Sistemas Embebidos II - IMT 322

Laboratorio 2: Recursos compartidos

Deadline: 20/03/2023

Introducción

En este laboratorio, se crea un entorno simulado basado en FreeRTOS. A partir de aquí, se explora algunos de los conceptos y algoritmos presentados durante las clases.

Entorno

Con la ayuda de RTOS, es posible al menos crear una ïlusión"de que las tareas se ejecutan en paralelo.

Analice el código del laboratorio del proyecto Keil uvision. Se implementara un código de tal forma que dos tareas compartan un recurso. Para fines de demostración, el recurso exclusivo aquí es un LED rojo (el acceso sin mutex no hace daño). Los LED azul y verde indican si la tarea 1 o la tarea 2 se están ejecutando respectivamente.

Verifique que la configuración para FreeRTOS sea correcta; en particular, asegúrese de que el algoritmo de programación sea Round Robin (ya que proporciona el .entorno"de multiprocesamiento). Se le pide que inserte funciones de delay (esta se brinda en el codigo) en algunas preguntas para asegurarse de que se produzca la preferencia durante el período de retraso, de modo que ambas tareas se puedan sincronizar y 'ejecutar en paralelo'. Siéntase libre de insertar la función de delay donde desee ver el resultado de la sincronización exacta entre dos tareas para simular el efecto del multiprocesamiento, pero asegúrese de que el período de delay sea lo suficientemente largo para que se produzca la preferencia. También puede probar otros algoritmos (no preventivos) durante el laboratorio y debería ver que el programa secuencial no suele implicar problemas concurrentes difíciles.

También puede usar un osciloscopio, en cuyo caso, debe conectar la sonda a PA_6 para el LED verde, PA_5 para el LED rojo y PA_7 para el LED azul (Por ejemplo).

Conceptos del Lab

- FreeRTOS
- Cronogramas
- Recursos compartidos
- Semaforos
- Mutex
- Tasks

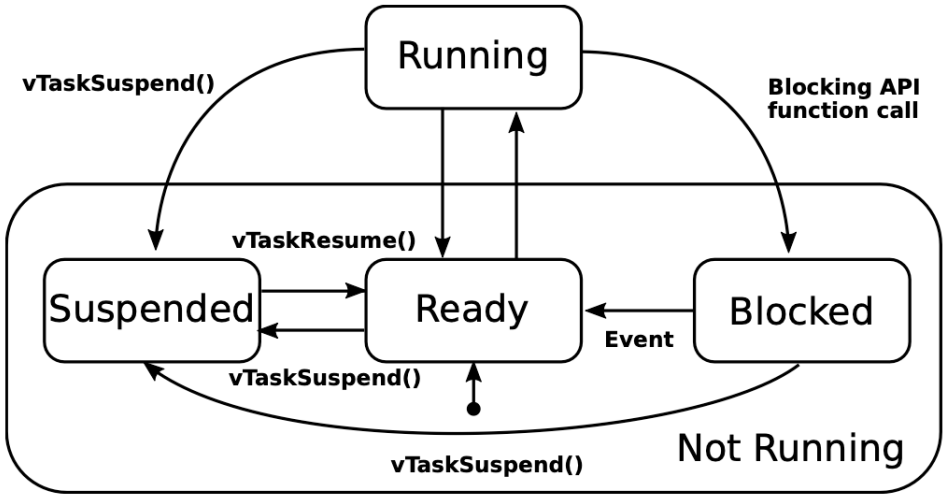

Imagen 1: Estados de una tarea en FreeRTOS

FreeRTOS

En FreeRTOS, las tareas pueden ser creadas usando la función xTaskCreate(). La declaración se muestra en el Codigo 1. Como puede ver esta función tiene varios parámetros. Revisemos que quiere decir cada uno de estos.

