ECE 5550G - Advanced Real-Time Systems
Final Project

Implement, test, and compare EDF against RM and DM.

Team Project By,

Vinodh Kumar Reddy Kamasani

Punitha Kamasani

What is this project about?

- Introduction to RM, DM and EDF
- Comparing the theoretical Implementation (RM, DM and EDF)
- Comparing various parameters like Runtime overhead, CPU Utilization and deadlines miss theoretically.
- Implementation of RM, DM, and EDF in FreeRTOS
- Simulation in Arduino IDE using FreeRTOS Library
- Observations and Conclusion

Introduction to RM, DM and EDF

Rate Monotonic Scheduling (RMS)

- **Type:** Static priority scheduling
- **Priority Assignment:** Priorities are assigned based on the task's period; tasks with shorter periods receive higher priority.
- **Pros:** Simple and predictable; optimal among fixed-priority preemptive scheduling algorithms for independent periodic tasks.
- **Cons:** Not as flexible as EDF, can underutilize CPU as task sets become more complex.

Deadline Monotonic Scheduling (DMS)

- **Type:** Static priority scheduling
- **Priority Assignment:** Priorities are assigned based on deadlines; tasks with shorter deadlines receive higher priority.
- **Pros:** More flexible than RM when tasks have deadlines that are not equal to their periods.
- **Cons:** Like RM, DM can also suffer from CPU underutilization in complex task scenarios.

Earliest Deadline First Scheduling (EDF)

- **Type:** Dynamic priority scheduling
- **Priority Assignment:** Priorities are assigned based on the deadlines; tasks with the earliest deadlines are given higher priority.
- **Pros:** Optimal among dynamic priority algorithms under preemptive scheduling, meaning it can schedule any set of tasks that are feasible.
- **Cons:** Higher runtime overhead due to dynamic priority calculation, and can be less predictable in practical real-time systems.

Comparing the theoretical Implementation (RM, DM, and EDF)

Example taskset:

Let's Consider a simple taskset to theoretically compare EDF, RM, and DM (all three scheduling algorithms can successfully schedule this task set)

Task	C	D	T
τ_1	100	400	400
τ_2	200	700	800
τ_3	150	1000	1000
τ_4	300	5000	5000

RM Scheduling:

Processor utilization, **U =** $\sum\left(\frac{C_i}{t_i}\right) = \frac{100}{400} + \frac{200}{800} + \frac{150}{1000} + \frac{300}{5000} = \frac{71}{100}$ = 0.71 or 71%

The priority of a task is inversely proportional to its period

Priority -> $\boldsymbol{\tau_1 > \tau_2 > \tau_3 > \tau_4}$

DM Scheduling:

Processor utilization, **U =** $\sum\left(\frac{C_i}{D_i}\right) = \frac{100}{400} + \frac{200}{700} + \frac{150}{1000} + \frac{300}{5000} = \frac{261}{350}$ = 0.74 or 74 %

Priority of a task is inversely proportional to its deadline

Priority -> $\boldsymbol{\tau_1 > \tau_3 > \tau_2}$

EDF Scheduling:

Processor utilization, **U =** $\sum\left(\frac{C_i}{t_i}\right) = \frac{100}{400} + \frac{200}{800} + \frac{150}{1000} + \frac{300}{5000} = \frac{71}{100}$ = 0.71 or 71%

Priorities are assigned based on the closest upcoming deadline

Priority -> **Dynamic priority**

Comparing the Parameters between EDF, RM and DM

1. CPU Utilization

CPU Utilization refers to the proportion of time the CPU is actively executing tasks as opposed to being idle. Higher utilization implies more efficient use of CPU resources.

Rate monotonic (RM) and Deadline monotonic (DM)

Utilization: As calculated previously, the total utilization for the task set is U=0.71. RM and DM are generally limited by bound($n(2^{1/n}-1)$), which is around 0.756 for four tasks. Hence, both RM and DM make efficient use of CPU resources up to their theoretical limits.

Earliest deadline first (EDF)

Utilization: EDF can theoretically utilize 100% of the CPU ($U \leq 1 U \leq 1$) without missing deadlines, assuming ideal conditions without context switching overheads. Therefore, EDF is potentially more efficient in terms of raw CPU utilization compared to RM and DM.

