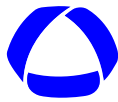


Licenciatura en Ciencias de la Computación

Sistemas Embebidos

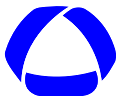
Unidad 3

Sistemas Operativos para Sistemas Embebidos

RTOS (Real Time Operating System)

- Hay **deadlines** (tiempos límite) para todas o algunas tareas.
- **Soft** RTOS: Si no se cumplen los deadlines, la **performance se degrada**, pero el sistema no se vuelve inutil o provoca una falla completa del sistema.
    - Ejemplos:
        - Sistema que debe medir variables ambientales a intervalos regulares de tiempo con timestamps.
        - Sistema que debe monitorear eventos, agregando timestamps.
- **Hard** RTOS: Si no se cumplen los deadlines, el sistema **falla por completo**.
    - Ejemplos:
        - Sistemas de seguridad: airbag, sensores de gases peligrosos, etc.

UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO


FACULTAD
DE INGENIERÍA

Licenciatura en Ciencias de la Computación

Sistemas operativos para sistemas embebidos

Embedded Linux	31%
FreeRTOS	27%
In-house/custom	16%
Debian (Linux)	15%
Ubuntu	14%
Android	14%
Microsoft (Windows 10)	12%
Texas Instruments RTOS	9%
Keil (RTX)	6%
Other	6%
Micrium (uC/OS-III)	5%
Red Hat (IX Lunix)	5%
Texas Instruments (DSP/BIOS)	5%
Wind River (VxWorks)	5%
Green Hills (INTEGRITY)	5%
QNX (QNX)	5%
Express Logic (ThreadX)	4%
Angstrom (Linux)	4%
Wittenstein HIS(OpenRTOS/SAFERTOS	4%
Micrium (uC/OS-II)	4%
Freescale MQX	3%
AnalogDevices (VDK)	3%
Segger (embOS)	3%
Wind River (Linux)	3%
Microsoft (Windows 7 Compact or earlier)	3%

Regional Breakout

APAC users will use FreeRTOS and Android much more than other regions and use Embedded Linux much less. **EMEA** will use Android less than other regions.

Most Used	World	Americas	EMEA	APAC
Embedded Linux	31%	32%	31%	26%
FreeRTOS	27%	25%	24%	37%
Android	14%	12%	10%	26%

Only Operating Systems with 3% more are shown

Figura obtenida de EETimes, “2019 Embedded Markets Study”.
https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

FreeRTOS

- **Hard RTOS** para microcontroladores y procesadores **pequeños**.
  - SO más utilizado para sistemas embebidos de bajo poder de procesamiento.
- Desarrollado por **Richard Barry**, luego mantenido por Real Time Engineers ltd., luego **Amazon Web Services**.
- Licencia **open source MIT license**.
  - Licencia libre permisiva.
  - Permite reutilizar software dentro de software propietario.
- Objetivos:
  - Velocidad, pequeño tamaño, simplicidad, compacto, confiabilidad.
- Plataformas: ARM, x86, Atmel AVR, TI MSP432, Xtensa, Microchip PIC etc.
- Foro de soporte muy activo y varios libros accesibles sin costo.

Soportado por: https://www.freertos.org/RTOS_ports.html

FreeRTOS

- Implementa:
  - Scheduler (tareas).
  - Exclusión mutua, semáforos, colas.
  - Temporizadores.
  - Modos bajo consumo de energía.
  - Manejo de interrupciones.
  - Notificaciones entre tareas.
  - Integración con servicios en la nube de AWS (AWS IoT services).
- No implementa:
  - Controladores de dispositivos (drivers).
  - Administración de memoria.
  - Cuentas de usuario.

FreeRTOS

- Características:
  - El kernel requiere entre **6KB** y **12KB** de memoria (El ATmega 328p posee 32 KB).
  - Muy escalable.
- Código fuente **lenguaje C**.
  - Puede ser compilado en más de 20 compiladores.
  - Puede ser compilado para más de 30 arquitecturas.
- **SafeRTOS**: Versión certificada para aplicaciones críticas como medicina, automóviles e industriales.

FreeRTOS

- **Tareas** (análogas a los procesos en un sistema operativo):
  - Se crean en tiempo de ejecución.
  - Por cada tarea, FreeRTOS:
    - Crea un **TCB** (Task Control Block): Contiene información de la tarea (análogo al PCB).
      - Contiene el estado de la tarea, prioridad, etc.
    - Asigna **Memoria**
      - Se asigna memoria cada vez que se crea una tarea
      - Se libera memoria cada vez que la tarea finaliza.

UNCUYO UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

Tareas - Estados

- **Running**: una tarea por núcleo.
- **Ready**: Esperando que el scheduler la pase a estado running.
- **Blocked**: No puede pasar a estado running. Pasa a estado Ready por:
  - Evento externo.
  - Timeout.
- **Suspended**: No puede pasar a estado running. Solo entra o sale con vTaskSuspend() and xTaskResume()

