HCMC University Of Technology, VNU-HCM
Faculty of Computer Science & Engineering

– Embedded System Lab 03 –

FreeRTOS Tasks Scheduling

Pham Hoang Anh & **Huynh Hoang Kha**

Goal In this lab, students are expected to ...

- Have a deeper look in FreeRTOS tasks
- Understand the task scheduling algorithm in the FreeRTOS
- Be able to configure the FreeRTOS scheduling algorithm

Content

- The idle task and idle task hook
- FreeRTOS scheduling algorithm and the `FreeRTOSConfig.h` header
- Preemptive scheduling versus Co-operative scheduling

Prerequisite Requirement

- Have basic knowledge of OS and micro-controller programming.

Grading policy

- 40% in-class performance
- 60% report submission

1 The idle task and idle task hook

1.1 The idle task

The idle task is created automatically when the scheduler is started. This ensure there must be at least one task that can enter the Running state at any time. The idle task does very little more than sit in a loop - so, it is always able to run. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.

Power consumption can be decreased significantly by placing the processor into a low power state each time the idle task runs. FreeRTOS also has a special tick-less mode. Using the tick-less mode allows the processor to enter a lower power mode than would otherwise be possible, and remain in the low power mode for longer.

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.

```
1  void vTaskFunction( void *pvParameters )
2  {
3      char *pcTaskName;
4      const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );
5
6      /* The string to print out is passed in via the parameter.
7      Cast this to a character pointer. */
8      pcTaskName = ( char * ) pvParameters;
9
10     /* As per most tasks, this task is implemented in
11     an infinite loop. */
12     for( ;; )
13     {
14         /* Print out the name of this task. */
15         vPrintString( pcTaskName );
16
17         /* Delay for a period. This time a call to vTaskDelay()
18         is used which places the task into the Blocked state
19         until the delay period has expired. The parameter takes
20         a time specified in "ticks", and the pdMS_TO_TICKS() macro
21         is used (where the xDelay250ms constant is declared) to
22         convert 250 milliseconds into an equivalent time in ticks.*/
23         vTaskDelay( xDelay250ms );
24     }
25 }
```

Code 1: A cyclic task printing the passed string argument every 250 milliseconds

Lets consider Code 1 and Code 2 below then examine how the FreeRTOS scheduler works in this case. In the task function as in shown Code 1:

- Line 1: The way the function was named is abiding the FreeRTOS coding convention. The "v" prefix is to indicate that the function returns nothing (void).

- Line 15: Because the FreeRTOS supports so many platforms, the `vPrintString()` routine used here is just an abstraction. On ESP32 development board, replace it with the usage of `printf()` function.

```
1  static const char *pcTextForTask1 = "Task 1 is running\r\n";
2  static const char *pcTextForTask2 = "Task 2 is running\r\n";
3  void app_main(void)
4  {
5      /* Create the first task at priority 1.
6      The priority is the second to last parameter. */
7      xTaskCreate( vTaskFunction, "Task 1", 1000,
8                   (void*)pcTextForTask1, 1, NULL );
9
10     /* Create the second task at priority 2,
11     which is higher than a priority of 1.
12     The priority is the second to last parameter. */
13     xTaskCreate( vTaskFunction, "Task 2", 1000,
14                  (void*)pcTextForTask2, 2, NULL );
15
16     /* Start the scheduler so the tasks start executing. */
17     vTaskStartScheduler();
18
19     /* Will not reach here. */
20 }
```

