Análisis de Modelización y Planificación de Procesos en FreeRTOS para ESP32

Terragno Angel Sebastian Eduardo*, Ramirez Matias Ariel†
Sistemas Operativos y Redes, Universidad Nacional de Río Negro, San Carlos de Bariloche, Argentina
Email: *cheviaset84@gmail.com, †armatias2000@gmail.com

***Resumen*—Este trabajo analiza la implementación de FreeRTOS en microcontroladores ESP32 mediante el framework ESP-IDF, enfocándose en cómo este sistema operativo de tiempo real (RTOS) modeliza y planifica tareas en una arquitectura multinúcleo. Se examina la adaptación para multiprocesamiento simétrico (SMP), detallando sus características clave como núcleos homogéneos y memoria compartida. Se profundiza en la estructura de bloque de control de tareas (TCB), los estados de las tareas y los algoritmos de planificación (prioridad, expropiación, Round Robin adaptativo) utilizados en el contexto dual-core del ESP32. Además, se exploran las estrategias de asignación de memoria dinámicas ofrecidas por FreeRTOS (heap_1 a heap_5) y su evolución en ESP-IDF mediante el sistema multi_heap. Se describe el uso del algoritmo TLSF (Two-Level Segregated Fit), adoptado a partir de ESP-IDF v4.3, como solución eficiente y determinista para la asignación dinámica en entornos de tiempo real. El objetivo es presentar una revisión detallada de estos mecanismos, fundamentales para el diseño de sistemas embebidos robustos sobre plataformas como el ESP32.**

I. Introducción

FreeRTOS es un sistema operativo de tiempo real (RTOS) de código abierto (licencia MIT) ampliamente utilizado en sistemas embebidos y microcontroladores como el ESP32 [3]. Su popularidad radica en su núcleo pequeño, modularidad y portabilidad a diversas arquitecturas. Un RTOS se diferencia de un sistema operativo convencional por su capacidad de garantizar respuestas temporales predecibles (determinismo), crucial en aplicaciones donde el tiempo de ejecución es crítico [1].

El framework ESP-IDF (Espressif IoT development Framework) es el entorno de desarrolló que permite programar dispositivos ESP32, ESP32-S entre otros, aprovechando sus funcionalidades como WI-FI, Bluetooth y otros perifericos. No utiliza la versión estándar de FreeRTOS, sino una variante modificada específicamente para aprovechar la arquitectura de doble núcleo (Xtensa LX7) del ESP32 [2]. Esta adaptación implementa multiprocesamiento simétrico (SMP), permitiendo que las tareas se ejecuten concurrentemente en ambos núcleos. Este estudio se centra en analizar cómo esta versión adaptada de FreeRTOS modeliza las tareas y gestiona su planificación en el entorno SMP del ESP32.

FreeRTOS tiene un núcleo liviano y modular, escrito principalmente en C para facilitar la portabilidad. Su planificación se basa en listas de tareas (Task), listas de espera (List) y colas de mensajes (Queue), que permiten la comunicación y sincronización entre tareas. El cambio de contexto lo gestiona un planificador activado por un tick periódico (configurable). Para adaptarse a distintas arquitecturas, FreeRTOS usa ports específicos que definen cómo se manejan interrupciones, registros y funciones críticas, algunas escritas en assembler. Estos ports están organizados en el directorio portable/, y permiten que el mismo núcleo funcione en microcontroladores ARM, RISC-V, x86, ESP32, entre otros.

II. Marco teórico

II-A. Multiprocesamiento Simétrico (SMP) en ESP32

Los microcontroladores ESP32 implementan una arquitectura SMP de doble núcleo con características específicas que FreeRTOS debe gestionar [5]:

- **Núcleos Homogéneos:** Denominados Core 0 y Core 1, son procesadores idénticos, lo que significa que el mismo código binario puede ejecutarse en cualquiera de ellos sin modificaciones, simplificando el desarrollo y la planificación.
- **Memoria Compartida:** La mayor parte de la memoria es accesible por ambos núcleos. El hardware se encarga de asegurar que los accesos simultáneos a la misma dirección de memoria se manejen en orden, usando el bus de memoria. Para operaciones que requieren atomicidad estricta (lectura-modificación-escritura), se utilizan instrucciones atómicas [1]. específicas del ISA (Instruction Set Architecture), como compare-and-swap, que FreeRTOS utiliza internamente para proteger sus estructuras de datos críticas.
- **Interrupciones Cruzadas (Inter-Processor Interrupts - IPI):** El sistema permite que un núcleo envíe una interrupción directamente al otro. FreeRTOS utiliza este mecanismo para tareas como solicitar un cambio de contexto en el otro núcleo (por ejemplo, si una tarea de alta prioridad se desbloquea y debe ejecutarse allí) o para la sincronización entre núcleos.

