RHmalloc:

A Very Large, Highly Concurrent Dynamic Memory Manager

Thesis submitted for the degree of

Doctor of Philosophy

Raymond John Huetter

Faculty of Information Technology

University of Technology, Sydney

February 2005

for Ziggi and Jazaan

believe in yourselves

© 2005 Raymond John Huetter

Certificate of Authorship/Originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of the requirements for a degree.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Production Note:
Signature removed prior to publication.

Raymond John Huetter
Faculty of Information Technology
University of Technology, Sydney
February 2005

Abstract

Dynamic memory management (DMM) is a fundamental aspect of computing, directly affecting the capability, performance and reliability of virtually every system in existence today. Yet oddly, the fifty year research into DMM has not taken memory capacity into account, having fallen significantly behind hardware trends. Comparatively little research work on scalable DMM has been conducted – of the order of ten papers exist on this topic – all of which focus on CPU scalability only; the largest heap reported in the literature to date is 600MB. By contrast, symmetric multi-processor (SMP) machines with terabytes of memory are now commercially available.

The contribution of our research is the formal exploration, design, construction and proof of a general purpose, high performance dynamic memory manager which scales indefinitely with respect to both CPU and memory – one that can predictably manage a heap of arbitrary size, on any SMP machine with an arbitrary number of CPU's, without *a priori* knowledge.

We begin by recognizing the scattered, inconsistency of the literature surrounding this topic. Firstly, to ensure clarity, we present a simplified introduction, followed by a catalog of the fundamental techniques. We discuss the melting pot of engineering tradeoffs, so as to establish a sound basis to tackle the issue at hand – large scale DMM. We review both the history and state of the art, from which significant insight into this topic is to be found. We then explore the problem space and suggest a workable solution.

Our proposal, known as RHmalloc, is based on the novel perspective that, a highly scalable heap can be viewed as, an unbounded set of finite-sized sub-heaps, where each sub-heap maybe concurrently shared by any number of threads; such that a suitable sub-heap can be found in O(1) time, and an allocation from a suitable sub-heap is also O(1).

Testing the design properties of RHmalloc, we show by extrapolation that, RHmalloc will scale to at least 1,024 CPU's and 1PB; and we theoretically prove that DMM scales indefinitely with respect to both CPU and memory.

Most importantly, the approach for the scalability proof alludes to a general analysis and design technique for systems of this nature.

Outline of this Dissertation

It is generally considered that the literature on dynamic memory management is inconsistent and scattered [WJNB1995]. According to Wilson et al. we know little more about DMM now than we did twenty years ago, and given its history, many misconceptions abound.

This thesis is therefore presented in the form of a text book: we start from first principles, by defining the basic concepts and terminology, and then build on that base, as we progressively work through the issues associated with proving the scalability of DMM with respect to both CPU and memory.

For readers conversant with the general topic of dynamic memory management, you may wish to skip Chapters 1, 2 and 3, which introduce the subject and cover the general engineering aspects. For those not interested in the history of DMM, you may wish to skip Chapter 4.

Chapter 1 Basic Concepts presents an overview of dynamic memory management sub-systems. We discuss the traditional concepts of allocation, freeing, splitting, coalescing, fragmentation, exhaustion, compaction and expansion.

Chapter 2 Fundamental Techniques supplies a catalog of dynamic memory management techniques. We discuss how such sub-systems can view and track memory using free lists, vectors, trees and bitmaps. Various allocation policies, such as first fit and best fit are presented. Header and footer boundary tags, as well as alignment issues, are also discussed.

Chapter 3 Engineering Issues discusses the time-space tradeoffs of DMM design. This includes issues such as patterned behavior of programs, performance, fragmentation, spatial and temporal locality, scalability, cache effects, robustness and reliability.

Chapter 4 History of the Art reviews the 50+ years of literature on this topic. We observe how the evolution of dynamic memory management is entwined with the evolution of hardware, operating systems, data structures and programming languages. We indicate where and when issues such as fragmentation and patterned behavior appeared and how these issues were subsequently refined.

Chapter 5 State of the Art examines the current state of DMM by investigating and comparing four dynamic memory managers which are currently in widespread use. It is here we observe that no specific work has been done on very large, highly concurrent dynamic memory management.

Chapter 6 Technical Requirements states our requirements for a very large, highly concurrent dynamic memory manager. We define the terms *very large* and *highly concurrent*, going on to argue that while a dynamic memory manager achieves its objectives it should exhibit efficient and predictable, linearly scalable, O(1) time-space behavior.

Chapter 7 Design Discussion explores the solution space via fourteen different thought experiments. We restate our objectives, divide our concerns and derive three principles which guide us to a workable solution.

Chapter 8 Formal Specification presents a formal specification for RHmalloc – a very large, highly concurrent dynamic memory manager. This includes graphical representations of the data structures, as well as pseudo-code for the main algorithms.

Chapter 9 Proving Scalability describes our implementation and testing of RHmalloc, tables our empirical and theoretical results proving the scalability of DMM to indefinitely large, highly concurrent environments.

Chapter 10 Conclusion and Summary states our final conclusions on dynamic memory scalability and discusses possible future work. We also provide a chapter-based summary of this thesis.

Acknowledgements

First and foremost, to Tom Hintz, my supervisor, thank you. Your wealth of knowledge, gentle wisdom, dexterous subtlety and good humor made the entire journey of this doctorate both enlightening and enjoyable. I have much to be grateful for.

I would also like to thank Bruce Howarth for taking the time to review this thesis, and provide valuable feedback; Mary O'Kane for her friendship and moral support; Tony Benson and Gary David for their belief in prodding me into this in the first place; as well as Michael Fry and Barry Jay for their sound advice when I first started.

I am grateful to Linda McEntee for her help in finding some of the rarer papers, and to IBM Australia for use of their resources in the final production of this thesis.

I remain forever appreciative of my undergraduate lecturers: Peter Cheeseman, John Debenham, Jenny Edwards, John Hughes, John Cady, John Colville and Ury Szewcow. The older I get, the more I realize, just how good an education I received.

I have had the privilege of working with a number of outstanding engineers, from whom I have learned so much: Rachna Hariharan, Michael Cahill, Adrian Tullock, Colin Pickup, Ian Steventon, Michael Hollins, Jim Wee, Frank Carnovale, Robbie Gates, Kim Haines, Gary Aitchison, Carlos Enciso, Jason Chan and Paul Smith. I would especially like to mention Alka Yamarti for the many years we have spent coding together.

I am indebted to Kay Hogan and Terry Slack for continuing to help me in countless ways, and I welcome the recent support from John and Lisa Veizades. I would be seriously remiss in failing to acknowledge Rachael Patterson – the person who finally taught me how to write, by sharing with me, her joy in writing. And a sincere thank you to the Krzyszton's and Zinkiewicz's for all the early encouragement.

I am deeply grateful for everything my parents Zygmunt and Maria Huetter have ever done for me; for their love, encouragement and example. All of my family has supported me with love and patience: Sarah, Liz, John, Graham and Brenda. I am particularly thankful to Irene Huetter and Derek Lucas who allowed me to use 'The Cottage' where most of this thesis was written.

And Jenni, the love of my life, above all, thank you for our two beautiful daughters.

Table of Contents

List of Figures

List of Tables

Chapter 1

1 Basic Concepts

Dynamic memory management (DMM) is the technique which enables a program to create/extend it's workspace during execution, catering for the situation where memory requirements are not known *a priori*, or when objects have indefinite lifetimes [Knu1973]. Dynamic memory management (a.k.a. *dynamic storage allocation* or *heap memory management*) is central to the capability, performance and reliability of virtually all computer systems in existence today.

We begin this thesis by introducing the basic concepts of dynamic memory management, by way of example.

1.1 Introduction

Let us imagine we are part of an engineering team, building an educational hand-held device called *Toby*. Toby is a spelling assistant that enables children to establish and maintain their own vocabulary. At the core of Toby is a program which manages a dictionary of words.

Assume the Toby device has, for the purposes of illustration, only 100 bytes available to hold its dictionary. This allows us to visualize Toby's memory pool as a simple 10 × 10 grid:

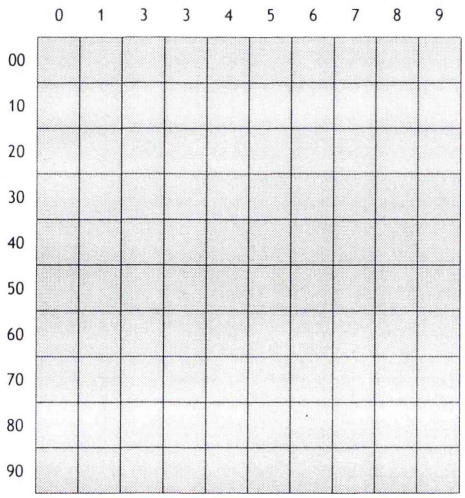


Figure 1.1: Visualizing Toby's memory pool as a 10 × 10 grid.

In order to maintain its dictionary, the Toby program requires a simple reusable data structure (such as a binary tree) to hold each word item; the constituents of which would have three elements: namely a left and right pointer, plus space for the word in question. In pseudo-code this item structure would be:

```
structure item
  left   item,
  right  item,
  word   string
end structure
```

Now, let us assume the device is initially empty, and our user enters the following set of words, in the order presented:

{ "DOG", "MOUSE", "DANCE", "GAME", "MONSTER", "GIGGLE", "LAUGHTER", "PLAYTIME", "ICECREAM", "CAT" }.

The core structure is a simple binary tree, so Toby's memory would be as follows:

	0	1	3	3	4	5	6	7	8	9
00	14	06	D	O	G	\	23	59	M	O
10	U	S	E	\	81		D	A	N	C
20	E	\		29	G	A	M	E	\	39
30		M	O	N	S	T	E	R	\	
40	48	G	I	G	G	L	E	\	70	
50	L	A	U	G	H	T	E	R	\	
60		P	L	A	Y	T	I	M	E	\
70			I	C	E	C	R	E	A	M
80	\			C	A	T	\			
90										

Figure 1.2: Toby's memory pool with initial set of words – with DOG highlighted.

To clarify, observe there is an item at location 00 that contains the word "DOG" (delimited by a '\'). The item has a left pointer of 14 (which points to the item "DANCE") and a right pointer of 6 (which points to the item "MOUSE"). The item "MONSTER" starts at location 29. This item has a left pointer of 39 (which points to item "GIGGLE") and a null right pointer (indicated by an empty square). Bytes from 87 through to 99 are unused.

1.1.1 An Architecture for Toby

Toby is, in some senses, quite a simple device. A well-engineered solution however, would still keep a clean separation of concerns, by layering the software into different subsystems – such as subsystems that manage the user interface, the dictionary routines, dynamic memory management, the underlying operating system and the physical device itself.

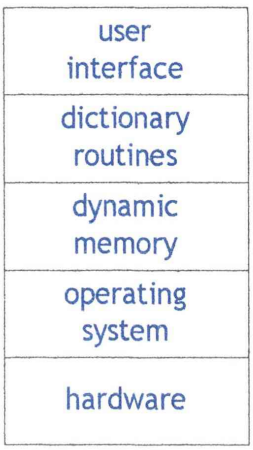


Figure 1.3: A high-level architecture for Toby.

Let us assume Toby is a well-engineered device along these lines.

1.1.2 Dynamic Memory Management

Traditionally, the subsystem that manages a memory pool in the manner just described is called a *dynamic memory manager*, a *heap memory manager* or sometimes just an *allocator*. A dynamic memory manager's job is to track free space in one or more memory pools referred to as a *heap*. A dynamic memory manager has two fundamental operations: allocation and deallocation, as described below.

DMM is the technique used when, the size and quantity of items held by a program, cannot be determined *a priori* at compile time. In the case of Toby, the number of words and the length of each word are unknown – it simply depends on the set of words the child chooses to hold in the device. One option would be to put an upper limit on the length of words that can be stored at, say ten characters, thereby fixing the device to store at most ten words. In the large however, this leads to both a very inefficient use of memory, since not all words are exactly ten letters long, and such a constraint is

artificial, since ten characters would eventually frustrate most users – particularly if a future version of Toby had 1TB of memory.

1.1.3 Allocation

Assume the user adds the word "SCHOOL". In order to do this, Toby's dynamic memory manager has to *allocate* space in memory for the new item[1]. Specifically, the allocator has to find a *block* of nine bytes that is available for allocation. If, for the moment, we assume Toby's allocator simply finds the first available block large enough for the new item, then after allocation, Toby's memory pool would be in the following state:

	0	1	3	3	4	5	6	7	8	9
00	14	06	D	O	G	\	23	59	M	O
10	U	S	E	\	81		D	A	N	C
20	E	\		29	G	A	M	E	\	39
30		M	O	N	S	T	E	R	\	
40	48	G	I	G	G	L	E	\	70	
50	L	A	U	G	H	T	E	R	\	
60	87	P	L	A	Y	T	I	M	E	\
70			I	C	E	C	R	E	A	M
80	\			C	A	T	\			S
90	C	H	O	O	L	\				

Figure 1.4: Toby's memory pool with SCHOOL added.

Observe that the block of unused bytes became shorter: specifically bytes 87 through 95 were allocated, leaving bytes 96 through 99 free, available for future use.

[1] The allocation operation often goes by the terms: *allocate*, *malloc* or *alloc*.

1.1.4 Deallocation

Assume the user now deletes the word "MONSTER". In order to do this, Toby's dynamic memory manager has to *deallocate* the space used by that item[2]. Specifically, the allocator has to make the ten byte block, occupied by the item containing the word "MONSTER", available for reuse.

Freeing "MONSTER" would leave Toby's memory pool in the following state:

	0	1	3	3	4	5	6	7	8	9
00	14	06	D	O	G	\	23	59	M	O
10	U	S	E	\	81		D	A	N	C
20	E	\		39	G	A	M	E	\	
30										
40	48	G	I	G	G	L	E	\	70	
50	L	A	U	G	H	T	E	R	\	
60	87	P	L	A	Y	T	I	M	E	\
70			I	C	E	C	R	E	A	M
80	\			C	A	T	\			S
90	C	H	O	O	L	\				

Figure 1.5: Toby's memory pool after MONSTER is deleted.

Observe there are now two free blocks of memory in Toby's pool: one block of ten bytes starting at location 29, and a second block of four bytes starting at location 96.

[2] The deallocation operation often goes by the terms: *deallocate*, *free* or *release*.

1.1.5 Coalescing

Assume the user deletes the word "GIGGLE". This frees bytes 39 through 47 (Figure 1.6). Notice this is adjacent to the space freed up when "MONSTER" was deleted. Depending on the design, Toby's allocator may keep the two free blocks separate, or it may *coalesce* those two free blocks into a single, larger free block.

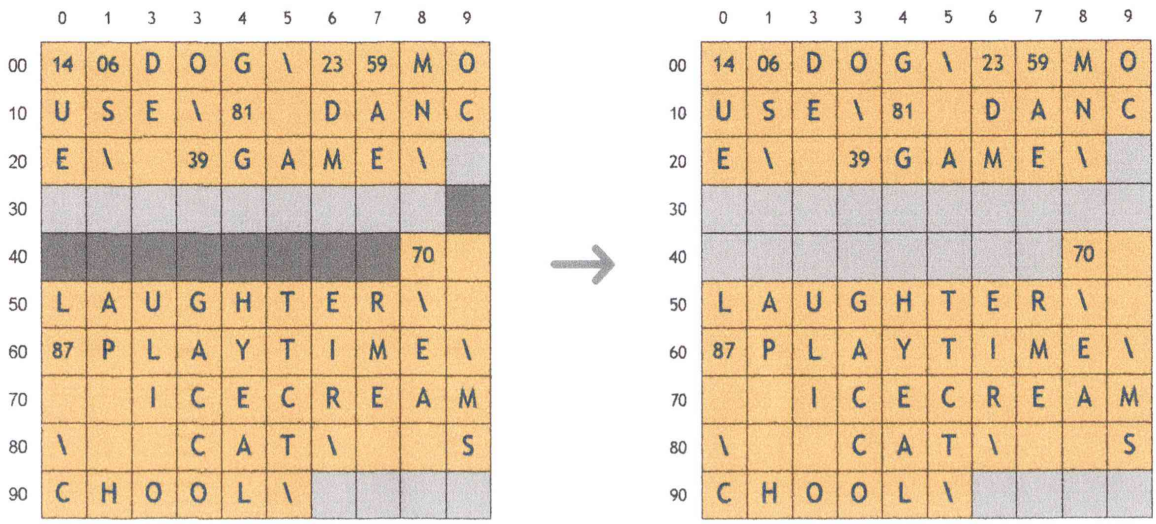


Figure 1.6: Coalescing two free blocks into a single block.

Coalescing free blocks is a form of "housekeeping". By coalescing the adjacent free blocks, at the time they are freed, means the allocator has a single resource readily capable of servicing a large allocation request in the future.

1.1.6 Splitting

For the purpose of this discussion, assume Toby's allocator had coalesced the two free blocks into one, and our user now adds the word "CHOCOLATE". Of the two free blocks available only the larger, previously coalesced one, is big enough to hold the new item. Depending on the design, Toby's allocator may *split* the large block into two separate blocks, with one being just large enough to satisfy the new item.

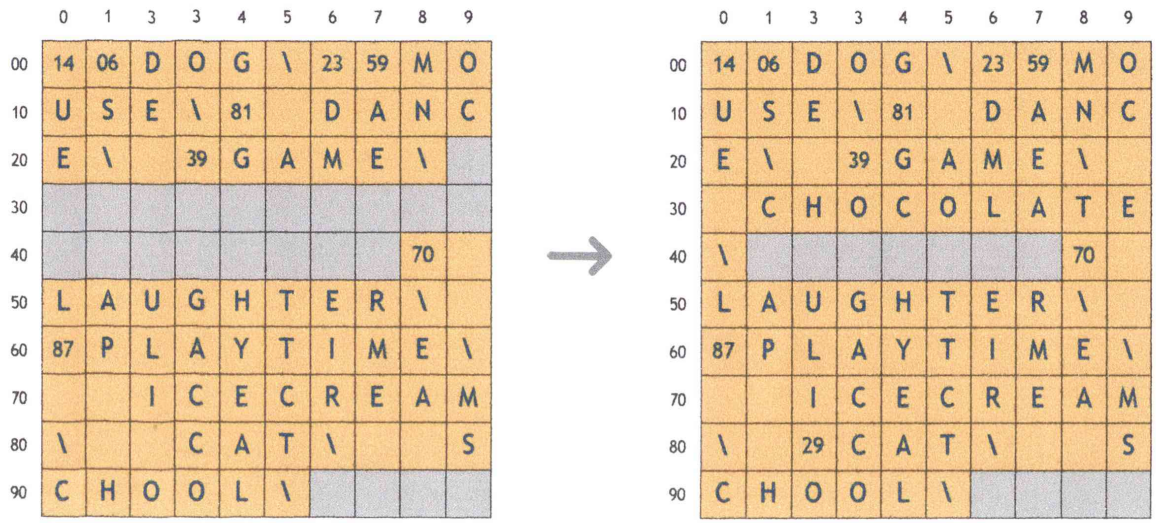


Figure 1.7: Splitting a free block.

Splitting an oversized block in this manner means the remainder is available for future allocation, and is not wasted.

1.1.7 Exhaustion

If the user wanted to add the word "BIBLIOGRAPHY" it would require Toby to have 15 bytes of free memory. At this point there is not enough space to hold that new word. Memory is said to have been *exhausted*.

1.1.8 Fragmentation

Assume the user now tries to add the word "AARDVARK". This requires 11 bytes of free space. Observe at this point that Toby's memory pool has 11 bytes of free space – but not as one contiguous block. Toby would therefore respond with an error indicating there is insufficient free space.

Not being easily thwarted, our user decides to delete some words – let's say "DOG", "DANCE" and "CAT" in order to make space – and then retries, only to be given the same error message. Toby's memory pool has become *fragmented.*

	0	1	3	3	4	5	6	7	8	9
00							23	59	M	O
10	U	S	E	\						
20				48	G	A	M	E	\	
30	6	C	H	O	C	O	L	A	T	E
40	\								70	
50	L	A	U	G	H	T	E	R	\	
60	87	P	L	A	Y	T	I	M	E	\
70			I	C	E	C	R	E	A	M
80	\						\			S
90	C	H	O	O	L	\				

Figure 1.8: Fragmented memory pool.

In aggregate, there is more than sufficient space in the pool to hold the new item, but there is no one single block large enough to do so.

1.1.9 Compaction

One possible solution, to the fragmentation problem, is to shuffle the items down in memory, so as to *compact* them. In this example, if we were to shuffle the items down, we would end up with the following arrangement:

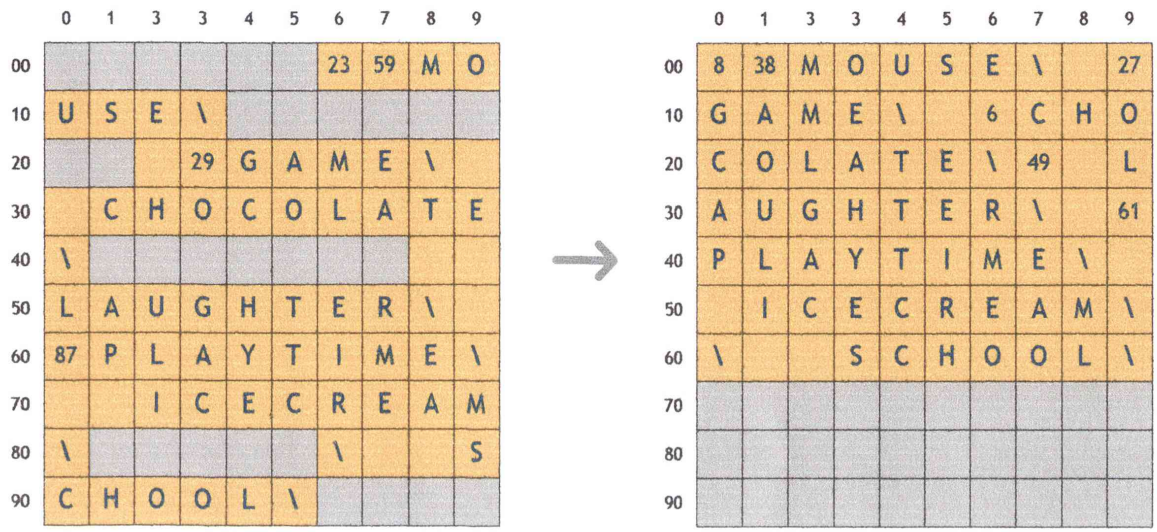


Figure 1.9: Compacting a memory pool.

Obviously as further new words are added, and old words are deleted, fragmentation can reappear.

1.1.10 Expansion

An alternate solution to fragmentation and exhaustion is to expand memory. In case of the Toby device, this might be possible by allowing the user to purchase a memory expansion kit, which added (say) 50% more memory. This would lead to the following scenario, where the Toby device now has 150 bytes in its memory pool:

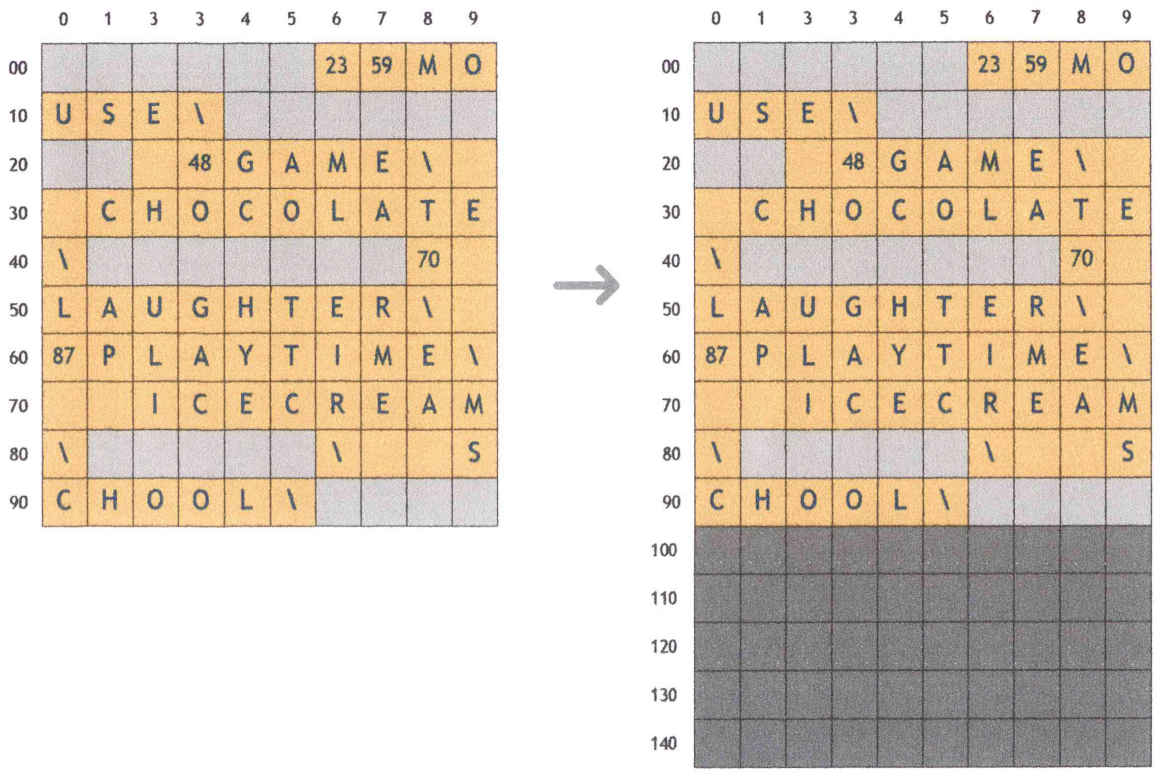


Figure 1.10: Expanding a memory pool.

Expansion only ameliorates the current situation. Should the user continue to add new words, ultimately we end up with the device being fully expanded and fully utilized – albeit with more words.

1.2 Formalizing Dynamic Memory Management

Having introduced the basic concepts, let us formalize dynamic memory management. Conceptually, an allocator is a sub-system, which typically sits directly above the virtual memory services of an operating system, in order to provide fine grained dynamic memory services to calling programs. Virtual memory systems manage memory in discrete units, typically called *pages*, while programs typically require memory in units which are not related to the page size of the given operating system[3].

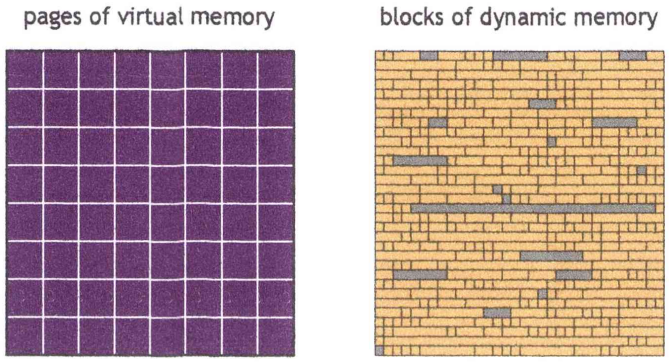


Figure 1.11: The page view of virtual memory versus the block view of dynamic memory.

Executing programs require such dynamic memory facilities when memory requirements are not known *a priori*, or when objects have indefinite lifetimes. As shown in the above diagram, an allocator supports an irregular fine-grained piecemeal view of virtual memory, independent of virtual memory pages.

The usual goal of an allocator is to maximize request throughput performance, while minimizing wasted space. As described in [WJNB1995], an allocator is a sophisticated algorithm, which must satisfy a number of well understood constraints.

---

[3] Linux, for example, has a page size of 4KB, 8KB, 16KB, 32KB or 64KB – depending on the particular machine architecture [Lov2003].

For the purposes of this thesis, a conventional allocator in a virtual address based environment is one which:

- Has an interface which supports requests to allocate and free blocks of dynamic memory;
- Must keep track of which blocks of dynamic memory are in use and which blocks are available for allocation;
- Has no *a priori* information – an allocator cannot determine or control the number, size, distribution of, sequence of, or frequency of allocation requests;
- Can only use free blocks to satisfy allocation requests;
- Cannot compact memory – once a block has been allocated that decision is irrevocable; and
- Can only deal with free blocks – once allocated the contents of a block of dynamic memory cannot be modified by the allocator.

To date, it has generally been considered that the fundamental problem that an allocator must deal with is fragmentation – there is no algorithm which guarantees efficient memory use in all cases [Rob1971]. Johnstone's [Joh1997] empirical experiments have shown though, that reasonably efficient memory use tends to occur in practice. Some consider that if this had not been the case, most systems simply would not work in practice and consequently, the edifice of computing, as we know it, would have already collapsed [WJNB1995].

In the next chapter, we begin the exploration of the techniques typically used for dynamic memory management. We discuss how memory allocators can view and track memory using free lists, vectors, trees and bitmaps. Various allocation policies, such as first fit and best fit, are also presented. Header and footer boundary tags, as well as alignment issues, are also discussed.

Chapter 2

2 Fundamental Techniques

Having introduced the basic concepts, we now look at DMM a little closer, by presenting a catalog of the fundamental techniques used by various allocators to date.

Cataloging these techniques is a valuable exercise, as it provides a structured basis for discussion, and a wealth of ideas for analysis and design. It should be noted, that depending on how the various techniques presented here are mixed and matched, there can be significant differences in performance and fragmentation – particularly as we address scalability. The next chapter will discuss the interplay and effects these techniques have when they are combined. For the moment, we will deal with the techniques in isolation.

2.1 Applicability

The most fundamental aspect of an allocator is whether the allocator has been designed with general or specific purpose use in mind.

2.1.1 General Purpose

Some allocators are designed for general purpose use – the objective is to provide satisfactory all round behavior across a broad spectrum of usage. For example the allocator known as DLmalloc [Lea2000] is designed to provide balanced behavior, good performance and low fragmentation across most programs[4].

2.1.2 Specific Purpose

Some allocators are designed for specific purposes – the objective is to provide deliberate behavior when used by a particular class of program. Such an allocator is typically designed to exploit specific behavioral patterns in the calling program and therefore makes certain assumptions, about how it is to be used. For example the

---

[4] Generally, DLmalloc has good all round behavior. Consequently, it has often been considered a benchmark to date, even though it is single threaded.

allocator known as VHmalloc [VH1999] is designed and tuned for symmetric multi-processor (SMP) environments which require very high allocation rates for one single block size.

2.2 Memory View

Memory is fundamentally a linear sequence of words identified by unique addresses. There are at least two different ways, however, that memory has been conceptualized by various dynamic memory managers over the years: free form or patterned.

2.2.1 Free Form

In the free form view, memory is seen somewhat fluidly. A block of any size can, in principal, be allocated at any location; blocks can be readily split into two blocks of any size; and two adjacent free blocks of any sized can be coalesced into a single larger block. There are few restrictions in this sense – other than compliance issues such as word alignment.

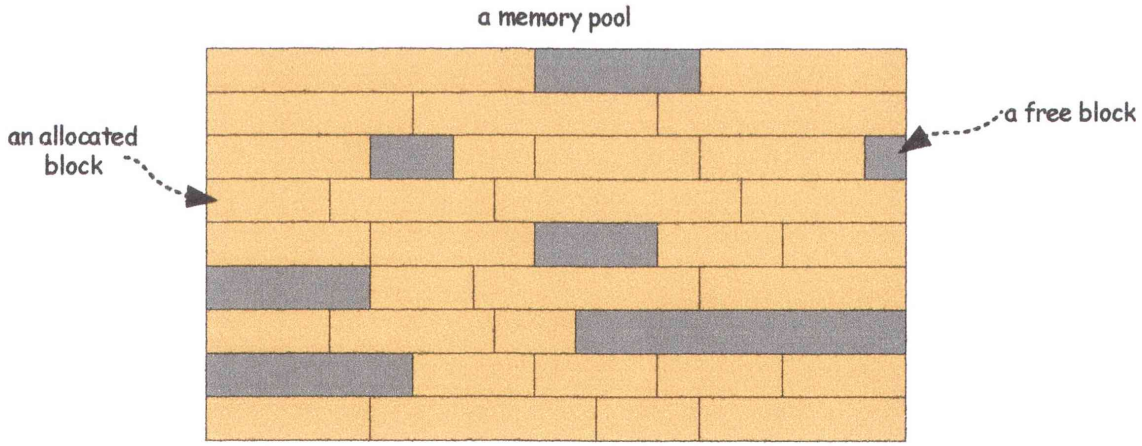


Figure 2.1: An example of a free form view of memory.

By way of example DLmalloc [Lea2000] and LKmalloc [LK1998] view memory as free form.

2.2.2 Patterned

In the patterned view, memory is seen somewhat crystalline. Memory is logically overlaid with some regular repeating mathematical pattern – which is typically some function of block size. Blocks of particular sizes can only be found at particular

locations in memory, based on the overlaying pattern. Consequently, there are restrictions on what size a block can be, given a particular location, as well as how and when blocks can be split and coalesced.

The following diagram depicts a hierarchically patterned view of memory. The pool can only be broken into blocks which are multiples of a power of two – so blocks are restricted to be of a certain size, depending on their location. For example, in a 32KB pool, a 16KB block can only be at two locations – either at the beginning or half way along. An 8KB block can only be in one of four locations; 4KB blocks can only be at one of eight locations. A block of 6KB cannot exist, while an 8KB block cannot be located $5/8^{th}$ the way into the pool.

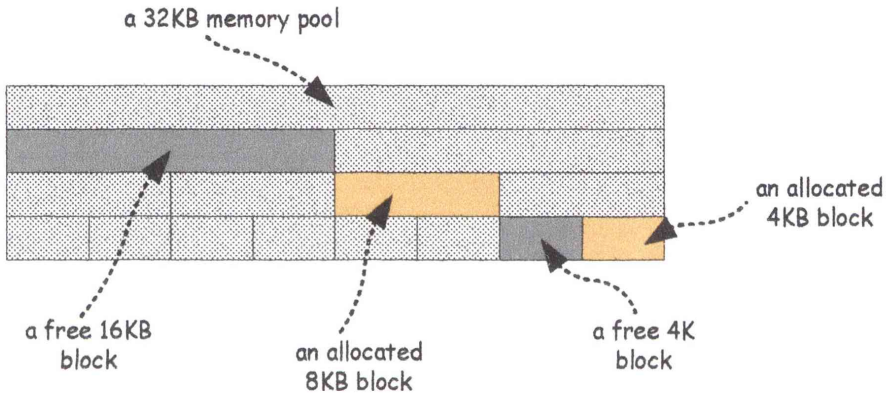


Figure 2.2: An example of a hierarchically patterned view of memory.

Historically the most popular form of patterned allocators were *buddy systems* [Kno1965, PS1970]. This idea was explored due to the ability to easily calculate possible locations based on size. The most common was the *binary buddy system* where all blocks sizes were a power of two. Given the address of a block finding its "buddy" is a simple bit manipulation of the address. This makes coalescing relatively simple.

Many variations of buddy systems have been explored over the years, based on other mathematical patterns including *fibonacci buddies* [Knu1973, Hir1973], *weighted buddies* [SP1974] and *double buddies* [WJNB1995]. Buddy systems appear to have fallen out of fashion though, as the constraint on block sizes and where blocks can be located, has been shown to lead to higher degrees of fragmentation, compared to other techniques.

2.3 Memory Partitioning

Another design decision is whether the available memory is treated as one large single pool, or whether the memory is partitioned into multiple pools.

2.3.1 Single Pool

In the non-partitioned model, the allocator manages the available memory as one large single pool. This is typically done for the sake of simplicity – multiple partitions have to be identified and tracked. DLmalloc [Lea2000] is an example of a non-partitioned allocator.

2.3.2 Multiple Pools

In the partitioned model, the heap is managed by dividing the memory available, into a number of smaller pools. Multiple pools provides a basis by which an allocator can make location choices, organize blocks in a particular way (such as according to size), or control threads by directing them into, or out of, specific pools. The number and size of pools may be either fixed or varying.

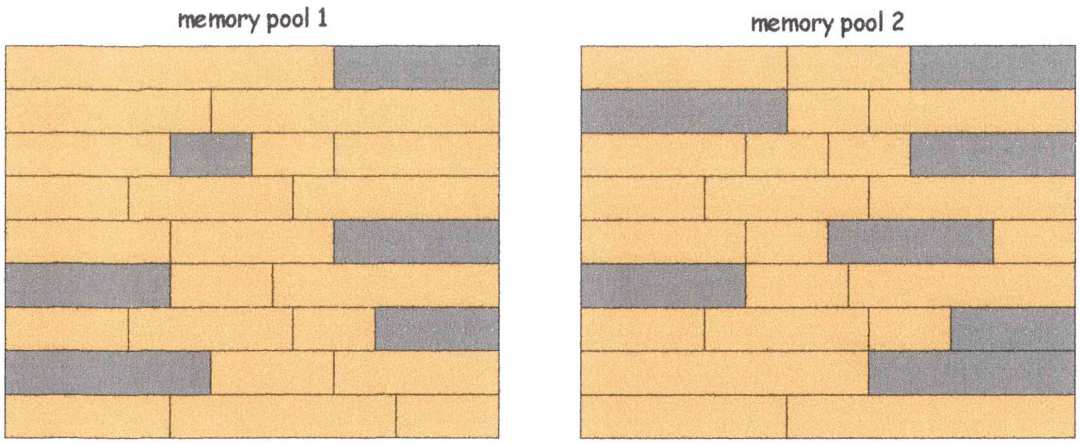


Figure 2.3: An example of a partitioned heap with two pools.

Sometimes a pool or partition is called a *superblock*. Conceptually, a superblock is a region of memory containing a number of blocks. Hoard [BMBW2000] partitions its heap into multiple superblocks. In other cases a pool is called an *arena* – particularly when a more scalable allocator is built by duplicating another, less sophisticated allocator in its entirety, a number of times [Glo2002].

2.4 Pool Constitution

Given an allocator uses multiple pools, there is a design choice as to whether the pools are basically the same (just many of them), or whether the pools are different in some way – certain blocks are sourced from certain pools.

2.4.1 Mixed Pools

In the mixed pool model, a pool is seen as containing blocks of any size – a pool is a collection of blocks of all different sizes. In principle, a mixed pool can be used to service a request of any size.

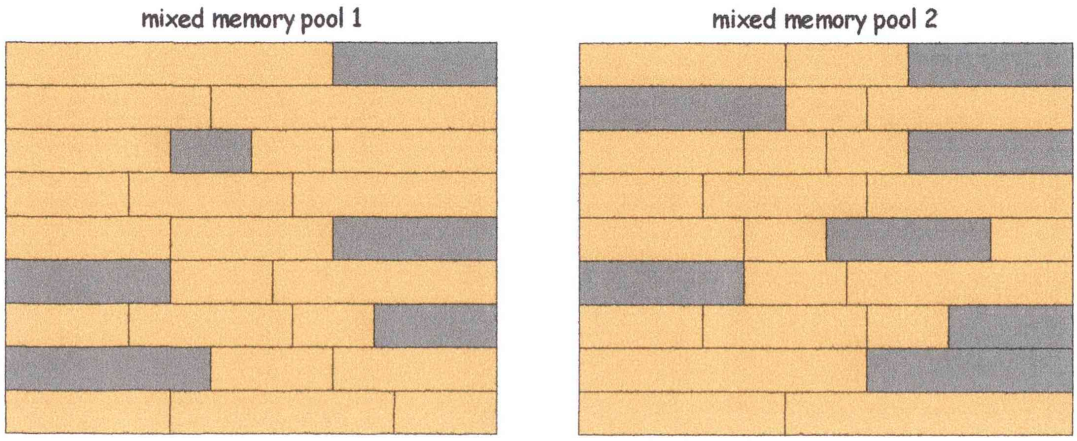


Figure 2.4: Example of a heap with two mixed pools.

PTmalloc [Glo2002] is an example of a multiple mixed pools.

2.4.2 Segregated Pools

In the segregated pool model, a pool is seen as a set of blocks of only one size, or maybe blocks of a particular range of sizes. Blocks of other sizes are to be found in other pools. A segregated pool, therefore, can only be used to service requests for particular sizes.

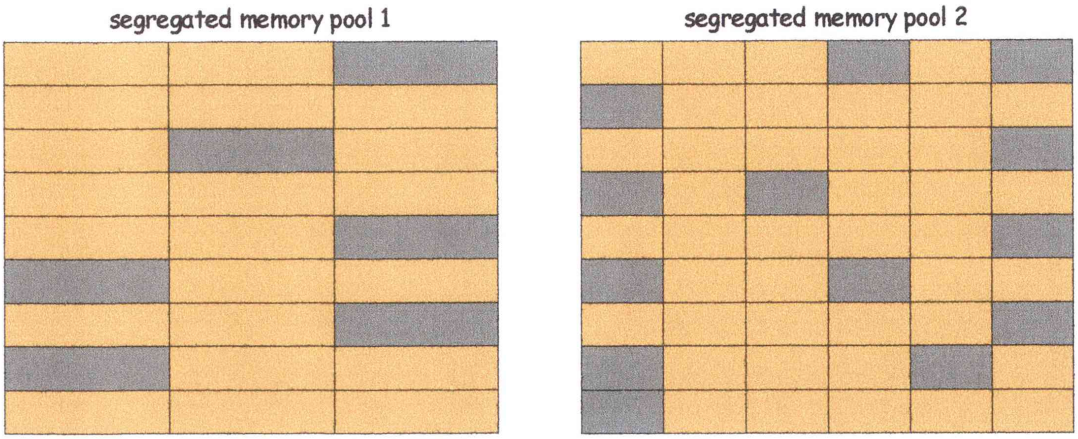


Figure 2.5: An example of a heap with two segregated pools.

In Hoard [BMBW2000], for example, each superblock is a pool holding blocks of one particular size.

2.5 Space Tracking

One of the basic jobs of an allocator is to keep an accurate record of free space, in such a manner that allocation requests can be serviced efficiently. A variety of techniques such as a free list, vector of free lists, tree of free lists and memory bitmaps have been used to date.

2.5.1 Free List

A single free list is the most basic technique for tracking free space. Many early allocators used this technique [Knu1973]. A free list is a set of blocks logically chained together – the space provided by each free block is used to hold a pointer to the next block in the list.

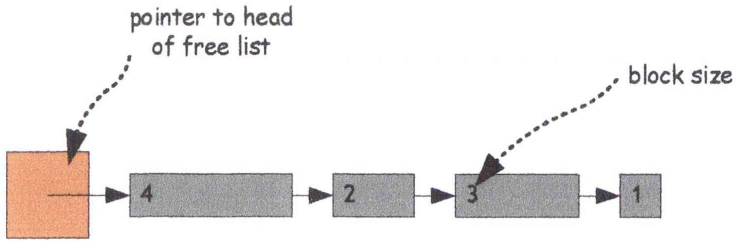


Figure 2.6: Example of a free list with four blocks of sizes 4, 2, 3 and 1 respectively.

The degree of linkage can be either single or double. In the single linked list case, each block only holds a pointer to the next block. In the double linked list case, each block holds two pointers: a pointer to the next block and a pointer to the previous block.

Additionally, the list can be either linear or circular. In the linear case, the pointers at the ends of the list are null, while in the circular case, the pointers at the ends of the list point to the other ends – logically forming a circle.

2.5.2 Vector of Free Lists

The vector of free lists technique uses an array to keep a set of free lists – each element of the array points to the head of a particular list. This technique is also known as quick fit [WW1988] and variants have been used in many allocators such as DLmalloc [Lea2000] and LKmalloc [LK1998]. Usually, there is a correlation between the element and the size of the blocks held in the list, pointed to by that element. For example, element one may point to a list which holds a set of blocks that are all one unit in size, element two may point to a list which holds a set of blocks that are all two units in size, and so on.

A list can be uniform or banded. In the uniform case, the blocks in the list are all of the same size – such as the arrangement just described. In the banded case, a list holds blocks which fall within some particular size range. For example, element one might point to a list of blocks which are between 100 and 199 units in size, element two might point to a list of blocks which are between 200 and 299 in size, and so.

