LAB 5 - Multitarea con FreeRTOS

Sistemes Encastats i Ubics

Estudiantes: Zixin Zhang & Tomeu Uris Tortella
Facultad: FIB - UPC
Curso: 2025-2026

Introducción

Esta práctica se centra en el estudio de la **multitarea y la sincronización** ofrecida por el *kernel* de FreeRTOS, buscando establecer una comprensión sólida de la **planificación de tareas** y la comunicación entre ellas.

Los principales retos abordados son:

- **Exclusión Mutua:** Implementación y análisis comparativo de **Mutexes** y **Semáforos Binarios** para proteger recursos de *hardware* (LEDs).
- **Modelo Productor/Consumidor (P/C):** Diseño de un sistema P/C que emplea **Colas de Mensajes** y **Semáforos de Conteo** para asegurar la alternancia de productores y la distribución equitativa de la carga de trabajo entre consumidores.
- **Análisis Avanzado:** Investigación del ***overhead*** del cambio de contexto del *kernel* y el análisis de la **Inversión de Prioridad**, incluyendo la efectividad del mecanismo de *Priority Inheritance*.

El trabajo demuestra la capacidad de diseñar y analizar sistemas multitarea síncronos en un entorno embebido.

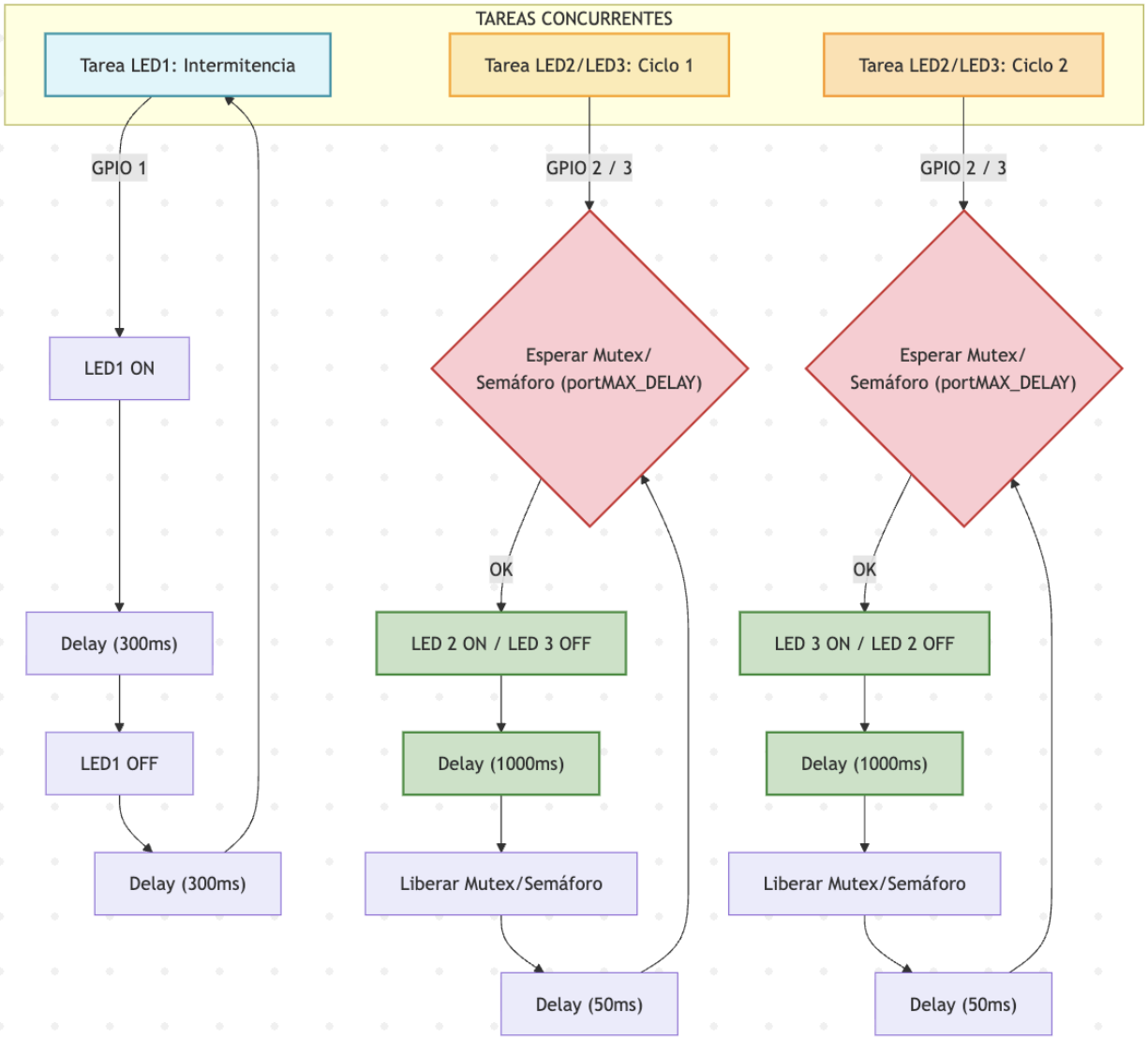
1. Scheduling por Defecto

El tipo de *scheduling* que utiliza FreeRTOS en un proyecto de ESP32-C6 por defecto es **Preemptivo de Prioridad Fija con *Time-Slicing***.

Características Clave

- **Prioridad Fija:** Cada tarea tiene una prioridad asignada al crearse. El planificador siempre ejecuta la tarea en estado **"Ready"** con la **prioridad más alta**.
- **Preemptivo:** Una tarea de baja prioridad que se esté ejecutando puede ser **interrumpida (desalojada)** inmediatamente si una tarea de mayor prioridad pasa al estado "Ready".
