UNIVERSIDAD POLITÉCNICA DE VALENCIA

DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Gestión de Memoria Dinámica en Sistemas de Tiempo Real

Tesis Doctoral Presentada Por:

Miguel Ángel Masmano Tello

Dirigida Por:

José Ismael Ripoll Ripoll

Jorge Vicente Real Sáez

Valencia, Mayo de 2007

Título:

Gestión de Memoria Dinámica en Sistemas de Tiempo Real

Autor:

Miguel Ángel Masmano Tello

Tribunal:
Presidente:
Vocales:

Acuerdan otorgar la calificación de:

Índice general

Resumen

La gestión de la memoria dinámica aparece a principios de los años cincuenta para reemplazar la hasta entonces utilizada gestión de memoria estática. Tales han sido las ventajas derivadas de su uso que, en la actualidad, es difícil encontrar sistemas que no la utilicen. Actualmente, la mayoría de sistemas operativos existentes así como algunos lenguajes de programación proveen facilidades para su uso, tanto de forma explícita como implícita.

Entre las pocas excepciones de sistemas que no hacen uso de la memoria dinámica se encuentran los sistemas de tiempo real. Estos sistemas, cuya corrección depende no solo de su lógica interna sino también del tiempo de respuesta, han evitado su uso principalmente por dos motivos: primero, la gestión de memoria siempre ha sido considerada por la comunidad de tiempo real como un problema inherentemente no determinista temporalmente. Segundo, el fenómeno de la fragmentación, el cual puede causar que un gestor de memoria fracase en una asignación de memoria, a pesar de que la cantidad de memoria no utilizada en ese momento sea igual o superior a la requerida.

Sin embargo, el reciente auge de lenguajes como Java, el cual hace un uso intensivo de la memoria dinámica de forma implícita, y su adopción en sistemas de tiempo real esta forzando a reconsiderar e introducir la gestión de memoria dinámica en este tipo de sistemas.

La presente tesis aborda la problemática de la gestión de memoria dinámica en sistemas de tiempo real. Primero, proponiendo un nuevo gestor de memoria dinámica, Two-Level Segregated Fit (TLSF), el cual presenta un tiempo de respuesta rápido y determinista, así como una fragmentación baja. Segundo, realizando un estudio de este nuevo algoritmo, a nivel teórico y práctico, y haciendo una comparación de los resultados de este algoritmo con los resultados obtenidos con los gestores ya existentes.

Resum

La gestió de la memòria dinàmica aparegué a principi dels anys cinquanta per a reemplaçar la gestió de memòria estàtica utilitzada fins a eixe moment. Tals han sigut els avantatges derivats del seu ús que a l'actualitat és difícil trobar sitemes que no la utilitzen. Actualment la majoria de sistemes operatius existents així com alguns llenguatges de programació proveixen facilitats pel seu ús, tant de forma explícita com implícita.

Entre les poques excepcions de sistemes que facen ús de la memoria dinàmica es troben els sistemes de temps real. Aquests sistemes, de quina correcció depen no sols de la seus lògica interna sino també del temps de resposta, han evitat el seu ús principalment per dos motius: Primer, la gestió de memòria sempre ha sigut considerada per la comunitat de temps real com un problema inherentment no determinista temporalment. Segon, el fenòmen de la fragmentació, què pot causar que un gestor de memòria fracase una asignació de memoria, a pesar de que la quantitat de memòria lliure en eixe moment siga igual o superior a la demandada.

A pesar de tot, la recient expansió dels llenguatges com Java, què fà una utilització intensiva de la memòria dinàmica implicitament, i la seua adopció en sistemes de temps reals està forçant a reconsiderar i, inclós, introduïr la gestió de memòria dinàmica en aquests tipus de sistemes.

La present tesis abarca la problemàtica de la gestió de memòria dinàmica en sistemes de temps reals. Primer, proposant un nou gestor de memòria dinàmica, Two-Level Segregated Fit (TLSF), què presenta un temps de responta ràpid y determinista, així com una fragmentació baixa. Segon, realitzant un estudi tant del temps de resposta com de la fragmentació, a nivell teòric i a nivell pràctic d'aquest nou algoritme i comparant els resultats amb dels altres gestors ja existents.

Abstract

Dynamic storage management appears in the early fifties to replace the previously existing static storage management. Due to the advantages derived of the use of dynamic memory, currently, it is fairly difficult to find systems which do not use it. Nowadays, most existing operating systems, as well as some programming languages enable, in any explicit or implicit way, to use dynamic memory.

Real-time systems belong to the few exceptions of systems which do not use dynamic memory at all. These systems, whose correctness lies in the well-working of their internal logic and in their response time, have avoided the use of dynamic memory due to reasons, firstly, the widely belief that all dynamic memory managers present an unbounded response time. Secondly, the fragmentation problem, which could cause the failure of an allocation, despite of the fact that there is more free memory than required.

However, the recent peak of languages as Java, which implicitly carries out an exhaustive use of the dynamic memory, and its adoption in real-time systems is forcing to reconsider dynamic memory management in this sort of systems.

This PhD thesis tackles the problem of dynamic memory in real-time systems. Proposing, firstly, a new dynamic memory manager, Two-Level Segregated Fit (TLSF), which shows a quick and deterministic response time, as well as, a low fragmentation. Secondly, performing a deep, theoretical and practical, study of the response time and the fragmentation of this algorithm. Comparing, subsequently, the results with those obtained with the rest of allocators.

Abreviaturas

$\mathcal{H}$	Tamaño del montículo del gestor de memoria.
M	Máxima cantidad de memoria que una aplicación puede tener asignada.
m	Máximo tamaño asignable en una única asignación.
n	Mínimo tamaño asignable en una asignación.
c	Grado de competitividad de un algoritmo *on-line*. Calculado mediante análisis competitivo.
r	Tamaño de bloque pedido al gestor memoria.
r'	Tamaño de bloque asignado por el gestor de memoria.
$\mathcal{I}$	Límite del índice de primer nivel en la estructura TLSF.
$\mathcal{J}$	Logaritmo en base 2 del límite del índice de segundo nivel en la estructura TLSF.
$2^{\mathcal{J}}$	Límite del índice de segundo nivel en la estructura TLSF.
i	Índice de primer nivel en la estructura TLSF para un tamaño dado.
j	Índice de segundo nivel en la estructura TLSF para un tamaño dado.
T_i	Tarea periódica, $T_i = (c_i, p_i, d_i, g_i, h_i)$.
c_i	Tiempo de cómputo en el peor de los casos de la tarea T_i.
p_i	Periodo de la tarea T_i.
d_i	Plazo de la tarea T_i.
g_i	Máxima cantidad de memoria que una tarea T_i puede pedir por periodo.
h_i	Máximo tiempo que una tarea T_i mantiene un bloque de memoria antes de liberarlo (Tiempo de posesión).
fi	Fragmentación interna absoluta.
fir	Fragmentación interna relativa.
fe	Fragmentación externa.

Capítulo 1
Planteamiento y objetivos de la tesis

El objetivo del presente capítulo es encuadrar el ámbito de la presente tesis así como realizar una declaración de objetivos y contribuciones.

1.1. Ámbito de la tesis

Hasta la aparición de la memoria dinámica, la memoria era asignada a la aplicación de forma estática en tiempo de compilación. Este esquema, aunque simple de implementar es claramente poco flexible y no permite desarrollar aplicaciones complejas. Un esquema de gestión de memoria más flexible era necesario. Durante el desarrollo del primer compilador a principio de los años 50 surge el concepto de gestión de memoria dinámica. La memoria dinámica dio respuesta a los problemas que conlleva la asignación estática, planteando nuevos retos. Algunos de estos problemas todavía hoy en día son considerados cuestiones abiertas.

El objetivo de un gestor de memoria dinámico es atender las peticiones de memoria de una aplicación o sistema en el menor tiempo posible, empleando para ello la menor cantidad de memoria posible.

Históricamente, el aspecto que más interés ha suscitado entre los investigadores ha sido el problema de la *fragmentación*. El cual se puede definir como:

> "La incapacidad de un gestor de memoria de asignar un bloque de tamaño r, a pesar de que el total de memoria asignada hasta el momento más r sea inferior o igual al total de la memoria libre."

Actualmente, la memoria dinámica es una parte indispensable de una gran multitud de sistemas. Por ejemplo, la mayor parte de los lenguajes orientados a objetos no podrían existir sin memoria dinámica.

Los sistema de tiempo real constituyen un área donde la utilización de memoria dinámica, a pesar de los beneficios que aporta, no ha tenido nunca cabida. Estos sistemas se distinguen del resto debido a que su corrección no depende solamente de su lógica interna sino, también, del tiempo en el que la respuesta es obtenida. Hasta el momento dos han sido las razones argumentadas por los diseñadores de este tipo de sistemas para no usar memoria dinámica:

- Tal como expresa I. Puaut en [96, 97], existe la creencia generalizada de que la gestión de memoria dinámica es en sí misma un problema *inherentemente* no determinista.

- A pesar de conocer con antelación la memoria total requerida por el sistema, el gestor de memoria podría fracasar debido al problema de la fragmentación.

Sin embargo, debido, en primer lugar, a la creciente complejidad de las aplicaciones de tiempo real y, en segundo lugar, a la creciente expansión de lenguajes orientados a objetos (Java) y su adopción en este tipo de sistemas, el interés por el uso de memoria dinámica en sistemas de tiempo real es creciente.

1.2. Objetivos de la tesis y contribuciones

La presente tesis se centra en el estudio de la gestión de memoria en el ámbito de sistemas de tiempo real, con la fuerte convicción de que, en pocos años, el uso de memoria dinámica en este tipo de sistemas será una práctica habitual.

En concreto, los objetivos principales de esta tesis son:

- Revisión del estado de la gestión de memoria dinámica y análisis de su problemática, haciendo especial énfasis en el problema de la fragmentación.
- Desarrollo de un nuevo gestor de memoria dinámica explícita apto para sistemas de tiempo real con tiempo de respuesta determinista y rápido y que produzca una baja fragmentación.
- Estudio comparativo, tanto a nivel práctico como teórico, del tiempo de respuesta del nuevo gestor de memoria.
- Estudio comparativo, tanto a nivel práctico como teórico, del comportamiento espacial del nuevo gestor de memoria.

Las principales contribuciones aportadas por la presente tesis son:

- Realización de un exhaustivo estudio del estado del arte de la gestión de memoria dinámica hasta la actualidad.
- Propuesta de un nuevo gestor de memoria dinámica explícita llamado Two-Level Segregated Fit (TLSF), cuyas principales características son: un tiempo de respuesta rápido y constante ($O(1)$) y una fragmentación relativamente baja.
- Análisis comparativo, a nivel teórico y practico, del tiempo de respuesta presentado por el gestor propuesto.
- Análisis comparativo, a nivel teórico y practico, del comportamiento espacial (fragmentación) presentado por el gestor TLSF.

- Extensión al modelo de tareas periódicas de tiempo real que modela el uso de memoria dinámica por parte de las tareas.
- Implementación de un simulador para el nuevo modelo de tareas propuesto.
- Implementación de un sistema mínimo de ejecución (SA-Tester), el cual permite estudiar el comportamiento de una aplicación sin las clásicas interferencias producidas por el sistema operativo convencional (fallos de página, interrupciones, interacción con otras aplicaciones).

1.3. Organización de la tesis

La presente tesis se encuentra estructurada en siete capítulos y cuatro apéndices.

- *Capítulo 2*: introducción a los sistemas de tiempo real y el modelo de tareas subyacente, resumiendo las contribuciones más destacadas de las distintas políticas de planificación existentes.
- *Capítulo 3*: revisión del estado del arte de la gestión de memoria explícita y su problemática, el problema de la fragmentación. Además se incluye una presentación de los gestores de memoria dinámica existentes.
- *Capítulo 4*: descripción del gestor de memoria Two-Level Segregated Fit (TLSF), específicamente diseñado para ser utilizado en sistemas de tiempo real.
- *Capítulo 5*: análisis asintótico y experimental del tiempo de respuesta del gestor TLSF. En este mismo capítulo también se comparan los resultados obtenidos con los de los gestores First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL y Half-Fit.
- *Capítulo 6*: análisis tanto a nivel teórico como experimental de la respuesta espacial (fragmentación) del gestor TLSF. En este mismo capítulo también se comparan los resultados obtenidos con los de los gestores First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL y Half-Fit.
- *Capítulo 7*: conclusiones más importantes del trabajo realizado y esbozo de las posibles líneas de trabajo futuro.
- *Apéndice A*: demostración de la función cota superior para la fragmentación interna del gestor de memoria TLSF.

- *Apéndice B*: descripción y estudio de las aplicaciones utilizadas en las diferentes evaluaciones realizadas al gestor TLSF en los capítulos 5 y 6. Además, se incluyen las funciones de distribución usadas por el modelo CDF, modelo propuesto por B. Zorn en [148], para la evaluación del tiempo de respuesta del gestor TLSF en el capítulo 5.
- *Apéndice C*: descripción del diseño del entorno de pruebas implementado específicamente para realizar la evaluación experimental del tiempo de respuesta del gestor TLSF en el capítulo 5.
- *Apéndice D*: breve reseña de la evolución del desarrollo del gestor TLSF (versiones e implementaciones) y descripción de sistemas que están actualmente utilizado el gestor de memoria TLSF.

Capítulo 2
Sistemas de tiempo real

Los sistemas de tiempo real son aquellos en los cuales la corrección del resultado no solo depende de la lógica interna del propio sistema sino también del tiempo. En el presente capítulo se realiza una introducción a los mismos.

2.1. Introducción

La tarea principal de un programa es transformar una secuencia de entrada, o estímulos, en una secuencia de salida. Para la mayoría del software, el concepto corrección significa que la lógica del programa ha sido la esperada. La consumición de una gran cantidad de tiempo por parte del programa no suele tener relación con la corrección del mismo, sino con su eficiencia. Ejemplos de esto sistemas son hojas de cálculo, procesadores de texto, etc.

Existe, sin embargo, una serie de sistemas en los cuales la producción de una salida correcta no solo depende de la lógica interna del mismo, sino principalmente de su tiempo de respuesta. Este tipo de sistemas, conocidos como sistemas de tiempo real, incluyen como principal requisito el tiempo.

Los principales campos de aplicación de los sistemas de tiempo real son los sistemas de control. Típicamente, un sistema de control está compuesto por un conjunto de sensores, una o varias unidades de proceso y, por último, un conjunto de actuadores. El software que se ejecuta en las unidades de proceso tiene el deber de monitorizar, a través de los sensores, el estado del sistema que controla, evaluando las posibles acciones de control y enviando a los actuadores las señales de control. En este tipo de sistemas, el sistema informático de control ha de cumplir una serie de requisitos temporales impuestos por la propia dinámica del sistema físico controlado y por el algoritmo de control utilizado. Ejemplos de este tipo de aplicaciones pueden ser: sistemas de producción industrial, control de radares y otros sistemas monitorizados, guiado de vehículos, control del tráfico aéreo, sistemas médicos, etc.

No hay que confundir entre un sistema *en* tiempo real y un sistema *de* tiempo real. Los primeros son sistemas en los que el factor más importante es la velocidad de respuesta para conseguir un funcionamiento realista. En estos sistemas, un retraso no provoca una situación peligrosa. Por contra, la característica más importante de los sistemas de tiempo real es la garantía y no la rapidez. Por garantía se entiende que se ha de poder predecir y asegurar que el sistema será capaz de ejecutar correctamente todos los trabajos encomendados [127].

2.2. Clasificación de los sistemas de tiempo real

El grado de requerimiento de determinismo temporal puede ser utilizado para clasificar los diferentes tipos de sistemas de tiempo real existentes y diferenciarlos de los que no lo son.

2.2.1. Sistemas de procesamiento por lotes

Muchas aplicaciones no tienen ningún requerimiento temporal en absoluto. Un ejemplo de este tipo de aplicaciones son los compiladores. La entrada para los programas de este tipo se prepara y está disponible antes de la ejecución del programa. La salida se calcula a partir de la entrada, después de esto el programa termina. Los sistemas diseñados para soportar este tipo de aplicaciones son conocidos como sistemas de procesamiento por lotes, debido a que la información se procesa por lotes. La corrección de la salida no se ve afectada por el tiempo que se ha tardado en producirla. Sin embargo, cierta eficiencia en el procesamiento es, por supuesto, siempre deseable, aunque no necesario. Por lo tanto, los sistemas de procesamiento por lotes no pueden ser considerados sistemas de tiempo real.

2.2.2. Sistema interactivos

Los computadores modernos a menudo interactúan con el usuario durante el procesamiento de la información. El usuario envía el comando necesario al programa, el cual ejecuta la acción pedida y muestra el resultado. Entonces, un nuevo comando puede ser enviado. Este tipo de sistemas se conocen como sistemas interactivos. Un procesador de textos es un ejemplo de este tipo de programas. Para que la interacción funcione correctamente, el tiempo de respuesta para ejecutar un comando tiene que ser razonablemente corto. Sino, el usuario podría considerar que el sistema va lento y si el retraso es inesperado, quizás el estado de la operación pudiera ser malinterpretado. Si, por ejemplo, un procesador de textos tardara varios segundos en mostrar una tecla pulsada por el usuario, el usuario podría interpretar que el procesador de textos no ha recibido correctamente la tecla pulsada, pulsándola más veces. El resultado no sería, por lo tanto, el esperado.

Los sistemas interactivos presentan a menudo algún grado de requerimientos temporales, incluso si estos son relativamente laxos. Tiempos de respuesta de hasta medio segundo son considerados, a menudo, aceptables para este tipo de aplicaciones. Retrasos menores de 0,1 segundos pueden ser en la mayoría de los casos no percibidos por todos los seres humanos. La perdida de un plazo en este tipo de sistemas no resulta crítica y no provoca la degradación del sistema. En un procesador de textos, por ejemplo, el usuario podría borrar algún carácter extra por error. Sin embargo, este puede ser fácilmente repuesto sin afectar, en absoluto, el funcionamiento del sistema.

Este tipo de sistemas, aunque pueda parecer que tienen requerimientos temporales, no pueden ser considerados sistemas de tiempo real sino sistemas en tiempo real, ya que el incumplimiento de estos no afecta al

funcionamiento del sistema ni lo degrada.

2.2.3. Sistemas de tiempo real no críticos

Normalmente, cuando un computador controla algún tipo de equipamiento externo, suelen aparecer ciertos requerimientos temporales. Los sistemas informáticos empotrados frecuentemente pertenecen a esta categoría. El tiempo de respuesta requerido es típicamente más corto que el de los sistemas interactivos, encontrándose a menudo en el rango comprendido entre 10 a 100 milisegundos. Para mantener el control sobre los equipamientos externos es importante que el sistema cumpla los plazos temporales. En los sistemas de tiempo real no críticos, sin embargo, perdidas de plazos ocasionales pueden ser toleradas.

Un ejemplo de un sistema de tiempo real no crítico puede ser un el control de un teléfono. Para servir eficientemente a un cliente, este control tiene que responder rápidamente a las acciones realizadas por la persona que llama. Cuando la persona que llama levanta el auricular, el teléfono debe generar el tono de dial y debe de estar preparado para aceptar un número de teléfono. Si el teléfono falla al hacer esto, la persona que llama podría tener la impresión de que el servicio no esta disponible en ese momento. Incluso aunque el sistema falle, la integridad del sistema no se ve perjudicada.

La línea divisoria entre los sistemas interactivos y los sistemas de tiempo real no críticos es a menudo difícil de dibujar. Un sistema de reproducción de audio podría, por ejemplo, ser considerado un sistema de tiempo real no estricto debido a sus plazos de respuesta temporal.

2.2.4. Sistemas de tiempo real críticos

Algunos sistemas de tiempo real están compuestos por procesos con requerimientos temporales críticos. La perdida de un plazo en este tipo de sistemas provoca a menudo el fallo completo del sistema, [20]. Muchos sistemas de control automático pertenecen a esta categoría. Este tipo de sistemas son, por ejemplo, un sistema de navegación aérea, el control de un brazo robot, el control de una planta industrial.

Este tipo de sistemas suelen requerir tiempos de respuesta muy rápidos, de orden menor al milisegundo. La pérdida de algún plazo en este tipo de sistemas suele provocar que los algoritmos de control se vuelvan inestables con resultados catastróficos.

2.3. Modelo de sistema

Un sistema de tiempo real está compuesto por varias tareas, al menos dos, que colaboran entre sí. Estas tareas pueden ser clasificadas en dos clasificaciones diferentes dependiendo del criterio que empleemos.

Si consideramos la importancia del trabajo que realiza cada tarea en el mantenimiento de la integridad del sistema, la clasificación existente es:

Críticas: Un fallo en una de estas tareas puede conducir al sistema a una situación catastrófica. El fallo puede deberse o bien a un error en el código del programa o bien a un retraso en la obtención del resultado.

Acrítica: Tareas que colaboran en el funcionamiento del sistema sin comprometer la integridad de éste, en caso de fallar o incluso de no ejecutarse.

En cambio, si atendemos a las características temporales de cada tarea, las tareas se pueden clasificar en:

Periódicas: Tareas que se ejecutan repetidamente con una cadencia fija. Cada activación ha de finalizar antes de un determinado plazo máximo de respuesta. Tres son los parámetros que caracterizan estas tares, $T_i = (C_i, D_i, P_i)$ [72, 27]:

- C_i: Tiempo de cómputo, en el peor de los casos, requerido en cada activación.
- D_i: Plazo relativo. Tiempo máximo del que dispone la activación para completar su ejecución. Si por cualquier causa, el resultado de la ejecución no está disponible en D_i unidades de tiempo (u.t.) después de iniciarse una activación, diremos que se ha producido un fallo.
- P_i: Periodo de activación. Distancia temporal entre dos peticiones consecutivas.

En los primeros trabajos de tiempo real [72], se asumía que el plazo coincidía con el periodo. Actualmente se suele eliminar esta restricción e incluso en algunos estudios, por ejemplo en [10], se pueden encontrar tareas con plazos mayores al periodo. En el presente capítulo asumiremos siempre que $D_i \leq P_i$.

En algunos sistemas, no todas las tareas han de comenzar su ejecución tan pronto se inicia el sistema, sino que algunas tareas tienen que comenzar su ejecución con un cierto *desfase*, S_i, desde el arranque del sistema. Diremos que un conjunto de tareas es *síncrono* si el desfase de todas las tareas periódicas es igual a cero.

Aperiódicas: Estas tareas, J_i, han de atender a eventos que aparecen de forma impredecible, como por ejemplo una orden dada por un operador o la aparición de un obstáculo. En función a su urgencia podemos, a su vez, subdividirlas en los siguientes grupos:

Sin plazo: Su ejecución no es crítica. No tienen ningún plazo límite para su finalización. Pueden emplearse para mejorar la precisión de las acciones de control o generar informes del estado del sistema. Si bien no tienen plazos límite de finalización, es deseable que el planificador las ejecute lo antes posible para conseguir un corto tiempo de respuesta. El único parámetro que las caracteriza es el tiempo de cómputo, $J_i = (C_i)$.

Con plazo firme: Al igual que las tareas aperiódicas sin plazo, éstas tampoco son críticas. El resultado de su ejecución sólo tiene utilidad para el sistema si finaliza antes de que expire su plazo. En caso de que no se pueda cumplir el plazo, el planificador debe cancelar lo antes posible su ejecución, con el fin de no *malgastar* tiempo de procesador. Este tipo de tareas se definen en base a dos parámetros, $J_i = (C_i, D_i)$.

Esporádicas: Este tipo de tareas poseen un plazo máximo de ejecución, y su incumplimiento puede resultar catastrófico para el sistema [84]. Por ello, el planificador debe garantizar sus correcta ejecución. Los parámetros de esta clase de tareas son los mismo que los de las tareas periódicas, con la salvedad que el periodo en lugar de indicar el tiempo entre activaciones representa la distancia mínima entre dos activaciones consecutivas, $T_i = (C_i,\, D_i,\, P_i)$.

En todos los trabajos [72, 70, 85, 126, 142] para garantizar la correcta ejecución, se basan en el conocimiento explícito de los periodos de las tartas o, al menos, en el conocimiento del tiempo mínimo entre peticiones. Por consiguiente, aquellas tareas consideradas críticas han de ser transformadas en periódicas, en caso de no lo sean. En concreto, las tareas esporádicas se han de considerar como si fueran periódicas durante la fase de diseño y análisis del sistema.

A partir de este punto asumiremos que tanto las características temporales de las tareas (tiempo de cómputo, plazo y periodo) como el tiempo (t) están representados por variables enteras. Este convenio no supone ningún tipo de restricción en la aplicación de la teoría a casos prácticos, y simplifica en gran medida la compresión de los resultados presentados. En cualquier caso, el extender los resultados para valores reales del tiempo no entraña

cambios fundamentales en las demostraciones presentadas. Una detallada discusión se puede encontrar en [11].

Aparte de las características temporales de las tareas, en los primeros estudios de tiempo real se asumían la siguiente serie de restricciones, si bien conforme ha ido evolucionando y madurando la teoría se han ido eliminando algunas restricciones:

- Todas las tareas periódicas son críticas.
- Todas las tareas pueden ser expulsadas del procesador en cualquier instante.
- Todas las tareas comienzan su ejecución tan pronto están activas.
- $C_i \leq D_i \leq P_i$.
- La sobrecarga debida al cambio de contexto se supone despreciable.
- No existe ninguna dependencia entre tareas, es decir, las tareas no comparten recursos comunes y no existe ninguna relación de precedencia entre ellas. Un conjunto de tareas con relaciones de precedencia puede ser transformado en otro nuevo conjunto sin problemas de precedencia mediante la modificación de los plazos [28].
- Ninguna tareas se suspende voluntariamente, a excepción de las expulsiones causadas por el algoritmo de planificación.

Llamaremos A_x a cada una de las activaciones que una tarea periódica o esporádica realiza. La activación k-ésima de un tarea T_i se produce en el instante $(k-1)\cdot P_i$. Para ejecutar correctamente la activación, el planificador deberá asignarle C_i u.t. a esta tarea en el intervalo $[(k-1)\cdot P_i, (k-1)\cdot P_i + D_i[$. Una activación está definida por la terna (s_x, c_x, d_x), donde $s_x = (k-1)\cdot P_i$ es el instante de inicio, $c_x = C_i$ el tiempo de cómputo, y $d_x = (k-1).\ P_j + D_j$ el plazo (absoluto) de finalización. Las activaciones se supondrán ordenadas por el instante de finalización d_x, de modo que $x < y \rightarrow d_x \leq d_y$. Obsérvese que se utilizan letras minúsculas para describir activaciones, mientras que las letras mayúsculas representan los parámetros de las tareas.

Representaremos por τ al conjunto de tareas periódicas (y esporádicas) que componen la carga del procesador. Asumiremos que n es el cardinal del conjunto τ. El parámetro más importante de la carga periódica es el *factor de utilización* del procesador, $\mathcal{U}_\tau$, el cual se define como:

$$\mathcal{U}_\tau = \sum_{i=1}^{n} \frac{C_i}{P_i}$$

El *hiperperiodo*, $\mathcal{P}$, es otra característica importante del conjunto de tareas. Se define como el mínimo común múltiplo del periodo de todas las tareas de τ, es decir, $\mathcal{P} = mcm(P_1, P_2, ..., P_n)$.

Un conjunto de tareas, τ, es **planificable** si y sólo si existe algún algoritmo de planificación capaz de cumplir los plazos de todas las activaciones de τ. Obsérvese que la condición de planificabilidad es una propiedad del conjunto de tareas y no del planificador. Un conjunto de tareas τ, es planificable por un planificador dado si y sólo si éste es capaz de cumplir los plazos de todas las activaciones de τ.

Partiendo de estas definiciones queda claro que pueden existir conjuntos de tareas que sean planificables pero que un determinado algoritmo de planificación sea incapaz de planificarlas correctamente. Se dice que un planificador es *óptimo* si y sólo si puede planificar correctamente todo conjunto de tareas planificables.

2.4. Requisitos de planificación

En los sistemas de informáticos clásicos, el objetivo principal del planificador suele ser minimizar el tiempo de respuesta de las tareas, es decir, Shorter Job First (SJF), o repartir lo más equitativamente posible el tiempo de procesador entre las distintas tareas, Round Robin (RR), o por último, evitar realizar cambios de contexto con el First Come First Served (FCFS). Ninguna de estas políticas es apropiada en sistemas de tiempo real [126, 128].

El principal objetivo de cualquier política de planificación de tiempo real debe de ser el asegurar que todas las tareas cumplan sus restricciones temporales. Además de este objetivo básico, es deseable que la política de planificación considere las siguientes cuestiones:

- Gestión de los recursos cuando son compartidos entre las distintas tareas del sistema.
- Recuperación de fallos.
- Minimización del tiempo de respuesta de todas las tareas.
- Ejecución de tareas acríticas sin plazo.
- Reemplazo de tareas en tiempo de ejecución (cambio de modo).
- Considerar la sobrecarga temporal del propio algoritmo de planificación, del cambio de contexto, etc.
- Planificación de tareas cuyo tiempo de computo no es conocido o predecible.

- Si es posible, conviene que sea sencillo y fácil de utilizar en realizaciones de prácticas.

La única forma de poder garantizar que un determinado algoritmo de planificación va a poder planificar correctamente un determinado conjunto de tareas es mediante el análisis, previo a la puesta en marcha del sistema, mediante métodos analíticos, o de forma exhaustiva. Como se apunta en [142], la simulación no es un método válido que garantice la correcta planificación, ya que:

> "La simulación muestra la presencia de errores, no la ausencia de estos" [33].

Por consiguiente, todo algoritmo de planificación de tiempo real ha de ir acompañado obligatoriamente de un test de planificación (test de garantía). Evidentemente, cualquier extensión del algoritmo de planificación se ha de reflejar en el test de garantía.

2.5. Planificación estática

También conocida como *ejecutivo cíclico* (ciclic scheduling) o *dirigido por tiempo* (time-driven) [63, 146, 142], es la clase de planificadores más empleados en la actualidad, debido a que fueron los primeros en aparecer. La mayor parte del trabajo de planificación en este tipo de planificadores se realiza durante el diseño del sistema. Se construye una tabla (llamada también plan estático o calendario) indicando los instantes en los que cada tarea debe ponerse en ejecución y cuando ha de finalizar. Cuando el sistema está en ejecución, lo único que ha de hacer el planificador es seguir repetidamente las instrucciones almacenadas en el plan. El tamaño de la tabla está determinado por el hiperperiodo del conjunto de tareas, lo que aveces resulta en un tamaño excesivo de ésta. La garantía de correcta ejecución de todas las tareas (análisis de planificabilidad) se hace implícitamente durante la construcción del propio plan.

Como principales ventajas de esta política podemos citar la predecibilidad y eficiencia en tiempo de ejecución. Mientras que entre los inconvenientes tenemos su ineficiencia para gestionar eventos aperiódicos, la poca flexibilidad para modificar las características de la carga (un pequeño cambio en alguna de las tareas implica el tener que reconstruir todo el plan), y el posible excesivo tamaño de la tabla.

A pesar de ello, J. Xu y D.L. Parnas hicieron una férrea defensa de los planificadores cíclicos en [142], y defendieron que algunos tipos de problemas sólo pueden ser resueltos con este tipo de algoritmos.

2.6. Planificación por prioridades

En contraposición a los planificadores cíclicos tenemos los planificadores controlados por prioridades. En estos, se elimina la necesidad de un plan prefijado, dejando la toma de decisiones en manos del planificador en ejecución.

El esquema básico de funcionamiento es el siguiente: En cada instante (típicamente cuando llega o finaliza alguna tarea), el planificador elige aquella tarea cuya *función de prioridad* tenga mayor valor de entre todas las tareas activas. Dependiendo de cual sea la función de prioridad tendremos: planificadores con *prioridades fijas* si a cada tarea se le asigna una prioridad fija inicialmente, y se emplea esta valor como función de prioridad; o planificadores con *prioridades dinámicas* en caso de que la función de prioridad tenga como parámetros datos relativos a la carga que haya en el instante de su evaluación.

Estos métodos precisan de un test previo que garantice que la política concreta de planificación empleada será capaz de planificar correctamente todas las tareas críticas del sistema.

2.6.1. Prioridades fijas

Dentro del esquema de prioridades fijas, los dos métodos más extendidos son el de *prioridad a la tarea más frecuente* (Rate Monotonic o RM) y el de *prioridad a la tarea más urgente* (Deadline Monotonic o DM). En los últimos años, los esquemas de planificación basados en prioridades fijas han recibido una gran atención por parte de la comunidad científica, alcanzando un alto grado de madurez y reconocimiento [125, 69, 68, 115, 20, 4]. En [6] se puede encontrar un estudio histórico de los avances en la planificación por prioridades fijas.

El RM fue introducido por C.L. Liu y J.W. Layland en [72], y consiste en asignar a las tareas de periodos cortos prioridades altas, esto es, la prioridad de cada tarea es inversamente proporcional a su periodo. Para poder aplicar esta política, el plazo de respuesta de cada tarea ha de coincidir con su periodo. La obtención del primer test de garantía se basó en la siguiente observación:

> Sea τ un conjunto de tareas periódicas. El tiempo de respuesta más largo para cualquier tarea ocurre cuando se activan todas las tareas al mismo tiempo. A este instante se le denomina instante crítico.

Definición 1. **(C.L. Liu y J.W. Layland [72])** Si el conjunto de tareas es síncrono (todas las tareas tienen la misma fase), el instante crítico inicial

es aquel que se produce a partir del instante cero.

El siguiente test representa una condición suficiente de planificabilidad:

Teorema 1. **(C.L. Liu y J.W. Layland [72])** El conjunto de tareas τ es planificable bajo el RM si:

$$\mathcal{U}_\tau \leq n \cdot (2^{1/n} - 1)$$

Posteriormente J.P. Lehoczky, L. Sha y Y. Ding [68] propusieron el siguiente método exacto de análisis de planificabilidad. Está basado en la utilización de cada tarea, condición que se da durante el instante crítico inicial.

Teorema 2. **(J.P. Lehoczky, L. Sha y Y. Ding [68])** El conjunto de tareas τ es planificable bajo el RM si y sólo si:

$$\forall_{1 \leq i \leq n} \min_{t \leq 0 \leq P_i} \sum_{\forall j \in hp(i)} \frac{C_i}{t} \cdot \left\lceil \frac{t}{P_j} \right\rceil \leq 1$$

J. Leung y J. Whitehead extendieron en [71] el RM para trabajar con tareas que tuvieran plazos menores que el periodo, asignando mayor prioridad a tareas con plazos más cortos, y demostrando que esta asignación de prioridades es óptima entre todas las asignaciones estáticas. Estamos hablando del DM, quedando el RM como un caso particular del DM. J.P. Lehoczky, L. Sha y Y. Ding ampliaron en [68] el análisis de planificabilidad para el DM, obteniendo como resultado un test muy similar al del RM:

Teorema 3. **(J.P. Lehoczky, L. Sha y Y. Ding [68])** El conjunto de tareas τ es planificable bajo el DM si y sólo si:

$$\forall_{1 \leq i \leq n} \min_{t \leq 0 \leq D_i} \sum_{\forall j \in hp(i)} \frac{C_i}{t} \cdot \left\lceil \frac{t}{P_j} \right\rceil \leq 1$$

El test consiste en verificar que la primera activación de cada tarea es capaz de cumplir su plazo de ejecución teniendo en cuenta la cantidad de tiempo de cómputo pedido por tareas más prioritarias. Este test no sólo puede ser empleado con asignaciones de prioridades basadas en el DM, sino que es un test exacto para cualquier asignación de prioridades [67]. Resultados similares fueron obtenidos por N.C. Audsley et al. [7].

La compartición de recursos entre distintas tareas también ha sido tenida en cuenta dentro de la teoría del DM. Algoritmos tales como el protocolo del techo de prioridad (Priority Ceiling Protocol o PCP) [114, 115] o el protocolo del techo del semáforo (Ceiling Semaphore Protocol o CSP) [8, 100], por nombrar algunos, se pueden emplear para gestionar el acceso a secciones

críticas. A la hora de analizar la planificabilidad del sistema, tan sólo se ha de incluir un termino que representa el factor de bloqueo en la inecuación del test. El factor de bloqueo representa la cantidad de tiempo que una tarea está bloqueada a causa de tareas menos prioritarias que están dentro de secciones críticas.

Una revisión interesante sobre los distintos métodos de control de recursos se puede encontrar en [5].

El reemplazo de tareas en tiempo de ejecución y los cambios de modo ha sido estudiados en [116, 2, 102].

2.6.1.1. Servicio de tareas aperiódicas

Otro aspecto importante que no ha sido descuidado y que ha desencadenado una gran cantidad de resultados [6], es el servicio de tareas aperiódicas. Desde los servidores de segundo plano (*background*) o por consulta (*pooling*) [113], con muy poca sobrecarga pero con pobres resultados; pasando por los algoritmos de mantenimiento del ancho de banda (*bandwidth-preserving*), como el servidor diferido (*deferable server*) [69], el intercambio de prioridades (*priority exchange*) [121], o el servidor esporádico (*sporadic server*) propuesto por B. Sprunt et al. [122], que ofrece un mejor tiempo de respuesta a las tareas aperiódicas a costa de una mayor complejidad en la gestión del tiempo disponible; para finalizar con los algoritmos de extracción de holgura, que estresan al máximo las tareas periódicas (sin llegar a producir la pérdida de ningún plazo) para conseguir dedicar la mayor cantidad de tiempo posible a servir tareas aperiódicas. Estas últimas propuestas son óptimas, esto es:

Teorema 4. **[101, 31]** Para cualquier conjunto de tareas periódicas planificado por un planificador por prioridades fijas y cualquier secuencia de peticiones aperiódicas servidas en orden FIFO, los extractores de holgura minimizan el tiempo de respuesta de cada una de las tareas aperiódicas, de entre los algoritmos que garantizan el cumplimiento de todos los plazos de todas las tareas periódicas (exceptuando los algoritmos clarividentes).

Dos son las propuestas basadas en la extracción de holgura, una estática y otra dinámica. En la versión estática [101] se dispone de una tabla en la que se tienen almacenados los instantes en los que se le puede *robar* tiempo de cómputo a las tareas periódicas. Cada vez que hay que servir una tarea aperiódica, se consulta esta tabla para buscar huecos de holgura disponibles, para asignárselos a la tarea. El espacio requerido para almacenar la tabla de consulta puede llegar a ser totalmente intolerable en una realización práctica, de hecho, tal como comentan los propios autores, la principal utilidad del algoritmo es la de servir de referencia para evaluar el rendimiento de otros algoritmos.

R.I. Davis, K.W. Tindell y A.J. Burns [31] propusieron la versión dinámica. El cálculo de la holgura se realiza totalmente en tiempo de ejecución, sin emplear para ello ninguna tabla precalculada. Esta solución evita la necesidad de un gran espacio de memoria para almacenar la tabla, pero por contra, sufre una mayor sobrecarga en tiempo de ejecución, debida al mayor número de cálculos necesarios.

2.6.2. Prioridades dinámicas

Los dos algoritmos más significativos, dentro de esta clase, son el de prioridad a la tarea más urgente (*Earliest Deadline First* o EDF) y el de prioridad a la tarea con menor holgura (*Least Laxity First* o LLF). El EDF asigna mayor prioridad a las activaciones con plazos más próximos. Obsérvese que a diferencia del planificador DM, donde las prioridades eran asignadas inicialmente (la prioridad era función del plazo de ejecución relativo máximo, D_i), el planificador EDF evalúa los plazos absolutos de las activaciones y asigna prioridad en función de estos plazos (la prioridad es una función del plazo ejecución absoluto máximo, d_i). El planificador LLF asigna mayor prioridad a las activaciones con menor holgura, entendiendo holgura como la longitud del intervalo desde el instante actual hasta el plazo de la activación menos el tiempo de cómputo que le reste por ejecutar.

Ambos algoritmos son óptimos (el algoritmo EDF deja de ser óptimo en sistemas multiprocesadores) [32, 86]. Por consiguiente, si alguna tarea pierde su plazo es seguro que ningún otro algoritmo habría sido capaz de evitarlo. Debido a esta característica, tanto EDF como LLF consiguen aprovechar al máximo la potencia del procesador, garantizando más tareas periódicas críticas que cualquier otro algoritmo. Esta clase de algoritmos se adaptan muy bien a entornos altamente dinámicos, en los cuales la carga del sistema puede no ser conocida inicialmente.

A pesar de estas interesantes propiedades, y a diferencia de la atención prestada a la planificación por prioridades fijas, esta clase de planificadores ha estado relegada, hasta hace algunos años, a un segundo plano dentro de la comunidad de tiempo real, debido principalmente a que presentaban los siguientes problemas:

1. La sobrecarga del planificador es mayor a la causada por planificadores estáticos.
2. No es estable ante sobrecargas. En caso de que alguna tarea consuma más tiempo del estimado, no hay forma de saber qué tarea perderá su plazo, si es que alguna lo pierde.
3. La mayoría de los sistemas operativos actuales disponen de un planificador basado en prioridades fijas, por lo que sería necesario construir

un planificador a medida.

C.L. Liu y J.W. Layland en [72] demostraron que el algoritmo EDF es capaz de garantizar la correcta ejecución de todas las tareas cuando la utilización del procesador es menor del 100 % (cuando el plazo era igual que el periodo). Por tanto, es óptimo para este tipo de tareas.

M. Dertouzos en [32] demostró que el EDF es también óptimo para tareas con plazos menores al periodo.

J. Leung y M.L. Merril [70] observaron que la planificación producida por cualquier planificador expulsivo se repite cíclicamente cada hiperperiodo. En otras palabras, el planificador realiza las mismas acciones en el instante t que en $t + k \cdot \mathcal{P}$, siendo k un número entero. Empleando esta importante propiedad, propusieron el primer test de planificabilidad para tareas con plazos menores al periodo. Un conjunto de tareas periódicas es planificable si y sólo si ninguna tarea pierde ningún plazo durante el primer hiperperiodo.

Un test de planificabilidad más eficiente fue propuesto por S.K. Baruah et al. [11], demostrando que para la mayor parte de conjuntos de tareas no es necesario comprobar la correcta ejecución de todas las tareas a lo largo de todo el hiperperiodo, sino que basta comprobarla durante un pequeño intervalo. También, en el mismo artículo, se demuestra que el problema de la planificabilidad para monoprocesadores es co-**NP**-completo en el sentido fuerte. Una característica que parecen poseer los problemas de este tipo es que pequeñas variaciones en la instancia del problema (en nuestro caso, podría consistir en modificar el periodo o el tiempo de cómputo de alguna tarea) resultan en importantes variaciones en la complejidad. Gracias a esta característica, el análisis de planificabilidad se ve notablemente simplificado en los casos reales.

I. Ripoll en [103] demostró el siguiente test de planificabilidad:

Teorema 5. **(I. Ripoll [103])** Sea $L = \dfrac{\sum_{i_1}^{n} C_i \cdot (1 - D_i/P_i)}{1 - \mathcal{U}_\tau}$ y $[0, \mathcal{R}[$ el intervalo crítico inicial (ICI). τ es planificable si y sólo si:

$$\forall t < \text{mín}(L, \mathcal{R}), H_\tau(t) \leq t$$

M.I. Chen y K.J. Lin [26] estudiaron el problema de la compartición de recursos para el algoritmo de prioridad más urgente. El protocolo del techo de prioridad dinámico (Dynamic Priority Ceiling Protocol o DPCP) evita los interbloqueos y los bloqueos encadenados.

Otro enfoque sobre el uso de recursos es el ofrecido por A.K. Mok en [84]. Las tareas que están ejecutando código dentro de alguna sección crítica no son expulsadas, pero a cambio el tiempo que permanecen dentro de la sección crítica ha de ser reducido.

2.6.2.1. Servicio de tareas aperiódicas

En lo relativo a la planificación conjunta de tareas periódicas y aperiódicas, los primeros trabajos fueron realizados por H. Chetto y M. Chetto [27] que investigaron la localización de los intervalos ociosos del procesador, y propusieron el algoritmo *earliest deadline last o EDL*. El EDL trata de planificar las tareas periódicas lo más tarde posible (pero sin llegar a violar ningún plazo), con lo que los intervalos ociosos son *desplazados* hacia adelante, pudiendo ser empleados para dar servicio a trabajo aperiódico. H. Chetto, M. Silly y T. Bouchentouf en [28] propusieron un test óptimo de aceptación de grupos de tareas aperiódicas para ser planificadas bajo el EDF. El conjunto de tareas aperiódicas es aceptado si ninguna tarea pierde su plazo en el intervalo $[t_{now}, t_{now} + \mathcal{P} + D_\alpha[$, donde T_{now} es el instante actual y D_α el plazo más largo de todas las tareas aperiódicas. El coste del algoritmo es alto, $O(N^2)$, siendo N el número de activaciones de tareas periódicas durante un hiperperiodo.

En 1990, S.K. Baruah, A.K. Mok y L.E. Rosier [10] presentaron un test de planificabilidad para tareas esporádicas que puede ser empleado para aceptar tareas aperiódicas con plazo firme. La complejidad de este algoritmo es ligeramente inferior al propuesto por H. Chetto et al. En el peor caso el coste es $O(n^2 \cdot \max(P_i - D_i)/\min(P_i))$, siendo n el número de tareas periódicas existentes en el sistema.

K. Schwan y H. Zhow [110] propusieron un algoritmo de planificación dinámico basado en el EDF y con un coste $O(n \cdot \log(n))$. El problema de su propuesta radica en que todas las tareas, incluidas las tareas periódicas, deben de pasar un test de aceptación. Las tareas que son rechazadas son tratadas por otro planificador de más alto nivel.

