

Seminar „Echtzeitfähige malloc()-Implementierungen“



Auftraggeber

Technische Universität Braunschweig
Institut für Datentechnik und Kommunikationsnetze
Prof. Dr.-Ing. Rolf Ernst
Hans-Sommer-Str. 66
38106 Braunschweig

Betreuer

Steffen Stein <stein@ida.ing.tu-bs.de>

Auftragnehmer

Martin Wegner <mw@mroot.net>
Matr.-Nr.: 2864217

Braunschweig, 31. März 2011

Inhaltsverzeichnis

1. Einleitung	3
1.1. Grundlegende Techniken und Begriffe	4
1.1.1. Laufzeit: Landau-Symbole	4
1.1.2. Fragmentierung	5
1.1.3. Splitting und Coalescing[WJNB95, S. 8]	6
1.2. Anforderungen an einen Allokator	7
2. Bisherige Allokatoren	8
2.1. Boundary Tags [Knu98][Kae01]	8
2.2. Sequential Fits	9
2.3. Segregated Free Lists	11
2.4. Buddy Systems	11
2.5. dlmalloc	12
3. Echtzeitfähige Allokatoren	13
3.1. Two Level Segregate Fit (TLSF)	13
3.1.1. Datenstruktur [MRBC08, S. 11]	15
3.1.2. Algorithmen für Allokation und Deallokation	16
3.2. Half-Fit	22
3.3. Evaluation	23
3.3.1. Laufzeitverhalten	24
3.3.2. Fragmentierung	28
4. Fazit	29
Anhang	
A. Listings	30
A.1. Einfache Allokation und Deallokation über Funktionsgrenzen	30
A.2. fls()-Implementierung	30

1. Einleitung

Eine der wichtigsten Ressourcen, die von jedem Computerprogramm benötigt wird, ist der Arbeitsspeicher. Dieser wird dazu genutzt, um den aktuellen Ausführungszustand zu speichern und zu verwalten sowie vom Programm benötigte Daten für einen schnellen Zugriff zu speichern oder zwischenzuspeichern.

Im Bereich der eingebetteten Systeme und hardwarenaher Programmierung finden häufig die Programmiersprachen C bzw. C++ Verwendung, da sie teils hohe Abstraktionen wie etwa objektorientierte Programmierung oder generische Datentypen mit hardwarenahen Aspekten wie z. B. der Möglichkeit beliebig und direkt auf Speicher zuzugreifen.

Ein Programm in diesen Programmiersprachen kann dabei auf drei verschiedene Arten von Speicher zugreifen:

Statischer Speicher wird für die gesamte Laufzeit eines Programms belegt. Er wird beim Programmstart reserviert (auch *allokiert*) und erst mit Beenden des Programms wieder freigegeben (auch *deallokiert*). Dies trifft z. B. auf den Speicher von globalen Variablen und statischen Funktions- bzw. Methodenvariablen und Klassenvariablen zu.

Automatischer Speicher wird durch den Compiler auf dem sog. *Stack* vergeben. Auf dem Stack wird z. B. für Funktionen und Methoden beim Aufruf Speicher für die Argumente in ihrer Größe reserviert. Auch für lokale Variablen innerhalb einer Funktion wird der benötigte Speicher auf dem Stack allokiert. Der so insgesamt allokierte Bereich auf dem Stack wird freigegeben, wenn die Funktion abgearbeitet ist.

Dynamischer Speicher wird zuletzt benötigt, um in einem Programm Daten oder Objekte dynamisch zur Laufzeit erzeugen zu können und diese auch außerhalb der zuvor genannten Lebenszyklen des statischen und automatischen Speichers verwalten zu können. So ist es oft nötig, berechnete Daten über Funktionsgrenzen hinaus performant weiterzugeben, d. h. ohne dass diese bei Übergabe kopiert werden müssten (automatischer Speicher) oder über die gesamte Laufzeit des Programms existieren (statischer Speicher). Letzteres würde auch zu einer sehr ineffizienten Speichernutzung führen, da etwa Zwischenergebnisse nicht für die gesamte Laufzeit benötigt werden und damit nicht permanent gespeichert werden müssen.

So kann dynamischer Speicher über Bibliotheksroutinen von einem Programm zu einem beliebigen Zeitpunkt angefordert und wieder freigegeben werden. In C geschieht dies über die Funktionsaufrufe `malloc()` bzw. `free()`, in C++ über die Operatoren `new` und `delete`.

Die Größe des benötigten Speichers muss bei `malloc()` explizit übergeben werden, beim `new`-Operator wird diese aus dem angegebenen Datentyp abgeleitet, z. B. bei Klassen aus

der Anzahl und der Größe der Instanzvariablen.

Speicheranfragen von laufenden Programmen (Prozessen) über diese Aufrufe werden i. d. R. an das Betriebssystem weitergeleitet, da dieses die physikalischen Ressourcen wie den Arbeitsspeicher verwaltet, aus dem diese Anfragen letztlich bedient werden können.

Das Betriebssystem verwaltet somit einen großen – zunächst freien – Speicherbereich (auch *Memory-Pool* oder *Heap* genannt). Im Betrieb müssen dann Speicheranfragen verschiedener Größe entgegengenommen und aus diesem zur Verfügung stehenden Pool bedient werden und die verbleibenden freien und die belegten Blöcke entsprechend verwaltet werden. Von Anwendungen nicht benötigter und somit wieder freigegebener Speicher muss in den freien Speicher wieder eingegliedert werden. Ein Algorithmus, der die Verwaltung des freien Speichers übernimmt, wird auch als *Allokator* bzw. *Deallokator* oder *Dynamic Storage Allocator* bezeichnet.

Einen schematischen Überblick über einen Speicherpool, aus dem bereits Speicherblöcke an Anwendungen zugeteilt wurden, gibt Abb. 1. Die zugewiesenen Blöcke (dunkel hervorgehoben) wurden dabei aus dem insgesamt zur Verfügung stehenden Pool herausgenommen. Die Programme erhalten einen *Zeiger* oder *Referenz* auf diese Blöcke, um auf diese zuzugreifen. Ein Beispiel für ein solches Programm kann im Anhang A.1 gefunden werden.

Für eine Speicheranfrage bestimmter Größe muss der Allokator also in den zur Verfügung stehenden, freien Blöcken nach einem passenden Block suchen, diesen reservieren und eine Referenz auf den Block an die Anwendung zurückgeben. Bei Freigabe des Blocks muss dieser entsprechend wieder in die Liste der freien Blöcke aufgenommen werden.

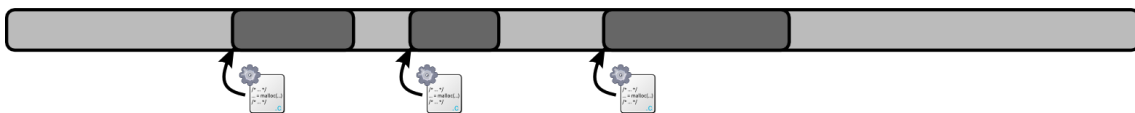


Abbildung 1: Schema: Verwaltung eines Speicherpools mit bereits erfolgten Allokationen.

1.1. Grundlegende Techniken und Begriffe

Im Folgenden sollen nun einige grundlegende Techniken und Begriffe eingeführt werden, die zum Verständnis der Kriterien für einen guten DSA-Algorithmus benötigt werden.

1.1.1. Laufzeit: Landau-Symbole

In der Informatik wird mit den sog. *Landau-Symbolen* das Laufzeit-Verhalten von Algorithmen und Programmen asymptotisch durch Klassen für Funktionen der Eingabegrößen beschrieben. Verschiedene Landau-Symbole geben dabei untere oder obere Schranken oder beides an.

Für die folgenden Betrachtungen werden zwei der Landau-Symbole benutzt: $\Theta(g)$ und $\mathcal{O}(g)$.

- In $\Theta(g)$ liegen Funktionen f , die durch die Funktion g sowohl von unten als auch von oben beschränkt werden. D. h. die Funktionen f wachsen *mindestens* so schnell wie g und *auch höchstens* so schnell wie g .

Mathematisch lässt sich das wie folgt beschreiben:

$$f \in \Theta(g) \Leftrightarrow \exists c_0 > 0 \quad \exists c_1 > 0 \quad \exists x_0 \quad \forall x > x_0 : \\ c_0 \cdot |g(x)| \leq |f(x)| \leq c_1 \cdot |g(x)|$$

Dies bedeutet, dass sich ein konstantes x_0 finden lässt, ab dem für alle größeren Werte von x alle (absoluten) Funktionswerte durch Funktionswerte von g derart beschränkt sind, dass ein konstantes Vielfaches des Funktionswertes $g(x)$ immer eine untere Schranke und ein anderes immer eine obere Schranke darstellt.

- Für Funktionen f in $\mathcal{O}(g)$ wird dann nur noch die obere Schranke gefordert, d. h. g beschränkt immer f ab einem bestimmten x_0 für alle Funktionswerte nach oben. Formal lässt sich das aufschreiben zu:

$$f \in \mathcal{O}(g) \Leftrightarrow \exists c > 0 \quad \exists x_0 \quad \forall x > x_0 : \\ |f(x)| \leq c \cdot |g(x)|$$

Die Laufzeit von Algorithmen wird i. A. über die Anzahl der nötigen Elementarschritte angegeben. Diese Schritte sind oft abhängig von der Größe der Eingabe bzw. Eingabedatenstrukturen.

So ist zum Beispiel die Laufzeit des Sortieralgorithmus' Quick-Sort im besten und im durchschnittlichen Fall beschränkt durch $\mathcal{O}(n \log n)$, jedoch im schlechtesten Fall („worst case“) lediglich durch $\mathcal{O}(n^2)$, wobei n die Anzahl der zu sortierenden Elemente angibt. Quick-Sort lässt sich somit nicht durch die gleiche Funktion von unten und nach oben beschränken.

Im Vergleich dazu ist bei dem Sortieralgorithmus Heap-Sort für jeden Fall die Laufzeit durch $\Theta(n \log n)$ beschränkt. D. h. Heap-Sort braucht nie mehr als $n \log n$ Schritte zum Sortieren einer Liste, aber auch niemals weniger.

1.1.2. Fragmentierung

Neben der Laufzeit ist die Fragmentierung eine wichtige Kenngröße für DSA-Algorithmen: Sie kennzeichnet, wie viel von dem insgesamt verfügbaren Speicher tatsächlich durch den Allokator den Anwendungen zur Verfügung stehen kann.

