Implementation of the Earliest Deadline First (EDF) Scheduling Algorithm on µC/OS-III Real-time OS

Authors:

Harri Sapto Wijaya	harri001@e.ntu.edu.sg	G1402202G
Shehzad Jalaluddin	shehzad001@e.ntu.edu.sg	G1402223L

M.Sc. in Embedded Systems, School of Computer Engineering
Nanyang Technological University, Singapore
[Edited]
As part of Embedded Operating System (ES6153) course assignment for phase-1 (18 March 2015).

Abstract:

We have implemented the Earliest Deadline First (EDF) as a Job-Fixed Priority scheduling algorithm on uC/OS-III real-time OS. Original fixed-priority-based, or Rate/Deadline Monotonic, scheduling algorithm in existing uC/OS-III was unable to meet the deadline of some tasks in given application. Our implemented EDF able to schedule those tasks in given application to meet their deadlines successfully. The implemented EDF was designed to work together with existing fixed-priority-based scheduling algorithm in uC/OS-III, hence minimize the impact on system tasks. Only if the task is opted-in to be scheduled under EDF, via task's option, then it will be managed under EDF, and queuing in a separated EDF List instead of existing Ready List. Various tests has been performed to test the correctness and generality of our implementation.

Keywords: scheduling, real-time OS/kernel, JFP, EDF, uC/OS-III.

1. Introduction

RM (or DM) has simpler implementation and predictability during overload (though only true for highest priority task); on the other hand, EDF allows a full processor utilization and better responsiveness of aperiodic activities [1]. While RM scheduling algorithm is widely used in industry, it might not meet certain applications' requirement. For example, the automotive softwares are mixed of the safety-critical control applications with stringent stability and performance requirement and the time-critical applications with stringent deadline constraint [2].

In this project, we will implement the EDF scheduling algorithm on the uC/OS-III for the deadline-sensitive application which fail to be scheduled under RM scheduling algorithm.

2. Scheduling algorithm on uC/OS-III

uC/OS-III is a preemptive, priority-based kernel. The part of the dispatcher in Micrium is responsible for determining which task to run next with respect to the appropriate priority assigned to each task. When an important (higher-priority) task made ready-to-run by an event or sending a message, uC/OS-III immediately switch to the higher-priority task and the control of the CPU is given to that task. Similar effects have been seen when a message is send by an ISR to a higher-priority task.

Task that are made ready-to-run goes into a predefined priority based array of Ready List, in which each task control block (TCB) is attached with respect to its priority assigned to it. Each task that made ready-to-run state is attached to the end of the list of tasks at that priority level assigned to it.

By initializing the Ready List, it makes some internal higher-priority task with fixed priorities, the Idle-task is assigned on [os_cfg_prio_max-1] as well as the [0] priority is assigned to the ISR handler task if the ISR deferred configuration is set to to 1, the doubly

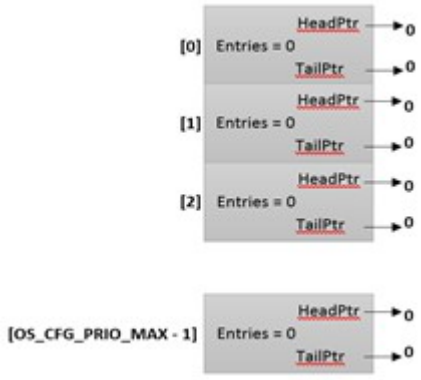

Figure 1: Array of linked list in Ready List.

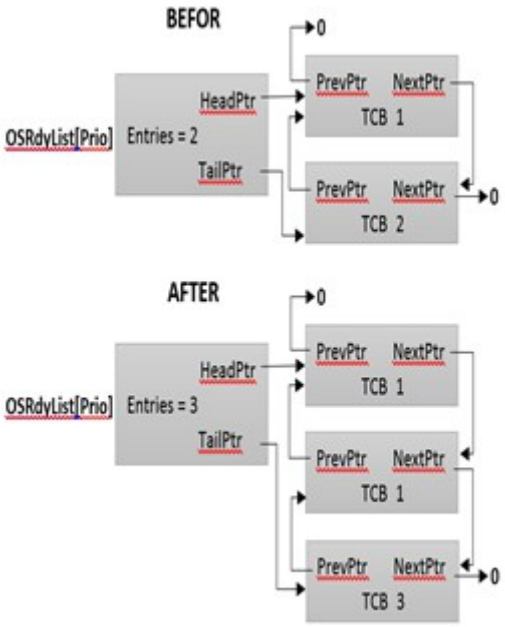

Figure 2: Inserting a new TCB at the end of Ready List.

linked list pointers of the Idle-task TCB is always points to NULL. The Tick-task, Timer-Task and Statistic-Task are also made ready-to-run state after the initialization with priority level assigns to them, typically, the priority of the tick task is always higher than the priority of timer task and that the timer task priority is always higher than the priority of statistic task.