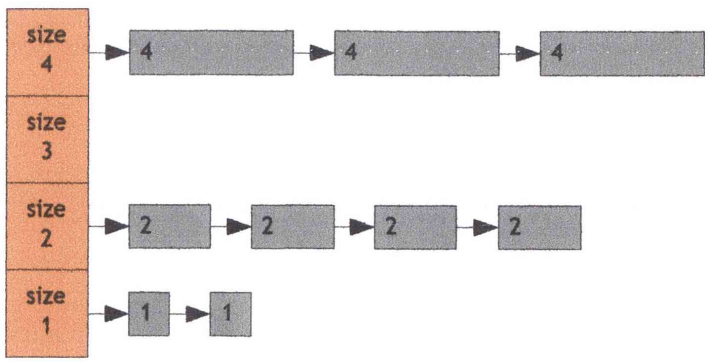


Figure 2.7: An example of a four element vector with three free lists.

DLmalloc [Lea2000] has uniform and banded lists (referred to as exact and sorted bins in Lea's nomenclature).

2.5.3 Tree of Free Lists

Trees are another technique for keeping track of free lists. This technique is often used when there is a large variation in block sizes, particularly when blocks for most of the possible sizes, do not happen to exist at any point in time. A large vector, whose elements are typically empty, might be considered wasteful – depending on the circumstance.

Vmalloc [Vo1996] uses a top-down splay tree as its tracking mechanism.

In the case of a tree of free lists, the tree itself is usually built out of the free blocks – the blocks in the tree point to specific free lists. Typically the tree is ordered by block size, but alternately, the tree could be ordered by address.

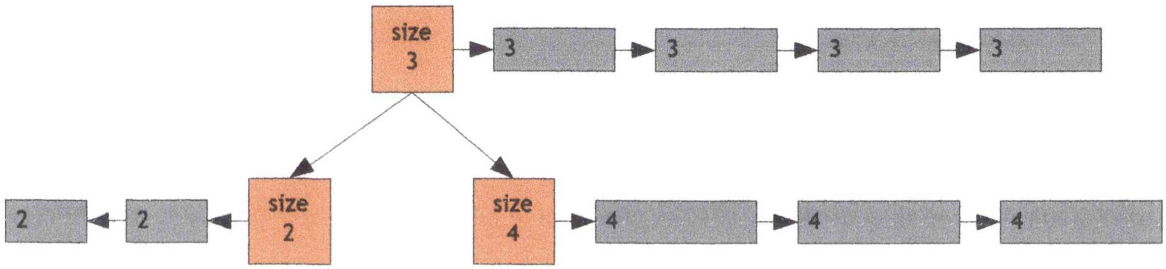


Figure 2.8: An example of a binary tree of three free lists.

2.5.4 Bitmaps

Rather than tracking free space via lists, a completely different technique is to use a bitmap (or bit table) to record what parts of a pool are free, versus what parts are allocated [IGK1971]. This is logically akin to space maps used for file systems.

In a bitmap of this kind, a single bit represents a number of bytes. Allocation and freeing are a matter of flipping the sequence of bits, which represent the state of the block in question.

An advantage of this technique is that, coalescing tends to happen automatically. Freeing a block requires flipping the representative bits. If the block happens to be next to an already freed block, then the new bit sequence simply forms one long run.

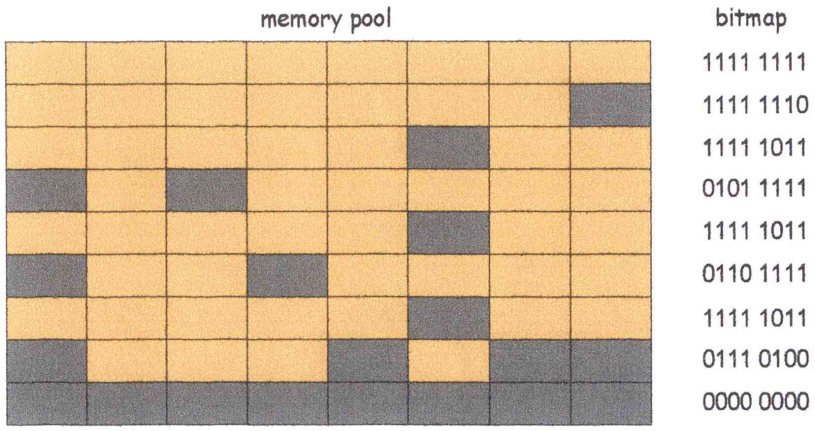


Figure 2.9: An example of a bitmap used to track free space in a memory pool.

A possible disadvantage of this technique is that the bitmap represents a fixed overhead – the memory used for the bitmap is usually not available for any other use.

2.6 List Ordering

In the case where an allocator tracks free space, via some form of list(s), there are design choices to be made about how the list is managed: FIFO, LIFO, size ordered and address ordered are standard choices. As will be discussed in the next chapter, list ordering has a significant effect on fragmentation, locality and performance.

2.6.1 FIFO

In the FIFO (first in first out) technique, blocks are put onto the tail of the list, when they become free, and blocks are taken from the head of the list, when allocated. This has the effect of using older free blocks, before younger ones.

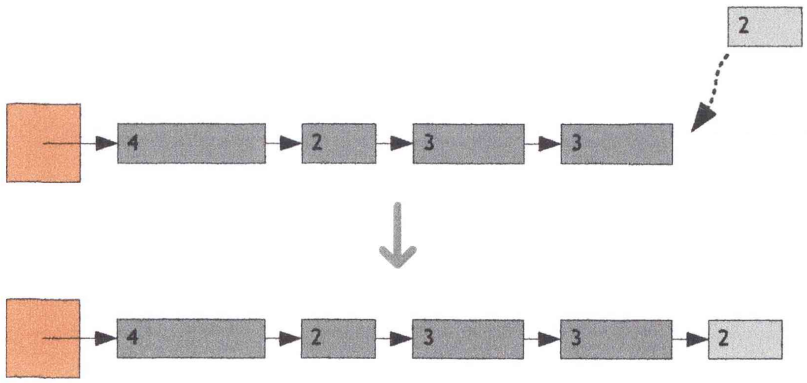


Figure 2.10: A FIFO ordered free list showing a block being added at the tail.

The FIFO technique is useful for delaying the reuse of blocks, so as to increase the probability of coalescence.

2.6.2 LIFO

In the LIFO (last in first out) technique, blocks are put onto the head of list, when they become free, and blocks are taken from the head of the list, when allocated. This has the effect of using younger free blocks, before older ones.

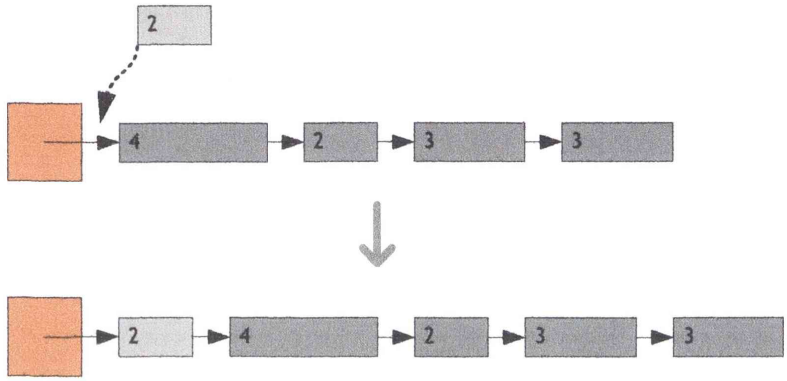


Figure 2.11: A LIFO ordered free list showing a block being added at the head.

The LIFO technique is useful for improving cache hit rates.

2.6.3 Size Ordered

In the SO (size ordered) technique, blocks are inserted into a position in the list, based on their size when they become free. This has the effect of keeping the blocks in the list sorted by size.

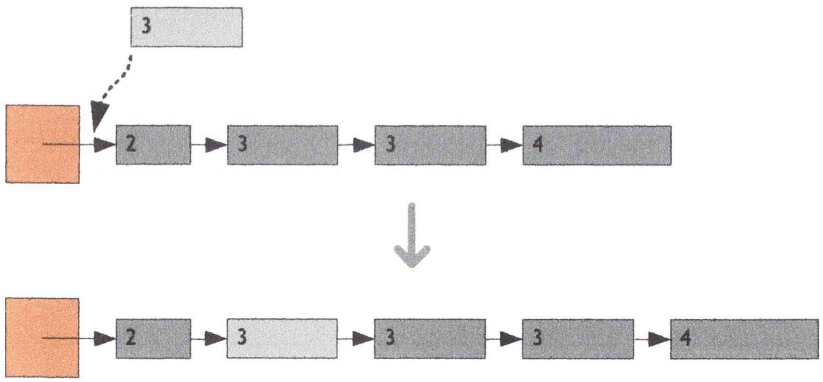


Figure 2.12: A size ordered free list showing a block being inserted midway.

The size ordered technique is useful in limiting searches. A search through a size ordered list can be terminated early, if it can be determined that no better sized block could be found by searching the remainder of the list – DLmalloc's sorted bins are size ordered free lists [Lea2000].

2.6.4 Address Ordered

In the AO (address ordered) technique, blocks are inserted into position within a free list, based on their address. This has the effect of keeping the blocks in the list sorted by address.

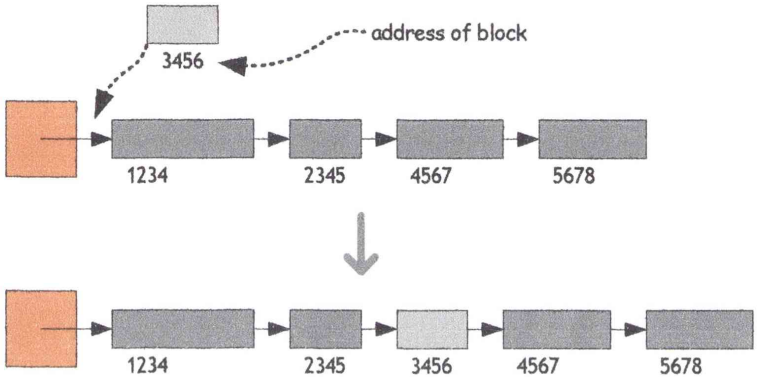


Figure 2.13: An address ordered free list with a block being inserted by address.

The address ordered technique is useful for finding the free block which is closest to some point in memory.

2.7 Allocation Policy

When an allocator receives an allocation request, a decision has to be made as to which free block will be used to service the request – given potentially many possible choices. Typical policies include first fit, next fit, best fit, worst fit, and exact fit. As will be discussed in the next chapter, allocation policy has a significant effect on fragmentation.

In the following discussion on each of the policies, recall that free lists may be FIFO, LIFO, size ordered or address ordered – depending on the design.

2.7.1 First fit

With the first fit allocation policy, the search for a suitable free block, in a given list, starts from the head of the list each time. First fit stops on the first free block, which could be used to satisfy the request – irrespective of the excess.

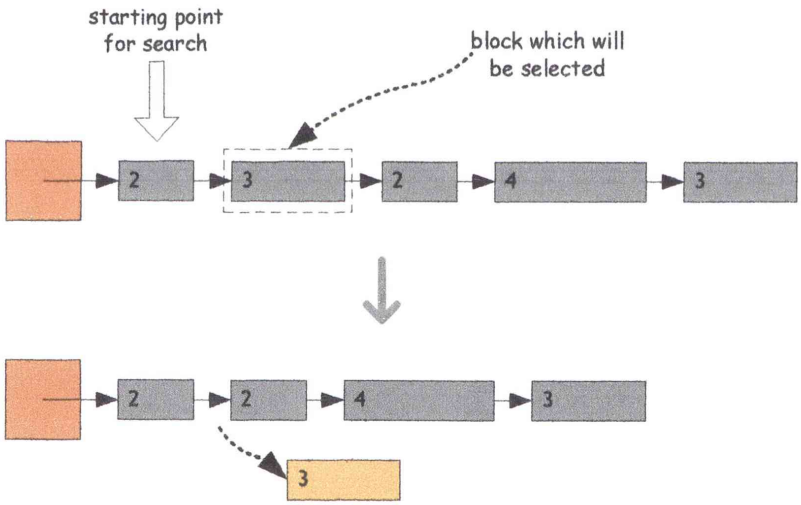


Figure 2.14: First fit allocation policy showing the list being searched from the head, for the first block which satisfies the request.

2.7.2 Next Fit

With the next fit allocation policy, the search for a suitable free block, in a given list, starts from wherever the last search left off – potentially wrapping at the end of the list, back to the head of the list. Next fit stops on the first free block which could be used to satisfy the request – irrespective of the excess.

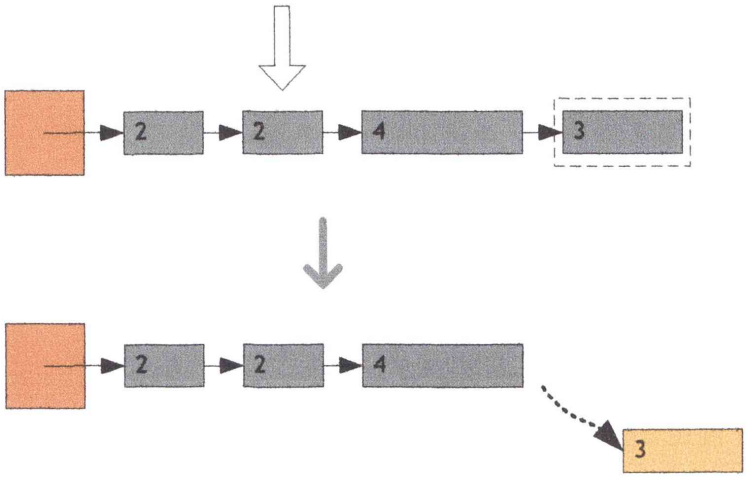


Figure 2.15: Next fit allocation policy showing the list being searched from where the last search ended, looking for the next block which satisfies the request.

2.7.3 Best Fit

With the best fit allocation policy, the search for a suitable free block, in a given list, starts from the head of the list each time. Best fit searches through the entire list, looking at each possibility, to determine the best possible fit – the block whose size is closest to the one required. The search can be terminated early, if an exact fit is found.

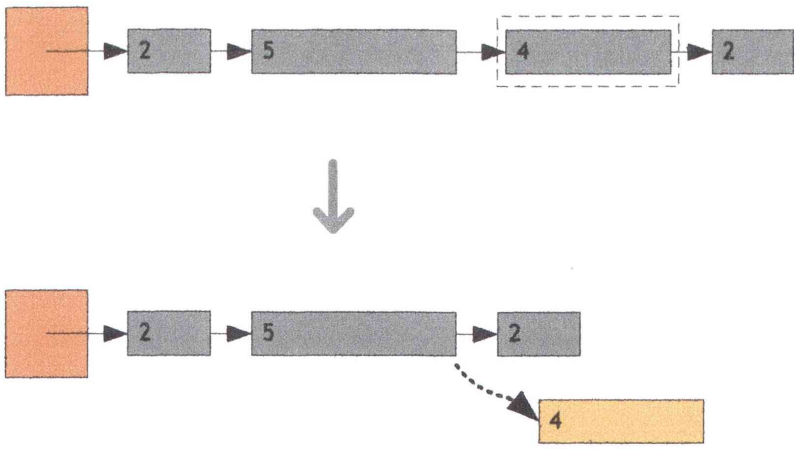


Figure 2.16: Best fit allocation policy showing the selection of the closest size which satisfies the request.

2.7.4 Worst Fit

With the worst fit allocation policy, the search for a suitable free block starts from the head of list each time. Worst fit searches through the entire list, looking at each possibility, to determine the worst possible fit – i.e. the largest block.

While this may sound unusual at first, the intent is to split the large block into two – one of which is the exact size. As will be explained in the next chapter, allocators can be designed to exploit the behavioral patterns of programs. If a program allocates blocks of particular sizes in an even distribution, then a worst fit algorithm tends to carve up memory into the sizes most needed by the program.

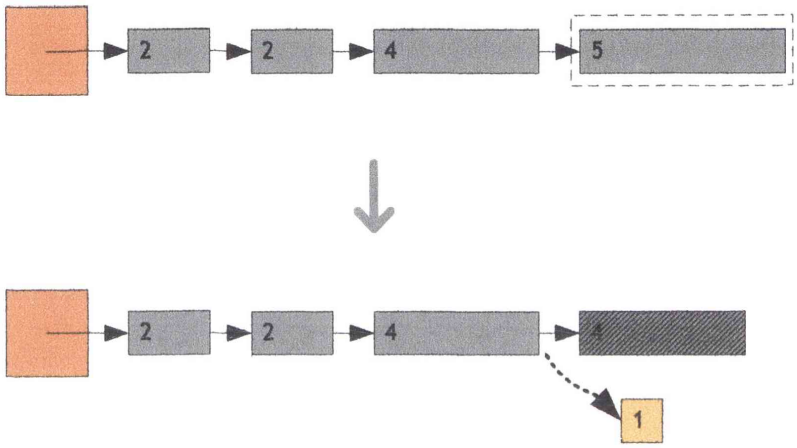


Figure 2.17: Worst fit allocation policy showing the largest block is selected and then carved up (remainder being returned to the list).

2.7.5 Exact Fit

With the exact fit allocation technique, the search for a suitable free block, in a given list, starts from the head of list each time. Exact fit searches through the entire list, looking at each possibility, for the first exact fit – i.e. the first block which is the same size as that required.

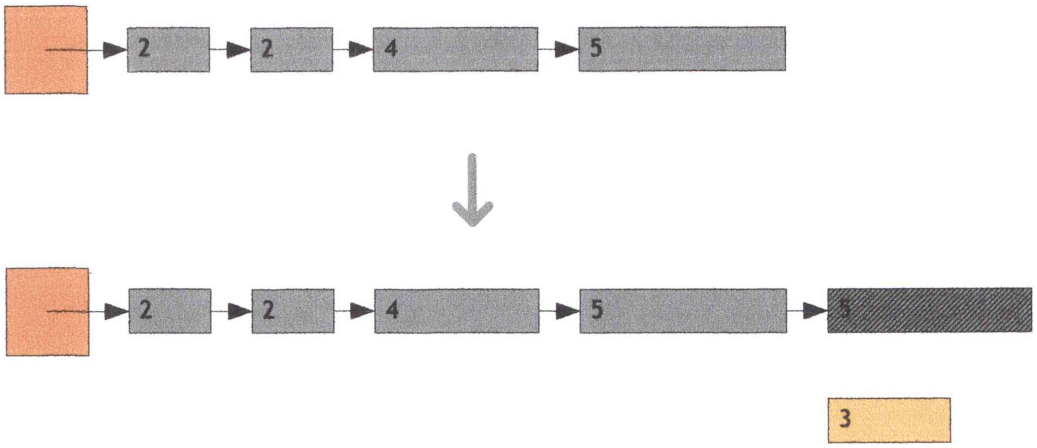


Figure 2.18: Exact fit allocation policy showing no suitable block was found so memory was extended (and then carved up) to satisfy the request.

If an exact fit is not found, then virtual memory is extended, and the new space is carved up into blocks of the size being request. Like worst fit, this is an attempt to exploit behavioral patterns in the calling program.

2.8 Split Policy

In response to an allocation request, an allocator will identify a candidate free block which can be used to satisfy the request. The design question arises: What happens when the choice is bigger than the request? Three choices are available: always split, never split or threshold split.

2.8.1 Always Split

In the always split policy, a block which is larger than needed, is always split into two – such that the split off block is put back into the heap, making that block available to be used to satisfy some future request.

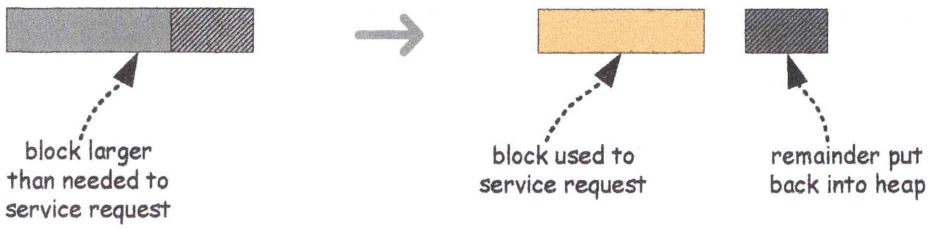


Figure 2.19: Always split policy.

2.8.2 Never Split

In the never split policy, whenever a suitable free block is found, the block is never split – even if that block is larger than what is needed. An allocator may implement a never split policy for performance reasons. Note, however, that a never split policy leads to external fragmentation.

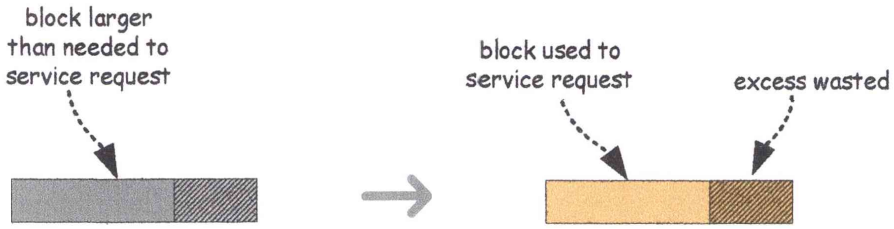


Figure 2.20: Never split policy.

2.8.3 Threshold Split

In the threshold split policy, a block which is larger than needed, is only split into two if the resulting, split off block, would be larger than some threshold size. The objective is to minimize the number of smaller blocks.

2.9 Coalesce Policy

When a block is returned to heap, the design choice arises of what to do when one, or both, of the neighboring blocks are also free. There are three choices: always coalesce, never coalesce or deferred coalescing.

2.9.1 Always Coalesce

With an always coalesce policy, a block being returned to the heap, will always be checked, to see if it can be joined with its neighbors to form a larger free block. Freeing always triggers coalescing

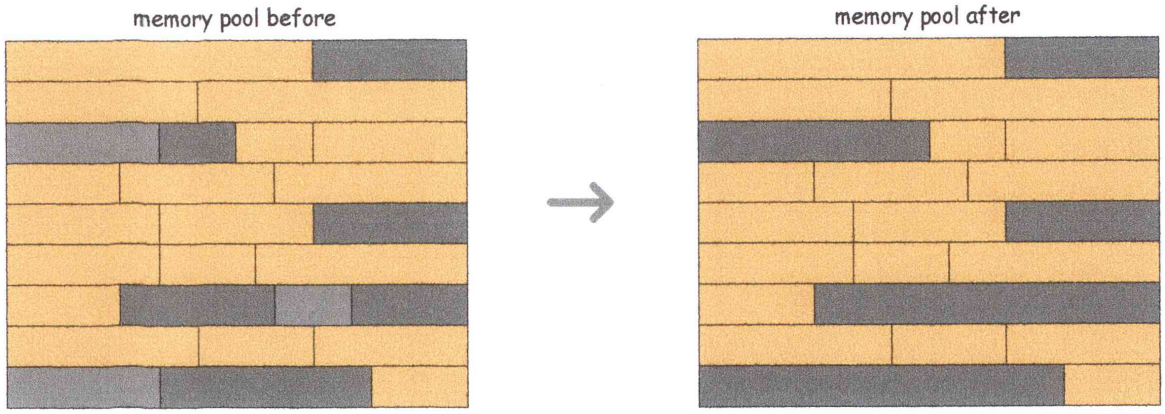


Figure 2.21: Always coalesce policy merges adjacent free blocks when they are freed.

2.9.2 Never Coalesce

With a never coalesce policy, freed blocks are simply never coalesced. Once a block is formed, the block remains that way for the rest of the pools existence – it is never joined with other free blocks, to form larger free blocks. Freeing never triggers coalescing.

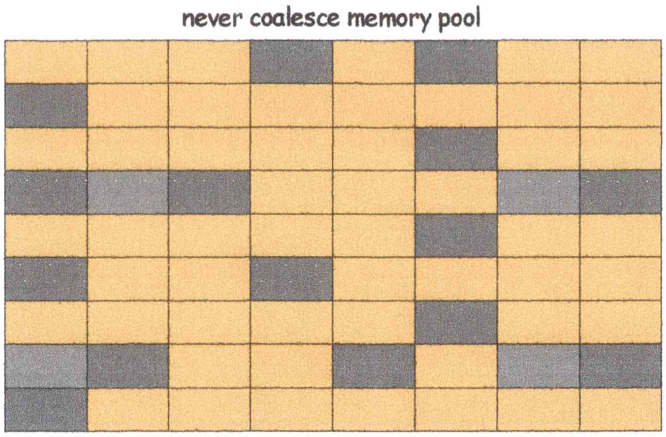


Figure 2.22: Never coalesce policy permits adjacent free blocks.

An allocator may choose a never coalesce policy for performance reasons.

2.9.3 Deferred Coalescing

With a deferred coalescing policy, blocks being returned to a heap do not necessarily trigger coalescing. Coalescing is deferred, until triggered by some other event – such as when an allocation cannot be satisfied. In which case, the allocator would first coalesce and then try again to satisfy the request, only extending memory as a last resort.

2.10 Memory Compaction

One technique to minimize fragmentation is to periodically compact memory. In many situations though, this is not possible, as compaction changes the address of allocated blocks. The two design choices are compacting or non-compacting.

2.10.1 Compacting

Some allocators, particularly those used specifically in handle-based environments with automatic garbage collection, can be compacting – they periodically shuffle a subset of blocks together, so as to maximize the free space pieces.

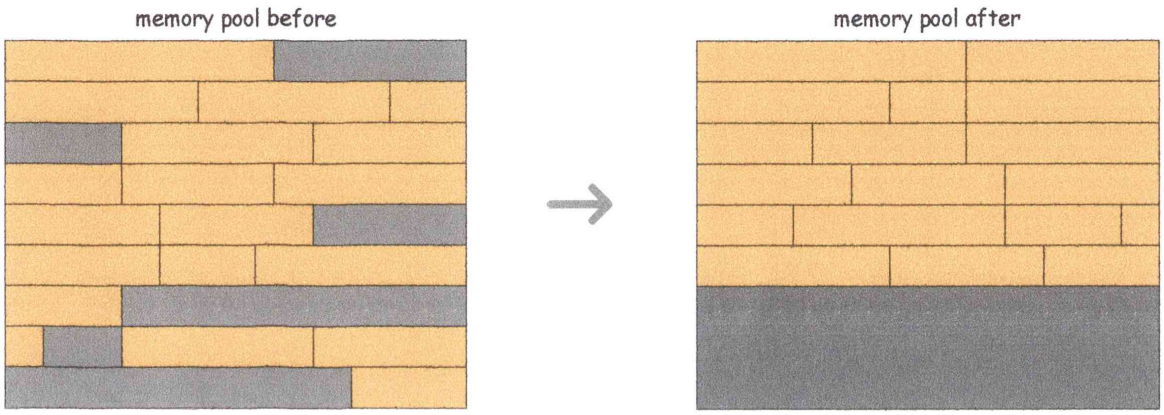


Figure 2.23: Compacting a memory pool shuffles allocated blocks together to combine free space.

2.10.2 Non-Compacting

In many cases allocators are non-compacting – they do not periodically move a subset of blocks, so as to maximize free space. Often, this is due to the fact that the allocator cannot inspect allocated blocks, in order to detect and change memory references.

2.11 Memory Extension

Most allocators would not request the full extent of virtual memory, when they are first invoked. Rather, allocators tend to request more virtual memory pages from the operating system, if and only, when required. A design decision has to be made about how to treat the new region of memory when the heap is extended.

2.11.1 Preserved

In the preserved model, when virtual memory is extended, the extension is treated as one large block and put into the heap as free space.

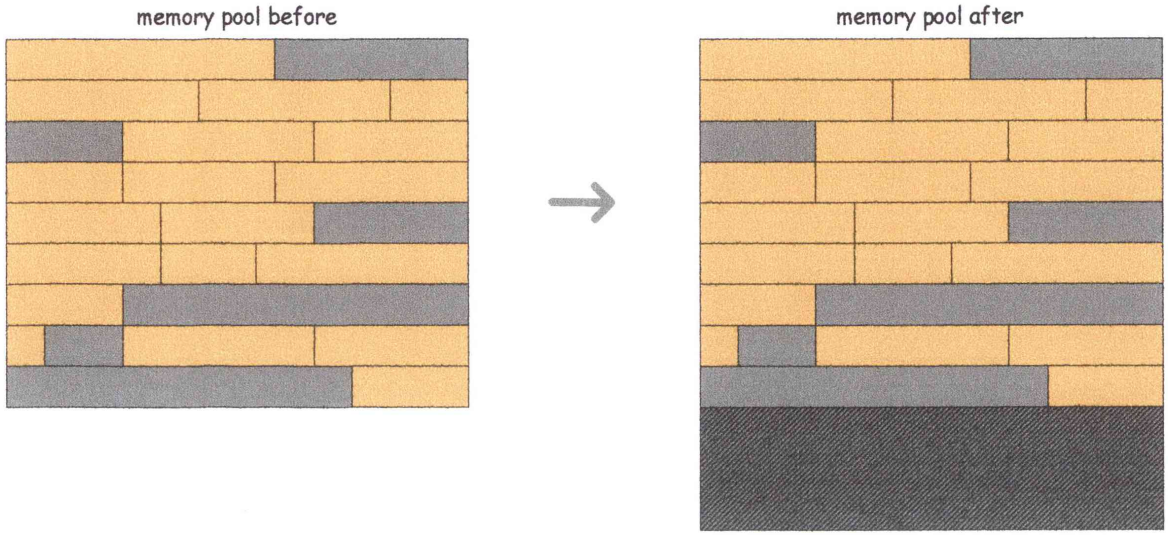


Figure 2.24: Preserving the memory extension by adding it as one large free block.

Also note that many allocators attempt to use the extension as conservatively as possible, leading to the concept of *wilderness preservation* [Lea2000]. The importance is that the extension should be used slowly, in order to dampen the rate by which memory is extended.

2.11.2 Carved Up

In the carved up model, when virtual memory is extended, the extension is immediately split into a number of blocks, using some chosen block size(s). The intent with such a technique is to "top up" the commonly used sizes.

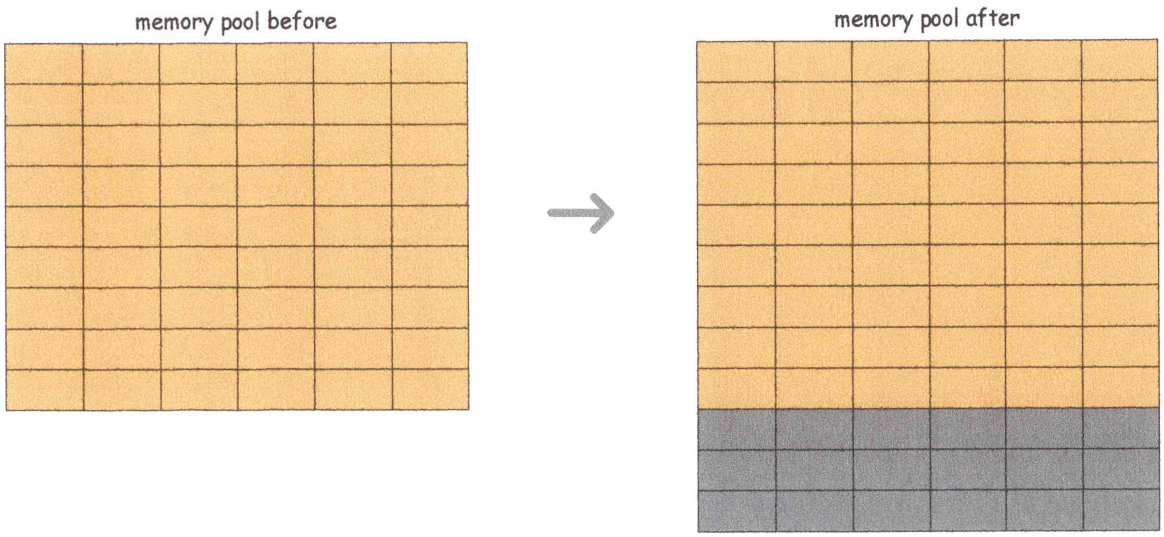


Figure 2.25: Carving up the memory extension into a commonly used size.

2.12 Memory Reduction

Memory reduction (a.k.a. *trimming* [Lea2000]) is the inverse of memory extension – the ability to return unneeded virtual memory back to the operating system

2.12.1 Trim

In the trim model, an allocator is designed to detect the situation when the heap is using more virtual memory than is required, and return pages back to the operating system. As is discussed in the next chapter, the ability to do this is important when programs go through different execution phases, requiring differing amounts of memory.

2.12.2 Never Trim

In the never trim model, an allocator is designed to never return memory to the operation system. Not testing for the specifics of this situation, and consequently not actually returning memory, is usually simpler and faster than doing so.

2.13 Boundary Tags

A certain amount of overhead is incurred in describing and tracking blocks. Traditionally, this information is kept in headers and footers (also known as *boundary tags* [Knu1973]) directly alongside/with each block, rather than in some other structure.

2.13.1 Header

A header is often used to keep a number of fields describing each block. This typically includes a field (which may be a single bit) indicating if the block is currently allocated or free, plus a field indicating how long the block is.

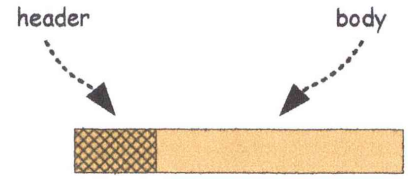


Figure 2.26: A block header.

The header may also have other information, such as which pool the block belongs to.

2.13.2 Footer

A footer is often used to indicate how long the previous block is, and whether that previous block is allocated or not – primarily in order to support coalescing. Footers appear after the main body of the block.

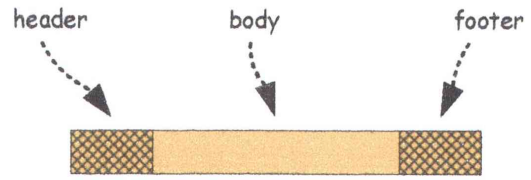


Figure 2.27: A block footer.

In order to minimize overheads, the footer is often factored out. This can be done by having an additional field, in each header, indicating the length of the previous block. To determine if the previous block is free, the size of the previous block is used to perform the appropriate address calculation, so that the previous block's header can be inspected.

2.14 Block Sizing

Digital computers are discrete devices. Consequently, memory is only divisible into discrete units. Coupled with overheads and alignment issues, there are certain albeit simple, restrictions on block sizes.

2.14.1 Minimum Size

Because there is an overhead in tracking blocks, there is usually a minimum size that can be allocated. In many allocators, this is typically two words of memory.

2.14.2 Size Increment

Because of word alignment requirements, there is often a minimum size increment. Again, in many allocators, this is two words. For example, a request for 9 bytes on a 32-bit machine may, in fact, return a block which is 16 bytes long. Observe that while 12 bytes would be long enough, it is three words. A 12 byte block would leave the next block on an odd word boundary, which in many cases, causes an alignment fault, if that next block were to contain a multiword object, such as a double floating point value.

2.14.3 Maximum Size

Many allocators place an upper limit on the size of the block which can be allocated on the heap itself. Above this limit, the allocator typically uses an alternate technique (such as direct memory mapping or virtual page allocation) to service those requests.

2.15 Concurrency

How a dynamic memory manager deals with concurrency (simultaneous access by more than one thread) will directly affect its performance and scalability – particularly for large SMP environments.

2.15.1 Single Threaded

In a single threaded model, only one thread is permitted to be executing within the dynamic memory sub-system, at any one time. If need be, the sub-system is guarded by a single lock. Obviously, this is likely to translate to the lowest throughput possible in an SMP environment.

2.15.2 Multithreaded

In a multithreaded model, more than one thread is permitted to be executing within the dynamic memory sub-system, at any one time. A multithreaded model can be *coarse* or *fine grained*. In a coarse grained model, there are multiple pools but only one thread

can access any one pool, at any point in time. In a fine grained model, more than one thread can access a given pool, at any time.

2.16 Summary

The following table summarizes the fundamental techniques presented in this chapter.

In the next chapter we discuss the time-space tradeoffs of allocator design. This includes issues such as patterned behavior of programs, performance, fragmentation, spatial and temporal locality, scalability, cache effects, robustness and reliability.

APPLICABILITY	• General purpose • Special purpose
MEMORY VIEW	• Free form • Patterned
MEMORY PARTITIONING	• Partitioned • Non-partitioned
POOL CONSTITUTION	• Mixed • Segregated
SPACE TRACKING	• List • Vector of lists • Tree of lists • Bitmap
LIST ORDERING	• FIFO • LIFO • Size ordered • Address ordered
ALLOCATION POLICY	• First fit • Next fit • Best fit • Worst fit • Exact fit
SPLIT POLICY	• Always split • Never split • Threshold split
COALESCE POLICY	• Always coalesce • Never coalesce • Deferred coalesce
MEMORY COMPACTION	• Compacting • Non-compacting
MEMORY EXTENSION	• Preserve • Carve-up
MEMORY REDUCTION	• Trim • Never trim
BOUNDARY TAGS	• Header • Footer • Footer factored out
BLOCK SIZING	• Minimum size • Maximum size • Size increment
CONCURRENCY	• Single threaded • Multi-threaded (coarse or fine grained)

Table 2.1: Summary of fundamental DMM issues and techniques.

Chapter 3

3 Engineering Issues

Virtually all sophisticated engineering activities are a complex tradeoff between conflicting issues, such as time and space. In computing, data structures and algorithms which yield high performance may be inefficient in their memory use, while conversely, memory efficient variants may not support high performance. Achieving both high performance and memory efficiency is rarely a straightforward exercise.

In this chapter, we take a more holistic approach and look at the relationship and interplay between the various factors, which must be considered during allocator design.

3.1 Behavioral Patterns

The role of an allocator is to provide dynamic memory services to a calling program. There are an infinite number of possible programs, and consequently, an infinite number of possible memory usage behaviors. At first blush, one may tend to consider the memory request stream to be completely random.

Programs, in practice though, tend to exhibit patterned behavior – such as the sequence in which blocks of particular sizes are allocated, and then freed. This is not surprising considering programs are constructed by human beings, using common disciplines and techniques, while computers are discrete symbol manipulation machines, repetitiously performing calculations and making structured decisions within nested loops.

The fact that programs exhibit behavioral patterns is arguably the quintessential quality affecting an allocator. Behavioral patterns that are sympathetic to the way a specific allocator works, can lead to good performance and low fragmentation, while behavioral patterns that are discordant, can readily lead to poor performance and high fragmentation.

A general purpose allocator cannot be concordant to all possible program behaviors, for there is insufficient information conveyed by the traditional interface. This gives rise to allocators which are designed for specific purposes [GZ1993], or to overloaded interfaces [Vo1996].

3.1.1 Various Behaviors

There are five behavioral aspects of interest:

1. **Sizes** – programs have a tendency to require blocks of particular sizes, rather than use all possible sizes. The majority of sizes have been shown to be relatively small – less than a few hundred bytes;

2. **Quantities** – programs have a tendency to require blocks of particular sizes in differing quantities, rather than use a similar number of blocks evenly distributed across particular sizes;

3. **Allocation** – programs tend to allocate blocks of particular sizes and quantities in particular temporal patterns;

4. **Freeing** – programs tend to free blocks of particular sizes and quantities in particular temporal patterns, that are quite distinct from allocation patterns; and

5. **Phases** – programs tend to change their size, quantity, allocation and freeing patterns periodically. For example, a program may read through a file in order to load data into a dynamically allocated data structure, thereby requiring blocks of a particular size. Once the data structure is created, the program may then perform some other activity, allocating and freeing blocks of certain sizes different to those in the first phase.

Behavioral patterns of programs, particularly with respect to dynamic memory management, have been poorly recognized to date [WJNB1995] and this remains an open topic.

3.2 Fragmentation

One of the main problems an allocator must deal with is fragmentation. As described in Chapter 1, fragmentation results in unusable free space. After a program has made a number of allocation and free requests, there is likely to be free space scattered throughout the heap. A non-compacting allocator may not be capable of satisfying a future allocation request (without extending memory), because there is no single unit of

free space large enough – even though, in aggregate, there is more than sufficient free space[5].

This leads to an extemporaneous decision of whether a heap is fragmented or not. A heap, at a given point in time with a particular arrangement of free space, may or may not be considered fragmented, if all future requests fit within those free spaces. Fragmentation tends to be more a matter of subjective degree – up until exhaustion occurs, when fragmentation is then absolute.

3.2.1 External versus Internal Fragmentation

Fragmentation, as originally described by Randell [Ran1969], occurs in two forms. The first form, known as *external fragmentation*, is the one just described – the heap records the fact that there is free space, but that free space is in units too small to be useful. External fragmentation is minimized by coalescing adjacent free blocks.

The second, more subtle form, known as *internal fragmentation*, occurs when a block is used to satisfy an allocation request, but the size of the block is a relatively poor fit – the block is larger than needed but is not split. A block may not be split because the remainder may not be a useful size, or because of alignment/rounding issues – the request size may not be a whole multiple of units the allocator deals with. Being larger than what is requested, the resulting block has some degree of waste internal to it.

3.2.2 Fragmentation is Insoluble

One of the interesting aspects about fragmentation is that it has been proven that no solution exists [Rob1971, GGU1972, Rob1974, Rob1977]. No matter the allocation policy there is always at least one pathological case that will force severe fragmentation.

This has important ramifications for many programs and systems – particularly long running ones [WJNB1995]. Of concern is that a program or system may execute for some time, exhibiting acceptable memory usage when, due to some perturbation, its memory usage pattern changes causing severe fragmentation leading to memory exhaustion and consequent failure. Analytical [Rob1977] and empirical evidence

---

[5] Loosely speaking, one might describe this as the "swiss-cheesy-ness" of the heap.

[Sho1975] however, have shown that even in constrained environments, exhaustion from fragmentation rarely ever occurs.

3.2.3 Causes of Fragmentation

[WJNB1995] established that there are two fundamental causes of fragmentation:

1. **Isolated freeing** – a heap becomes fragmented when it becomes pocked by free space. This occurs when adjacent blocks are not freed at similar times; and
2. **Time-varying behavior** – fragmentation arises from changes in the way a program uses memory. A latter execution phase may use different sized blocks from that of a previous phase. The free space left over from a previous phase may not be suitable for a latter one.

Fragmentation is a function of block sizes, juxtaposition and the variation of these over time.

3.2.4 Fragmentation is Manageable

A seminal piece of research into fragmentation was by Johnstone [Joh1997]. Johnstone explored the fragmentation properties of 53 different allocation policies – from first fit address ordered to buddy systems. And he did such using eight different programs, which were known to have quite different and varying memory behaviors.

The relevance of Johnstone's work is that he empirically shows there are a number of allocation policies, which yield acceptable low degrees of fragmentation. Conversely, he also shows there are a number of allocation policies, which yield unacceptably high degrees of fragmentation.

Fragmentation is insoluble, but on the whole, it is a manageable problem.

3.3 Locality

Programs typically exhibit a property called locality of reference – they tend to reuse memory locations that have been recently used. A widely held rule of thumb is that a program spends 90% of its execution time, in only 10% of its code. This property is a fundamental driving force behind the memory hierarchy designs of modern computer

systems – largely due to the fact that processor speed continues to double every eighteen months, while memory speeds are increasing at only five percent per annum [Sei2000].

3.3.1 Spatial versus Temporal Locality

There are two kinds of locality [Joh1997]:

1. **Spatial locality** – memory locations, whose addresses are near one another, tend to be referenced close together in time; and
2. **Temporal locality** – memory locations, which have been accessed more recently, tend to be accessed again.

3.3.2 The Implications of Locality

Locality has a number of important implications for an allocator. For example, an allocation algorithm which searches through a long list, looking for the best fit, may have the tendency to invalidate the cache. An allocator, which implemented an address ordered allocation policy, may provide very good spatial locality to the associated program. An allocator which implemented a LIFO allocation policy may perform well because the memory blocks last used, may still be in cache – leading to fewer cache misses/CPU stalls.

Indirect interaction between an allocator and its associated program may also be an issue – the behavior of the two may complement, or interfere with, each other [GZH1993].

Locality, and its implications with reference to DMM, has had minimal study to date.

3.3.3 Locality versus Fragmentation

The interplay and tradeoffs between locality and fragmentation need to be considered when designing an allocator. An allocation policy that exhibits good locality, may also exhibit high fragmentation – or vice versa.

