Tareas en FreeRTOS

Sistemas multitarea:

- Soft Real Time: si una respuesta se produce después del límite, el sistema no es inutilizable
- Hard Real Time: el sistema debe responder dentro del tiempo límite establecido, sino representa la falla absoluta del sistema (ejemplo airbag)

Tarea: es un programa con un punto de entrada, correrá en un lazo infinito y nunca saldrá. En FreeRTOS no deben retornar, si no se requiere más, debe ser explícitamente borrada (vTaskDelete). Una función de tarea puede instanciarse muchas veces.

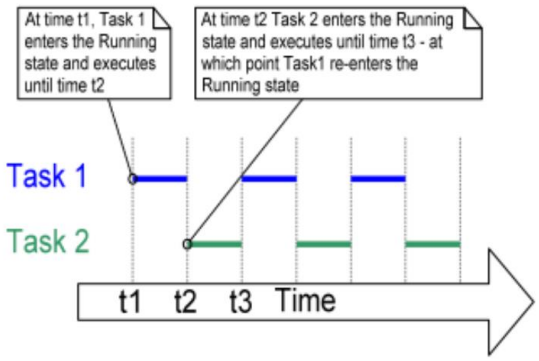
Aplicación: puede consistir en muchas tareas. El scheduler es el encargado de ir cambiando la tarea en ejecución en un momento dado.

Creación de tareas:

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask
                         );
```

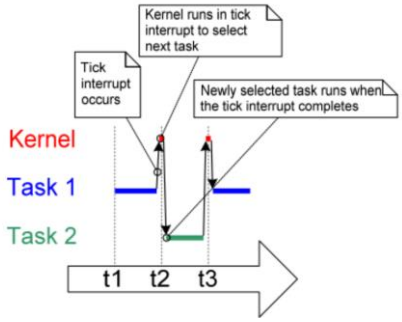
- pvTaskCode: puntero a la función que implementa la tarea
- pcName: nombre descriptivo de la tarea
- usStackDepth: tamaño de la pila para la tarea (número de palabras, no de bytes)
- pvParameters: parámetro del tipo puntero a void
- uxPriority: prioridad con la que la tarea se ejecutará
- pxCreatedTask: handler a la tarea creada, se usa para referenciar la tarea desde otras tareas
- Valor de retorno: pdTRUE indica que la tarea se creó correctamente, sino se indica que la tarea no se pudo crear porque no había suficiente RAM disponible

Cuando se crean 2 tareas con el mismo nivel de prioridad, van compartiendo el tiempo de ejecución:

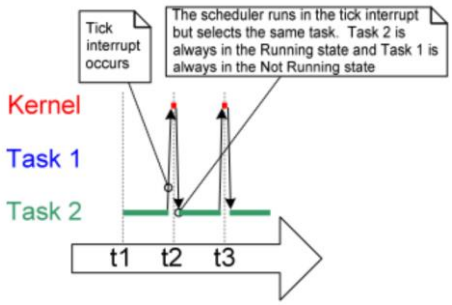


La prioridad de las tareas puede modificarse mediante la API vTaskPrioritySet(). Se puede definir la cantidad de prioridades que se desee, pero a mayor cantidad de prioridades, mayor consumo de RAM. La prioridad más baja posible es 0. El scheduler siempre ejecutará la tarea de mayor prioridad que esté disponible para ser ejecutada, alternando entre aquellas que

tienen el mismo nivel de prioridad. La interrupción periódica que permite esta alternancia es el "Tick", que puede ser ajustada por el usuario con la constante configTICK_RATE_HZ.



Si una tarea de mayor nivel de prioridad siempre está lista para ejecutarse, las tareas de menor prioridad jamás serán ejecutadas.



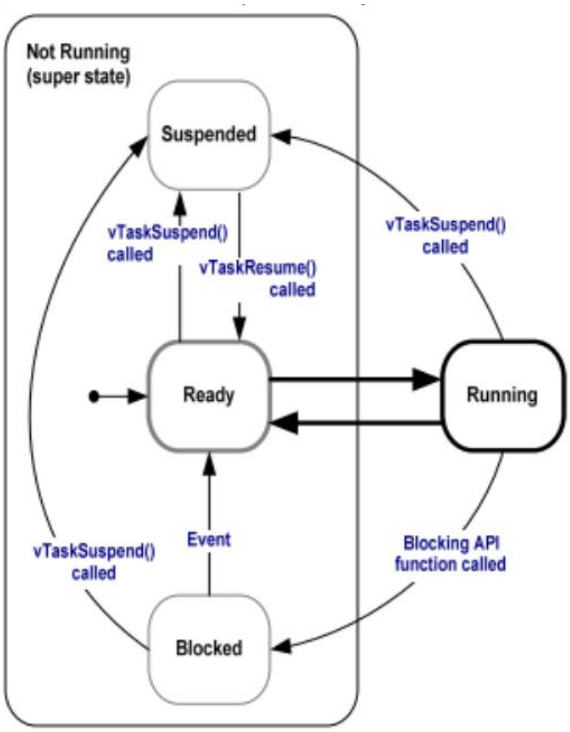
Estado de "no ejecución": las tareas de procesamiento continuo son aquellas que siempre están disponibles para ser ejecutadas. Las tareas manejadas por eventos son aquellas que sólo pueden entrar en estado de ejecución cuando ocurre dicho evento, de esta forma, tareas con menor prioridad pueden ser ejecutadas.

Estado bloqueado: cuando una tarea está esperando que ocurra un evento.

- Eventos temporales: la tarea espera un cierto tiempo que debe cumplirse
- Eventos de sincronización: el evento es generado por otra tarea, una interrupción, colas, semáforos.

Estado suspendido: la única manera de que entren en este estado es llamando a vTaskSuspend, y la única manera de salir es llamando a vTaskResume o vTaskResumeFromISR.

Estado disponible: están listas para ser ejecutadas, pero no están en estado de ejecución.



vTaskDelay: es una API que coloca a la tarea en estado bloqueado por una determinada cantidad de ticks. Mientras se encuentra bloqueada, no consume procesamiento. La constante portTICK_RATE_MS se usa para convertir milisegundos en ticks.

Cuando no hay ninguna tarea disponible para ser ejecutada, se ejecuta la tarea ociosa o idle task.

vTaskDelayUntil: esta función especifica el valor exacto que debe tener la cuenta de interrupciones tick para que la función pase de bloqueada a disponible. Se usa cuando se requiere un periodo fijo de ejecución.