2. Deadline miss rate

Deadline miss rate measures the frequency of tasks failing to meet their deadlines. Lower rates are desirable as they indicate higher reliability

RM, DM, and EDF

Given that our prior calculations and theoretical conditions (ignoring context switches and other practical considerations) suggest that all tasks meet their deadlines under the CPU utilization provided, the deadline miss rate of RM, DM, and EDF would be zero for this particular task set.

3. Scalability

Scalability refers to the ability of the scheduling algorithm to effectively handle increasing numbers of tasks or changes in the characteristics of the task set (e.g., periods, execution time)

RM and DM

Scalability: RM and DM scalability is generally limited due to the fixed priority scheme. As the number of tasks increases, the probability that a low-priority task will miss its deadline also increases due to the “priority inversion” phenomenon. They scale less effectively when the system complexity or number of tasks increases beyond certain bounds.

EDF

Scalability: EDF typically scales better than RM and DM in complex systems because it dynamically assigns priorities based on deadlines, reducing the probability of a deadline miss as task count increases. However, it can suffer from increased overhead due to frequent priority recalculations.

4. Predictability

Predictability is the degree to which future system behavior (like task execution and completion) can be accurately predicted based on known parameters and past performance.

RM and DM

Predictability: RM and DM offer high predictability due to their fixed priority assignment. The system behavior can be more easily analyzed and predicted because the interference patterns from higher priority tasks are consistent and can be calculated.

EDF

Predictability: EDF, while more flexible and potentially more efficient, offers lower predictability compared to RM and DM. This is because priorities can change dynamically based on the deadline landscape at any given moment, making worst-case execution time analysis and interference analysis more complex.

RM Implementation in FreeRTOS

Implementing macros in schdeduler.h file:

#define schedSCHEDULING_POLICY_RMS 1

This line defines a macro schedSCHEDULING_POLICY_RMS with a value of 1. This typically represents the Rate Monotonic Scheduling (RMS) policy. Assigning it the number 1 makes it a unique identifier for this scheduling policy within the system.

#define schedSCHEDULING_POLICY schedSCHEDULING_POLICY_RMS

This line sets the scheduling policy of the system to Rate Monotonic Scheduling by using the value associated with schedSCHEDULING_POLICY_RMS.

Implementation in scheduler.cpp file:

```
// Define a function to set fixed priorities for tasks
static void prvSetFixedPriorities( void )
{
   // Record the start time for measuring function execution time
   TickType_t StartTime = xTaskGetTickCount();
   BaseType_t xIter, xIndex; // Loop counters for iterating through tasks
   TickType_t xShortest, xPreviousShortest=0; // Track the shortest period/deadline found
   SchedTCB_t *pxShortestTaskPointer, *pxTCB; // Pointers to task control blocks

   // Determine the highest priority that can be assigned to tasks
   #if( schedUSE_SCHEDULER_TASK == 1 )
      BaseType_t xHighestPriority = schedSCHEDULER_PRIORITY;
```

```
    #else
        BaseType_t xHighestPriority = configMAX_PRIORITIES;
    #endif

    // Iterate over all tasks to set their priorities
    for( xIter = 0; xIter < xTaskCounter; xIter++ )
    {
        xShortest = portMAX_DELAY;

        // Search for the task with the shortest period/deadline
        for( xIndex = 0; xIndex < xTaskCounter; xIndex++ )
        {
            pxTCB = &xTCBArray[ xIndex ];
            configASSERT( pdTRUE == pxTCB->xInUse ); // Ensure task is in use

            // Skip tasks that already have their priorities set
            if(pdTRUE == pxTCB->xPriorityIsSet)
            {
                continue;
            }

            // Additional logic to update xShortest and pxShortestTaskPointer...
        }
    }
}
#if( schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_RMS )
if( pxTCB->xPeriod <= xShortest )
{
xShortest = pxTCB->xPeriod;
pxShortestTaskPointer = pxTCB;
}
```

if(pxTCB->xPeriod <= xShortest): This line performs a comparison between the period of the current task (accessed through the task control block pointer pxTCB->xPeriod) and the current shortest period found (xShortest).

- xShortest = pxTCB->xPeriod;: Update xShortest to this task's period, as it is the new shortest period found.
- pxShortestTaskPointer = pxTCB;: Set pxShortestTaskPointer to point to this task's control block (pxTCB), marking it as the task with the current shortest period. This pointer may later be used to assign the highest priority to this task.

