
CSCE 4114 FreeRTOS

David Andrews

dandrews@uark.edu



Agenda

- Defining and Creating Tasks
- Intro to Task Scheduler
- Tick Interrupt



Some Definitions to Start

- Processes and Threads: Independent sequences of execution.
 - Threads: run in shared memory space
 - Processes: run in separate memory spaces
- We will work with Tasks == Threads
 - Have their own context (PC, Reg File, Stack)
 - Will assign Priorities
 -Visible by an OS Scheduler



Simple View of Task "State"

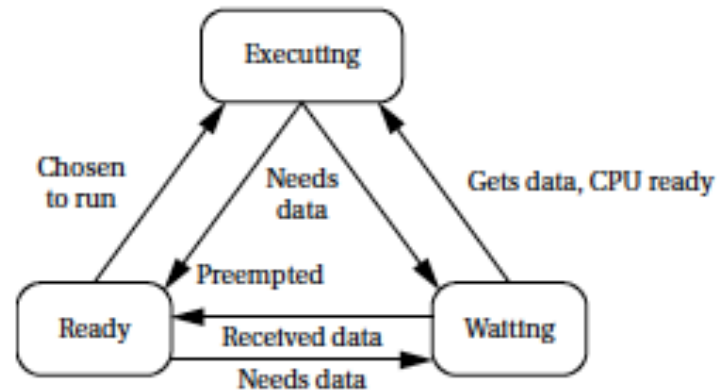


FIGURE 6.6

Scheduling states of a process.

- Executing: only 1 currently on CPU
- Ready Tasks: Not blocked or waiting on anything. Can have many-sort by priority
- Waiting Tasks: Blocked or Suspended



FreeRTOS Task State Machine

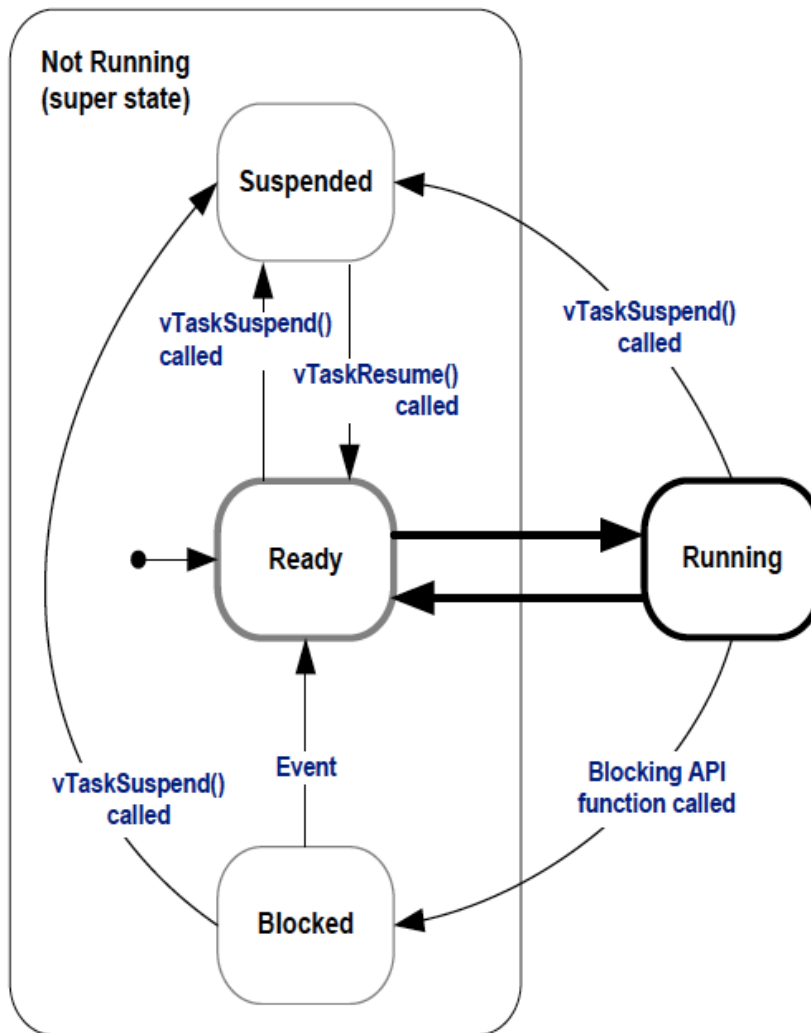


Figure 15. Full task state machine



Scheduler: How is it invoked ?

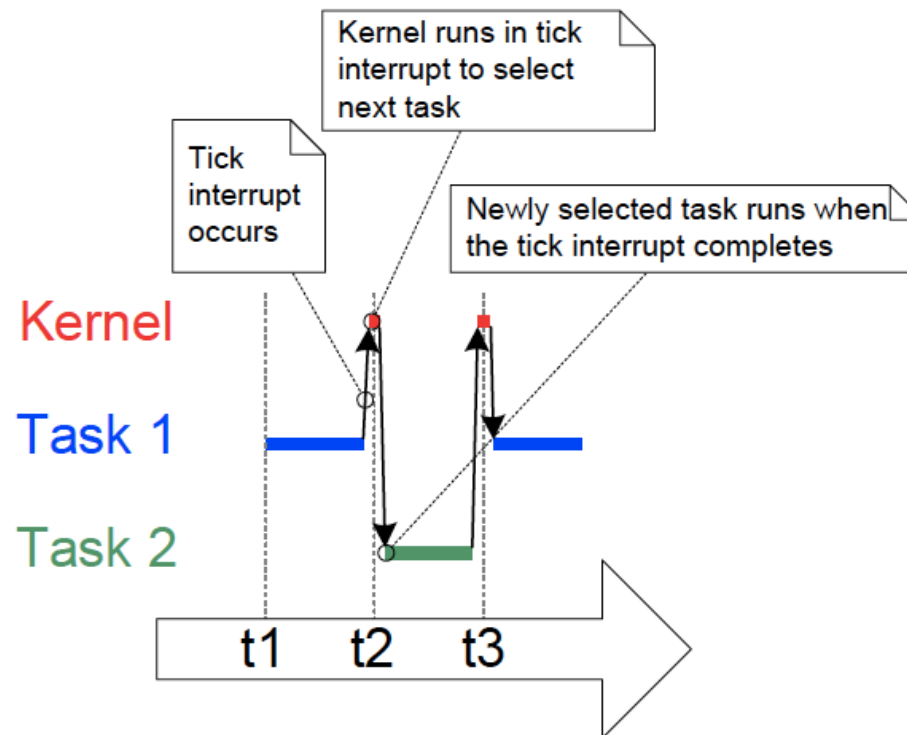


Figure 12. The execution sequence expanded to show the tick interrupt executing

- Simplest: Timer Tick (periodic interrupt)
- Set in FreeRTOS using `TICK_RATE_HZ`



