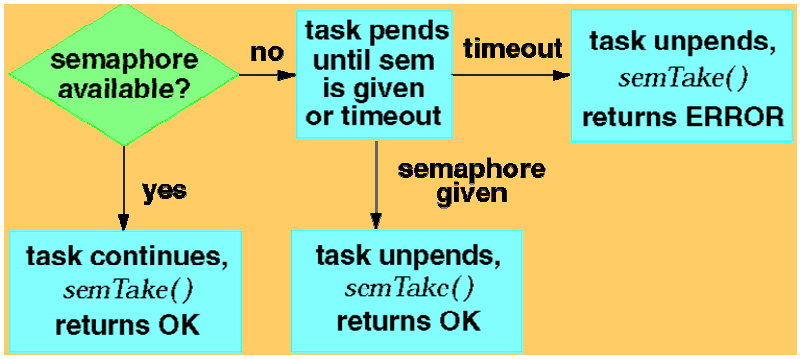
ARM PROGRAMMING

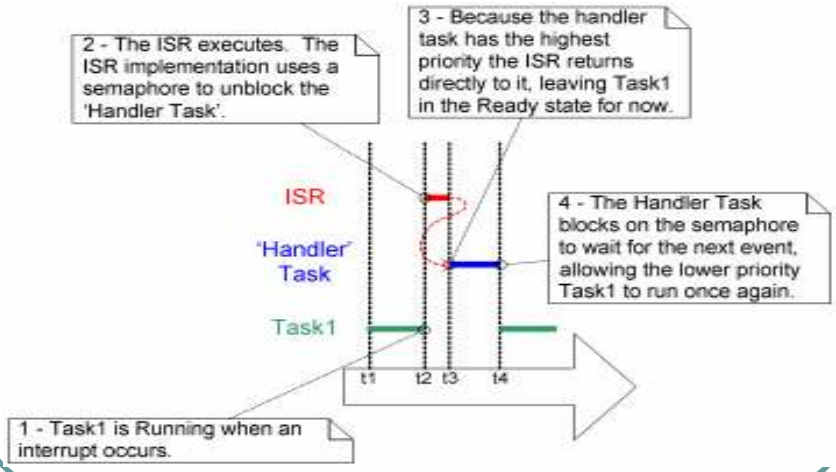
Bùi Quốc Bảo

- FreeRTOS cho phép sử dụng ngắt.
- Tất cả các API có tên kết thúc bằng “FromISR” hoặc “FROM_ISR” thì được sử dụng trong ISR.

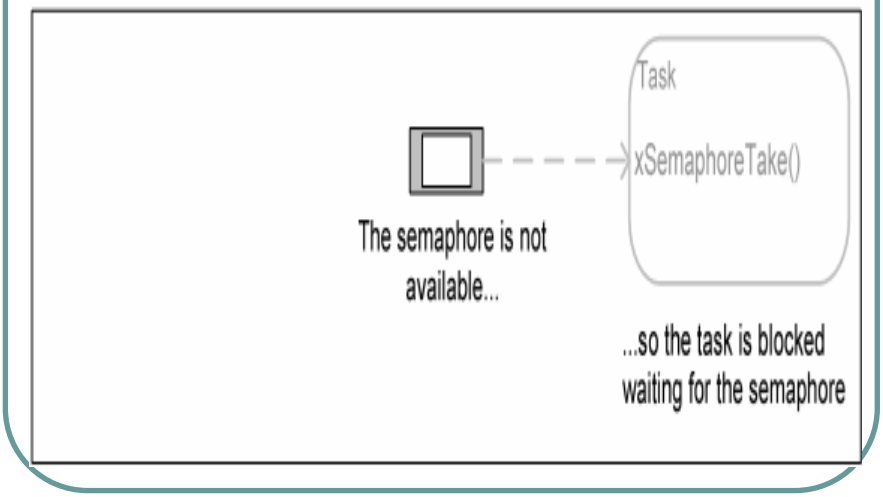
Binary semaphore use for synchronization