Another important contribution by Johnstone [Joh1997] is the empirical proof that locality and fragmentation do not need to be traded off against each other – there exist a number of allocation policies which achieve good locality and low fragmentation. We examine those results in Chapter 4.

3.4 Performance

Performance is probably one of the main reasons why so many allocators have been built over the years. The data structures and algorithms being used, directly affect performance – particularly in relation to the amount of memory being managed. A simple linked list, for example, may have O(n) performance, while a tree structure may have O(log n) performance. Such performance may be suitable in constrained applications, but O(n) and O(log n) would be unacceptable for very large environments.

DLmalloc [Lea2000] is generally considered a good performing allocator. Johnstone's research work into fragmentation and locality showed that DLmalloc achieves low fragmentation and excellent locality in practice [Joh1997]. This demonstrates the existence of engineering solutions which balance performance, fragmentation and locality – at least for single threaded allocators.

The degree of concurrency is also likely to be an influencing factor. A single threaded allocator is likely to perform quite poorly for multi-threaded applications – particularly on large scale SMP machines. The most recent studies into allocator performance can be found in [Vo1996, LK1998, VH1999, BMBW2000, Mic2004].

3.4.1 Block Caching

Coalescing blocks in situations which only ever allocate a restricted number of sizes may be a waste of CPU – the larger blocks, formed by the act of coalescing, are simply re-split to create the same sized blocks, as before coalescing took place. In this case, deferred coalescing may be a sensible design choice. One technique, usually referred to as *caching*[6], is to not coalesce blocks when they are first returned to the heap, but keep them in a cache, separate from the general set of free lists. Allocation requests are serviced from the cache first, and then the general lists, second.

DLmalloc [Lea2000] uses this technique. DLmalloc defers coalescing until a request cannot be serviced from its cache. DLmalloc then moves the blocks from the cache into the free lists (known as *bins* in DLmalloc), coalescing the blocks as it proceeds. As an

[6] This should not be confused with hardware caching.

extra performance measure, blocks are only singularly linked when they are in a DLmalloc cache.

3.4.2 Pre-Allocation

One technique for further improving performance, particularly in relation to caching, is to pre-allocate blocks for frequently requested sizes. Rather than splitting off blocks from larger blocks, one at a time, the idea is to pre-split many at once. This is normally faster than splitting blocks one at a time [Lea2000].

3.4.3 Lookaside Buffers

An interesting technique used for simultaneously improving performance and locality is the concept of a *lookaside* buffer [Boz1984]. A lookaside buffer summarizes another structure, into a more compact form. Rather than exhaustively search the larger structure, it is quicker and less likely to cause cache invalidation, to search the condensed lookaside. As an example, Isoda et al. describe a compact bit table approach for a binary buddy system in [IGK1971].

3.5 Alignment

On modern CPU architectures integers, floating point and double floating point values typically need to be aligned on boundaries which are multiples of the word size. This is due to the fact that bus hardware (typically) cannot access multi-byte objects at any arbitrary address. Usually, word-sized objects need to be aligned to word boundaries, double word-sized objects to double-word boundaries and so on. Should a program attempt to access an object that is incorrectly aligned, a bus error of some form occurs.

The address of any block, returned by an allocator, must conform to those boundary constraints. If the block being returned is to hold a data structure, for example, then the block address should be such that, the first item in the structure could be any valid machine object – such as an integer, floating point or double floating point value. Consequently, block sizes are usually rounded up to the next double word boundary to fully cater for alignment – causing a marginal degree of wastage.

Alignment is a systemic cause of internal fragmentation.

3.6 Overheads

Free space management usually takes some spatial overhead – often more so if performance is a consideration. As discussed in Chapter 2, free space is often tracked via multiple lists, organized as vectors or trees. The multiplicity of these structures increases overheads, largely due to any additional memory the structures themselves require.

Furthermore, the existence of boundary tags used to demark blocks reduces the amount of memory which can be used by an application proper. Given boundary tags are typically a small fixed number of words, the ratio of boundary tag size to block size diminishes as the block size increases.

Johnstone [Joh1997] showed that existing allocators typically have spatial overheads between 10 and 35%.

3.7 Scalability

There are two aspects to the scalability of an allocator:

1. **CPU scalability** – how does the performance of an allocator change as we add CPU's? A linearly scalable heap would double its throughput as we double the available amount of CPU in the machine; and
2. **Memory scalability** – how does the performance of an allocator change as we add memory? An allocator with an O(1) rate of allocation and freeing would not change as we add memory, but would handle a larger heap.

There has been very little work on CPU scalability and we can find no specific studies on memory scalability. In 1996 Iyengar [Iye1996] made a very important observation that, to achieve throughput scalability requires a multiplicity of resources (when those resource are locked during various operations), in order to minimize contention:

> *"For machines with many processors, dividing the heap into different regions becomes a necessity. Algorithms sustaining higher throughputs will require fewer subdivisions of the heap for good performance. ... Each processor allocates storage from a different area of the heap. Consequently, the throughput of the heap allocator scales with the number of processors."*

This is the key to all CPU scalable allocators such as Hoard, LKmalloc and VHmalloc. We explore this issue further in Chapter 7. More recent studies into scalability can be found in [LK1998, VH1999, BMBW2000, Mic2004].

3.8 False Sharing

One of the causes of poor performance of SMP applications in general, is *false sharing*. False sharing occurs when threads, concurrently executing on distinct CPU's, share words in the same hardware cache line, without actually sharing data. The result is that, even though the CPU's are accessing different words, the cache line has to be continuously refreshed for each CPU. Cache lines are typically 32-128 bytes; whole lines, not words are moved between memory levels. Excessive refresh of this form leads to thrashing – with a significant reduction in throughput.

Allocators can inadvertently induce false sharing in their calling programs [Ber2002]. This occurs when an allocator services multiple allocation requests for distinct threads, by using the same contiguous region of memory.

3.9 Blowup

Blowup is the phenomena where an allocator fails to be able to reallocate previously freed blocks, consequently consuming an ever increasing amount of memory over time, ultimately leading to premature memory exhaustion [BMBW2000].

Blowup is a subtle form of design defect for allocators which use multiple sub-heaps, particularly occurring when the heap is divided into sub-heaps for scalability reasons (as Iyengar pointed out above), but the sub-heaps are statically assigned to one thread or processor, or are not circulated in a balanced fashion. Even though there is available free memory, it is in part(s) of the heap which is inaccessible to a thread servicing an allocation request, which could otherwise be satisfied.

3.10 Robustness

Dynamic memory management has become such a pervasive technique that it affects virtually all programs and systems. Consequently, the robustness and reliability of an allocator, impacts the robustness and reliability of all programs and systems which use that allocator.

This is particularly important for long running programs and continuously executing systems – where restarting a poorly behaving system is generally not acceptable. In such cases, the quality of the engineering is paramount. An allocator, that would be used in a system which is some way connected to physical safety, for example, must have a very low defect density otherwise the whole system is compromised.

3.11 Evaluation

How the various aspects of allocator behavior are evaluated, is a topic often debated in the literature [WJNB1995]. There are two schools of thought on how this should be done:

1. **Synthetic tests** – in this approach, a test harness is used to drive the allocator while its behavior is being measured. This can be done by having the test harness call the allocator directly, or indirectly, by using previously generated data files; and

2. **Real tests** – in this approach, real world programs are used to drive the allocator, again either directly or indirectly.

Both approaches are valid in our opinion – particularly depending on the circumstance. For example (and in the case in question) if a very large dynamic memory manager does not exist, then *ipso facto*, programs which use very large heaps do not yet exist. In order to test the very first very large dynamic memory manager, synthetic test(s) of some form will be required.

Therefore, one could argue synthetic tests are first order tests, while real tests are second, or subsequent order tests.

In the next chapter we pause to review the fifty years of literature on this topic. We observe how the evolution of dynamic memory management is entwined with the evolution of hardware, operating systems, data structures and programming languages.

Chapter 4

4 History of the Art

In the modern genre of scalable SMP computing, we often take for granted the ability to safely execute multiple multi-threaded programs simultaneously on a single machine; where such programs dynamically synthesize large, sophisticated and mutable data structures as directed graphs of nodes with multiple attributes in non-sequential locations in memory.

Fifty years ago this was anything but the case.

The history of dynamic memory management is intrinsically linked with the evolution of the CPU, memory architectures, operating systems, data structures and the very concept of the program itself. The literature, particularly from the 1950's and 60's, embodies not just the latter, but records an important legacy – a glimpse into the development of the mentality and thought processes which lead to that evolution.

In the coming chapters we will describe the state of the art and then advance it, by specifically exploring very large, highly concurrent dynamic memory management. For the moment, it is worth pausing to retrace the path by which the computing community arrived at this point.

Before we begin our journey into the history of dynamic memory management, it is important to state that the literature records only a subset of the work done by a larger community. Some papers record invention and specific breakthroughs, while others form part of a much larger mosaic of topics being grappled with by that community. A significant number of people contributed to the development of scalable SMP computing – history only records a few[7].

[7] We also refer the reader to Appendix C for a timeline on the history of DMM.

4.1 The 1950's and 60's

By way of stark contrast, early machine architectures were substantially different in their programming model. Prior to the use of transistors, many instructions sets had each machine instruction carry with it the address of the next instruction. This was referred to as *one-plus-one addressing* – there was no implicit program counter [Ros1969].

One-plus-one addressing arose because of the mechanical device known as a *drum memory* [HK1954], which was used as a supplement to core memory for machines like the IBM 650[8] [Knu1960, Knu1973]. Because of the mechanical nature of drum memory, when a location was read from the drum the next serial location had already passed the head, so it was not optimal to read the next serial location. Consequently, when programs were loaded for execution, the constituent machine instructions were distributed around the drum according to optimal rotational access patterns – calculated *a priori*.

The modern reader would appreciate that, by implication, programming environments were far more restricted and complex, compared to what we enjoy today[9].

4.1.1 The UNIVAC – Circa 1952

One of the earliest papers to mention memory management was by Grace Murray Hopper[10] of Remington Rand Corporation [Hop1952]. This paper describes the programming model of the UNIVAC machine[11]. From that paper one can garner that computing at that time was mainly considered the province of mathematical

---

[8] The rotating magnetic drum for the IBM 650 had a 2,000 word capacity and random access time of 2.496 ms.

[9] An excellent (and somewhat entertaining) description of the life of a 1950's programmer can be found in [Ros1986].

[10] Hopper later went on to be instrumental in the development of COBOL. In 1969 she was awarded the first ever Computer Science Man-of-the-Year Award from the Data Processing Management Association.

[11] UNIVAC-1 had 5,200 vacuum tubes, weighed 29,000 pounds, and consumed 125 kilowatts of power.

computation – there appears to be no support for strings let alone complex data structures.

Historically it is worth noting that many programming artifacts were already in place by this stage – compilation, libraries, subroutines, threading, arguments, results, constants, variables along with fixed and floating point values. String manipulation, trees and structures/records with multiple attributes, however, had not yet emerged – yet alone dynamic memory management.

4.1.2 FORTRAN – Automatic Programming

In 1954 a team headed by John Backus developed FORTRAN the first high level language [Bac1978] to provide "automatic programming in order to overcome the machines shortcomings" [12]. FORTRAN I did not have any support for records/structures or dynamic memory allocation, quoting Backus:

> *"In our naïve unawareness of language design problems – of course we knew nothing of many issues which were later thought to be important, e.g. block structure, conditional expressions, type declarations – it seemed to us that once one had the notions of the assignment statement, the subscripted variable, and the DO statement in hand, ... then the remaining problems of language design were trivial."*

Backus recounts that many of the difficulties they faced were due to the idiosyncrasies of the machines at that time – notably instruction sets with arbitrary restrictions (e.g. AND instructions but not OR), as well as the lack of index registers.

4.1.3 Emergence of Symbolic List Processing

By the mid 1950's mathematicians and scientists had started attempting to use computers to do symbolic manipulation – rather than just numerical calculation. The first software tool which supported such endeavors, was a language called IPL (which stood for *Information Processing Language*) developed by Newell, Shaw and Simon.

---

[12] This paper gives a rare insight into the nature of programming and what it was like in the 1950's.

IPL-II was designed and built in the spring of 1956 on Rand Corporation's JOHNNIAC[13] computer [NT1960], and used in writing one of the first AI programs called the LOGIC THEORIST[14].

IPL was the first symbolic list processing language. Note that by the term *symbolic list* we mean a sequence like (x, (y, b, c), d, e), as opposed to the data structure called a list – the general class of data structures had not yet emerged.

In IPL a symbolic list was maintained by a very simple construct – a linked set of nodes (a node was called a *cell* in IPL parlance), each consisting of an attribute and a symbolic name reference to the next node. IPL programs were held in the same format and an interpreter stepped through the program lists in order to execute the program. There was a very limited amount of working storage; programmers using a stack for additional work space.

From a memory management perspective it is important to note that IPL supported a structure called the *Available Space List*[15], which enabled the programmer to acquire and release nodes dynamically. Quoting from [NT1960]:

> *"This device frees the programmer from problems of memory assignment, and allows him to apply at will various processes that modify the structure of memory."*

Dynamic memory had emerged.

The concepts of IPL were often described as NSS (after the inventors' initials) and a number of library packages, supplying similar functionality for FORTRAN, appeared soon thereafter [GHG1960].

---

[13] Named after John von Neumann despite his protestations. JOHNNIAC'S CPU was vacuum tube based and had 4K magnetic core memory supplemented with a 12K drum.

[14] The LOGIC THEORIST could solve theorems presented in *Principia Mathematica* – a book of theorems and proofs by Alfred Whitehead and Bertrand Russell.

[15] Often referred to as the "list of available space list" to remind the programmer that the structure was identical to a program as well as the other data structures used in a program.

4.1.4 LISP

A very important milestone in the evolution of symbolic processing, data structures and memory management was the development of LISP by John McCarthy in 1959 [McC1960, McC1999, BR1964]. LISP provided very sophisticated symbolic list processing capabilities, and enabled programs to be specified in mathematical function notation.

From the very first version memory was managed automatically in LISP through the use of a *Free-Storage List*. McCarthy is also credited for inventing garbage collection[16]:

> *"This process, because it is entirely automatic, is more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists."*

Given the machine architectures of the day this would have to be considered an extraordinary feat – contemporary work at the time was grappling with problems such as the lack of nested subscripting in languages such as FORTRAN [Car1959].

4.1.5 Threaded and Knotted Lists

As the popularity of symbolic list processing grew, the issues of how to traverse symbolic lists underwent considerable investigation. Of notable concern was that recursive traversal required back tracking information to be held in some form of explicit stack – block structured languages (notably ALGOL) which implicitly supported recursion had yet to rise to prominence. This was compounded by the desire to support more complex arrangements – such as sub-lists being attached to multiple super-lists – and the desire to traverse by iteration rather than by recursion.

Threaded lists [PT1960, EPV1961] were developed so as to eliminate the need for explicit push down stacks, by having the last node point to the parent list[17]. *Knotted*

---

[16] For those readers interested in garbage collection one of the earliest papers on reference counting (designed to improve on McCarthy's *reclamation cycle*) can be found in [Col1960].

[17] Alan. J. Perlis was awarded the first ever ACM Turing Award in 1966 for his influence in the area of advanced programming techniques and compiler construction.

lists [Wei1962] were developed to allow multiple programs to traverse a single list, by moving the push down stack into the list itself.

4.1.6 Automatic Storage Allocation

While languages and data structures were in their early stages of evolution in the early 1960's, so were operating systems. One of the critical problems that needed to be addressed was storage allocation – specifically how multiple concurrent/overlapping program and data segments could be assigned to and released from memory in a dynamic fashion [Cod1960, Coh1961, Fot1961, HL1961, Hol1961, Kel1961, One1961, Ris1961, Rob1961, Sam1961a, Sam1961b, RK1968].

Collins [Col1961] gave a rare insight into the state of operating system development, referring to operating systems as *programming systems*, in accordance with contemporary nomenclature. At that time a programming system, consisting of an execution control module, an input/output supervisor, task lists, task processors, a memory map and a storage allocator was being developed for the IBM 7090[18].

The memory allocator as an identifiable sub-system had emerged.

Quoting from [Col1961]:

> *"an allocator which maintains the memory map and upon request assigns space in the high-speed memory as well as the system files."*

Collins then went on to discuss the issues they faced in designing an allocator, decomposing the problem into two: choosing locations for items and keeping account of the choices. In order to "*assist us in the design of the algorithm for assigning random-access memory, we have conducted a set of experiments in which we tested various assignment procedures under simulated conditions of requests for and release of memory space*".

Here we see early examples of computer simulation being used to build some of the first fundamental operating system components.

---

[18] Part of the 700 series, the IBM 7090 was the first commercial transistorized computer. Built in 1960, the 7090 computer was the fastest computer in the world at the time.

The various assignment procedures Collins tested included best fit, first fit, worst fit and random fit. It is interesting to note that, although the tests only ran for 100 iterations, they concluded best fit gave the best overall results, followed closely by first fit[19].

4.1.7 ALGOL 60

Peter Naur was the editor of the report on the algorithmic language ALGOL 60 [Nau1960]. ALGOL is of historical importance because it was the first block structured imperative language. In its first version ALGOL did not have records (data structure support). An early storage allocation scheme for ALGOL 60 on a machine with core memory and a magnetic drum was described in [JMN1961].

4.1.8 Language Support

About the same time Maher [Mah1961] described a dynamic storage allocation system for the Burroughs B5000[20] – although it appears to be functionally overloaded, by comparison, with modern allocators:

> *"This routine is responsible for the loading of programs, the assignment of memory locations for the program segments, the assignment of input-output areas (with provisions for semiautomatic tanking (sic.) if memory space permits), and the assigning of input-output unit designation for all files."*

Maher then went on to describe possible ways in which languages, notable ALGOL, could interact dynamically with the allocation system.

Sattley [Sat1961] and Ingerman [Ing1961] described extensions to ALGOL to support dynamic memory allocation by introducing "*a pseudo-vector referred to as 'Memory'*". The array represented the remaining memory after a program had been loaded.

4.1.9 Plex: A Shift in Perspective

Up until about 1961 it appears that lists and trees consisted of nodes which were single words. Consequently, when packed, nodes could only contain a limited amount of data

---

[19] Which, 36 years later Johnstone [Joh1997] concurred after exhaustive empirical studies.

[20] A dual processor machine with up to eight 4K core memory modules and two 32K drums.

– perhaps two values or addresses and a flag or two[21]. Being single words, such nodes were often referred to as *singlets* [Com1964].

In 1961 Douglas Ross published a paper [Ros1961] on a technique he called a *plex* – being an abbreviation for the word plexus[22]. The important property of a plex was that it was a directed graph of nodes, where the nodes had an arbitrary number of attributes. Nodes were described as "n-component elements".

Quoting Ross:

> *"The components of an element may be of any form. They may be one-bit quantities, machine addresses, machine instructions, symbolic information, or numerical data in any appropriate number form. In particular (and this is the means whereby the system includes all other known symbol manipulation schemes[23]), a component of an element may be a 'link' or reference to another element."*

The generic data structure had arrived[24]. Ross however made no mention of how these structures were to be represented in memory.

4.1.10 Abstract Trees

A year later Gerald Salton published a paper describing the use of generic trees [Sal1962]. Salton described a tree as:

> *"A special type of directed graph consisting of elements (nodes) and associations between pairs of elements (branches)... The initial and final nodes of a maximal path are called respectively a root and leaf of the tree."*

---

[21] Example machines of the day often had 36 or 48-bit words, with only 24-bit addressing. Consequently, words could be packed with multiple items.

[22] Plex: any intertwined or interwoven mass; an intricate arrangement or collection of things; a web; a network – The New Shorter Oxford English Dictionary.

[23] As an aside, imagine the consequences had Ross been able to patent his plex arrangement!

[24] Wirth and Hoare credit Ross as being the pioneer of the concept of records [WH1966].

Salton discussed how trees could be used for a broad range of problems including indexing in classification systems, such as the Universal Decimal Classification System for the Library of Congress, as well as for syntactic and semantic analysis.

Salton then proceeded to show how trees can be represented, searched and manipulated within a matrix. One possible and reasonable inference is that, the software tools and techniques for representing trees directly in memory were not yet generally available.

4.1.11 A Complete Subsystem

Iliffe and Jodeit [IJ1962] presented an example of the dynamic storage interface being developed for the Rice University computer[25]. The interface hinged around a technique called *codewords*. A codeword was a single word, packed with five fields, which described a *block*. Quoting Iliffe and Jodeit:

> *"Consider a store S of consecutively numbered locations in the range (L,H). A sequence of consecutive locations within S will be termed a block and identified by the pair [F,N] giving N(>0), the number of words or elements in the block, and F, the address of the first element."*

The concept of a memory block, as we know it today, had emerged.

Iliffe and Jodeit then proceeded to describe how codewords, being memory descriptors, could be created statically by a compiler, or dynamically by a program to allocate and free memory as required. Within this arrangement, free memory was kept in an *inactive chain*.

Allocation was described as three staged: initially first fit was attempted, failing that coalescing would be done until a block of sufficient size was found, and failing that compaction was done.

It is interesting to note that, the usage envisaged by Iliffe and Jodeit, was for dynamically sized arrays and not for more general use, such as dynamically constructed directed graphs.

---

[25] A custom built 54-bit vacuum tube machine capable of addressing a maximum of 32,768 words of main memory.

4.1.12 The List Data Structure

An important paper was published by Weizenbaum in 1963 [Wei1963], entitled "Symmetric List Processor" a.k.a. SLIP – a FORTRAN based library which was the first implementation of the list data structure. This paper records the emergence of the list data structure (as opposed to the symbolic list). Quoting Weizenbaum:

> *"Slip is a list processing system in which each list cell carries both a forward and a backward link as well as a datum."*

One can judge its perceived importance in its time, by the following excerpt from the editor's note, which accompanied the papers publication (including the SLIP source code) in Communications of the ACM:

> *"Publication of the detailed appendix to the paper 'Symmetric List Processor' is a departure from the normal practice of the Programming Languages department of omitting detailed FORTRAN listings except for examples. This exception is being made because of considerable interest in the SLIP system..."*

With SLIP any number of lists and sub-lists could be created. The programmer however, had to declare (via the FORTRAN dimension statement) a single dimension array which would become, borrowing from IPL terminology the *list of available space*. All SLIP cells (which were word pairs) were taken from and returned to this list – so they were still not truly dynamic with respect to memory.

Bowlden [Bow1963] described a technique for managing a dynamic table of arbitrary length strings using two word cells in ALGOL. Other languages and packages were being experimented with at the time, notably COMIT [Yng1963] and SASP [BBB1964]. A good comparison of COMIT, IPL, LISP and SLIP can be found in [BR1964].

4.1.13 Multiword List Items

In 1964 Comfort put forward the proposition that a node, in a two-way or symmetric list, could contain multiple words – much along the lines of the PLEX structure proposed by Ross [Ros1961] – and in fact could be varying in length. Comfort noted that such an approach leads to more optimal memory use and has diverse applications, such as in representing sparse matrices.

Comfort went on to discuss how a generalized space list, sorted by address with immediate coalescing, could be used to support such an arrangement. Comfort compared this to the arrangement proposed by Iliffe and Jodeit [IJ1962].

The concept of a directed graph of arbitrary length nodes mapped directly onto memory blocks had emerged.

In the course of Comfort's dialogue, he took the reader through a form of induction – discussing single word, double word, triple word and then the general case of multiple word items. In discussing how one might approach managing the set of free blocks, he postulated that one could keep a free list for each available size – thereby hitting on the idea which later becomes the segregated fit or vector of free lists.

The implications of Comfort's work were immediately picked up by Berztiss [Ber1965] in proposing STRIL – a string manipulation library for FORTRAN which used "*blocks of indefinite size*".

4.1.14 A Possible Counter Example

Much of the above discussion surrounded the use of computers for numeric and symbolic computing. Eric Wolman of Bell Telephone Laboratories published a paper [Wol1965] describing the use of computer for message processing. Of concern at the time was that the length of each message was not known as it arrived, yet space had to be allocated. This lead to two problems:

> *"Difficulties occur both in assigning storage space and in keeping track of the locations of messages."*

The solution proposed by Wolman was to create blocks of fixed size which matched the "*fixed optimum cell-size for records of various lengths*", where optimum was interpreted as the "*cell size that minimizes the mean (with respect to message lengths) amount of space per message.*"

4.1.15 A Fast Storage Allocator

In 1965 Kenneth Knowlton published a paper [Kno1965] describing an allocator for the language known as LLLLLL or L^6 (Bell Telephone Laboratories Low-Level List Language). The technique employed became known as the binary buddy system:

> *"This scheme ... makes available blocks of computer registers in several different sizes: the smaller blocks are obtained by successively splitting larger ones in half, and the larger blocks are reconstituted if and when their parts are simultaneously free."*

The described allocator was capable of handling blocks in the range 1-128 words. In his paper, Knowlton acknowledged he became aware at the time, of publication of a variation of the idea, which was used in SIMSCRIPT in 1963 – although it only handled size 1-8 words. Knuth [Knu1973] also recalls that "*independent invention*".

Knowlton also discussed delayed coalescing being based on a number of different heuristics, such as quantity thresholds, not just memory exhaustion.

4.1.16 ALGOL Gets Records

The support for records in ALGOL was first proposed in 1966, by Wirth and Hoare [WH1966]. Quoting from the paper:

> *"This reference may be assigned as the value of a suitable field in another record, with which the given record has some meaningful relationship. In this way, groups of records may be linked in structural networks of any desired complexity."*

Records were adopted in ALGOL 68.

4.1.17 Still Constrained

In 1967 John Abramowich [Abr1967] eluded to the issue that, computers were still quite constrained devices, by describing a storage allocation technique for eliminating records, out of an input table(s) as they were processed, to make room for output table(s).

4.1.18 The AED Free Storage Package

In 1967 Douglas Ross (who had proposed the plex structure) described the AED[26] Free Storage Package, being developed since 1959 by the Computer Applications Group of the MIT Electronic Systems Laboratory [Ros1967]. This allocator was very sophisticated – even by modern standards. With over 50 interface routines described, it may in some respects, be one of the most sophisticated ever built. This was a true sub-system, automatically managing all of available free memory, as opposed to having free space declared by the programmer *a priori*.

The AED Free Storage Package had many rich features, including segregated hierarchical views of memory controllable by the programmer, the ability to allocate blocks of any size, multiple allocation techniques, and programmer extension via callbacks. It was also designed to be highly portable, with releases scheduled for the IBM 7094, the IBM 360, the UNIVAC 1108 and the GE 645.

Ross described fragmentation as "*physical memory peppered full of holes*".

4.1.19 PL/I List Processing

In 1967 the PL/I programming language was formally enhanced by IBM to support list processing [Law1967]. The controlled storage class was added to PL/I type system and the terms *based variable* and *pointer variable* were added to the nomenclature. Dynamic storage management also correspondingly appeared:

> *"The example will illustrate the ability to create and manipulate a list structure where the elements are dynamically created via the ALLOCATE statement. A based variable named in the ALLOCATE statement causes an amount of main storage equal to the size of the based variable to be reserved. In addition a SET clause in the statement names a pointer variable which will be set to 'point to' the allocated main storage. The storage allocated may also be released via a FREE statement which names the based variable".*

---

[26] Automated Engineering Design or ALGOL Extended for Design.

We see here the earliest example of the elegant, two function allocator interface, which is common today.

4.1.20 External v. Internal Fragmentation

Randell [Ran1969] introduced the distinction between external and internal fragmentation. He noted that fragmentation, at that time, was also known as "*fracturing or checker-boarding*".

Randell and his team at IBM were exploring the tension between storage allocation and hardware, operating systems and compiler design. Of particular interest were the implications for paging versus segment overlay systems. Randell noted that MULTICS had two different pages sizes, being 64 and 1024 words.

In order to better understand fragmentation, Randell conducted a simulation experiment, comparing a best fit and a random fit policy against a theoretical policy (which compacted memory after each release). He discovered best fit to be superior to random fit.

Additionally, he experimented with rounding up allocation requests to certain threshold sizes – in order to minimize the number of block sizes the allocator had to deal with. To his surprise, more storage was lost from internal fragmentation, than what was saved by lower external fragmentation[27].

4.2 The 1970's

4.2.1 Measuring Segment Sizes

Batson et al. [BJW1970] examined the distribution of segment sizes, on a Burroughs B5500 at the University of Virginia, running mainly ALGOL programs. The Burroughs B5500 was capable of allocating core storage in segments which varied from 1 to 1023 words. At the time, there seems to have been only one other experimental study on the distribution of segment/blocks sizes, by Totschek at SDC.

---

[27] Which is an important observation to make – particularly for buddy systems which round up implicitly.

They discovered that 60% of the segments in use contained less than 40 words. This phenomenon is still prevalent today.

4.2.2 Statistical Properties of the Buddy System

Purdom and Stigler [PS1970] performed a mathematical analysis on the behavior of buddy systems. In particular, they examined the effect arrival and service times have on the interaction between, the bottom level of a buddy system, and the next level up.

The paper provides a very good description of how a buddy system works.

4.2.3 Virtual Memory

Peter Denning [Den1970] discussed the evolution of virtual memory, tracing its origins from manual overlay techniques in the 1950's. Denning described virtual memory as the separation of address space from memory space and specifically states that list processing (along with machine independence and program modularity) was one of the key factors driving that evolution.

4.2.4 Optimal Fit Policy

Campbell [Cam1971] observed that first fit tended to cause a high degree of fragmentation, while best fit tended to perform poorly. Campbell suggested and evaluated a technique, called *optimal fit*, which produced lower fragmentation than first fit.

In the optimal fit policy, a certain number of blocks are examined as a sample, then the next block which is better than all sampled, is selected. Campbell relates this to the stopping problem on a Markov chain of given length.

Campbell stated that, one of his motivations was to find a good dynamic storage algorithm to support LISP programs, being used for differential equations in celestial mechanics.

4.2.5 Bit Tables

Isoda et al. [IGK1971] explored a bitmap technique which used one bit to record the state of one word. They refer to their technique as the *tail lamp system* – as only the last

bit for a varying length block was flipped on allocation/deallocation. They observed that this lead to efficient coalescing, as only one bit is flipped to merge two blocks.

They used that technique as the control mechanism for a buddy system[28]. They also noted error checking could be introduced via an equivalent *heap lamp system* – where the first word in a varying length block is asserted before being flipped on allocation or deallocation.

4.2.6 Operating System Research

Margolin et al. [MPS1971] claimed that observed properties of actual programs should be considered when designing DMM algorithms. The paper described their research work leading up to the storage management subsystem of the IBM CP-67[29] version 3. Dynamic memory was not only required for list processing but for I/O and supervisor services.

Their research was in three stages and entirely empirical[30]. The first stage, involved collecting trace data, pertaining to every allocation and deallocation of memory from a production system, which they captured to tape[31]. This included call timings and block sizes.

The second stage consisted of development and experimentation with various algorithms. This included pre-allocation for given sizes, rounding up request sizes to given thresholds and segregating into sub-pools by size[32]. A new algorithm, based on pre-allocation and pooling of specific sizes with deferred coalescing, was developed.

The third stage was to systematically test the final algorithm, by booting the old and new versions of the operating system, on different days over a two week period.

---

[28] Implemented on a FACOM 270-20/30 – a 16-bit word machine with a 65K word core.

[29] CP: "*a virtual machine control program which provides for each logged in user the environment of a System/360.*"

[30] System observed to manage between 48KB and 100KB.

[31] A full reel of data recorded 960,000 calls and took approximately two hours to collect.

[32] This predates Berger's work on Hoard by nearly 30 years. Refer §4.5.1.

Their conclusion was that the new algorithm resulted in a speed up of 7 or 8 to 1.

4.2.7 Storage Analysis

Robson [Rob1971] conducted one of the first mathematical analyses of dynamic storage allocation. He showed that the worst-case requirement for memory has a lower bound, which is a logarithmic function of the size of the blocks allocated i.e. $N \times M \log_2 n$; where n is the maximum size of blocks allocated, M is the extant of memory requested and N is some constant.

Garey et al. [GGU1972] performed mathematical worst-case analysis of first fit and best fit. They compare memory allocation to industrial bin packing problems such as *cutting stock* and *assembly-line balancing*, as well as other computer science problems such as *table formatting*, *pre-paging* and *file allocation.*

4.2.8 Donald E. Knuth

Knuth's classic textbook [Knu1973], first published in 1968, provided an excellent introduction, and comprehensive survey, of dynamic storage allocation. Knuth instructs that free space lists can be constructed using the space in the free blocks themselves. Knuth then went on to discuss first fit, best fit, next fit (a.k.a. modified first fit), boundary tags, buddy systems, splitting, coalescing (a.k.a. collapsing) and thresholds.

Knuth noted that at the time, the behavior and performance of dynamic storage algorithms was based entirely on Monte Carlo experiments (i.e. random synthetic trace simulations). From his own experiments he derived the fifty percent rule – that under certain conditions the length of the free list(s) will tend towards half the number of allocated blocks[33].

4.2.9 Fibonacci System

Following Knuth's suggestion [Knu1973], Hirschberg [Hir1973] built a variant of the buddy system, called the *fibonacci system,* in which the various levels follow a Fibonacci pattern, as opposed to a binary pattern.

---

[33] This "rule" later gets disproved.

Hirschberg conducted a simulation experiment comparing the fibonacci and buddy systems, using Poisson arrival times and exponential service times. The experiment showed the fibonacci system had a lower degree of fragmentation than the buddy system – largely due to the wider range of available block sizes.

4.2.10 Bounds of Dynamic Storage Allocation

Robson [Rob1974] extended his prior work [Rob1971] and examined the amount of memory required by a dynamic storage allocation system to guarantee its operations – given a number of bounds. This is a complex mathematical problem, so Robson made the simplifying assumption that block sizes are always powers of two.

Robson showed that the worst case memory usage for an optimal algorithm[34] was somewhere between 0.5 $M \log_2 n$ and about 0.84 $M \log_2 n$.

4.2.11 Weighted Buddy System

Shen and Peterson [SP1974] acknowledged that the major benefit of the binary buddy system is its performance, but also acknowledged its weakness is memory loss due to internal fragmentation – stemming from the restricted number of blocks sizes available. Consequently, they proposed the *weighted buddy system.*

In the weighted buddy system, block sizes may either be a power of two, or three times a power of two. Thus in their system, block sizes may be 1, 2, 3, 4, 6, 8, 12, 16 ... compared to a binary buddy system which would only permit 1, 2, 4, 8, 16 ...

Shen and Peterson tested their system via simulation using synthetic traces. Two tests were done. The first test used a uniform distribution of block sizes, while the second test used an exponential distribution. Service times (block lifetimes) were a uniform distribution.

The experiments showed that, for uniform block size distributions, the binary buddy system had a lower overall fragmentation, compared to weighted buddy system (27% compared to 34%). For an exponential distribution, the binary buddy system had a

---

[34] Where: M is the total amount of memory in use at any time; and n is the largest block allocated.

higher overall fragmentation, compared to weighted buddy system (29% compared to 22%).

4.2.12 Fibonacci Buddy Coalescing

Hinds [Hin1975] discussed the similarities between the binary buddy system and the fibonacci buddy system. In particular, he noted that the original fibonacci technique proposed by Knowlton [Kno1965], required a lookup table to determine possible block locations, and that lookup table had to be calculated *a priori* for a specific sequence, such as the Fibonacci series.

Hinds proposed an algorithm based on a generalized Fibonacci sequence, within which each block kept a *left buddy count*, indicating how many levels to the left of the block its buddy would be found. This improved the speed of coalescing but required an additional word in the control field of the block.

Citing Hinds' work, Cranston and Thomas [CT1975] discussed the time and space problems, of the known algorithms, for coalescing blocks in the fibonacci buddy system. They present a new algorithm, which represented the state of memory as a binary tree, thereby enabling the fibonacci system to perform as well as binary buddy, while only requiring two additional control bits per block.

They developed and implemented this scheme as part of the SIMPL programming language.

4.2.13 Comparing Best Fit and First Fit

Concerned with the consistency of previous experiments Shore [Sho1975] conducted a series of simulation experiments to "*obtain better data on the relative performance of first fit and best fit and a better understanding of the reasons underlying observed differences.*"

They cited an example from 1963 where best fit had been chosen for the Burroughs D-825 operating system AOSP – Automatic Operating and Scheduling Program:

> "*Best fit had been chosen, in the absence of any literature advising to the contrary, because it was both simple and intuitively efficient. Debilitating*

fragmentation, which had been anticipated with plans for periodic memory compaction, never occurred[35]. "

Shore introduced a measurement of fragmentation called the time-memory product efficiency, which measured the memory utilization of an algorithm over its runtime. The experiments showed that first fit and best fit generally performed within 1 to 3% of each other. They concluded that first fit tended to outperform best fit for exponential and hyper exponential distributions of block sizes, while best fit tended to outperform first fit for normal and uniform distributions.

Shore hypothesized that, when first fit does outperform best fit, it does so because it preferentially allocates blocks at one end of memory, thereby permitting blocks at the other end to age and coalesce, forming much larger free blocks.

4.2.14 Influences on Disk Storage Allocation

Burton [Bur1976] observed that some disk drives (notably ICL) at the time had word, sector, track and cylinder arrangements, which would allow buddy systems to be used to track disk storage allocation. Extending the work by Cranston and Thomas [CT1975], Burton used a variation of the fibonacci buddy system for allocating disk storage for geometrical data[36].

4.2.15 Tracing Execution

Batson and Brundage [BB1977] discussed the concept of an *execution trace* – recording the activity of a program and its requests for resources – in order to examine the size and lifetime of segments in ALGOL programs. The idea for which came from *contour models* [Joh1971] and from block structured languages such as OREGANO [Ber1971].

They modified a Burroughs B5500 ALGOL compiler to add sampling instrumentation to programs. This instrumentation used a software controlled hardware counter, based on

---

35 The fact that debilitating fragmentation never occurred, and this was observed as far back as 1963, is an important fact which seems to have been overlooked in the literature.

36 In a system called BEAUTIFUL – Burton's East Anglia University Topographical Information Facility Utilizing Location.

the 1MHz clock of the B5500. Data samples were written to tape and analyzed on a CDC 6400.

All in all, 34 B5500 ALGOL production programs were examined – including a BASIC compiler, a linear programming package, a differential equation solver and some standard statistical routines[37].

Their experiments indicated that for the programs sampled, 80% of memory blocks were less than 50 words, and had a mean lifetime of around 150-200 machine instructions.

Batson and Brundage concluded by saying, they had no success in fitting their empirical data to the distributions used in probabilistic modeling to date. They suggested that their source language level measurement technique could be used in other performance evaluation studies.

4.2.16 Comparing Next Fit

Bays [Bay1977] used the term *next fit* to refer to modified first fit [Knu1973, Sho1975]. He described a simulation test written in BASIC on a PDP-11 using doubly linked lists.

Using the time-memory product efficiency proposed by Shore [Sho1975], he concluded that next fit is "*decidedly inferior*" to first fit and best fit, when the mean size of blocks is less than $1/16^{th}$ of memory. Beyond which all three were similar.

4.2.17 A More Comprehensive Test

In order to improve the quality of computer based simulations, Nielsen [Nie1977] investigated the performance of 35 dynamic memory allocation algorithms on 18 different test cases – described as "*typical to extreme*". The algorithms were coded in FORTRAN IV and run on an IBM 360/15. The tests were synthetic, with arrival rates being constant, normal, uniform and exponential distributions, while service times being normal, uniform and exponential. Each test had between 3,000 and 30,000 allocations.

---

[37] Virtual memory space requirement ranged from 162 words to 89,976 words.

Nielsen compared the memory consumption, processing time, and external fragmentation of the algorithms under different configurations.

Nielsen observed that algorithms:

- Which maintained separate free space lists for each size, tended to perform quite well compared with other algorithms;
- Operating on memory ordered lists (without any free list) performed "*surprisingly well*"; and
- Employing power-of-two block sizes (buddy systems) had favorable processing times, but generally unfavorable memory usage.

4.2.18 The Reality of Buddy Systems

Peterson and Norman [PN1977] presented two generalized forms of buddy systems, thereby subsuming the binary, fibonacci and weighted variants – one of which permitted any number of buddies to be created when splitting a block.

They performed both mathematical analysis and detailed simulation of the various forms concluding that, as variations of buddy systems reduce internal fragmentation (by increasing the number of block sizes), they increase external fragmentation, owing to a larger number of smaller blocks. They observed that total fragmentation remained relatively constant, with 25 to 40% of memory lost.

4.2.19 Worst Case Fragmentation

Robson [Rob1977] continued his previous mathematical analysis of storage allocation strategies [Rob1971, Rob1974], and proved that the worst case performance of first fit is about $M \log_2 n$ words[38] (which is not far from the optimum), while the worst case for best fit (and similarly for next fit) is $M n$ (which is much worse).

---

[38] Where: M is the total amount of memory in use at any time; and n is the largest block allocated.

4.2.20 Anomalies in Knuth's Rule

Shore [1977] reported on simulation experiments with first fit addressed-ordered memory allocation that empirically showed that, the average free-to-allocated-block ratio could vary considerably from Knuth's "*50 percent rule*" [Knu1973].

Shore argued that *systematic placement* introduces an asymmetry in the allocation process, which subtly interacts with the release process, such that the number of free blocks is not always $\frac{1}{2}pN$ at equilibrium[39]:

> *"Systematic placement, which refers to the natural convention of always allocating storage requests against the same end of the free block selected by the allocation strategy, tends to order blocks within contiguous groups according to their allocation time. The degree of anomalous behavior depends on the extent to which allocated blocks are released in the order of their allocation."*

4.2.21 Standard Interface

According to Vo, in 1979 Doug McIlroy established the now familiar ANSI-C standard interface generally known as *malloc* [Vo1996].

4.2.22 Compacting Lists

Bobrow and Clark [BC1979] discussed contemporary LISP implementations, noting that many implementations were inefficient when the word size was larger than the address size[40]. They studied memory usage in five different programs, ranging from a chemical structure verifier to a speech recognition system, and concluded that lists could be encoded in a compact form, leading to a doubling in space efficiency. They also observed that, for a microcodable processor said improvement in space efficiency, could be gained for little or no additional cost in processing time.

---

[39] Where: p is the ratio of the probability of allocation over release; and N is number of allocated blocks.

[40] The PDP-10 LISP system used 36-bit words, while the XEROX and MIT LISP Machines were both 32-bit words. All used 24-bit addresses.

4.3 The 1980's

4.3.1 BSD Unix

According to Vo, in 1980 Chris Kingsley wrote a binary buddy allocator, which was distributed with versions of BSD UNIX [Vo1996]. It was fast but had very poor memory utilization.

4.3.2 Fragmentation Analysis

Over the course of several years Reeves [Ree1979, Ree1980, Ree1982, Ree1983] performed an exhaustive analytical analysis of fragmentation. Motivated by Knuth's Fifty Percent Rule [Knu1973], Reeves attempts to characterize the state of memory at equilibrium, under a random fit strategy:

> *"In equilibrium, for each block size, the average rate of reservation and release are the same and so the size distribution of release is the same as for requests. Conditions are sought for statistical equilibrium between reservations and releases in a store of N locations operating under a random fit strategy."*

In his very first paper, Reeves made the assertion that "*Knuth's Fifty Percent Rule presupposes a random fit strategy*". Reeves made the claim that first fit and next fit are crude approximations "*to an idealization which we may call 'random fit'*", going on to propose random fit as "*a yardstick against which more efficient practical algorithms may be assessed*".

In rereading Knuth this appears not to be the case. Knuth clearly starts from a first fit strategy and then tests best fit, as well as binary buddy. Knuth clearly uses random sizes and lifetimes, but makes no mention of random fit at all.

What is particularly interesting to note is, that Knuth specifically stated that the rule was only an approximation[41]. He specifically gives examples where observations do not match the rule exactly, himself commenting that "*the fifty-percent rule is subject to statistical variations*".

---

[41] It is entirely possible that by "rule" Knuth meant "rule of thumb".

Shore [Sho1977] had shown that some variations can be attributed to systematic placement. In order to eliminate the source of variations, Reeves used a circular model of memory. Reeves was able to prove that, under certain highly constrained conditions, the lower bound was 48%.