II-B. Adaptación SMP de FreeRTOS en ESP-IDF

La versión de FreeRTOS en ESP-IDF explota estas características SMP, pero introduce una gestión asimétrica de ciertas responsabilidades para optimizar y evitar condiciones de carrera:

- **Núcleo 0 (Core 0):** Actúa como el procesador principal del tiempo, asumiendo responsabilidad completa sobre:

[1]Se pueden considerar como una instrucción formada por muchas instrucciones pero que se ejecutan desde el principio hasta el final, sin ser interrumpidas

- **Mantenimiento del tiempo global:** Actualiza el contador central de ticks que sirve como referencia temporal para todo el sistema
- **Gestión de despertar:** Identifica y reactiva las tareas cuyo período de bloqueo temporal ha concluido
- **Planificación global:** Ejecuta el algoritmo de planificación si alguna tarea cambia su estado a Ready"
- **Hooks de usuario:** Invoca la función vApplicationTickHook() si el desarrollador la ha implementado. Las funciones Hooks son rutinas que el desarrollador de la aplicacion puede rregistrar para que se ejecuten automaticamente cuando el planificador no tiene tareas listas para ejecutar en algun nucleo determinado permitiendo aprovechar el tiempo de CPU que de otra manera se desperdiciaria en unbucle de espera (Tarea Idle).

.

- **Núcleo 1 (Core 1):** El Núcleo 1 mantiene un rol más limitado respecto al manejo de interrupciones de tick:
  - No modifica el contador global de ticks
  - No interviene en el despertar de tareas bloqueadas por tiempo
  - No participa en la planificación global del sistema

  Sin embargo, el Núcleo 1 sí asume responsabilidades locales:
  - **Time slicing local:** Gestiona la rotación de tareas de igual prioridad dentro de su propio contexto de ejecución
  - **Hooks de usuario:** También invoca vApplicationTickHook() si está configurada, permitiendo que el código de usuario se ejecute periódicamente en ambos núcleos

II-C. Gestión de Memoria Dinámica en FreeRTOS y ESP-IDF

En sistemas operativos embebidos como FreeRTOS, el manejo eficiente de memoria dinámica es fundamental para garantizar determinismo y evitar errores como la fragmentación o fugas de memoria. FreeRTOS proporciona cinco estrategias distintas para gestionar el heap (montículo), definidas en tiempo de compilación mediante el parámetro `configFRTOS_MEMORY_SCHEME`:

- `heap_1`: Asignación secuencial sin liberación. Muy simple y determinista. No soporta `free()`.
- `heap_2`: Asignación dinámica con liberación, usando una estrategia de "mejor ajuste". (Obsoleto).
- `heap_3`: Utiliza `malloc()` y `free()` de la biblioteca estándar (`newlib`). Aporta compatibilidad, pero no garantiza predictibilidad.
- `heap_4`: Similar a `heap_2` (utiliza "primer ajuste"), pero con mejor manejo de fragmentación y coalescencia de bloques libres.
- `heap_5`: Permite manejar múltiples regiones de memoria no contiguas, basado en `heap_4`. Ideal para microcontroladores con RAM fragmentada.

Estas estrategias funcionan bien en arquitecturas simples. Sin embargo, en plataformas más complejas como el ESP32, con varios tipos de memoria (DRAM, IRAM, PSRAM), se requiere una gestión más sofisticada.

(NOTA: Una fuga de memoria ocurre cuando un programa reserva memoria dinámicamente (por ejemplo, con `malloc()`), pero luego pierde toda referencia a esa memoria sin liberarla con `free()`. La memoria queda ocupada pero no puede reutilizarse ni accederse, porque ya no existe un puntero válido que la apunte.)

II-D. Sistema `multi_heap` y el algoritmo TLSF en ESP-IDF

ESP-IDF reemplaza las estrategias tradicionales de FreeRTOS con su propio sistema de asignación de memoria llamado `multi_heap`, que permite gestionar múltiples regiones de memoria con distintas capacidades (`MALLOC_CAP_DMA`, `MALLOC_CAP_EXEC`, etc.) y tipos (interna, externa, IRAM, etc.).

Hasta la versión ESP-IDF v4.2, `multi_heap` usaba algoritmos simples como "primer ajuste"(first fit). A partir de ESP-IDF v4.3, se incorpora el algoritmo TLSF (Two-Level Segregated Fit) como núcleo del sistema de asignación.

II-D1. Algoritmo TLSF: TLSF es un algoritmo de tiempo constante $O(1)$ para las operaciones de `malloc()` y `free()`, ideal para sistemas de tiempo real. Sus características principales son:

- Clasifica los bloques de memoria usando dos niveles de índices: FLI (First Level Index) y SLI (Second Level Index), permitiendo una búsqueda rápida del bloque más ajustado.
- Mantiene listas enlazadas segmentadas por tamaño y bitmaps para una localización eficiente de bloques libres.
- Soporta la coalescencia (fusión) de bloques libres adyacentes de manera inmediata tras la liberación, reduciendo la fragmentación externa.
- Permite gestionar múltiples regiones de heap gracias a su integración con la arquitectura de `multi_heap` en ESP-IDF.

Cuando se realiza una asignación con `heap_caps_malloc(size, caps)`, ESP-IDF busca un heap compatible con las capacidades solicitadas y llama internamente a `tlsf_malloc()`, que ubica el bloque más adecuado en tiempo constante. Esta integración permite al ESP32 bajo FreeRTOS mantener una gestión de memoria eficiente, determinista y adecuada para entornos con restricciones temporales estrictas.

III. Modelización de Procesos

El planificador de ESP-IDF FreeRTOS es *preventivo* (preemptive), basado en *prioridades fijas* y con soporte para *time slicing* (Round Robin) [5]. Lo que significa que:

- Cada tarea recibe una prioridad constante al crearse. El planificador ejecuta la tarea lista con mayor prioridad.