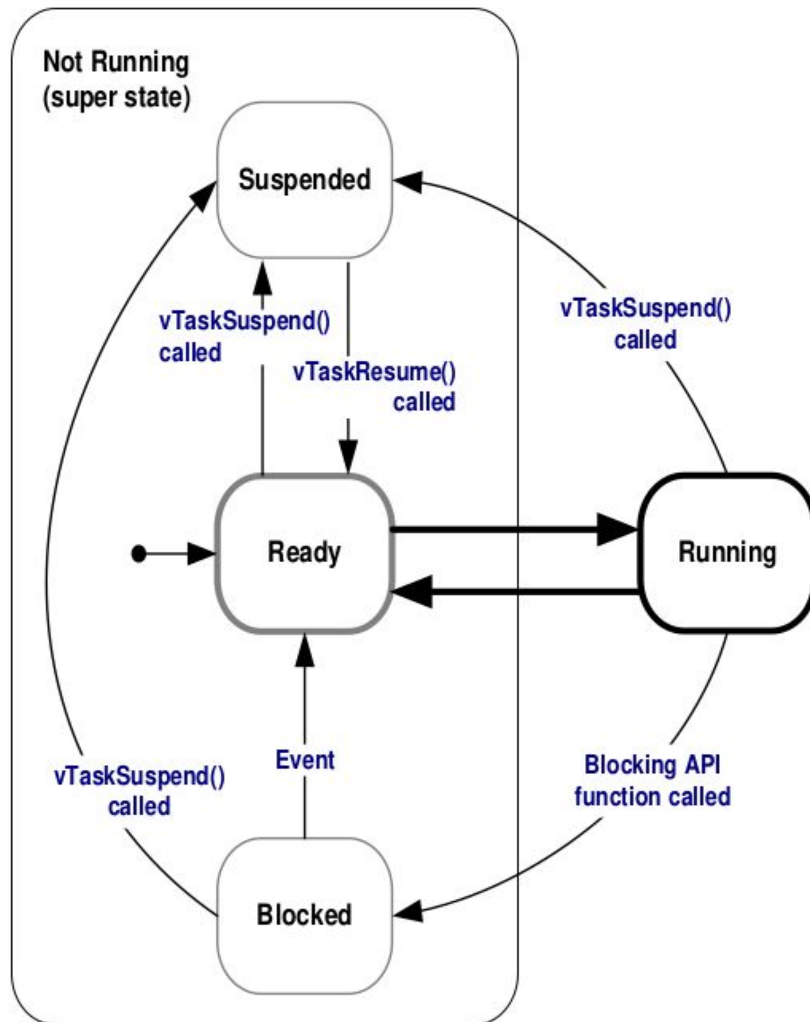


Figura obtenida de Richard Barry. "Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide". Página 67

FreeRTOS

Scheduler

- **Con prioridades** y **Preemptive** (puede interrumpirse la ejecución de una tarea para pasar a ejecutar otra de mayor prioridad).
- **Round-robin**, **time sliced** (intervalos de tiempo fijo).
- Prioridades: Números desde 0 a un máximo. **Mayor valor, mayor prioridad**. Fijas.
  - Se ejecutan siempre tareas de mayor prioridad.
    - Tareas de mayor prioridad deben entrar en modo bloqueado o suspendido, de lo contrario, se ejecutarán todo el tiempo.
- Tarea **Idle task**: creada cuando el scheduler arranca. Tiene la más baja prioridad. La única función que ejecuta es liberar memoria asignada a tareas eliminadas.

FreeRTOS

- Se ejecuta sobre procesadores:
  - Simple core, simetric multicore: una sola instancia de FreeRTOS para todos los núcleos.
  - Asymmetric multicore: Cada core ejecuta su instancia de FreeRTOS.

Implementación de una tarea

```
void vATaskFunction( void *pvParameters )
    {
        for( ;; )
        {
            -- Task application code here. --
        }
        /*No se debe llamar a return*/
        vTaskDelete( NULL ); /*Finaliza la tarea*/
    }
```

FreeRTOS

Creación de una tarea

```
xTaskCreate(
    TaskBlink13,   //Nombre de la función que implementa la tarea.
                   //Variable tipo "TaskFunction_t" (vector).
    "Blink",       //Nombre entendible por humanos. No usado por freeRTOS
    128,           //tamaño del stack asignado a la tarea. Tipo: "uint16_t"
    pvParameters,  //Parámetros. Tipo "void *" (se pasan por referencia).
                   //Si no hay parámetros, se escribe NULL.
    2,             // Prioridad
    pxCreatedTask,//Manejador de tarea. Puede invocarse desde otra tarea
                   //para realizar operaciones como cambiar su prioridad
                   //o eliminar la tarea.
                   //Variable tipo "TaskHandle_t *" (se pasa por referencia).
                   //Puede escribirse NULL si la tarea no será modificada.
);
```

FreeRTOS

Creación de una tarea

- La función xTaskCreate() retorna:
  - **pdPASS** si la tarea fue creada exitosamente.
  - **pdFAIL** si hubo errores al crear la tarea (por ejemplo, si no hay suficiente RAM).
- Pueden crearse varias tareas desde una misma función.
- **Tick Interrupt**: Interrupción que se ejecuta para que el scheduler ejecute la siguiente tarea.

Scheduler FreeRTOS

Ejecución de dos tareas periódicas con **igual prioridad**.

