CE4053 Real-time Operating Systems
Phase 2: Resource Sharing Protocols
Group Swensens

Jonathan Liem Zhuan Kim	**U1520775F**
Nicholas Koh Ming Xuan	**U1522485D**

Table of Contents

1. Resource Sharing Protocol 3
   1.1. Micrium kernel objects for resource sharing 3
   1.2. Existing resource access protocols in Micrium 3
   1.3. Stack Resource Protocol(SRP) 3
   1.4. SRP implemented in Micrium 4
      1.4.1. Modifications & Additions 4
      1.4.2. Stack & System Ceiling 5
      1.4.3. OSMutexPend() 5
      1.4.4. OSMutexPost() 6
      1.4.5. Red-Black tree & Blocked tasks 6
      1.4.6. OSSched() 7
      1.4.7. Resource Access Scenarios 8
   1.5. Limitations of current SRP design 9
2. Phase 1 Feedback (Synchronous release) 10
3. Benchmarking 11
   3.1. Scheduling 11
   3.2. Periodicity 11
   3.3. Mutex acquire/release 11
   3.4. Task execution time (Average time for each Task, for our example each task just do a series of printfs so that it is easy for us to know which task is running during debugging) 12
   3.5. Interrupt latency 12

1. Resource Sharing Protocol

1.1. Micrium kernel objects for resource sharing

In Micrium, we are provided with the basic implementation of Mutexes. The API allows us to create mutexes through OSMutexCreate(). It also allows any task to post or pend a mutex through the use of OSMutexPend() and OSMutexPost(). It should be noted that a task can actually pend a mutex multiple times, which is known as a mutex nesting. When this happens, the same task needs to release it the same number of times for the mutex to be fully available for other tasks to obtain, else there will be problems.

Micrium also has other services to manage shared resources, such as:

- Disabling the scheduler
- Disabling interrupts
    - The Micrium OS does this to ensure the atomicity of critical section code.
- Semaphores
- Monitors
    - Used for advanced resource management and synchronisation

1.2. Existing resource access protocols in Micrium

Micrium OS implements full-priority inheritance, and thus if a higher priority task requests an already held resource, the priority of the owner task will be raised to the priority of the new requestor. In essence, Priority Inheritance Protocol is already implemented in the OS. This is done in the kernel object OSMutex.

1.3. Stack Resource Protocol(SRP)

In general, SRP is identical in concept to I-PCP (Immediate-Priority Ceiling Protocol). However, the difference lies in that priority levels are not considered. This is important as the scheduler we are using is EDF which is based on deadlines to be met for each task. Each task has a preemption level which is inversely proportional to the its deadline. Hence, for the implementation of SRP in Micrium, it is important to take note that priorities no longer matter, and any comparison made between tasks are based on their preemption levels instead.

New variables will be required for this implementation. A resource ceiling for each resource is required as we will need to statically define the highest preemption level task that can use each resource. A new variable to be added is the system ceiling. This changes during runtime. This value represents the largest resource ceiling at any time. As the number of mutexes being held changes through runtime, this variable will change. To handle the constantly changing system ceiling, we have implemented a new data structure, a stack.

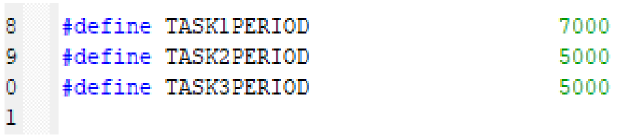
To allow a job J1 of a task T1 to preempt another job J2 of task T2, it needs to satisfy the following two conditions: J1 have higher priority than J2 in terms of job deadlines, and preemption level of T1 is higher than the system ceiling. It is important to note that since system ceiling changes through runtime, we need to code it such that system ceiling gets updated every time a mutex gets acquired or released.

If a job fails the SRP, it will be blocked. This is where a data structure to manage blocked tasks become useful. This data structure will need to know when to block and unblock the tasks. This data structure to be used is the Red-Black tree, a tree similar to the AVL-tree used in Phase 1 for the implementation of EDF.