The dispatcher selects the task by scanning the bitmap that contains all the priorities for ready-to-run state tasks, the width of the bitmap depends on the processor that usually varies between 8-32 bits, the scheduler picks the highest priority task with its corresponding bit set to 1 in the bitmap and Count Leading Zeros on that index to get the highest priority. The search on the bitmap has been done to ensure optimized searching time.

Dispatcher handles the events placement from interrupts either direct or differed waiting methods, if the task is preempted by the ISR and eventually an event made a higher-priority task ready-to-run the CPU vectors to serve the highest-priority task as in the deferred post uC/OS-III makes a special queue called ISR Handler Task ready-to-run and the uC/OS-III switches to the higher-priority task. Scheduling also occurs based on different conditions (Scheduling Points) by which uC/OS-III automatically switches to run the higher-priority tasks:

- Task sends messages to another task.
- Calling time delay events.
- Task waits for an event that yet to occur.
- Task aborts the wait of other tasks.
- Task is created.
- Task is deleted.
- Task changes priority of itself or another task.
- Task suspend itself.
- Task resumes another task.
- End of all nested interrupts.
- Dispatcher unlocked.
- User calls the scheduler.

3. Earliest Deadline First (EDF) scheduling algorithm

3.1. Problem analysis of provided application

The provided application consist of 3 main tasks (excluding system tasks and repeater tasks) run-to-completion type, labeled as T1, T2, and T3, with the characteristic of:

```
T1 (prio=12, T=3, C=1),
T2 (prio=11, T=5, C=1),
T3 (prio=10, T=5, C=2);
```

prio denotes the priority, *T* denotes the period of repeated task creation (in this application the relative deadline *D* is considered to be the same as the period), and *C* denotes the cost or execution time. Figure 3 show the theoretical (or ideal) time line of those 3 tasks. It shown that while T2 and T3 are meeting their deadline, T1 is suffering from missing the deadline.

The actual time line of running those 3 tasks on uC/OS-III is shown in Figure 4. While the trend of tasks execution order and deadline missing are similar, there are exist the differences. The first different is due to *the execution time different* from ideal case, which could be contributed from misadjustment of delay for-loop in applications, dynamic of the hardware (e.g. variation time in receiving the I2C status when displaying to the LCD), additional code for logging feature, etc. The second different is due to *the repeater task will repeat at faster rate once detect a deadline missed*. This was done to ensure that a deadline missed do not postpone a next task creation for too long (i.e. deadline-missed-task's period), thus repeated at faster rate to re-create the task as soon as possible (while considering switching overhead) once its get deleted. Hence we can see from Figure 4, once T1 has completed after missing its deadline, T1 is not immediately re-created thus Idle Task appear from time stamp 3.816 since no others tasks are

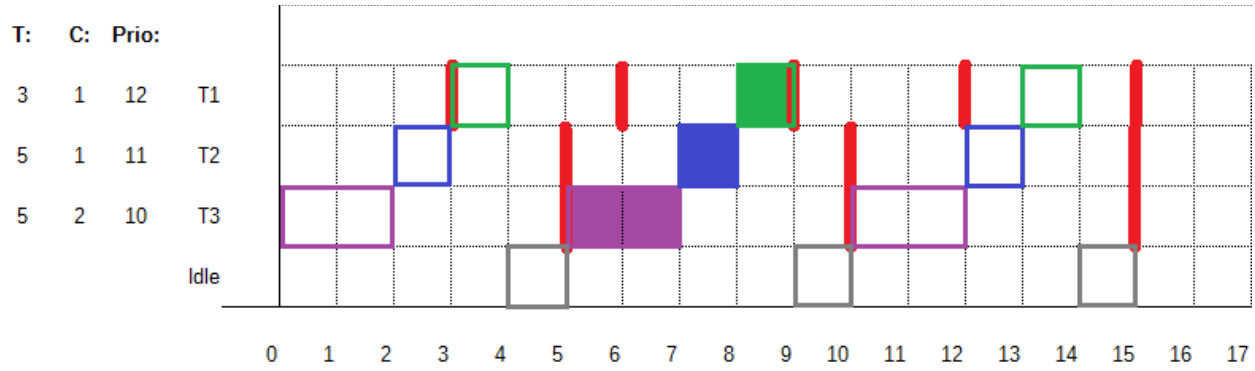

Figure 3: Theoretical time line of provided application under Task-Fixed Priority (TFP) algorithm.