- ***Time-Slicing*:** Si hay **múltiples tareas** con la **misma prioridad más alta**, el planificador alterna su ejecución (*round-robin*) en cada *tick* del sistema, asegurando que todas compartan equitativamente el tiempo de CPU.

2. Implementación de la Aplicación



La aplicación se implementa con tres tareas:

1. **led_blink_task_1**: Controla **LED1**. Realiza una intermitencia (ON/OFF) de 0.3 segundos (300 ms) cada ciclo. Esta tarea corre de forma **independiente** y en paralelo a las otras.
2. **led_control_task_2**: Controla **LED2** (ON) y **LED3** (OFF). Entra en la sección crítica, realiza su ciclo de 1 segundo (1000 ms), y luego libera el recurso.
3. **led_control_task_3**: Controla **LED3** (ON) y **LED2** (OFF). Entra en la sección crítica, realiza su ciclo de 1 segundo (1000 ms), y luego libera el recurso.

Las tareas 2 y 3 comparten el recurso (la alternancia LED2/LED3) y usan un mecanismo de exclusión mutua (Mutex o Semáforo Binario) para garantizar que solo una de ellas pueda modificar los LEDs compartidos a la vez, logrando así la alternancia de 1 segundo.

Comparativa: Mutex vs. Semáforo Binario

Característica	Mutex	Semáforo Binario
Uso Principal	Exclusión Mutua (proteger recursos compartidos)	Sincronización (señalización de eventos)
Propiedad	**Sí**. Solo la tarea que lo *toma* puede *liberarlo*.	**No**. Cualquier tarea puede *liberarlo*.
Inversión de Prioridad	**Manejo automático** de la Inversión de Prioridad mediante **Priority Inheritance** (Herencia de Prioridad).	**No** soporta Herencia de Prioridad.
Uso en esta Práctica	Ideal para proteger el recurso compartido (LED2/LED3)	Puede usarse como Mutex (contador a 1) si se respeta la propiedad.

❓ ¿Existe alguna diferencia en el comportamiento?

No, en esta implementación específica no se observa una diferencia significativa en el comportamiento de la alternancia LED2/LED3.