DM Implementation in FreeRTOS

Implementing macros in schdeduler.h file:

#define schedSCHEDULING_POLICY_DMS 2

This line defines a macro schedSCHEDULING_POLICY_DMS with a value of 2. This typically represents the Deadline Monotonic Scheduling (DMS) policy. Assigning it the number 2 makes it a unique identifier for this scheduling policy within the system.

#define schedSCHEDULING_POLICY schedSCHEDULING_POLICY_DMS

This line sets the scheduling policy of the system to Deadline Monotonic Scheduling by using the value associated with schedSCHEDULING_POLICY_DMS.

Implementation in scheduler.cpp file:

```
// Define a function to set fixed priorities for tasks
static void prvSetFixedPriorities( void )
{
   // Record the start time for measuring function execution time
   TickType_t StartTime = xTaskGetTickCount();
   BaseType_t xIter, xIndex; // Loop counters for iterating through tasks
   TickType_t xShortest, xPreviousShortest=0; // Track the shortest period/deadline
found
   SchedTCB_t *pxShortestTaskPointer, *pxTCB; // Pointers to task control blocks

   // Determine the highest priority that can be assigned to tasks
   #if( schedUSE_SCHEDULER_TASK == 1 )
      BaseType_t xHighestPriority = schedSCHEDULER_PRIORITY;
   #else
      BaseType_t xHighestPriority = configMAX_PRIORITIES;
   #endif

   // Iterate over all tasks to set their priorities
   for( xIter = 0; xIter < xTaskCounter; xIter++ )
   {
      xShortest = portMAX_DELAY;

      // Search for the task with the shortest period/deadline
      for( xIndex = 0; xIndex < xTaskCounter; xIndex++ )
      {
         pxTCB = &xTCBArray[ xIndex ];
         configASSERT( pdTRUE == pxTCB->xInUse ); // Ensure task is in use

         // Skip tasks that already have their priorities set
         if(pdTRUE == pxTCB->xPriorityIsSet)
         {
            continue;
         }
```

```
            // Additional logic to update xShortest and pxShortestTaskPointer...
        }
    }
}

    #if( schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_RMS )
        if( pxTCB->xPeriod <= xShortest )
        {
            xShortest = pxTCB->xPeriod;
            pxShortestTaskPointer = pxTCB;
        }

    #elif( schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_DMS )
        if( pxTCB->xRelativeDeadline <= xShortest )
        {
            xShortest = pxTCB->xRelativeDeadline;
            pxShortestTaskPointer = pxTCB;
        }
```

#elif(schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_DMS): This part of the code is compiled and executed only if the schedSCHEDULING_POLICY is set to schedSCHEDULING_POLICY_DMS

- if(pxTCB->xRelativeDeadline <= xShortest): This statement checks if the relative deadline of the current task (pxTCB->xRelativeDeadline) is less than or equal to the shortest deadline found so far (xShortest). This comparison is crucial for finding the task with the most imminent deadline.
    - pxTCB is a pointer to the current task's Control Block, where task-specific data like its deadline are stored.
    - xShortest holds the shortest deadline encountered during the current iteration over all tasks.

EDF Implementation in FreeRTOS

Implementing macros in schdeduler.h file:

#define schedSCHEDULING_POLICY_EDF 3

This line defines a macro schedSCHEDULING_POLICY_EDF with a value of 3. This typically represents the Earliest Deadline First Scheduling (EDF) policy. Assigning it the number 3 makes it a unique identifier for this scheduling policy within the system.

#define schedSCHEDULING_POLICY schedSCHEDULING_POLICY_EDF

This line sets the scheduling policy of the system to Earliest Deadline First Scheduling by using the value associated with schedSCHEDULING_POLICY_EDF.

```
#if( schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_EDF)
        #define schedEDF_NAIVE 1
        #define schedEDF_EFFICIENT 0
#endif
```

This block checks if the current scheduling policy (schedSCHEDULING_POLICY) is set to Earliest Deadline First (EDF), denoted by schedSCHEDULING_POLICY_EDF.
Inside the conditional block, it sets schedEDF_NAIVE to 1 and schedEDF_EFFICIENT to 0.

```
#define schedUSE_TIMING_ERROR_DETECTION_DEADLINE 1
#define schedUSE_TIMING_ERROR_DETECTION_EXECUTION_TIME 1
```