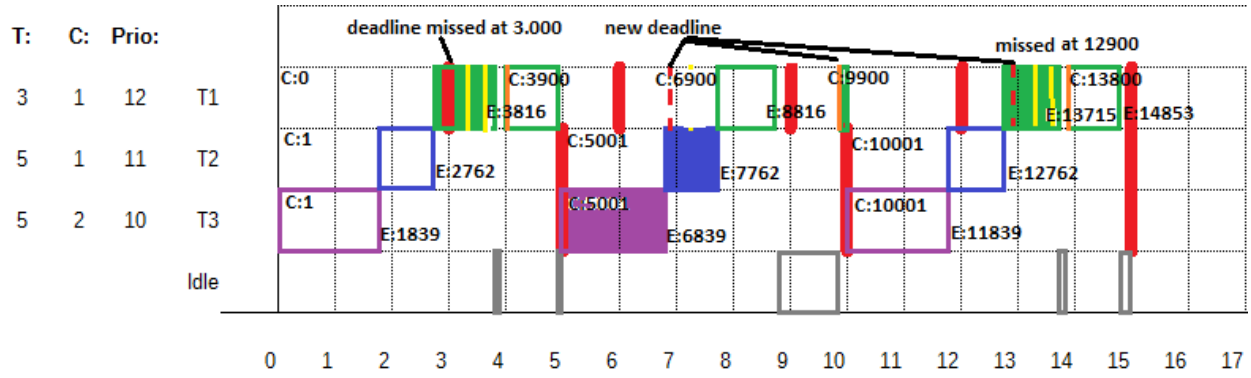

Figure 4: Actual time line of provided application running on uC/OS-III (which implement TFP algorithm). No SD Card.

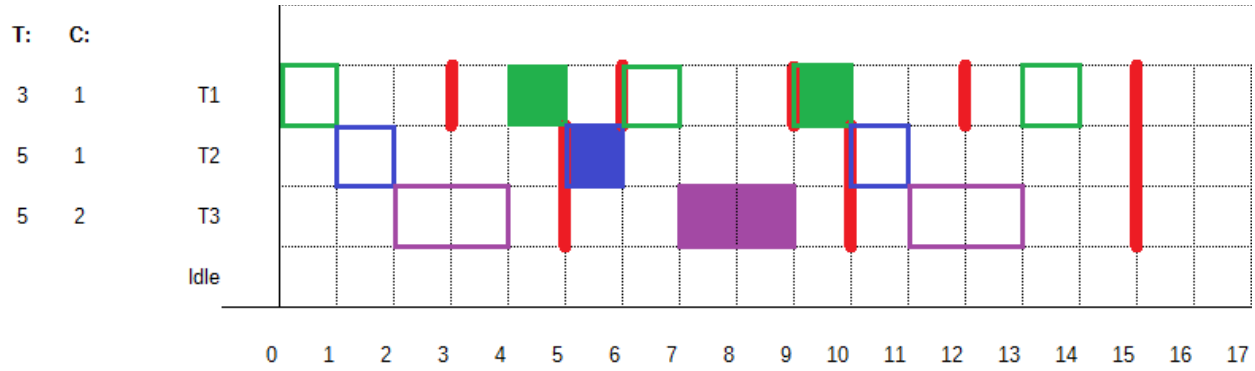

Figure 5: Theoretical time line of provided application under Earliest Deadline First (EDF) algorithm.

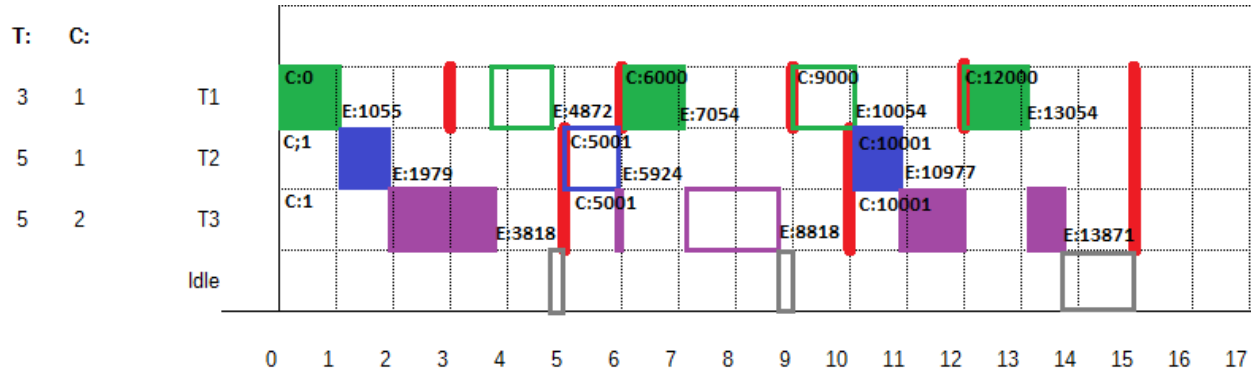

Figure 6: Actual time line of provided application running on modified uC/OS-III with EDF algorithm. No SD Card.

ready. The T1 is re-created at time stamp 3.900 (min(n) for 3.000+n0.300 > 3.816).