4.3.3 Parallel Memory Allocation

In what appears to be one of the first efforts on parallel memory allocation, Harold Stone [Sto1982] presented a parallel first fit algorithm, which used the FETCH-AND-ADD instruction for concurrency control, in order to achieve a high degree of parallelism in a multiprocessor system. Stone was examining the parallelization of Knuth's first fit algorithm [Knu1973] (which consists of a single doubly linked list) and noted:

> *"The general problem is quite challenging as the number of processors becomes very large because the algorithm for allocating and deallocating memory appears to require serialization. But a serialization of memory allocation could severely compromise performance of a highly parallel system."*

Stone decomposed the first fit algorithm into five fundamental actions (*processes* in his nomenclature) and then showed how to use FETCH-AND-ADD (an atomic instruction which enabled multiple CPU's to simultaneously increment a memory location without data loss) to create a concurrent implementation. The implementation required explicitly detecting interference with other threads and restoring original values before retrying.

No test results were published.

4.3.4 How Optimal is Optimal Fit?

Page [Pag1982] investigated (analytically and using random trace simulation) the frequency by which first fit chooses the best possible free block, as well as the mean time taken to scan the free list[42]. Results were compared with optimal fit by Campbell [Cam1971].

---

[42] Tests were very simple. Memory model was only 1,024 words over a run of only 1,000 steps.

Page observed that first fit performed better than optimal fit in both respects – a counter result to [Cam1971]. Page concluded that this was due to the tendency of first fit to arrange segments in size order – a phenomena first observed by Shore [Sho1975].

4.3.5 Release Match Policy

Motivated by Shore's work in exploring fragmentation [Sho1975, Sho1977], Beck [Bec1982] observed that fragmentation is caused (or increased) by differences in release times between adjacent blocks. Beck mused that fragmentation might be reduced, if free blocks were selected for allocation, based on the release times of the surrounding allocated blocks. This led him to propose the *release match* policy:

> *"Satisfy a storage request by allocating an area at one end of memory, whenever this is possible. Otherwise, allocate the new block adjacent to an existing allocated block whose release time is as close as possible to that of the new block."*

Beck acknowledged that this required knowing the time of deallocation, at the time of allocation – which could either be determined *a priori*, or by observing statistically valid correlations during runtime.

In the cases where it was not possible to do so, Beck also proposed the *age match* policy – choosing a free block, adjacent to an existing allocated block, whose allocation time is as close as possible to that of the new block.

The two new policies were tested, using both random and actual traces. As per Beck's intuition, release match performed better than first fit and best fit, while age match did not perform as well – usefulness decreased as the variation of lifetimes increased.

4.3.6 Adaptive Lists

Leverett and Hibbard [LH1982] described a technique they called *Adaptive Creation of Free Lists* or ACFL. The objective of ACFL was to improve performance, by dynamically creating separate free lists, for each of the different block sizes, which are most frequently allocated.

Leverett and Hibbard were motivated by the observation that, the dynamic storage allocation system used in Carnegie-Mellon's ALGOL-68 implementation, suffered severely from splintering (as described by Knuth [Knu1973]):

> *"The existence of a small number of sizes that are frequently requested wreaks havoc on the structure of the general list."*

The concept of ACFL capitalizes on program pattern behavior producing a high frequency of specific sizes. ACFL was tested against five programs and a performance improvement of approximately 10% was observed.

4.3.7 Fast Fits

Stephenson [Ste1983] introduced a set of techniques called *fast fits* in which free blocks are kept in a Cartesian tree[43] [Vui1980]. His motivation was to reduce the number of blocks that have to be visited, in order to satisfy an allocation or release request. Stephenson's discussion implies that, as at 1983, when first fit and best fit are mentioned one would naturally assume address ordering of the free list (as opposed to FIFO or LIFO).

Stephenson made careful note that a Cartesian tree may become unbalanced and that, in general, cannot be explicitly rebalanced – as nodes are implicitly ordered by address and size.

Tests showed that, for larger memory sizes, fast fits performed better (time wise) than first fit or best fit – due to the lower number of blocks visited (about $1/60^{th}$). It appears the degree of fragmentation did not change[44].

4.3.8 SUN OS

According to Vo, in 1983 Aoki and Adams implemented an allocator using Stephenson's Cartesian tree, which became part of SUNOS [Vo1996].

---

[43] A form of binary tree where (in this case) the nodes are primarily ordered by address and secondarily by block size.

[44] Tests were 10,000 blocks with an average size of 100 words.

4.3.9 UNIX System V Release 4

In 1983 Kiem-Phong Vo implemented a best fit allocator, using a bottom-up splay tree, which became part of UNIX System V Release 4 [Vo1996].

4.3.10 Choosing a new Algorithm for VM/SP

Following on from Margolin et al. [MPS1971], Bozman et al. [BBDT1984] further explored various dynamic storage allocation algorithms, culminating in a new algorithm incorporated in VM/SP HPO[45] Release 2. The impetus for this work came from the observation that, the then current dynamic storage algorithm, was consuming between 11 and 20% of supervisor-state CPU. The algorithm was single threaded:

> *"The high CPU time caused high lock holding time for the primitive lock on the multi-processor that prevents the concurrent execution of the dynamic storage allocator. The high lock time in turn caused longer lock wait time on the other processor."*

Bozman et al. also noticed cache invalidation, due to searching, left the cache "*full of data that are very unlikely to be referenced after the search*".

Like Margolin et al. the team wanted to collect live data in order to drive simulations, but was concerned about perturbing live systems. Consequently, they decided not to collect every request, but rather collect statistical snapshots from three different systems, from which they could derive the mean arrival and service times for each block size. They then assumed exponential distributions for both.

After performing a number of simulations tests comparing the Margolin et al. algorithm with AO, FIFO and LIFO variations of first fit, next fit and best fit, as well as Cartesian trees and buddy systems, Bozman et al. concluded that algorithms which search a single free list (such as first fit, next fit and best fit) are too slow for large systems.

The final chosen algorithm was a variant of Margolin et al.'s, with two levels of sub-pools – one level with lists holding two discrete sizes, the other level with lists holding

---

[45] High Performance Option.

32 discrete sizes. The impact was to reduce supervisor time to 4-5% in some cases, and reduce overall execution time by 10%

4.3.11 Two More Buddy Variants

Kaufman [Kau1984] introduced two new variants of buddy systems – the *tailored list buddy system* and the *recombination delaying buddy system.*

In the tailored list buddy system, a separate list was kept for each block size. The scheme attempted to keep the free lists populated to their relative optimum lengths, in anticipation of demand, by choosing to split or coalesce when satisfying an allocation request. Similarly, for deallocation, a block could be put onto its free list or combined with its buddy, in order to maintain the free list lengths. It was assumed that the size distributions were known *a priori.*

In the recombination delaying buddy system, the allocation scheme was similar to the tailored list buddy system, but coalescing was deferred on deallocation – coalescing being triggered by allocation failure.

Kaufman tested the algorithms using a simulator written in PL/I on a UNIVAC 1100/81, using both synthetic and real traces. He concluded that, for low memory usage/loading scenarios, the tailored list buddy system gave marginally better runtime and fragmentation results, than recombination delaying buddy system. Both of which (in the experiments conducted) performed better than traditional binary buddy. In heavily loaded systems, Kaufman observed no real differences between the three.

4.3.12 The Software Lookaside Buffer

Bozman [Boz1984] discussed the detrimental effects searching a linked list had on cache. He proposed the software analogy of a hardware cache, to hold condensed information about recently used items, held in one or more lists to ameliorate the effect.

Bozman gave three examples of how a software lookaside buffer can be used – one of which was dynamic storage allocation.

By way of evaluation, Bozman conducted a synthetic simulation of first fit and best fit, using real size distributions collected from an IBM VM/SP system. He observed a

reduction in search overhead (number of blocks inspected) of between 73.4% (for best fit) and 81.4% (for first fit)[46].

4.3.13 Using First Fit for Tape Archival

Coffman et al. [CKS1985] observed that, if first fit results in low fragmentation, it could be used to place files onto tapes, by reusing space made available from previously erased files. Compare this to the then current practice, of rewriting the tape so as to find the end marker.

Coffman et al. conjectured that, first fit is asymptotically optimal, in the sense that, the ratio of expected empty space to expected occupied space tends towards zero, as the occupied space tends to infinity[47].

4.3.14 Adaptive List Sizes

Oldehoeft and Allan [OA1985] examined a variation of Bozman's software lookaside buffer [Boz1984] in order to further improve runtime performance. In their technique they used a working set cache to manage free block sizes, and pointers to free lists of those sizes – based on a policy using a time window. When a size was in the working set, the system maintained a list of free blocks for that size – without any coalescing. If a size was evicted from the working set, the blocks on the associated list were returned to a FIFO pool, and coalesced where appropriate.

Using one real trace, and two synthetic traces generated from real distributions, the working set policy exhibited higher block hit rates than FIFO.

Two problems with the technique were considered. Firstly, delayed coalescing could increase the degree of fragmentation, and secondly, phase transitions could cause the working set to be purged, resulting in unpredictable (and potentially unacceptable) execution times thereafter.

---

[46] This may explain the success of DLmalloc's fastbins caching technique. Refer §4.3.18

[47] This is an important point to remember when we get to designing arbitrarily large heaps.

4.3.15 Finer Compacting Algorithms

Baker et al. observed the high processing costs of compacting all of memory upon exhaustion [BCW1985], and proposed two new algorithms, which moved a subset of blocks on demand, to create a contiguous space large enough to satisfy a request. They observed that finding a minimum cost sequence of such moves, is NP-hard.

Both of the proposed schemes used binary power placement strategies, similar to buddy systems, to restrict the number of candidate locations. The schemes differed in the manner by which they selected a region to compact. The first scheme, called *Not Full*, compacted a region of memory based on finding a suitable interval of memory that was not completely full, while the second, called *Not Too Full*, compacted a region of memory based on finding a suitable interval of memory that was, relatively speaking, not more full (percentage wise) than all of memory.

The difficulty of such schemes was keeping the volume of backtracking information required – particularly while compacting large tracts of memory. Baker et al. suggested a few techniques to ameliorate some of the difficulties (such as pruning and keeping such information in the available free space), but it is not clear if the algorithms were ever implemented. The algorithms have O(log n) performance and would be difficult to implement in a concurrent fashion.

4.3.16 Dual Buddy System

In an attempt to address the fragmentation issues associated with buddy systems, Page and Hagins [PH1986] proposed the dual buddy system[48] – a modified form of the weighted buddy system[49] [SP1974].

Previous attempts to reduce internal fragmentation did so by increasing the range of supported sizes, but gave rise to a commensurate increase in external fragmentation. This tradeoff was first observed by Peterson and Norman [PN1977]. Page and Hagins

---

[48] Wilson et al. [WJNB1995] attribute the double buddy system to David Wise citing an unpublished report from 1978.

[49] Recall that the weighted buddy system intermingles two sequences of sizes: 2^n and 3×2^n.

were motivated by the observation that, increasing the block range required increasing the associated tree height – their conjecture was that this was the root cause of increased external fragmentation.

In the dual buddy system, the two series are maintained in two disjoint regions of memory. Consequently, the two individual trees have the same height as the original binary buddy system, so if Page and Hagins' conjecture was right, this would lead to reduced external fragmentation.

Simulation experiments (using synthetic traces), conducted by Page and Hagins, empirically showed that, the dual buddy system had lower overall fragmentation, than the binary buddy and weighted buddy. In particular, the dual buddy system had lower internal fragmentation than binary buddy and substantially lower external fragmentation than weighted buddy.

4.3.17 More Efficient First Fit

Brent [Bre1989] proposed a technique for implementing address order first fit, using a form of balanced binary tree. The advantage of Brent's technique was that it had O(log n) performance and, being a balanced structure, it did not suffer from the potential problem of degrading into O(n), like that of the Cartesian tree technique [Ste1983].

Brent implemented his algorithm in both FORTRAN and PASCAL and tested it on a VAX/VMS system, using Bozman et al.'s distributions [BBDT1984]. As predicted, the algorithm showed excellent results as the overall heap size increased[50].

Additionally, Brent discussed the importance of distinguishing between strategy (which blocks are allocated) and algorithm (how blocks are allocated). He made the point that there can be many algorithms which implement the same strategy. This is later rediscovered by Wilson et al. [WJNB1995]. Brent also notes that, depending on the tests involved, the first fit procedure typically emerges as the best strategy – or close to the best[51].

---

[50] Though only a maximum heap of 306KB was tested.

[51] This was again confirmed by Johnstone [Joh1997].

4.3.18 Doug Lea's Allocator

In 1987 Doug Lea began writing an allocator, which according to the author has never been formally named, but now goes by the name *DLmalloc* [Lea2000]. This allocator is still maintained by Lea today – the latest version being v2.7.2. Quoting the author [Lea2000]:

> *"Why use this malloc? This is not the fastest, most space-conserving, most portable, or most tunable malloc ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for malloc-intensive programs"*

There are two core elements in DLmalloc, which have remained unchanged since the earliest versions:

1. **Boundary Tags** – each block has a header which contains size and status information; and
2. **Binning** – free blocks are maintained in bins[52]. There are 128 bins, approximately logarithmically spaced in size. Bins for sizes less than 512 bytes each hold exactly one size (spaced eight bytes apart). Searches for free blocks are processed in smallest-first, best fit order.

Another key feature of DLmalloc is caching – a method for implementing deferred coalescing. Small blocks are kept in fastbins – an array of lists, holding most recently freed small blocks, which are not coalesced until an allocation request cannot be satisfied.

The major weaknesses of DLmalloc are that it is single threaded and the larger bins hold free blocks sorted by size. Consequently, DLmalloc is not a scalable allocator.

---

[52] Binning is another name for a vector of free lists.

4.4 The 1990's

4.4.1 Fast Allocation

Furusou et al. [FMY1991] discussed the subtleties of the interaction between allocators and garbage collectors for high performance[53] concurrent systems. Their goal was to produce practical and effective implementations of concurrent OO-languages on various parallel architectures.

In order to reduce synchronization overheads, Furusou et al. described two algorithms. The first, called *basic*, employed a novel technique of having the garbage collecting threads do the actual allocation, on behalf of the application threads – application threads issued their requests to the garbage collecting threads via IPC mechanisms.

The basic algorithm was implemented on a variety of machines[54]. While they noted it worked well, it didn't reach their performance objective, due to demand overload on the garbage collecting threads, and the high cost of IPC messages in a multiprocessor environment.

Consequently, a second algorithm, called *extended*, was proposed. The extended algorithm attempted to address these issues, by shifting the responsibility of allocation back to the application threads. A higher degree of parallelism, with little or no synchronization, would be achieved by assigning each application thread a set of superblocks[55], from which it could satisfy allocation requests without any form of locking. Higher performance would be achieved by having only one size per superblock – resulting in a very simple and efficient algorithm[56]. No size to superblock mapping was discussed.

---

[53] Where they define high performance as one million allocations per second.

[54] A LUNA 88K (quad MC88100 running MACH 2.5) and a NEXT (MC68040 running NEXT OS 2.0).

[55] Furusou et al. used the term *chunk*, but this word is now generally used to describe a free block.

[56] Conceptually, very similar to Hoard [Ber2002].

4.4.2 A Persistent Heap

In one of the earliest papers on persistent programming languages, Hosking [Hos1991] briefly discussed the notion of a persistent heap:

> *"The heap is stored in some file on disk and then loaded into memory in its entirety when the environment begins executing."*

This is one of the first papers in what becomes the field of object oriented databases. Hosking discusses concepts such as *object identifiers, object faulting* and *swizzling*. It is interesting to note how quintessential heaps are to so many computing disciplines.

4.4.3 Lifetime Prediction

Barrett and Zorn [BZ1993] explored the possibility of using object lifetime prediction to improve memory allocation performance. They examined five, allocation intensive C programs, and observed that more than 90% of allocations were short-lived. Their hypothesis was that, an allocator could take advantage of that knowledge to clump short-lived objects together, thereby improving reference locality, memory overhead and in some cases performance.

Barrett and Zorn defined *object lifetime* to be: the total number of bytes allocated between the time a specific object is allocated and when it is deallocated. Their definition of *short-lived* is somewhat more intriguing and could possibly be described as a "first order guess". They defined short-lived to be an object lifetime of 32KB or less, admitting it was an arbitrary choice. Their rationale though, is intuitive:

> *"We must realize that the longer we consider 'short-lived' to be, the more objects we will be able to predict as short-lived... On the other hand, the shorter we predict short-lived objects to live, the smaller the region of memory we need to set aside for them. Making this region as small as possible has two advantages. First it decreases the total heap size... Second, it localizes the references to short-lived objects, reducing the cache and page miss rates. In addition to balancing these concerns, we must also consider that each program will have different lifetime distributions."*

The five programs they examined had a maximum heap size of between 35KB and 2.1MB. 32KB was therefore between 3 to 99 % of the base heap size.

Their technique was essentially a profile-based optimization. They used allocation site (including call stack depth) and object size to provide a framework, by which they could identify and track object lifetime. The resulting information was then encoded as a hash-table, accessible by an allocator, which used that information for optimized runs.

Simulation testing produced good results, with some notable exceptions. For the larger program, a 48% reduction in memory consumption was observed. It was also observed however, that accurate or inaccurate lifetime prediction could improve or degrade performance – depending on the circumstance.

4.4.4 Hardware Assistance

Gehringer and Chang [GC1993] explored hardware assisted dynamic memory management. They described a cache coprocessor which, from the point of view of hardware, lay between the processor and memory, appearing as a second-level cache. The coprocessor allocated objects in its cache and performed reference counting, in order to remove most objects before they were written to main memory[57].

The memory allocation technique was called *modified buddy system* – a variation of the buddy system, capable of allocating variable length blocks (due to its hardware implementation) using a bit vector to track allocations.

Gehringer and Chang reported that in-cache allocation eliminated 53-63% of fetch traffic, with less garbage being generated. No performance improvements were stated.

---

[57] This follows Barrett and Zorn's observation [BZ1993] that most objects are short-lived.

4.4.5 Cache Locality

Using a cache simulator called TYCHO, Grunwald et al. [GZH1993] explored the effect different memory allocators had on cache locality[58]. Five allocators were tested (FIRSTFIT, GNU G++, BSD, GNU LOCAL and QUICKFIT) on five different programs[59].

They observed that allocation strategy had a significant impact on cache locality, which affected overall execution performance by as much as 25%, with significant variation between the various allocators. This was in relation to a 16KB cache, and when the cache was increased to 256KB, the cache miss rate dropped to below 0.5%; with little significant variation between the allocators – with the exception of FIRSTFIT which sorts and scans free lists.

Consequently, they drew two important conclusions:

- Maintaining free lists in sorted order takes considerable CPU time; and
- Searching a free list is disastrous for cache locality and page reference.

Their measurements showed that CPU efficient allocators, such as BSD and QUICKFIT, also provided the best locality of reference.

4.4.6 Allocation Costs

Detlefs et al. [DDZ1993] used a profiling tool called QPT [BL1992] to profile the memory allocation costs in 11 large[60], allocation-intensive C and C++ programs[61]. The average instruction count per allocation ranged from 52 to 611; and from 2 to 81 per deallocation. Total time spent in the allocators ranged from 6 to 36%. Heap expansion, due to fragmentation, ranged from 1.5 to 5.5 times the base memory requirement.

---

[58] This is a particularly interesting paper which we later use in guiding the design of our very large dynamic memory manager.

[59] Maximum heap size ranged from 60KB to 4.1MB.

[60] Maximum heap size ranged from 21KB to 3.8MB.

[61] Tests were conducted on a DECSTATION 5000/240 with 112MB of memory.

4.4.7 Parallel Allocation

Iyengar [Iye1993] observed that very limited parallelism is possible with an allocator that manages free storage using a single list. In order to increase parallelism, multiple lists or trees are required. Quick fit [WW1988] is a very good candidate for parallelism due to the number of lists kept – a list for each small block size and one single list (called a *misc* list) for larger sizes, with coalescing usually deferred.

Iyengar noted that while quick fit had a higher degree of parallelism, because different free lists could be search concurrently, it could degenerate for larger block sizes[62]:

> *"In the worst case, every request is for a large block. Quick fit then degenerates into a first fit system with deferred coalescing. First fit systems with deferred coalescing are the slowest storage allocators we studied."*

Iyengar explored three variations of quick fit:

1. **Multiple Free List Fit I (MFLF I)** – similar to quick fit but there were multiple *misc* lists for the larger sizes. The *misc* lists were partitioned based on size ranges;
2. **Modified Quick Fit (MQF)** – similar to quick fit but a concurrent B-tree was used, ordered by block size, to manage a free list per size for larger sizes; and
3. **Multiple Free List Fit II (MFLF II)** – a variation of MFLF I and MQF. It used *misc* lists for commonly requested large block sizes and a B-tree for the rest.

Simulation tests[63] showed MFLF I to be the best algorithm when large blocks of many different sizes were requested – largely due to superior concurrency[64]. MFLF II was the best algorithm when large blocks of only a few different sizes were requested – largely

---

[62] Another important observation to be taken into consideration for a very large DMM.

[63] In an environment called ID WORLD which simulated an ideal parallel machine.

[64] We start to see here that scalability in an SMP environment is achieved by having a multiplicity of lockable resources.

due to reduced searching[65]. MFLF II and MQF utilized memory more efficiently than quick fit and MFLF I.

Additionally, quick fit and MFLF I were implemented on a MONSOON dataflow multiprocessor[66] and tested against six different programs. In some cases MFLF I achieved substantially better performance than quick fit. MFLF I had consistently good performance across different distributions. Iyengar notes that they never encountered real applications where MFLF I performed badly.

No heap sizes are given.

4.4.8 Synthesizing Custom Allocators

Grunwald and Zorn [GZ1993] described CUSTOMALLOC – a program which synthesized a memory allocator, customized for a specific application. The CUSTOMALLOC allocator first measured the memory allocation behavior of an application and then generated a customized allocator using that information. The general structure for a customized allocator consisted of a specifically generated front end allocator and a general back end allocator.

The front end allocator was similar to that of QUICKFIT [WW1988], in that commonly request object sizes were stored on distinct singly-linked lists. The difference being that sizes, gleaned from analyzing the applications memory usage, were explicitly encoded in the generated code.

The goals of CUSTOMALLOC were to: distinguish between frequent and infrequent object sizes; configure the front end allocator; and select an appropriate back end allocator (either a fast non-coalescing allocator or a slower coalescing allocator) – all by measuring program behavior. Measurements showed that synthesized allocators can be from two to ten times faster, than general purpose allocators.

---

[65] We start to see here that searching needs to be bounded in some manner to guarantee throughput.

[66] Consisting of eight 64-bit pipelined processing elements (each interleaving up to eight threads at a time) and eight interleaved memory modules.

4.4.9 Model Evaluation

Zorn and Grunwald [ZG1994] examined the suitability and implications of simulations, which used actual traces versus synthetic traces, as the event stimulus. They evaluated the accuracy of five different models of allocation behavior, against six allocation-intensive C programs, using four different dynamic storage allocation algorithms, in order to see how well the models predicted the actual behavior.

The evaluation models were:

1. **Mean-Value Model (MEAN)** – characterized program behavior by computing three means: the mean object size, object lifetime and request arrival time. The simplest of the five models;

2. **Actual CDF Model (CDF)** – characterized program behavior by computing the cumulative distribution functions of the above three means;

3. **Size-Class-Dependent Model (SIZE)** – computed the three means as a function of each of the object size classes;

4. **Time-Dependent Model (TIME)** – computed the three means over a number of evenly distributed time periods; and

5. **Size Transition Model (TRANS)** – a refinement of the Size-Class-Dependent model, in which there is an assumption of dependence between successive allocations based on size. A conditional probability density, of an allocation of size Y follows an allocation of size X, was computed and held in a matrix.

Using these models, Zorn and Grunwald presented a taxonomy of possible methods to model allocation behavior – including an unnamed class which explored time-size dependencies.

The tests[67] showed wide variations in the average relative error: 25% for MEAN, 18% for CDF, 19% for SIZE, 80% for TIME and 20% for TRANS.

---

[67] Maximum heap size ranged from 17KB to 2.7MB.

Based on detailed analysis of the results, Zorn and Grunwald concluded that relatively simple synthetic models could effectively emulate the allocation behavior of well-behaved programs, but the more complex models could only roughly approximate the behavior of more complex programs.

4.4.10 GNU C

In 1994 Mike Haertel developed an allocator which segregated blocks of the same size into pages specifically for that size[68]. The allocator was distributed with GNU C library [Vo1996].

4.4.11 Customizable C++ Framework

Attardi et al. [AF1994, AFI1998] described a customizable dynamic memory management framework for C++ called CMM – which included facilities for garbage collection. Attardi et al. had observed two important points: firstly, optimal performance can not always be achieved by a uniform general purpose solution; and secondly, it can be very difficult, if not impossible, to mix languages and libraries which have different dynamic memory management and garbage collection approaches. The aim of their work was to address both these issues simultaneously.

In CMM several policies could coexist. While a few default variants were supplied, programmers could also implement their own, via delegation. CMM permitted multiple sub-heaps, each with its own policy[69], with the calling program being able to determine which sub-heap an allocation request was to be satisfied in. This enabled the programmer to influence performance/runtime characteristics, by taking advantage of program/object behaviors.

4.4.12 Critical Review

Wilson et al. [WJNB1995] presented a comprehensive survey and critical review of dynamic storage allocation as at that date.

---

[68] It is not clear from [Ber2002] if Haertel's work influenced the design of Hoard.

[69] Attardi et al. acknowledged the idea of sub-heaps with policies extended back as far as [Col1961].

They covered the full gamut of allocator design, including:

- Defining what an allocator is and what it must do;
- Fragmentation, splitting and coalescing;
- Low-level mechanisms such as boundary tags, alignment; and
- Basic mechanisms such as sequential fits, segregated free lists, buddy systems, indexed fits and bitmapped fits.

Wilson et al. discussed at length the issue of fragmentation, what it is, what causes it, and how it should be studied. They observed that in many cases, allocators are evaluated using probabilistic analysis, noting that the mathematics is too difficult[70] to be able to reliably predict behaviors, or via simulation using synthetic (random) traces, noting the incongruity that most programs exhibit patterned (non-random) behavior. Consequently, they not only strongly agree with Zorn and Grunwald [ZG1994] that synthetic models only approximate behavior, they make the claim that:

> *"From this Zorn and Grunwald conclude that the only reliable method currently available for studying allocators is trace driven simulation with real traces. While this result has received too little attention, we believe that this was a watershed experiment, invalidating most of the prior experimental work in memory allocation."*

Wilson et al. extended the structural separation of concerns (proposed as early as Brent [Bre1989]) to three levels:

1. **Strategy** – the fragmentation minimization philosophy of the allocator, usually attempting to exploit regularities in the request stream;
2. **Policy** – an implementable decision procedure for placing blocks in memory; and
3. **Mechanism** – a set of algorithms and data structures that implement the policy.

---

[70] This was earlier noted by Shore [Sho1977].

Using this tri-level approach Wilson et al. presented a taxonomy of allocators, followed by a short chronological review of the literature – particularly from the view of methodology used to study fragmentation. They noted, many times, the possible invalidity of previous results due to Zorn and Grunwald's observation of incongruity.

A somewhat astounding observation was that, only less than about 20 real programs had been used to test fragmentation of allocators to date. Many classes of programs, such as long-running systems, scientific computation, business data analysis, graphical programs, interactive programming environments and heavily object-oriented environments, have not been tested.

Consequently, Wilson et al. made the following comments:

> *"There is considerable confusion about the nature of memory allocators, and about the problem of memory allocation in general. Worse, this confusion is often unrecognized, and allocators are widely thought to be fairly understood. In fact, we know little more about allocators than was known twenty years ago. The literature on the subject is rather inconsistent and scattered, and considerable work appears to be done using approaches that are quite limited."*

Wilson et al. also noted that locality of reference is becomingly increasingly more important, as the difference between CPU and memory speed continues to grow.

4.4.13 Heap Profiling

Runciman and Röjemo [RR1996] discussed the concept of a *heap profiler* – a tool which generated a chart showing the contents of dynamic memory throughout a computation. Two general approaches were discussed: profiling a heap during execution and profiling the heap post-mortem using logs taken during the execution.

Runciman and Röjemo proposed a novel technique for doing both approaches by running the program twice – the first time to collect the logs and the second to watch the heap in action comparing it to future activity (garnered from the logs).

Their approach assumed deterministic computation in order to get two comparable data sets:

> *"The obvious overhead of two-pass profiling is the extra time for a repeated computation. We also need extra space to store the input to ensure identical computations in each run of the program."*

4.4.14 Heap Scalability

Extending his previous work on parallelism [Iye1996] Iyengar specifically explored the scalability (with respect to CPU) of his proposed algorithm, MLFL I, comparing it with quick fit, first fit (two variants), best fit and binary buddy.

Iyengar used the ID WORLD programming environment[71] to test the scalability of the above six algorithms. He used three different block size distributions: small blocks only, small and large blocks of even distributions, and small and large blocks of ten specific sizes.

The results were surprising. MFLF I performed substantially better than all of the other algorithms across all sizes – in some cases nearly five times better than the next best[72]. The one exception to this was quick fit for small block sizes. In this case, MLFL I and quick fit were the same.

Also of particular interest was the maximum possible throughput. Being able to vary the number of processors enabled Iyengar to demonstrate that MFLF I could scale to at least 130 processors – when said processors allocated from different areas of the heap[73].

---

[71] Which can simulate a shared memory parallel machine where the number of processors can be varied from one to infinity.

[72] What becomes obvious here is that in a parallel/SMP environment scalability is achieved by having a multiplicity of resources (such as lists) which can be scaled 1:1 with CPU's. MFLF I has five times as many searchable lists as the others.

[73] This is an important result for consideration in designing a very large, highly concurrent dynamic memory manager.

4.4.15 Vmalloc

Kiem-Phong Vo [Vo1996] introduced *Vmalloc* – a customizable allocator conforming to the malloc interface. Vmalloc introduced the idea of organizing memory into separate regions, each with a *discipline* to get raw memory from the operating system, and a *method* to manage allocation. Additionally, applications could select allocation methods using environment variables, so they could tune performance or perform other tasks, such as profiling memory usage, generating traces or debug memory errors.

Vmalloc was supplied with a general purpose allocator called VMBEST – a best fit, deferred coalescing algorithm, with free blocks kept in a top-down splay tree. In simulation tests[74], using real traces from nine different programs, VMBEST performed competitively to the best allocators tested – with respect to both time and space.

4.4.16 An Exhaustive Study

In 1997 Mark Johnstone [Joh1997] presented the results of an extraordinarily comprehensive exploration into the fragmentation and locality properties of 53 different allocators tested against eight different programs[75].

Over a number of successive iterations, Johnstone factored out overheads (such as header and footer boundary tags) and accounted for minimum alignment and object size. The objective was to get a true measure of fragmentation and locality, as a consequence of allocator policy, independent of implementation issues. He was very strict about his methodology and procedures.

---

[74] Maximum heap size was from 20KB to 6.8MB.

[75] Maximum heap size was from 1.3MB to 104MB.

In summary, the better allocators were:

RANK	ALLOCATOR	% FRAGMENTATION	% CACHE MISS RATE
1	Best Fit FIFO	2.23%	0.00910%
2	Best Fit AO	2.27%	0.00912%
3	DLmalloc v2.6.1	2.27%	0.00861%
4	Best Fit LIFO	2.30%	0.00907%
5	First Fit AO	2.30%	0.00898%
6	First Fit FIFO	3.14%	0.00954%

Table 4.1: The better allocators from Johnstone's fragmentation and locality study.

In summary, the worst allocators were:

RANK	ALLOCATOR	% FRAGMENTATION	% CACHE MISS RATE
51	Binary Buddy	53.35%	0.01043%
52	2^N & 3 x 2^N	61.50%	0.01305%
53	Simple 2^N	73.61%	0.01621%

Table 4.2: The worst allocators from Johnstone's fragmentation and locality study.

The majority of the tests requested additional virtual memory from the operating system in 4KB units. Two tests (known as First Fit AO 8K and Best Fit AO 8K) were done requesting memory in 8KB units and achieved 0.77% and 0.83 % fragmentation respectively[76].

The importance of the result is that these five policies achieve both low fragmentation and good locality – one does not have to trade fragmentation off with locality.

---

[76] However since an 8KB unit was not uniformly tested across the entire allocator suite one is inclined to marginalize the 8KB results.

Johnstone also observed phenomena already observed by others:

- Objects allocated at the same time tend to die at the same time – refer [Sho1975, Bec1982, BZ1993];
- Programs tend to allocate only a few sizes – refer [Ros1967, BJW1970, MPS1971, BB1977];
- Small policy variations can lead to large fragmentation variations – refer [Cam1971, Nei1977, Bec1982]; and
- The policies which perform best immediately coalesce memory and preferentially reallocate objects that have died recently over those that have died further in the past – refer [Bec1982]. The exception to this is First Fit FIFO – refer [Sho1975].

4.4.17 Long-Running Systems

Larson and Krishnan [LK1998] pointed out that virtually none of the research work done to that date had addressed the issues of dynamic memory management in long-running systems – systems which run indefinitely, such as mail, web and database servers.

In their paper Larson and Krishnan described their efforts in building LKmalloc – a dynamic memory allocator targeted for both traditional "one-shot" applications, as well as continuously running systems, with specific emphasis on CPU scalability.

As a reference point, they discussed the requirement of an eight processor SMP machine to be able to sustain one million allocations and one million deallocations per second, noting that larger systems should be capable of handling millions of operations per second. They also note that very little work on parallel allocators had been done to date, citing only Iyengar's work [Iye1993, Iye1996].

Apart from speed, utilization, locality and scalability, Larson and Krishnan stated four other, rather critical, requirements which do not appear to have been previously formalized.

Quoting from their paper:

1. ***Size independence*** *– speed should not be greatly affected by the size distribution and ordering of allocation requests. Given that object sizes can be very varied on server systems, some dependent on the nature of incoming requests, it is important to have size independence;*

2. ***Thread independence*** *– a block must not be tied to the thread that allocated it. A block should be free to migrate among threads, that is, it should be possible for one thread to allocate a block, a second thread to use it, and a third thread to free it (even if the original thread has already died). This makes it possible to pass objects from thread to thread in a server system. This is a strict requirement for server applications;*

3. ***Predictable speed*** *– servers are (soft) real-time applications and need to exhibit not only low but also predictable response times. This means that the time it takes to allocate or free a block should, ideally, be constant and independent of its size, amount of memory in use, allocation history, etc. In particular, occasional long pauses for garbage collection or large-scale coalescing are unacceptable;*

4. ***Stability*** *– for long running systems it is very important that the memory allocator's performance remains stable over time. In other words, memory utilization should not decrease or allocation times increase over time if the load on the system remains stable.*

Larson and Krishnan claimed long-running systems pose different requirements (notably the four above) than traditional single shot applications, and these requirements have not been addressed[77].

[77] It is a tenet of this thesis that not only have these not been addressed, they have not been addressed for very large heaps at all.

In designing LKmalloc they adopted three key features from DLmalloc [Lea2000]:

1. **Binning** – a combination of 64 quick lists for small blocks and 64 *misc* lists for larger blocks[78]; Unlike DLmalloc the *misc* lists are not kept in size sorted order[79];

2. **Approximate best fit** – the effect of the above binning is that fitness is a combination of best fit and first fit[80]; and

3. **Immediately coalescing** – reduces fragmentation, improves memory utilization, avoids postponing work, providing more predictable speed.

LKmalloc used a number of sub-heaps determined *a priori* – each of which could grow to the full extent of memory. A thread was assigned a specific sub-heap, based on a hash of its thread id[81].

A series of experiments were used to test the scalability of LKmalloc against other allocators, such as DLmalloc and PTmalloc. On uniprocessor systems, LKmalloc was about 25% slower than the fastest. On four and eight CPU machines, LKmalloc scaled better than the others[82] (some tests showing negative scalability).

Interesting LKmalloc exhibited very poor memory utilization – between 10 and 50% – due to the fact that the multiple sub-heaps were not shareable between threads.

---

[78] Refer to [Iye1993] for good description of quick lists versus *misc* lists.

[79] This concurs with the observations by [GZH1993].

[80] This concurs with the efficiency and effectiveness of quick lists or sub-pooling at least as far back as [MPS1971].

[81] This is a possible weakness in the model. Should the size of one sub-heap outstrip the size of other sub-heaps, the threads of the smaller heaps may falsely detect exhaustion. Space is available in other sub-heaps but threads can only ever allocate from one sub-heap.

[82] Again confirming scalability in an SMP environment requires a multiplicity of lockable resources which scale 1:1 with CPU's.

4.4.18 Fragmentation Solved?

Summarizing and reevaluating their previous work [WJNB1995, Joh1997], Johnstone and Wilson [JW1998] put forward the proposition that, with respect to fragmentation, good allocation policies already existed at that time; efficiency of implementation was now the major challenge.

4.4.19 Scalable SMP Allocator

Vee and Hsu [VH1999] presented an experiment with a CPU scalable and efficient allocator (VHmalloc[83]), designed for time-critical parallel programs. VHmalloc achieved its properties from two qualities: firstly, it allocated fixed sized blocks, and secondly, it had very low lock contention. Vee and Hsu specifically noted that there are cases where sizes are known *a priori*, and high performance and scalability are critical.

VHmalloc had a local pool per thread, with a set of blocks available for allocation. There was also a global pool of blocks. Threads allocated and freed from their local pool without locking; threads transferred blocks to or from the global pool, guarded by a lock, in order to top up or empty out their local pool as needed.

Vee and Hsu showed, both theoretically and empirically, that their allocator incurred very low lock contention. Simulation experiments[84] showed VHmalloc had very fast allocation and release rates, and scaled near linearly to four CPU's.

From the point of view of hardware evolution, Vee and Hsu made the following observation:

> *"A 16-processor machine nowadays typically has at least a few gigabytes of memory. A payoff of only 256kbytes for a better performance on such machines is definitely worthwhile."*

Note that 256KB represents only 0.02% of a gigabyte.

---

[83] Although Vee and Hsu did not fomally name their allocator, we refer to it as *VHmalloc*.

[84] On a four processor 250MHz UltraSPARC Sun Enterprise 3000.

4.4.20 PTmalloc

Given DLmalloc had good all-round performance but was single threaded, Wolfram Gloger took the novel step and built a "synthetic allocator" called *PTmalloc* [Glo2002]. Behind its API, PTmalloc gangs together multiple copies of DLmalloc[85], to present a multithreaded allocator. PTmalloc manages the locking of an instance of DLmalloc, and creates a new instance of DLmalloc, when all known instances are currently locked.

At first blush this sounds an elegant solution to concurrency. There are simple pathological cases though, which cause blowup (premature memory exhaustion) [BMBW2000]. This is largely due to relatively simple interference patterns which occur at this granularity, effectively locking what are entire separate heaps. Not withstanding this, PTmalloc is a highly effective multi-threaded allocator and is in current popular use.

4.5 The 2000's

4.5.1 Hoard

Berger et al. [BMBW2000] observed that even though dynamic storage allocation is one of the most studied topics in computer science, very little work has been done on concurrent memory allocators to date. They presented a concurrent CPU scalable allocator called *Hoard*. Their motivation was the recognition that memory allocation can be a bottleneck for parallel, multi-threaded applications.

Hoard was similar to the arrangement in VHmalloc [VH1999] in that it utilized one global heap and one (or more) local heaps per CPU. Unlike VHmalloc (which transferred individual blocks to and from the global heap), Hoard transferred superblocks.

In Hoard a superblock is a region of virtual memory containing an array of allocatable blocks, and contains a free list of available blocks maintained in LIFO order to improve locality. Like Haertel's allocator [Vo1996], within Hoard, the superblocks always contain blocks of the same size.

---

[85] PTmalloc calls a single instance of DLmalloc an *arena*.

One of the objectives of Hoard was to limit the degree of *blowup*. Blowup is the phenomena where an allocator fails to be able to reallocate previously freed blocks, consequently consuming an ever increasing amount of memory over time, ultimately leading to memory exhaustion. Blowup is a subtle form of design defect for allocators which use multiple sub-heaps[86], particularly occurring when sub-heaps are statically assigned to one thread or processor, or are not circulated in a balanced fashion between threads or processors.

Hoard bounded blowup, by firstly, moving superblocks from a local heap back to the global heap when a superblock crosses an *emptiness threshold* – calculated on the basis of the number of free blocks in the superblock relative to the amount allocated from the local heap. Secondly, Hoard segregated the global heap into bins based on fullness and preferentially selects fuller superblocks when moving superblocks from the global heap back into a local heap.

Additionally Hoard showed that, while false sharing cannot be eliminated, it can be minimized. The false sharing phenomenon, where multiple processors share words in the same cache line without actually sharing data, can lead to cache thrashing. False sharing can be minimized by allocating (as best possible) non-overlapping memory intervals to distinct threads.

Tests[87] against 11 different programs showed that Hoard scaled near linearly to 14 CPU's. However, as Berger et al. pointed out, Hoard achieved low synchronization costs and fragmentation, but traded memory consumption to do so. For the 11 programs tested, Hoard generally achieved low-levels of fragmentation. The exception was one program, for which Hoard used three times as much memory, as required by the program.

---

[86] PTmalloc is known to suffer from this problem. As pointed out, this may also occur in LKmalloc.

[87] Run on a 14-processor Sun Enterprise 5000 with 2GB of memory, 400MHz UltaSparc CPU's with 4MB of level 2 cache. Maximum heap size range from 400KB to 614MB.

Hoard appeared to suffer higher fragmentation when there was a large range of infrequently used sizes – a superblock gets created for each individual size[88]. An example of a pathological case would be a single block of each possible size, leading to large amounts of wasted space in each superblock.

4.5.2 Avon

Seidl [Sei2000] discussed a memory allocation system called *Avon*. The objective of Avon was to improve execution performance, by identifying and capitalizing on the skew in reference density between objects – a small percentage of the objects in a system receive a large percentage of the references. Seidl was able to show that references were non-uniform, and consequently, were predictable and exploitable.

The Avon system was a profile driven optimizer. It analyzed program runtime behavior in order to predict object reference behavior. This information was then used to alter the program, so as to segregate objects into separate heap areas, based on their reference behavior and lifetime.

Objects were segregated into four types of memory region: highly referenced, short-lived, not highly referenced and other. This can be seen as an extension of the work by Zorn and Grunwald [BZ1993], which identified short-lived objects in order to improve execution.

Seidl discovered that object segregation can result in a reduction of virtual memory paging activity, and data TLB miss rates:

> *"The combination of these two affects improves performance and reduces execution time in applications that use a large amount of dynamic memory."*

Seidl used two cache simulators (TYCHO [GZH1993] and a custom simulator) to simulate the cache of a Dec Alpha 21164 processor. Tests with seven different

---

[88] Hoard does not appear to uphold Larson and Krishnan's requirements for size, thread and speed independence. Refer §4.4.17.

programs[89] showed that, increasing the reference locality of dynamically allocated objects, could improve the execution performance between 10 and 50%.

4.5.3 Debugging Tools

In trying to understand memory allocation behavior in the functional programming language SCHEME, Serrano and Boehm [SB2000] developed two tools. The first, KPROF, was a profiler which reported the number of times allocations were made from each function. It could report on such items as heap size and number of live objects, as well as browse the call stack. The second, KBDB, was an interactive heap inspector. It displayed the live objects, and the chains of pointers that linked those objects in the heap, and was primarily used to fix memory leaks.

Printezis and Jones [PJ2002] presented GCSPY – an adaptable heap visualization framework. As a framework, the GCspy architecture allowed it to be incorporated into any dynamic memory management system. Fundamentally, GCspy provided for the collection, transmission, storage and replay of dynamic memory usage.

4.5.4 A CPU Scalable Lock-Free Allocator

Michael [Mic2004] described his work on a dynamic memory manager similar to Hoard [Ber2002], except that it used atomic memory operations to avoid using mutual exclusion locking techniques (such as spin locks). Experiments on a 16 processor SMP machine[90] showed near linear CPU scalability, with performance throughput better than the other allocators tested (Hoard, PTmalloc and libc).

There was no mention of the size of the heaps being tested.

---

[89] Maximum heap size of 86MB.

[90] Sixteen 375MHz POWER3-II processors, with 24GB of memory, 4MB second level caches.

