FreeRTOS

FreeRTOS Primitives, Data structures, and inter-task communication

Binary Semaphore

Semaphores are used to signal/synchronize tasks as well as protect resources.

A binary semaphore can (and should) be used as a means of signaling a task. This signal can come from an interrupt service routine or from another task. A semaphore is an RTOS primitive and is guaranteed to be thread-safe.

Design Pattern

Wake Up On Semaphore

The idea here is to have a task that is waiting on a semaphore and when it is given by an ISR or an other task, this task unblocks, and runs its code. This results in a task that usually sleeping/blocked and not utilizing CPU time unless its been called upon. In FreeRTOS, there is a similar facility provided which is called 'deferred interrupt processing'. This could be used to signal an emergency shutdown procedure when a button is triggered, or to trigger a procedure when the state of the system reaches a fault condition. Sample code below:

```
/* Declare the instance of the semaphore but not that you have to still 'create' it which is done in th
SemaphoreHandle_t xSemaphore;
```

```
void vWaitOnSemaphore( void * pvParameters )
{
    while(1)
    {
        /* Wait forever until a the semaphore is sent/given */
        if(xSemaphoreTake(xSemaphore, portMAX_DELAY))
        {
            printf("Semaphore taken\n");
            /* Do more stuff below ... */
        }
    }
}
void vSemaphoreSupplier( void * pvParameters )
{
    while(1)
    {
        if(checkButtonStatus())
        {
            xSemaphoreGive(xSemaphore);
        }
        /* Do more stuff ... */
    }
}
int main()
{
    /* Semaphore starts 'empty' when you create it */
    xSemaphore = xSemaphoreCreateBinary();

    /* Create the tasks */
    const uint32_t STACK_SIZE_WORDS = 128;
    xTaskCreate(vWaitOnSemaphore, "Waiter", STACK_SIZE_WORDS, NULL, tskIDLE_PRIORITY+1, NULL);
    xTaskCreate(vSemaphoreSupplier, "Supplier", STACK_SIZE_WORDS, NULL, tskIDLE_PRIORITY+1, NULL);

    /* Start Scheduler */
    vTaskStartScheduler();}
```

Code Block 1. How to use Semaphores and use as a wake up pattern

Semaphore as a flag

The idea of this is to have a code loop that checks the semaphore periodically with the 'block time' of your choice. The task will only react when it notices that the semaphore flag has been given. When your task takes it, it will run an if statement block and continue its loop. Keep in mind this will consume your flag, so the consumer will loop back and check for the presence of the new flag in the following loop.

```
void vWaitOnSemaphore( void * pvParameters )
{
    while(1)
    {
        /* Check the semaphore if it was set */
        if(xSemaphoreTake(xSemaphore, 0))
        {
            printf("Got the Semaphore, consumed the flag indicator.");
            /* Do stuff upon taking the semaphore successfully ... */
        }

        /* Do more stuff ... */
    }}
```

Code Block 2. Semaphores as a consumable flag

Interrupt Signal from ISR

This is useful, because ISRs should be as short as possible as they interrupt the software or your RTOS tasks. In this case, the ISR can defer the work to a task, which means that the ISR runtime is short. This is important because when you enter an interrupt function, the interrupts are disabled during the ISRs execution. The priority of the task can be configured based on the importance of the task reacting to the semaphore.

You may not want to defer interrupt processing if the ISR is so critical that the time it takes to allow RTOS to run is too much. For example, a power failure interrupt.

```
void systemInterrupt()
{
    xSemaphoreGiveFromISR(xSemaphore);
}
void vSystemInterruptTask(void * pvParameter)
{
    while(1)
    {
        if(xSemaphoreTake(xSemaphore, portMAX_DELAY))
        {
            // Process the interrupt
        }
    }}
```

?

Code Block 3. Semaphore used within an ISR

NOTICE: *The* ***FromISR*** *after the* ***xSemaphoreGive*** *API call? If you are making an RTOS API call from an ISR, you must use the* ***FromISR*** *variant of the API call. Undefined behavior otherwise like freezing the system.*

Mutexes

?