These lines define constants for enabling timing error detection features. The schedUSE_TIMING_ERROR_DETECTION_DEADLINE likely enables detection for deadline

```
#if( schedUSE_POLLING_SERVER == 1 || schedSCHEDULING_POLICY ==
schedSCHEDULING_POLICY_EDF || schedUSE_TIMING_ERROR_DETECTION_DEADLINE == 1 ||
schedMAX_NUMBER_OF_SPORADIC_JOBS == 1)
        #define schedUSE_SCHEDULER_TASK 1
#else
        #define schedUSE_SCHEDULER_TASK 0
#endif
```

This **#if** directive evaluates a logical OR condition among several configuration settings:

- **schedUSE_POLLING_SERVER == 1**: Checks if a polling server feature is enabled.
- **schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_EDF**: Checks if the EDF scheduling policy is active.
- **schedUSE_TIMING_ERROR_DETECTION_DEADLINE == 1**: Checks if deadline-related timing error detection is enabled.
- **schedMAX_NUMBER_OF_SPORADIC_JOBS == 1**: Evaluates whether the system supports or limits to only one sporadic job.

Implementation in scheduler.cpp file:

```
#if( schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_RMS || schedSCHEDULING_POLICY ==
schedSCHEDULING_POLICY_DMS )

        static void prvSetFixedPriorities( void );

#if( schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_RMS || schedSCHEDULING_POLICY ==
schedSCHEDULING_POLICY_DMS )
```

RM and DM code goes here

```
            #elif( schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_EDF )
              /* Initializes task priorities for EDF scheduling */
```

```
static void prvInitEDF( void )
{
  SchedTCB_t *pxTCB;

  #if( schedEDF_NAIVE == 1 )
    /* Set the highest task priority based on scheduler usage */
    #if( schedUSE_SCHEDULER_TASK == 1 )
      UBaseType_t uxHighestPriority = schedSCHEDULER_PRIORITY - 1;
    #else
      UBaseType_t uxHighestPriority = configMAX_PRIORITIES - 1;
    #endif
```

Depending on whether a scheduler task is being used (schedUSE_SCHEDULER_TASK), the initial highest priority is set either just below the scheduler's priority or as the highest available system priority (configMAX_PRIORITIES - 1).

```
// Setup pointers to begin and end of task list
const ListItem_t *pxTCBListEndMarker = listGET_END_MARKER( pxTCBList );
ListItem_t *pxTCBListItem = listGET_HEAD_ENTRY( pxTCBList );

// Iterate through task list
while( pxTCBListItem != pxTCBListEndMarker )
{
  // Assign highest available priority and get the task control block
  pxTCB = listGET_LIST_ITEM_OWNER( pxTCBListItem );
  pxTCB->uxPriority = uxHighestPriority--;

  // Move to the next list item
  pxTCBListItem = listGET_NEXT( pxTCBListItem );
}
```

After assigning a priority to a task, the highest priority is decremented (uxHighestPriority--), preparing the next priority level for the subsequent task in the list.

```
#elif( schedEDF_EFFICIENT == 1 )
  // Retrieve the first task in the ready list
  ListItem_t *pxTCBListItem = listGET_HEAD_ENTRY( pxTCBReadyList );

  // Get the task control block from the list item
  pxCurrentTCB = listGET_LIST_ITEM_OWNER( pxTCBListItem );

  // Set the task's priority to a running state priority
  pxCurrentTCB->uxPriority = schedPRIORITY_RUNNING;
#endif /* schedEDF_EFFICIENT */
```

Simulation in Arduino IDE using FreeRTOS Library.

