

Usando el Kernel de Tiempo Real FreeRTOS:

(Traducción de la Guía práctica por Richard Barry)

Capítulo 4: Administración de Recursos

4.1 Introducción y alcance de este capítulo:

Existe un conflicto potencial que puede surgir en un sistema multitarea, si una tarea comienza a acceder a un recurso, pero no completa su acceso antes de una transición que la quita del estado de ejecución. Si la tarea deja el recurso en un estado incoherente, entonces el acceso a este recurso por cualquier otra tarea o interrupción podría dar lugar a la corrupción de datos u otro error similar.

Algunos ejemplos:

1. Acceso a periféricos:

Considere el siguiente escenario donde dos tareas intentan escribir en un LCD:

- La tarea A se ejecuta y comienza a escribir "Hola mundo" en el LCD.
- La tarea B se desbloquea y se adelanta a la tarea A, justo cuando esta había escrito solo "Hola m".
- La tarea B escribe "Abortar, Reintentar, Fallo?" en el LCD y luego entra en el estado bloqueado.
- La tarea A continúa desde el punto donde había quedado, y completa la frase colocando "undo".

El LCD mostrará la siguiente frase "Hola mAbortar, Reintentar, Fallo?undo" .

2. Operaciones de lectura, modificación y escritura:

En el Listado 57 se muestra una línea de código C, y su salida correspondiente en assembler (ARM7). Se puede observar que el valor del Puerto A es leído primero desde la memoria hacia un registro, es modificado en este registro, y luego es escrito nuevamente en la memoria. Esto es una operación de lectura, modificación y escritura.

/ Código C que será compilado. */*

155: PORTA |= 0x01;

/ Código Assembler producido. */*

0x00000264 481C LDR R0,[PC,#0x0070]	; Obtiene la dirección del Puerto A
0x00000266 6801 LDR R1,[R0,#0x00]	; Lee el valor del Puerto A hacia R1
0x00000268 2201 MOV R2,#0x01	; Mueve la constante absoluta 1 a R2
0x0000026A 4311 ORR R1,R2	; OR R1 (Puerto A) con R2 (constante 1)
0x0000026C 6001 STR R1,[R0,#0x00]	; Guarda el nuevo valor de nuevo en el Puerto A

Listado 57 – Ejemplo de una secuencia de lectura, modificación y escritura.

Esta es una operación "no-atómica" porque se necesita más de una instrucción para completarla, y puede ser interrumpida. Considere el siguiente escenario donde dos tareas intentan actualizar un registro asignado en memoria llamado PORTA:

- La tarea A carga el valor de PORTA en un registro (parte de lectura de la operación).
- La tarea B abandona el estado bloqueado y adelanta a la tarea A, antes que esta complete la modificación y la escritura de la misma operación.
- La tarea B actualiza el valor de PORTA y luego se bloquea.
- La tarea A continúa desde el punto donde había dejado. Modifica la *copia* de PORTA, que ya estaba en un registro, y luego vuelve a escribirlo en PORTA.

La tarea A ha actualizado y escrito un valor desactualizado de PORTA. La tarea B había modificado a PORTA entre las acciones de leído y modificación, que realiza la tarea A. Cuando la tarea A escribe en PORTA, está sobrescribiendo la modificación que había realizado la tarea B.

Este ejemplo utiliza un registro periférico, pero el mismo principio se aplica cuando se realiza la operación de lectura, modificación, y escritura sobre las variables globales.

3. Acceso no atómico a variables:

Actualización de múltiples miembros de una estructura o actualizar una variable que es mayor que el tamaño de la palabra natural de la arquitectura (por ejemplo, la actualización de una variable de 32 bits en una máquina de 16 bits) son ejemplos de operaciones no atómicas. Si son interrumpidas pueden provocar la pérdida de datos o la corrupción de los mismos.

4. Función reentrante:

Una función es reentrante si es seguro llamar a la función desde más de una tarea, o de tareas e interrupciones.

Cada tarea mantiene su propia pila y su propio conjunto de valores de los registros centrales. Si una función no tiene acceso a datos distintos de los que se le asignan a la pila, o que se llevan a cabo en un registro entonces la función es reentrante. El Listado 58 es un ejemplo de una función reentrante. El Listado 59 es un ejemplo de una función que no es reentrante.

/ Un parámetro se pasa a la función. Este será o bien pasado en la pila o en un registro de la CPU. De cualquier manera, es seguro dado que cada tarea mantiene su propia pila y su propio conjunto de registro de valores. */*

`long lAddOneHundered(long lVar1)`

`{`

/ Esta variable de la función también se destinará a la pila o a un registro, dependiendo del compilador y del nivel de optimización. Cada tarea o interrupción que llama a esta función tendrán su propia copia de lVar2. */*

`long lVar2;`

`lVar2 = lVar1 + 100;`

/ Lo más probable es que el valor de retorno se coloque en un registro del CPU, aunque también podría ser colocado en la pila. */*

`return lVar2;`

`}`

Listado 58 – Ejemplo de Función Reentrante.

```
/* En este caso lVar1 es una variable global, por lo tanto, cada tarea que llama a la función se accede a la
misma copia de la variable. */
long lVar1;

long lNonsenseFunction( void )
{
    /* Esta variable es estática por lo que no se asigna en la pila. Cada tarea que llama a la función accede a la
    misma copia de la variable. */
    static long lState = 0;
    long lReturn;

    switch( lState )
    {
        case 0 : lReturn = lVar1 + 10;
                lState = 1;
                break;

        case 1 : lReturn = lVar1 + 20;
                lState = 0;
                break;
    }
}
```

Listado 59 – Ejemplo de una Función No Reentrante.

Exclusión mutua:

El acceso a un recurso que se comparte, ya sea entre las tareas o entre tareas e interrupciones, debe gestionarse utilizando una técnica de 'exclusión mutua' para asegurar que la consistencia de datos se mantiene en todo momento. El objetivo es asegurar que una vez que una tarea comienza a acceder a un recurso compartido, esta tarea tiene acceso exclusivo hasta que el recurso ha sido devuelto.

FreeRTOS ofrece varias características que pueden ser usadas para aplicar la exclusión mutua, pero el mejor método es (siempre que sea posible) diseñar la aplicación de tal manera que los recursos no se compartan y cada recurso únicamente se acceda desde una sola tarea.

Alcance:

Este capítulo tiene como objetivo dar a los lectores una buena comprensión de:

- Cuándo y por qué es necesaria la gestión y control de los recursos.
- Qué es una sección crítica.
- Qué significa la exclusión mutua.
- Qué significa suspender el scheduler.
- Cómo utilizar un mutex.
- Cómo crear y utilizar un guardián de tarea.

- Qué es la inversión de prioridades y cómo la herencia de prioridad puede reducir (pero no eliminar) su impacto.

