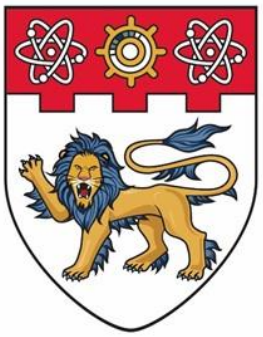


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

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

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. This is mentioned in the Micrium OS User's Manual. Thus, 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. Red-Black Tree

The Red-Black tree, similar to the AVL tree, is a self-balancing binary search tree. Each node of the binary has an additional value, namely, the color. The AVL tree's counterpart is the height.

The balancing of the tree is assisted by the properties of the Red-Black tree, namely:

- Each node is either colored Red or Black
- All leaves (NIL nodes) are by default colored Black
- If a node is Red, both its children must be Black
- Every path from a given node to any of its descendant NIL nodes (or leaves) contains the same number of Black nodes
- Critically: The path from the root to the farthest leaf is no more than ***twice*** as long as the path from the root to the nearest leaf.

Thus, these properties result in the tree being ***approximately*** balanced. Even though the tree is not perfectly balanced, the tree is balanced enough such that time complexities are still manageable, as shown below.

Search	$O(\log n)$
Insert	$O(\log n)$
Delete	$O(\log n)$

The time complexities are due to the times being proportional to the height of the tree. However, to keep the properties true for every insertion and deletion, operations to fix the tree is required. Like the AVL tree, rotation of subtrees is used. Additionally, re-painting of nodes are required, to keep the alternating colors of the Red-Black tree.

1.5. SRP implemented in Micrium

1.5.1. 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.

```
8   #define TASK1PERIOD                  7000
9   #define TASK2PERIOD                  5000
0   #define TASK3PERIOD                  5000
1
```

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;
```

1.5.2. Stack & System Ceiling

The data structure we are using to manage the mutexes held by the tasks is a stack. The concept is 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.

```
2   struct stack_node
3   {
4       OS_MUTEX* data;
5       struct stack_node* next;
6   };
```

The system ceiling on the other hand, is a variable that holds the current minimum period of all tasks that are holding resources. That value is updated in OSMutexPost and OSMutexPend. By default, if the stack is empty, the system ceiling value is set to 999999, a large number, such that any task will be able to run.

1.5.3. OSMutexPend()

```
412         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.

```
737     OS_SYSTEM_CEILING = stack_find_min_deadline(OS_MUTEX_STACK_HEAD);
738     /*#########################################################################
```

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.

```
60    deadline = cur->data->Resource_Ceiling->Deadline;
61    while (cur != 0){
62      if (cur->data->Resource_Ceiling->Deadline < deadline) {
63        deadline = cur->data->Resource_Ceiling->Deadline;
64      }
65      cur = cur->next;
66    }
67  }
68  return deadline;
69 }
```

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.

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

1.5.4. OSMutexPost()

```
731     OS_MUTEX_STACK_HEAD = stack_pop(OS_MUTEX_STACK_HEAD);
732
733     OS_SYSTEM_CEILING = stack_find_min_deadline(OS_MUTEX_STACK_HEAD);
734     /*#########################################################################
```

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).

```
738     /*#########################################################################################
739     if (OS_BLOCKED_RDY_TREE.root != 0)
740     {
741       //       after releasing the mutex, we check if a task is currently being blocked in rbtree
742       struct RBNode* cur = _rbtree_minimum(OS_BLOCKED_RDY_TREE.root);
743       while ((cur!=0)&&(cur->value->Deadline < OS_SYSTEM_CEILING))
744       {
745         //      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;
}
```

We added certain modifications towards the end of the OSMutexPost() to account for a proper preemption. For instance, if a task A releases its mutexes and hereby unblocks another task B from the Red Black tree that wants to use the mutex, and satisfy SRP conditions, system should jump straight running of the code of task B, postponing the remainder of task A code to later.

```
p_pend_list = &p_mutex->PendList;
if (p_pend_list->NbrEntries == (OS_OBJ_QTY)0) {
    p_mutex->OwnerTCBPtr     = (OS_TCB       *)0;
    p_mutex->OwnerNestingCtr = (OS_NESTING_CTR)0;
    OS_CRITICAL_EXIT();
    if ((opt & OS_OPT_POST_NO_SCHED) == (OS_OPT)0) {
      OSSched();
    }
    *p_err = OS_ERR_NONE;
    return;
```

This is achieved by allowing the system to enter OSSched upon releasing of mutex. Prior to this change, the system makes a return as it satisfy the if block above. By adding another if block to allow OSSched() to be called, we mitigate this problem since it allows the scheduler call the task B to run now even if there are still code left to run for task A.

