Real Time Operating System “FreeRTOS” Part 3

Prepared By: AMIT Learning

Agenda

- Idle Task Call Back
- Changing Tasks priorities
- Scheduling Wrap up

The Idle Task and the Idle Task Hook (Idle Task CallBack)

- An Idle task is automatically created by the scheduler when vTaskStartScheduler() is called.
- The idle task does very little more than sit in a loop
- The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state
- Idle task is always pre-empted by higher priority tasks
- idle hook (or idle callback) function—a function that is called automatically by the idle task once per iteration of the idle task loop
- Common uses for the Idle task hook include:
  - Executing low priority, background, or continuous processing.
  - Measuring the amount of spare processing capacity.
  - Placing the processor into a low power mode
- configUSE_IDLE_HOOK must be set to 1 within FreeRTOSConfig.h for the idle hook function to get called.

Example 7: Idle Task CallBack

```
/* Declare a variable that will be incremented by the hook function. */
unsigned long ulIdleCycleCount = 0UL;

/* Idle hook functions MUST be called vApplicationIdleHook(), take no parameters,
and return void. */
void vApplicationIdleHook( void )
{
    /* This hook function does nothing but increment a counter. */
    ulIdleCycleCount++;
}
```

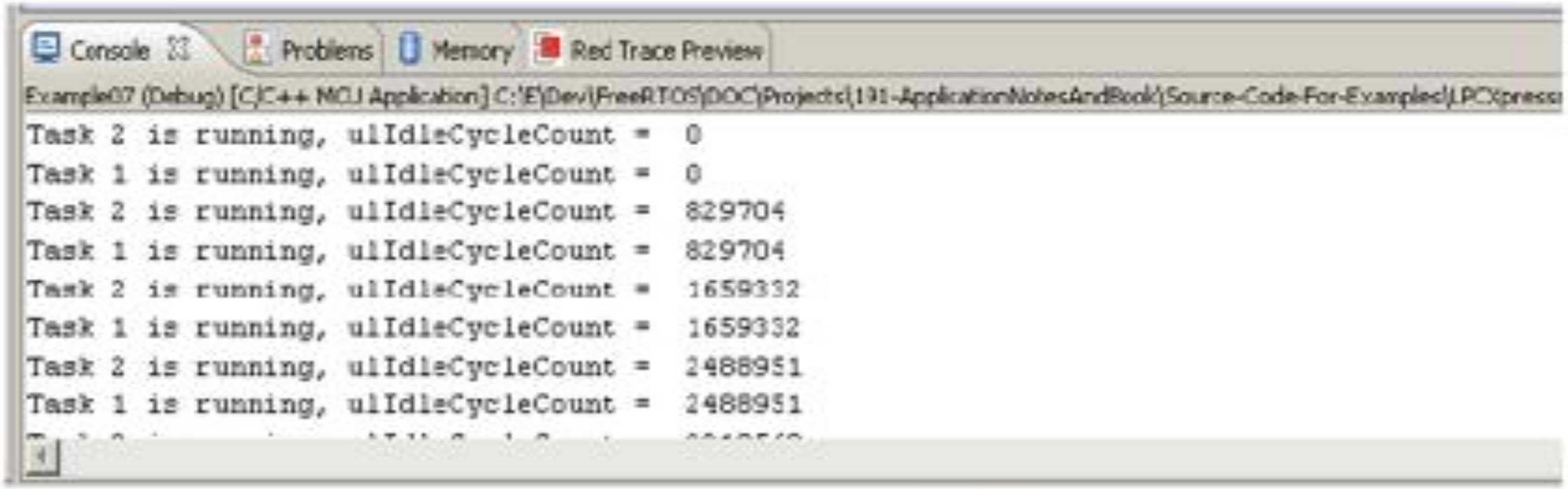
Listing 18. A very simple Idle hook function

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task AND the number of times ulIdleCycleCount
        has been incremented. */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

        /* Delay for a period of 250 milliseconds. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Listing 19. The source code for the example task prints out the ulIdleCycleCount value