4.2 Secciones críticas y suspensión del scheduler:

Secciones Críticas Básicas:

Las secciones críticas básicas son regiones de código que están rodeadas por llamados a las macros `taskENTER_CRITICAL()` y `taskEXIT_CRITICAL()` respectivamente, como se demuestra en el Listado 60. Las secciones críticas también se conocen como regiones críticas.

```
/* Garantiza que el acceso al registro PORTA no sea interrumpido, colocándolo dentro de una sección crítica. */
```

```
taskENTER_CRITICAL();
```

```
/* Un cambio a otra tarea no puede ocurrir entre la llamada a taskENTER_CRITICAL () y la llamada a taskEXIT_CRITICAL (). Las interrupciones aún pueden ejecutar en los puertos FreeRTOS que permiten la anidación de interrupciones, pero sólo interrupciones cuya prioridad está por encima del valor asignado a la constante configMAX_SYSCALL_INTERRUPT_PRIORITY (y a esas interrupciones no se les permite llamar a funciones API de FreeRTOS). */
```

```
PORTA |= 0x01;
```

```
/* Hemos terminado de acceder PORTA por lo que puede salir de la sección crítica de forma segura. */
```

```
/taskEXIT_CRITICAL();
```

Listado 60 – Uso de una sección crítica para resguardar el acceso a un registro.

Los proyectos de ejemplo que acompañan a este libro usan una función llamada `vPrintString()` para escribir cadenas a la salida estándar (que es la ventana de terminal para ejecutables Open Watcom DOS). `vPrintString()` se llama de muchas tareas diferentes por lo que, en teoría, su implementación podría proteger el acceso a la salida estándar utilizando una sección crítica, como se muestra en el Listado 61.

```
void vPrintString( const portCHAR *pcString )
{
    /* Escribir la cadena en la salida estándar, utilizando una sección crítica como un método crudo de exclusión mutua. */
    taskENTER_CRITICAL();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    taskEXIT_CRITICAL();
}
```

```
/* Permite que cualquier tecla detenga la ejecución de la aplicación. Una aplicación real
que utiliza un valor de teclado debe proteger el acceso a este también. */
if( kbhit() )
{
    vTaskEndScheduler();
}
}
```

Listado 61 – Posible implementación de la función vPrintString().

Las secciones críticas realizadas de esta manera son un método muy crudo de proporcionar la exclusión mutua. Trabajan simplemente mediante la desactivación de las interrupciones o bien completamente, o hasta la prioridad de interrupción establecido por configMAX_SYSCALL_INTERRUPT_PRIORITY (dependiendo del puerto FreeRTOS siendo utilizado).

Sólo pueden ocurrir cambios de contexto dentro de una interrupción, así que mientras las interrupciones permanecen inhabilitadas la tarea que llama taskENTER_CRITICAL () está garantizada que permanezca en el estado de ejecución hasta que se salga de la sección crítica.

Las secciones críticas deben ser muy cortas, de lo contrario afectarán negativamente a los tiempos de respuesta de interrupción. Cada llamado a taskENTER_CRITICAL () debe estar estrechamente vinculado con una llamada a taskEXIT_CRITICAL (). Por esta razón la salida estándar no debe ser protegida mediante una sección crítica (como se muestra en el Listado 61) porque la escritura en el terminal puede ser una operación relativamente larga. También la forma en que el emulador de DOS y Open Watcom manejan la salida del terminal no es compatible con esta forma de exclusión mutua, dado que las llamadas de librerías dejan interrupciones habilitadas. Los ejemplos de este capítulo exploran soluciones alternativas.

Es seguro para las secciones críticas convertirse en anidadas porque el kernel mantiene una cuenta de la profundidad de anidamiento. La sección crítica sólo se abandona cuando la profundidad de anidación vuelve a cero, que es cuando una llamada a taskEXIT_CRITICAL () se ha ejecutado para cada llamada anterior para taskENTER_CRITICAL ().

Suspensión (o bloqueo) del Programador:

Las secciones críticas también se pueden crear mediante la suspensión del scheduler. Suspender el scheduler también se conoce como bloquear” el scheduler.

Las secciones críticas básicas protegen una región de código de acceso de otras tareas e interrupciones. Una sección crítica implementada suspendiendo el programador sólo protege una región de código de acceso de otras tareas, porque las interrupciones permanecen habilitadas.

Una sección crítica que es demasiado larga para ser implementada simplemente deshabilitando las interrupciones, puede implementarse mediante la suspensión del scheduler, pero la reanudación de mismo puede resultar ser una operación relativamente larga, por lo que se debe considerar en cada caso particular cuál es el mejor método para su uso.

Función API vTaskSuspendAll():

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

Listado 62 – Prototipo de la Función API vTaskSuspendAll().

El scheduler se suspende llamando a la función vTaskSuspendAll (). Suspender el scheduler evita que ocurra un cambio de contexto, pero deja las interrupciones habilitadas. Si una interrupción solicita un

cambio de contexto mientras que el scheduler está suspendido, la solicitud se mantiene en espera y sólo se realiza cuando el scheduler se reanuda (deja de estar suspendido).

Las Funciones API de FreeRTOS no deberían ser llamadas mientras el scheduler está suspendido.

Función API xTaskResumeAll():

portBASE_TYPE xTaskSuspendAll (void);

Listado 63 – Prototipo de la Función API xTaskResumeAll().

El scheduler reanuda su funcionamiento (deja de estar suspendido) con un llamado a la función xTaskResumeAll().

Tabla 18 Valor de retorno de la función xTaskResumeAll().

Valor de Retorno	Descripción
Valor de Retorno	Los cambios de contexto que se solicitaron mientras el scheduler fue suspendido se mantienen en espera y sólo se llevan a cabo cuando el scheduler se reanuda. Un cambio de contexto en espera, anterior a xTaskResumeAll () devuelve pdTRUE. En todos los demás casos xTaskResumeAll () devolverá pdFALSE.

Es seguro para las llamadas a vTaskSuspendAll () y xTaskResumeAll () convertirse en anidada porque el kernel mantiene una cuenta de la profundidad de anidamiento. El scheduler sólo se reanudará cuando la profundidad de anidación vuelve a cero, que es cuando una llamada a xTaskResumeAll () se ha ejecutado para cada llamada anterior para vTaskSuspendAll ().

El Listado 64 muestra la aplicación real de vPrintString (), que suspende el scheduler para proteger el acceso a la salida de la terminal.

```
void vPrintString( const portCHAR *pcString )
{
    /* Escribir la cadena en la salida estándar, suspendiendo el scheduler como método de exclusión mutua. */
    vTaskSuspendScheduler();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    xTaskResumeScheduler();

    /* Permite que cualquier tecla detenga la aplicación en ejecución. Una aplicación real que utiliza un valor de teclado debe proteger el acceso a este también. */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