Los tres algoritmos de conservación de ancho de banda: priority exchange, deferable server y sporadic server han sido adaptados para trabajar con el algoritmo EDF por T.M. Ghazalie y T.P. Baker [49] y posteriormente por M. Spuri y G.C. Buttazzo [123]. A pesar de no ser algoritmos óptimos, mantienen un buen compromiso entre coste y eficiencia.

T.S. Tia, J.W. Liu y M. Shankar [135] desarrollaron un servidor aperiódico basado en la extracción de holgura. Lo más llamativo de esta solución es su elevado coste espacial, $O(N^2)$. Otro problema es que en cada punto de planificación se han de actualizar ciertas variables con un coste $O(n)$. Aunque el servidor es óptimo, el elevado coste espacial lo convierte en poco práctico para aplicaciones reales.

2.7. Gestión de memoria dinámica

Actualmente, el uso de memoria dinámica en sistemas de tiempo real es prácticamente inexistente. De hecho, solamente la especificación de Java para sistemas de tiempo real [34] aborda la problemática de memoria dinámica en este tipo de sistemas.

Por lo general, el único uso de memoria dinámica en los sistemas de tiempo real se produce durante la inicialización y terminación de los mismos tal como se sucede en el lenguaje de programación Ada [37] (El perfil Ravenscar no permite el uso de memoria dinámica).

Cuando el uso de gestión de memoria dinámica es una necesidad impuesta, el único gestor considerado ha sido Binary Buddy [62], debido a su respuesta temporal logarítmica, $O(\log_2(n))$.

I. Puaut en [97] apunta dos causas para explicar esta situación:

- Los gestores de memoria dinámica existentes o bien no ofrecen una respuesta temporal determinista o bien, en caso de ser determinista, no es lo suficientemente rápida.
- El problema de la fragmentación, descrito en la sección 3.3 puede provocar el fallo de la aplicación a pesar de que se disponga de suficiente memoria libre.

2.8. Conclusiones

Un sistema de tiempo real es un sistema en el cual no sólo la salida del sistema determina el éxito o el fracaso del mismo. El momento temporal en el cual se produce la salida también tiene una gran importancia. El sistema debe presentar un tiempo de respuesta a estímulos externos menor que un cierto límite predeterminado para que la respuesta sea correcta. Se dice por lo tanto que este tipo de sistemas tienen plazos temporales.

Debido a la complejidad inherente, estos sistemas se encuentran formados por multitud de subtareas, normalmente, cooperativas donde cada una de ellas presenta una serie de requerimientos temporales específicos para el buen funcionamiento global del sistema.

Los sistemas de tiempo real plantean una problemática compleja que, actualmente, cuenta con una solida teoría.

Capítulo 3
Gestión de memoria dinámica explícita

Este capítulo presenta un estudio del estado del arte de la gestión de memoria explícita, centrándose especialmente en los gestores de memoria más significativos propuestos hasta el presente.

3.1. Introducción

La gestión de *memoria dinámica explícita* aparece a finales de los años cincuenta para reemplazar la gestión de *memoria estática*, la cual había sido empleada hasta entonces. La gestión de memoria estática consiste simplemente en definir todos los *entes* (objetos) que formaban a un programa en tiempo de compilación. Este esquema es muy simple de implementar, sin embargo, presenta ciertas desventajas. Primero, el tamaño de todas las estructuras de datos tiene que ser conocido antes de comenzar la ejecución. Segundo, no es posible construir estructuras de datos dinámicamente dependiendo de la entrada recibida por la aplicación. Un esquema más flexible era necesario. La introducción de la memoria dinámica, la cual permite la asignación de memoria en tiempo de ejecución, resolvió estos inconvenientes al tiempo que planteó nuevos retos.

Actualmente, todos los sistemas operativos modernos [120, 132, 131] proveen facilidades que permiten la implementación del *área de memoria dinámica* o *montículo*[1] , la cual es utilizada por el proceso para alojar la memoria dinámica. El sistema operativo suele ofrecer un soporte mínimo consistente en una llamada al sistema para aumentar el tamaño del montículo del proceso. Por ejemplo, POSIX define la llamada al sistema `brk()`.

Sin embargo, el uso directo de esta primitiva por parte de los procesos no suele ser práctico ya que trabaja a nivel de página física, lo que significa que solo permite asignar/liberar páginas físicas completas al/del montículo. El uso de una página completa por parte del proceso para almacenar una estructura compuesta por tan solo unos pocos bytes se traduce en una gran ineficiencia. En su lugar, los procesos utilizan los llamados *gestores de memoria dinámica*. Estos gestores, normalmente implementados como bibliotecas, son los encargados de reservar memoria para el montículo y posteriormente gestionarla, permitiendo al proceso realizar peticiones de tamaño menor a una página física.

Desde el punto de vista del proceso, el problema de la gestión de memoria está resuelto, ya que ésta se delega a los gestores de memoria dinámica. Sin embargo, los gestores de memoria se enfrentan a un problema de difícil solución, el problema de la *fragmentación*: una sucesión de peticiones de asignación y liberación de memoria provoca una progresiva partición del montículo inicial en bloques de tamaño más pequeño. El gestor puede, después de un cierto tiempo de funcionamiento, no ser capaz de satisfacer una petición de asignación, a pesar incluso de que la suma de los tamaños de los bloques libres disponibles sea mayor que el bloque de tamaño requerido.

[1] En este contexto, el término *montículo* no debe ser confundido con el árbol binario balanceado del mismo nombre.

3.2. Definición del problema

La gestión de memoria dinámica apareció para permitir la asignación y liberación de memoria a las aplicaciones durante su ejecución. Dotando así a las aplicaciones de un esquema de gestión más flexible que el hasta entonces utilizado, la memoria estática. Actualmente existen dos formas de gestionar la memoria dinámica:

- Memoria dinámica explícita o manual: el programador tiene el control directo sobre cuándo se asigna y se libera la memoria. Normalmente esto se realiza mediante llamadas explícitas a las funciones que ofrece el gestor de memoria (por ejemplo, `malloc`/`free` en el lenguaje C).
- Memoria dinámica implícita o automática: el programa solicita memoria conforme la necesita, pero no indica explícitamente qué bloques de memoria han quedado libres. Para localizar los bloques libres, el gestor de memoria debe buscarlos, lo cual se conoce como "recolección de basura" o *garbage colletion*.

Aunque la gestión de memoria explícita es más rápida que la gestión implícita, requiere de una gran disciplina y conocimiento de la aplicación para utilizarla correctamente. Por otra parte, la gestión implícita permite eliminar una de las principales fuentes de fallos de programación: el uso de punteros.

Aunque la gestión de memoria automática resulta atractiva, a menudo consume una gran cantidad de tiempo de CPU y no suele presentar un tiempo de respuesta determinista. Esta tesis se encuentra únicamente centrada en la gestión de memoria dinámica explícita y por lo tanto este estudio del arte también.

Desde el punto de vista algorítmico, la gestión de memoria dinámica es un problema de optimización, que consiste en minimizar la cantidad de memoria necesaria para satisfacer una serie de peticiones de asignación y liberación. Sin embargo, el problema presenta una complejidad adicional a los problemas de optimización clásicos, la secuencia de peticiones no es conocida a priori. Este desconocimiento resulta crítico en la resolución del problema [18].

El *análisis competitivo* es la herramienta existente para abordar este tipo de problemas. El análisis competitivo consiste en plantear el problema como si de un juego de dos jugadores se tratara. Por una parte, está el jugador en línea, el cual se encarga de ejecutar el algoritmo competitivo con la entrada proporcionada por el adversario. El adversario, basándose en el conocimiento que dispone acerca del algoritmo ejecutado por el jugador en línea, construye la peor entrada posible para así maximizar el grado de competitividad.

En análisis competitivo, la calidad de un algoritmo competitivo frente a cada secuencia de entrada viene expresada por el grado de competitividad del mismo. Este grado se expresa normalmente como una constante o función c y se calcula como:

$$ALG(I) \leq c \cdot OPT(I) + \alpha$$

Donde $ALG(I)$ es el resultado del algoritmo competitivo ante una entrada I y $OPT(I)$ es el resultado obtenido por el algoritmo óptimo no competitivo ante la misma entrada I.

Tal como se puede suponer y tal como indican A. Borodin y R. El-Yaniv en [18], el adversario malicioso siempre será capaz de encontrar una entrada I para la cual producirá una $c > 1$ para cualquier algoritmo competitivo *determinista*.

Por tanto, la imposibilidad de encontrar un algoritmo óptimo para la resolución del problema de la gestión de memoria dinámica ha provocado la necesidad de diseñar una gran cantidad de gestores que se aproximen al resultado óptimo $c = 1$, como por ejemplo, Best-Fit, First-Fit, Next-Fit, Worst-Fit, Binary Buddy, etc.

3.3. El problema de la fragmentación

La fragmentación ha sido considerado, sin duda alguna, como el principal problema en la gestión de memoria dinámica.

Tal como señalaron P.R. Wilson et al. en [140], cabe destacar el reducido número de publicaciones teóricas frente a la gran cantidad de trabajos experimentales que abordan este problema.

El problema de la fragmentación se puede definir como:

> "La incapacidad de un gestor de memoria para satisfacer una petición de asignación de r bytes a pesar de que el total de memoria asignado hasta el momento más r sea inferior o igual a $\mathcal{H}$, es decir, al tamaño del montículo."

Tradicionalmente se han distinguido dos tipos de fragmentación: la fragmentación *externa* y la fragmentación *interna*.

La fragmentación externa se presenta cuando un gestor de memoria fracasa ante una petición de memoria de tamaño r a pesar de que la memoria total libre disponible (memoria no asignada en peticiones previas) en ese momento es igual o mayor que r.

Por otra parte, la fragmentación interna aparece cuando, ya sea por requerimientos del gestor o por alguna decisión de diseño del mismo (como puede ser que toda asignación esté alineada), el gestor asigna más memoria

de la que ha sido requerida. A este tipo de fragmentación se le conoce como interna precisamente porque la memoria que está siendo desaprovechada se encuentra en el interior del bloque que ha sido asignado.

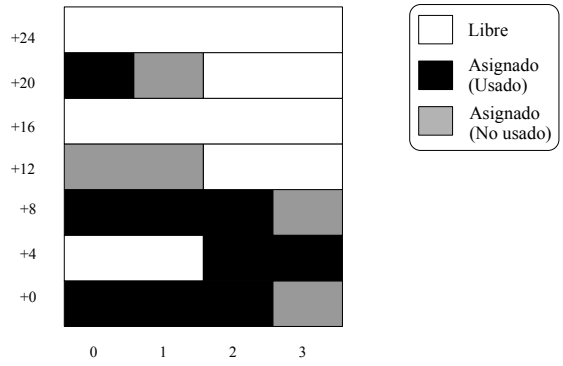


Figura 3.1: Ejemplo de fragmentación interna y externa.

La figura 3.1 muestra un ejemplo de un mapa de memoria con fragmentación interna, representado como una matriz de celdas. Cada celda representa un byte, la unidad mínima asignable. Las celdas contiguas en una misma fila pueden ser agrupadas para crear bloques mayores (varios bytes). Los bloques nunca pueden ser creados a partir de celdas verticalmente contiguas. Cuando se llega al final de una fila, se considera como celda contigua la primera celda de la siguiente fila.

En la figura 3.1 se muestra tres tipos de celdas:

1. Celdas no coloreadas, las cuales representan memoria libre.
2. Celdas coloreadas de negro, que representan bloques asignados a una aplicación. Esta memoria se encuentra en uso por la aplicación ya que fue realmente requerida por ella.
3. Celdas coloreadas de gris, las cuales también representa memoria asignada a la aplicación. Sin embargo, estos bytes no fueron requeridos por la aplicación, por lo cual no están siendo utilizados (fragmentación interna).

   Un ejemplo de creación de celdas negras es cuando la aplicación requiere 2 celdas y el gestor, debido a una cuestión de diseño, asigna 4 celdas. Las dos primeras celdas serían celdas negras mientras que las dos celdas restantes serían grises.

Tal como se ha comentado anteriormente, no se puede hacer referencia a la existencia de fragmentación externa, ya que ésta aún no se ha producido. En el caso de la figura 3.1, el gestor de memoria fracasará debido a la

fragmentación externa ante una petición de tamaño en el rango [7, 14], ya que existen un total de 14 celdas y el bloque libre más grande disponible se compone de 6 celdas libres.

Es importante resaltar que no todos los algoritmos de gestión de memoria tienen por qué padecer a la vez de las dos clases de fragmentación existentes pero sí de alguna de ellas [140].

Históricamente se han seguido dos aproximaciones completamente diferentes para el estudio de la fragmentación. Por una parte se ha estudiado la fragmentación mediante una metodología totalmente experimental [119, 57, 17]. Es decir, se ha estudiado el impacto de las diferentes estrategias de gestión de la memoria según los diferentes programas que las utilizan. Este tipo de estudios han sido claramente los que han llegado a conclusiones más optimistas, la principal de las cuales puede ser resumida como:

> "El problema de la fragmentación es más una falta de entendimiento de la propia fragmentación que un problema en sí mismo [57]."

Por otra parte, se ha realizado el estudio de la fragmentación mediante un análisis formal teórico de la misma [105, 106, 107, 45], en los cuales se concluye que es posible crear escenarios de peor caso de fragmentación que requieran un montículo considerablemente más grande que la cantidad de memoria pedida.

A partir de los estudios tanto teóricos como experimentales se puede llegar a la siguiente conclusión: en teoría es posible construir secuencias de peticiones de asignación y liberación capaces de generar una fragmentación externa extrema, sin embargo, en la práctica, la probabilidad de aparición de los mismos es extremadamente baja.

Para un mejor entendimiento de los estudios realizados hasta ahora, en las siguientes secciones se explica con mayor detalle los resultados obtenidos en ambos tipos de estudios para cada una de las clases de fragmentación.

3.3.1. Fragmentación externa

El problema de la fragmentación externa se manifiesta cuando, ante una petición de tamaño r, el gestor fracasa, a pesar de que el total de memoria libre es superior o igual a r. Esto se debe a que, tras satisfacer una secuencia de operaciones de asignación y liberación, el montículo puede quedar troceado en un conjunto de bloques libres físicamente no contiguos. Por ejemplo, suponiendo un montículo de tamaño $\mathcal{H} = 32$ bytes, ante la siguiente secuencia de peticiones: b_1 =Asignar(14), b_2 =Asignar(4), Liberar(b_1), b_3 =Asignar(22), la última petición de asignación de 22 bytes fallaría, ya

que los 28 bytes libres existentes se encuentran repartidos en dos bloques libres de 14 bytes cada uno.

Este fenómeno solamente puede aparecer si el gestor de memoria puede gestionar bloques de memoria de diferentes tamaños. Por el contrario, si el gestor solo es capaz de asignar bloques de un único tamaño entonces este tipo de fragmentación nunca se produce.

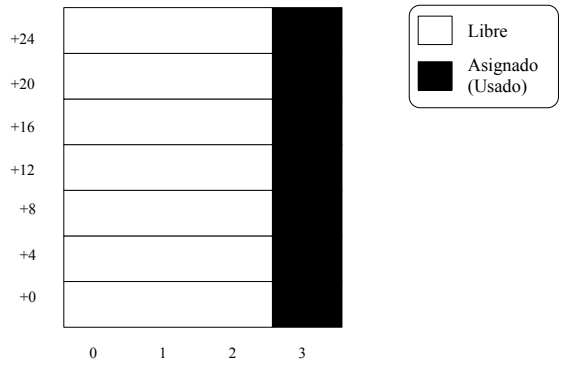


Figura 3.2: Escenario de peor caso para la fragmentación externa.

A continuación se enumeran algunos de los resultados más relevantes en el plano teórico sobre gestión de memoria dinámica y fragmentación. Se ha de tener en cuenta que todos los resultados indicados a continuación dependen básicamente de dos variables: M, la máxima cantidad de memoria viva que la aplicación puede tener asignada; y m que es el tamaño de bloque máximo que la aplicación puede pedir.

- J.M. Robson [105, 106, 107], y M.R. Garey, R.L. Graham y J.D. Ullman [45], basándose en análisis competitivo, demostraron de manera independiente la cota superior asintótica de fragmentación máxima que un gestor de memoria puede producir como $M \cdot m$. Además, ambos trabajos identificaron el escenario de peor caso asociado a dicha cota. La figura 3.2 refleja un ejemplo de dicho escenario, en dicha figura se puede observar un mapa de memoria el cual ha degenerado a una secuencia de bloques libres de tamaño $m - 1$ (en este caso $m = 4$ celdas). Dichos bloques libres no son contiguos, ya que cada bloque libre está separado del resto por un bloque asignado de tamaño 1.
- J.M. Robson determinó en [105, 106] la cota asintótica de fragmentación producida por una hipotética estrategia de gestión de memoria competitiva óptima. Tal estrategia es inviable ya que plantea que el jugador en línea puede decir cual es el mejor movimiento a realizar tras

la jugada de su adversario, seleccionando la mejor opción defensiva a su alcance. El propio Robson en este trabajo tilda de no implementable dicha estrategia. Ante esta estrategia óptima, Robson determinó que la cota superior se encontraría entre $\frac{1}{2} \cdot M \cdot \log_2(m)$ y $0{,}86 \cdot M \cdot \log_2(m)$.

- D.E. Knuth demostró en [62] que la función cota superior para los sistemas Buddy Binarios se podía calcular como $2 \cdot M \cdot \log_2(m)$.
- Varios años después, J.M. Robson calculó en [107] la cota superior de las políticas First-Fit y Best-Fit. En el caso de la política First-Fit, la función cota superior se calcula como $M \cdot (1 + \log_2(m))$. Por otra parte, en el caso de la política Best-Fit, la cota superior es *al menos* $(M - 4 \cdot m + 11) \cdot (m - 2)$, siendo esta cota de orden asintótico $O(M \cdot m)$.
- F. Gavril en [47] y J.M. Robson en [108] demostraron que el problema de gestión de memoria dinámica en su versión no competitiva (el algoritmo tiene un conocimiento absoluto de las peticiones futuras) es un problema NP-Completo. Además, en [46] se puede encontrar un resumen de las técnicas utilizadas por ambos para realizar dicha demostración.
- H.A. Kierstead modeló en [59] la política First-Fit mediante la teoría de grafos y calculó el grado de competitividad del mismo como 80. Posteriormente, Kierstead propone en [60], varias aproximaciones polinómicas del algoritmo óptimo no competitivo y mediante estas calcula un nuevo grado de competitividad de 6.
- J. Gergov calculó en [48] un grado de competitividad de 5 para la política First-Fit.
- M.G. Luby, J. Naor y A. Orda proponen en [74] utilizar un nuevo parámetro en el análisis competitivo del problema de la gestión de memoria dinámica: la duración de cada una de las asignaciones.
- B. Kalyanasundaram y K.R. Pruhs concluyen en [58] que si el gestor de memoria dinámica conoce la duración de las asignaciones, este conocimiento no es de gran ayuda en el peor de los casos. Además, muestran que el grado competitivo de todo algoritmo competitivo aleatorio (no determinista) ante un adversario ignorante es $\Omega\left(\frac{\log(x)}{\log(\log(x))}\right)$, donde tal como expresan en el propio trabajo, x podría ser cualquier parámetro expresado en la literatura existente: relación entre el mayor y menor bloque pedido; relación entre el mayor y el menor *holding time*; el número de distintas duraciones, etc.
- G. Confessore, P. Dell′Olmo y S. Giordani en [30] plantearon un algoritmo próximo al óptimo para resolver el problema de la asignación periódica de memoria, un caso particular de la gestión de memoria donde las peticiones son periódicas y se conoce la duración de la asignación

a priori, utilizando para ellos la teoría de grafos. En esta propuesta se impone fuertes restricciones en el patrón de peticiones que la aplicación puede realizar.

Tal como se puede observar, todos los resultados teóricos existentes son bastante pesimistas. Aplicando estos resultados a un caso numérico, si quisiéramos utilizar un algoritmo First-Fit con una montículo de memoria de $M = 10$ Kbytes y un tamaño de bloque máximo de asignación de $m = 512$ bytes, necesitaríamos, al menos, $10 \cdot 2^{10} \cdot (1 + log_2\,(512)) = 100$ Kbytes para garantizar que el algoritmo nunca fracasará debido a la fragmentación externa. En el caso de la estrategia Best-Fit, los resultados son todavía mucho más pesimistas. Con los datos anteriores, necesitaríamos aproximadamente $10 \cdot 2^{10} \cdot 512 = 5$ Mbytes para garantizar que el gestor nunca falle.

Considerando únicamente estos resultados es difícil considerar la memoria dinámica como una herramienta de programación útil. Sin embargo, contrastan enormemente con los resultados experimentales que se presentan a continuación:

- J.E. Shore comparó en [119], las políticas Best-Fit y First-Fit entre sí, utilizando para ello distribuciones de probabilidad (en concreto una distribución normal, una distribución uniforme y una distribución hiperexponencial), concluyendo que ambas políticas causaban una fragmentación baja, con una diferencia máxima entre ellas nunca superior al 3 %.
- M.S. Johnstone y P.R. Wilson analizaron en [57] la fragmentación producida por un conjunto de gestores de memoria dinámica (First-Fit, Best-Fit, First-Fit AO, Best-Fit AO, DLmalloc, etc) ante una selección de aplicaciones reales (GCC, Espresso, Perl, Ghostscript, etc), concluyendo que el problema de la fragmentación es en realidad un problema de mal diseño de los gestores de memoria y que en el caso de las aplicaciones analizadas, las políticas existentes no sufren casi de fragmentación *real*.
- A. Bohra y E. Gabber realizaron en [17] una nueva comparativa de varias implementaciones de gestores de memoria (DLmalloc, PhK/BSD malloc, Binary Buddy, Quick Fit, etc) con dos aplicaciones concretas: Hummingbird y Emacs. Hummingbird es un sistema de ficheros ligero utilizado como proxy cache de los sistemas web. Su principal interés consiste en que al contrario de las aplicaciones convencionales, estas aplicaciones suelen ejecutarse ininterrumpidamente por un largo periodo de tiempo. Las conclusiones de este trabajo pueden resumirse como que aunque el problema de la fragmentación parece resuelto en el caso de las aplicaciones de corta duración con una demanda de bloques pequeños, este no es el caso para las aplicaciones de larga duración con

demandas más atípicas.

Por lo tanto, se puede concluir que todo gestor de memoria dinámica existente presenta algún caso patológico de fragmentación, el cual, en caso de darse, provoca la máxima fragmentación posible. Sin embargo, tal como se desprende de los estudios experimentales existentes, esto casos rara vez se dan.

3.3.2. Fragmentación interna

La fragmentación interna, al contrario que la fragmentación externa, solamente sucede cuando, debido a la estrategia de asignación de bloques utilizada por el algoritmo de gestión de memoria dinámica, a una petición se le asigna una mayor cantidad de memoria que la que realmente había sido pedida. Como consecuencia, la memoria extra no es utilizada por la aplicación, debido principalmente al desconocimiento por parte de la aplicación de su asignación. La fragmentación interna suele deberse a decisiones de compromiso en el diseño del gestor de memoria. Por ejemplo, se suele asignar más memoria de la requerida para evitar asignar bloques no alineados o para evitar gestionar bloques demasiado pequeños (puede dar lugar a fragmentación externa).

La fragmentación interna también puede deberse a requerimientos de la propia estrategia de asignación utilizada. Por ejemplo en el algoritmo Binary Buddy (véase la sección 3.4.5.5) los tamaños de bloque son redondeados a potencias de dos.

De forma general, los algoritmos cuyas estructuras de datos solamente gestionan un conjunto discreto de tamaños de bloque, como por ejemplo los algoritmos Binary Buddy, TLSF y Half-Fit, no pueden dividir los bloques a tamaños diferentes que los que tienen prefijados, por lo tanto asignan dichos tamaños a pesar de que la petición requiriese un tamaño menor.

Por el contrario, algoritmos como Best-Fit (véase sección 3.4.5.2), First-Fit (véase sección 3.4.5.1), Next-Fit (véase sección 3.4.5.3) y AVL (véase sección 3.4.5.7) no la padecen ya que tienen la capacidad de gestionar cualquier tamaño de bloque libre en su estructura.

Al contrario que en el caso de la fragmentación externa, la fragmentación interna es propia de cada algoritmo y tiene que ser estudiada particularmente para dicho algoritmo e incluso para una implementación concreta del mismo. Es inusual encontrar en la literatura un estudio sobre este tipo de fragmentación.

3.3.3. Métricas de fragmentación

Actualmente, medir la fragmentación producida por un gestor de memoria concreto es uno de los grandes problemas con los que se enfrenta todo aquel que estudia la gestión de memoria dinámica. De hecho, hasta el momento, no existe un acuerdo acerca del mejor método para medir fragmentación.

M.S. Johnstone y P.R. Wilson propusieron en [57] cuatro métricas de fragmentación, las cuales se encuentran actualmente muy aceptadas. Estas métricas se calculan a partir de dos parámetros extraídos durante la ejecución de una aplicación: la cantidad de memoria que utiliza esta para ejecutarse, M_R y la cantidad de memoria que requiere el gestor de memoria para satisfacer todas las peticiones de la aplicación, M_G.

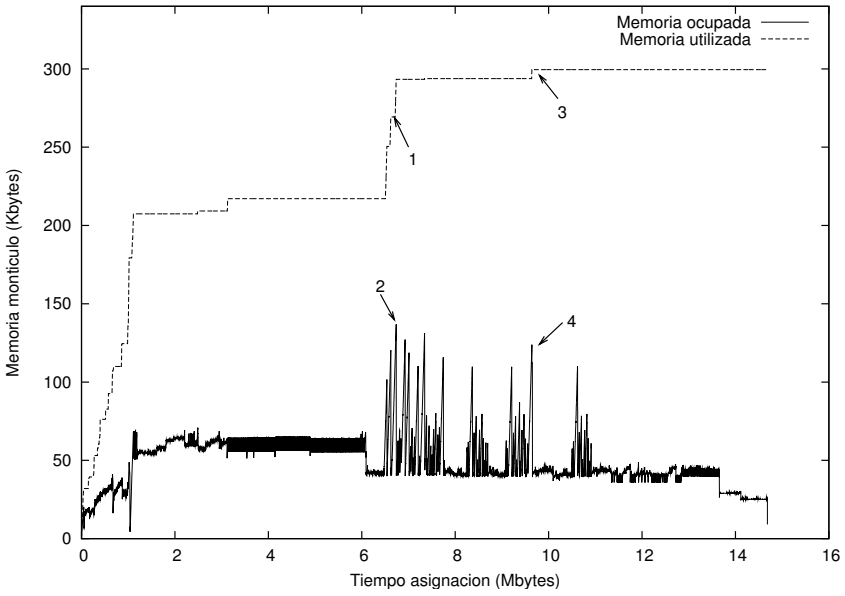


Figura 3.3: Fragmentación de Espresso utilizando Binary Buddy.

A continuación se describen cada una de estas métricas, ilustrándolas con el ejemplo de la gráfica 3.3, la cual muestra una traza de la utilización de la memoria por parte de la aplicación Espresso obtenida al utilizar el gestor de memoria Binary Buddy. Esta gráfica muestra dos fenómenos diferentes, por una parte muestra la cantidad de memoria utilizada por la aplicación (en Kbytes). Por otra parte muestra la cantidad de memoria re-

querida por el gestor de memoria para satisfacer la memoria requerida por la aplicación (también en Kbytes). Las cuatro métricas consideradas para medir la fragmentación producida durante la ejecución de un programa son:

Métrica 1: la cantidad de memoria utilizada por el gestor de memoria relativa a la cantidad de memoria requerida por la aplicación, *promediada en todos los puntos a través del tiempo*, es decir, $Métrica_{\#1} = \overline{M}_G/\overline{M}_R$ relativa a M_R. En la figura 3.3, esto es equivalente a promediar la cantidad de memoria utilizada por el gestor Binary Buddy relativa a la cantidad de memoria requerida por el programa Espresso durante toda la ejecución del programa. En el caso de la aplicación Espresso utilizando el gestor Binary Buddy, usando esta métrica, se obtiene una fragmentación del 445,60 %. El problema de esta métrica es que tiende a esconder los picos de utilización de memoria, siendo estos picos donde la fragmentación se convierte en un verdadero problema.

Métrica 2: la cantidad de memoria usada por el gestor de memoria relativa a la máxima cantidad de memoria requerida por la aplicación *en el punto de máxima utilización de la memoria requerida por la aplicación*, es decir, $Métrica_{\#2} = M_G/M_R@\,\text{máx}(M_R)$ relativa a M_R. En la figura 3.3 esta medida se obtiene como el cociente entre el valor del punto 1 y el valor del punto 2. Para el caso del programa Espresso utilizando el gestor Binary Buddy esta métrica muestra una fragmentación de 146,09 %. El inconveniente de esta métrica es que el punto de mayor utilización de memoria por parte del programa no suele corresponderse con el punto en el que el gestor de memoria acusa mayor fragmentación.

Métrica 3: la máxima cantidad de memoria utilizada por el gestor relativa a la cantidad de memoria requerida por la aplicación *en el punto de máximo uso de memoria por parte del gestor*, es decir, $Métrica_{\#3} = M_G/M_R@\,\text{máx}(M_G)$ relativa a M_R. En la figura 3.3, el punto 3 corresponde a la máxima cantidad de memoria utilizada por el algoritmo Binary Buddy en la aplicación Espresso y el punto 4 corresponde a la memoria requerida por la aplicación Espresso en ese punto. La fragmentación obtenida en ese punto es 141 %. El inconveniente asociado a esta métrica es que tenderá a presentar altas fragmentaciones en programas que realicen una gran petición de memoria cuando solamente un mínimo de la misma esté siendo utilizada.

Métrica 4: la máxima cantidad de memoria usada por el gestor relativa a la máxima cantidad de memoria utilizada por la aplicación, es decir, $Métrica_{\#4} = \text{máx}(M_G)/\,\text{máx}(M_R)$ relativa a M_R. *Estos dos puntos no tienen que ocurrir necesariamente en el mismo momento temporal de ejecución del programa.* Respecto a la figura 3.3, el punto 2 corresponde a la máxima cantidad de memoria utilizada por el gestor de memoria

dinámica y el punto 3 corresponde a la máxima cantidad de memoria utilizada por la aplicación. La fragmentación obtenida utilizando esta métrica es 118 %.

Tal como se puede observar, todas las métricas presentadas en esta sección resultan válidas para medir la fragmentación producida por un gestor de memoria dinámica. De hecho, aunque en el estudio de fragmentación de M.S. Johnstone y P.R. Wilson [57] solamente se hace uso de la tercera y cuarta métrica, se deja claro que las otras dos serán utilizadas en futuros estudios.

3.4. Algoritmos de gestión de memoria

El problema de gestión de memoria dinámica se traduce a un problema de optimización de espacio, donde las peticiones de memoria no son conocidas a priori. Borodin en [18] demostró que debido a las características del problema, no existe una solución óptima, es decir, algoritmo que ante cualquier entrada produzca una fragmentación mínima.

Sin embargo, a pesar de no poder diseñar un algoritmo óptimo, sí que es posible proponer *estrategias* específicas para adaptarse a los requerimientos de un sistema o conjunto de sistemas concretos. Así, por ejemplo, existe una serie de gestores diseñados para ser utilizados por sistemas "generales", sin requerimientos temporales específicos con un comportamiento espacial bastante aceptable. Ejemplos de estos gestores son: First-Fit, Best-Fit, Lazy-Fit [143] y DLmalloc [66].

A su vez, también se han descritos gestores para aplicaciones de alto rendimiento temporal como es el gestor Simple Segregated Storage (SSS) y variaciones del mismo [61, 53], así como estudios sobre la escalabilidad de estos algoritmos en entornos paralelos [56]. Sin embargo, en el caso de este gestor, el consumo espacial es relativamente elevado. Por último cabe indicar que existen también gestores como Hoard [65, 14], específicos para ser utilizados por aplicaciones multiprocesador.

Otro grupo de gestores son los conocidos como gestores *a medida*, diseñados específicamente para ser utilizados por una única aplicación. Básicamente este tipo de gestores son creados para aprovecharse de ciertas características de la aplicación que el resto de gestores no aprovecharían.

Tal como concluye B. Zorn en [147, 52, 15], el uso de gestores construidos a medida puede ser ventajoso, ya que toda aplicación presenta un patrón en el uso de su memoria. Un ejemplo de estas ventajas es que si se detecta que la aplicación no requiere memoria libre mayor de un determinado tamaño, el gestor de memoria puede entonces dimensionarse a dicho tamaño. Sin embargo, el propio Zorn en [16], tras realizar un estudio comparativo de una

serie de gestores de memoria a medida (Apache, GCC, etc) con el gestor de propósito general DLmalloc, concluye que en todas las comparativas el algoritmo DLmalloc resulta claramente superior. Este resultado puede tener la siguiente interpretación: a pesar que los gestores a medida pueden llegar a ser claramente superiores, los existentes actualmente no han sabido interpretar adecuadamente el patrón de la aplicación para la cual han sido diseñados.

La construcción de un gestor de memoria puede parecer extraordinariamente laboriosa, sin embargo, como demuestran P.R. Wilson et al. en [140], todos los algoritmos de gestión de memoria dinámica existentes ofrecen los mismos servicios (a veces con diferentes denominaciones) y pueden ser reducidos a un mismo modelo operacional, lo cual facilita enormemente su estudio y su construcción.

Para una mejor comprensión de los algoritmos de gestión de memoria, la presente sección realiza un estudio funcional exhaustivo de los algoritmos de gestión de memoria existentes: por una parte se presentan los servicios que ofrecen los gestores de memoria. Después se presentan las diferentes *políticas* y *mecanismos* existentes, los cuales son utilizados para implementar las diferentes *estrategias* de diseño de cada uno de los gestores de memoria. Posteriormente se presenta el modelo operacional común, el cual puede ser encontrado en cada uno de los gestores de memoria existente. Para terminar la sección, se presentan un conjunto de gestores de memoria ampliamente utilizados hoy en día.

Diferencia entre estrategia, política y mecanismo. Hasta el momento se han utilizado los términos *estrategia*, *política* y *mecanismo* sin definirlos. A continuación se ofrece dicha definición: la estrategia tiene en cuenta regularidades en el comportamiento de la aplicación y determina un rango de políticas para ser utilizadas en la asignación de bloques libres. La política seleccionada se implementa mediante un mecanismo, el cual es un conjunto de algoritmos y estructuras de datos. En el contexto de memoria dinámica:

- Una estrategia intenta explotar las regularidades en el comportamiento de la aplicación.
- Una política es una decisión implementable.
- Un mecanismo es un conjunto de algoritmos y estructuras de datos que implementan una política.

Por ejemplo, después de analizar el patrón de uso de memoria de una aplicación se llega a la conclusión de que dicha aplicación requiere bloques de tamaño relativamente pequeños, los cuales permanecen un largo tiempo

asignados, y bloques grandes, los cuales, por contra, son liberados en cortos espacios de tiempo.

Una posible estrategia a seguir para aprovechar al máximo el patrón observado es *evitar trocear áreas libres de tamaño grande para asignar bloques pequeños*. Esta estrategia, tal como ha sido definida no resulta implementable. Sin embargo, existe una política de asignación de bloques libres, la política de mejor ajuste que se corresponde con la estrategia definida. En concreto, la política de mejor ajuste especifica justamente que *siempre se ha de utilizar el bloque más pequeño de tamaño suficientemente grande para satisfacer la petición*. El siguiente paso, una vez identificada la política que se corresponde con la estrategia a seguir, es seleccionar el mecanismo (estructura de datos) más adecuado para implementar dicha política. En este caso, mecanismos válidos pueden ser una lista doblemente enlazada o un árbol binario de búsqueda.

3.4.1. Servicios ofrecidos

Históricamente, todos los gestores de memoria han ofrecido dos servicios básicos:

Asignación de memoria: servicio que permite a la aplicación realizar una petición de memoria libre. El servicio requiere que la aplicación especifique el tamaño requerido. La denominación del servicio varía de implementación a implementación aunque las más extendidas son: `malloc` en el lenguaje C [38], `new` en la mayoría de lenguajes orientados a objetos (Ada [37], C++ [36], Java [51, 130]), etc.

Liberación de memoria: servicio recíproco al anterior, el cual permite liberar un bloque de memoria previamente asignado. Normalmente suele requerir que se indique la dirección de comienzo de la memoria a liberar. Existen, sin embargo, versiones de este servicio que también requieren conocer el tamaño de la memoria que fue asignada. Al igual que el servicio anterior, este servicio ha recibido históricamente diferentes denominaciones como: `free` en el lenguaje C [38], `deallocate`, `delete` y `dispose` en la mayoría de lenguajes orientados a objetos (Ada [37], C++ [36], Java [51, 130]), etc.

Adicionalmente, algunas especificaciones como ISO/IEC programming language C [38], además de los servicios descritos arriba (`malloc` y `free`) propone una serie de servicios derivados de estos:

`realloc`**:** esta función permite modificar el tamaño de un bloque previamente asignado.

`calloc`**:** esta función permite a una aplicación pedir un vector de un número determinado de elementos. La función acepta como parámetros el núme-

ro de elementos y el tamaño de cada uno de estos elementos. Además la semántica de esta función especifica que la memoria asignada habrá sido inicializada al valor inicial cero.

3.4.2. Principales políticas

Una vez decidida la estrategia o estrategias a seguir por el gestor de memoria, el siguiente paso en el diseño del mismo es la selección de la política o políticas de asignación y liberación de memoria que permitirán implementar dicha estrategia. Es importante señalar que la política o políticas seleccionadas tienen un enorme impacto en la fragmentación causada por el gestor. Por ejemplo, se ha demostrado empíricamente que la política de mejor ajuste, la cual consiste en asignar el bloque de tamaño más cercano por exceso al tamaño requerido por petición causa menos fragmentación que el uso de la política de primer ajuste, la cual consiste en asignar el primer bloque que satisfaga la petición [119].

Resulta además curioso señalar que todos los gestores de memoria existentes implementan alguna o algunas de las políticas que se enumeran a continuación.

A continuación se listan las principales políticas de asignación de memoria existentes:

- Ajuste exacto (Exact-Fit): se busca un bloque libre de igual tamaño que el bloque pedido. En caso de no encontrase, el gestor toma memoria del final del montículo. Ejemplos de gestores basados en esta política son el gestor Simple Segregated Storage (SSS) [61] y el gestor Adaptative Exact-Fit publicado por R.R. Oldehoeft y S.J. Allan en [93].
- Mejor ajuste (Best-Fit): ante una petición de un tamaño determinado, se asignará el bloque libre que más se aproxime, en exceso, a dicho tamaño. Esta es una de las políticas más estudiadas y que mejor funcionan en la práctica [119, 57]. Normalmente, todos los algoritmos que implementan esta política suelen requerir una búsqueda exhaustiva en sus estructura de datos con el consiguiente coste temporal. Ejemplo de algoritmos que implementan esta política son Best-Fit o el gestor AVL.
- Buen ajuste (Good-Fit): es una variación de la política de mejor ajuste. Esta política asigna un bloque libre de tamaño próximo al requerido, sin embargo es posible que exista algún bloque de tamaño aún más adecuado. Esta política permite el uso de mecanismos que evitan una búsqueda exhaustiva en la estructura de datos. Un algoritmo que utiliza esta política es TLSF [78, 82] y Half-Fit [90].
- Primer ajuste (First-Fit): ante una petición, se asigna el primer bloque encontrado en la estructura de datos del gestor cuyo tamaño satisfa-

ga la petición, es decir, el tamaño del bloque debe ser igual o mayor que el tamaño pedido. Es importante señalar que la búsqueda de dicho bloque siempre comenzará desde la misma posición (considerada la inicial) y que será dependiente del mecanismo utilizado. A pesar de lo que podría parecer, esta política suele presentar unos buenos resultados de fragmentación en la práctica, casi tan buenos como la estrategia Best-Fit [134, 133]. El gestor First-Fit es un ejemplo de implementación de esta política.

- Ajuste siguiente (Next-Fit): variación de la política primer ajuste, donde la búsqueda no comienza siempre al principio de la estructura del gestor sino en la posición donde finalizó la búsqueda anterior. C. Bays realizó un estudio comparativo en [13] de esta política con la política de primer ajuste y la política de mejor ajuste, concluyendo que de las tres políticas, esta es la que peores resultados, obtiene en términos de fragmentación. El algoritmo Next-Fit es un ejemplo de gestor que implementa esta política.
- Peor ajuste (Worst-Fit): esta política se basa en asignar siempre el bloque libre de mayor tamaño disponible. Esta política se utiliza raramente, sin embargo, se ha introducido aquí por ser una de las políticas más básicas existentes [62].

La segunda gran decisión que se debe tomar en el diseño de un gestor de memoria dinámica es la política que se va a utilizar respecto a los bloques de memoria recién liberados. Es decir, cuando un bloque es liberado y existen bloques libres físicamente contiguos al mismo, qué acción va a realizar el gestor de memoria. Las posibles políticas existentes son:

- Fusión inmediata: cuando se libera un bloque que es físicamente contiguo a otro u otros libres, estos son unidos en un único bloque inmediatamente. La ventaja de utilizar esta estrategia consiste en que siempre se consumirá el mismo tiempo realizando la fusión de bloques libres. Los gestores First-Fit, Best-Fit y Next-Fit, por ejemplo, implementan esta política.
- Fusión diferida: esta política consiste en posponer la unión de bloques libres, permitiendo la reutilización de los bloques recientemente liberados. Esta política presenta una ventaja clara: la reutilización de bloque en aplicaciones con una gran reutilización de bloques del mismo tamaño. La principal desventaja presentada por el uso de esta política consiste en la gran cantidad de tiempo que debe de consumir el gestor cuando decide realizar la fusión de bloques libres físicamente contiguos. Este inconveniente puede resultar de poca importancia para la mayoría de los sistemas existentes, sin embargo es un problema importante en el caso de los sistemas de tiempo real.

El gestor DLmalloc [66] es un claro ejemplo de gestor que implementa esta política de fusión de bloques libres.

- No realizar ningún tipo de fusión: esta política consiste en no fusionar nunca áreas de memoria libres físicamente contiguas.

Es importante señalar que un gestor de memoria puede implementar más de una política a la vez. Un ejemplo de gestores que utilizan varias políticas de asignación es el gestor DLmalloc, el cual dependiendo del tamaño de bloque requerido puede utilizar una política de mejor ajuste o de ajuste exacto.

3.4.3. Principales mecanismos

Una vez seleccionada la política o políticas que mejor se adaptan a la estrategia de diseño del gestor a implementar, el siguiente paso es diseñar los mecanismos que las implementen. Debido a la gran cantidad de gestores existentes y a los múltiples mecanismos implementables, en la presente sección se han agrupado estos algoritmos (con una descripción del mecanismo que utilizan) en una serie de categorías [140]:

- **Ajuste secuencial**: esta categoría contiene los algoritmos más básicos. Estos algoritmos suelen estar basados en el uso de una lista doblemente enlazada y en el recorrido de ésta para encontrar un bloque con una cierta condición (que depende de la política que esté utilizando el algoritmo en cuestión). Ejemplos de algoritmos que pertenecen a esta categoría son: First-Fit, Best-Fit, Next-Fit y Worst-Fit.
- **Listas segregadas**: los algoritmos pertenecientes a esta categoría utilizan un vector de listas de bloques libres, donde cada una de estas listas contiene bloques que cumplen una cierta condición. Por ejemplo, tener un cierto tamaño o pertenecer a un cierto rango de tamaños determinado. Los bloques de estas listas se encuentran segregados o agrupados lógicamente, no es necesario que se encuentren físicamente segregados o agrupados. Ejemplos del uso de esta estrategia son: el algoritmo conocido como Fast-Fit [129], el cual utiliza un vector para indexar bloques de tamaño muy pequeño y un árbol binario para gestionar bloques grandes; el algoritmo desarrollado por Doug Lea [66], el algoritmo SSS (Simple Segregated Fit) [61] y sus variaciones [53].
- **Sistemas de colegas (Buddy systems)**: una variante a la estrategia de listas segregadas son los sistemas Buddy [62, 95, 61]. Estos algoritmos utilizan un vector de listas de bloques libres, las cuales son indexadas mediante algún tipo de función matemática ($f(x) = \log_2(x)$ en la variante Binary Buddy [62] o la función de Fibonacci en la variante Fibonacci Buddy [55, 95, 21]). Los sistemas de colegas además

utilizan una estrategia de división de bloques un tanto limitada aunque muy eficiente. Esta estrategia se basa en que cada vez que se parte un bloque, este es partido en dos partes las cuales pueden ser insertadas en alguna lista existente. Respecto a la fusión, los sistemas de colegas fusionan bloques físicamente contiguos en un nuevo bloque más grande sí y solo sí este nuevo bloque había existido previamente. Es decir, dos bloques son fusionados solamente si habían sido previamente colegas. Saber si dos bloques provienen de un mismo bloque inicial puede ser calculado de forma muy eficiente a partir de la posición de memoria que ocupan, tal como se describe en [54]. Existen muchísimos ejemplos de algoritmos que pertenecen a esta categoría: Binary Buddy [62], Fibonacci Buddy [55, 95, 21], Weighted Buddy [118] y Double Buddies [141, 139, 94]. Los sistemas de colegas suelen mostrar unos buenos tiempos de respuesta. Sin embargo, uno de los mayores inconvenientes presentado por esta familia de algoritmos es una gran fragmentación interna, de hasta un 50 % en el peor de los casos [57].