```
void vTaskDelayUntil(portTickTyper *pxPreviousWakeTime, portTickType xTimeIncrement);
```

- pxPreviousWakeTime: es el tiempo en que la tarea dejó por última vez el estado bloqueado, se usa como punto de referencia para calcular el próximo momento en que la tarea debe dejar de estar bloqueada. Debe ser inicializada una vez y luego vTaskDelayUntil se encarga de actualizarla.
- xTimeIncrement: setea la frecuencia con que se ejecuta la tarea. Se puede usar portTICK_RATE_MS para pasar de ticks a milisegundos.

Cómo usar pxPreviousWakeTime:

```
portTickType xLastWakeTime;

xLastWakeTime = xTaskGetTickCount();

/* --- */

vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) )
```

Idle Task: el procesador necesita siempre algo para ejecutar, es decir que siempre debe haber al menos una tarea que puede entrar en estado de ejecución. Para esto, el scheduler crea automáticamente la tarea idle, que tiene la prioridad más baja posible para asegurar que no

impida la ejecución de las tareas de mayor prioridad. Se pueden añadir funcionalidades a la idle task (función de enlace):

- ejecución de baja prioridad o de procesamiento continuo
- medir la cantidad de capacidad de procesamiento extra
- colocar al procesador en modo bajo consumo

Pero a su vez la tarea idle no puede:

- intentar bloquear o suspender, sólo se ejecutará cuando no hay otra tarea capaz de hacerlo
- como la tarea idle se encarga de limpieza de los recursos del kernel, la función de enlace no puede permanecer permanentemente en ejecución

Ejemplo de función de enlace de tarea idle:

```
unsigned long ulIdleCycleCount = 0UL;

void vApplicationIdleHook( void )

{

        ulIdleCycleCount++;

}
```

configUSE_IDLE_HOOK debe establecerse en 1 dentro de FreeRTOSConfig.h para que la función de enlace pueda ser llamada.

Como cambiar la prioridad de una tarea:

```
void vTaskPrioritySet(xTaskHandle pxTask, unsignedportBASE_TYPE uxNewPriority);
```

- pxTask: handler de la tarea a modificar, NULL si es a sí misma
- uxNewPriority: la prioridad a establecer en la tarea

Consultar la prioridad de una tarea:

```
unsigned portBASE_TYPE uxTaskPriorityGet(xTaskHandle pxTask);
```

- pxTask: handler de la tarea a consultar, NULL si es a sí misma
- valor de retorno: prioridad de la tarea consultada

Cómo eliminar una tarea:

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

- pxTaskToDelete: handler de la tarea que se desea borrar, NULL si es a sí misma

Las tareas eliminadas ya no pueden entrar en ejecución nuevamente.

Resumen del Algoritmo del Scheduler:

- a cada tarea se le asigna una prioridad
- cada tarea puede existir en uno de varios estados
- solo una tarea puede estar en ejecución en un momento dado
- el scheduler siempre selecciona la tarea disponible de mas alto nivel de prioridad para entrar en ejecución

A este algoritmo se lo llama “Programación con prioridad fija y derecho preferente”. Prioridad fija porque no puede ser alterada por el kernel, y derecho preferente porque una tarea que entra en estado disponible siempre será ejecutada si su prioridad es mayor que el resto de las tareas disponibles.

Selección de la prioridad de las tareas: las que implementan funciones de tiempo real dura se le asignan prioridades por encima de las de tiempo real suave. También debe tenerse en cuenta los tiempos de ejecución y el uso del procesador.

El scheduling de tasa monótona es una técnica de asignación de prioridad que da un único valor de prioridad a las tareas que se ejecutan con la misma frecuencia. A mayor frecuencia de ejecución, mayor nivel de prioridad. Esto logra maximizar la planificación de las aplicaciones, pero los cálculos de periodicidad de las tareas son complejos.

Planificación cooperativa: FreeRTOS puede usar este tipo de planificación. En este tipo, el cambio de contexto sólo se produce cuando una tarea pasa a estado bloqueado o cuando una tarea en estado de ejecución hace un llamado a taskYIELD(). Es más simple, pero puede dar lugar a un sistema menos sensible.

También se pueden implementar sistemas híbridos, donde se usan rutinas de servicio de interrupción (ISR) para causar explícitamente un cambio de contexto. Esto permite eventos de sincronización, no eventos temporales.

Colas en FreeRTOS

Las colas son usadas por todas las comunicaciones y mecanismos de sincronización. Una cola puede contener un número finito de elementos de datos de tamaño fijo. El numero máximo de elementos que una cola puede contener es su longitud. La longitud y el tamaño de cada elemento se definen en la creación de la cola.

Las colas son buffers FIFO donde los datos se escriben en el final de la cola y se retiran en la parte frontal. La escritura se lleva a cabo copiando byte por byte los datos a ser almacenados. La lectura de los datos provoca que se eliminen de la cola. Acceso por múltiples tareas: que una cola tenga múltiples escritores es muy común, pero que tenga múltiples lectores es poco común, aunque posible.

Cuando una tarea intenta leer de una cola, puede especificar un tiempo de bloqueo, durante el cual puede quedarse esperando a que los datos estén disponibles. Una vez que los mismos estén disponibles, pasará a estado listo. Si hay múltiples tareas esperando por datos de una cola, la primera en desbloquearse es la de mayor prioridad, si tienen igual prioridad, es la que lleva mayor tiempo de espera.

También se puede establecer un tiempo de bloqueo cuando se escribe en una cola, tiempo durante el cual permanece bloqueada hasta que se libere espacio en la cola para poder escribirla. En el caso de que haya múltiples escritores, la que se desbloquea primero es la de mayor prioridad. Su tienen la misma prioridad, se desbloquea la que lleva esperando más tiempo.

Crear una cola:

```
xQueueHandle xQueueCreate (unsigned portBASE_TYPE uxQueueLength, unsigned
portBASE_TYPE uxItemSize);
```

- uxQueueLength: máximo numero de ítems que la cola puede contener
- uxItemSize: tamaño en bytes de cada elemento de datos que pueden ser almacenados en la cola
- valor de retorno: handler a la cola creada o NULL si no hay suficiente RAM disponible

`xQueueSendToFront`: se usa para enviar datos a la parte frontal de una cola.

```
portBASE_TYPE xQueueSendToFront( xQueueHandle xQueue, const void * pvItemToQueue,
portTickType xTicksToWait);
```

`xQueueSendToBack`: se usa para enviar datos a la parte frontal de una cola.

```
portBASE_TYPE xQueueSendToBack ( xQueueHandle xQueue, const void * pvItemToQueue,
portTickType xTicksToWait);
```

- xQueue: handler de la cola a la que se están enviando los datos
- pvItemToQueue: puntero a los datos que se copiarán a la cola
- xTicksToWait: cantidad máxima de tiempo que la tarea puede permanecer bloqueada esperando a que el espacio esté disponible en la cola, si es que ya está llena. portMAX_DELAY hará que el tiempo de espera sea indefinido siempre que INCLUDE_vTaskSuspend esté seteado en 1 en FreeRTOSConfig.h
- valor de retorno:
    - pdPASS: si los datos fueron enviados correctamente
    - errQUEUE_FULL: si los datos no pudieron ser escritos en la cola por estar llena

xQueueReceive: para leer un elemento de una cola. El elemento se elimina de la cola.

```
portBASE_TYPE xQueueReceive(xQueueHandle xQueue, const void * pvBuffer, portTickType xTicksToWait);
```

xQueuePeek: para leer un elemento de una cola sin que sea eliminado.

```
portBASE_TYPE xQueuePeek(xQueueHandle xQueue, const void * pvBuffer, portTickType xTicksToWait);
```

- xQueue: handler de la cola desde la que se están leyendo los datos
- pvBuffer: puntero a la memoria en que se copiarán los datos
- xTicksToWait: cantidad de tiempo que la tarea debe permanecer bloqueada a que el dato esté disponible en la cola, si es que está vacía
- Valor de retorno:
    - pdPASS: si los datos fueron leídos exitosamente
    - errQUEUE_FULL: si los datos no pudieron leerse porque la cola está vacía

xQueueMessagesWaiting: para consultar el número de elementos que se encuentran en la cola

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

- xQueue: handler de la cola
- valor de retorno: número de elementos que la cola contiene

Colas para transferir tipos compuestos: mediante estructuras

Colas para transferir datos grandes: usar punteros en lugar de copiar los datos. Es más eficiente en procesamiento y en la RAM necesaria para crearla. Pero se debe tener cuidado:

- El propietario de la RAM siendo apuntado sea claramente definido. Es decir que ambas tareas no modifiquen los contenidos de la memoria al mismo tiempo para evitar inconsistencias.
- La memoria RAM siendo apuntada sigue siendo válida: sólo una tarea debe ser responsable de liberar la memoria y ninguna tarea debe intentar acceder a ella luego de que fue liberada.

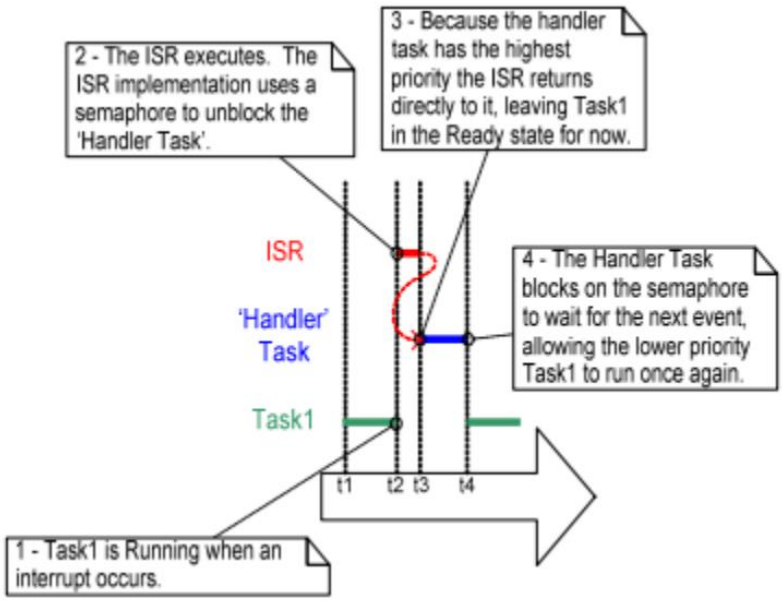
Interrupciones en FreeRTOS

Los sistemas embebidos de tiempo real tienen que tomar acciones en función de los eventos que ocurren en su entorno. En general, para detectar un evento, se usan interrupciones.

Procesamiento diferido de interrupción

Semáforos binarios usados para sincronización: un semáforo binario se puede usar para desbloquear una tarea cada vez que se produce una interrupción. Esto permite que los eventos de procesamiento de interrupción sean implementados dentro de la tarea sincronizada. Se dice entonces que el proceso de interrupción ha sido diferido a una tarea manipuladora.

Si el proceso de interrupción es muy crítico, se puede setear la prioridad de la tarea manipuladora para asegurarse que se adelante a la otra tarea. De esta forma tiene el efecto de ejecutar de forma contigua en el tiempo, somo si todo se hubiera implementado en el ISR.



Cuando se usan en sincronización de interrupciones, los semáforos se pueden pensar como una cola de un solo elemento, entonces siempre estará llena o vacía. Llamar a xSemaphoreTake mientras la cola está vacía hace que se bloquee la tarea. Cuando ocurre el evento, se llama a xSempahoreGiveFromISR, llenando la cola y haciendo que la tarea manipuladora salga del estafo bloqueado y obtenga el elemento de la cola, dejándola de nuevo vacía. Cuando la tarea manipuladora intenta volver a ejecutarse, intenta leer nuevamente a la cola vacía, entrando entonces en estado bloqueado, esperando al próximo evento.

Crear un semáforo:

```
void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

- xSemaphore: semáforo que se creará

Tomar un semáforo: es sinónimo de obtener o recibir un semáforo, si está disponible.

```
portBASE_TYPE xSemaphoreTake(xSemaphoreHandle xSemaphore, portTickType xTicksToWait);
```

- xSemaphore: semáforo que se va a tomar, tiene que haber sido creado antes

- xTicksToWait: máximo tiempo que debe permanecer bloqueada la tarea esperando al semáforo si es que no está disponible. Si se usa portMAX_DELAY la tarea esperará indefinidamente a que el semáforo esté disponible.
- Valor de retorno: hay 2 posibles valores
    - pdPASS: si se obtuvo el semáforo
    - pdFALSE: si el semáforo no estaba disponible (expiró el tiempo de espera)

Entregar un semáforo:

```
portBASE_TYPE xSemaphoreGiveFromISR( xSemaphoreHandle xSemaphore, portBASE_TYPE
*pxHigherPriorityTaskWoken );
```

- xSemaphore: semáforo a entregar
- pxHigherPriorityTaskWoken: es posible que un semáforo tenga bloqueada a más de una tarea. xSemaphoreGiveFromISR puede hacer que el semáforo esté disponible y hacer que una tarea deje el estado bloqueado. Si una tarea deja el estado bloqueado y tiene prioridad más alta que la que está en ejecución xSemaphoreGiveFromISR establecerá internamente pxHigherPriorityTaskWoken a pdTRUE. Si esto ocurre, un cambio de contexto se debe realizar entes de salir de la interrupción. Esto asegurará que la interrupción regresa directamente a la tarea disponible con más alta prioridad. (Tenemos que forzar nosotros el cambio de contexto antes de salir del handler de la interrupción)
- Valor de retorno: hay 2 posibles valores:
    - pdPASS: si el llamado a la función es satisfactorio
    - pdFAIL: si el semáforo ya estaba disponible no puede ser retornado nuevamente

Semáforos Counting: en los semáforos binarios, el flujo de ejecución es:

- ocurre una interrupción
- la ISR es ejecutada, dando el semáforo para desbloquear la tarea manipuladora
- la tarea manipuladora es ejecutada cuando la interrupción es completada, lo primero que hace la tarea manipuladora es tomar el semáforo
- la tarea manipuladora procesa el evento antes de intentar volver a tomar el semáforo nuevamente, entrando en establo bloqueado si el semáforo no estaba nuevamente disponible

Esta secuencia es útil cuando la interrupción ocurre a frecuencias relativamente bajas. Si otra interrupción ocurre antes de que la tarea manipuladora complete su procesamiento, el semáforo binario estaría nuevamente disponible, permitiendo que se vuelva a procesar el evento inmediatamente después de terminar de procesar el primero. La tarea manipuladora no entraría en estado bloqueado entre el procesado del 1er y 2do evento. Un semáforo binario puede retener como máximo un evento de interrupción. Si ocurren más eventos mientras el semáforo está disponible, se perderán. Para eso se pueden usar los semáforos counting, que pueden tener un largo de más de un elemento. Cada vez que un semáforo counting es dado, un espacio de la cola es liberado. El número de ítems de la cola es la cuenta del semáforo.