4.6 A Reflection on the Literature

The history of DMM is certainly colorful. As we can see, DMM was one of the grand challenges in the history of computing, being a key driving force behind the evolution of machines, operating systems, data structures, algorithms and the very concept of the program itself. Consequently, the nomenclature, perspectives and orientations changed considerably over the formative years leading to the general sub-system we know today.

Yet most of the work on DMM was conducted prior to the 1990's. Consequently, the bulk of DMM research occurred before multi-threading emerged in operating systems on SMP architectures. In fact, most of that work was done in view or consideration of one-shot programs (such as compilers) on comparatively limited machines. Very little work on CPU scalability has been done. No study of DMM within a continuously executing multithreaded system has been published to date, nor any study with respect to memory scalability.

Over the last fifty years hundreds, possibly thousands of allocators have been written, with relatively few surviving the test of time. In the next chapter we examine the state of the art in DMM, inspecting and comparing four allocators which are currently in widespread use.

5 State of the Art

Having reviewed the history of DMM we now move on to examine the state of the art. In this chapter we inspect in detail a small number of very different, but highly successful allocators and take a clinical look at how they work.

Our objective is to understand if, and consequently why, these allocators scale with respect to both CPU and memory. Ultimately though, we show that no allocator has been specifically designed to scale with respect to both CPU and memory.

5.1 Introduction

One of the difficulties in discussing and comparing dynamic memory managers is the variation in nomenclature, adopted by the respective designers. For example, some allocators are described as managing memory resources using a single heap, others multiple heaps. Some allocators describe their multiple heaps as a set of sub-heaps, others generically as a set of heaps. Alternatively one might describe a heap as a two or three level hierarchy of sub-heaps, or simply an indefinite recursion of sub-heaps.

For the sake of clarity, we chose to use the following definitions and attempt to apply them consistently. We basically define a *heap* to be the collection of memory resources managed by one instance of an allocator. A heap may be recursively composed of a set of *sub-heaps*, where each sub-heap is either a set of sub-heaps or a set of memory resources. In the case where we are describing heaps in a hierarchical fashion, we use the term heap to refer to the top most tier, a sub-heap as the second most tier and *micro-heap* as the third tier – in practice, we do not need more than three levels to describe the current state of the art.

5.2 Representative Set

Our approach is to examine in detail a small number of very different but highly successful allocators – allocators which have attained industrial grade reliability and are in broad usage. Our objective is to understand how those allocators work and how they scale with respect to both CPU and memory.

The allocators we have chosen are: DLmalloc, PTmalloc, LKmalloc and Hoard.

We begin with DLmalloc because it is generally recognized as one of the best general purpose allocators built to date, and has antecedents stemming back to 1987.

We chose PTmalloc because it is a multithreaded version of DLmalloc. We chose LKmalloc because it (at least from our perspective) is a multithreaded variation of the techniques used by DLmalloc.

We chose to examine Hoard because it represents the most recent and complete work in concurrent dynamic memory management, focusing particularly on scalability with respect to CPU. Hoard is also of architectural interest, in that it takes a completely different approach compared to DLmalloc, PTmalloc and LKmalloc.

5.3 Berger's Taxonomy

Berger et al. [BMBW2000] described a taxonomy of DMM algorithms which is a very useful tool for examining the modern set of allocators. Berger's taxonomy consists of the following five categories:

1. **Serial Single Heap** – there is one heap of memory resources and only one thread may perform any given operation within that one heap at any one point in time. The single heap is protected by a single lock.
   This approach serializes operations leading to contention. Consequently, this class exhibits very poor scalability with respect to CPU;

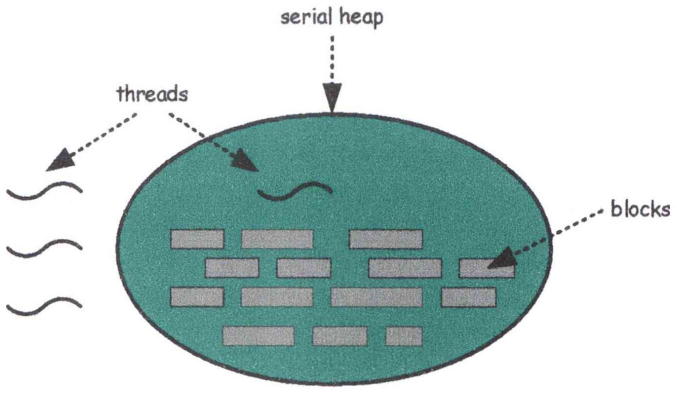


Figure 5.1: Serial Single Heap. Figure shows one thread accessing the heap, with three other threads waiting.

2. **Concurrent Single Heap** – there is one heap of memory resources, but multiple threads may be performing various operations within that single heap at the same time. The single heap is typically implemented as a concurrent data structure (such as concurrent B-tree or a vector of lists with a lock per list). This approach can often degenerate into serialized behavior when requests are mainly for a small number of sizes;

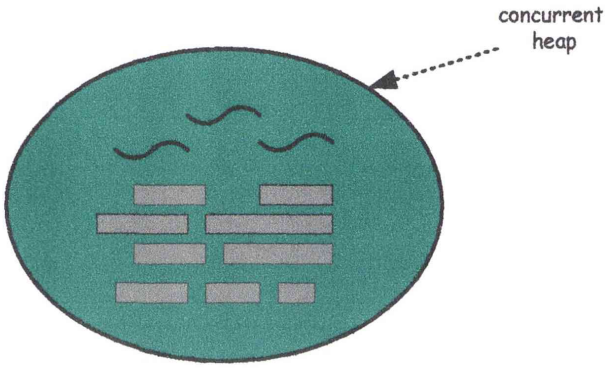


Figure 5.2: Concurrent Single Heap. Figure shows three threads which are concurrently accessing the same heap.

3. **Pure Private Sub-Heaps** – each thread has its own sub-heap. A block of memory which is freed is put into to the sub-heap associated with the freeing thread. This may or may not be the same thread as the thread which allocated the block, so the memory block may be returned to a different sub-heap than the one from which it was allocated. Each sub-heap is considered private, in the sense that each thread never accesses any other sub-heap other than its own for any operation. Implementations can be made very fast and scalable with respect to CPU as no concurrency mechanisms (such as locks) are required.
This approach readily leads to situations (blowup) where, memory resources are available in one heap, but cannot be accessed by any thread other than the thread which owns the heap;

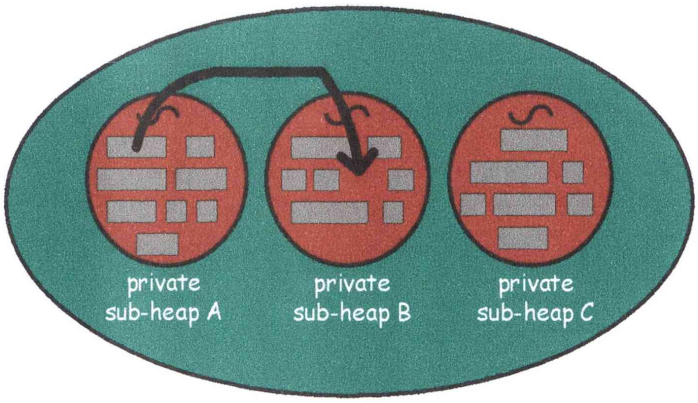


Figure 5.3: Pure Private Sub-Heaps. Where a thread owning private sub-heap A, allocates a block which is then freed by a thread owning private sub-heap B. Block is returned to private sub-heap B.

4. **Private Sub-Heaps with Ownership** – each thread has its own sub-heap but memory blocks are always returned to the original sub-heap when freed. This approach can also readily lead to situations (blowup) where, memory resources are available in one sub-heap, but cannot be accessed by any thread other than the thread which owns the sub-heap; and

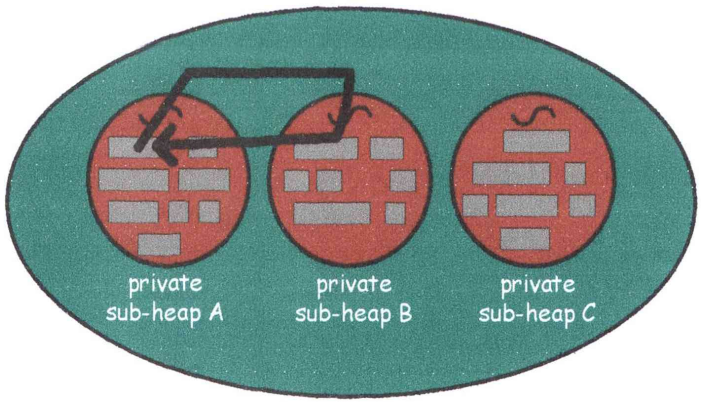


Figure 5.4: Private Sub-Heaps with Ownership. Where the memory block allocated by a thread of private sub-heap A and then freed by a thread of sub-heap B is returned to the original sub-heap i.e. private sub-heap A.

5. **Private Sub-Heaps with Thresholds** – each thread has its own sub-heap which can hold a limited amount of free memory. Typically a sub-heap is composed of a number of micro-heaps which can be moved between sub-heaps. This enables excess free memory to be returned to a common sub-heap.

The objective of this approach is to recycle memory resources, in order to enable those resources to be accessible by multiple threads.

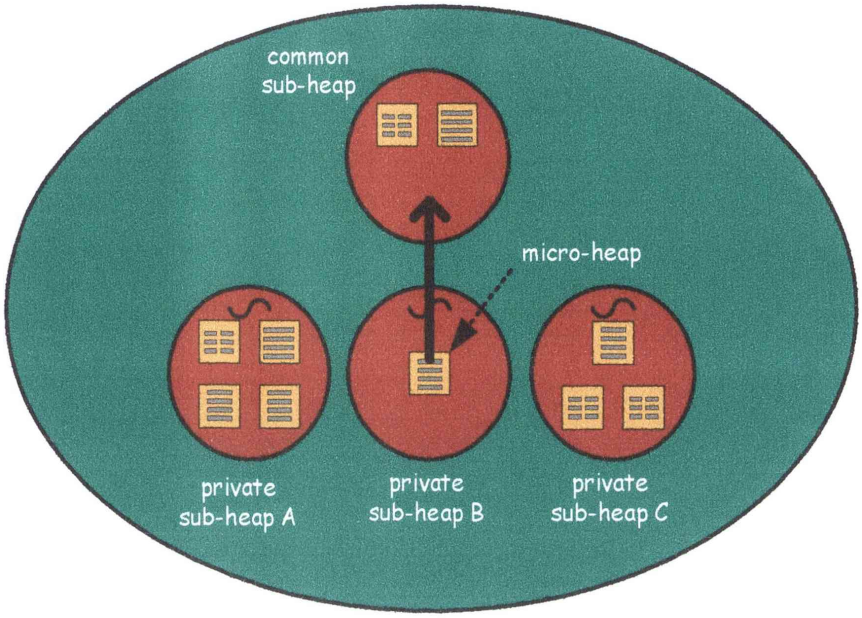


Figure 5.5: Private Sub-Heaps with Thresholds. Figure shows a micro-heap being shifted from private sub-heap B into the common sub-heap.

5.4 Dissecting DLmalloc

The core of DLmalloc utilizes vectors to keep track of a set of free block lists. DLmalloc refers to this technique as *binning* and it is similar to the quick fit technique. Doug Lea himself notes [Lea2000] that this technique is in fact an implementation of a best fit policy[91].

DLmalloc uses a vector with 64 elements to track free blocks which are in the range 0 – 63 units in size. In this arrangement, each element of the vector is a pointer to a list of free blocks of a particular size, where that size directly corresponds to its index in the vector – element zero points to a list of free blocks where each block in that specific list is zero units in size[92], element one points to a list of free blocks each of one unit, and so on up to 63 units.

---

[91] Refer to Johnstone's results in §4.4.16.

[92] Note that it is considered legitimate in the ANSI C/C++ malloc interface definition to request a memory block of zero bytes in length.

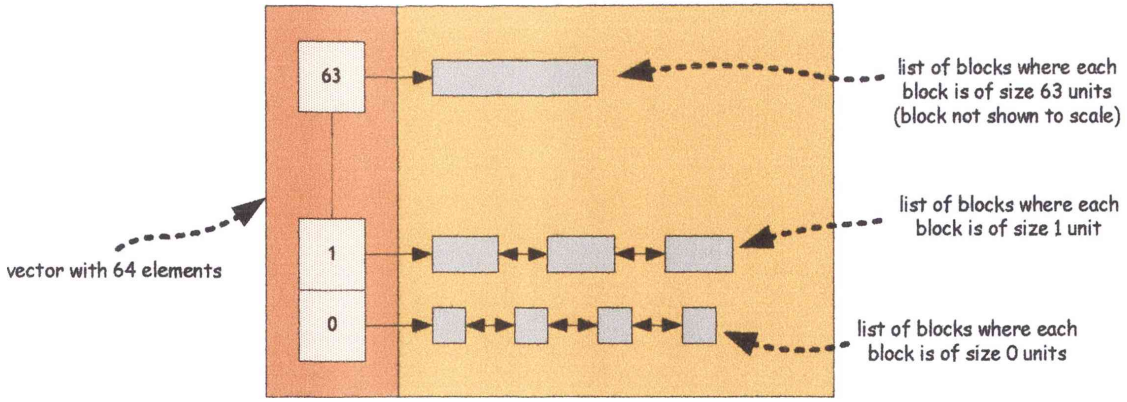


Figure 5.6: Depicting the exact bins in DLmalloc as a vector of lists.

According to Berger's taxonomy, DLmalloc is a serial single heap.

Because of the one-to-one correspondence between element index and block size, DLmalloc refers to this kind of grouping as *exact bins* – each list is conceptualized as a holding bin, which have free blocks all of the same size. Allocation of blocks in this range (depending on the state of memory at the time) can be as simple as using the requested size to inspect the corresponding sized bin and removing the first block found. Inversely freeing a block can be as simple as pushing the block into the bin based directly on its size.

Consequently, access to blocks within the smaller exact size range in DLmalloc are O(1).

The exact bins technique is very suitable in practice. Many studies have shown that programs typically allocate blocks of the sizes covered by the exact bins.

5.4.1 Larger Sizes

We observed that using a vector of free lists is a highly effective technique for managing a small range of sizes. The question arises on how DLmalloc organizes and manages free blocks which are larger, and cover a much broader range, than the exact bin sizes.

The approach that DLmalloc takes is that it uses another vector of 64 elements to manage another set of 64 lists, such that each list holds a range or band of sizes, rather than a single size. DLmalloc refers to this set of lists as *sorted bins*.

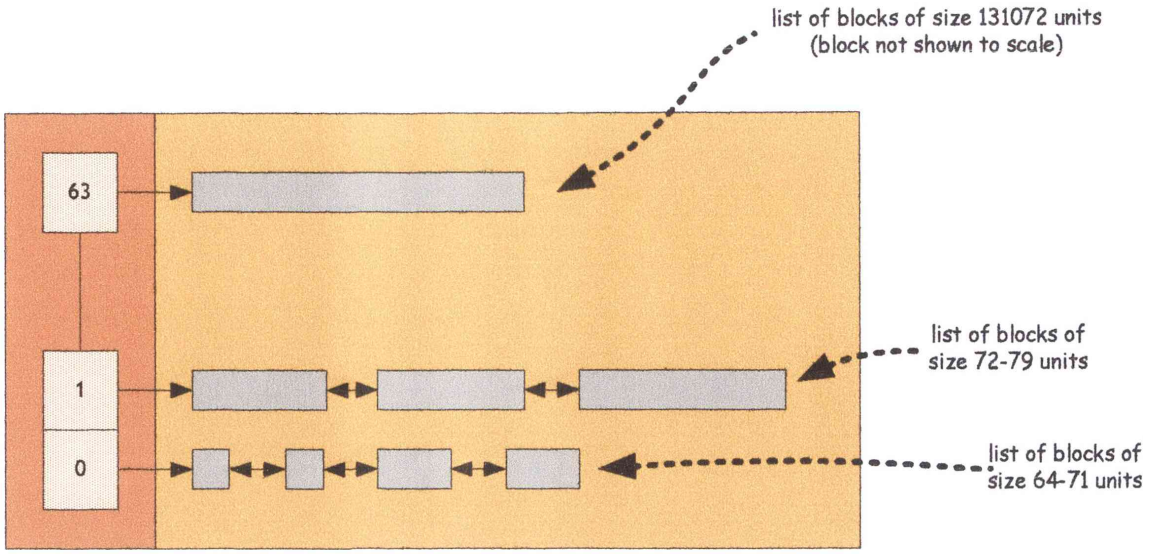


Figure 5.7: Depicting the sorted bins in DLmalloc as a vector of lists.

In DLmalloc the blocks within these lists are maintained in order sorted by the size of the block[93] (hence the name sorted bins) using a very simple algorithm: for allocation the list is searched one block at a time, starting at the head, until the first suitably sized block is found; and for freeing, the list is inspected one block at a time, starting at the head, until the appropriate position for insertion based on size is found.

Thus access to blocks within the larger, sorted size range in DLmalloc are O(n).

DLmalloc chooses to maintain these larger size lists in sorted order as a time-space optimization. DLmalloc attempts to minimize fragmentation by using vectors of free lists to implement a best fit policy. By keeping the lists in size sorted order, DLmalloc can also (in certain cases) limit searches during allocation – when scanning the blocks in the list, the first block found will be the best size.

In DLmalloc the sizes for the sorted bins are (approximately) logarithmically spaced[94]. So on average, each sorted bin covers a range of about 512 sizes. DLmalloc is capable of using sorted bins to manage sizes up to 1MB. By default it manages sizes up 128KB.

---

[93] This was not the case in earlier versions of DLmalloc.

[94] In DLmalloc version 2.7.2 the first 32 sorted bins are spread 64 bytes apart, the next 16 are spread 512 bytes apart, the next 8 are spread 4,096 bytes apart, the next 4 are spread 32,768 bytes apart, the next 2 are spread 262,144 bytes apart. The final bin contains anything else.

Sizes above these limits, are expected to be managed by facilities in the virtual memory management system of the underlying operating system[95].

In both the exact and sorted cases, the lists within the bins are kept in FIFO order [Lea2000]:

> *"Chunks of the same size are linked with the most recently freed at the front, and allocations are taken from the back. This results in LRU (FIFO) allocation order, which tends to give each chunk an equal opportunity to be consolidated with adjacent freed chunks, resulting in larger free chunks and less fragmentation."*

5.4.2 Deferred Coalescing

It is interesting to note that, due to coalescing, allocating and freeing blocks in the exact size range does not always yield O(1) performance. The progressive freeing and coalescing of blocks, when adjacent to other free blocks, causes larger free blocks to be formed such that they "percolate up" through the sorted bins – incurring O(n) service times at those points.

As an optimization Lea argues for deferred coalescing in order to maximize CPU utilization[96]. DLmalloc uses a technique it calls *fast bins*, which are a duplicate set of bins mirroring the normal bins described above.

---

[95] Such as the mmap() function available with many Unix kernels or the VirtualAlloc() function in Microsoft Windows NT.

[96] It is interesting to note that Larson and Krishnan [LK1998] argue against this, claiming the reduction in CPU usage in practice is minimal and the added code complexity not worth the effort.

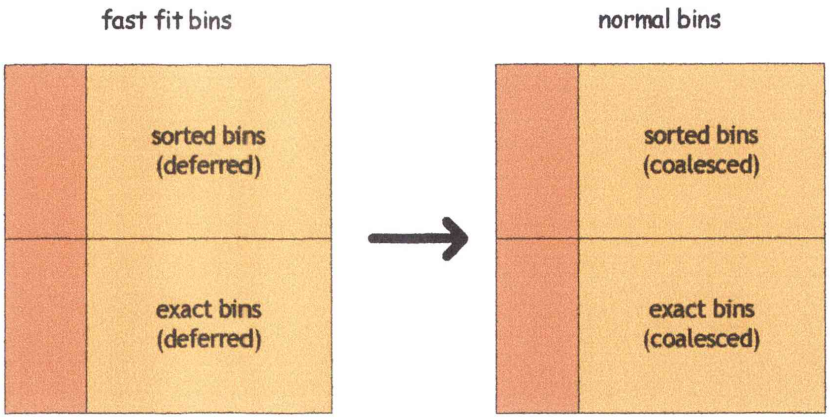


Figure 5.8: Fast bins are deferred coalescing bins mirroring the normal bins.

When blocks are in the fast bins they are not eligible for coalescing. The fast bins are singly linked lists, kept in LIFO order to maximize cache locality and minimize CPU utilization (singly linked lists are slightly faster than doubly linked lists). Blocks are allocated from, and returned to, the fast bins first without coalescing. When an allocation request cannot be satisfied from the fast bins, the fast bins are emptied into the normal bins (being coalesced whenever possible) and the request is then satisfied from the normal bins.

5.4.3 CPU Scalability

DLmalloc is single threaded – the allocator is guarded by a single lock. While any thread is allocating or freeing, it locks and therefore bars, any other thread from the heap. Fundamentally this means DLmalloc does not scale with respect to CPU. In an SMP environment, as we add CPU's into the model, one and only one thread can be using the allocator at any given point in time.

While there are no specific reasons given for the single threaded nature of DLmalloc, one could surmise it is due to its history. DLmalloc was first created in 1987, and was not originally designed as a multithreaded library.

5.4.4 Memory Scalability

DLmalloc is not scalable with respect to memory either. In particular, the sorted bins can contain an indefinite number of blocks. Searching sorted bins is unbounded. An example of a pathological case is the circumstance where DLmalloc searches an entire

list. This happens when the size being searched for is not in the list, and the size is larger than all the other sizes in the list.

Consequently, as memory tends to infinity, sorted bins tend to infinity, and so would service times.

5.5 Dissecting PTmalloc

There exists a variation of DLmalloc, called PTmalloc, which achieves a higher degree of concurrency – essentially by replicating DLmalloc entirely. PTmalloc is effectively a list of DLmalloc instances. Each instance is guarded by a lock, which allows only one thread access to any given instance, at any point in time.

PTmalloc refers to an instance of DLmalloc as an *arena*.

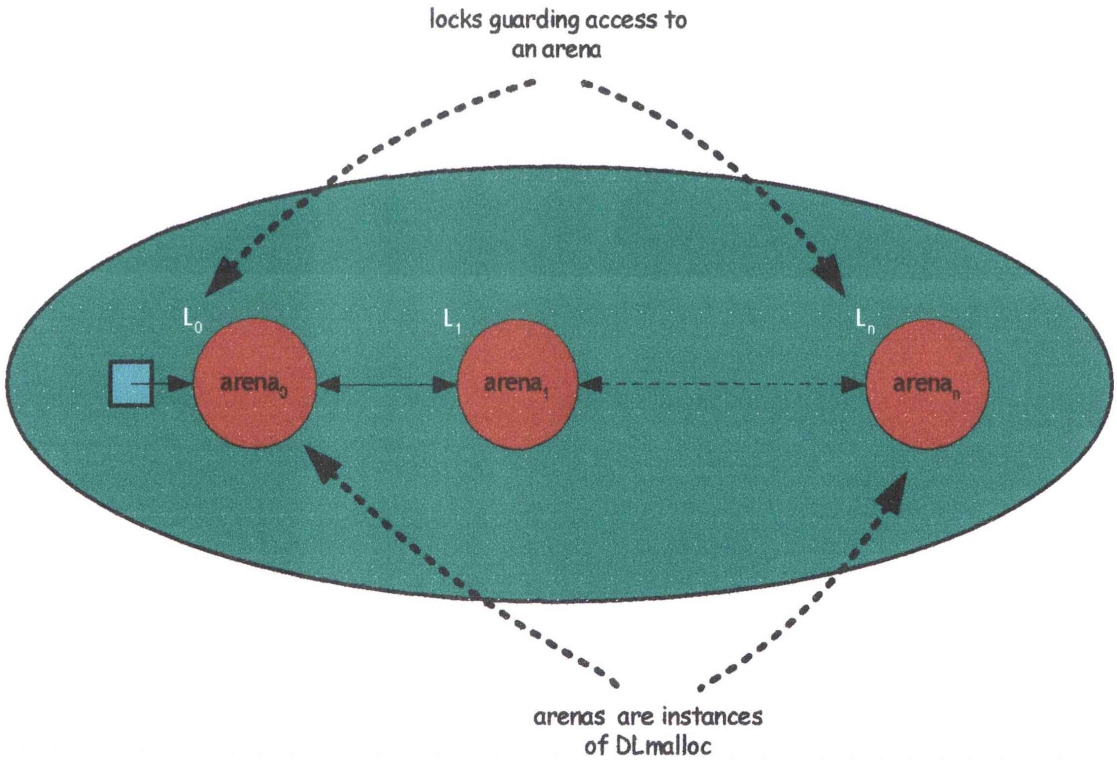


Figure 5.9: PTmalloc is a list of guarded DLmalloc instances.

According to Berger's taxonomy, PTmalloc is classified as private sub-heaps with ownership.

5.5.1 CPU Scalability

When a thread requests an allocation of dynamic memory, PTmalloc chooses an unlocked arena by scanning the list of arenas, starting each time from the head of the list. Restating: each and every allocation request requires scanning the list of arenas to find an unlocked instance of DLmalloc, and then executing the allocation procedure of DLmalloc as described above. Tests by Berger et al. [BMBW2000] show that this heap selection strategy generates substantial bus traffic, which limits PTmalloc's scalability to about six CPU's.

PTmalloc creates a new arena according to some very simple rules based on detecting thread collisions. While PTmalloc achieves a *higher* degree of concurrency (in that more than one thread can be allocating and freeing at the same time within PTmalloc) PTmalloc does not achieve a *finer* degree of concurrency (in that two or more threads can not be accessing any one arena simultaneously). Concurrency within PTmalloc could be described as *coarse or large grained.*

5.5.2 Memory Scalability

PTmalloc is not scalable with respect to memory. An arena is locked by a thread when it is allocating or freeing, and new arenas are created when all arenas are locked at the point when they are tested. Consequently, this can lead to simple pathological cases where allocating threads collide with freeing threads, causing new arenas to be created unnecessarily, thereby leading to blowup – premature memory exhaustion.

5.6 Dissecting LKmalloc

LKmalloc is a derivative work of DLmalloc [LK1998]. LKmalloc uses multiple sub-heaps, where each sub-heap has the same vector arrangement of 128 bins as per DLmalloc. In LKmalloc a sub-heap is, first and foremost, an extensible region of memory: any block which is allocated from a sub-heap will be returned to that same specific heap when freed[97].

---

[97] Compare this to a sub-heap which is first and foremost a collection of blocks. Sub-heaps which are block collections enable blocks to be returned to a different sub-heap from where they were allocated.

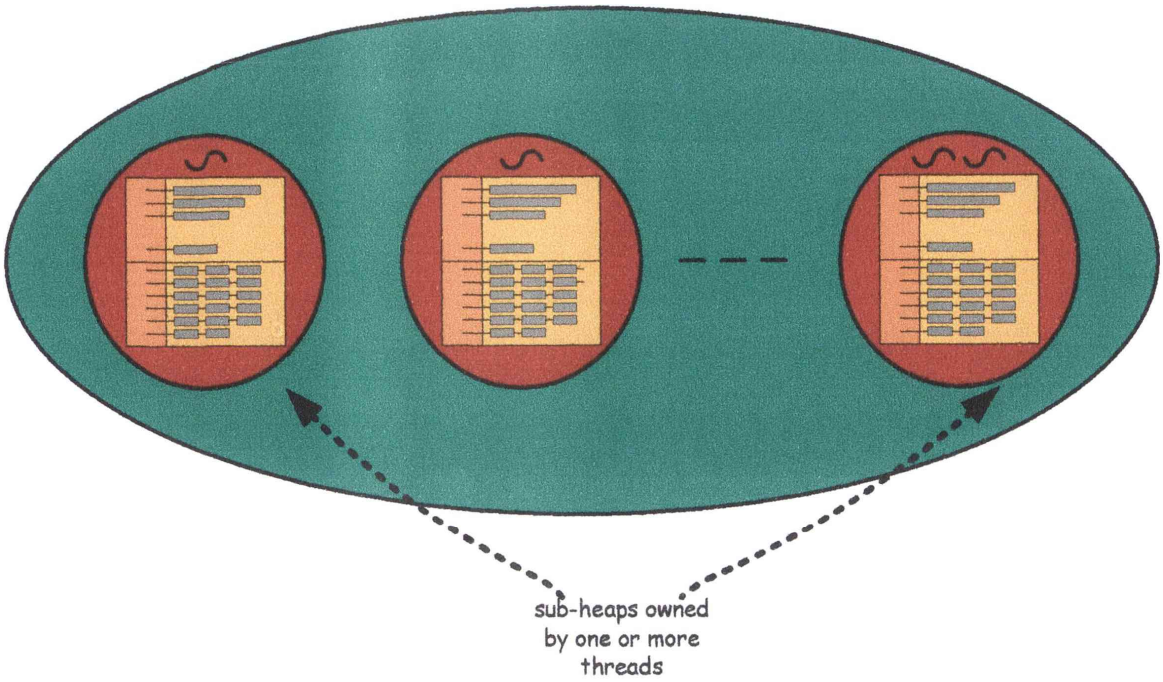


Figure 5.10: LKmalloc is a set of private sub-heaps with ownership.

According to Berger's taxonomy, LKmalloc is classified as private sub-heaps with ownership.

5.6.1 CPU Scalability

The number of sub-heaps is determined at library initialization time. LKmalloc attempts to create a sub-heap per thread based on the number of CPU's in the machine. This is not an accurate technique, resulting at times in more than one thread using the same sub-heap, or excess sub-heaps being created. In LKmalloc, each list is guarded by a lock. So in practice, more than one thread can be simultaneously allocating from the same sub-heap, providing the threads are allocating from different lists.

LKmalloc was shown to have good scalability. On tests up to eight CPU's LKmalloc exhibited up to 5.3 times the performance of a uniprocessor machine.

We later used this observation to derive another taxonomy catering for all possible allocators. Refer to Appendix D.

5.6.2 Memory Scalability

Because a thread is permanently assigned to a sub-heap, the memory blocks which are available in one sub-heap are not available to other threads assigned to other sub-heaps. This gives rise to a simple pathological case which leads to premature memory exhaustion: one thread could allocate all of memory and then free it, upon which it ends up in one sub-heap, thereby making it unavailable to other threads.

It is this behavior, where resources are not recycled amongst consumers, that leads to the case of *blowup,* as described by Berger et al. [BMBW2000]. Quoting Larson and Krishnan [LK1998]:

> *"LKmalloc's memory utilization is definitely too low when multiple sub-heaps are used. For ten-sub-heaps and five or more concurrent threads, it needed almost twice as much memory as the most efficient allocators tested. We are looking into ways of automatically adjusting the number of sub-heaps based on the level of concurrency exhibited by the application. The problem is how to improve memory utilization without loosing too much in scalability."*

5.7 Dissecting Hoard

The primary goal of Hoard is CPU scalability. At its core, Hoard is fundamentally different from the DLmalloc family described above. While Hoard views a heap as a set of sub-heaps, a sub-heap in the Hoard model is not a single entity. Rather Hoard's memory orientation is that it first divides memory into regions called *superblocks* – where a superblock is a multiple of a virtual memory page[98].

In Hoard a sub-heap is a set of superblocks.

Each superblock is an array of some number of blocks. All of the blocks in any given superblock are the same size. So by definition, if a program requests ten different sizes of blocks, there will be at least ten different superblocks within that instance of Hoard. Each superblock contains a free list of its available blocks maintained in LIFO order to

[98] By default, a Hoard superblock is 8KB.

improve locality. Thus, one might consider a superblock to be a homogenous *micro-heap*.

In order to achieve a high degree of concurrency, Hoard arranges sub-heaps in the following manner: there is one global sub-heap and (at least) one sub-heap per CPU. Unlike LKmalloc, Hoard can shift resources between sub-heaps. Should a CPU sub-heap become exhausted, then a superblock can be shifted from the global sub-heap (if it is available) into the CPU sub-heap. Allocation can then continue. Note that allocation of blocks never occurs directly out of the global sub-heap. The global-sub-heap is an exchange mechanism by which memory resources can be shifted (in a two stage fashion) between CPU sub-heaps.

Additionally, Hoard moves superblocks out of CPU sub-heaps, and back into the global sub-heap, when superblocks become relatively empty. Hoard does this in order to bound blowup – the excessive use of resources caused by the inability to recycle resources.

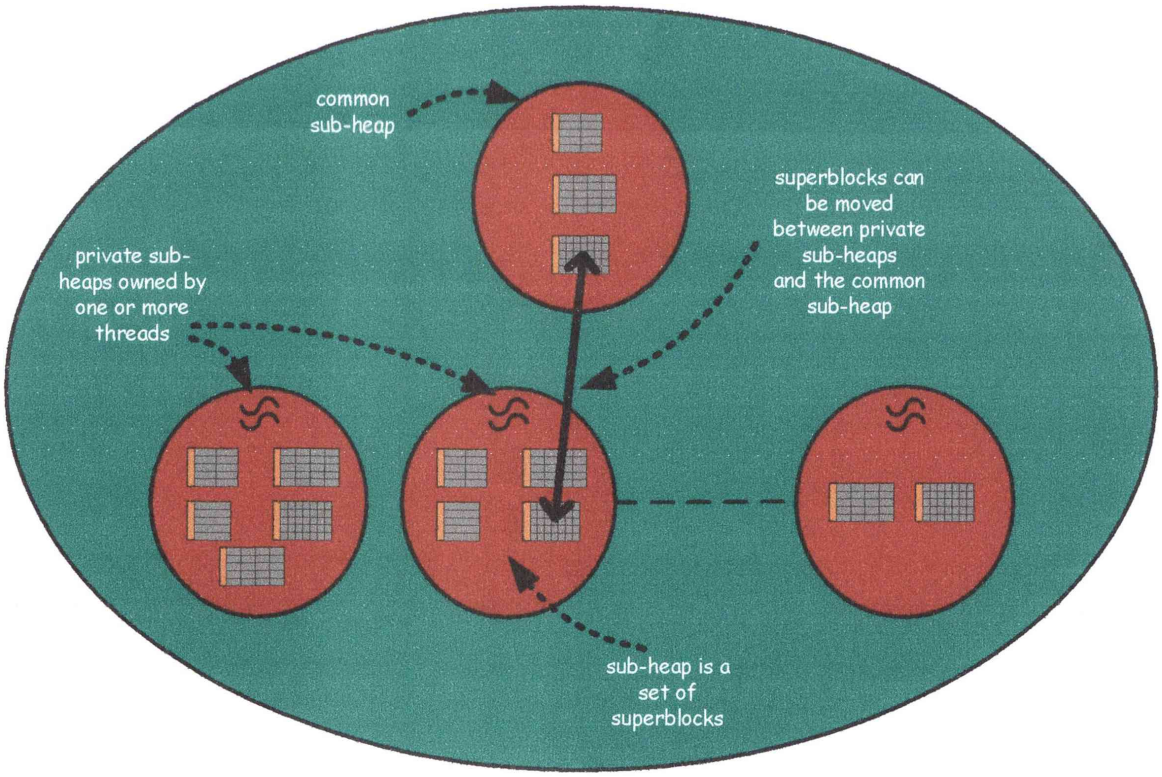


Figure 5.11: Hoard is as a set of private sub-heaps consisting of a set of superblocks. Superblocks can move between private sub-heaps and a common sub-heap.

According to Berger's taxonomy, Hoard is classified as private sub-heaps with thresholds.

5.7.1 CPU Scalability

In order to manage concurrency, Hoard locks an entire superblock when a block is allocated or freed. Hoard needs to do this as more than one thread could be allocating or freeing at similar times (given context switching etc.). The time, however, for which a lock is held, is very short: a superblock is effectively a stack of blocks. So allocating and freeing are simple pop and push actions. Hoard locks the entire global sub-heap while shifting superblocks in and out of it.

Berger et al. [BMBW2000] showed that this concurrency strategy leads to high degree of throughput and scales particularly well with respect to CPU. In their tests they used two sub-heaps per CPU to minimize lock contention. In a number of cases Hoard achieved close to linear scalability on a 14 CPU machine.

5.7.2 Memory Scalability

Hoard is designed to be highly concurrent, trading memory utilization to achieve that goal. Hoard performs best, and uses appropriate amounts of memory, when the calling application allocates and frees a large number of blocks evenly distributed over a small number of sizes.

There are cases where Hoard uses an excessive amount of memory. This is due to the homogenous nature of superblocks. Because a superblock can only service one specific size, there are pathological cases of the kind where an application allocates a small number of blocks over large number of sizes – at least one superblock must be created per size – possibly more, depending on the state of the sub-heaps at the time the request are made. Quoting Berger et al.:

> *"We are also looking at ways to remove the one size class per superblock restriction. This restriction is responsible for increased fragmentation and a decline in performance for programs which allocate objects from a wide range of size classes..."*

5.8 Summary

As shown in our literature review, there clearly has been an enormous amount of work which has been done over the years in relation to dynamic memory management. We must, however, concur with Berger et al. [BMBW2000] that there has been very little work on concurrent scalable allocators.

In particular, we observe that not only has there been little work on scalability, the term *scalability* is almost invariably used synonymously with CPU scalability. Memory scalability is never mentioned as an issue in its own right, relegated as it were, to simply being an issue of memory utilization – a function of memory fragmentation. This is certainly reflected in allocators such as DLmalloc, PTmalloc, LKmalloc and Hoard which are representative of the state of the art.

It is our conclusion therefore, that no work (certainly none that we can find in the last 50 odd years) has been done on very large, highly concurrent dynamic memory management.

In the next chapter we state our requirements for a very large, highly concurrent dynamic memory manager. We define the terms very large and highly concurrent, going on to argue that while an allocator achieves its objectives it should exhibit efficient and predictable, linearly scalable, O(1) time-space behavior.

Chapter 6

6 Technical Requirements

Having both reviewed a representative cross section of the literature, and inspected a representative set of modern allocators, we have observed that no allocator has been specifically designed and built to address scalability with respect to both CPU and memory.

All things being equal[99], an allocator, which was linearly scalable with respect to CPU, would achieve twice the allocation and freeing rate if the number of CPU's in the machine were doubled. An allocator, which was linearly scalable with respect to memory, would maintain the same allocation and freeing rate when the amount of memory was doubled, but would support a heap twice the size.

In this chapter we specify our formal requirements for such an allocator. This statement of requirements is presented as a prelude to the upcoming design discussion.

6.1 Discussion

Our motivation stems from the observation made in the prior chapters that past work has only formally explored heap sizes of the order of 600MB [BMBW2000]. Some work on CPU scalability has been done [LK1998, BMBW2000, Mic2004], but it appears that no work has been done on memory scalability – let alone scalability with respect to both CPU and memory.

Commercially available 64-bit SMP machines currently possess physical memory capacities in hundred's of gigabytes (in some cases more than a terabyte) with hundreds of processors [SGI2003]. Programmers wishing to fully utilize the capabilities of such machines, must be able to write programs which create heaps that extend to the entire CPU and memory range.

---

[99] Please observe our use of 'all things being equal'. We assume, for the purposes of this thesis, that doubling the number of CPU's in an SMP machine linearly doubles the available amount of CPU. This may, or may not be true in practice, depending on the architecture of the machine in question.

The purpose of this research is to explore the issues associated with scalability, with respect to both CPU and memory, so as to provide the engineering basis for a commercially available very large, highly concurrent, dynamic memory manager.

6.2 Definition of Terms

Restating, our particular concern is the ability of dynamic memory managers to handle *very large* amounts of memory on *highly concurrent* machines.

How large is very large? Two decades ago one megabyte was considered large. Given hardware trends, one megabyte twenty years from now will likely seem miniscule. In order to factor out this shift in subjective judgment, we choose to define very large in a timeless way – we define very large to be an arbitrary amount of memory. This implies the full address range of all current and future machines: 64-bit, 128-bit, 256-bit and so on. In practice (at this point in time of hardware evolution) this means very large is in the range 100GB to 10TB[100].

How concurrent is highly concurrent? Again to factor out any shifts in subjective judgment, we define highly concurrent to be in principle, an arbitrary number of CPU's. In practice (at this point in time of hardware evolution) this means highly concurrent is in the range 128 to 1,024 CPU's.

6.3 A Sense of Scale

A terabyte (2^{40} or 10^{12}) is a substantial amount of memory and can be difficult to comprehend[101] – particularly when it is in a single machine available to a single executing process. For example, one terabyte is sufficient to keep a 100 byte record on each of 10 billion items. So it is entirely feasible, that a modern machine with one terabyte of memory could be used to load and process a census of the entire planet and do so entirely in physical memory[102].

---

[100] As at 2004, machines with 10TB of memory are quite likely to be available within the next decade.

[101] For an interesting discussion on this, refer to the chapter "On Number Numbness" in [Hof1985].

[102] Compare this to the first machine used for census work, the UNIVAC-I [Ros1969]. Refer §4.1.1.

Following Moore's Law, CPU performance also increases over time. To get a sense of proportion, let us assume there is a hypothetical, highly scalable allocator that can allocate on average, a block of any size in 1,000 machine instructions.

Let us also assume, for the purposes of simplicity and clarity, that memory access time is completely uniform irrespective of the amount of memory in the machine. Please note that on some existing architectures memory access time may not be uniform – by design.

The following table shows the time taken to fully allocate a particular quantity of memory by such an allocator, given differing CPU speeds and memory capacities, assuming that the average block size is 100 bytes

		Memory Capacity							
	Instx/sec	1GB	10GB	100GB	1TB	10TB	100TB	1PB	1EB
Instruction Rate	1B	10	100	1,000	10,000	100,000	1,000,000	10,000,000	10,000,000,000
	10B	1	10	100	1,000	10,000	100,000	1,000,000	1,000,000,000
	100B	0	1	10	100	1,000	10,000	100,000	100,000,000
	1T	0	0	1	10	100	1,000	10,000	10,000,000
	10T	0	0	0	1	10	100	1,000	1,000,000

Table 6.1: Time (in seconds) to allocate all of memory just once given instruction rates and memory capacity – assuming average block size of 100 bytes and 1,000 instructions to allocate a block.

Typical high-end CPU speeds are currently in the 1B to 10B instructions per second range and typical high-end memory capacities are currently in the 100GB to 1TB range. As we can see from this table, the time taken for a single high-end CPU to fully allocate memory in that region, just once, would be in the range of one hundred seconds to just over 2 ¾ hours – which equates to 10^{15} to 10^{16} instructions for memory allocation, yet alone any application computation which may accompany it.

A terabyte heap would be 1,600 times larger than what has been reported in the literature to date, and we note that SMP machines, which could physically support such a heap, already exist today.

6.4 The Limits of Scalability

It is important to note that there are potential problems in scaling SMP machines. SMP machines give a view of a single shared memory by using complex protocols that manage such issues as cache line invalidation. These protocols are simple on a single bus but lose performance rapidly as the number of busses is increased.

Consequently doubling the number of CPU's and the amount of memory in a machine may in fact not produce a machine which is simply twice as fast with twice the memory capacity. Additionally there may be physical limits – particularly with current architectures – to the number of CPU's and the amount of memory a machine may feasibly have.

Predicting the general behavior of programs as the underlying machine is scaled is beyond the scope of this thesis. Not withstanding this, for the purposes of this thesis, we have embraced the previously mentioned "all things being equal" assumption that machines can be made bigger without penalty – doubling the CPU and memory in a machine does in fact produce a machine which is double the performance and capacity.

6.5 Feasibility

It is one of the tenets of this thesis that, the feasibility of such a very large, highly concurrent dynamic memory manager relies on the ability to allocate and free memory in constant time, irrespective of the heap size, and to allow any number of threads to concurrently do so – all the while maintaining good CPU and memory utilization.

It is important to note, that by constant time, we do not mean that each, and every action, must take exactly the same time, or require exactly the same number of instructions. Rather, we mean that the time taken, to allocate or free a memory block, must be independent of the size of the heap.

A very large allocator would therefore exhibit *O(1) performance*. Minor variations in performance due to circumstances (such as whether a block happens to get coalesced with other blocks because its neighbors are free or not, or having to request more virtual memory from the operating system) are entirely acceptable. A very large allocator, one which can handle an arbitrary amount of memory, simply does not slow down as the heap gets larger.