Example 7: Idle Task CallBack

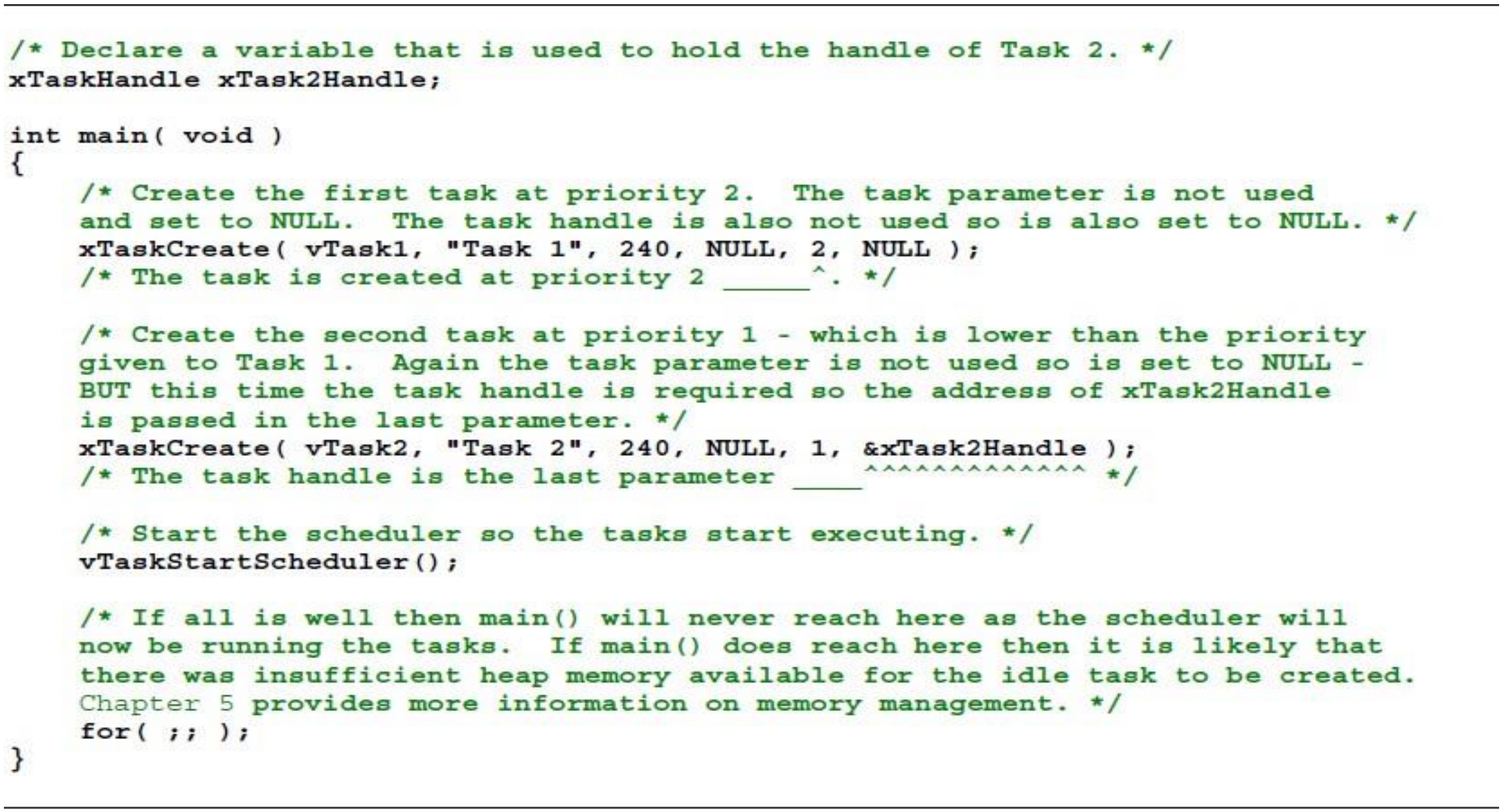
How many times IdelTask is called between tasks calls?



```
Example07 (Debug) [C/C++ MCU Application] C:\E\Dev\FreeRTOS\DOC\Projects\191-ApplicationNotesAndBook\Source-Code-For-Examples\LPCXpress
Task 2 is running, ulIdleCycleCount =  0
Task 1 is running, ulIdleCycleCount =  0
Task 2 is running, ulIdleCycleCount =  829704
Task 1 is running, ulIdleCycleCount =  829704
Task 2 is running, ulIdleCycleCount =  1659332
Task 1 is running, ulIdleCycleCount =  1659332
Task 2 is running, ulIdleCycleCount =  2488951
Task 1 is running, ulIdleCycleCount =  2488951
```