Semaphores vs Mutexs

Semaphores and mutexes are nearly the same construct except that mutexes have the feature of priority inheritance, where in a low priority task can inheritate the priority of a task with greater priority if the higher priority task attempts to take a mutex that the low priority task possess.

Priority Inversion Using a Semaphore

Below is an illustration of the scenario where using a semaphore can cause priority inversion.

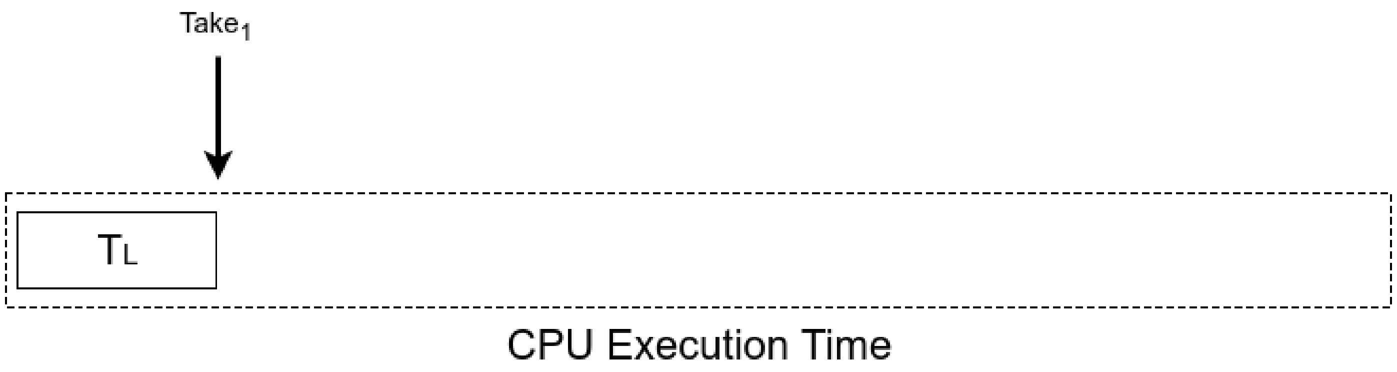

Figure 1. Low priority task is currently running and takes a semaphore.

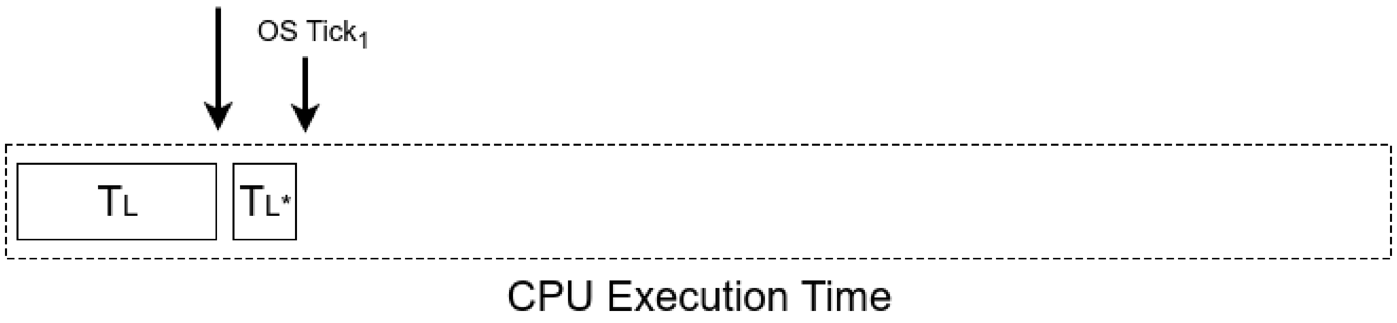

Figure 2. OS Tick event occurs.