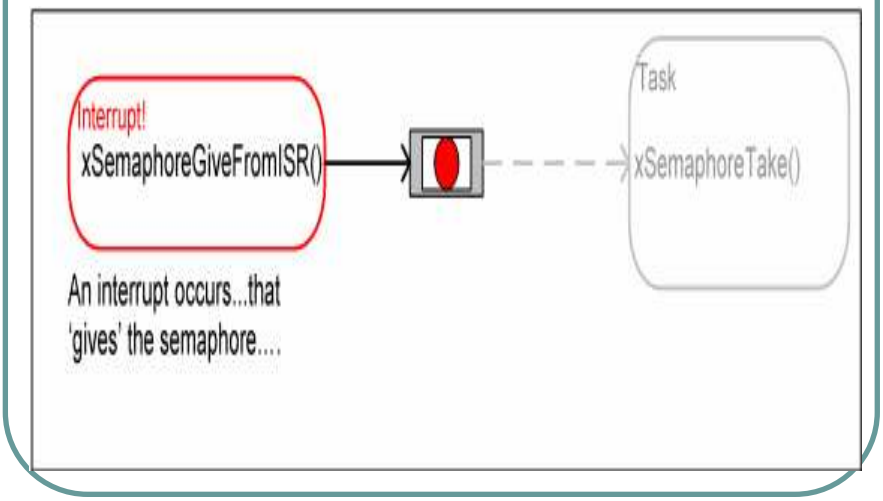
Binary semaphore use for synchronization



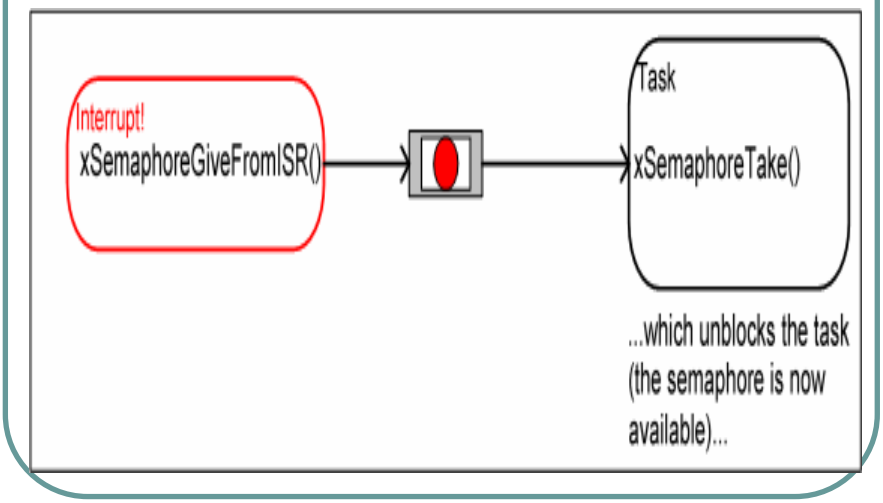
Binary semaphore use for synchronization



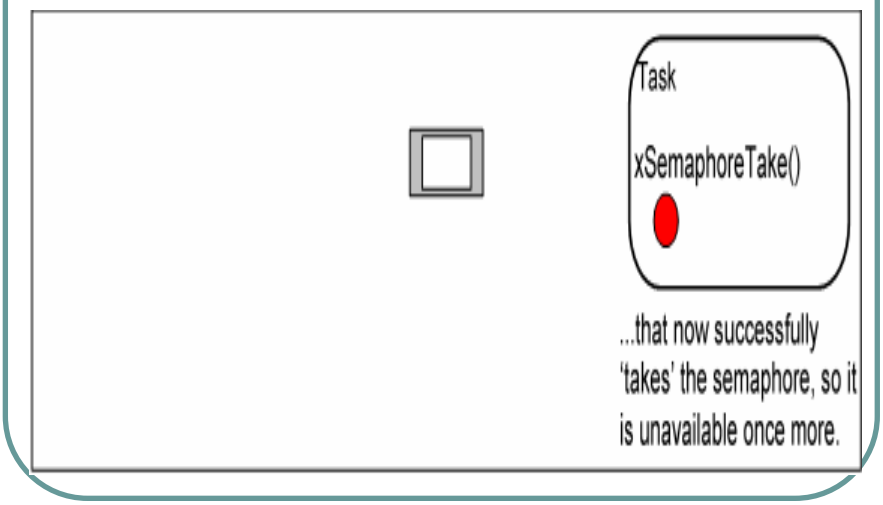
Binary semaphore use for synchronization