As briefly mentioned that the deadline missed was due to the fact that even though the deadline of T1 is more tight but its assigned priority is low, and current scheduling algorithm in uC/OS-III is a Task-Fixed Priority algorithm. Hence implies that to improve the result, i.e. avoid deadline missed, uC/OS-III scheduling algorithm must be aware of the task's deadline and ordering the tasks base on their deadline. Thus the priority must be adjusted at each job/instance level.

3.2. The EDF algorithm and design

As the name suggested, in EDF scheduling algorithm, ***the tasks are ordered in the list base on their deadline***. The list could be reusing existing ready list or separated ready list for EDF. As generalization, let label the list which has deadline-based-ordering as ***EDF List***. The assignment of the deadline is done in task creation (i.e. job/instance) for task type of run-to-completion (as in this project); for task type of infinite loop (or *sporadic task*), a new API might need to be inserted just after the pending point (e.g. pending for semaphore or message) within the task entry function as the offset of deadline and to trigger the reordering of EDF List.

In provided application, under Task-Fixed Priority (which currently implemented in uC/OS-III) those 3 tasks' instances will always be ordered in same order in ready list, i.e. {T3, T2, T1}, while under Job-Fixed Priority (e.g. EDF) the order in EDF List could vary at run-time. For example, at time stamp 0.0, the order is {T1 (D=3.0), T2 (D=5.0), T3 (D=5.0)} (or {T1, T3, T2} depend on whether ordering in FIFO of arrival time is used or not), but at time stamp 3.0, even though previously T1 has “higher priority” it cannot preempt the T3, since at time stamp 3.0 the T1 has deadline of 6.0 while T3 has deadline of 5.0 hence the EDF List is {T3, T1}, hence now T3 has “higher priority” than T1. Figure 5 show the time line of the 3 tasks under EDF algorithm, it is shown that no deadlines are missed.

In order to minimize the impact of new scheduling algorithm to the system tasks (Tick Task, Timer Task, Idle Task, etc.), we design the algorithm and data structure for the EDF implementation as the following.

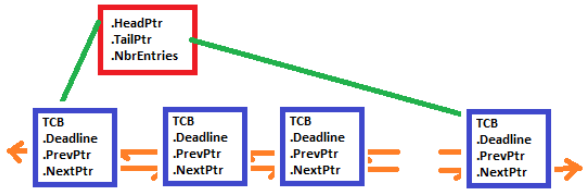

Figure 7: Data srcuture for EDF List.

a) List allocation and the TCB

The list for the EDF will be ***separated*** from existing Ready List. The task that selected to be scheduled under EDF will goes to the EDF List, otherwise will goes to the Ready List. In order to know which task is opt-in for EDF, new ***task option*** need to be added, e.g. `OS_OPT_TASK_EDF`. The task's ***deadline*** will need to be stored in the TCB (Task Control Block) as well, and will only be used if task's option contain the `OS_OPT_TASK_EDF` option.

b) List data structure and its functions

Many consideration in choosing appropriate data structure for EDF List are (1) should we statically allocate the priority/deadline, (2) represented using array or linked list, and (3) should we group together the TCB (i.e. representation of the task instance) of same priority/deadline.

While Ready List statically allocated the priorities in array (1 array index representing fixed predefined priority level, usually same as array index), it is possible since we can limit the priority level. But in deadline-based-priority, we could not

Table 1*: Compare ring buffer versus linked list for EDF List*

Ring buffer	**Linked list**
Pros.: Faster in finding the node (can use bisection search instead of scan one-by-one).	*Pros.*: Shifting is not required when inserting new node.
Cons.: Shifting is required when inserting new node (since placed adjacent to each others).	*Cons.*: Slower in finding the node (scan one-by-one is required).
Complexity of inserting (in term of write): O(n)	*Complexity of inserting (in term of write)*: search time + O(1)
Complexity of finding: O(log n) [3]	*Complexity of finding*: O(n)

```
void  OSTaskCreate (OS_TCB          *p_tcb,
                    CPU_CHAR        *p_name,
                    OS_TASK_PTR      p_task,
                    void            *p_arg,
                    OS_PRIO          prio,
                    OS_TICK          deadline,
                    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)
```

```
OSTaskCreate((OS_TCB     *)&AppTaskOneTCB,
             (CPU_CHAR   *)"App Task One",
             (OS_TASK_PTR ) AppTaskOne,
             (void       *) 0,
             (OS_PRIO     ) APP_TASK_ONE_PRIO,
             (OS_TICK     ) (TASK1PERIOD * 1000),   --> relative deadline
             (CPU_STK    *)&AppTaskOneStk[0],
             (CPU_STK_SIZE) APP_TASK_ONE_STK_SIZE / 10u,
             (CPU_STK_SIZE) APP_TASK_ONE_STK_SIZE,
             (OS_MSG_QTY  ) 0u,
             (OS_TICK     ) 0u,                                          --> opt-in for EDF
             (void       *) (CPU_INT32U) (TASK1PERIOD*1),
             (OS_OPT      ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR | OS_OPT_TASK_EDF),
             (OS_ERR     *)&err);
```

