

# **TLSF: Two-Level Segregated Fit**

## **¿Qué es TLF?** (2 minutos)

**Two-Level Segregated Fit** es un algoritmo de asignación dinámica de memoria diseñado específicamente para **sistemas operativos de tiempo real**.

### **Características Principales**

- **Tiempo constante O(1)** para malloc() y free()
- **Baja fragmentación** de memoria
- **Comportamiento predecible** y determinístico
- **Eficiencia en espacio** con overhead mínimo

### **¿Por qué es Importante?**

#### **Problemas de algoritmos tradicionales:**

- First Fit: O(n) en el peor caso, mucha fragmentación.
- Best Fit: O(n) siempre.
- Buddy System: Fragmentación interna significativa

#### **Aplicaciones de TLF:**

- Sistemas embebidos (IoT, microcontroladores)
- Cualquier sistema que requiera predictibilidad temporal

## **Estructura de Dos Niveles** (6 minutos)

### **Concepto Fundamental**

TLF organiza la memoria libre en una jerarquía de **dos niveles de segregación**:

#### **Primer Nivel (FLI - First Level Index)**

- Divide el espacio de memoria en clases de tamaño por potencia de 2.
- Se usa bitmap para indicar que clases tienen bloques libres.

#### **División por potencias de 2:**

FLI=4: [16-31] bytes

FLI=5: [32-63] bytes

FLI=6: [64-127] bytes

FLI=7: [128-255] bytes

## Segundo Nivel (SLI - Second Level Index)

- Subdivide cada clase del primer nivel en subclases más pequeñas
- Típicamente cada clase se divide en 16 o 32 subclases
- También usa un bitmap por cada clase del primer nivel

**Subdivisión linear dentro de cada FLI:** Con J=3 (8 subdivisiones):

FLI=5: [32-63] bytes

FLI=5: [32-35] [36-39] [40-43] [44-47] [48-51] [52-55] [56-59] [60-63]

SLI=0 SLI=1 SLI=2 SLI=3 SLI=4 SLI=5 SLI=6 SLI=7

## Fórmulas de Mapeo

(size)]

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

Ejemplo para 44 bytes:

$$(44)] = 5$$

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

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

## Cálculo de Rangos por Posición

**Fórmula para calcular rangos:**

$$Base = 2^{FLI}$$

$$Intervalo = \frac{Base}{2^J}$$

$$Rango min = Base + (SLI * Intervalo)$$

$$Rango max = Base + ((SLI + 1) * intervalo) - 1$$

**Ejemplo: Calcular rango para [6][0]**

$$Base = 2^6 = 64$$

$$Intervalo = \frac{64}{2^3} = \frac{64}{8} = 8$$

$$Rango min = 64 + (0 * 8)$$

$$Rango max = 64 + (1 * 8) - 1 = 71$$

[6][0] cubre 64-71 byte

## Tabla de rangos para FLI=6:

```
[6][0]: 64-71 [6][1]: 72-79 [6][2]: 80-87 [6][3]: 88-95  
[6][4]: 96-103 [6][5]: 104-111 [6][6]: 112-119 [6][7]: 120-127
```

# Estructuras de Control

### **Matriz de Listas Enlazadas:**

- Array bidimensional `listas[FLI][SLI]`
  - Cada posición apunta a lista de bloques libres

### **Mapas de Bits para Búsqueda O(1):**

Mapa\_Bit\_i: bitmap de primer nivel (qué FLI tienen bloques)

Mapa\_Bit\_j[i]: bitmap de segundo nivel (qué SLI tienen bloques en FLI=i)

**Búsqueda instantánea:** Operación find-first-set por hardware

# Organización de Listas Dblemente Enlazadas

## Estructura de Control Principal

TLSF Control Block	
Mapa Bit i = [11110000]	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] = [00000000]	Bitmap FLI=4
Mapa_Bit_j[3] = [00000000]	Bitmap FLI=3
Matriz de Punteros a Listas	
listas[FLI][SLI] → Cabeza de lista enlazada	

## Matriz de Listas enlazadas

SLI→	0	1	2	3	4	5	6	7
FLI=6	64-71 Libre	72-79	80-87	88-95	96-103	104-111	112 - 119	120 - 127
FLI=5	32-35	36-39	40-43 Libre	44-47	48-51	52-55 Libre	56-59	60-63
FLI=4	16-17	18-19	20-21	22-23	24-25	26-27	28-29	30-31



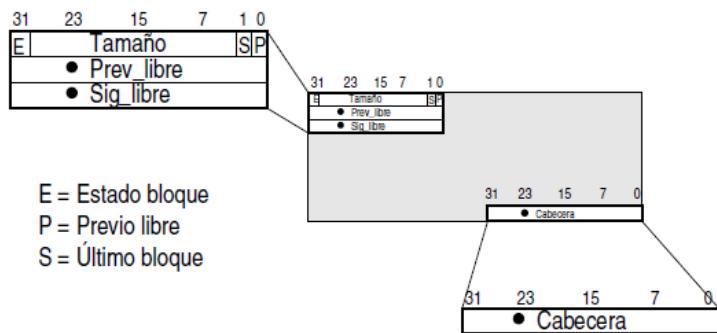
Bloques libres disponibles:

- [6][0]: 68 bytes (rango 64-71)
- [5][2]: 42 bytes (rango 40-47)
- [5][5]: 52 bytes (rango 52-59)
- [4][0]: 17 bytes (rango 16-17)
- [4][3]: 22 bytes (rango 22-23)

Bitmaps:

```
Mapa_Bit_j[6] = [10000000] // Posición 0
Mapa_Bit_j[5] = [00100100] // Posiciones 2 y 5
Mapa_Bit_j[4] = [10010000] // Posición 0 y 3
Mapa_Bit_i = [01110000] // FLI 6,5,4 disponible
```

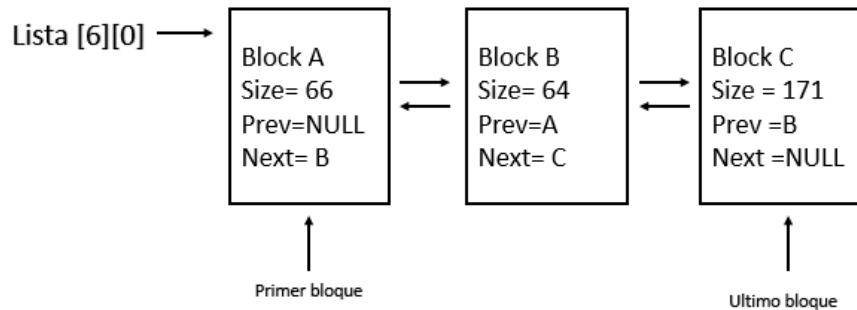
## Estructura de un Bloque Libre



- **Tamaño**, el bit 31 (E) de este campo codifica el estado del bloque (libre u ocupado). El bit 1 (S) se utiliza para codificar si este bloque es el límite físico del montículo de memoria. El bit 0 (P) permite conocer si el bloque físicamente previo se encuentra libre o asignado. El resto de bits se utilizan para codificar el tamaño del bloque.
- **Dos punteros**, **Prev\_libre** y **Sig\_libre**, los cuales enlazan al bloque con las correspondientes listas segregadas. Se utilizan dos punteros debido a que esta lista es una lista doblemente enlazada.

Además de esta cabecera, los bloques libres utilizan la última palabra del bloque como puntero a la cabecera. Este campo se denomina cabecera.  
Mediante el uso de este campo, el bloque siguiente, físicamente contiguo, puede encontrar en tiempo constante la cabecera de este.

## Vista de una lista enlazada



**IMPORTANTE:** En cada lista pueden coexistir bloques de diferentes tamaños dentro del rango permitido. Todas satisfacen solicitudes en el rango permitido 64-71

## Caso Real: Heap de 2KB (8 minutos)

### Estado del Sistema

i\j	0	1	2	3	4	5	6	7
6	64 Libre	72	80	88	96	104	112	120
5	32	36	40 Libre	44	48	52 Libre	56	60
4	16 Libre	18	20	22 Libre	24	26	28	30

Bitmaps:

```

Mapa_Bit_j[6] = [10000000] // Posición 0
Mapa_Bit_j[5] = [00100100] // Posiciones 2 y 5
Mapa_Bit_j[4] = [10010000] // Posición 0 y 3
Mapa_Bit_i = [01110000] // FLI 6,5,4 disponible
    
```

### Ejemplo 1: malloc(44) - Búsqueda en SLI Superior

Solicitud: 44 bytes de memoria

#### Paso 1: Calcular posición

```

FLI = log2(44) = 5
SLI = (44 - 32) / 4 = 3
→ Buscar en [5][3] (rango 44-47)
    
```

#### Paso 2: Verificar disponibilidad

```

Mapa_Bit_j[5] = [00100100]
Bit en posición 3 = 0 → No hay bloques

```

### Paso 3: Estrategia Good-Fit

Buscar SLI superiores en FLI=5:

- SLI=4: vacío (No hay bloques libres)
- SLI=5: ¡disponible! → bloque de 52 bytes

### Paso 4: Asignación

División: 44 bytes (usuario) + 8 bytes (fragmento pequeño, descartado)

### Estado del Sistema

i\j	0	1	2	3	4	5	6	7
6	64 Libre	72	80	88	96	104	112	120
5	32	36	40 Libre	44	48	52	56	60
4	16 Libre	18	20	22 Libre	24	26	28	30

Bitmaps:

```

Mapa_Bit_j[6] = [10000000] // Posición 0
Mapa_Bit_j[5] = [00100000] // Posiciones 2
Mapa_Bit_j[4] = [10010000] // Posición 0 y 3
Mapa_Bit_i = [01110000] // FLI 6,5,4 disponible

```

### 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

### Estado del Sistema

i\j	0	1	2	3	4	5	6	7
6	64 Libre	72	80	88	96	104	112	120
5	32	36	40	44	48	52	56	60
4	16 Libre	18	20	22 Libre	24	26	28	30

