Práctica 2
Avanzando con FreeRTOS
(Versión: marzo 2022)

Justificación

FreeRTOS es un sistema operativo de tiempo real para microcontroladores y pequeños microprocesadores extendido a multitud de plataformas. En la actualidad es el líder del mercado y ostenta una cuota de mercado del sector de los sistemas operativos de tiempo real (RTOS) superior al 25% según el informe "*Embedded Software Market*" elaborado por el grupo *Global Market Insights*. Sus orígenes de desarrollo en colaboración con los principales fabricantes de chips a lo largo de más de 2 décadas, unido a la reciente (desde 2017) administración del proyecto por parte de Amazon Web Services, hacen de este proyecto el referente principal del mundo de los sistemas operativos de tiempo real (RTOS). Además, su código se distribuye libremente bajo licencia *open source* MIT y cuenta con múltiples librerías que complementan su *kernel* y hacen del mismo un sistema operativo fiable y de muy fácil uso.

En esta práctica se van a introducir algunos de los conceptos más avanzados de este RTOS, que nos permitirán poder realizar desarrollos reales hacia el final de la asignatura.

Se recuerda al alumno que dispone de documentación completa sobre FreeRTOS en la página oficial, disponible en los siguientes enlaces:

- ✓ https://www.freertos.org/Documentation/RTOS_book.html
- ✓ https://www.freertos.org/features.html
- ✓ https://www.freertos.org/a00106.html

Así mismo, el código fuente del proyecto completo y actualizado se puede encontrar en el repositorio de GitHub:

- ✓ https://github.com/FreeRTOS
- ✓ https://github.com/FreeRTOS/FreeRTOS

Duración

4 horas presenciales (2 sesiones de 2h cada una). Se recuerda que según la *guía del plan de estudios* por cada hora presencial el alumno debe de realizar 1 hora y media de trabajo personal.

Entorno de trabajo

En las prácticas de esta parte de la asignatura, tanto para las relativas a FreeRTOS como al planificador de tareas SimSo, vamos a trabajar usando la máquina virtual XUbuntu disponible en el aula de UBUVirtual (la contraseña de acceso es "*osboxes.org*"). No obstante, aquellos alumnos que deseen trabajar de manera nativa en Windows, pueden usar el *port* disponible para Windows (https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html).

Así mismo, usaremos un proyecto de base, con las librerías de FreeRTOS, que nos sirva de plantilla y que podéis descargaros también desde el aula de UBUVirtual.

Actividades

2.1. Tareas periódicas frente a aperiódicas y esporádicas

Tal y como se ve en la parte teórica de la asignatura, existen tareas que se ejecutan de manera periódica cada cierto intervalo de tiempo, pero también es posible tener tareas que se ejecutan bajo demanda frente a eventos concretos. Dentro de este segundo tipo de tareas no periódicas, las mismas pueden ser críticas en cuanto al tiempo real o no. Su programación en FreeRTOS no diferirá demasiado y podemos indicar la criticidad o no de las mismas basado en la prioridad que asignemos a dicha tarea. Hasta esta práctica hemos trabajado únicamente con tareas periódicas, pero en este apartado veremos cómo podemos tener tareas que no ejecuten de manera periódica. Para ello podemos utilizar dos enfoques:

1) Ejecutar la acción asociada a la tarea una única vez, procediendo a la eliminación de la misma cuando termine la primera iteración. *Hints*: vTaskDelete().
2) Simular una tarea esporádica de ejecución aleatoria bloqueando la tarea durante un tiempo aleatorio en lugar de un intervalo fijo, como sucede con las tareas periódicas. *Hints*: vTaskDelay(), vTaskDelayUntil(), #include <stdlib.h>, random(), srandom().

2.2. Dependencia entre tareas

Cuando una tarea debe esperar a que otra finalice para poderse ejecutar, por ejemplo, porque ésta depende de datos generados por la otra, debemos ser capaces de bloquear la ejecución hasta que la tarea de la que depende haya finalizado. Una forma de conseguir este efecto es usando semáforos en FreeRTOS, aunque podemos usar directamente una cola en el caso de que los datos a leer se escriban en la misma. El uso de semáforos nos permite independizar, de la escritura/lectura en la cola, la dependencia entre tareas, o incluso gestionar esas dependencias cuando las tareas no necesitan intercambiar información entre las mismas.

En este apartado se pide crear tres tareas (T1-T3) con dependencia entre sí, de manera que la T1 se ejecute cada 3 segundos de manera periódica. La tarea T3 se ejecutará cuando T1 le ceda un semáforo, que deberá ser antes de bloquearse T1. La tarea T2 se ejecutará cuando T3 le ceda otro semáforo, que deberá ser antes de bloquearse T3 en espera del semáforo de T1.

Hints: xSemaphoreGive(), xSemaphoreTake(), xSemaphoreCreateBinary().