```
    }  
}
```

Listado 64 – Implementación de vPrintString().

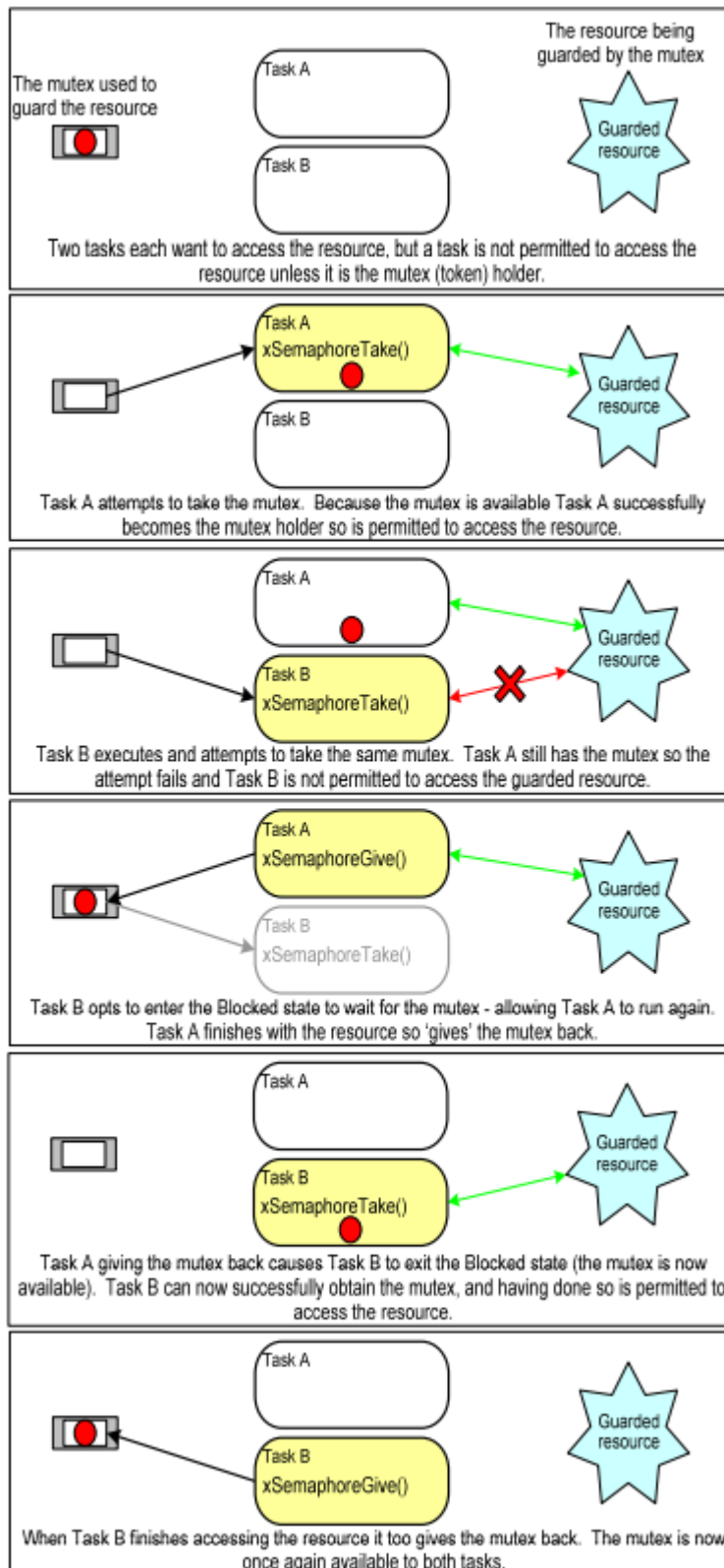
4.3 Semáforos Mutex (y binarios):

Un mutex es un tipo especial de semáforo binario que se utiliza para controlar el acceso a un recurso que se comparte entre dos o más tareas. La palabra MUTEX proviene de "exclusión mutua".

Cuando se usa en un escenario de exclusión mutua, el mutex puede ser conceptualmente pensado como un "token" que está asociado con el recurso que se comparte. Para que una tarea acceda legítimamente al recurso, debe primero tomar el token. Cuando ha terminado de usar el recurso, debe devolver el token. Sólo cuando el token se ha vuelto otra tarea puede tomarlo, y luego acceder de forma segura al mismo recurso compartido. No está permitido que una tarea acceda al recurso compartido a menos que esta posea el token. Este mecanismo se muestra en la Figura 36.

A pesar de que los mutex y los semáforos binarios comparten muchas características, el escenario mostrado en la Figura 36 (donde se utiliza un mutex para la exclusión mutua) es completamente diferente a la mostrada en la Figura 30 (donde se utiliza un semáforo binario para la sincronización). La principal diferencia es lo que ocurre con el semáforo después de que se ha obtenido:

- Un semáforo que se usa para exclusión mutua debe ser devuelto siempre.
- Un semáforo que se usa para sincronizar tareas normalmente es descartado y no se devuelve.



El mutex es usado para resguardar el recurso. Dos tareas quieren acceder al recurso, pero no pueden hacerlo a menos que tengan el mutex (token).

La tarea A intenta obtener el mutex, como este estaba disponible, logra obtenerlo y por lo tanto puede acceder al recurso.

La tarea B se ejecuta e intenta obtener el mismo mutex. La tarea A aún tiene el mutex, por lo que la tarea B no logra obtenerlo.

La tarea B se bloquea, esperando a que el mutex esté disponible. La tarea A vuelve a ejecutarse y termina de usar el recurso. Luego de esto devuelve el mutex.

Cuando la tarea A devuelve el mutex, la tarea B sale del estado bloqueado, toma el mutex y ahora si puede acceder al recurso.

Cuando la tarea B termina de usar el recurso devuelve el mutex, quedando nuevamente disponible para las dos tareas.

Figura 36 – Exclusión mutua implementada usando un mutex.

El mecanismo funciona exclusivamente a través de la disciplina del diseñador de la aplicación. No hay razón para que una tarea no pueda acceder al recurso en cualquier momento, pero cada tarea "acuerda" no acceder si previamente no pudo obtener el mutex.

Función API xSemaphoreCreateMutex():

Un mutex es un tipo de semáforo. Los manipuladores de todos los tipos de semáforos de FreeRTOS se almacenan en una variable de tipo xSemaphoreHandle. Antes que un mutex se pueda utilizar, en primer lugar debe ser creado. Para crear un semáforo tipo mutex se debe utilizar la función API xSemaphoreCreateMutex().

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

Listado 65 – Prototipo de la función xSemaphoreCreateMutex().

Tabla 19 Valor de retorno de la función xSemaphoreCreate().

Parámetro/ Valor de Retorno	Descripción
Valor de Retorno	Si se devuelve NULL, el mutex no se pudo crear porque la memoria disponible no era suficiente. Un valor de retorno distinto de NULL indica que el mutex se ha creado correctamente. El valor devuelto debe ser almacenado como el manipulador al mutex creado.

Ejemplo 15. Rescritura de vPrintString() para utilizar un semáforo.

En este ejemplo se crea una nueva versión de vPrintString () llamada prvNewPrintString (), luego llama a la nueva función desde múltiples tareas. prvNewPrintString () tiene una funcionalidad idéntica a vPrintString () pero utiliza un mutex para controlar el acceso a la salida estándar en lugar de una sección crítica básica. La implementación de prvNewPrintString () se muestra en el Listado 66.

```
static void prvNewPrintString( const portCHAR *pcString )
{
    /* El mutex se crea antes de iniciar el scheduler de modo que ya existe en el momento en
    esta tarea se ejecuta por primera vez.