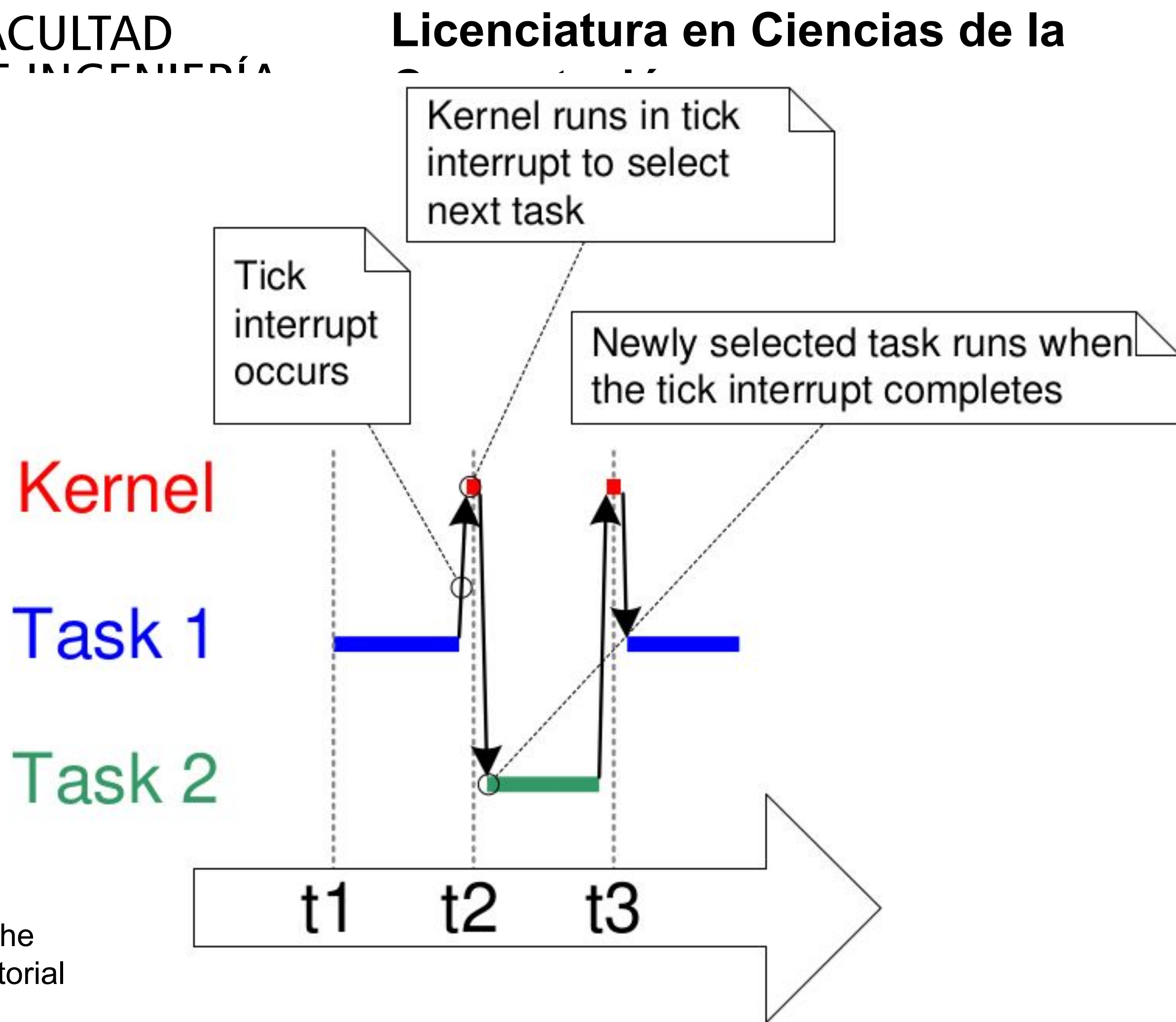


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 62

Scheduler FreeRTOS

Ejecución de dos tareas periódicas con distinta prioridad. La tarea 2 posee mayor prioridad.

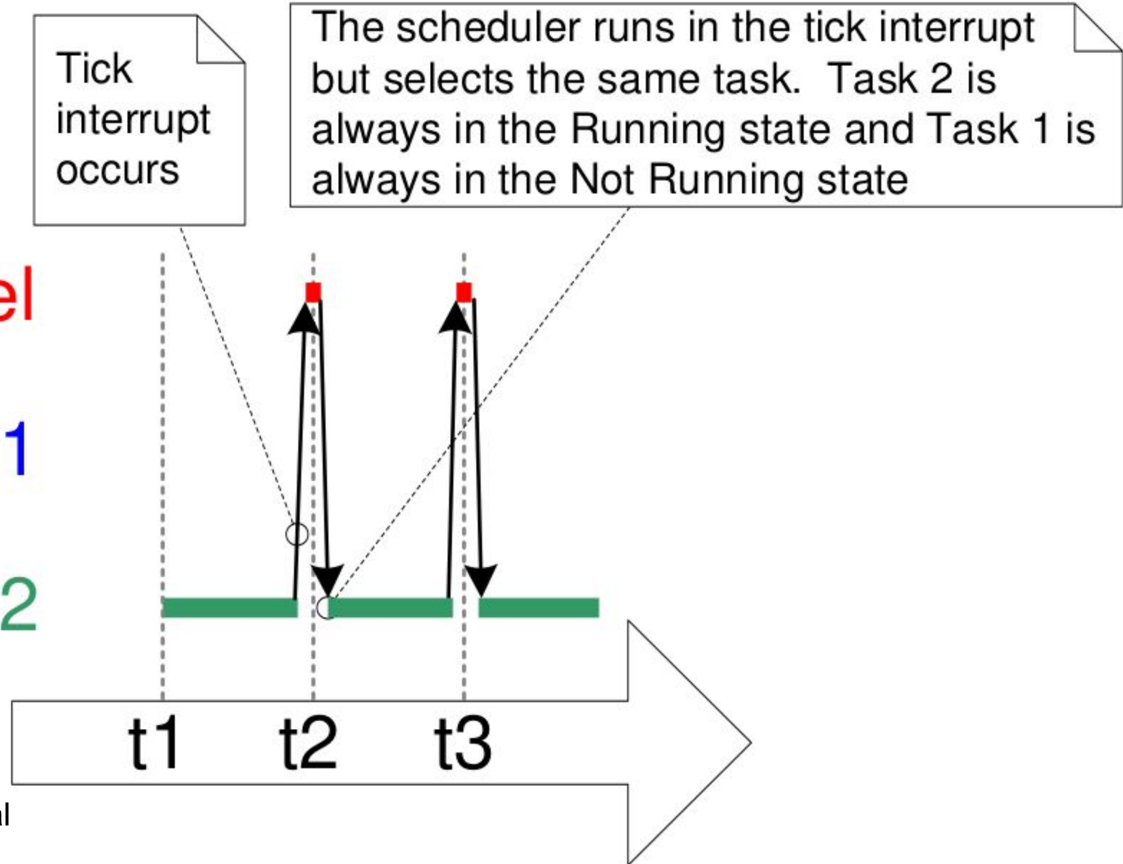


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 64

FreeRTOS

- Primitivas interesantes:
  - **vTaskDelay**(TickType_t   xTicksToDelay): Coloca la tarea en modo **Blocked** hasta que transcurra el tiempo indicado (en Arduino, 1 xTicksToDelay es 15 ms).
  - **pdMS_TO_TICKS**(xTimeInMs): Transforma ms a Ticks.
  - **vTaskDelayUntil**(TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement): Coloca la tarea en modo Blocked hasta un momento de tiempo indicado.
    Ejemplo:
    ```
    TickType_t xLastWakeTime;
    xLastWakeTime = xTaskGetTickCount();
    vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );
    ```
  - void **vTaskDelete**(TaskHandle_t   pxTaskToDelete): Elimina una tarea. Si se ejecuta dentro de la tarea a eliminar, pxTaskToDelete puede valer NULL.

Scheduler FreeRTOS

Ejecución de dos tareas manejadas por eventos. La tarea 2 posee mayor prioridad.

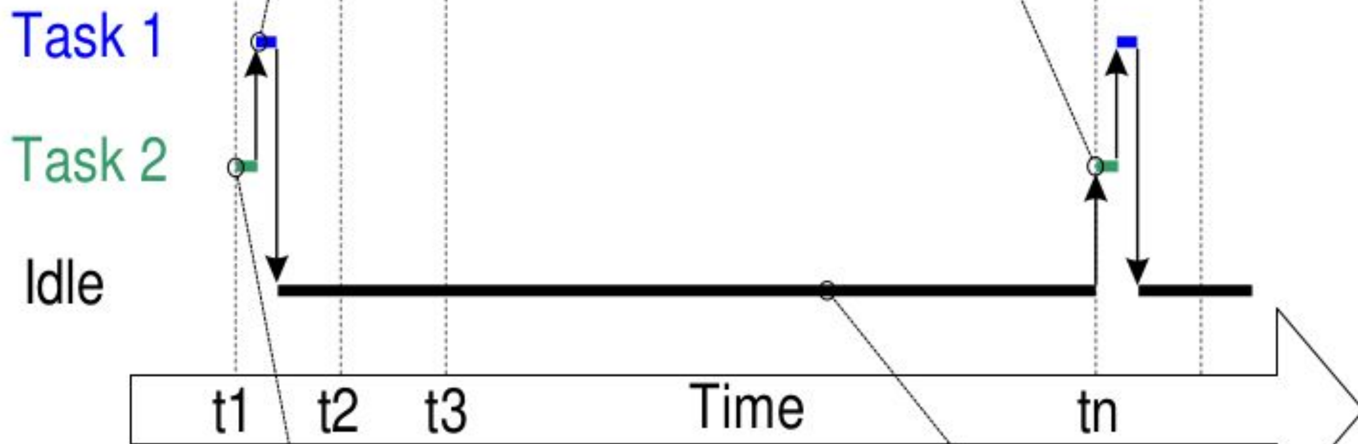
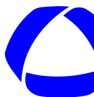


