-Aman Kanwar

Semaphores implementation:

From FreeRTOS reference manual it is mentioned:

Binary Semaphores – A binary semaphore used for synchronization does not need to be ‘given’ back after it has been successfully ‘taken’ (obtained).
Task synchronization is
implemented by having one task or interrupt ‘give’ the semaphore, and another task ‘take’ the semaphore (see the xSemaphoreGiveFromISR() documentation).

and

Mutexes – The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to ‘inherit’ the priority of the task that is attempting to ‘take’ the same mutex. The inherited priority will be ‘disinherited’ when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

Here with mutex, there is an implementation to avoid the Priority Inversion issue!

Example

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    /* Attempt to create a semaphore.
    NOTE: New designs should use the xSemaphoreCreateBinary() function, not the
    vSemaphoreCreateBinary() macro. */
    vSemaphoreCreateBinary( xSemaphore );

    if( xSemaphore == NULL )
    {
        /* There was insufficient FreeRTOS heap available for the semaphore to
        be created successfully. */
    }
    else
    {
        /* The semaphore can now be used. Its handle is stored in the xSemaphore
        variable. */
    }
}
```

"Direct to task notifications normally provide a lighter weight and faster alternative to binary semaphores."

Binary semaphores and mutexes are very similar, but do have some subtle differences. Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between an interrupt and a task), and mutexes the better choice for implementing simple mutual exclusion.

From FreeRTOS

-Aman Kanwar

7.3 Mutexes (and Binary Semaphores)

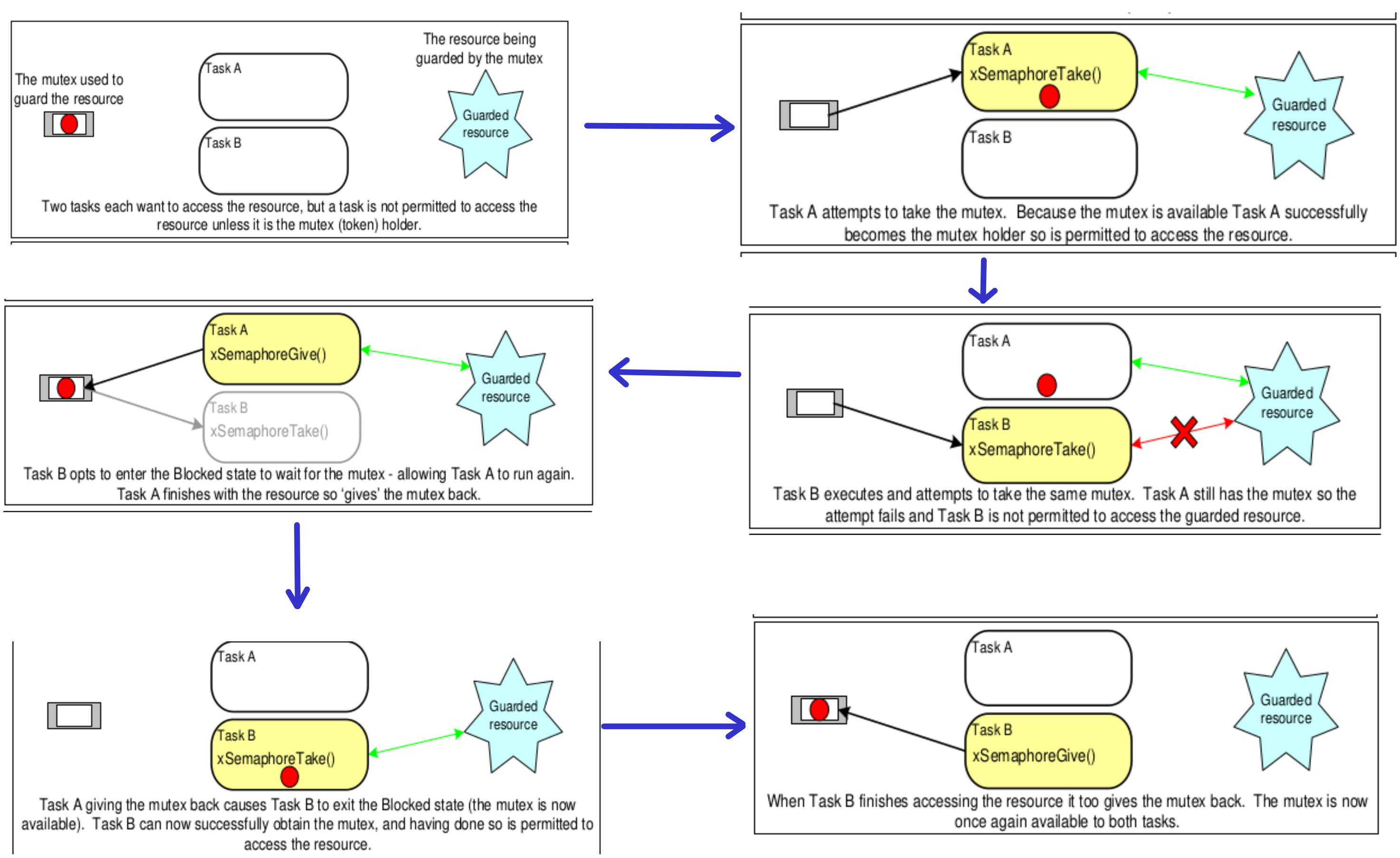
A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks. The word MUTEX originates from 'MUTual EXclusion'. configUSE_MUTEXES must be set to 1 in FreeRTOSConfig.h for mutexes to be available.

When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared. For a task to access the resource legitimately, it must first successfully 'take' the token (be the token holder). When the token holder has finished with the resource, it must 'give' the token back. Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource. A task is not permitted to access the shared resource unless it holds the token. This mechanism is shown in Figure 63.

Even though mutexes and binary semaphores share many characteristics, the scenario shown in Figure 63 (where a mutex is used for mutual exclusion) is completely different to that shown in Figure 53 (where a binary semaphore is used for synchronization). The primary difference is what happens to the semaphore after it has been obtained:

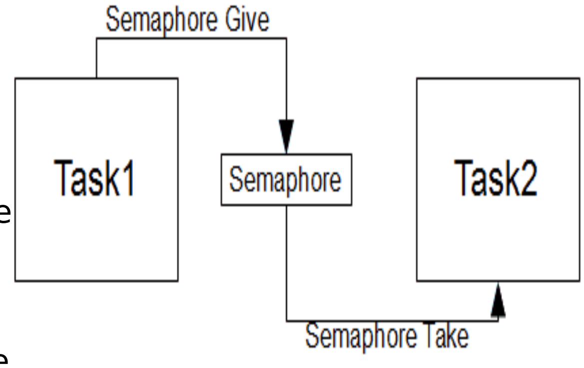
- A semaphore that is used for mutual exclusion must always be returned.
- A semaphore that is used for synchronization is normally discarded and not returned.

Problem statement:



Above problem statment can be implemented using the following scenario

Create two tasks:



Producer Task PRIORITY 1

This task takes the semaphore on its first run.
Once the semaphore is acquired. It should block for 8 ms and after 8 ms it should release the semaphore and delete itself after setting led 15 high

Consumer Task. PRIORITY 2

This task should try to take the semaphore and once semaphore is taken successfully, it should make LED 13 high and delete itself.

The priority of Producer task should be lower than the consumer task.

We can create the following task model to create.