    Intenta tomar el mutex, si falla se bloquea de forma indefinida, esperando a que esté dis-
    ponible. El llamado a xSemaphoreTake call () sólo retornará cuando el mutex se haya obte-
    nido con éxito, por lo que no hay necesidad de comprobar el valor de retorno de la fun-
    ción. Si se ha utilizado cualquier otro periodo de retardo, entonces el código debe compro-
    bar que xSemaphoreTake() devuelve pdTRUE antes de acceder al recurso compartido (que
    en este caso es la salida estándar). */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* La siguiente línea sólo se ejecutará una vez que el mutex se ha obtenido exito-
        samente. Ahora la salida estándar puede ser accedida libremente, y tan sólo una
        tarea puede tener el mutex en cualquier momento dado. */
        printf( "%s", pcString );
        fflush( stdout );

        /* El mutex DEBE ser devuelto.*/
    }
    xSemaphoreGive( xMutex );

    /* Permite que al presionar cualquier tecla se detenga la aplicación en ejecución. Una apli-
    cación real que utiliza un valor ingresado por teclado para detenerse debe proteger el ac-
    ceso a este. */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

Listado 66 – Implementación de prvNewPrintString().

prvNewPrintString () se llama repetidamente por dos instancias de una tarea llamada prvPrintTask(). Un tiempo de retardo aleatorio se utiliza entre cada llamada. El parámetro de tarea se utiliza para pasar una cadena única en cada instancia de la tarea. La implementación de prvPrintTask () se muestra en el Listado 67.

```
static void prvPrintTask( void *pvParameters )
{
    char *pcStringToPrint;
    /* Dos instancias de esta tarea se crean, por lo que la cadena que la tarea enviará a
    prvNewPrintString () se pasa a la tarea utilizando el parámetro tarea.
    Castear esto al tipo requerido. */
    pcStringToPrint = ( char * ) pvParameters;

    for( ;; )
    {
        /* Imprime la cadena mediante la función que acaba de definir. */
        prvNewPrintString( pcStringToPrint );

        /* Espera un momento pseudo aleatorio. Tenga en cuenta que rand() no es nece-
        sariamente reentrante, pero en este caso, en realidad no importa, ya que al código
        no le importa qué valor se devuelve. En una aplicación más segura una versión
        de rand() que se sabe que es reentrante debe ser usada. */
        vTaskDelay( ( rand() & 0x1FF ) );
    }
}
```

Listado 67 – Implementación de prvPrintTask() para el Ejemplo 15.

Como es normal, el main() simplemente crea el mutex, crea las tareas, y luego comienza el scheduler. La aplicación se muestra en el Listado 68.

Las dos instancias de prvPrintTask () se crean en diferentes prioridades por lo que la tarea de menor prioridad a veces es adelantada por la tarea de mayor prioridad. Como se utiliza un mutex para asegurar que cada tarea tiene acceso mutuamente excluyente al terminal, incluso cuando una tarea es de mayor prioridad que la otra, las cadenas que se muestran serán correctas.

```

int main( void )
{
    /* Antes de utilizar un semáforo, este debe ser explícitamente creado. En este ejemplo se
    crea un semáforo tipo mutex. */
    xMutex = xSemaphoreCreateMutex();

    /* Las tareas van a utilizar un retraso pseudo aleatorio, se preselecciona el generador de
    números aleatorios. */
    srand( 567 );

    /* Verifica que el semáforo se ha creado correctamente antes de crear las tareas. */
    if( xMutex != NULL )
    {
        /* Se crean dos instancias de las tareas que escriben en la salida estándar. La ca-
        dena que escriben se pasa como parámetro de tarea. Las tareas se crean en dife-
        rentes prioridades para que ocurran algunos adelantos de tareas. */
        xTaskCreate( prvPrintTask, "Print1", 1000,
                    "Task 1 *****\r\n", 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 1000,
                    "Task 2 ----- \r\n", 2, NULL );

        /* Iniciar el programador para las tareas creadas comiencen a ejecutarse. */
        vTaskStartScheduler();
    }

    /*Si todo anda bien, el main() nunca debería alcanzar este punto, dado que el Scheduler va
    a estar siempre ejecutando las tareas. Si el main() alcanza este punto, entonces el probable
    que la memoria disponible no era suficiente para crear la tarea ociosa. El capítulo 5 provee
    más información sobre el manejo de memoria. */
    for( ;; );
}

```

Listado 68 – Implementación del main() para el Ejemplo 15.

La salida producida cuando se ejecuta el Ejemplo 15 se muestra en la figura 37. Una posible se-
cuencia de ejecución se muestra en la Figura 38.

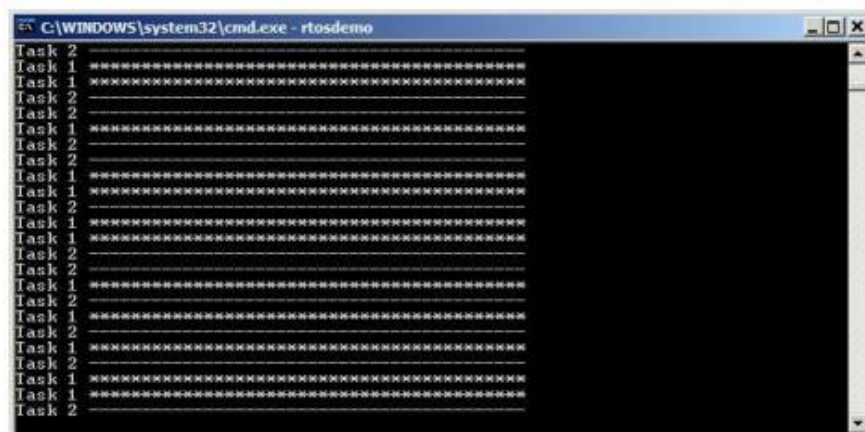


Figura 37 – Salida producida cuando se ejecuta el Ejemplo 15.

La figura 37 muestra que, como era de esperar, no hay corrupción en las cadenas que se muestran en el terminal. El orden aleatorio es un resultado de los periodos de retardo aleatorios usados por las tareas.

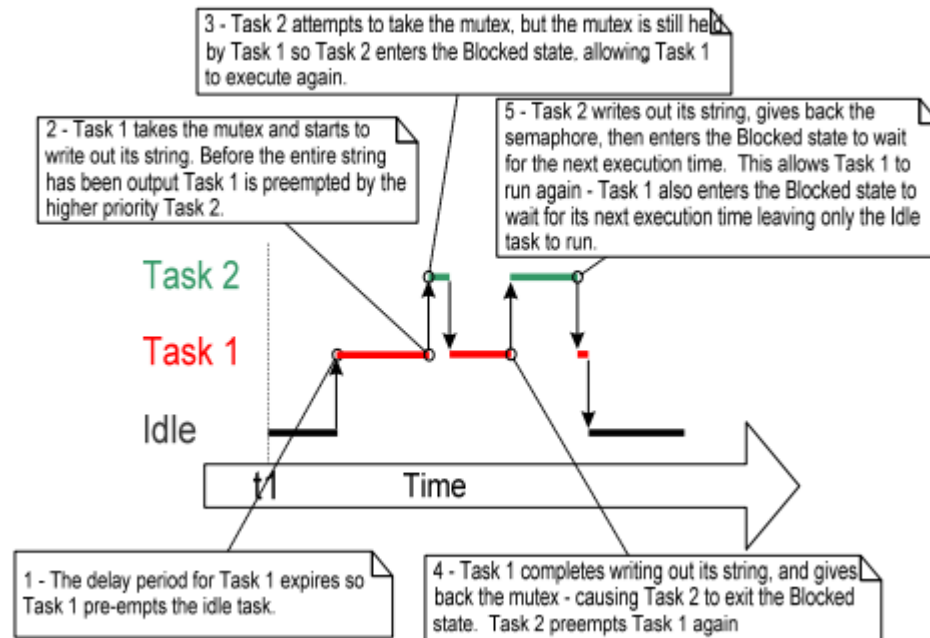


Figura 38 – Una posible secuencia de ejecución producida por el Ejemplo 15.