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 70

UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO



Scheduler FreeRTOS

Ejecución de dos tareas periódicas y una tarea manejada por eventos de mayor prioridad.

4 - At time t5 the tick interrupt finds that the Periodic task block period has expired so moved the Periodic task into the Ready state. The Periodic task is the highest priority task so immediately then enters the Running state where it prints out its string exactly once before calling vTaskDelayUntil() to return to the Blocked state.

1 - Continuous task 1 runs for a complete tick period (time slice between times t1 and t2) - during which time it could print out its string many times.

5 - The Periodic task entering the Blocked state means the scheduler has again to choose a task to enter the Running state - in this case Continuous 1 is chosen and it runs up to the next tick interrupt - during which time it could print out its string many times.

Periodic

Continuous 1

Continuous 2

Idle

The Idle task never enters the Running state as there are always higher priority task that are able to do so.

t1 t2 t3 Time t5

2 - The tick interrupt occurs during which the scheduler selects a new task to run. As both Continuous tasks have the same priority and both are always able to run the scheduler shares processing time between the two - so Continuous 2 enters the Running state where it remains for the entire tick period - during which time it could print out its string many times.

3 - At time t3 the tick interrupt runs again, causing a switch back to Continuous 1, and so it goes on.

Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 75

Scheduler FreeRTOS

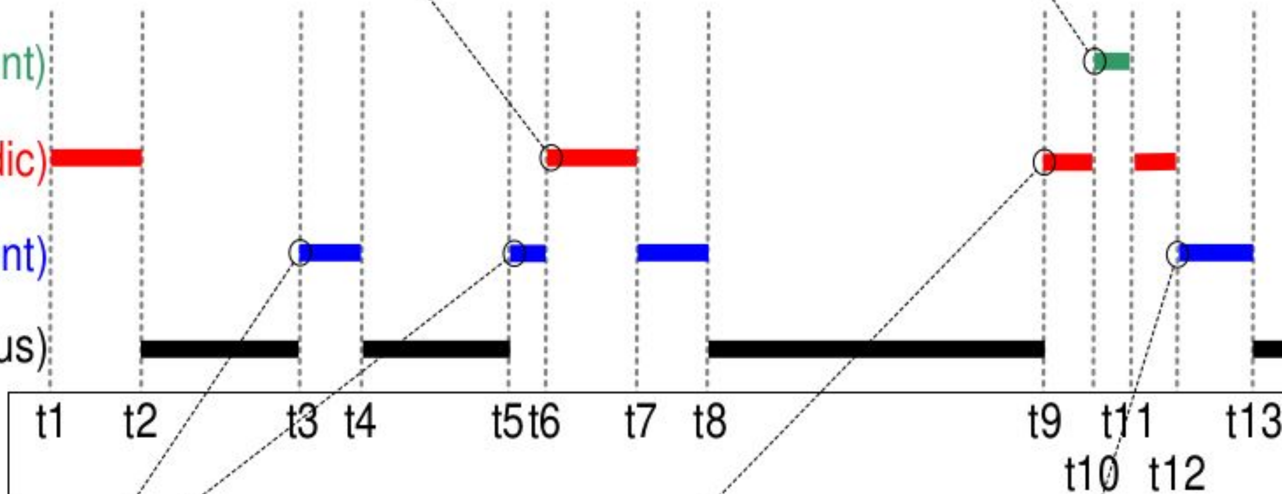
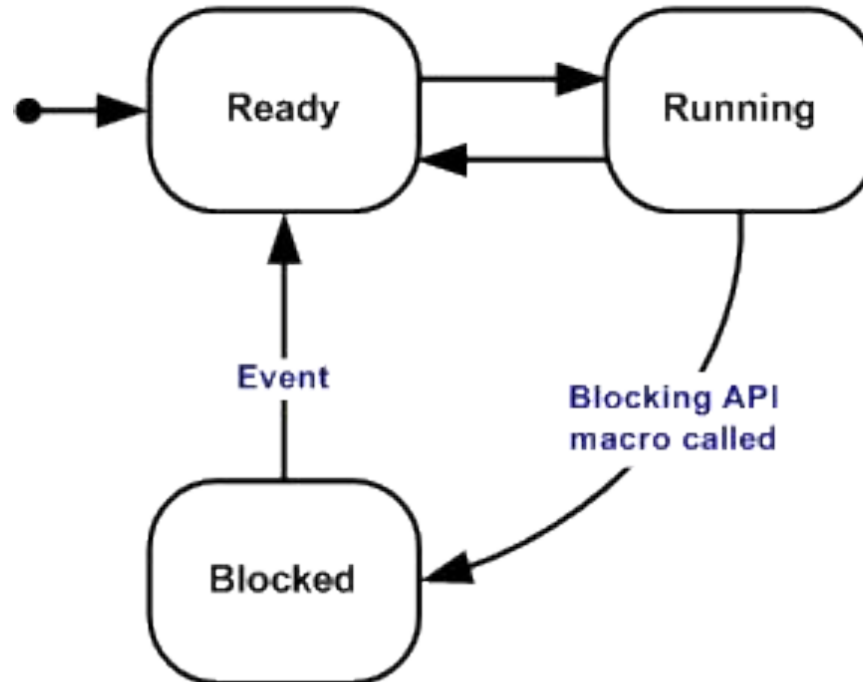


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 93

Co-rutinas

- Destinadas a procesadores de altas restricciones de RAM.