Even O(log n) performance would make very large heaps impractical – as the size of memory approaches infinity, so does performance. For many (soft) real-time applications, this means that responses times could not be guaranteed within reasonable confidence limits. Non-O(1) performance would prohibit the programming community from taking full advantage of machines which support very large memory capacities.

Being scalable with respect to CPU means the allocator should exhibit *linearly scalable* O(1) performance – doubling the number of CPU's should in practice deliver close to a doubling in the allocation and freeing rate, irrespective of heap size. An allocator which was not scalable with respect to CPU would not be able to take full advantage of the processing power available in large multiprocessor environments – throughput would be suboptimal.

6.6 Practical Considerations

There are a number of other considerations, apart from CPU and memory scalability, which need to be taken into account, in order for any general purpose allocator to be practical. This includes time and space efficiency, as well as long term stability.

6.6.1 Time Efficiency

In 1998 Larson and Krishnan [LK1998] commented that, at the time, a server may encounter a peak load which required one million allocations and deallocations per second, in order to service its tasks at hand in a timely and responsive manner. While they did not formally state the required allocation or deallocation rate in terms of CPU time, it is certainly possible to do so.

Over successive hardware generations, CPU speeds have tended to increase far more rapidly than memory speeds [Sei2000]. Consequently, CPU stalls (when measured in terms of lost instructions) due to cache misses tend to become relatively more expensive over successive hardware generations. This is further exacerbated, due to the fact that, as memory capacities increase over successive hardware generations, there is a further increase in the probability of cache invalidation and lost instructions due to CPU stalls.

Despite this, we should not shy away from stipulating a reasonable estimated cost for allocating or deallocating a memory block – particularly from an engineering perspective, where we require requests to be processed in constant O(1) time. For the

purposes of this research, we consider a reasonable first order estimate of 1,000 instructions to allocate or deallocate a block of memory, on average, in an SMP environment to be reasonable.

One may consider 1,000 instructions to be too small or too large – depending on ones perspective. This budget potentially includes such steps as identifying which sub-heap to allocate from or release to, locking various structures, requesting or releasing pages of virtual memory and some housekeeping. We believe therefore, that 1,000 instructions on average to be a reasonable and achievable first order estimate.

6.6.2 Space Efficiency

Memory utilization is also important – particularly for very large heaps. A very large heap which had a 100% spatial overhead would waste half of all memory. Inversely, it would only be able to support half the actual number of blocks. Either way, a terabyte of memory is not cheap, so wastage needs to be minimized.

Some overhead of course is required to provide the dynamic nature of heap management. Johnstone [Joh1997] showed that wastage due to memory fragmentation can be less than 5%. When one accounts for boundary tags and other structural issues, best practice (such as DLmalloc) indicates that total wastage can be kept to 20-25%.

Locality has also been shown to affect an allocators behavior. A state of the art allocator should attempt to allocate blocks near each other when those blocks are used together. Johnstone also showed that space efficiency and locality do not need to be traded off against each other.

6.6.3 Time v. Space Tradeoff

There has been considerable debate over performance and fragmentation – and the tradeoff there between. We agree with [WJNB1995] that it should be possible to build allocators which are both CPU and memory efficient. An allocator which achieved both CPU and memory scalability at the expense of both CPU and memory efficiency may be of theoretical interest, but would be of questionable value in practice.

One of our main objectives is to provide the engineering basis for a commercial product. Consequently, a realistic very large, highly concurrent dynamic memory manager should be efficient with respect to CPU and memory utilization.

6.6.4 Distribution Independence

Much has been said about the distribution of block sizes, lifetimes and request arrival patterns. The majority of this commentary has been about whether the evaluation of allocators has been valid or not – particularly with respect to fragmentation and to a lesser extent speed. From the perspective of stipulating requirements, we strongly concur with Larson and Krishnan [LK1998], that the speed of a contemporary allocator should not be affected by these distributions.

Just as important, we take note of Johnstone's conclusion [Joh1997] that there are a number of policies which yield low fragmentation in practice. We conclude that a very large, highly concurrent allocator should be capable of being fast and memory efficient in most real world circumstances.

6.6.5 Long-Term Stability

As observed by Larson and Krishnan [LK1998], the work on DMM to date has mainly been focused on supporting one-shot applications (short-lived programs which perform one particular task and then terminate), as opposed to long-running server applications (programs which perform a set of tasks and in principle may never terminate). [WJNB1995] noted that very little, if anything, is formally known about the long-term behavior of allocators.

By implication, large heaps will tend to be associated with long running programs – the larger the heap, the longer it will take to populate and use. Even one-shot programs can allocate and free many times more than the available memory, so indefinitely long running programs could, in principle, allocate and free an indefinite amount of memory.

For programs which run for long or indefinite times, the behavior of the heap manager (particularly with respect to CPU and memory efficiency) should not vary over time. The behavior of a heap manager, after running for say ten years, on some highly reliable machine which permitted this, should be the same as it was at invocation; otherwise programs will not behave predictably over time.

Berger et al. [BMBW2000] observed that concurrent allocators will exhibit blowup when resources (such as blocks and sub-heaps) are tied to specific threads and cannot be shared or reassigned. We conclude that a state of the art concurrent allocator should have evenly shared resources, independent of threads.

6.7 Requirements

We formally state our requirements for very large, highly scalable dynamic memory managers as follows:

1. **General Purpose** – such an allocator should be capable of being used for a broad range of applications;

2. **CPU Scalability** – such an allocator should exhibit near linear scalability with respect to CPU. Doubling the number of CPU's close to doubles the allocation and freeing rate. Only a very modest amount of CPU time should be expended in ensuring critical synchronization between multiple threads on multiple CPU's;

3. **Memory Scalability** – such an allocator should exhibit linear scalability with respect to memory, thereby being able to handle an arbitrary amount of memory;

4. **CPU Utilization** – such an allocator should be highly efficient with respect to CPU, able to on average allocate or deallocate a block of memory in the order of 1,000 instructions;

5. **Memory Utilization** – such an allocator should be highly efficient with respect to memory, able to require less than 20% space overhead to represent and manage a heap of any size, with no more than an additional 5% lost due to fragmentation;

6. **Maximal Locality** – such an allocator should maximize locality by attempting to allocate blocks of memory that are typically used together near each other;

7. **Distribution Independence** – such an allocator should not be affected by block size, block lifetime or request arrival distributions;

8. **Resource Sharing** – such an allocator should have resources independent of threads. Blocks allocated by one thread should be capable of being freed by any

thread. Sub-heaps should be capable of being shared by, or be re-assignable between, multiple threads thereby eliminating blowup;

9. **Avoid False Sharing** – such an allocator should to minimize the phenomena of false sharing of cache lines, thereby minimizing the possibility of cache thrashing; and

10. **Predictable Behavior** – such an allocator should have time independent behavior. The allocator should be capable for running indefinitely without any of the above behaviors changing.

6.8 Definition

With consideration to the above requirements, we arrive at the following definition:

> ***A very large, highly concurrent dynamic memory manager exhibits efficient and predictable, linearly scalable, O(1) time-space behavior.***

Such an allocator would be time and space independent, while being time and space efficient. This is the quintessential quality which would enable an allocator to handle an arbitrary sized heap, on a machine with an arbitrary number of CPU's, with highly predictable behavior.

In the next chapter we explore the solution space via a number of thought experiments. We restate our objectives, divide our concerns and derive three principles which guide us to a workable solution.

Chapter 7

7 Design Discussion

Designing and implementing a dynamic memory manager, for the modern 64-bit class of machines and beyond, which scales linearly with respect to both CPU and memory, is a complex task. The problem space is large, any possible solutions are non-obvious, and subtle variations in implementation are known to lead to significant variations in behaviour.

We use a number of thought experiments to explore the spectrum of potential solutions, along with their associated problems, and arrive at a feasible solution. We then sketch an informal design for an allocator, which satisfies our stated requirements.

We present a formal specification for our allocator in the next chapter.

7.1 Introduction

We have surveyed a representative cross section of the literature, and examined the current state of the art in allocators, so let us begin our design discussion by attempting to design a scalable allocator, on the basis of an educated guess, and then critically analyzing it to see how we fare.

Let us assume a hypothetical machine, where CPU's, both in number and speed, are plentiful, as is memory. Following a basic "keep it simple" approach, our initial best guess is to have the following arrangement:

- Blocks conform to fixed sizes, rounded up to the next power of two;
- The free blocks are managed by a single vector of lists; and
- Each list is a guarded by a lock.

The allocation process would be:

- Calculate the block size to be allocated (this can be done very efficiently using bit masking and shifting [War2002]);

- If the list for that size has blocks, lock the list, pop the first block, and then release the lock; and
- If the list is empty, search for the next biggest list, lock the list, pop the block, release the lock, split the block appropriately and push the split off block(s) into their appropriate lists (locking accordingly). Note there may be more than one split off block, because all blocks must be a power of two.

We can visualize our, keep it simple power-of-two allocator, as follows:

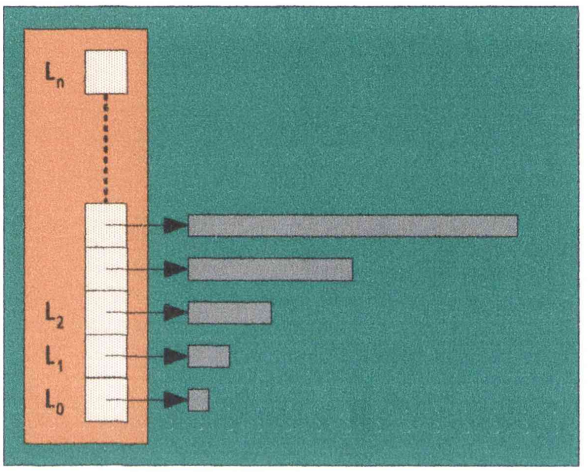


Figure 7.1: A simple power-of-two allocator.

This seems an elegant arrangement which, given a very large machine, with plentiful CPU and abundant memory, should on first blush produce excellent results:

- A large vector should be able to support a large number of threads accessing different elements;
- There is no list searching in this arrangement; and
- The amount of waste should be about half the difference between successive block sizes. In this case, this should equate to a 25% overhead, which for an elegant high speed solution is arguably acceptable.

However, from the experiences in the literature we know:

- The power-of-two approach leads, in practice, to unexpectedly high memory wastage [Nei1977]. Johnstone observed as much as 85% total wastage using real world programs, primarily due to very high external fragmentation [Joh1997];

- Such allocators generate very high cache miss rates – Johnstone observed simple power-of-two allocators generated the highest cache miss rates of the 53 allocators he tested; and

- Tests by Larson and Krishnan [LK1998], Berger et al. [BMBW2000] and Michael [Mic2004] have consistently confirmed that variations of this kind of allocator do not scale under load, in many cases, beyond two or four CPU's and/or lead to memory blowup.

With respect to the last point, real world programs tend to request dynamic memory from a small set of discrete sizes. Consequently, certain lists become hotspots. Multiple caches exacerbates the problem: as we add CPU's into our model, the hotspot memory locations are in demand across an ever increasing population of caches. The contents of those hotspots get sloshed between caches (over buses) leading to thrashing.

Quoting Larson and Krishnan:

> *"To build a highly scalable allocator, it is not sufficient to minimize lock contention. One must also reduce bus traffic by reducing the frequency of access to shared, fast-changing data items like list heads, counters, block headers and block footers."*

As we will show, there is no simple tweak to the "keep it simple arrangement" which yields a satisfactory answer[103]. Solving fragmentation requires managing a much finer range of sizes than just power-of-two; solving scalability requires a design which is concordant with the nature of scalable hardware architectures.

One of the harsh realities is that CPU speeds are continuing to increase much faster than memory and bus speeds [Sei2000]. To a large extent, the performance of modern processors therefore relies heavily on hardware cache, as well as carefully tuned instruction sequences which maximize cache utilization. Brute force searching of large data structures, for example, tends to invalidate cache.

---

[103] It is entirely possible that as further heap scalability work occurs, this elegant guarded vector, power-of-two allocator will be rediscovered and discarded many times – just as the author and his colleagues did.

Modern machines, though increasingly more computationally powerful and spatially abundant, are increasingly more complex devices – the product of many sophisticated electronic sub-systems, layered on top of each other. The keep it simple allocator, while conceptually elegant, is substantially discordant with the physical nature of those electronic sub-systems.

The following diagram is an abstract visualization of an indefinitely scalable SMP machine, one in which we can always add more CPU's and memory:

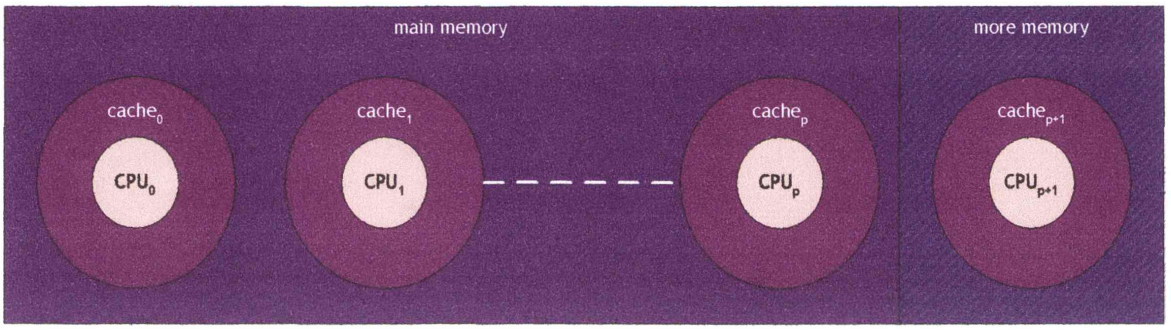


Figure 7.2: Abstract model of an indefinitely scalable SMP machine. The borders of the circles may be thought of as buses.

To a very large extent, to achieve scalable software on multi-processor, multi-cache machines requires the software to be capable of multiplexing its data structures simultaneously across multiple caches, such that cache hit rate is maximized, while cache invalidation and sloshing is minimized – with consequential minimization of bus traffic.

What we are fundamentally searching for, in the following design discussion, is an arrangement which can, loosely speaking, stretch and expand indefinitely across any number of CPU/caches and amount of memory, without any inherent loss of efficiencies. Such an arrangement needs to maximize memory efficiency – by maintaining a fine range of blocks sizes; while maximizing CPU efficiency – by maximizing the cache occupancy of relevant data in all caches; all the while minimizing the co-residency of such data in multiple caches.

7.2 Our Philosophy

Given the shear volume, breadth and evolutionary nature of the work within this subject, it is more than likely that many of the experiences and lessons garnered from

that work are quite relevant for large scale, highly concurrent dynamic memory management. While there is the likely need for invention and novelty, in order to solve some of the thornier issues, it is quite likely that the overall solution will draw heavily from techniques which, in some cases, have been in existence for decades[104]. Understanding the workings of a feasible solution (however poor the first version is) is as important as the solution itself.

In order to maximize the probability of success (whilst minimizing the degree of difficulty and associated costs) the view we hold in starting the design phase of this project, is that the literature abounds with many valuable lessons – which, if we are to achieve our goal, simply cannot be ignored, but rather should be embraced as guideposts to success. We should therefore uphold and reuse, as best as possible, the mentality and experiences embodied in the literature, yet being careful to identify and address problematic areas[105].

7.3 Basic Concerns

We begin by stating our fundamental concerns and working outwards from that point. As stated, our objective is to build a very large, highly concurrent dynamic memory manager: that is, an allocator that exhibits efficient and predictable O(1) behavior, on a machine which has an arbitrary amount of memory, and an arbitrary number of CPU's.

We break our objective into two separate concerns:

- An allocator that can manage an arbitrary amount of memory with O(1) service times; and
- An allocator that scales linearly with respect to CPU.

Reasoning from first principles about the first concern, we can see that in order to have O(1) service times, the operations must be independent of the total memory in the

---

[104] This is a polite way of saying our gut instinct is that, a feasible solution is likely to be an 80% utilization of existing techniques and 20% new – as opposed to say 80% new.

[105] This does not mean we believe prior work to be flawed, but rather we view prior work as valuable insight into a complex topic, which has not yet been fully addressed.

machine. Each operation must be an action, which is finite in some manner that does not change, as we vary the memory in the model. Failure to achieve this would mean that, service times would approach infinity, as memory approaches infinity.

Reasoning from first principles about the second concern, we can see that in order to be able to have any number of concurrent threads (which exclusively lock and hold resources during operations), implies there must also be an increasing multiplicity of those shared resources – in order to increase throughput as we add more CPU's into the model. Failure to achieve this, would mean throughput would at best remain constant, or decay to zero, as the number of CPU's approaches infinity.

7.4 Guiding Principles

We start with two guiding principles, to assist us, in our search for an appropriate solution:

- Each operation must be an action which is finite in some manner, and which does not change as we add more memory into the model; and
- There must be an increasing multiplicity of shared resources, in order to increase throughput, as we add more CPU's into the model.

7.5 Thought Experiment #1

Let us begin our search for a solution, by crafting an example of the simplest allocator possible, enabling us to acutely show the problems of CPU and memory scalability for the purposes of discussion. Such an allocator, which we will name TEX-1, would be a thread safe variant of that described in [Knu1973] – a single list of free blocks, where the list is guarded by a lock. Using Berger's taxonomy, TEX-1 would be a serial single heap.

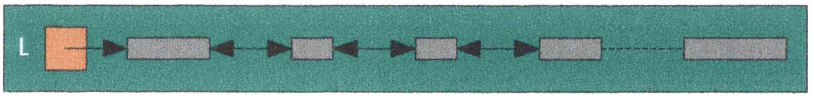


Figure 7.3: TEX-1 – a single linked list of free blocks guarded by lock L.

Let us assume the list is kept in LIFO order. To allocate, we first lock the list, then linearly search the list looking for the first block which will fit, remove that block, and

then release the lock. To free, we lock the list, then push the block onto the front of the list, and then release the lock.

In this arrangement we are linearly searching a single list. On average, we would need to search half the list, to find an appropriate block, so allocation service time is O(n). As memory approaches infinity, so does allocation service time. Freeing is simply a push action which requires no searching, so freeing service time is O(1).

In this arrangement we are locking a single resource. Only one thread can access the list at any one time. As we increase the number of CPU's no improvement in throughput can be achieved[106].

If the list was maintained in some order other than LIFO (such as address or size order) then when we free a block, the thread would first lock the list, search through the list for the appropriate insertion point (based on address or size), insert the block at that point, and then release the lock. In this case, the problem is worse – as freeing service time has become O(n).

As we can see, TEX-1 fails both of our guiding principles: our actions are not independent of the amount of memory in the model, and there is only one single shared resource (the guarded list) which does not increase as we increase the number of CPU's.

7.6 An Observation on Delayed Coalescing

With delayed coalescing, the computational cost of coalescing is not spread or amortized across each free request. Rather, at some future point, a request will have to perform coalescing. That coalescing action, if not bounded in some way, could take an indeterminate amount of time and still may not even yield a free block of suitable size.

As we increase the amount of memory, deferred coalescing of this kind may lead to periodic inordinately long response times. So we arrive at the conclusion that for very large dynamic memory management, if deferred coalescing is to be employed, then it must be limited or dampened in some way.

---

[106] In fact, depending on the physical architecture of the machine, adding CPU's may cause throughput to decay – possibly to zero. This can be due to acute bus saturation and cache sloshing, causing the machine to thrash, so that no effective work gets done.

7.7 Thought Experiment #2

Let us craft an example of an allocator which has multiple guarded lists, which we will call TEX-2. Multiple guarded lists have been explored by Iyengar [Iye1993, Iye1996]. One feasible technique is to have a vector of guarded lists. Using Berger's taxonomy, TEX-2 would be a concurrent single heap.

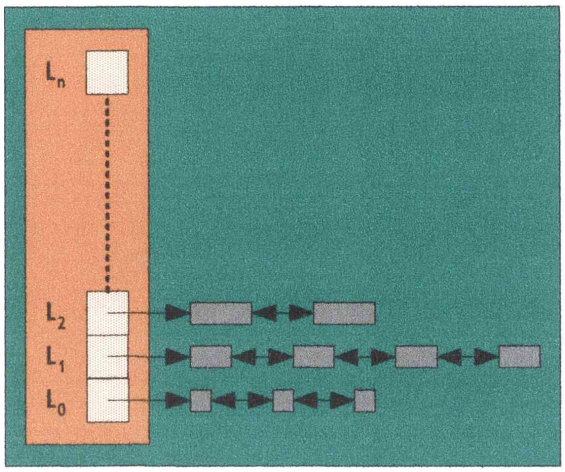


Figure 7.4: TEX-2 – a vector of guarded lists.

Let us assume that the allocator only allocates a small, finite number of different sized blocks – say 64 different sizes – and that lists are kept in LIFO order. To allocate, we first calculate which list to lock (based on the size of the requested block), we then lock that list, pop the first block off the list, and then release the lock. To free, we first calculate which list to lock (based on the size of the block being freed), we lock that list, push the block onto the front of the list, and then release the lock.

In this arrangement we are not linearly searching any of the lists, so allocation and freeing service time is O(1). As memory approaches infinity, allocation and freeing time remain constant.

In this arrangement we are locking one of a finite set of resources. Although only one thread can access any one list at any one time, as many threads as lists can be operating simultaneously. However, as we add CPU's into the model no real improvement in throughput can be achieved.

As we can see, TEX-2 fails our second guiding principle: the shared resources (the set of guarded lists) do not increase as we increase the number of CPU's.

Additionally, to be useful, an allocator needs to be able to manage a reasonably broad set of sizes. This takes us to TEX-3.

7.8 Thought Experiment #3

Let us craft an example of an allocator which could be described as a concurrent version of DLmalloc. This allocator has two vectors of guarded lists: one which handles exact sizes (for the smaller sizes) and the other which handles sorted sizes (for the larger sizes). We will call this allocator TEX-3. Using Berger's taxonomy, TEX-3 would be a concurrent single heap.

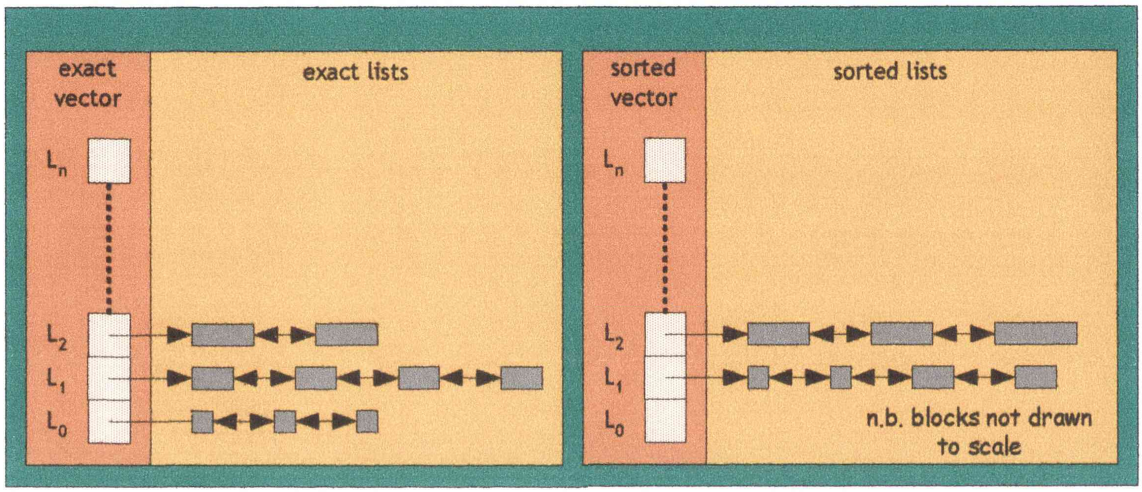


Figure 7.5: TEX-3 – a vector of guarded exact lists and a vector of guarded sorted lists.

Let us assume that, the sorted lists effectively group blocks into lists which are of similar size; that like DLmalloc the groups are roughly logarithmically spaced; that the exact lists are kept in LIFO order; and that the sorted lists are kept in size order.

To allocate, we first calculate whether we are allocating an exact size or a sorted size. This determines which vector we access. If it is an exact size we then calculate which exact list to lock (based on the size of the requested block), we lock that exact list, pop the first block off the list, and then release the lock. To free an exact size we first calculate which exact list to lock (based on the size of the block being freed), we lock that list, push the block onto the front of the list, and then release the lock. If it is a sorted size we first calculate which sorted list to lock (based on the size of the requested block), we then lock that sorted list, linearly scan the list looking for the first appropriate block, remove that block from the list, and then release the lock. To free a

sorted size we first calculate which sorted list to lock (based on the size of the block being freed), we lock that list, linearly scan the list looking for the appropriate position to insert the block, insert the block, and then release the lock.

In this arrangement, we are linearly searching the sorted lists, so allocation and freeing service time is O(n). As memory approaches infinity, the time to allocate and free also approaches infinity.

From a locking perspective, TEX-3 is the same as TEX-2: we are locking one of a finite set of resources. As we add CPU's into the model, no real improvement in throughput can be achieved. As stated for TEX-2, real world programs exhibit patterned behavior, often allocating and freeing most blocks from a small sub-set of sizes. In the worst case, where a multithreaded program were to only allocate and free one size, TEX-3 also degenerates into TEX-1.

As we can see TEX-3 fails both of our guiding principles: our actions are not independent of the amount of memory in the model, and there is only one single shared resource (the set of guarded lists) which does not increase as we increase the number of CPU's.

7.9 Thought Experiment #4

Let us craft a variation of TEX-3, in which the lists that manage the larger sizes are not sorted. This is similar to a sub-heap in LKmalloc. We will call this allocator TEX-4. Using Berger's taxonomy TEX-4, would be a concurrent single heap.

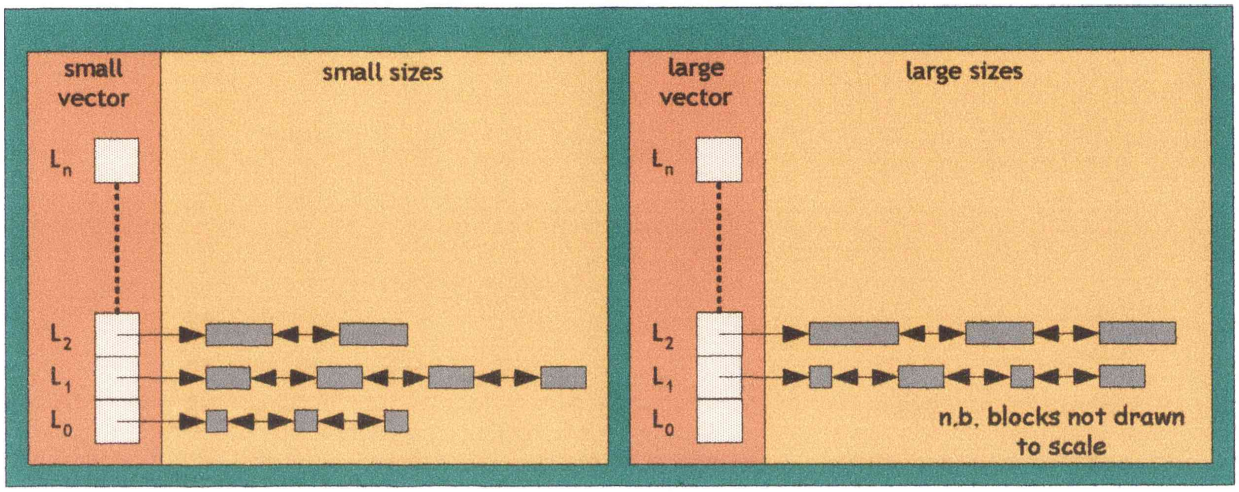


Figure 7.6: TEX-4 – a vector of guarded small size lists and a vector of guarded large size lists.

All other things being equal, the only real difference of note between TEX-4 and TEX-3 is that freeing becomes O(1) – allocation remains O(n).

To allocate a large size block, we calculate which large list to lock (based on the size of the request), then lock that large list, linearly scan the list looking for the first appropriate block, remove that block from the list, and then release the lock. To free a large size, we first calculate which large list to lock, lock that list, push the block onto the front of the list, and then release the lock.

In this arrangement we are still linearly searching the large list, so allocation time is O(n). So as memory approaches infinity, allocation time also approaches infinity.

From a locking perspective TEX-4 is the same as TEX-3 and TEX-2: we are locking one of a finite set of resources. As we continue to increase the number of CPU's, no improvement in throughput can be achieved. Given that real world programs exhibit patterned behavior (often allocating and freeing most blocks from a small sub-set of

sizes), in the worst case where a multithreaded program only allocated and freed one size, TEX-4 also degenerates into TEX-1.

As we can see TEX-4 fails both of our guiding principles: our actions are not independent of the amount of memory in the model, and there is only one single shared resource (the guarded list) which does not increase as we increase the number of CPU's.

7.10 Thought Experiment #5

Let us craft an allocator that is a larger version of TEX-2, in which the vector covers all possible sizes we are interested in. We will call this allocator TEX-5. Using Berger's taxonomy, TEX-5 would be a concurrent single heap.

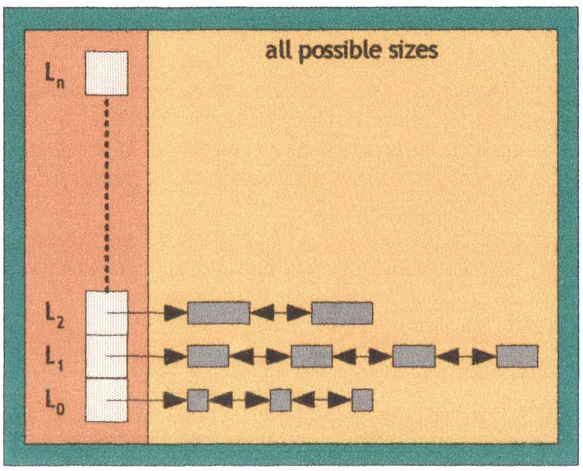


Figure 7.7: TEX-5 – a vector of guarded lists explicitly covering all possible sizes.

For the sake of simplicity, let us assume that the lists are kept in LIFO order. To allocate, we calculate which list to lock based on the request size, we then lock that list, pop the first block off the list, and then release the lock. To free, we calculate which list to lock based on the size of the block being freed, we lock that list, push the block onto the front of the list, and then release the lock.

In this arrangement, we are not linearly searching any of the lists, so allocation and freeing service time is O(1). As memory approaches infinity, allocation and freeing time remain constant.

The Achilles' heal in this arrangement is that we are still only locking one of a finite set of resources. As we continue to increase the number of CPU's, no improvement in throughput can be achieved. Additionally, real world programs exhibit patterned

behavior, often allocating and freeing most blocks from a small sub-set of sizes. In the worst case, where a multithreaded program were to only allocate and free one size, TEX-5 degenerates into TEX-1.

TEX-5 fails our second guiding principle: the shared resources (the set of guarded lists) do not increase as we increase the number of CPU's.

7.11 Interim Conclusion #1

At this point it is reasonable to conclude that a very large, highly concurrent dynamic memory manager (one that exhibits efficient and predictable O(1) behavior on a machine which has an arbitrary amount of memory and an arbitrary number of CPU's), will not be some variation of a concurrent single heap – no matter how that heap is optimized. There appears to be no possible arrangement of concurrent single heap which will scale indefinitely with respect to both CPU and memory.

A further disadvantage in our designs so far, is the hidden assumption that access to the elements is evenly spread. Research into program execution patterns, indicate that memory allocation is far from an even distribution across all possible sizes. Most programs exhibit highly uneven distributions, producing very high demands for specific sizes. This has the effect for producing "hot spots" – areas in memory which are highly contested amongst threads.

In this case a subset of elements would tend to be highly contested. While the vector may have a large number of elements in it, only a small number of hot elements may ever be accessed – possibly even just one in extreme cases. Excessive thread collision rapidly leads to CPU wastage (particularly if spin locks are being used as the locking mechanism) and increased burden being placed on the operating system scheduler [Lov2003].

These observations led to the work by Berger on Hoard.

We arrive at the conclusion that, a single vector is not an acceptable solution, in that it will not provide good overall performance with respect to throughput. In fact, it may provide very poor performance in many cases.

7.12 Pushing Forward

Since a concurrent single heap is not likely to be the answer, then let us explore multiple sub-heaps.

If a single vector is not likely to achieve good sustainable throughput for most cases, then we arrive at the conclusion that we require multiple vectors – each supporting concurrent access. Many questions then arise:

- How many vectors should we have?
- Is the number of vectors fixed or dynamic?
- How many threads, running on distinct CPU's, could be supported by a single vector and still achieve good throughput?
- How much memory should be tracked by a vector?
- What is the correlation between the number of CPU's, the number of threads, the amount of memory, the size of cache, the bus performance and the number of vectors? – All of which is arguably *a priori* knowledge which may vary from machine to machine.

One possible answer is that we should have a dynamic number of vectors. That is, a new vector should be created when the circumstances demand it.

Keeping the described experiences with PTmalloc in mind, it appears important to dampen/control the rate/situation when new vectors are created, so as to avoid pathological run-away cases, which prematurely exhaust memory.

Note that (unlike PTmalloc) each of the proposed vectors is multithreaded – they can, in principle, support an indefinite number of threads accessing them concurrently (though they may queue when they collide for specific elements). It seems reasonable therefore, to only create a new vector (and have threads switch to it) when thread collision exceeds a threshold which is likely to be causing throughput problems.

7.13 Thought Experiment #6

Let us craft an example of simple multiple sub-heaps. Such an allocator, which we will name TEX-6, would be a variant of TEX-1, where each thread has a sub-heap consisting of a single list of free blocks for its exclusive use. Since each list is for the exclusive use of the associated thread, there is no need for a lock. Using Berger's taxonomy, TEX-6 would be classified as pure private sub-heaps.

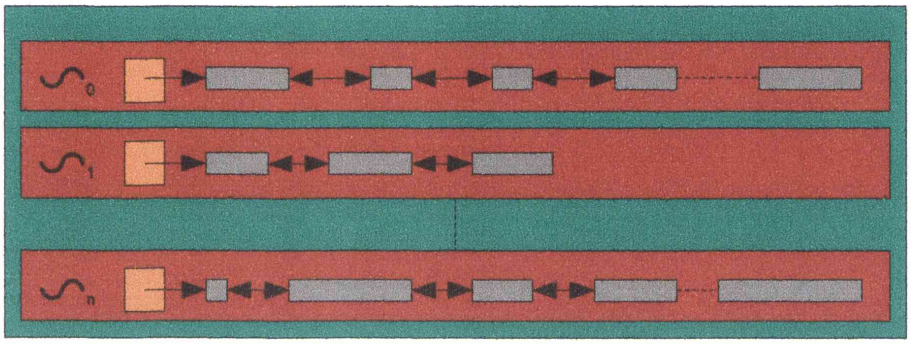


Figure 7.8: TEX-6 – one private sub-heap per thread.

Let us assume the lists are kept in LIFO order. To allocate, a thread linearly searches its list looking for the first block which will fit, and then removes that block from the list. To free a block, a thread simply pushes the block onto the front of the appropriate list.

The strength of this arrangement is that it can support any number of threads; as the number of CPU's approach infinity there is no concurrency bottleneck.

There are, however, several major weaknesses in this arrangement. First and foremost, purely private sub-heaps are simply not feasible for a general purpose allocator. Private sub-heaps imply that, the thread which frees a block must always be the thread that allocated the block in the first place. This is an untenable restriction.

Secondly, as we scale memory, any one list may grow indefinitely long. Since service times are O(n), then an allocation may take an indefinitely long time.

Thirdly, list lengths are not balanced in any way. In principal, a list may grow much larger than any other. Memory, which is available in one sub-heap, is not available to a different thread. This leads to the phenomena of blowup.

TEX-6 fails the first guiding principle, in that its actions are not independent of the amount of memory in the model.

7.14 A Third Guiding Principle Emerges

TEX-6 is an interesting experiment. One of its major weaknesses is that over time one or more of the lists may grow significantly longer than the others. In that sense, the system becomes *unbalanced*. A workable solution is likely to be one that tends to remain balanced over time.

This leads to a third guiding principle:

- Resources in the system will tend to remain balanced over time.

7.15 Thought Experiment #7

The fundamental flaw with TEX-6 was that free blocks could not be moved between lists. Let us craft a multiple sub-heap allocator, where free blocks can move between lists. Such an allocator, which we will name TEX-7, would be a variant of TEX-6 where each thread has a sub-heap consisting of a single list of free blocks. Allocation is performed by taking a suitable block from the sub-heap associated with the thread. Any thread can free any given block, adding it to its own sub-heap, not the sub-heap it was allocated from. Since lists are now accessed by multiple threads they need to be locked. Using Berger's taxonomy, TEX-7 would be classified as private sub-heaps with ownership.

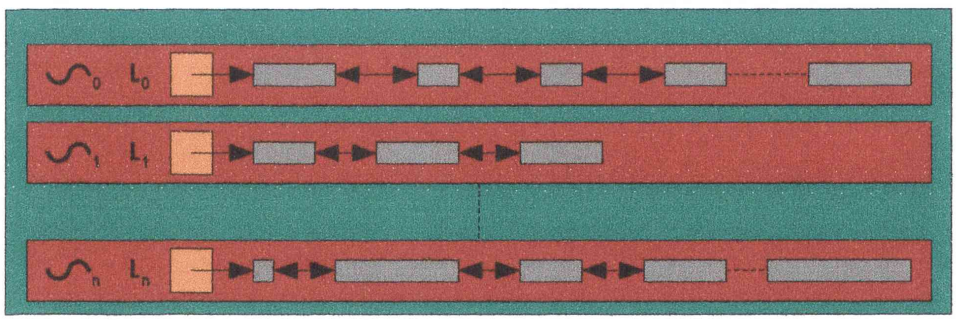


Figure 7.9: TEX-7 – one guarded sub-heap per thread, freed blocks being added to the sub-heap associated with freeing thread.

The major weakness with this model is that it violates our third guiding principle: even though blocks can move between sub-heaps the system can easily become unbalanced. A simple example of this is a programming model where there are garbage collection

threads. All blocks eventually end up on the sub-heap(s) associated with the garbage collecting thread(s), thereby starving the other threads of memory resources.

7.16 Thought Experiment #8

The fundamental flaw with TEX-7 was that the system could become unbalanced, because freed blocks were added into the sub-heap associated with the freeing thread. Let us craft a multiple sub-heap allocator, where blocks are returned to the sub-heap from which they were allocated. Such an allocator, which we will name TEX-8, would be a variant of TEX-7, such that each thread has a sub-heap consisting of a guarded list of free blocks. This class of allocator is not identified within Berger's taxonomy[107]. However, we could classify TEX-8 as multiple concurrent sub-heaps with ownership.

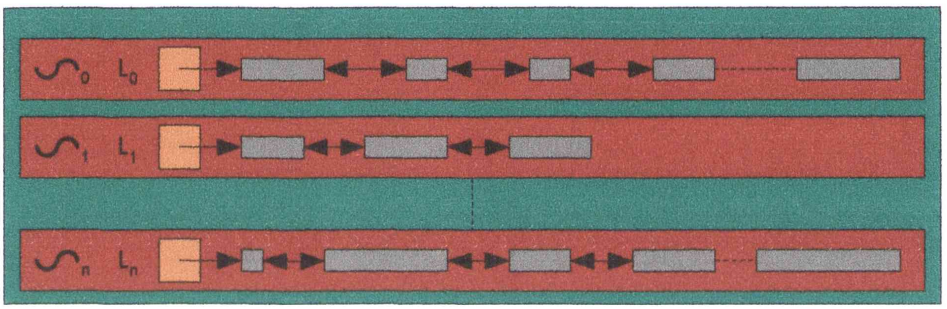


Figure 7.10: TEX-8 – one guarded sub-heap per thread, freed blocks being returned to the sub-heap from which they were allocated.

The major weakness with this model is that it still violates our third guiding principle of balance. Returning blocks to the sub-heap from which they were allocated, eliminates the problem of a freeing thread ending up with all of memory, but it does not eliminate the problem that, memory in one sub-heap can only be allocated by the owning thread. Consequently, one sub-heap could still end up with all of memory.

Additionally, as any one list gets longer, service times will elongate – as memory approaches infinity service times can as well.

---

[107] Refer to Appendix D for our proposed taxonomy.

7.17 Interim Conclusion #2

Based on thought experiments 6, 7 and 8 it is reasonable to conclude that a workable solution will not be some variation of private sub-heaps, or private sub-heaps with ownership.

The principle problem is that resources are not being dynamically shared. There appear to be two ways to address that problem. Either move the resources between sub-heaps, or have the threads move between sub-heaps.

7.18 Thought Experiment #9

Let us craft an allocator which moves resources between sub-heaps. Such an allocator, which we will call TEX-9 could have a sub-heap per thread and a common sub-heap. Threads attempt to allocate from their own sub-heap. If the allocation cannot be satisfied from a threads private sub-heap, the thread attempts to allocate from a common sub-heap. Using Berger's taxonomy, TEX-9 would be described as private heaps with thresholds.

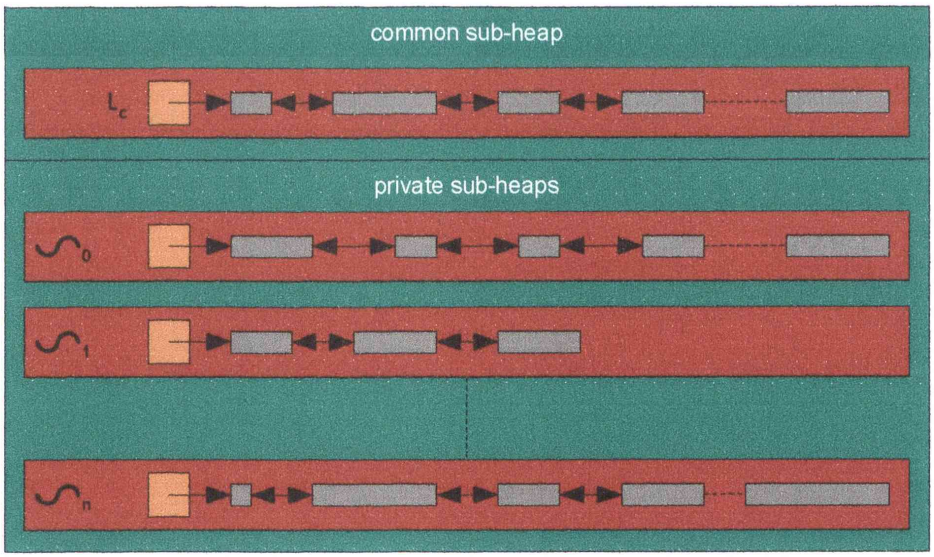


Figure 7.11: TEX-9 – common sub-heap accessible by all threads, plus a sub-heap private to each thread.

The private sub-heaps are not guarded, as each sub-heap is only accessed by the associated thread – threads allocate from, and free to, their own sub-heap. In this model, threads allocate from their own sub-heap, unless the request cannot be satisfied, where upon which they lock the common sub-heap and allocate from there. In order for

the model to remain balanced, threads at some point must then also move blocks back to the common sub-heap. The trigger, for freeing to the common sub-heap, could be based on a number of different events, such as the threads sub-heap becoming "full" in some sense. This could be an absolute fullness, based on number or total size of blocks in a sub-heap, or by some relativity to other sub-heaps. For the purposes of discussion, let us assume that a thread can detect some "fullness" property, upon which it frees blocks to the common sub-heap.

The strength of this arrangement is that resources are shared between threads. There are also a number of weaknesses. Firstly, the common sub-heap can grow indefinitely large, and since that searching the list is O(n), allocation service time approaches infinity as memory approaches infinity. Secondly, since there is only one common sub-heap with a single lock, the common sub-heap becomes a concurrency bottle neck as the number of threads is increased. Thirdly, even though resources are shared, they are only shared to a limited degree. There can be an indefinite amount of memory in private sub-heaps – none of which is accessible to any thread other than the thread associated with the sub-heap – thereby having the potential to exhibit blowup.