Figure 8: Task Create API and how to use it for EDF.

limit the deadline, or more precisely we can limit the deadline if needed but on how to discretize the space in between is tricky. For example, in fixed priority, in between priority of 4 and 8 we can be sure to have 3 spaces (for 5, 6, and 7); but in deadline-base-priority, in between 1000 ticks and 2000 ticks, we cannot be sure how to discretize in between, whether discretize for 1 tick (1001, 1002, 1003, …, 1998, 1999) or bigger chunk (1100, 1200, …, 1800, 1900), and how to handle insertion of 1100 and 1999 ticks deadline. Hence we decided to do not statically allocate the priority/deadline, but we just put them ***sorted adjacent*** to each others and ***no limitation in deadline values***.

After we consider to put adjacent to each others, next question is whether to represent it as *ring buffer* or *linked list* (bare array cannot be used unless we limit the deadline values). Table 1

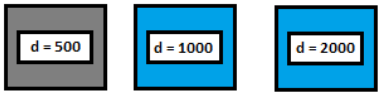

Figure 10: insertion of new TCB with sortest deadline. Head pointer is changed.

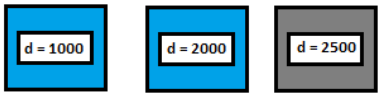

Figure 11: insertion of new TCB with longest deadline, or same as Tail's deadline. Tail pointer is changed.

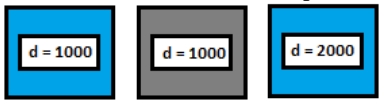

Figure 12: insertion of new TCB with deadline in-between Head's and Tail's deadline, or same with Head's or existing deadline. No change in Head and Tail pointer.

OSLogSwitchDbg[49].Event	34
OSLogSwitchDbg[49].TaskTCB	AppTaskThreeTCB (0x20002AE4)
OSLogSwitchDbg[49].TaskName	0x83B4 "App Task Three"
OSLogSwitchDbg[49].TickCtr	13054
OSLogSwitchDbg[49].Deadline	15001
OSLogSwitchDbg[50].Event	51
OSLogSwitchDbg[50].TaskTCB	AppTaskThreeTCB (0x20002AE4)
OSLogSwitchDbg[50].TaskName	0x83B4 "App Task Three"
OSLogSwitchDbg[50].TickCtr	13871
OSLogSwitchDbg[50].Deadline	15001
OSLogSwitchDbg[51].Event	34
OSLogSwitchDbg[51].TaskTCB	OSIdleTaskTCB (0x20002D24)
OSLogSwitchDbg[51].TaskName	0x835C "uC/OS-III Idle Task"
OSLogSwitchDbg[51].TickCtr	13871
OSLogSwitchDbg[51].Deadline	0
OSLogSwitchDbg[52].Event	34
OSLogSwitchDbg[52].TaskTCB	AppTaskRepeatOneTCB (0x20002B74)
OSLogSwitchDbg[52].TaskName	0x83C4 "Repeat Task One"
OSLogSwitchDbg[52].TickCtr	15000
OSLogSwitchDbg[52].Deadline	0

Figure 9: Example content of the logging array.

summarize the comparison. Since insertion (and removal) have more action in 1 iteration (i.e. copy the while structure fields) compared to finding (1 or 2 read action), we decided to use ***linked list*** due to small complexity in inserting, i.e. no shifting required.

Now we have decided to put the TCB in EDF List sorted adjacent to each others in linked list, shall we group together the TCBs with same deadline (just like in Ready List, which now will required 2D linked list) or not (just normal 1D linked list). Since we cannot do bisection search in linked list, there is no benefit in doing such grouping. Hence we decided to lay down all TCBs ***linearly*** in a 1D linked list, be it same or different deadline, as shown in Figure 7.

c) Integration with existing scheduler

We have decided to create separated EDF List from existing Ready List. Integration of both lists in the priority-based scheduler is done such these.

- Both Ready List and EDF List produce pointer to each *highest priority* (via bitmap) and *shortest deadline* (simply head of the list) TCB.
- The priority of any TCB from EDF List is decided to be ***fixed*** (predefined value).
- Scheduler *compare* which TCB has higher priority, then switch to that.

The predefined priority for any TCB of EDF List is defined in `OS_CFG_EDF_PRIO` in os_cfg.h.

d) Functions on EDF List

There are only 3 functions to modify EDF List as following.

`OS_EDFListInit()`: to initialize the EDF List, called once from `OSInit()`.

`OS_EDFListInsert()`: to insert a TCB in the

EDF List, including sorting it in incrementing deadline order. Figure Figure 10 to 12 illustrate various scenario on inserting new TCB in the list. Note that when new TCB with deadline is same as existing ones, it will be placed after the last same deadline, i.e. FIFO order of arrival time.