Co-rutinas

- Implementación
- Se crean con xCoRoutineCreate().
- Las tareas tienen prioridad sobre las corrutinas.

```
void vACoRoutineFunction( CoRoutineHandle_t xHandle,
                          UBaseType_t uxIndex )
{
    crSTART( xHandle );

    for( ;; )
    {
        -- Co-routine application code here. --
    }

    crEND();
}
```

FreeRTOS
Sintaxis

- Variables:
  - **TickType_t**: tipo de variable de la interrupción tick interrupt. Sirve para medir tiempos.
  - **BaseType_t**: Tipo de datos más eficiente de la arquitectura (depende del número de bits de la arquitectura).
  - Prefijos:
    - c: para char.
    - s: para int16_t (short).
    - l: para int32_t (long).
    - x: para BaseType_t
    - u: unsigned
    - p: punteros

FreeRTOS
Sintaxis

- Prefijos en las funciones:
  - **vTask**PrioritySet(): retorna **void** y está definida en **task.c**.
  - **xQueue**Receive(): retorna variable del tipo **BaseType_t** y está definida en **queue.c**.
  - **pvTimer**GetTimerID(): retorna un **puntero** y está definida en **timers.c**.
  - Funciones **privadas** en el alcance de un archivo añaden el prefijo **prv**.

FreeRTOS
Comunicación entre tareas - Colas

- **Buffers** tipo **FIFO** permiten **comunicación** entre:
  - Tareas hacia tareas.
  - Tareas hacia interrupciones.
  - Interrupciones hacia tareas.
- Dos tipos:
  - Cola por copia: el dato es copiado en la cola, byte por byte.
  - Cola por referencia: La cola contiene un puntero a los datos.
- Una tarea que intenta leer datos de una cola vacía pasa a estado blocked.
- Una tarea que intenta escribir datos en una cola llena pasa a estado blocked.

FreeRTOS - Comunicación entre tareas - Colas

Task A

```
int x;
x = 20;
```

Queue: [ ] [ ] [ ] [20] [10]

Send

Task B

```
int y;
```

Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.

Task A

```
int x;
x = 20;
```

Queue: [ ] [ ] [ ] [20] [10]

Receive

Task B

```
int y;
// y now equals 10
```

Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).

1 - The Receiver task runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Sender 2 runs after the Receiver has blocked.

3 - The Receiver task empties the queue then enters the Blocked state again. This time Sender 1 runs after the Receiver has blocked.

Receiver

Sender 2

Sender 1

t1

Time

2 - Sender 2 writes to the queue, causing the Receiver to exit the Blocked state. The Receiver has the highest priority so pre-empts Sender 2.

4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on ....

Primitivas para las colas

QueueHandle_t **xQueueCreate**(UBaseType_t **uxQueueLength**, UBaseType_t **uxItemSize**);

- uxQueueLength: número máximo de ítems que la cola puede tener.
- uxItemSize: Tamaño de cada elemento (en bytes).
- Return Value:
  - **NULL**: la cola no pudo crearse.
  - **non-NULL**: la cola se creó con éxito. Se retorna el manejador.

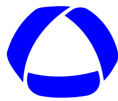
```
QueueHandle_t arrayQueue;
arrayQueue=xQueueCreate(10, //Queue length
                        sizeof(int)); //Queue item size
if(arrayQueue!=NULL){
```

Primitivas para las colas

BaseType_t **xQueueSendToFront**(QueueHandle_t **xQueue**, const void * **pvItemToQueue**, TickType_t **xTicksToWait**);
(copia datos al frente de la cola)

BaseType_t **xQueueSendToBack**(QueueHandle_t **xQueue**, const void * **pvItemToQueue**, TickType_t **xTicksToWait**);
(copia datos al final de la cola. Equivalente a xQueueSend()).

- xQueue: Manejador de la cola.
- pvItemToQueue: Puntero a los datos a escribir en la cola.
- xTicksToWait: Máxima cantidad de tiempo a esperar si la cola está llena (La tarea se bloquea).
- Returned value:
  - **pdPASS**: El dato se escribió correctamente en la cola.
  - **errQUEUE_FULL**: El dato no se escribió por estar la cola llena.



Primitivas para las colas

BaseType_t **xQueueReceive**(QueueHandle_t **xQueue**, void * const **pvBuffer**, TickType_t **xTicksToWait**);

- xQueue: Manejador de la cola
- pvBuffer: puntero a un buffer donde se escribirán los datos recibidos.
- xTicksToWait: tiempo máximo a esperar si no hay datos disponibles en la cola. La tarea se bloquea.
- Returned value:
  - **pdPASS**: Se leyeron datos de la cola satisfactoriamente.
  - **errQUEUE_EMPTY**: No se leyeron datos por estar la cola vacía.

Interrupciones en FreeRTOS

- Las **interrupciones tienen prioridad sobre las tareas**
  - La ISR de menor prioridad interrumpirá a la tarea de mayor prioridad.
  - Nunca una tarea interrumpirá a una ISR.
- Dos versiones de las funciones definidas por la API:
  - Funciones de la API para ser llamadas desde las tareas (pueden colocar la tarea en estado bloqueado).
  - Funciones de la API para ser llamadas desde las ISR (No se puede colocar una ISR en estado bloqueado).
    - Se agrega **FromISR** al nombre de la función.
      - **xQueueSendToBack**(...)
      - **xQueueSendToBackFromISR**(....)

Procesamiento de interrupción diferida

- Objetivo: que la duración de la rutina de servicio sea mínima.
  - La rutina de servicio solo:
    - **Desbloquea una tarea que atenderá el evento** que causó la interrupción.
    - **Realiza alguna tarea mínima**.
- Ventajas:
  - Evitar que las rutinas de servicio causen variaciones temporales en la ejecución de las tareas (afecten comienzo y finalización de las tareas).
  - Evitar el anidamiento de ISRs
    - Cada ISR requiere recursos.
    - Cada ISR requiere tiempo de ejecución (Se puede perder previsibilidad).