7.19 Thought Experiment #10

Although it is unlikely to be the solution, for the sake of completeness, let us explore the issues associated with multiple common sub-heaps. Such an allocator, which we will call TEX-10, could have one private sub-heap per thread, as well as access any number of common sub-heaps. Threads first attempt to allocate from their own sub-heap. If the allocation cannot be satisfied from the thread's private sub-heap, the thread selects a common sub-heap and attempts to allocate from that common sub-heap. As in TEX-9, this allocator frees to a common sub-heap when its own private sub-heap is full. Using Berger's taxonomy, TEX-9 would be described as private heaps with thresholds.

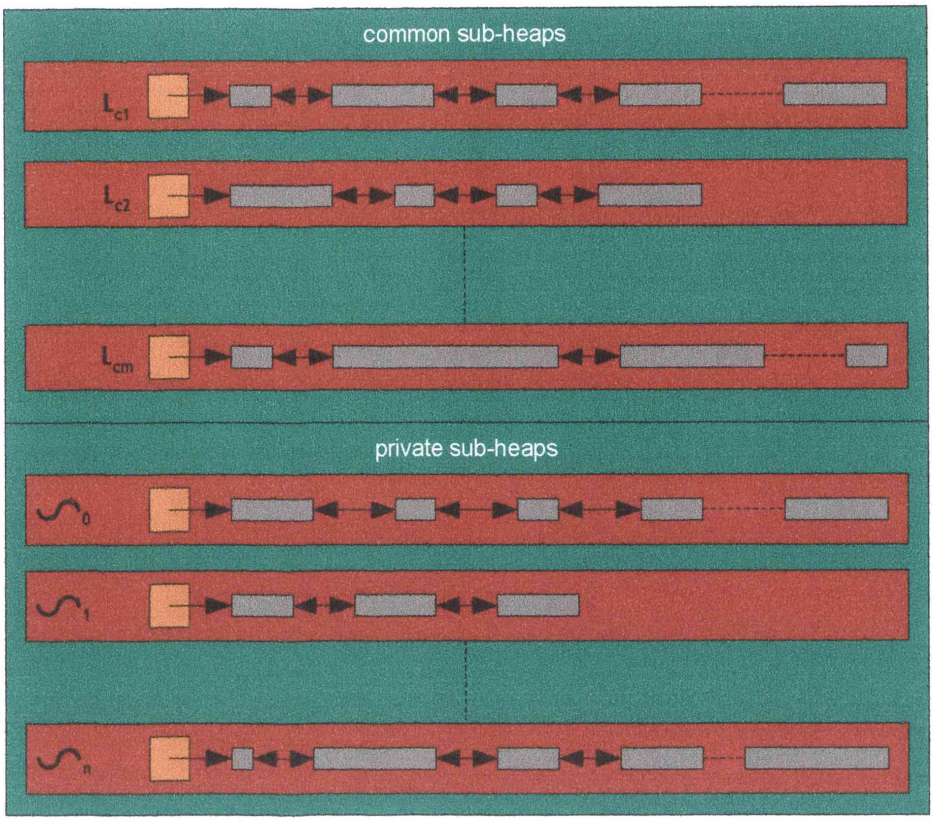


Figure 7.12: TEX-10 – multiple common sub-heaps accessible by all threads, plus a sub-heap private to each thread.

The strength of this arrangement is that resources are shared between threads. Additionally, if the number of common sub-heaps can be varied in some manner, as we add CPU's into the model, then sub-heaps are no longer an intrinsic bottleneck.

There still are a number of weaknesses though. Firstly, each of the common sub-heaps can grow indefinitely large, and since searching a lists is O(n), then allocation service time approaches infinity as memory approaches infinity. This could be alleviated by ensuring that common sub-heaps are bounded in some manner, such that they can only grow to a certain size, however as memory approaches infinity, then the number of common but bounded sub-heaps also approaches infinity.

The problem has now shifted to finding a suitable common sub-heap from an indefinite number of common sub-heaps.

7.20 A Valuable Insight

It is from TEX-10 that a valuable insight emerges. It appears, that in order to place an upper limit on the service times, sub-heaps need to be bounded in some way – by the number of blocks they contain, or the amount of memory they manage, for example.

If we bound the size of a sub-heap, then as memory approaches infinity, we would have an infinite number of finitely sized sub-heaps. The problem then shifts from finding a suitable sized block from an unbounded set of blocks, to finding a suitable sub-heap from an unbounded set of sub-heaps.

Up until this point, generally speaking, we have had models within which, there were private sub-heaps permanently associated with threads. What happens if we sever that tie and turn our thinking "inside-out", as it were?

Let us explore what happens when, instead of shifting memory resources between sub-heaps, threads themselves shift between sub-heaps.

7.21 Thought Experiment #11

Let us craft TEX-11, which is an allocator that consists only of a set of common bounded sub-heaps. When a thread allocates, it allocates from the sub-heap it last allocated from. If the request cannot be serviced from that sub-heap, the thread searches for a different sub-heap, shifts to that sub-heap and then allocates from that. When a thread frees a block, it returns that block to the sub-heap it was allocated from.

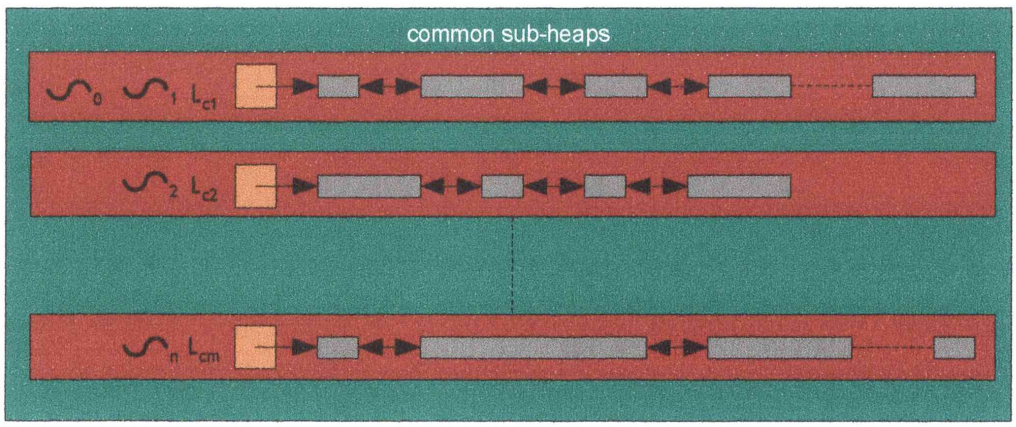


Figure 7.13: TEX-11 – a heap consisting of a set of common sub-heaps.

There are several advantages with this arrangement. Allocating from any given sub-heap is O(1) – since there is an upper bound on the size of the sub-heap. Likewise freeing is O(1). Another advantage is that, with respect to memory, the system remains balanced – no one sub-heap can end up with an excess amount of memory.

There are two main disadvantages, the principal one being that there are an indefinite number of sub-heaps. A linear search for an appropriate sub-heap is not feasible, as this leads to indefinite service times as memory approaches infinity. Secondly, any number of threads could be attached to any given sub-heap. Unless otherwise dampened, this would lead to scalability bottlenecks as we add CPU's into the model.

7.22 Critical Insight #1

Any given sub-heap will have a set of free blocks available to be allocated at any point in time. If (as in TEX-11) we limit sub-heaps based on the amount of memory they manage, then each sub-heap has a finite amount of memory available for allocation, at any one time.

A critical insight is that, the amount of memory a sub-heap has for allocation, is a property which could be indexed.

Consequently, if we can devise an indexing strategy which discovers a suitable sub-heap, in a manner which is independent of the number of sub-heaps, then we potentially have a feasible solution.

And so we discover a very large, highly concurrent dynamic memory manager could be one which manages an unbounded set of fixed sized sub-heaps, where each sub-heap is a finite set of varying sized blocks, such that finding a suitable sub-heap is O(1), and allocating from and freeing to any given sub-heap is also O(1).

The question now becomes: What is a suitable indexing technique?

7.23 Using Trees

One design option would be to use a tree structure, of some form, to construct an index of sub-heaps. The use of a balancing tree (such as a red-black tree) will typically have an access time of O(log n). While this is better than O(n) it does mean that a very large

dynamic memory manager, based on a tree of sub-heaps, will slow down as we add memory into the model.

Two other factors need to be considered. Firstly, there is a computational overhead in maintaining a tree structure in balance. Secondly, balancing a tree in a multithreaded context is complex – two or more threads may simultaneously be performing potentially conflicting rebalancing operations, at the same time while one or more threads may be attempting to read through the tree.

The resolution of these tensions would typically require techniques such as read locks, deadlock/livelock avoidance, detection and resolution, as well as scheduling issues such as priority inversion. Such complexity also conveys with it, higher degrees of difficulty of conceptualization/comprehension, code volume plus defect identification and correction – all leading to increased implementation costs.

7.24 Using Vectors

The complexity associated with high performance, finely concurrent rebalancing tree structures and algorithms, which at best deliver O(log n), provides motivation to explore an alternative solution. Let us reexamine the vector approach used by DLmalloc and see if it can be applied to indexing sub-heaps.

Firstly, recall that it is generally considered acceptable to handle very large requests by allocating virtual memory directly from the operating system [Lea2000] – simply round the requested size up to a whole number of pages. For the purposes of this discussion, assume we are interested in managing sizes in the range 0 – 1MB; therefore it is possible to index the sub-heaps, based on the largest free block, in each of the sub-heaps. For example, if a sub-heap has 100 free blocks, of which the largest is 1KB, then its index would be based on 1KB, not 100 blocks, nor the sum of the space of those 100 blocks.

7.25 Thought Experiment #12

Let us craft TEX-12, which is an allocator that tracks sub-heaps using two vectors. The first vector tracks sub-heaps which have a small block (in the range 0 – 63 units in size) as their largest free block. The second vector tracks sub-heaps which have a large block

as their largest free block. For the purposes of discussion, let us assume that the large vector represents bins which are logarithmically spaced (as in DLmalloc).

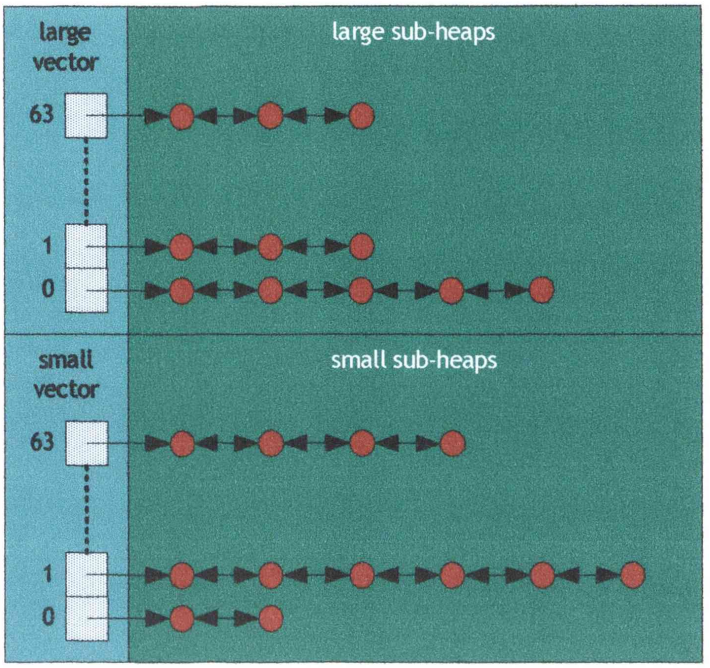


Figure 7.14: TEX-12 – tracking sub-heaps with two vectors – the smaller one is exact, while the other is logarithmically spaced.

In this arrangement, to find a small sub-heap is O(1) – irrespective of the number of sub-heaps on any given small list. The first sub-heap on any small list will suffice. The problem is the larger sizes. There is an indefinite number of sub-heaps, and so any number of sub-heaps could exist on any large sub-heap list, but because they are grouped in logarithmically spaced bins, the lists must be linearly searched for an appropriate sub-heap – leading to O(n) behavior.

7.26 Thought Experiment #13

Given that, for the purposes of this discussion, we are only interested in managing memory blocks up to 1MB in size, one possible arrangement would be to have a very large vector, covering each of the possible sizes. Let us call this arrangement TEX-13. For a 64-bit machine this would require a vector with 65,536 elements in it[108].

[108] On a terabyte machine a vector this large only represents 0.000005% of memory.

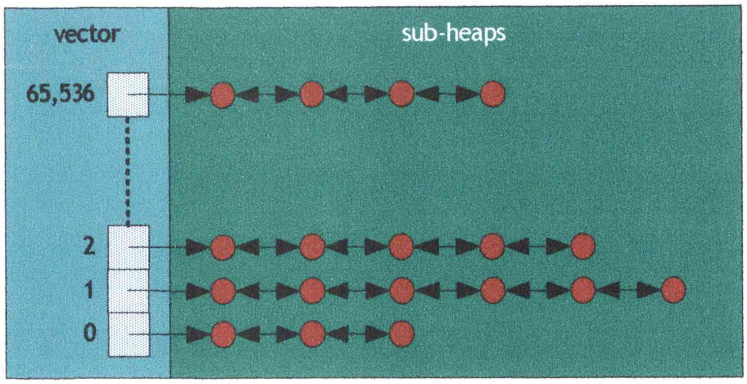


Figure 7.15: TEX-13 – a single vector of sub-heaps.

A simple implementation for finding a suitable sub-heap would simply be to inspect the corresponding element of the sub-heap vector to see if a sub-heap exists – one which has as its largest free block, the specific space required.

The main problem with this approach is that a sub-heap with a free block of the specific required size may not be available, so a sub-heap with a larger free-block has to be used – or a new sub-heap has to be created. Consequently, if a suitable sub-heap is not available, then a linear search upwards is required until the next biggest size is found, or the search fails.

This has an average upper search limit of half the vector. To search a vector in this manner would be an estimate of between 10 and 100 instructions per element. This would be between 65,536 ÷ 2 × 10 and 65,536 ÷ 2 × 100 instructions – between 327,680 and 3,276,800 instructions on a 64-bit machine. This seems somewhat large, particularly when we consider that brute force searching of this kind, also tends to lead to excessive cache invalidation.

7.27 Introducing Bit Indexes

One technique for reducing search time and cache invalidation is to employ a *bit index* – a form of bitmap used to summarize the state of the vector [IGK1971]. For example, the 64 bits of a 64-bit word, can be used to summarize whether each of 64 different words have a non-null value in them. If an index word of this kind is zero, this would indicate that all of the associated 64 words are null. Should the index word be non-zero then the bits of the index word can quickly be examined, to identify which associated words have a value [War2002].

Let us refine the brute force approach by adding a bit index. To summarize a vector with 65,536 elements would require 1,024 words. A simple bit index of this kind reduces the search time, by about the word size of the machine. Searching half of the bit index requires between 5,120 and 51,200 instructions.

Depending on the frequency of searching for sub-heaps, this may or may not be acceptable. Observe though, that the bit indexing technique can be applied to any number of levels. Another level of bit indexing consisting of 16 words (summarizing the other bit index of 1,024 words) can be searched in 80 – 800 instructions. This reduces the overall average number of instructions to between 400 and 4,000.

For the purposes of this discussion, we will assume two levels of bit indexing.

7.28 Bit Index Optimization

Note that the bits in a bit index are only changed, when the words they summarize change from null to a non-null value, and back again. Bits in a bit index are not updated, when the words they summarize change from some non-null value, to some other, non-null value.

Restating: Bits in the lowest level index are set on, for the first block that is inserted into a list. Bits are only set off, when the last block is removed from a list. The action of inserting a block into an existing list, or removing a block from a list with two or more blocks in it, does not cause the bit index to be changed.

7.29 Bit Indexes and Concurrency

A major advantage of a bit index is that the structure necessary to hold the index is static – the physical structure does not change because the data has changed. Compare this to most tree like structures, which change shape in a homologous fashion reflecting the data they contain. By not changing shape when data changes, makes concurrency significantly simpler – a bit index can be read concurrently without locking, and can be updated concurrently using light-weight test-and-set mechanisms available on all modern microprocessors [Mic2002].

7.30 Thought Experiment #14

This brings us to TEX-14, an allocator which uses a vector which explicitly covers each possible size of sub-heap, and a two level bit index for optimally searching that vector.

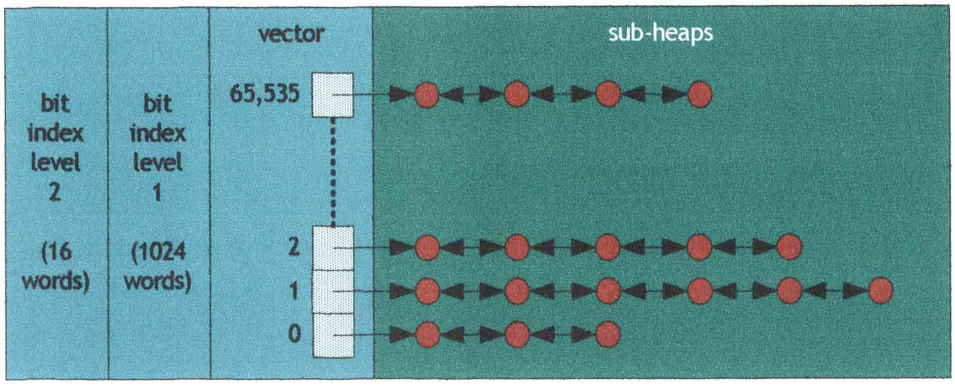


Figure 7.16: TEX-14 – a vector of sub-heaps with a two level bit index.

In this arrangement, finding a suitable sub-heap is O(1), and with bit indexing it should be capable of being done quite efficiently. Allocation from, or freeing to, any sub-heap is also O(1), as the overall capacity of a sub-heap is bounded.

TEX-14 appears to be a tenable, first order solution for a very large dynamic memory manager. Let us pursue TEX-14 in more detail.

7.30.1 Managing a List of Sub-Heaps

When a thread searches the vector of sub-heaps and finds an entry in the vector, that entry represents a set of suitable sub-heaps – there can be any number of sub-heaps in any given list. The question arises: which sub-heap in the list should a thread choose?

The simplest answer is the first one in the list. However, unless something else is done, the first sub-heap in any list will be chosen by any number of threads. Having too many threads in any one sub-heap, will lead to bottlenecks and thrashing, as threads contend for limited resources within a sub-heap.

An elegant answer to this problem is that, when a thread selects a sub-heap, the thread moves that sub-heap to the end of the list. Making the sub-heap lists circular would make this action a simple and efficient rotation.

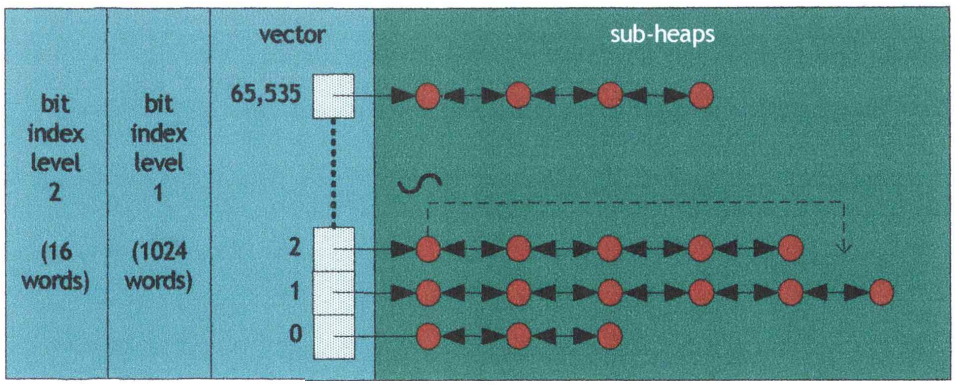


Figure 7.17: Figure depicts a thread selecting a sub-heap of size two and shifting it to the end of that list (refer dashed line).

Moving sub-heaps in this manner keeps the list of sub-heaps in FIFO order, thereby having the tendency to automatically spread threads out, over the set of sub-heaps.

7.30.2 Detecting Excessive Collisions

Even with the above list rotation mechanism, it is quite possible for any number of threads to be attempting to use the same sub-heap. In cases where those threads were allocating and freeing similar sized blocks, then bottlenecks are still possible.

The solution to this problem is to use a self-governing heuristic, which detects when threads are colliding too often, and have threads either move to a different sub-heap or create a new one.

Such a heuristic might work in the following manner:

- When there are very few sub-heaps in existence simply create a new sub-heap;
- When there are a reasonable number of sub-heaps, preferentially move to a different sub-heap, at least attempt to do so a certain number of times, before creating a new sub-heap;
- When the system is approaching memory exhaustion, always move to a different sub-heap, if one of suitable size is available; and
- If memory is exhausted, then retry a certain number of times (suitable sized blocks may become freed while trying) before reporting an error.

7.30.3 Shifting Sub-Heaps between Lists

As threads allocate and free blocks, those actions will change the largest available block in the sub-heap. Threads will therefore need to shift sub-heaps between lists, when an action changes the largest chunk of free space in a sub-heap.

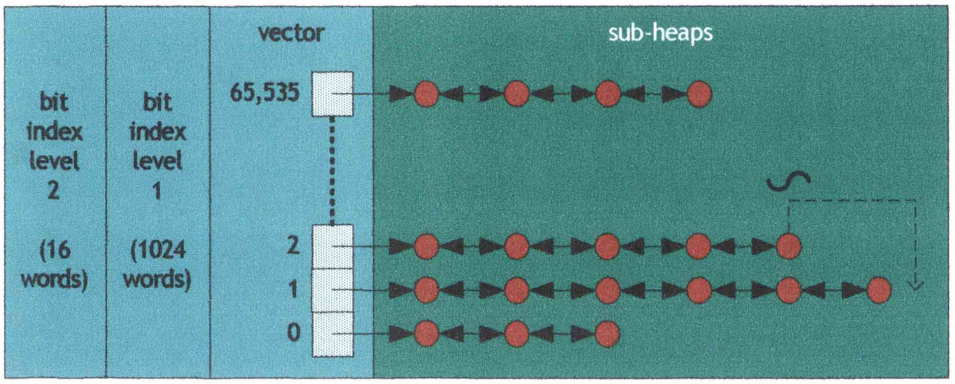


Figure 7.18: Figure depicts a thread shifting a sub-heap from list 2 to list 1 (refer dashed line).

7.30.4 Releasing Sub-Heaps

One final issue needs to be addressed at this level before we have a complete conceptual model: sub-heaps should be released (i.e. the virtual memory pages a sub-heap is managing should be returned to the operating system) when all of the blocks become free.

This enables the entire heap to grow and shrink, as needs be, over an indefinite period of time.

7.30.5 Cross-Check: Cache Efficiency

As noted in the beginning of this chapter, cache utilization is becoming increasingly more important, particularly as the difference between CPU and memory speeds continues to grow [WJNB1995]. It is essential therefore that before we formally specify and build RHmalloc, we pause and check how our proposed solution fares with cache related issues.

Cache related effects have been specifically observed by:

1. **[GZH1993]** – allocation policy has a significant effect on cache locality affecting overall execution performance by as much as 25%;
2. **[Joh1997]** – memory locations whose addresses were near one another tend to be referenced close together in time (spatial locality);
3. **[Joh1997]** – memory locations which have been accessed more recently tend to be accessed again (temporal locality);
4. **[BBDT1984, GZH1993]** – cache is invalidated by searching;
5. **[Boz1984]** – maintaining summarized information can substantially reduce searching overheads and consequently improves cache utilization; and
6. **[Ber2002]** – false sharing, which can lead to cache thrashing, can be minimized by having distinct threads allocate from different memory regions.

More general, memory efficiency related effects have been observed by:

1. **[Nie1977]** – algorithms which maintain separate free lists for each size tend to perform quite well compared to other algorithms;
2. **[Nie1977]** – algorithms employing power-of-two block sizes have favorable processing times but generally unfavorable memory usage;
3. **[Col1961]** – best fit tends to give the best overall results followed by first fit;
4. **[BZ1993]** – the majority of blocks tend to be short lived;
5. **[BJW1970, BB1977]** – the majority of blocks sizes tend to be small;
6. **[Sho1975, BZ1993, Joh1997]** – blocks allocated at the same time tend to get freed at the same time;
7. **[Bec1982]** – fragmentation is caused by (or increased) by differences in release times between adjacent blocks;
8. **[Sho1975]** – blocks at the end of lists tend to age, coalescing over time to form much larger free blocks (thereby reducing fragmentation);

9. **[Joh1997, LK1998]** – allocators which immediately coalesce memory and preferentially reallocate blocks that have been freed most recently tend to perform better; and

10. **[LK1998, BMBW2000]** – allocators which permanently tie subsections of memory to threads or CPU's suffer from simple pathological cases which lead to blowup.

An important point made by Lea [Lea2000] is that the vector of lists mechanism (as used by DLmalloc) approximates best fit – which is one of the better known all-round policies with respect to memory efficiency. The overall efficiency of this mechanism was first observed by Nielson [Nie1977], and Johnstone specifically confirmed the memory (fragmentation) and cache efficiency of best fit LIFO, and best fit FIFO, in his exhaustive study [Joh1977]. Refer §4.4.16 for his results.

We therefore expect RHmalloc to be CPU, memory, cache and bus efficient for the following reasons:

- RHmalloc tracks sub-heaps via a vector of circular lists. Threads select the first sub-heap in a list, and then rotate the list. This has the effect of dispersing threads over the available sub-heaps. This minimizes the chance of thread contention, as well the chance of sub-heaps being co-resident in multiple caches;

- Many operating system schedulers, such as the Linux O(1) Scheduler, support soft CPU-affinity where, by default, they try and schedule a thread back onto the CPU it was last executing on [Lov2003]. RHmalloc records the sub-heap last used by a thread (using thread local storage) and attempts to satisfy a request using that sub-heap before searching for a different one. This has the consequence that the bitmap and vector pertinent to a sub-heap may still be in the cache of the CPU assigned to a thread for its next time-slice – probably even more so when two or more threads assigned to the same CPU happen to be sharing the same sub-heap;

- The vector of LIFO lists arrangement we have chosen, for managing the blocks within sub-heaps, is a very similar approach to DLmalloc – and is specifically

not a power-of-two arrangement. Best fit LIFO tends to have low fragmentation, preferentially allocating blocks that have just been freed;

- Best fit LIFO also generates few cache misses, reusing blocks just freed. List searching does not occur at all for the exact sized lists, and because the sub-heap is finite, we expect few large blocks in the inexact lists;
- A sub-heap is a bounded region or interval of memory. Blocks allocated from the same sub-heap are close to each other address wise, which supports spatial locality. This has the effect of optimizing cache utilization;
- RHmalloc should minimize false sharing. Threads tend to be scheduled onto to distinct sub-heaps, and then allocate from that sub-heap as long as possible, before shifting to a different one. A thread, therefore, will tend to allocate its blocks from contiguous memory regions separate to that of other threads;
- Bozman [Boz1984] observed that by holding condensed information, searching overheads could be reduced (including a reduction in cache invalidation) in some cases by as much as 80+%. Since RHmalloc uses bitmaps to concentrate data, so as to minimize vector searching, we can expect a commensurate level of optimization;
- RHmalloc also uses the same bitmapped, best-fit-approximating vector approach in tracking the sub-heaps, but the lists are in FIFO order. Best fit FIFO was the best overall approach of the 53 Johnstone tested; and
- In RHmalloc sub-heaps and threads are not permanently tied together. If there is available memory, then threads will find the pertinent sub-heaps. RHmalloc only extends memory, given there is available memory, when there is unabated thread contention for that memory. Such contention tends to indicate that said available memory is likely to become unavailable in the very near short term. RHmalloc should not exhibit blowup.

7.30.6 Visualizing RHmalloc on an SMP Machine

The following diagram shows how RHmalloc, as an abstract design, maps over our abstract model of a scalable SMP machine. The stylized diagram shows that:

1. Sub-heaps are primarily in memory;
2. Some sub-heaps are in cache (either partially or fully);
3. Some parts of sub-heaps are in the registers of the CPUs;
4. Because of operating system level, soft thread-to-CPU affinity and RHmalloc level, thread to sub-heap binding, threads tend to be rescheduled onto the processor they were last executing and consequently the contents of that CPU's cache will tend to be relevant;
5. Because of collision avoidance, threads tend to stay away from each other and therefore tend not to share data across multiple caches; but
6. There is still a chance, albeit diminished, that two or more threads may share the same sub-heap, and those threads may simultaneously be on different processors – and consequently in different caches.

RHmalloc is independent of the number of CPU's or amount of memory in a machine.

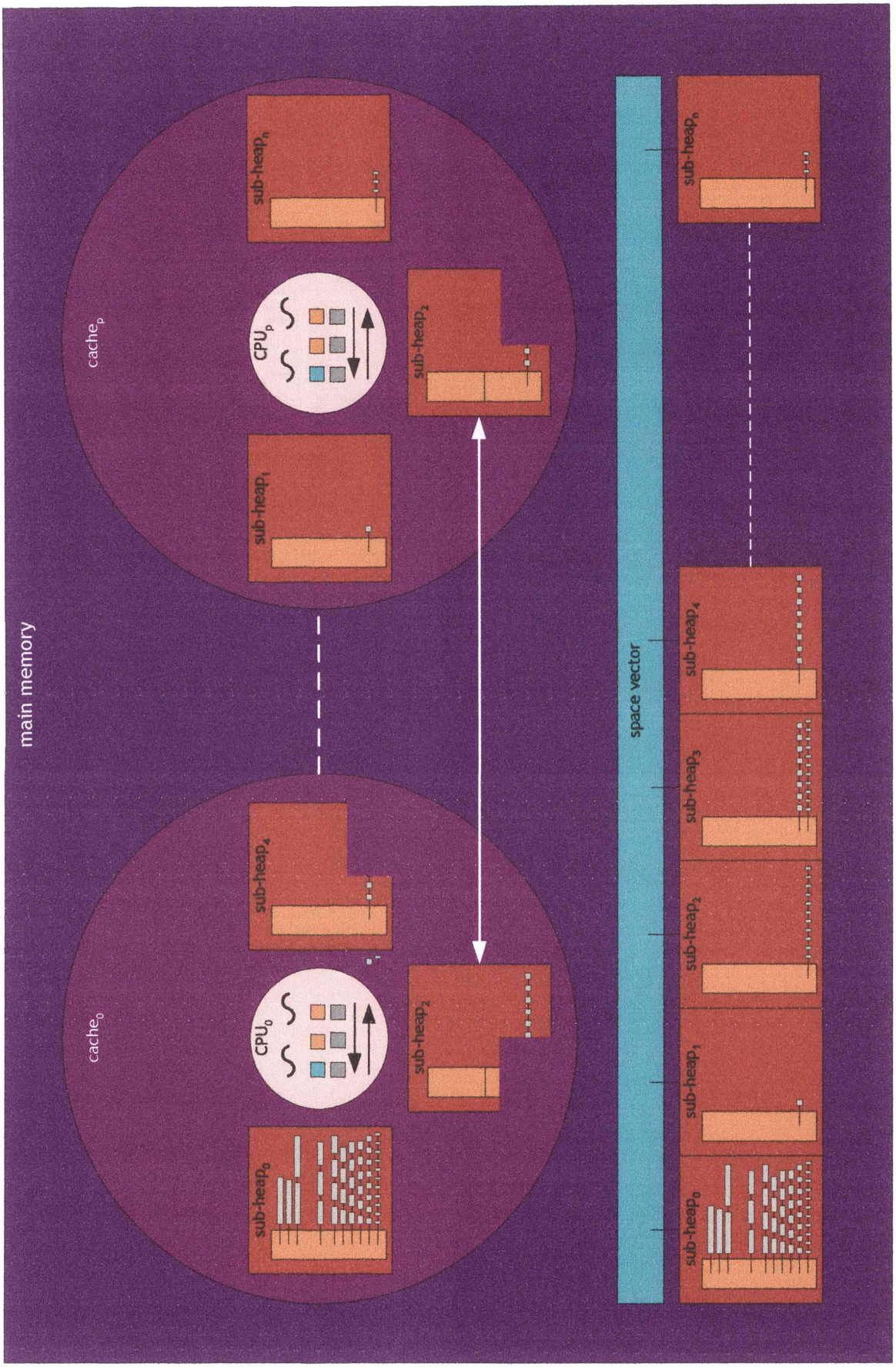


Figure 7.19: A stylized depiction of how RHmalloc maps over an SMP machine.

In summary, RHmalloc is an SMP CPU, memory and cache optimized heap of sub-heaps of blocks: a bitmapped best fit FIFO vector of bitmapped best fit LIFO vectors of regionalized blocks, coupled with thread-to-cache affinity and collision avoidance scheduling. RHmalloc uses two of the three best memory and cache efficient choices from the 53 specifically explored by Johnstone [Joh1997], and supported by the observations of at least 12 other independent researchers [BB1977, BBDT1984, Bec1982, Ber2002, BJW1970, Boz1984, BZ1993, Col1961, GZH1993, LK1998, Nie1977, Sho1975] from over the last 40+ years.

RHmalloc should not only scale indefinitely with respect to CPU and memory, it is likely to be one of the most efficient ways to do so – particularly with respect to cache utilization. If CPU and memory speeds continue to diverge, then this aspect of RHmalloc is an inherently valuable property.

This substantiates our philosophy, as stated at the beginning of this chapter, that prior work would provide valuable insights into a complex topic, and that the solution to large scale DMM would likely be an 80% utilization of existing techniques.

7.31 Summary: A First Order Sketch

At this stage, we have a rough sketch for a workable very large, highly concurrent dynamic memory manager. The following diagram provides a simplified depiction of RHmalloc.

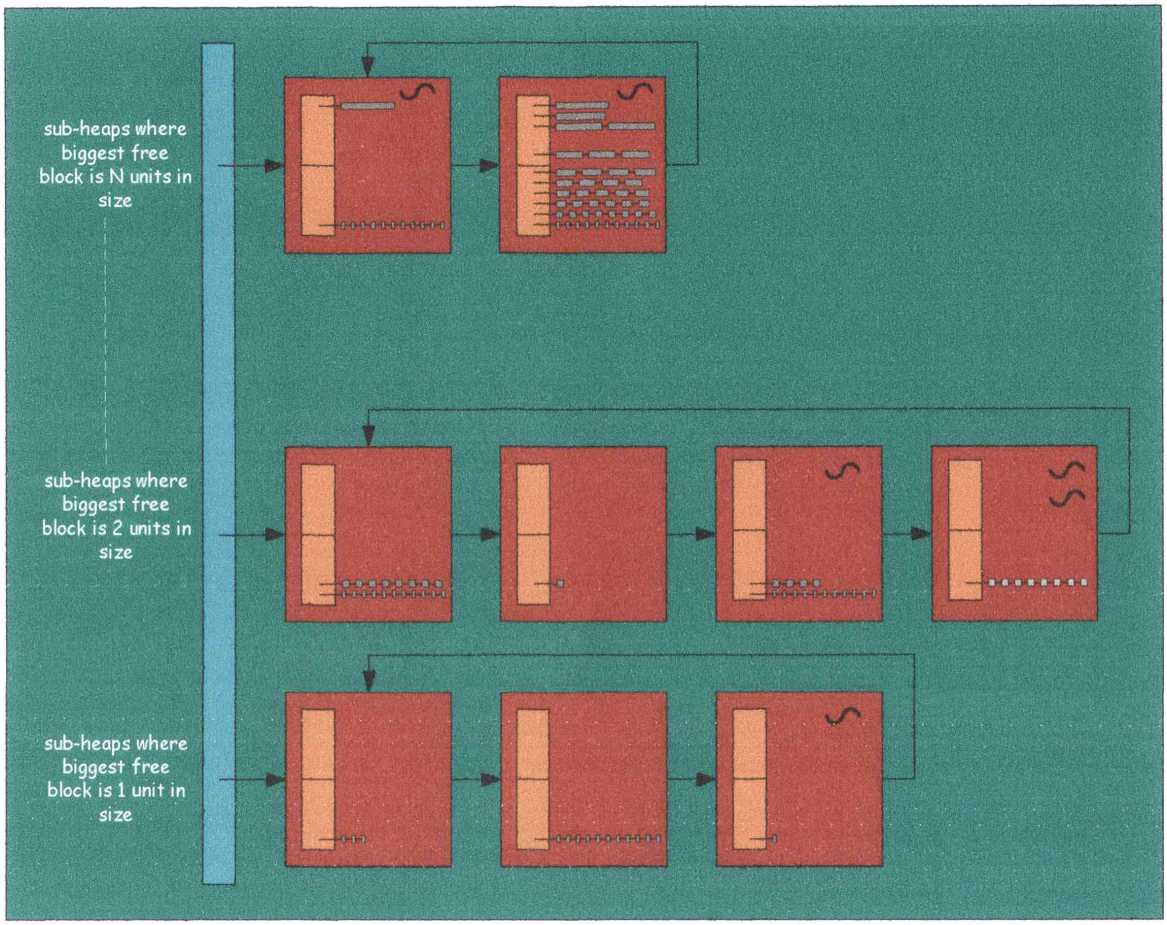


Figure 7.20: A first-order sketch of RHmalloc.

RHmalloc can be described in the following manner:

1. A very large heap is viewed as an unbounded set of sub-heaps;
2. The allocator manages blocks up to particular limit (e.g. 1MB), above which it is requested from the operating system;
3. A sub-heap is a bounded region of virtual memory containing a bounded number of free blocks;

4. Sub-heaps are tracked using as a single vector of FIFO ordered lists called the sub-heap vector;
5. Sub-heaps are always in one of the sub-heap vector lists, based on the size of the largest free block they contain;
6. As a performance optimization, there is a bit index over the sub-heap vector indicating which lists of particular sizes exist or not;
7. Threads attempt to allocate from the sub-heap they last allocated from, until that sub-heap cannot be used to satisfy a particular request;
8. Threads select a different sub-heap by inspecting the sub-heap vector, looking for a sub-heap with a large enough free block to satisfy the current request;
9. Upon selecting a sub-heap, a thread moves the sub-heap to the end of its list, in order to minimize the probability that two or more threads will be allocating from the same sub-heap;
10. Sub-heaps are shifted between sub-heap lists when the size of the largest free block changes (due to allocation or freeing);
11. A sub-heap manages free blocks as two vectors of lists – one for small (exact) sizes and one for larger sizes;
12. The strategy for minimizing fragmentation is to recognize there typically are regularities in the request stream which can readily be exploited by a best fit LIFO policy [Joh1997];
13. Threads shift to a different sub-heap, or create a new sub-heap, based on a heuristic which weighs thread collision against memory fullness;
14. Freed blocks are returned to the sub-heap they were allocated from; and
15. A sub-heap is freed (thereby releasing the virtual memory it represents) when all blocks it is managing become free.

In the next chapter we present a formal specification for RHmalloc – a very large, highly concurrent dynamic memory manager. This includes graphical representations of the data structures as well as pseudo-code for the main algorithms.

8 Formal Specification

This chapter presents the formal specification of RHmalloc: a very large, highly concurrent dynamic memory manager.

8.1 Data Structures

We begin our formal specification, by stating the first order view of how RHmalloc relates to its principal resource – memory.

Our definition of an indefinite heap is an unbounded set of finite sized sub-heaps. Two fundamental data structures are immediately evident: the heap and the sub-heap. The heap structure is a control block, which manages the list of sub-heaps which constitute the heap at any point in time.

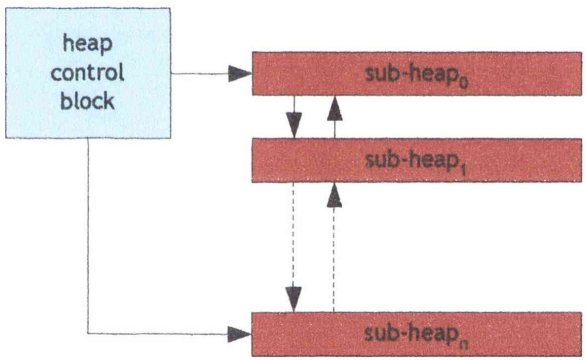


Figure 8.1: Heap control block manages the set of sub-heaps.

RHmalloc maintains the list of sub-heaps in address order.

8.1.1 The Sub-Heap

8.1.1.1 Sub-Heap Numbering

The sub-heap is the foundation by which RHmalloc manages the block-wise allocation and de-allocation of dynamic memory. Sub-heaps are uniquely identified by their *sub-heap number*, which usually starts from zero.

8.1.1.2 Virtual Memory Mapping

A sub-heap is a region of virtual memory, from which blocks of dynamic memory are allocated and freed in a piecemeal fashion. Sub-heaps functionally map over virtual memory ranges in a non-overlapping fashion. Sub-heap$_0$, for example, always maps over the same virtual memory address range – the first address range available given the architecture of the machine. Sub-heap$_1$ maps over the virtual memory range which borders sub-heap$_0$ – the address range after sub-heap$_0$ and before sub-heap$_2$.

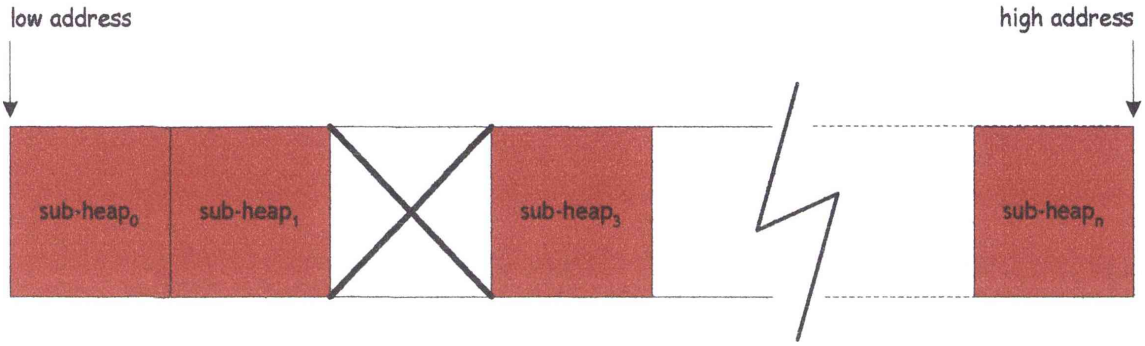


Figure 8.2: Sub-heaps map onto non-overlapping virtual memory ranges.

8.1.1.3 Virtual Memory Clash

It is possible that some virtual memory addresses are not available, because devices or other libraries directly allocate virtual memory from the operating system, consuming part or all of that range. In this case, the sub-heap which would clash with any such addresses cannot be created, and consequently does not appear in the address-ordered list of sub-heaps.

The previous diagram shows an example where sub-heap$_2$ cannot be created due to such a clash.

8.1.1.4 Virtual Memory Management

Virtual memory pages are requested from the operating system, as part of the act of creating a sub-heap. RHmalloc destroys sub-heaps when all blocks within a sub-heap are freed and returns the virtual memory pages back to the operating system. Consequently, any given subset of sub-heaps may exist at any point in time.

8.1.1.5 Bounded Range of Virtual Memory

In the following diagram, we show that a sub-heap only manages the blocks relating to a particular region of virtual memory. Consequently, all of the blocks in a given sub-heap are in the same address range. Observe that as we add memory into this model, we create new sub-heaps – we do not extend existing sub-heaps. The amount of memory represented by any given sub-heap is finite.

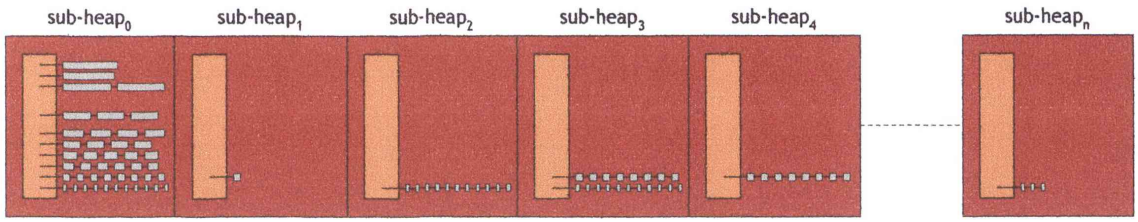


Figure 8.3: An unbounded heap viewed as an indefinite set of finite sub-heaps.

8.1.1.6 Sub-Heap Size

The amount of memory to be managed by a sub-heap is an interesting choice. If sub-heaps are too small then the allocator may tend to request and return virtual memory from the operating system too frequently – consuming CPU and thereby reducing throughput. Too large and other inefficiencies may emerge: threads can spend too long searching through free lists, consequently thread collisions would tend to increase, thereby reducing throughput.