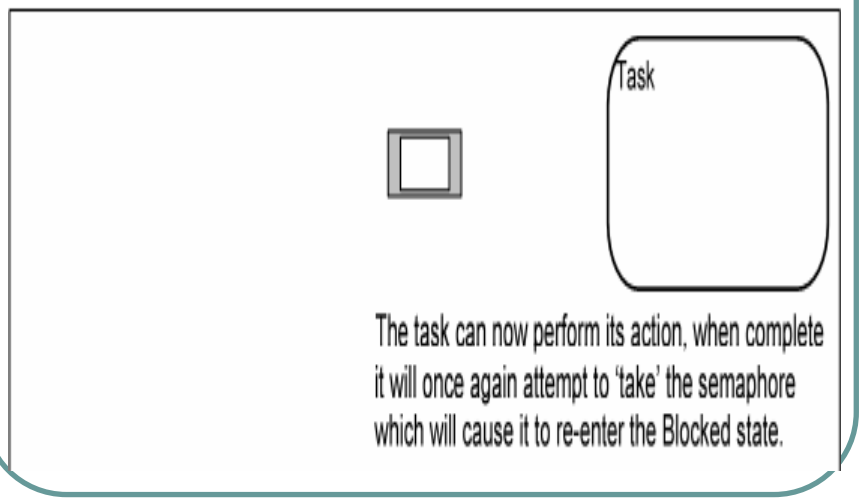
Binary semaphore use for synchronization



Binary semaphore use for synchronization



Binary semaphore use for synchronization



Create a binary semaphore vSemaphoreCreateBinary()

- void vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore);
  - xSemaphore The semaphore being created.

Get the semaphore xSemaphoreTake

- portBASE_TYPE xSemaphoreTake(xSemaphoreHandle xSemaphore, portTickType xTicksToWait);
  - xSemaphore: The semaphore being 'taken'.
  - xTicksToWait:
    - 0: the function will return immediately if the semaphore is not available
    - portMAX_DELAY: the task will wait indefinitely if INCLUDE_vTaskSuspend is set to 1
  - Return value:
    - pdPASS: success
    - pdFALSE: Fail

Give semaphore from ISR xSemaphoreGiveFromISR

- portBASE_TYPE xSemaphoreGiveFromISR(xSemaphoreHandle xSemaphore, portBASE_TYPE *pxHigherPriorityTaskWoken);
  - pxHigherPriorityTaskWoken
    - pdTRUE: a higher priority task will pre-emp current task
    - pdFALSE: there are no higher priority task waiting for the semaphore

Nếu pxHigherPriorityTaskWoken là pdTRUE, ISR phải thực hiện "contextSwitch" trước khi thoát

Give semaphore from ISR xSemaphoreGiveFromISR

```
static void __interrupt __far vExampleInterruptHandler( void )
{
static portBASE_TYPE xHigherPriorityTaskWoken;

        xHigherPriorityTaskWoken = pdFALSE;

   /* 'Give' the semaphore to unblock the task. */
   xSemaphoreGiveFromISR( xBinarySemaphore,
   &xHigherPriorityTaskWoken );

   if( xHigherPriorityTaskWoken == pdTRUE )
   {
        portSWITCH_CONTEXT();
   }
}
```

Give semaphore from ISR xSemaphoreGiveFromISR

```
static void vPeriodicTask( void *pvParameters )
{
   for( ;; )
   {
      /* This task is just used to 'simulate' an interrupt by generating a
      software interrupt every 500ms. */

      vTaskDelay( 500 / portTICK_RATE_MS );
    vPrintString( "Periodic task - About to generate an interrupt.\r\n" );

      __asm{ int 0x82 } /* This line generates the interrupt. */

      vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n" );
   }
}
```