1.5.5. 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 1.5.7.

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.

```
3  #define RED 1
4  #define BLACK 0
5
6  #define LEFT 1
7  #define RIGHT 2
8
9  #define IS_NULL(node) ((node) == 0)
10 #define IS_RED(node) ((node) != 0 && (node)->color == RED)
11
12 typedef struct RBNode {
13         struct RBNode *parent;
14         void *key;
15         OS_TCB *value;
16         struct RBNode *left;
17         struct RBNode *right;
18         int color;
19 } RBNode;
20
21 typedef struct RBTree {
22         struct RBNode *root;
23         int (*rbt_keycmp)(void *, void *);
24 } RBTree;
25
26 void rbtree_init(RBTree* tree, int (*rbt_keycmp)(void *, void *));
27 void rbtree_insert(RBTree *tree, RBNode *new_node);
28 void *rbtree_del(RBTree *tree, void *key);
29 //int rbt_keycmp(void *a, void *b);
30 struct RBNode *_rbtree_minimum(struct RBNode *node);
31
```

1.5.6. 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().

```
452         }
453         else if (tree_smallest->Deadline < OS_SYSTEM_CEILING)
454         {
455           /*        the task must have higher preemption/lower
456           we just need check condition 2, since condition 1 ful
457           */
458           OSPrioHighRdy = tree_prio;
459           OSTCBHighRdyPtr = tree_smallest;
460           break;
461         }
```

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. Without this check, the task that was previously running will not be able to run, even though it is the one holding the mutexes.

```
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.

```
470         //########################    END of New Code        ##########
471         else{
472           /* does not fulfil SRP Requirements
473           move to blocking tree
474           */
475           query.deadline=tree_smallest->Deadline;
476           cur = avl_search(&OS_AVL_TREE, &query.avl, cmp_func);
477           node = _get_entry(cur, struct os_avl_node, avl);
478           if (node->p_tcb1 == tree_smallest) node->p_tcb1 = 0;
479           else if (node->p_tcb2 == tree_smallest) node->p_tcb2 = 0;
480           else if (node->p_tcb3 == tree_smallest) node->p_tcb3 = 0;
481           avl_remove(&OS_AVL_TREE, cur);
482           OS_RdyListRemove(tree_smallest);
483           RB_NODE_ARR[rb_count].parent = 0;
484           RB_NODE_ARR[rb_count].key = (void *)tree_smallest->Deadline;
485           RB_NODE_ARR[rb_count].value = tree_smallest;
486           RB_NODE_ARR[rb_count].left = 0;
487           RB_NODE_ARR[rb_count].color = RED;
488   //        ts_start1 = CPU_TS_Get32();
489           rbtree_insert(&OS_BLOCKED_RDY_TREE, &RB_NODE_ARR[rb_count]);
```

After blocking the selected task, a new task is picked again from the EDF scheduler and the same checks are conducted.

1.5.7. 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.

```
8   #define TASK1PERIOD                 7000
9   #define TASK2PERIOD                 5000
0   #define TASK3PERIOD                 5000
1
```

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:

- 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.

1.6. Limitations of current SRP design

Our implementation of the stack requires a guaranteed push and pop of the mutex every single time a mutex is acquired and released respectively. This is rather expensive (in terms of CPU cycles) as the updating of system ceilings do not require the push and pop of every mutex. However, this design opens the possibility of expanding the current design to handle

non-nested mutex acquiring and releasing. This, requires the stack to be replaced by another structure that can be traversed through instead, like a linked list or a tree structure.

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. This memory stability may or may not be favored in certain real-time applications.

Lastly, an improvement can be made in the implementation of the two binary tree structures, the AVL tree, and Red-Black tree. An improvement can be made to allow for a new data structure to be attached to the nodes of both trees, allowing only for a single node for each key value. In other words, if there are multiple tasks with similar deadlines, a new node will not be created instead, but attached to the existing node with the same deadline.

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).

```
755     if (syncRelease==0){
756       syncRelease = 1;
757     }
```

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.

```
394     if (syncRelease == 0){
395       return;
396     }
```

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.