Explicación del Motivo

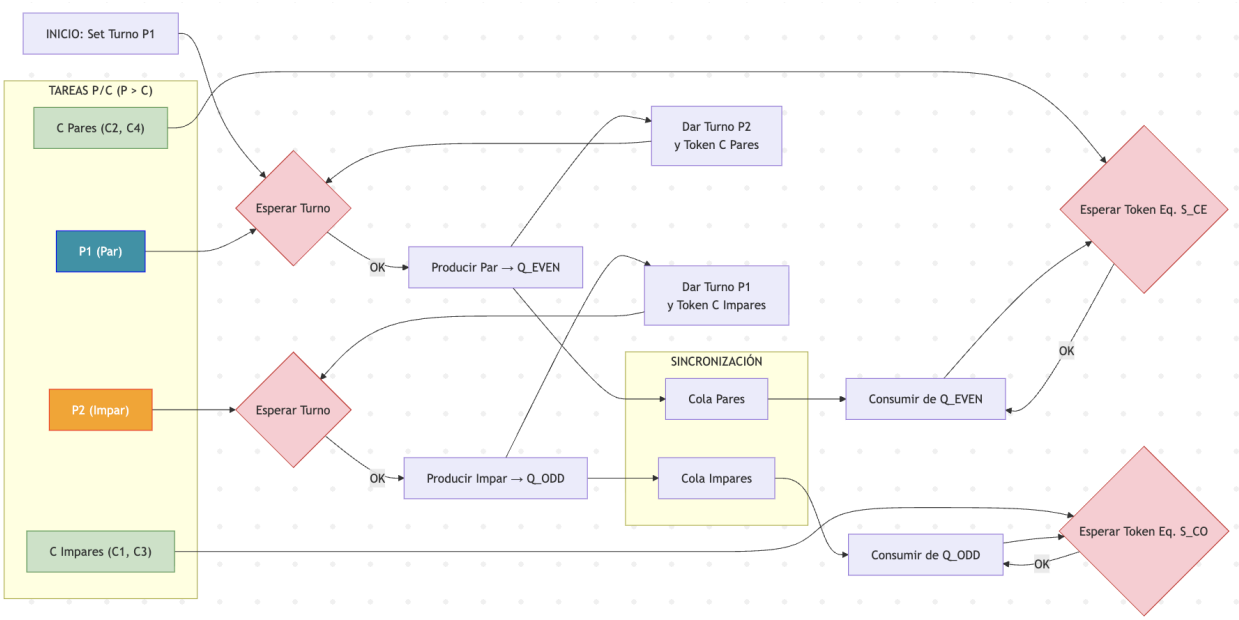
La razón de la similitud se debe a la **Ausencia de Inversión de Prioridad** en el diseño del ejercicio.

1. **Exclusión Mutua Básica:** Ambos objetos cumplen eficazmente la función básica de **exclusión mutua** sobre la sección crítica.
2. **Misma Prioridad:** La ventaja clave del Mutex es su soporte de **Herencia de Prioridad**. Sin embargo, dado que las tareas competidoras (*led_control_task_2* y *led_control_task_3*) tienen la

misma prioridad (Prioridad 5), el escenario de conflicto de prioridades (donde una tarea de prioridad media interrumpe a una de baja que bloquea a una de alta) nunca se produce.

3. **Resultado:** Al no existir un conflicto de prioridades, la característica avanzada del Mutex **nunca se activa**. Por lo tanto, el Mutex funciona en este contexto de manera idéntica al Semáforo Binario.

3. Programa de Productor/Consumidor



El sistema productor/consumidor (P/C) se implementa utilizando **colas separadas** para números pares e impares y una combinación de **semáforos binarios** y **semáforos de conteo** para garantizar la alternancia y la equidad.

Implementación y Sincronización

1. **Productores (P1 y P2)**
   - **Alternancia (Turno):** P1 (pares) y P2 (impares) usan dos **semáforos binarios** (*p1_turn_semaphore* y *p2_turn_semaphore*) para asegurar que solo uno produzca a la vez, garantizando una alternancia estricta. P1 comienza con el *token*.
   - **Prioridad:** Ambos productores se ejecutan con **mayor prioridad** (*PRODUCER_PRIORITY*) que los consumidores, garantizando que las colas se llenen rápidamente.