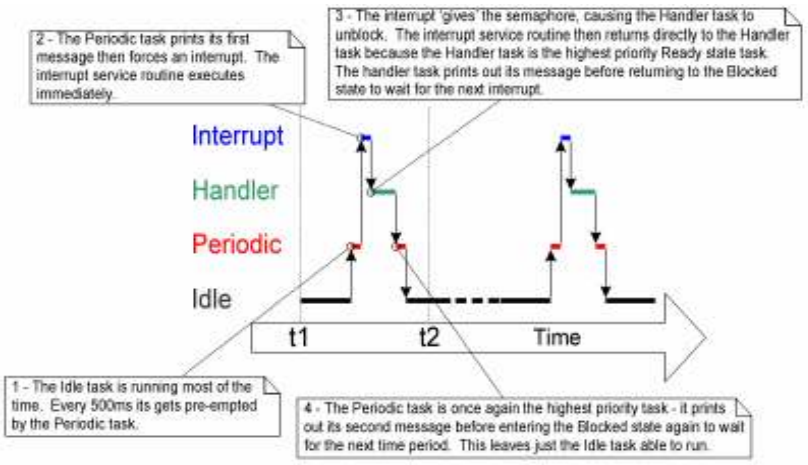
Give semaphore from ISR xSemaphoreGiveFromISR

```
static void vHandlerTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented
    within an infinite loop. */
    for( ;; )
    {
        xSemaphoreTake( xBinarySemaphore,
                                    portMAX_DELAY );
        vPrintString( "Handler task - Processing
                            event.\r\n" );
    }
}
```

Give semaphore from ISR xSemaphoreGiveFromISR

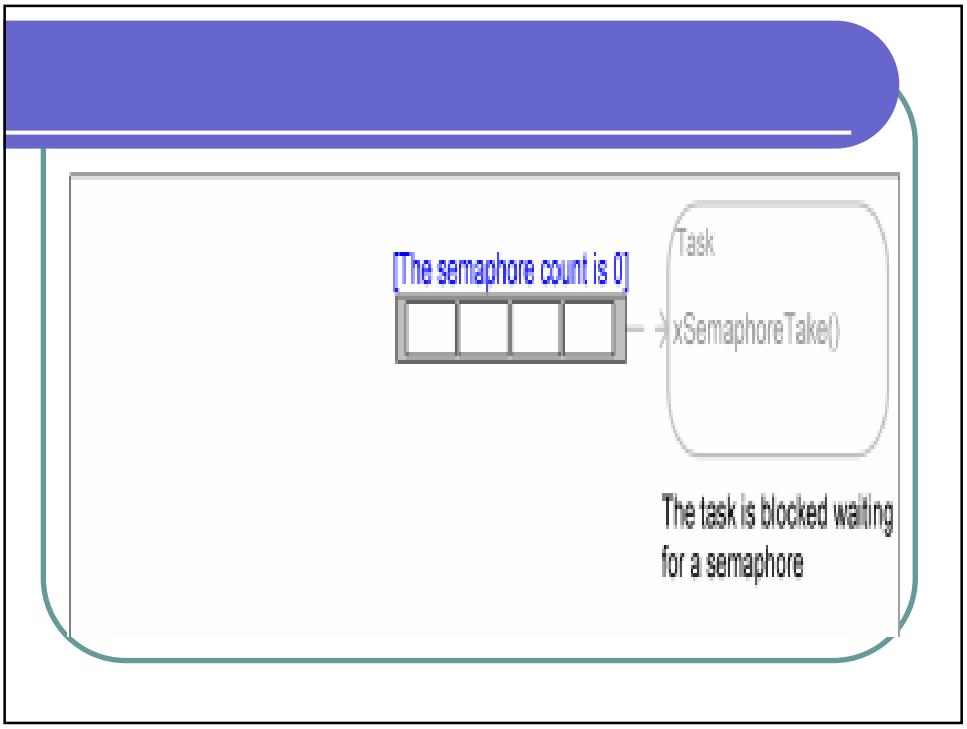
```
int main( void )
{
vSemaphoreCreateBinary( xBinarySemaphore );
 _dos_setvect( 0x82, vExampleInterruptHandler );

   /* Check the semaphore was created successfully. */
   if( xBinarySemaphore != NULL )
   {
 xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );
 xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, NULL );
 vTaskStartScheduler();
   }
   for( ;; );
}
```