Los semáforos counting son usados para:

- Contar eventos: un evento da un semáforo cada vez que ocurre, entonces la cuenta del semáforo se incrementa cada vez que se da un semáforo. La tarea manipuladora toma un semáforo cada vez que procesa un evento, haciendo que la cuenta se reduzca. La

cuenta del semáforo es la diferencia entre la cantidad de eventos que se han procesado. En este caso los semáforos se crean con un valor inicial 0.
- Gestión de recursos: la cuenta del semáforo indica el número de recursos disponibles. Para tomar un recurso, una tarea debe primero obtener un semáforo, reduciendo el valor de la cuenta de semáforos. Cuando la cuenta llega a 0, no hay más recursos libres. Cuando una tarea termina de usar un recurso devuelve el semáforo, incrementando el valor de la cuenta. En este caso los semáforos se crean con un valor inicial igual al número de recursos disponibles.

Crear un semáforo counting:

```
xSemaphoreHandle xSemaphoreCreateCounting( unsignedportBASE_TYPE uxMaxCount,
unsignedportBASE_TYPE uxInitialCount );
```

- uxMaxCount: valor máximo de la cuenta de semáforo. Es el número máximo de eventos que pueden ser guardados en el semáforo.
- uxInitialCount: valor inicial de la cuenta.
- Valor de retorno: NULL si no se pudo crear porque no había memoria disponible. No nulo si se creó satisfactoriamente.

Los semáforos se usan para comunicar eventos. Las colas se usan tanto para comunicar eventos como para transferir datos. Cuando queremos usar colas desde una interrupción, se deben usar las siguientes funciones:

```
portBASE_TYPE xQueueSendToFrontFromISR( xQueueHandle xQueue, void *pvItemToQueue,
portBASE_TYPE *pxHigherPriorityTaskWoken);

portBASE_TYPE xQueueSendToBackFromISR( xQueueHandlexQueue, void *pvItemToQueue,
portBASE_TYPE *pxHigherPriorityTaskWoken);
```

- xQueue: manipulador de la cola a la que se enviará el dato
- pvItemToQueue: puntero al dato que será copiado en la cola. El tamaño de cada ítem que la cola puede guardar es establecido cuando se crea la cola
- pxHigherPriorityTaskWoken: se usa para lo mismo que antes, para asegurar el cambio de contexto a una tarea de mayor prioridad.
- Valor de retorno: hay 2 posibilidades:
    - pdPASS: dato enviado a la cola
    - errQUEUE_FULL: el dato no se pudo enviar porque la cola estaba llena

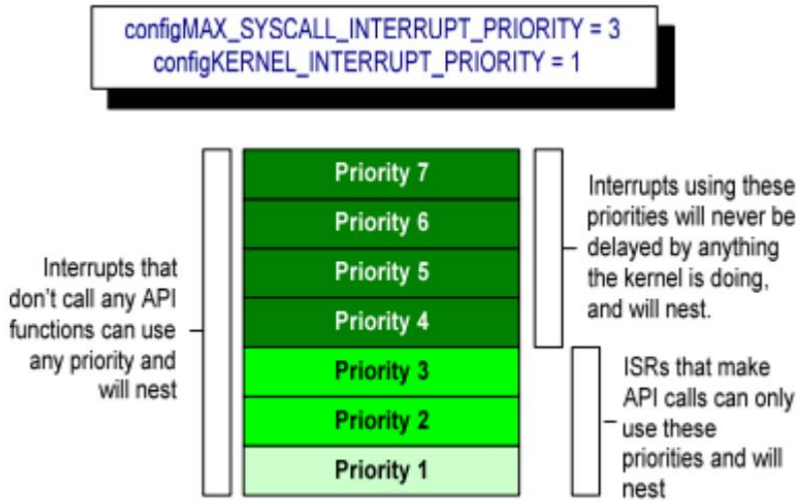
Uso eficiente de colas: los ejemplos suelen ser un driver de UART que pasan caracteres a través del handler de la interrupción. Sin embargo, enviar caracteres individuales a través de colas es ineficiente. Es mejor:

- ubicar el carácter recibido en un buffer RAM y luego usar un semáforo para desbloquear una tarea que procese el buffer una vez que el mensaje completo es recibido
- interpretar los caracteres recibidos directamente dentro de la ISR, luego usar una cola para enviar los comandos interpretados y decodificados. Esto sólo es válido si la interpretación de la cadena es lo suficientemente rápida para realizarse dentro de la interrupción.

Anidación de interrupciones: para eso se necesitan 2 constantes que deben ser definidas en FreeRTOSConfig.h:

- configKERNEL_INTERRUPT_PRIORITY: setea la prioridad de la interrupción tick.
- configMAX_SYSCALL_INTERRUPT_PRIORITY: setea el valor más elevado de prioridad las interrupciones.

Un modelo de anidación de interrupciones completa se crea mediante el establecimiento de configMAX_SYSCALL_INTERRUPT_PRIORITY a una prioridad mayor que configKERNEL_INTERRUPT_PRIORITY. La diferencia entre prioridades de trabajo y prioridades de interrupción es que las de interrupción están basadas en la arquitectura del microcontrolador y son controladas por hardware. Las tareas no se ejecutan en ISR por lo que la prioridad de software asignado a una tarea no está relacionada con la prioridad de hardware asignado a una fuente de interrupción.