```
92  ********************************************************************************************************
93  */
94  if(syncRelease == 1)
95  {
96    for (int k = 0; k < 3; k++)
97    {
98      p_tcb = OS_REC_HEAP.node_arr[k]->p_tcb;
99
100     CPU_SR_ALLOC();
101     /*insert into AVL tree as well */
102     //    &node = (struct os_avl_node *)realloc(sizeof(struct os_avl_node));
103     new_avl_nodeArr[avl_count].deadline = p_tcb->Deadline;
104     new_avl_nodeArr[avl_count].p_tcb1 = p_tcb;
```

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

```
135     }
136     syncRelease = 2;    //    sync release flag set to 2;
137   }
```

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.

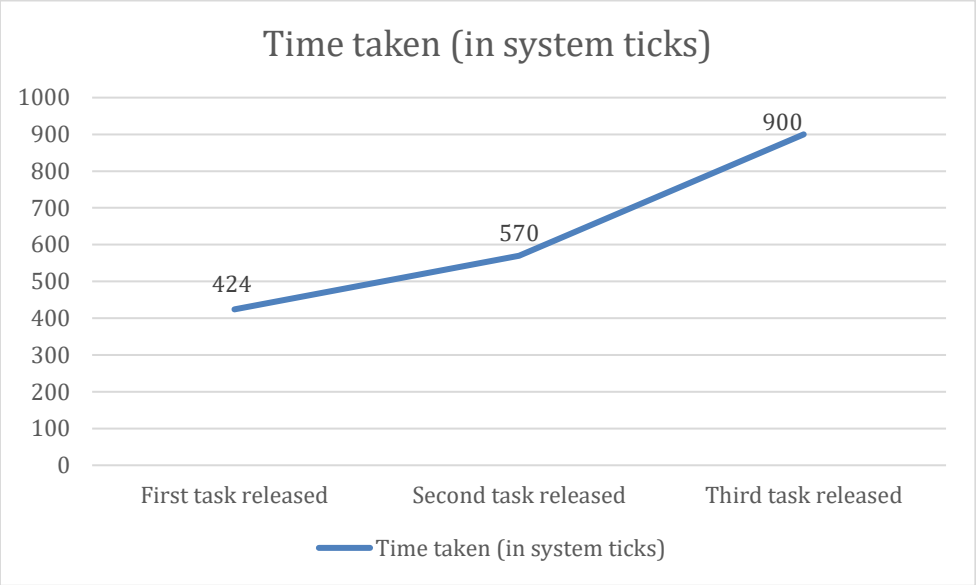
There may be certain discrepancies in our measurements, as we had to account for the fact that in some functions, there were nested calls to other functions such as OSSched(), which makes measuring the exact time from start to end of the function itself impossible. Hence, for certain functions we chose specific locations in the code to get a timestamp. Below are the **average** system ticks measured for the specified functions and implementations of our code.
** **average**: ran 4-5 times to obtain an average system tick value

3.1. Scheduling

- *OSSched()* without EDF Scheduling *(due to synchronous release not done)*: **24**
- *OSSched()* with EDF Scheduling
    - If task already running and scheduled again: **255**
    - Any task that is less than system ceiling: **249**
    - If task is the one holding mutex, and wants to run: **291**
    - Blocking of task: **810**

Remarks:
OSSched() has multiple checks to ensure that the next task to run is the correct one. Thus there are multiple values of OSSched() with EDF. If all the checks failed, the task picked out by EDF cannot run due to SRP, and will be blocked, resulting in the large number of system ticks.

3.2. Periodicity



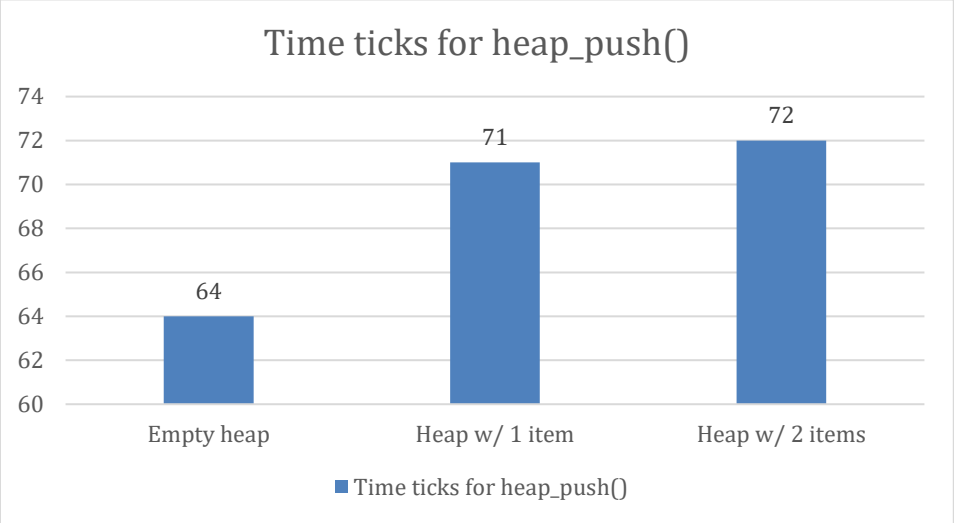
- *OS_revive_rec_task()*: Time taken to do Synchronous Release
    - Synchronous Release of all 3 tasks: **1900**

Remarks:
According to the graph above, the time taken increases with more tasks being released. This is due to the additional overhead of balancing the AVL tree (our EDF ready list). However, the trend at this point is still unclear as the system only has three tasks. The sample size is too small to declare that the trend does not conform to O(log(n)).

