**Implementación de Mutex y esquivando Dead Locks (Puntos Muertos)**

Nota: Las respuestas pueden variar dependiendo de su tarjeta.

**OBJETIVO**

En este ejercicio simularemos un ambiente basado en RTX RTOS. De aquí explicaremos los conceptos y algoritmos introducidos durante las presentaciones futuras.

**EL AMBIENTE DE DESARROLLO**

Usar una ST-Nucleo como procesador púnico. De allí ud encontrará problemas de preemption pero no de multi-procesadores. Con la ayuda de RTX, sin embargo, es posible al menos crear la “ilusión” que las tareas corren en paralelo.

Se proveerá un código de proyecto en Keil. Ud. Podrá ver que el código posee dos tareas. Las cuales comparten un recurso exclusivo. Para propósitos demostrativos, el recurso exclusivo aquí son LEDs (non-mutex no tienen ningún efecto negativo). El LED azul y verde indican donde las tareas 0 y 1 están corriendo respectivamente.

Verificar que la configuración es correcta para RTX; in particular, asegurese que el algoritmo del “scheduler” es Round Robin (a como es provisto en ambientes multiprocesamiento). Será su labor inserter una tarea de retardo en algunas preguntas para asegurarse que el preemption ocrurre durante un period de retardo así que ambas taras pueden ser sincronizadas en paralelo. Sientase libre de ingresar una función en cualquier lugar para ver el efecto de sincronía entre tareas simultnaneas de multiprocesamiento, pero asegurese de que el retardo del period es lo suficientemente largo para “preemption” y que tome lugar. Ud. También puede tartar otros algoritmos (non-preemptive) durante el laboratorio y ud debería ver que el programa secuencial no involucre dificultades en la concurrencia. Debe también usar un osciloscopio, en cuyo caso, debe pegar la punta a PA\_6 para el verde y PA\_5 para el rojo, PA\_7 para el azul.

**PROCEDIMIENTO DEL LABORATORIO**

1. Abrir la plantilla de Proyecto y ver el código.
2. Compile, descargue y corra el código. Verá que la task0 finalisa su labor (LED rojo y azul en off) sin competir con la tarea 1.
3. Qué pasa si descomenta la línea *//taskID1 = os\_tsk\_create( Task1, 0);* intente corer el código. Que vé y que se supone debe pasar.

Ahora intentará implementar mutex en las dos tareas. Ud. puede crear una variable global y añadir Nuevo código en task0 y task1, pero, no cambie la secuencia de código en las secciones críticas. Ud. puede asumir que el código es atómico (se puede ver antes pero no su valor intermedio).but do not change the sequence of the code in the critical sections.

1. El primer mutex sugiere que puede verificar si el recurso está accedido antes de entrar a la sección crítica. Si el recurso está disponible, entrar a la sección, de lo contrario un busy-waiting ocurriría. Use la función de verificar (check). Check (RED), por ejemplo, retorna un 1 cuando el LED rojo está ON y 0 cuando esta OFF. Implementar esto añadiendo solo una línea justo después de la sección crítica para ambas tareas (mientras el lazo cerrado debería ser suficiente), correr y describir que ve. ¿Esto refuerza el Mutex?
2. Ahora asuma que tiene un while loop como (Check(RED)){} para la pregunta anterior. Trate de añadir esta línea Delay(rand() % RDIV\*RAMT + RMIN); justo entre el while y la sección crítica de ambas taras. Esta mantiene la tarea realizando nada por momentos mientra entra en la sección crítica. Correr nuevamente el código. ¿Como afecta esto el resutlado?. ¿Hay algún momento que lost res LEDs están encendidos? ¿Qué implica el resultado?
3. El segundo esquema de mutex sugiere lo siguiente. Primero cree una variable de temporizador global llamada *token* e inicialícela en 0. Esta variable indica que tarea is utilizada para usarse como recurso crítico. Segundo, añada este código para proteger cada sección crítica de código crítico. Siendo *i*∈ {0,1} e *i’* be *1-i*.

*while ( token != i ) { }*

*/\* critical section \*/*

*token = i’;*

Implemente esto para ambas tareas, corra y describa lo que ve. ¿Esto refuerza el mutex? Si lo hace, ¿Cuales son los principals problemas para estos mutex?

1. El tercer esquema de mutes sugiere utilizar banderas como sigue (note que *i’* es igual a *1-i*):

*flag[i] = true;*

*while ( flag[i’] ) { }*

*/\* critical section \*/*

*flag[i] = false;*

Aquí, levanter una bandera que indica cual ejecutar. Implemente el código superior para ambas tareas, corra y describa lo que ve. ¿Esto refuerza el mutex? Si lo hace, comparelo con el segundo esquema; ¿Como se mejora el mutex?

1. Como en la sección 5. Añada esta línea de código: Delay(rand() % RDIV\*RAMT + RMIN); entre donde ud ajustó la bandera y puso el lazo while. ¿Cómo esto afectará el resultado? Corra el código y describa lo que ve. ¿Qué implica el resultado?
2. El cuarto esquema de mutex podifica el tercero para resolver el deadlock siendo “polite” y ajustando la bandera de non-preemptible del recurso a una de recurso preemptible:

*flag[i] = true;*

*while ( flag[i’] ) {*

*flag[i] = false;*

*Delay(Random\_Time);*

*flag[i] = true;*

*}*

*/\* critical section \*/*

*flag[i] = false;*

Implemente para ambas tareas, repita las preguntas de la sección 8 para este esquema. Corra describa lo que ve. ¿Será que este esquema resuelve el problema que tenía el tercer esquema?

1. El cuarto esquema se cree es el problemático. Potencialmente causará un **“livelock”** en donde dos tareastartan de ser “polite”, así que la tarea actual consume cualquier recurso para cada una. Trate de usar la función de retardo a través de mostrar como puede pasar esto.

Pista: la función de retardo es efectivamente un mecanismo de sincronización entre tareas; uselo siempre y cuando ud desee que ambas tareas corran “realmente” en paralelo.

1. Implemente un algoritmo de Dekker para ambas tareas. Una version de pseudocódigo puede ser encontrada aquí en el archivo de solución o en las filminas. Debe de esetar reusando el token y las banderas implementadas para las siguientes preguntas. Entienda como funciona. Pista: El cuarto esquema es muy cercano al algoritmo de dekker; intente combinar el segundo esquema y el cuarto esquema.
2. (Opcional) Otro algoritmo para mutex exitoso es el de Peterson. Busque en un libro de sistemas operativos o en línea.[[1]](#footnote-2) Muestre como funciona y porqué este esquema es mejor

*flag[i] = true;*

*turn = i’;*

*while ( flag[i] && turn == i’ ) { }*

*/\* critical section \*/*

*flag[i] = false;*

Este algoritmo es más “polite” que el del cuarto esquema. Sin embargo, lo más importante es que no depende de operaciones atómicas.

Como habrá notado del laboratorio, todos los mutex aplican en un estado de busy-wait. Además en algunos casos, es deseable e incurre en malgastar ciclos del CPU y recursos. Puede utilizar os\_tsk\_pass() en vez de un mecanismo de busy-wait, pero los servicios del sistema operative como semaforos son mejor alternative en muchos de los casos y también más sencillos de utilizar.

1. e.g., <https://en.wikipedia.org/wiki/Peterson%27s_algorithm> [↑](#footnote-ref-2)