- **Ajuste indexado**: esta categoría contiene aquellos algoritmos que utilizan estructuras complejas para indexar bloques de memoria libre. Estas estructuras suelen presentar características muy interesantes, como por ejemplo la versión del algoritmo Fast-Fit, que utiliza un árbol cartesiano para almacenar e indexar los bloques de memoria libres mediante dos características, el tamaño de los mismos y su dirección física [129].
- **Ajuste mediante mapas de bits**: esta categoría es una variante de la categoría anterior (ajuste indexado). A diferencia del ajuste indexado, en la cual se necesita explorar el contenido de la estructura para saber si existen bloque libres, los algoritmos pertenecientes a esta categoría utilizan mapas de bits para conocer cuáles de los bloques se encuentran libres y cuáles están asignados. Un ejemplo de este tipo de algoritmos es el algoritmo Half-Fit [90]. Esta aproximación presenta la ventaja de almacenar toda la información necesaria para realizar la búsqueda de un bloque libre en un fragmento de memoria relativamente pequeño, típicamente una palabra de 32 bits. Por lo tanto esta estrategia reduce la probabilidad de producir fallos de caché, mejorando la respuesta temporal, tal como se deduce de los trabajos publicados por F. Sebek [111] y S. Basumallick y por K. Nilsen [12].

3.4.4. Modelo operacional

Todo gestor de memoria dinámica puede ser descrito como un tipo abstracto de datos (TAD), el cual es usado para gestionar una o varias áreas de memoria (montículos de memoria). Esta gestión se realiza mediante el almacenamiento de información acerca del estado (libre o usado) de cada uno

de los bloques de memoria que van siendo creados a partir de la progresiva fragmentación del montículo inicial. Por lo tanto, todo gestor de memoria comparte un mismo modelo operacional, en el cual las únicas diferencias son ciertas decisiones de diseño.

De hecho, algunos autores [15, 3] proponen implementar todas las políticas existentes (decisiones de diseño) en una biblioteca de tal forma que es la aplicación la que construye el algoritmo de gestión según sus necesidades y requerimientos.

Para un mejor entendimiento del impacto de la estructura utilizada es interesante analizar cómo los algoritmos de gestión de memoria dinámica gestionan los bloques de memoria libre.

Ante una petición de asignación de memoria:

1. El gestor comprueba la disponibilidad en su estructura de datos de un bloque libre que pueda satisfacer la petición recibida. Esta comprobación depende de la política implementada para la búsqueda de bloques libres. Por ejemplo, si el algoritmo utiliza una política de primer ajuste, el primer bloque libre encontrado que sea de tamaño igual o superior al tamaño requerido será suficiente. Sin embargo, si el gestor de memoria implementa una política de mejor ajuste será imprescindible realizar una búsqueda exhaustiva en la estructura de datos. Ante la existencia de múltiples bloques libres, la elección del más apropiado es realizada por la implementación del algoritmo.
2. En caso de que no se encuentre ningún bloque y si el algoritmo implementa una política de fusión de bloques diferida, se lleva a cabo la fusión de los bloques físicamente contiguos. Si al realizar la fusión, se encuentra un bloque lo suficientemente grande para satisfacer la petición, se sigue al siguiente paso. En caso de no encontrar ningún bloque, el algoritmo fracasa. En sistemas donde existan llamadas al sistema para incrementar el montículo inicial[2] , el gestor hará uso de ellas, en caso contrario el algoritmo finaliza el servicio con fallo.
3. Si el bloque seleccionado es de tamaño superior al requerido, el gestor divide el bloque seleccionado en dos nuevos bloques libres: uno de tamaño igual al requerido o aproximadamente igual, según la política y otro con el tamaño restante. El primer bloque será asignado para satisfacer la petición mientras que el segundo será devuelto a la estructura de datos del gestor.

Ante una petición de liberación de memoria:

[2] El sistema operativo suele proveer una serie de llamadas al sistema que permiten al gestor de memoria incrementar o reducir el montículo inicial, en el caso de los sistemas UNIX®, esta llamada es `brk`.

1. En caso de que el algoritmo implemente una política de fusión inmediata, se realiza la fusión con los dos bloques colindantes: el anterior y el posterior, en caso de encontrarse libres.
2. Se introduce el bloque liberado en la estructura del gestor de memoria.

Para llevar a cabo estas operaciones (asignación y liberación de bloques), cada gestor de memoria suele implementar una serie de operaciones para trabajar con su estructura de bloques libres (inserción, búsqueda, extracción). Estas operaciones de apoyo son descritas a continuación con un mayor nivel de detalle:

- **Inserción de un bloque libre**: esta operación puede ser requerida tanto por la operación de liberación de memoria como por la operación de asignación. En el primer caso, el bloque a insertar es el bloque que ha sido liberado por la aplicación o un bloque mayor resultado de la fusión del bloque liberado por la aplicación con bloques libres, físicamente contiguos. En el caso de la operación de asignación, el bloque insertado es la parte no asignada de un bloque seleccionado para ser asignado pero cuyo tamaño es mayor que el del bloque pedido por la aplicación.
- **Búsqueda de un bloque libre de un tamaño dado o de tamaño mayor**: esta operación se encarga de encontrar un bloque libre de tamaño igual o superior al tamaño requerido por una petición de una aplicación. Esta operación es la encargada de implementar la estrategia que definirá el comportamiento del algoritmo para encontrar un bloque libre (Primer ajuste o First-Fit, Mejor Ajuste o Best-Fit, etc). Normalmente esta elección marcará el rendimiento global del gestor.
- **Búsqueda de un bloque adyacente**: para la fusión de un bloque libre es necesario encontrar los bloques libres físicamente adyacentes a él, si estos existen. Esta operación no suele ser costosa y suele ser utilizada tanto por la operación de asignación como por la operación de liberación de memoria.
- **Extracción de un bloque libre**: esta operación podría ser también necesaria tanto para la operación de liberación como para la operación de asignación de memoria. En el primer caso se utiliza cuando un bloque va a ser fusionado con otro bloque libre adyacente que ya se encontraba insertado en la estructura. En este caso el bloque que se encontraba insertado en la estructura tiene que ser extraído de ella. El segundo caso ocurre cuando una operación de asignación de memoria encuentra un bloque adecuado para una asignación en la estructura de datos de bloques libres.

3.4.5. Gestores representativos

Para terminar la exposición sobre los algoritmos de gestión de memoria dinámica, se presentan a continuación los gestores más utilizados (o más representativos) actualmente.

Se ha evitado deliberadamente la introducción de los gestores conocidos como "a medida" (gestores diseñados específicamente para una aplicación en concreto, como por ejemplo el gestor utilizado por el servidor web Apache [137]). Aunque el diseño de un gestor de memoria adaptado a la aplicación que va a servir pueda parecer a primera vista como la opción idónea [15], no es sencillo conocer con precisión el comportamiento de la aplicaciones, y por tanto, los gestores de memoria a medida suelen ofrecer peores prestaciones que los gestores genéricos. Según [16], en muchas ocasiones los diseñadores de aplicaciones tienen una noción errónea de cómo se comportan sus aplicaciones en realidad.

3.4.5.1. First-Fit

El algoritmo First-Fit [62, 119, 19] es uno de los algoritmos más conocidos y estudiados además de ser uno de los más simples. Tal como indica su nombre este algoritmo implementa una política de primer ajuste, normalmente mediante el uso de una lista doblemente enlazada, donde se almacenan para su uso posterior los bloques libres.

El punto de comienzo de la búsqueda del primer bloque adecuado para satisfacer una petición determinada varía según la implementación, pudiendo seguir los siguientes esquemas:

- Orden físico: los bloques son insertados en la estructura del algoritmo siguiendo una ordenación por direcciones físicas.
- Orden LIFO (Last In First Out): las inserciones en la estructura de datos siempre se producen en la cabeza de la estructura de datos.
- Orden FIFO (First In First Out): las inserciones en la estructura de datos siempre se producen en la cola de la estructura de datos.

La lógica interna de la operación `malloc` es como sigue:

1. Se realiza una búsqueda iterativa (comenzado siempre desde la cabeza de la lista) en la lista doblemente enlazada hasta que se encuentra un bloque de tamaño lo suficientemente grande para satisfacer la petición recibida. Si no se encuentra un bloque lo suficientemente grande, se retorna con fallo. En caso de que se encuentre un bloque adecuado se salta al paso 2.
2. Si el tamaño del bloque encontrado coincide con el tamaño requerido entonces el bloque es asignado y la función retorna con éxito. En caso contrario, se parte el bloque encontrado en dos nuevos bloques. El

tamaño de uno de estos bloques coincidirá con el tamaño requerido, con lo cual será asignado a la petición. Por otra parte, el otro bloque será insertado otra vez en la lista doblemente enlazada.

La lógica de la operación `free` es como sigue:

1. Se comprueba si los bloques físicamente contiguos al bloque recién liberado se encuentran libres. De ser así se extraen de la lista doblemente enlazada y se fusionan con el bloque recién liberado para crear un bloque de tamaño mayor.
2. El bloque recién liberado se inserta o bien en la cabeza de la lista o bien en la cola de la lista (según el mecanismo de asignación de bloques libres (FIFO, LIFO u orden físico)).

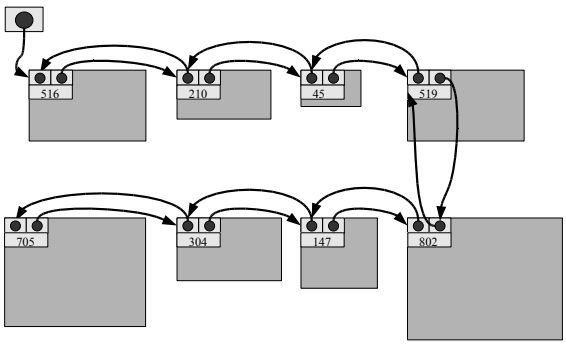


Figura 3.4: Política First-Fit implementada mediante una lista doblemente enlazada.

La figura 3.4 muestra una implementación de un algoritmo First-Fit mediante una lista doblemente enlazada. Esta lista contiene en concreto ocho bloques libres de diferentes tamaños. Los tamaños se indican en la cabecera de cada bloque, que contiene también los punteros que conforman el doble enlace.

3.4.5.2. Best-Fit

El algoritmo Best-Fit consiste en una implementación de la política de mejor ajuste, utilizando para ello una lista doblemente enlazada, al igual que en el algoritmo First-Fit. Sin embargo, esta estructura de datos no es la única opción existente, otros algoritmos como el AVL utilizan estructuras avanzadas (árbol AVL) para implementar la misma política con una respuesta temporal mejor.

Para implementar la política de mejor ajuste, este algoritmo realiza una búsqueda exhaustiva en la estructura de datos, seleccionando el bloque de tamaño más parecido a la petición recibida. En caso de existir varios bloques libres de dicho tamaño, la elección del bloque depende de la implementación del mecanismo. Posibles implementaciones son:

- Orden físico: el bloque asignado es aquel que tenga la dirección física más baja.
- FIFO (First In First Out): Se asigna el bloque más tardíamente insertado en la estructura de datos.
- LIFO (Last In First Out): se asigna el bloque más recientemente insertado en la estructura de datos.

A pesar de que este algoritmo consume, por lo general, más tiempo de cómputo por la búsqueda exhaustiva que los algoritmos First-Fit y Next-Fit, Knuth [62] y Bays [13] demostraron que en la práctica su uso de memoria es menor (es decir, causa un menor grado de fragmentación).

La lógica interna de la operación `malloc` es como sigue:

1. Se realiza una búsqueda iterativa (comenzado siempre desde la cabeza de la lista) en la lista doblemente enlazada, seleccionando el bloque que más se aproxime al tamaño requerido. En caso de encontrarse un bloque de tamaño requerido o de terminar la búsqueda con un bloque adecuado se salta al paso 2. En caso de terminar la búsqueda sin encontrar un bloque adecuado, el algoritmo ha fracasado y se retorna con fallo.
2. Si el tamaño del bloque encontrado coincide con el tamaño requerido entonces el bloque es asignado y la función retorna con éxito. En caso contrario, se parte el bloque encontrado en dos nuevos bloques. El tamaño de uno de estos bloques coincidirá con el tamaño requerido, con lo cual será asignado a la petición. Por otra parte, el otro bloque será insertado otra vez en la lista doblemente enlazada.

La lógica de la operación `free` es como sigue:

1. Se comprueba si los bloques físicamente contiguos al bloque recién liberado se encuentran libres, de ser así se extraen de la lista doblemente enlazada y se fusionan con el bloque recién liberado para crear un bloque de tamaño mayor.
2. El bloque recién liberado se inserta o bien en la cabeza de la lista o bien en la cola de la lista (según el mecanismo de asignación de bloques libres (FIFO, LIFO u orden físico)).

3.4.5.3. Next-Fit

El algoritmo Next-Fit es una variante del algoritmo First-Fit. En este caso se implementa una política de ajuste siguiente, donde la búsqueda del

bloque adecuado para la asignación no comienza siempre en la cabeza de la lista sino en la posición donde terminó la última búsqueda realizada. Por lo demás este algoritmo tiene el mismo comportamiento que el algoritmo First-Fit.

3.4.5.4. Worst-Fit

El algoritmo Worst-Fit es una implementación de la política de peor ajuste mediante una lista doblemente enlazada. Este algoritmo es muy parecido a Best-Fit, con la única diferencia de que asigna el bloque más grande encontrado que satisfaga la petición de memoria. Para ello este algoritmo realiza una búsqueda exhaustiva en la estructura de datos.

D. K. Knuth demostró en [62] que, en la práctica, este es el algoritmo que causa mayor utilización de memoria (mayor grado de fragmentación).

3.4.5.5. Binary Buddy

El gestor de memoria Binary Buddy es una de las múltiples variantes pertenecientes a la categoría de los sistemas de los colegas. Este algoritmo, al igual que el resto de algoritmos pertenecientes a dicha categoría, utiliza un vector de listas de bloques libres, donde cada lista contiene solo bloques libres de un tamaño determinado (es decir, si una lista representa el tamaño 64 bytes, todos los bloques contenidos en esta lista tendrán un tamaño de 64 bytes). La única diferencia de este algoritmo con el resto de su categoría es la función que utiliza para insertar los bloque libres en las diferentes listas. En el caso de este algoritmo se utiliza la función potencias de dos, es decir:

$$i = \lfloor \log_2 (r) \rfloor \tag{3.1}$$

donde r representa el tamaño del bloque que se quiere bien insertar en la estructura de bloques libres o bien buscar en el interior de dicha estructura, e i corresponde a la entrada correspondiente en el vector.

Aparte de esta diferencia, este gestor no difiere en nada más con el resto de algoritmos pertenecientes a su categoría.

La figura 3.5 muestra el vector de listas utilizado por el algoritmo Binary Buddy para gestionar los bloques libres. Como se puede ver en este ejemplo, cada lista o bien está vacía o bien contiene bloques con un tamaño potencia de dos del índice de la entrada del vector correspondiente. Por ejemplo, la entrada 7 contiene dos bloques con un tamaño 128 bytes (2^7).

A continuación se describe la lógica interna de la operación `malloc` de este algoritmo:

1. El tamaño de bloque requerido es redondeado en exceso para que coincida con alguna de las entradas de la estructura Binary Buddy. Por

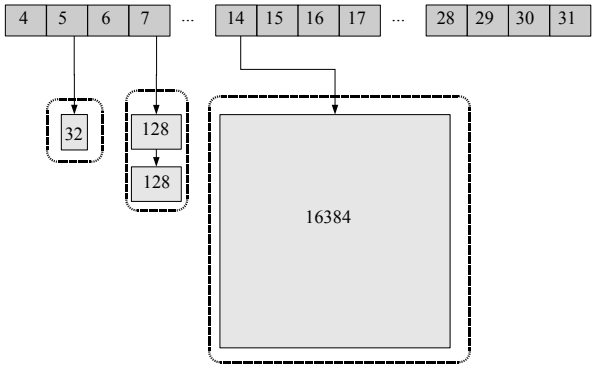


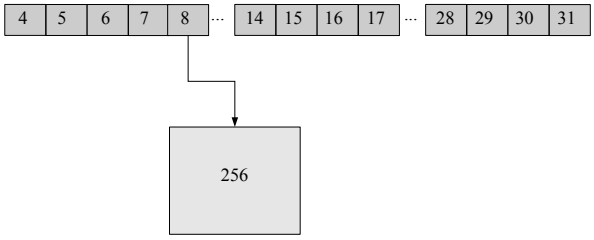
Figura 3.5: Estructura de bloques libres del gestor Binary Buddy.

ejemplo, ante un petición de 110 bytes, la petición será redondeada a la siguiente potencia de dos, 128 bytes.

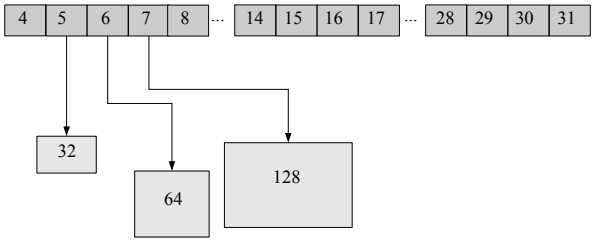
2. Se utiliza la función (3.1) para encontrar la entrada correspondiente al tamaño requerido. En el ejemplo anterior, donde la petición se había redondeado a 128 bytes, la entrada encontrada con dicha función será la 7.
3. Si en dicha entrada existe un bloque libre entonces es asignado, finalizando la operación y retornando con éxito. En caso de que dicha entrada se encuentre vacía, se realiza una búsqueda iterativa incremental por las entradas superiores hasta encontrar una entrada no vacía. Si se encuentra dicha entrada no vacía se salta al paso 4 del algoritmo. En caso de que no se encuentra dicha entrada, el algoritmo fracasa retornando un error.
4. Si se ha encontrado un bloque en alguna entrada superior, este bloque tendrá un tamaño mayor al requerido. El primer paso es extraer dicho bloque de la lista. Una vez extraído dicho bloque se parte, recursivamente hasta obtener el bloque de tamaño potencia de dos más pequeño superior al tamaño solicitado. Los bloques generados en el proceso de particionado son insertados en las listas correspondientes. Dicho bloque será asignado y el algoritmo retornará con éxito. Utilizando el ejemplo anterior, supongamos que encontramos un bloque en la entrada 9, este bloque tendrá por lo tanto un tamaño de 512 bytes. Una vez encontramos decrementamos a la entrada 8, partimos el bloque en dos trozos de 256 bytes. Uno de estos trozos lo indexamos en la entrada 8. Volvemos a realizar la misma operación decrementando a la entrada 7 (bloques de 128 bytes). Como nos encontramos justamente con la entrada perte-

neciente al bloque que se nos pedía, volvemos a partir el bloque de 256 bytes que disponíamos en dos trozos de 128 bytes, uno de estos trozos es asignado a la petición mientras que el otro es insertado otra vez en la estructura.

La figura 3.6 muestra en dos imágenes la asignación de 32 bytes cuando solamente se tiene disponible un bloque de 256 bytes. La imagen superior muestra el estado de la estructura antes de realizar la asignación. La imagen inferior muestra el estado después de realizar la asignación. Como se puede ver en la imagen inferior, el bloque libre de 256 bytes del que se disponía inicialmente ha sido partido en cuatro bloques de tamaños logarítmicamente decrecientes: 128, 64, 32 y 32. Cada uno de estos bloques ha sido insertado en una entrada decreciente. Por ejemplo el bloque de 128 bytes ha sido insertado en la entrada 7. La única excepción es el último bloque de 32 bytes, este bloque ha sido utilizado para satisfacer la petición de 32 bytes.



(a) Estado previo a la asignación de 32 bytes.



(b) Estado posterior a la asignación de 32 bytes.

Figura 3.6: Asignación de un bloque por el algoritmo Binary Buddy.

A continuación se describe la lógica interna de la operación `free`:

1. Se utiliza la función (3.1) para encontrar la entrada correspondiente al tamaño del bloque recién liberado.
2. De manera iterativa, a partir de la entrada devuelta por (3.1) se comprueba que en dicha entrada exista un bloque físicamente contiguo al

recién liberado y que previamente haya formado conjuntamente con el recién liberado un bloque de tamaño doble al correspondiente a la entrada en la que estamos situados. De ser así, los dos bloques libres se fusionan formando uno de tamaño doble al que teníamos. Esta operación se repite iterativamente incrementado la entrada de la estructura donde nos encontramos hasta que en una entrada no exista ningún bloque que cumpla la condición descrita anteriormente (que esté libre y que sea físicamente contiguo al bloque que queremos insertar en la estructura y que previamente ambos formaran un único bloque), en ese momento se inserta el bloque en la lista correspondiente y se retorna.

3.4.5.6. Half-Fit

A continuación se describe el algoritmo de gestión de memoria dinámica Half-Fit [90]. Este algoritmo es considerado el primer gestor de memoria dinámica diseñado explícitamente para ser utilizado en sistemas de tiempo real. Además, es el primer algoritmo conocido que utiliza mapas de bits para la gestión de bloques libres, lo que le permite acelerar la realización de búsquedas en la estructura de bloques libres.

La estructura de datos consiste en un vector de listas, cada una de las cuales contiene bloques libres de tamaños comprendidos entre la potencia de dos del índice que ocupa dicha lista en el vector y la potencia de dos del índice de la siguiente lista, de forma parecida al algoritmo Binary Buddy.

La figura 3.7 muestra un ejemplo de la estructura Half-Fit en funcionamiento donde en la lista correspondiente a la entrada 5 del vector (bloques pertenecientes al rango $\left[2^5, 2^6\right[$) existen 5 bloques de tamaños 43, 55, 33, 62 y 38 bytes.

A diferencia del algoritmo Binary Buddy, este algoritmo siempre fusiona bloques libres físicamente adyacentes.

Para un acceso rápido a las listas contenidas en el vector, el algoritmo Half-Fit utiliza dos funciones diferentes para traducir el tamaño de un bloque a un índice en su estructura de bloques libres.

La primera de estas funciones es utilizada para insertar bloques libres en la estructura Half-Fif. Esta función traduce el tamaño del bloque libre a la entrada del vector que contiene la lista que representa dicho tamaño. Por ejemplo, si queremos insertar un bloque libre con un tamaño de 40 bytes, esta función devolverá la entrada 5, la cual corresponde a bloques de tamaño comprendidos entre $\left[2^5, 2^6\right[$. La función es:

$$i = \lfloor \log_2 (r) \rfloor \tag{3.2}$$

Donde r indica el tamaño del bloque a insertar e i indica el índice correspondiente a la lista que representa dicho tamaño.

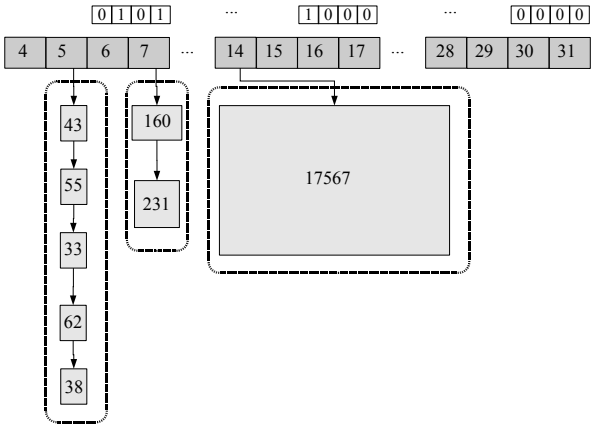


Figura 3.7: Estructura de bloques libres del gestor Half-Fit.

La segunda función es utilizada para encontrar una lista cuyo contenido siempre satisfaga la petición de memoria recibida (esta función no tiene en cuenta si la lista se encuentra vacía). Por ejemplo, si se realiza una petición de 35 bytes, esta función nos devolverá la entrada 6. Debido a que todos los bloques contenidos en dicha lista tendrán siempre un tamaño superior a 35, cualquier lista a partir de la entrada 6 será válida para satisfacer la petición de 35 bytes. La función es:

$$i = \begin{cases} 0 & \text{si } r = 1 \\ \lfloor \log_2 (r-1) + 1 \rfloor & \text{en cualquier otro caso} \end{cases} \tag{3.3}$$

A continuación se describe la lógica interna de la operación `malloc` de este algoritmo:

1. Mediante el uso de la función 3.3 se traduce el tamaño requerido al índice de la primera lista correspondiente a dicho tamaño.
   Si la lista está vacía, es necesario buscar alguna lista superior (de mayor índice en el vector) que no lo esté. Para realizar esta búsqueda se utiliza el mapa de bits junto con operaciones de búsqueda de bits disponibles en la mayoría de los procesadores. En caso de que se encuentre una lista no vacía, se extrae el primer bloque libre de dicha lista y se salta al paso 2. En caso de que no se encuentre ninguna lista que contenga algún bloque libre, el algoritmo retorna sin éxito.
2. Si el bloque encontrado es mayor que el pedido entonces se parte en dos nuevos bloques, uno de ellos con el tamaño que ha sido requerido para

satisfacer la petición, y la otra parte será insertada en la lista indexada por el índice obtenido por la función 3.2.

A continuación se describe la lógica interna de la operación `free`:

1. La primera operación a realizar es la de fusionar el bloque recién liberado con los bloques libres físicamente contiguos a él.
2. Mediante el uso de la función 3.2 se calcula el índice adecuado y se inserta el bloque en la cabeza de dicha lista.

Aunque no se ha especificado explícitamente en ninguna de las operaciones anteriores, en el caso de inserción y extracción de bloques libres en las listas, siempre se realizará la siguiente comprobación: en el caso de que la lista quede vacía después de una extracción, o en el caso de que la lista se encontrara vacía y se le inserte un nuevo bloque, el mapa de bits será modificado para reflejar dicha situación.

Este algoritmo fue uno de los primeros intentos conocidos por construir un gestor de memoria con un comportamiento determinista. Sin embargo, la fragmentación producida por este algoritmo (50 % en el peor de los casos) lo hace poco recomendable para ser utilizado en sistemas de tiempo real.

3.4.5.7. Árboles balanceados AVL

Los árboles balanceados AVL son una variante de los árboles binarios de búsqueda, desarrollados por Adelson-Velskii y Landis. Se puede obtener una descripción de los árboles AVL en [112].

Un árbol AVL es un árbol binario de búsqueda al cual se le añade una condición de equilibrio. Esta condición consiste en que la diferencia de altura de los subárboles izquierdo y derecho de cualquier nodo tiene que ser a lo sumo en 1.

Básicamente un árbol AVL permite:

- **Inserción**: la inserción de un elemento en un árbol AVL es idéntica a la de un árbol binario de búsqueda con la salvedad de la comprobación de la propiedad de equilibrio del árbol. En caso de que se produzca un desequilibrio, se realiza el balanceo del árbol para mantener la condición de equilibrio impuesta por la estructura.
- **Extracción**: la operación de extracción se realiza igual que en el caso de los árboles binarios de búsqueda salvo que al igual que en la operación de inserción, se realiza una comprobación de la propiedad de equilibrio del árbol y en caso de que no se cumpla, se realiza un balanceo del mismo.
- **Búsqueda**: como árbol de búsqueda binaria, el árbol AVL cumple la siguiente propiedad: a partir de un nodo, todos los nodos pertenecientes al subárbol izquierdo almacenarán valores menores al valor alma-

cenado por dicho nodo, mientras que los nodos del subárbol derecho almacenarán valores mayores. Así pues cualquier búsqueda realizada en un árbol AVL mostrará una respuesta temporal asintótica acotada por $\Theta(\log_2(n))$.

Existen cuatro mecanismos de reequilibrado (balanceo) de este tipo de árboles. El esquema de rotación simple izquierda, el simple derecha, el doble izquierda y el doble derecha, los cuales pueden ser encontrados en [112].

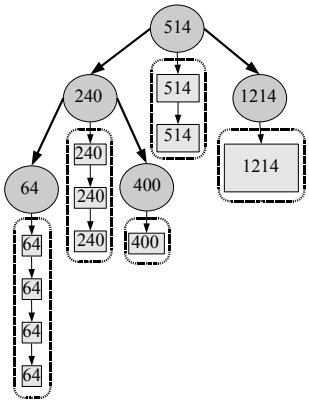


Figura 3.8: Árbol AVL utilizado por el gestor AVL.

Como se ha visto anteriormente, el uso de los árboles AVL es interesante debido a que cualquier búsqueda en esta estructura presenta un tiempo de respuesta asintóticamente logarítmico.

Tal como se ve en la figura 3.8, el árbol AVL es utilizado por este algoritmo para contener bloques de diferentes tamaños, cada hoja del árbol representa un tamaño dado (el cual es utilizado como clave[3] de la hoja) y contiene bloques únicamente de dicho tamaño.

A continuación se describe la lógica interna de la operación `malloc`:

1. Realizar una búsqueda en el árbol AVL hasta que se encuentre un nodo que contenga bloques de tamaño igual o superior al buscado.
2. Si se encuentra el nodo, se extrae la cabeza de la lista. Si la lista se queda vacía, el nodo se extrae del árbol AVL. En caso de que el bloque extraído sea mayor que el tamaño pedido, el bloque se trocea en dos bloques nuevos, uno de los bloques será asignado a la petición, el otro será reinsertado en el árbol: o bien a una lista ya existente o bien se creará un nodo nuevo en el árbol para indexar dicho bloque.

[3] La clave de una hoja es utilizada en los árboles de búsqueda para decidir la posición ocupada por dicha hoja en el árbol.

A continuación se describe la lógica interna de la operación `free`:

1. Se intenta fusionar el bloque recién liberado con los bloque físicamente contiguos a él.
2. Se inserta el nodo recién liberado en el árbol AVL.

Tal como se ha visto, el algoritmo de gestión de memoria AVL es un algoritmo más avanzado que el algoritmo Best-Fit, debido a que utiliza una estructura más avanzada y con mejores propiedades de búsqueda que la lista doblemente enlazada. Sin embargo, la estrategia implementada es la misma que en aquel algoritmo, por lo cual el uso de este algoritmo mejora notablemente el tiempo de respuesta de peor caso, $O(n)$ en el caso del gestor Best-Fit contra $O(\log_2(n))$ en el caso del gestor AVL.

3.4.5.8. El gestor de Doug Lea (DLmalloc)

Uno de los gestores de memoria dinámica más usados actualmente es el desarrollado e implementado por Doug Lea [66], también conocido como *dlmalloc*.

Actualmente este gestor de memoria, o algunas de sus variantes, como por ejemplo ptmalloc [50], es utilizado por las bibliotecas C y C++ de GNU [40, 41],

A continuación se describe el funcionamiento interno de este algoritmo tal como se describe en [66]. Es importante resaltar que en versiones posteriores del algoritmo se han variado ciertos detalles del mismo. Por ejemplo, la versión 2.7.2 utiliza mapas de bits para acelerar las búsquedas de bloques libres en vez de la búsqueda iterativa de la versión anterior. Sin embargo, la estrategia principal se ha mantenido inalterada.

El algoritmo dlmalloc utiliza una estructura híbrida para almacenar los bloques de memoria libres. Por una parte utiliza un vector, en el cual indexa los bloques libres de tamaño menor a una constante prefijada. Por otra parte utiliza una lista doblemente enlazada para bloques de tamaño mayor.

Las primeras entradas del vector de bloques pequeños únicamente contiene bloques de un tamaño fijo, permitiendo una asignación inmediata de dichos bloques. El resto de entradas contiene rangos de tamaños, ordenados por tamaño dentro de cada lista. En la terminología del algoritmo, a los bloques indexados en la estructura se les conoce como "*Bins*".

La figura 3.9 muestra la estructura utilizada por el algoritmo dlmalloc. La estructura indexa de manera exacta los bloques libres menores de 512 bytes mientras que los bloques de hasta 128 Kbytes son agrupados en rangos.

El algoritmo dlmalloc utiliza diferentes estrategias de asignación según el tamaño demandado, en concreto:

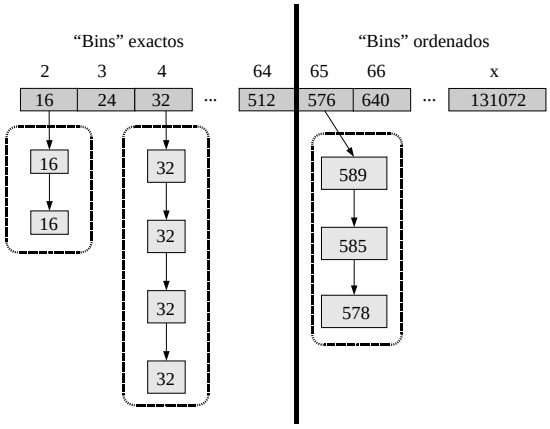


Figura 3.9: Estructura de bloques libres del gestor Doug Lea's malloc.

- Para peticiones de **tamaños pequeños ($\leq$ 64 bytes)**, el algoritmo se comporta como asignador cache, utilizando una política FIFO de asignación.
- Para peticiones de **tamaños intermedios ($>$ 64 bytes y $<$ 512 bytes)**, el algoritmo es configurable para tener un comportamiento FIFO o bien Best-Fit.
- Para peticiones de **tamaños grandes ($\geq$ 512 bytes)**, se utiliza una estrategia puramente Best-Fit.
- Para peticiones de **tamaños muy grandes ($\geq$ 128 Kbytes)**, el algoritmo confía en las funciones de gestión de memoria del sistema operativo subyacente, por ejemplo la función `brk` en el caso de los sistemas UNIX®.

A continuación se describe la lógica interna de la operación `malloc` considerando que gestiona "Bins" exactos de hasta 64 bits:

1. Incrementa el tamaño del bloque requerido para que así el bloque que se suministre pueda contener una cabecera con la información necesaria para ser gestionado.
2. En este paso el algoritmo inicia una búsqueda por la estructura para encontrar un bloque libre de tamaño adecuado según el tamaño pedido:
   a) Para **peticiones pequeñas ($\leq$ 64 bytes por defecto, puede ser configurado hasta un máximo de 512 bytes)**: se traduce el tamaño requerido a un índice y en caso de que la entrada corres-

pondiente de la estructura dlmalloc no esté vacía, se asigna a la petición el primer bloque de dicha entrada.
En caso de que dicha entrada esté vacía, se salta al paso 3.

b) Para **peticiones de tamaño grande ($\geq$ 64 bytes por defecto)**: se traduce el tamaño requerido a un índice, y se utiliza para realizar una búsqueda en la lista indexada por la entrada correspondiente, la estrategia utilizada en la búsqueda es Best-Fit.
En caso de que dicha entrada se encuentre vacía, se salta al paso 3.

c) Para **peticiones de tamaño muy grande ($\geq$ 128 Kbytes)**: se utilizan las llamadas al sistema proporcionadas (algunos ejemplos de este tipo de funciones son `brk`, `sbrk`, `mmap`, `munmap`) por el sistema operativo para asignar dicha memoria.

3. La ejecución de este paso se produce porque la lista que indexa los bloques libres de tamaño pedido se encuentra vacía. Por lo cual se procede a compactar la memoria libre existente (unir bloques libres físicamente contiguos) y a realizar una búsqueda de la primera lista no vacía que pueda satisfacer la petición. En caso de que esta búsqueda fracase, pasar al paso 4.
4. El algoritmo se ha quedado sin suficiente memoria libre para asignar a la petición, por lo cual, utiliza las funciones del sistema operativo subyacente para incrementar el tamaño del montículo y así poder satisfacer la petición.

A continuación se describe la lógica interna de la operación `free`:

1. Si el tamaño del bloque liberado es menor de 128 Kbytes, este se inserta dentro de la lista que indexa dicho tamaño. Es importante señalar que en este paso no se realiza fusión de bloques libres físicamente contiguos.
2. Si por el contrario, el bloque liberado es mayor de 128 Kbytes, este bloque es liberado al sistema operativo.

Tal como se acaba de ver, así como la operación `malloc` es una función laboriosa, la operación `free` es una operación muy rápida ya que con la excepción de que el bloque recién liberado sea físicamente contiguo al *tope*, el bloque recién liberado no será fusionado y será insertado en la entrada apropiada de forma ordenada.

3.5. Gestión de memoria dinámica y sistemas de tiempo real

La gestión de memoria dinámica en sistemas de tiempo real es un área casi inexplorada. Dos parecen haber sido las causas. Por una parte, la creencia generalizada de que los gestores de memoria o bien no presentan una

respuesta determinista o si la presentan, esta es muy alta. El segundo motivo que ha frenado el uso de memoria dinámica es el problema de la fragmentación.

El primer problema ha sido abordado de diferentes formas. Por un lado se han presentado gestores deterministas con un bajo tiempo de respuesta, tanto implementados en software como por ejemplo, el gestor Half-Fit [90], como implementados en hardware como el algoritmo propuesto por V. H. Lai en [64] o el algoritmo propuesto por M.A. Shalan en [117]. También existen propuestas híbridas como el gestor de memoria implícita propuesto por K.D. Nilsen y W.J. Schmidt en [89, 87, 88], que se implementa en software pero con soporte hardware externo.

Por otra parte, se han implementado algunos de los gestores ya existentes en hardware, disminuyendo así los tiempos de respuesta. Ejemplos de esta segunda aproximación son las diferentes versiones del gestor Binary Buddy implementadas por [61, 98, 24, 124, 25].

Además de los trabajos anteriores, existen otros que se centran más en si la gestión de memoria en sí misma puede ser admisible en sistemas de tiempo real.

R. Ford publicó en [39] un análisis del problema de la inversión de prioridad cuando un número determinado de tareas realizan peticiones de asignación y liberación de memoria sobre una única área de memoria dinámica. Este trabajo, centrado en el control de concurrencia, concluye con un algoritmo libre de interbloqueos, basado en accesos concurrentes, seguidos por una fase de validación y recuperación.

W.J. Schmidt y K.D. Nilsen realizaron en [109] un estudio de rendimiento del gestor de memoria dinámica implícita con soporte hardware externo propuesto por ellos mismos en [89].

I. Puaut [97, 96] presentó un análisis de prestaciones de un conjunto de gestores de memoria dinámica y analizó su posible utilización en sistemas de tiempo real. Para el análisis experimental utilizó los siguientes programas: mpg123, CFRAC y Espresso. Este artículo concluye que los algoritmos de gestión analizados no son adecuados para los sistemas de tiempo real debido a su respuesta temporal indeterminista. Sin embargo, es significativo que en este estudio no se tuviera en cuenta el algoritmo Half-Fit [90], publicado 7 años antes.

3.6. Conclusiones

La memoria dinámica es una herramienta muy útil en la programación de aplicaciones, permitiendo un uso más eficiente del recurso memoria. Así, la aplicación puede utilizar solamente la memoria que necesita, liberándola

cuando deje de ser útil. De hecho, la mayoría de los sistemas operativos actuales proporcionan soporte de memoria dinámica a los programas.

Sin embargo, la gestión de esta memoria introduce un grave problema conocido como el problema de la fragmentación, para el cual, debido a sus características intrínsecas, encontrar un algoritmo óptimo que lo resuelva es teóricamente inalcanzable. Sin embargo, es posible proponer estrategias de gestión de la memoria aceptables según las necesidades de la aplicación o rango de aplicaciones que las vayan a emplear.

Actualmente, en la literatura se pueden encontrar una gran cantidad de algoritmos de gestión de memoria dinámica, cada uno de ellos diseñado para satisfacer un ámbito de uso, siendo objetivo común en todos ellos reducir la fragmentación provocada así como proporcionar un tiempo de respuesta bajo.

En el caso de los sistemas de tiempo real, esto no ha sido precisamente así. De hecho, hasta la fecha, debido a sus características, los gestores existentes han sido considerados no adecuados para este tipo de sistemas. Sin embargo, la publicación de gestores como el Half-Fit [90], pueden forzar a replantear dicha creencia.

Capítulo 4
Two-Level Segregated Fit (TLSF)

Este capítulo introduce un nuevo gestor de memoria dinámico, Two-Level Segregated Fit (TLSF), específicamente diseñado para sistemas de tiempo real.

4.1. Introducción

Actualmente, la memoria dinámica es una parte fundamental de la mayoría de los sistemas informáticos, la cual puede ser utilizada o bien de forma explícita, mediante llamadas a bibliotecas, o bien de forma implícita, enmascarada por el lenguaje de programación utilizado. Lenguajes de programación como Ada [37], C++ [36] y Java [51], por poner unos pocos ejemplos, hacen uso de la misma en el propio soporte de ejecución.

En el presente capítulo se introduce un nuevo gestor de memoria dinámica, Two-Level Segregated Fit (TLSF) [78, 82, 80], el cual ha sido diseñado específicamente para ser utilizado en sistemas de tiempo real. Sus dos características de diseño más relevantes son: un tiempo de respuesta rápido y totalmente predecible y una baja fragmentación.

Para lograr estos objetivos, TLSF implementa mediante listas segregadas y mapas de bits una política de buen ajuste (Good-Fit). Esta política produce unos resultados parecidos a la política de mejor ajuste (Best-Fit), la cual se sabe que produce los mejores resultados de fragmentación en la práctica [57].

4.2. Criterios de diseño

Las prestaciones que requieren las aplicaciones de tiempo real difieren bastante de las mostradas, por ejemplo, por las aplicaciones de alto rendimiento. Mientras que las primeras siempre imponen determinismo en la respuesta temporal, las segundas exigen un tiempo medio de respuesta bajo.

Por ello, a continuación se enumeran las restricciones y los requisitos presentes en los sistemas de tiempo real así como una serie de guías utilizadas en el diseño del gestor TLSF para un cumplimiento de las mismas:

- Con frecuencia, estas aplicaciones se ejecutan en entornos confiables, donde los programadores que tienen acceso al sistema no son maliciosos. Esto significa que ellos no intentarán intencionalmente robar o corromper datos de la aplicación que está siendo ejecutada. La protección se realiza a nivel de interfaz de usuario final no a nivel de programación.
- La cantidad de memoria física disponible suele ser limitada.
- En la mayoría de ocasiones, no se dispone del hardware necesario para soportar memoria virtual (MMU) o si se dispone del mismo no suele ser aconsejable debido a los requisitos temporales existentes.

Por todo ello, en el diseño del gestor TLSF se han seguido los siguientes criterios:

Fusión inmediata: cuando se libera un bloque de memoria, si los bloques físicamente contiguos también se encuentran libres, el gestor tiene varias opciones. El gestor puede unirlos inmediatamente (fusión inmediata), generando un bloque mayor, o bien puede no realizar dicha unión, posponiéndola (fusión diferida). DLmalloc [66] es un claro ejemplo de gestor que utiliza la fusión diferida. Aplazar la fusión de bloques libres es una estrategia útil en sistemas cuyas aplicaciones utilizan repetidamente bloques de un mismo tamaño. En estos casos, se elimina la sobrecarga temporal que supone unir y dividir el mismo bloque varias veces.
Aunque este aplazamiento suele mejorar el rendimiento temporal medio del gestor, el uso de esta estrategia hace más difícil predecir su comportamiento temporal ya que no se conoce a priori cuándo se realizará la fusión ni tampoco cuántos bloques serán fusionados. Por lo tanto, todo gestor diseñado para sistemas de tiempo real debería hacer uso únicamente de la fusión inmediata. TLSF utiliza por lo tanto una estrategia de fusión inmediata de bloques libres.

Umbral de división: el tamaño del bloque más pequeño de memoria asignable son 32 bytes. Esta decisión ha sido tomada debido a que la gran mayoría de aplicaciones, y las de tiempo real no son una excepción, no suelen demandar memoria para almacenar datos simples, como pueden ser enteros, punteros o números en coma flotante. En lugar de ello, normalmente, la memoria requerida suele utilizarse para almacenar estructuras de datos complejas que contienen al menos un puntero y un conjunto de datos.
Además, esta limitación viene impuesta porque así el algoritmo puede utilizar esta memoria en el caso de los bloques libres para almacenar los punteros necesarios para gestionar las listas de bloques libres. Esta aproximación optimiza el uso de la memoria, ya que se disminuye el tamaño necesario por parte del bloque para almacenar su información de estado.

Estrategia Good-Fit: TLSF *intentará* asignar el bloque de memoria más pequeño que sea lo suficientemente grande para contener el bloque requerido. Tal como se muestra en [140, 57], la mayoría de aplicaciones existentes solamente requieren bloques de memoria comprendidos en un pequeño rango de tamaños. En este tipo de aplicaciones, el uso de una política Best-Fit, respecto a las demás políticas existentes, suele producir la menor fragmentación posible. La estrategia Best-Fit (o casi Best-Fit, también conocida como estrategia *Good-Fit*) puede ser implementada de una manera predecible y muy eficiente mediante el uso de listas segregadas.
TLSF implementa una estrategia Good-Fit, esto es, se utiliza un gran

conjunto de listas libres, donde cada lista contiene una lista de bloques libres **no ordenados** cuyos tamaños se encuentran dentro del rango del tamaño representado por dicha lista.

No se realiza reubicación: se asume que la memoria libre inicial (montículo) es un único gran bloque de memoria libre, y que no se encuentra disponible la función `brk()` ya que los sistemas operativos de propósito general (como por ejemplo UNIX®) proveen memoria virtual, la gran mayoría de algoritmos de gestión de memoria dinámica existentes han sido diseñados para tomar ventaja de esta característica. Mediante el uso de esta llamada al sistema, es posible diseñar un algoritmo de gestión de memoria dinámica optimizado para gestionar bloques de memoria relativamente pequeños, delegando la gestión de bloques mayores al sistema operativo subyacente mediante el aumento dinámico del montículo.

TLSF ha sido diseñado para proveer un soporte completo para gestión de memoria dinámica, esto significa que puede ser utilizado tanto por aplicaciones, como por el sistema operativo sin necesidad de utilizar ningún hardware de soporte de memoria virtual.