Üblicherweise wird Fragmentierung in zwei verschiedene Arten unterschieden (vgl. [WJNB95, S. 8]): *interne* und *externe* Fragmentierung. Zur Verdeutlichung stellt Abb. 2 schematisch eine mögliche Speicherbelegung dar.

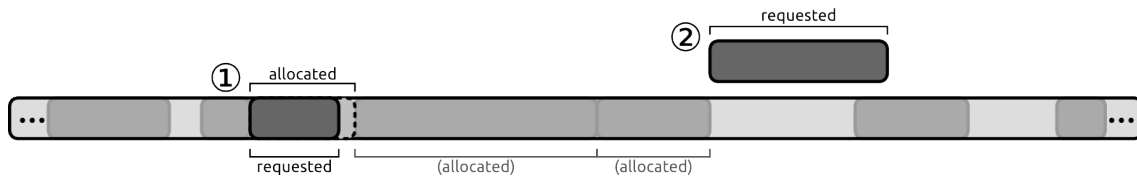


Abbildung 2: Schema: Interne (1) und externe Fragmentierung (2).

Zu einer internen Fragmentierung kommt es, wenn ein Allokator-Algorithmus einer Speicheranfrage von einer Anwendung einen größeren Block zuteilt, als diese angefordert hat (vgl. Abb. 2, (1)). Der überschüssige Speicher wird dann nicht durch die Anwendung genutzt, kann aber auch nicht durch den Allokator anderweitig vergeben werden. Dass ein Allokator so implementiert wird, dass dieser einen (leicht) größeren Block vergibt als nötig wäre, kann verschiedene Ursachen haben (s. [WJNB95, S. 9]): Zum einen können Laufzeit-Anforderungen oder eine gewünscht geringe Komplexität dazu führen, dass ein Allokator nicht genau passende Blöcke vergibt, sondern z. B. nur Blöcke bestimmter Größenklassen (vgl. *Segregated Free Lists*, 2.3). Zum anderen kann es bei einem Allokator gewünscht sein, den Block in einen passenden und einen kleinen aufzuteilen, etwa um die externe Fragmentierung zu verringern (s. *Splitting*, 1.1.3).

Von externer Fragmentierung spricht man, wenn eine Anwendung einen Speicherblock von einer Größe anfordert, die größer ist als die Größen aller freien Blöcke (vgl. Abb. 2, (2)). In diesem Fall kann der Allokator die Speicheranfrage nicht bedienen, obwohl insgesamt genügend Speicher frei wäre, aber nicht genügend zusammenhängender Speicher zur Verfügung steht.

Insgesamt wird Fragmentierung durch sich über die Zeit veränderndes Verhalten einer Anwendung bzgl. Speicheranforderungen und -freigaben verursacht (s. [WJNB95, S. 22]). Insbesondere sind dabei Anfragespitzen mit unterschiedlichen Anfragegrößen am problematischsten. Fragmentierung lässt sich zudem am besten untersuchen, indem die Anforderungen von tatsächlichen Programmen aufgezeichnet werden (sog. *Traces*), und diese als Daten-Grundlage für Anforderungen an einen DSA-Algorithmus in Simulationen genutzt werden (s. ebd.)

1.1.3. Splitting und Coalescing [WJNB95, S. 8]

Zwei wichtige Teilalgorithmen eines DSA-Algorithmus' sind *Splitting* und *Coalescing* von freien Speicherblöcken.

Splitting bezeichnet dabei das Vorgehen, einen freien Block für eine eingehende Speicheranfrage in einen passenden und einen weiterhin freien, kleineren Block aufzuteilen (vgl. Abb. 3, (1)). Hierbei kann auch ein z. B. aus Effizienzgründen auf die nächsthöhere Größenklasse aufgerundeter Block als passend gelten. Mit Splitting kann u. a. der internen Fragmentierung

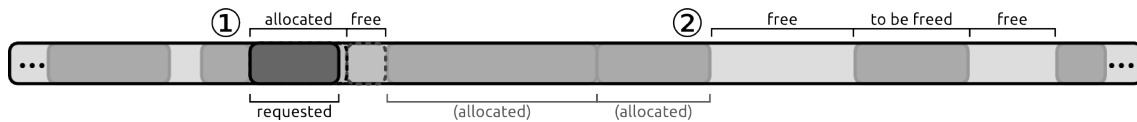


Abbildung 3: Schema: Splitting (1) und Coalescing (2).

begegnet werden.

Beim *Coalescing* wird ein freizugebender Speicherblock mit seinen evtl. freien Nachbarn wieder zu einem gemeinsamen, großen Block zusammengefügt (vgl. Abb. 3, (2)). Auf diese Weise kann sichergestellt werden, dass externe Fragmentierung möglichst vermieden wird, da zu jedem Zeitpunkt größtmögliche freie Speicherblöcke zur Verfügung stehen. Unabhängig davon bleibt durch das Verhalten des Programms ausgelöste externe Fragmentierung weiterhin bestehen.

Coalescing kann zudem zu unterschiedlichen Zeitpunkten durchgeführt werden: Sofort bei Freigabe eines Blocks (*immediate*) oder aber verzögert (*deferred*), z. B. zeitgesteuert oder wenn ein entsprechend großer Block benötigt wird.

1.2. Anforderungen an einen Allokator

Mit den bisher eingeführten Begriffen lassen sich damit folgende Kriterien an einen Allokator-Algorithmus formulieren:

- *Speicheranfragen unterschiedlicher Größe* von verschiedenen Anwendungen müssen bedient werden.
- Mindestens im durchschnittlichen Fall soll der Algorithmus eine *kurze, beschränkte Laufzeit* aufweisen.
- Durch Minimierung von Effekten wie interner und externer Fragmentierung soll der größte Teil des verwalteten Speichers tatsächlich den Anwendungen zur Verfügung stehen, so dass eine *hohe Speichereffizienz* erreicht wird.

2. Bisherige Allokatoren

In diesem Kapitel soll ein kurzer Überblick über Grundideen bisheriger Dynamic Storage Allocator Algorithmen gegeben werden. Es werden drei Klassen an Algorithmen (vgl. [WJNB95]) vorgestellt: *Sequential Fits* (Kap. 2.2), *Segregated Free Lists* (Kap. 2.3) und *Buddy Systems* (Kap. 2.4). Am Ende wird in Kap. 2.5 der bekannte hybride Allokator *dmalloc* vorgestellt, der sowohl das Konzept Sequential Fits als auch Segregated Free Lists implementiert.

Viele der Konzepte verwenden sog. *Boundary Tags*, um freie – teils auch belegte – Speicherblöcke in Listen zu verwalten. Daher wird im Folgenden zunächst dieses Konzept eingeführt.

2.1. Boundary Tags [Knu98][Kae01]

Den *Boundary Tags* liegt die Idee zugrunde, dass die Informationen zur Organisation der Listen von Blöcken in den Speicherblöcken selbst verwaltet werden können. Im Fall von freien Blöcken ist der Speicher nicht belegt und somit kann dieser optimal dazu genutzt werden. Bei belegten Blöcken wird ein leichter Overhead – üblicherweise nicht größer als zwei Maschinenwörter (entspricht etwa auf 32bit-Systemen 8 Bytes Overhead) in Kauf genommen.

Tabelle 1 gibt eine mögliche Struktur für einen Boundary Tag eines *freien* Speicherblocks an. Hierbei wird zunächst die Größe des vorherigen Blocks gefolgt von der eigenen Größe des Blocks

Freier Block	
Größe vorheriger Block	
Größe	Vorheriger Block in Benutzung?
Zeiger auf vorherigen freien Block (optional)	
Zeiger auf folgenden freien Block	
... weiterer nicht genutzter Speicher in diesem Block ...	

Tabelle 1: Mögliches Format für einen Boundary Tag eines freien Blocks (nach [Knu98] und [Kae01]).

gespeichert. Auf diese Weise können anhand der physikalischen Blockadresse die Adressen der benachbarten Blöcke auf einfache Weise berechnet werden, indem die Größen der Blockadresse abgezogen bzw. hinzu addiert werden. Dies kann u. a. zum *Immediate Coalescing* verwendet werden und diese Operation deutlich vereinfachen. Da die unteren Bits der Größe eines Blocks i. d. R. nicht benötigt werden, da Speicherblöcke z. B. aus Gründen des Speicher-Alignments immer eine Größe von einem Vielfachen von 4 oder 8 Bytes haben müssen, können diese als Status-Flags mit unterschiedlichen Bedeutungen genutzt werden. Im hier vorgestellten Format wird das unterste Bit der Größe genutzt, um anzuzeigen, ob der vorherige Block frei ist. Auf

die Größenangaben folgt mindestens ein Zeiger auf den nächsten freien Block für eine einfach verkettete Liste. Ergänzt werden kann dies durch einen weiteren Zeiger auf den vorherigen freien Block für doppelt verkettete Listen. Insgesamt wird durch diesen Boundary Tag eine minimale Blockgröße von 16 Bytes auf einem 32bit-System impliziert.

Tabelle 2 gibt das Format eines Boundary Tags für einen belegten Speicherblock an. Die

Belegter Block	
Größe vorheriger Block	
Größe	Vorheriger Block in Benutzung?
... tatsächlicher Speicherinhalt einer Anwendung ...	

Tabelle 2: Mögliches Format für einen Boundary Tag eines belegten Blocks (nach [Knu98] und [Kae01]).

Felder haben hier die gleiche Bedeutung wie zuvor bei den Boundary Tags freier Blöcke. Der Speicher für die Zeiger auf einen vorherigen bzw. nächsten freien Block fallen weg und ihr Platz im Speicherblock kann somit bei belegten Blöcken zusätzlich der Anwendung zur Verfügung gestellt werden.

2.2. Sequential Fits

Allokatoren dieses Konzepts verwenden i. d. R. eine einzige Liste, in der alle freien Speicherblöcke verwaltet werden. Diese Liste kann über Boundary Tags verwaltet werden und einfach oder doppelt verkettet und evtl. auch zirkulär sein (s. [WJNB95]). Unabhängig von der implementierten Suchstrategie kann die Liste dabei auf verschiedene Arten sortiert werden: So ist eine Sortierung der Liste nach Größe der in ihr gespeicherten freien Blöcke (vgl. etwa dmalloc, Kap. 2.5) oder aber nach deren physikalischer Speicheradresse denkbar (vgl. [WJNB95]) oder freie Blöcke können in der Liste nach den Prinzipien *last in, first out* (LIFO) bzw. *first in, first out* (FIFO) organisiert werden (s. ebd.). Es werden Splitting und Immediate Coalescing angewandt.