Code 2: Create tasks using task function as shown in Code 1

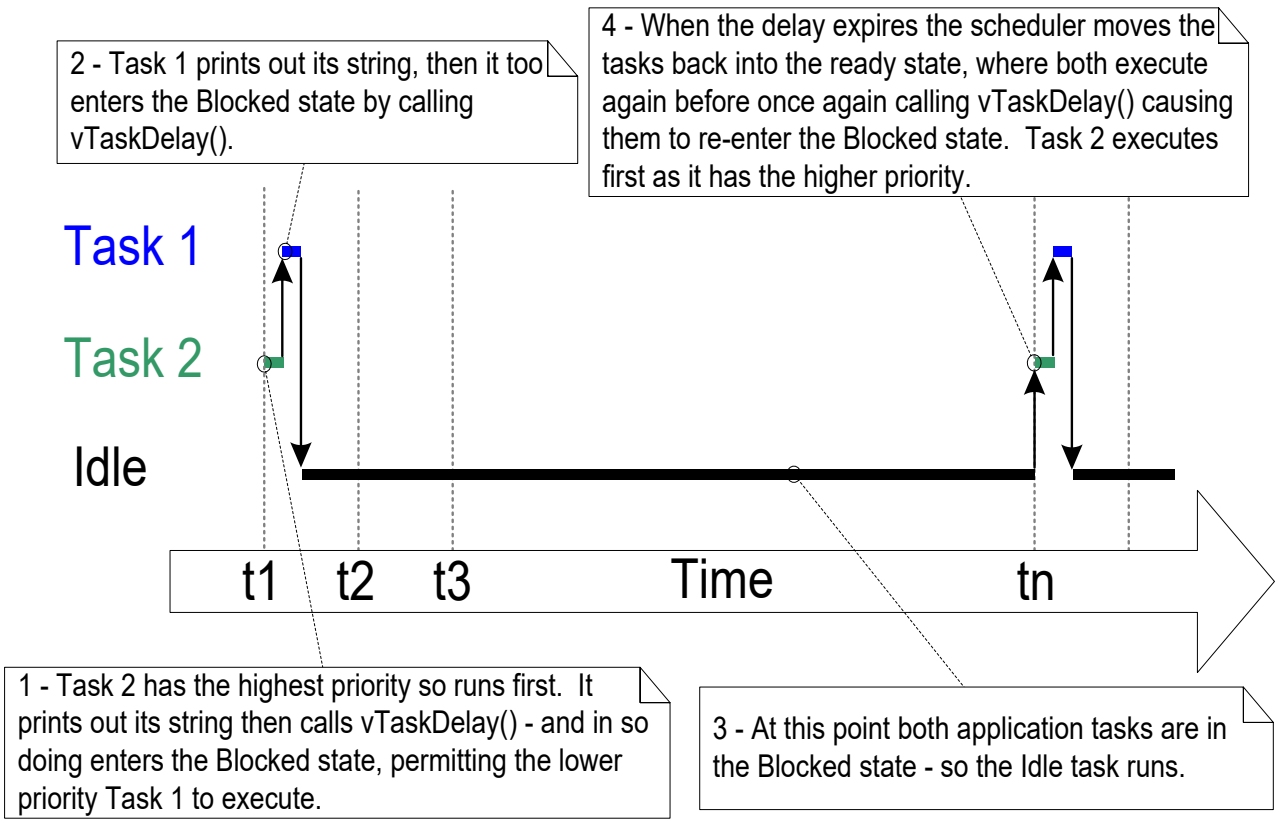

Figure 1: The execution sequence when the tasks use `vTaskDelay()`

Executing Code 2, there would be two tasks created using the same task function but different priorities. The execution sequence should be as described in Figure 1

1.2 The idle task hook

It is possible to add application specific functionality directly into the idle task through the use of an 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 functionality.
- Measuring the amount of spare processing capacity. (The idle task will run only when all higher priority application tasks have no work to perform; so measuring the amount of processing time allocated to the idle task provides a clear indication of how much processing time is spare.)
- Placing the processor into a low power mode, providing an easy and automatic method of saving power whenever there is no application processing to be performed.

Idle task hook functions must adhere to the following rules that will be considered as **limitations on the Implementation of Idle Task Hook Functions**.

1. An Idle task hook function must never attempt to block or suspend.
   Note: Blocking the idle task in any way could cause a scenario where no tasks are available to enter the Running state.
2. If the application makes use of the vTaskDelete() API function, then the idle task hook must always return to its caller within a reasonable time period. This is because the idle task is responsible for cleaning up kernel resources after a task has been deleted. If the idle task remains permanently in the idle hook function, then this clean-up cannot occur.

Idle task hook functions must have the name and prototype shown by Code 3 and Code 4 is a very simple example of an idle task hook function.

```
1 void vApplicationIdleHook(void);
```

Code 3: The idle task hook function name and prototype

```
1 /* Declare a variable that will be incremented by the hook function. */
2 volatile uint32_t ulIdleCycleCount = 0UL;
3
4 /* Idle hook functions MUST be called vApplicationIdleHook(), take no
     parameters, and return void. */
```

```
5
6  void vApplicationIdleHook(void)
7  {
8      /* This hook function does nothing but increment a counter. */
9      ulIdleCycleCount++;
10 }
```

Code 4: A very simple Idle hook function

2 FreeRTOS scheduling algorithm and the `FreeRTOSConfig.h` header

A Recap of Task States and Events

The task that is actually running (using processing time) is in the Running state. On a single core processor there can only be one task in the Running state at any given time.

Tasks that are not actually running, but are not in either the Blocked state or the Suspended state, are in the Ready state. Tasks that are in the Ready state are available to be selected by the scheduler as the task to enter the Running state. The scheduler will always choose the highest priority Ready state task to enter the Running state.

