Recapitulando

Modelización de procesos TCB (Task Control Block)

tasks: Unidades concurrentes dentro de un único espacio de memoria

Espacio de memoria compartida: Todas las tareas comparten el mismo espacio de memoria. Esto significa que una tarea puede acceder directamente a la memoria de otra tarea, lo que facilita la comunicación, pero requiere mecanismos de sincronización adecuados.

Memoria en ESP32

Tipo de Memoria	Capacidad	Ubicación	Uso Principal	Características Especiales
SRAM Total	**520 KB**	Interna	Datos y código	Dividida en DRAM e IRAM
DRAM	**320 KB**	Interna	Datos de aplicación	Límite de 160 KB estáticos
IRAM	**200 KB**	Interna	Código crítico	Acceso rápido, sin cache miss
ROM	**448 KB**	Interna	Boot loader y librerías	Solo lectura
RTC Fast Memory	**8 KB**	Interna	Datos en deep sleep	Retiene datos sin alimentación
RTC Slow Memory	**8 KB**	Interna	Datos ULP	Ultra low power coprocessor
External RAM	**Hasta 4/8 MB**	Externa (PSRAM)	Expansión de memoria	Vía SPI, menor velocidad

Memoria FreeRTOS

La RAM necesaria para alojar objetos del kernel como tareas, colas, semáforos
puede asignarse estáticamente en tiempo de compilación o dinámicamente en tiempo de ejecución.

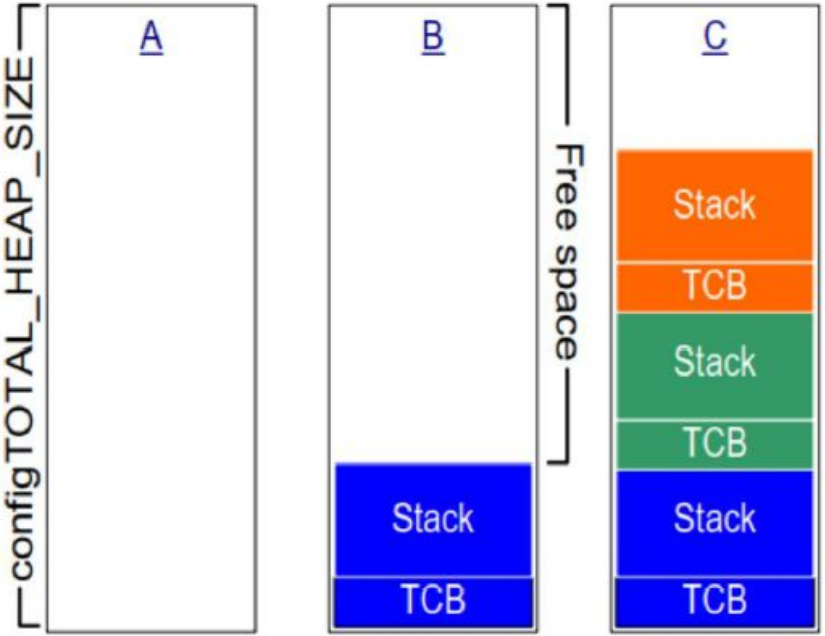
FreeRTOS proporciona 5 esquemas diferentes de asignación de memoria dinámica, cada uno diseñado para diferentes tipos de aplicaciones y requisitos. Estos esquemas se implementan en los archivos heap_1.c a heap_5.c ubicados en
FreeRTOS/Source/portable/MemMang/.

Heap 1

Características Principales:
Algoritmo: Subdivisión simple de array
Liberación de memoria: NO implementa Free()
Fragmentación: No puede ocurrir
Determinismo: Completamente determinístico O(1)
Thread-Safe: Deshabilitación de interrupciones

Funcionamiento:

- Subdivide un array estático de tamaño configTOTAL_HEAP_SIZE.
- Solo asigna memoria de forma secuencial.
- Una vez asignada, la memoria nunca se libera

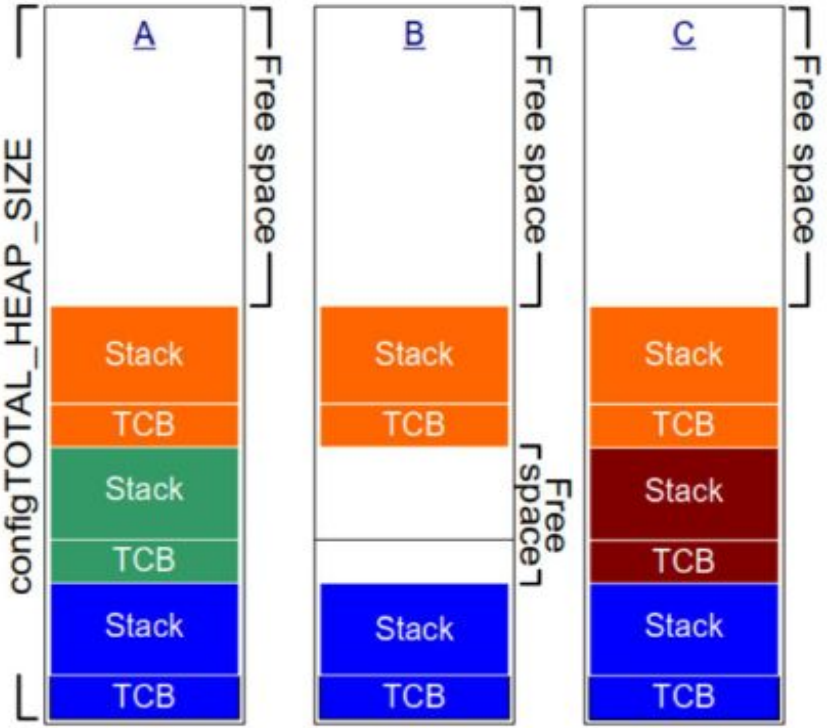


Heap 2 (Best Fit - obsoleto)

Características Principales:
Algoritmo: Best-fit (mejor ajuste)
Liberación de memoria: Implementa Free()
Fragmentación: Si
Determinismo: No determinístico
Coalescencia: No combina bloques libres adyacentes
Estado: Reemplazado por heap_4
Thread-Safe: Deshabilitación de interrupciones

Funcionamiento:

- Busca el bloque libre más pequeño que satisfaga la solicitud.
- Divide bloques grandes cuando es necesario.
- No combina bloques libres adyacentes al liberar memoria



Heap 3

Características Principales:
Algoritmo: Implementación de la librería C estándar
Liberación de memoria: Sí (usa free() estándar)
Fragmentación: Depende de la implementación de la librería
Determinismo: Generalmente no determinístico
Thread-safe: Mediante suspensión temporal del scheduler
Tamaño del heap: Definido por la configuración del enlazados.