Für die Suche nach einem freien Block in dieser Liste können dabei die folgenden verschiedenen Strategien angewandt werden (vgl. [WJNB95]):

Best Fit Diese Strategie beginnt die Suche nach einem passenden freien Block am Beginn der Liste und sucht nach einem möglichst genau passenden freien Speicherblock. Die Suche wird abgebrochen, sobald ein entsprechender Block gefunden wurde. Im Allgemeinen wird mit diesem Algorithmus die Fragmentierung deutlich gering gehalten, während die

Laufzeit mit der Länge der Liste der freien Blöcke skaliert und bei entsprechend langen Listen schlecht ausfallen kann.

First Fit Hier wird ab Anfang der Liste der freien Blöcke ein passender freier Block gesucht und die Suche beim ersten, der groß genug ist, um die Anfrage zu erfüllen, abgebrochen. Der zugeteilte Block wird evtl. gesplittet, wenn dieser größer ist als angefragt.

Bei dieser Strategie gibt es verschiedene Möglichkeiten freizugebende Blöcke in die Liste einzufügen: Ein freizugebender Block kann so immer zu Beginn der Liste eingefügt werden oder aber die Liste wird nach der physikalischen Adresse der Blöcke sortiert (vgl. [WJNB95, Kap. 3.4]). Während bei ersterem die Deallokation eines Blocks schnell durchgeführt werden kann, produziert die zweite Variante eine deutlich geringere Fragmentierung und wird in der Praxis häufiger verwendet (s. ebd.).

Next Fit Um einen freien Block mit dieser Strategie zu suchen, wird ein *laufender Zeiger* auf der – üblicherweise zirkulären – Liste der freien Blöcke implementiert. Dieser steht immer an der Stelle der Liste, an der zuletzt eine Speicheranfrage erfolgreich bedient werden konnte. Sukzessive wird jede weitere Suche von einem Block von dieser Stelle an begonnen. Ziel dieser Strategie ist es, die Laufzeit der Suche zu verringern (vgl. [WJNB95, Kap. 3.4]). Der Nachteil ist, dass durch die Natur des laufenden Zeigers zeitlich nahe Speicheranfragen aus physikalisch weit voneinander entfernten Bereichen bedient werden können. Werden diese Speicherblöcke in der Anwendung zeitlich ebenfalls nah beieinander verwendet, führt diese geringe Lokalität der Blöcke zu häufigen sog. Cache Misses und beeinträchtigt die Laufzeit so negativ (s. ebd.).

Worst Fit Diese Strategie verhält sich genau gegenteilig zu dem zuvor genannten Best Fit. Es wird versucht, den größtmöglichen Block einer Speicheranfrage zuzuteilen, der zuvor gesplittet wird und dessen Rest wieder in die Liste der freien Blöcke eingefügt wird. Auf diese Weise soll sichergestellt werden, dass der frei verbleibende Rest eines gesplitteten Blocks möglichst groß ist, so dass Reste kleiner, nicht nutzbarer Größen vermieden werden.

Isoliert betrachtet schneidet diese Strategie bei Tests mit synthetisch generierten Speicheranfragen deutlich schlechter ab als die bisher vorgestellten (vgl. [WJNB95, Kap. 3.5]), grundsätzlich aber kann Worst Fit in Kombination mit anderen Verfahren Anwendung finden (s. ebd.).

Insgesamt kommen die Autoren in [WJNB95, Kap. 3.5] zu dem Ergebnis, dass Implementierungen mit den Strategien *Best Fit* sowie *First Fit* mit einer nach physikalischer Adresse sortierten Liste gute Ergebnisse zeigen. Ebenso wird vermutet, dass eine First-Fit-Variante mit

einer FIFO-Liste ähnlich gute Resultate aufweisen kann, ausführliche Studien zu dieser oder anderen Strategien und ihren Varianten fehlen aber bisher (vgl. ebd.).

2.3. Segregated Free Lists

Dieses Konzept verwendet im Allgemeinen unterschiedliche Listen für Speicherblöcke unterschiedlicher Größen. Dabei gibt es sowohl verschiedene Möglichkeiten, die Größen auszuwählen und anzuordnen (z. B. immer in Zweierpotenzen 2^x) sowie exakte Größen oder Größenklassen zu benutzen. Bei ersterem enthält jede Liste freie Speicherblöcke von genau der gleichen Größe. Anfragen werden auf die nächstgrößere Liste aufgerundet und aus dieser bedient. Werden in den Listen Blöcke bestimmter Größenklassen gespeichert, z. B. Blöcke der Größen 256 bis 512 Bytes, so können die Sequential Fit Algorithmen (vgl. Kap. 2.2) verwendet werden, um einen passenden Block innerhalb dieser Listen zu finden.

2.4. Buddy Systems

Bei den Buddy Systems wird der initiale Speicherbereich (Heap) bei einer Speicheranfrage zunächst in zwei Bereiche aufgeteilt. Diese werden jeweils sukzessive weiter in zwei Bereiche unterteilt, bis auf diese Weise ein Block der minimal passenden Größe für die Anfrage erzeugt werden konnte (vgl. [WJNB95, Kap. 3.7]). Auf diese Weise entsteht eine Baumstruktur mit möglichen freien Blöcken auf jeder Ebene als Blätter.

Dabei können die Bereiche auf jeder Ebene nach unterschiedlichen Schemata aufgeteilt werden: Bei den sog. *Binary Buddies* wird jede Ebene in genau zwei gleich große Speicherblöcke unterteilt. Die Blöcke haben somit immer eine Größe von 2^i , wodurch Adressberechnungen, z. B. zur Bestimmung benachbarter oder übergeordneter Blöcke, stark vereinfacht werden (s. ebd.). Gleichzeitig entsteht aber durch die fest vorgegebenen Block-Größen von $b = 2^i$ Bytes i. d. R. eine hohe interne Fragmentierung, da Speicheranfragen einer bestimmten Größe immer auf die nächsthöhere Blockgröße b aufgerundet werden müssen (vgl. [WJNB95, Kap. 3.7]).

Dieses Problem der hohen internen Fragmentierung kann mit anderen Varianten wie zum Beispiel den *Fibonacci Buddies* reduziert werden: In dieser Variante werden die Blöcke immer im Verhältnis zweier Fibonacci-Zahlen aufgeteilt, so dass gemäß der Definition der Fibonacci-Zahlen die Größen F_{i-1} und F_{i-2} zweier Blöcke auf einer Ebene immer die Größe des übergeordneten Blocks $F_i = F_{i-1} + F_{i-2}$ ergeben. Dadurch, dass Fibonacci-Zahlen näher beieinander liegen als etwa Zweierpotenzen 2^i , kann die interne Fragmentierung reduziert werden, da Aufrundungen auf die nächste Blockgröße geringer ausfallen. Gleichzeitig wird ein höherer Aufwand zur Berechnung der Position eines Blocks in der Datenstruktur in Kauf genommen. Dieser nö-

tige Mehraufwand im Vergleich zu Binary Buddies kann aber z. B. durch die Verwendung von Look-up-Tabellen vermindert werden (vgl. [WJNB95, Kap. 3.7]).

2.5. dlmalloc

dlmalloc [Lea00] ist ein von Doug Lea entwickelter *hybrider Allokator*, der Segregated Lists – in dlmalloc als *Bins* bezeichnet – und Sequential Fit vereinigt. dlmalloc findet in abgewandelter Form z. B. Anwendung in der *GNU C Library* (glibc) unter Linux.

Ziele der Implementierung des Allokators sind eine gute durchschnittliche Laufzeit bei gleichzeitig geringer Fragmentierung und möglichst hoher Lokalität der Speicherblöcke (vgl. [Lea00]).

dlmalloc verwendet für Speicheranfragen bis 512 Bytes sog. *Exact Bins*, die immer Speicherblöcke fester Größen enthalten. Die Größen liegen immer 8 Bytes auseinander, so dass sich Bins für die Größen 16, 24, 32, ... ergeben. Für Speicherblöcke, die größer als 512 Bytes sind,

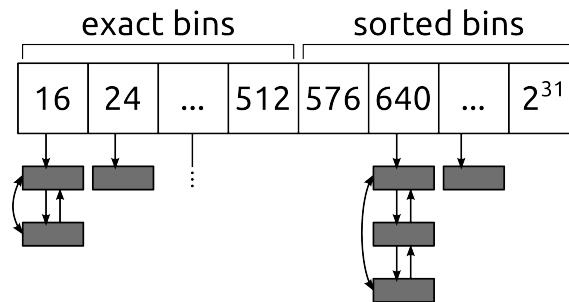


Abbildung 4: Schematische Darstellung der dlmalloc-Datenstruktur zur Verwaltung freier Blöcke mit *Exact Bins* und *Sorted Bins* für Größenklassen nach [Lea00].

werden sog. *Sorted Bins* verwendet, die logarithmisch verteilt sind (s. ebd.) und somit Blöcke einer Größenklasse enthalten. Die in diesen Bins enthaltenen Listen von freien Speicherblöcken sind nach Größe der Blöcke sortiert. Für die Suche wird die *Best Fit* Strategie implementiert, um die Fragmentierung gering zu halten. Die Datenstruktur, die dlmalloc somit zum Verwalten von freien Blöcken benutzt, ist in Abb. 4 schematisch dargestellt. Zur Verwaltung der Listen werden ebenfalls Boundary Tags (vgl. Kap. 2.1) eingesetzt. Ist zur Bedienung einer Speicheranfrage das Splitten eines Blocks nötig, so erzeugt dlmalloc durch Splitten eines deutlich größeren Blocks weitere kleine freie Blöcke dieser Größe (sog. *Preallocation*, vgl. [Lea00]). Weiter führt dlmalloc ein *Deferred Coalescing* durch.

3. Echtzeitfähige Allokatoren

Echtzeit-Anwendungen wie etwa Video-Streaming und -Decodierung oder etwa regeltechnische Systeme z. B. in Autos, Robotern, etc. sind absolut zeitkritisch. D. h. Anwendungen unterliegen hier nicht nur den Anforderungen der korrekten, fehlerfreien Ausführung sondern auch Zeitanforderungen, die unbedingt (harte Echtzeit) oder möglichst immer (weiche Echtzeit) eingehalten werden müssen.

Um diese Anforderungen einhalten zu können, muss die Laufzeit von den Komponenten einer Echtzeit-Anwendung auch im schlechtesten Fall nach oben begrenzt sein.