- El planificador puede cambiar la ejecución a otra tarea sin la cooperación de la tarea que se está ejecutando actualmente.
- El planificador alterna periódicamente la ejecución entre tareas listas con la misma prioridad mediante un sistema de turnos rotatorios. La segmentación de tiempo se rige por una interrupción de tick.

La arquitectura multinúcleo del ESP32 introduce adaptaciones importantes.

III-A. Planificación Multinúcleo: Prioridad y Afinidad

Cada núcleo ejecuta su propio ciclo de planificación independiente pero coordinado. Al seleccionar la siguiente tarea, un núcleo siempre buscará ejecutar la tarea de mayor prioridad en estado 'Ready', pero con dos condiciones adicionales impuestas por SMP:

1. **Compatibilidad de Afinidad:** La tarea debe tener permiso para ejecutarse en ese núcleo específico (controlado por 'uxCoreAffinityMask').
2. **Disponibilidad:** La tarea no debe estar ejecutándose simultáneamente en el otro núcleo.

Esto significa que el sistema no siempre ejecutará las dos tareas de mayor prioridad global si, por ejemplo, ambas tienen afinidad por el mismo núcleo.

III-B. Expropiación (Preemption) Multinúcleo

La expropiación (preemption) se mantiene: si una tarea de mayor prioridad que la actual en un núcleo pasa a 'Ready' y es compatible, desalojará a la de menor prioridad. La adaptación SMP clave ocurre cuando una tarea de alta prioridad *sin afinidad específica* ('unpinned') se desbloquea. Si ambos núcleos ejecutan tareas de menor prioridad, ESP-IDF aplica una **"preferencia local"**: el núcleo donde se originó el evento que desbloqueó a la tarea de alta prioridad (por ejemplo, el núcleo que ejecutaba la tarea que liberó un semáforo) será el que intente ejecutarla primero, realizando la expropiación si es necesario. Si ese núcleo no puede (quizás ya ejecuta otra tarea de igual o mayor prioridad), entonces se señalará al otro núcleo (vía IPI) para que intente ejecutarla. Esta estrategia busca minimizar la latencia y la comunicación inter-núcleo, asumiendo que los datos relevantes podrían estar ya en la caché del núcleo local.

III-C. Round Robin adaptativo de multinúcleo

Para tareas de igual prioridad, FreeRTOS usa Round Robin (basado en 'time slicing' por el 'tick') para compartir lo más equitativamente posible el tiempo de CPU. En ESP-IDF, debido a la afinidad y ejecución paralela, no se garantiza un Round Robin perfecto y global. Se implementa un "Best Effort Round Robin"[3]:

- Cada núcleo, en su 'tick', intenta rotar entre las tareas de igual prioridad más alta listas y compatibles con él.
- Una tarea que consume su 'time slice' y sigue lista, pasa al final de la lista de su prioridad.
- Un núcleo puede 'saltarse' tareas en la lista si tienen afinidad exclusiva con el otro núcleo o ya corren allí.

Ejemplo ilustrativo:

- **AX:** Tarea A sin afinidad específica (X = unpinned)
- **B0:** Tarea B asignada al Núcleo 0
- **C1:** Tarea C asignada al Núcleo 1
- **D0:** Tarea D asignada al Núcleo 0

Evolución del planificador:

1. **Estado inicial:**
   Lista: [AX, B0, C1, D0]
2. **Interrupción de tick en Núcleo 0:**
   - Selecciona tarea A (compatible con cualquier núcleo)
   - Nueva lista: [B0, C1, D0, AX]
3. **Interrupción de tick en Núcleo 1:**
   - B0 es incompatible (restringida al Núcleo 0) $\rightarrow$ se omite
   - Selecciona tarea C1 (compatible con Núcleo 1)
   - Nueva lista: [B0, D0, AX, C1]
4. **Nueva interrupción de tick en Núcleo 0:**
   - Selecciona tarea B0 (compatible con Núcleo 0)
   - Nueva lista: [D0, AX, C1, B0]
5. **Nueva interrupción de tick en Núcleo 1:**
   - D0 es incompatible (restringida al Núcleo 0) $\rightarrow$ se omite
   - Selecciona tarea AX (sin restricciones de núcleo)
   - Nueva lista: [D0, C1, B0, AX]

III-D. Consideraciones Prácticas

- **Comportamiento no secuencial:** A diferencia del FreeRTOS original, no se puede garantizar una ejecución estrictamente secuencial de tareas de igual prioridad.
- **Omisión selectiva:** El planificador puede necesitar omitir tareas incompatibles con el núcleo actual debido a restricciones de afinidad.
- **Garantía de progreso:** Con suficientes interrupciones de tick, todas las tareas eventualmente recibirán tiempo de CPU.
- **Descenso de prioridad:** Si un núcleo no encuentra tareas ejecutables en el nivel de prioridad más alto, descenderá a niveles inferiores en busca de tareas elegibles.

III-E. Influencia del Programador y Resumen de Estrategias

El desarrollador guía al planificador mediante la configuración de prioridades, afinidad, bloqueos y retardos. La Tabla II resume las estrategias en el contexto SMP de ESP-IDF.