Inside Tick Interrupt

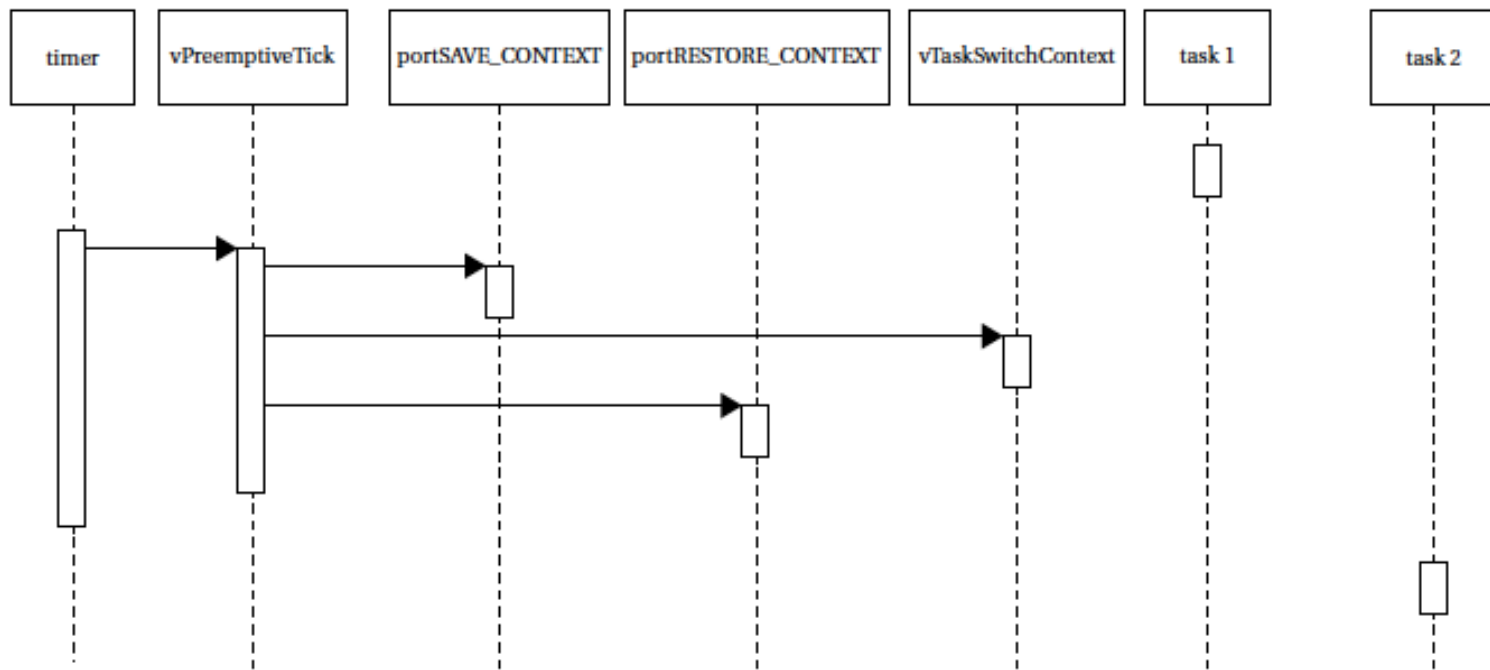


FIGURE 6.9

Sequence diagram for freeRTOS.org context switch.



Priority Based Scheduling

- Pre-emptive: Task on the Processor can be interrupted.
 - -Use Timer Tick to check
- Non pre-emptive: Once on CPU Tasks cannot be interrupted.
 - -Don't use Timer Tick
 - Current Tasks can suspend itself. That's it !



Priorities

- Static: Assign once and forget.
Scheduler cannot change
- Dynamic: Assigned but Scheduler can change as the system runs

- Static: RMS Rate Monotonic Scheduling
- Dynamic: EDF Earliest Deadline First



RMS

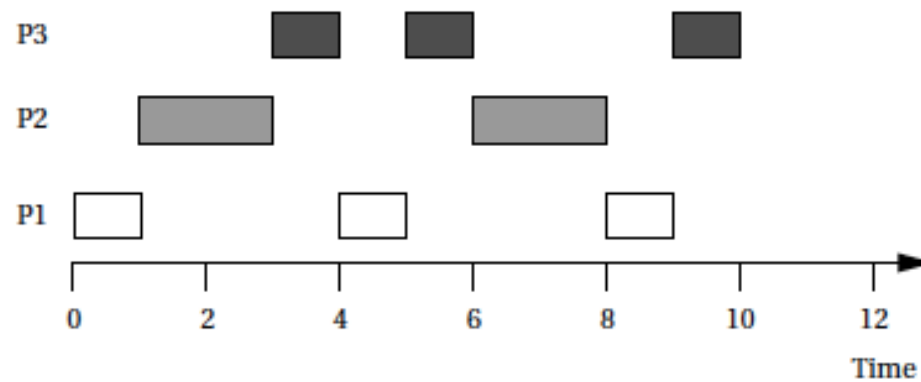
Example 6.3

Rate-monotonic scheduling

Here is a simple set of processes and their characteristics.

Process	Execution time	Period
P1	1	4
P2	2	6
P3	3	12

Applying the principles of RMA, we give P1 the highest priority, P2 the middle priority, and P3 the lowest priority. To understand all the interactions between the periods, we need to construct a time line equal in length to hyperperiod, which is 12 in this case.



Rate Monotonic Analysis

$$U = \sum_{i=1}^n \frac{T_i}{\tau_i}$$

- Familiar Utilization Equation

$$U = m(2^{1/m} - 1)$$

- New Constraint on Utilization
- What is value as $m \rightarrow$ infinity ?
- Only guarantees a schedule if utilization is $\leq 69\%$
- Doesn't mean can't happen for higher utilizations just not guaranteed !



Pretty Simple Code for RMS

```
/* processes[] is an array of process activation records,
   stored in order of priority, with processes[0] being
   the highest-priority process */
Activation_record processes[NPROCESSES];

void RMA(int current) { /* current = currently executing
process */
    int i;
    /* turn off current process (may be turned back on) */
    processes[current].state = READY_STATE;
    /* find process to start executing */
    for (i = 0; i < NPROCESSES; i++)
        if (processes[i].state == READY_STATE) {
            /* make this the running process */
            processes[i].state == EXECUTING_STATE;
            break;
        }
    }
}
```

FIGURE 6.12

C code for rate-monotonic scheduling.



Earliest Deadline First

Example 6.4

Earliest-deadline-first scheduling

Consider the following processes:

Process	Execution time	Period
P1	1	3
P2	1	4
P3	2	5

The hyperperiod is 60. In order to be able to see the entire period, we write it as a table:

- Dynamic Adjustment of Priority as Tasks run



Tasks....

```
void ATaskFunction( void *pvParameters )
```

```
/* Variables can be declared just as per a normal function. Each instance of a task created using this example function will have its own copy of the lVariableExample variable. This would not be true if the variable was declared static - in which case only one copy of the variable would exist, and this copy would be shared by each created instance of the task. (The prefixes added to variable names are described in section 1.5, Data Types and Coding Style Guide.) */
```

```
int32_t lVariableExample = 0;
```

```
/* A task will normally be implemented as an infinite loop. */
```

```
for( ;; )
```

```
{
```

```
    /* The code to implement the task functionality will go here. */
```

```
/* Should the task implementation ever break out of the above loop, then the task must be deleted before reaching the end of its implementing function. The NULL parameter passed to the vTaskDelete() API function indicates that the task to be deleted is the calling (this) task. The convention used to name API functions is described in section 0, Projects that use a FreeRTOS version older than V9.0.0 must build one of the heap_n.c files. From FreeRTOS V9.0.0 a heap_n.c file is only required if configSUPPORT_DYNAMIC_ALLOCATION is set to 1 in FreeRTOSConfig.h or if configSUPPORT_DYNAMIC_ALLOCATION is left undefined. Refer to Chapter 2, Heap Memory Management, for more information.
```

```
Data Types and Coding Style Guide. */
```

```
vTaskDelete( NULL );
```

```
}
```

Listing 12. The structure of a typical task function



“Creating” the Task

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        uint16_t usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask );
```

pvTaskCode

Pointer to your task

pcName

Just convenient name for your interest

usStackDepth

Each Task has it's own Stack

*pvParameters

Void pointer to optional parameters

uxPriority

Used by Scheduler

*pxCreatedTask

Handle Identified for Task



Two Different Example Tasks

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There
            nothing to do in here. Later examples will replace this crud
            loop with a proper delay/sleep function. */
        }
    }
}

void vTask2( void *pvParameters )
{
    const char *pcTaskName = "Task 2 is running\r\n";
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```



Using xTaskCreate() in Main

```
int main( void )
{
    /* Create one of the two tasks. Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate( vTask1, /* Pointer to the function that implements the task. */
                "Task 1", /* Text name for the task. This is to facilitate
                debugging only. */
                1000, /* Stack depth - small microcontrollers will use much
                less stack than this. */
                NULL, /* This example does not use the task parameter. */
                1, /* This task will run at priority 1. */
                NULL ); /* This example does not use the task handle. */