- *OS_revive_rec_task():* For Reviving of Periodic Task (happens after sync release done)
    - Per recursive task to revive: **1950**
    - No Tasks to revive: **42**

Remarks:
Re-releasing recursive tasks are expensive, as it essentially performs the same actions of OSTaskCreate. Additionally, the newly refreshed task needs to be added to the ready list and AVL tree, resulting in even more time ticks to pass.

- *heap_push()*:
  - Inside *OSRecTaskCreate()*



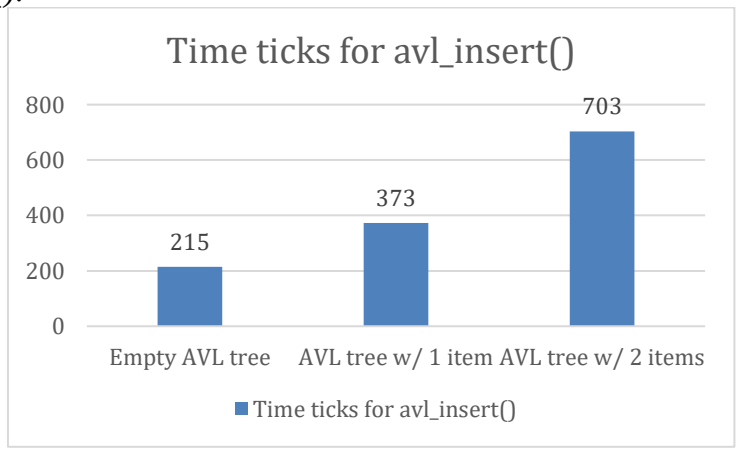
Remarks:
Balancing for a binary heap with 1 or 2 items is relatively easy, and thus do not require much time cycles. Thus, there is no trend visible yet. With more items in the heap, the time taken would be O(n*log(n)) instead. Note that this heap_push() is called during the initial creation of the recursive tasks, in AppTaskStart.

  - Inside *OS_revive_rec_task()*
    - Always done with 2 items left: **72**

- *heap_pop()*:
  - Inside *OS_revive_rec_task()*
    - Always pop heap with 3 items (max) inside: **90**

Remarks:
For our example, heap_push() and heap_pop() is also called when tasks are re-released into the system, as the deadline needs to be updated. As the heap is always full of all the three tasks, in this example, the function heap_push() and heap_pop() will always take the same amount of time. For more details, please look at our phase 1 report.

- *avl_insert():*

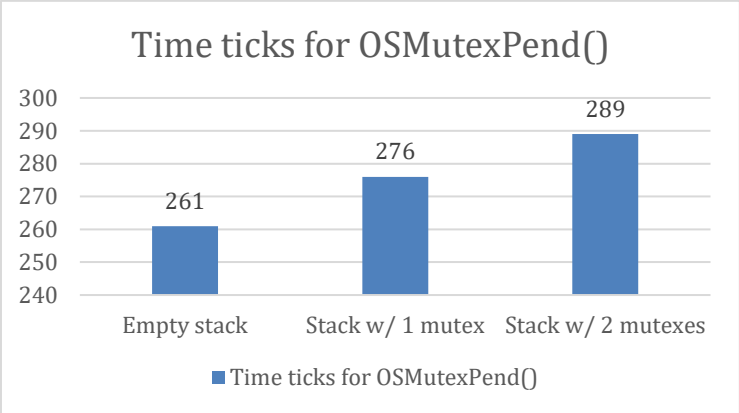


- *avl_remove():*
  - Max: **277**
  - Min: **118**

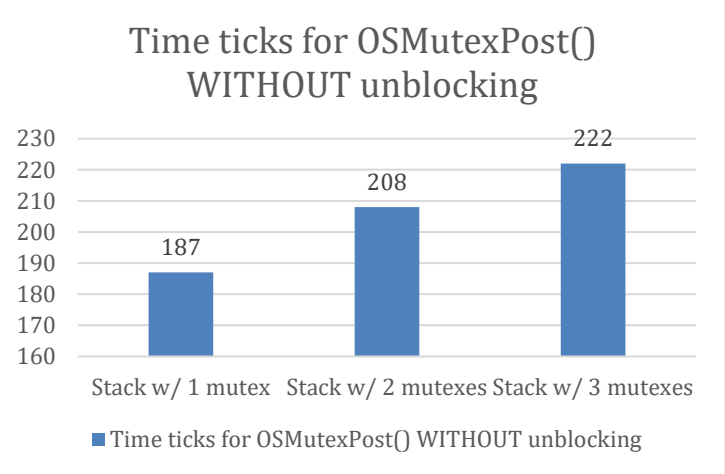
Remarks:
avl_insert() with 2 items takes a lot more time, as balancing is required to be done with the now new tree with 3 nodes. Insertion with 1 node and empty tree is relatively cheap, as not much balancing is needed to be done. Insertion into a tree with an existing item only consists of adding the new node to the left or right of the tree, without any balancing required.

3.3. Mutex acquire/release

- *OSMutexPend()*:



- *OSMutexPost()*:



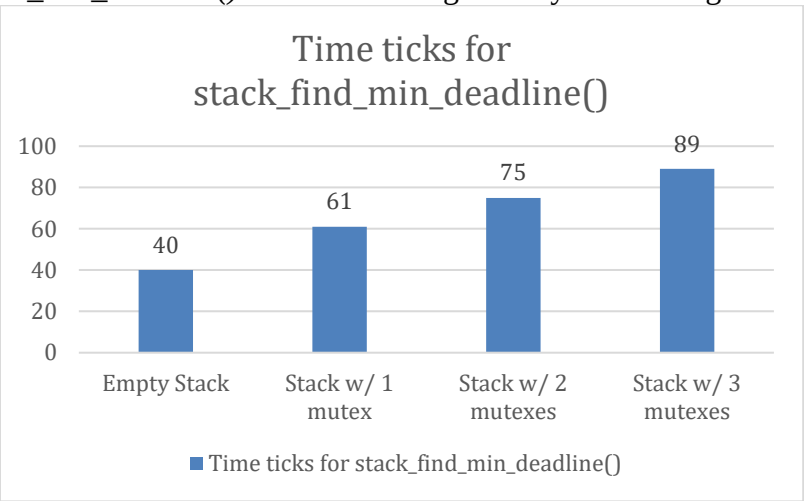
  - For every task we can unblock: **we add an average of 720 to above values**

Remarks:
The time taken does not include the compulsory call to OSSched(). The time taken for OSMutexPost() increases with the size of the stack, as the system ceiling has to be modified. This modification, namely, stack_find_min_deadline() will search through the whole stack for the new system ceiling, thus will scale with the size of the stack.