- Las interrupciones que usan prioridades entre 1 y 3 no podrán ejecutarse mientras el kernel o la aplicación se encuentren dentro de la sección crítica, pero pueden usar las funciones API de interrupciones seguras
- Las interrupciones con prioridad 4 o mayor no son afectadas por las secciones críticas, entonces nada de lo que haga el kernel impedirá que estas interrupciones se ejecuten inmediatamente. Funcionalidades que requieren tiempos muy precisos usarán prioridades mayores a configMAX_SYSCALL_INTERRUPT_PRIORITY para segurar que el scheduler no introduzca variaciones en los tiempos de respuesta de la interrupción
- Las interrupciones que no hacen uso de llamados a funciones API de FreeRTOS pueden tener cualquier prioridad

Administración de Recursos en FreeRTOS

En los sistemas multitarea se puede dar el caso en que una tarea comienza a acceder a un recurso, pero no completa su acceso antes de que una transición le quite el estado de ejecución. Si la tarea deja al recurso en un estado incoherente, entonces el acceso al recurso por parte de otra tarea puede dar lugar a corrupción de datos. Ejemplos pueden ser: escribir en un LCD, acceso no atómico a variables, etc.

Exclusión mutua: se usa para asegurar la consistencia de los datos de un recurso compartido, dando acceso exclusivo al recurso a la tarea hasta que la misma termine de usarlo. Siempre lo mejor es diseñar la aplicación para evitar que los recursos se compartan.

Secciones críticas: son regiones de código que están rodeadas por las macros taskENTER_CRITICAL y taskEXIT_CRITICAL. Estas macros lo único que hacen es desactivar las interrupciones para evitar los cambios de contexto, garantizando que la tarea se mantendrá en estado de ejecución mientras esté accediendo al recurso crítico.

Las secciones críticas también se pueden crear mediante la suspensión del scheduler. Esta forma sólo protege una región de código de acceso de otras tareas, porque las protecciones permanecen habilitadas. Para secciones críticas largas se puede usar este método, pero la reanudación puede ser demasiado larga, por lo que se debe considerar cada caso particular.

vTaskSuspendAll: al suspender el scheduler se evita que ocurra un cambio de contexto, pero deja las interrupciones habilitadas. Si una interrupción solicita un cambio de contexto mientras el scheduler está suspendido, la solicitud se mantiene en espera y sólo se realiza cuando el scheduler se reanuda. Las API de FreeRTOS no deberían ser llamadas mientras el scheduler está suspendido.

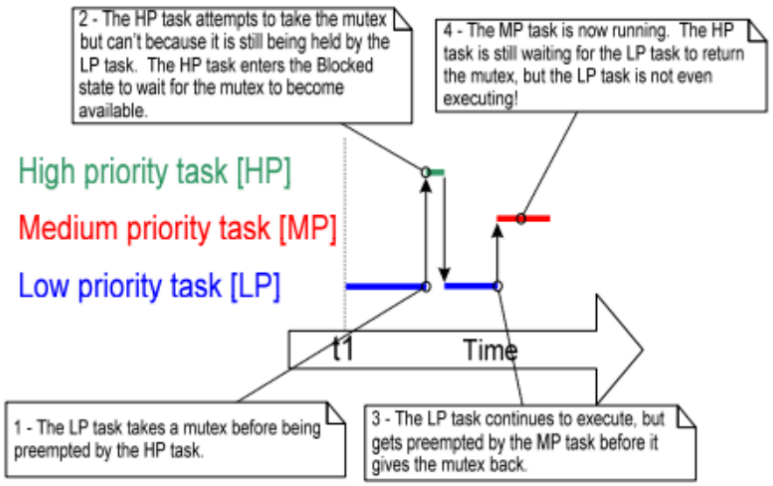
sTaskResumeAll: el scheduler reanuda su funcionamiento. Para usar estas llamadas es conveniente implementar anidamiento porque el kernel mantiene una cuenta de profundidad de anidación. El scheduler entonces sólo se reanudará cuando la profundidad de anidación vuelve a cero, es decir cuando una llamada a resume se ha ejecutado para cada suspend.

Semáforos mutex: es un tipo de semáforo que se usa para controlar el acceso a un recurso que se comparte entre 2 o más tareas. Conceptualmente es como asociar un token al recurso. Para que una tarea pueda acceder al recurso, debe tomar el token. Cuando termina de usar el recurso, debe devolver el token. Sólo cuando el token se ha devuelto otra tarea puede tomarlo y luego acceder de forma segura al mismo recurso compartido. El mecanismo funciona si el programa está bien diseñado.

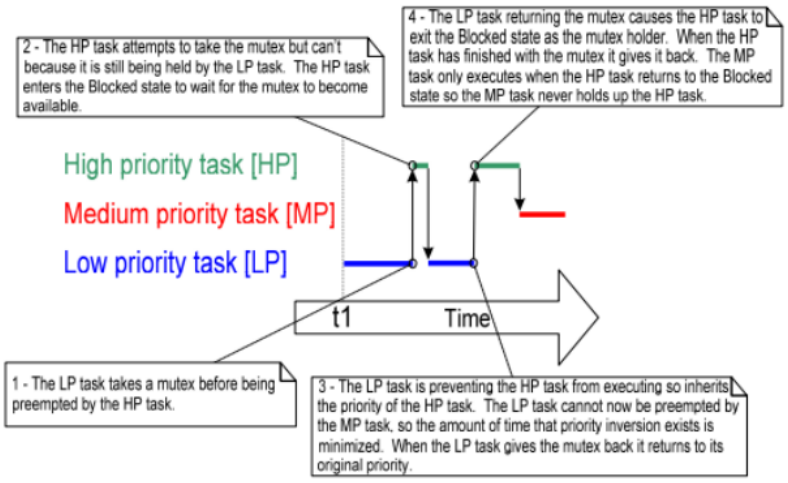
```
xSemaphoreHandle xSemaphoreCreateMutex ( void );
```

- Valor de retorno: si devuelve NULL es porque no hay memoria suficiente para crear el semáforo. Si es distinto de NULL, el semáforo pudo crearse con éxito y el valor devuelto es el manipulador del semáforo.