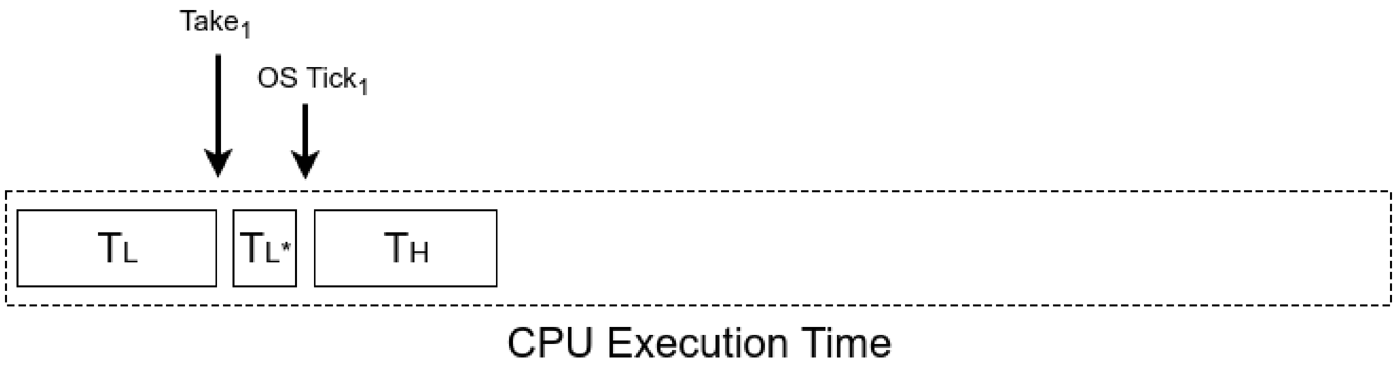

Figure 3. High priority task is ready to run and selected to run.

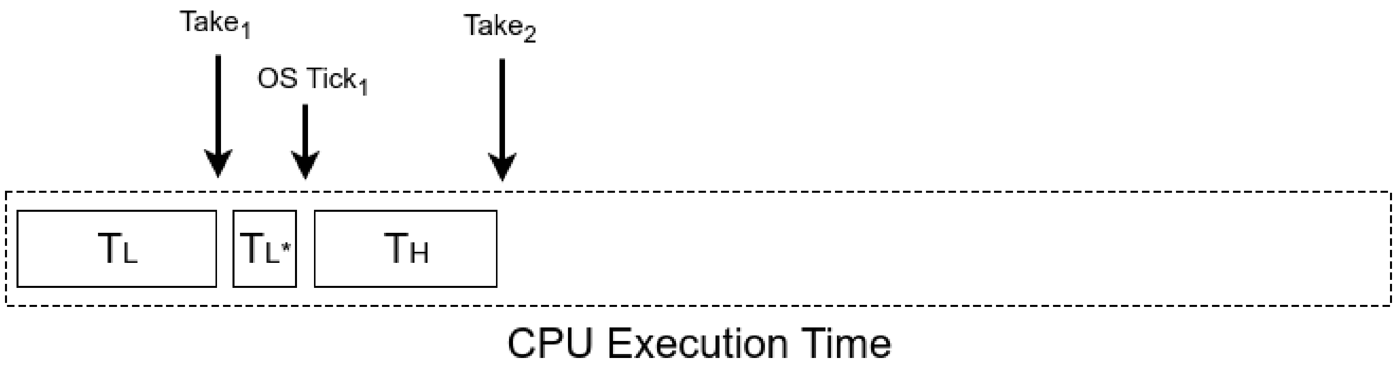

Figure 4. High priority task attempts to take semaphore and blocks.