Bei den hier bisher vorgestellten Dynamic Storage Allokatoren war dies nicht der Fall: Allokation und/oder Deallokation laufen z. B. in linearer, nicht begrenzter Zeit. Design-Entscheidungen wie *Deferred Coalescing* wie bei *dmalloc* (vgl. [Lea00]) führen zwar zu einer *amortisierten*, konstanten Laufzeit, weisen aber durch das zeitverzögerte Zusammenlegen von freien Blöcken zu nicht vorhersagbaren Zeitpunkten eine unsichere Komponente auf, so dass eine Laufzeitgarantie nicht mehr abgegeben und Echtzeit-Anforderungen (Deadlines) nicht mehr eingehalten werden können.

Gleichzeitig sind bei Echtzeitsystemen auch an die Effizienz der Speichernutzung höhere Anforderungen zu richten, da meist nur begrenzter Speicher zur Verfügung steht und Anwendungen auf diesen Systemen oft deutlich länger laufen als gewöhnliche Computerprogramme (vgl. [MRRC04, S. 2]).

Im Folgenden werden zwei Algorithmen vorgestellt und bzgl. ihrer allgemeinen Eigenschaften sowie den besonderen Anforderungen von Echtzeit-Systemen evaluiert: *Two Level Segregate Fit* (TLSF) in Kap. 3.1 sowie *Half Fit* in Kap. 3.2.

3.1. Two Level Segregate Fit (TLSF)

Der *Two Level Segregate Fit*-Algorithmus wurde daher so entwickelt, dass Echtzeit-Anforderungen unter allen Bedingungen nachgekommen werden kann. Als Design-Ziele (vgl. [MRRC04], [MRBC08]) wurden daher verfolgt:

- Bekannte, beschränkte Laufzeit von Allokation und Deallokation im schlechtesten Fall. Zudem soll diese Zeit konstant beschränkt sein, so dass die Laufzeit des Algorithmus' durch $\Theta(1)$ beschränkt ist.
- Zusätzlich wird gefordert, dass die tatsächliche Laufzeit – z. B. gemessen in benötigten CPU-Zyklen bzw. -Instruktionen – mindestens mit derer konventioneller DSA-Algorithmen vergleichbar ist und somit auch im mittleren bzw. allgemeinen Fall nicht ineffizienter als die bisheriger DSA-Implementierungen ist.

- Speicheranfragen sollen erst fehlschlagen, wenn der physikalisch vorhandene Speicher erschöpft ist. Dazu ist vor allem die interne Fragmentierung niedrig zu halten.

Weiterhin wurden die folgenden Annahmen getroffen (vgl. [MRCR04, S. 4]), die ebenfalls größtenteils auf Echtzeitsysteme zutreffen, deren Ausschluss aber auch für andere Anwendungen nicht von Nachteil ist:

- Es steht nur eine vergleichsweise kleine Menge an zu verwaltendem Speicher zur Verfügung.
- Auf Ebene des Allokators werden keine Schutzmechanismen eingebaut, die vor böswilligem Zugriff einer Anwendung auf fremden Speicher schützen.
- Spezielle Hardware-Unterstützung wie etwa eine *Memory Management Unit* (MMU) oder die System-Calls `brk()` bzw. `sbrk()`¹ stehen nicht zur Verfügung und werden daher auch nicht von TLSF vorausgesetzt und benutzt.

Aus diesen Designzielen und Annahmen leiten sich eine Reihe von Konsequenzen für die Implementierung von TLSF ab (vgl. [MRCR04, S. 4f]):

- TLSF implementiert eine optimale *Best-Fit*-Strategie nur annähernd, auch als *Good-Fit* bezeichnet. Auf diese Weise können für Speicheranfragen Blöcke mit (annähernd) der genauen Größe zurückgegeben werden, was die interne Fragmentierung minimiert.
- Anforderungen aller verschiedenen möglichen Größen werden auf die gleiche Art behandelt, um auszuschließen, dass die Laufzeit durch verschiedene Strategien für verschiedene Blockgrößen schwierig zu bestimmen ist (vgl. `dmalloc`, 2.5).
- Die minimale Blockgröße, die von einem Programm angefordert werden kann, liegt bei 16 Bytes. Auf diese Weise können *Boundary Tags* (vgl. 2.1) verwendet werden, um freie und belegte Blöcke in doppelt verketteten Listen zu verwalten, um auf diese effizient zugreifen zu können. Da in der Regel nahezu alle Anfragen an den dynamischen Speicher für komplexe Datenstrukturen und nicht für primitive Datentypen erfolgen, ist dies keine Einschränkung, die zu einer höheren internen Fragmentierung führt.
- Freizugebende Blöcke werden sofort mit evtl. freien benachbarten Blöcken zu einem großen freien Block zusammengefasst (*Immediate Coalescing*).
- Allokierter Speicher bzw. freizugebender Speicher wird nicht (z. B. mit Nullen) initialisiert oder überschrieben. Hierdurch wird eine von der Größe – und damit von Eingabedaten abhängige – Laufzeit vermieden.

¹Mit `brk()/sbrk()` kann ein Prozess – wie etwa ein Allokator – dynamisch seinen eigenen Speicherbereich vergrößern und so vom Betriebssystem mehr Speicher anfordern.

3.1.1. Datenstruktur [MRBC08, S. 11]

TLSF benutzt als Datenstruktur *Segregated Lists* (vgl. 2.3), die auf zwei Ebenen angeordnet sind, um freie Blöcke zu verwalten. Für jede Ebene wird ein Index eingeführt: der *First Level Index* (FLI) und der *Second Level Index* (SLI) respektive.

Die initiale Größe des Speicherpools muss immer durch eine Zweierpotenz darstellbar sein, beträgt also $\mathcal{H} = 2^t$. Auf der ersten Ebene werden über den First Level Index Speicherblöcke der jeweils gemeinsamen Größenklasse $[2^{fli}, 2^{fli+1}[$ mit $fli \leq t$ verwaltet. Der Second Level Index für den \mathcal{J} Bits verwendet werden, unterteilt diese Größenklassen auf der zweiten Ebene nun linear in $2^{\mathcal{J}}$ gleich große Bereiche $[2^{fli} + 0 \cdot 2^{\mathcal{J}}, 2^{fli} + 1 \cdot 2^{\mathcal{J}}[, [2^{fli} + 1 \cdot 2^{\mathcal{J}}, 2^{fli} + 2 \cdot 2^{\mathcal{J}}[, \dots$. Auf diese Weise werden kleinere Größenklassen feingranularer unterteilt als größere, was bei typischen Anwendungsszenarien von Vorteil ist, da kleinere Blöcke am häufigsten angefordert werden.

Jede Kombination eines First und Second Level Index (fli, sli) verweist dabei auf den Beginn einer Segregated List, die nur freie Blöcke mit einer Größe in genau dieser Größenklasse enthält. Die freien Blöcke dieser Liste sind mittels Boundary Tags doppelt verkettet organisiert.

Beispiel 3.1.1. *Im Folgenden wird für die Beispiele eine Heapgröße von $\mathcal{H} = 2^9 = 512$ Bytes angenommen, sowie ein Second Level Index mit $\mathcal{J} = 3$ Bits, so dass sich $2^3 = 8$ Second-Level-Speicherbereiche ergeben.*

Für eine Beispielanforderung eines Speicherblocks der Größe $r = 67_{10}$ Bytes ergeben sich somit die Indizes wie folgt:

$$r \in [64, 128[= [2^6, 2^7[\\ \Rightarrow fli = 6$$

$$[64, 128[\rightarrow [64, 72[, [72, 80[, [80, 88[, \dots \\ \Rightarrow r \in [64, 72[\\ \Rightarrow sli = 0$$

Formal können die Indizes der Listen, in denen ein Block der Größe r zu verwalten ist, wie folgt berechnet werden (nach [MRR⁺07]):

$$\text{mapping_insert}(r) = \begin{cases} fli & = \lfloor \log_2(r) \rfloor \\ sli & = \left\lfloor \frac{r - 2^{fli}}{2^{fli - \mathcal{J}}} \right\rfloor \end{cases}$$

Die Boundary Tags haben in TLSF das folgende Format (vgl. [MRCR04]):

Freier Block		
Größe	L	F
Zeiger auf physikalisch vorherigen Block		
Zeiger auf vorherigen freien Block		
Zeiger auf folgenden freien Block		

Belegter Block		
Größe	L	F
Zeiger auf physikalisch vorherigen Block		

Bei freizugebenden Blöcken kann die Adresse des physikalisch folgenden Blocks durch die Adresse des betrachteten Blocks addiert mit seiner Größe ermittelt werden. Der physikalisch vorhergehende Block kann über den entsprechenden Zeiger ebenfalls ermittelt werden, so dass ein sofortiges Coalescing möglich wird.

Bei freien Blöcken werden zusätzlich noch die Zeiger der doppelt verketteten Listen gespeichert. L und F sind zwei Flags (1 Bit): L gibt an, ob es sich um den physikalisch letzten Block handelt und F gibt an, ob der Block frei ist. Hierdurch ergibt sich die Beschränkung, dass die Größe vergebener Speicherblöcke immer ein Vielfaches von 4 Bytes sein muss.

3.1.2. Algorithmen für Allokation und Deallokation

Zunächst ist hierbei nicht ersichtlich wie die Indizes effizient in konstanter Zeit berechnet werden können. Betrachtet man die Binärdarstellung einer gegebenen Größe, so wird jedoch deutlich, dass der First und Second Level Index effizient mittels Bitsuchoperationen und Binäroperatoren berechnet werden können. Der First Level Index wird dabei über das höchst signifikante gesetzte Bit der Binärdarstellung ermittelt, der Second Level Index über den Wert der darunter liegenden \mathcal{J} Bits.

Beispiel 3.1.2. So kann für die Größe $r = 67_{10}$ abgelesen werden:

$$r = 67_{10} = \overbrace{1}^{fli=6} \underbrace{000}_{sli=0} 011_2$$

Das höchst signifikante gesetzte Bit lässt sich dabei effizient über die Prozessor-Instruktion `fls` ermitteln oder, falls diese nicht vorhanden ist, über eine konstante Zahl von geeigneten `if-`

Abfragen². Der Second Level Index kann über Bitverschiebe-Operatoren und die Subtraktion bestimmt werden.

Die Berechnung der Indizes lässt sich so in konstanter Zeit ($\mathcal{O}(1)$ bzw. $\Theta(1)$) implementieren:

```

1 void mapping_insert(unsigned int r,
2                   unsigned int *fli, unsigned int *sli) {
3     *fli = fls(r); // O(1)
4     *sli = (r >> (*fli - SLI_SIZE)) - (2 << SLI_SIZE); // O(1)
5 }

```

Zur Verwaltung der Segregated Lists werden nun Bitmaps genutzt, in denen ein gesetztes Bit für einen gegebenen First Level Index resp. Second Level Index angibt, ob freie Blöcke der entsprechenden Größenklasse vorhanden sind. Auf diese Weise kann auch die Suche nach freien Blöcken über Bitoperationen realisiert werden.

