NANYANG TECHNOLOGICAL UNIVERSITY

CE4053 Embedded Operating Systems

Part 1: Task Recursion & Scheduling

Group: Swensens

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

Introduction

Many real time systems are embedded systems, and part of the ever growing infrastructure. A significant amount of them are hard real-time systems, where missing deadlines results in critical failures, which may involve human lives. µC/OS-III is a real-time operating system designed to provide a solid framework and foundation for embedded systems. According to µC/OS-III User's Manual, "Tasks typically take the form of an infinite loop". However, this does not result in the task running periodically. One can of course add timers and other OS objects to ensure periodic release, but this adds complexity to the code. Thus, this report shows a new API that embedded developers can use to simply create recursive tasks. Additionally, µC/OS-III uses a simple priority-based scheduler, which may not be suitable for all embedded applications. This report additionally shows an implementation of Earliest Deadline First scheduling.

Flow of events

1. Instead of OS_TaskCreate, OS_RecTaskCreate is called instead.
2. Creation of TCB and stack occurs as per OS_TaskCreate.
    a. Additionally, the TCB pointer is stored in the heap OS_REC_HEAP
    b. OS_RecTaskCreate exits and calls the scheduler
3. Resume creation of other recursive tasks (if applicable)
4. When OSTickCtr == 5, OS_revive_rec_task runs and goes through the whole heap and push each TCB into the ready list and AVL tree OS_AVL_TREE.
5. Run highest priority task
    a. Compare TCB that is extracted from OS (i.e. priority bitmap and ready list) and AVL tree (i.e. recursive TCB with smallest deadline).
    b. If priority of TCB extracted from OS is more than 3, next task to run is TCB extracted from AVL tree.
    c. Else, run TCB extracted from OS.
    d. Note in the background, OS_TickTask & OS_revive_rec_task is called every 1ms.
6. When the task ends, OS_RecTaskDel is called instead. This function does not delete the TCB, unlike OS_TaskDel.
    a. The TCB is also removed from the AVL tree.
7. Repeat step 5-6 until no more tasks to run OR it is time to release a recursive task to the ready list.
8. When it is time to release a task, OS_revive_rec_task runs.
    a. It peeks at the TCB at the top of the heap (i.e. the one requiring release)
    b. It refreshes the TCB, like in OS_TaskCreate, and pushes it to the ready list.
    c. The TCB is popped off the heap, and pushed into the heap again, but with the next release time.

d. The TCB is again inserted to the AVL tree.
e. Call the OS Scheduler.

9. Repeat step 5.

Task Recursion

API Functions

OSRecTaskCreate() - Recursive Task Creation

µC/OS-III creates a task through the API provided in **OSTaskCreate()**. It consists of the following arguments:

```
void  OSTaskCreate (OS_TCB        *p_tcb,
                    CPU_CHAR      *p_name,
                    OS_TASK_PTR    p_task,
                    void          *p_arg,
                    OS_PRIO        prio,
                    CPU_STK       *p_stk_base,
                    CPU_STK_SIZE   stk_limit,
                    CPU_STK_SIZE   stk_size,
                    OS_MSG_QTY     q_size,
                    OS_TICK        time_quanta,
                    void          *p_ext,
                    OS_OPT         opt,
                    OS_ERR        *p_err)
```

In order to properly implement periodicity, user needs to have the choice of entering a period value for the task. Hence, we made a new function **OSRecTaskCreate()** which includes period as an argument.

```
void  OSRecTaskCreate (OS_TCB        *p_tcb,
                       CPU_CHAR      *p_name,
                       OS_TASK_PTR    p_task,
                       void          *p_arg,
                       OS_PRIO        prio,
                       OS_PERIOD      period,
                       CPU_STK       *p_stk_base,
                       CPU_STK_SIZE   stk_limit,
                       CPU_STK_SIZE   stk_size,
                       OS_MSG_QTY     q_size,
                       OS_TICK        time_quanta,
                       void          *p_ext,
                       OS_OPT         opt,
                       OS_ERR        *p_err)
```

This period will be saved accordingly into the p_tcb, together with a new value which we called “deadline” in our code. This deadline actually refers to the next instance each task is supposed to run at, and in our case is calculated by period+OSTickCtr.

```
549     p_tcb->Period        = period;
550     p_tcb->Deadline      = period+OSTickCtr;
```

However, it is important to note that the TCB is not being pushed into the ready list for the OS Scheduler yet. This is so that we can perform a synchronous release of all the tasks later on at t=0. Also, we made use of a heap data structure to manage these recursive tasks, using it to store the respective period, deadlines and TCB of each recursive task.

```
582     REC_TASK_ARR[OS_REC_HEAP.count].period = p_tcb->Period;
583     REC_TASK_ARR[OS_REC_HEAP.count].deadline = p_tcb->Deadline;
584     REC_TASK_ARR[OS_REC_HEAP.count].p_tcb = p_tcb;
```

OS_RecTaskDel() - Recursive task deletion

In the original **OS_TaskDel()**, the TCB is reset to default values, which is not what we want when we delete a recursive task.

```
750
751     OS_TaskInitTCB(p_tcb);                                  /* Initialize the TCB to def
752     p_tcb->TaskState = (OS_STATE)OS_TASK_STATE_DEL;         /* Indicate that the task wa
753
```

This is because we are trying to keep the recursive tasks inside the heap data structure, which points to the TCB value, which if cleared, will be pointing to nothing instead. Hence, instead of calling **OS_TaskDel()**, we will need to call a new function that does a “pseudo delete”.