In the case for when tasks are unblocked from the Red-Black tree, the additional time ticks required are approximately 720 ticks, as there are multiple operations, such as removal of the

task from the Red-Black tree, addition of the task to the AVL tree, and addition of the task to the Ready List.

- *stack_find_min_deadline():* Time taken to get the system ceiling from stack



Remarks:
As mentioned in the benchmarking of OSMutexPost(), this function is the main reason for the increasing time cost for OSMutexPost(). The larger the size of the stack, the more time taken due to iteratively search the stack.

- *stack_push()*: Time taken to push mutex into stack
  - With no mutex in stack: **41**
  - With one mutex already in stack: **41**
  - With two mutex already in stack: **41**
- *stack_pop()*: Time taken to pop mutex into stack
  - With three mutex in stack: **33**
  - With two mutex already in stack: **33**
  - With one mutex already in stack: **33**

Remarks:
Pushing and popping to and from the stack takes O(1) constant time, and the benchmarks show that.

3.4. Task execution time

This is the average time for each Task. For our example, each task executes a series of printfs so that it is easy for us to know which task is running during debugging.

Note: values below may differ from actual ones left in code, as changes were made to tasks during demonstration.

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

3.5. Interrupt latency

- *OSTickTask()* excluding *OS_revive_rec_task()* and *OS_TickListUpdate()*: **26**
- *OS_TickListUpdate()*: **190**

Remarks:
OSTickTask can take very little time if there is nothing to update in the Tick List, and if there is no recursive tasks to be re-released.

- *rbtree_insert()*: time taken to insert task into blocking tree: **140**
- *rbtree_del()*: time taken to delete task from blocking tree: **200**
- *_rbtree_minimum():* time taken to retrieve the task with minimum deadline from blocking tree: **38**

Remarks:
Finding the minimum of the Red-Black tree is relatively easy, as the Red-Black tree is still a binary tree. Traversing to the left and to the end is cheap. However, insertion and deletion is relatively expensive due to the required balancing of the tree afterwards.