Beispiel 3.1.3. Somit ergibt sich für das Beispiel nach den bisherigen Definitionen die Datenstruktur zur Verwaltung von freien Blöcken wie in Abb. 5 dargestellt. Dort ist eine mögliche Situation nach bereits erfolgten Allokationen und Deallokationen dargestellt.

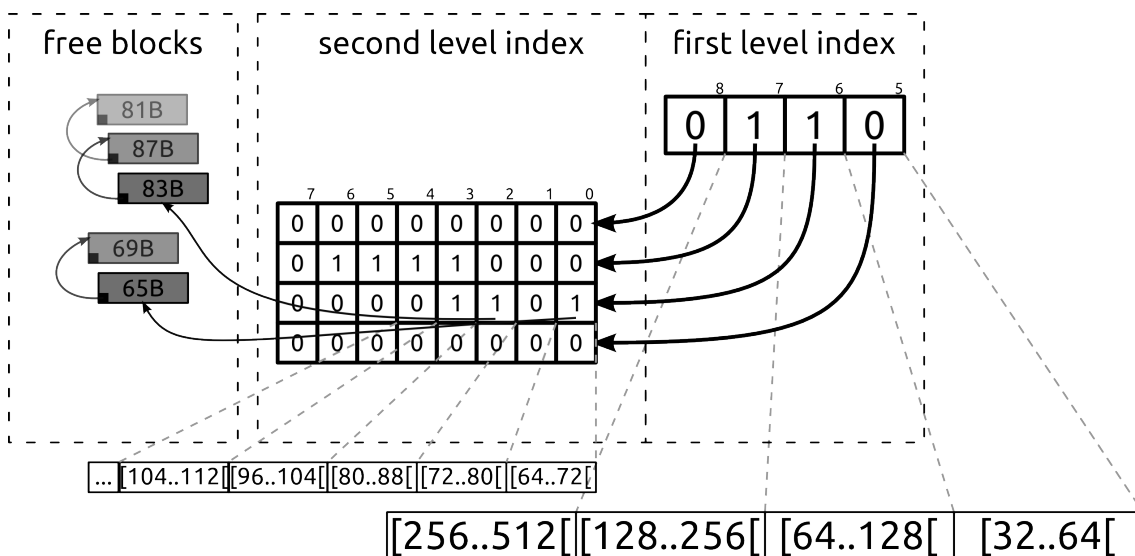


Abbildung 5: Schematische Darstellung der TLSF-Datenstruktur zur Verwaltung freier Blöcke.

Für die Größe $r = 67$ kann so über den First Level Index $fli = 6$ und den Second Level Index $sli = 0$ durch das gesetzte Bit ermittelt werden, in welcher Liste freie Blöcke der gleichen Größenklasse zu finden sind. Dies kann z. B. benutzt werden, um einen freien Block dieser Größe der Liste hinzuzufügen. Für die Suche bei einer Speicheranforderung einer Anwendung nach

²Vgl. dazu z. B. die `fls()`-Implementierung im Linux Kernel, <http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=include/asm-generic/bitops/fls.h;h=0576d1f42f43fc34fb5efa8e27969afc5dbdf0b4;hb=HEAD>, auch im Anhang unter A.2

einem passenden Block eignet sich dies jedoch noch nicht, da die Liste auch Blöcke kleinerer Größen als der angeforderten enthalten kann. In der Abbildung ist dies z. B. direkt für den ersten Block der Liste der Fall.

Da für Speicheranfragen einer gegebenen Größe die Liste der freien Blöcke in dieser Größenklasse auch kleinere Blöcke enthalten kann, wird eine Anfrage immer auf die nächstgrößere Klasse aufgerundet. Somit ergibt sich für den First und Second Level Index folgende formale Definition zur Bestimmung der Größenklasse, aus der *alle* freien Blöcke – und somit auch der Kopf der Liste, auf den in konstanter Zeit zugegriffen werden kann – für die angefragte Größe passend sind:

$$\text{mapping_search}(r) = \begin{cases} fli & = \lfloor \log_2(r + 2^{\lfloor \log_2(r) \rfloor - \mathcal{J}} - 1) \rfloor \\ sli & = \lfloor \frac{r + 2^{\lfloor \log_2(r) \rfloor - \mathcal{J}} - 1 - 2^{fli}}{2^{fli - \mathcal{J}}} \rfloor \end{cases}$$

Auch diese Berechnung lässt sich mittels Bitoperationen effizient in konstanter Zeit implementieren (nach [MRR⁺07]):

```

1 void mapping_search(unsigned int r,
2                     unsigned int *fli, unsigned int *sli) {
3     r = r + (1 << (f1s(r) - SLI_SIZE)) - 1; // O(1)
4     *fli = f1s(r); // O(1)
5     *sli = (r >> (*fli - SLI_SIZE)) - (2 << SLI_SIZE); // O(1)
6 }

```

Mithilfe der auf diese Weise berechneten Werte für den First und Second Level Index und der First und Second Level Bitmaps, die Listen mit verfügbaren freien Blöcken kennzeichnen, kann nun mittels weiterer Bitoperationen³ in konstanter Zeit ein freier passender Speicherblock gefunden werden (nach [MRR⁺07]):

```

1 void *find_suitable_block(unsigned int fli, unsigned int sli) {
2     unsigned int temp_bitmap, non_empty_fli, non_empty_sli;
3     temp_bitmap = SL_bitmaps[fli] & (0xffffffff << sli); // O(1)
4     if(temp_bitmap != 0) {
5         non_empty_sli = ffs(temp_bitmap); // O(1)
6         non_empty_fli = fli; // O(1)
7     }
8     else {
9         temp_bitmap = FL_bitmap & (0xffffffff << (fli + 1)); // O(1)
10        non_empty_fli = ffs(temp_bitmap); // O(1)
11        non_empty_sli = ffs(SL_bitmaps[non_empty_fli]); // O(1)
12    }

```

³U. a. kommt hier die `ffs()`-Operation, die – ähnlich der `f1s()`-Funktion – das niedrigst signifikant gesetzte Bit ermittelt.

```

13  return head_of_list(non_empty_fli, non_empty_sli);    // O(1)
14  }

```

In Zeile 3 und 5 werden in der Second Level Bitmap des gegebenen First Level Indexes nur die gesetzten Bits betrachtet, die mindestens der Größenklasse der gesuchten Indizes entsprechen. Mit der `ffs`-Operation kann dann in konstanter Zeit aus diesen die kleinste, passende Größenklasse ermittelt werden, die noch freie Blöcke enthält.

Enthält keine der Second Level Größenklassen noch freie Blöcke, so wird in den Zeilen 9 bis 11 die nächsthöhere First Level Größenklasse, deren Second Level Listen noch freie Blöcke enthalten, bestimmt sowie die zugehörigen Indizes berechnet.

In Zeile 13 kann so der Kopf der Liste als passender Speicherblock zurückgegeben werden.

Mit den so definierten Hilfsoperationen kann nun eine `malloc()`-Implementierung für die TLSF-Datenstruktur angeben, die eine Allokation eines passenden Speicherblocks mit mindestens der angeforderten Größe durchführt. Dazu werden die folgenden Schritte durchgeführt:

1. Bestimmen des First Level Indexes und Second Level Indexes mit `mapping_search()`.
2. Basierend auf den Indizes Ermitteln eines passenden freien Blocks mithilfe der Funktion `find_suitable_block()`.
3. Entfernen des Blocks aus der doppelt verketteten Liste der freien Blöcke.
4. Evtl. Splitten des Blocks, falls dieser über einer bestimmten Schwelle größer ist als die angeforderte Größe.

```

1  void *malloc(unsigned int size) {
2    unsigned int fli, sli;
3    void *free_block, *remaining_block;
4    mapping_search(size, &fli, &sli);                // O(1)
5    free_block = find_suitable_block(fli, sli);      // O(1)
6    if(free_block != NULL) {
7        remove_head(free_block);                    // O(1)
8        if(size_of_block(free_block) - size > SPLIT_SIZE_THRESHOLD) {
9            remaining_block = split(free_block, size); // O(1)
10           mapping_insert(size_of_block(remaining_block), &fli, &sli); // O(1)
11           insert_block(remaining_block, fli, sli);  // O(1)
12        }
13    }
14    return free_block;
15 }

```

Dabei sind die Operationen Entfernen aus und Einfügen eines Blocks in eine doppelt verkettete Liste sowie das Splitten eines Blocks in konstanter Zeit durchführbar (s. [MRR⁺07, S. 16]).

Insgesamt ist somit die Laufzeit einer Allokation mit dem TLSF-Algorithmus beschränkt durch $\Theta(1)$ und damit in konstanter Zeit möglich.

Die Deallokation eines zuvor belegten Blocks kann nun ebenfalls angegeben werden. Da diese ein Immediate Coalescing vornehmen soll, werden die Hilfsfunktionen `merge_prev()` und `merge_next()` (nach [MRR⁺07]) eingeführt, die einen freizugebenden Block mit den jeweils benachbarten Blöcken zusammenführen, falls diese frei sind:

```

1 void *merge_prev(void *block) {
2     if (is_prev_free(block)) { // O(1)
3         unsigned int fli, sli;
4         void *prev_block = prev_physical(block); // O(1)
5         mapping_insert(size_of_block(prev_block), &fli, &sli); // O(1)
6         remove_block(prev_block, fli, sli); // O(1)
7         merge(prev_block, block); // O(1)
8     }
9     return block;
10 }

```

```

1 void *merge_next(void *block) {
2     if (is_next_free(block)) { // O(1)
3         unsigned int fli, sli;
4         void *next_block = next_physical(block); // O(1)
5         mapping_insert(size_of_block(next_block), &fli, &sli); // O(1)
6         remove_block(next_block, fli, sli); // O(1)
7         merge(block, next_block); // O(1)
8     }
9     return block;
10 }

```

Dabei können über die Boundary Tags und die Adresse des Blocks die benachbarten Blöcke und deren Status in konstanter Zeit bestimmt werden. In den Segregated Lists der TLSF-Datenstruktur sind die freien Blöcke wie bereits angegeben ebenfalls über die Boundary Tags in doppelt verketteten Listen organisiert. Über die Hilfsfunktion `remove_block()` können so die benachbarten Blöcke aus diesen doppelt verketteten Liste entfernt werden, wobei je nach Position des Blocks in der Liste drei Fälle zu unterscheiden sind, nämlich der zu entfernende Block befindet sich:

1. Am Kopf der Liste (*fli*, *sli*): In diesem Fall wird – falls vorhanden – der nächste Block zum Kopf der Liste und der Rückzeiger dieses Blocks wird so angepasst, dass dieser den zu entfernenden Block nicht mehr referenziert. Gibt es keinen weiteren Block in der Liste, ist diese nun leer, d. h. das entsprechende Bit in der TLSF-Datenstruktur für (*fli*, *sli*) wird so gesetzt, dass dieses nun eine leere Liste angibt.