- 1- El periodo de retardo para la tarea 1 expira, entonces la tarea 1 se adelanta a la tarea ociosa y comienza a ejecutarse.
- 2- La tarea 1 toma el mutex y comienza a escribir la cadena. Antes que toda la cadena fuese escrita por completo, la tarea 2 adelanta a la tarea 1.
- 3- La tarea 2 intenta tomar el mutex pero no lo logra (porque ya lo tiene la tarea 1) entonces se bloquea.
- 4- La tarea 1 se ejecuta nuevamente y termina de escribir la cadena. Devuelve el mutex causando que la tarea 2 salga del estado bloqueado, y adelante nuevamente a la tarea 1.
- 5- La tarea 2 escribe su cadena y devuelve el semáforo, luego se bloquea esperando su próxima ejecución. Esto permite que la tarea 1 se ejecute de nuevo. La tarea 1 también se bloquea esperando a su próxima ejecución, dejando que se ejecute la tarea ociosa nuevamente.

Inversión de Prioridad:

La figura 38 muestra uno de los peligros potenciales de la utilización de un mutex para proporcionar la exclusión mutua. La posible secuencia de ejecución representada muestra la Tarea 2, de mayor prioridad, teniendo que esperar a que la Tarea 1, de menor prioridad, a que le ceda el control del mutex. Una tarea de mayor prioridad que se retrasó por una tarea de menor prioridad de esta forma, se llama "inversión de prio-

ridades". Este comportamiento no deseable sería exagerado aún más si una tarea de prioridad media comenzó a ejecutar mientras que la tarea de alta prioridad estaba esperando el semáforo (el resultado sería una tarea de alta prioridad a la espera de una tarea de baja prioridad, y sin la tarea de baja prioridad pudiendo ser capaz de ejecutar y devolver el mutex. Este es el peor de los casos, y se muestra en la Figura 39.

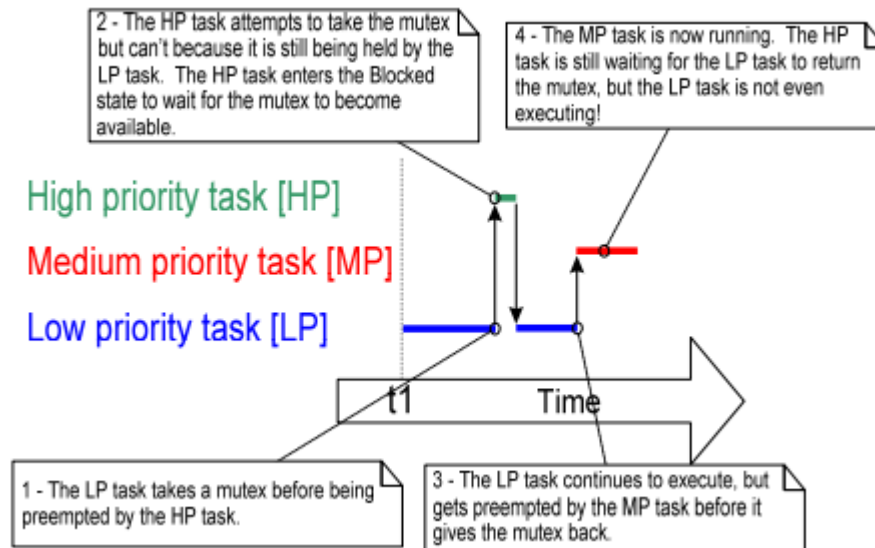


Figura 39 – Caso de inversión de prioridades.

Inversión de prioridades puede ser un problema importante, pero en un sistema embebido pequeño a menudo puede ser evitado considerando cómo se accede a los recursos en tiempo de diseño del sistema.

Herencia de Prioridades:

Los semáforos Mutex y los semáforos binarios de FreeRTOS son muy similares, la única diferencia es que los mutex proporcionan automáticamente un mecanismo básico de "herencia de prioridad". La herencia de prioridad es un esquema que minimiza los efectos negativos de la inversión de prioridades (no arregla la inversión de prioridades, simplemente disminuye su impacto). La herencia de prioridad hace que el análisis matemático del comportamiento del sistema sea un ejercicio más complejo, por lo que no se recomienda depender de la herencia de prioridad si se puede evitar.

La herencia de prioridad consiste en elevar temporalmente la prioridad de la tarea que posee el mutex, al mismo nivel de prioridad de la tarea de más alta prioridad, que está tratando de obtener el mismo mutex. La tarea de baja prioridad que tiene el mutex "hereda" la prioridad de la tarea que está esperando poder tomar el mutex. Esto se demuestra en la Figura 40. La prioridad de la tarea que tenía el mutex se restablece automáticamente a su valor original cuando ésta ya lo devolvió.

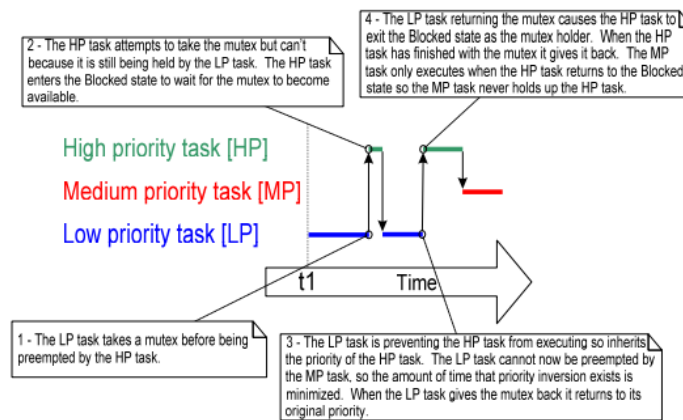


Figura 40 – Herencia de prioridad minimizando el efecto de la inversión de prioridades.

- 1- La tarea de baja prioridad toma el mutex y luego es adelantada por una tarea de alta prioridad.
- 2- La tarea de alta prioridad intenta tomar el mutex pero no puede, porque lo tiene la tarea de baja prioridad. Entonces se bloquea y espera a que el mutex esté disponible.
- 3- La tarea de baja prioridad está impidiendo que la tarea de alta prioridad se ejecute, entonces, hereda la misma prioridad que la tarea de alta prioridad para evitar que la tarea de prioridad media pueda adelantarse. Con esto se logra minimizar el tiempo en el que existe la inversión de prioridad. Cuando la tarea de baja prioridad devuelve el mutex, esta recupera su prioridad original.
- 4- Cuando la tarea de baja prioridad devuelve el mutex, este es tomado por la tarea de alta prioridad y abandona el estado bloqueado. Cuando la tarea de alta prioridad termina de usar el mutex lo devuelve. La tarea de prioridad media solo se ejecutará cuando la tarea de alta prioridad pase a estar bloqueada nuevamente.

Debido a que la preferencia es para evitar la inversión de prioridades, en primer lugar, y porque FreeRTOS está dirigido a limitaciones de memoria de los microcontroladores, el mecanismo de la herencia de prioridad implementado por mutex es sólo una forma básica que asume que una tarea sólo llevará a cabo una sola exclusión mutua en un momento dado.