`OS_EDFListRemove()`: to search and remove a TCB from EDF List.

3.3. Implementation in uC/OS-III

This section will summarize the modification made to the uC/OS-III in order to implement EDF scheduling algorithm. The code is available in `ssh://git@codeventure.sce.ntu.edu.sg:7999/~harri001/newyear.git`.

a) TCB

We add one extra field in TCB to hold the value of the ***deadline***. This field will only be used if it is specified in the *option bit* (`OS_OPT_TASK_EDF`) that the task is opted-in for EDF scheduling algorithm, otherwise unused.

b) Task Creation

On top of existing task creation, if the `OS_OPT_TASK_EDF` option bit is set, instead of goes to the Ready List, the ***TCB will goes to the EDF List*** via `OS_EDFListInsert()`, and its priority will be set to fixed-predefined value of `OS_CFG_EDF_PRIO`. The deadline will be set to `OSTickCtr + deadline`, i.e. current tick value plus supplied relative deadline.

c) Task Deletion

On top of existing task deletion, it will remove the TCB from Ready List or from EDF List (via `OS_EDFListRemove()`) depending of the `OS_OPT_TASK_EDF` option bit of the TCB-to-be-deleted, whether clear or set.

d) Scheduler

In `OSSched()` and `OSIntExit()`, it will ***produce highest-priority-TCB from Ready List*** as per normal, and also ***shortest-deadline-TCB from EDF List***. ***Compare the priority*** of both TCB, whichever higher (smaller in numerical value) will be put in `OSTCBHighRdyPtr` to let the `OS_CPU_PendSVHandler` (in os_cpu_a.asm) context switch to it. Hence, we still keep the task in current TFP scheduling, unless the task opted-in for EDF. Note that the system tasks are still in current existing TFP scheduling,, while it is possible to include them in EDF algorithm, we decided to keep them unmodified.

e) API usage in application

Figure 8 show the API modification to the OSTaskCreate() and how to use it to include the to-be-created task under EDF scheduling algorithm.

4. Experiments and Results

4.1. Task switch logging feature

In order to be able to observe the tasks execution (especially the task switching), we created a logging feature to log into array some of the TCB information whenever task switching occurred. Figure 9 Show the example content of the logging array, with event 34 meaning switched-into, 51 meaning task completed/deleted, and 17 meaning task created. The task's name and tick stamp are also logged.

We do selective logging to exclude certain system task, i.e. Tick Task and Timer Task, but do allow Idle Task; otherwise the log will be full quickly since there is task switching to-and-from Tick Task every tick. To complete this exclusion, we also exclude if the task-to-be-switched-in is same as previous index in the log. So, we know when the task start, when it is preempted (if happen), and when it is ended.

4.2. Test on provided application

Figure 6 shown the actual time line of provided application running on modified uC/OS-III with EDF scheduling algorithm. The actual time line is very similar as expected by theoretical time line, except at tick stamp 12'000. At that tick stamp, instead of continue running, T3 is preempted by T1, although at tick stamp 12'000 all the tasks' instances should have same deadline. However the fact that T2 and T3 is created 1 tick later than T1 has drifted the deadlines of T2 and T3 by 1 tick. So at that time, the deadline of T2 and T3 is 15'001 ticks (10'001+5'000) while the deadline of T1 is 15'000 ticks (12'000+3'000), i.e. earlier than T3, thus T1 preempt T3. If necessary, this problem can be avoided by limit the resolution of the deadline, hence get the effect of low-pass filter. For example to the resolution of 10 ticks, so 121 and 122 ticks deadline will be considered as same 120 ticks deadline.

Figure 13 shown the actual time line when writing to SD Card for logging is enabled. The