2. Mitten in der Liste (fli, sli): In diesem Fall werden die Rück- und Vorwärtszeiger der benachbarten Blöcke so angepasst, dass die Blöcke sich nun gegenseitig und nicht mehr den freizugebenden Block referenzieren.
3. Am Ende der Liste (fli, sli): Hier wird der vorhergehende Block zum Ende der Liste, indem im vorhergehenden Block der Vorwärtszeiger auf den zu entfernenden Block zurückgesetzt wird.

In allen Fällen lassen sich die benachbarten Blöcke in konstanter Zeit bestimmen, der zu entfernende Block kann aus den Listen durch die doppelte Verkettung – da nur Anpassungen an den Rück- bzw. Vorwärtszeigern der benachbarten Blöcke notwendig sind – ebenfalls in konstanter Zeit entfernt werden. Nachdem ein benachbarter Block so aus der TLSF-Datenstruktur entfernt wurde, kann das Zusammenfügen der benachbarten freien Blöcke letztlich ebenfalls in konstanter Zeit mit der Hilfsfunktion `merge()` durchgeführt werden: Diese gibt die Adresse des physikalischen kleineren Blocks zurück und setzt in dem zugehörigen Boundary Tag zu Beginn des Blocks das Feld für die Größe des Blocks auf die gemeinsame, addierte Größe beider Blöcke.

Die Implementierung von `free()` ergibt sich mit diesen Funktionen schließlich zu (nach [MRBC08]):

```

1 void free(void *block) {
2     unsigned int fli, sli;
3     block = merge_prev(block);           // O(1)
4     block = merge_next(block);          // O(1)
5     mapping_insert(size_of_block(block), fli, sli); // O(1)
6     insert_block(block, fli, sli);      // O(1)
7 }
```

Somit ist auch die Deallokation in $\Theta(1)$ möglich.

3.2. Half-Fit

Bereits 1995 wurde mit *Half Fit* [Oga95] ein weiterer echtzeitfähiger Dynamic Storage Allocator vorgeschlagen, der ebenfalls in Echtzeitsystemen zum Einsatz kommen kann, da auch dieser theoretisch und in seiner Implementierung eine beschränkte Laufzeit in $\Theta(1)$ aufweist.

Half Fit, der zunächst im Echtzeit-Bereich keine Bekanntheit erlangte und erst bei der Entwicklung von Two Level Segregate Fit „wiederentdeckt“ wurde (vgl. [MRBC08, S. 2]), benutzt dabei letztlich die gleiche Idee wie der First Level Index bei TLSF.

Auch hier werden die verfügbaren Speicherblöcke in Größenklassen unterteilt, die ebenfalls jeweils Größen in den Bereichen $[2^i, 2^{i+1}[$ abdecken. Zur Verwaltung der Segregated Lists für jede Größenklasse wird ebenfalls eine Bitmap benutzt. Mittels der Bitsuchoperationen `fls()` und `ffs()` wird für eine Speicheranfrage r auch in Half Fit in konstanter Zeit die Liste mit freien Blöcken passender Größen ermittelt. Fällt ein angeforderter Speicherblock der Größe r in den Bereich $[2^i, 2^{i+1}[$ gibt Half Fit genau wie TLSF der nächstgrößeren Liste mit freien Blöcken zurück, also i. d. R. aus dem Größenbereich $[2^{i+1}, 2^{i+2}[$.

Da die Größenbereiche der Listen bei Half Fit immer in Zweierpotenzen auseinander liegen, ergibt sich bei Half Fit mehr als bei TLSF das Problem der Fragmentierung (vgl. Kap. 3.3.2). Außerdem tritt noch ein weiteres Problem – ebenfalls sehr viel stärker als bei TLSF – in Erscheinung, das in [Oga95] als „*incomplete memory use*“ beschrieben wird. Hiermit ist ein Problem gemeint, bei dem die Suchstrategie, in der nächstgrößeren Liste zu suchen, bei einer Speicheranfrage in bestimmten Szenarien zu einem Fehler führen kann, obwohl ein freier Block der passenden Größe vorhanden ist.

Beispiel 3.2.1. Gegeben sei ein Heap der Größe $\mathcal{H} = 2^{10} = 1024$ Bytes. Werden nun Blöcke der Größen 32, 496 und 496 Bytes angefordert – die beiden letzteren seien zur vereinfachten Betrachtung nicht physikalisch benachbart⁴ – und die beiden letzten wieder freigegeben, so befinden sich diese in der Liste $i = 8 \rightarrow [256, 512[$. Wird nun wieder ein Block der Größe 496 Bytes angefordert, beginnt die Suche in der freien Liste mit $i = 9 \rightarrow [512, 1024[$, die jedoch keine freien Blöcke enthält, wodurch die Speicheranfrage nicht bedient werden kann.

In TLSF kommt dies zunächst weniger zum Tragen, da durch den Second Level Index feinere Unterteilungen der Größenklassen vorgenommen werden (s. [MRBC08, S. 16]). Weitergehend kann in TLSF die Strategie insofern erweitert werden, dass bei einer Speicheranfrage der Größe r diese immer automatisch auf die untere Grenze der nächstgrößeren Liste aufgerundet wird. So steht der Block nach seiner Freigabe in der korrekten Liste zur Verfügung, gleichzeitig wird aber eine höhere interne Fragmentierung erzeugt.

⁴In komplexeren Szenarien kann diese Situation einfach erzeugt werden bzw. häufig auftreten (vgl. [MRBC08, S. 15]).

3.3. Evaluation

In [MRBC08, Kap. 6ff] wird der *Two Level Segregate Fit* Allokator mit bestehenden Allokatoren verglichen. Hierzu werden u. a. Implementierungen von TLSF mit solchen von *First Fit*, *Best Fit*, *Binary Buddy*, *dmalloc* (vgl. Kap. 2) und *Half Fit* (vgl. Kap. 3.2) in verschiedenen Szenarien auf ihre Eigenschaften getestet.

Im Folgenden werden die zum Vergleich genutzten Testfälle, ihre Durchführung und die Ergebnisse – teils auszugsweise – beschrieben und aufgeführt. Für die vollständigen Ergebnisse sei an dieser Stelle auf [MRBC08, Kap. 6-8] verwiesen.

Um Aussagen über das Laufzeitverhalten der Algorithmen sowie die auftretende Fragmentierung zu erhalten, wurden für jeden Algorithmus Tests durchgeführt, um die Laufzeit im schlechtesten Fall (Worst Case Execution Time, WCET), die durchschnittliche Laufzeit in realen Anwendungsszenarien sowie die dabei auftretende Fragmentierung zu bestimmen. Die Tests wurden wie im Folgenden beschrieben durchgeführt:

Laufzeit im schlechtesten Fall (WCET) Um die Laufzeit im schlechtesten Fall zu bestimmen, wurden für jeden Allokator künstlich Szenarien konstruiert, die bei Allokation und Deallokation den schlechtesten Fall bzw. einen möglichst schlechten Fall darstellen. Letzteres kann lediglich z. B. bei *dmalloc* für die Allokation angegeben werden, da ein Beweis, dass dies der schlechteste Fall ist, noch nicht erbracht wurde bzw. evtl. nicht erbracht werden kann (s. ebd.).

Eine eigens entwickelte minimale Testumgebung namens *μlayer* wurde hierzu entwickelt, die alle notwendigen Basis-Funktionen zur Ausführung der Allokatoren wie z. B. mathematische Funktionen und solche zur Behandlung von Zeichenketten zur Verfügung stellt. Jeder Allokator wurde direkt nach einem Boot der Testumgebung durchgeführt.

Um für die Allokatoren vergleichbare Resultate in den verschiedenen Szenarien zu erhalten, wurden die Anzahl der ausgeführten Prozessor-Instruktionen für die Allokation und Deallokation je Allokator mittels des `ptrace()`-System-Calls ermittelt.

Durchschnittliche Laufzeit (Avg. Execution Time) Zur Bestimmung der durchschnittlichen Laufzeit sowie deren Standardabweichung wurden Testszenarien mit realen Programmen aufgesetzt und benutzt. Zur Vergleichbarkeit mit anderen Allokator-Studien wurden zuvor bereits definierte Szenarien mit folgenden üblichen Anwendungsprogrammen durchgeführt (vgl. [MRBC08, Kap. 7.2 sowie Anhang A]):

- *CFRAC*, Faktorisierung von Integern
- *Espresso*, Optimierung von Schalttermen logischer Schaltungen
- *GNU awk*, AWK Script Interpreter

- *Ghostscript*, PostScript und PDF Interpreter
- *Perl*, Perl-Interpreter.

Zur Untersuchung wurde ein minimales Testsystem (vgl. [MRBC08, Kap. 6.2]) eingerichtet, dass aus einem Linux-System im Runlevel 1 (single user, kein Netzwerk, minimal gestartete Prozesse), um Interferenzen mit anderer Software und System Interrupts zu vermeiden bzw. zu minimieren.

Fragmentierung Zur Bestimmung der Fragmentierung der Allokatoren wurden die gleichen Testszenerarien sowie die Testumgebung benutzt wie zur Bestimmung der durchschnittlichen Laufzeit.

Die Fragmentierung \mathcal{F} wird dabei in Prozent angegeben und beschreibt, wie viel Speicher \mathcal{H} vom Allokator insgesamt tatsächlich benötigt wird, um den maximalen Speicher \mathcal{M} Anwendungen zur Verfügung zu stellen. \mathcal{F} lässt sich bestimmen über

$$\mathcal{F} = \frac{\mathcal{H} - \mathcal{M}}{\mathcal{M}}$$

Mit den so definierten Testfällen und -metriken können zum einen die echtzeitfähigen Implementierungen miteinander und diese zum anderen wieder mit klassischen Allokatoren verglichen werden und es kann überprüft werden, inwiefern TLSF den eingangs definierten Designzielen (vgl. Kap. 3.1) gerecht werden kann.