Cuadro I: Resumen de Estrategias de Planificación en ESP-IDF FreeRTOS (SMP)

Estrategia	Comportamiento SMP	Restricción / Consideración Clave
Prioridad Fija	Cada núcleo ejecuta la tarea lista compatible de mayor prioridad.	La afinidad de núcleo puede impedir que la tarea de mayor prioridad global se ejecute si está 'pinned' a un núcleo ocupado.
Expropiación (Preemption)	Un núcleo puede ser expropiado por una tarea compatible de mayor prioridad. Preferencia local para tareas 'unpinned'.	La expropiación solo ocurre si la tarea entrante es compatible con el núcleo y tiene estrictamente mayor prioridad.
Round Robin (Time Slicing)	"Mejor esfuerzo"por núcleo para rotar tareas de igual prioridad compatibles.	No garantiza orden global estricto debido a afinidad y ejecución paralela. Puede requerir 'saltar' tareas.

IV. Planificación de Procesos

El planificador de ESP-IDF FreeRTOS es *preventivo* (preemptive), basado en *prioridades fijas* y con soporte para *time slicing* (Round Robin) [5]. Lo que significa que:

- Cada tarea recibe una prioridad constante al crearse. El planificador ejecuta la tarea lista con mayor prioridad.
- El planificador puede cambiar la ejecución a otra tarea sin la cooperación de la tarea que se está ejecutando actualmente.
- El planificador alterna periódicamente la ejecución entre tareas listas con la misma prioridad mediante un sistema de turnos rotatorios. La segmentación de tiempo se rige por una interrupción de tick.

La arquitectura multinúcleo del ESP32 introduce adaptaciones importantes.

IV-A. Planificación Multinúcleo: Prioridad y Afinidad

Cada núcleo ejecuta su propio ciclo de planificación independiente pero coordinado. Al seleccionar la siguiente tarea, un núcleo siempre buscará ejecutar la tarea de mayor prioridad en estado 'Ready', pero con dos condiciones adicionales impuestas por SMP:

1. **Compatibilidad de Afinidad:** La tarea debe tener permiso para ejecutarse en ese núcleo específico (controlado por 'uxCoreAffinityMask').
2. **Disponibilidad:** La tarea no debe estar ejecutándose simultáneamente en el otro núcleo.

Esto significa que el sistema no siempre ejecutará las dos tareas de mayor prioridad global si, por ejemplo, ambas tienen afinidad por el mismo núcleo.

IV-B. Expropiación (Preemption) Multinúcleo

La expropiación (preemption) se mantiene: si una tarea de mayor prioridad que la actual en un núcleo pasa a 'Ready' y es compatible, desalojará a la de menor prioridad. La adaptación SMP clave ocurre cuando una tarea de alta prioridad *sin afinidad específica* ('unpinned') se desbloquea. Si ambos núcleos ejecutan tareas de menor prioridad, ESP-IDF aplica una **"preferencia local"**: el núcleo donde se originó el evento que desbloqueó a la tarea de alta prioridad (por ejemplo, el núcleo que ejecutaba la tarea que liberó un semáforo) será el que intente ejecutarla primero, realizando la expropiación si es necesario. Si ese núcleo no puede (quizás ya ejecuta otra tarea de igual o mayor prioridad), entonces se señalará al otro núcleo (vía IPI) para que intente ejecutarla. Esta estrategia busca minimizar la latencia y la comunicación inter-núcleo, asumiendo que los datos relevantes podrían estar ya en la caché del núcleo local.

IV-C. Round Robin adaptativo de multinúcleo

Para tareas de igual prioridad, FreeRTOS usa Round Robin (basado en 'time slicing' por el 'tick') para compartir lo más equitativamente posible el tiempo de CPU. En ESP-IDF, debido a la afinidad y ejecución paralela, no se garantiza un Round Robin perfecto y global. Se implementa un "Best Effort Round Robin"[3]:

- Cada núcleo, en su 'tick', intenta rotar entre las tareas de igual prioridad más alta listas y compatibles con él.
- Una tarea que consume su 'time slice' y sigue lista, pasa al final de la lista de su prioridad.
- Un núcleo puede 'saltarse' tareas en la lista si tienen afinidad exclusiva con el otro núcleo o ya corren allí.

Ejemplo ilustrativo:

- **AX:** Tarea A sin afinidad específica (X = unpinned)
- **B0:** Tarea B asignada al Núcleo 0
- **C1:** Tarea C asignada al Núcleo 1
- **D0:** Tarea D asignada al Núcleo 0

Evolución del planificador:

1. **Estado inicial:**
   Lista: [AX, B0, C1, D0]
2. **Interrupción de tick en Núcleo 0:**
   - Selecciona tarea A (compatible con cualquier núcleo)
   - Nueva lista: [B0, C1, D0, AX]
3. **Interrupción de tick en Núcleo 1:**
   - B0 es incompatible (restringida al Núcleo 0) $\rightarrow$ se omite
   - Selecciona tarea C1 (compatible con Núcleo 1)
   - Nueva lista: [B0, D0, AX, C1]
4. **Nueva interrupción de tick en Núcleo 0:**
   - Selecciona tarea B0 (compatible con Núcleo 0)
   - Nueva lista: [D0, AX, C1, B0]
5. **Nueva interrupción de tick en Núcleo 1:**
   - D0 es incompatible (restringida al Núcleo 0) $\rightarrow$ se omite
   - Selecciona tarea AX (sin restricciones de núcleo)
   - Nueva lista: [D0, C1, B0, AX]

IV-D. Consideraciones Prácticas

- **Comportamiento no secuencial:** A diferencia del FreeRTOS original, no se puede garantizar una ejecución estrictamente secuencial de tareas de igual prioridad.