Figure 13. The output produced when Example 7 is executed

Changing task priorities; Example 8



```
/* Declare a variable that is used to hold the handle of Task 2. */
xTaskHandle xTask2Handle;

int main( void )
{
    /* Create the first task at priority 2.  The task parameter is not used
    and set to NULL.  The task handle is also not used so is also set to NULL. */
    xTaskCreate( vTask1, "Task 1", 240, NULL, 2, NULL );
    /* The task is created at priority 2 _____^. */

    /* Create the second task at priority 1 - which is lower than the priority
    given to Task 1.  Again the task parameter is not used so is set to NULL -
    BUT this time the task handle is required so the address of xTask2Handle
    is passed in the last parameter. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, &xTask2Handle );
    /* The task handle is the last parameter ____^^^^^^^^^^^^^ */

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 24. The implementation of main() for Example 8

Changing task priorities; Example 8

```
void vTask1( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* This task will always run before Task 2 as it is created with the higher
    priority.  Neither Task 1 nor Task 2 ever block so both will always be in either
    the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\n" );

        /* Setting the Task 2 priority above the Task 1 priority will cause
        Task 2 to immediately start running (as then Task 2 will have the higher
        priority of the two created tasks).  Note the use of the handle to task
        2 (xTask2Handle) in the call to vTaskPrioritySet().  Listing 24 shows how
        the handle was obtained. */
        vPrintString( "About to raise the Task 2 priority\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* Task 1 will only run when it has a priority higher than Task 2.
        Therefore, for this task to reach this point Task 2 must already have
        executed and set its priority back down to below the priority of this
        task. */
    }
}
```

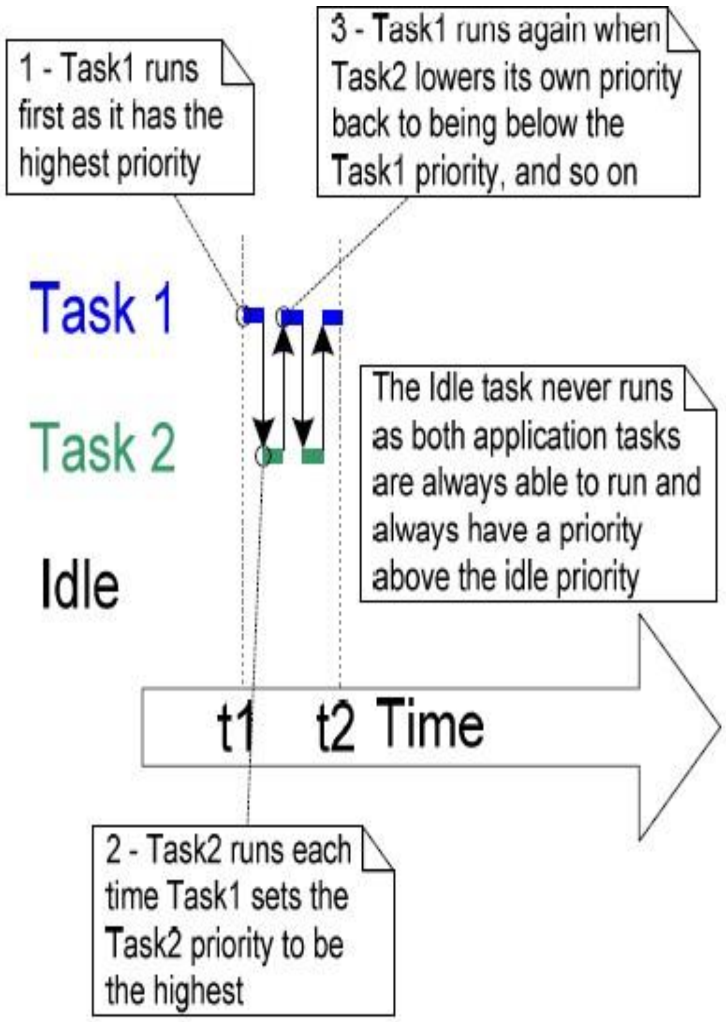
Listing 22. The implementation of Task 1 in Example 8

Changing task priorities; Example 8

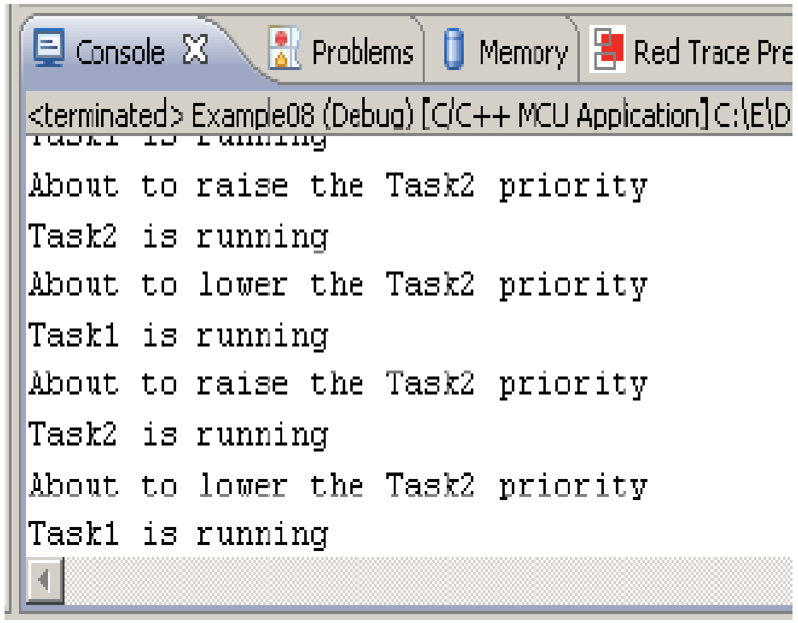
```