2. **Colas de Mensajes**
    - Se usan **dos colas** de capacidad 16, una para **pares** (*even_number_queue*) y otra para **impares** (*odd_number_queue*), lo que simplifica la lógica de los consumidores.
3. **Consumidores (C1, C2, C3, C4)**
    - **Encaminamiento:** C1 y C3 leen de la cola de impares. C2 y C4 leen de la cola de pares.
    - **Equidad (Round-Robin):** Se utilizan dos **semáforos de conteo** (*odd_consumer_semaphore* y *even_consumer_semaphore*) con capacidad 2 e inicializados a 0. Cada vez que un productor envía un mensaje, da un *token* al semáforo de conteo del consumidor correspondiente.
    - **Mecanismo de Equidad:** Dado que C1/C3 y C2/C4 tienen la **misma prioridad** (*CONSUMER_PRIORITY*), el planificador de FreeRTOS aplica *time-slicing* (Round-Robin) al distribuir los *tokens* de consumo. Esto obliga a los dos consumidores del mismo tipo a tomar los mensajes de la cola de forma equitativa (50/50), asegurando la oportunidad de consumo para todos.

4. Cálculo de overhead

El objetivo de este ejercicio es **cuantificar el tiempo de *overhead*** introducido por el *Kernel* de FreeRTOS al realizar un **cambio de contexto** entre dos tareas competidoras que utilizan un mecanismo de **exclusión mutua**. Este *overhead* incluye el tiempo de la **interrupción del sistema**, la ejecución del **planificador (*scheduler*)** y el proceso de **guardar el estado de la tarea saliente y cargar el estado de la tarea entrante** (cambio de contexto propiamente dicho).

La medición se implementa sobre el código del Ejercicio 2, donde las tareas *LED2_Control_Task* y *LED3_Control_Task* compiten por un **recurso compartido**, protegido por un **Mutex** (*shared_mutex*). El tiempo de *overhead* se mide desde el instante en que la **tarea saliente libera el Mutex** hasta el instante en que la **Tarea Entrante lo toma con éxito**. Este ciclo incluye:

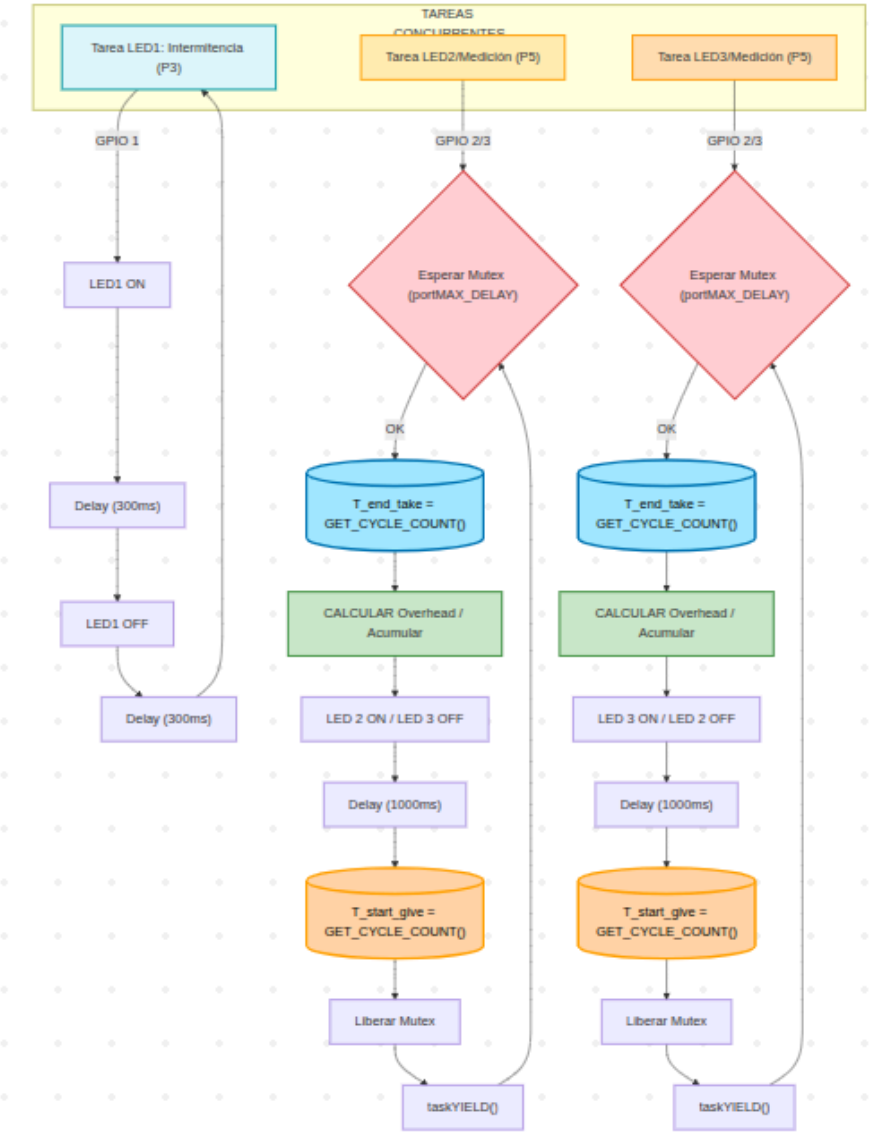
- **Inicio de la Medición (tarea saliente):** Se registra el tiempo en ciclos de CPU (**T_start_give**) inmediatamente antes de llamar a *xSemaphoreGive* en la tarea saliente (p. ej., *LED3_Control_Task*).
- **Transición del Kernel: FreeRTOS toma el control**, ejecuta el *scheduler* y realiza el **cambio de contexto** a la Tarea Entrante (p. ej., *LED2_Control_Task*).