Funcionamiento:

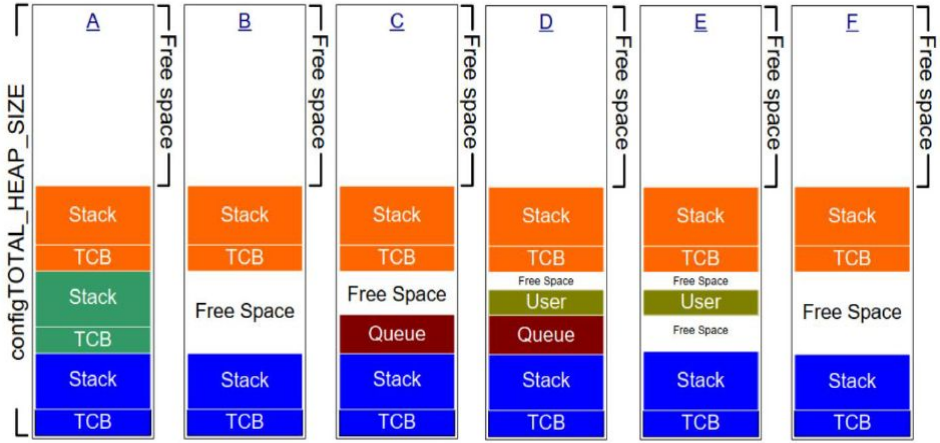
- malloc() y free() estándar con thread-safety
- Suspende temporalmente el scheduler FreeRTOS durante las operaciones
- Compatibilidad con librerias C estandar

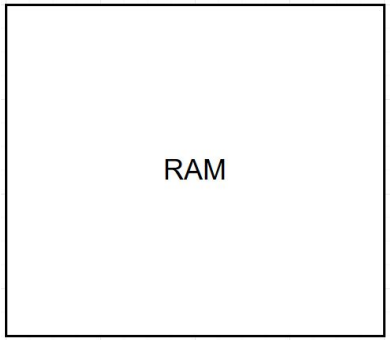
Heap 4

Características Principales:
Algoritmo: First-fit con coalescencia
Liberación de memoria: Implementa Free()
Fragmentación: Minimizada mediante coalescencia
Determinismo: No determinístico pero rápido
Coalescencia: Combina bloques libres adyacentes
Thread-Safe: Deshabilitación de interrupciones

Funcionamiento:

- Usa el primer bloque libre suficientemente grande
- Combina automáticamente bloques libres adyacentes
- Divide bloques grandes cuando es necesario
- Mantiene una lista de bloques libres ordenada (es clave, permite que la coalescencia sea eficiente, porque puede encontrar rápidamente bloques adyacentes.)
- Proteccion contra accesos concurrentes



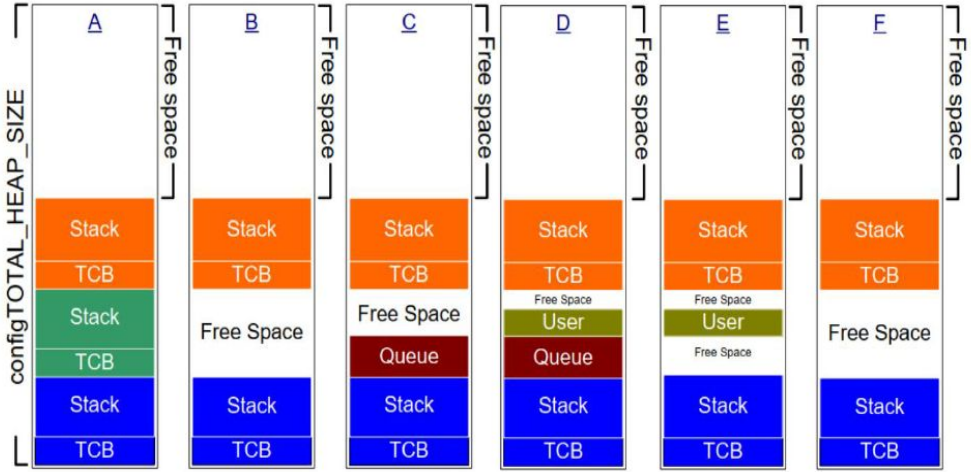


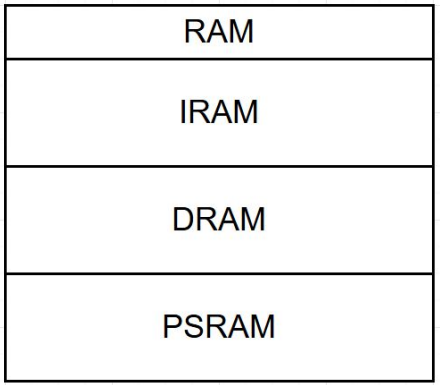
Heap 5

Características Principales:
Algoritmo: Lo mismo que heap_4 (first-fit + coalescencia)
Liberación de memoria: Implementa Free()
Fragmentación: Minimizada mediante coalescencia
Determinismo: No determinístico
Regiones múltiples: Puede usar múltiples áreas de memoria no contiguas
Inicialización: Requiere vPortDefineHeapRegions() antes de uso
Thread-Safe: Deshabilitación de interrupciones.

Funcionamiento:

- Identico a heap_4 pero puede combinar múltiples regiones de RAM
- Requiere inicialización explícita mediante vPortDefineHeapRegions()
- Las regiones deben definirse en orden de direcciones de memoria
- Puede aprovechar RAM rápida y lenta en el mismo heap
- Permite que FreeRTOS vea y use automáticamente diferentes "pedazos" de memoria que están en direcciones físicas completamente separadas, como si fuera un solo heap grande inteligente.





Resumen Asignación Memoria FreeRTOS

Características	Heap 1	Heap 2	Heap 3	Heap 4	Heap 5
Liberación memoria	NO	SI	SI	SI	SI
Algoritmo de búsqueda	Lineal (Simple)	Best Fit	Según implementación	First Fit optimizado	First Fit por región
Coalescencia	-----	NO	Depende del sistema	Si. Inmediata	Si inmediata
Fragmentación	Ninguna	Alta	Depende	Baja	Baja
Velocidad asignación	Muy rápida	Rápida	Variable	Rápida	Media
Múltiples regiones	No	No	No	No	Si
Recomendado	Casos específicos	NO	Casos específicos	Si	ESP32 con PSRAM

ESP32 ESP-IDF

Algoritmo TLSF

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

¿Que es TLSF?.

TLSF (**Two Level Segregate Fit**) es un algoritmo de asignación de memoria Diseñado para sistemas de tiempo real.

- Tiempo constante O(1) para malloc() y Free().
- Baja fragmentación de memoria.
- Comportamiento predecible y determinístico

Estructura de dos Niveles:

TLSF organiza la memoria libre en jerarquía de dos niveles de segregación.

Primer nivel de segregación (FL - First Level)

- Divide los bloques de memoria por potencias de dos.

Segundo nivel de segregación (SL - Second Level)

- Cada clase del primer nivel se subdivide en varios bloques más pequeños.

TLSF mantiene:

- Una **matriz de listas libres** organizada por (FL, SL).
- **Bitmaps** para cada nivel que indican rápidamente qué listas tienen bloques disponibles.