- **Omisión selectiva:** El planificador puede necesitar omitir tareas incompatibles con el núcleo actual debido a restricciones de afinidad.
- **Garantía de progreso:** Con suficientes interrupciones de tick, todas las tareas eventualmente recibirán tiempo de CPU.
- **Descenso de prioridad:** Si un núcleo no encuentra tareas ejecutables en el nivel de prioridad más alto, descenderá a niveles inferiores en busca de tareas elegibles.

IV-E. Influencia del Programador y Resumen de Estrategias

El desarrollador guía al planificador mediante la configuración de prioridades, afinidad, bloqueos y retardos. La Tabla II resume las estrategias en el contexto SMP de ESP-IDF.

Cuadro II: Resumen de Estrategias de Planificación en ESP-IDF FreeRTOS (SMP)

Estrategia	Comportamiento SMP	Restricción / Consideración Clave
Prioridad Fija	Cada núcleo ejecuta la tarea lista compatible de mayor prioridad.	La afinidad de núcleo puede impedir que la tarea de mayor prioridad global se ejecute si está 'pinned' a un núcleo ocupado.
Expropiación (Preemption)	Un núcleo puede ser expropiado por una tarea compatible de mayor prioridad. Preferencia local para tareas 'unpinned'.	La expropiación solo ocurre si la tarea entrante es compatible con el núcleo y tiene estrictamente mayor prioridad.
Round Robin (Time Slicing)	"Mejor esfuerzo"por núcleo para rotar tareas de igual prioridad compatibles.	No garantiza orden global estricto debido a afinidad y ejecución paralela. Puede requerir 'saltar' tareas.

V. Manejo y Asignación de Memoria

El manejo de memoria dinámica constituye uno de los aspectos más críticos en sistemas operativos de tiempo real, especialmente en sistemas embebidos con recursos limitados como el ESP32. FreeRTOS ofrece múltiples estrategias para gestionar el heap, mientras que ESP-IDF extiende estas capacidades mediante un sistema más robusto y determinista basado en múltiples regiones de memoria y el algoritmo TLSF.

V-A. Estrategias de Heap en FreeRTOS

FreeRTOS implementa la gestión de memoria como parte de la capa portable, separándola del núcleo del sistema operativo. Esta arquitectura responde a la diversidad de requisitos de los sistemas embebidos en términos de asignación dinámica y restricciones temporales. FreeRTOS abstrae la gestión mediante `pvPortMalloc()` y `pvPortFree()`, análogas a `malloc()` y `free()` estándar.

V-A1. `heap_1`: Asignación Secuencial Determinista: La implementación `heap_1` es la más básica, diseñada para sistemas donde todos los objetos del kernel se crean antes de iniciar el planificador y nunca se liberan. Opera subdividiendo secuencialmente un array estático (`uint8_t`) definido por `configTOTAL_HEAP_SIZE`. No implementa `vPortFree()`.

V-A1a. Características: Determinismo absoluto ($O(1)$ en asignación), ausencia de fragmentación, simplicidad, consumo predecible (todo el heap se asigna estáticamente). Es ideal para aplicaciones críticas que prohíben la asignación dinámica post-inicialización.

V-A1b. Consideraciones: Cada tarea creada dinámicamente usa dos llamadas a `pvPortMalloc()` (TCB y pila).

V-A2. `heap_2`: Mejor Ajuste con Liberación (Obsoleto): **Nota importante:** `heap_2` ha sido reemplazado por `heap_4`. Se mantiene por compatibilidad con versiones anteriores. `heap_2` subdivide un array estático (`configTOTAL_HEAP_SIZE`) e implementa `pvPortMalloc()` y `vPortFree()` usando un algoritmo de *mejor ajuste*: busca el bloque libre más pequeño que satisfaga la solicitud. Si un bloque es más grande, se divide.

V-A2a. Limitación Crítica: No puede combinar bloques libres adyacentes (coalescencia), lo que lleva a una alta fragmentación. Solo adecuado si los bloques asignados y liberados son siempre del mismo tamaño o para sistemas legados (desactualizados).

V-A3. `heap_3`: Delegación a Biblioteca Estándar: `heap_3` delega la gestión a `malloc()` y `free()` de la biblioteca C estándar (e.g., `newlib`). El tamaño del heap se define en la configuración del enlazador, no por `configTOTAL_HEAP_SIZE`.

V-A3a. Característica Clave: Thread-Safety: `heap_3` añade seguridad para hilos suspendiendo temporalmente el planificador de FreeRTOS durante las llamadas a `malloc()`/`free()`, previniendo condiciones de carrera.

V-A3b. Consideraciones: El rendimiento y determinismo dependen de la implementación de `malloc()`/`free()` de la biblioteca. La suspensión del planificador puede afectar la respuesta en tiempo real.

V-A4. `heap_4`: Primer Ajuste con Coalescencia: `heap_4` es una mejora de `heap_2`, utilizando un algoritmo de *primer ajuste* (first-fit) y, crucialmente, *coalescencia*. Opera sobre un array estático (`configTOTAL_HEAP_SIZE`).

V-A4a. Mecanismo de Asignación y Coalescencia: El primer ajuste selecciona el primer bloque libre encontrado que sea suficientemente grande. La coalescencia fusiona automáticamente bloques libres adyacentes cuando `vPortFree()` es llamado, examinando bloques anterior y posterior. Esto reduce significativamente la fragmentación.