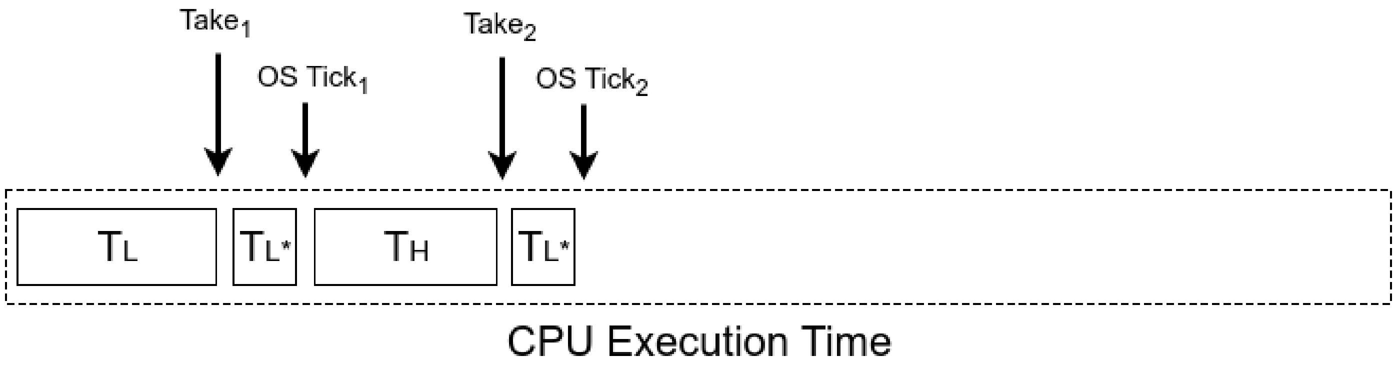

Figure 5. Since high priority task is blocked, the next ready task that can run is the low priority task. The OS tick event occurs.

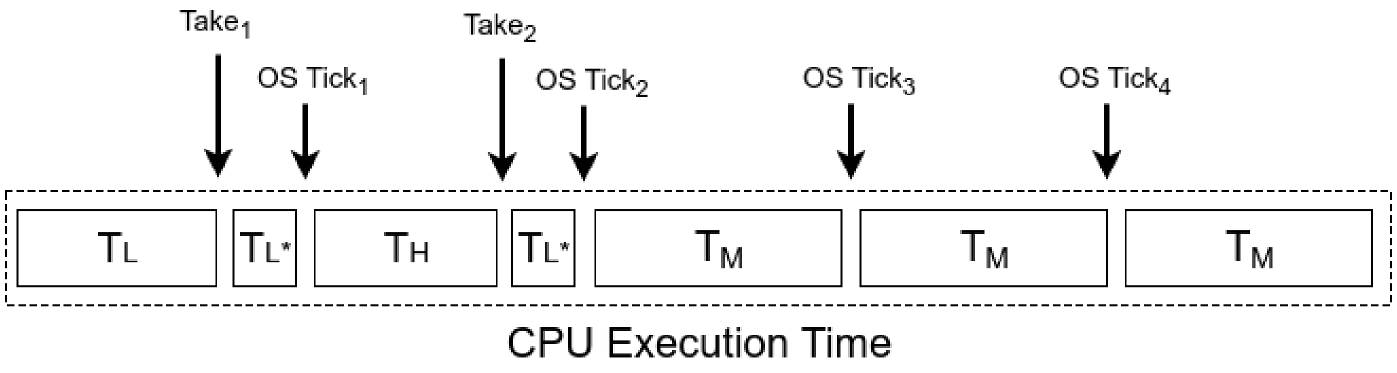

Figure 6. The OS tick event occurs, a middle priority task, that never sleeps is ready to run, it begins to run, high priority task is blocked on semaphore and low priority task is blocked by the middle priority task. This is priority inversion, where a medium priority task is running over a higher priority task.

Priority Inheritance using Mutex

Priority inheritance is the means of preventing priority inversion.

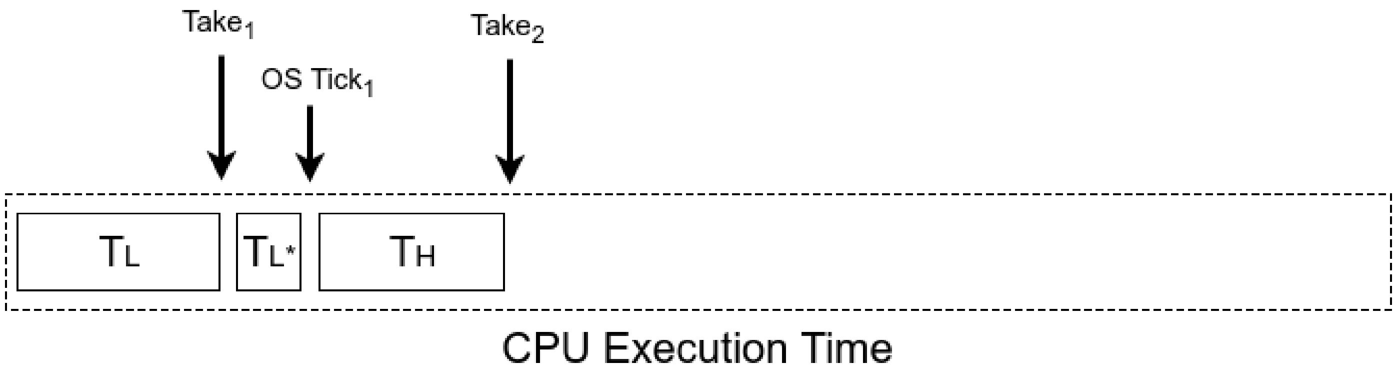

Figure 7. Moving a bit further, the high priority task attempts to take the Mutex