Tasks can wait in the Blocked state for an event and are automatically moved back to the Ready state when the event occurs. Temporal events occur at a particular time, for example, when a block time expires, and are normally used to implement periodic or timeout behavior. Synchronization events occur when a task or interrupt service routine sends information using a task notification, queue, event group, or one of the many types of semaphore. They are generally used to signal asynchronous activity, such as data arriving at a peripheral.

Configuring the Scheduling Algorithm

The scheduling algorithm is the software routine that decides which Ready state task to transition into the Running state.

All the examples so far have used the same scheduling algorithm, but the algorithm can be changed using the `configUSE_PREEMPTION` and `configUSE_TIME_SLICING` configuration constants. Both constants are defined in `FreeRTOSConfig.h`

A third configuration constant, `configUSE_TICKLESS_IDLE`, also affects the scheduling algorithm, as its use can result in the tick interrupt being turned off completely for extended periods. `configUSE_TICKLESS_IDLE` is an advanced option provided specifically for use in applications that must minimize their power consumption. The descriptions provided in this section assume `configUSE_TICKLESS_IDLE` is set to 0, which is the default setting if the constant is left undefined.

In all possible configurations the FreeRTOS scheduler will ensure tasks that share a priority are selected to enter the Running state in turn. This "take it in turn" policy is often referred to as "Round Robin Scheduling". A Round Robin scheduling algorithm does not guarantee time is shared equally between tasks of equal priority, only that Ready state tasks of equal priority will enter the Running state in turn.

3 FreeRTOS scheduling algorithm's properties: Preemptive and time-slicing

The two most important properties in FreeRTOS are preemptive and time-slicing. Pre-emptive scheduling algorithms will immediately 'pre-empt' the Running state task if a task that has a priority higher than the Running state task enters the Ready state. Being pre-empted means being involuntarily (without explicitly yielding or blocking) moved out of the Running state and into the Ready state to allow a different task to enter the Running state.

Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state. Scheduling algorithms described as using 'Time Slicing' will select a new task to enter the Running state at the end of each time slice if there are other Ready state tasks that have the same priority as the Running task. A time slice is equal to the time between two RTOS tick interrupts.

These properties can be combined as follow:

- Preemptive with time-slicing
- Preemptive without time-slicing
- Non-preemptive (also called co-operative)

When the co-operative scheduler is used, a context switch will only occur when the Running state task enters the Blocked state, or the Running state task explicitly yields (manually requests a re-schedule) by calling `taskYIELD()`. Tasks are never pre-empted, that is the reason why time-slicing cannot be used.

3.1 Prioritized Pre-emptive Scheduling with Time Slicing

Look at Figure 2

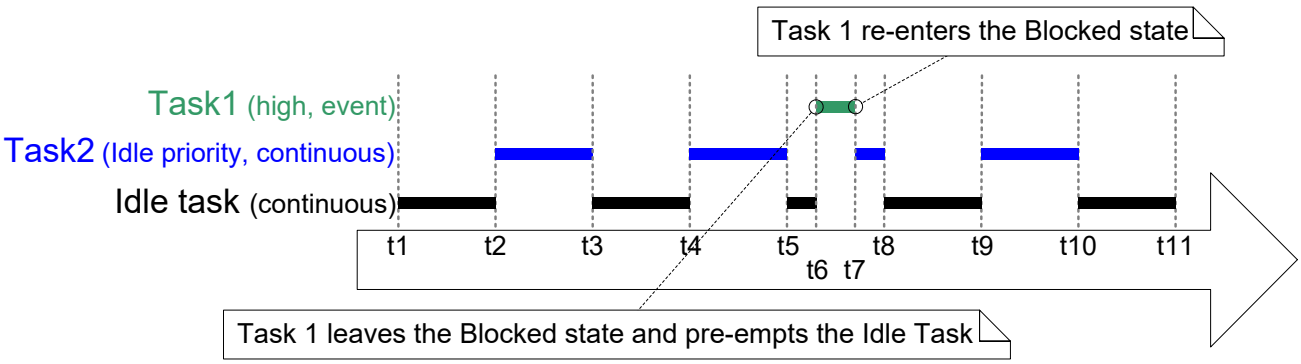

Figure 2: Preemptive scheduling with time-slicing example execution sequence

The Idle Task and Task 2

The Idle task and Task 2 are both continuous processing tasks, and both have a priority of 0 (the lowest possible priority). The scheduler only allocates processing time to the priority 0 tasks when there are no higher priority tasks that are able to run, and shares the time that is allocated to the priority 0 tasks by time slicing. A new time slice starts on each tick interrupt, which in Figure 27 is at times t1, t2, t3, t4, t5, t8, t9, t10 and t11.