We are using the same task-set we used for the theoretical example earlier, here is the implementation of the taskset in Arduino IDE and the respected ouputs;

1. vSchedulerPeriodicTaskCreate(testFunc1, "t1", configMINIMAL_STACK_SIZE, &c1, 4, &xHandle1, pdMS_TO_TICKS(0), pdMS_TO_TICKS(400), pdMS_TO_TICKS(100), pdMS_TO_TICKS(400));

2. vSchedulerPeriodicTaskCreate(testFunc2, "t2", configMINIMAL_STACK_SIZE, &c2, 3, &xHandle2, pdMS_TO_TICKS(0), pdMS_TO_TICKS(800), pdMS_TO_TICKS(200), pdMS_TO_TICKS(700));

3. vSchedulerPeriodicTaskCreate(testFunc3, "t3", configMINIMAL_STACK_SIZE, &c3, 2, &xHandle3, pdMS_TO_TICKS(0), pdMS_TO_TICKS(600), pdMS_TO_TICKS(150), pdMS_TO_TICKS(600));
4. vSchedulerPeriodicTaskCreate(testFunc4, "t4", configMINIMAL_STACK_SIZE, &c4, 1, &xHandle4, pdMS_TO_TICKS(0), pdMS_TO_TICKS(5000), pdMS_TO_TICKS(300), pdMS_TO_TICKS(5000));

In order to perform RM, we use **#define schedSCHEDULING_POLICY schedSCHEDULING_POLICY_RMS** in Scheduler.h

Here is the RM output from the serial monitor:

```
100   vSchedulerInit();
101   vSchedulerPeriodicTaskCreate(testFunc1, "t1", configMINIMAL_STACK_SIZE, &c1, 4, &xHandle1, pdMS_TO_TICKS(0), pdMS_TO_TICKS(40
102   vSchedulerPeriodicTaskCreate(testFunc2, "t2", configMINIMAL_STACK_SIZE, &c2, 3, &xHandle2, pdMS_TO_TICKS(0), pdMS_TO_TICKS(80