- **Fin de la Medición (tarea entrante):** Se registra el tiempo en ciclos de CPU (**T_end_take**) inmediatamente después de que *xSemaphoreTake* retorna con éxito en la tarea entrante.

El tiempo de overhead (**T_overhead**) en ciclos se calcula como:

$$T\_overhead = T\_end\_take - T\_start\_give$$

El mecanismo del *Kernel* utilizado es el **Contador de Ciclos de CPU**. Se utiliza la función *esp_cpu_get_cycle_count()* para acceder al **contador de ciclos de CPU de 32 bits del ESP32-C6**. A continuación se utilizan algunas variables globales (*T_start_give*, *Total_Mutex_Overhead_Cycles*, *Measurements_Total*) para **transferir el punto de inicio de la medición de una tarea a la otra**.



5. Inversión de Prioridades

En el código inicial, el trabajo largo dentro de la sección crítica de task_low se simula mediante llamadas a *vTaskDelay*. El problema es que *vTaskDelay* **cede el control al scheduler y pone la tarea en estado de bloqueo (Blocked)** por el tiempo especificado. Dado que *task_low* entra en estado **Blocked** repetidamente, el *scheduler* de FreeRTOS tiene la oportunidad de ejecutar *task_medium* (P=5). Como consecuencia, **[MEDIUM] no necesita desalojar a [LOW]** (que ahora tiene P=7 por Inheritance), porque **[LOW] ya está bloqueada por el delay**. **[MEDIUM] se ejecuta libremente**, y el tiempo que **[HIGH] debe esperar es prolongado artificialmente** por la suma de los *delays* y la ejecución de [MEDIUM]. **[HIGH] sigue esperando**, pero el mecanismo de **Priority Inheritance no se puede observar ni demostrar correctamente**, ya que la tarea bloqueadora nunca está en estado **Ready** siendo interrumpida.

Para que el mecanismo de **Herencia de Prioridad** funcione correctamente y se pueda observar el fenómeno, la tarea que tiene el recurso bloqueado (**LOW**) debe simular **un trabajo largo que consuma CPU activamente sin ceder el control al *scheduler***. La corrección consiste en reemplazar las llamadas a *vTaskDelay* dentro de la sección crítica de *task_low* por **un bucle que consuma ciclos de CPU (Busy-Wait)**, como se muestra en el código corregido:

```
for (int i = 0; i < 10; i++) {
      // Bucle que solo consume CPU. LOW (ahora P7) NO puede ser interrumpida por MEDIUM (P5).
    for (uint32_t j = 0; j < LONG_WORK_ITERATIONS; j++) {
        __asm__ __volatile__ ("nop"); // Operación que consume CPU
    }
    printf("[LOW] Treball lent... %d/10\n", i+1);
}
```