The Idle task and Task 2 enter the Running state in turn, which can results in both tasks being in the Running state for part of the same time slice, as happens between time t5 and time t8.

Task 1

The priority of Task 1 is higher than the Idle priority. Task 1 is an event driven task that spends most of its time in the Blocked state waiting for its event of interest, transitioning from the Blocked state to the Ready state each time the event occurs.

The event of interest occurs at time t6, so at t6 Task 1 becomes the highest priority task that is able to run, and therefore Task 1 pre-empts the Idle task part way through a time slice. Processing of the event completes at time t7, at which point Task 1 re-enters the Blocked state.

Allocating that much processing time to the Idle task might not be desirable if the Idle priority tasks created by the application writer have work to do, but the Idle task does not. The `configIDLE_SHOULD_YIELD` compile time configuration constant can be used to change how the Idle task is scheduled:

- If `configIDLE_SHOULD_YIELD` is set to 0 then the Idle task will remain in the Running state for the entirety of its time slice, unless it is preempted by a higher priority task.

- If `configIDLE_SHOULD_YIELD` is set to 1 then the Idle task will yield (voluntarily give up whatever remains of its allocated time slice) on each iteration of its loop if there are other Idle priority tasks in the Ready state.

The execution pattern shown in Figure 2 is what would be observed when `configIDLE_SHOULD_YIELD` is set to 0. The execution pattern shown in Figure 3 is what would be observed in the same scenario when `configIDLE_SHOULD_YIELD` is set to 1.

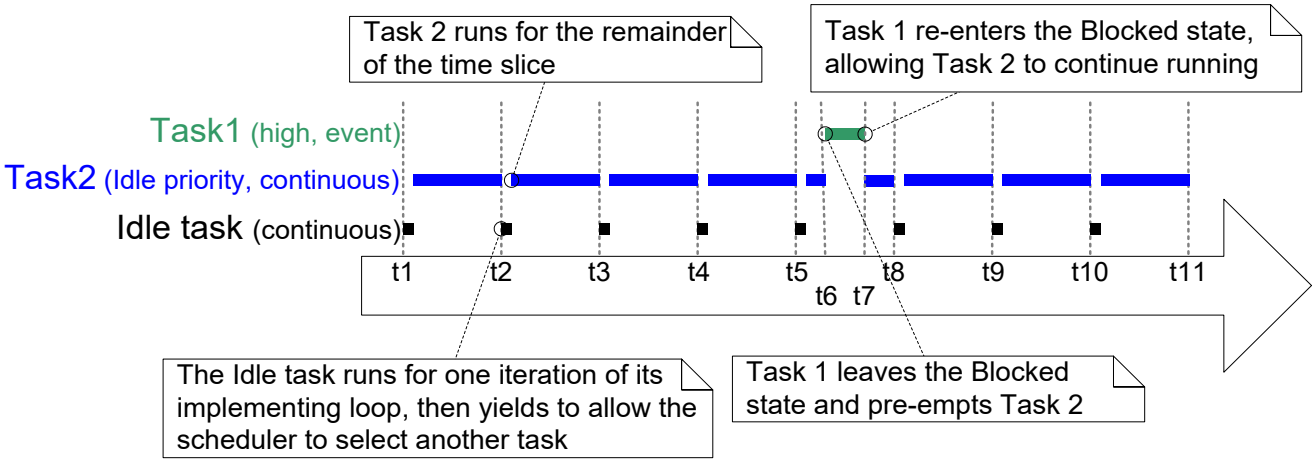

Figure 3: The execution pattern for the same scenario as shown in Figure 2, but this time with `configIDLE_SHOULD_YIELD` set to 1

3.2 Prioritized Pre-emptive Scheduling (without Time Slicing)

Prioritized Preemptive Scheduling without time slicing maintains the same task selection and pre-emption algorithms as described in the previous section, but does not use time slicing to share processing time between tasks of equal priority.

As was demonstrated in Figure 2, if time slicing is used, and there is more than one ready state task at the highest priority that is able to run, then the scheduler will select a new task to enter the Running state during each RTOS tick interrupt (a tick interrupt marking the end of a time slice). If time slicing is not used, then the scheduler will only select a new task to enter the Running state when either:

- A higher priority task enters the Ready state.
- The task in the Running state enters the Blocked or Suspended state.