La misma estrategia para todos los tamaños de bloque: se utiliza la misma estrategia para cualquier tamaño de bloque pedido. Uno de los gestores de memoria dinámica más eficientes y también más ampliamente utilizados, el gestor DLmalloc [66], utiliza cuatro estrategias diferentes según el tamaño de bloque pedido: First-Fit, bloques cache, listas segregadas y las facilidades suministradas por el sistema operativo subyacente para aumentar el tamaño del montículo. Esta clase de algoritmos no provee un comportamiento uniforme, sino que depende de la estrategia utilizada en un momento dado, por lo que su tiempo de ejecución en el peor de los casos es normalmente alto y difícil de determinar.

La memoria no se inicializa: ni el montículo inicial ni los bloques liberados son inicializados a cero. Los algoritmos de gestión de memoria dinámica utilizados en entornos multiusuarios tienen que inicializar el contenido de la memoria (normalmente rellenándola con ceros) para evitar problemas de seguridad. No inicializar la memoria antes de asignarla al usuario puede ser considerado un grave problema de seguridad, ya que cualquier programa malicioso podría obtener información confidencial.

Sin embargo, se asume que el algoritmo TLSF será utilizado en un entorno confiable, donde las aplicaciones son escritas por programadores bien intencionados. Por lo tanto, inicializar la memoria es una característica innecesaria que introduce una considerable sobrecarga.

4.3. Detalles de implementación

El algoritmo TLSF utiliza el mecanismo de ajuste segregado para implementar la política de buen ajuste (Good-Fit). Esto se debe a que el gestor no realiza una búsqueda exhaustiva en su estructura de datos para encontrar el bloque de tamaño más cercano al tamaño requerido. En lugar de ello, se asigna un bloque de tamaño próximo al tamaño requerido. Para ello, TLSF utiliza una estructura bidimensional, una matriz de tamaño $\mathcal{I} \times 2^{\mathcal{J}}$. Cada entrada $(i,\ j)$ de esta matriz representa una lista segregada que contiene bloques con tamaños comprendidos en el rango $[2^i{+}2^{(i-\mathcal{J})}{\cdot}j,\ 2^i{+}2^{(i-\mathcal{J})}{\cdot}(j{+}1)[$.

El número total de filas ($\mathcal{I}$) indica el tamaño máximo de bloque ($2^{\mathcal{I}+1}{-}1$) gestionable. Por otra parte, el número total de columnas ($2^{\mathcal{J}}$) establece el número de listas segregadas del algoritmo TLSF y por tanto, los tamaños de los bloques que el algoritmo TLSF podrá manejar.

Por ejemplo, una estructura TLSF configurada con $2^{\mathcal{J}} = 1$ tendrá un comportamiento parecido al gestor Binary-Buddy, obviando las reglas de asignación y liberación de bloques de dicho gestor, porque ambos algoritmos solamente serían capaces de gestionar tamaños potencias de dos.

Para acelerar el proceso de búsqueda de bloques libres, se ha asociado un mapa de bits de tamaño $2^{\mathcal{J}}$ con cada una de las entradas de segundo nivel (un total de $\mathcal{I}$). El bit j del mapa de bits i indica si la lista $(i,\ j)$ contiene algún bloque o no. Este vector de mapas de bits se denomina `mapa_bits_j[]`.

Además, se ha asociado un mapa de bits de tamaño $\mathcal{I}$, mapa_bits_i, con el vector de mapas de bits anterior. El bit i de este mapa de bits indica si mapa_bits_j[i] tiene algún bit a 1 o no.

Respecto a la información necesaria para gestionar cada uno de los bloques de memoria tanto libres como ocupados (tamaño bloque, estado, etc) es almacenada en el interior de los propios bloques, en una estructura conocida como cabecera.

La figura 4.1 muestra la estructura de datos utilizada por el TLSF para $\mathcal{J} = 3$. En este ejemplo en particular, existen bloques libres en las listas segregadas $(i,\ j) = (5,\ 1)$, $(6,\ 5)$ y $(16,\ 2)$ tal como viene reflejado en los mapas de bits (mapa_bits_i y mapa_bits_j [i]).

4.3.1. Parámetros de configuración

La estructura utilizada por el algoritmo TLSF se caracteriza básicamente por los tres parámetros siguientes:

- **Límite del índice de primer nivel** ($\mathcal{I}$): este parámetro indica el número de filas que tendrá la estructura TLSF. Este parámetro define por tanto el tamaño máximo de bloque que podrá ser almacenado

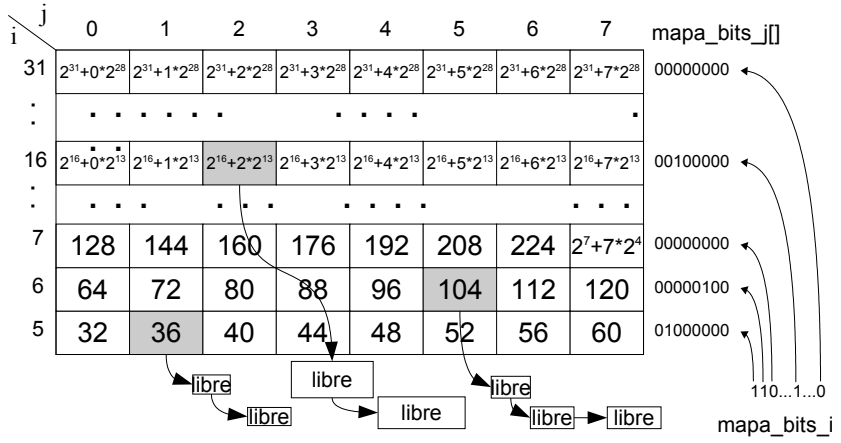


Figura 4.1: Estructura del algoritmo TLSF.

en la estructura. Por ejemplo, si prefijamos $\mathcal{I} = 20$, TLSF solamente podrá gestionar bloques de tamaño hasta $2^{21} - 1$ bytes, es decir, un Mbyte. Por razones de eficiencia, este parámetro nunca debería exceder el tamaño de una palabra de procesador, permitiendo así que con un mapa de bits del tamaño de una palabra se conozca qué clases contienen bloques.

- **Límite del índice de segundo nivel** ($\mathcal{J}$): este parámetro indica el número de columnas en la estructura TLSF, indicando, por tanto, el número de listas segregadas que existen. La elección del valor de este parámetro resulta crítica. Por una parte, un valor demasiado alto provocará una estructura TLSF de varios Kbytes. Por otra parte, un valor demasiado bajo aumentará la cantidad de fragmentación interna producida. Por razones de eficiencia, la potencia de dos de este valor no debería exceder el tamaño de una palabra de procesador. Esto se debe al uso de mapas de bits para conocer las listas no vacías. Valores razonables para este parámetro en una arquitectura de 32 bits son 4 o bien 5.
- **Tamaño de bloque mínimo** (TBM): este parámetro define el tamaño mínimo del bloque. En la primera versión de TLSF, este valor estaba fijado a 32 bytes, sin embargo, con la revisión de la estructura de TLSF, este valor ha sido modificado a 4 bytes.

A partir de los parámetros definidos anteriormente ($\mathcal{I}$, $\mathcal{J}$ y TBM) se pueden definir los siguientes parámetros derivados:

- El número de total de listas existentes: $2^{\mathcal{J}} \cdot \mathcal{I}$.
- El rango de tamaño de bloques aceptado por cada lista según sus índices (i, j):

$$rango\_bloques(i, j) = \left[2^i + j \cdot \frac{2^i}{2^{\mathcal{J}}},\ 2^i + (j+1) \cdot \frac{2^i}{2^{\mathcal{J}}}\right[$$

- El tamaño total de la estructura de datos de TLSF:

$$t\_estructura_{\text{TLSF}} = TFH + PS \cdot 2^{\mathcal{J}} \cdot \mathcal{I}$$

Donde TFH indica el tamaño de la estructura necesaria para gestionar las listas segregadas (40 bytes) y PS de los punteros (4 bytes).
Suponiendo un montículo inicial máximo de 4 Gbytes ($\mathcal{I} = 32$) y un $\mathcal{J} = 5$, el tamaño de la estructura de datos requerida es 3624 bytes.
Si, en lugar de 4 Gbytes, suponemos un montículo inicial de 8 Mbytes ($\mathcal{I} = 23$) y un $\mathcal{J} = 4$, el tamaño de la estructura de datos requerida es 408 bytes.

4.3.2. Funciones de traducción

La inserción de los bloques en la estructura del gestor TLSF se realiza mediante el uso de una función de traducción, $finsertar$, la cual convierte el tamaño r del bloque en el índice de una lista (i, j). Esta función se define como:

$$finsertar(r) = \begin{cases} i = \lfloor \log_2(r) \rfloor \\ j = \lfloor \left(r - 2^i\right) / \left(2^{i-\mathcal{J}}\right) \rfloor \end{cases}$$

Por ejemplo, suponiendo $\mathcal{J} = 5$, un bloque de 265 bytes será insertado en la lista (8, 1).

Además, para evitar realizar búsquedas exhaustivas, TLSF dispone de la función $fbuscar$, la cual, dado una petición de tamaño r, obtiene la primera lista cuyos bloques satisfarán dicha petición. Esta función se define como:

$$fbuscar(r) = \begin{cases} i = \lfloor \log_2(r + 2^{\lfloor \log_2(r) \rfloor - \mathcal{L}} - 1) \rfloor \\ j = \lfloor \left(r + 2^{\lfloor \log_2(r) \rfloor - \mathcal{L}} - 1 - 2^i\right) / \left(2^{i-\mathcal{J}}\right) \rfloor \end{cases}$$

Por ejemplo, suponiendo $\mathcal{J} = 5$, ante una petición de 265 bytes, la primera lista apta para extraer un bloque será (8, 2).

Tal como se puede apreciar, la primera función, $finsertar$ indica la lista que representa el bloque a insertar. La segunda función $fbuscar$ indica la lista siguiente, consiguiendo la propiedad deseada de que todos los bloques contenidos en dicha lista sean mayores o iguales al requerido.

4.3.3. Gestión de bloques

Para gestionar cada bloque de memoria de forma eficiente, TLSF necesita conocer cierta información acerca de los mismos. Entre otras cosas, por cada bloque, TLSF necesita saber el tamaño, el estado actual (libre u ocupado) y varios punteros necesarios para enlazarlo en la estructura TLSF. Esta información se codifica al principio de cada bloque en una área llamada cabecera de bloque.

La cantidad de información necesaria difiere si el bloque está libre u ocupado.

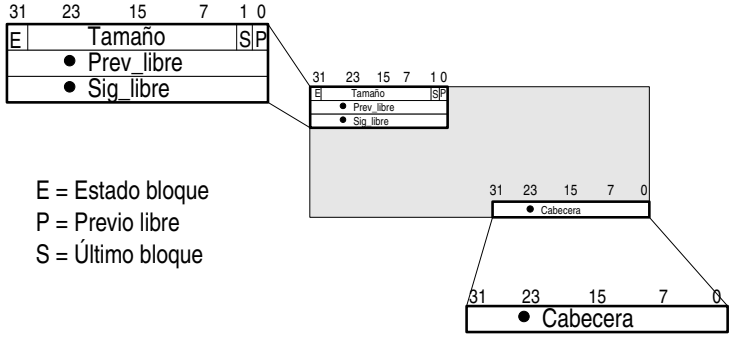


Figura 4.2: Cabecera y cola de un bloque libre.

La figura 4.2 muestra la cabecera utilizada para los bloques libres. Esta cabecera está formada por los siguientes campos:

- **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 éste.

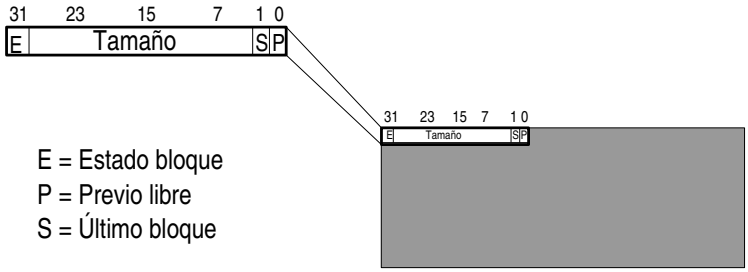


Figura 4.3: Cabecera de un bloque asignado.

La figura 4.3 muestra la cabecera de un bloque asignado. Tal como se puede apreciar, esta cabecera es muy reducida y se compone de un único campo, el campo *tamaño*. Tal como pasaba en el caso de los bloques libres, el bit 31 (E) de este campo se utiliza para codificar el estado del bloque (libre u ocupado). El bit 1 (S) permite conocer si este bloque está situado al final del montículo. Y el bit 0 (P) codifica si el bloque previo está libre u ocupado. El resto de bits codifican el tamaño del bloque.

4.3.4. Pseudocódigo

La mejor forma de estudiar un algoritmo consiste en estudiar su pseudocódigo. En esta sección se incluye detalladamente el pseudocódigo de las dos funciones principales del gestor TLSF: `malloc()` y `free()`. Y sus funciones auxiliares: `mapping_search()`, `mapping_insert()`, `fls()`, etc.

Para facilitar el estudio del pseudocódigo del gestor TLSF, las diferentes funciones han sido clasificadas en cinco categorías: funciones aritméticas, mapas de bits, listas doblemente enlazadas, estructura TLSF y funciones principales.

Como se verá más adelante, el gestor TLSF ha sido implementado sin ningún bucle. Es decir, no se utiliza la iteración ni la recursividad en el código.

4.3.4.1. Funciones aritméticas

Esta categoría engloba los cálculos matemáticos que el gestor TLSF necesita realizar. Por ejemplo, $\log_2(x)$, $finsertar$ y $fbuscar$. Para garantizar el determinismo temporal, todas estas funciones han sido implementadas

utilizando operaciones aritméticas simples, la suma, la resta y desplazamientos de bits. Las funciones incluidas en esta categoría son: `fls()`, `ffs()`, `insert_func` y `search_func`.

`fls()`

La función `fls()` (find last set) recibe como entrada una palabra y devuelve la posición del bit a 1 más significativo. Por ejemplo, ante la entrada 356_{10} (101100100_2) esta función retorna 8. Esta función calcula la función $\lfloor \log_2(x) \rfloor$. Algunos procesadores implementan directamente esta función en ensamblador, por ejemplo, la arquitectura x86 de Intel® dispone de la instrucción `bsr`. En caso de no estar disponible, el algoritmo tiene un coste logarítmico con el tamaño de palabra. Para un tamaño de palabra de 32 bits, la implementación sería:

```
 1  function fls(word: in integer) return integer is
 2      position: integer := 32;
 3  begin
 4      if word = 0 then
 5          return -1;
 6      end if
 7      if (word and FFFF0000#16#) = 0 then
 8          word:= word left_shift 16;
 9          position := position - 16;
10      end if
11      if (word and FF000000#16#) = 0 then
12          word:= word left_shift 8;
13          position := position - 8;
14      end if
15      if (word and F0000000#16#) = 0 then
16          word:= word left_shift 4;
17          position := position - 4;
18      end if
19      if (word and C0000000#16#) = 0 then
20          word:= word left_shift 2;
21          position := position - 2;
22      end if
23      if (word and 80000000#16#) = 0 then
24          word:= word left_shift 1;
25          position := position - 1;
26      end if
27      return position;
28  end fls;
```

`ffs()`

La función `ffs()` (find first set) recibe como entrada una palabra y devuelve la posición del bit a 1 menos significativo. Por ejemplo, ante la entrada 356_{10} (101100100_2) esta función retorna 2. Al igual que en la función `fls`, algunos procesadores implementan en su juego de instrucciones esta función, en el caso de la arquitectura x86 de Intel® la instrucción en ensamblador es `bsf`. En caso de que se tenga que implementar, el algoritmo tiene un coste logarítmico. A continuación se implementa la versión para máquinas de 32 bits.

```
 1  function ffs(word: in integer) return integer is
 2      position: integer := 0;
 3  begin
 4      if word = 0 then
 5          return −1;
 6      end if
 7      if (word and FFFF#16#) = 0 then
 8          word:= word right_shift 16;
 9          position:= position + 16;
10      end if
11      if (word and FF#16#) = 0 then
12          word:= word right_shift 8;
13          position:= position + 8;
14      end if
15      if (word and F#16#) = 0 then
16          word:= word right_shift 4;
17          position:= position + 4;
18      end if
19      if (word and 3#16#) = 0 then
20          word:= word right_shift 2;
21          position:= position + 2;
22      end if
23      if (word and 1#16#) = 0 then
24          word:= word right_shift 1;
25          position:= position + 1;
26      end if
27      return position;
28   end ffs;
```

`insert_func()`

La función `insert_func()` implementa la función de traducción $finsertar$ (ver sección 4.3.2), la cual obtiene, a través de un tamaño de bloque r, el

índice (i, j) de la lista donde el bloque debe de ser insertado.

```
1  procedure insert_func(r: in integer; i, j: out integer) is
2  begin
3        i:= fls(r);
4        j:= (r right_shift (i − L)) − 2^L;
5  end insert_func;
```

En la línea 4, $2^{\mathcal{L}}$ corresponde a una constante, no es necesario calcularla en la función.

Tal como se puede apreciar, el calculo de esta función se puede realizar de forma eficiente debido a que las operaciones que se realizan son, básicamente, desplazamientos de bits y operaciones aritméticas básicas.

`search_func()`

La función `search_func()` implementa la función de traducción $fbuscar$, la cual obtiene, a través de un tamaño r de bloque, el índice (i, j) de la primera lista que puede satisfacer dicho tamaño.

```
1  procedure search_func(r: in out integer; i, j: out integer) is
2  begin
3        r:= r + (1 left_shift (fls(r) − L) − 1;
4        i:= fls(r);
5        j:= (r right_shift (i − L)) − 2^L;
6  end search_func;
```

En la línea 5, $2^{\mathcal{L}}$ corresponde a una constante, no es necesario calcularla en la función. **Nótese que el tamaño de bloque r es redondeado al tamaño representado por la lista superior más cercana**.

Los cálculos realizados por esta función, al igual que pasaba con la función *insert_func()*, se realizan de forma muy eficiente debido a la simplicidad de las operaciones que realiza.

4.3.4.2. Funciones de mapas de bits

Para acelerar la búsqueda de bloques libres en la estructura, TLSF asocia a cada fila de la estructura (un total de $\mathcal{I}$) un mapa de bits con el mismo tamaño que la fila ($\mathcal{J}$). Todos estos mapas de bits son agrupados en un único vector llamado J_bitmaps[]. El bit j del mapa i representa el estado (vacía o no vacía) de la lista (i, j).

Además, TLSF asocia un único mapa de bits (I_bitmap) de tamaño $\mathcal{I}$ con cada mapa de bits contenido en J_bitmaps[], donde el bit i indica si en el J_bitmaps[i] existe o no alguna lista no vacía.

El pseudocódigo perteneciente a las funciones relacionadas con mapas de bits no ha sido incluido por considerarse trivial.

Para trabajar con los mapas de bits, TLSF utiliza las siguientes funciones:

`set_bit()`

Esta función pone a 1 el bit indicado en el mapa de bits indicado.

`clear_bit()`

Esta función pone a 0 el bit indicado en el mapa de bits indicado.

`is_bit_set()`

Esta función retorna el valor de un bit determinado.

4.3.4.3. Funciones de lista doblemente enlazada

Cada lista segregada contenida en la estructura TLSF se implementa por medio de una lista doblemente enlazada no ordenada. El puntero que indica la cabeza de la lista se encuentra en la propia estructura TLSF. Cada bloque insertado en la lista mantiene en su cabecera dos punteros: Prev_libre y Sig_libre.

El pseudocódigo perteneciente a las funciones relacionadas con las listas doblemente enlazadas no ha sido detallado por considerarse trivial. Para el manejo de listas doblemente enlazadas se dispone de las siguientes primitivas:

`head_list()`

Esta función devuelve el bloque que se encuentra en la cabeza de la lista pero no lo extrae.

`insert_list()`

Esta función inserta un bloque en la cabeza de la lista.

`remove_list()`

Dado un bloque, esta función lo extrae de la lista doblemente enlazada.

4.3.4.4. Funciones de la estructura TLSF

Estas funciones, de más alto nivel que las anteriores, permiten gestionar la estructura TLSF, extrayendo e insertando bloques. Además se

ocupan de mantener actualizados los mapas de bits. Estas funciones son: `find_block()`, `insert_block()`, `remove_block()`, `merge_prev()`, `merge_next()` y `split()`.

`find_block()`

La función `find_block()` encuentra el primer bloque libre contenido en la lista indicada por los índices (i, j) o superior. Para ello utiliza los mapas de bits de la estructura TLSF.

```
 1  function find_block(i, j: in integer) return address is
 2  begin
 3       bitmap_tmp:= J_bitmaps[i] and
 4                    (FFFFFFFF#16# left_shift j);
 5       if  bitmap_tmp ≠ 0 then
 6            non_empty_j:= ffs(bitmap_tmp[i]);
 7            non_empty_i:= i;
 8       else
 9            bitmap_tmp:= I_bitmap and
10                         (FFFFFFFF#16# left_shift (i+1));
11            non_empty_i:= ffs(bitmap_tmp);
12            non_empty_j:= ffs(J_bitmaps[non_empty_i]);
13       end if
14       return head_list(non_empty_i, non empty_j);
15  end find_block;
```

Debido a que los mapas de bits son de tamaño constante, buscar en un mapa de tamaño 32 cuesta a lo máximo de 5 pasos, con lo cual esta función se ejecuta con un tiempo de respuesta constante.

`insert_block()`

La función `insert_block()` inserta un bloque de memoria libre en la estructura TLSF, para ello recibe el índice (i, j) de la lista correspondiente. Además, esta función también actualiza el estado de los mapas de bits.

```
1  procedure insert_block(block: in address; i, j: in integer) is
2  begin
3       insert_list  (TLSF_struct[i][j], block);
4       set_bit (J_bitmaps[i], j);
5       set_bit (I_bitmap, i);
6  end insert_block;
```

`remove_block()`

La función `remove_block()` extrae un bloque de memoria libre de la estructura TLSF. Además, esta función actualiza el estado de los mapas de bits.

```
1  procedure remove_block(block: in address; i, j: in integer) is
2  begin
3      remove_list (TLSF_struct[i][j], block);
4      clear_bit (J_bitmaps[i], j);
5      if J_bitmaps[i] = 0 then
6          clear_bit (I_bitmap, i);
7      end if
8  end remove_block;
```

`merge_prev()`

Esta función funde el bloque que recibe como argumento con el bloque físicamente previo a éste, en el caso en que dicho bloque se encuentre libre.

```
1  function merge_prev(block: in address) return address is
2  begin
3      prev_block:= get_prev_block(block);
4      if is_free (prev_block) then
5          remove_block(prev_block);
6          block:= prev_block + block ;
7      end if
8      return block;
9  end merge_prev;
```

La función `is_free()` (línea 4) indica si el bloque se encuentra libre u ocupado a través de la información extraída de la cabecera del mismo. La función `get_prev_block()` (línea 3) devuelve un puntero al bloque físicamente anterior a través de la información extraída de la cabecera del mismo. Dado un bloque b, el puntero al bloque físicamente previo p se consigue comprobando en la cabecera de b si p está libre u ocupado. En caso de que p este libre, el puntero inmediato a la cabecera de b (puntero *cabecera*, ver imagen 4.2) será válido y apuntará a la cabecera de p.

`merge_next()`

Esta función funde el bloque que recibe como argumento con el bloque físicamente siguiente a éste, en el caso en que dicho bloque se encuentre libre.

```
1  function merge_next(block: in address) return address is
2  begin
```

```
3       next_block:= get_next_block(block);
4       if is_free (next_block) then
5           remove_block(next_block);
6           block:= block + next_block;
7       end if
8       return block;
9   end merge_next;
```

En la línea 4, la función `is_free` indica si el bloque se encuentra libre u ocupado a través de la información extraída de la cabecera del mismo. La función `get_next_block()` (línea 3) devuelve un puntero al bloque físicamente siguiente, esto requiere solamente aritmética de punteros, ya que, consiste en sumar el tamaño del bloque actual al puntero que apunta a dicho bloque. Dado un bloque b, el bloque físicamente siguiente se calcula sumando al puntero b el tamaño de b.

`split()`

La función `split()` acepta como primer argumento un bloque de memoria b de tamaño r_b y como segundo argumento, un tamaño r. Esta función divide b en dos nuevos bloques, $b1$ y $b2$. El primero, $b1$, con un tamaño r y el segundo, $b2$, con un tamaño $r_b - r$.

No se especifica el código de esta función por considerarse trivial, ya que consiste solamente en desplazamientos de punteros.

4.3.4.5. Funciones principales

Como funciones principales se entienden las funciones exportadas por el gestor, es decir, `malloc()` y `free()`.

`malloc()`

La función `malloc()` se encarga de asignar memoria. Recibe como argumento un tamaño en bytes y devuelve, si la operación ha tenido éxito, un bloque de dicho tamaño o mayor. En caso de que el gestor no disponga de ningún bloque libre de dicho tamaño o mayor, esta función retorna un error.

```
1   function malloc(r: in integer) return address is
2   begin
3       search_func(r, i, j);
4       free_block := find_block (r, i, j);
5       if not(free_block) then return error;
6       remove_block(free_block);
```

```
 7        if size (free_block ) > split_size_threshold   then
 8              remaining_block:= split(free_block , r);
 9              insert_func (size (remaining_block), i , j );
10              insert_block (remaining_block, i , j );
11        end if
12        return free_block;
13   end malloc;
```

La constante split_size_threshold (línea 7) indica el tamaño de bloque mínimo (TBM), fijado a 4 en la última versión del gestor TLSF. La función size (línea9) retorna el tamaño del bloque de memoria que se le pasa como parámetro.

`free()`

La función `free()` se encarga de liberar un bloque de memoria previamente asignado. Insertándolo otra vez en la estructura TLSF.

```
1   procedure free(block: in address) is
2   begin
3         merged_block:= merge_prev(block);
4         merged_block:= merge_next(merged_block);
5         insert_func (size (merged_block), i, j );
6         insert_block (merged_block, i, j );
7   end free;
```

4.3.5. Optimizaciones

Desde que fue inicialmente diseñado hasta la redacción de la presente tesis, la estructura de datos utilizada por el algoritmo TLSF ha sufrido ligeras modificaciones para adaptarse mejor a las aplicaciones reales. Sin embargo, dichos cambios no han afectado al funcionamiento global del algoritmo ni a sus propiedades: respuesta temporal rápida y predecible, y baja fragmentación.

Esta visión de evolución y cambio de dicha estructura no debería entenderse como un error de planteamiento del algoritmo en sí mismo, sino como una corrección de una de las hipótesis de diseño del mismo y posterior adaptación del algoritmo. Concretamente, la siguiente hipótesis de diseño:

> "La gran mayoría de aplicaciones, y las aplicaciones de tiempo real no son una excepción, no suelen demandar memoria para almacenar datos simple, como pueden ser enteros, punteros o número

> en coma flotante. En lugar de ello, normalmente, la memoria requerida suele utilizarse para almacenar estructuras de datos complejas que contienen al menos un puntero y un conjunto de datos."

Resultó ser falsa, ya que todas las aplicaciones reales estudiadas (CFRAC, Espresso, GAWK, GS y Perl, ver apéndice B.1 para una descripción de las mismas) utilizan con mucha frecuencia, y sin excepción, bloques de tamaño menor a 32 bytes. Por lo tanto, fue necesario introducir pequeñas modificaciones en la estructura TLSF para una mejor gestión de bloques pequeños, así como contravenir el umbral de división al tamaño de palabra del procesador.

Revisión de la estructura TLSF Para mejorar la fragmentación causada por la versión inicial del gestor TLSF se ha introducido un vector adicional de listas segregadas. Este nuevo vector contiene listas de bloques libres con los siguientes tamaños: [12, 16[, [16, 20[, [20, 24[, [24, 28[y [28, 32[.

Las políticas de inserción y búsqueda de bloques permanecen prácticamente inalteradas. Ante una inserción de un bloque pequeño, el gestor utiliza la siguiente función de traducción $finsertar(r)$:

$$finsertar(r) = \{i = \lfloor r/4 \rfloor - 3$$

Esta función devuelve como resultado el índice i correspondiente a la lista donde el bloque debe ser insertado.

Si por el contrario, la acción a realizar es una operación de búsqueda de bloque libre, se utiliza la siguiente función $fbuscar$:

$$fbuscar(r) = \{i = \lceil r/4 \rceil - 3$$

La cual devuelve el índice i correspondiente a la primera lista cuyos bloques pueden satisfacer dicha petición. El siguiente paso consiste en buscar la primera lista a partir del índice i no vacía.

Para acelerar las búsquedas en este vector adicional se ha añadido un nuevo mapa de bits de tamaño 7. El bit i de este mapa de bits indica si la lista i del vector adicional contiene algún bloque o no.

La figura 4.4 muestra las modificaciones sufridas por la estructura TLSF, es decir, la adición de un conjunto de listas para gestionar bloques de tamaño menor a 32 bytes. Además se ha introducido un nuevo mapa de bits para acelerar la gestión de estas listas, el resto de la estructura permanece inalterada.

Como se puede apreciar, la revisión no plantea ningún coste temporal adicional a la versión del algoritmo previamente planteada, además la fragmentación producida al gestionar bloques de tamaño menor de 32 bytes es reducida, ya que ahora dichos bloques no se redondean.

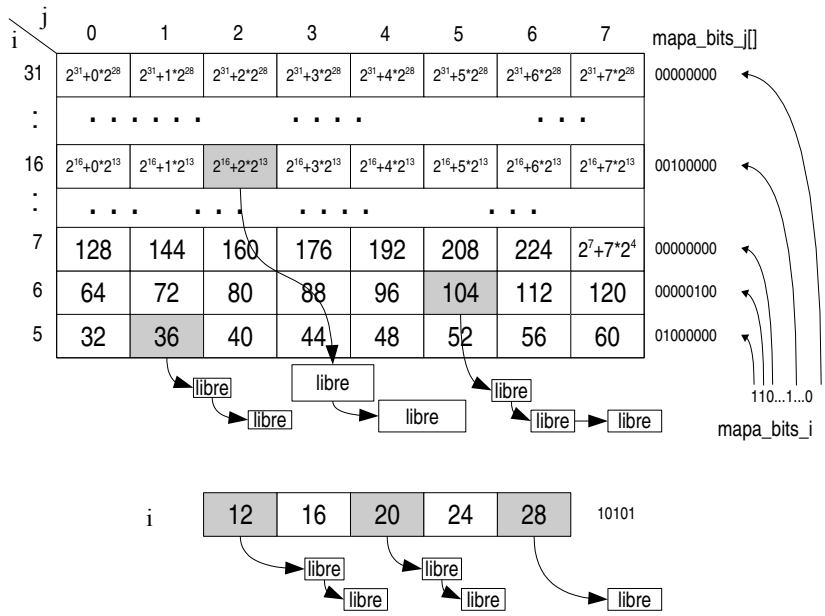


Figura 4.4: Estructura revisada del algoritmo TLSF.

4.4. Conclusiones

TLSF es un nuevo algoritmo de gestión de memoria dinámica que proporciona una respuesta temporal determinista y rápida. Esta característica se debe a que TLSF no realiza búsquedas exhaustivas para encontrar el bloque más adecuado en su estructura de datos. En su lugar, utiliza un conjunto de listas segregadas, cada una de las cuales indexa un rango de tamaños de bloques. El acceso a dichas listas se realiza en tiempo constante debido a que TLSF utiliza una función de traducción bidireccional para seleccionar una lista dado un tamaño de bloque. Esta función se basa en el uso de desplazamientos de bits con lo cual es muy eficiente.

Prueba del tiempo de respuesta constante del gestor TLSF es su pseudocódigo, el cual ha sido diseñado sin un solo bucle.

Todas estas características convierten a TLSF en un algoritmo específicamente apto para los sistemas de tiempo real, cuyo principal requisito es la obtención de una respuesta temporal determinista.

Capítulo 5
Comportamiento temporal del gestor TLSF

Este capítulo presenta un estudio temporal, asintótico y experimental, comparativo del algoritmo TLSF con un conjunto representativo de gestores de memoria dinámica: First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL malloc y Half-Fit.

5.1. Introducción

La mayor parte de los trabajos sobre memoria dinámica se han centrado en los aspectos relativos a fragmentación. Se ha primado la eficiencia espacial frente a la eficiencia temporal. Cualquier gestor de memoria con un comportamiento temporal de orden lineal, $O(n)$, o inferior suele ser considerado aceptable en la mayoría de sistemas que utilizan memoria dinámica.

En el capítulo anterior se presentaba un nuevo algoritmo de gestión de memoria dinámica, TLSF, diseñado para ser utilizado en sistemas de tiempo real. En el presente capítulo se realiza un estudio de su comportamiento temporal.

Además, se ha realizado un estudio comparativo temporal con un conjunto representativo de gestores: First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL malloc y Half-Fit. Estudiando la idoneidad de estos gestores frente al gestor TLSF.

5.2. Estudio asintótico

El estudio temporal asintótico de un algoritmo indica una cota superior que el algoritmo nunca superará. El coste temporal asintótico de una función se calcula como el coste de cada una de las instrucciones más el coste de las funciones que ejecuta, donde cada instrucción básica tiene coste unitario.

A continuación se analizan asintóticamente, una a una, cada una de las funciones que utiliza al gestor TLSF.

`fls()` **y** `ffs()`

Ambas funciones (ver paginas70 y 71) están compuestas por 6 sentencias condicionales secuenciales, sin bucle alguno. El coste temporal es, por lo tanto, constante, $O(1)$.

`insert_func()`

La función `insert_func()` (ver página 72) traduce un tamaño r de bloque al índice (i, j) de la lista donde el bloque ha de ser insertado. Esta función no contiene ningún bucle, solamente varias operaciones aritméticas y lógicas sencillas, y una llamada a la función `fls()`. El coste computacional de esta función es acotado, $O(1)$.

`search_func()`

La función `search_func()` (página 72) traduce un tamaño r al índice

(i, j) correspondiente a la primera lista que puede satisfacer dicho tamaño. Esta función se compone, básicamente, de funciones aritméticas sencillas y de una llamada a la función `fls()`, siendo, por lo tanto, constante y acotado ($O(1)$) el coste temporal de esta función.

`set_bit()`, `clear_bit()` y `is_bit_set()`

Todas estas funciones permiten trabajar con mapas de bits, la función `set_bit()` (ver página 73) cambia un bit determinado a 1, `clear_bit()` (ver página 73) cambia un bit determinado a 0 y `is_bit_set()` (ver página 73) devuelve el estado de un bit determinado. Todos los mapas de bit usados por el gestor TLSF tienen un tamaño igual al tamaño de la palabra de procesador. Por lo tanto, todas estas funciones presentan un tiempo de respuesta constante, $O(1)$.

`head_list()`, `insert_list()` y `remove_list()`

Todas estas funciones permiten trabajar con una lista doblemente enlazada. La función `head_list()` (página 73) devuelve un puntero al bloque que se encuentra en la cabeza de la lista. La función `insert_list()` (página 73) modifica el puntero de la cabeza y enlaza los punteros anterior y posterior. Y `remove_list()`(página 73) elimina una bloque de la lista, esta función recibe como argumento dicho bloque. Son funciones elementales de operaciones sobre listas con un coste de $O(1)$.

`find_block()`

La función `find_block()` (página 74) busca en la estructura TLSF, comenzando desde la lista apuntada por los índices (i, j), un bloque de memoria libre. Esta función no realiza ninguna búsqueda lineal en la estructura TLSF, simplemente busca una lista no vacía que sea posterior a (i, j). El primer bloque de dicha lista es devuelto. Para realizar la búsqueda en tiempo contante se utilizan mapas de bits. Esta función no contiene ni iteración ni recursividad en su código y todas las funciones que utiliza (`ffs()` y `head_list()`) son temporalmente constantes, por lo tanto, el coste temporal asintótico de esta función es constante, $O(1)$.

`insert_block()`

La función `insert_block()` (página 74) inserta el bloque dado en la lista indexada por los índices i y j y actualiza dos mapas de bits de tamaños constantes. Todas las funciones usadas por esta función (`insert_list()` y `set_bit()`) tienen un coste temporal constante, $O(1)$.

`remove_block()`

Esta función (ver página 75) extrae un bloque prefijado de la lista a la que pertenece y actualiza los mapas de bits en consecuencia. Esta función contiene únicamente código secuencial y todas las funciones auxiliares que utiliza (`remove_list()` y `clear_bit()`) tienen un coste temporal constante. Por ello, el coste computacional de esta función es constante $O(1)$.

`merge_prev()`

La función `merge_prev()` (página 75) fusiona un bloque dado con el bloque físicamente previo, en caso de que éste esté libre. Esta función está compuesta por una sentencia condicional secuencial, sin ningún bucle y una llamada a las funciones `get_prev_block()`, `is_free()` y `remove_block()`, ambas temporalmente constante. Por lo tanto, el coste temporal de esta función es constante, $O(1)$.

`merge_next()`

La función `merge_next()` (ver página 75) fusiona un bloque dado con el bloque físicamente siguiente, en caso de que éste esté libre. Esta función está compuesta únicamente por una sentencia condicional secuencial, sin bucles y una llamada a la funciones `get_next_block()`, `is_free()` y `remove_block()`, ambas temporalmente constante. Por lo tanto, el coste temporal de esta función es constante, $O(1)$.

`split()`

La función `split()`(ver página 76) divide un bloque de memoria b en dos bloques de memoria más pequeños $b1$ y $b2$. Esta función consiste en aritmética de punteros (crear una nueva cabecera en el bloque ya existente) y modificar la cabecera del bloque a dividir, por esto, el coste temporal asintótico de esta función es constante, $O(1)$.

`malloc()`

La función `malloc()` (ver página 76) es la encargada de asignar memoria. Esta función no contiene ni recursividad ni iteración, solamente código secuencial. Todas las funciones auxiliares usadas por `malloc` (`search_func()`, `find_block()`, `remove_block()`, `size()`, `split()`, `insert_func()` y `insert_block()`) tienen un coste temporal constante. Por lo tanto, el coste temporal asintótico de esta función es constante, $O(1)$.

`free()`

La función `free()` (ver página 77) se encarga de liberar los bloques previamente asignados. Al igual que la función `malloc()`, esta función no contiene ni iteración ni recursividad y solamente utiliza funciones auxiliares (`merge_prev()`, `merge_next()`, `insert_func()`, `size()` y `insert_block()`) de coste constante. El coste temporal de esta función es constante, $O(1)$.

Por lo tanto, las dos funciones principales, `malloc()` y `free()`, del gestor TLSF presentan un coste asintóticamente constate:

`malloc()`	`free()`
$O(1)$	$O(1)$

Tabla 5.1: Coste asintótico del algoritmo TLSF.

Esta característica lo convierte en un algoritmo idóneo para sistemas de tiempo real.

5.3. Análisis experimental

La evaluación experimental del tiempo de respuesta de un algoritmo permite estudiar el comportamiento del mismo cuando se ejecuta con un conjunto de cargas reales o sintéticas.

En la presente sección, el gestor TLSF ha sido evaluado utilizando ambos tipos de cargas. Por una parte, se ha empleado un conjunto de aplicaciones: CFRAC, Espresso, GAWK, GS y Perl. El apéndice B.1 incluye un estudio detallado de cada una de estas aplicaciones y de sus patrones de uso de memoria dinámica. Estas aplicaciones han sido seleccionadas por haber sido previamente empleadas en estudios sobre memoria dinámica [147, 52, 148]. Por otra parte, se ha construido el escenario de peor respuesta temporal. Por último, tal como propone B. Zorn en [148, 29], se ha implementado un simulador que, a partir de trazas de aplicaciones reales, reproduce el patrón de petición de memoria de las mismas.

Para terminar, se ha estudiado también el comportamiento temporal de todos los algoritmos considerados en la sección 3.4.5, comparando los resultados obtenidos con los de TLSF.

5.3.1. Tiempo de respuesta: métricas

Actualmente se dispone de varios métodos diferentes para el medir tiempo de respuesta de un algoritmo. Por ejemplo, se puede contar el número de instrucciones ejecutadas, contar el número de ciclos de procesador trans-

curridos, o utilizar un temporizador. Cada método presenta una serie de ventajas y desventajas frente al resto.

En el presente capítulo se han utilizado tres métricas para medir el parámetro *tiempo de respuesta*:

Instrucciones ejecutadas: el mismo programa ejecutado en diferentes implementaciones de una misma arquitectura (por ejemplo, Pentium, K7, Via o C3) ejecuta el mismo número de instrucciones. Además, un mismo algoritmo alimentado dos veces con la misma entrada siempre proporciona el mismo número de instrucciones ejecutadas.

Ciclos de procesador sin ejecutivo: esta segunda métrica consiste en obtener el tiempo requerido por el gestor de memoria dinámica cuando el programa es el único programa ejecutado por el procesador. Para obtener esta métrica se ha implementado un entorno de ejecución mínimo, llamado SA-Tester. El SA-Tester provee un entorno de ejecución que permite ejecutar una aplicación sencilla en modo monotarea. Las interrupciones se mantienen en todo momento inhibidas. El apéndice C contiene una descripción completa del mismo.

Esta métrica proporciona un resultado más realista que el obtenido en un sistema con múltiples programas ejecutándose concurrentemente.

Ciclos de procesador con ejecutivo: esta última métrica se obtiene ejecutando la aplicación sobre un sistema operativo. A diferencia de la métrica anterior, donde la aplicación tenía asegurada la falta de interferencias por parte de otras aplicaciones o por parte del propio núcleo, en esta métrica la aparición de esta interferencia no puede ser descartada. Sin embargo, esta métrica ha sido obtenida debido principalmente a que es perfectamente lógico pensar que el algoritmo TLSF será ejecutado en un entorno con núcleo. Para realizar esta métrica se ha optado por utilizar el sistema operativo Linux, sin ningún servicio (demonio) activo.

Obtener el número de instrucciones ejecutadas es costoso y no siempre puede hacerse. Sin embargo, los resultados obtenidos por esta métrica no se ven afectados por el procesador utilizado. Por otra parte, en el caso de la arquitectura x86 de Intel® se dispone del registro TSC, el cual indica el número de ciclos de procesador transcurridos desde el último *reset* del procesador. Por lo tanto, en esta arquitectura, el cálculo del número de ciclos de procesador usados en la ejecución de una prueba resulta trivial. Esta métrica, sin embargo, se ve afectada por diversos factores relacionados con el procesador utilizado (tipo de procesador, tamaño de cache, etc).

No siempre ha sido posible utilizar todas las métricas propuestas en todos los experimentos realizados. En el caso de cargas reales, por ejemplo, ha sido indispensable utilizar un sistema operativo que soporte dichas

aplicaciones, con lo cual solamente se ha podido utilizar la tercera métrica.

5.3.2. Escenario de peor caso

I. Puaut propuso en [97, 96] la evaluación de los gestores de tiempo real mediante el estudio de su comportamiento temporal en un hipotético peor escenario.

Sin embargo, construir un escenario de peor caso para un algoritmo no resulta una tarea trivial. Además de fabricarlo, es necesario demostrarlo formalmente. Debido a que esta tarea queda fuera del ámbito de esta tesis, se ha optado en utilizar los escenarios de peor caso previamente demostrados. En el caso de no disponer de dichos escenarios para algún gestor, se ha optado por proponer un escenario que consigue un mal tiempo de respuesta y definir el escenario como *escenario de mal caso* en lugar de peor caso. Un *mal caso* de un gestor de memoria obtiene un mal tiempo de respuesta de dicho gestor, sin embargo, podría existir un escenario peor.

Dos han sido los objetivos en esta sección: primero, fabricar el escenario de peor/mal caso del algoritmo TLSF y de cada uno de los gestores descritos en la sección 3.4.5 (First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL y Half-Fit). Segundo, evaluar la respuesta de cada uno de estos gestores con cada escenario de peor caso y compararla.

Se han utilizado dos métricas para medir el tiempo de respuesta: el número de instrucciones ejecutadas y el número de ciclos de procesador sin ejecutivo.

La nomenclatura seguida para describir los escenarios de peor/mal caso es la siguiente: $\mathcal{H}$ indica el tamaño del montículo inicial y n indica el tamaño del bloque mínimo asignable por el gestor.

5.3.2.1. Asignación de memoria: escenario de peor caso

Para la función `malloc`, los escenarios de peor/mal caso son los siguientes:

First-Fit/Best-Fit: dos condiciones se tienen que cumplir para que el escenario creado sea el de peor caso para estos gestores: primero, la lista que almacena los bloque libres debe ser de longitud máxima. Segundo, el bloque libre buscado se debe encontrar al final de ésta. Este escenario fuerza a ambos algoritmos a recorrer toda la lista de bloques libres para satisfacer la asignación. Para fabricar este escenario, se realizan peticiones del tamaño mínimo asignable por el gestor (n) hasta dejar al gestor exhausto. Posteriormente, suponiendo que el algoritmo inserta en la cabeza de la lista los bloque libres, se liberan dos bloques físicamente contiguos. Estos bloques serán unidos creando un bloque de tamaño $2{\cdot}n$.

A continuación solo resta liberar los bloques pares. La lista de bloques libres estará formada por lo tanto de una sucesión de bloques de tamaño n y un último bloque de tamaño $2 \cdot n$. El coste de una búsqueda será, por lo tanto, $O(\mathcal{H}/(2 \cdot n))$.

Binary Buddy: en el caso de este algoritmo, el escenario de peor caso se presenta cuando se realiza una petición inicial de tamaño mínimo n. El algoritmo necesita, entonces, dividir el montículo inicial en potencias de dos un total de $\lceil \log_2(\mathcal{H}/n) \rceil$ veces. Siendo el coste total del algoritmo: $O(\log_2(\mathcal{H}/n))$.