Esto permite:

- Búsqueda en **O(1)** para el mejor ajuste disponible.
- Liberación en **O(1)** con fusión de bloques vecinos libres.

malloc(size)

1. Se determina la clase (FL, SL) adecuada.
2. Se busca la lista libre más cercana al tamaño solicitado (usando los bitmaps).
3. Se toma un bloque y, si es más grande que lo necesario, se divide.

free(ptr)

1. Se agrega el bloque liberado a la lista correspondiente.
2. Se intenta fusionar con bloques libres adyacentes.

TLSF Control Block	
Mapa_Bit_i = [11110001]	Bitmap Primer Nivel
Mapa_Bit_j[7] = [01010000]	Bitmap FLI=7
Mapa_Bit_j[6] = [10100000]	Bitmap FLI=6
Mapa_Bit_j[5] = [00110000]	Bitmap FLI=5
Mapa_Bit_j[4] = [10000010]	Bitmap FLI=4
Mapa_Bit_j[3] = [00000000]	Bitmap FLI=3
Mapa_Bit_j[2] = [00000000]	Bitmap FLI=2
Mapa_Bit_j[1] = [00000000]	Bitmap FLI=1
Mapa_Bit_j[0] = [00110000]	Bitmap FLI=0
Matriz de Punteros a Listas Lista[FLI][SLI] → Cabeza de lista enlazada	

Fórmulas de Mapeo

$$FLI = \lfloor \log_2(size) \rfloor$$

$$SLI = \lfloor \frac{size - 2^{FLI}}{2^{FLI-J}} \rfloor$$

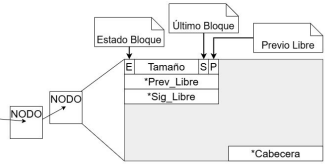
Ejemplo para 44 bytes:

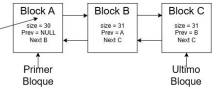
$$FLI = \lfloor \log_2(44) \rfloor = 5$$

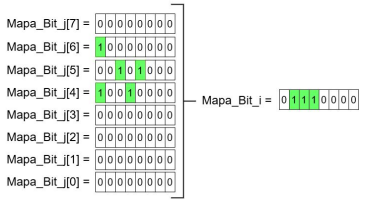
$$SLI = \lfloor \frac{size - 2^{FLI}}{2^{FLI-J}} \rfloor = \lfloor \frac{44 - 32}{2^{5-3}} \rfloor = 3$$

Posición [5][3] para rango 44-46 bytes

FLI \ SLI	0	1	2	3	4	5	6	7
6	64-71 Libre	72-29	80-87	88-95	96-103	104-111	112-119	120-127
5	32-35	36-39	40-43 Libre	44-47	48-51 Libre	52-55	56-59	60-63
4	16-17 Libre	18-19	20-21	22-23 Libre	24-25	26-27	28-29	30-31







¿Dónde caen los bloques ocupados?

TLSF no almacena bloques ocupados en la matriz.

Cuando un bloque se asigna (malloc):

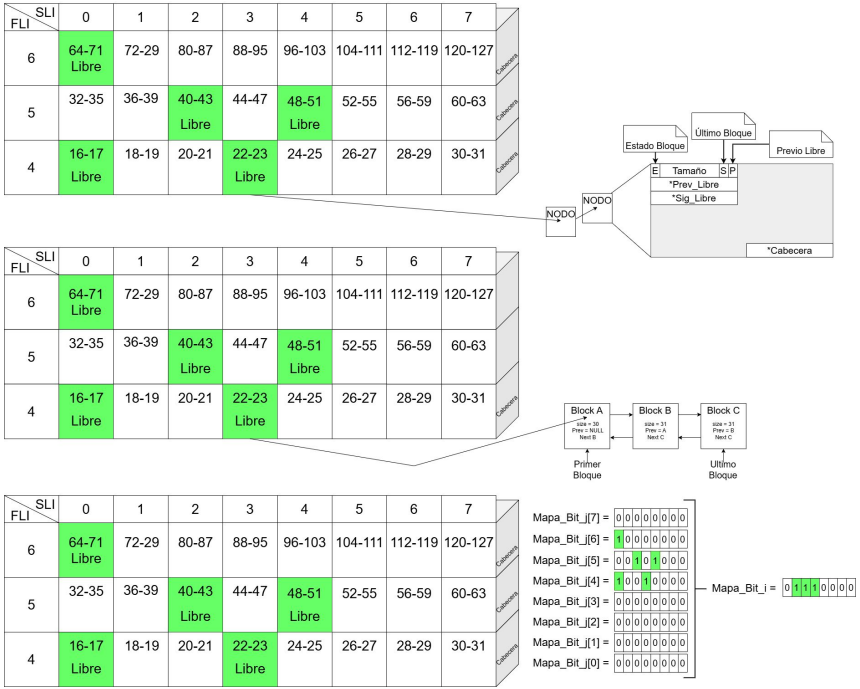
- Se **saca** de su lista libre correspondiente ([FL][SL]).
- Ya **no forma parte de la matriz de listas libres**.
- No se guarda en ninguna lista global de ocupados.

¿Dónde “están” los bloques ocupados entonces?

- **En memoria** siguen existiendo, pero **no están registrados en la estructura de TLSF**.
- Solo se accede a ellos mediante el puntero que el sistema devuelve al asignar.
- Cada bloque asignado tiene una **cabecera** donde se guarda información como:
  - Tamaño total del bloque.
  - Estado: libre u ocupado.
  - Flags para fusión con vecinos en free.

Cuando se libera un bloque (free):

- Se **marca como libre**.
- Se **intenta fusionar** con bloques libres vecinos.
- Finalmente, se lo **reincorpora** a su lista [FL][SL] correspondiente en la matriz.



Ejemplos Malloc()

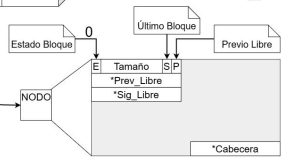
Ejemplo 1: malloc(44) - Búsqueda en SLI Superior
Solicitud: 44 bytes de memoria
Paso 1: Calcular posición
FLI = $\log_2(44) = 5$
SLI = (44 - 32) / 4 = 3
→ Buscar en [5][3] (rango 44-47)

Paso 2: Verificar disponibilidad
Mapa_Bit_j[5] = [00101000]
Bit en posición 3 = 0 → No hay bloques

FLI \ SLI	0	1	2	3	4	5	6	7
6	64-71 Libre	72-29	80-87	88-95	96-103	104-111	112-119	120-127
5	32-35	36-39	40-43 Libre	44-47 **Buscado**	48-51 Libre	52-55	56-59	60-63
4	16-17 Libre	18-19	20-21	22-23 Libre	24-25	26-27	28-29	30-31