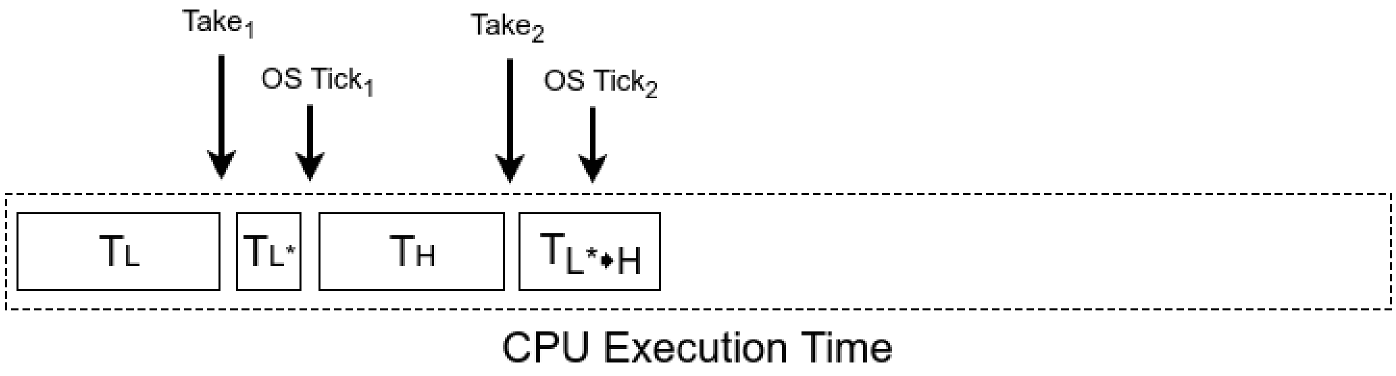

Figure 6. Low priority task inherates the highest priority of the task that attempts to take the mutex it posses.

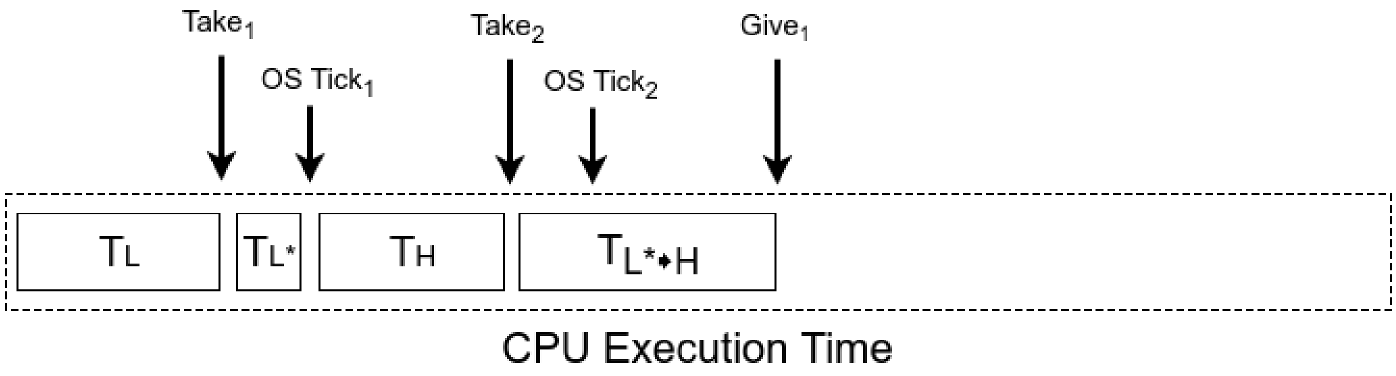

Figure 7. OS Tick$_2$ occurs, and medium priority task is ready, but the low priority task has inheritated a higher priority, thus it runs above the medium priority task.