3.3.1. Laufzeitverhalten

Zunächst seien je Allokator die Szenarien angegeben, die zu dem schlecht möglichsten Laufzeitverhalten führen (nach [MRBC08, Kap. 7.1]):

- First Fit und Best Fit:

Allokation Zunächst werden Blöcke der minimalen Größe angefordert, bis der gesamte Speicher belegt ist. Dann werden zwei benachbarte Blöcke freigegeben, sowie jeder zweite der zuvor angeforderten Blöcke. Die ersten beiden werden dabei zu einem größeren Block durch Coalescing zusammengeführt und zu Beginn der Liste der freien Blöcke angefügt, ebenso wie die restlichen freigegebenen Blöcke. Zuletzt wird der größte Block durch die Anwendung angefordert, der sich nun am Ende der Liste der freien Blöcke befindet. Die zur Allokation eines passenden Blocks für diese Anfrage benötigte Laufzeit liegt somit – begrenzt durch die Länge der Liste – in $\mathcal{O}\left(\frac{\mathcal{H}}{2\mathcal{M}}\right)$.

Deallokation Freizugebende Blöcke werden – falls nötig – mit den Nachbarn zu einem größeren freien Block zusammengefügt und an den Kopf der Liste der freien

Blöcke eingefügt. Die maximale Laufzeit wird erreicht, wenn ein Block mit beiden physikalischen Nachbarn zusammengefügt werden kann, liegt aber auch dann in $\mathcal{O}(1)$.

- Binary Buddy:

Allokation Die schlechteste zu erwartende Laufzeit bei Allokation tritt hier auf, wenn der gesamte Speicherbereich noch frei ist (Initialzustand) und ein Block der minimalen Blockgröße angefordert wird. Zur Allokation muss der Speicherbereich sukzessive so lange in Blöcke der Größe 2^i unterteilt werden, bis ein minimal passender Block gefunden wird. Die hierfür benötigte Laufzeit liegt in $\mathcal{O}\left(\log_2\left(\frac{H}{M}\right)\right)$.

Deallokation Genau umgekehrt entsteht hier der schlechteste Fall: Wird der soeben allokierte Block wieder freigegeben, müssen alle Unterteilungen wieder zurückgenommen werden, wodurch die Laufzeit ebenfalls in $\mathcal{O}\left(\log_2\left(\frac{H}{M}\right)\right)$ liegt.

- dlmalloc:

Allokation Eine möglichst schlechte Laufzeit kann für dlmalloc verursacht werden, indem ein Szenario verursacht wird, durch das das *Deferred Coalescing* anspringt. Hierzu werden Blöcke der minimalen Blockgröße angefordert, bis kein weiterer Speicher zur Verfügung steht. Darauf werden alle Blöcke wieder freigegeben und befinden sich zunächst in der Liste der zurückgestellten freien Blöcke, die zusammengefügt werden können. Bei der nächsten Speicheranfrage werden dann durch das *Deferred Coalescing* alle diese Blöcke zusammengefügt, die Laufzeit dafür liegt in $\mathcal{O}\left(\frac{H}{M}\right)$.

Deallokation dlmalloc gibt in allen Fällen den Block frei und fügt diesen einer Liste hinzu. Die Laufzeit liegt immer in $\mathcal{O}(1)$.

- Half Fit und TLSF:

Allokation Bei beiden Allokatoren tritt der schlecht möglichste Fall ein, wenn nur ein großer freier Block (initial) existiert und eine Anfrage für die minimale Blockgröße eintrifft, da dann ein Splitting vorgenommen werden muss. Die Laufzeit hierfür liegt in $\mathcal{O}(1)$ (vgl. Kap. 3.1.2).

Deallokation Auch hier tritt der schlechteste Fall ein, wenn zwei physikalisch benachbarte Blöcke mit dem freizugebenden Block zusammengelegt werden müssen. Die Laufzeit ist auch hier beschränkt durch $\mathcal{O}(1)$ (vgl. ebd.).

Alle Allokatoren wurden mit allen diesen Szenarien getestet. Tabelle 3 gibt einen Überblick über die in [MRBC08] gemessenen Laufzeiten der Allokatoren. Hierzu wurde in jedem Testfall

das Worst-Case-Szenario wie oben beschrieben aufgebaut und dann eine einzige Speicheranfrage mittels eines `malloc()`-Aufrufs durchgeführt, dessen Laufzeit in ausgeführten Prozessor-Instruktionen wie angegeben gemessen wurde. Die Laufzeiten der Allokatoren in dem jeweils schlechten Fall sind dort hervorgehoben.

Szenario \ Impl.	First Fit	Best Fit	Binary Buddy	dlmalloc	Half Fit	TLSF
First Fit / Best Fit	81995	98385	115	109	162	197
Binary Buddy	86	94	1403	729	162	188
dlmalloc	88	96	1113	721108	164	197
Half Fit / TLSF	88	96	1287	729	164	197

Tabelle 3: Dauer eines `malloc()`-Aufrufs in Zahl der Prozessor-Instruktionen in versch. Worst-Case-Szenarien nach [MRBC08].

Insgesamt fällt bei den gemessenen Ergebnissen auf, dass alle klassischen Allokatoren deutlich schlechtere Laufzeiten in ihrem jeweiligen Worst-Case-Szenario aufweisen als in den jeweils anderen Szenarien. Lediglich Half Fit und TLSF können auch hier eine (nahezu) konstante Laufzeit aufweisen.

Zudem wird bei `dlmalloc` deutlich, dass durch Deferred Coalescing eine unvorhersehbare Zeitkomponente in einigen Szenarien gegeben ist: Während der Allokator in vielen Fällen – so auch in realen Anwendungsfällen – gute durchschnittliche Laufzeiten liefert, kann eine einzelne Allokation im schlechtesten Fall eine sehr hohe Laufzeit aufweisen: Die höchste, die in der gesamten Evaluation gemessen wurde. Daher eignet sich `dlmalloc` nicht zum Einsatz in Echtzeit-Anwendungen und -Systemen.

Half Fit weist insgesamt bessere Laufzeiten im schlechtesten Fall auf als TLSF. Dieses Ergebnis wird jedoch durch die höhere Fragmentierung (vgl. Kap. 3.3.2) relativiert.

TLSF erweist sich somit im Sinne der Design-Ziele als eine Implementierung, die sowohl die Laufzeit als auch die Fragmentierung gering hält (vgl. ebd.).

In Tabelle 4 sind die Laufzeiten der Deallokation der einzelnen DSA-Algorithmen in Prozessor-Instruktionen nach [MRBC08] angegeben. Die Laufzeiten wurden dabei jeweils aufgrund ihrer Ähnlichkeit nur für den jeweils eigenen schlechtesten Fall angegeben. Da die Deallokation bei allen Allokatoren außer Binary Buddy aus Listen-Operationen auf verketteten Listen und evtl. Coalescing besteht, fallen die Ergebnisse entsprechend gleichförmig aus. `dlmalloc` kann dadurch, dass kein Coalescing bei Freigabe eines Blocks erfolgt, diese Operation am schnellsten durchführen.

free()-Impl.	First Fit	Best Fit	Binary Buddy	dmalloc	Half Fit	TLSF
Anzahl Prozessor-Instr.	115	115	1379	51	169	187

Tabelle 4: Dauer eines free()-Aufrufs in Zahl der Prozessor-Instruktionen in versch. Worst-Case-Szenarien nach [MRBC08].

Der Binary Buddy DSA muss im schlechtesten Fall bis zu $\log_2(\mathcal{H})$ Blöcke zusammenführen und weist entsprechend die höchste Laufzeit auf.

Tabelle 5 gibt einen Überblick über die durchschnittliche Laufzeit der DSA-Algorithmen inkl. der Standardabweichung der malloc()- und free()-Funktionen in den Ghostscript-Testfällen nach [MRBC08]. Dabei sind die Laufzeiten in Prozessor-Zyklen je Allokator und Testfall von malloc() bzw. free() angegeben, in Klammern ist darunter die jeweils ermittelte Standardabweichung der Operationen zu finden. Ghostscript soll an dieser Stelle beispielhaft angegeben werden, da es im Test eine hohe Varianz der angeforderten Blockgrößen aufwies, wodurch die Allokatoren besonders gefordert wurden (s. [MRBC08, Kap. 8.2]). Bei den CFRAC, Espresso und GNU awk Tests werden so überwiegend lediglich Blöcke kleiner Größen angefordert (s. ebd.). Die Ergebnisse zeigen zunächst, dass dmalloc in allen Tests die geringste Abwei-

malloc()/free() (Standardabw.)	First Fit	Best Fit	Binary Buddy	dmalloc	Half Fit	TLSF
Test 1	1857/170 (3031/277)	1834/148 (2933/214)	1554/241 (4827/232)	2042/131 (2970/190)	227/170 (189/202)	685/413 (964/693)
Test 2	196/196 (486/166)	389/286 (512/286)	345/301 (622/217)	801/143 (882/127)	343/322 (301/298)	344/202 (245/207)
Test 3	1918/192 (3278/372)	1849/184 (2997/323)	1130/321 (3395/377)	1903/128 (3206/211)	214/176 (182/257)	419/257 (533/454)

Tabelle 5: Durchschnittliche Dauer der malloc()-/free()-Aufrufe in Prozessor-Zyklen in versch. Ghostscript-Tests (vgl. [MRBC08, Anhang A]) nach [MRBC08].

chung in den durchschnittlichen Laufzeiten aufweist, was dem Design-Ziel – der Optimierung der durchschnittlichen Laufzeit – entspricht. Insgesamt gehören TLSF und Half-Fit mit ihren Laufzeiten am häufigsten zu den schnellen Implementierungen. Auch hier lässt sich beobachten, dass Half Fit dabei noch bessere Laufzeiten als TLSF erreicht, wobei auch hier die schlechteren Ergebnisse von Half Fit bei der Fragmentierung zu berücksichtigen bleiben.

Insgesamt wird in [MRBC08] beobachtet, dass bei den Tests mit Ghostscript unter den stärkeren Anforderungen – wie der zuvor beschriebenen höheren Varianz bei der Größe ange-

forderter Speicherblöcke – die klassischen Allokatoren deutliche Laufzeiteinbußen aufweisen: So ist die `dlmalloc`-Laufzeit im Vergleich zu den `CFRAC`, `Espresso` und `GNU awk` Tests um das 22-fache erhöht, die Laufzeit des `Binary Buddy DSA` um das 15-fache sowie die von `First Fit` um das zehnfache (s. [MRBC08, S. 29]). Gleichzeitig ergeben sich für `Half Fit` und `TLSF` nur um das Zwei- bis Dreifache erhöhte Laufzeiten, wodurch darauf geschlossen wird, dass sich die Laufzeiten der beiden Algorithmen auch unter besonderen realen Anforderungen stabil verhalten.