void vTask2( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* Task 1 will always run before this task as Task 1 is created with the
    higher priority.  Neither Task 1 nor Task 2 ever block so will always be
    in either the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* For this task to reach this point Task 1 must have already run and
        set the priority of this task higher than its own.

        Print out the name of this task. */
        vPrintString( "Task2 is running\n" );

        /* Set our priority back down to its original value.  Passing in NULL
        as the task handle means "change my priority".  Setting the
        priority below that of Task 1 will cause Task 1 to immediately start
        running again – pre-empting this task. */
        vPrintString( "About to lower the Task 2 priority\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

Listing 23. The implementation of Task 2 in Example 8

Changing task priorities; Example 8



1 - Task1 runs first as it has the highest priority
3 - Task1 runs again when Task2 lowers its own priority back to being below the Task1 priority, and so on
Task 1
Task 2
Idle
The Idle task never runs as both application tasks are always able to run and always have a priority above the idle priority
t1
t2 Time
2 - Task2 runs each time Task1 sets the Task2 priority to be the highest



```
<terminated> Example08 (Debug) [C/C++ MCU Application] C:\E\D
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
```

Figure 14. The sequence of task execution when running Example 8

The Scheduling Algorithm—A Summary

- Fixed Prioritized Pre-emptive Scheduling
  - Each task is assigned a priority.
  - Each task can exist in one of several states.
  - Only one task can exist in the Running state at any one time.
  - The scheduler always selects the highest priority Ready state task to enter the Running state.

- **Fixed Priority’** because each task is assigned a priority that is not altered by the kernel itself (only tasks can change priorities);

- **‘Pre-emptive’** because a task entering the Ready state or having its priority altered will always pre-empt the Running state task, if the Running state task has a lower priority.