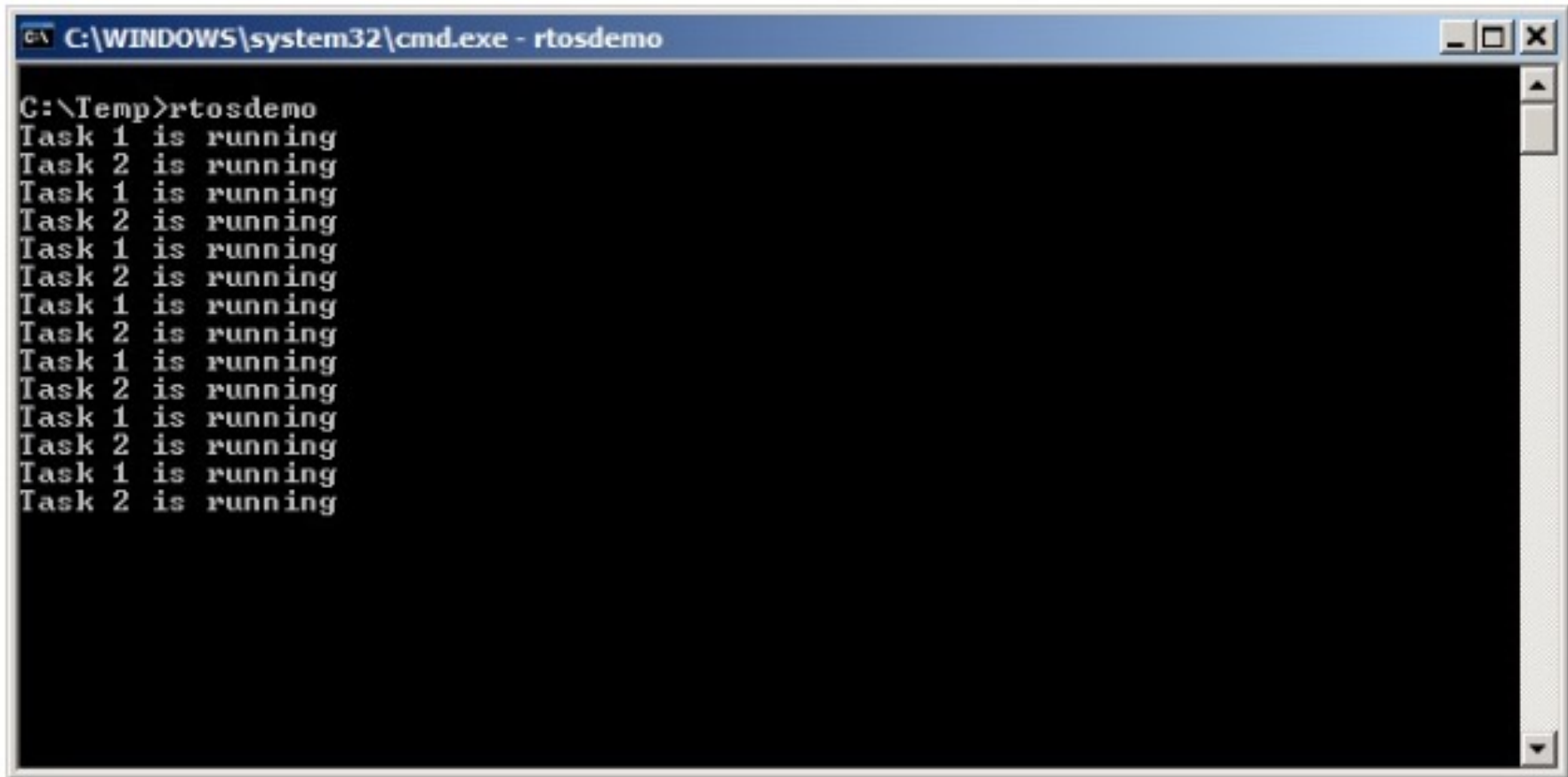
    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* 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 2 provides more information on heap memory management. */
    for( ;; );
}
```



Output

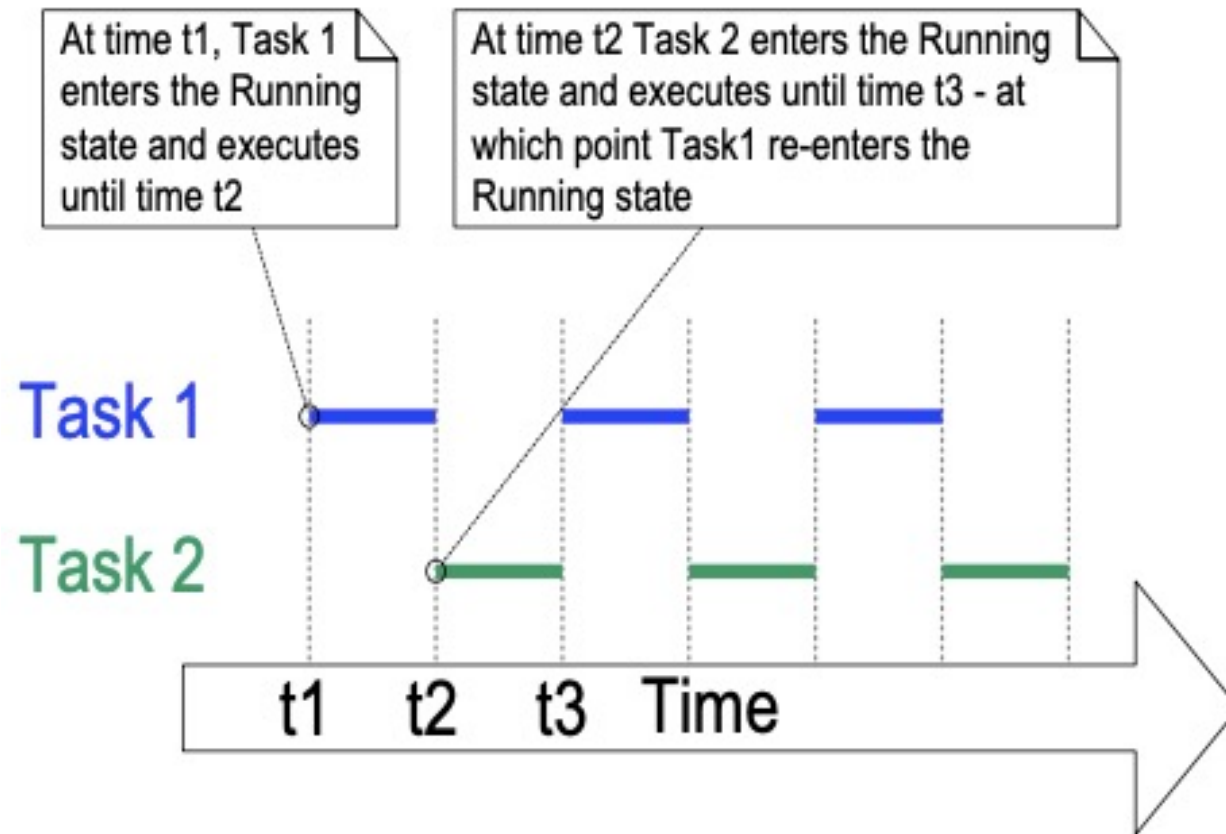


```
C:\WINDOWS\system32\cmd.exe - rtosdemo
C:\Temp>rtosdemo
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
```

Figure 10. The output produced when Example 1 is executed¹



Timer ticks



One function.....

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

/* 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. */
vPrintString( pcTaskName );

/* Delay for a period. */
for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
{
/* This loop is just a very crude delay implementation. There is
nothing to do in here. Later exercises will replace this crude
loop with a proper delay/sleep function. */
}
}
}
```

Listing 18. The single task function used to create two tasks in Example 2



As multiple "tasks"

```
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate(    vTaskFunction,          /* Pointer to the function that
                                                    implements the task. */
                  "Task 1",                /* Text name for the task. This is to
                                                    facilitate debugging only. */
                  1000,                    /* Stack depth - small microcontrollers
                                                    will use much less stack than this. */
                  (void*)pcTextForTask1,   /* Pass the text to be printed into the
                                                    task using the task parameter. */
                  1,                        /* This task will run at priority 1. */
                  NULL );                  /* The task handle is not used in this
                                                    example. */

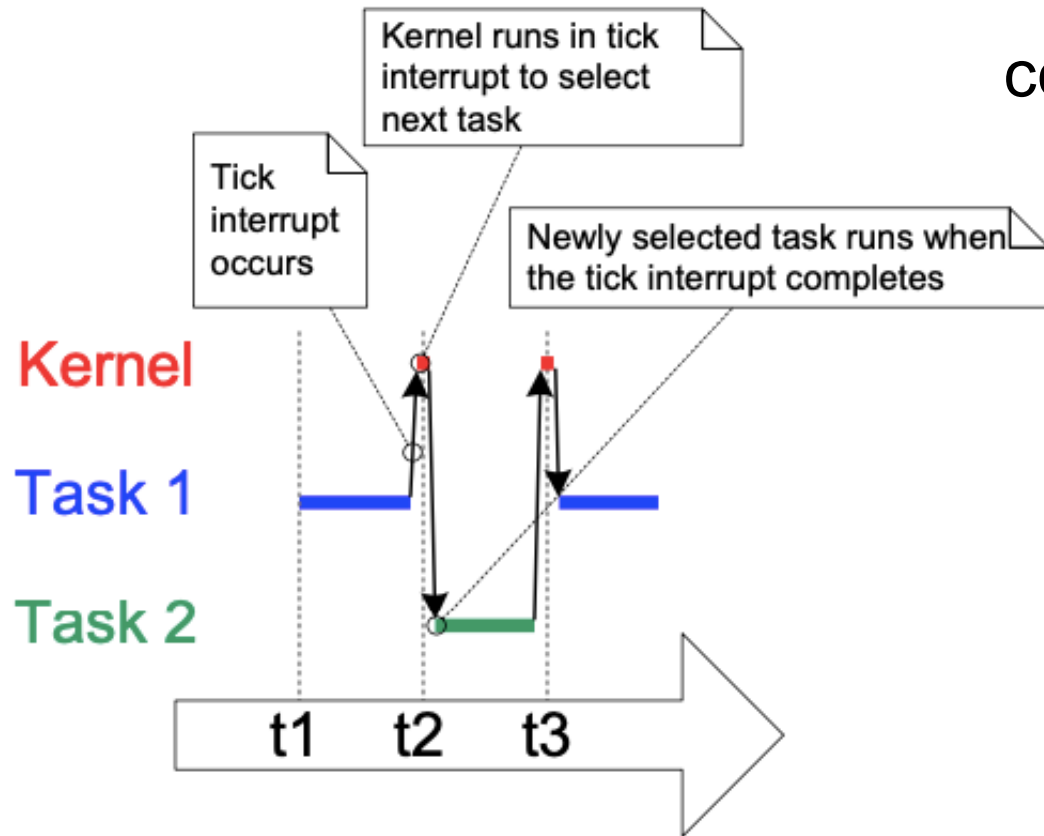
    /* Create the other task in exactly the same way. Note this time that multiple
    tasks are being created from the SAME task implementation (vTaskFunction). Only
    the value passed in the parameter is different. Two instances of the same
    task are being created. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

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



Time Measurement and Tick Interrupt

In FreeRTOSConfig.h
`configTICK_RATE_HZ`



Handy Macro function.....