Punto muerto:

Un punto muerto es otro problema potencial del uso de semáforos mutex para la exclusión mutua. Se produce cuando dos tareas no pueden continuar porque ambos están a la espera de un recurso que está en manos de la otra. Considere el siguiente escenario donde la Tarea A y B ambas necesitan adquirir el mutex X y el mutex Y, con el fin de realizar una acción:

- 1- La tarea A se ejecuta y toma el mutex X.
- 2- La tarea B adelanta a la tarea A.
- 3- La tarea B toma el mutex Y, y luego intenta tomar el mutex X, pero no lo logra dado que lo tiene la tarea A. La tarea B entonces se bloquea, esperando a que el mutex X esté disponible.
- 4- La tarea A vuelve a ejecutarse e intenta tomar el mutex Y, pero no lo logra dado que la tarea B tiene tomado este mutex. Entonces la tarea A también se bloquea, esperando a que el mutex Y esté disponible.

Al final de este escenario, la Tarea A está esperando al mutex tomado por la Tarea B, y Tarea B está esperando a un mutex retenido por la Tarea A. Se produce entonces un interbloqueo, porque ni tarea puede proceder más allá.

Al igual que con la inversión de prioridades, el mejor método para evitar esto es considerar su potencial existencia en el momento del diseño, y diseñar el sistema para que simplemente esto no pueda ocurrir. En la práctica, los puntos muertos no son un gran problema para los sistemas embebidos pequeños, porque los diseñadores del sistema pueden tener una buena comprensión de toda la aplicación y así identificar y eliminar las zonas donde podría llegar a ocurrir.

4.4 Tareas guardianas:

Las tareas guardianas proporcionan un método limpio de implementación de la exclusión mutua sin la preocupación de la inversión de prioridad o punto muerto.

Una tarea guardiana es una tarea que tiene la propiedad exclusiva de un recurso. Sólo la tarea guardiana tiene permitido el acceso al recurso, cualquier otra tarea que tenga la necesidad de acceder a este recurso, sólo pueden hacerlo indirectamente mediante el uso de los servicios del guardián.

Ejemplo 16. Reescritura de `vPrintString()` para usar una tarea guardiana:

El ejemplo 16 también proporciona una implementación alternativa para `vPrintString()`, esta vez una tarea guardiana se utiliza para administrar el acceso a la salida estándar. Cuando una tarea quiere escribir un mensaje en el terminal, no llama a una función de impresión directa, sino que le envía el mensaje a la tarea guardiana.

Esta tarea utiliza una cola FreeRTOS para serializar el acceso a la terminal. La implementación interna de la tarea no necesita considerar la exclusión mutua, ya que es la única tarea permitida para acceder al terminal directamente.

La tarea guardiana pasa la mayor parte de su tiempo en el estado bloqueado a la espera de mensajes para llegar a la cola. Cuando llega un mensaje simplemente lo escribe en la salida estándar antes de regresar al estado bloqueado para esperar el siguiente mensaje. La implementación de la tarea guardiana se muestra en el Listado 70.

Las interrupciones pueden enviar datos a las colas, por lo que también pueden utilizar con seguridad los servicios para escribir mensajes al terminal que ofrece la tarea guardiana. En este ejemplo, una función de enlace tick se utiliza para escribir un mensaje cada 200 ticks.

Un enlace tick es una función que se llama por el núcleo durante cada interrupción tick. Para utilizar una función de enlace tick:

- Setear `configUSE_TICK_HOOK` en 1 en `FreeRTOSConfig.h`
- Proporcionar la implementación de la función de enlace, utilizando el nombre de la función exacta y el prototipo mostrado en el Listado 69.


```
void vApplicationTickHook( void );
```

Listado 69 – Nombre y prototipo para la función de enlace tick.

Las funciones de enlace tick se ejecutan en el contexto de las interrupciones tick, por lo que deben ser mantenidas muy cortas, sólo utilizan una cantidad moderada de espacio de pila, y no llaman a las funciones API de FreeRTOS cuyo nombre no termina con 'FromISR ()'.

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
    char *pcMessageToPrint;
    /* Esta es la única tarea que puede escribir en el terminal de salida. Cualquier otra tarea
    que quiera escribir una cadena no podrá acceder al terminal directamente, sino que tendrá
    que enviarle la cadena esta tarea. Como sólo esta tarea accede a la salida estándar, no hay
    exclusión o problemas mutuos a tener en cuenta dentro de la aplicación de la tarea en sí.*/
    for( ;; )
    {
        /* Espera a que un mensaje llegue. Se especifica un tiempo de bloqueo indefini-
        do, por lo que no hay necesidad de comprobar el valor de retorno. */
        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

        /* Imprime la cadena recibida. */
        printf( "%s", pcMessageToPrint );
        fflush( stdout );
        /* Ahora sólo tiene que esperar al siguiente mensaje. */
    }
}
```

Listado 70 – La tarea guardiana.

La tarea que imprime el mensaje es similar a la utilizada en el Ejemplo 15, excepto que esta vez la cadena se envía en la cola a la tarea guardiana, en lugar de imprimirse directamente. La implementación se muestra en el Listado 71. Al igual que antes, se crean dos instancias independientes de la tarea, cada una de las cuales imprime una cadena única que se le pasa a través del parámetro de tareas.

```
static void prvPrintTask( void *pvParameters )
{
    int iIndexToString;
    /* Se crean dos instancias de esta tarea. El parámetro de tarea se utiliza para pasar un ín-
    dice en un vector de cadenas en la tarea. Castear esto al tipo requerido. */
    iIndexToString = ( int ) pvParameters;
    for( ;; )
    {
        /* Imprime la cadena, no directamente, sino en lugar haciendo pasar un puntero
        a la cadena a la tarea guardiana, a través de una cola. Se crea la cola antes que se
```

```

        inicie el scheduler, así ya existe en el momento en que esta tarea se ejecuta por
        primera vez. Un tiempo de bloqueo no está especificado, porque siempre debe
        haber espacio en la cola. */
        xQueueSendToBack( xPrintQueue, &(amp; pcStringsToPrint[ iIndexToString ] ), 0 );
        /* Espera un momento pseudo aleatorio. Tenga en cuenta que rand() no es nece-
        sariamente por reentrante, pero en este caso, en realidad no importa ya que el
        código no le importa qué valor se devuelve. En una aplicación más segura, una
        versión de rand() que se sabe que si es reentrante debe ser usado, o las llamadas
        a rand() deben ser protegidas mediante una sección crítica. */
        vTaskDelay( ( rand() & 0x1FF ) );
    }
}

```

Listado 71 – Implementación de la tarea de impresión para el Ejemplo 16.

La función de enlace tick simplemente cuenta el número de veces que se llama, enviando su mensaje a la tarea guardiana cada vez que la cuenta llegue a 200. Sólo para fines de demostración, el enlace de tick escribe en el frente de la cola y de las tareas de impresión escriben en la parte de atrás de la cola. La implementación del enlace de tick se muestra en el Listado 72.

```

void vApplicationTickHook( void )
{
    static int iCount = 0;
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* Imprime un mensaje cada 200 ticks. El mensaje no es escrito directamente, sino que es
    enviado a la tarea guardiana. */
    iCount++;
    if( iCount >= 200 )
    {
        /* En este caso el último parámetro (xHigherPriorityTaskWoken) no se utiliza, pe-
        ro debe ser suministrado. */
        xQueueSendToFrontFromISR (    xPrintQueue,
                                      &( pcStringsToPrint[ 2 ] ),
                                      &xHigherPriorityTaskWoken );

        /* Restablece la cuenta, para imprimir la cadena de nuevo en 200 ticks. */
        iCount = 0;
    }
}

```

Listado 72 – Implementación del enlace tick.

Como es normal, el main () crea las colas y las tareas necesarias para ejecutar el ejemplo, luego se inicia el scheduler. La ejecución de main () se muestra en el Listado 73.