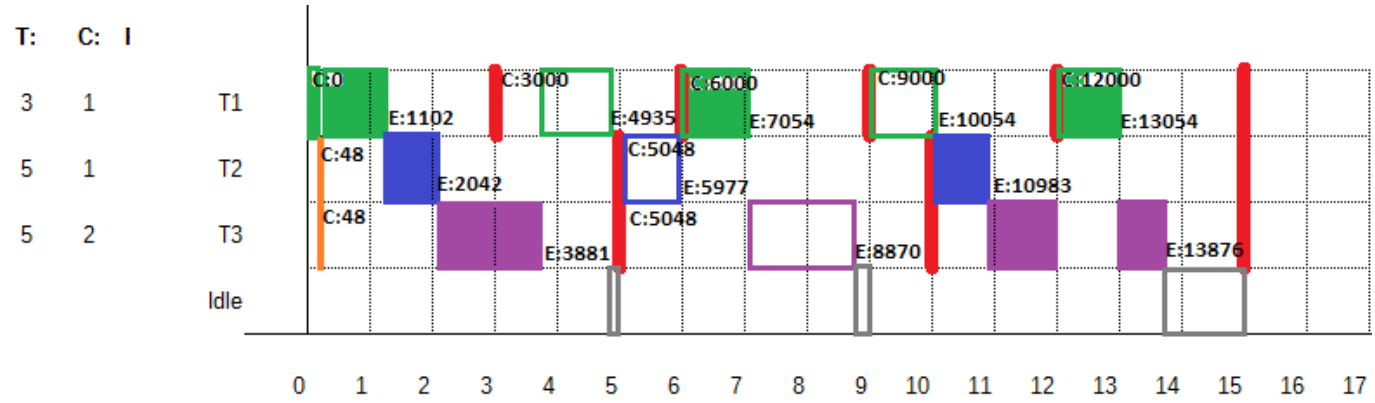

Figure 13: Actual time line of provided application running on modified uC/OS-III with EDF algorithm. With SD Card.

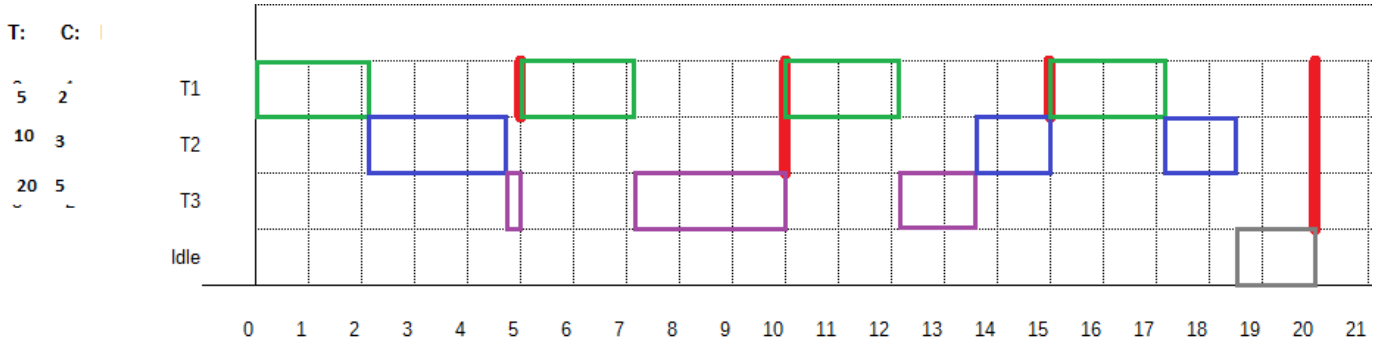

Figure 14: Actual time line of application from Tutorial 4 Question 1. No SD Card.

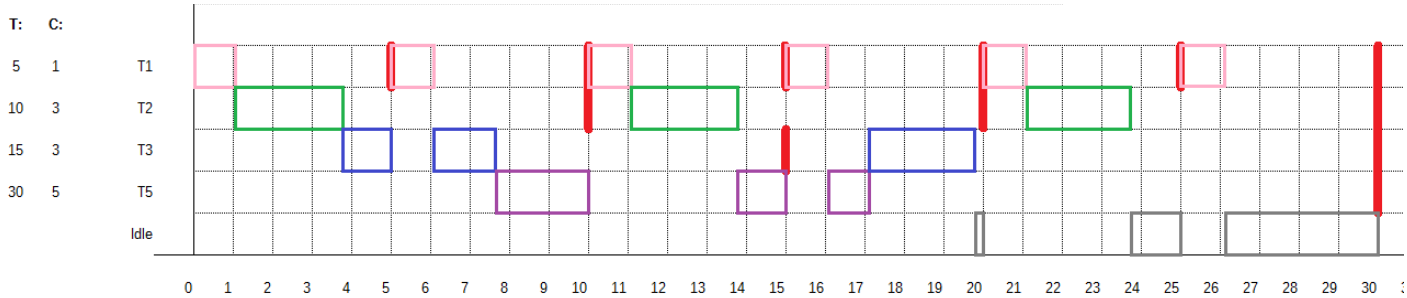

Figure 15: Actual time line of the application with 4 tasks. No SD Card.