```
TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```



Introducing Priorities

```
/* Define the strings that will be passed in as the task parameters. These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create the first task at priority 1. The priority is the second to last
parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2, which is higher than a priority of 1.
The priority is the second to last parameter. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

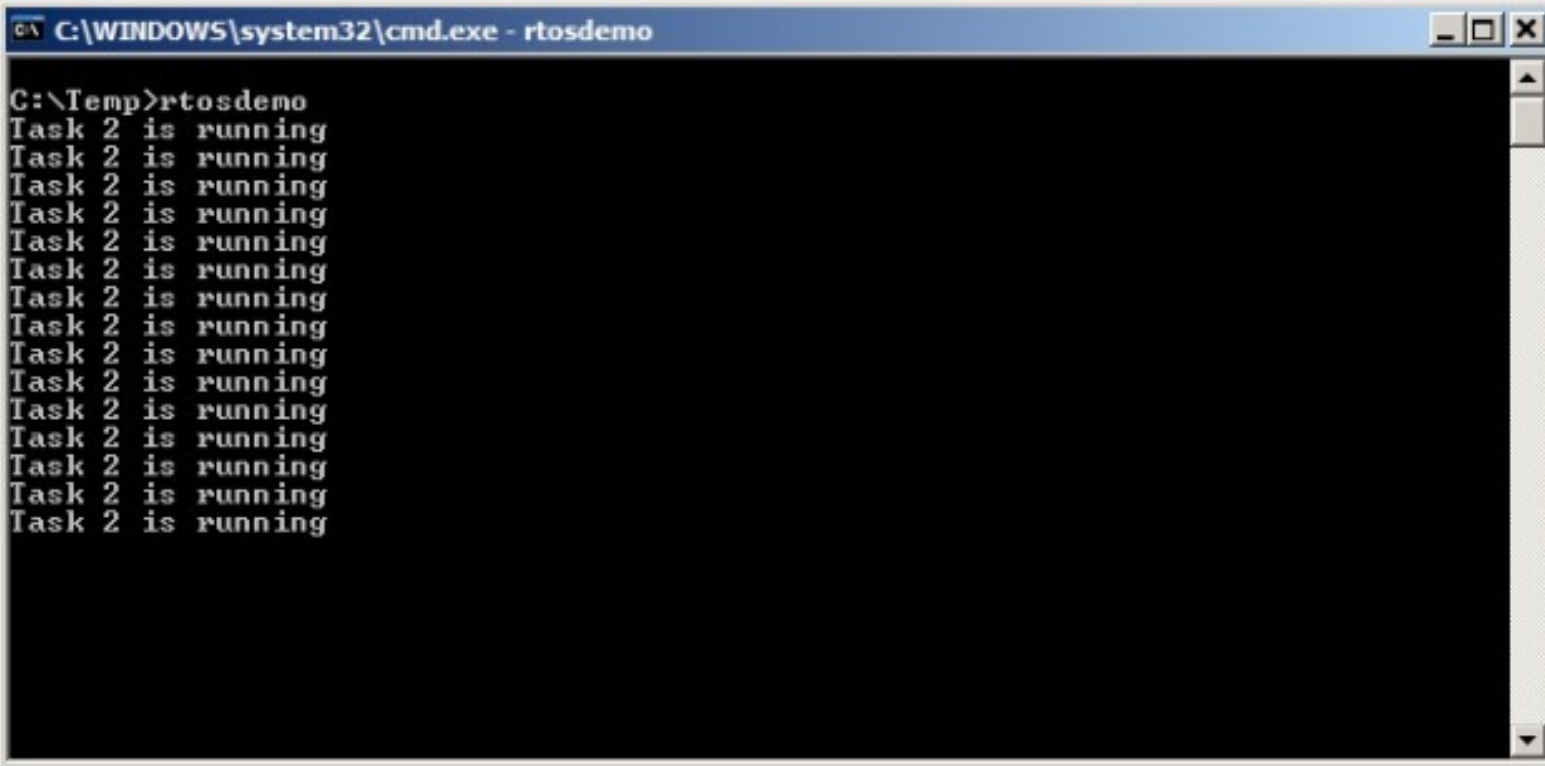
    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* Will not reach here. */
    return 0;
}
```

For prior example, what would be the output ?



Introducing Priorities



```
C:\WINDOWS\system32\cmd.exe - rtsdemo
C:\Temp>rtsdemo
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
```

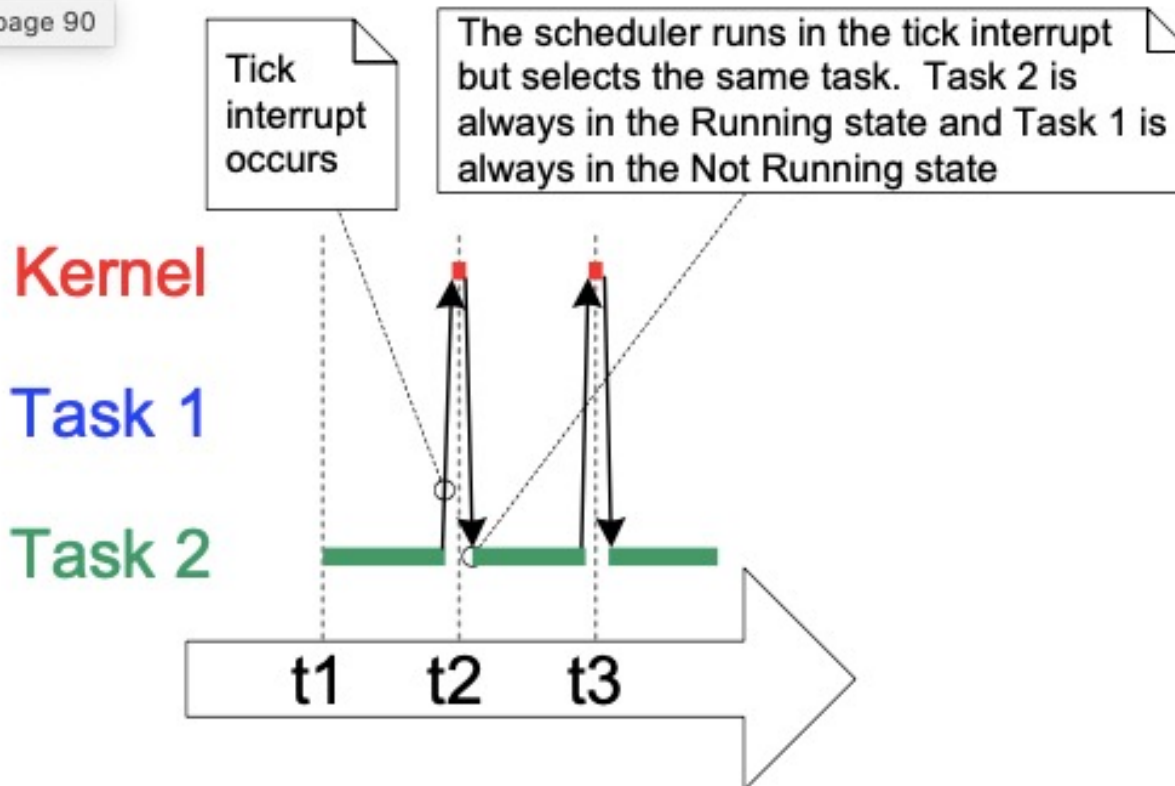
Figure 13. Running both tasks at different priorities

Starvation ! What went wrong (if anything ?)