Doug Lea's malloc: el escenario de mal caso propuesto intenta explotar la sobrecarga temporal que produce el uso de la técnica de fusión aplazada. Un caso extremo ocurre cuando todo el montículo ha sido asignado como bloques de tamaño mínimo y posteriormente todos estos bloques son liberados. Esto produce una gran lista de bloques de tamaño mínimo. Cuando un tamaño mayor es requerido y no existen bloques de ese tamaño o superior, los bloques contenidos en la lista serán fusionados antes de atender la petición. El siguiente pseudocódigo genera el caso:

```
1  -- Inicialización del escenario
2  for i in 1.. Heap_Size/Mim_Size loop
3        allocated[i]:= malloc(Min_Size);
4  end loop;
5  for i in 1.. Heap_Size/Min_Size loop
6        free (allocated[i]);
7  end loop;
8
9  malloc(4233); -- Esta asignación representa un mal caso.
```

El número de bloques libres que han sido fusionados es $\mathcal{H}/n$. Por lo que la complejidad asintótica resultante es: $O(\mathcal{H}/n)$.

Árbol AVL: existen varias operaciones en este algoritmo que contribuyen al coste del algoritmo. Buscar y eliminar de la estructura un bloque adecuado:

$$O(1{,}44 \cdot \log_2(\mathcal{H}/n))$$

Insertar el bloque restante (solamente si el bloque encontrado es de tamaño mayor que el pedido), y rebalancear el árbol:

$$O(1{,}44 \cdot \log_2(\mathcal{H}/n))$$

No es simple encontrar el escenario de peor caso debido al comportamiento altamente dinámico de la estructura de datos AVL. Por lo tanto, se propone el siguiente mal caso: construir el árbol AVL más grande posible y requerir un bloque que únicamente pueda ser servido mediante

el uso de un bloque que se encuentre indexado por una hoja del árbol. El tamaño de bloque requerido provocará la partición del bloque encontrado en dos, obligando al algoritmo una inserción y un rebalanceado. Por lo cual, el coste es $O\left(2 \cdot 1{,}44 \cdot \log_2\left(\mathcal{H}/n\right)\right)$.

Half-Fit/TLSF: en el caso de estos algoritmos, ya que sus operaciones no dependen del número de bloques libres que gestionan y no existe ningún bucle en sus códigos, solamente se pueden esperar pequeñas variaciones en la ejecución temporal dependiendo en el código condicional ejecutado. Por lo tanto, podemos considerar el siguiente caso como mal caso: una petición de tamaño n, cuando el montículo inicial se encuentra todavía entero. El coste es $O(1)$.

5.3.2.2. Liberación de memoria: escenarios de peor caso

Los escenarios de peor caso o mal caso para la función `free` son los siguientes:

First-Fit/Best-Fit: la operación de liberación de ambos gestores es muy simple: une el bloque liberado con los vecinos físicos, en caso de que estos estuvieran libres e inserta en la cabeza de la lista dicho bloque. Por lo tanto, el peor caso se da cuando ambos vecinos físicos al bloque liberado se encuentran libres. El coste de esta operación es $O(1)$.

Binary-Buddy: el peor caso de liberación de memoria para este algoritmo se presenta cuando existe únicamente un bloque de tamaño mínimo (n) asignado. Al ser liberado, este bloque será iterativamente unido a todos los bloques existentes en la estructura de datos del gestor hasta generar el montículo inicial. El coste asintótico de esta operación es $O\left(\log_2\left(\mathcal{H}/n\right)\right)$.

Doug Lea's malloc: ya que el algoritmo Doug Lea's malloc pospone la operación de fusión de los bloques liberados y la realiza según sea necesaria en la operación `malloc`, la operación `free` es extremadamente rápida. Cualquier liberación de bloque tiene siempre un coste similar, el cual es $O(1)$.

AVL: para el algoritmo AVL se ha diseñado y utilizado el siguiente mal caso: en un árbol de altura máxima (un árbol que almacena tantos bloques de tamaño diferentes como le es posible), se libera un bloque que fuerza la fusión con sus dos vecinos físicos libres. El bloque resultante de esta unión será insertado como un nodo hoja en el árbol. El coste es $O\left(3 \cdot 1{,}44 \cdot \log_2\left(\mathcal{H}/n\right)\right)$.

Half-Fit/TLSF: los gestores Half-Fit y TLSF solamente tienen tres casos diferentes ante la operación de liberación de memoria. Primero, que el bloque liberado no pueda ser unido con ningún bloque existente. Se-

gundo, solamente uno de los bloques físicamente contiguos al bloque se encuentre libre. Por lo tanto se produce una única fusión. Tercero, los dos bloques físicamente contiguos al bloque liberado se encuentran libres. Por lo tanto se producen dos fusiones. El peor escenario se presenta, por lo tanto, en el último caso expuesto. Siendo el coste temporal asintótico $O(1)$.

	Asignación	Liberación
First-fit/Best-fit	$O\left(\mathcal{H}/\left(2 \cdot n\right)\right)$	$O(1)$
Binary-buddy	$O\left(\log_2\left(\mathcal{H}/n\right)\right)$	$O\left(\log_2\left(\mathcal{H}/n\right)\right)$
DLmalloc	$O\left(\mathcal{H}/n\right)$	$O(1)$
AVL-tree	$O\left(2 \cdot 1{,}44 \cdot \log_2\left(\mathcal{H}/n\right)\right)$	$O\left(3 \cdot 1{,}44 \cdot \log_2\left(\mathcal{H}/n\right)\right)$
Half-fit/TLSF	$O(1)$	$O(1)$

Tabla 5.2: Resumen del coste de los peores/malos casos asignación/liberación

5.3.2.3. Escenarios de peor/mal caso: resultados

En todas las pruebas realizadas se ha utilizado un AMD® Athlon XP 2000 con 1 GByte de memoria física. El montículo de memoria ($\mathcal{H}$) ha tenido un tamaño de 4 Mbytes en todas las pruebas. El mínimo tamaño de bloque asignable por los gestores (n) ha sido 16 bytes.

Se han utilizado dos métricas diferentes para obtener el tiempo de respuesta: el número de instrucciones ejecutadas, esta métrica es independiente de la máquina utilizada; y los ciclos de procesador sin operativo, lo que nos permite valorar el impacto de la máquina subyacente (cache, tiempo de respuesta de la memoria, etc).

Al no verse afectado por la máquina utilizada, el número de instrucciones ha sido obtenido utilizando el siguiente pseudocódigo:

```
1  reset_intruction_number();
2  malloc() o free ();
3  instructions := read_instruction_number();
```

La línea 1 inicializa el contador de instrucciones y la línea 3 obtiene el valor actual. Para contar en la arquitectura Intel® IA32 el número de instrucciones ejecutadas en un programa, primero, se intercepta la interrupción hardware 1 con una rutina que incrementa una variable global cada vez que se ejecute esta interrupción; segundo, se pone al procesador en modo depuración; en este modo, el procesador produce una interrupción 1 cada vez que se ejecuta una instrucción.

La obtención de los ciclos de procesador ha sido un proceso más complejo. Esta métrica se ve afectada por diversos factores: la máquina utilizada, el tamaño de cache, el número de pasos de la instrucción de desplazamiento, etc. Para minimizar el impacto de estos factores, y debido que el código ejecutado para realizar las pruebas es relativamente simple (no necesita servicios del sistema operativo), estas han sido ejecutadas sin operativo. En su lugar se ha utilizado el SA-Tester, un entorno mínimo de ejecución, programado específicamente para estos experimentos. La función del SA-Tester consiste en arrancar la máquina, dejarla en un estado estable con interrupciones inhibidas y ejecutar la correspondiente prueba. En el apéndice C se explica de forma más detallada este entorno de ejecución.

Para realizar la medidas, se ha utilizado el siguiente código, el cual evita el efecto de la cache, inicializándola justamente antes de medir (línea 2).

```
 1  disable_interrupts ();
 2  flush_cache ();
 3  cpuid();
 4  t1:= read_tsc ();
 5  malloc() o free ();
 6  cpuid();
 7  t2:= read_tsc ();
 8  cpuid();
 9  t3:= read_tsc ();
10  enable_interrupts ();
11  time:= (t2 − t1) − (t3 − t2);
```

Las líneas 3, 6 y 8 permiten serializar la ejecución de las instrucciones, asegurando que cuando se lee el número de ciclos en ese momento todas las instrucciones máquina anteriores han completado su ejecución. La función `read_tsc()` devuelve el valor del contador TSC, contador que se incrementa automáticamente en cada ciclo del procesador.

Las tabla 5.2(b) y 5.2(a) muestran los resultados de los escenarios de peor/mal caso (*Worst-Case*/*Bad-Case*) para la operación de asignación. Cada gestor ha sido ejecutado con cada escenario de peor caso construido. El resultado de ejecutar un gestor con su peor caso ha sido resaltado en negrita. Los resultados muestran que cada algoritmo obtiene su peor tiempo de ejecución en su peor escenario. La tabla 5.2(c) presenta los ciclos por instrucción (CPI) de cada gestor, calculados a partir de las dos primeras tablas.

Como se esperaba, los gestores First-Fit y Best-Fit se comportan muy mal bajo sus peores casos. La baja localidad espacial ocasiona un alto porcentaje de fallos de cache, lo que produce un tiempo de respuesta incluso peor de lo que en un principio se esperaba. El número de ciclos por ins-

(a) Instrucciones de procesador

Malloc	FF	BF	BB	DL	AVL	HF	TLSF
FF WC	**81995**	**98385**	115	109	699	162	197
BB WC	86	94	**1403**	729	353	162	188
DL BC	88	96	1113	**721108**	353	164	197
AVL BC	5085	6093	252	56093	**3116**	162	197
TLSF/HF WC	88	96	1287	729	3053	**164**	**197**

(b) Ciclos de procesador

Malloc	FF	BF	BB	DL	AVL	HF	TLSF
FF WC	**161326**	**158755**	1445	1830	6471	1633	2231
BB WC	1168	1073	**3898**	4070	3580	1425	2388
DL BC	1203	1227	3208	**331325**	3844	1651	2251
AVL BC	105835	101497	1703	13216	**11739**	1629	2149
TLSF/HF WC	1168	1074	3730	4124	3580	**1690**	**2448**

(c) CPI

Malloc	FF	BF	BB	DL	AVL	HF	TLSF
FF WC	**19**	**16**	12	16	9	10	11
BB WC	13	11	**2**	5	10	8	12
DL BC	13	12	2	**4**	10	10	11
AVL BC	20	16	6	2	**3**	10	10
TLSF/HF WC	13	11	2	5	1	**10**	**12**

Tabla 5.3: Peores Casos (WC) y Malos Casos (BC) de asignación.

trucción (CPI) es alto: 19 CPI en el caso del First-Fit y 16 CPI en el del Best-Fit.

Es interesante señalar que, aunque el gestor TLSF y el gestor Half-Fit tienen una alta localidad espacial, presentan un alto número de CPI, 12 y 10 CPI respectivamente. Esto se debe a dos factores: primero, el uso de operaciones aritméticas y lógicas para encontrar directamente el bloque libre adecuado. Y segundo, la complejidad de la estructura de datos utilizada para organizar las listas segregadas. El mismo conjunto de pruebas ejecutadas sin invalidar la cache del procesador reducía a 150 ciclos de procesador en el caso de TLSF. Pero dejaba invariante el número de ciclos en el caso del First-Fit y el Best-Fit. En cualquier caso, los CPI obtenidos son muy estables y dependen más del diseño del procesador (los procesadores Intel® muestran unos resultados en términos de CPI mejores que los AMD® en estos experimentos) que de la distribución de datos utilizada.

El escenario de mal caso del gestor AVL, el cual genera una gran cantidad de bloques de diferente tamaño, ha resultado ser también, un escenario de mal caso para la mayoría de algoritmos. Solamente Binary Buddy, Half-Fit y TLSF se comportan correctamente en este escenario. Esto se debe a que el coste de la mayoría de los gestores analizados depende tanto del número total de bloques libres como del número de bloques de tamaño diferente.

El gestor DLmalloc es un buen ejemplo de algoritmo diseñado para optimizar el tiempo de respuesta promedio. DLmalloc, presenta unos tiempos de respuesta muy bajos. Sin embargo, en su mal caso, muestra el peor tiempo de respuesta de todos los algoritmos. Ello se debe a su política de retrasar la unión de bloques libres físicamente contiguos. Cuando al final la debe realizar, une todos los bloques cuya unión había previamente pospuesto, provocando una gran sobrecarga temporal. DLmalloc muestra el peor de todos los tiempos de respuesta, por lo tanto, no es aconsejable su uso en sistemas de tiempo real.

Los gestores Half-Fit, TLSF, Binary Buddy y AVL presentan un tiempo de respuesta razonablemente bajo. Half-Fit y TLSF además muestran una gran estabilidad y uniformidad en los resultados, tanto en tiempo de respuesta como en número de instrucciones. Half-Fit es el gestor que mejores tiempos de respuesta muestra, aventajando a TLSF en aproximadamente un 20 %.

Por otra parte, aunque el coste temporal del gestor AVL no es tan bueno como el de Half-Fit y el de TLSF, su cota asintótica mostrada $O\left(\log\left(\mathcal{H}/n\right)\right)$ es aceptable para la mayoría de las aplicaciones. Además, ya que AVL implementa una política Best-Fit y el bloque asignado tiene siempre el tamaño requerido, este algoritmo no sufre de fragmentación interna.

La tabla 5.4 resume el número de instrucciones y los ciclos de procesador requeridos por la operación de liberación. Ya que todos los gestores muestran

un coste de liberación muy estable y uniforme se ha optado por mostrar solamente los resultados de cada gestor con su peor/mal caso.

Free	FF	BF	BB	DL	AVL	HF	TLSF
Instr. proc.	115	115	1379	51	1496	130	187
Ciclos proc.	1241	1289	4774	955	7947	1110	2151
CPI	10	11	3	18	5	8	11

Tabla 5.4: Peores/malos casos (WC/BC) de liberación.

Todos los gestores, con las excepciones de AVL y Binary Buddy, no realizan ningún tipo de búsqueda en la operación de liberación, es decir, no contienen iteración ni recursividad. Por lo tanto, todos estos gestores presentan un tiempo de respuesta muy bajo y estable. Binary Buddy realiza la operación de fusión de bloques hasta un máximo de $\log_2(\mathcal{H})$ veces, por ello su coste temporal es alto. El alto coste presentado por el gestor AVL se debe a la complejidad de eliminar un nodo en un árbol AVL. Una operación de liberación puede requerir, en el peor de los casos, dos extracciones y una inserción.

La política de posponer la unión de bloques, implementada por el gestor DLmalloc, convierte a este último en el algoritmo que mejores tiempo de respuesta presenta en la operación de liberación.

5.3.3. Cargas reales

La evaluación del escenario de peor caso proporciona justamente un resultado requerido por los sistemas de tiempo real: el tiempo de ejecución en el peor escenario. Tal como concluyó J. M. Robson en [107], la distancia entre el escenario de peor caso y el caso promedio es muy amplia, ya que el escenario de peor caso raramente se presenta.

En esta sección se ha estudiado el comportamiento del gestor TLSF al ejecutarlo con un conjunto de aplicaciones reales. Además, los resultados han sido comparados con los obtenidos por el resto de gestores dinámicos descritos en la sección 3.4.5.

El conjunto de aplicaciones utilizadas han sido seleccionadas por su gran popularidad en los estudios previos sobre memoria dinámica [140, 57, 148]. En el apéndice B.1 se incluye una descripción exhaustiva de las mismas. Estas aplicaciones y las pruebas ejecutadas son:

CFRAC: el programa CFRAC permite factorizar números enteros a través del método de fraccionar dichos enteros continuamente. A continuación se describen las entradas utilizadas para realizar las diferentes pruebas con este programa.

- Test 1: factorización de 1000000001930000000057.
- Test 2: factorización de 327905606740421458831903.
- Test 3: factorización de 417576463441248601459380302877l.
- Test 4: factorización de 41757646344123832613190542166099121.

Espresso: el programa Espresso, del cual se ha utilizado la versión 2.3, es un programa de optimización lógica. Es decir, como entrada acepta un circuito combinacional y como resultado se obtiene dicho circuito optimizado. Los tests utilizados para su evaluación han consistido en:

- Test 1: se optimiza un circuito combinacional con 7 entradas y 10 salidas.
- Test 2: se optimiza un circuito combinacional con 8 entradas y 8 salidas.
- Test 3: se optimiza un circuito combinacional con 24 entradas y 109 salidas.
- Test 4: se optimiza un circuito combinacional con 16 entradas y 40 salidas.

GAWK: el programa Gnu AWK, del cual se ha utilizado la versión 3.1.3, es un intérprete para el lenguaje AWK, utilizado para realizar informes. Los tests utilizados han sido:

- Test 1: procesamiento de un texto pequeño para crear un *checksum*.
- Test 2: ajuste de las líneas que conforman un texto según las opciones dadas, utilizando un texto pequeño (unos pocos Kbytes).
- Test 3: igual que el test anterior, esto es ajuste de las líneas que conforman un texto según las opciones dadas, utilizando un texto grande (varios MBytes).

GS: GhostScript, versión 7.07.1, es un intérprete libre para el lenguaje de descripción de formato de páginas Postscript. Los tests utilizados han sido:

- Test 1: procesamiento de una única página que contiene dos gráficas.
- Test 2: procesamiento de la guía de usuario de la biblioteca C++ de GNU.
- Test 3: procesamiento del manual del lenguaje SELF, desarrollado en la universidad de Stanford.

Perl: el programa Perl, del cual se ha utilizado la versión 5.8.4, es un intérprete del lenguaje del mismo nombre, optimizado para realizar búsquedas arbitrarias en ficheros de texto, permitiendo extraer información de dichos ficheros y crear informes basados en dicha información.

- Test 1: ordenación de las entradas contenidas en un fichero utilizando para ello la clave primaria que se da al final de cada una de las entradas de dicho fichero.

- Test 2: script que traduce un fichero /etc/hosts en formato unixops a formato CS.
- Test 3: ajuste de las líneas que conforman un texto según las opciones dadas, utilizando un texto pequeño (unos pocos Kbytes).
- Test 4: igual que el test anterior, esto es ajuste de las líneas que conforman un texto según las opciones dadas, utilizando un texto grande (varios MBytes).

La ejecución de aplicaciones reales es más compleja que la ejecución de escenarios sintéticos. Las aplicaciones requieren de los servicios proporcionados por el sistema operativo (sistema de ficheros, etc.). Por lo tanto, el tiempo de respuesta ha sido obtenido midiendo los ciclos de procesador. Además, tampoco se ha podido contar el número de instrucciones ejecutadas, a pesar de que es factible, por el gran coste temporal que conlleva. Para minimizar el impacto del sistema operativo todos los experimentos han sido realizados en la distribución de Linux Gentoo [44], en modo monousuario, sin ningún servicio activo. El pseudocódigo utilizado para obtener los resultados ha sido:

```
 1  disable_interrupts ();
 2  cpuid();
 3  t1:= read_tsc();
 4  malloc() o free();
 5  cpuid();
 6  t2:= read_tsc();
 7  cpuid();
 8  t3:= read_tsc();
 9  enable_interrupts ();
10  time:= (t2 − t1) − (t3 − t2);
```

A diferencia del código utilizado en el caso de los escenarios de peor/mal caso, aquí no se está vaciando la memoria cache, por lo cual, el estado previo de dicha memoria es desconocido.

El tamaño del montículo utilizado ha sido de 16 Mbytes, mientras que el tamaño del bloque más pequeño asignable por los gestores ha sido 16 bytes.

5.3.3.1. Carga real: resultados

Las tablas 5.5, 5.6, 5.7, 5.8 y 5.9 muestran los resultados obtenidos, en ciclos de procesador, por cada uno de los gestores de memoria considerados cuando son ejecutados con cada una de las aplicaciones. En negrita se han resaltados los mejores resultados para cada aplicación.

El comportamiento de las aplicaciones (secuencia de operaciones `malloc` y `free`) tienen un gran impacto en la respuesta de los gestores. No resulta

Malloc/Free	Test 1	Test 2	Test 3	Test 4
FF media	94 / 65	131 / 69	129 / 72	132 / 87
desv.std.	40 / 38	520 / 32	505 / 32	486 / 41
BF media	96 / 65	138 / 69	142 / 75	144 / 89
desv.std.	47 / 36	511 / 30	518 / 32	519 / 44
BB media	1131 / 114	265 / 145	254 / 153	241 / 182
desv.std.	7843 / 49	1904 / 98	1705 / 106	1202 / 107
DL media	**90** / **30**	120 / **24**	115 / **25**	**108** / **24**
desv.std.	127 / 16	442 / 18	423 / 8	440 / 15
AVL media	512 / 335	750 / 603	780 / 762	503 / 801
desv.std.	703 / 244	810 / 385	867 / 478	931 / 497
HF media	107 / 143	**108** / 98	**114** / 116	114 / 118
desv.std.	121 / 210	69 / 50	68 / 55	61 / 43
TLSF media	231 / 204	146 / 116	149 / 121	158 / 121
desv.std.	451 / 305	87 / 85	102 / 87	85 / 57

Tabla 5.5: Aplicación CFRAC, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3	Test 4
FF media	**77** / 73	**73** / 73	**74** / 69	**74** / 71
desv.std.	124 / 29	74 / 30	105 / 27	56 / 27
BF media	132 / 70	158 / 69	196 / 60	201 / 71
desv.std.	109 / 32	76 / 32	111 / 29	75 / 30
BB media	131 / 132	130 / 138	128 / 128	128 / 136
desv.std.	440 / 89	288 / 92	268 / 84	121 / 92
DL media	88 / **25**	85 / **25**	110 / **26**	93 / **24**
desv.std.	319 / 10	262 / 9	845 / 11	905 / 8
AVL media	1132 / 683	1144 / 709	1305 / 708	1242 / 624
desv.std.	373 / 468	339 / 505	448 / 556	356 / 505
HF media	80 / 104	79 / 104	78 / 98	78 / 102
desv.std.	20 / 38	15 / 35	12 / 32	12 / 33
TLSF media	126 / 108	123 / 108	114 / 95	122 / 105
desv.std.	36 / 47	30 / 47	31 / 42	20 / 41

Tabla 5.6: Aplicación Espresso, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3
FF media	136 / 64	**77** / 80	**77** / 79
desv.std.	659 / 38	36 / 34	25 / 33
BF media	160 / 59	165 / 58	161 / 56
desv.std.	609 / 41	65 / 31	64 / 30
BB media	213 / 112	103 / 97	103 / 98
desv.std.	1506 / 80	78 / 55	66 / 58
DL media	131 / **28**	132 / **28**	130 / **27**
desv.std.	641 / 16	249 / 13	247 / 13
AVL media	1081 / 424	1161 / 810	1148 / 806
desv.std.	1388 / 373	422 / 461	423 / 455
HF media	**81** / 109	88 / 115	89 / 115
desv.std.	38 / 48	15 / 36	14 / 36
TLSF media	120 / 92	124 / 101	123 / 99
desv.std.	62 / 58	34 / 49	35 / 47

Tabla 5.7: Aplicación GAWK, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3
FF media	1857 / 170	**196** / 196	1918 / 192
desv.std.	3031 / 277	486 / 166	3278 / 372
BF media	1834 / 148	389 / 286	1849 / 184
desv.std.	2933 / 214	512 / 286	2997 / 323
BB media	1554 / 241	345 / 301	1130 / 321
desv.std.	4827 / 232	622 / 217	3395 / 377
DL media	2042 / **131**	801 / **143**	1903 / **128**
desv.std.	2970 / 190	882 / 127	3206 / 211
AVL media	3312 / 693	2483 / 1799	3788 / 568
desv.std.	3041 / 965	1558 / 979	3827 / 919
HF media	**227** / 170	343 / 322	**214** / 176
desv.std.	189 / 202	301 / 298	182 / 257
TLSF media	685 / 413	344 / 302	419 / 257
desv.std.	964 / 693	245 / 207	533 / 454

Tabla 5.8: Aplicación GS, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3	Test 4
FF media	350 / 119	126 / 132	83 / 82	83 / 91
desv.std.	1900 / 149	363 / 65	81 / 26	52 / 30
BF media	263 / 95	170 / 105	131 / 65	135 / 58
desv.std.	1000 / 61	352 / 54	86 / 35	56 / 32
BB media	372 / 156	192 / 146	111 / 111	112 / 112
desv.std.	2376 / 115	761 / 84	167 / 54	108 / 55
DL media	237 / **34**	**81 / 60**	**34 / 24**	**33 / 24**
desv.std.	970 / 34	403 / 66	79 / 6	51 / 6
AVL media	981 / 669	1593 / 813	1353 / 647	1338 / 640
desv.std.	1381 / 501	758 / 485	465 / 449	487 / 435
HF media	**95** / 125	99 / 126	87 / 114	76 / 92
desv.std.	52 / 59	29 / 47	19 / 29	16 / 27
TLSF media	149 / 141	218 / 180	109 / 82	123 / 100
desv.std.	96 / 106	164 / 157	51 / 52	42 / 48

Tabla 5.9: Aplicación Perl, `malloc` y `free` (Ciclos de procesador).

sencillo modelar o describir cómo una aplicación utiliza la memoria dinámica. Además, un pequeño cambio en la secuencia de peticiones `malloc`/`free` puede ocasionar un gran impacto en el rendimiento global del algoritmo. Por ejemplo, en el caso de los algoritmos con ajuste segregado, si el tamaño requerido no se encuentra al comienzo de la correspondiente lista segregada, entonces el gestor tiene que hacer un trabajo extra para encontrar un bloque adecuado.

Entre otros factores, los siguientes patrones de petición son relevantes: la cantidad de tamaños diferentes pedidos; la distancia entre estos tamaños; el número de operaciones de liberación (algunas aplicaciones, mientras se ejecutan, solamente liberan un pequeño porcentaje de la memoria pedida, liberándola al completo cuando terminan su ejecución); el tiempo de vida de cada bloque asignado (esto es, el tiempo transcurrido entre que un bloque es asignado y liberado); etc.

Las aplicaciones CFRAC, Espresso y GAWK se comportan de forma similar con respecto a los tamaños de bloque que requieren. Todas ellas hacen un uso intensivo de bloques de pequeño tamaño. Ghostscript (GS), por el contrario, exhibe un patrón de tamaños diferente (ver apéndice B.1). GS utiliza bloques de tamaño muy variado, desde unos pocos bytes a varios Kbytes. Las pruebas realizadas a GS han resultado ser las más exigentes para todos los gestores, incluyendo TLSF y Half-Fit. Pero, mientras la mayoría de gestores sufren una pérdida significativa de rendimiento (DLmalloc requiere 22 veces más tiempo que con las otras aplicaciones; Binary Buddy

15 veces; First-Fit 10 veces y AVL 8 veces), el tiempo de respuesta de los gestores TLSF y Half-Fit solamente es 2 o 3 veces peor que con el resto de aplicaciones, lo que confirma su estabilidad bajo diferentes tipos de cargas.

Respecto al tiempo de liberación, DLmalloc resulta imbatible, mientras que la operación de unión de bloques libres en los gestores AVL, Binary Buddy, Half-Fit y TLSF puede llegar a requerir hasta dos extracciones y una inserción. DLmalloc siempre inserta el bloque libre en la cabeza de una lista. AVL, por el contrario es el gestor que presenta los tiempos de liberación más elevados.

Cuando se considera el efecto combinado de la asignación y liberación de memoria, el gestor DLmalloc muestra los mejores tiempos de respuesta medios, salvo en el caso de la aplicación GS, donde Half-Fit es el que mejor se comporta.

5.3.4. Modelos sintéticos

La evaluación de un gestor de memoria dinámica mediante aplicaciones reales presenta la desventaja de que la secuencia de peticiones viene prefijada por la propia aplicación. B. Zorn en [148] propone la utilización de modelos sintéticos, obtenidos a partir de patrones de aplicaciones reales. La ventaja de utilizar estos modelos consiste en que la ejecución de la simulación puede ser arbitrariamente larga (tanto como se desee). Sin embargo, el principal inconveniente consiste en que actualmente no existe ningún modelo que simule de forma fidedigna el comportamiento de las aplicaciones.

Los parámetros más comúnmente utilizados para fabricar modelos sintéticos son:

Tiempo de posesión (*HT*): este parámetro indica el tiempo que transcurre entre que se asigna un bloque a la aplicación y ésta lo libera.

Tiempo entre llegadas (*IAR*): este parámetro indica el tiempo que transcurre entre una petición de asignación y la siguiente.

Tamaño de los objetos (*SC*): este parámetro indica el tamaño medio (bytes) de bloque requerido al gestor de memoria dinámica.

Normalmente, los modelos hacen uso de la media y la desviación típica de estos parámetros para, a partir de una distribución estadística, simular la aplicación.

Los dos modelos considerados para la presente sección han sido:

MEAN: el modelo MEAN consiste en caracterizar el comportamiento de una aplicación mediante tres parámetros: *HT*, *IAR* y *SC*. A partir de dichos parámetros se utiliza una distribución uniforme (desde cero hasta el doble de la media) para simular la aplicación. Una variante del modelo consiste en simular la aplicación utilizando una distribución normal a

partir de la media y la varianza de dichos parámetros. Este modelo no requiere una implementación demasiado compleja, lo cual explica su gran difusión [148, 139, 97, 96].

CDF: el modelo CDF consiste en construir funciones de probabilidad acumulativa a partir de los tres parámetros *HT*, *IAR* y *SC*. Posteriormente, mediante dichas funciones y una distribución uniforme entre $[0, 1[$, se simula el comportamiento de la aplicación. Un ejemplo de estas funciones de probabilidad es:

$$\texttt{SC CDF}\,(x) = \begin{cases} 0{,}0 & x < 32 \\ 0{,}3 & 32 \leq x < 64 \\ 0{,}6 & 64 \leq x < 512 \\ 1{,}0 & x \geq 1024 \end{cases}$$

Modelo CDF. Ejemplo función de distribución.

La cual indica que existe la misma probabilidad de que se requiera un bloque de 32 de 64 o de 512 bytes.

Como aplicaciones a ser simuladas se han utilizado CFRAC, Espresso, GAWK, GS y Perl, es decir, las utilizadas en la sección anterior. En el apéndice B.1 se encuentran la media y la desviación típica de los parámetros *HT*, *IAR* y *SC* de cada una de estas aplicaciones.

Para medir tiempo de respuesta se ha utilizado la misma métrica que en la sección anterior, ciclos de procesador con ejecutivo. Permitiéndonos, así establecer una comparativa de los resultados.

5.3.4.1. Modelo de valor medio (MEAN)

En esta sección, se ha fabricado el modelo MEAN de cada una de las aplicaciones estudiadas en el apéndice B.1 (CFRAC, Espresso, GAWK, GS y Perl). Posteriormente, se ha simulado, usando una distribución normal, con cada modelo y con cada gestor 10000 pasos de simulación. Las tablas 5.10, 5.11, 5.12, 5.13 y 5.14 muestran los ciclos de procesador obtenidos (medias y desviaciones típicas) en cada prueba. Para facilitar el estudio de los resultados, se ha subrayado en negrita las mejores medias obtenidas en cada prueba.

Como concluyó B. Zorn en [148], la simulación del modelo MEAN no se corresponde con los resultados obtenidos al ejecutar aplicaciones reales.

Por una parte, en el caso de los modelos MEAN para CFRAC, Espresso, GAWK y Perl (tablas 5.10, 5.11, 5.12 y 5.14) se observa, en general, mejores resultados, con disminución de las medias y las desviaciones típicas obtenidas. Esto se debe principalmente a dos razones: primero, el HT generado por el simulador suele ser muy bajo, con lo cual la memoria asignada es liberada

Malloc/Free	Test 1	Test 2	Test 3	Test 4
FF media	59,3 / 53,7	60,6 / 57,5	60,9 / 52,2	87,8 / 56,2
desv.std.	43,9 / 51,2	17,6 / 26,5	13,8 / 34,6	707,4 / 714,5
BF media	63,8 / 51,3	66,4 / 58,5	73,5 / 53,1	94,5 / 55,2
desv.std.	55,0 / 67,3	39,4 / 50,7	18,5 / 54,5	697,4 / 705,3
BB media	146,2 / 123,2	121,0 / 103,6	126,0 / 104,3	143,3 / 119,9
desv.std.	133,3 / 156,2	70,0 / 94,5	82,9 / 109,8	99,9 / 128,4
DL media	**49,0 / 26,5**	**41,2 / 22,0**	**40,0 / 21,6**	**45,2 / 23,0**
desv.std.	75,3 / 86,3	22,5 / 41,7	17,1 / 38,0	34,6 / 52,4
AVL media	324,8 / 194,1	317,5 / 194,6	343,7 / 184,7	335,9 / 190,7
desv.std.	303,8 / 402,8	203,4 / 324,2	953,9 / 1001,1	709,3 / 765,7
HF media	103,4 / 98,6	105,3 / 94,2	103,4 / 95,3	105,0 / 96,1
desv.std.	8,4 / 33,8	9,7 / 48,6	8,1 / 42,0	10,1 / 44,8
TLSF media	155,7 / 116,8	156,0 / 146,5	158,3 / 118,1	157,4 / 119,4
desv.std.	14,7 / 105,1	16,0 / 57,1	18,0 / 108,0	18,3 / 105,5

Tabla 5.10: Simulación MEAN CFRAC, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3	Test 4
FF media	**58,2** / 59,7	**56,0** / 55,6	**89,9** / 59,1	**74,4** / 58,1
desv.std.	28,8 / 26,2	10,9 / 13,1	734,8 / 743,9	367,1 / 372,1
BF media	65,1 / 60,0	62,7 / 58,1	95,3 / 61,0	81,7 / 58,2
desv.std.	21,7 / 33,8	14,9 / 28,4	713,4 / 723,0	392,9 / 399,2
BB media	133,8 / 99,2	146,8 / 116,8	182,7 / 109,0	150,2 / 97,5
desv.std.	99,2 / 134,7	82,8 / 122,0	876,6 / 896,0	407,7 / 425,8
DL media	182,5 / **42,3**	167,2 / **39,1**	183,6 / **35,6**	141,8 / **30,9**
desv.std.	161,1 / 240,7	111,1 / 197,4	828,4 / 854,6	552,3 / 572,5
AVL media	320,5 / 189,6	334,7 / 186,7	400,6 / 191,5	441,8 / 187,2
desv.std.	289,1 / 389,7	272,8 / 390,6	734,6 / 821,4	1123,4 / 1199,1
HF media	100,9 / 95,4	98,9 / 94,8	98,6 / 126,9	102,0 / 95,4
desv.std.	14,3 / 36,9	7,5 / 30,0	6,6 / 10,0	22,4 / 43,9
TLSF media	165,7 / 114,1	164,7 / 113,3	183,4 / 114,6	185,0 / 146,6
desv.std.	38,4 / 127,1	33,7 / 124,8	72,9 / 162,6	72,2 / 135,5

Tabla 5.11: Simulación MEAN Espresso, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3
FF media	**96,3** / 58,3	**87,2** / 56,6	**76,2** / 59,3
desv.std.	559,1 / 567,4	548,3 / 554,6	380,7 / 385,2
BF media	106,3 / 57,1	92,4 / 56,3	84,6 / 58,4
desv.std.	618,1 / 628,0	491,8 / 499,2	434,6 / 440,7
BB media	242,1 / 101,4	197,7 / 110,6	168,3 / 111,2
desv.std.	1446,1 / 1470,6	991,9 / 1009,4	404,0 / 425,1
DL media	141,9 / **30,1**	132,0 / **30,8**	128,0 / **30,9**
desv.std.	588,6 / 607,9	458,4 / 477,9	493,6 / 511,0
AVL media	491,1 / 182,4	443,4 / 172,8	488,1 / 178,6
desv.std.	1062,0 / 1162,1	844,4 / 941,8	1374,5 / 1453,5
HF media	98,5 / 96,8	98,2 / 94,1	98,2 / 94,4
desv.std.	13,5 / 24,9	5,0 / 29,9	4,7 / 28,8
TLSF media	212,2 / 113,3	193,2 / 113,3	196,1 / 113,1
desv.std.	99,2 / 206,4	82,4 / 177,9	88,0 / 183,8

Tabla 5.12: Simulación MEAN GAWK, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3
FF media	1095,5 / 59,4	360,8 / 57,3	1008,2 / 57,8
desv.std.	2528,4 / 2769,7	1412,0 / 1464,1	2342,0 / 2559,3
BF media	1047,2 / 61,1	357,4 / 57,7	1023,4 / 59,7
desv.std.	2402,2 / 2633,8	1415,7 / 1466,8	2644,7 / 2846,6
BB media	705,3 / 178,0	544,8 / 167,4	684,4 / 218,7
desv.std.	2214,0 / 2329,2	1674,7 / 1762,6	1989,8 / 2101,3
DL media	1186,2 / **42,2**	381,7 / **37,8**	1111,5 / **48,4**
desv.std.	2650,2 / 2918,8	1373,1 / 1432,3	2601,8 / 2840,2
AVL media	3281,7 / 188,6	1821,9 / 182,1	3151,9 / 183,8
desv.std.	3535,9 / 4846,2	2741,4 / 3304,2	3308,6 / 4584,3
HF media	**120,6** / 62,9	**107,2** / 94,3	**119,9** / 94,9
desv.std.	55,5 / 117,7	43,3 / 67,9	52,9 / 91,1
TLSF media	267,8 / 117,6	206,4 / 114,0	221,0 / 113,4
desv.std.	45,0 / 34,8	29,5 / 39,5	93,5 / 23,8

Tabla 5.13: Simulación MEAN GS, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3	Test 4
FF media	**54,7** / 58,1	**57,7** / 56,3	62,1 / **54,8**	100,6 / 59,7
desv.std.	18,2 / 0,0	13,2 / 19,2	12,8 / 32,6	1121,2 / 1128,7
BF media	63,1 / 60,9	63,9 / 57,7	97,9 / 56,9	**65,3** / 61,7
desv.std.	18,8 / 26,2	19,9 / 34,4	851,9 / 860,2	12,2 / 25,4
BB media	140,1 / 109,1	175,0 / 109,2	137,9 / 111,3	129,6 / 104,7
desv.std.	94,9 / 130,8	1058,6 / 1071,8	82,8 / 117,3	89,8 / 118,8
DL media	122,6 / **33,4**	158,0 / **38,5**	**49,8** / 56,1	76,0 / **21,6**
desv.std.	86,8 / 147,5	121,2 / 196,2	35,4 / 25,0	1009,0 / 1015,7
AVL media	341,5 / 187,9	341,5 / 179,3	364,9 / 180,5	321,7 / 110,9
desv.std.	115,7 / 310,6	50,5 / 296,6	873,4 / 934,3	30,1 / 304,8
HF media	99,3 / 94,9	100,7 / 95,9	102,9 / 94,3	106,0 / 95,9
desv.std.	11,7 / 33,7	31,4 / 44,7	7,7 / 43,2	15,8 / 48,6
TLSF media	186,8 / 112,7	166,0 / 114,9	155,6 / 115,6	149,7 / 117,0
desv.std.	78,4 / 170,0	24,8 / 123,3	22,3 / 107,7	12,7 / 95,2

Tabla 5.14: Simulación MEAN Perl, `malloc` y `free` (Ciclos de procesador).

en un corto espacio de tiempo. Segundo, el rango de tamaños generados por el simulador (SC) es, también, muy reducido. Estos dos factores favorecen, por lo tanto, a una gran reutilización de bloques, con lo cual todos los gestores son favorecidos. First-Fit y DLmalloc, obtienen los mejores resultados en estas pruebas.

Por otra parte, en el caso del modelo MEAN para GS (tabla 5.13), se observa un empeoramiento general, salvo en el caso de los gestores Half-Fit y TLSF, los cuales mantienen su estabilidad. Esto se debe sin duda a que, este modelo genera peticiones de tamaños muy dispares. Es decir, el rango de tamaños producidos por el simulador (SC) es muy amplio. Esto entorpece la reutilización de bloques, forzando a los gestores, en general a realizar búsquedas de bloques libres más costosas. En el caso de los gestores Half-Fit y TLSF, dichas búsquedas no son necesarias, con lo cual, los tiempos obtenidos son prácticamente los esperados.

Respecto a la operación `free`, no existen grandes variaciones entre los resultados de la simulación MEAN y los resultados obtenidos por las aplicaciones reales. Esto era lo esperado, ya que, por lo general, la lógica interna de la operación `free` suele ser bastante simple y no necesita la realización de ninguna operación de búsqueda en la estructura de datos del algoritmo.

Por lo tanto, se puede llegar a la conclusión de que el modelo MEAN no consigue reproducir correctamente el patrón de peticiones de las aplicaciones reales, debido a que: o bien el uso de solamente los tres parámetros (HT, SC y IAR) utilizados para modelizar las aplicaciones es insuficiente. O bien el

comportamiento de las aplicaciones no se ajusta a una distribución normal (o uniforme, según la variante del simulador MEAN).

Por lo tanto, este modelo no resulta adecuado para la reproducción del comportamiento respecto a la memoria dinámica de una aplicación. Sin embargo, debido a su facilidad de implementación y, a que a veces, resulta útil disponer de alguna manera de producir peticiones de asignación/liberación de memoria, este modelo sigue en uso [97, 96].

5.3.4.2. Modelo CDF

Las siguientes funciones de probabilidad muestran el modelo CDF de la aplicación Espresso en el caso de la prueba 2.

$$\texttt{HT}(x) = \begin{cases} 0{,}00 & x < 1 \\ 0{,}25 & x < 2 \\ 0{,}29 & x < 4 \\ 0{,}39 & x < 8 \\ 0{,}46 & x < 16 \\ 0{,}53 & x < 32 \\ 0{,}61 & x < 64 \\ 0{,}69 & x < 128 \\ 0{,}79 & x < 256 \\ 0{,}87 & x < 512 \\ 0{,}95 & x < 1K \\ 0{,}97 & x < 2K \\ 1{,}00 & x \geq 128K \end{cases} \quad \texttt{IAR}(x) = \begin{cases} 0{,}00 & x < 1 \\ 0{,}92 & x < 2 \\ 0{,}96 & x < 4 \\ 0{,}98 & x < 8 \\ 1{,}00 & x \geq 32 \end{cases} \quad \texttt{SC}(x) = \begin{cases} 0{,}00 & x < 4 \\ 0{,}01 & x < 8 \\ 0{,}31 & x < 16 \\ 0{,}32 & x < 32 \\ 0{,}84 & x < 64 \\ 0{,}91 & x < 128 \\ 0{,}95 & x < 256 \\ 0{,}96 & x < 512 \\ 1{,}00 & x \geq 1K \end{cases}$$

Modelo CDF. Espresso, test 2.

Los modelos del resto de aplicaciones pueden ser consultados en el apéndice B.1.4.

Cada simulación realizada a partir de este modelo ha sido ejecutada durante 10000 pasos de simulación.

Las tablas 5.15, 5.16, 5.17, 5.18 y 5.19 muestran los resultados obtenidos (medias y desviaciones típicas). Para facilitar su lectura, se ha resaltado en negrita las mejores medias en cada prueba.

Al igual que pasaba en el caso del modelo MEAN, el modelo CDF no consigue reproducir el comportamiento de las aplicaciones originales. En general se observan mejores resultados que los que se observaban en las aplicaciones originales. El gestor First-Fit es, sin duda, el gestor que mejores resultados obtiene, debido, principalmente a la gran reutilización de bloques.

La gran desviación típica obtenida en la mayoría de gestores muestra, también, que todos los gestores funcionan mejor ante un patrón de peticiones no aleatorio. La introducción de aleatoriedad en el modelo introduce variabilidad en los resultados.