Inversión de prioridad: esto se da cuando una tarea de menor prioridad toma un mutex, y la tarea de mayor prioridad debe esperar a que la otra lo libere. Un caso extremo sería que, mientras una tarea de baja prioridad tiene el mutex, la tarea de alta prioridad intente tomarlo y no pueda. Y encima, antes que la tarea de baja prioridad libere el mutex, una tarea de prioridad media le gane a la de baja prioridad, entonces ambas tareas se están ejecutando antes que la de mayor prioridad.



Herencia de prioridades: los semáforos mutex proporcionan un mecanismo de herencia de prioridad. Esto minimiza los efectos negativos de la inversión de prioridades. Consiste en elevar temporalmente la prioridad de la tarea que tiene el mutex al mismo nivel que la tarea de mayor prioridad que está intentando tomar el mutex. La tarea que tiene el mutex “hereda” la prioridad de la tarea que está esperando poder tomar el mutex. La prioridad de la tarea que tenía el mutex se reestablece a su valor original una vez que lo devolvió. Esto minimiza el caso anterior, porque ahora la tarea de prioridad intermedia no puede pisar a la de menor prioridad mientras esta tiene el mutex.



Punto muerto: es otro problema potencial de uso de mutex para exclusión mutua. Se produce cuando 2 tareas no pueden continuar porque ambos están a la espera de un recurso que está en manos de la otra.

- La tarea A toma el mutex X
- La tarea B adelanta a la tarea A
- La tarea B toma el mutex Y e intenta tomar el mutex X, pero no lo logra porque lo tiene la A. Entonces se bloquea esperando al X.
- La tarea A vuelve a ejecutarse e intenta tomar el mutex Y, pero no puede porque la B lo está tomando. Entonces la tarea B también se bloquea.

El escenario descripto se denomina interbloqueo. La mejor manera de contrarrestarlo es tener en cuenta que puede ocurrir al momento de diseñar el sistema.

Tareas guardianas: son una forma de implementar la mutua exclusión sin preocuparse por caer en inversión de prioridad o punto muerto. Una tarea guardiana tiene la propiedad exclusiva de algún recurso. Cualquier otra tarea que deba acceder a ese recurso, sólo puede hacerlo indirectamente a través de la guardiana. Por ejemplo, para imprimir en una pantalla, se puede implementar una tarea que reciba lo que se quiere imprimir mediante una cola y sólo esta tarea es la que puede imprimir sobre la pantalla.

Manejo de memoria: el kernel tiene que asignar dinámicamente la memoria RAM cada vez que se crea una tarea, cola o semáforo. Las funciones malloc y free pueden usarse pero no siempre están disponibles en los sistemas embebidos pequeños, su implementación puede ocupar demasiado espacio, no suelen ser seguras, no son determinísticas, pueden sufrir fragmentación de memoria y pueden complicar la vinculación.

FreeRTOS tiene sus implementaciones para mejorar la portabilidad del código: pvPortFree y pvPortMalloc.

Esquemas de asignación de memoria:

- Heap_1.c
  Implementa una versión básica de pvPortMalloc y no implementa pvPortFree. Cualquier aplicación que no borra tareas, colas o semáforos pueden usar heap_1, que siempre es determinística. El tamaño total de la memoria disponible es fijado de forma estática por la definición de configTOTAL_HEAP_SIZE dentro de FreeRTOSConfig.h
- Heap_2.c
  Usa un algoritmo que mejora la asignación de memoria y permite liberarla. El tamaño máximo también es definido estáticamente mediante configTOTAL_HEAP_SIZE. El algoritmo permite usar la porción de memoria que más se ajusta (el bloque libre que está más cerca en tamaño a la cantidad de bytes solicitados.
  Es un algoritmo adecuado cuando la aplicación crea y elimina tareas, siempre que el tamaño máximo de la pila no cambie.
  Heap_2 no es determinística, pero es más eficiente que la mayoría de las implementaciones de la biblioteca estándar de malloc y free.
- Heap_3.c
  Usa la librería estándar de malloc y free pero hace que las llamadas sean seguras, suspendiendo temporalmente el scheduler. El tamaño de la pila no lo define configTOTAL_HEAP_SIZE, lo asigna el linker.

Solución de problemas en FreeRTOS

Los problemas más comunes en FreeRTOS están relacionados con el desbordamiento de la pila.

printf-stdarg.c: es una versión mínima y eficiente de la pila de sprintf que se puede usar en lugar de la librería estándar, requiriendo mucho menor espacio que la pila.

Para cuando se produce el desbordamiento de la pila, FreeRTOS tiene APIs para depuración:

uxTaskGetStackHighWaterMark: se usa para consultar cuán cerca una tarea está de desbordar el espacio de pila que le sea asignado. Este valor se denomina “cota máxima” de la pila.

```
unsigned portBASE_TYPE uxTaskGetStackHighWaterMark( xTaskHandle xTask );
```

- xTask: el handler de la tarea que está siendo consultada. Se puede consultar el estado de la propia tarea pasando NULL.
- Valor de retorno: cantidad mínima de espacio de pila restante que estuvo disponible desde que la tarea inició la ejecución. Es la cantidad de pila que no fue utilizada cuando el uso de la pila estaba en su máximo valor. Cuanto más cerca la cota máxima está del valor 0, más cerca estuvo la tarea de desbordar la pila.

