FreeRTOS en ESP32.

Introducción

El microcontrolador ESP32 viene con un RTOS (Real Time Operating System) integrado, el freeRTOS. Sin embargo no utiliza la versión vanilla de este, sino que utiliza una versión modificada para aprovechar el hecho de que el MCU dispone de dos unidades de procesamiento. Esta versión modificada provee Multriprocesamiento Simétrico de Dos Cores, nombrada como ESP-IDF FreeRTOS, la versión actual está basada en Vanilla FreeRTOS v10.4.3.

Nota: Este FreeRTOS puede ser configurado para que use un solo core, seteando la directiva CONFIG_FREERTOS_UNICORE = 1.

Multiprocesamiento Simétrico de Dos Cores

Es una arquitectura de procesamiento en donde dos (o más) CPUs idénticas son conectadas a través de una misma memoria principal compartida, y controladas por un solo sistema operativo. En general cumple que:

- Tiene varios cores ejecutándose de forma independiente. Cada uno con sus propios registros, interrupciones, etc.
- Presenta una vista idéntica de la memoria para cada core. Consecuentemente, un fragmento de código que acceda a una posición específica de memoria tiene el mismo efecto en cualquier core que se ejecute.

Sus principales ventajas son:

- Mayor capacidad de creación de hilos.
- Debido a la memoria compartida, los programas pueden cambiar de core durante su ejecución. Permitiendo mejores optimizaciones de CPU.

Si bien está permitido que un hilo cambie de core en tiempo de ejecución, existen escenarios en donde las tareas deben ejecutarse en un core en particular. Es por esto que existe el parámetro “Core Affinity” (Afinidad de núcleo) que permite especificar en qué core está permitida la ejecución del hilo.

- Un hilo pineado a un core en particular solo puede ejecutarse en ese core.
- Un hilo que no está pineado puede cambiar de cores en tiempo de ejecución.

SMP en ESP

- Dos cores idénticos conocidos como CPU0 y CPU1.

- CPU0 también es conocido como Protocol CPU (PRO_CPU) debido a que usualmente se le asignan las tareas correspondientes a las conexiones inalámbricas (Wi-Fi, Bluetooth, etc.).
- CPU1 también es conocido como Application CPU (APP_CPU) debido a que a este se le asigna el resto de la aplicación que no es asignado a PRO_CPU.

Tareas

Creación

Las tareas deben ser asignadas a un core específico al momento de ser creadas. Es por esto que disponemos de las siguientes funciones:

- **xTaskCreatePinnedToCore()**: Crea una tarea con una Core Affinity particular. La memoria de la tarea se aloca dinámicamente.
- **xTaskCreateStaticPinnedToCore()**: Ídem. A la anterior pero la memoria se aloca estáticamente (Asignada por el usuario).

A estas tareas se les debe pasar un parámetro adicional xCoreID el cual se corresponde a la Core Affinity de la tarea, este puede tener 3 valores:

- **0** pinea la tarea a CPU0
- **1** pinea la tarea a CPU1
- **tskNO_AFFINITY** Permite que la tarea se ejecute en cualquiera de los dos CPUs

Ejecución

Las tareas en ESP-IDF FreeRTOS siguen la misma anatomía que en Vanilla FreeRTOS. Mas específicamente las Tareas:

- Están en alguno de estos estados: Corriendo, Lista, Bloqueada o Suspendida.
- Se implementan a través de un loop infinito
- Nunca deben de retornar.

Borrado

Se implementa a través de la función **vTaskDelete()**, con la siguientes modificaciones:

- Cuando un core elimina una tarea pineada al otro core, la memoria reservada por esa tarea es liberada por la “Idle Task” del otro core (Debido a que se deben limpiar registros de la FPU)

- Cuando un core elimina una tarea que se encuentra ejecutándose en el otro core, se dispara un “Yield” en el otro core y la memoria de la tarea es liberada por una de las “Idle Tasks”(Dependiendo de la Core Affinity de la Tarea).
- La memoria de una tarea eliminada es liberada inmediatamente si:
    - La tarea está ejecutándose en un core en el cual está pineado
    - La tarea no se está ejecutando y no está pineada a ningún core