Bitmaps:

```

Mapa_Bit_j[6] = [10000000] // Posición 0
Mapa_Bit_j[5] = [00000000] // Ninguno libre
Mapa_Bit_j[4] = [10010000] // Posición 0 y 3
Mapa_Bit_i = [01010000] // FLI 6,4 disponible

```

### 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

#### Estado del Sistema

i\j	0	1	2	3	4	5	6	7
6	64 Libre	72	80	88	96	104	112	120
5	32	36	40	44	48	52	56	60
4	16 Libre	18 Libre	20	22 Libre	24	26	28	30

Bitmaps:

```

Mapa_Bit_j[6] = [10000000] // Posición 0
Mapa_Bit_j[5] = [00000000] // Ninguno libre
Mapa_Bit_j[4] = [11010000] // Posición 0,1,3 libre
Mapa_Bit_i = [01010000] // FLI 6,4 disponible

```

## 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]

Detalle del punto 2

Información para Coalescencia - Estructura TLSF

Size y flag: Tamaño del bloque actual + flag → permite calcular dirección siguiente

Bit P: Estado del bloque anterior → determina si el anterior es libre/ocupado

Footer: (puntero a cabecera) → permite encontrar el header del anterior

Bit E: Estado del bloque actual → determina si puede coalescer.

### Navegación física:

- Bloque siguiente: dirección\_actual + tamaño\_actual
- Bloque anterior: footer\_anterior → puntero\_a\_cabecera (solo si bit P=0)

## Estado del Sistema

i\j	0	1	2	3	4	5	6	7
6	64 Libre	72	80	88	96	104	112	120
5	32	36	40	44	48 Libre	52	56	60
4	16	18 Libre	20	22	24	26	28	30

Bitmaps:

```
Mapa_Bit_j[6] = [10000000] // Posición 0
Mapa_Bit_j[5] = [00001000] // posición 4
Mapa_Bit_j[4] = [01000000] // Posición 1,1, y 3 libre
    Mapa_Bit_i = [01110000] // FLI 6,5,4 disponible
```

## **Ventajas y Rendimiento** (2 minutos)

### **Garantías de Rendimiento**

#### **Operaciones en tiempo constante:**

- malloc(): O(1) garantizado
- free(): O(1) garantizado
- Búsqueda: ~10-30 ciclos CPU

#### **Comparación de rendimiento:**

Algoritmo tradicional: 100-10,000 ciclos (variable)

TLSF: 50-100 ciclos (constante y predecible)

### **Eficiencia de Memoria**

#### **Fragmentación mínima:**

- Externa: <5% (excelente coalescencia)
- Interna: ~10-15% (overhead razonable)
- Utilización efectiva: 80-85%

#### **Overhead del sistema:**

- Control TSLF: ~600 bytes (0.4% para 160KB)
- Headers por bloque: 4-8 bytes

### **Aplicaciones Ideales**

- **Sistemas de tiempo real:** Comportamiento predecible
- **IoT y embebidos:** Eficiencia en recursos limitados

## **Aplicación Real: ESP32 (1 minuto)**

### **Configuración para ESP32**

#### **Especificaciones del microcontrolador:**

- RAM disponible: ~160 KB
- Arquitectura: 32 bits
- Frecuencia: 240 MHz

#### **Parámetros TSLF optimizados:**

Bloque mínimo: 16 bytes (alineación 32-bit)  
FLI máximo: 17 (cubre hasta 128KB)  
SLI subdivisiones: 8 ( $J=3$ )  
Total de listas: 144  
Overhead de control: ~610 bytes (0.4% del heap)

## Casos de Uso Típicos en ESP32

### Gestión eficiente para aplicaciones IoT:

- **Buffers WiFi:** Asignaciones frecuentes de 64-1024 bytes
- **JSON envió de datos:** Fragmentos variables de 32-512 bytes
- **Sensor data:** Pequeños bloques de 16-64 bytes
- **Audio/señales:** Bloques grandes de 2-8 KB

### Ventajas específicas:

- **Latencia predecible:** Crítico para comunicaciones en tiempo real
- **Bajo Overhead:** Preserva memoria valiosa en sistema embebido
- **Sin fragmentación:** Mantiene heap utilizable durante operación prolongada

### Rendimiento en ESP32:

- `malloc()`: ~0.2-0.4 µs (50-100 ciclos a 240MHz)
- `free()`: ~0.15-0.3 µs (40-80 ciclos)
- **10x más rápido** que `malloc()` estándar en escenarios complejos

## Conclusiones

### TLSF revoluciona la gestión de memoria porque combina:

1. **Predictibilidad absoluta:** O(1) garantizado siempre
2. **Eficiencia práctica:** Baja fragmentación y overhead mínimo
3. **Escalabilidad:** Desde 8 bytes hasta gigabytes
4. **Simplicidad operacional:** Operaciones de bits ultrarrápidas

**Resultado final:** Memoria dinámica confiable y eficiente para sistemas donde cada microsegundo cuenta.

## Preguntas