1.4. SRP implemented in Micrium

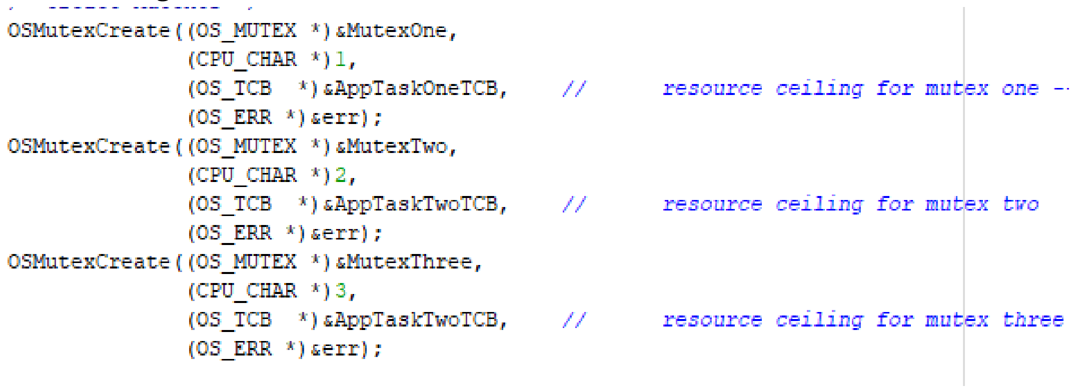
1.4.1. Modifications & Additions

We have added extra code into functions such as the OSMutexPend and OSMutexPost. Certain modifications also had to be made to phase 1 code due to the concept of a blocking tree being introduced. We had to add extra code into the mutex API in order to accomodate a resource ceiling for each resource. We statically define the resource ceiling for each mutex prior to runtime. This is determined by their preemption levels, inversely proportional to their deadlines.



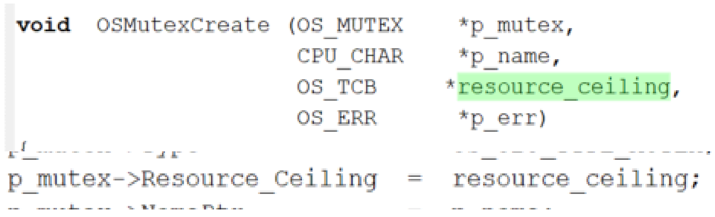
```
#define TASK1PERIOD                    7000
#define TASK2PERIOD                    5000
#define TASK3PERIOD                    5000
```

In our example, task one uses all three mutexes, task two uses mutexes two and three, and task three using none.



```
OSMutexCreate((OS_MUTEX *)&MutexOne,
              (CPU_CHAR *)1,
              (OS_TCB   *)&AppTaskOneTCB,    //      resource ceiling for mutex one -
              (OS_ERR *)&err);
OSMutexCreate((OS_MUTEX *)&MutexTwo,
              (CPU_CHAR *)2,
              (OS_TCB   *)&AppTaskTwoTCB,    //      resource ceiling for mutex two
              (OS_ERR *)&err);
OSMutexCreate((OS_MUTEX *)&MutexThree,
              (CPU_CHAR *)3,
              (OS_TCB   *)&AppTaskTwoTCB,    //      resource ceiling for mutex three
              (OS_ERR *)&err);
```

This is done by passing the entire task TCB as an argument for each resource. In the picture above, task one would be the resource ceiling for mutex one, and task two for mutexes two and three.



```
void  OSMutexCreate  (OS_MUTEX    *p_mutex,
                      CPU_CHAR    *p_name,
                      OS_TCB     *resource_ceiling,
                      OS_ERR      *p_err)
```

```
p_mutex->Resource_Ceiling  =  resource_ceiling;
```

1.4.2. Stack & System Ceiling

The data structure we are using to manage the mutexes held by the tasks is a stack. The concept is pretty simple: acquiring of a mutex simply pushes the mutex variable into the stack, and releasing a mutex does a pop. For this purpose, we have included stack API that takes in mutexes.

```c
struct stack_node
{
    OS_MUTEX* data;
    struct stack_node* next;
};
```

When a mutex gets acquired, we push it into the stack data structure.

1.4.3. OSMutexPend()

```c
        OS_MUTEX_STACK_HEAD = stack_push(OS_MUTEX_STACK_HEAD, &STACK_NODE_ARR[stack_count]);
```

Upon doing so, we update the system ceiling by calling stack_find_min_deadline() in our stack API. Note that OS_SYSTEM_CEILING is an OS_DEADLINE variable.

```c
    OS_SYSTEM_CEILING = stack_find_min_deadline(OS_MUTEX_STACK_HEAD);
    /*######################################################################
```

```c
OS_DEADLINE OS_SYSTEM_CEILING;
```