103   vSchedulerPeriodicTaskCreate(testFunc3, "t3", configMINIMAL_STACK_SIZE, &c3, 2, &xHandle3, pdMS_TO_TICKS(0), pdMS_TO_TICKS(60
```

Output Serial Monitor ×

Message (Enter to send message to 'Arduino Mega or Mega 2560' on 'COM3') New Line

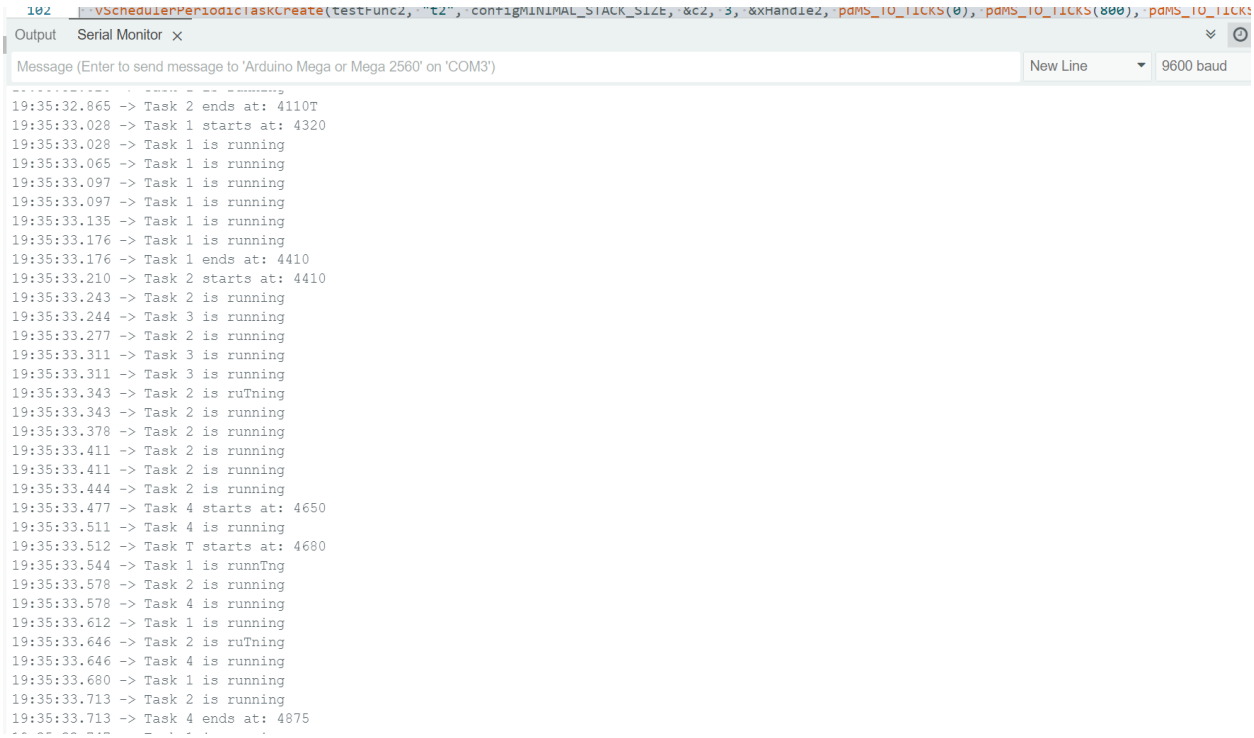
```
19:24:23.634 -> Task 3 starts at: 555
19:24:23.668 -> Task 3 is running
19:24:23.668 -> TaskT2 is running
19:24:23.701 -> Task 3 is running
19:24:23.735 -> Task 2 is ruTning
19:24:23.735 -> Task 3 ends at: 660
19:24:23.769 -> Task 2 ends at: 660
19:24:23.803 -> Task 1 starts at: 720
19:24:23.803 -> Task 1 is running
19:24:23.836 -> Task 1 is running
19:24:23.869 -> Task 2 starts aT: 735
19:24:23.869 -> Task 2 is running
19:24:23.903 -> Task 1 is running
19:24:23.936 -> Task 2 is runniTg
19:24:23.936 -> Task 1 is running
19:24:23.970 -> Task 2 is running
19:24:24.004 -> Task 1 ends at: 855
19:24:24.004 -> Task 2 is running
19:24:24.037 -> Task 2 is running
19:24:24.071 -> Task 2 is running
19:24:24.071 -> Task 2 is running
19:24:24.104 -> Task 2 is running
19:24:24.138 -> Task 2 is running
19:24:24.138 -> Task 2 is running
19:24:24.171 -> Task 2 is running
19:24:24.206 -> Task 2 is running
19:24:24.206 -> Task 2 ends at: 1050
19:24:24.238 -> Task 1 starts at: 1080
19:24:24.272 -> Task 1 is running
19:24:24.272 -> Task 1 is running
```