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,      // Pointer to the task function
                        const char * const pcName,      // Descriptive name of the task
                        uint16_t usStackDepth,     // Stack size allocated for the task
                         void *pvParameters ,          // Task specific parameters
                         UBaseType_t uxPriority ,      // Priority of the task
                         TaskHandle_t *pxCreatedTask     // Handle to the task
                      );
```

Codigo 1: Declaracion de xTaskCreate()

- pvTaskCode - este argumento apunta a una función que contiene la lógica de la tarea. Esta función nunca debe terminarse. Por lo tanto, la funcionalidad de la tarea debe encapsularse en un bucle infinito. El Código 2 muestra un ejemplo de una función de tarea de este tipo.
- pcName - este argumento debe elegirse para que sea un nombre descriptivo para la tarea. FreeRTOS no utiliza este nombre, pero facilita el proceso de depuración.
- usStackDepth - para cada tarea, se preasignará un espacio de memoria. El programador debe especificar el tamaño del stack según los requisitos de memoria de la tarea. En este laboratorio, configuramos usStackDepth en un valor predeterminado de configMINIMAL_STACK_SIZE. En práctica, el tamaño del stack requerido se puede determinar con las herramientas de FreeRTOS y luego se puede configurar en un valor razonablemente pequeño.
- pvParameters - este argumento permite a los usuarios pasar información adicional a la función de la tarea para la inicialización, como parámetros que describen un comportamiento más detallado de la lógica de la tarea. Para un uso genérico, esto se logra mediante un puntero. Por lo tanto, cualquier argumento debe pasarse a través de un puntero a un objeto vacío.

- uxPriority - este argumento se puede utilizar para asignar una prioridad a la tarea. El papel de las prioridades se explicará en el Ejercicio.
- pxCreatedTask - este argumento permite al usuario recibir un identificador de la tarea creada. Algunas funciones de FreeRTOS requieren un identificador para realizar operaciones en una tarea específica durante el tiempo de ejecución, por ejemplo, para cambiar la prioridad de una tarea o eliminar una tarea.

Importante: Note que para este laboratorio se obviara el loop infinito. Esto es solo con fines demostrativos para los conceptos explorados en clases.

```
1  void vTaskFunction(void *pvParameters)
2  {
3    // Some code here
4
5  }
```

Codigo 2: Declaracion funcion del task

Setup: FreeRTOSConfig

1. Para este laboratorio asegurese de tener funcionando FreeRTOS en su plataforma
2. Descargue el proyecto ejemplo y abra a FreeRTOSConfig.h
3. Configure el cronograma con Time Slicing y que este sea preventivo
4. En el macro configTICK_RATE_HZ con un valor de 5
5. Configure tres pines como salida (a su eleccion). Esto debe realizarlo en la funcion setup_GPIO y setup_RCC. Note que puede usar bare metal programming o alguna capa de abstraccion como HAL o LL.
6. Una vez haya configurado estos pines como salida conectelos de la siguiente forma, por ejemplo:
   - LED Rojo - PA.1 - Los Task 1 y Task 2 intentaran acceder a este recurso al mismo tiempo. Por tanto, el LED Rojo hara el rol de un recurso compartido. Modifique adecuadamente las funciones, $Access$ y $Release$. Estas son encargadas de simular el acceso y la liberacion del recurso.
   - LED Azul - PA.2 - El Task 1 se hara cargo de este LED. Lo prendera y apagara. El Task 2 no tiene acceso a este recurso. Modifique el Task 1 de tal forma que este prenda y apague los LEDs.
   - LED Verde - PA.3 - El Task 2 se hara cargo de este LED. Lo prendera y apagara. El Task 1 no tiene acceso a este recurso. Modifique el Task 1 de tal forma que este prenda y apague los LEDs.

Ejercicio 1: Recursos compartidos

1. Una vez haya realizado el Setup correspondiente. Analice el código. Verifique que los LEDs se prendan y apaguen sin un sistema operativo para luego realizar los siguientes pasos. Una vez lo haya verificado vaya al paso 2.
2. Compile, cargue y ejecute el código. Puede ver que $vTask1$ finaliza su tarea (los LED rojo y azul se encienden y apagan) sin competencia del task2.

3. ¿Qué pasaría si se descomenta la línea $result = xTaskCreate(vTask2...$ pruebe y ejecute el código. ¿Que observa? ¿Es eso lo que esperabas? Ahora se intentara implementar un mutex para estas dos tareas. Se puede crear nuevas variables globales y agregar código nuevo en task1 y task2, pero no se debe cambiar la secuencia del código en las secciones críticas. Puedes asumir que todos las instrucciones son atómicas.

4. El primer esquema mutex sugiere verificar si se accede al recurso antes de ingresar a la sección crítica. Si el recurso está disponible, ingrese la sección crítica; de lo contrario, se producirá una espera ocupada. Implemente y utilice la función de comprobación. $Check(RED)$, por ejemplo, esta debe devuelver 1 cuando el LED rojo está encendido y 0 cuando está apagado. Es decir, 1 cuando es recurso esta siendo usado y 0 cuando este esta libre. Implemente y utilice esto agregando lo justo antes de la sección crítica para ambas tareas (use el ciclo while y la funcion Check), ejecute y describa lo que ve. ¿Esto impone mutex?

5. Ahora, suponga que tiene el ciclo while como while(Check(RED))__NOP() para la pregunta 4. Intente agregar esta línea:

```
Delay(rand() %RDIV*RAMT + RMIN);
```

Codigo 3: Random delay

justo entre el bucle while y las secciones críticas de ambas tareas. Mantenga la tarea sin hacer nada por un tiempo antes de entrar en la sección crítica. Vuelva a ejecutar su código. ¿Cómo afectará esto al resultado? ¿Hay algún momento en que los tres LED estén encendidos? ¿Qué implica el resultado?

6. El segundo esquema mutex sugerido es el siguiente. Primero, cree una variable de tipo entero global llamada token e inicialícela como 0. Esta variable indica a qué tarea le toca usar el recurso crítico. En segundo lugar, agregue este código para proteger cada sección crítica. Sean $i \in 0, 1$ y $i' = 1 - i$.