Introducing Priorities

Go to page 90



Task 2 is always in “running” state

Need to either suspend or block it from being considered



Rounding out scheduler FSM

Move Task 2 into a non-running waiting state:

- Blocked
- Suspended

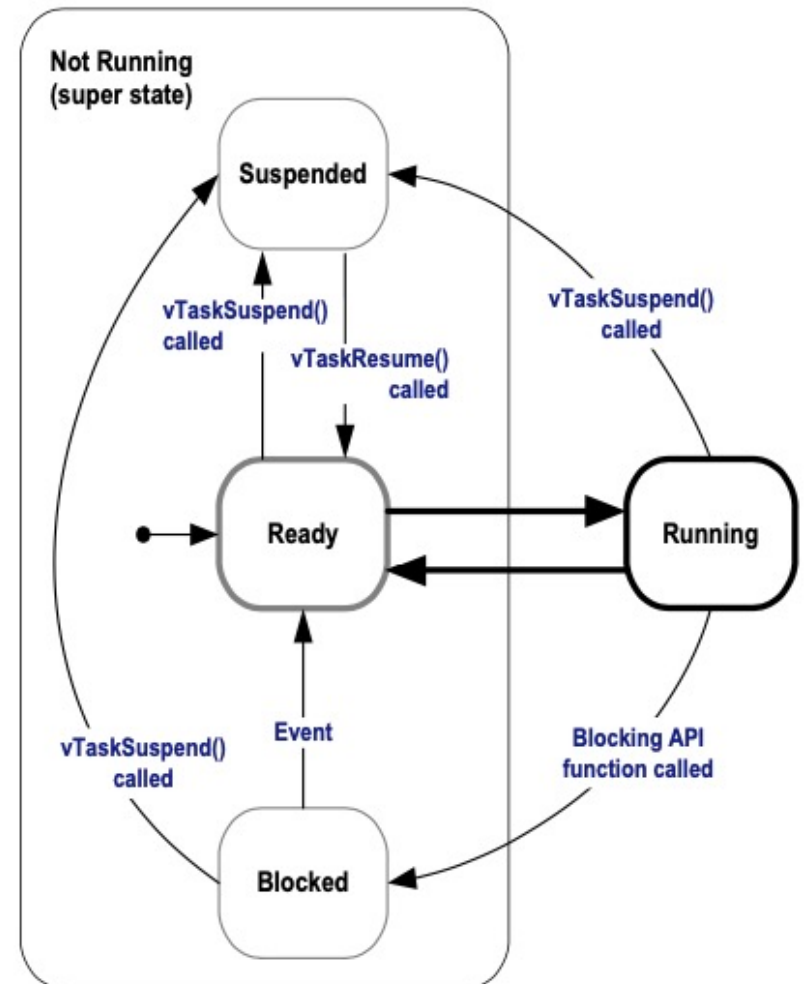


Figure 15. Full task state machine



Implicit Time Delay

```
void vTaskDelay( TickType_t xTicksToDelay );
```

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

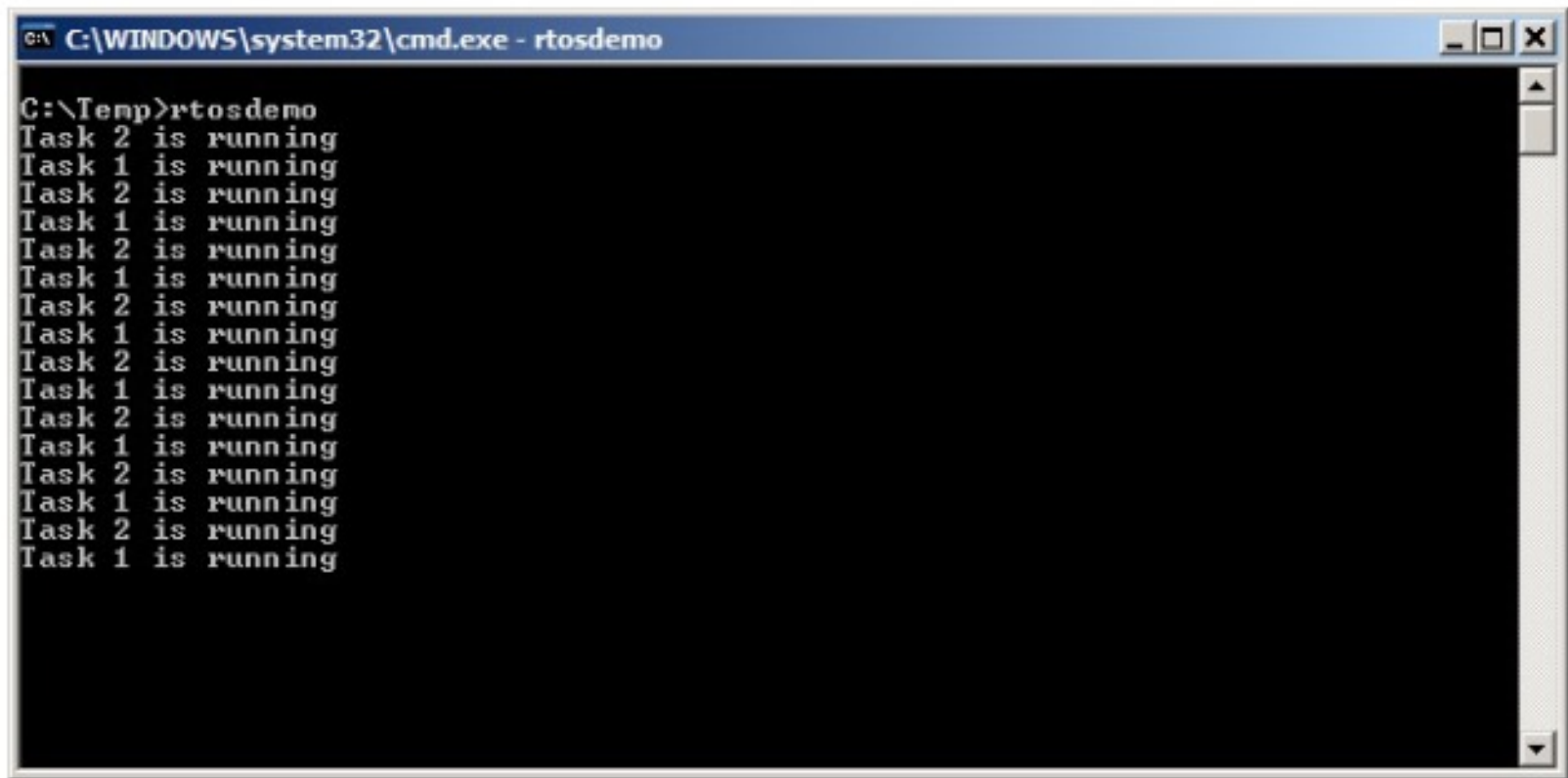
/* The string to print out is passed in via the parameter. Cast this to a
character pointer. */
pcTaskName = ( char * ) pvParameters;

/* As per most tasks, this task is implemented in an infinite loop. */
for( ;; )
{
    /* Print out the name of this task. */
    vPrintString( pcTaskName );

    /* Delay for a period. This time a call to vTaskDelay() is used which places
the task into the Blocked state until the delay period has expired. The
parameter takes a time specified in 'ticks', and the pdMS_TO_TICKS() macro
is used (where the xDelay250ms constant is declared) to convert 250
milliseconds into an equivalent time in ticks. */
    vTaskDelay( xDelay250ms );
}
}
```



New Output



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
C:\Temp>rtosdemo
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
```

Figure 16. The output produced when Example 4 is executed



Better but is it periodic ?

vTaskDelay() does not guarantee frequency at which task runs is fixed.

~Why ?~



Better but is it periodic ?

`vTaskDelay()` does not guarantee frequency at which task runs is fixed.

~Why ?~

Time when task leaves the Blocked state is ***relative to when `vTaskDelay()` called*** .



Better but is it periodic ?

vTaskDelay() does not guarantee frequency at which task runs is fixed.

~Why ?~

Time when task leaves the Blocked state is ***relative to when vTaskDelay() called*** .

A better solution: TaskDelayUntil()



The vTaskDelayUntil()

```
void vTaskDelayUntil(  
TickType_t* pxPreviousWakeTime,  
TickType_t xTimeIncrement );
```

Automatically updated!