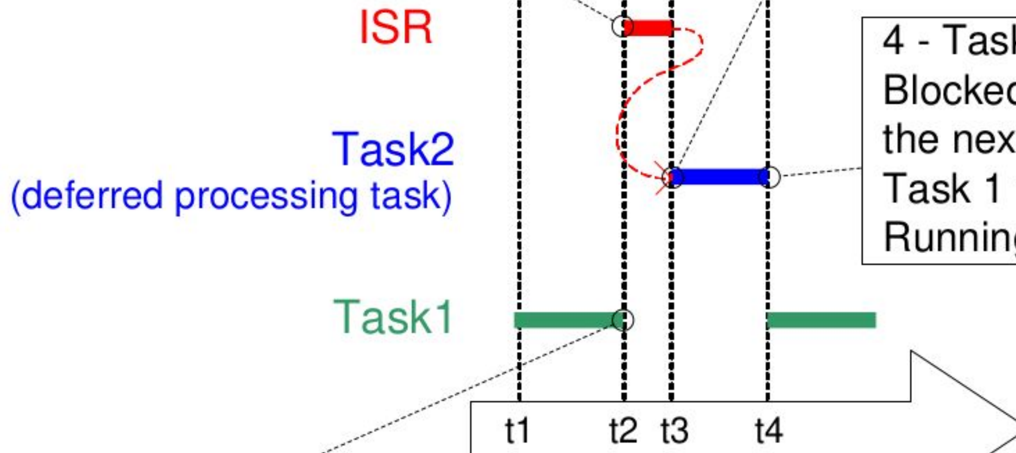
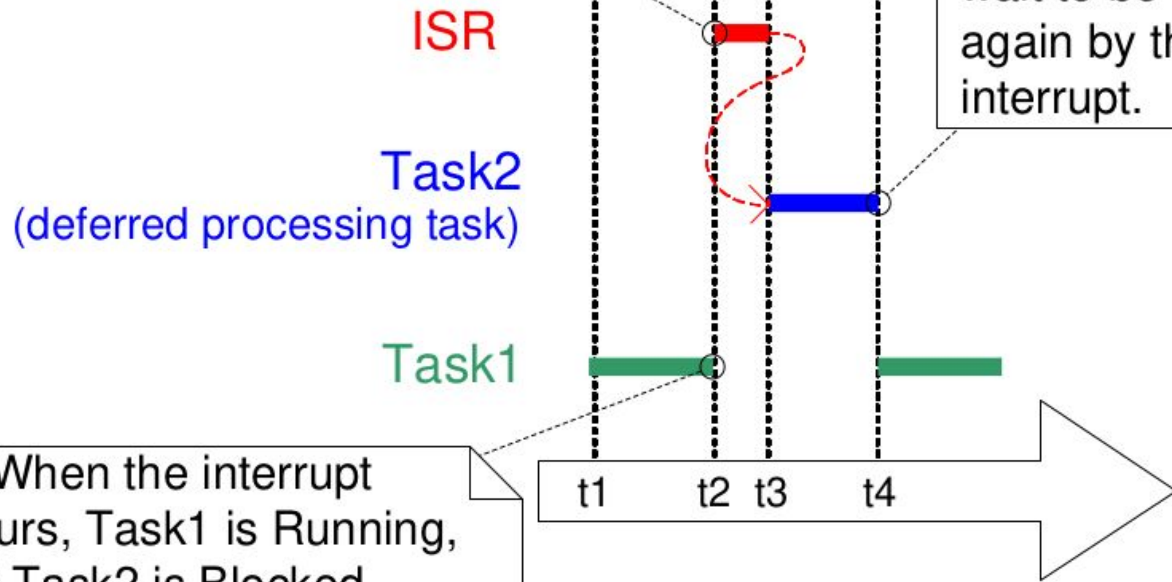


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 190.

2 - The ISR executes, handles the interrupting peripheral, clears the interrupt, then ‘gives’ a semaphore to unblock Task 2.

3 - Task 2 completes any further processing necessitated by the interrupt, then blocks on the semaphore to wait to be unblocked again by the next interrupt.

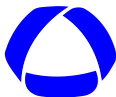


1 - When the interrupt occurs, Task1 is Running, and Task2 is Blocked waiting for a semaphore.

Semáforos binarios

- Mecanismo de sincronización (Dijkstra, 1965).
  - Pueden estar en estado "**disponible**" o "**no disponible**".
  - Si una tarea intenta tomar un semáforo no disponible, se bloquea.
  - Cualquier tarea puede hacer disponible un semáforo en estado no disponible (no necesariamente la tarea que lo tomó).
- SemaphoreHandle_t **xSemaphoreCreateBinary**(void);
  - Crea un semáforo. Devuelve NULL si el semáforo no pudo crearse (falta de memoria).

UNCUYO
UNIVERSIDAD NACIONAL DE CUYO


FACULTAD DE INGENIERÍA

Semáforos binarios

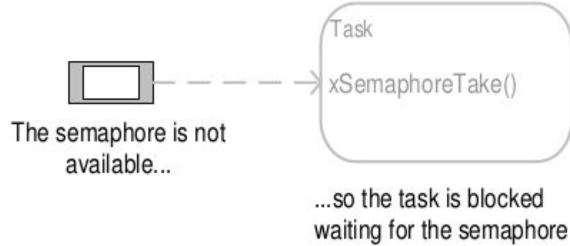
- BaseType_t **xSemaphoreTake**(SemaphoreHandle_t **xSemaphore**, TickType_t **xTicksToWait**);
  - Intenta tomar el semáforo (si está disponible). Si no está disponible, la tarea pasa a estado bloqueado.
  - xSemaphore: Manejador del semáforo.
  - xTicksToWait: Tiempo máximo a esperar (en ticks).
    - Si es configurado como **portMAX_DELAY**, la tarea esperará indefinidamente.
  - Valor retornado:
    - **pdPASS**: El semáforo fue obtenido exitosamente.
    - **pdFALSE**: El tiempo de espera expiró.

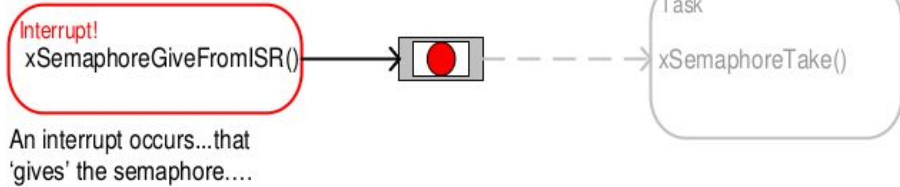
Semáforos binarios

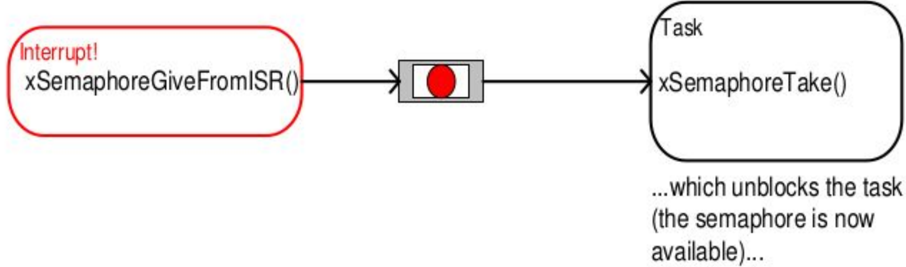
- **xSemaphoreGive**(SemaphoreHandle_t **xSemaphore**);
  - Otorga un semáforo. Para todos los tipos de semáforos (binarios, contador, etc).