```
/* Define las cadenas que las tareas y las interrupciones imprimirán a través de la tarea
guardiana. */
static char *pcStringsToPrint[] =
{
"Task 1 *****\r\n",
"Task 2 ----- \r\n",
"Message printed from the tick hook interrupt #####\r\n"
};
/*-----*/
/* Declara una variable de tipo xQueueHandle. Esto se utiliza para enviar mensajes desde las tareas
de impresión y la interrupción de tick, a la tarea guardiana. */
xQueueHandle xPrintQueue;

/*-----*/
int main( void )
{
    /* Antes de usar una cola, se debe crear de forma explícita. Se crea la cola para contener
    un máximo de 5 punteros de caracteres. */
    xPrintQueue = xQueueCreate( 5, sizeof( char * ));

    /* Las tareas van a utilizar un retraso pseudo aleatorio. */
    srand( 567 );

    /* Verifica que la cola se ha creado correctamente. */
    if( xPrintQueue != NULL )
    {
        /* Se crean dos instancias de las tareas que envían mensajes al guardián. El índice
        de la cadena que la tarea utiliza se pasa a través del parámetro de tareas (cuarto
        parámetro para xTaskCreate ()). Las tareas se crean en diferentes prioridades por
        lo que la tarea de mayor prioridad se adelantará ocasionalmente a la tarea de
        menor prioridad. */
        xTaskCreate( prvPrintTask, "Print1", 1000, ( void * ) 0, 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

        /* Se crea la tarea guardiana. Esta es la única tarea que tiene permitido acceder
        directamente a la salida estándar. */
        xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL, 0, NULL );
        /* Se inicia el scheduler para que las tareas comiencen a ejecutarse. */
        vTaskStartScheduler();
    }