Set task period here



Prior Example.....

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

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

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count. Note that this is the only time the variable is written to explicitly.
    After this xLastWakeTime is automatically updated within vTaskDelayUntil(). */
    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* This task should execute every 250 milliseconds exactly. As per
        the vTaskDelay() function, time is measured in ticks, and the
        pdMS_TO_TICKS() macro is used to convert milliseconds into ticks.
        xLastWakeTime is automatically updated within vTaskDelayUntil(), so is not
        explicitly updated by the task. */
        vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );
    }
}
```

Listing 25. The implementation of the example task using vTaskDelayUntil()



Introducing the Idle Task...

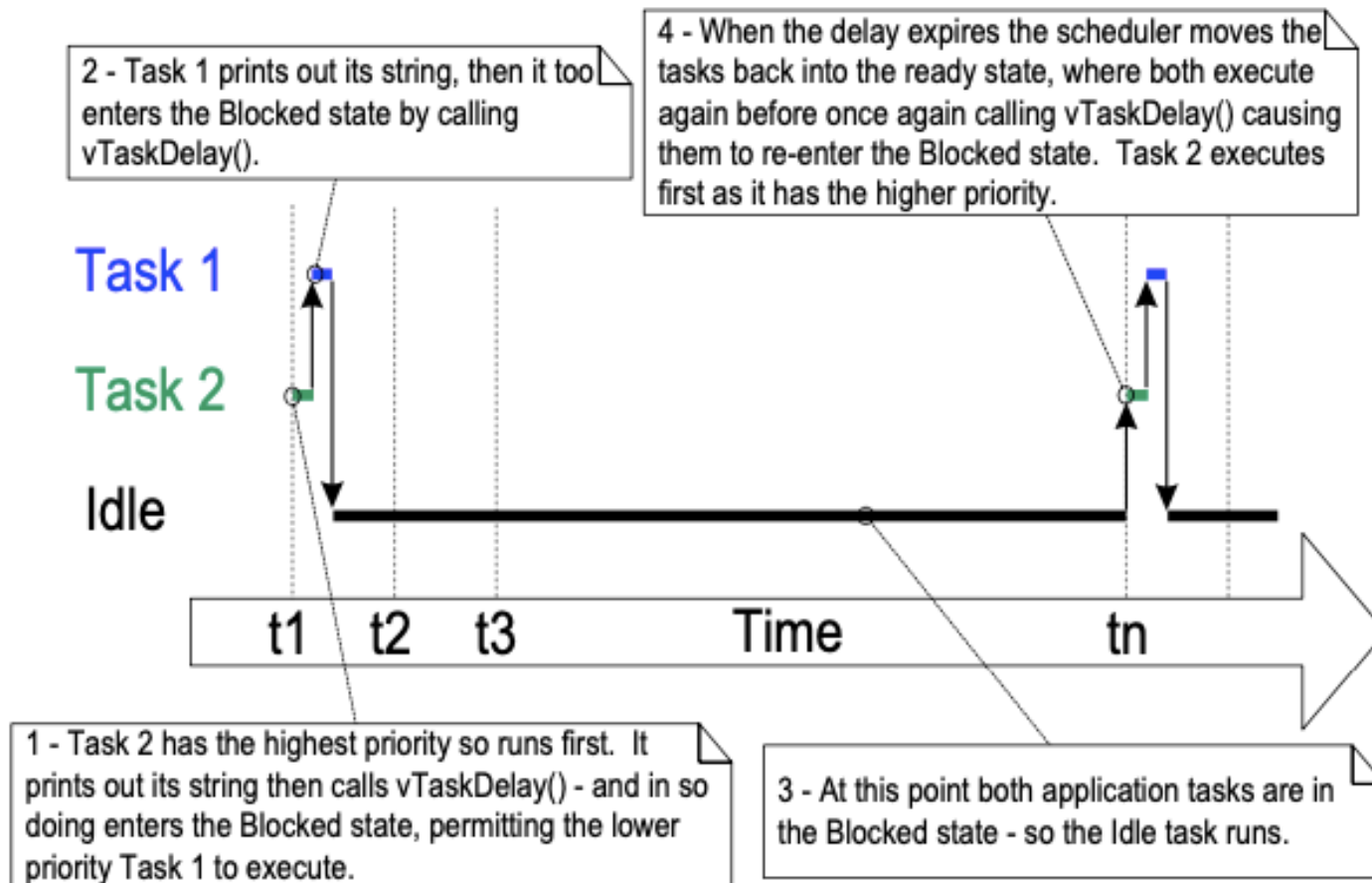


Figure 17. The execution sequence when the tasks use `vTaskDelay()` in place of the NULL loop



Mixing behaviors....

1. Two tasks are created at priority 1. These do nothing other than continuously print out a string.
2. A third task is then created at priority 2, (above the priority of the other two tasks). The third task also just prints out a string, but this time periodically, so uses the `vTaskDelayUntil()` API function to place itself into the Blocked state between each print iteration.



ing tasks

```
void vContinuousProcessingTask( 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. This task just does this repeatedly
        without ever blocking or delaying. */
        vPrintString( pcTaskName );
    }
}
```

Listing 26. The continuous processing task used in Example 6

```
void vPeriodicTask( void *pvParameters )
{
    TickType_t xLastWakeTime;
    const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count. Note that this is the only time the variable is explicitly written to.
    After this xLastWakeTime is managed automatically by the vTaskDelayUntil()
    API function. */
    xLastWakeTime = xTaskGetTickCount();

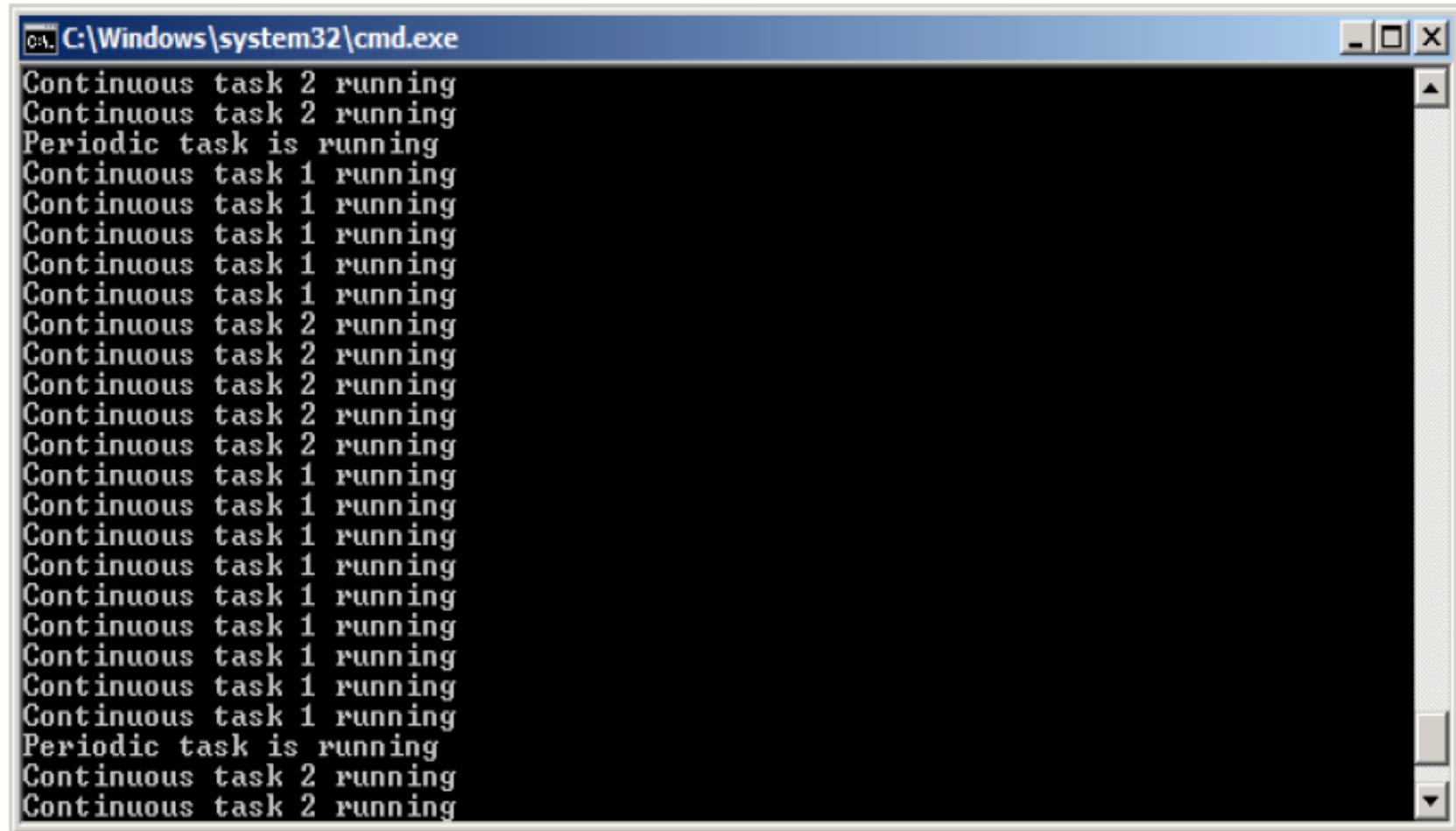
    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Periodic task is running\r\n" );