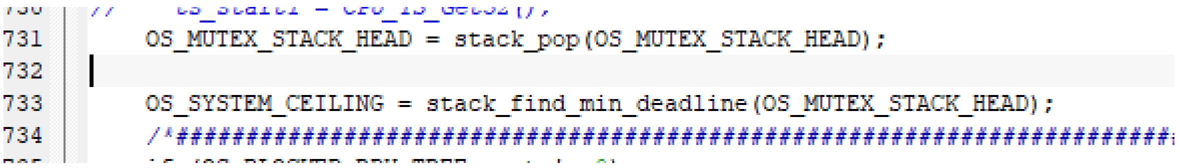
stack_find_min_deadline() basically iterates through the stack structure to get the lowest resource ceiling (in terms of deadline) by checking against every resource ceiling of each mutex inside.

```c
    deadline = cur->data->Resource_Ceiling->Deadline;
    while (cur != 0){
      if (cur->data->Resource_Ceiling->Deadline < deadline) {
        deadline = cur->data->Resource_Ceiling->Deadline;
      }
      cur = cur->next;
    }
  }
  return deadline;
}
```

It is also important to actually reset the system ceiling so that it doesn't crash if there are no more mutexes inside the stack. In this case, we assign it to be high enough so that if there are no mutexes being held, any task that wants to preempt, when compared to this large deadline value, can be preempted. This assignment is done inside stack_find_min_deadline() in the stack API.

```c
  */
  if(cur==0)    //      nothing in mutex
  {
    return 99999999;
  }
```

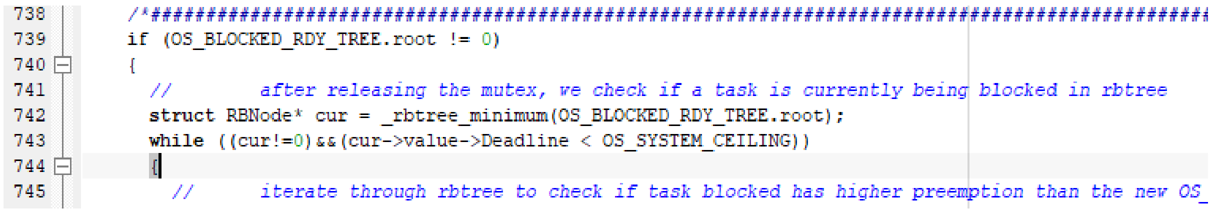
1.4.4. OSMutexPost()



```
//    os_start1 = cpu_ts_get32();
    OS_MUTEX_STACK_HEAD = stack_pop(OS_MUTEX_STACK_HEAD);

    OS_SYSTEM_CEILING = stack_find_min_deadline(OS_MUTEX_STACK_HEAD);
    /*####################################################################
```

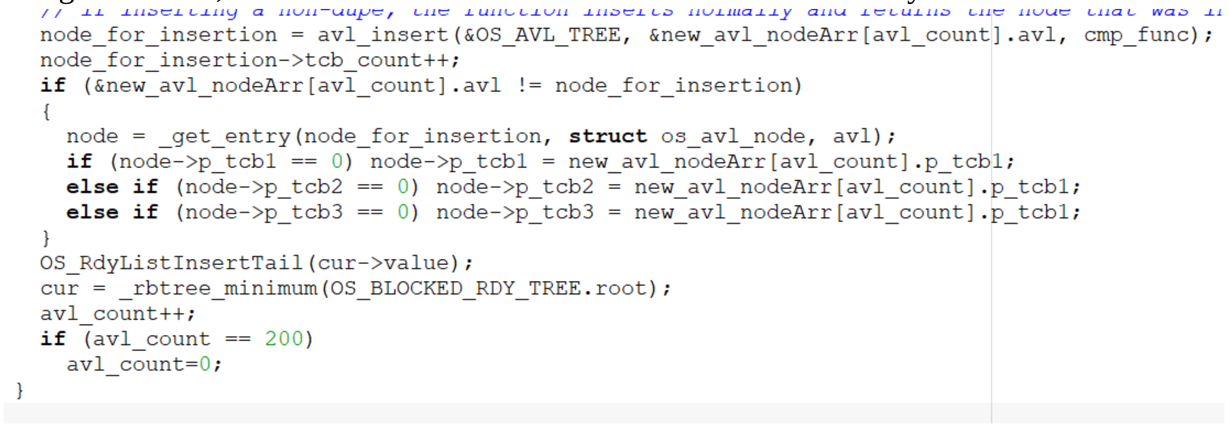
As shown above, posting of a mutex does a pop as mentioned, and we update the system ceiling accordingly.