```
/* Declare a variable of type SemaphoreHandle_t.  This is used to reference the
semaphore that is used to synchronize a task with an interrupt. */
SemaphoreHandle_t xBinarySemaphore;

void producer(void * ptr)
{
    /*
     * Attempting to take the semaphore as soon as the task started
     * */
    xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
    const TickType_t xDelay8ms = pdMS_TO_TICKS(8);
    while(1) {
        vTaskDelay(xDelay8ms);
        xSemaphoreGive(xBinarySemaphore);
        HAL_GPIO_WritePin(GPIOD, GPIO_PIN_15, GPIO_PIN_SET);
        vTaskDelete(NULL);
    }
}
void consumer(void * ptr)
{
    const TickType_t xDelay1ms = pdMS_TO_TICKS(1);

    while(1)
    {
        vTaskDelay(xDelay1ms);
        /* Use the semaphore to wait for the event.  The semaphore was created
        before the scheduler was started so before this task ran for the first
        time.  The task blocks indefinitely meaning this function call will only
        return once the semaphore has been successfully obtained - so there is
        no need to check the returned value. */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );
        HAL_GPIO_WritePin(GPIOD, GPIO_PIN_13, GPIO_PIN_SET);
        vTaskDelete(NULL);
    }
}
```

```

  xBinarySemaphore = xSemaphoreCreateMutex();
  if (!xBinarySemaphore) {
      while(1);
  }

   xTaskCreate(producer, "ProducerTask2", 200, NULL, 1, NULL);
   xTaskCreate(consumer, "ConsumerTask1", 200, NULL, 2, NULL);

   vTaskStartScheduler();
```

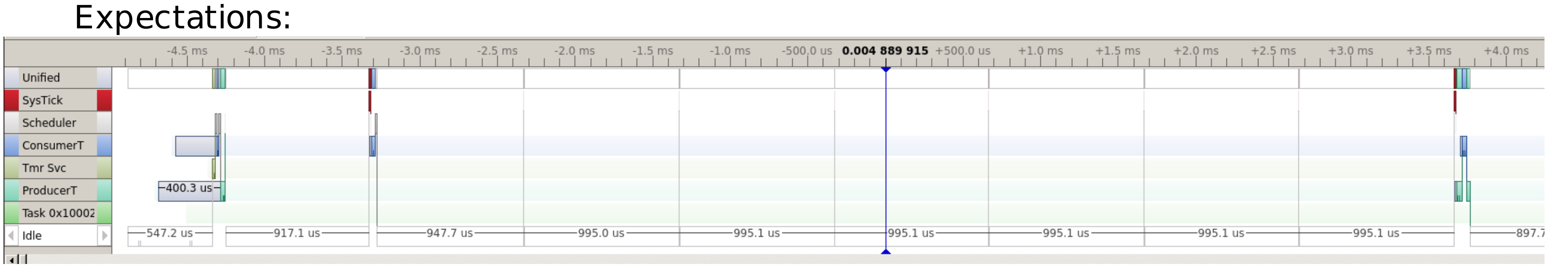
```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