3.3.2. Fragmentierung

Wie eingangs beschrieben wurde bei allen DSA-Algorithmen in Tests mit den typischen Anwendungsprogrammen `CFRAC`, `Espresso`, `GNU awk`, `Ghostscript` und `Perl` auch die Fragmentierung getestet.

Tabelle 6 gibt einen Überblick über die Ergebnisse für die Fragmentierung \mathcal{F} . Da Echtzeit-Anwendungen wie zu Beginn des Kapitels beschrieben oft über längere Zeit laufen als gewöhnliche Anwendungen wurden von allen getesteten Anwendungen die Testfälle (vgl. [MRBC08, Anhang A]) mit der längsten Laufzeit angeführt. Werden die Ergebnisse bei den Laufzeit-Tests

Fragmentierung \mathcal{F}	First Fit	Best Fit	Binary Buddy	dlmalloc	Half Fit	TLSF
CFRAC Test 5	92,3%	32,1%	89,8%	34,6%	81,5%	38,0%
Espresso Test 4	67,4%	7,1%	88,7%	7,3%	17,4%	8,0%
GNU awk Test 3	61,2%	7,5%	44,5%	7,5%	7,9%	9,1%
Ghostscript Test 3	12,5%	0,3%	26,8%	0,3%	1,1%	0,4%
Perl Test 4	16,5%	8,9%	34,4%	9,3%	21,2%	10,3%

Tabelle 6: Fragmentierung \mathcal{F} in Prozent der verschiedenen DSA-Algorithmen in Anwendungsfällen nach [MRBC08] (vgl. auch [MRBC08, Anhang A]).

berücksichtigt, so kann bei den Testfällen zwar teils eine besserer Fragmentierungsfaktor bei den klassischen Allokatoren `Best Fit` und `dlmalloc` beobachtet werden, diese Algorithmen sind wegen ihrer nicht beschränkten Laufzeit aber nicht echtzeitfähig.

Für die echtzeitfähigen Allokatoren `Half Fit` und `TLSF` lässt sich dagegen der bereits zuvor erwähnte Unterschied bei der Fragmentierung deutlich erkennen: Dadurch, dass `Half Fit` nur das obere Level an `Segregated Lists` implementiert, fällt die Fragmentierung im Vergleich zu `TLSF` deutlich höher aus.

Insgesamt kann somit festgehalten werden, dass TLSF gemäß seiner Design-Ziele (vgl. Kap. 3.1) eine Balance zwischen einer konstanten Laufzeit und einer möglichst geringen Fragmentierung gelingt.

4. Fazit

Es wurden zwei echtzeitfähige Implementierungen für einen Dynamic Storage Allocator Algorithmus vorgestellt: *Half Fit* und *Two Level Segregate Fit*. Für beide Algorithmen wurde dargelegt, dass die Laufzeit der zentralen Operationen `malloc()` zur Allokierung von Speicher einer bestimmten Größe sowie `free()` zur Freigabe zuvor allokierten Speichers in $\Theta(1)$ liegen und somit konstant beschränkt sind und unabhängig von den Eingabegrößen wie Größe des angeforderten Speicherblocks und den zur Verwaltung freier Blöcke genutzten Datenstrukturen sind.

In der Evaluation der Algorithmen im Vergleich zu klassischen DSA-Algorithmen wurde gezeigt, dass auch die real gemessene Laufzeit in Prozessor-Instruktionen und -Zyklen begrenzt ist, gerade auch in speziell für jeden Allokator entworfenen Worst-Case-Szenarien. Auch bei der Fragmentierung zeigen die echtzeitfähigen Implementierungen im Allgemeinen gute Ergebnisse gegenüber den klassischen Allokatoren.

Zudem wurde gezeigt, dass der TLSF-Allokator gegenüber *Half Fit* insgesamt besser abschneidet, wenn als Faktoren in diese Bewertung sowohl die Laufzeit als auch die Fragmentierung in standardisierten realen Anwendungsszenarien zugrunde gelegt werden.

Somit existiert mit *Two Level Segregate Fit* eine Implementierung eines Dynamic Storage Allocators, dessen Laufzeit zum einen konstant beschränkt ist, wodurch es ermöglicht wird, den Aufwand dynamischer Speicherverwaltung genau einzuschätzen, so dass nachgewiesen werden kann, dass Deadlines eingehalten werden können. Gleichzeitig wird die in Echtzeit-Computersystemen – evtl. knappe – Ressource Speicher dahingehend möglichst optimal verwaltet, dass ein Großteil des verfügbaren Heaps auch tatsächlich den Anwendungen zur Verfügung steht. TLSF trifft bzgl. beider Faktoren eine gute Abwägung und erreicht gute Resultate.

A. Listings

A.1. Einfache Allokation und Deallokation über Funktionsgrenzen

```
1 #include <stdlib.h>
2 #include <string.h>
3
4 void *alloc42(size_t const size) {
5     void *mem = malloc(size);
6     memset(mem, 42, size);
7     return mem;
8 }
9
10 int main(int argc, char *argv[]) {
11     char *array42 = alloc42(10);
12     int i = 0;
13     for (i = 0 ; i < 10 ; i++) {
14         array42[i] += 23;
15     }
16     free(array42);
17     array42 = 0;
18     return 0;
19 }
```

A.2. fls()-Implementierung

```
1 int fls(int x) {
2     int r = 32;
3
4     if (!x)
5         return 0;
6     if (!(x & 0xffff0000u)) {
7         x <<= 16;
8         r -= 16;
9     }
10    if (!(x & 0xff000000u)) {
11        x <<= 8;
12        r -= 8;
13    }
14    if (!(x & 0xf0000000u)) {
15        x <<= 4;
16        r -= 4;
17    }
18    if (!(x & 0xc0000000u)) {
19        x <<= 2;
20        r -= 2;
21    }
22    if (!(x & 0x80000000u)) {
23        x <<= 1;
24        r -= 1;
25    }
26    return r;
27 }
```

Quelle: Linux Kernel 2.6.38, include/asm-generic/bitops/fls.h.

Literatur

- [DM99] DEMAINE, Erik D. ; MUNRO, J. I.: Fast Allocation and Deallocation with an Improved Buddy System. In: *In proceedings of the 19th Conference on the Foundations of Software Technology and Theoretical Computer Science (fst & tcs'99), Lecture Notes in Computer Science*, 1999, S. 13–15
- [Kae01] KAEMPF, Michel: *Vudo - An object superstitiously believed to embody magical powers*. Version: April 2001. <http://phrack.org/issues.html?issue=57&id=8&mode=txt>, Abruf: 18. Nov. 2010 2, 8, 9, 32
- [Knu98] KNUTH, D.: *The Art of Computer Programming volume 1: Fundamental Algorithms*. Addison-Wesley, 1998. – ISBN 978–0201485417 2, 8, 9, 32
- [Lea00] LEA, Doug: *A Memory Allocator*. Version: April 2000. <http://g.oswego.edu/dl/html/malloc.html>, Abruf: 18. Nov. 2010 12, 13, 32
- [MRBC08] MASMANO, Miguel ; RIPOLL, Ismael ; BALBASTRE, Patricia ; CRESPO, Alfons: A constant-time dynamic storage allocator for real-time systems. In: *Real-Time Systems* 40 (2008), 149-179. <http://dx.doi.org/10.1007/s11241-008-9052-7>. – ISSN 0922–6443. – 10.1007/s11241-008-9052-7 2, 13, 15, 21, 22, 23, 24, 25, 26, 27, 28, 32
- [MRCR04] MASMANO, M. ; RIPOLL, I. ; CRESPO, A. ; REAL, J.: TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In: *16th Euromicro Conference on Real-Time Systems*. Catania, Italy : IEEE, July 2004, S. 79–88 13, 14, 16
- [MRR⁺07] MASMANO, M. ; RIPOLL, I. ; REAL, J. ; CRESPO, A. ; WELLINGS, A. J.: Implementation of a constant-time dynamic storage allocator. In: *Implementation of a constant-time dynamic storage allocator* 38 (2007). <http://dx.doi.org/10.1002/spe.858>. – DOI 10.1002/spe.858 15, 18, 19, 20
- [Oga95] OGASAWARA, Takeshi: An Algorithm with Constant Execution Time for Dynamic Storage Allocation. In: *Real-Time Computing Systems and Applications, International Workshop on* 0 (1995), 21. <http://doi.ieeecomputersociety.org/10.1109/RTCSA.1995.528746>. ISBN 0–8186–7106–8 22
- [WJNB95] WILSON, Paul R. ; JOHNSTONE, Mark S. ; NEELY, Michael ; BOLES, David: Dynamic Storage Allocation: A Survey and Critical Review. In: *International Workshop on Memory Management* (1995). <ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps> 2, 6, 8, 9, 10, 11

Abbildungsverzeichnis

1.	Schema: Verwaltung eines Speicherpools mit bereits erfolgten Allokationen.	4
2.	Schema: Interne (1) und externe Fragmentierung (2).	6
3.	Schema: Splitting (1) und Coalescing (2).	7
4.	Schematische Darstellung der dmalloc-Datenstruktur zur Verwaltung freier Blöcke mit <i>Exact Bins</i> und <i>Sorted Bins</i> für Größenklassen nach [Lea00].	12
5.	Schematische Darstellung der TLSF-Datenstruktur zur Verwaltung freier Blöcke.	17

Tabellenverzeichnis

1.	Mögliches Format für einen Boundary Tag eines freien Blocks (nach [Knu98] und [Kae01]).	8
2.	Mögliches Format für einen Boundary Tag eines belegten Blocks (nach [Knu98] und [Kae01]).	9
3.	Dauer eines <code>malloc()</code> -Aufrufs in Zahl der Prozessor-Instruktionen in versch. Worst-Case-Szenarien nach [MRBC08].	26
4.	Dauer eines <code>free()</code> -Aufrufs in Zahl der Prozessor-Instruktionen in versch. Worst-Case-Szenarien nach [MRBC08].	27
5.	Durchschnittliche Dauer der <code>malloc()</code> -/ <code>free()</code> -Aufrufe in Prozessor-Zyklen in versch. Ghostscript-Tests (vgl. [MRBC08, Anhang A]) nach [MRBC08].	27
6.	Fragmentierung \mathcal{F} in Prozent der verschiedenen DSA-Algorithmen in Anwendungsfällen nach [MRBC08] (vgl. auch [MRBC08, Anhang A]).	28