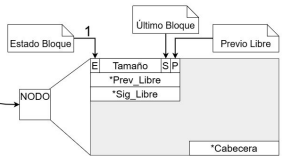
Mapa_Bit_j[7] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[6] = 1 0 0 0 0 0 0 0
Mapa_Bit_j[5] = 0 0 1 0 1 0 0 0
Mapa_Bit_j[4] = 1 0 0 1 0 0 0 0
Mapa_Bit_j[3] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[2] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[1] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[0] = 0 0 0 0 0 0 0 0
Mapa_Bit_i = 0 1 1 1 0 0 0 0

Paso 3: Estrategia Good-Fit
Buscar SLI superiores en FLI=5:
- SLI=4: vacío (No hay bloques libres)
- SLI=5: ¡disponible! → bloque de 48 bytes
Paso 4: Asignación
División: 44 bytes (usuario) + 4 bytes (fragmento pequeño, descartado)



FLI \ SLI	0	1	2	3	4	5	6	7
6	64-71 Libre	72-29	80-87	88-95	96-103	104-111	112-119	120-127
5	32-35	36-39	40-43 Libre	44-47	48-51	52-55	56-59	60-63
4	16-17 Libre	18-19	20-21	22-23 Libre	24-25	26-27	28-29	30-31

Mapa_Bit_j[7] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[6] = 1 0 0 0 0 0 0 0
Mapa_Bit_j[5] = 0 0 1 0 0 0 0 0
Mapa_Bit_j[4] = 1 0 0 1 0 0 0 0
Mapa_Bit_j[3] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[2] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[1] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[0] = 0 0 0 0 0 0 0 0
Mapa_Bit_i = 0 1 1 1 0 0 0 0



Ejemplo 2: malloc(35) - Búsqueda en FLI Superior
Solicitud: 35 bytes de memoria
Proceso con traducción:
1. Calcular: FLI=5, SLI=0 → [5][0]
2. Verificar: posición vacía (No hay bloques libres)
3. Buscar SLI superiores en FLI=5:
   - SLI=2: ¡disponible! → bloque de 42 bytes
Asignación con:
Bloque encontrado: 42 bytes
Fragmento interno: 7 bytes no utilizados

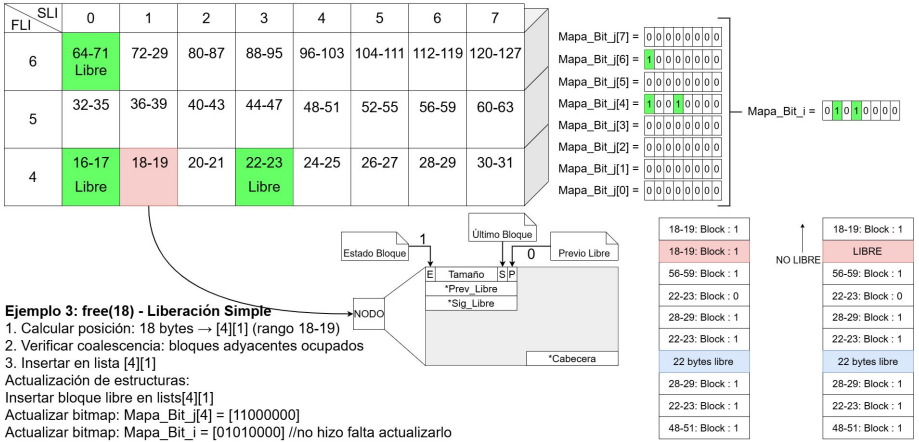
FLI \ SLI	0	1	2	3	4	5	6	7
6	64-71 Libre	72-29	80-87	88-95	96-103	104-111	112-119	120-127
5	32-35 **Buscado**	36-39	40-43 Libre	44-47	48-51	52-55	56-59	60-63
4	16-17 Libre	18-19	20-21	22-23 Libre	24-25	26-27	28-29	30-31

Mapa_Bit_j[7] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[6] = 1 0 0 0 0 0 0 0
Mapa_Bit_j[5] = 0 0 1 0 0 0 0 0
Mapa_Bit_j[4] = 1 0 0 1 0 0 0 0
Mapa_Bit_j[3] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[2] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[1] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[0] = 0 0 0 0 0 0 0 0
Mapa_Bit_i = 0 1 1 1 0 0 0 0

FLI \ SLI	0	1	2	3	4	5	6	7
6	64-71 Libre	72-29	80-87	88-95	96-103	104-111	112-119	120-127
5	32-35	36-39	40-43	44-47	48-51	52-55	56-59	60-63
4	16-17 Libre	18-19	20-21	22-23 Libre	24-25	26-27	28-29	30-31

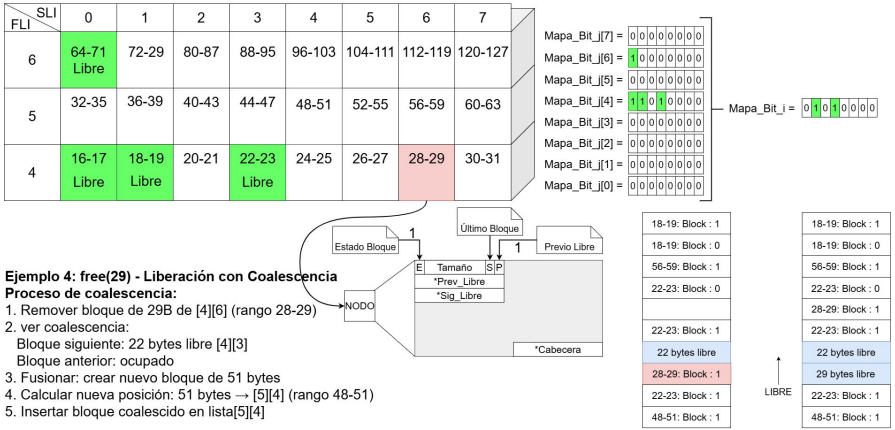
Mapa_Bit_j[7] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[6] = 1 0 0 0 0 0 0 0
Mapa_Bit_j[5] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[4] = 1 0 0 1 0 0 0 0
Mapa_Bit_j[3] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[2] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[1] = 0 0 0 0 0 0 0 0
Mapa_Bit_j[0] = 0 0 0 0 0 0 0 0
ACTUALIZACIÓN
Mapa_Bit_i = 0 1 0 1 0 0 0 0

Ejemplos Free()