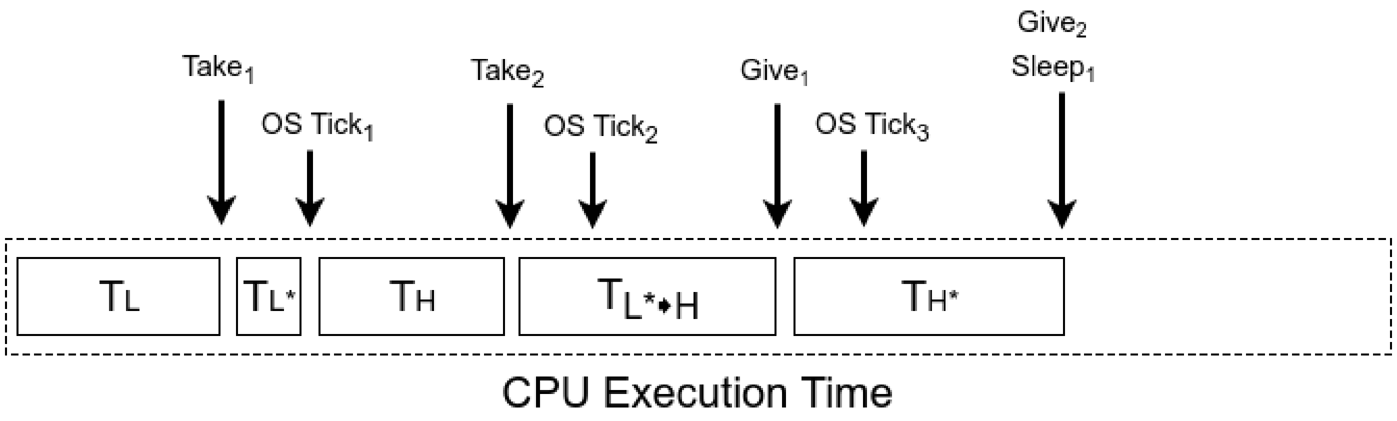

Figure 7. Low priority task gives the mutex, low priority task de-inheritates its priority, and the high task immediately begins to run. It will run over the medium task.

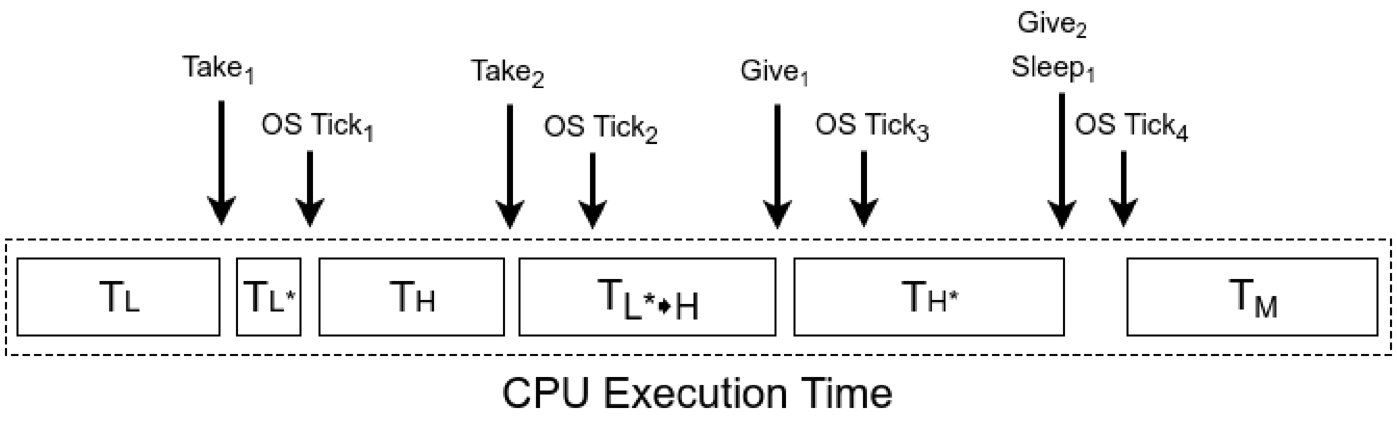

Figure 7. At give$_2$ high priority task releases the mutex and sleeps. Some time elapses, and then the medium task begins to run. No priority inversion occurs in this scenario, the RTOS rule of highest priority runs first is held.