After the posting of the mutex in OSMutexPost, the Red-Black tree will be examined to see what tasks shall be unblocked and moved to both the ready list and AVL tree (for EDF).



```
/*##################################################################################
if (OS_BLOCKED_RDY_TREE.root != 0)
{
  //        after releasing the mutex, we check if a task is currently being blocked in rbtree
  struct RBNode* cur = _rbtree_minimum(OS_BLOCKED_RDY_TREE.root);
  while ((cur!=0)&&(cur->value->Deadline < OS_SYSTEM_CEILING))
  {
    //      iterate through rbtree to check if task blocked has higher preemption than the new OS_
```

The tasks to be unblocked is determined by the deadline of the blocked tasks. If the deadline of the blocked tasks are strictly smaller than the current system ceiling, it is unblocked. After being unblocked, the task is then added into the AVL tree and the ready list.

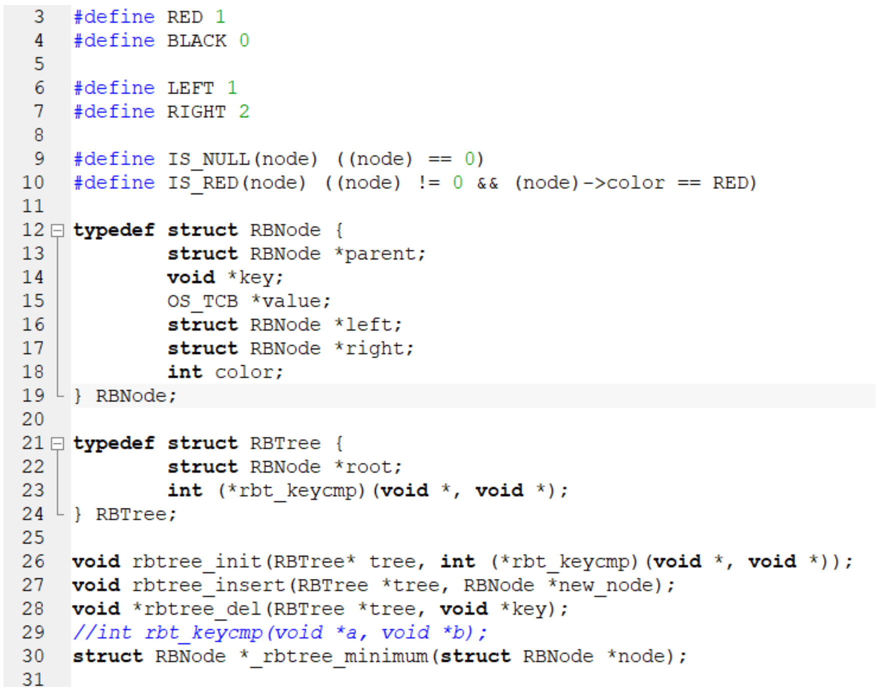


```
  // if inserting a non-dupe, the function inserts normally and returns the node that was i
  node_for_insertion = avl_insert(&OS_AVL_TREE, &new_avl_nodeArr[avl_count].avl, cmp_func);
  node_for_insertion->tcb_count++;
  if (&new_avl_nodeArr[avl_count].avl != node_for_insertion)
  {
    node = _get_entry(node_for_insertion, struct os_avl_node, avl);
    if (node->p_tcb1 == 0) node->p_tcb1 = new_avl_nodeArr[avl_count].p_tcb1;
    else if (node->p_tcb2 == 0) node->p_tcb2 = new_avl_nodeArr[avl_count].p_tcb1;
    else if (node->p_tcb3 == 0) node->p_tcb3 = new_avl_nodeArr[avl_count].p_tcb1;
  }
  OS_RdyListInsertTail(cur->value);
  cur = _rbtree_minimum(OS_BLOCKED_RDY_TREE.root);
  avl_count++;
  if (avl_count == 200)
    avl_count=0;
}
```

1.4.5. Red-Black tree & Blocked tasks

As mentioned before, the data structure holding blocked tasks is a Red-Black tree. Unblocking occurs in OSMutexPost, while blocking occurs in OSSched. Blocking in OSSched will be explained in the later section 1.4.7.