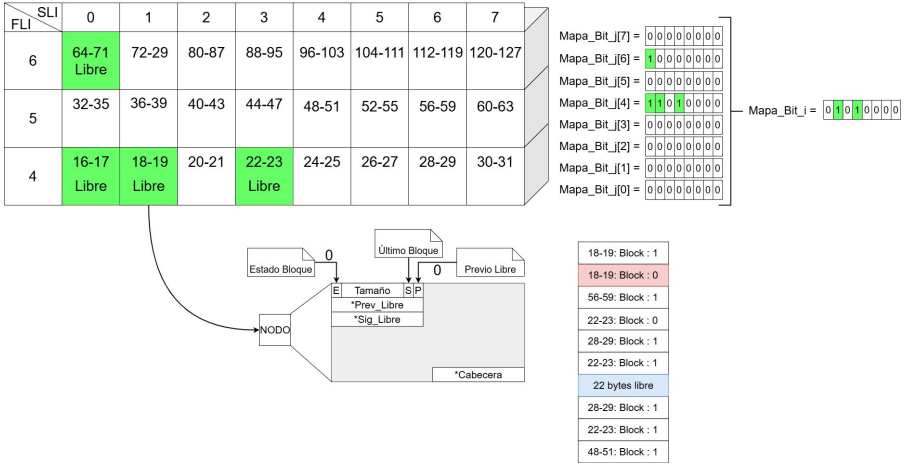


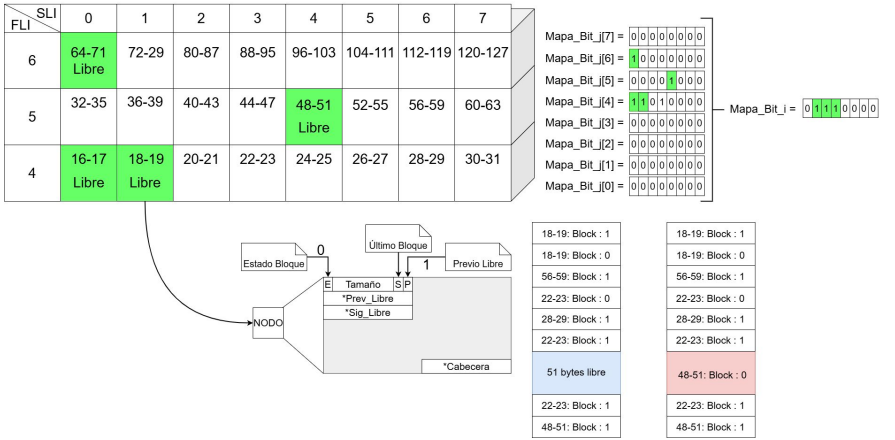
Ejemplo 3: free(18) - Liberación Simple
1. Calcular posición: 18 bytes → [4][1] (rango 18-19)
2. Verificar coalescencia: bloques adyacentes ocupados
3. Insertar en lista [4][1]

Actualización de estructuras:
Insertar bloque libre en lists[4][1]
Actualizar bitmap: Mapa_Bit_j[4] = [11000000]
Actualizar bitmap: Mapa_Bit_i = [01010000] //no hizo falta actualizarlo



Ejemplo 4: free(29) - Liberación con Coalescencia
Proceso de coalescencia:
1. Remover bloque de 29B de [4][6] (rango 28-29)
2. ver coalescencia:
   Bloque siguiente: 22 bytes libre [4][3]
   Bloque anterior: ocupado
3. Fusionar: crear nuevo bloque de 51 bytes
4. Calcular nueva posición: 51 bytes → [5][4] (rango 48-51)
5. Insertar bloque coalescido en lista[5][4]





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

Análisis Matemático de Fragmentación: La fragmentación en TLSF está matemáticamente acotada.

Fragmentación_máxima ≤ (1/8) × Memoria_total + O($\log_2$(Memoria_total))

- **12.5% máximo garantizado** - mucho mejor que First Fit o Best Fit

Fragmentación Interna:

- **Acotada por SLI** (típicamente 3% con SLI=32)
- **Controlada por granularidad** de las sublistas

Fragmentación Externa:

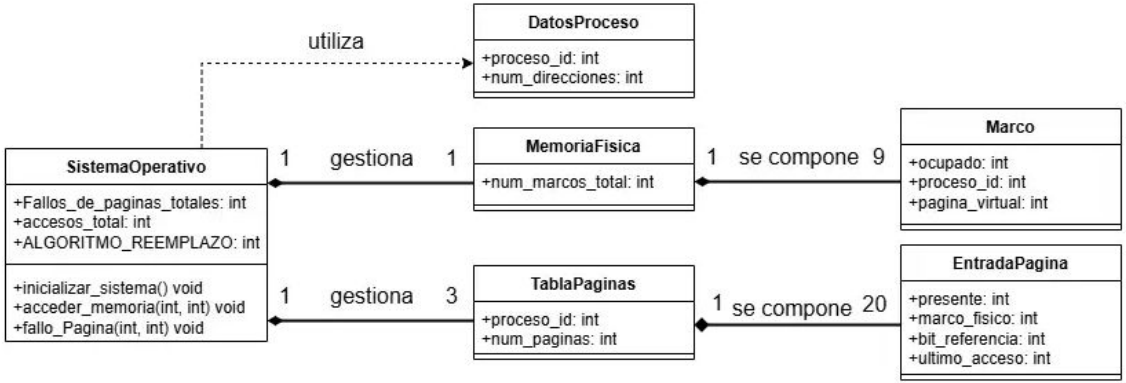
- **Minimizada por coalescencia inmediata**
- **Elimina huecos pequeños** automáticamente

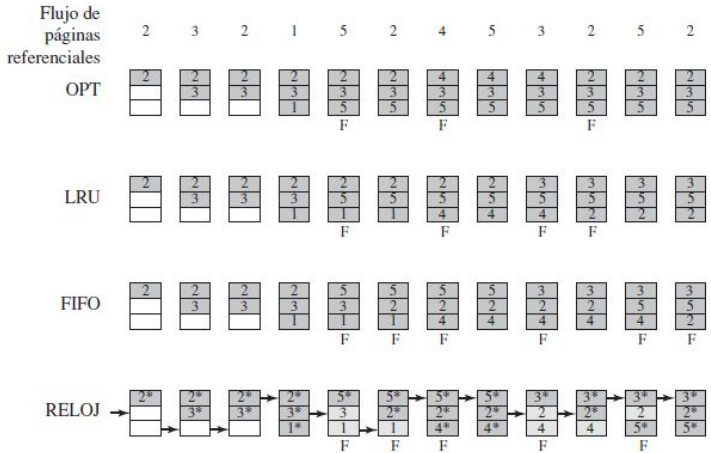
Eficiencia Garantizada:

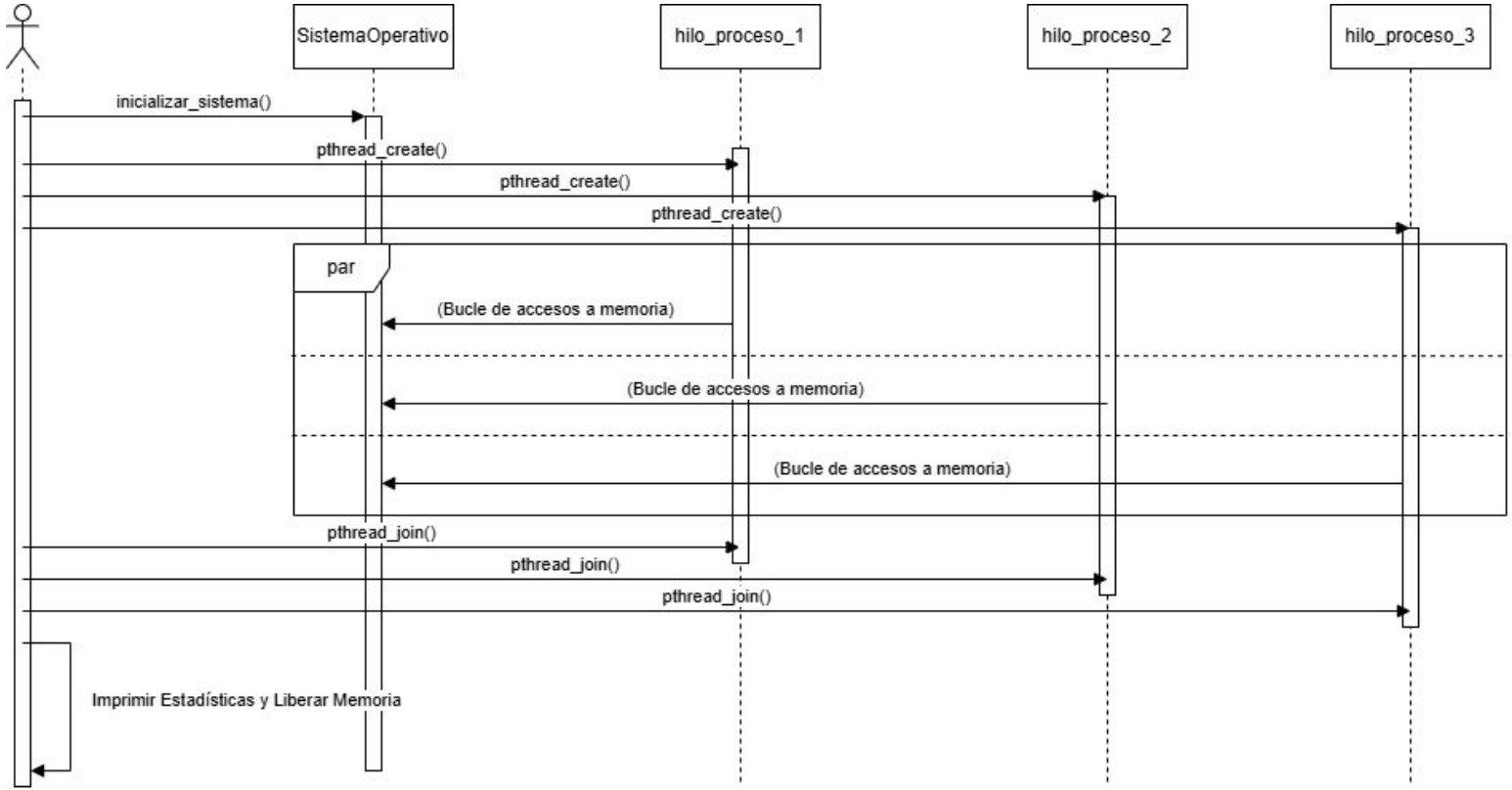
- **87.5% mínimo** de memoria útil garantizada
- **92-95% típico** en aplicaciones reales

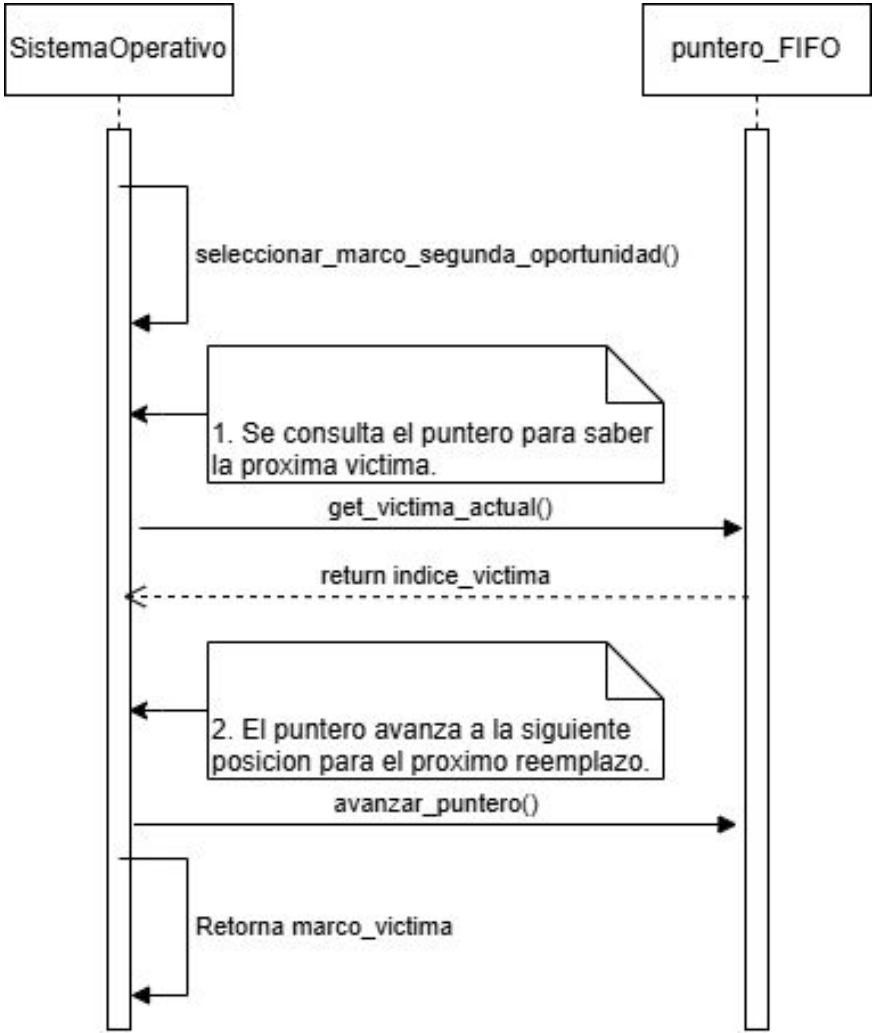
Esta **garantía matemática** es lo que hace a TLSF especialmente valioso para sistemas de tiempo real donde la **predictibilidad es crítica**.