    /* Si todo anda bien, el main() nunca debería alcanzar este punto, dado que el Scheduler va
    a estar siempre ejecutando las tareas. Si el main() alcanza este punto, entonces es probable
    que la memoria disponible no era suficiente para crear la tarea ociosa. El capítulo 5 provee
    más información sobre el manejo de memoria. */
    for( ;; );
}
```

}

Listado 73 – Implementación del main() para el Ejemplo 16.

La salida producida cuando el Ejemplo 16 se ejecuta, se muestra en la Figura 41. Como se puede ver, las cadenas procedentes de las tareas y las cadenas procedentes de la interrupción, son todas impresas correctamente, sin errores.

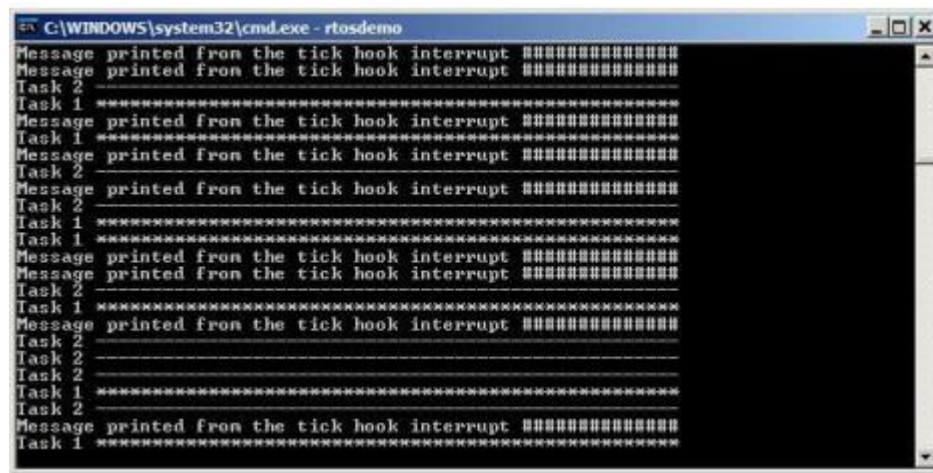


Figura 41 – Salida producida cuando el Ejemplo 16 es ejecutado.

A la tarea guardiana se le asignó una prioridad más baja que las tareas de impresión, por lo que los mensajes enviados al guardián permanecieron en la cola hasta que ambas tareas de impresión se bloquearon. En algunas situaciones, sería apropiado asignarle al guardián una mayor prioridad, para que los mensajes se procesan más rápido, pero hacerlo sería a costa de retrasar las tareas de menor prioridad hasta que hubiera completado el acceso al recurso protegido.

Capítulo 5: Manejo de Memoria

5. 1 Introducción y Alcance de este capítulo:

El kernel tiene que asignar dinámicamente la memoria RAM cada vez que se crea una tarea, cola o semáforos. Las funciones de librería malloc() y free() estándares pueden usarse, pero también pueden sufrir de uno o más de los siguientes problemas:

1. No están siempre disponibles en los pequeños sistemas embebidos.
2. Su implementación puede resultar relativamente extensa, y ocupar demasiado espacio.
3. Rara vez son seguras.
4. No son determinísticas. La cantidad de tiempo que toma ejecutar las funciones varía entre un llamado y otro.
5. Pueden sufrir fragmentación de memoria.
6. Pueden complicar la vinculación.

Diferentes sistemas embebidos tienen variaciones de asignación de memoria RAM y distintos requerimientos temporales, por lo que un solo algoritmo de asignación de memoria RAM será solamente el apropiado para un subconjunto de aplicaciones. Por lo tanto, FreeRTOS trata la asignación de memoria como parte de la capa portátil (en oposición a una parte de la base de código del núcleo). Esto permite a las aplicaciones individuales para proporcionar su propia implementación específica cuando sea apropiado.

Cuando el kernel requiere memoria RAM, en lugar de llamar a la función malloc(), directamente llama la función pvPortMalloc(). Cuando la memoria RAM se está liberando, en lugar de llamar a la función free() directamente el kernel llama a la función vPortFree(). pvPortMalloc() tiene el mismo prototipo que malloc(), y vPortFree() tiene el mismo prototipo como free().

FreeRTOS viene con tres ejemplos de implementaciones de pvPortMalloc () y vPortFree (), los cuales están documentados en este capítulo. Los usuarios de FreeRTOS pueden utilizar uno de los ejemplos de implementaciones, o proporcionar uno propio.

Los tres ejemplos se definen en los archivos heap_1.c, heap_2.c y heap_3.c respectivamente, los cuales se encuentran en el directorio FreeRTOS\Source\portable\MemMang. El esquema de agrupación de memoria y la asignación de bloque original utilizado por versiones muy tempranas de FreeRTOS ha sido eliminados por el esfuerzo y la comprensión que era necesaria para dimensionar los bloques y las agrupaciones.

Es común que los pequeños sistemas embebidos solo creen las tareas, las colas y los semáforos antes de iniciar el scheduler. Cuando este es el caso, la memoria es asignada dinámicamente antes que la aplicación comience a realizar cualquier funcionalidad verdadera en tiempo real, y la memoria una vez asignada, nunca es liberada. Esto significa que el esquema de asignación elegido no tiene que tener en cuenta ninguna de las cuestiones más complejas como el determinismo y la fragmentación, y puede en cambio sólo tener en cuenta los atributos tales como el tamaño del código y la simplicidad.

Alcance:

Este capítulo tiene como objetivo dar a los lectores una buena comprensión de:

- Cuando FreeRTOS asigna RAM.
- Los tres ejemplos de esquemas de asignación de memoria suministrados con FreeRTOS.

5. 2 Ejemplo de esquemas de asignación de memoria:

Heap_1.c

Heap_1.c implementa una versión muy básica de `pvPortMalloc()` y no implementa `vPortFree()`. Cualquier aplicación que nunca borra una tarea, cola o semáforo, tiene el potencial de utilizar heap_1. Heap_1 siempre es determinista.

El esquema de asignación simplemente subdivide una matriz sencilla en bloques más pequeños como llamadas a `pvPortMalloc()` se hacen. La matriz es el heap de FreeRTOS.

El tamaño total (en bytes) de la matriz es fijado por la definición de `configTOTAL_HEAP_SIZE` dentro `FreeRTOSConfig.h`. Definir un conjunto amplio de esta manera puede hacer que la aplicación parezca que consume una gran cantidad de memoria RAM.

Cada tarea creada requiere un bloque de control de tareas (TCB) y una pila que se asignarán del heap. La figura 42 muestra cómo heap_1 subdivide la matriz a medida que se crean tareas. Refiriéndose a la Figura 42:

- A muestra la matriz antes que las tareas se creen, por lo que la matriz está vacía.
- B muestra la matriz luego que se creó una tarea.
- C muestra la matriz luego que se crearon tres tareas.

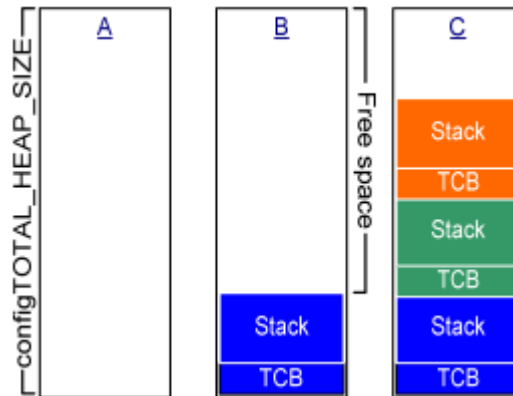


Figura 42 – RAM siendo asignada en una matriz cada vez que se crea una tarea.

Heap_2.c

Heap_2.c también utiliza una matriz sencilla dimensionada por `configTOTAL_HEAP_SIZE`. Utiliza un algoritmo mejor en condiciones de asignar memoria y, a diferencia heap_1 sí permite liberar memoria. Una vez más, la matriz se declaró estática, lo que hará que la aplicación parezca que consume una gran cantidad de memoria RAM.

El algoritmo que mejor se ajusta, asegura que la función `pvPortMalloc()` utiliza el bloque libre de la memoria que está más cerca en tamaño a la cantidad de bytes solicitados. Por ejemplo, considere la situación donde:

1. El montón contiene tres bloques de memoria libre que son 5 bytes, 25 bytes y 100 bytes respectivamente.
2. La función `pvPortMalloc()` es llamada y requiere 20 bytes de memoria RAM.

El bloque libre más pequeña de RAM en la que el número solicitado de bytes se ajusta es el bloque de 25 bytes, así `pvPortMalloc()` divide el bloque de 25 bytes en un bloque de 20 bytes y un bloque de 5 bytes antes de devolver un puntero al bloque de 20 bytes. El nuevo bloque de 5 bytes permanece a disposición de las futuras convocatorias de `pvPortMalloc()`.

`Heap_2.c` no combina bloques libres adyacentes en un solo bloque más grande por lo que puede sufrir fragmentación, sin embargo, la fragmentación no será un problema si los bloques que se asignan y posteriormente son liberados son del mismo tamaño. `Heap_2.c` es adecuado para una aplicación que crea y elimina tareas siempre que el tamaño de la pila asignada a las tareas creadas no cambia.

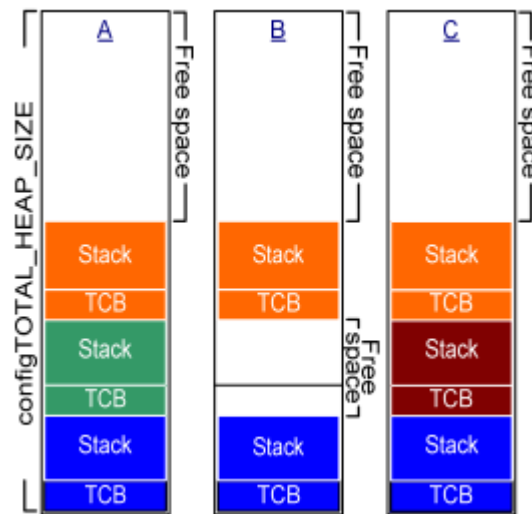


Figura 43 – RAM siendo asignada desde la matriz a medida que las tareas son creadas y borradas.

La figura 43 muestra cómo funciona el mejor algoritmo cuando se crea una tarea, se elimina, y luego se crea de nuevo. Refiriéndose a la Figura 43:

- A muestra la matriz después que se han creado tres tareas. Un bloque libre grande permanece en la parte superior de la matriz.
- B muestra la matriz después que una de las tareas ha sido eliminada. El gran bloque libre en la parte superior de la matriz permanece. Ahora hay también dos bloques libres más pequeños que se asignaron previamente a la TCB y a la pila de la tarea eliminada.
- C muestra la situación después de haber creado otra tarea. La creación de la tarea resultó en dos llamadas a `pvPortMalloc()`, uno para asignar un nuevo TCB y uno para asignar la pila (las llamadas a `pvPortMalloc()` se producen internamente dentro de la función API `xTaskCreate()`).

Cada TCB es exactamente del mismo tamaño, por lo que el mejor algoritmo de ajuste asegura que el bloque de memoria RAM que habían sido asignados a la TCB de la tarea eliminada se vuelve a utilizar para asignar el TCB de la nueva tarea.

El tamaño de la pila asignada a la tarea es idéntico al asignado a la tarea previamente eliminada, por lo que el mejor algoritmo de ajuste asegura que el bloque de memoria RAM que había sido asignado a la pila de la tarea eliminada se vuelve a utilizar para asignar a la pila de la nueva tarea.

El bloque no asignado más grande en la parte superior de la matriz permanece intacto.

Heap_2.c no es determinista, pero es más eficiente que la mayoría de las implementaciones de la biblioteca estándar de malloc() y free().

Heap_3.c

Heap_3.c simplemente utiliza la librería estándar malloc() y la función free(), pero hace que las llamadas sean seguras, suspendiendo temporalmente el scheduler. La aplicación se muestra en el Listado 74.

El tamaño de la pila no se ve afectada por configTOTAL_HEAP_SIZE y en su lugar se define por la configuración del enlazador (linker).

```
void *pvPortMalloc( size_t xWantedSize )
{
    void *pvReturn;
    vTaskSuspendAll();
    {
        pvReturn = malloc( xWantedSize );
    }
    xTaskResumeAll();
    return pvReturn;
}

void vPortFree( void *pv )
{
    if( pv != NULL )
    {
        vTaskSuspendAll();
        {
            free( pv );
        }
        xTaskResumeAll();
    }
}
```

Listado 74 – Implementación del Heap_3.c