Malloc/Free	Test 1	Test 2	Test 3	Test 4
FF media	46,0 / 38,6	106,0 / 40,3	**138,9** / 41,5	328,4 / 55,9
desv.std.	17,3 / 30,5	625,4 / 633,7	841,1 / 852,0	1498,0 / 1533,8
BF media	51,8 / 38,9	114,7 / 40,5	143,7 / 40,8	321,3 / **50,3**
desv.std.	18,2 / 38,8	645,7 / 655,2	828,8 / 840,6	1384,1 / 1421,1
BB media	716,3 / 293,0	1239,3 / 416,3	1338,7 / 244,3	2228,9 / 279,3
desv.std.	5941,0 / 6049,2	6336,7 / 6521,5	5995,9 / 6188,1	1283,4 / 13185,8
DL media	**43,4** / **19,2**	147,8 / **30,6**	176,6 / **34,5**	425,1 / 117,3
desv.std.	99,1 / 106,6	646,1 / 662,7	781,3 / 800,8	2666,0 / 2699,3
AVL media	602,1 / 201,1	531,2 / 239,4	597,1 / 260,0	573,0 / 253,6
desv.std.	2071,1 / 20743,5	1293,2 / 1378,8	1448,9 / 1546,3	1567,6 / 1651,0
HF media	110,9 / 93,6	**101,8** / 101,1	156,9 / 98,3	**116,0** / 110,3
desv.std.	489,1 / 493,1	330,3 / 330,9	4011,4 / 4015,7	35,6 / 50,9
TLSF media	145,5 / 110,9	182,3 / 117,0	175,7 / 116,3	180,7 / 173,0
desv.std.	28,4 / 98,6	83,1 / 162,9	347,4 / 371,8	91,9 / 106,1

Tabla 5.15: Simulación CDF CFRAC, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3	Test 4
FF media	**68,5** / **38,6**	107,4 / 55,7	**104,2** / **39,6**	140,4 / 39,6
desv.std.	452,3 / 456,1	2621,4 / 2625,1	560,7 / 569,6	768,0 / 780,5
BF media	65,9 / **38,6**	**72,9** / **38,0**	104,9 / 40,0	142,7 / **38,7**
desv.std.	272,5 / 277,8	748,3 / 751,5	547,9 / 557,1	780,2 / 793,0
BB media	695,8 / 274,3	710,4 / 285,6	778,5 / 244,3	872,2 / 256,4
desv.std.	5795,3 / 5901,0	5605,2 / 5711,2	6014,4 / 6157,6	6321,6 / 6453,4
DL media	135,9 / 27,1	148,3 / 43,7	270,3 / 41,5	501,2 / 110,0
desv.std.	490,4 / 508,5	477,5 / 498,4	3292,0 / 3306,8	1667,6 / 16699,5
AVL media	384,0 / 212,8	617,0 / 219,5	485,8 / 195,0	618,3 / 235,6
desv.std.	3073,3 / 3091,7	1716,6 / 17185,0	1329,2 / 1403,4	3915,1 / 3960,6
HF media	100,7 / 98,7	141,3 / 98,8	108,5 / 94,4	**113,6** / 94,3
desv.std.	10,4 / 23,0	2822,6 / 2826,7	392,1 / 396,2	542,7 / 546,9
TLSF media	204,7 / 125,2	159,4 / 112,3	185,3 / 111,8	199,0 / 111,8
desv.std.	1815,2 / 1823,5	47,7 / 122,9	89,1 / 172,9	104,8 / 195,4

Tabla 5.16: Simulación CDF Espresso, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3
FF media	211,6 / **38,9**	**57,9** / **38,3**	**59,1** / 42,5
desv.std.	2047,5 / 2060,1	187,8 / 192,9	214,7 / 218,7
BF media	172,0 / 39,7	61,0 / **38,3**	69,2 / 43,5
desv.std.	898,4 / 914,8	196,8 / 202,6	429,6 / 433,3
BB media	836,5 / 254,6	761,2 / 298,2	743,0 / 281,1
desv.std.	5906,4 / 6007,8	5857,7 / 5970,9	5914,7 / 6002,5
DL media	373,9 / 44,7	177,2 / 41,1	173,3 / **36,0**
desv.std.	3363,3 / 3387,1	240,3 / 296,0	263,1 / 313,2
AVL media	641,7 / 214,2	595,9 / 192,8	343,5 / 369,5
desv.std.	1707,7 / 1813,5	18158,5 / 18181,8	525,5 / 0,0
HF media	**103,4** / 99,4	99,0 / 93,7	98,9 / 98,3
desv.std.	19,1 / 34,7	12,8 / 34,8	13,3 / 17,5
TLSF media	213,3 / 111,1	165,4 / 112,6	166,9 / 118,4
desv.std.	120,2 / 218,5	45,3 / 129,5	55,4 / 130,1

Tabla 5.17: Simulación CDF GAWK, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3
FF media	**104,0** / **38,6**	**59,3** / 38,7	**99,8** / **41,3**
desv.std.	563,8 / 572,5	200,7 / 205,8	651,7 / 658,8
BF media	118,1 / 39,0	66,9 / 39,4	101,8 / 38,1
desv.std.	826,1 / 834,3	385,5 / 389,7	628,0 / 635,9
BB media	848,3 / 254,4	730,2 / 283,9	689,0 / 229,7
desv.std.	5795,9 / 5946,7	5707,1 / 5816,2	6005,9 / 6089,5
DL media	246,2 / 41,9	89,8 / **20,6**	234,1 / 76,7
desv.std.	633,6 / 679,0	572,3 / 579,5	578,4 / 620,0
AVL media	648,1 / 189,6	308,1 / 207,2	412,5 / 227,6
desv.std.	1736,9 / 1738,9	237,5 / 329,7	897,6 / 962,5
HF media	107,6 / 100,7	103,0 / 98,4	98,7 / 99,3
desv.std.	339,8 / 342,2	337,5 / 339,3	15,0 / 12,0
TLSF media	176,5 / 112,9	155,4 / 116,8	178,6 / 112,1
desv.std.	360,2 / 385,2	43,4 / 111,6	77,5 / 159,5

Tabla 5.18: Simulación CDF GS, `malloc` y `free` (Ciclos de procesador).

Malloc/Free	Test 1	Test 2	Test 3	Test 4
FF media	**66,8** / **39,1**	**61,8** / **37,7**	**54,8** / 37,7	**47,8** / 38,2
desv.std.	509,3 / 512,6	234,9 / 240,3	108,7 / 115,8	482,5 / 483,9
BF media	75,7 / **39,1**	123,0 / 38,3	58,0 / 38,8	63,0 / 38,2
desv.std.	383,2 / 388,9	3188,9 / 3196,2	101,2 / 110,1	349,8 / 353,7
BB media	795,2 / 244,1	696,2 / 276,7	708,8 / 267,8	882,6 / 279,6
desv.std.	6140,5 / 6261,8	5602,7 / 5684,5	5778,0 / 5838,5	6383,2 / 6489,7
DL media	236,5 / 36,0	123,2 / 44,4	141,9 / **30,1**	138,2 / **28,9**
desv.std.	2561,5 / 2574,2	294,6 / 316,7	405,5 / 428,8	211,7 / 251,4
AVL media	400,1 / 226,8	330,1 / 177,9	309,5 / 192,5	317,0 / 189,2
desv.std.	839,1 / 902,2	433,9 / 516,3	404,0 / 471,5	380,2 / 458,0
HF media	107,5 / 99,9	95,7 / 94,3	102,7 / 94,4	97,7 / 98,4
desv.std.	477,1 / 479,1	337,3 / 338,3	342,8 / 345,4	8,8 / 0,0
TLSF media	190,2 / 111,1	207,5 / 115,9	165,8 / 111,0	173,9 / 110,9
desv.std.	346,0 / 379,2	2845,0 / 2854,8	333,5 / 355,8	666,4 / 680,4

Tabla 5.19: Simulación CDF Perl, `malloc` y `free` (Ciclos de procesador).

Los gestores TLSF y Half-Fit, al contrario del resto, mantienen su estabilidad. Este resultado se debe a su política de búsqueda de bloques, es decir, no realizar búsquedas exhaustivas.

5.4. Conclusiones

Los sistemas de tiempo real requieren determinismo temporal. De todos los gestores estudiados en este capítulo solamente dos han mostrado una respuesta temporal estable e independiente de la entrada recibida. Estos gestores han sido TLSF y Half-Fit. Su respuesta temporal se debe principalmente a su política de búsqueda de bloques libres: no se realiza una búsqueda exhaustiva para encontrar un bloque libre. En su lugar, ambos algoritmos utilizan una estructura de datos de tamaño fijo, donde los bloques libres son almacenados y una función de traducción les permite en tiempo constante realizar la búsqueda.

Half-Fit consigue un tiempo de respuesta de casi la mitad que TLSF. Sin embargo, tal como se verá en el siguiente capítulo, la fragmentación causada por Half-Fit es muy superior a la causada por TLSF, haciéndolo poco apropiado para sistemas con limitaciones de memoria.

Otras contribuciones de este capítulo han sido:

- Todos los gestores de memoria dinámica estudiados tienen un comportamiento temporal muy aceptable (respecto a media y desviación típica)

en aplicaciones con el siguiente patrón: uso de bloques de memoria relativamente pequeños ($\leq$ 256 bytes) y alta reutilización de bloques de memoria.

- Las aplicaciones que realizan peticiones superiores a 512 bytes afectan negativamente al rendimiento de los gestores. Con la excepción del gestor TLSF y la del Half-Fit, cuyos resultados permanecen inalterables.
- Los modelos MEAN y CDF no parecen reproducir el patrón de uso de la memoria de las aplicaciones simuladas, lo cual resulta en un incremento significativo en la media y la desviación típica obtenida en todos los gestores salvo en el caso de los gestores TLSF y Half-Fit, los cuales se mantienen invariables.
- La inclusión de cierta aleatoriedad (como se ha visto en los modelos MEAN y CDF) en el patrón de petición de memoria produce un empeoramiento claro en el funcionamiento de todos los gestores salvo en el de TLSF y el de Half-Fit.
- En todos los gestores de memoria estudiados la operación de liberación (`free`) obtiene mejores resultados que la operación de asignación (`malloc`). Esto se debe a que la operación de asignación conlleva una búsqueda.

Por lo tanto, se puede concluir que TLSF ofrece los resultados para los que fue diseñado, un tiempo de respuesta bajo y determinista. Lo cual le convierte en un gestor de memoria adecuado para sistemas de tiempo real.

Capítulo 6
Comportamiento espacial del gestor TLSF

Este capítulo presenta un estudio, teórico y experimental de la fragmentación producida por el gestor TLSF. En el caso del estudio experimental, los resultados obtenidos han sido comparados con los obtenidos por un conjunto representativo de gestores de memoria dinámica: First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL malloc y Half-Fit.

6.1. Introducción

El principal requerimiento de un sistema de tiempo real es, sin lugar a dudas, el determinismo temporal. Es decir, la corrección del sistema no solo depende de que la lógica del programa sea correcta, sino también del instante temporal en el que los resultados son obtenidos. Sin embargo, ésta no es la única característica requerida en los gestores de memoria para sistemas de tiempo real.

Las aplicaciones de tiempo real, a diferencia del resto de aplicaciones, suelen ser ejecutadas por un largo periodo de tiempo. En el caso de que una aplicación de tiempo real utilizara memoria dinámica, el fracaso de la misma podría deberse a dos situaciones. La primera, el incumplimiento de algún plazo de ejecución. La segunda, al fenómeno conocido como *fragmentación*, el cual fue definido en el capítulo 3.3, que puede causar que el gestor de memoria no pueda satisfacer una petición de asignación de memoria.

Tradicionalmente, existen dos casos de fragmentación, la fragmentación interna 3.3.2 y la fragmentación externa 3.3.1.

La fragmentación interna, causada por motivos de diseño del propio gestor de memoria, se produce cuando ante una petición de r bytes el gestor asigna un tamaño r', tal que $r' > r$.

Por otra parte, la fragmentación externa se produce cuando, ante una petición de tamaño r, el gestor fracasa, a pesar de que el total de memoria libre disponible es superior o igual a r. Esto se debe a que, el gestor de memoria va troceando el montículo para satisfacer cada una de las peticiones, convirtiendo al montículo, tras un tiempo de ejecución, en un conjunto de bloques libres físicamente no contiguos siendo el tamaño de cada uno de ellos inferior a r.

Por lo tanto, cualquier gestor de memoria apto para sistemas de tiempo real debe cumplir dos propiedades básicas, una respuesta temporal determinista y aceptable, y una respuesta espacial predecible.

En el capítulo anterior se demostró que el gestor TLSF presentaba un comportamiento temporal rápido y determinista. El presente capítulo muestra los resultados de un estudio teórico realizado sobre la fragmentación interna y externa producida por el gestor TLSF. Además se presentan los resultados obtenidos por el gestor TLSF y otros gestores de memoria (First-Fit, Best-Fit, Binary-Buddy, DLmalloc, AVL y Half-Fit) en un estudio comparativo con diferentes tipos de cargas reales y sintéticas.

6.2. Estudio teórico

El gestor TLSF presenta cuatro fuentes de fragmentación:

1. Por definición, todo tamaño requerido es redondeado al tamaño representado por la lista inmediatamente superior al tamaño pedido. Por ejemplo, suponiendo $\mathcal{J} = 5$, el tamaño de una petición de 318 bytes será redondeado a 320 bytes, es decir, el tamaño representado por la lista más próxima $(i, j) = (8, 8)$.
2. Tal como se vio en la sección 4.3.3, TLSF asigna un tamaño constante extra a cada bloque para almacenar información para su gestión.
3. TLSF permite la gestión de bloques de diferentes tamaños sin poder realizar una reubicación dinámica de los mismos.
4. El gestor TLSF no realiza búsquedas exhaustivas en su estructura de datos para encontrar el bloque libre más adecuado, en su lugar busca la primera lista no vacía cuyo tamaño representado sea superior o igual al requerido, asignando uno de los bloques almacenados en dicha lista. Esto puede provocar que el gestor falle una asignación a pesar de que exista algún bloque en la estructura de datos que pueda satisfacer la petición. Por ejemplo, suponiendo $\mathcal{J} = 5$ y un único bloque libre de 1055 bytes indexado, por definición, en la lista $(i, j) = (10, 0)$. Ante una petición de 1055 bytes, el gestor comenzará la búsqueda a partir de la lista $(i, j) = (10, 1)$, fracasando por tanto en la asignación a pesar de la existencia del bloque de 1055 bytes.

Mientras que 1 y 2, debido a que obligan la asignación de más memoria de la requerida, causan fragmentación interna, 3 y 4 causan fragmentación externa, ya que pueden provocar el fallo de una asignación a pesar de que exista la suficiente memoria libre para satisfacer dicha petición. Además 4 es un tipo especial de fragmentación externa, bautizada por el autor de esta tesis como *fragmentación estructural*, que solamente se había visto con anterioridad en el gestor de memoria Half-Fit.

A continuación se estudia con un mayor nivel de detalle cada una de estas fragmentaciones y el impacto que tienen en el funcionamiento del gestor TLSF.

6.2.1. Fragmentación interna

Dos son la causas por las cuales el gestor TLSF produce fragmentación interna, el redondeo del tamaño solicitado al tamaño representado por la lista superior más cercana a dicho tamaño y el espacio extra constante asignado por el gestor para almacenar información para la gestión del bloque (cabecera del bloque). A partir de ahora vamos a considerar que el tamaño extra introducido por la cabecera, 4 bytes, es despreciable, quedándonos el redondeo realizado por el gestor como única fuente de fragmentación interna.

Más formalmente, la fragmentación interna (fi) producida por un gestor

de memoria en la asignación de un bloque de memoria se calcula como la diferencia entre el tamaño del bloque asignado r' y el tamaño requerido r, es decir, $\mathrm{fi}(r) = r' - r$.

En el caso del gestor TLSF esta diferencia, $\mathrm{fi}_{\text{TLSF}}$, se calcula como:

$$\mathrm{fi}_{\text{TLSF}}(r) = \overbrace{2^{i(r)} + j(r) \cdot 2^{i(r)-\mathcal{J}}}^{r'} - r$$

Por ejemplo, suponiendo $\mathcal{J} = 5$ y una petición de $r = 5886$ bytes, la lista superior más próxima es $(i,\, j) = (12,\, 14)$, la cual representa el tamaño $r' = 5888$, por lo cual la petición será satisfecha con un bloque de dicho tamaño, creando una fragmentación interna de 2 bytes.

Como se puede observar, la fragmentación interna en el gestor TLSF depende exclusivamente de la distancia, d, existente entre la lista $(i,\, j)$ y la lista siguiente, $(i,\, j+1)$ si $j < \left(2^{\mathcal{J}} - 1\right)$ o $(i+1,\, 0)$ en otro caso, la cual se calcula como:

$$d = \begin{cases} \overbrace{\left(2^{i+1} + 2^{i+1-\mathcal{J}} \cdot 0\right)}^{(i+1,\ 0)} - \overbrace{\left(2^{i} + 2^{i-\mathcal{J}} \cdot \left(2^{\mathcal{J}} - 1\right)\right)}^{(i,\ 2^{\mathcal{J}}-1)} = 2^{i-\mathcal{J}} & \text{si } j = 2^{\mathcal{J}} - 1 \\ \overbrace{\left(2^{i} + 2^{i-\mathcal{J}} \cdot (j+1)\right)}^{(i,\ j+1)} - \overbrace{\left(2^{i} + 2^{i-\mathcal{J}} \cdot j\right)}^{(i,\ j)} = 2^{i-\mathcal{J}} & \text{en otro caso} \end{cases}$$

Es decir,

$$d = 2^{i-\mathcal{J}}$$

Como se puede apreciar, la distancia entre dos listas consecutivas solamente depende del índice de primer nivel i, existiendo la misma distancia entre todas las listas que compartan el mismo valor para i.

La distancia entre dos listas consecutivas determina, entre otras cosas, la fragmentación máxima producible *ante una sola petición* para un primer nivel i como $2^{i-\mathcal{J}} - 1$.

La fragmentación relativa (memoria desperdiciada respecto al total de memoria asignada por el gestor) viene dada por la siguiente formula:

$$\mathrm{fir}_{\text{TLSF}}^{\text{máx}} \leq \frac{1}{1 + 2^{\mathcal{J}}}$$

Una demostración formal de esta función cota superior aparece en el apéndice A.

Esta función indica que, por ejemplo, suponiendo $\mathcal{J} = 5$, ante cualquier tamaño de montículo, la máxima fragmentación interna relativa producible es un 3 % del tamaño del montículo.

6.2.2. Fragmentación externa

El gestor TLSF gestiona diferentes tamaños de bloques lo cual es fuente de fragmentación externa tal como se explicó en la sección 3.3.1.

J.M. Robson demostró en [107] que el peor caso de este tipo de fragmentación, que se produce cuando una petición de memoria de tamaño r fracasa a pesar de que el total de memoria libre existente en el gestor es igual o mayor que r, para la política de mejor ajuste o Best-Fit era, *al menos*, $(M - 4 \cdot n + 11) \cdot (n - 2)$, la cual es orden asintótico de $O(M \cdot n)$. Esta cota asintótica ya había sido demostrada por J.M. Robson en [105] como la peor cota posible de fragmentación externa para un gestor de memoria dinámica.

Por ejemplo, según este resultado teórico, un gestor de memoria con un tamaño máximo de asignación a la aplicación $M = 1$ Mbyte y un tamaño máximo de asignación por petición $n = 10$ Kbytes, que implemente una política de mejor ajuste, necesita un montículo de memoria $\mathcal{H} = 1 \cdot 10240 = 10240$ Mbytes para evitar un fallo por fragmentación externa.

El gestor TLSF implementa una política de buen ajuste o Good-Fit, similar en todos los sentidos a la política de mejor ajuste. Debido a estas similitudes es posible asimilar los resultados obtenidos para la política de mejor ajuste en la de buen ajuste.

Además, debido a que TLSF no realiza una búsqueda exhaustiva para asignar el bloque libre más adecuado ante una petición, una asignación de tamaño r puede fracasar a pesar de que exista un bloque de tamaño r' siendo $r' \geq r$. Por ejemplo, suponiendo $\mathcal{J} = 5$ y un único bloque de memoria libre de 2058 bytes, indexado en la lista (11, 0), una petición de 2050 bytes fracasaría, ya que la lista (11, 1) se encuentra vacía.

Este tipo de fragmentación externa, llamado durante esta tesis como fragmentación estructural, no es exclusiva del gestor TLSF ya que también la sufre Half-Fit, presentada en [90] bajo el nombre *uso incompleto de memoria*, y en general, todo gestor de memoria que no realice una búsqueda exhaustiva para encontrar el bloque libre más adecuado ante una petición.

La fragmentación estructural máxima producible por TLSF se encuentra acotada superiormente por $2^{i(M)-\mathcal{J}} - 1$. Esta cota se deduce a partir del peor caso que produzca un fallo del gestor por fragmentación estructural, es decir, suponiendo un tamaño de bloque máximo asignable M, indexado por la lista con índices (i_M, j_M), el peor caso se da cuando se pide un

bloque de tamaño mínimo r tal que $(M - r)$ sea de tamaño igual al tamaño representado por alguna lista con $(i,\, j) = (i_M,\, j > 0)$ menos uno.

6.3. Análisis experimental

Tal como concluyó J.M. Robson en [107] tras comparar sus resultados teóricos con los resultados obtenidos por J.S. Shore en [119], aunque en teoría existen casos patológicos de fragmentación, en la práctica raramente se dan. Este resultado resalta la importancia de realizar un estudio experimental a la vez que un estudio teórico sobre la fragmentación.

Para llevar a cabo el estudio experimental del TLSF, en la presente sección se han utilizado tanto cargas reales como sintéticas.

Se ha utilizado el mismo conjunto de cargas reales que el utilizado en el estudio temporal: CFRAC, Espresso, GAWK, GS y Perl. Las cuales han sido utilizas a su vez como referencia en estudios de otros investigadores [147, 52, 148]. El apéndice B.1 incluye una descripción detallada de cada una de estas aplicaciones y del patrón de uso de memoria de las mismas.

Para la generación de carga sintética, se ha propuesto un modelo de tareas periódico que incluye los aspectos de petición y liberación de memoria dinámica. A partir de dicho modelo se ha implementado un simulador.

Además del gestor TLSF, también se han tenido en cuenta los gestores First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL malloc y Half-Fit, lo que permite realizar un estudio comparativo de los mismos.

6.3.1. Métricas de fragmentación

Uno de los principales problemas al medir fragmentación es la falta de consenso existente en la literatura. En la sección 3.3.3, se presentaban las cuatro métricas de fragmentación propuestas por M.S. Johnstone y P.R. Wilson en [57], las cuales cuentan actualmente con una buena aceptación. En el presente capítulo para medir fragmentación se ha optado por utilizar tres de estas cuatro métricas, en concreto se ha eliminado la primera ya que no aporta información extra a los resultados del resto de métricas.

6.3.2. Cargas reales

Estudios como los de M.S. Johnstone y P.R. Wilson en [57] o el de A. Bohra y E. Gabber en [17] ponen de manifiesto la importancia del uso de aplicaciones reales (CFRAC, Espresso, GCC, Emacs, HummingBird, Make) para estudiar la fragmentación producida por un conjunto de gestores de

memoria dinámica (First-Fit, Best-Fit, DLmalloc, Binary Buddy, Double Binary Buddy).

En esta sección, lo más adecuado hubiera sido estudiar la respuesta espacial proporcionada por los gestores de memoria considerados con varias aplicaciones de tiempo real. Sin embargo, este tipo de estudio no es posible ya que las aplicaciones de tiempo real existentes evitan el uso de memoria dinámica. Por lo tanto, para realizar este estudio, en la presente sección se ha optado por utilizar un conjunto de aplicaciones estándar.

Las tablas 6.1, 6.2, 6.3, 6.4 y 6.5 muestran los resultados obtenidos, es decir, la fragmentación medida mediante las métricas descritas anteriormente (métricas #2, #3 y #4). Se han resaltado en negrita los mejores resultados en cada prueba.

Gestor	Test 1	Test 2	Test 3	Test 4
FF #2 #3 #4	103,83	102,12	80,79	71,09
BF #2 #3 #4	102,78	99,54	80,05	70,45
BB #2 #3 #4	135,30	132,17	103,95	89,63
DL #2 #3 #4	74,45	71,68	52,27	42,01
AVL #2 #3 #4	102,78	99,54	80,05	70,45
HF #2 #3 #4	96,23	93,12	94,11	82,14
TLSF #2 #3 #4	**47,48**	**44,85**	**32,84**	**23,68**

Tabla 6.1: Fragmentación de CFRAC expresado en tanto por cien.

En algunos casos, las tres métricas utilizadas para calcular la fragmentación han obtenido resultados idénticos, lo cual indica que el punto de máxima utilización de memoria por parte de la aplicación coincide con el instante que más memoria requiere el gestor.

El gestor TLSF es el gestor que mejor comportamiento espacial muestra, produciendo la menor cantidad de fragmentación por regla general, salvo en el caso de la aplicación GS. Estos resultados, mejores incluso que los obtenidos por el gestor Best-Fit, se deben principalmente a dos razones. La primera, TLSF utiliza una política de buen ajuste, la cual tiene un comportamiento próximo al mostrado por la política de mejor ajuste. En la práctica, la política de mejor ajuste suele, normalmente, presentar los mejores resultados de fragmentación [119, 57]. Segundo, TLSF, por motivos de diseño, redondea los tamaños requeridos, consiguiendo una mejor reutilización de los bloques asignados. Por ejemplo, ante la siguiente secuencia de peticiones: b_1 =Asignar(65), b_2 =Asignar(4), Liberar(b_1), b_3 =Asignar(66), el gestor TLSF asignará un bloque de 66 bytes, por lo cual le será posible reutilizarlo para satisfacer la petición de 66 bytes. En el caso del gestor Best-Fit, la primera petición será satisfecha con un bloque de 65 bytes, lo que

Gestor	Test 1 (%)	Test 2 (%)	Test 3 (%)	Test 4 (%)
FF #2	100,46	270,50	104,87	346,44
FF #3	135,69	303,83	105,66	350,45
FF #4	100,14	270,50	108,47	351,13
BF #2 #3 #4	23,71	44,05	23,27	39,41
BB #2	73,12	56,12	48,07	69,36
BB #3	73,12	58,65	54,86	69,34
BB #4	73,12	65,87	55,63	69,36
DL #2	10,41	25,48	10,42	22,53
DL #3	56,55	97,00	46,72	52,92
DL #4	10,67	27,47	10,45	22,55
AVL #2 #3 #4	23,71	44,05	23,27	39,41
HF #2	44,87	100,53	44,77	80,48
HF #3	44,87	106,94	44,77	92,46
HF #4	44,87	104,61	44,77	85,46
TLSF #2	**3,95**	**17,96**	**5,89**	**16,99**
TLSF #3	**3,95**	**40,54**	**5,89**	**16,99**
TLSF #4	**3,95**	**18,83**	**5,89**	**16,99**

Tabla 6.2: Fragmentación Espresso. Tests 1, 2, 3 y 4.

impide reutilizar este bloque para atender la próxima petición de 66 bytes. En el caso de la aplicación GS, la única aplicación que necesita bloques de tamaños relativamente *grandes*, el gestor TLSF obtiene buenos resultados, sin embargo, no los mejores. Esto se debe a la influencia de la fragmentación interna producida por el propio gestor (1,61 %) en estas pruebas.

Cabe destacar también la similitud entre los resultados obtenidos por los gestores Best-Fit y AVL. Debido a que ambos gestores implementan la misma política de asignación de bloques, mejor ajuste (FIFO), implementándola mediante estrategias diferentes, una lista doblemente enlazada y un árbol de búsqueda AVL respectivamente. Esto se traduce a un comportamiento idéntico, es decir, misma fragmentación, pero diferente comportamiento temporal ya que la lista doblemente enlazada tiene una respuesta temporal lineal mientras que la del árbol AVL es logarítmica.

El gestor First-Fit, como era de esperar [119], presenta, en la mayoría de las pruebas unos resultados levemente peores a los gestores que utilizan una política de mejor ajuste.

El gestor DLmalloc utiliza, según el tamaño del bloque a asignar, dos políticas de asignación diferentes. Por una parte, para los bloques pequeños, DLmalloc sigue una política de ajuste exacto. Por otra parte, para los bloques mayores, DLmalloc utiliza una política de mejor ajuste. Esto le permite obtener al gestor DLmalloc resultados buenos, mejores incluso que los ob-

Gestor	Test 1 (%)	Test 2 (%)	Test 3 (%)
FF #2	36,42	70,11	129,74
FF #3	36,42	80,30	130,30
FF #4	36,42	81,46	132,35
BF #2	11,10	15,82	16,63
BF #3	11,11	15,91	17,14
BF #4	11,10	15,82	17,13
BB #2	88,67	78,12	77,30
BB #3	88,69	79,77	79,77
BB #4	88,67	78,12	77,30
DL #2	8,17	12,92	12,57
DL #3	8,17	13,97	13,38
DL #4	8,17	12,92	13,37
AVL #2	11,10	15,82	16,63
AVL #3	11,11	15,91	17,14
AVL #4	11,10	15,82	17,13
HF #2	31,19	50,17	55,38
HF #3	31,19	54,62	54,10
HF #4	31,19	55,87	50,01
TLSF #2	**6,71**	**11,33**	**10,41**
TLSF #3	**6,71**	**12,36**	**11,03**
TLSF #4	**6,71**	**11,33**	**11,02**

Tabla 6.3: Fragmentación GAWK. Tests 1, 2 y 3.

Gestor	Test 1 (%)	Test 2 (%)	Test 3 (%)
FF #2	0,28	0,37	0,27
FF #3	0,28	2,43	0,27
FF #4	0,28	1,39	0,27
BF #2 #3 #4	0,17	0,16	0,10
BB #2	31,45	50,37	44,13
BB #3	31,45	54,11	44,13
BB #4	31,45	50,37	44,13
DL #2 #3 #4	**0,12**	**0,10**	**0,04**
AVL #2 #3 #4	0,17	0,16	0,10
HF #2	10,26	15,35	10,51
HF #3	10,26	18,47	10,51
HF #4	10,26	17,37	10,51
TLSF #2 #3 #4	1,38	1,49	1,61

Tabla 6.4: Fragmentación GS. Tests 1, 2 y 3.

Gestor	Test 1 (%)	Test 2 (%)	Test 3 (%)	Test 4 (%)
FF #2 #3 #4	9,37	22,18	13,75	13,84
BF #2 #3 #4	5,35	19,28	10,34	10,29
BB #2 #3 #4	63,99	54,83	75,52	75,28
DL #2 #3 #4	**3,38**	14,71	5,80	5,65
AVL #2 #3 #4	5,35	19,28	10,34	10,29
HF #2 #3 #4	15,74	32,74	23,76	23,60
TLSF #2 #3 #4	4,30	**13,40**	**5,08**	**4,94**

Tabla 6.5: Fragmentación Perl. Tests 1, 2, 3 y 4.

tenidos por el gestor Best-Fit.

El gestor Binary Buddy, debido a la gran fragmentación interna que genera para satisfacer peticiones que no son potencia de dos, ha obtenido los peores resultados, de dos a tres veces superiores a los obtenidos por el resto de gestores.

Por último, el gestor Half-Fit, el cual a pesar de utilizar una política de buen ajuste para asignar bloques libres, muestra una gran cantidad de fragmentación. Los malos resultados obtenidos por este gestor se deben a la estrategia utilizada para implementar dicha política, ya que, ante una petición de r bytes, Half-Fit busca el primer bloque libre de tamaño igual o superior a r' tal que $r' = 2^{\lceil \log_2(r) \rceil}$. Sin embargo, una vez encontrado un bloque de tamaño igual o mayor que r', dicho bloque es partido para conseguir un bloque de tamaño r. Siendo, por último, este bloque de tamaño r asignado. Eso causa que muchos bloques recién liberados no puedan ser reutilizados ni siguiera para satisfacer el tamaño para el que fueron inicialmente asignados. Por ejemplo, ante la siguiente secuencia, b_1 =Asignar(55), b_2 =Asignar(100), Liberar($b1$), b_3 =Asignar(55), aunque parezca insólito, para satisfacer la última petición de 55 bytes, Half-Fit tendrá que asignar un nuevo bloque libre, en lugar de reutilizar b_1. Por lo tanto, esta estrategia de asignación aunque no produce fragmentación interna, a largo plazo produce una gran cantidad de fragmentación externa.

6.3.3. Modelos sintéticos

El uso de cargas sintéticas para evaluar la fragmentación ha sido una práctica bastante generalizada en la literatura, por ejemplo, J.E. Shore ya estudió en [119] la fragmentación producida por las políticas First-Fit y Best-Fit ante diferentes funciones de probabilidad.

Además, en el contexto de esta tesis, el uso de cargas sintéticas permite estudiar la fragmentación producida por un gestor de memoria dinámica en

un sistema de tiempo real, a pesar de la falta de aplicaciones de tiempo real que utilicen memoria dinámica. Para ello, se ha añadido una extensión al modelo de tareas periódicas para tiempo real para permitir el uso de memoria dinámica por parte de las tareas.

6.3.3.1. Modelo de tareas

Sea $\tau = \{T_1, ..., T_n\}$ un sistema de tareas periódicas. Cada tarea $T_i \in \tau$ tiene los siguientes parámetros $T_i = (c_i, p_i, d_i, g_i, h_i)$. Donde c_i es el peor tiempo de ejecución, p_i es el periodo, d_i es el plazo, g_i es la máxima cantidad de memoria que una tarea T_i puede pedir por periodo y h_i es el máximo tiempo que una tarea mantiene un bloque de memoria antes de liberarlo (*holding time*).

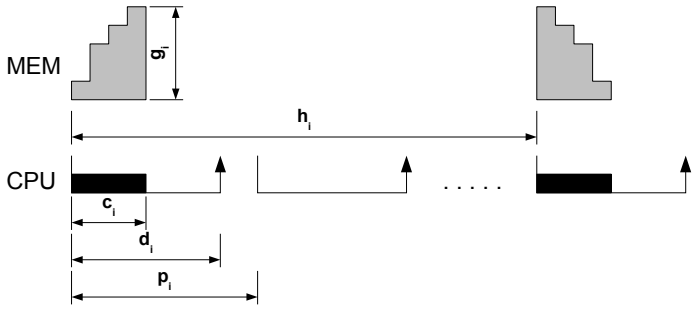


Figura 6.1: Modelo de tareas.

La figura 6.1 muestra el modelo propuesto, el cual es una extensión del modelo de tareas ya existente, donde se permite que cada tarea pueda reservar hasta un máximo de g_i bytes, el tiempo máximo que transcurre entre la asignación y la liberación es de h_i periodos.

Siendo $\mathcal{H}$ el tamaño de montículo, y r el tamaño del bloque de memoria de una petición.

Definición 1. Sea $g_{ij} = \sum_{m=1}^{k} r_{im}$ la cantidad de memoria requerida por una tarea T_i en su j^{esima} activación, la cual es menor o igual a g_i.

Definición 2. La máxima cantidad de memoria viva requerida por la tarea T_i se denota como l_i y se calcula como $l_i = g_i \cdot h_i$.

Definición 3. Sea $\mathcal{L} = \sum_{i=0}^{n} l_i$ la máxima cantidad de memoria requerida por un sistema de tareas periódica τ.

Definición 4. Dado un gestor a, sea $\mathcal{K}^a$ la máxima cantidad de memoria requerida para garantizar $\mathcal{L}$. Incluyendo la cantidad de memoria requerida

por cada una de las tareas más la cantidad de memoria requerida para evitar un fallo debido al fenómeno de la fragmentación.

A partir de este modelo, se puede derivar los parámetros propuestos por la especificación Java de tiempo real [34].

$$\begin{aligned} maxMemoryArea &= g_i \cdot \left(\frac{h_i}{p_i} + 1\right) \\ allocationRate &= \frac{g_i}{c_i} \end{aligned}$$

6.3.3.2. Simulador

Para realizar la simulación del modelo propuesto, se ha implementado un simulador en Linux. Básicamente, este simulador se compone de dos partes bien diferenciadas: primero, un planificador Rate-Monotonic que permite ejecutar tareas periódicas. Segundo, un conjunto de gestores de memoria dinámica, implementados como bibliotecas de carga dinámica (DLL).

Para su ejecución, el simulador obtiene a partir de un fichero de texto la descripción de la carga a ejecutar. Esto es, la descripción del conjunto de tareas que van a ser simuladas. Esta definición incluye el periodo p_i de cada una de las tareas a simular, el plazo d_i, el tiempo de cómputo c_i, el tiempo máximo a transcurrir antes de liberar un bloque de memoria asignado h_i (expresado como una media avg_h_i y una desviación estándar std_h_i), el tamaño máximo de memoria que puede ser asignado por periodo g_i y el rango de tamaños de bloque r_i que cada tarea puede requerir por periodo hasta sumar g_i (expresado como un máximo $r_{\text{máx}}$ y un mínimo r_{min}). Además, este fichero de configuración también especifica el gestor a ser utilizado.

Siguiendo el modelo propuesto, cada tarea puede pedir bloques de memoria de tamaños comprendidos entre $r_{\text{mín}}$ y $r_{\text{máx}}$ (el tamaño de dichos bloques se calcula mediante una distribución de probabilidad uniforme entre $r_{\text{mín}}$ y $r_{\text{máx}}$) hasta sumar un total de g_i, el cual es recargado en el siguiente periodo de la tarea. Por cada bloque asignado, el simulador calcula el instante t a partir del cual el bloque será liberado utilizando para ello una distribución normal *normal(*avg_h_i*,* std_h_i*)*.

6.3.3.3. Experimentos

A continuación se describen los experimentos realizados con el simulador y los resultados obtenidos.

Todos los experimentos realizados se han centrado en estudiar el efecto de h, es decir, el tiempo transcurrido entre la asignación de un bloque y su liberación y la máxima cantidad de memoria asignable por periodo g.

Todas las simulaciones han ejecutado 50000 pasos de simulación y cada prueba ha sido ejecutada 100 veces, variando cada vez la semilla del generador de números aleatorios.

De las métricas propuestas en la sección 6.3.1 para medir la fragmentación causada en los siguientes experimentos, y ante los resultados obtenidos en los experimentos previos donde las métricas #2 y #3 no introducían información adicional, se ha optado por utilizar solamente la métrica #4.

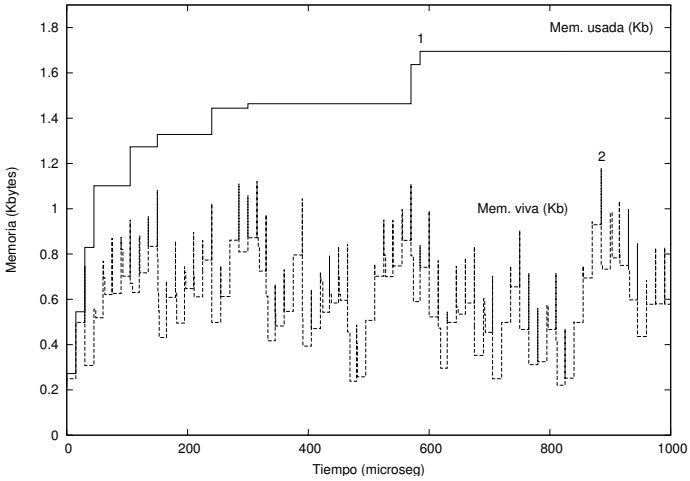


Figura 6.2: Uso de memoria por el gestor Best-Fit con 1 tarea.

La gráfica 6.2 muestra los resultados obtenidos a partir de la primera prueba realizada. Esta gráfica representa la memoria usada por una única tarea $T1$ definida como $T1 = (c_i = 3,\ p_i = 15,\ d_i = 15,\ g_i = 256,\ h_i = 60)$, cuando es servida por el gestor Best-Fit. La gráfica muestra, por una parte, la memoria en uso en el sistema (memoria viva) en cada instante de tiempo. Por otra parte, se representa la memoria requerida por el gestor de memoria para satisfacer cada una de las peticiones de la tarea (memoria usada).

Como se esperaba, las peticiones de memoria se realizan de forma periódica, las subidas en la memoria viva coinciden con el periodo de la tarea (15 u.t.) y las caídas con el fin del h de cada asignación (60 u.t.). Tal como se puede apreciar, una de las ventajas del modelo propuesto es que la cantidad máxima de memoria viva requerida por el total de la carga

$((60/15 + 1) \cdot 256 = 1280)$ bytes puede ser calculada por anticipado.

La fragmentación producida en este caso mediante la métrica #4 propuesta en 6.3.1 es 47,06 % y se calcula como máxima cantidad de memoria usada por el gestor (punto 1) relativa a la máxima cantidad de memoria en uso por la tarea (punto 2).

A continuación se describen una serie de cargas introducidas en el simulador y los resultados obtenidos:

Carga 1: la primera carga consiste en 100 tareas periódicas con periodos armónicos[1] donde los parámetros r_i y g_i de cada tarea han sido generados mediante una distribución $uniforme(16, 4096)$ y $uniforme(8, 2048)$, respectivamente, con las siguientes restricciones: $\forall g_i,\ r_i : g_i > 2 \cdot r_i$. El parámetro h_i de cada tarea ha sido calculado mediante una distribución uniforme, $uniforme(4 \cdot p_i, 12 \cdot p_i)$.

Carga 2: esta segunda carga está compuesta por 200 tareas periódicas con periodos no armónicos. Como en la carga previa, se ha utilizado una distribución uniforme para calcular los parámetros de cada una de las tareas con los siguientes valores: $r_i = uniforme(100, 8192)$, $g_i = uniforme(200, 1638)$, $h_i = uniforme(4 \cdot p_i, 6 \cdot p_i)$. Para generar g_i se utiliza la misma restricción que la utilizada en la carga 1.

Carga 3: la tercera carga esta formada solamente por 50 tareas periódicas con periodos no armónicos. Como en los casos anteriores, se ha utilizado una distribución uniforme para calcular los parámetros de cada una de las tareas que componen la carga. Los valores de dichas distribuciones son: $r_i = uniforme(1024, 10240)$, $g_i = uniforme(2048, 20480)$, $h_i = uniforme(p_i, 16 \cdot p_i)$. Para generar g_i, se ha considerado la restricción ya descrita en la carga 1.

Carga 4: la cuarta carga, similar a la tercera, está compuesta solamente por 50 tareas con periodos armónicos. Al igual que en el resto de cargas, los parámetros de cada una de las tareas han sido calculados mediante una distribución uniforme con los siguientes valores: $r_i = uniforme(8, 256)$, $g_i = uniforme(16, 512)$, $h_i = uniforme(p_i, 3 \cdot p_i)$. Para generar g_i, se ha considerado la restricción ya descrita en la carga 1.

La carga 1 pretende el estudio de la fragmentación causada por un conjunto grande de tareas periódicas, 100 tareas; pidiendo cada una de estas tareas bloques pequeños, de hasta 4096 bytes; estos bloques son liberados tras un largo plazo de tiempo, hasta 12 veces el periodo de una tarea.

La carga 2 permite observar la fragmentación provocada por un conjunto de tareas periódicas aun mayor que en el caso anterior, 200 tareas; requiriendo cada una de estas tareas bloques de memoria de hasta 1638 bytes

[1] Los periodos de las tareas son múltiplos del periodo más pequeño del sistema.

y liberándolos tras un plazo de tiempo no superior a 6 el periodo de una tarea.

Por último, el objetivo de las cargas 3 y 4 es la evaluación de la fragmentación causada por el mismo conjunto de tareas, las cuales, en el caso de la carga 3 requieren bloques de tamaño grande, hasta 20480; liberándolos tras un largo espacio de tiempo, hasta 16 veces el periodo de una tarea. Mientras que en el caso de la carga 4, las tareas requieren bloques de tamaño pequeño, hasta 512 bytes; liberándolos tras un corto espacio de tiempo, hasta 3 veces el periodo de una tarea.

Gestor	FF	BF	BB	DL	AVL	HF	TLSF
C1 (%) Media	100,01	4,64	46,37	5,19	4,64	82,23	**4,33**
Desv.Tip.	4,96	0,61	0,74	1,03	0,61	1,06	**0,55**
C2 (%) Media	85,01	**4,54**	45,00	6,09	**4,54**	75,15	4,99
Desv.Tip.	4,92	0,67	0,98	0,92	0,67	1,52	**0,59**
C3 (%) Media	112,51	**7,01**	48,63	10,43	**7,01**	99,10	7,69
Desv.Tip.	8,53	1,13	1,90	1,54	1,13	2,61	**0,98**
C4 (%) Media	109,71	22,09	69,59	40,63	22,09	73,58	**12,52**
Desv.Tip.	14,04	1,84	15,13	10,33	1,84	5,55	**1,57**

Tabla 6.6: Fragmentación por carga sintética.

La tabla 6.6 sintetiza los resultados obtenidos para cada uno de los gestores considerados ante estas cargas. Para facilitar la interpretación de la tabla se ha resaltado en negrita los mejores resultados de cada prueba.

Tal como se observa en dicha tabla, y como ya se observó en los resultados con cargas reales, el gestor Best-Fit y el gestor AVL han obtenido los mismos resultados. Esto se explica debido a que a pesar de utilizar estrategias diferentes, ambos implementan la misma política de asignación de bloques, comportándose, espacialmente hablando, de forma idéntica.

TLSF es el gestor que mejor resultados obtiene, superando incluso a los gestores Best-Fit y AVL en la mayoría de las pruebas. Esto se debe a los redondeos, por diseño, realizados por el gestor TLSF a los bloques de memoria requeridos, permitiendo su posterior reutilización para satisfacer peticiones de tamaños similares. Por ejemplo, suponiendo $\mathcal{J} = 5$, ante una petición de 130 bytes, TLSF siempre asignará un bloque de 132 bytes, por lo que este bloque podrá ser utilizado posteriormente para satisfacer peticiones de hasta 132 bytes.

Los gestores Best-Fit y AVL muestran los segundos mejores resultados, próximos a los mostrados por el gestor TLSF. Demostrando, tal como se esperaba [119], que la política de mejor ajuste produce muy buenos resultados

en la práctica.

El gestor DLmalloc, debido a su política híbrida, obtiene resultados próximos a los del gestor Best-Fit, sin embargo, no llega a igualarlos.

Por otra parte, tanto el gestor Binary Buddy como el gestor Half-Fit obtienen altas fragmentaciones por diversas razones. En el caso de Binary Buddy esta se obtiene por la excesiva fragmenta interna que se produce cuando el tamaño de bloque requerido no es potencia de dos. Por ejemplo, ante una petición de 4097 bytes, Binary Buddy asignará un bloque de 8192 bytes, generando una fragmentación interna de 4095 bytes (50 %). Half-Fit, por contra, únicamente genera fragmentación externa y lo que el autor de este gestor llama *uso incompleto de la memoria*.

Por último, en estas pruebas, First-Fit es el gestor que peores resultados obtiene, con los mayores factores de fragmentación (112,51 %) y las peores desviaciones típicas. Esto se debe a que el gestor First-Fit suele partir bloques grandes para satisfacer peticiones de tamaño pequeño, impidiendo su uso para futuras peticiones.

El último experimento considerado en esta sección consiste en, dada una carga, ver el impacto en la fragmentación de la variación de los parámetros g y h propuestos en el modelo.

Para este último experimento se han considerado dos tipos de cargas:

1. Esta carga se compone solamente de una tarea ($T1$) con la siguiente definición $T1 = (c_1 = 3,\ p_1 = 15,\ d_1 = 15,\ g_1 = variable,\ h_1 = variable)$.
2. Esta carga se compone de 5 tareas con las siguientes definiciones $T_i = (c_i = 3,\ p_i = [5,\ 10,\ 15,\ 20,\ 25],\ d_i = p_i,\ g_i = variable,\ h_i = 300)$.