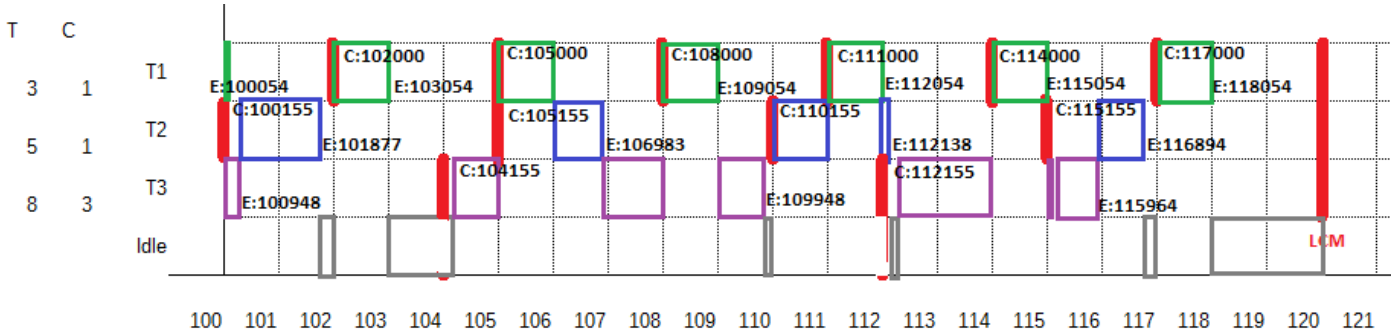

Figure 16: Actual time line of the Test1 application. With SD Card.

drift is increased from 1 ticks to about 48 ticks. However the implemented EDF scheduling algorithm still able to schedule those 3 tasks to meet their deadlines, either relative to creation time or relative to absolute 0 ticks.

4.3. Test on modified application

In order to test the implemented algorithm more, we took example scenario from from Tutorial 4 Question 1 where there are 3 tasks with the following characteristic.

```
T1 (T=5,  C=2),
T2 (T=10, C=3),
T3 (T=20, C=5);
```

Figure 14 show the actual time line result. Note there is small different in actual execution time, a little longer for T1 and a little shorter for T2, resulting T3 to be run before tick stamp 5'000. The causes of this different has been discussed in Section 3.1.

4.4. Test on more number of tasks

In order to test the generality of our implementation, we also test on the application with more number of tasks. Due to code size constraint, we only able to create 1 more task, hence we tested on application with 4 tasks with the characteristic as the following.

```
T1 (T=5,  C=1),
T2 (T=10, C=3),
T3 (T=15, C=3);
T5 (T=30, C=5);
```

Figure 15 show the actual time line result. Note that the theoretical time line might predict more than single time line, due to when there are 2 or more tasks with same priority or deadline, it is depend on the implementation to choose which task to run first, e.g. whether the ordering is considering FIFO with respect to arrival time. Another factor is as has been discussed in in Section 3.1.

4.5. Test on provided Test1 application

The **Test1** application has been provided to test more on the implemented algorithm. The characteristic of tasks in Test1 is the following.

```
T1 (T=3, C=1),
T2 (T=5, C=1),
T3 (T=8, C=3);
```

Due to the LCM of those tasks is quite big, i.e. 120 ticks, we only draw the time line near to the tick amount of LCM. The reason is because when there is deadline miss, the scenario near LCM will be suffered more due to shifted deadline-relative-to-task-creation from deadline-relative-to-0-ticks. The result near LCM is shown in Figure 16. It also show that no deadline missed with the provided Test1 application.

EDIT: additional implementation.
Problem: it is required to follow the deadline relative to the absolute-0-ticks **not** to the release time, otherwise when release time is shifted so does the deadline will be shifted all the way. Originally we thought this is not a problem.
Solution: modify deadline calculation in OSTaskCreate() to ensure the deadline is relative to the absolute-0-ticks, i.e. multiple of relative-deadline, hence to absorb the shift time.
Commit: `0098250` done in the lab during the demo.

5. Conclusion and Discussion

The Task-Fixed Priority, or DM/RM, could not guarantee the tasks to meet their deadline, since they are agnostic of the deadline and use fixed priority for all its instances all the time. While Job-Fixed Priority taking care of the task's deadline and adjust the “priority” of its instance at every creation (for task type of run-to-completion), hence it able to schedule the tasks to meet their deadline. Our implemented ***Earliest Deadline First*** (EDF) scheduling algorithm is able to schedule the provided tasks to meet their deadlines successfully.

The implemented EDF scheduling algorithm has ***tested to be able to handle various scenarios***, including various tasks' characteristic (period and deadline), various overhead and time creation drift (enabling of SD Card writing feature), as well as various number of tasks to be managed under EDF scheduling algorithm.

The implemented EDF could be fine tuned to be more robust to small variation in deadline due to time creation drift by limiting the resolution of the deadline. When the number of tasks to be managed is huge, the EDF List could be implemented using ring buffer instead of linked list to reduce the complexity in TCB insertion by utilizing the bisection search but shifting will still be needed.

The Task Switch Logging feature can also be improved. In current implementation, if a task is created from highest priority task in ready list, after the creation the switch will not happen, i.e. immediate exit, hence that task creation event is not logged under that condition.

References:

[1] G. Buttazzo, “Rate Monotonic vs. EDF: Judgment Day”, Real-Time Systems Journal: 29(5-26), 2005.

[2] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakrabothy, “Time-triggered Implementations of Mixed-Criticality Automotive Software”, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012.

[3] http://bigocheatsheet.com/