        /* The task should execute every 3 milliseconds exactly - see the
        declaration of xDelay3ms in this function. */
        vTaskDelayUntil( &xLastWakeTime, xDelay3ms );
    }
}
```

Listing 27. The periodic task used in Example 6



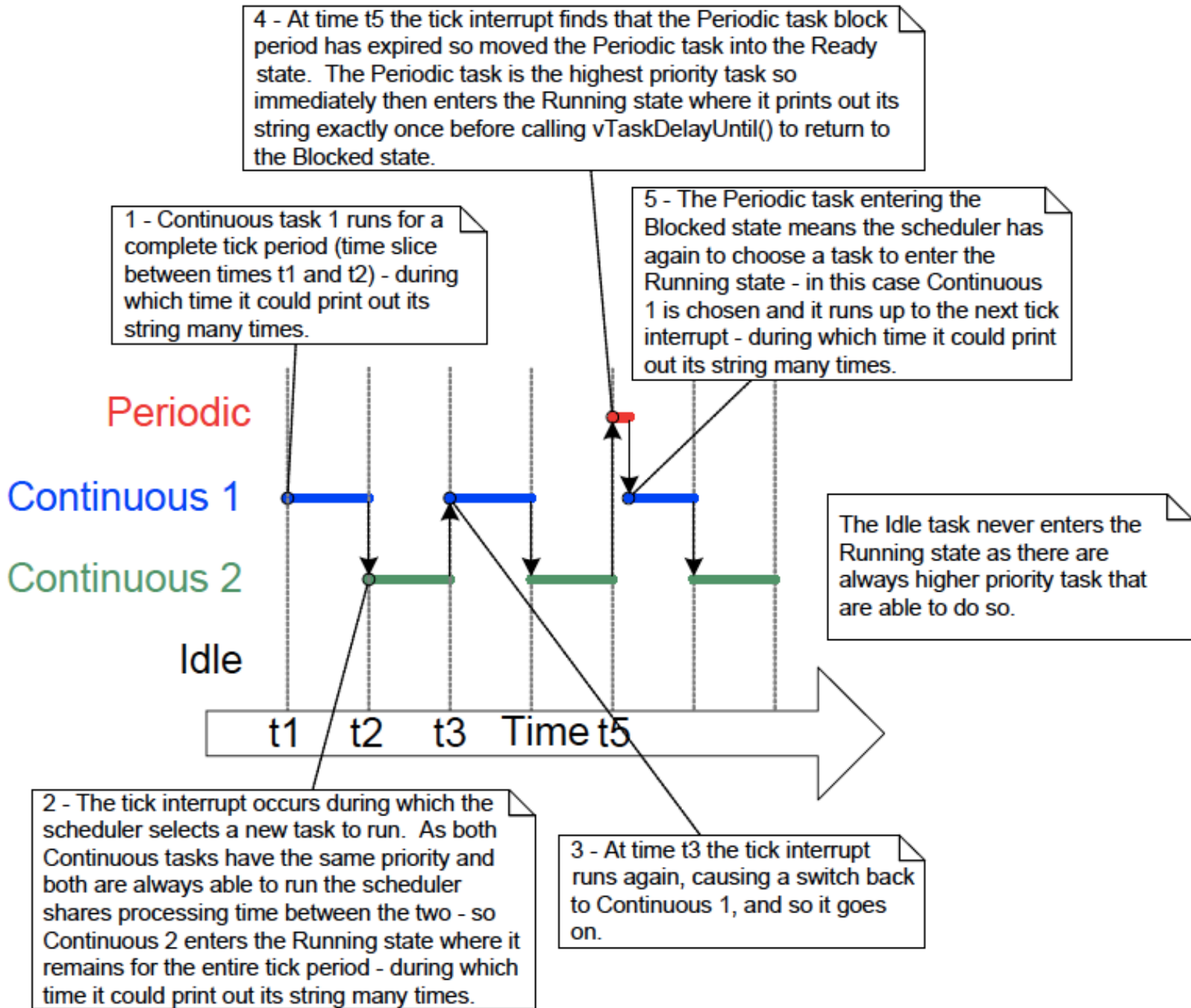
Output