2.3. Acceso a recursos compartidos

En el acceso a recursos compartidos (variables, periféricos, sensores, actuadores, etc.), debemos garantizar que una tarea bloquee el recurso para evitar el acceso simultáneo que pueda dar lugar a inconsistencias en las lecturas y escrituras sobre el mismo.

En este apartado se pide programar dos tareas en FreeRTOS, donde cada una de ellas necesita acceder y escribir sobre dos variables compartidas (globales). Se realizará en ambas tareas la escritura sobre una de las variables, con un tiempo de 2 segundos entre medias, quedando la tarea bloqueada mediante un vTaskDelay, hasta escribir en la segunda de las variables. La sección de código que incluye ambas escrituras y el retardo

estará marcada como sección crítica mediante el uso de un mutex. Para garantizar el acceso a la misma de manera atómica, debemos implementar un mutex que nos evite realizar modificaciones sobre las mismas si la otra tarea está ejecutando la misma sección crítica. Comprobar que el acceso a la escritura de ambas variables es siempre secuencial por parte de la misma tarea, sin que se escriba la misma variable seguidamente por las dos tareas.

Hints: xSemaphoreCreateMutex().

2.4. Guardado y análisis de trazas con Tracealyzer 4

El código base utilizado en todas las prácticas de FreeRTOS de la asignatura tiene activado el guardado de trazas. Esta característica avanzada nos permite registrar los diferentes eventos de la ejecución de un programa para poder analizar que la ejecución es adecuada. Para analizar el archivo "Trace.dump" con la traza que se genera al finalizar la ejecución de nuestro programa, usaremos el software Tracealyzer de Percepio, que podéis acceder en los ordenadores del laboratorio o desde vuestros equipos usando UBULabs.

En este apartado se pide que carguéis la traza referente a los apartados 2.1 y 2.2 y exploréis las opciones que ofrece el software de análisis de trazas. Así mismo, debéis comprobar que la ejecución de las tareas de cada apartado es correcta y cumple con los requisitos especificados en el enunciado.

2.5. Comparativa planificador FreeRTOS usando y sin usar "preemption"

En este apartado vamos a comparar la ejecución de dos tareas cuando se utiliza y cuando no se utiliza la opción de poder interrumpir la ejecución de una tarea de prioridad más baja cuando una tarea de mayor prioridad está disponible. Para ello, debemos configurar en el fichero `FreeRTOSConfig.h`, escogiéndose el valor de la macro `configUSE_PREEMPTION` al valor 1 (si queremos activarla) o al valor 0 (si queremos deshabilitarla).

En este apartado se pide modificar el valor de la macro `configUSE_PREEMPTION` para el código del apartado 1.2 (es posible que tengáis que cambiar algún parámetro de las tareas para ver diferencias) y analizar en Tracealyzer el comportamiento del planificador en ambos escenarios (con valor de macro 0 vs. 1).

2.6. Personalizando el planificador (scheduler)

En este apartado vamos a ver cómo podemos conseguir que el planificador en vez de ejecutarse basado en prioridades, implemente una planificación basada en plazos (*deadline*). En este caso, programaremos la modificación de las prioridades de las tareas en base al plazo de ejecución de cada una de ellas, priorizando tareas con el *deadline* próximo frente a aquellas que tengas un plazo más holgado para ejecutarse. Para ello, debemos crear una estructura de datos en la que almacenar los plazos de cada tarea y modificar las prioridades de acuerdo con ello. Así mismo, debemos tener una tarea de elevada prioridad que modifique las prioridades de las tareas del sistema para conseguir ejecutar con antecedencia las tareas que vencen en un plazo más cercano.

2.7. Herencia de prioridades para una tarea que bloquea a otra

En este apartado vamos a ver cómo podemos conseguir que una tarea de menor prioridad, pero de cuyo resultado depende poder ejecutar otra tarea de más alta prioridad, pueda heredar la prioridad de la tarea que está esperando a que finalice para poder tener una planificación de tareas que evite la inversión de prioridad.

Para ello, debemos programar 3 tareas periódicas:

T1) Prioridad 4

T2) Prioridad 3

T3) Prioridad 5

La tarea T1 se ejecutará de manera continua con periodo de 5 segundos, consumiendo 3 segundos de procesador mediante un bucle, y bloqueándose hasta el comienzo del siguiente periodo.

La tarea T2 se ejecutará con periodo de 10 segundos, adquiriendo un semáforo mutex (S1) antes de su ejecución, consumiendo 3 segundos de procesador, y liberando dicho semáforo al final, tras lo cual se bloquea hasta el comienzo del siguiente periodo.

La tarea T3 se ejecutará con un período de 5 segundos. Antes de su ejecución deberá adquirir un semáforo mutex (S1), consumiendo 1 segundo de procesador y liberando el semáforo al final, tras lo cual se bloquea hasta el comienzo del siguiente periodo.

Programar el sistema descrito anteriormente y verificar que la tarea T2 hereda la prioridad de T3 para que el bloqueo de la misma no paralice su ejecución de máxima prioridad.