Verificación del tiempo de ejecución de pila: FreeRTOS incluye 2 mecanismos para la verificación del tiempo de ejecución de la pila. Son controlados por la constante configCHECK_FOR_STACK_OVERFLOW. Ambos métodos incrementarán el tiempo que se tarda en realizar un cambio de contexto. El enlace de desbordamiento de pila es una función que es llamada por el kernel cuando detecta un desbordamiento de la pila. Para poder usarla hay que:

- Establecer configCHECK_FOR_STACK_OVERFLOW en 1 o 2 en FreeRTOSConfig.h
- Proporcionar la implementación de la función de enlace, usando el nombre de la función y prototipo de la función adecuadamente.
  ```
  void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR
  *pcTaskName );
  ```

El enlace de desbordamiento de la pila se usa para encontrar y depurar errores de pila más fácilmente, pero no hay forma de recuperarse de un desborde una vez que ya ocurrió. El enlace de desbordamiento de pila puede ser llamado desde el contexto de una interrupción. Algunos microcontroladores generarán una excepción de falta cuando detectan un acceso de memoria incorrecta y es posible que una falta se active antes que el kernel tenga la oportunidad de llamar a la función de enlace de desbordamiento.

Método 1: configCHECK_FOR_STACK_OVERFLOW en 1. El contexto de ejecución entero de una tarea se guarda en su pila cada vez que sale del estado de ejecución. Es probable que este es el tiempo en el que el uso de la pila alcanza su pico. Cuando configCHECK_FOR_STACK_OVERFLOW está en 1, el kernel verifica que el puntero de pila permanece dentro del espacio de pila válida. Si el puntero está fuera del rango válido, se llama el enlace de desbordamiento de pila. Este método es rápido de ejecutar, pero se pueden perder desbordamientos que se produzcan entre los cambios de contexto.

Método 2: realiza comprobaciones adicionales al método 1. configCHECK_FOR_STACK_OVERFLOW debe estar en 2. Cuando se crea una tarea, su pila se rellena con un patrón conocido. El método 2 camina los últimos 20 bytes válidos de la pila para comprobar que el patrón no se sobrescribió. La función de enlace de desbordamiento se llama si alguno de los 20 bytes ha cambiado su valor. No es tan rápido como el método 1, pero es probable que pueda detectar todos los desbordamientos de pila.

Otros posibles errores que pueden surgir:

- Agregar una tarea a una demostración causa que la aplicación deje de funcionar. Muchos de los proyectos dimensionan el tamaño del bloque de memoria para que sólo puedan soportar las tareas de demostración, si se crean tareas adicionales, la memoria será insuficiente. La tarea idle se crea automáticamente cuando se llama a vTaskStartScheduler. Esta función devuelve sólo si no hay suficiente memoria para que se cree la idle task. Poner un loop después de vTaskStartScheduler puede hacer que este error sea más fácil de detectar.
- Usar una API dentro de una interrupción hace que la aplicación deje de funcionar. Hay que usar sólo APIs que terminen en fromISR dentro de los servicios de interrupción.
- La aplicación deja de funcionar dentro durante una ISR. Hay que chequear que la interrupción no esté causando un desbordamiento de la pila. Garantizar la prioridad asignada a cada interrupción teniendo en cuenta que los números bajos se usan para representar interrupciones lógicamente de alta prioridad.
- El scheduler deja de funcionar cuando intenta iniciar la primera tarea: asegurarse que el procesador está en modo de supervisor antes de realizar el llamado a vTaskStartScheduler. La forma más sencilla de lograr esto es colocar el procesador en modo supervisor dentro del código de inicio C antes de llamar al main (para ARM7)
- Las secciones críticas no se anidan correctamente: no altere los bits de habilitación de interrupciones o los flags de prioridad del microcontrolador usando cualquier método que no sea las llamadas a taskENTER_CRITICAL y taskEXIT_CRITICAL. Estas macros mantienen una cuenta de la profundidad de anidamiento de las llamadas para asegurar que las interrupciones sólo se habiliten de nuevo cuando la anidación ha vuelto completamente a cero.
- La aplicación deja de funcionar incluso antes de iniciar el scheduler. Una ISR que podría causar un cambio de contexto no debe permitirse que se ejecute antes de que se ha iniciado el planificador. Lo mismo es para cualquier ISR que intenta enviar o recibir de una cola o semáforos. Un cambio de contexto no puede ocurrir hasta después que se haya iniciado el planificador. Muchas API no pueden llamarse antes de ser iniciado el planificador, es mejor restringir el uso de APIs para la creación de tareas, colas y semáforos hasta después que haya sido llamada vTaskStartScheduler
- Llamados a funciones API mientras el scheduler está suspendido causa que la aplicación deje de funcionar. El scheduler se suspende llamando a vTaskSuspendAll() y es reanudado llamando xTaskResumeAll(). No llame a funciones de la API, mientras que el programador esté suspendido.
- El prototipo pxPortInitialiseStack() causa que la compilación falle. Lo más probable es que la macro se establece de forma incorrecta para el puerto que se utiliza