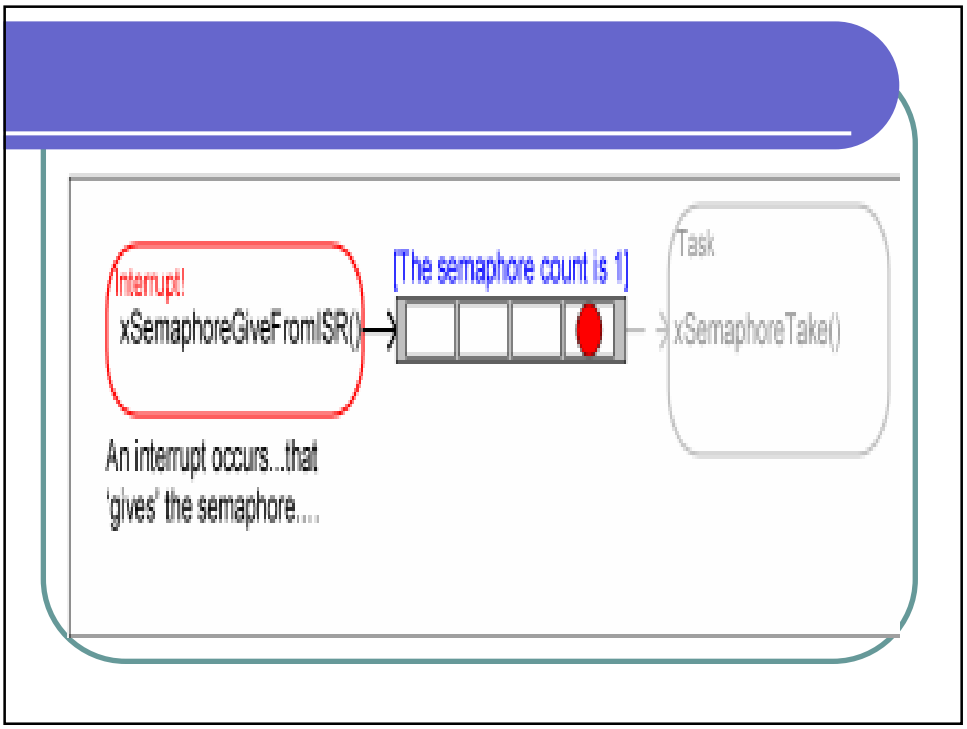
Give semaphore from ISR xSemaphoreGiveFromISR