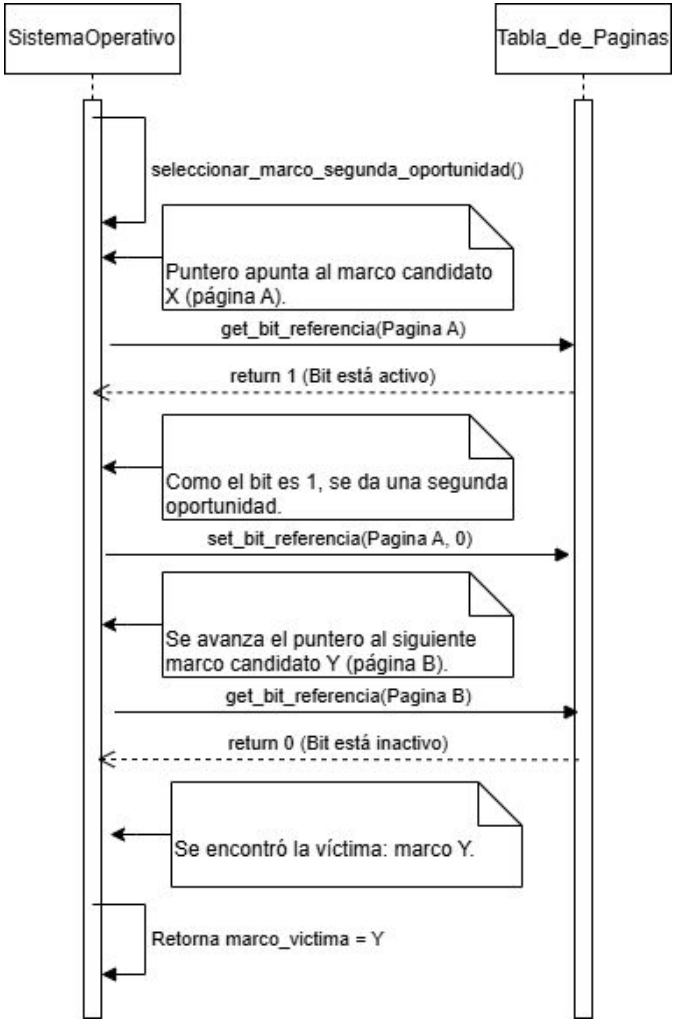
Estructuras de programa simulador de Memoria Virtual

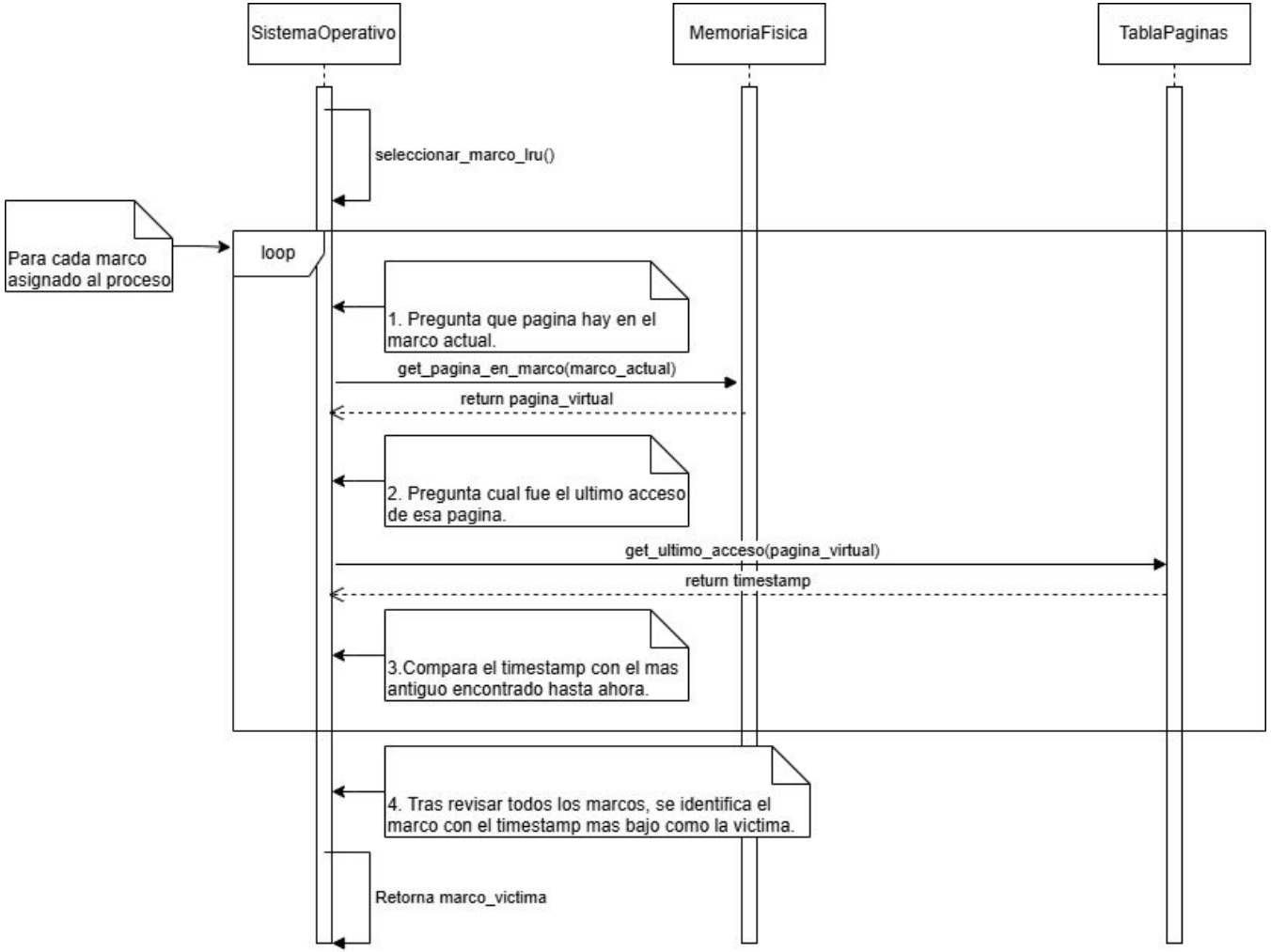





SistemaOperativo
hilo_proceso_1
hilo_proceso_2
hilo_proceso_3
inicializar_sistema()
pthread_create()
pthread_create()
pthread_create()
par
(Bucle de accesos a memoria)
(Bucle de accesos a memoria)
(Bucle de accesos a memoria)
pthread_join()
pthread_join()
pthread_join()
Imprimir Estadísticas y Liberar Memoria


SistemaOperativo
puntero_FIFO
seleccionar_marco_segunda_oportunidad()
1. Se consulta el puntero para saber la proxima victima.
get_victima_actual()
return indice_victima
2. El puntero avanza a la siguiente posicion para el proximo reemplazo.
avanzar_puntero()
Retorna marco_victima