V-A4b. Características: Minimiza fragmentación, versátil para tamaños variables, generalmente más rápido que `malloc()` estándar, pero no estrictamente determinista. Es una opción robusta y recomendada si la aplicación asigna y libera bloques de memoria de varios tamaños.

V-A5. `heap_5`: Múltiples Regiones No Contiguas: `heap_5` extiende `heap_4` para gestionar múltiples regiones de memoria RAM físicamente separadas y no contiguas. Utiliza el mismo algoritmo de primer ajuste y coalescencia.

V-A5a. Inicialización Obligatoria: Requiere inicialización explícita con `vPortDefineHeapRegions()` antes de crear cualquier objeto del kernel. Esta función recibe un

array de estructuras `HeapRegion_t`, cada una definiendo la dirección de inicio y tamaño de una región.

Listing 1: Estructura HeapRegion_t para heap_5

```
typedef struct HeapRegion {
    uint8_t *pucStartAddress;
// Direccion de inicio del bloque de memoria
    size_t xSizeInBytes;
// Tamanio en bytes de la region
} HeapRegion_t;
```

V-A5b. Gestión y Casos de Uso: Ideal para arquitecturas con RAM interna/externa (SRAM, PSRAM) o memoria fragmentada. El algoritmo trata las múltiples regiones como un heap lógico único. Definir regiones como arrays estáticos mejora la robustez (direcciones resueltas por el enlazador, previene solapamientos).

V-B. Sistema `multi_heap` en ESP-IDF

ESP-IDF introduce `multi_heap` para la arquitectura de memoria heterogénea del ESP32, que incluye:

- **DRAM (Data RAM)**: Memoria principal para datos y heap de aplicación.
- **IRAM (Instruction RAM)**: Memoria rápida para código crítico (ej. interrupciones).
- **PSRAM (Pseudo-Static RAM)**: Memoria externa opcional para expandir capacidad.
- **RTC Memory**: Memoria de bajo consumo para persistencia en modos deep sleep.

`multi_heap` permite definir heaps en estas diferentes memorias, cada una con atributos o "capacidades" (`MALLOC_CAP_DMA`, `MALLOC_CAP_EXEC`, `MALLOC_CAP_SPIRAM`, etc.).

V-C. Evolución hacia el Algoritmo TLSF en ESP-IDF

Desde ESP-IDF v4.3, TLSF se convirtió en el algoritmo central de `multi_heap`, mejorando el determinismo y la eficiencia en la gestión de memoria para aplicaciones de tiempo real en el ESP32.

V-C1. Características Fundamentales del Algoritmo TLSF:

V-C1a. Determinismo Temporal Garantizado y Arquitectura de Dos Niveles: TLSF asegura operaciones `malloc()` y `free()` en tiempo constante $O(1)$, sin bucles ni recursión, esencial para sistemas de tiempo real. Su arquitectura de mapeo de dos niveles es clave:

- FLI (First Level Index): Clasifica bloques por tamaño usando una representación logarítmica (base 2).

$$f = \lfloor \log_2(\text{size}) \rfloor$$

.

- SLI (Second Level Index): Implementa una subdivisión lineal dentro de cada clase FLI.

$$s = \left\lfloor \frac{\text{size} - 2^f}{2^f / 2^{\text{SLI\_param}}} \right\rfloor$$

, donde SLI_param es configurable (usualmente 4 o 5, definiendo $2^4 = 16$ o $2^5 = 32$ subdivisiones).

Esta estructura, junto con el uso de bitmaps para marcar listas de bloques libres no vacías e instrucciones de hardware como `ffs()` (find first set), permite una búsqueda extremadamente rápida.

V-C1b. La Matriz de Listas de Dos Niveles y su Dimensión: La estructura central del algoritmo se representa como una matriz de cabeceras de listas doblemente enlazadas, cuya dimensión (FLI x SLI) es el resultado directo de la jerarquía de dos niveles que utiliza para clasificar los bloques de memoria libre.

- **Primer Nivel (FLI - First Level Index):** Realiza una clasificación logarítmica y general, dividiendo el espacio de memoria en "clases de tamaño" basadas en potencias de 2. El FLI determina la primera dimensión (las filas) de la matriz y permite al algoritmo localizar rápidamente la clase de tamaño general de un bloque. Por ejemplo, `FLI=6` agrupa todos los bloques en el rango de [64, 127] bytes.
- **Segundo Nivel (SLI - Second Level Index):** Toma cada clase general del FLI y la subdivide linealmente en un número fijo de subclases más precisas (usualmente 8 o 16). El SLI determina la segunda dimensión (las columnas) de la matriz. Por ejemplo, dentro de `FLI=6`, la subclase `SLI=0` podría corresponder al rango [64-71] bytes, `SLI=1` a [72-79] bytes, y así sucesivamente.
- **Listas Doblemente Enlazadas:** Cada celda `[i][j]` de la matriz actúa como la cabecera (puntero) de una lista doblemente enlazada. Esta lista agrupa todos los bloques de memoria actualmente libres cuyo tamaño corresponde al rango definido por esa combinación específica de `FLI` y `SLI`. La elección de una lista *doblemente* enlazada es crucial, ya que permite la inserción y remoción de bloques en tiempo constante ($O(1)$), una característica esencial para el rendimiento determinista del algoritmo.