In this new function **OS_RecTaskDel()**, we ensure that the TCB and message queue are not reset by commenting out the code section. Hence, this ensures that after running and deleting a recursive task, the task’s TCB values are still in tact inside the heap to be recursively called later on. However, we made sure to change the state of the recursive task to be deleted inside this function, so as to prevent Micrium internals from misinterpreting the task state as ready since it still exists inside our heap structure. This task state will be changed in **OS_revive_rec_task()** later on.

```
851     //OS_TaskInitTCB(p_tcb);
852     p_tcb->TaskState = (OS_STATE)OS_TASK_STATE_DEL;
```

At OSTickCtr == 5 - Synchronous Release of Tasks

In order to implement the synchronous release of tasks into the ready list upon starting the robot, at tick counter = 5, we iterate through the heap data structure that stores all the recursive tasks, and push them into the ready list in order for the OS Scheduler to schedule them accordingly. This is done inside **OS_revive_rec_task(),** being the first thing checked.

```
88  if (OSTickCtr == 5) {
89    //  iterate through heap (size 5)
90    //  add them all into readylist at one go
91    for (int k = 0; k < 5; k++)
92    {
93      p_tcb = OS_REC_HEAP.node_arr[k]->p_tcb;
94      CPU_SR_ALLOC();
95
96      OS_PrioInsert(p_tcb->Prio);
97      OS_RdyListInsertTail(p_tcb);
```

In this code section, we push these tasks into an AVL tree for the first time, in order to manage the EDF scheduling later on as each of the task's deadline approaches, which will be explained later on.

```
100       new_avl_nodeArr[avl_count].deadline = p_tcb->Deadline;
101       new_avl_nodeArr[avl_count].p_tcb = p_tcb;
102
103       avl_insert(&OS_AVL_TREE, &new_avl_nodeArr[avl_count].avl, cmp_func);
```

Min-Heap - Recursive task management

There is a need to manage the recursive tasks that gets created. In our case, we make use of a Min-Heap to manage this. At all times, the root of the heap should be minimum value, which in our case, refers to the next timing to be released ("deadline" in code) of the TCB that is pushed inside. Hence we always compare the tick counter against the next timing belonging to the root index of the heap, to check against the task that is supposed to run next.

OS_revive_rec_task() - Periodic release of recursive tasks

OS_revive_rec_task is the function that gets called every time **OS_TickTask** is called (every 1ms). By doing so, it checks against the min-heap as mentioned above to see if the tick counter is equal to the nearest time-to-release task in the heap. This is done by comparing the tick counter to the corresponding value belonging to the first index of the heap (which will be the lowest of all elements inside, due to minimum nature of heap). (Note that this will be done after the synchronous release of all the tasks mentioned above.)

```
128   while (OSTickCtr == OS_REC_HEAP.node_arr[0]->deadline)
129   {
130     p_tcb = OS_REC_HEAP.node_arr[0]->p_tcb;
```

If this condition is true, it pseudo-creates the task by making the system p_tcb value the one the heap is pointing to, clearing and reinitialising the stack. The time to next release of the task will also be updated by Period+OSTickCtr once again. The p_tcb will then be added to the ready list.

Since the p_tcb's next time to release has just been updated, there is a need to update the this value in the heap, else this p_tcb will remain as the root node with the new value even though it might not be the lowest value anymore in the heap. Hence, we need to heapify the heap by popping the heap (essentially removing the root node, which is the task mentioned),

and pushing the same p_tcb. Also, it is important to change the task state from deleted to ready before pushing them into the ready list.

```
173         p_tcb->TaskState  = (OS_STATE)OS_TASK_STATE_RDY;

220         new_nodeArr[count].deadline = p_tcb->Deadline;
221         new_nodeArr[count].period = p_tcb->Period;
222         new_nodeArr[count].p_tcb = p_tcb;

234          heap_push(&OS_REC_HEAP,&new_nodeArr[count]);
```

Scheduling

Earliest Deadline First

µC/OS-III by default, makes use of a preemptive priority based kernel, allowing for multiple tasks to have the same priority level. The task with the highest priority will be made ready to run first. However, this may not be the best algorithm for real-time systems, with critical deadlines to meet. Thus, a better scheduling method that could be used is Earliest Deadline First (EDF) scheduling, where each instance of a task is assigned priorities based on the closeness to the deadline as compared to other tasks in the system.The task that is picked to run is the task with the closest deadline.