Tasks start at very close time intervals, with Task 1 starting first, followed closely by Tasks 2, 3, and 4. This suggests that either the tasks have very similar periods or the system's scheduling ticks allow for quick switches and executions.

The tasks appear to preempt each other frequently. For example, Task 1 starts but is quickly interrupted by Task 3, which in turn is interrupted by Task 2. This indicates a preemptive scheduling strategy, where tasks with higher priority (based on RMS) interrupt those with lower priority.

In order to perform DM, we use **#define schedSCHEDULING_POLICY schedSCHEDULING_POLICY_DMS** in Scheduler.h

Here is the DM output from the serial monitor:

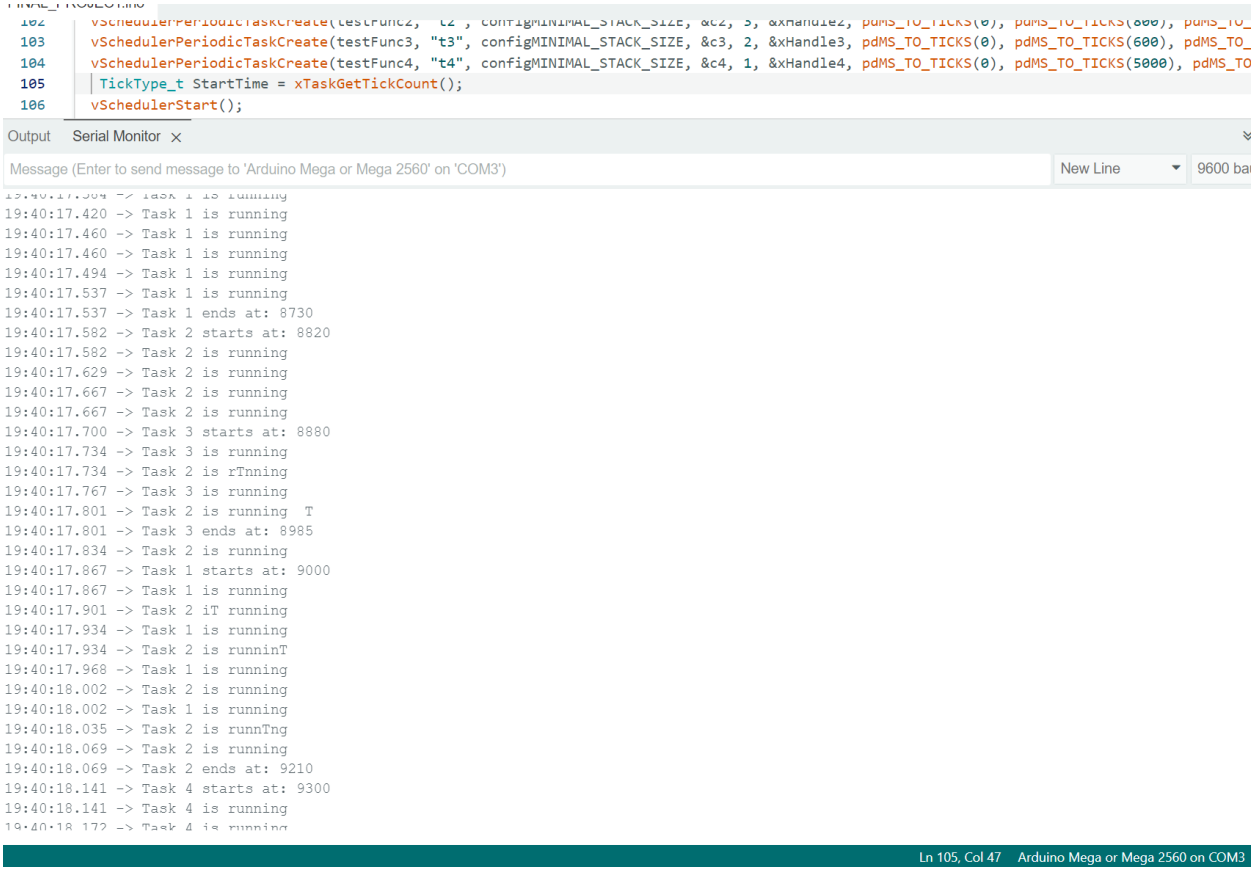


The log entries show interleaved executions between Task 1 and Task 2, indicating that these tasks are being preempted and resumed frequently. Task 3 also starts execution but is seen interleaved between Task 2's executions.

This interleaving indicates a system where tasks with relatively close deadlines or different priorities preempt each other. The time spans show these tasks might be having overlapping deadlines or their deadlines are very close, causing frequent context switching.

In order to perform EDF, we use **#define schedSCHEDULING_POLICY schedSCHEDULING_POLICY_EDF** in scheduler.h

Here is the EDF output from the serial monitor:



The log shows frequent switches between Task 1, 2, and 3. This reflect tasks with very short execution slices, and tasks that are being frequently interrupted due to new tasks with earlier deadlines becoming ready to run.

Notably, Tasks 1 and 2 appear prominently, with Task 4 occasionally interrupting, showing it have later deadlines compared to the other tasks.

Observations

1. Task interleaving and preemption:
The output logs from RMS, DMS, and EDF shows that the tasks are interleaving where multiple tasks run concurrently and preempt each other. This indicates active task management where tasks with higher priority (under RMS and DMS) or nearer deadlines (under EDF) preempt lower priority or later-deadline tasks.

2. Frequent context switching:
Across all scheduling policies, there is a frequent context switch. This behavior indicates a highly dynamic system. While this is beneficial for responsiveness it may cause potential overhead due to the high rate of task switching.

3. Deadline and period management:
Under RMS and DMS, tasks seem to be scheduled based on static priorities derived from their periods and deadlines, respectively. EDF showed dynamic priority adjustments based on deadlines, but the rapid switches suggest either overly aggressive deadline settings or very short task durations.

4. System load and task overheads:
The system appears to be under significant load, possibly operating near its capacity given the frequent task activations and context switches. This might be indicative of either an overloaded system or inefficiencies in task design and scheduling configurations.

Conclusion:

All the Scheduling algorithms are implemented successfully in FreeRTOS. The respective outputs are observed in the serial monitor. The implemented scheduling strategies, while fundamentally sound in theory, show practical challenges in the embedded context of Arduino with FreeRTOS. The high frequency of task switching points to a need for optimizing task periods, deadlines, and execution times to better suit the capabilities of the system and reduce scheduling overhead.