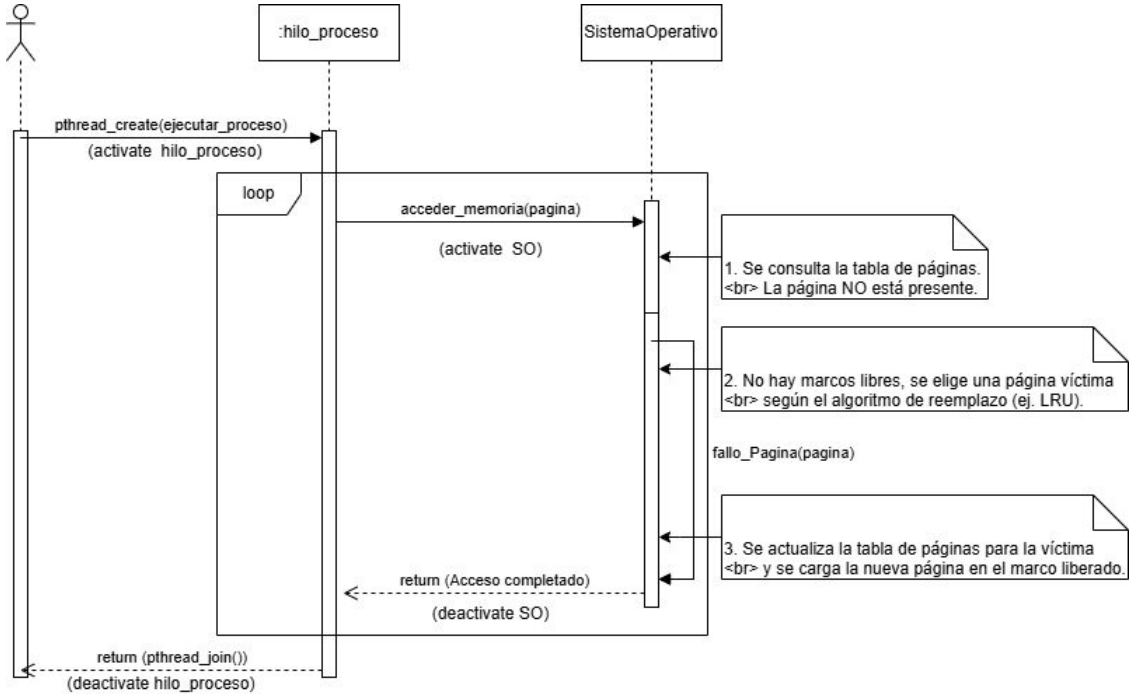

SistemaOperativo
Tabla_de_Paginas
seleccionar_marco_segunda_oportunidad()
Puntero apunta al marco candidato X (página A).
get_bit_referencia(Pagina A)
return 1 (Bit está activo)
Como el bit es 1, se da una segunda oportunidad.
set_bit_referencia(Pagina A, 0)
Se avanza el puntero al siguiente marco candidato Y (página B).
get_bit_referencia(Pagina B)
return 0 (Bit está inactivo)
Se encontró la víctima: marco Y.
Retorna marco_victima = Y


SistemaOperativo
MemoriaFisica
TablaPaginas
seleccionar_marco_lru()
Para cada marco asignado al proceso
loop
1. Pregunta que pagina hay en el marco actual.
get_pagina_en_marco(marco_actual)
return pagina_virtual
2. Pregunta cual fue el ultimo acceso de esa pagina.
get_ultimo_acceso(pagina_virtual)
return timestamp
3.Compara el timestamp con el mas antiguo encontrado hasta ahora.
4. Tras revisar todos los marcos, se identifica el marco con el timestamp mas bajo como la victima.
Retorna marco_victima

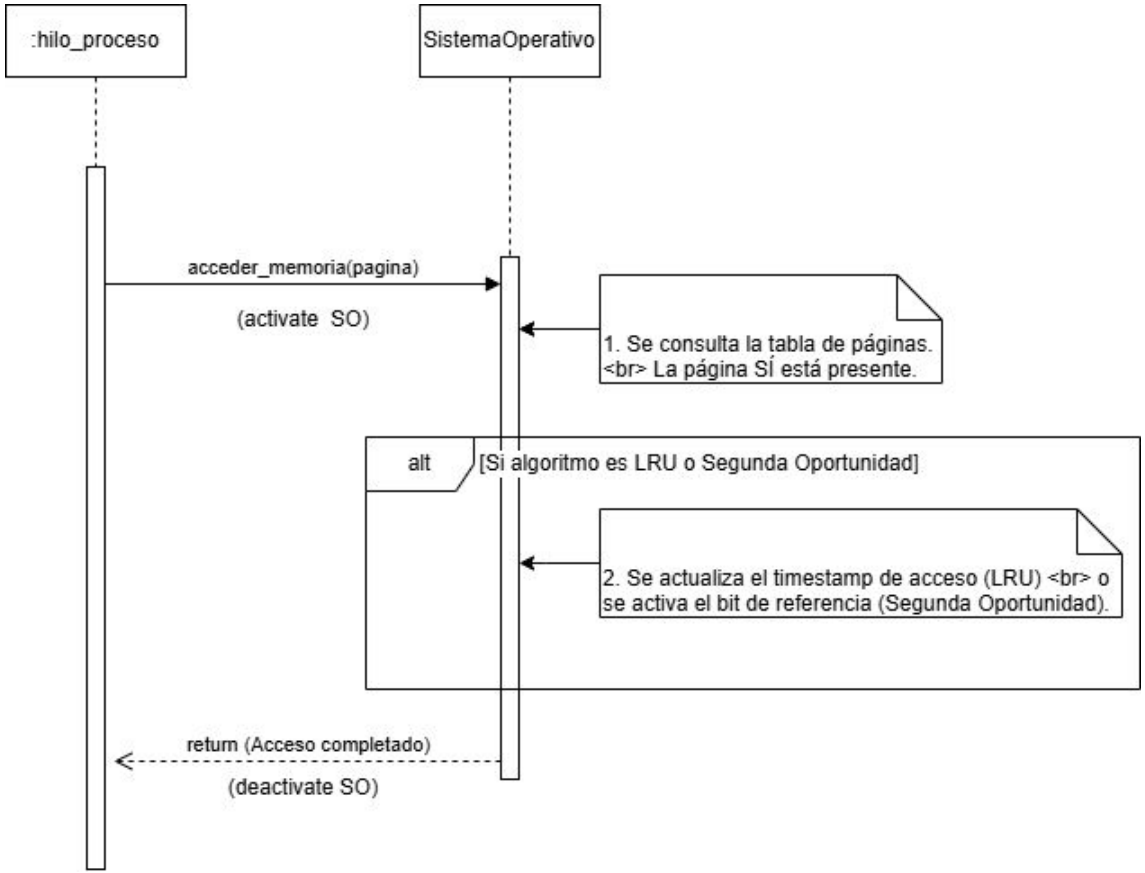
Caso: **Un proceso que solicita acceso a una página de memoria que no se encuentra cargada, lo que provoca un fallo de página y requiere un reemplazo.**

Participantes (Lifelines) del Diagrama:

- :main: La función principal que inicia todo el sistema.
- :hilo_proceso: Representa a uno de los hilos de pthread que está ejecutando la simulación de un proceso.
- :SistemaOperativo: Este es un participante lógico que representa el conjunto de funciones y variables globales que gestionan la memoria (acceder_memoria, fallo_Pagina, tablas_paginas, memoria_fisica, etc.). Actúa como el "kernel" de nuestra simulación.



Caso: **Que sucede cuando un proceso solicita una página que ya se encuentra cargada en uno de sus marcos de memoria.**
Participantes (Lifelines) del Diagrama:



Caso intermedio. El proceso solicita una página que **no está en memoria**, pero a diferencia del primer ejemplo, **hay un marco de página vacío** asignado a ese proceso.