Debemos evitar invocar a la función vTaskDelete() a una tarea que se está ejecutando en otro core, esto puede llevar a comportamientos indeseados como:

- Borrar una tarea que está bloqueando un mutex
- Borrar una tarea que todavía debe liberar memoria dinámica que haya reservado

Es por esto que se recomienda que las tareas se eliminen sólo en estados previamente planificados o conocidos. Por ejemplo:

- Que la tarea se borre a ella misma usando vTaskDelete(NULL) luego de haber finalizado su ejecución y liberado toda la memoria dinámica reservada.
- Que la tarea se suspensa sola (Usando **vTaskSuspend()**), antes de que otra tarea la elimine.

Scheduler

El Scheduler de Vanilla FreeRTOS es descrito como un **scheduler apropiativo, de prioridades fijas y con Time Slicing** cumple con las siguientes características:

- Cada tarea es asignada una prioridad en su creación que no cambia en ejecución. El Scheduler ejecuta la tarea con mayor prioridad que esté Ready.
- El Scheduler puede cambiar la ejecución de una tarea a otra sin la cooperación de la tarea que estaba siendo previamente ejecutada.
- El Scheduler va cambiando la ejecución de tareas de misma prioridad que estén listas y en ejecución usando la lógica de round robin. Time Slicing es implementado por una interrupción periódica de timer.

El Scheduler de ESP-IDF FreeRTOS cumple con todo esto con las siguientes modificaciones:

- Cada CORE planifica su propio conjunto de tareas. A la hora de elegir una tarea para ejecutar seleccionará la próxima tarea que esté lista para ejecutarse con mayor prioridad. Una tarea estará disponible para el Core si:
    - Tiene una Core Affinity compatible (Está pineado al core o a ninguno).

    - No está siendo ejecutada por otro core.
- Esto puede llevar a que no necesariamente todas las tareas de mayor prioridad se ejecuten antes de las que tengan menor prioridad (Ya que si están pineadas a un core no pueden ser ejecutadas en otro).
- Si una tarea unpinneada está lista y es de mayor prioridad que las tareas ejecutándose en ambos cores, el scheduler se apropiará de la CPU en la cual se esté ejecutando el mismo.
- No es posible implementar un Round Robin perfecto entre tareas de la misma prioridad, debido a la core affinity que pueden llegar a tener estas. Es por esto que implementa un Best Effort Round Robin, seleccionando para ejecutar a la tarea de misma prioridad, con ninguna afinidad o afinidad al core del scheduler, que más tiempo pasó sin ser ejecutada. Por ende no se puede asumir que las tareas de misma prioridad se ejecuten en secuencia.

Interrupción de Timer

Se encarga de las siguientes tareas:

- Incrementar el tickCount del Scheduler
- Desbloquear aquellas tareas bloqueadas que ya cumplieron su tiempo
- Ejecutar el lazo de tick de la aplicación

En ESP-IDF FreeRTOS cada core tiene su propia interrupción periódica que ejecuta su propio gestor de interrupción. Estos tienen el mismo período, pero pueden tener distinta fase. Sin embargo, no tienen las mismas responsabilidades:

- CPU0 se encarga de todas las responsabilidades descriptas previamente. (Por ende cualquier evento que retrase este Scheduler causará que todo el sistema se retrase)
- CPU1 solo valida el timeSlicing y ejecuta el lazo de tick de la aplicación.

Tareas Idle

Se crea una tarea “Idle” en cada core con prioridad 0 cuando el Scheduler se inicia. Esta se ejecuta cuando no hay ninguna tarea ejecutándose y sus responsabilidades son:

- Liberar la memoria de las tareas eliminadas
- Ejecutar el lazo de ejecución del Idle
- Suspensión del planificador
- La suspensión del planificador en FreeRTOS permite detener temporalmente la planificación de tareas y luego reanudarla de manera controlada.
- En Vanilla FreeRTOS, se puede suspender y reanudar el planificador con las funciones vTaskSuspendAll() y xTaskResumeAll(). Durante la suspensión:

    - Se deshabilita el cambio de tareas, pero las interrupciones permanecen habilitadas.
    - No se pueden usar funciones de bloqueo o de yield, y se desactiva la división de tiempo entre tareas.
    - El contador de "ticks" se detiene, pero la interrupción de "tick" sigue ocurriendo para ejecutar el "tick hook" de la aplicación.
- En ESP-IDF FreeRTOS, suspender el planificador solo afecta al núcleo en el que se llama vTaskSuspendAll(). El cambio de tareas se deshabilita solo en ese núcleo, y las interrupciones del mismo núcleo siguen habilitadas.
- Las tareas desbloqueadas por una interrupción durante la suspensión irán a las listas de tareas pendientes de sus núcleos asignados o al núcleo en el que se llamó la interrupción si no tienen asignación de núcleo.
- Al reanudar el planificador con xTaskResumeAll():
    - Se reanudan las tareas que se agregaron a la lista de tareas pendientes del núcleo donde se llama.
    - En CPU0 de ESP-IDF, se ponen al día los "ticks" pendientes.
- Es importante destacar que la suspensión del planificador no garantiza la exclusión mutua entre tareas cuando acceden a datos compartidos. Se deben usar primitivas de bloqueo apropiadas, como mutexes o spinlocks, para lograr la exclusión mutua de manera segura.

Desactivación de Interrupciones

En Vanilla FreeRTOS, es posible desactivar y activar interrupciones mediante las funciones taskDISABLE_INTERRUPTS y taskENABLE_INTERRUPTS, respectivamente.

ESP-IDF FreeRTOS proporciona la misma API, pero con la diferencia de que las interrupciones solo se desactivan o activan en el núcleo actual.

Advertencia: Desactivar interrupciones es un método válido para lograr la exclusión mutua en Vanilla FreeRTOS (y sistemas de un solo núcleo en general). Sin embargo, en un sistema SMP (Multiprocesamiento simétrico), desactivar interrupciones NO es un método válido para garantizar la exclusión mutua. Consulte Secciones Críticas para obtener más detalles.

Secciones Críticas

- En Vanilla FreeRTOS, se utilizan secciones críticas desactivando interrupciones para lograr la exclusión mutua en el acceso a recursos compartidos.
- En ESP-IDF FreeRTOS, en sistemas SMP (Multiprocesamiento simétrico), las secciones críticas se implementan utilizando spinlocks en lugar de desactivar interrupciones.

- Las APIs de secciones críticas de ESP-IDF FreeRTOS incluyen un parámetro de spinlock adicional para gestionar las secciones críticas.
- En Vanilla FreeRTOS, las secciones críticas se gestionan con las funciones taskENTER_CRITICAL(), taskEXIT_CRITICAL(), taskENTER_CRITICAL_FROM_ISR(), y taskEXIT_CRITICAL_FROM_ISR().
- En ESP-IDF FreeRTOS, las funciones para gestionar secciones críticas incluyen un parámetro de spinlock adicional, como taskENTER_CRITICAL(&spinlock) y taskEXIT_CRITICAL(&spinlock).

Implementación

- En ESP-IDF FreeRTOS, cuando un núcleo entra en una sección crítica, primero desactiva sus interrupciones (o la anidación de interrupciones) hasta un nivel específico de prioridad.
- Luego, el núcleo espera adquirir un spinlock mediante una operación atómica de comparación y configuración.
- Una vez adquirido el spinlock, la función retorna y el código dentro de la sección crítica se ejecuta con las interrupciones (o la anidación de interrupciones) desactivadas.
- Al salir de una sección crítica, el núcleo libera el spinlock y reactiva las interrupciones (o la anidación de interrupciones).

Restricciones y Consideraciones

- Las secciones críticas deben ser lo más breves posible para evitar retrasos en las interrupciones pendientes.
- Se recomienda que una sección crítica acceda solo a unas pocas estructuras de datos o registros de hardware.
- Si es posible, se debe evitar llamar a funciones de FreeRTOS desde una sección crítica.
- Nunca se deben usar funciones de bloqueo o de yield dentro de una sección crítica, ya que podrían causar bloqueos o comportamiento inesperado.