Design Pattern

The design pattern for a mutex should be exclusively used as a **protection token**. Mutexes can be used in place of as semaphores but the addition work of priority inheritance will cause this approach to take longer and thus be less efficient than a semaphore.

```
// In main(), initialize your Mutex:
SemaphoreHandle_t spi_bus_lock = xSemaphoreCreateMutex();
void vTaskOne()
{
    while(1) {
        if(xSemaphoreGet(spi_bus_lock, 1000)) {
            // Use Guarded Resource

            // Give Semaphore back:
            xSemaphoreGive(spi_bus_lock);
        }
    }
}
void vTaskTwo()
{
    while(1) {
        if(xSemaphoreGet(spi_bus_lock, 1000)) {
            // Use Guarded Resource

            // Give Semaphore back:
            xSemaphoreGive(spi_bus_lock);
        }
    }
}
```

Queues

RTOS Queues

There are standard queues, or <vector> in C++, but RTOS queues should almost always be used in your application because they are thread-safe (no race conditions with multiple tasks), and they co-operate with your RTOS to schedule the tasks. For instance, your task could optionally sleep while receiving data if the queue is empty, or it can sleep while sending the data if the queue is full.

Queues vs. Semaphore for "Signalling"

Semaphores may be used to "signal" between two contexts (tasks or interrupts), but they do not contain any payload. For example, for an application that captures a keystroke inside of an interrupt, it could "signal" the data processing task to awake upon the semaphore, however, there is no payload associated with it to identify what keystroke was input. With an RTOS queue, the data processing task can wake upon a payload and process a particular keystroke.

The data-gathering tasks can simply send the key-press detected to the queue, and the processing task can receive items from the queue, and perform the corresponding action. Moreover, if there are no items in the queue, the consumer task (the processing one) can sleep until data becomes available. You can see how this scheme lends itself well to having multiple ISRs queue up data for a task (or multiple tasks) to handle.

Design Pattern

After looking through the sample code below, you should then watch this video.

Let's study an example of two tasks communicating to each other over a queue.

```
QueueHandle_t q;
void producer(void *p)
{
  int value_put_sent_on_queue = 0;

  while (1) {
    vTaskDelay(100);
    xQueueSend(q, &value_put_sent_on_queue, 0); // TODO: Find out the significance of the parameters o
    ++value_put_sent_on_queue;
  }
}
void consumer(void *p)
{
  int variable_to_store_value_retreived_from_queue;
  while (1) {
    // We do not need vTaskDelay() because this task will sleep for up to 100ms until there is an item
    if (xQueueReceive(q, &variable_to_store_value_retreived_from_queue, 100)) {
      printf("Got %i\n", variable_to_store_value_retreived_from_queue);
    }
    else {
      puts("Timeout --> No data received");
    }
  }
}
void main(void)
{
  // Queue handle is not valid until you create it
  q = xQueueCreate(10, sizeof(int));
}
```

Example Queue usage with Interrupts

```
// Queue API is special if you are inside an ISR
void uart_rx_isr(void)
{
  xQueueSendFromISR(q, &x, NULL); // TODO: Find out the significance of the parameters
}
void queue_rx_task(void *p)
{
  int x;
  // Receive is the usual receive because we are not inside an ISR
  while (1) {
    xQueueReceive(q, &x, portMAX_DELAY);
  }
}
```

Additional Information

Queue Management (Amazon Docs)

Queue API (FreeRTOS Docs)

Task Suspension and Resumption

A freeRTOS task that is currently running can be suspended by another task or by its own task. A suspended task will not get any processing time from the micro-controller. Once suspended, it can only be resumed by another task.