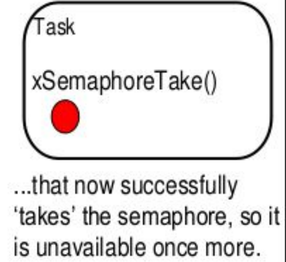
- Versión para utilizar desde interrupciones:
  - BaseType_t **xSemaphoreGiveFromISR**(SemaphoreHandle_t **xSemaphore**, BaseType_t *__pxHigherPriorityTaskWoken__);
    - Otorga el semáforo desde una ISR.
    - pxHigherPriorityTaskWoken es seteada a pdTRUE si se produce un cambio de contexto.

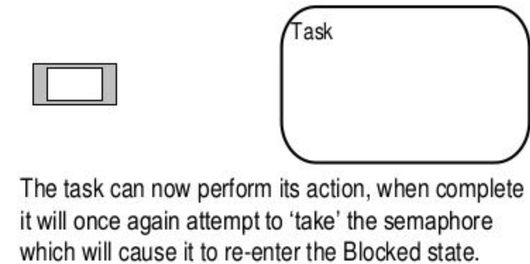
Licenciatura en Ciencias de la Computación









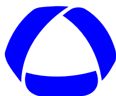


En este caso, la tarea se bloqueará hasta que **otra tarea** libere el semáforo.

Semáforos contadores

- Objetivos en los sistemas embebidos:
  - Contar eventos.
  - Manejar recursos.
- SemaphoreHandle_t **xSemaphoreCreateCounting**(UBaseType_t **uxMaxCount**, UBaseType_t **uxInitialCount**);
  - Valor de retorno:
    - NULL: El semáforo no pudo crearse.
    - non-NULL. El semáforo fue creado con éxito y devuelve un manejador de semáforo.
- Otorgar y tomar el semáforo (funciones vistas antes):
  - **xSemaphoreGive**(SemaphoreHandle_t **xSemaphore**);
  - BaseType_t **xSemaphoreTake**(SemaphoreHandle_t **xSemaphore**, TickType_t **xTicksToWait**);

UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO



FACULTAD
DE INGENIERÍA

Licenciatura en Ciencias de la Computación

Semáforos múltiples

[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....

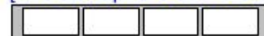
Interrupt!
xSemaphoreGiveFromISR()

[The semaphore count is 1]

Task
xSemaphoreTake()

...which unblocks the task (the semaphore is now available)...

[The semaphore count is 0]

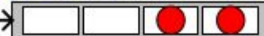


Task
vProcessEvent()

...that now successfully 'takes' the semaphore, so it is unavailable once more. The task now starts to process the event.

Interrupt!
xSemaphoreGiveFromISR()

[The semaphore count is 2]

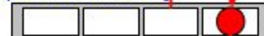


Task
vProcessEvent()

Another two interrupts occur while the task is still processing the first event. Both ISRs 'give' the semaphore, 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 the task has finished processing the first event it calls xSemaphoreTake() again. Another two semaphores are already 'available', one is taken without the task ever entering the Blocked state, leaving one 'latched' semaphore still available.

Exclusión mutua

Multitasking + recursos compartidos: Recursos estado inconsistentes.

- Una tarea1 comienza una operación, otra tarea2 interrumpe a tarea1 antes de que tarea1 finalice. El recurso puede quedar en estado inconsistente.
  - Ejemplos:
    - Tarea1 escribe “Hello world”, y tarea2 escribe “exit?”, el estado final puede ser: Hello worexit?ld.
    - Operaciones Read, Modify, Write.
    - Acceso a variables no atómicas.
- Exclusión mutua: Asegurar que una tarea tiene acceso exclusivo a un recurso.
- Algunos mecanismos de exclusión mutua:
  - Regiones críticas.
  - Suspender el Scheduler
  - Mutex.

Regiones críticas

- Regiones críticas: Un switch a otra tarea no puede ocurrir dentro de una región crítica.
  - Deshabilita interrupciones con prioridades menor a cierto umbral (el umbral depende de la implementación).
  - No se puede llamar a funciones desde una ISR que fue llamada desde una sección crítica.

```
taskENTER_CRITICAL();
….Región crítica…..
taskEXIT_CRITICAL();
```

Versión para interrupciones:

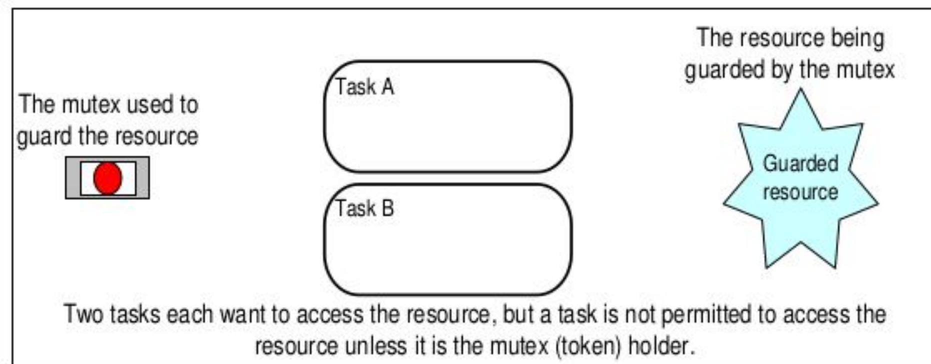
```
UBaseType_t  uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();
taskEXIT_CRITICAL_FROM_ISR(UBaseType_t  uxSavedInterruptStatus );
```