There are fewer task context switches when time slicing is not used than when time slicing is used. Therefore, turning time slicing off results in a reduction in the scheduler's processing overhead. However, turning time slicing off can also result in tasks of equal priority receiving greatly different amounts of processing time, a scenario demonstrated by Figure 4. For this reason, running the scheduler

without time slicing is considered an advanced technique that should only be used by experienced users.

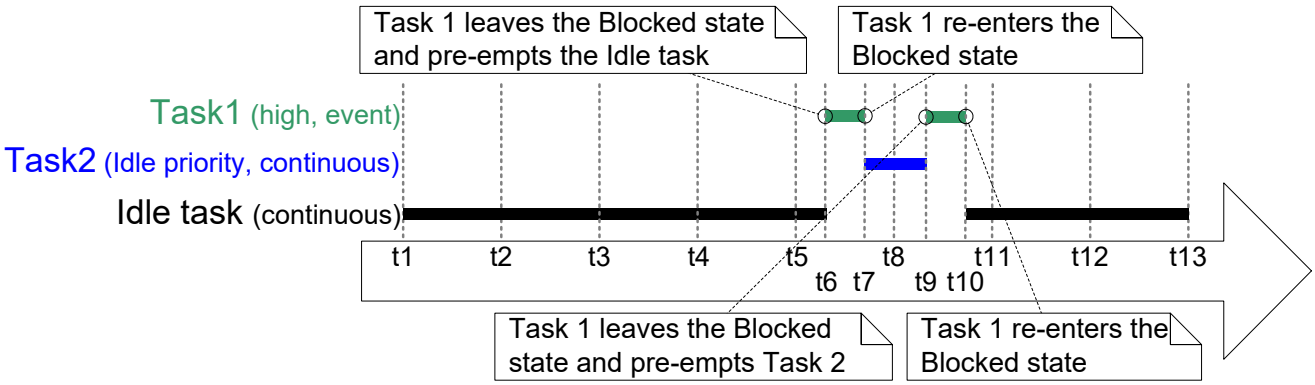

Figure 4: Execution pattern that demonstrates how tasks of equal priority can receive hugely different amounts of processing time when time slicing is not used

3.3 Co-operative Scheduling

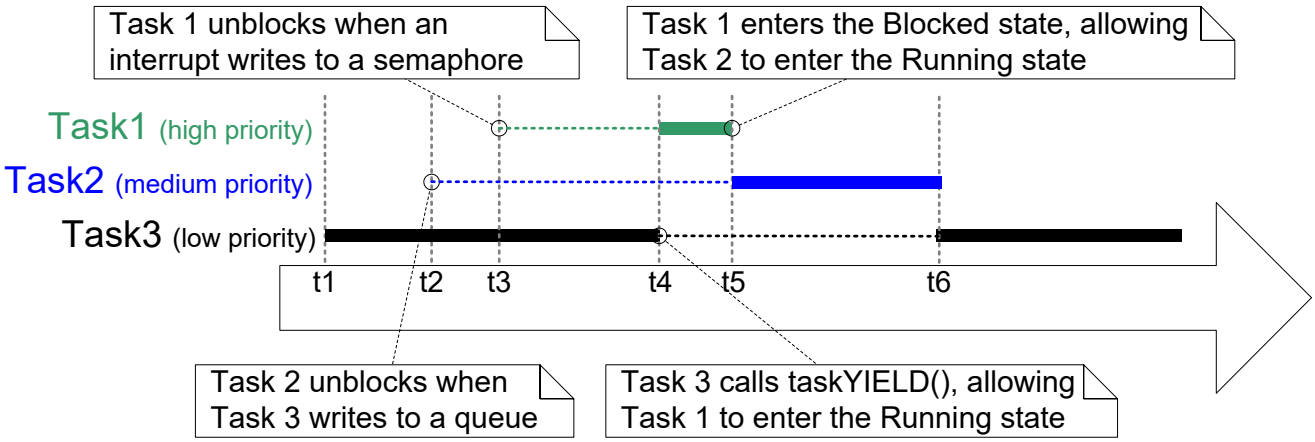

Figure 5: Execution pattern demonstrating the behavior of the co-operative scheduler

In Figure 5, Task 2 was Ready at t2 and Task 1 was Ready at t3. But they don't "preempt" the Task 3.

At t4, the Task 3 called `taskYIELD()` so the scheduler select a Ready task with the highest priority to execute.

4 Exercises

Design and implement examples for each scheduling mentioned above, they are:

- Prioritized Pre-emptive Scheduling with Time Slicing.

- Prioritized Pre-emptive Scheduling (without Time Slicing)
- Co-operative Scheduling

Extra exercise: Implement a program that use the idle task hook to monitor the CPU utilization. Note that, the ESP32 is dual-core.