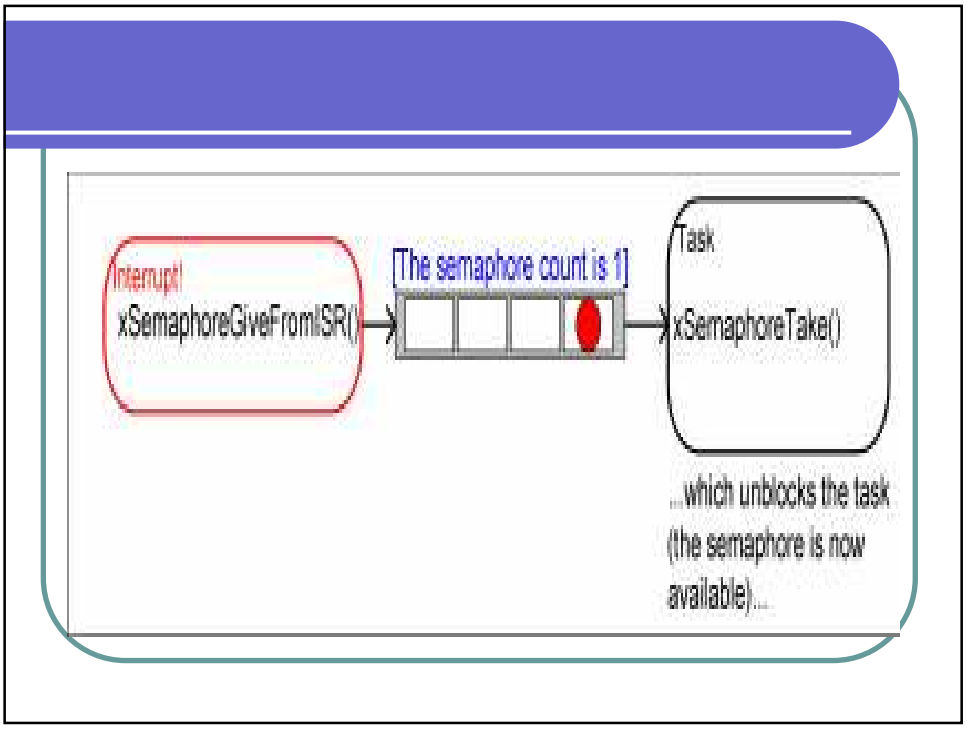


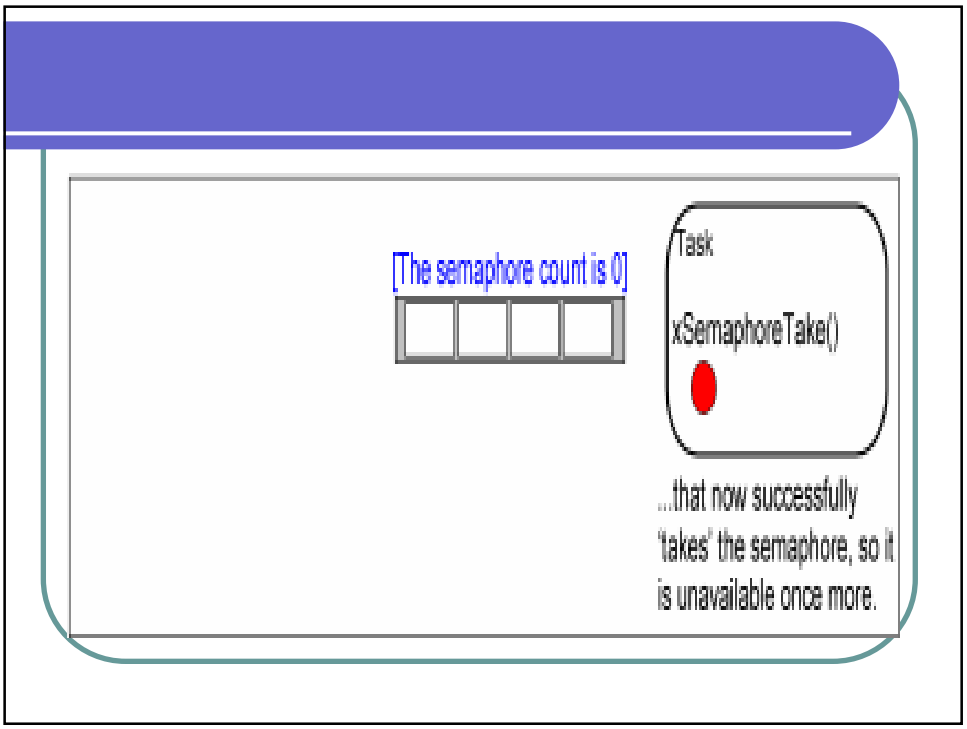
Counting semaphore

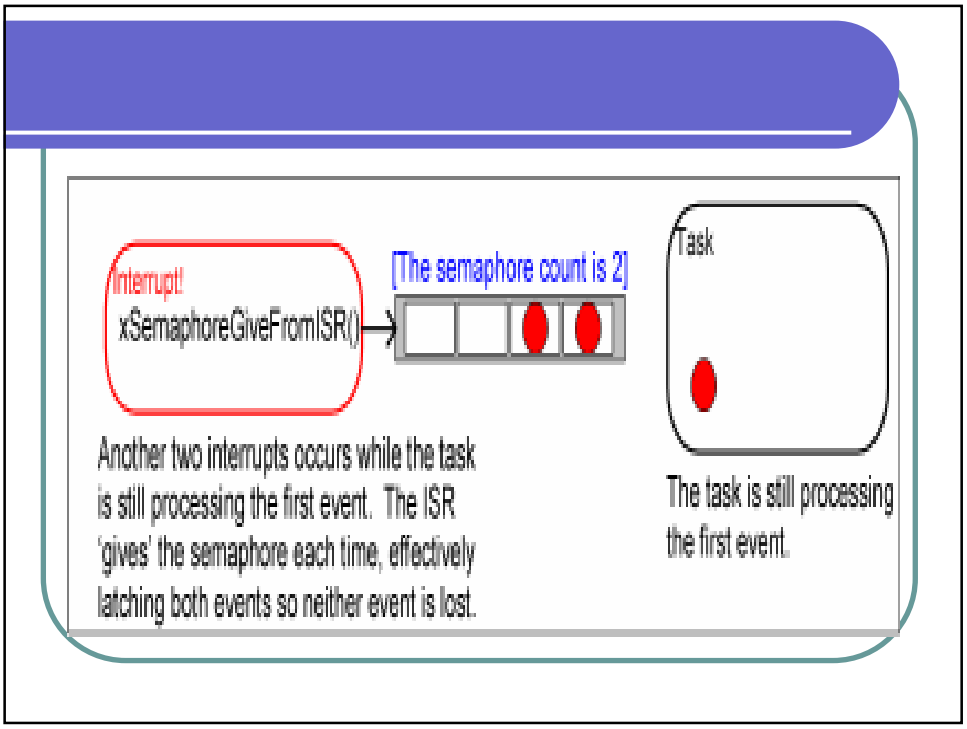
- Nếu trong quá trình task đang thực thi, ISR xảy ra và gọi hàm xSemaphoreGiveFromISR nhiều lần, chỉ có 1 lần tác vụ được thực thi.
- Counting semaphore có thể được coi như là 1 queue các semaphore.
- Số lượng các phần tử trong queue là độ rộng của semaphore.

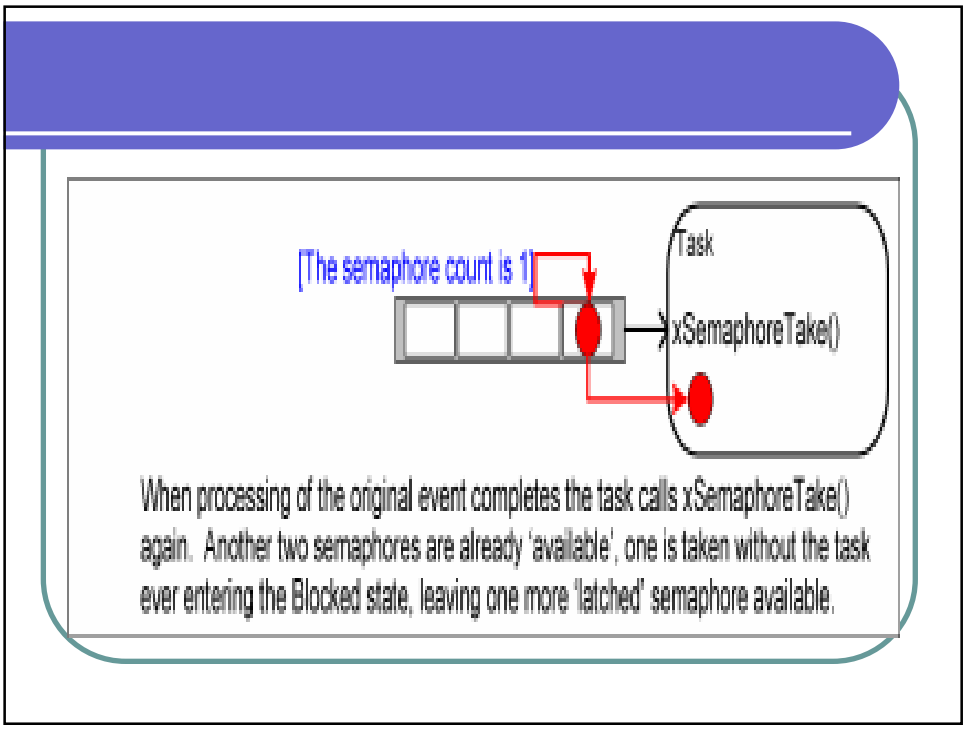

[The semaphore count is 0]
Task
xSemaphoreTake()
The task is blocked waiting for a semaphore


Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....


Interrupt
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...


[The semaphore count is 0]
Task
xSemaphoreTake()
...that now successfully 'takes' the semaphore, so it is unavailable once more.


Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 2]
Task
Another two interrupts occurs while the task is still processing the first event. The ISR 'gives' the semaphore each time, effectively latching both events so neither event is lost.
The task is still processing the first event.