```
while ( token != i ) { __NOP();}

// comienza seccion critica
...
// finaliza seccion critica

token = i';
```

Codigo 4: Segundo esquema

Implemente esto para ambas tareas, ejecute y describa lo que ve. ¿Esto impone mutex? Si es así, ¿cuáles son los principales problemas de dicho mutex?

7. El tercer esquema mutex sugiere el uso de flags. Cree una variable global para las flags, esta es un array de dos elementos:

```
flags[2] = {0, 0};
```

Codigo 5: Definicion de flags

Luego, agregue el siguiente codigo a cada task: (tenga en cuenta que i' es igual a $1 - i$)

```
flag[i] = 1;

while (flag[i']) {__NOP();}

// comienza seccion critica
...
```

```
// finaliza seccion critica

flag[i] = 0;
```

Codigo 6: Tercer esquema

Aquí, una flag establecida indica el deseo de ejecutarse. Implemente el código anterior para ambas tareas, ejecute y describa lo que ve. ¿Esto impone mutex? Si es así, compárelo con el segundo esquema; ¿Cómo mejora esto el mutex?

8. Como en el inciso 5, agregue esta línea de código:

```
Delay(rand() %RDIV*RAMT + RMIN);
```

Codigo 7: Random delay

entre donde establece la flag y el ciclo while. ¿Cómo afectará esto al resultado? Ejecute el código y describa lo que ve. ¿Qué implica el resultado?

9. El cuarto esquema de exclusión mutua modifica el tercero para resolver el problema del deadlock siendo 'cortés' y convirtiendo la flag de un recurso no interrumpible en un recurso interrumpible:

```
flag[i] = 1;

while (flag[i']) {

  flag[i] = 0;

  Delay(tiempo_aleatorio);

  flag[i] = 1;

}

// comienza seccion critica
...
// finaliza seccion critica

flag[i] = 0;
```

Codigo 8: Cuarto esquema

Implemente esto para ambas tareas; repita la pregunta 8 para este esquema. Corre y describa lo que ve. ¿Este esquema resuelve el problema que tiene el tercer esquema?

10. Parece que el cuarto esquema es problemático. Potencialmente causará un **livelock** donde dos tareas son demasiado 'cortés', de modo que ninguna tarea consuma ningún recurso mientras espera a la otra. Intente usar la función de retraso nuevamente para mostrar cómo podría suceder eso. (Antes de la seccion critica)

    Sugerencia: la función de retraso es efectivamente un mecanismo de sincronización entre las tareas; utilícelo siempre que desee que ambas tareas se ejecuten en paralelo 'real'.

11. Implemente el algoritmo de Dekker para ambas tareas. Se puede encontrar una versión en pseudocódigo en el recurso mas abajo. Puede comenzar reutilizando el token y las flags implementadas para las preguntas anteriores. Comprenda cómo funciona.

    Pista: el cuarto esquema en realidad es bastante parecido al algoritmo de Dekker; intente combinar el segundo esquema y el cuarto esquema.

Como habrá notado en el laboratorio, todos los mutexes aplican el mecanismo de espera ocupada. Aunque en algunos casos esto es deseable, generalmente es una pérdida de recursos de la CPU. Por esto, los servicios del sistema operativo como los semáforos y timers son mejores alternativas en la mayoría de los casos.

Ejercicio 2: Semaforos

FreeRTOS usa el algoritmo de cronograma llamado 'Fixed Priority Pre-emptive Scheduling with Time Slicing'. En el libro de FreeRTOS se explica de la siguiente forma.

- **Fixed Priority** - Un algoritmo de cronograma descrito como Fixed Priority no cambia la prioridad asignada a la tarea programada, pero tambien no previene que la tarea cambie su propia prioridad u otra prioridad de otra tarea.
- **Pre emptive** - Se 'interrumpira' la tarea que esta en el estado Running si la nueva tarea tiene una prioridad mayor que la tarea en el estado running. Ser preventivo quiere decir que se puede quitar a una tarea del estado running de forma involuntaria para permitir que otra tarea ocupe ese estado siempre y cuando esta ultima tenga mayor prioridad.
- **Time Slicing** - Para poder compartir el tiempo en el procesador entre tareas con la misma prioridad.

A cada tarea se le puede asignar una prioridad con el argumento uxPriority de la función xTaskCreate(). El programador en FreeRTOS usa estas prioridades para establecer tareas. Se ejecutará la tarea con la prioridad más alta que esté lista para ejecutarse, en el estado Ready. Si dos tareas tienen la misma prioridad y están listas para ejecutarse, se utiliza la división de tiempo para compartir el tiempo de procesamiento entre estas tareas.

Mutex - A veces, una operación en un recurso compartido no se debe adelantar ya que, de lo contrario, no se garantiza la funcionalidad correcta del programa. En nuestro caso, las tareas creadas en el Ejercicio 1 tienen la misma prioridad y, en consecuencia, el intervalo de tiempo se adelanta a una tarea para permitir que la otra también se ejecute. Por ejemplo, esto sucede durante una transmisión UART, la salida se corromperá.

Se puede utilizar un mutex para garantizar el acceso exclusivo a un recurso compartido. Esto se puede considerar como un token único atribuido al recurso compartido. Una tarea solo puede usar el recurso si primero puede obtener el token con éxito. A partir de ese momento, el token no está disponible para ninguna otra tarea. Cuando la tarea ha terminado de usar el recurso, debe devolver el token, lo que permite que otras tareas reclamen el recurso. En FreeRTOS, se utilizan tres funciones para la exclusión mutua:

- la creación de una exclusión mutua se realiza con xSemaphoreCreateMutex(),
- se puede tomar un token específico con xSemaphoreTake()
- y devolverlo con xSemaphoreGive() (consulte los Chapters del 'FreeRTOS reference' 4.6, 4.13, 4.16) .