Las figuras 6.3 y 6.4 muestran los resultados de simulación en estos experimentos cuando g_i =Eje-X y $h_i = 300$. La figura 6.5 muestra los resultados de simular la primera carga propuesta cuando $g_i = 4096$ y h_i =Eje-X. Para una mejor interpretación de las gráficas, se ha representado el máximo número de bloques asignados al mismo tiempo en cada prueba.

Como se puede observar en todas las figuras, todos los gestores de memoria son más sensibles a las variaciones de g, ya que su incremento causa un decremento en la fragmentación global obtenida, que a las variaciones de h, cuyo incremento parece no afectar apenas a la fragmentación producida.

Ambos parámetros, g y h, se encuentran fuertemente relacionados con la cantidad de memoria asignada en un momento dado y con el número de bloques asignados (memoria viva). Un incremento en alguno de los mismos tiene un efecto similar, un incremento de la cantidad de memoria viva. Sin embargo, existe una gran diferencia en como cada uno de estos parámetros afecta al gestor para conseguir este mismo efecto. Mientras que el incremento del parámetro g causa un mayor porcentaje de asignaciones, el incremento de h causa un decremento en el número de bloques liberados. El hecho de

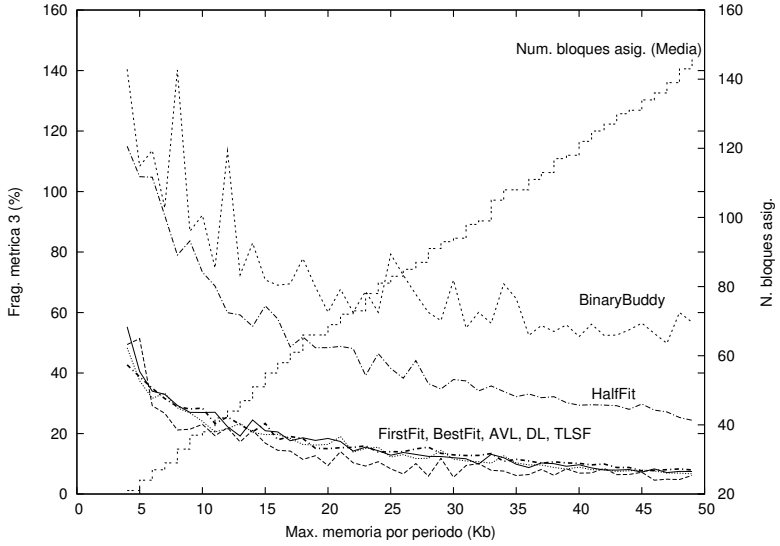


Figura 6.3: Impacto de g con 1 tarea.

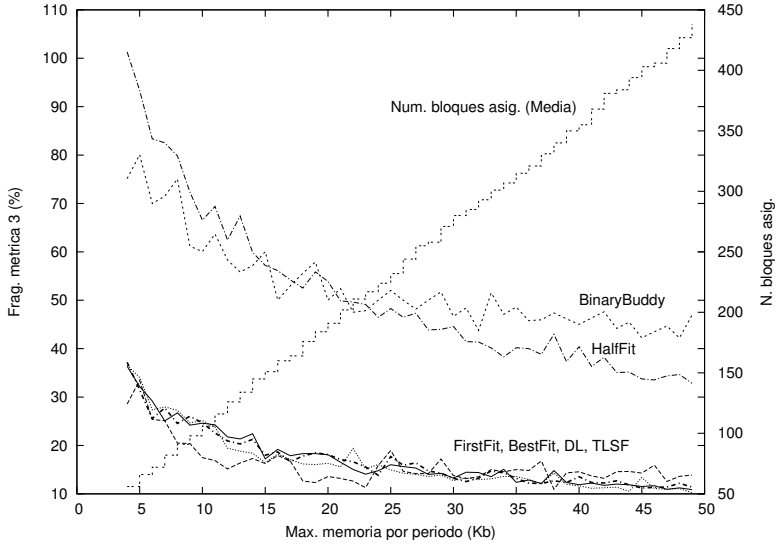


Figura 6.4: Impacto de g con 5 tareas.

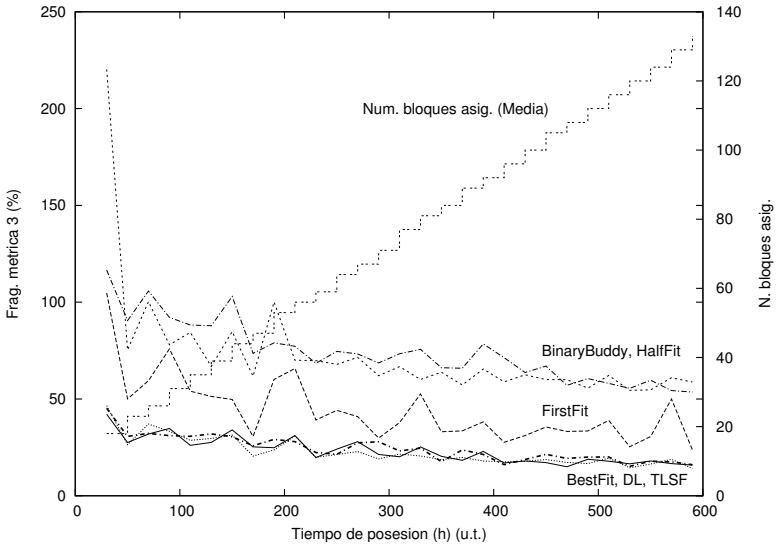


Figura 6.5: Impacto de h con 1 tarea.

que más cantidad de memoria sea asignada, con un h bajo, permite incrementar la reutilización de los bloques, consiguiendo que la fragmentación relativa disminuya (no así la fragmentación absoluta, que con seguridad aumentará). Por otra parte, el hecho de que cada bloque permanezca más tiempo asignado, con el mismo número de asignaciones, no varía en absoluto el porcentaje de reutilización de memoria, permaneciendo la fragmentación relativa sin grandes cambios.

Por último, como era de esperar, TLSF y Best-Fit son los gestores que mejores resultados obtienen en estos experimentos.

6.4. Conclusiones

El principal requerimiento de una aplicación de tiempo real por definición es el determinismo temporal. Sin embargo, cuando se considera el uso de memoria dinámica en los sistemas de tiempo real, el determinismo temporal del gestor de memoria dinámica no es suficiente, ya que, existe otro factor que puede provocar el fallo de la aplicación, la fragmentación.

En este capítulo se ha estudiado la fragmentación, tanto a nivel teórico como práctico causada por el gestor TLSF. Además se ha realizado una

comparativa de los resultados del gestor TLSF a nivel práctico con los resultados mostrados por los gestores First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL y Half-Fit.

Respecto al análisis de fragmentación teórico realizado, la fragmentación interna relativa causada por el gestor TLSF se encuentra acotada superiormente por la función:

$$\mathrm{fir}_{\text{TLSF}}^{\text{máx}} \leq \frac{1}{1+2^{\mathcal{J}}}$$

La fragmentación externa, como ya demostró J.M. Robson en [107], se encuentra acotada por la función:

$$\mathrm{fe}_{\text{TLSF}} \leq M \cdot m$$

Siendo esta la misma cota existente para todos los algoritmo que implementen una política de asignación de bloques de mejor ajuste o buen ajuste.

Respecto al análisis de fragmentación experimental, la situación ideal sería haber utilizado aplicaciones de tiempo real ya existentes para estudiar la fragmentación producida por la misma con cada gestor de memoria considerado en el estudio. Sin embargo, debido a que las aplicaciones de tiempo real existentes no utilizan memoria dinámica, en este capítulo se ha optado por, primero, realizar este estudio con aplicaciones estándar (CFRAC, Espresso, GAWK, GS y Perl). Segundo, se ha propuesto un modelo basado en el modelo de tareas periódicas, donde, bajo ciertas restricciones, cada tarea puede requerir y liberar memoria dinámica. Posteriormente se ha implementado un simulador de dicho modelo y se ha medido la fragmentación causada.

Las conclusiones obtenidas a partir de este estudio comparativo práctico pueden ser resumidas como:

- Aunque teóricamente existen casos patológicos de fragmentación, en la práctica, estos casos raramente se dan. Por ejemplo, en el caso del gestor Best-Fit, la máxima fragmentación producible, teóricamente, es de hasta m veces, donde m es el tamaño de bloque más grande asignable por el gestor, el tamaño $\mathcal{H}$ del montículo. En la práctica, en ningún caso del gestor Best-Fit se ha superado el doble de $\mathcal{H}$.
- El gestor TLSF ha obtenido, en la mayoría de los experimentos, los mejores resultados seguido por los gestores Best-Fit y AVL.
- Los otros dos gestores considerados de tiempo real, Binary Buddy y Half-Fit, provocan una gran fragmentación por razones diferentes. Binary Buddy produce una gran fragmentación interna cuando asigna

tamaños que no son potencia de dos. Half-Fit, por el contrario, no produce fragmentación interna, sin embargo por su estrategia de búsqueda de bloques libres produce una gran fragmentación externa.

- Los gestores Best-Fit y AVL producen, prácticamente, la misma fragmentación, esto se debe a que ambos implementan la política de mejor ajuste para la búsqueda de bloques libres, aun utilizando diferentes estrategias para implementarla.
- Según el modelo de tareas con memoria dinámica propuesto en este capítulo, el aumento del tiempo de posesión h de los bloques de memoria asignados o el aumento de la cantidad de memoria g que una tarea puede pedir por periodo presentan el mismo efecto: un aumento en la cantidad de memoria utilizada por el sistema. Sin embargo, el efecto en la fragmentación no es el mismo. El aumento de h apenas tiene efecto alguno en la fragmentación relativa. El aumento de g produce una disminución de la misma.

Para concluir, la fragmentación es una de las causas por las cuales se ha evitado el uso de memoria dinámica en los sistemas de tiempo real.

Sin embargo, en este capítulo se muestra que existen, en teoría, cotas superiores de fragmentación. Y que, además, aunque las cotas existentes puedan ser un tanto desproporcionadas (hasta n veces el tamaño $\mathcal{H}$ del montículo), estos casos, en la práctica, raramente se dan. Por ejemplo en el caso del gestor TLSF, en ningún experimento práctico realizado se ha alcanzado una cota superior a 1,5 veces el tamaño del montículo. En el caso de los gestores Binary Buddy y Half-Fit, los máximos resultados alcanzados no han superado en ningún caso en 2,2 veces dicho tamaño.

Por lo tanto, aunque sería necesario un análisis más profundo del fenómeno de la fragmentación, sobre todo de la fragmentación externa, todos los indicios indican que esta no debería plantear ningún problema grave en el futuro para utilizar memoria dinámica en los sistemas de tiempo real.

Capítulo 7
Conclusiones y líneas de trabajo futuro

Este capítulo presenta las conclusiones generales obtenidas en esta tesis así como las líneas de trabajo futuro.

7.1. Conclusiones generales

El principal objetivo en esta tesis ha consistido en permitir el uso de memoria dinámica en sistemas de tiempo real. Por lo tanto, las principales aportaciones de esta tesis consisten en, primero, un profundo estudio del estado del arte de la gestión de memoria dinámica. Segundo, un nuevo gestor de memoria dinámica, el gestor TLSF, que cumple los requisitos temporales necesarios para atender a sistemas de tiempo real. Tercero un estudio comparativo del tiempo de respuesta y de la fragmentación, a nivel teórico y experimental, del gestor TLSF con respecto a un conjunto representativo de gestores. Cuarto, se amplía el modelo de tareas periódicas de tiempo real para incluir la gestión de memoria dinámica.

A continuación se explica con mayor nivel de detalle cada una de las aportaciones realizadas.

Estado del arte de la gestión de memoria dinámica

La gestión de memoria dinámica ha sido un tema ampliamente estudiado durante muchos años, produciéndose una gran cantidad de resultados. Resulta curioso que no todos los resultados más relevantes son conocidos por todos los investigadores. Por ejemplo, los importantes estudios sobre fragmentación realizados por J.M. Robson [105, 106, 107, 108] han sido completamente ignorados en el trabajo de M.S. Johnstone et al. [57], y en el de A. Bohra et al. [17].

La primera contribución de esta tesis consiste en un estudio del estado del arte sobre memoria dinámica. Este estudio se ha centrado, primero, en los resultados existentes sobre el problema de la fragmentación, y segundo, en los gestores de memoria dinámica existentes y en el modelo operacional de estos. Además, se estudia en profundidad seis gestores de memoria: First-Fit, Best-Fit, Binary Buddy, Doug Lea's malloc, AVL malloc y Half-Fit.

Este estudio del estado del arte sobre memoria dinámica sienta unas buenas bases para estudiar el posible uso de memoria dinámica en sistemas de tiempo real.

El gestor TLSF

Hasta el momento, dos son las razones argumentadas para rechazar el uso de la memoria dinámica en sistemas de tiempo real. Por un parte, los gestores de memoria dinámica existentes suelen presentar una respuesta temporal no predecible, lo cual los hace inadecuados para los sistemas de tiempo real. Por otra parte, el problema de la fragmentación puede hacer que la aplicación de tiempo real se quede sin memoria y fracase.

La segunda contribución de esta tesis ha consistido en un nuevo gestor

de memoria dinámica, el gestor Two-Level Segregated Fit (TLSF), específicamente diseñado para ser utilizado en sistemas de tiempo real. Este gestor de memoria dinámica muestra, en primer lugar, un tiempo de respuesta rápido y determinista y, en segundo lugar, una fragmentación baja.

La respuesta temporal constante $O(1)$ obtenida por el gestor TLSF se debe a que no se realizan búsquedas exhaustivas para encontrar bloques libres. En lugar de ello, TLSF utiliza mapas de bits de tamaño constante y una función de traducción.

Estudio comparativo del tiempo de respuesta y la fragmentación

En los capítulos 5 y 6 se realizan dos estudios del gestor TLSF, uno teórico y otro experimental. Además, se realiza una comparación de los resultados obtenidos con los resultados del resto de gestores considerados: First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL y Half-Fit.

Las principales conclusiones obtenidas a partir de estos estudios son:

- Tal como concluyó J.M. Robson en [107], en teoría, existen casos de fragmentación catastróficos. Sin embargo, en la práctica, estos casos no se suelen dar. De hecho, en ningún caso de fragmentación estudiado, se ha obtenido una fragmentación superior a 2,5 veces el total de la memoria viva existente.
- Hasta ahora existía la falsa creencia de que un tiempo de respuesta rápido y una baja fragmentación eran conceptos incompatibles.
  En estos estudios se ha comprobado que, justamente los gestores que mejor tiempos de respuesta han obtenido (TLSF, DLmalloc y AVL malloc) son los gestores que, por termino medio, menos fragmentación han causado.
- Tal como se esperaba, TLSF ha obtenido unos tiempos de respuesta bajos y muy estables, casi superiores al del resto de gestores en todas las pruebas. Además, este gestor ha causado los resultados más bajos de fragmentación en casi todas las pruebas realizadas. Estas características lo convierten en un gestor bastante apropiado para ser utilizado en sistemas de tiempo real o en cualquier tipo de sistemas que requieran un gestor de memoria dinámica con tiempo de respuesta muy bajo y una fragmentación, también, muy baja.
- Ambos gestores, Half-Fit y Binary Buddy, han obtenido muy buenos tiempos de respuesta, bajos y predecibles. Sin embargo, ambos han mostrado una alta fragmentación, externa e interna respectivamente, en todas las pruebas realizadas, lo cual desaconseja su uso para sistemas de tiempo real.
- El gestor AVL ha mostrado unos resultados temporales muy estables aunque más altos que los mostrados por otros gestores como TLSF,

Half-Fit, DLmalloc y Binary Buddy. Los resultados de fragmentación han sido, además, bastante buenos, lo cual lo convierte en una alternativa de gestor de memoria apto para ser utilizado en sistemas de tiempo real.

- Tal como concluyó B. Zorn en [148], ni el modelo MEAN ni el modelo CDF consiguen reproducir correctamente el comportamiento de una aplicación real. Sin embargo, ambos resultan útiles si lo que se pretende es conseguir una secuencia de operaciones de asignación y liberación para estudiar a los gestores de memoria dinámica.

Modelo de tareas con gestión de memoria dinámica

Ninguno de los modelos de tareas existentes contemplan el uso de la gestión de memoria dinámica. En el capítulo 6 se ha extendido el modelo de tareas periódicas para tiempo real con la incorporación del uso de memoria dinámica por parte de las tareas del sistema.

En este modelo, cada tarea T_i existente en el sistema se define por la tupla $T_i = (c_i, p_i, d_i, g_i, h_i)$. Donde c_i es el peor tiempo de ejecución, p_i es el periodo, d_i es el plazo, g_i es la máxima cantidad de memoria que una tarea T_i puede pedir por periodo y h_i es el máximo tiempo que una tarea mantiene un bloque de memoria antes de liberarlo (tiempo de posesión).

A partir de este modelo, en el capítulo 6, se ha estudiado la fragmentación producida por cada uno de los gestores de memoria considerados (First-Fit, Best-Fit, Binary Buddy, DLmalloc, AVL, Half-Fit y TLSF), llegando a las siguientes conclusiones:

- El uso del modelo propuesto da como resultado un uso periódico de la memoria dinámica. Esto podría permitir, en el futuro, deducir la cantidad de memoria necesaria para el sistema así como el total de fragmentación que va a ser causada por un gestor de memoria determinado.
- Ambos parámetros pertenecientes al modelo, h_i y g_i, tienen un gran impacto en la cantidad total de memoria utilizada por el sistema. Un incremento de h_i causa un aumento de la vida de los bloques asignados mientras que un incremento de g_i causa la asignación de un mayor número de bloques. Sin embargo, su impacto en la fragmentación relativa causada es diferente, mientras que el incremento de h_i no modifica la fragmentación causada, el incremento de g_i decrementa la fragmentación relativa total. Según lo observado, los mejores resultados se obtienen con h_i bajos y g_i altos, ya que esto permite una mejor reutilización de los bloques de memoria libres.

7.2. Líneas de trabajo futuro

Muchas son las líneas que quedan abiertas a partir de los resultados de la presente tesis.

Ajustar la función cota superior de fragmentación para la política de mejor ajuste

J.M. Robson concluyó en [107] que la fragmentación del escenario de peor caso en la política de mejora ajuste es *al menos* $(M - 4 \cdot m + 11) \cdot (m - 2)$, lo cual es orden asintótico de $O(M \cdot m)$.

Una cota superior más ajustada repercutiría positivamente en las cotas superiores de fragmentación de una gran cantidad de gestores existentes, más concretamente en todos aquellos gestores que implementan una política de mejor ajuste.

Estudio de la gestión de memoria dinámica implícita

La gestión de la memoria dinámica puede realizarse de dos formas diferentes. La gestión explícita o manual, donde la propia aplicación es la responsable de realizar las peticiones de memoria y su liberación posterior. Y la gestión implícita o automática, donde la aplicación solamente se encarga de pedir memoria, siendo el propio gestor de memoria el encargado de liberarla cuando detecte que ya no se requiere la memoria asignada.

La presente tesis se ha centrado exclusivamente en la gestión de memoria dinámica explícita. Sin embargo, la gestión de memoria implícita presenta aun mayores retos. Por un lado, el problema de la fragmentación sigue presente, por otro lado. aparece el problema de la *recolección de memoria* o *garbage collection*, el cual consiste en, primero, decidir que bloques de memoria pueden ser liberados. Segundo, decidir cuando estos bloques deben ser liberados.

El problema de la recolección de memoria dificulta aun más su adopción en sistemas de tiempo real.

Gestión de memoria implícita y el gestor TLSF

Al igual que en la gestión de memoria explícita, en la gestión implícita se requiere usar un gestor para asignar y liberar la memoria libre disponible.

Modificar el gestor TLSF para que permita la gestión automática puede tener un gran interés, ya que permite a la aplicación ignorar la liberación de la memoria.

Estudiar el modelo de tareas con memoria dinámica propuesto

En el capítulo 6 se propone un modelo de tareas periódicas para sistemas de tiempo real que incluye memoria dinámica. En dicho capítulo este modelo es utilizado para evaluar la fragmentación del gestor TLSF, sin embargo, el propio modelo no es evaluado.

La evaluación de la validez del modelo propuesto es interesante ya que actualmente ninguno de los modelos de tareas existentes contemplan el uso de memoria dinámica.

Apéndice A
Demostración cota fragmentación interna

En el presente apéndice se demuestra formalmente la función cota superior para la fragmentación interna producida por el gestor TLSF.

A.1. Definiciones

Antes de entrar en el cuerpo de la demostración, a continuación se presentan una serie de propiedades y definiciones que luego se emplearán para demostrar la función cota superior de la fragmentación interna en el gestor TLSF.

Definición 1. Dada una petición de tamaño r, la fragmentación interna en dicha petición, $\mathrm{fi}(r)$, se define como la diferencia entre el tamaño asignado r' y el tamaño pedido r. Es decir, $\mathrm{fi}(r) = r' - r$.

El resultado de esta operación no puede ser nunca negativo, debido a que esto significa un error en la implementación del gestor. Por otra parte, un resultado igual a cero significa que no se ha producido fragmentación interna en dicha petición.

En el caso del gestor TLSF, la fragmentación interna producida en una asignación de tamaño r, $\mathrm{fi}_{\text{TLSF}}(r)$, se calcula como:

$$\mathrm{fi}_{\text{TLSF}}(r) = \overbrace{2^{i(r)} + j(r) \cdot 2^{i(r)-\mathcal{J}}}^{r'} - r$$

Donde r' es el tamaño representado por la lista superior más cercana al tamaño requerido.

Por ejemplo, suponiendo $\mathcal{J} = 5$, una petición de $r = 1120$ bytes causa una fragmentación interna igual a 0, ya que la lista más cercana es $(i(r)\ ,\ j(r)) = (10\ ,\ 3)$, lista que contiene bloques en el rango $[2^{10} + 3 \cdot 2^{10-5}\ ,\ 2^{10}+4\cdot 2^{10-5}[= [1120\ ,\ 1152[$, y asigna bloques de tamaño $r' = 1120$. En cambio, una petición de $r = 1121$ es atendida por la lista (10, 4), la cual asigna bloques de tamaño $r' = 1152$, causando una fragmentación interna de 31 bytes.

Definición 2. En el gestor TLSF, la distancia entre cualquier lista $(i,\ j)$ y la lista siguiente, $(i,\ j+1)$ si $j < \left(2^{\mathcal{J}} - 1\right)$ o $(i+1,\ 0)$ en otro caso, se calcula como:

$$d = \begin{cases} \overbrace{\left(2^{i+1} + 2^{i+1-\mathcal{J}} \cdot 0\right)}^{(i+1,\ 0)} - \overbrace{\left(2^{i} + 2^{i-\mathcal{J}} \cdot \left(2^{\mathcal{J}} - 1\right)\right)}^{(i,\ 2^{\mathcal{J}}-1)} = 2^{i-\mathcal{J}} & \text{si } j = 2^{\mathcal{J}} - 1 \\ \overbrace{\left(2^{i} + 2^{i-\mathcal{J}} \cdot (j+1)\right)}^{(i,\ j+1)} - \overbrace{\left(2^{i} + 2^{i-\mathcal{J}} \cdot j\right)}^{(i,\ j)} = 2^{i-\mathcal{J}} & \text{en otro caso} \end{cases}$$

Es decir,

$$d = 2^{i-\mathcal{J}}$$

Por ejemplo, suponiendo $\mathcal{J} = 5$, la distancia entre la lista (11, 3), la cual asigna bloques de 2240 bytes, y la lista (11, 4), que asigna bloques de 2304 bytes, es de 64 bytes.

Corolario 1. La distancia entre dos listas consecutivas solamente depende del índice de primer nivel i y del índice máximo de segundo nivel $2^{\mathcal{J}}$, siendo totalmente independiente del índice de segundo nivel j.

Por ejemplo, suponiendo $\mathcal{J} = 5$, la distancia entre cada una de las lista con $i = 11$ y la siguiente es de 64 bytes.

Corolario 2. Dado un índice de primer nivel i se causa la máxima fragmentación interna en dicho nivel cuando se pide un bloque de tamaño igual al representado por alguna lista de dicho nivel más uno. Dicha fragmentación se calcula como:

$$\text{fi}(r) = 2^{i(r)-\mathcal{J}} - 1 \tag{A.1}$$

Por ejemplo, suponiendo $\mathcal{J} = 5$, la máxima fragmentación interna que puede ser generada por las listas pertenecientes a $i = 11$ es de 63 bytes, y se da ante cualquiera de las siguientes peticiones: 2049, 2113, 2177, 2241, ..., 4033.

A.2. Función cota superior fragmentación interna

Teorema 1. La función cota superior de la fragmentación interna relativa producida por el gestor TLSF es:

$$\text{fir}_{\text{TLSF}}^{\text{máx}} \leq \frac{1}{1 + 2^{\mathcal{J}}}$$

Demostración. La máxima fragmentación provocada por un bloque de tamaño r viene dada por expresión A.1. Puesto que para todos los bloques cuyo tamaño esté en el rango del primer índice, se obtiene la misma fragmentación, la fragmentación relativa será máxima cuando el bloque sea el menor de ese rango. Esto es, cuando r sea de la forma $2^z + 1$, siendo z un entero positivo. El tamaño de bloque asignado por TLSF para este tipo de peticiones es $2^z + 2^{z-\mathcal{J}}$.

Supongamos que seguimos el siguiente proceso de petición de memoria (posteriormente veremos que los resultados son válidos para cualquier secuencia de peticiones):

1. Sea $\mathcal{H}_k = \mathcal{H}$.
2. Pedimos un bloque r_0 tal que $r_0 = 2^{i(\mathcal{H}_k)} + 1$;

3. TLSF asignará un bloque de tamaño $2^{i(\mathcal{H}_k)} + 2^{i(\mathcal{H}_k)-\mathcal{J}}$. La memoria restante la llamaremos $\mathcal{H}_{k+1}$ y es igual a $\mathcal{H}_k - \left(2^{i(\mathcal{H}_k)} + 2^{i(\mathcal{H}_k)-\mathcal{J}}\right)$.
4. Repetir puntos 2 y 3 con el nuevo valor de $\mathcal{H}_{i+1}$ hasta que la memoria remanente sea cero ($\mathcal{H}_{k+1} = 0$).

Al finalizar esta secuencia de peticiones hemos consumido toda la memoria del montículo, o lo que es lo mismo, la suma de toda la memoria asignada es igual al tamaño inicial del montículo. Si definimos W como el conjunto de los n valores que ha tomado $\mathcal{H}_k$, entonces:

$$\mathcal{H} = \sum_{r \in W} \left(2^{i(r)} + 2^{i(r)-\mathcal{J}}\right) \tag{A.2}$$

Igualmente, a partir del conjunto W podemos calcular la cantidad de fragmentación interna como:

$$\sum_{r \in W} \left(2^{i(r)-\mathcal{J}} - 1\right) \tag{A.3}$$

La fragmentación relativa es el cociente entre la memoria desperdiciada en fragmentación interna y la memoria total utilizada:

$$\mathrm{fir}^{\text{máx}}_{\text{TLSF}}(\mathcal{H}) = \frac{\displaystyle\sum_{r \in W} \left(2^{i(r)-\mathcal{J}} - 1\right)}{\displaystyle\sum_{r \in W} \left(2^{i(r)} + 2^{i(r)-\mathcal{J}}\right)} \tag{A.4}$$

Considerando que n es el cardinal del conjunto W y separando los términos de los sumatorios, la ecuación A.4 se puede reescribir como:

$$\mathrm{fir}^{\text{máx}}_{\text{TLSF}}(\mathcal{H}) = \frac{2^{-\mathcal{J}} \cdot \displaystyle\sum_{r \in W} \left(2^{i(r)}\right) - n}{\displaystyle\sum_{r \in W} \left(2^{i(r)}\right) + 2^{-\mathcal{J}} \cdot \sum_{r \in W} \left(2^{i(r)}\right)}$$

Dividiendo numerador y denominador por la expresión no nula $\sum_{r \in W} \left(2^{i(r)}\right)$, obtenemos:

$$\mathrm{fir}^{\text{máx}}_{\text{TLSF}}(\mathcal{H}) = \frac{2^{-\mathcal{J}} - \left(\dfrac{n}{\sum_{r \in W} \left(2^{i(r)}\right)}\right)}{1 + 2^{-\mathcal{J}}}$$

Finalmente, eliminando el termino positivo $n / \sum_{r \in W} 2^{i(r)}$ y dividiendo por la constante $2^{-\mathcal{J}}$, obtenemos la desigualdad que queríamos demostrar:

$$\mathrm{fir}_{\text{TLSF}}^{\text{máx}} \leq \frac{1}{1+2^{\mathcal{J}}} \tag{A.5}$$

Hasta ahora se ha supuesto que todo bloque del conjunto W produce la máxima fragmentación y es el más pequeño del primer nivel al que corresponde. Si consideramos que W es un conjunto arbitrario de peticiones tales que toda la memoria disponible es asignada, entonces, la expresión A.2 queda como sigue:

$$\mathcal{H} = \sum_{r \in W} \left(2^{i(r)} + k \cdot 2^{i(r)-\mathcal{J}} \right) \tag{A.6}$$

Donde k pertenece al rango $[0, 2^{\mathcal{J}}]$.

La fragmentación producida por el conjunto W será menor o igual al obtenido en la expresión A.3.

k sólo será igual a 0 cuando el bloque pedido sea exactamente de la forma $r = 2^{i(r)}$. En este caso, la fragmentación es cero ya que el bloque pedido coincide con la cabeza de una lista, con lo que los bloques de la forma $r = 2^{i(r)}$ pueden ser eliminados del conjunto W en la expresión A.4 al no aportar sumandos al numerador, quedado el conjunto de valores de k en el rango $[1, 2^{\mathcal{J}}]$. Por tanto, la expresión A.4 queda como sigue:

$$\mathrm{fir}_{\text{TLSF}}^{\text{máx}}(\mathcal{H}) \leq \frac{\displaystyle\sum_{r \in W} \left(2^{i(r)-\mathcal{J}} - 1 \right)}{\displaystyle\sum_{r \in W} \left(2^{i(r)} + k_{\in [1,2^{\mathcal{J}}]} \cdot 2^{i(r)-\mathcal{J}} \right)} \leq \frac{\displaystyle\sum_{r \in W} \left(2^{i(r)-\mathcal{J}} - 1 \right)}{\displaystyle\sum_{r \in W} \left(2^{i(r)} + 2^{i(r)-\mathcal{J}} \right)}$$

A partir de esta expresión se obtiene la misma cota que la obtenida en A.5.

□

Apéndice B
Análisis experimental: cargas

En los capítulos 5 y 6 se realiza un estudio experimental, temporal y espacial, de un conjunto representativo de gestores de memoria, incluyendo al gestor TLSF. La carga usada en dichas pruebas está compuesta por las aplicaciones CFRAC, Espresso, GAWK, GS y Perl. Este capítulo contiene una descripción y un estudio de cada una de estas aplicaciones, incluyendo las funciones de distribución pertenecientes al modelo CDF propuesto por B. Zorn en [148].

B.1. Aplicaciones estudiadas

Cinco han sido las aplicaciones utilizadas en los estudios experimentales realizados: CFRAC, Espresso, GAWK, GS y Perl. Esta selección se ha realizado con dos objetivos en mente, que todas las aplicaciones fueran libremente accesibles (estando todas las aplicaciones seleccionadas bajo la licencia GPL [42]) y que ya hubieran sido utilizadas previamente en estudios sobre memoria dinámica [140, 57, 148].

B.1.1. Descripción

CFRAC: el programa CFRAC permite factorizar números enteros a través del método de fraccionar dichos enteros continuamente. A continuación se describen las entradas utilizadas para realizar las diferentes pruebas con este programa.

- Test 1: factorización de 1000000001930000000057
- Test 2: factorización de 327905606740421458831903.
- Test 3: factorización de 417576463441248601459380302877l.
- Test 4: factorización de 41757646344123832613190542166099121.

Espresso: el programa Espresso, del cual se ha utilizado la versión 2.3, es un programa de optimización lógica. Es decir, como entrada acepta un circuito combinacional y como resultado se obtiene dicho circuito optimizado. Los tests utilizados para su evaluación han consistido en:

- Test 1: se optimiza un circuito combinacional con 7 entradas y 10 salidas.
- Test 2: se optimiza un circuito combinacional con 8 entradas y 8 salidas.
- Test 3: se optimiza un circuito combinacional con 24 entradas y 109 salidas.
- Test 4: se optimiza un circuito combinacional con 16 entradas y 40 salidas.

GAWK: el programa Gnu Awk, del cual se ha utilizado la versión 3.1.3, es un intérprete para el lenguaje AWK, el cual es utilizado para realizar informes y extraer información. Los tests utilizados han sido:

- Test 1: procesamiento de un texto pequeño para crear un checksum.
- Test 2: ajuste de las líneas que conforman un texto según las opciones dadas, utilizando un texto pequeño.
- Test 3: igual que el test anterior, esto es ajuste de las líneas que conforman un texto según las opciones dadas, utilizando un texto grande.

GS: GhostScript, versión 7.07.1, es un intérprete libre para el lenguaje de descripción de formato de páginas Postscript. Los tests utilizados han sido:

- Test 1: procesamiento de una única página que contiene dos gráficas.
- Test 2: procesamiento de la guía de usuario de la biblioteca C++ de GNU.
- Test 3: procesamiento del manual del lenguaje SELF, desarrollado en la universidad de Stanford.

Perl: el programa Perl, del cual se ha utilizado la versión 5.8.4, es un intérprete del lenguaje del mismo nombre, optimizado para realizar búsquedas arbitrarias en ficheros de texto, permitiendo extraer información de dichos ficheros y crear informes basados en dicha información.

- Test 1: ordenación de las entradas contenidas en un fichero utilizando para ello la clave primaria que se da al final de cada una de las entradas de dicho fichero.
- Test 2: script que traduce un fichero /etc/hosts en formato unixops a formato CS.
- Test 3: ajuste de las líneas que conforman un texto según las opciones dadas, utilizando un texto pequeño.
- Test 4: igual que el test anterior, esto es ajuste de las líneas que conforman un texto según las opciones dadas, utilizando un texto grande.

B.1.2. Patrón de uso de memoria dinámica: Parámetros

B. Zorn propuso en [148] tres parámetros que permiten evaluar la utilización de la memoria dinámica por parte de una aplicación. Estos parámetros son:

Tiempo de posesión (*HT*): este parámetro indica el tiempo (en kilociclos de procesador) que transcurre entre la asignación y la liberación de un bloque de memoria dinámica.

Tiempo entre llegadas (*IAR*): este parámetro indica el tiempo (en kilociclos de procesador) que transcurre entre una petición de asignación y la siguiente.

Tamaño de los objetos (*SC*): este parámetro indica el tamaño medio (bytes) de bloque pedido al gestor de memoria dinámica, además también se ha calculado la desviación típica.

Para reflejar mejor los resultados, se ha optado por mostrar el valor medio y la desviación típica de cada métrica en cada aplicación así como

mostrar en una gráfica la evolución de la cantidad de memoria reservada por cada una de las aplicaciones durante su ejecución.

B.1.3. Evaluación de las aplicaciones

- Aplicación **CFRAC**: CFRAC es una aplicación de cálculos matemáticos con gran consumo de CPU y poco consumo de recursos de E/S. Las medias y desviaciones típicas obtenidas reflejan que se trata de una aplicación que suele pedir bloque de memoria de tamaño pequeño, con un SC medio ≤ 32 bytes y una desviación típica también pequeña. Y que dichos bloques los conserva por una gran cantidad de tiempo (> 32 kCP (kilo ciclos de procesador)). Las gráficas reflejan además que la cantidad de memoria requerida por la aplicación es monótona creciente, es decir, que la aplicación rara vez libera memoria. Esto refleja una mala técnica de programación. El programa simplemente reserva memoria y espera que al terminar el sistema operativo sea el encargado de devolverla.
- Aplicación **Espresso**: tal como se puede observar en la tablas pertenecientes a esta aplicación, la aplicación Espresso hace un gran uso de la memoria dinámica aunque ninguno de los bloques que requiere supera los 512 bytes. Por lo general, esta aplicación utiliza bloques menores de 80 bytes. En las gráficas se muestra claramente tres fases en la ejecución de la aplicación: la inicialización, donde se realiza una gran asignación de memoria. La ejecución del cálculo, donde aunque existen requerimientos de memoria, la cantidad de memoria se suele mantener estable. Y por último, la finalización de la aplicación donde se libera prácticamente toda la memoria reservada con anterioridad. Sin embargo, igual que pasaba en CFRAC, Espresso no devuelve toda la memoria que ha reservado y confía dicha función al sistema operativo.
- Aplicación **GAWK**: los resultados obtenidos con aplicación GAWK muestran otra vez una aplicación con requerimiento de bloques de tamaño pequeños (no mayores de 1200 bytes). Esta aplicación también requiere una gran cantidad de memoria durante su ejecución. En el caso de esta aplicación, en las gráficas solamente se pueden apreciar dos fases: la inicialización de la aplicación, donde se reserva una gran cantidad de memoria y la fase de ejecución estable, donde aunque se continua reservando memoria, se observa una cierta estabilidad (tendencia a mantener siempre la misma cantidad de memoria reservada). Por último esta aplicación no libera memoria al finalizar su ejecución. Debe de ser el sistema operativo el que realice dicha liberación.
- Aplicación **GS**: tal como se observa en las tablas, la aplicación GS es

una aplicación atípica ya que llega a requerir bloques de gran tamaño (de hasta 32 Kbytes). Además de que dichos bloques los conserva durante un largo periodo de tiempo. Las gráficas muestran otra vez nada más que dos fases claras: la inicialización, donde se realiza la mayor cantidad de peticiones de memoria y posteriormente una fase de estabilidad donde la cantidad de memoria o bien se mantiene estable o bien crece lentamente. Tal como se ha mencionado anteriormente, esta aplicación no muestra una fase de finalización, es decir, no libera la memoria previamente asignada. Otra vez será el sistema operativo el encargado de realizar dicha liberación.

- Aplicación **Perl**: tal como se aprecia en las tablas, el intérprete Perl hace un gran uso de la memoria dinámica, utilizando siempre bloques de tamaño pequeño ($\leq$ 300 bytes). Perl demuestra otra vez la tendencia de mala programación que consiste en no liberar la memoria asignada y dejar dicha tarea al sistema operativo. Las gráficas obtenidas así lo demuestra, solo mostrando otra vez dos fases claras en la ejecución del intérprete Perl: la inicialización, mostrada como una línea monótonamente creciente, hasta llegar a una meseta, la fase de estabilidad, donde la cantidad de memoria reservada no varía (lo cual indica que se libera la misma cantidad de memoria que se vuelve a reservar). Otra vez la liberación total de la memoria previamente asignada es inexistente, dejando dicho trabajo al sistema operativo.

CFRAC. *SC*, *HT* y *IAR*. Tests: 1, 2, 3, 4 y 5.

CFRAC	*SC* (bytes)	*HT* (kCP)	*IAR* (kCP)
Test 1 media	14.8	106175	141
desv.std.	17.6	48570660	73640
Test 2 media	15.1	192242	221
desv.std.	18.8	112630900	161828
Test 3 media	16.4	2233049	726
desv.std.	19.9	5233106000	1621562
Test 4 media	17.5	8267087	1536
desv.std.	23.1	22831750000	5511209

Espresso. *SC*, *HT* y *IAR*. Tests: 1, 2, 3 y 4.

Espresso	*SC* (bytes)	*HT* (kCP)	*IAR* (kCP)
Test 1 media	42.6	2161	8
desv.std.	106.5	26327370	115303
Test 2 media	52.3	1865	8
desv. Típ.	149.5	44503190	287397
Test 3 media	80.9	35504	32
desv.std.	411.6	409653200	3838668
Test 4 media	63.9	46570	32
desv.std.	430.4	22890540000	28438810

GAWK. *SC*, *HT* y *IAR*. Tests: 1, 2 y 3.

GAWK	*SC* (bytes)	*HT* (kCP)	*IAR* (kCP)
Test 1 media	118.5	4191	144
desv.std.	1069.7	25886680	917964
Test 2 media	72.0	3059	50
desv.std.	738.0	3227578000	54396320
Test 3 media	73.2	3375	52
desv.std.	747.6	10268470000	161809700

GS. *SC*, *HT* y *IAR*. Tests: 1, 2 y 3.

GS	*SC* (bytes)	*HT* (kCP)	*IAR* (kCP)
Test 1 media	15861.8	839148	368
desv.std.	25477.8	151220000	129206
Test 2 media	9480.7	35063	127
desv.std.	12492.8	60457460	2959041
Test 3 media	29639.6	685133	695
desv.std.	29959.2	326155400	654780

Perl. *SC*, *HT* y *IAR*. Tests: 1, 2, 3 y 4.

Perl	*SC* (bytes)	*HT* (kCP)	*IAR* (kCP)
Test 1 media	63.1	16829	56
desv.std.	324.4	3719220	35276
Test 2 media	23.5	61134	157
desv.std.	137.3	78788870	1292210
Test 3 media	6.5	256099	262
desv.std.	26.5	1352481000	66067840
Test 4 media	6.7	251266	257
desv.std.	15.3	1194827000	183626600

B.1.4. Simulación CDF: funciones de distribución

B. Zorn propuso en [148] el modelo CDF, el cual modeliza la secuencia de peticiones de memoria dinámica de una aplicación mediante tres funciones de probabilidad. La función de probabilidad *HT*, la función de probabilidad *IAR* y la función de probabilidad *SC* (la descripción de estos parámetros puede ser consultada en B.1.2).

A partir de estas funciones de distribución y de una distribución uniforme es posible simular el comportamiento de la aplicación.

Una clara ventaja de utilizar este modelos sintéticos es la posibilidad de ejecutar tantas operaciones de gestión de memoria como se deseen. Sin embargo, tal como se concluye en [148], ningún modelo existente reproduce fidedignamente el comportamiento respecto a la memoria dinámica de ninguna aplicación.

A continuación se encuentran las funciones de probabilidad calculadas para las aplicaciones CFRAC, Espresso, GAWK, GS y Perl.

Se ha evitado poner en este apéndice los histogramas obtenidos a partir de dichas aplicaciones debido a que no aportan mayor información que la aportada por las siguientes funciones de probabilidad.

CFRAC Test 1

Prob.	HT	IAR	SC
0.00	< 512	< 1	< 4
0.01	$< 1K$	-	-
0.02	$< 2K$	-	-
0.05	$< 4K$	-	-
0.13	$< 8K$	-	-
0.18	-	< 2	-
0.20	-	-	< 8
0.30	-	< 4	-
0.32	$< 16K$	-	-
0.56	-	< 8	-
0.57	-	-	< 32
0.78	-	< 16	-
0.86	-	< 32	-
0.92	-	< 64	-
1.00	$\geq 32K$	≥ 128	≥ 256

CFRAC Test 2

Prob.	HT	IAR	SC
0.00	$< 1K$	< 1	< 8
0.01	$< 2K$	-	-
0.03	$< 4K$	-	-
0.06	$< 8K$	-	-
0.13	$< 16K$	-	-
0.16	-	< 2	-
0.21	-	-	< 16
0.27	-	< 4	-
0.38	$< 32K$	-	-
0.53	-	< 8	-
0.55	-	-	< 32
0.70	-	< 16	-
0.77	-	< 32	-
0.85	-	< 64	-
0.91	-	< 128	-
1.00	$\geq 64K$	≥ 256	≥ 512

CFRAC Test 3

Prob.	HT	IAR	SC
0.00	$< 16K$	< 1	< 16
0.01	$< 32K$	-	-
0.03	$< 64K$	-	-
0.04	-	< 2	-
0.07	-	< 4	-
0.09	$< 128K$	-	-
0.13	-	< 8	-
0.22	$< 256K$	-	-
0.25	-	< 16	-
0.50	-	< 32	-
0.57	-	-	< 32
0.77	-	< 64	-
0.82	-	< 128	-
0.88	-	< 256	-
0.94	-	< 512	-
1.00	$\geq 512K$	$\geq 1K$	< 64

CFRAC Test 4

Prob.	HT	IAR	SC
0.00	$< 128K$	< 1	< 16
0.02	$< 256K$	-	-
0.03	-	< 2	-
0.05	-	< 4	-
0.06	$< 512K$	-	-
0.09	-	< 8	-
0.16	$< 1M$	-	-
0.17	-	< 16	-
0.32	-	< 32	-
0.44	-	< 64	-
0.55	-	< 128	-
0.58	-	-	< 32
0.76	-	< 256	-
0.86	-	< 512	-
0.94	-	$< 1K$	-
1.00	$\geq 2M$	$\geq 2K$	≥ 64

Tabla B.1: Aplicación CFRAC. Tests 1, 2, 3, 4.

Espresso Test 1			
Prob.	HT	IAR	SC
0.00	< 1	< 1	< 4
0.01	-	-	< 8
0.31	< 2	-	-
0.36	< 4	-	-
0.41	-	-	< 16
0.44	-	-	< 32
0.47	< 8	-	-
0.55	< 16	-	-
0.64	< 32	-	-
0.72	< 64	-	-
0.77	< 128	-	-
0.81	-	-	< 64
0.82	< 256	-	-
0.87	< 512	-	-
0.88	-	< 2	-
0.90	-	-	< 128
0.91	$< 1K$	-	-
0.95	$< 2K$	-	< 256
0.96	-	< 4	< 512
0.98	$< 4K$	< 8	-
1.00	$\geq 32K$	≥ 16	$\geq 1K$

Espresso Test 2			
Prob.	HT	IAR	SC
0.00	< 1	< 1	< 4
0.01	-	-	< 8
0.25	< 2	-	-
0.29	< 4	-	-
0.31	-	-	< 16
0.32	-	-	< 32
0.39	< 8	-	-
0.46	< 16	-	-
0.53	< 32	-	-
0.61	< 64	-	-
0.69	< 128	-	-
0.79	< 256	-	-
0.84	-	-	< 64
0.87	< 512	-	-
0.91	-	-	< 128
0.92	-	< 2	-
0.95	$< 1K$	-	< 256
0.96	-	< 4	< 512
0.97	$< 2K$	-	-
0.98	-	< 8	-
1.00	$\geq 128K$	≥ 32	$\geq 1K$

Tabla B.2: Aplicación Espresso. Tests 1 y 2.