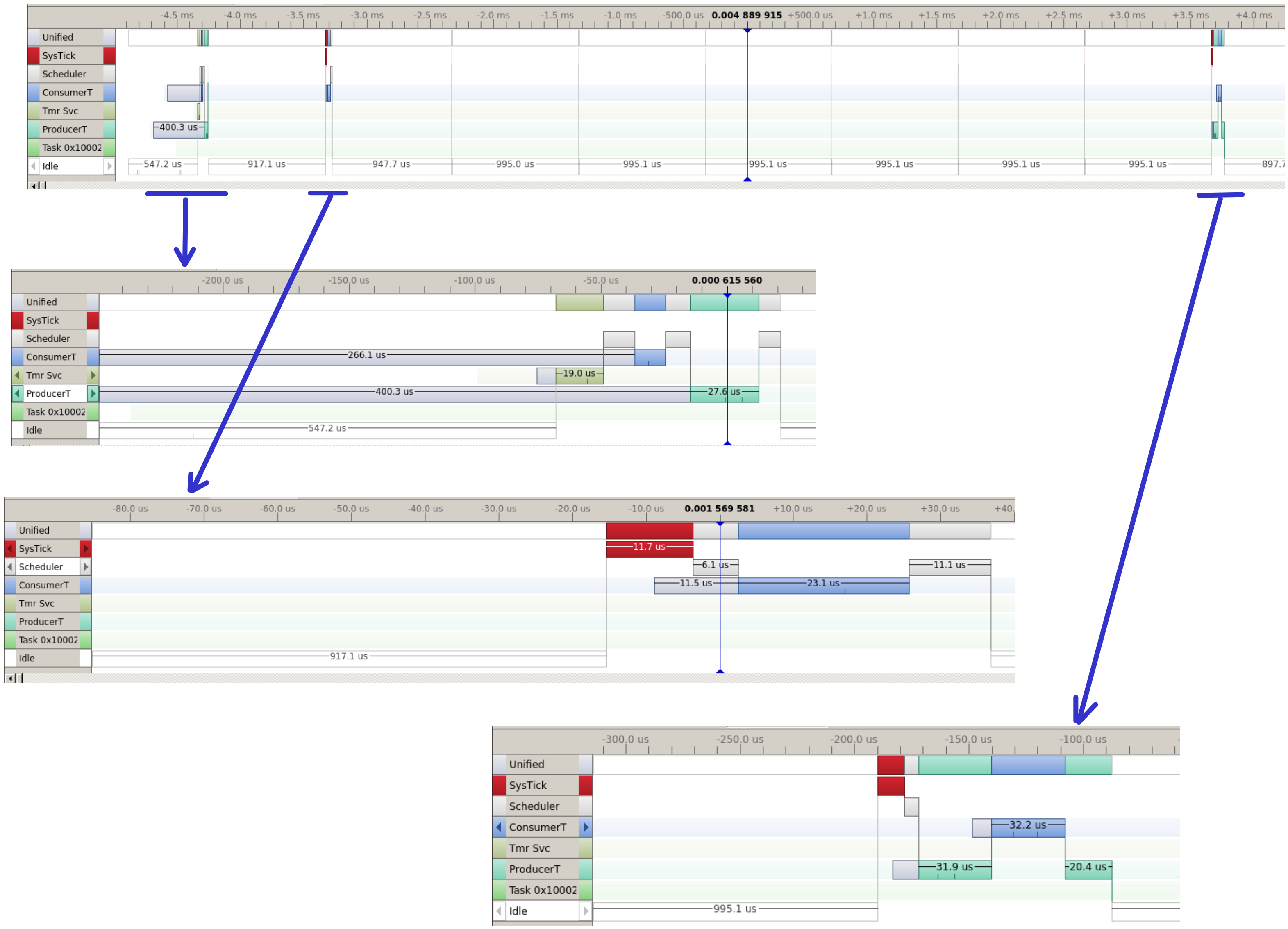
Listing 90. The xSemaphoreTake() API function prototype

Table 34. xSemaphoreTake() parameters and return value

Parameter Name/ Returned Value	Description
xSemaphore	The semaphore being 'taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t. It must be explicitly created before it can be used.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for the semaphore if it is not already available. If xTicksToWait is zero, then xSemaphoreTake() will return immediately if the semaphore is not available. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds to a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without a timeout) if INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Expectations:





As cleary seen from the dumped trace, our two task are created of different priorties. Here the semaphore is first taken by the task of lower priority and later the task of higher priority tries to take the semaphore and gets blocked.

Later the producer task blocks itself for 8 ms and after 8 ms, the producer task gives the semaphore and delete itself.

Once the semaphore is available consumer task gets unblocked immediately and it deletes itself. Once the consumer task deletes itself, the producer task runs and deletes itself too.