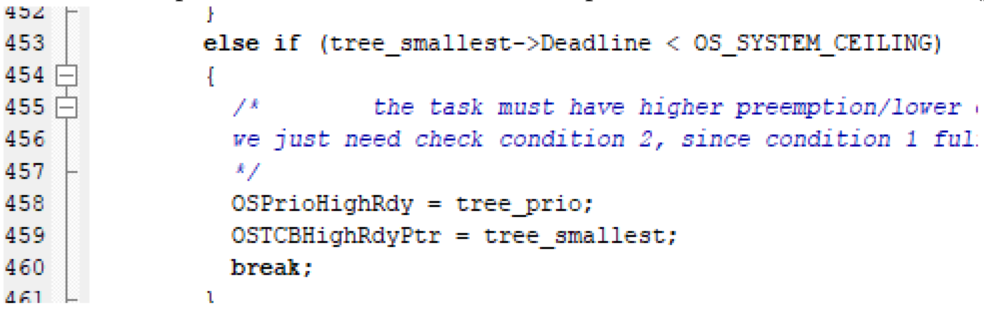
The Red-Black tree is implemented using an external redblack.c and redblack.h file. The tree is initialised in OSInit. Similar to the AVL tree, it is a self-balancing binary tree. However, instead of using heights to balance, each node is instead assigned a colour, red or black. The rule used to balance the tree is that the path to every descendant leaf contains the same number of black nodes. Anytime a node is inserted or deleted, the tree has to balanced, similar to the AVL tree.



```
#define RED 1
#define BLACK 0

#define LEFT 1
#define RIGHT 2

#define IS_NULL(node) ((node) == 0)
#define IS_RED(node) ((node) != 0 && (node)->color == RED)

typedef struct RBNode {
        struct RBNode *parent;
        void *key;
        OS_TCB *value;
        struct RBNode *left;
        struct RBNode *right;
        int color;
} RBNode;

typedef struct RBTree {
        struct RBNode *root;
        int (*rbt_keycmp)(void *, void *);
} RBTree;

void rbtree_init(RBTree* tree, int (*rbt_keycmp)(void *, void *));
void rbtree_insert(RBTree *tree, RBNode *new_node);
void *rbtree_del(RBTree *tree, void *key);
//int rbt_keycmp(void *a, void *b);
struct RBNode *_rbtree_minimum(struct RBNode *node);
```

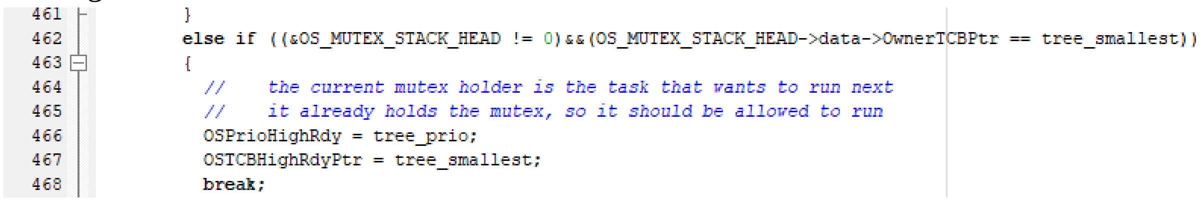
1.4.6. OSSched()

As we were already doing EDF scheduling back in phase 1, any task that gets selected by our AVL tree will automatically have the lowest deadline out of all the tasks that is waiting to run fulfilling the first SRP condition. Hence, we just need to check if the task have a higher preemption level than the system ceiling in order to fulfil all the preemption requirements of SRP. This implementation can be seen in the picture below inside OSSched().



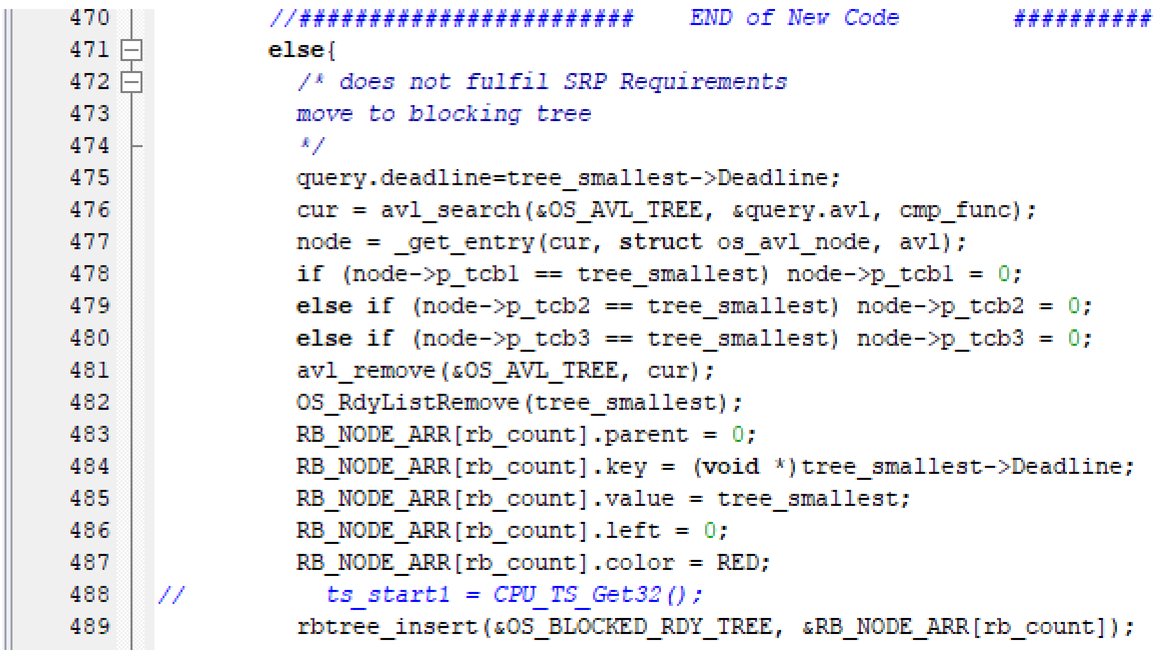
```
          }
          else if (tree_smallest->Deadline < OS_SYSTEM_CEILING)
          {
            /*         the task must have higher preemption/lower
            we just need check condition 2, since condition 1 ful
            */
            OSPrioHighRdy = tree_prio;
            OSTCBHighRdyPtr = tree_smallest;
            break;
          }
```