The Scheduling Algorithm—A Summary

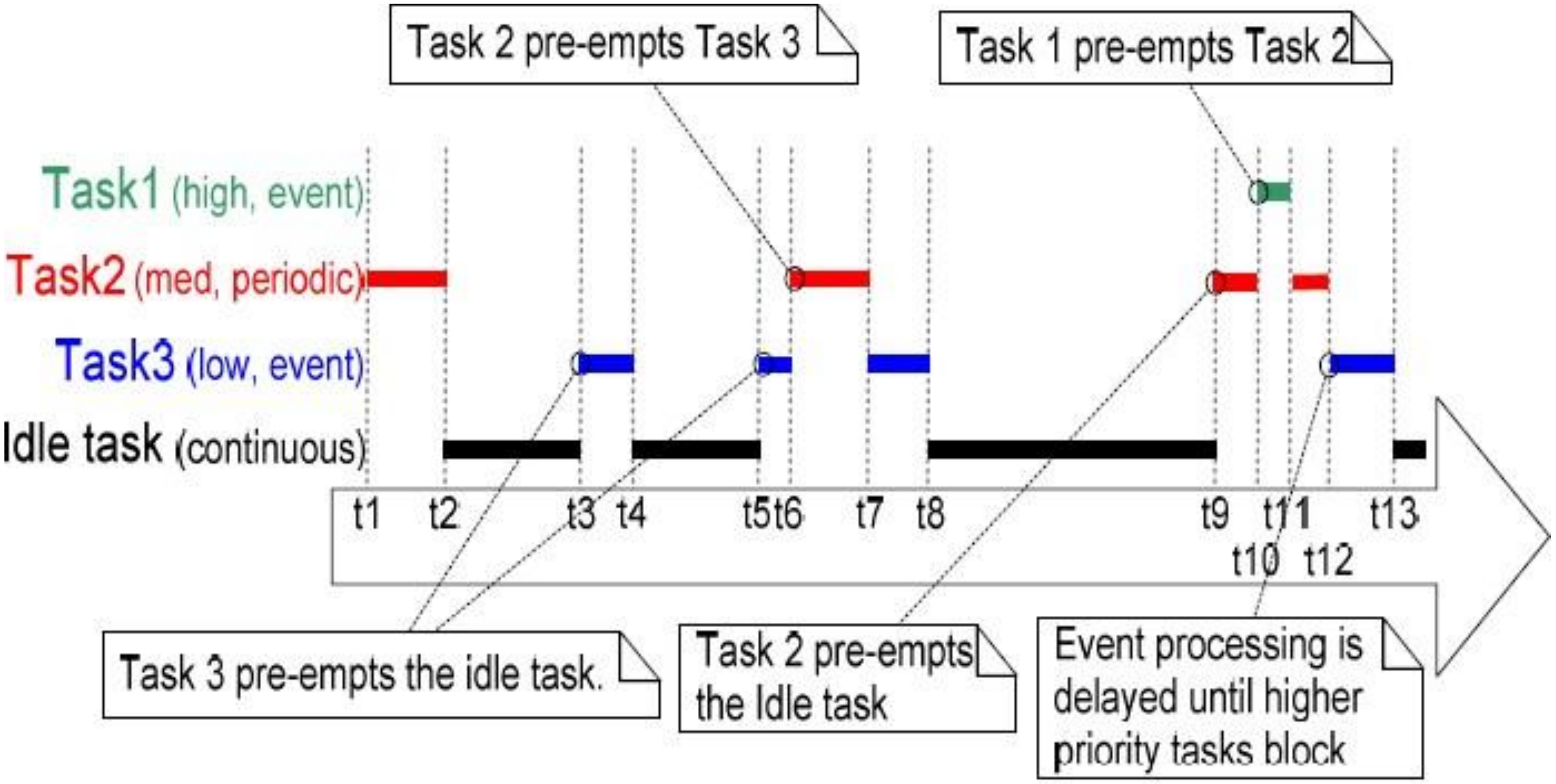


Figure 18. Execution pattern with pre-emption points highlighted