```
1 // Funcion para crear un mutex nuevo y obtener su handle
2 SemaphoreHandle_t xSemaphoreCreateMutex(void);
3
4 // Funcion para tomar el mutex
5 xSemaphoreTake( SemaphoreHandle_t xSemaphore,           // Handle al mutex
6                  TickType_t xTicksToWait  );           // Numero de ticks que hay
7 //que esperar antes de que se devuelva el token de forma automatica
8
```

```
// Funcion para devolver el mutex
xSemaphoreGive ( SemaphoreHandle_t xSempahore );    // Handle del mutex
```

Codigo 9: Declaracion semaforos

Ejercicio 2.1

Para este ejercicio puede continuar usando el proyecto anterior o puede crear uno nuevo. Reescriba la funciónes utilizando un mutex de modo que la tarea no se pueda interrumpir durante el acceso al recurso compartido. Primero, declare un identificador SemaphoreHandle_t global e inicialice el identificador en main() con la función xSemaphoreCreateMutex(). Para ello, asegúrese de incluir semphr.h. Luego, use xSemaphoreTake y xSemaphoreGive para modificar la funcion correspondiente. Puede usar la definición global portMAX_DELAY de FreeRTOS como entrada para el argumento xTicksToWait de xSemaphoreTake. Verifique que el recurso sea exclusivo. ¿En qué orden se ejecutan las tareas?

Ejercicio 2.2

Cambie las prioridades de sus tareas de modo que la primera tarea tenga una prioridad de 1 y la segunda tarea una prioridad de 2. ¿Qué puede observar?

Retraso sin bloqueo - Nuestra función de tarea tiene un gran problema. Después de que el task acceda al recurso, la tarea espera un momento. En principio, la tarea está inactiva y durante este tiempo, la segunda tarea podría acceder al recurso. Sin embargo, la función vSimpleDelay() se implementa como un ciclo for vacío que simplemente itera continuamente la variable contador. Desde la perspectiva del programador, la tarea todavía está en estado Running porque está procesando el bucle for. FreeRTOS ofrece una mejor solución para esperar un tiempo determinado: la función vTaskDelay(), que se basa en timers.

```
void vTaskDelay( TickType_t xTicksToDelay);
```

Codigo 10: Declaracion delay non-blocking (Revisar 'FreeRTOS Referenca Chapter 2.9')

Para comprender completamente cómo funciona la función, primero debemos echar un vistazo los estados de las tareas más detallada en la Figura 1 se muestra estos estados.

vTaskDelay() cambia el estado de la tarea de llamada del estado Running al estado Blocked durante un período de tiempo determinado. Por lo tanto, la tarea estará en el superestado Not Running y se puede ejecutar otra tarea, que está en el estado Ready. El tiempo que la tarea estará en el estado Blocked se especifica en ticks mediante el argumento xTicksToDelay. Ticks es una unidad de tiempo utilizada internamente por FreeRTOS. Para mayor comodidad, puede utilizar la función pdMS_TO_TICKS(), que convierte milisegundos en ticks. Cuando ha transcurrido el tiempo de delay especificado, se produce un evento que mueve la tarea al estado Ready.