Since there is the possibility that the current mutex holder is the one that wants to run, we just schedule it as per normal as seen below. Thus, the mutex holder will be able to resume running.



```
          }
          else if ((&OS_MUTEX_STACK_HEAD != 0)&&(OS_MUTEX_STACK_HEAD->data->OwnerTCBPtr == tree_smallest))
          {
            //    the current mutex holder is the task that wants to run next
            //    it already holds the mutex, so it should be allowed to run
            OSPrioHighRdy = tree_prio;
            OSTCBHighRdyPtr = tree_smallest;
            break;
```

However if all the conditions fail, it means that by SRP the task should be blocked, and hence we put it into the Red-Black Tree, and remove it accordingly from the Ready List and AVL tree.

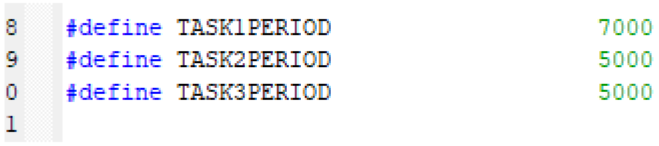


```
        //########################    END of New Code        ##########
        else{
          /* does not fulfil SRP Requirements
          move to blocking tree
          */
          query.deadline=tree_smallest->Deadline;
          cur = avl_search(&OS_AVL_TREE, &query.avl, cmp_func);
          node = _get_entry(cur, struct os_avl_node, avl);
          if (node->p_tcb1 == tree_smallest) node->p_tcb1 = 0;
          else if (node->p_tcb2 == tree_smallest) node->p_tcb2 = 0;
          else if (node->p_tcb3 == tree_smallest) node->p_tcb3 = 0;
          avl_remove(&OS_AVL_TREE, cur);
          OS_RdyListRemove(tree_smallest);
          RB_NODE_ARR[rb_count].parent = 0;
          RB_NODE_ARR[rb_count].key = (void *)tree_smallest->Deadline;
          RB_NODE_ARR[rb_count].value = tree_smallest;
          RB_NODE_ARR[rb_count].left = 0;
          RB_NODE_ARR[rb_count].color = RED;
//          ts_start1 = CPU_TS_Get32();
          rbtree_insert(&OS_BLOCKED_RDY_TREE, &RB_NODE_ARR[rb_count]);
```

1.4.7. Resource Access Scenarios

For all resource access scenarios, we are assuming that we are working on a given task set that is schedulable under EDF.

For scenarios with non-nested or nested mutex acquiring and releasing, SRP works, as a task can only run if the task is guaranteed to have all the resources required.



```
#define TASK1PERIOD                    7000
#define TASK2PERIOD                    5000
#define TASK3PERIOD                    5000
```

One example can be done with the existing base code. The periods given are as above. The resource ceiling for both MutexTwo and MutexThree is AppTask2.
Note that TimerTask runs every 1ms.

Timeline (Abridged):

- Starting from when AppTaskStart is created, AppTaskStart runs
- AppTaskStarts creates the 3 AppTasks, BUT does not add the TCBs to the Ready list or AVL tree
- OSTaskDelete runs, deletes AppTaskStart, and sets a flag for synchronous release
- Synchronous release occurs
- By EDF, the lowest deadline task is chosen to run, namely AppTask2 for this example.
- AppTask2 will be allowed to run by SRP, as the system ceiling is at the max value of 999999. AppTask2's deadline is 5000, which is smaller than 999999.
- AppTask2 will take MutexTwo and MutexThree, changing the system ceiling to 5000.
- During this, TickTask will continually interrupt AppTask2 and run every 1ms.