In this example, we will use a self-balancing binary tree, the Adelson-Velsky and Landis tree, or AVL tree. To not affect the base operations of µC/OS-III, the AVL tree will be added on top of the existing priority bitmap and ready list. In a sense, this is second ready list, specially made for EDF and will only contain TCBs of recursive tasks. The variable that is the AVL tree is labelled as OS_AVL_TREE in the code.The tree, as with the heap, is initialised in the OSInit function.

As the scheduler is very closely tied to the priority bitmap and ready list, the changes that have been made to enable EDF scheduling is in OSSched, OSRecTaskDel and OS_revive_rec_task.

In OSSched, we allow the original operations of extracting the highest priority from the priority bitmap to occur, as seen below.

```
OSPrioHighRdy   = OS_PrioGetHighest();                 /* Find the highest priority ready
OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;


/* Search for node with smallest deadline greater than 3, and get tcb  & priority*/
if (OS_AVL_TREE.root != 0)
{
  query.deadline=0;
  cur = avl_search_greater(&OS_AVL_TREE, &query.avl, cmp_func);
  node = _get_entry(cur, struct os_avl_node, avl);
  tree_smallest = node->p_tcb;
  tree_prio = node->p_tcb->Prio;
  /* Compare AVL tree and original */
  /* if more than 3, means non internal task, i.e. recursive tasks */
  if (OSPrioHighRdy > 3)
  {
    OSPrioHighRdy = tree_prio;
    OSTCBHighRdyPtr = tree_smallest;
  }
}
```

Afterwards, we conduct a search for the node with the smallest deadline. We extract the OS_TCB and its corresponding priority. If the priority retrieved from the bitmap is more than 3, this indicates that the task that is next to run is one of the application tasks, and not the internal tasks. We then overwrite the OSTCBHighRdyPtr with the TCB that was extracted from the AVL tree, to ensure that the next task to run is the recursive task with the earliest (or in this case, smallest) deadline.

In OSRecTaskDel, when the recursive task has run to completion, in addition to removing the TCB from the ready list, we also remove the TCB from the AVL tree as well. (See below.)

```
switch (p_tcb->TaskState) {
    case OS_TASK_STATE_RDY:
        /* remove node with this tcb from AVL tree */
        query.deadline=p_tcb->Deadline;
        cur = avl_search(&OS_AVL_TREE, &query.avl, cmp_func);
        avl_remove(&OS_AVL_TREE, cur);
        /* remove from ready list as well */
        OS_RdyListRemove(p_tcb);
         break;
```

In OS_revive_rec_task, when all the recursive tasks are pushed into the ready list together for a synchronous release, their TCBs are also inserted into the AVL tree.

During the periodic releases of the recursive tasks, (e.g. OSTickCtr == 5000 for LEDBlink) the TCB of the task to be released as also added into the AVL tree.

In this manner, whenever there is a change in the ready list related to the recursive tasks, the AVL will change appropriately as well. At all times, similar to the ready list, the AVL tree will hold all the TCBs that are ready to run, sorted by their deadline.

Overheads

With the addition of several functions, timing tests are required, especially in real-time systems.

Below shows the overheads for the specific functions, measured using **CPU_TS_Get32().**

Function	**Time (cycles)**	**Function**	**Time (cycles)**
OSTaskCreate	1350	OSRecTaskCreate	1350
OSTaskDel	512	OSRecTaskDel	736
heap_pop	120	heap_push	45
avl_insert()	200	avl_remove()	380
OS_revive_rec_task (synch release for 5 tasks)	3000	OS_revive_rec_task (no tasks to revive)	32
OS_revive_rec_task (PER task to revive)	2200		

OSRecTaskCreate, as compared to OSTaskCreate, they are very similar in timings as the only operations omitted in OSRecTaskCreate is the addition of the TCB to the ready list. This addition is delegated to the OS_revive_rec_task function, for synchronous release.

For OSRecTaskDel, does not execute certain lines of code in OSTaskDel to prevent the deletion of the TCB from the OS. However, the additional operation of the removal of the TCB from the AVL tree, takes up quite a lot of clock cycles and caused the time taken by the OSRecTaskDel function to increase.

For OS_revive_rec_task, there are three versions to assess. The first, during the synchronous release of all five recursive tasks, consists of setting the priorities in the bitmap, and pushing the TCBs into the ready list and AVL tree. This is relatively expensive as the data structure operations are similarly expensive. This value can be expected to increase linearly as more recursive tasks are added into the system.

The second version, is when it is not time to release a recursive task. This is short as only a statement has to be checked. The function then exits immediately.

The third version, when it is time to release a recursive task, takes a lot of clock cycles, as operations include resetting the TCB in question for running again, and the addition of the aforementioned TCB to the heap and AVL tree. Similar to during the synchronous release of tasks, the time taken would linearly increase per task that requires release at that time.

As for the operations regarding the data structures (heap and AVL tree), the number of clock cycles taken for pop (heap), push (heap), insert (AVL), remove (AVL) operations start small in the hundreds, but will generally increase as there are more nodes existing in their respective data structure. This is due to the requirements of the data structures, which require them to keep their heap property (for heap), binary tree property (AVL) and balance factor property (AVL). Keeping these properties require an increasing amount of operations when the data structure gets bigger and bigger.