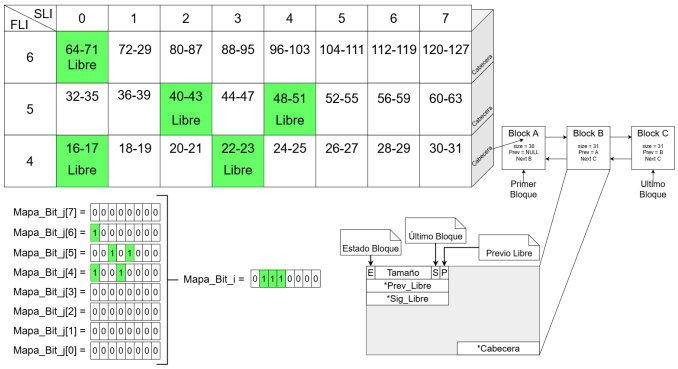

Figura 1: Representación de la Matriz de Listas de Dos Niveles, los Nodos de las Listas Doblemente Enlazadas, y los Bitmaps.

La matriz `listas[FLI][SLI]` es una tabla de búsqueda directa. Dado un tamaño de bloque, el algoritmo calcula sus

índices `i` y `j` en $O(1)$, y accede inmediatamente a la cabecera de la lista correcta. Esta organización, apoyada por mapas de bits que indican qué listas no están vacías, es la que permite a TLSF garantizar sus operaciones en tiempo constante.

V-C1c. Gestión Avanzada de Fragmentación y Estructuras de Bloques: TLSF combate la fragmentación mediante coalescencia inmediata y una estrategia "good-fit" (retorna el bloque más pequeño que satisface la solicitud). La fragmentación interna y externa están matemáticamente acotadas. Los bloques de memoria (libres u ocupados) contienen metadatos:

- **Bloque Libre**: Incluye tamaño, puntero al bloque físico anterior (boundary tag para coalescencia), y punteros al siguiente y anterior bloque libre en su lista segregada. También bits de estado (libre/ocupado, último bloque físico).
- **Bloque Ocupado**: Requiere menos metadatos, principalmente el tamaño y el boundary tag del bloque físico anterior para facilitar la coalescencia cuando se libere.

El tamaño mínimo de bloque (MBS) suele ser de 16 bytes para acomodar estos metadatos.

V-C1d. Parámetros de Configuración TLSF:

- **FLI**: Se calcula durante la inicialización basado en el tamaño del pool: FLI $=$ $\text{mín}(\lfloor \log_2(\text{memory\_pool\_size}) \rfloor, 31)$.
- **SLI (parámetro)**: Definido por el usuario (e.g., 4 o 5), debe ser potencia de 2.

V-C1e. Operaciones Fundamentales TLSF (Simplificado): Algoritmo `malloc(size)`:

1. Mapear 'size' a índices 'fl' y 'sl' ($O(1)$).
2. Buscar un bloque adecuado en la lista 'free_lists[fl][sl]' o en la siguiente lista no vacía usando bitmaps ($O(1)$).
3. Si se encuentra un bloque: removerlo de la lista libre ($O(1)$).
4. Si el bloque es más grande que 'size', dividirlo. El remanente se reinserta en la lista de libres apropiada ($O(1)$).
5. Marcar el bloque como ocupado y retornarlo.

Algoritmo `free(ptr)`:

1. Obtener el puntero al inicio del bloque a partir de 'ptr' y marcarlo como libre ($O(1)$).
2. Fusionar (coalesce) con el bloque físico posterior si está libre ($O(1)$).
3. Fusionar (coalesce) con el bloque físico anterior si está libre ($O(1)$).
4. Mapear el tamaño del nuevo bloque (posiblemente fusionado) a 'fl' y 'sl' ($O(1)$).
5. Insertar el bloque en la lista 'free_lists[fl][sl]' ($O(1)$).

V-D. Integración TLSF con `multi_heap` en ESP-IDF

V-D1. Flujo de Asignación Integrado: Cuando se invoca `heap_caps_malloc(size, caps)`:

1. El sistema `multi_heap` identifica los heaps disponibles que cumplen con las 'caps' (capacidades) solicitadas (e.g., `MALLOC_CAP_DMA`, `MALLOC_CAP_INTERNAL`).
2. Para cada heap compatible, se invoca internamente `tlsf_malloc()` sobre la región de memoria de ese heap.
3. TLSF localiza el bloque óptimo en $O(1)$.
4. Si la asignación falla en un heap (por falta de espacio o fragmentación que impide satisfacer la solicitud), `multi_heap` puede intentar automáticamente con el siguiente heap compatible, si la política lo permite.
5. Se retorna un puntero al bloque asignado o `NULL` si no hay memoria suficiente en ningún heap compatible.

V-D2. Ventajas del Sistema Integrado:

V-D2a. Para Desarrolladores:

- API unificada (`heap_caps_malloc`) que abstrae la complejidad de múltiples tipos de memoria.
- Especificación declarativa de requisitos de memoria mediante capacidades.
- Compatibilidad con funciones estándar de C (`malloc`, `free` usualmente redirigen a `heap_caps_malloc` con capacidades por defecto).
- Herramientas de depuración avanzadas en ESP-IDF para fugas y fragmentación.

V-D2b. Para el Sistema:

- Determinismo temporal garantizado por TLSF.
- Eficiencia de memoria superior y fragmentación controlada.
- Escalabilidad: el rendimiento no se degrada significativamente con múltiples regiones o tamaño del pool.
- Sincronización eficiente para compatibilidad SMP (multiprocesamiento simétrico) en ESP32.