- AppTask2 will be allowed to resume, even though its deadline is equal to the system ceiling, as it is the current mutex owner. (see 1.4.6.)
- Once AppTask2 is done, it will post the mutexes it currently holds. Thus the system ceiling will again be changed to the maximum values of 999999. The blocked tree, if it was not empty, will be inspected to see if any tasks should be unblocked. However, the blocked tree is empty.
- The scheduler will be called again, and by EDF, the next task is selected to run, which is AppTask3 in this case.

Special Cases (Blocking of tasks):
If for example, AppTask2 is released while AppTask1 is running, and AppTask2 has a smaller absolute deadline than AppTask1, the scheduler will be called. By EDF, AppTask2 will be chosen to run. However, as AppTask1 is running the system ceiling is modified such that AppTask2 cannot be chosen to run. This is because AppTask1 uses all 3 mutexes, causing the system ceiling (in this example) to be the deadline of AppTask2. So, AppTask2 is not allowed to run since it has to strictly be higher than system ceiling to be able to run. (see 1.4.6.) Thus, AppTask2 will be blocked and moved into the Red-Black tree. Only when AppTask1 finished and posts its held mutexes, will AppTask2 be unblocked, and assigned by the scheduler to finally run.

1.5. Limitations of current SRP design

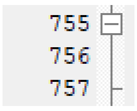
The system ceiling maximum is currently set to 999999. As our design has not found a way to handle OSTickCtr overflow, at some point, the deadlines of the recursive tasks will exceed the system ceiling maximum, and SRP will prevent any task from running. Thus our design can only run for a limited time.

In this design, we have opted to use a fixed memory partition for use in our data structures, instead of using malloc(), realloc() and free(). Thus this design can only handle a limited number of tasks, in this case, a 100 tasks. However our choice results in an easier memory management as memory leaks are not an issue, as compared to using dynamic memory allocation.

2. Phase 1 Feedback (Synchronous release)

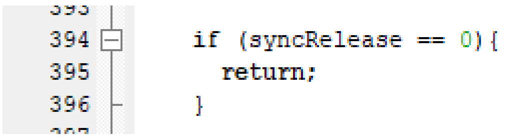
In order to implement synchronous release properly, we have made use of a flag as mentioned in the feedback from Phase 1. Previously, we were using a check for when the OSTickCtr is a certain value (5 in our case). In our new implementation, we make use of a flag syncRelease. This is a value where we will assign an integer number, either 0, 1 or 2 to ensure that all recursive tasks are synchronously released into the readylist before our EDF scheduler starts scheduling them.

This flag can be found inside three functions: OSSched(), OSTaskDel(), and OS_revive_rec_task(). We know synchronous release should happen after AppTaskStart creates all the tasks via OSRecTaskCreate(), we can have the flag inside OSTaskDel(), such that only when AppTaskStart is deleted, the flag becomes assigned to 1 (its default value is 0). Note that this flag is assigned 1 only after all tasks are recursively created (present in the heap structure).



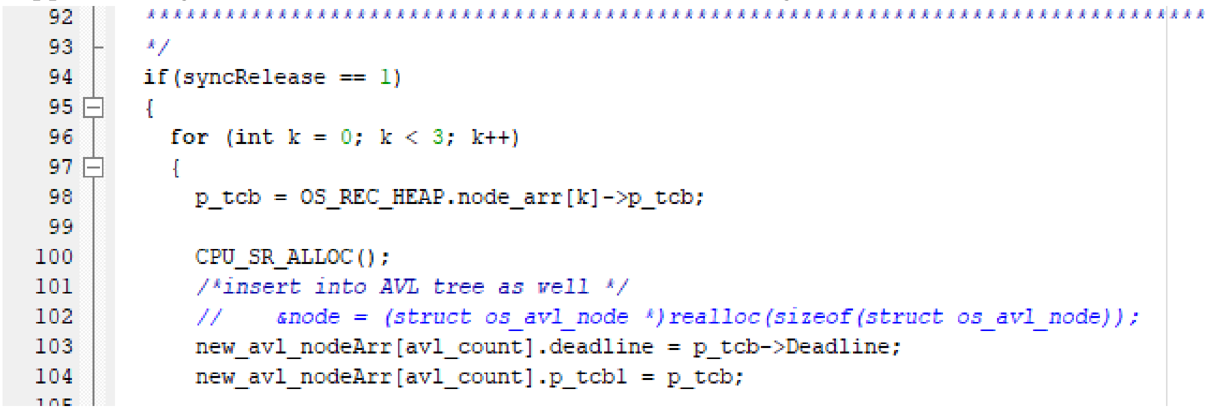
```
    if (syncRelease==0){
      syncRelease = 1;
    }
```

Inside OSSched(), we will not run our EDF scheduler so long as syncRelease is still 0. This is because we want to wait for all tasks to be inside the readylist. Hence, the first check OSSched() does is to see if syncRelease is 0, and if yes, make a return without scheduling any task. This is similar to locking the scheduler, with the Micrium OS variable OSSchedLockNestingCtr.