[The semaphore count is 1]
Task
xSemaphoreTake()
When processing of the original event completes the task calls xSemaphoreTake() again. Another two semaphores are already 'available', one is taken without the task ever entering the Blocked state, leaving one more 'latched' semaphore available.

Counting semaphore

- Counting semaphore dùng để:
  - Lưu số lần xảy ra sự kiện
  - Quản lý tài nguyên

Create counting semaphore xSemaphoreCreateCounting

- xSemaphoreHandle xSemaphoreCreateCounting(unsigned portBASE_TYPE uxMaxCount, unsigned portBASE_TYPE uxInitialCount);
  - uxMaxCount: The maximum value the semaphore will count to
  - uxInitialCount: The initial count value
  - Returned value:
    - NULL: fail
    - Non-NUL: the semaphore handle

Using queue in ISR

- Các hàm
  - xQueueSendToFrontFromISR()
  - xQueueSendToBackFromISR()
  - xQueueReceiveFromISR()

  được sử dụng để tương tác với queue trong ISR

Send message to back of queue xQueueSendToBackFromISR

- portBASE_TYPE xQueueSendFromISR(
  xQueueHandle xQueue,
  void *pvItemToQueue
  portBASE_TYPE *pxHigherPriorityTaskWoken);
- portBASE_TYPE xQueueSendFromISR(
  xQueueHandle xQueue,
  void *pvItemToQueue
  portBASE_TYPE *pxHigherPriorityTaskWoken);

- xQueue: The handle of the queue
- pvItemToQueue: A pointer to the data
- pxHigherPriorityTaskWoken:
  - Will be true if there is another task with higher priority waiting for the queue
- Returned value:
  1. pdPASS
  2. errQUEUE_FULL

```
static void vIntegerGenerator( void *pvParameters ) {
portTickType xLastExecutionTime;
unsigned portLONG ulValueToSend = 0;
int i;
xLastExecutionTime = xTaskGetTickCount();
   for( ;; )
   {
vTaskDelayUntil( &xLastExecutionTime, 200 / portTICK_RATE_MS );
for( i = 0; i < 5; i++ ) {
    xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
        ulValueToSend++; }
    vPrintString( "Generator task - About to generate an interrupt.\r\n"
    );
      __asm{ int 0x82 } /* This line generates the interrupt. */
   vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );
   } }
```

```
static void __interrupt __far        vExampleInterruptHandler( void )
{
static portBASE_TYPE                 xHigherPriorityTaskWoken;
static unsigned long ulReceivedNumber;
static const char *pcStrings[] =
{
   "String 0\r\n",
   "String 1\r\n",
   "String 2\r\n",
   "String 3\r\n"
};
    xHigherPriorityTaskWoken = pdFALSE;
```

```
while( xQueueReceiveFromISR( xIntegerQueue,
                        &ulReceivedNumber,
                        &xHigherPriorityTaskWoken ) !=
    errQUEUE_EMPTY )
  {
        ulReceivedNumber &= 0x03;
        xQueueSendToBackFromISR( xStringQueue,
                        &pcStrings[ ulReceivedNumber ],
                        &xHigherPriorityTaskWoken );
  }
if( xHigherPriorityTaskWoken == pdTRUE )
  {
        portSWITCH_CONTEXT();
  }
}
```

```
static void vStringPrinter( void *pvParameters )
{
char *pcString;

   for( ;; )
   {
      /* Block on the queue to wait for data to arrive. */
      xQueueReceive( xStringQueue, &pcString,
    portMAX_DELAY );

      /* Print out the string received. */
      vPrintString( pcString );
   }
}
```

```
int main( void )
{
    xIntegerQueue = xQueueCreate( 10, sizeof( unsigned long ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );
    _dos_setvect( 0x82, vExampleInterruptHandler );
    xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );
    xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );

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

Interrupt nesting

- configKERNEL_INTERRUPT_PRIORITY:
  - Sets the interrupt priority used by the tick interrupt.
- configMAX_SYSCALL_INTERRUPT_PRIORITY:
  - Sets the highest interrupt priority from which interrupt safe FreeRTOS API functions can be called