V-E. Análisis de Rendimiento y Fragmentación

V-E1. Complejidad Temporal Formal: El análisis formal de TLSF confirma que todas las operaciones fundamentales (`malloc`, `free`, `realloc`) tienen una complejidad temporal de $O(1)$ en el peor caso. Esto se logra evitando búsquedas lineales o bucles dependientes del número de bloques.

V-E2. Análisis Matemático de Fragmentación: La fragmentación en TLSF está matemáticamente acotada. La fragmentación externa (espacio libre inutilizable entre bloques ocupados) se minimiza por la coalescencia inmediata y la estrategia de ajuste. La fragmentación interna (espacio desperdiciado dentro de un bloque asignado porque es más grande que lo solicitado) también es limitada. La fórmula de fragmentación máxima mencionada en el texto original,

$$\text{Fragmentación máxima} = \frac{2^{\text{FLI}_{\text{pool}}} - \text{size}}{2^{\text{SLI}_{\text{param}}}} - 2$$

, es una simplificación o interpretación específica. En la práctica, la fragmentación total depende de muchos factores, pero el diseño de TLSF busca mantenerla baja. Un ejemplo para un ESP32 con 4MB de RAM (FLI_{pool} tal que $2^{\text{FLI}_{\text{pool}}} \approx 4MB = 2^{22}$ bytes) y $\text{SLI}_{\text{param}} = 5$ (32 subdivisiones por clase FLI) podría dar una fragmentación máxima por "mala elección" de clase de unos $\frac{2^{22}}{32} \approx 128$KB en un escenario muy específico, representando un pequeño porcentaje del pool total.

V-E3. Resultados Experimentales y Comparativos: Estudios y benchmarks comparativos (como los referenciados por los autores de TLSF y en contextos de RTOS) demuestran consistentemente que TLSF:

- Mantiene tiempos de respuesta bajos y acotados, sin los picos impredecibles de otros algoritmos.
- Exhibe una fragmentación significativamente menor que algoritmos como First-Fit, Best-Fit o sistemas Buddy tradicionales en muchos escenarios de carga.
- Ofrece una eficiencia computacional superior, siendo más rápido que implementaciones estándar de `malloc`/`free` en bibliotecas C comunes para muchos casos de uso.
- Escala bien con el tamaño del pool de memoria y el número de bloques activos.

La adopción de TLSF en ESP-IDF es un avance importante, permitiendo a los desarrolladores construir sistemas embebidos fiables y de alto rendimiento sobre el ESP32.

VI. Conclusiones

El análisis de FreeRTOS en el ESP32 a través de ESP-IDF revela una adaptación sofisticada de un RTOS clásico a una arquitectura embebida multinúcleo. Los puntos clave son:

- **Adaptación especializada para arquitectura multinúcleo:** ESP-IDF FreeRTOS se ha modificado respecto a la versión estándar para aprovechar los dos núcleos del ESP32, implementando un sistema de multiprocesamiento simétrico (SMP) que permite una distribución eficiente de tareas entre procesadores.
- **Distribución asimétrica de responsabilidades:** A pesar de tener núcleos homogéneos, el Core 0 asume funciones críticas del sistema como la gestión temporal global, mientras que el Core 1 tiene un rol más limitado.
- **Planificación adaptativa para entornos multinúcleo:** El planificador implementa tres estrategias adaptadas al contexto multinúcleo:
  - Planificación por prioridad considerando la afinidad de núcleo
  - Preemption con preferencia por el núcleo local
  - Round Robin adaptativo de ”mejor esfuerzo”
- **Evolución avanzada en gestión de memoria dinámica**:
  - Sistema `multi_heap`: Gestiona múltiples regiones de memoria (DRAM, IRAM, PSRAM).
  - Algoritmo TLSF: Operaciones `malloc()`/`free()` en $O(1)$.
  - Fragmentación acotada.
- **Control indirecto por parte del programador:** Aunque el planificador toma decisiones automáticas, el programador influye significativamfente en estas a través de la configuración de prioridades, afinidad de núcleo, bloqueos y retardos temporales.

Comprender estas adaptaciones es esencial para diseñar sistemas embebidos eficientes y fiables sobre la plataforma ESP32 con FreeRTOS.

Referencias

[1] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[2] Espressif Systems, “ESP32-S3 Series Datasheet,” Version 1.2, 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf

[3] R. Barry, “Mastering the FreeRTOS Real Time Kernel - A Practical Guide,” Real Time Engineers Ltd. [Online]. Available: https://www.freertos.org/Documentation/RTOS_book.html

[4] R. Barry, “Using The FreeRTOS Real Time Kernel - A Practical Guide - PIC32 Edition,” Real Time Engineers Ltd.

[5] Espressif Systems, “ESP-IDF Programming Guide - FreeRTOS (SMP),” latest. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos_smp.html

[6] FreeRTOS Kernel. [Online]. Available: https://github.com/FreeRTOS/FreeRTOS-Kernel

[7] Miguel Angel Masmano Tello, *Tesis Doctoral: Gestión de Memoria Dinámica en Sistemas de Tiempo Real*. Universidad Politécnica de Valencia - Departamento de Informática de Sistemas y Computadores, España, 2007.

[8] M. Masmano, I. Ripoll, A. Crespo, J. Real *TLSF: a New Dynamic Memory Allocator for Real-Time Systems*. Universidad Politécnica de Valencia, España, 2007.