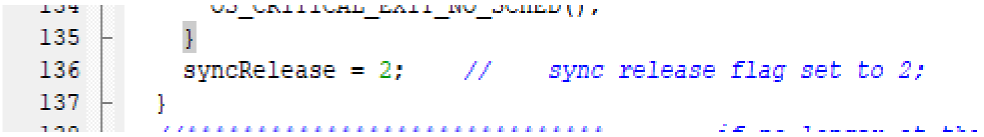
```
    if (syncRelease == 0){
      return;
    }
```

Once syncRelease becomes assigned to 1, it allows the synchronous release of all tasks to happen. The synchronous release method works the same way as in Phase 1.



```
*****************************************************************************************
*/
if(syncRelease == 1)
{
  for (int k = 0; k < 3; k++)
  {
    p_tcb = OS_REC_HEAP.node_arr[k]->p_tcb;

    CPU_SR_ALLOC();
    /*insert into AVL tree as well */
    //    &node = (struct os_avl_node *)realloc(sizeof(struct os_avl_node));
    new_avl_nodeArr[avl_count].deadline = p_tcb->Deadline;
    new_avl_nodeArr[avl_count].p_tcbl = p_tcb;
```

However, at the end of this synchronous release, the flag gets set to 2.



```
    OS_CRITICAL_EXIT_NO_SCHED();
    }
    syncRelease = 2;    //    sync release flag set to 2;
  }
```

This is done so that the next time OS_revive_rec_task() is called, synchronous release code doesn't get run again, and also to allow our EDF scheduler to work properly without ever returning (it returns before the EDF scheduling code if we set the flag back to 0).

3. Benchmarking

For benchmarking of our implementation, we have made use of CPU_TS_Get32() as in phase 1 to obtain the system ticks required for specific functions. This API helps us to get the current 32-bit CPU timestamp. For most functions, benchmarks are obtained by placing a variable ts_start1 at the start of the function, and ts_end1 at the end.

However, for unknown reasons, we were unable to obtain a subtracted value by doing ts_end1 = CPU_TS_Get32() - ts_start1 as recommended in the lecture. We were also not able to obtain any numerical value for the ts_end1 and ts_start1 variables in the watch list of the IAR. Hence, a breakpoint was used inside of the CPU_TS_Get32() function in order to get an accurate timestamp value, where we were able to obtain a numerical value of the timestamp to be returned. We then subtract it accordingly when we reach ts_start1 and ts_end1.

There may be certain discrepancies in our measurements, as we had to account for the fact that in some functions, there were nested calls to other functions such as OSSched(), which makes measuring the exact time from start to end of the function itself impossible. Hence, for certain functions we chose specific locations in the code to get a timestamp. Below are the system ticks measured for the specified functions and implementations of our code.

3.1. Scheduling

- *OSSched()* with EDF Scheduling: 277
- *OSSched()* without EDF Scheduling *(due to synchronous release not done)*: 116

3.2. Periodicity

- *OS_revive_rec_task(),* Synchronous Release of tasks: 1291
- *OS_revive_rec_task(),* Recursive task to revive: 1925
- *OS_revive_rec_task(),* No Tasks to revive: 37
- *heap_push()*: 45
- *heap_pop()*: 120

3.3. Mutex acquire/release

- *OSMutexPend()*: 225
- *OSMutexPost()*: 168
- *stack_push()*: 37
- *stack_pop()*: 29

3.4. Task execution time

Average time for each Task, for our example each task just do a series of printfs so that it is easy for us to know which task is running during debugging)

- *AppTaskOne()*: 102672660
- *AppTaskTwo()*: 51551027
- *AppTaskThree()*: 58418197

3.5. Interrupt latency

- *OSTickTask()* excluding *OS_revive_rec_task()* and *OS_TickListUpdate()*: 26
    - Total time depends on state of system, whether there are tasks to revive etc.
- *OS_TickListUpdate()*: 190
- *rbtree_insert()*, time taken to insert task into blocking tree: 140
- *rbtree_del()*, time taken to delete task from blocking tree: 200