API which can suspend a single task is:

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend );
```

> Refer this link to explore more details on the API. **https://www.freertos.org/a00130.html**

API to suspend the scheduler is:

```
void vTaskSuspendAll( void );
```

> Refer this link to explore more details on the API. **https://www.freertos.org/a00134.html**

API to resume a single task:

```
void vTaskResume( TaskHandle_t xTaskToResume );
```

> Refer this link to explore more details. **https://www.freertos.org/a00131.html**

?

API to resume the scheduler:

```
BaseType_t xTaskResumeAll( void );
```

?

> Refer this link to explore more details. **https://www.freertos.org/a00135.html**

?

?

Although this may seem useful, do refrain from using these.

Many engineers will use task resume and suspend to control the flow of the system, but this is rarely done elegantly, and can result in weird behavior if abused.

Use these sparingly and only when all other options have been deemed nonoptimal and this is the most optimal solution.

Event Groups

Event group APIs can be used to monitor a set of tasks. A software watchdog in an embedded system can make use of event groups for a group of tasks and notify/alert the user if any of the task misbehaves.

Each task uses an event bit. After every successful iteration of the task, the bit can be set by the task to mark completion. The event bits are then checked in the watchdog task to see if all the tasks are running successfully. If any of the bits are not set, then watchdog task can alert about the task to the user.

Below are the APIs that can be used. Refer to each of the API to understand how to use them in your application.

- xEventGroupCreate
- xEventGroupCreateStatic
- xEventGroupWaitBits
- xEventGroupSetBits
- xEventGroupSetBitsFromISR
- xEventGroupClearBits
- xEventGroupClearBitsFromISR
- xEventGroupGetBits
- xEventGroupGetBitsFromISR
- xEventGroupSync
- vEventGroupDelete

Watchdogs

Please follow the steps precisely in order to complete the objectives of the assignment.

1. Create a **producer** task that takes 1 temperature sensor value every 1ms.
    - After collecting 100 samples (after 100ms), compute the average.
    - Write average value every 100ms (avg. of 100 samples) to a **sensor queue**.
    - Use medium priority for this task (see `util/rtos.hpp`)
    - See `examples/Temperature` for an example on how to use the temperature sensor driver.

2. Create a **consumer** task that pulls the data off the **sensor queue**.
    - Use infinite timeout value during queue receive API
    - Open a file, **sensor.txt**, and append the data to an output file on the SD card.
    - Save the data in this format: **`printf("%i, %i\n", time, temperature);`**
    - Note that if you write and close a file every 100ms, it may be very inefficient, so try to come up with a better method such that the file is only written once a second or so ...
    - Use medium priority for this task.

3. At the end of the loop of each task, set a bit using FreeRTOS event group API.
    - At the end of each loop of the tasks, set a bit using the **`xEventGroupSetBits()`**
    - Task 1 should set bit 1, Task 2 should set bit 2 etc ...

4. Create a **watchdog** task that monitors the operation of the two tasks.
    - Use high priority for this task.
    - Wait 1 second for all of the task bits to be set. If there are two tasks, wait for bit1, and bit2 etc...
    - If you fail to detect the bits are set, that means that the other tasks did not reach the end of the loop.
    - In the event that a task failed to set its event group bit, append to a file, **stuck.txt**, with the information about which task is "stuck"
    - Open the file, append the data, and close the **stuck.txt** file to flush out the data immediately.

5. Create a terminal command to "suspend" and "resume" a task by name.
    - "task suspend task1" should suspend a task named "task1"

- "task resume task2" should suspend a task named "task2"
- Use the `examples/Commandline` project as an

6. Run the system, and under normal operation, you will see a file being saved with sensor data values.
   - Plot the file data in Excel to demonstrate the variation in the data.
   - Make sure the data has some variation. This can be done by touching the top of the temperature sensor.

7. Suspend the producer task. The watchdog task should display a message and save relevant info to the SD card.
8. Let the system run for a while, and note down the CPU usage in your text file.

What you created is a "software watchdog". This means that in an event when a loop is stuck, or a task is frozen, you can save relevant information such that you can debug at a later time.

> You may use any built in libraries for this lab assignment.

> For File I/O refer examples/FileIO project
> And the API documentation here: http://elm-chan.org/fsw/ff/00index_e.html

> **Extra Credit**
> Every sixty seconds, save the CPU usage info to a file named cpu.txt. See command "info" as a reference.

?

?

?