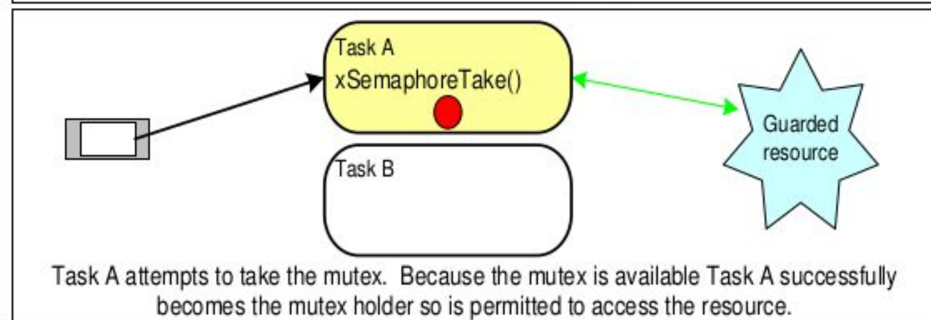
Suspender el Scheduler

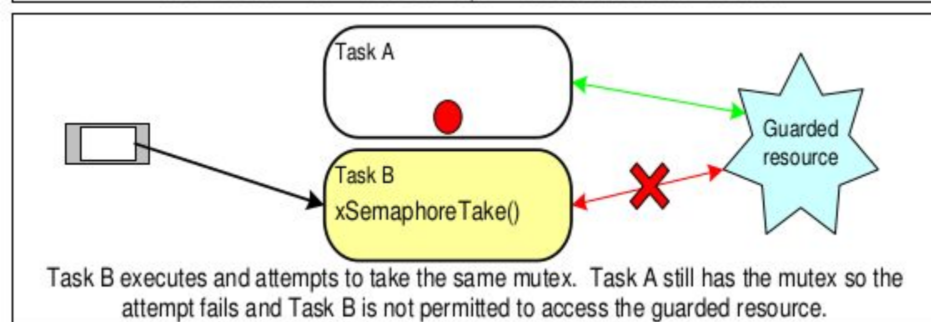
- Suspender el Scheduler.
  - Evita un switch de una tarea hacia otra.
  - No deshabilita interrupciones.
  - void **vTaskSuspendAll**(void);
  - BaseType_t **xTaskResumeAll**(void);
    - Valor de retorno:
      - **pdTRUE** si un cambio de contexto está pendiente.
      - **pdFALSE** si no hay cambio de contexto pendiente.

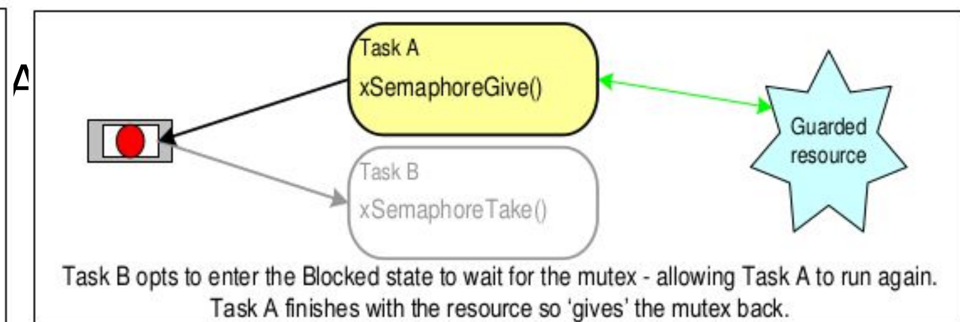
MUTEX

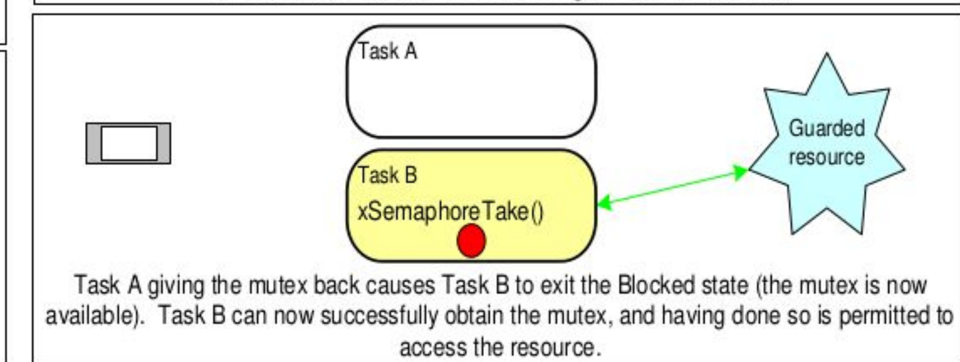
- MUTEX (MUTual EXclusion): Tipo de semáforo.
- Funcionamiento: similar a un **token**.
  - La tarea que necesita acceder a un recurso debe tomar el token.
  - Una vez que la tarea finaliza su trabajo con el recurso, debe liberar el token.
- **El mutex debe siempre ser devuelto** (en los semáforos no es obligatorio).











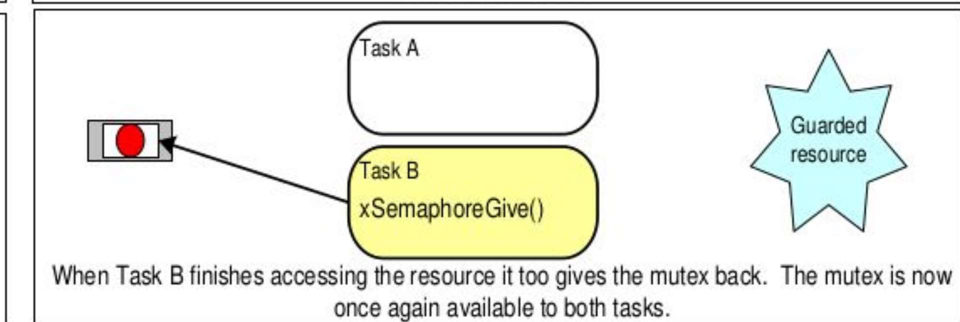


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 245.

3 - Task 2 attempts to take the mutex, but the mutex is still held by Task 1 so Task 2 enters the Blocked state, allowing Task 1 to execute again.

2 - Task 1 takes the mutex and starts to write out its string. Before the entire string has been output Task 1 is preempted by the higher priority Task 2.

5 - Task 2 writes out its string, gives back the semaphore, then enters the Blocked state to wait for the next execution time. This allows Task 1 to run again - Task 1 also enters the Blocked state to wait for its next execution time leaving only the Idle task to run.

Task 2

Task 1

Idle

t1

Time

1 - The delay period for Task 1 expires so Task 1 pre-empts the idle task.

4 - Task 1 completes writing out its string, and gives back the mutex - causing Task 2 to exit the Blocked state. Task 2 preempts Task 1 again

Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 250.

- SemaphoreHandle_t **xSemaphoreCreateMutex**(void);
  - Returned value:
    - **NULL**: El mutex no pudo ser creado.
    - non-NULL: El mutex pudo crearse. Retorna un manejador.
- BaseType_t **xSemaphoreTake**(SemaphoreHandle_t **xMutex**, TickType_t **xTicksToWait**);
- **xSemaphoreGive(** SemaphoreHandle_t **xSemaphore**);