```
C:\Windows\system32\cmd.exe
Continuous task 2 running
Continuous task 2 running
Periodic task is running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Periodic task is running
Continuous task 2 running
Continuous task 2 running
```

Figure 19. The output produced when Example 6 is executed





FreeRTOS Sched. Options

FreeRTOSConfig.h.

configUSE_PREEMPTION /*

configUSE_TIME_SLICING /*no round robin for tasks
of equal priority*/

configUSE_TICKLESS_IDLE /*turns tick interrupt off */



Preemption=1, time slicing=1

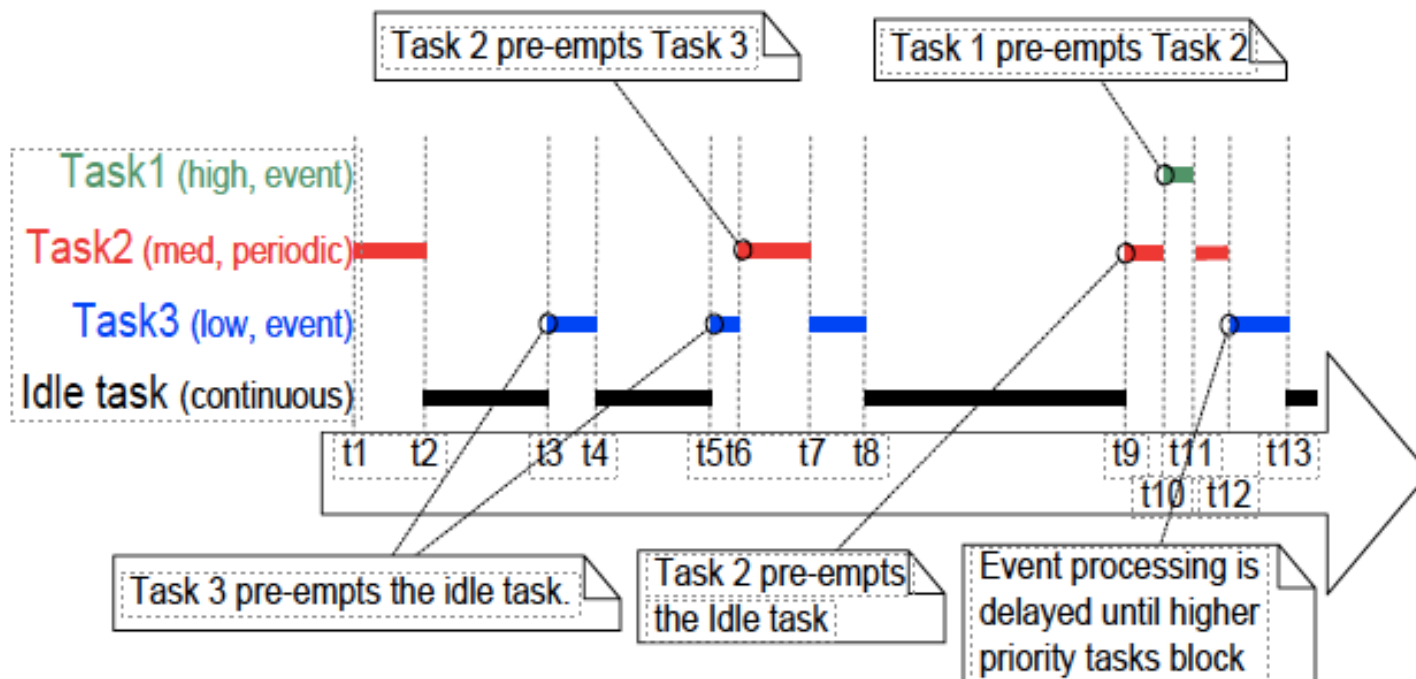


Figure 26. Execution pattern highlighting task prioritization and pre-emption in a hypothetical application in which each task has been assigned a unique priority

Preemption=0, time slicing=1

Two tasks, TaskL=low priority, TaskH=high priority. Assume TaskL has processor.

Q: When does TaskH get to run ?

A: When TaskL gives up processor.

How ?

Yield() or block on mutex.

Lets see.....



Preemption=0, time slicing=1

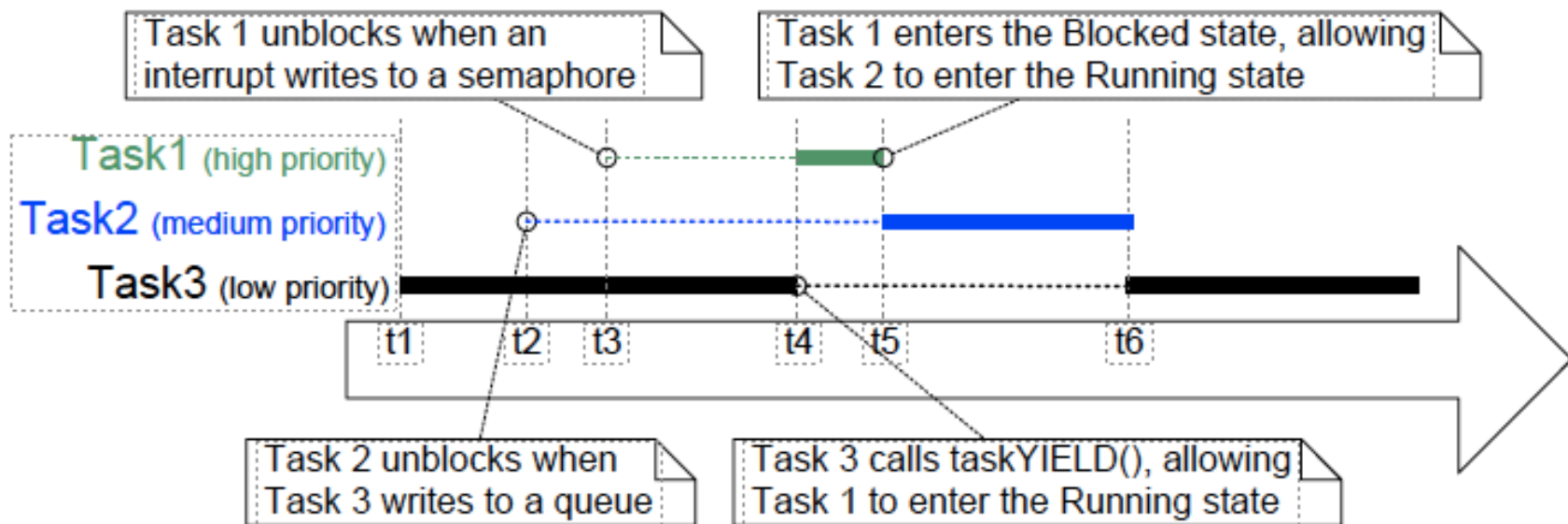


Figure 30 Execution pattern demonstrating the behavior of the co-operative scheduler



Preemption=1, time slicing=0

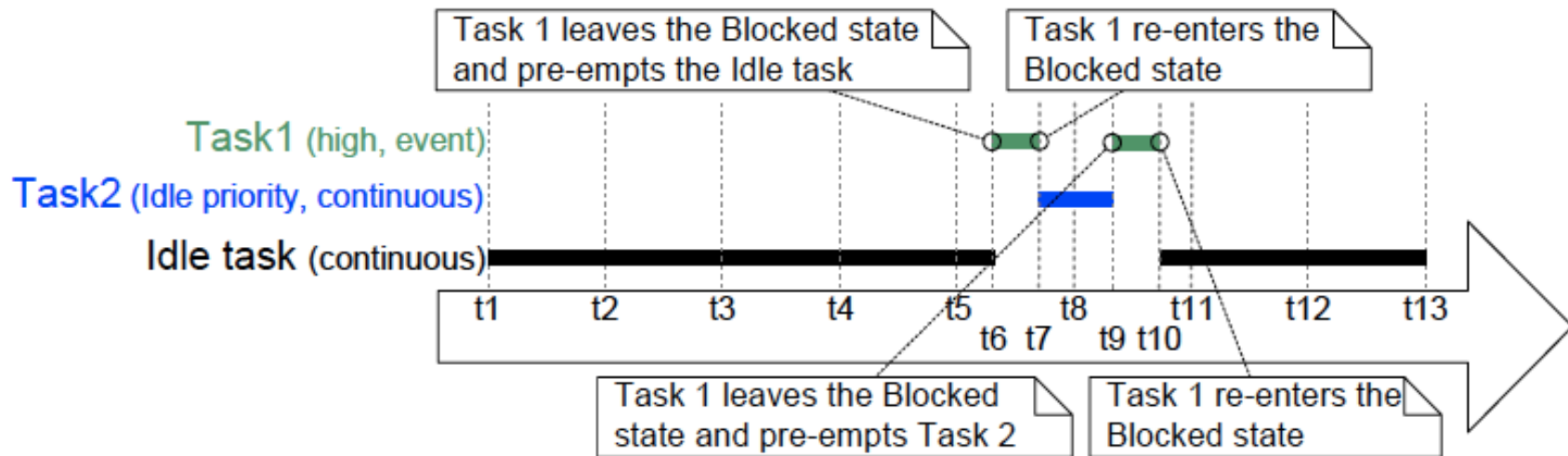


Figure 29 Execution pattern that demonstrates how tasks of equal priority can receive hugely different amounts of processing time when time slicing is not used



Semaphores in FreeRTOS

6.4 Binary Semaphores Used for Synchronization	191
The xSemaphoreCreateBinary() API Function	194
The xSemaphoreTake() API Function	194
The xSemaphoreGiveFromISR() API Function	196
Example 16. Using a binary semaphore to synchronize a task with an interrupt ...	198
Improving the Implementation of the Task Used in Example 16	202
6.5 Counting Semaphores	208
The xSemaphoreCreateCounting() API Function	210
Example 17. Using a counting semaphore to synchronize a task with an interrupt .	211



Semaphores in FreeRTOS

6.4 Binary Semaphores Used for Synchronization	191
The xSemaphoreCreateBinary() API Function	194
The xSemaphoreTake() API Function	194
The xSemaphoreGiveFromISR() API Function	196
Example 16. Using a binary semaphore to synchronize a task with an interrupt ...	198
Improving the Implementation of the Task Used in Example 16	202

SemaphoreHandle_t xSemaphoreCreateBinary(void);

SemaphoreHandle_t xSemaphoreCreateMutex(void)



Semaphores in FreeRTOS

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

```
SemaphoreHandle_t xSemaphoreCreateMutex( void )
```

```
BaseType_t xSemaphoreTake(  
    SemaphoreHandle_t xSemaphore,  
    TickType_t xTicksToWait );
```

Can be used on either binary/counting semaphores and mutexes



Semaphores in FreeRTOS

SemaphoreHandle_t xSemaphoreCreateBinary(void);

SemaphoreHandle_t xSemaphoreCreateMutex(void)

**BaseType_t xSemaphoreTake(
SemaphoreHandle_t xSemaphore,
TickType_t xTicksToWait);**



Semaphores in FreeRTOS

SemaphoreHandle_t xSemaphoreCreateBinary(void);

SemaphoreHandle_t xSemaphoreCreateMutex(void)

**BaseType_t xSemaphoreTake(
 SemaphoreHandle_t xSemaphore,
 TickType_t xTicksToWait);**

Amount of time to suspend if not successful

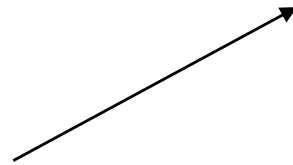


Semaphores in FreeRTOS

SemaphoreHandle_t xSemaphoreCreateBinary(void);

SemaphoreHandle_t xSemaphoreCreateMutex(void)

**BaseType_t xSemaphoreTake(
 SemaphoreHandle_t xSemaphore,
 TickType_t xTicksToWait);**



Amount of time to suspend if not successful

0 = asynchronous, non blocking

X = suspend for x timer ticks

portMAX_DELAY

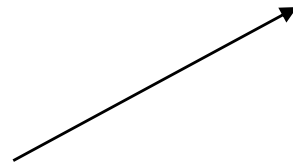


Semaphores in FreeRTOS

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

```
SemaphoreHandle_t xSemaphoreCreateMutex( void )
```

```
BaseType_t xSemaphoreTake(  
                        SemaphoreHandle_t xSemaphore,  
                        TickType_t xTicksToWait );
```



Amount of time to suspend if not successful

0 = asynchronous, non blocking

X = suspend for x timer ticks

portMAX_DELAY