Espresso Test 3

Prob.	HT	IAR	SC
0.00	< 1	< 1	< 8
0.02	-	-	< 16
0.03	-	-	< 32
0.21	< 2	-	-
0.26	< 4	-	-
0.29	< 8	-	-
0.38	< 16	-	-
0.45	< 32	-	-
0.52	< 64	-	-
0.57	< 128	-	-
0.61	< 256	-	-
0.64	< 512	-	-
0.66	$< 1K$	-	-
0.67	-	< 2	-
0.69	$< 2K$	-	-
0.74	$< 4K$	-	-
0.77	-	< 4	-
0.80	$< 8K$	-	-
0.83	-	-	< 64
0.86	$< 16K$	-	-
0.88	-	< 8	-
0.90	-	-	< 128
0.92	$< 32K$	-	-
0.93	-	-	< 256
0.96	-	-	< 512
0.97	-	< 16	$< 1K$
1.00	$\geq 1M$	≥ 32	$\geq 2K$

Espresso Test 4

Prob.	HT	IAR	SC
0.00	< 1	< 1	< 4
0.01	-	-	< 8
0.03	-	-	< 16
0.18	< 2	-	-
0.20	< 4	-	-
0.22	< 8	-	-
0.29	< 16	-	-
0.34	-	-	< 32
0.35	< 32	-	-
0.43	< 64	-	-
0.50	< 128	-	-
0.56	< 256	-	-
0.61	< 512	-	-
0.66	$< 1K$	-	-
0.70	-	< 2	-
0.71	$< 2K$	-	-
0.75	$< 4K$	-	-
0.79	-	< 4	-
0.81	$< 8K$	-	-
0.86	$< 16K$	-	< 64
0.89	-	< 8	-
0.91	$< 32K$	-	-
0.93	-	-	< 128
0.95	$< 64K$	-	-
0.96	-	< 16	< 256
0.97	-	-	< 512
1.00	$\geq 128K$	≥ 32	$\geq 4K$

Tabla B.3: Aplicación Espresso. Tests 3 y 4.

GAWK Test 1

Prob.	HT	IAR	SC
0.00	< 1	< 1	< 1
0.01	-	-	< 2
0.02	-	-	< 4
0.07	-	< 2	-
0.10	-	< 4	-
0.13	-	< 8	-
0.17	< 2	-	-
0.19	< 4	-	-
0.21	< 8	-	-
0.22	-	-	< 8
0.25	-	< 16	-
0.29	< 16	-	-
0.35	< 32	-	-
0.43	< 64	-	-
0.49	< 128	-	-
0.55	< 256	-	-
0.60	< 512	-	-
0.65	$< 1K$	-	-
0.70	$< 2K$	-	-
0.75	$< 4K$	-	-
0.80	$< 8K$	-	-
0.86	$< 16K$	-	-
0.90	-	< 32	-
0.91	$< 32K$	-	< 16
0.94	$< 64K$	-	< 32
0.95	-	-	< 64
0.97	-	-	< 256
0.98	$< 128K$	-	-
1.00	$\geq 2M$	≥ 128	$\geq 16K$

GAWK Test 2

Prob.	HT	IAR	SC
0.00	< 1	< 1	< 1
0.03	-	-	< 2
0.04	-	-	< 4
0.08	< 4	< 2	-
0.15	< 8	-	-
0.19	< 16	-	-
0.30	< 32	-	-
0.32	< 64	-	-
0.34	-	-	< 8
0.41	< 128	-	-
0.45	< 256	-	-
0.47	< 512	-	-
0.52	-	-	< 16
0.67	-	< 4	-
0.69	-	-	< 32
0.70	-	< 8	-
0.76	-	-	< 64
0.83	$< 1K$	-	-
0.91	-	< 16	-
0.93	-	-	< 128
0.97	$< 2K$	-	-
1.00	$\geq 4K$	≥ 64	≥ 256

Tabla B.4: Aplicación GAWK. Tests 1 y 2.

GAWK Test 3

Prob.	HT	IAR	SC
0.00	< 1	< 1	< 1
0.03	-	-	< 2
0.04	-	-	< 4
0.07	< 4	< 2	-
0.15	< 8	-	-
0.18	< 16	-	-
0.29	< 32	-	-
0.32	< 64	-	-
0.35	-	-	< 8
0.41	< 128	-	-
0.44	< 256	-	-
0.47	< 512	-	-
0.52	-	-	< 16
0.66	-	< 4	-
0.69	-	< 8	< 32
0.76	-	-	< 64
0.81	$< 1K$	-	-
0.91	-	< 16	-
0.93	-	-	< 128
0.96	$< 2K$	-	-
1.00	$\geq 4K$	≥ 64	≥ 256

Tabla B.5: Aplicación GAWK. Test 3.

GS Test 1

Prob.	HT	IAR	SC
0.00	< 1	< 1	< 2
0.01	-	-	< 8
0.02	-	-	< 16
0.03	-	-	< 32
0.06	-	-	< 128
0.25	< 2	-	< 256
0.27	-	-	< 512
0.29	< 4	-	-
0.32	< 8	-	-
0.34	< 16	< 2	-
0.39	< 32	-	-
0.46	< 64	-	-
0.55	-	< 4	-
0.59	< 128	-	-
0.64	< 256	-	-
0.66	-	-	$< 1K$
0.67	-	-	$< 2K$
0.68	-	$< 8K$	$< 4K$
0.69	-	$< 512K$	$< 16K$
0.70	-	-	$< 32K$
0.71	-	< 8	-
0.76	-	< 16	-
0.79	-	< 32	-
0.83	-	< 64	-
0.88	-	< 128	-
0.93	-	< 256	-
0.97	-	< 512	-
1.00	$\geq 512K$	$\geq 2K$	$\geq 64K$

GS Test 2

Prob.	HT	IAR	SC
0.00	< 8	< 1	< 128
0.01	< 16	-	-
0.26	< 32	-	-
0.36	-	< 2	-
0.41	-	< 4	-
0.49	-	< 8	-
0.63	-	-	$< 32K$
0.66	< 64	< 16	-
0.74	-	< 32	-
0.79	-	< 64	-
0.97	< 128	-	-
0.98	$< 1K$	-	-
1.00	$\geq 8K$	≥ 128	$\geq 32K$

Tabla B.6: Aplicación GS. Tests 1 y 2.

GS Test 3

Prob.	HT	IAR	SC
0.00	< 8	< 1	< 128
0.01	< 16	-	-
0.26	< 32	-	-
0.36	-	< 2	-
0.41	-	< 4	-
0.49	-	< 8	-
0.63	-	-	$< 32K$
0.66	< 64	< 16	-
0.74	-	< 32	-
0.79	-	< 64	-
0.97	< 128	-	-
0.98	$< 1K$	-	-
1.00	$\geq 8K$	≥ 128	$\geq 32K$

Tabla B.7: Aplicación GS. Test 3.

Perl Test 1

Prob.	HT	IAR	SC
0.00	< 2	< 1	< 1
0.02	-	-	< 2
0.03	< 4	-	-
0.04	< 8	-	-
0.06	< 16	-	-
0.10	-	-	< 4
0.15	< 32	-	< 8
0.20	< 64	-	-
0.22	-	< 2	-
0.29	< 128	-	< 16
0.38	-	< 4	-
0.42	< 256	-	-
0.46	-	-	< 32
0.55	< 512	-	-
0.62	$< 1K$	-	-
0.64	$< 2K$	-	-
0.71	$< 4K$	-	-
0.75	$< 8K$	-	-
0.77	-	-	< 64
0.81	-	< 8	-
0.91	-	< 16	-
0.92	-	-	< 128
0.96	-	< 32	< 512
0.98	-	< 64	-
1.00	$\geq 16K$	≥ 128	$\geq 1K$

Perl Test 2

Prob.	HT	IAR	SC
0.00	< 2	< 2	< 1
0.01	< 4	-	-
0.02	-	< 2	-
0.04	-	< 4	-
0.07	-	< 8	-
0.19	< 16	-	-
0.27	-	-	< 2
0.28	-	-	< 4
0.29	-	-	< 8
0.38	< 32	-	-
0.39	< 64	-	-
0.43	-	-	< 16
0.45	-	< 16	-
0.49	< 128	-	-
0.50	< 256	-	-
0.59	-	-	< 32
0.72	-	< 32	-
0.89	-	-	< 64
0.90	-	< 64	-
0.91	-	-	< 128
0.95	< 512	-	-
0.96	$< 2K$	-	-
1.00	$\geq 256K$	≥ 128	≥ 256

Tabla B.8: Aplicación Perl. Tests 1 y 2.

Perl Test 3

Prob.	HT	IAR	SC
0.00	< 8	< 16	< 1
0.01	-	-	< 2
0.22	< 16	-	-
0.23	< 32	-	-
0.33	-	-	< 4
0.36	-	< 32	-
0.37	< 64	-	-
0.38	-	-	< 8
0.56	< 128	-	-
0.70	-	< 64	-
0.71	-	-	< 16
0.75	< 256	-	-
0.87	< 512	-	-
1.00	$\geq 1K$	≥ 128	≥ 32

Perl Test 4

Prob.	HT	IAR	SC
0.00	< 8	< 16	< 1
0.01	-	-	< 2
0.22	< 16	-	-
0.23	< 32	-	-
0.33	-	-	< 4
0.36	< 64	-	-
0.37	-	< 32	-
0.38	-	-	< 8
0.56	< 128	-	-
0.64	-	-	< 16
0.75	< 256	-	-
1.00	≥ 512	≥ 64	≥ 32

Tabla B.9: Aplicación Perl. Tests 3 y 4.

Apéndice C
Entorno de pruebas (SA-Tester)

En este apéndice se describe el entorno de pruebas SA-Tester, específicamente diseñado para evitar las interferencias causadas por el sistema operativo en el estudio temporal realizado durante el capítulo 5.

C.1. Introducción

El tiempo de respuesta de una operación de asignación o liberación suele ser unos pocos microsegundos. La interferencia causada por un sistema operativo convencional debido al tratamiento de interrupciones, cambios de contexto, paginación, etc, suele ser de varias decenas de microsegundos. Por esta razón resulta bastante difícil medir con precisión el tiempo de respuesta de las operaciones de un gestor de memoria. Para facilitar la obtención del tiempo de respuesta de los gestores estudiados en el capítulo 5, se ha implementado un entorno de ejecución mínimo llamado *SA-Tester*. Básicamente permite ejecutar una aplicación mínima sin la interferencia propia de un sistemas operativo.

C.2. Criterios de diseño del entorno SA-Tester

El objetivo del entorno SA-Tester consiste en ejecutar una aplicación mínima con la mínima interferencia de este y de los eventos externos. Por lo tanto se han tomado las siguientes decisiones:

- **Ejecución como entorno autónomo**: el entorno desarrollado se ejecuta de manera autónoma sin necesidad de ningún sistema operativo que le proporcione ningún tipo de soporte, el propio entorno se encarga de la gestión del hardware que necesite utilizar (por ejemplo la pantalla, el puerto serie, las interrupciones, el reloj del sistema, etc.).
- **Soporte mínimo**: el entorno desarrollado implementa el soporte mínimo para ejecutar una aplicación mínima. Este soporte proporciona:
  - Salida por pantalla/serie: proporcionar algún mecanismo para mostrar los resultados es totalmente necesario, en el caso del SATester se ha optado por dos mecanismos diferentes, salida por pantalla y salida por el puerto serie.
  - Funciones de biblioteca básicas: funciones para trabajar con cadenas, funciones para trabajar con memoria, funciones matemáticas.
  - Funciones de abstracción del hardware: funciones para vaciar la cache, gestión de interrupciones, lectura de ciclos, control del reloj, etc.
  - Funciones para obtener el tiempo de respuesta: funciones que permiten medir el número de instrucciones ejecutadas, ciclos de procesadores, etc.
- **Mantener inhabilitadas las interrupciones**: para minimizar el impacto de las interrupciones en las métricas, estas permanecen inhabilitadas.

- **No usar ningún mecanismo de memoria virtual**: nuestro entorno de pruebas no implementa ningún mecanismo de memoria virtual y no necesita el uso de la unidad TLB ni de la unidad MMU. Por lo cual se permite eliminar la incertidumbre introducida por estas dos unidades.
- **No proporcionar un sistema de ficheros**: SA-Tester no pretende ejecutar aplicaciones complejas, solamente medir el tiempo de respuesta de un gestor de memoria en una aplicación sencilla sin interacción con el sistema de ficheros. Por lo tanto no se provee ningún sistema de ficheros.
- **No proporcionar llamadas al sistema**: el entorno de ejecución proporciona un entorno mínimo con una interferencia mínima. Proporcionar servicios aumentaría la interferencia por lo que resulta inadmisible.

C.3. Implementación del entorno SA-Tester

El entorno mínimo SA-Tester está disponible para la arquitectura x86 de Intel®. El SA-Tester ha sido implementado como una biblioteca compuesta de la siguiente funcionalidad:

- **Inicialización de la máquina**: SA-Tester implementa el estándar Multiboot [92]. Requiere, por lo tanto un cargador que también lo implemente como por ejemplo GRUB [91] o Etherboot [138]. Estas funciones se encargan de inicializar la tabla global de descriptores (GDT) y los segmentos del sistema. Las interrupciones y la paginación permanecen inhabilitados, por lo tanto no es necesario inicializar la tabla global de interrupciones (IDT).
- **Funciones de biblioteca**: compuestas por funciones para trabajar con cadenas de texto, con memoria, con operaciones matemáticas, etc.
- **Funciones de abstracción del hardware**: permiten imprimir en pantalla, gestionar el reloj del sistema, vaciar la memoria cache, vaciar la TLB.
- **Funciones para leer el tiempo de respuesta**: permiten obtener el número de instrucciones ejecutadas, ciclos de procesador, etc.

El entorno mínimo SA-Tester ha sido implementado como un conjunto de bibliotecas, las cuales son enlazadas directamente con la aplicación a ejecutar creando un núcleo ejecutable compatible con el estándar Multiboot [92]. Este entorno no suministra llamadas al sistema ni sistema de ficheros, lo que restringe enormemente el número de aplicaciones que pueden se ejecutadas sobre él.

Apéndice D
Versiones y ámbito de uso de TLSF

En este apéndice se presentan, por una parte, las diferentes versiones e implementaciones que existen del gestor TLSF. Por otra parte, se describen los diferentes proyectos en los cuales el gestor TLSF está siendo utilizado.

D.1. Versiones e implementaciones de TLSF

La implementación del algoritmo TLSF es uno de los aspectos clave de la eficiencia del mismo, ya que, aunque teóricamente se define perfectamente las claves que convierten al algoritmo en temporalmente constante, es la implementación del mismo la que debe proporcionar este tipo de comportamiento.

Actualmente se dispone de tres implementaciones diferentes del algoritmo TLSF:

1. Una versión genérica para arquitecturas de 32 bits, la cual no depende de ninguna funcionalidad proporcionada por el procesador.
2. Una versión para al arquitectura Intel® x86, la cual utiliza las instrucciones específicas de dicho procesador para trabajar con bits. En concreto se utilizan la instrucción `bsf` y la instrucción `bsr`, las cuales indican el primer bit menos o más significativo a uno respectivamente en una palabra. Se ha comprobado que esta versión del algoritmo obtiene tiempos de respuesta menores que la versión genérica.
3. Además existe una versión implementada por M. Candela [23] para procesadores de 16 bits. La cual no ha podido ser comprobada por no encontrarse públicamente disponible.

De todas las versiones existentes, solamente las versiones para la arquitectura de 32 bits han sido implementadas por el autor de esta tesis. La versión de 16 bits ha sido implementada por M. Candela para su uso comercial en telefonía móvil.

D.1.1. Historial de versiones

Desde que fue inicialmente diseñado, la implementación del gestor TLSF ha evolucionado mucho, se han eliminado muchos errores, se ha mejorado enormemente la legibilidad del código y se ha mejorado la portabilidad.

Para permitir un mejor seguimiento de las versiones, se optó por utilizar un número de versión compuesto por tres cifras **X.YZ**. La cifra **X** indica el número de versión y las cifras **YZ** el número de subversión. Un cambio en el número de versión implica una cambio drástico en la implementación, como por ejemplo una reescritura de la misma. Un cambio en la **Y** representa la eliminación de uno o varios errores graves, mientras que un cambio en **Z** representa un modificación de poca importancia en el algoritmo, normalmente cosmético.

El historial de versiones es el siguiente:

- Versión 2.0: Por razones de portabilidad y de legibilidad, el gestor TLSF ha sido completamente reimplementado en esta versión. Todo el código

existente ha sido, otra vez implementado en un único fichero. En esta versión, las estructuras de datos han sido reimplementadas como matrices estáticas, lo que traduce en una mayor velocidad de acceso a las mismas.

- Versión 1.4: Tras el estudio del comportamiento de fragmentación del algoritmo en aplicaciones reales se decide crear listas de gestión de primer nivel para gestionar bloques de tamaño menor a 16 bytes, lo cual decrementa enormemente la fragmentación causada por el mismo.
- Versión 1.3: Cambio de concepto, en todas las versiones previas la estructura TLSF era una única estructura estática, la cual gestionaba el bloque de memoria (montículo) que el usuario le proporcionaba. A partir de esta versión TLSF tiene la capacidad de gestionar más de una estructura TLSF a la vez, esto lo consigue creando, bajo petición del usuario, una nueva estructura TLSF al principio del montículo. Esto permite trabajar con varios montículos a la vez. Además se añaden funciones que permiten incrementar el montículo así como añadir áreas de memoria que no son contiguas a un montículo preexistente. Esta versión del TLSF incorpora además funciones de depuración internas, las cuales permiten comprobar dinámicamente el estado del montículo en busca de errores.
- Versión 1.2: Arreglados más errores (errores de alineamiento de memoria), a partir de esta versión, por razones de eficiencia, el algoritmo ya no es capaz de trabajar con bytes sino con palabras. Incluida la versión TLSF para Intel 86, la cual saca ventaja del uso de las instrucciones específicas del procesador para trabajar con bits.
- Versión 1.1.1: Hasta el momento, el algoritmo había sido implementado en un único fichero, a partir de esta versión, se divide en una serie de ficheros, lo cual facilita enormemente su lectura.
- Versión 1.1: Varios errores severos arreglados (problemas en la gestión de los límites del montículo inicial). En esta versión ya no es necesario introducir el índice máximo de primer nivel. Añadido el prefijo `rtl_` a todas las funciones ofrecidas por el algoritmo (ahora el algoritmo ofrece por ejemplo `rtl_malloc` en lugar de `malloc`). Comenzando por esta versión, el algoritmo es adoptado como el gestor de memoria por defecto del sistema operativo MaRTE OS [1] a partir de su versión 1.2.
- Versión 1.0: Reimplementación cuidadosa completa del algoritmo, se fija el tamaño máximo de los mapas de bits a una palabra, se sustituyen las búsquedas lineales en los mapas de bits por búsquedas logarítmicas. El código del algoritmo se independiza del operativo RTLinux [9, 145] haciéndolo directamente utilizable por cualquier operativo de 32 bits. Sin embargo aún es necesario introducir el índice máximo de primer

nivel. Introducidas las funciones `realloc` y `calloc`.

- Versión 0.64 .. 0.1: Primeras implementaciones del algoritmo, las cuales fueron inicialmente desarrolladas para pruebas e investigación del mismo. El código contenía una gran cantidad de errores, lo cual puede ser apreciado en la gran cantidad de subversiones que aparecieron, nueve en total. Estas primeras versiones implementaban búsquedas lineales en los mapas de bits (los cuales eran de tamaño fijo), convirtiendo a la operación búsqueda en una operación de gran coste. Además estas versiones fueron diseñadas para trabajar únicamente con el operativo RTLinux [9, 145] dentro del contexto del proyecto OCERA [136], con lo cual no se podía utilizar fácilmente en otro sistema operativo. Estas primeras versiones proveían la funcionalidad básica: `malloc` y `free`.

La mayoría de estas versiones pueden ser conseguidas directamente de la página web [104], ya que todas ellas han sido liberadas bajo una estrategia de licencias dual, es decir, el código se suministra bajo los términos de la licencia GPL [42] y LGPL [43], pudiéndose elegir la licencia que más se adecua a los requisitos de cada usuario.

D.2. Ámbito de uso de TLSF

Desde su implementación, el gestor TLSF ha sido utilizado con éxito en una gran cantidad de proyectos relacionados con el área de tiempo real, e, incluso a veces, en otras áreas.

D.2.1. Proyectos propios

A continuación se describen cada unos de los proyectos del autor de esta tesis donde el gestor TLSF ha sido utilizado:

- **RTLGnat**: RTLGnat [76] es una adaptación del compilador de Ada [37] GNAT al núcleo de tiempo real estricto RTLinux [145]. La mayoría de facilidades descritas por el estándar Ada necesitan de un soporte de ejecución (GNARL en GNAT) con memoria dinámica para ser utilizadas. El gestor TLSF es usado por el soporte de ejecución para gestionar la memoria dinámica.
- **GLADE para RTLinux**: Adaptación de GLADE al sistema operativo RTLinux [75]. GLADE es una implementación del anexo D (Sistemas distribuidos) perteneciente al estándar Ada, el cual describe una serie de facilidades para implementar este tipo de sistemas. GLADE para RTLinux, al contrario que RTGlade [22], no proporciona garantías temporales, por lo que solo permite comunicar un sistema de tiempo real

con sistemas sin requerimientos temporales. El gestor TLSF es utilizado por el suporte de ejecución de GLADE, conocido como GARLIC.

- **MaRTE OS en Linux**: Adaptación del sistema operativo MaRTE OS al sistema operativo Linux [77]. Este proyecto permite ejecutar a MaRTE OS como si de una aplicación de Linux se tratara. El gestor TLSF es usado para gestionar la memoria de MaRTE OS.
- **C++ para RTLinux**: Adaptación del soporte de ejecución del compilador GNU C++ al sistema operativo RTLinux, lo que permite a dicho compilador construir ejecutables para RTLinux. El soporte de ejecución utiliza al gestor TLSF para gestionar la memoria dinámica.
- **RTLJava**: Adaptación de la máquina virtual Java, JamVM [73]. al sistema RTLinux [83]. JamVM es una pequeña máquina virtual escrita en C para sistemas empotrados. El gestor TLSF permite gestionar la memoria dinámica a la máquina JamVM.
- **XtratuM**: Nano-kernel desarrollado para permitir la ejecución simultánea de varios sistemas operativos donde al menos uno de ellos es un sistema de tiempo real [81, 79]. Para la primera implementación se ha utilizado una aproximación similar a la utilizada por el nano-kernel Adeos [144]. El sistema operativo Linux es el encargado de inicializar la máquina y de gestionar los diferentes dispositivos físicos. El nano-kernel es implementado como un módulo de Linux. Una vez insertado toma el control de las interrupciones, permitiendo que se ejecuten varios sistemas sobre él. El gestor TLSF gestiona la memoria dinámica utilizada para cargar los diferentes sistemas operativos.

D.2.2. Proyectos externos

El TLSF ha sido empleado con éxito en multitud de proyectos no relacionados con el autor de esta tesis. A continuación se enumeran una breve descripción de cada uno de ellos:

- **MaRTE OS**: Núcleo de tiempo real para aplicaciones empotradas que implementa la interfaz definida por el estándar POSIX.13 para sistemas de tiempo real mínimos. Este núcleo ha sido diseñado e implementado por M. Aldea y M. González [1]. La mayoría de su código se ha escrito en Ada [37] con algunas partes en C [38] y en ensamblador. El gestor TLSF fue introducido con éxito a partir de la versión 1.4 (Diciembre 2003).
- **RTLinux**: Núcleo de tiempo real que permite ejecutar al sistema operativo Linux como una tarea más del sistema con mínima prioridad [145]. Este sistema implementa una interfaz compatible con el estándar POSIX.13 para sistemas de tiempo real mínimos. El gestor TLSF fue in-

troducido para gestionar la memoria dinámica a partir de la versión 3.2-rc1.

- **RTAI**: Proyecto derivado de RTLinux. Al igual que RTLinux, este núcleo ejecuta al sistema Linux como una tarea de mínima prioridad [35]. El gestor TLSF gestiona la memoria dinámica de las aplicaciones ejecutadas sobre RTAI.
- **Lightweight IP para RTLinux**: Adaptación realizada por S. Pérez [99] del proyecto Lightweight IP (pila TCP/IP) para RTLinux. El gestor TLSF se utiliza para gestionar los paquetes de red enviados y recibidos.
- **Teléfonos móviles de Neoseven**: M. Candela de Neoseven [23] ha implementado una versión del gestor TLSF para arquitecturas de 16 bits y lo utiliza con éxito en todos los teléfonos móviles de esta compañía.
- **Videojuegos en las plataformas PS2 y XBox**: Una gestión de memoria eficiente resulta crucial en el desarrollo de un videojuego. Las vídeo-consolas actuales incluyen una cantidad de memoria muy reducida (32 Mbytes en el caso de la consola PS2) lo que hace muy probable el fallo del programa debido al problema de la fragmentación.
  Hasta hace relativamente poco, los programadores de videojuegos habían utilizado el gestor DLmalloc por considerarlo muy eficiente. Sin embargo, desde la aparición del gestor TLSF, cada vez hay más desarrolladores que han admitido utilizarlo.

Bibliografía

[1] M. Aldea and M. González-Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. volume 2043, pages 305–316, 2001.

[2] A. Alonso-Muñoz. Extensiones a los Métodos de Planificación de Sistemas de Tiempo Real Críticos Basados en Prioridades. Master's thesis, Universidad Politécnica de Madrid, Facultad de Informática, Madrid, Spain, February 1994.

[3] D. Atienza, S. Mamagkakis, F. Catthoor, J.M. Mendias, and D. Soudris. Dynamic Memory Management Design Methodology for Reduced Memory Footprint in Multimedia and Wireless Network Applications. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10532, Washington, DC, USA, 2004. IEEE Computer Society.

[4] A.N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[5] N.C. Audsley. Resource Control For Hard Real-Time Systems: A Review. Technical Report YCS 159, 1991.

[6] N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell, and A.J. Wellings. Fixed Priority Preemptive Scheduling: An Historical Perspective. *Real-Time Systems Symposium*, 8(2-3):173–198, 1995.

[7] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard Real-Time Scheduling: the Deadline Monotonic Approach. In *8th Workshop on real-time operating systems and software*, pages 127–132, 1991.

[8] T. Baker. Protected Records, Time Management and Distribution. *Ada Lett.*, X(9):17–28, 1990.

[9] M. Barabanov. A Linux-Based Real-Time Operating System. Master's thesis, New Mexico Institute of Mining and Technology, Socorro, New Mexico, June 1997.

[10] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[11] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. *Real-Time Systems Symposium*, 2(4):301–324, 1990.

[12] S. Basumallick and K.D. Nilsen. Cache Issues in Real-Time Systems. 1994.

[13] C. Bays. A Comparision of Fext-Fit, First-Fit and Best-Fit. *Communications of the ACM*, 20(3):191–192, 1977.

[14] E.D. Berger, K.S. McKinley, R.D. Blumofe, and P.R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.

[15] E.D. Berger, B.G. Zorn, and K.S. McKinley. Composing High-Performance Memory Allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, jun 2001.

[16] E.D. Berger, B.G. Zorn, and K.S. McKinley. Reconsidering Custom Memory Allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on object oriented programming systems languages and applications*, pages 1–12, 2002.

[17] A. Bohra and E. Gabber. Are Mallocs Free of Fragmentation? In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 105–117, Berkeley, CA, USA, 2001. USENIX Association.

[18] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1988.

[19] R. Brent. Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(3):388 – 403, 1982.

[20] A. Burns. Scheduling Hard Real-Time Systems: A Review. *Software Engineering Journal*, 6(3):116–128, 1991.

[21] W. Burton. A Buddy System Variation for Disk Storage Allocation. *Communications of the ACM*, 19(7):416–417, 1976.

[22] J. López Campos, J.J. Gutiérrez, and M. González. The Chance for Ada to Support Distribution and Real-Time in Embedded Systems. In *Ada-Europe*, pages 91–105, 2004.

[23] M. Candela. Neoseven. http://www.neoseven.com.

[24] J.M. Chang and E.F. Gehringer. A High-Performance Memory Allocator for Object-Oriented Systems. *IEEE Transactions on Computers*, 45(3):357–366, 1996.

[25] J.M. Chang, W. Srisa-an, C.-T. Dan Lo, and E.F. Gehringer. DMMX: dynamic memory management extensions. *J. Syst. Software*, 63(3):187–199, 2002.

[26] M.I. Chen and K.J. Lin. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. pages 325–346, February 1990.

[27] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions Software Engineering*, 15(10):1261–1269, 1989.

[28] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic Scheduling of Real-Time Tasks Under Precedence Constraints. *Real-Time Systems Symposium*, 2(3):181–194, 1990.

[29] T. Chilimbi, R. Jones, and B. Zorn. Designing a Trace Format for Heap Allocation Events. In Tony Hosking, editor, *ISMM2000 International Symposium on Memory Management*, pages 35–49, Minneapolis, MN, October 2000. ACM Press.

[30] G. Confessore, P. DellÓlmo, and S. Giordani. An Approximation Result for a Periodic Allocation Problem. *Discrete Applied Mathematics*, 112(1-3):53–72, 2001.

[31] R.I. Davis, K.W. Tindell, and A.J. Burns. Scheduling Slack Time in Fixed Priority Preemptive Systems. pages 222–231, December 1993.

[32] M. Dertouzos. Control Robotics: The Procedural Control of Physical Processors. pages 807–813, 1974.

[33] E.W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.

[34] G. Bollella et al. *Real-Time Specification for Java*. 2004.

[35] P. Mantegazza et al. Real-Time Application Interface (RTAI). http://rtai.org.

[36] International Organization for Standardization (ISO). ISO/IEC 14882, Programming language C++.

[37] International Organization for Standardization (ISO). ISO/IEC 8652:1005:TC1:2000, Programming language Ada.

[38] International Organization for Standardization (ISO). ISO/IEC 9899:1990, Programming language C.

[39] R. Ford. Concurrent Algorithms for Real-Time Memory Management. *IEEE Software*, 5(5):10–23, September/October 1996.

[40] Free Software Foundation. GNU C Library. http://www.gnu.org/-software/libc/libc.html.

[41] Free Software Foundation. GNU C++ Library. http://www.gnu.-org/directory/gpp.html.

[42] Free Software Foundation. GNU General Public License (GPL). http://www.gnu.org/licenses/.

[43] Free Software Foundation. GNU Lesser General Public License (LGPL). http://www.gnu.org/licenses/.

[44] Gentoo Foundation. Gentoo Linux distribution. http://www.gentoo.-org.

[45] M.R. Garey, R.L. Graham, and J.D. Ullman. Worst-Case Analysis of Memory Allocation Algorithms. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 143–150, New York, NY, USA, 1972. ACM Press.

[46] M.R. Garey and D.S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[47] F. Gavril. Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph. *SIAM Journal on Compututing*, 1(2):180–187, 1972.

[48] J. Gergov. Approximation Algorithms for Dynamic Storage Allocations. In *ESA '96: Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 52–61, London, UK, 1996. Springer-Verlag.

[49] T.M. Ghazalie and T.P. Baker. Aperiodic Servers in a Deadline Scheduling Environment. pages 31–67, September 1995.

[50] W. Gloger. Ptmalloc Memory Allocator. http://www.malloc.de/en/.

[51] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification. Third Edition*. Addison-Wesley Professional, 2005.

[52] D. Grunwald and B. Zorn. CustoMalloc: Efficient Synthesized Memory Allocators. *Software Practice and Experience*, 23(8):851–869, 1993.

[53] Y. Hasan and J.M. Chang. A Hybrid Allocator. pages 214 – 222, March 2003.

[54] J.A. Hinds. An Algorithm for Locating Adjacent Storage Blocks in the Buddy System. *Communications of the ACM*, 18(4):221–222, 1975.

[55] D.S. Hirschberg. A Class of Dynamic Memory Allocation Algorithms. *Communications of the ACM*, 16(10):615–618, 1973.

[56] A. Iyengar. Scalability of Dynamic Storage Allocation Algorithms. In *FRONTIERS '96: Proceedings of the 6th Symposium on the Frontiers*

of Massively Parallel Computation, page 223, Washington, DC, USA, 1996. IEEE Computer Society.

[57] M.S. Johnstone and P.R. Wilson. The Memory Fragmentation Problem: Solved? In *Proceedings of the International Symposium on Memory Management (ISMM'98), Vancouver, Canada*. ACM Press, 1998.

[58] B. Kalyanasundaram and K. Pruhs. Dynamic Spectrum Allocation: The Impotency of Duration Notification. *Lecture Notes in Computer Science*, 1974:421–428, 2000.

[59] H.A. Kierstead. The Linearity of First-Fit Coloring of Interval Graphs. *SIAM J. Discret. Math.*, 1(4):526–530, 1988.

[60] H.A. Kierstead. A Polynomial Time Approximation Algorithm for Dynamic Storage Allocation. *Discrete Math.*, 87(2-3):231–237, 1991.

[61] K.C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–625, 1965.

[62] D.E. Knuth. *The Art of Computer Programming, volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, USA, 1973.

[63] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, 9(1):25–40, 1989.

[64] V.H. Lai. Hardware-Based Dynamic Storage Management for High-Performance and Real-Time Systems. Master's thesis, Washington University in St. Louis, Washington, December 2003.

[65] P. Larson and M. Krishnan. Memory allocation for Long-Running Server Applications. pages 176 – 185, 1998.

[66] D. Lea. A Memory Allocator. *Unix/Mail*, 6/96, 1996.

[67] J.P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *IEEE Real-Time Systems Symposium*, pages 201–213, 1990.

[68] J.P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterisation and Average Case Behaviour. In *IEEE Real-Time Systems Symposium*, 1989.

[69] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[70] J. Leung and M.L. Merrill. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11(3):115–118, nov 1980.

[71] J. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. volume 2, pages 237–250, December 1982.

[72] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. pages 179–194, 2002.

[73] R. Lougher. JamVM. http://jamvm.sourceforge.net.

[74] M.G. Luby, J. Naor, and A. Orda. Tight Bounds for Dynamic Storage Allocation. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 724–732, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[75] M. Masmano, J. Real, A. Crespo, and I. Ripoll. Distributing Criticality Across Ada Partitions. *Lecture Notes in Computer Science*, 3555:196–207, 2005.

[76] M. Masmano, J. Real, I. Ripoll, and A. Crespo. Running Ada on Real-Time Linux. *Lecture Notes in Computer Science*, 2655:322–333, 2003.

[77] M. Masmano, J. Real, I. Ripoll, and A. Crespo. Extending the capabilities of real-time applications by combining MaRTE-OS and Linux. *Lecture Notes in Computer Science*, 3063:144–155, 2004.

[78] M. Masmano, I. Ripoll, and A. Crespo. Dynamic Storage Allocation for Real-Time Embedded Systems. *Real-Time Systems Symposium, Work-in-Progress Session*, 2003.

[79] M. Masmano, I. Ripoll, and A. Crespo. An Overview of the XtratuM Nanokernel. In *1st Intl. Workshop on Operating Systems Platforms for Embedded Real-Time applications. OSPERT 2005*, 2005.

[80] M. Masmano, I. Ripoll, and A. Crespo. Description of the TLSF Memory Allocator Version 2.0. Technical report, Universidad Politécnica de Valencia, 2005.

[81] M. Masmano, I. Ripoll, and A. Crespo. XtratuM: Un Nanokernel para el Particionado de Sistemas Críticos Empotrados. In *I Congreso Español de Informática (CEDI). I Simposio sobre Sistemas de Tiempo Real (STR).* Ed. Thompson, 2005.

[82] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *16th Euromicro Conference on Real-Time Systems*, pages 79–88, Catania, Italy, July 2004. IEEE.

[83] M. Masmano, I. Ripoll, J. Real, and A. Crespo. Early Experience with an Implementation of Java on RTLinux. In *Sixth Real-Time Linux Workshop*, 2004.

[84] A.K. Mok. Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment. Master's thesis, MIT Laboratory for Computer Science, Cambridge, Massachusetts, May 1983.

[85] A.K. Mok. The design of real-time programming systems based on process models. *Real-Time Systems Symposium, Work-in-Progress Session*, pages 5–17, 1984.

[86] A.K. Mok and M. Dertouzos. Multiprocessor Scheduling in a Hard Real-Time Environment. November 1978.

[87] K.D. Nilsen. Progress in Hardware-Assisted Real-Time Garbage Collection. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 355–379, London, UK, 1995. Springer-Verlag.

[88] K.D. Nilsen. High-Level Dynamic Memory Management for Object-Oriented Real-Time Systems. *SIGPLAN OOPS Mess.*, 7(1):86–93, 1996.

[89] K.D. Nilsen and W.J. Schmidt. Cost-Effective Object Space Management for Hardware-Assisted Real-Time Garbage Collection. *ACM Lett. Program. Lang. Syst.*, 1(4):338–354, 1992.

[90] T. Ogasawara. An Algorithm with Constant Execution Time for Dynamic Storage Allocation. *2nd International Workshop on Real-Time Computing Systems and Applications*, page 21, 1995.

[91] Y.K. Okuji. GRand Unified Bootloader, 1999. http://www.gnu.org-/software/grub/.

[92] Y.K. Okuji, B. Ford, E.S. Boleyn, and K. Ishiguro. The Multiboot Specification, 2002. http://www.gnu.org/software/grub/manual-/multiboot/.

[93] R.R. Oldehoeft and S.J. Allan. Adaptive Exact-Fit Storage Management. *Communications of the ACM*, 28(5):506–511, 1985.

[94] I.P. Page and J. Hagings. Improving the Performance of Buddy Systems. *IEEE Transactions on Computers*, 35(5):18 – 25, 1986.

[95] J.L. Peterson and T.A. Norman. Buddy Systems. *Communications of the ACM*, 20(6):421–431, 1977.

[96] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. Technical Report 1429, Institut de Recherche en Informatique et Systemes Aleatories, 2002.

[97] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. *14 th Euromicro Conference on Real-Time Systems (ECRTS'02)*, page 41, 2002.

[98] E. Puttkamer. A Simple Hardware Buddy System Memory Allocator. *IEEE Transactions on Computers*, 24(10):953 – 957, 1975.

[99] S. Pérez and J. Vila. Augmenting RT-Linux GPL Capabilities with TCP/IP. In *2nd Intl Workshop on Real-time LANs in the Internet Age*, 2003.

[100] R. Rajkumar, L. Sha, and J.P. Lehoczky. An Experimental Investigation of Synchronisation Protocols. *Proceedings 6th Workshop on Real-Time Operating Systems and Software*, pages 11–17, May 1989.

[101] S. Ramos-Thuel and J.P. Lehoczky. On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems. pages 160–171, December 1993.

[102] J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a new Proposal. *Real-Time Systems*, 26(2):161–197, March 2004.

[103] I. Ripoll. Planificación con Prioridades Dinámicas en Sistemas de Tiempo Real Crítico. Master's thesis, Universidad Politécnica de Valencia, Valencia, Spain, June 1996.

[104] I. Ripoll and M. Masmano. Real-Time Dynamic Storage Allocators. http://rtportal.upv.es/rtmalloc.

[105] J.M. Robson. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. *Journal of the Association for Computing Machinery*, 18(3):416–423, 1971.

[106] J.M. Robson. Bounds for Some Functions Concerning Dynamic Storage Allocation. *Journal of the Association for Computing Machinery*, 21(3):491–499, 1974.

[107] J.M. Robson. Worst Case Fragmentation of First-Fit and Best-Fit Storage Allocation Strategies. *Computer Science Theory*, pages 242–244, 1977.

[108] J.M. Robson. Storage Allocation is NP-hard. *Information Processing Letters*, 11(3):119–125, 1980.

[109] W.J. Schmidt and K.D. Nilsen. Performance of a Hardware-Assisted Real-Time Garbage Collector. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 76–85, New York, NY, USA, 1994. ACM Press.

[110] K. Schwan and H. Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads. *IEEE Transactions on Software Engineering*, 18:736–747, August 1992.

[111] F. Sebek. Cache Memories in Real-Time Systems. Technical Report 01/37, Mälardalen Real-Time Research Centre, Sweden, Department of Computer Engineering, Mälardalen University, Sweden, October 2 2001.

[112] R. Sedgewick. *Algorithms in C. Third Edition.* Addison-Wesley, Reading, Massachusetts, USA, 1998.

[113] L. Sha, R. Rajkumar, and J.P. Lehoczky. Solutions for Some Practical Problems in Prioritised Preemptive Scheduling. pages 181–191, December 1986.

[114] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Technical report, CMU-CS-87-181, Computer Science Department, Carnegie-Mellon University, December 1987.

[115] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[116] L. Sha, R. Rajkumar, J.P. Lehoczky, and K. Ramamritham. Mode Change Protocols for Priority-Driven Preemptive Scheduling. *Real-Time Systems*, 1(3):243–264, 1989.

[117] M.A. Shalan. Dynamic Memory Management for Embedded Real-Time Multiprocessor System on a Chip. Master's thesis, Georgia Institute of Technology, Georgia, November 2003.

[118] K.K. Shen and J.L. Peterson. A Weighted Buddy Method for Dynamic Memory Allocation. *Communications of the ACM*, 17(10):558–562, 1974.

[119] J.E. Shore. On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies. *Communications of the ACM*, 18:433–440, 1975.

[120] A. Silberschatz and P.B. Galvin. *Operating Systems.* Addison Wesley Longman, 1999.

[121] B. Sprunt, J.P. Lehoczky, and L. Sha. Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm. pages 251–258, December 1989.

[122] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. pages 27–60, 1989.

[123] M. Spuri and G. Buttazzo. Scheduling Aperidoc Tasks in Dynamic Priority Systems. pages 179–210, March 1996.

[124] W. Srisa-an, C.-T. Dan Lo, and J. M. Chang. A Hardware Implementation of Realloc Function. In *WVLSI '99: Proceedings of the IEEE Computer Society Workshop on VLSI'99*, page 106, Washington, DC, USA, 1999. IEEE Computer Society.

[125] J.A. Stankovic. Tutorial on Hard Real-Time Systems. *IEEE Computer Society Press*, 1987.

[126] J.A. Stankovic. Misconceptions About Real-Time Computing: A serious problem for Next Generation Systems. *IEEE Computer*, 21(10):10–19, 1988.

[127] J.A. Stankovic. Real-Time Computing Systems: The Next Generation. pages 14–37, 1988.

[128] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, 28(6):16–25, 1995.

[129] C.J. Stephenson. Fast Fits: New Methods for Dynamic Storage Allocation. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, volume 17 of *Operating Systems Review*, pages 30–32. ACM Press., 1995.

[130] F. Yellin T. Lindholm. *The Java Virtual Machine Specification*. Addison-Wesley Professional.

[131] A.S. Tanenbaum. *Modern Operating Systems. Second Edition*. Prentice Hall, 2001.

[132] A.S. Tanenbaum and A.S. Woodhull. *Operating Systems. Design and Implementation. Second Edition*. Prentice Hall, 2000.

[133] A. Tenembaum. Memory Utilization Efficiency Under a Class of First-Fit Algorithms. pages 186–190, 1980.

[134] A. Tenembaum and E. Widder. A Comparison of First-Fit Allocation Strategies. pages 875–883, 1978.

[135] T.S. Tia, J.W. Liu, and M. Shankar. Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed-Priority Preemptive Systems. pages 23–43, January 1996.

[136] UPV, SSSA, CTU, CEA, UC, MNIS, and VT. OCERA: Open Components for Embedded Real-Time Applications. 2002. IST 35102 European Research Project. (http://www.ocera.org).

[137] Múltiples voluntarios. Apache: a Web Server. http://www.apache.org.

[138] Múltiples voluntarios. EtherBoot Project. http://www.etherboot.-org/.

[139] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boks. Memory Allocation Policies Reconsidered. *Technical report*, 1995.

[140] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In H.G. Baker, editor, *Proceedings of the International Workshop on Memory Management (IWMM'95), Kinross, Scotland, UK*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer-Verlag, Berlin, Germany, 1995.

[141] D.S. Wise. The Double Buddy System. *Technical report*, 1979.

[142] J. Xu and D.L. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. *IEEE Transactions Software Engineering*, 19(1):70–84, 1993.

[143] Y. Chung and S. Moon. Memory Allocation with Lazy Fits. *SIGPLAN Not.*, 36(1):65–70, 2001.

[144] K. Yaghmour. Adaptative Domain Environment for Operating Systems. http://www.opersys.com/ftp/pub/Adeos/rtosoveradeos.pdf.

[145] V. Yodaiken. The RTLinux Manifesto.

[146] J. Zamorano, J.L. Redondo, C. Blanco, J.I. Tortosa, and J.A. de la Puente. Ejecutivo Cíclico, Manual de Usuario.

[147] B.G. Zorn and D. Grunwald. Empirical Measurements of Six Allocation-Intensive C Programs. *ACM SIGPLAN Notices*, 27(12):71 – 80, 1992.

[148] B.G. Zorn and D. Grunwald. Evaluating Models of Memory Allocation. *ACM Transactions on Modeling and Computer Simulation*, pages 107 – 131, 1994.