If the size of a sub-heap was 16MB then a 1TB virtual address space would be managed by 65,536 sub-heaps. This seems a reasonable first-order estimate. One option is to make the sub-heap size a configurable setting.

8.1.1.7 Sub-Heap Layout

Each sub-heap reserves a small amount of space for control information and the remainder is deemed *block space* available for allocation.

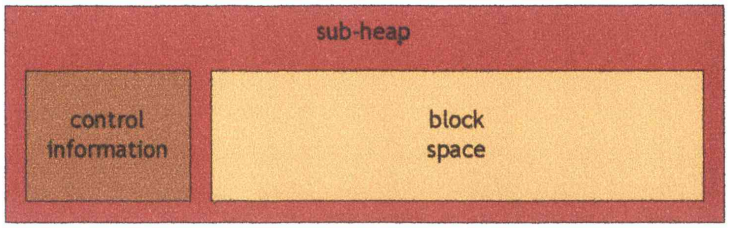


Figure 8.4: Sub-heaps are a structure containing information used to control allocation from the remaining space in the structure.

8.1.1.8 Free List Vector

Within the sub-heap control structure, there is a vector which holds a set of pointers to doubly-linked lists of free blocks in the block space area. This vector is known as the *free list vector*. The size of the free list vector reflects the word size of the machine.

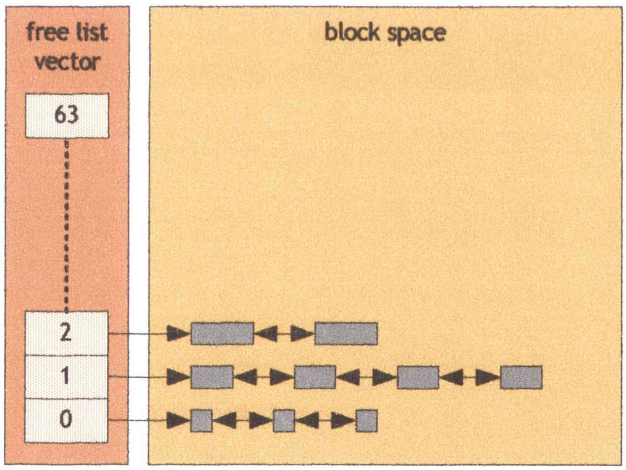


Figure 8.5: The free list vector.

On a 64-bit machine the free list vector would hold 64 pointers; on a 256-bit machine the free list vector would hold 256 pointers. The free lists are maintained in LIFO order, so as to minimize fragmentation [Joh1997].

As described below, RHmalloc only manages the allocation of block sizes up to a particular limit. Requests for blocks larger than that limit will be satisfied by directly requesting virtual memory pages from the operating system.

8.1.1.9 Free Bitmap

At any given point in time there may, or may not be, lists of blocks of a particular size. Therefore, a request for a particular size may need to be satisfied by using a larger free block and splitting it. In order to minimize searching, the free list vector is summarized by a one word bit vector, called the *free bitmap*.

Each bit in the free bitmap corresponds to an element in the free list vector. For example, bit two in the free bitmap would be on, if there was a free list of blocks of size two in the free list vector; the bit would otherwise be off.

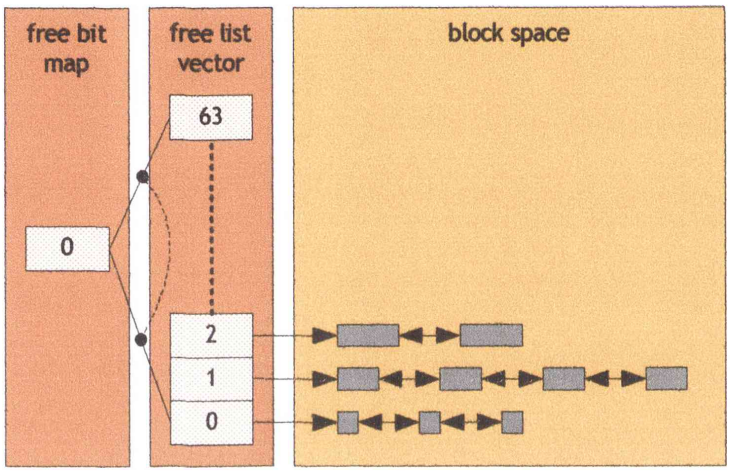


Figure 8.6: The free bitmap.

The choice of having the length of the free list vector reflect the word size is deliberate, as it enables the free list vector to be summarized by a single word. Searching for free lists is then a matter of inspecting the bit pattern in the free bit map – which can be done quite efficiently using standard bit manipulation techniques [War2002].

Being a single word, the free bit map can be safely updated using atomic compare and swap instructions – which are available on all modern microprocessors [Mic2002].

8.1.1.10 Free Lock Vector

Sub-heaps are multithreaded. In order to achieve thread safety, the free lists are guarded by a vector of spin-locks called the *free lock vector*. Threads must acquire the spin-lock corresponding to the free list, before they access that list, and then release the spin-lock immediately afterwards.

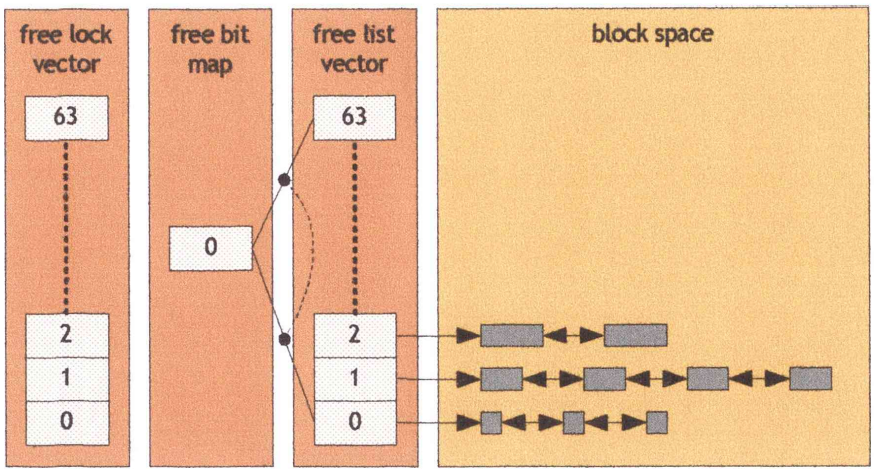


Figure 8.7: The free lock vector.

8.1.1.11 Exact v. Banded Free Size Lists

The length of the free list vector reflects the word size of the machine. Consequently, there are a small number of possible lists, which must cover the range of possible blocks sizes – which for the first version will be 0 through to 1MB[109].

The set of lists are notionally broken into two halves. The lists in the lower half are called the *exact size free lists*; while the upper half are called the *banded size free lists*. Blocks that are in an exact size free list are all the same size; blocks in a particular banded size free list are within a particular size range.

[109] A 1MB limit seems reasonable for a 64-bit machine. This may need to be reexamined for 128-bit machines and larger.

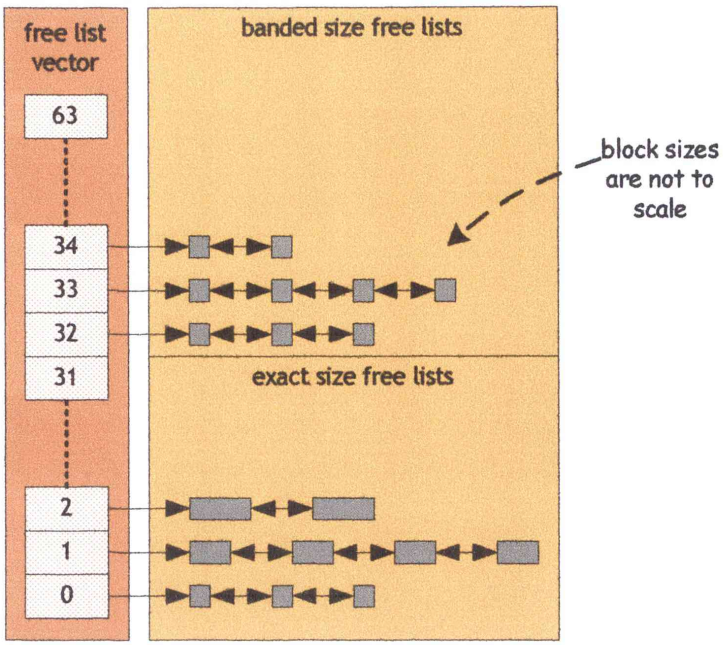


Figure 8.8: Banded size versus exact size free lists.

8.1.1.12 Block Sizes

Blocks will always be allocated as a multiple of two words, due to alignment requirements. The managed range of 0 through to 1MB, therefore, yields 65,536 discrete block sizes on a 64-bit machine. The exact size free lists cover the range 0 through to 4096 bytes, while the banded size free lists are evenly distributed across the remainder.

Given a sub-heap of 16MB on a 64-bit machine there would be, at worst case, of the order of 32,768 blocks in the smallest of the banded free lists, with over 2,048 different possible sizes. In practice, the probability of having such a list occur, and having to search it entirely, is very low. This should be capable of being observed during verification.

8.1.1.13 Complete View of Sub-Heap

Having discussed the various structural elements, we can now present an entire picture of an RHmalloc sub-heap.

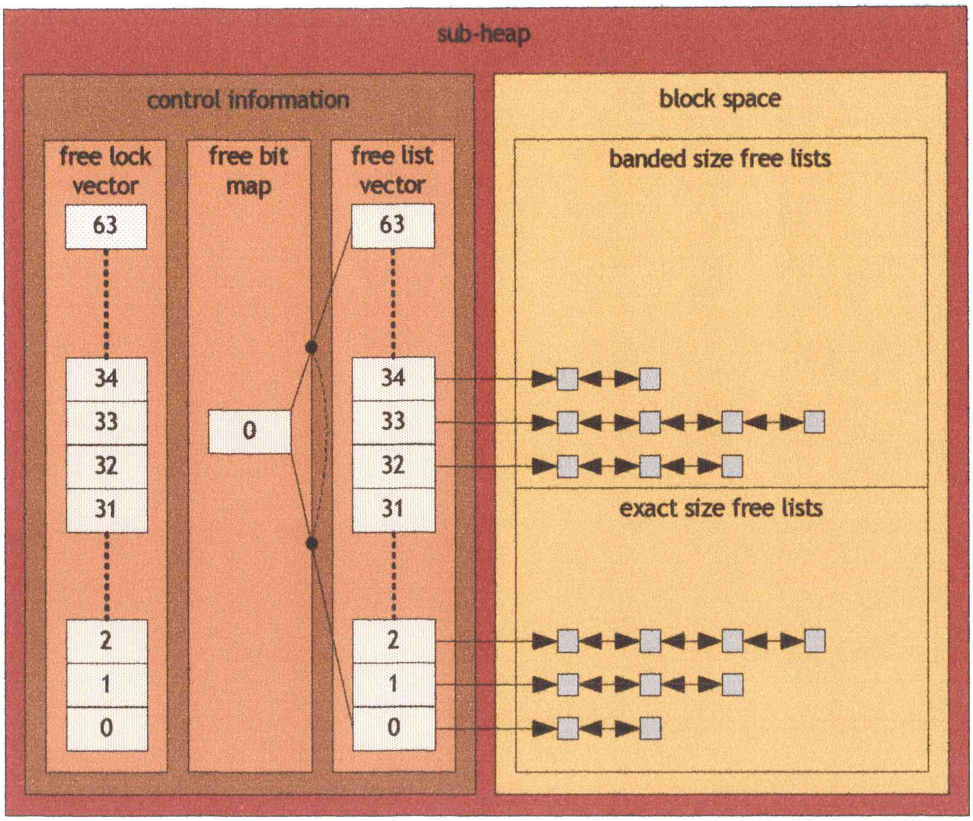


Figure 8.9: The RHmalloc sub-heap (block sizes not drawn to scale).

8.1.2 The Heap

8.1.2.1 Sub-Heap Management

An application thread will attempt to allocate blocks from a given sub-heap, until that sub-heap can no longer be used to satisfy a given request[110]. At which point, the application thread searches for a different sub-heap, one with sufficient space that can be used to satisfy the allocation request. If no such sub-heap exists at that point, then a new sub-heap is created.

8.1.2.2 Service Capacity

At any point in time, a given sub-heap may, or may not, have the capacity to satisfy a particular request. Therefore, one way of viewing a sub-heap is its instantaneous

[110] This does not mean the sub-heap is completely exhausted. There may be any number of smaller sized free blocks in the sub-heap – they just are not big enough to satisfy the request at hand.

capacity to service requests. Rephrasing: "What is the largest block that could be allocated from this sub-heap at this point in time?"

Free space is not fluid, so knowing the total amount of free space is less important than knowing the largest serviceable request. Since larger blocks can be split to service smaller requests, knowing the largest serviceable request is the property, or quality by which, RHmalloc indexes sub-heaps.

8.1.2.3 Space List Vector

In order for RHmalloc to scale indefinitely, a suitable sub-heap must be capable of being found in a manner which is not affected by the number of sub-heaps within the heap at any point in time[111]. RHmalloc maintains a *space list vector* – a vector which organizes the sub-heaps into lists, according to the available space they have.

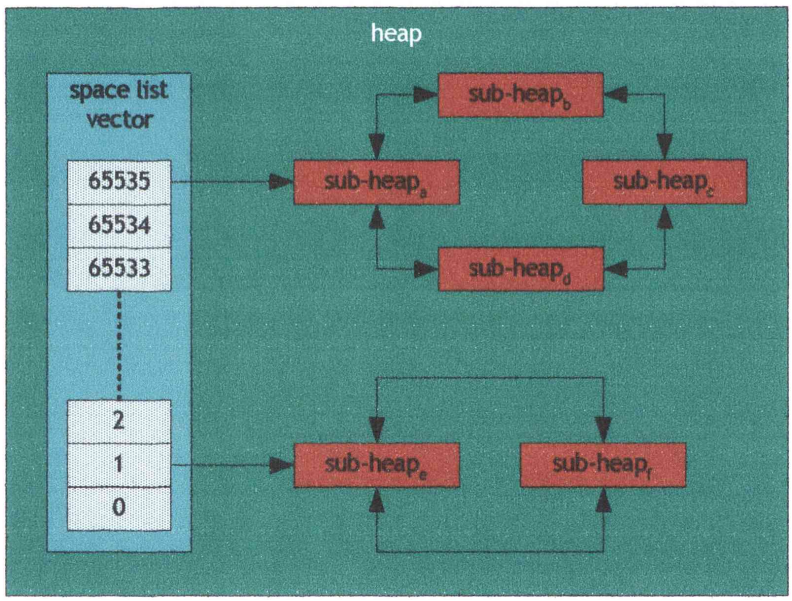


Figure 8.10: Sub-heaps organized by available space.

The length of the space list vector is SIZELIMIT ÷ (2 × BYTESPERWORD). This means for example, that in a 64-bit environment a space list vector manages 65,536 discrete sizes. Each space list is a circular doubly-linked list of sub-heaps. The presence of a sub-heap in a particular space list indicates that the sub-heap has the capacity to service a request, corresponding to the size represented by the list.

---

[111] This is a good "working definition" of *constant time*.

8.1.2.4 Space Bitmap

When a thread searches for a new sub-heap, it is in fact looking for a sub-heap, which has a serviceable space that is greater than, or equal to, the size of the allocation request. A simple brute force search, by iterating through the vector is feasible but, as discussed in §7.26, a more CPU and cache efficient technique is to use a bit index. In RHmalloc this bit index is called the *space bitmap*.

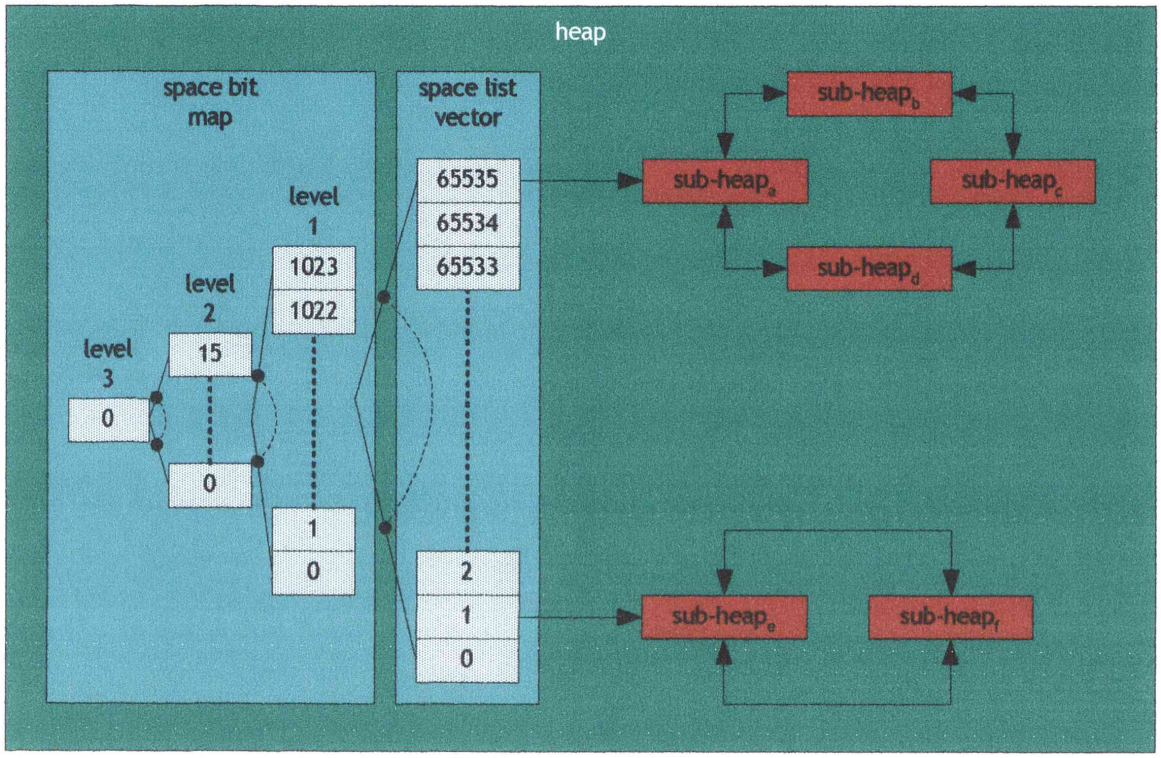


Figure 8.11: The space bitmap.

As discussed in §7.27 the space bitmap is a multileveled bit vector. The first level is a multiword bit vector which summarizes the space list vector. The length (in words) of the first level will be (SIZELIMIT ÷ (2 × BYTESPERWORD) ÷ BITSPERWORD). The levels above the first level summarize the levels underneath them.

For the first version of RHmalloc, a single level space bitmap is considered acceptable.

8.1.2.5 Space Lock Vector

The spaces lists are guarded by a vector of spin-locks, called the *space lock vector*. Threads must acquire the spin-lock, corresponding to the space list, before they access that list, and then release the spin-lock immediately afterwards.

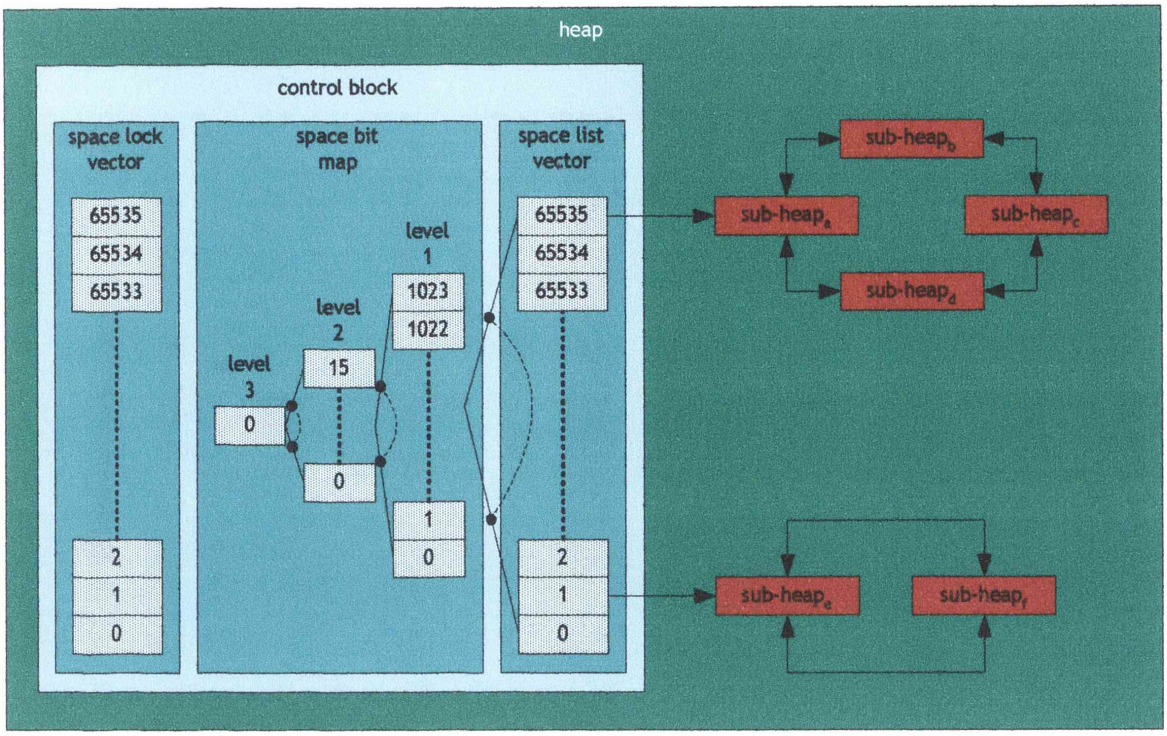


Figure 8.12: The space lock vector.

8.1.2.6 Rotating Space Lists

Sub-heaps are always in one of the space lists. Threads rotate a space list after selecting a sub-heap to satisfy the current request[112]. This is done in order to minimize the probability that, two or more threads choose the same sub-heap at similar times (and therefore minimize the probability of thread collision). This also allows sub-heaps not actively being used by threads to age, thereby increasing the probability, that blocks being freed, will coalesce into larger amounts of free space.

Being a circular doubly-linked list, rotating a space list is simply a matter of changing the pointer, in the space list vector, to reference the next sub-heap in the list.

[112] Or create a new sub-heap, if no sub-heap can be found to satisfy the request.

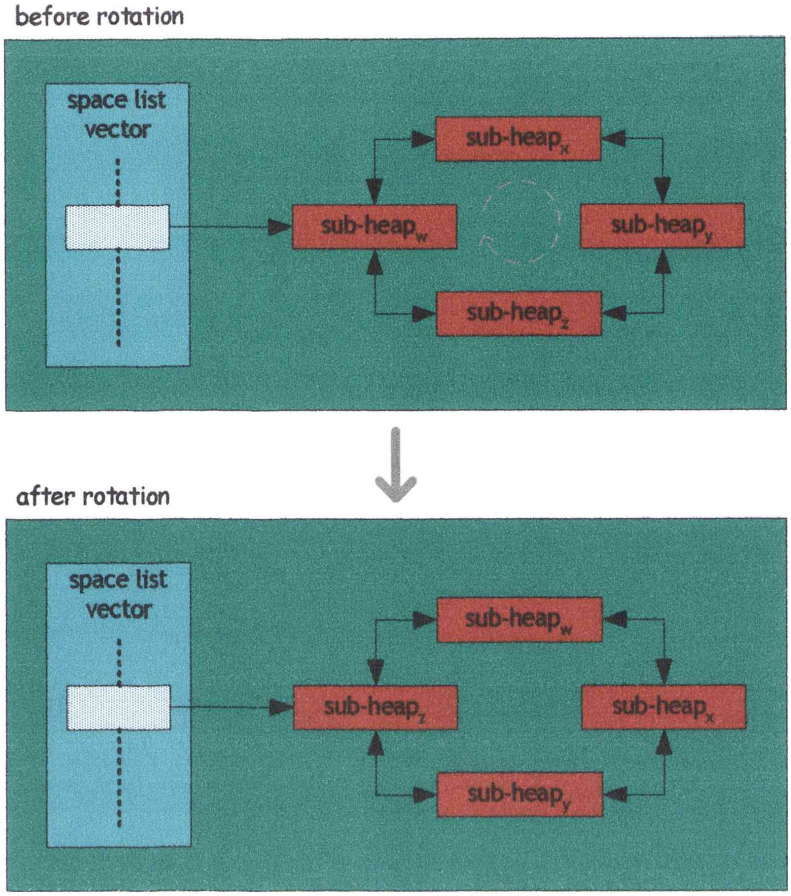


Figure 8.13: Rotating a space list.

8.1.2.7 Shifting Sub-Heaps

By course of allocating and freeing, any given sub-heap's spatial serviceability changes, on a moment by moment basis. Consequently, when a sub-heap's serviceability does change, the sub-heap must be shifted within the space list vector.

In the case of allocation, when a thread is unable to satisfy an allocation request, that thread shifts the sub-heap it was using, to the appropriate position in the space list. The thread then proceeds to find a different sub-heap which can satisfy the request.

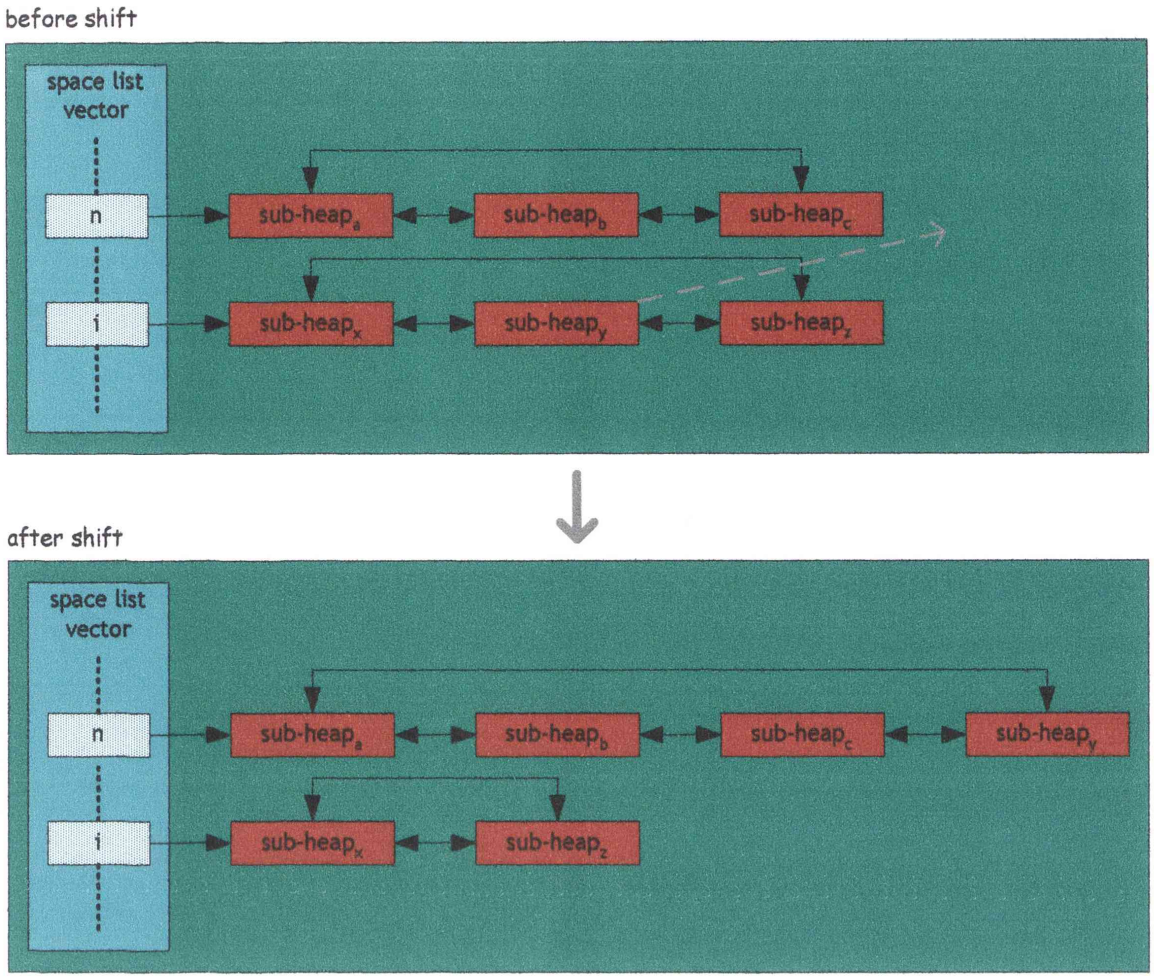


Figure 8.14: Shifting a sub-heap within the space lists.

A thread which frees a block checks whether the final, fully coalesced block being returned to the sub-heap, changes the sub-heap's serviceability; and if it does, the thread promotes the sub-heap within the space vector.

Such a regime is balanced in its actions, in that, when a sub-heap is being reduced via allocation, it is progressively demoted within the space vector, and when it is being rejuvenated via freeing, it is progressively promoted within the space vector. The regime is also efficient, by virtue of the fact that, demotion only occurs at service failure, and promotion only occurs when serviceability improves – and not on every request. Additionally, both tests are efficient/lightweight tests.

The following diagram is a stylized depiction, of a single sub-heap being shifted within the space list vector; downwards as blocks are allocated from the sub-heap, and upwards as blocks are freed back to the sub-heap.

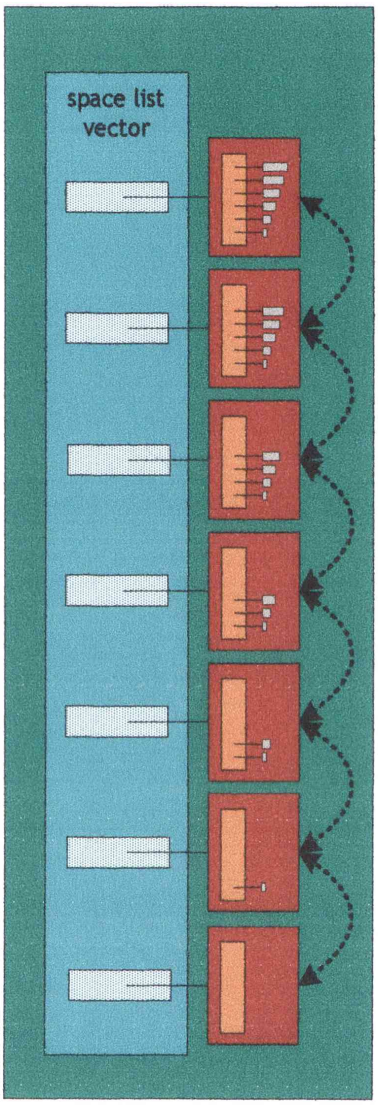


Figure 8.15: Demoting and promoting a sub-heap within the space list vector.

8.2 Algorithms

We present the main algorithms for RHmalloc. Please note that RHmalloc uses list level locking – which can be observed in Pop Chunk, Push Chunk, Find Sub-Heap and Shift Sub-Heap.

8.2.1 Allocate Block

block ← malloc(bytes)

```
1.  get sub-heap last used by calling thread
2.  calculate block size from bytes
3.  repeat following attempt a limited number of times
4.      foreach list in sub-heap of size >= block size
5.          attempt to pop first chunk off list
6.          if chunk popped
7.              if chunk too large
8.                  split chunk into two
9.                  push second chunk onto appropriate list
10.             end if
11.             calculate block address from first chunk
12.             return block address
13.         end if
14.     end foreach
15.     shift current sub-heap to appropriate heap list
16.     find different sub-heap
17. end repeat
18. return failure
```

8.2.2 Free Block

free (block)

1. calculate chunk address from block address
2. calculate sub-heap address from chunk address
3. attempt to coalesce chunk with neighbors
4. calculate list index based on size of chunk
5. push chunk onto list within sub-heap
6. if sub-heap completely empty
7. attempt to destroy sub-heap
8. if sub-heap destroyed
9. return
10. end if
11. end if
12. if chunk larger than previous largest chunk
13. shift sub-heap in heap
14. end if

8.2.3 Pop Chunk

chunk ← popChunk (sub-heap, index)

1. calculate address of list lock using list index
2. acquire list lock
3. if list empty
4. release lock
5. return failure
6. end if
7. remove first chunk from list
8. if list now empty
9. turn off bit index entry for list
10. end if
11. release list lock
12. return first chunk

8.2.4 Push Chunk

pushChunk (sub-heap, chunk)

1. calculate list index based on chunk size
2. calculate address of list lock using list index
3. acquire list lock
4. mark chunk as being free
5. remember if list is currently empty
6. push chunk onto list
7. if list was empty
8. turn on bit index entry for list
9. end if
10. release lock

8.2.5 Split Chunk

splitChunk (sub-heap, chunk, units)

1. calculate address of split-off chunk
2. create split-off chunk
3. update size of previous chunk in chunk after split-off chunk
4. update size of previous chunk in split-off chunk
5. update size of original chunk
6. push split-off chunk into sub-heap

8.2.6 Coalesce Chunks

chunk ← coalesceChunks (sub-heap, chunk)

```
1.  # first, try and merge with next chunk
2.  calculate address of next chunk
3.  if next chunk is free
4.      attempt to remove next chunk from its list
5.      if next chunk was removed from list
6.          calculate size of merge with next chunk
7.          update previous size in chunk after next chunk
8.          update size in chunk to reflect merge
9.      end if
10. end if
11. # second, try and merge with previous chunk
12. calculate address of previous chunk
13. if previous chunk is free
14.     attempt to remove previous chunk from its list
15.     if previous chunk was removed from its list
16.         calculate size of merge with previous chunk
17.         update previous size in next chunk
18.         update size in chunk to reflect merge
19.     end if
20. end if
21. return resultant chunk
```

8.2.7 Find Sub-heap

```
sub-heap ← findSub-heap (current sub-heap, units)
1.  repeat following attempt a limited number of times
2.      foreach heap list with sub-heaps of size >= units
3.          acquire list lock
4.          if list empty or list only contains current sub-heap
5.              release list lock
6.              continue next list
7.          end if
8.          if head of list is current sub-heap
9.              rotate list so head becomes tail
10.         end if
11.         new sub-heap is head of list
12.         rotate list so head becomes tail
13.         release list lock
14.         break repeat
15.     end foreach
16. end repeat
17. if no new sub-heap found
18.     if memory exhausted
19.         return error
20.     else
21.         create new sub-heap
22.     end if
23. end if
24. record sub-heap against calling thread
25. reset number of collisions for calling thread
26. return sub-heap
```

8.2.8 Shift Sub-heap

shiftSub-heap (sub-heap)

1. acquire lock on sub-heap
2. acquire list lock
3. remove sub-heap from its size list
4. release list lock
5. calculate new heap list for sub-heap based on largest free block
6. acquire list lock
7. push sub-heap onto new heap list
8. release list lock
9. release lock on sub-heap

8.2.9 Create Sub-heap

sub-heap ← createSub-heap ( )

1. calculate virtual memory address for sub-heap
2. reserve and commit virtual memory for sub-heap
3. generate id for sub-heap
4. calculate boundary of sub-heap
5. initialize vector of lists
6. initialize vector of locks
7. initialize bit index
8. convert free space into chunk
9. push chunk into sub-heap
10. add sub-heap to heap
11. calculate sub-heap size
12. push sub-heap onto space list
13. return sub-heap

8.2.10 Destroy Sub-heap

sub-heap ← destroySub-heap ( )

1. detach sub-heap from heap
2. release virtual memory represented by sub-heap

8.2.11 Initialize Heap

initializeHeap ( )

1. initialize the heap lock
2. allocate thread local storage indexes
3. initialize the heap list and lock vectors
4. create first sub-heap and put into heap
5. attach calling thread to first sub-heap

The next chapter describes our implementation and testing of RHmalloc, tables our empirical and theoretical results proving the scalability of dynamic memory management to indefinitely large, highly concurrent environments.

Chapter 9

9 Proving Scalability

We are now in a position to use RHmalloc to study scalability of heaps in practice. Our motivation comes from the observation that the fifty year research into DMM has not taken memory capacity into account, and has fallen significantly behind hardware trends. Very little work on scalable DMM has been done – of the order of ten papers exist on this topic – all of which focus on CPU scalability. The largest heap reported in the literature to date is only 600MB, yet SMP machines with terabytes of real memory are now commercially available. Our objective is to formally determine if dynamic memory management scales with respect to both CPU and memory – or to identify any limits if it does not.

We postulate that any well formed, very large, highly scalable dynamic memory manager is likely to have mathematically provable response time related properties – which are independent of the number of CPU's, and the amount of memory in a machine. We identify those properties in our dynamic memory manager, and consequently propose and conduct a series of tests, which systematically examines response time behavior as we vary both CPU and memory.

Our systematic technique, coupled with the empirical evidence from these tests, enable us to claim practical scalability to at least 1PB. Further, our approach – which we have termed "extrapolation" – enables us to show theoretically that dynamic memory management can scale indefinitely, with respect to both CPU and memory.

We also discover a fourth maxim of scalability, as well as argue that the rate of block coalescence is a potential measure of entropy in a heap.

9.1 Discussion

We designed RHmalloc to explore heap scalability. Our original goal was to demonstrate scalability of dynamic memory management up to one terabyte on machines with up to 100 CPU's. Today, terabyte SMP class machines are available [SGI2003] but are still rare and very expensive[113].

So how can we demonstrate such scalability? And then, even if we do demonstrate terabyte SMP class scalability, what happens when petabyte (and subsequently exabyte) SMP machines become a reality?

Instead, can we use RHmalloc to show that dynamic memory management can be made to scale to any size, on machines with any number of CPU's and any amount of memory?

9.2 Proposition

One possible way to show scalability, is to examine the response time properties of RHmalloc on a gigabyte class machine (32-bit), in a systematic manner, such that we can prove by extrapolation, that heaps can scale theoretically to any number of CPU's, with any amount of memory.

Further, we believe it is reasonable to assert that any well formed, very large, highly scalable dynamic memory manager, will have response time related properties which can be mathematically proven to be independent of CPU and memory. Consequently, if we can identify and demonstrate such properties in RHmalloc, then we have formally proven that dynamic memory management scales indefinitely.

---

[113] Personal discussion with Bob Bishop, CEO of Silicon Graphics, Inc. The only three terabyte class machines built by SGI to date are currently at NASA performing space shuttle simulations.

9.3 Reviewing RHmalloc

We begin by reviewing the design of RHmalloc. RHmalloc is based on:

(a) Dividing a heap into an indefinite number of sub-heaps, each of finite size **S**, arranged such that, within any given sub-heap the response time **r** is O(1), being independent of the number of blocks **b**, and the block size **B**;

(b) Managing an indefinite number of sub-heaps **s** arranged such that, the time **f** to find a suitable sub-heap to satisfy a request is O(1), being independent of the number of sub-heaps **s**, and the sub-heap size **S**; and

(c) Selecting sub-heaps so as to minimize contention such that over time **f** + **r** remains independent of the number of threads **t** in the system.
We note though that certain request patterns can always produce contention hotspots such that **f** + **r** would be affected – potentially adversely. This is a corollary to [Rob1971, GGU1972, Rob1974, Rob1977] who showed that no solution to fragmentation exists.

The overall response time **R** to allocate/free any given block is given by **f** + **r**.

The size of a heap **H** is given by **S** × **s**.

9.4 Using a 32-bit Machine

The first version of RHmalloc was built and tested on a dual AMD Athlon 2100 (128KB L1 cache) with 2GB of DDR RAM, running Microsoft Windows XP using Microsoft Visual C++ 6.0.

On such a 32-bit machine we can readily vary the:

1. Size of the sub-heap **S** from 1KB to 1GB;
2. Number of sub-heaps **s** from 1 through 1M; and
3. Number of threads **t** from 1 through 1,024, thereby simulating a machine with up to 1,024 CPU's.

9.5 1PB by Extrapolation

Although we cannot have a heap consisting of 1M sub-heaps, each 1GB in size on a 32-bit machine, if we can show that (a), (b) and (c) hold true for testable combinations of the above, then by extrapolation we can claim scalability to at least 1M sub-heaps of 1GB each for 1,024 CPU's:

H = max(**S**) x max(**s**) for max(**t**)

= 1GB x 1M for 1,024 threads

= 1PB on a 1,024 CPU machine

Restating: if the within sub-heap response time **r** is the same for the range min(**S**) to max(**S**), and we have tested max(**s**) for min(**S**) for max(**t**), then by extrapolation we have shown max(**S**) for max(**s**) for max(**t**), since **r** is an invariant of **S**, **s** and **t**.

9.6 Theoretically Indefinite

Furthermore, if we show that **R** = **f** + **r** is independent of **B**, **b**, **S**, **s** and tends to remain independent of **t** for most cases, then we have shown theoretically that dynamic memory management can in fact be made to scale indefinitely.

9.7 Test Suite

Consequently, we propose three tests:

1. **Test 1 – Within Sub-Heap Response Time**
   Allocate and then free all of memory within a single sub-heap, and measure the within sub-heap response time **r**. We vary the size of the sub-heap **S,** and the block size **B** – thereby testing a large range of blocks **b** in the sub-heap. We observe if the within sub-heap response time **r** is independent of **S, B** and **b**;

2. **Test 2 – Find Suitable Sub-Heap Time**
   Prearrange memory such that each and every allocation request requires finding a suitable sub-heap, and measure the time **f** to find a sub-heap. We vary the number of sub-heaps **s**, and the size of the sub-heaps **S**. We observe if the time **f** to find a sub-heap is independent of the number of sub-heaps **s**, and the size of the sub-heaps **S**; and

3. **Test 3 – Overall Response Time**

   Program a thread to allocate a random number of random sized blocks and then free them, and measure the overall response time **R**. We vary the number of executing threads **t,** both the number of sub-heaps **s**, and the size of the sub-heaps **S**. We observe if, and under what conditions, the overall response time **R** remains independent of the number of executing threads **t**.

If our design and implementation is sound, tests 1 and 2 should readily show (a) & (b). Test 3 is more complex, but the question is: Will the overall response time **R** remain relatively consistent, i.e. independent of **B**, **b**, **S**, **s** and **t**, and under what conditions?

If **R** does tend to remain consistent, we have shown that dynamic memory management can scale indefinitely with respect to both CPU and memory, to any size.

9.8 Instrumentation

We instrumented the code to collect data from various parts of RHmalloc.

The following table itemizes the statistics collected in each run.

VALUE	DESCRIPTION
#RQ	Total number of requests made
RQtime/rq	Average time per request
#AB	Total number of block allocations
ABtime	Total time spent allocating blocks
ABtime/a	Average time per block allocation
#FB	Total number of block frees
FBtime	Total time spent freeing blocks
FBtime/f	Average time per block free
#SB	Total number of block splits
SBtime	Total time spent splitting blocks
SBtime/sb	Average time per block split

VALUE	DESCRIPTION
%rq SB	Percentage of requests that incurred a block split
SBtime/rq	Average time spent splitting blocks per request
#CB	Total number of block coalesces
time	Total time spent coalescing blocks
CBtime/cb	Average time per block coalesce
%rq CB	Percentage of requests that incurred a block coalesce
CBtime/rq	Average time spent coalescing blocks per request
#SF	Total number of times a sub-heap was shifted to a different space list
SFtime	Total time spent shifting sub-heaps
SFtime/sb	Average time per sub-heap shift
%rq SF	Percentage of requests that incurred a sub-heap shift
SFtime/rq	Average time spent shifting sub-heaps per request
#FS	Total number of times a sub-heap was searched for (a.k.a. find)
FStime	Total time spent finding sub-heaps
FStime/sb	Average time per sub-heap find
%rq FS	Percentage of requests that incurred a sub-heap find
FStime/rq	Average time spent finding sub-heaps per request
#HM	Total number of times the heap map was adjusted
HMtime	Total time spent adjusting the heap map
HMtime/sb	Average time per heap map adjustment
%rq HM	Percentage of requests that incurred a heap map adjustment
HMtime/rq	Average time spent adjusting heap map per request
#SM	Total number of times a sub-heap map was adjusted
SMtime	Total time spent adjusting sub-heap maps

VALUE	DESCRIPTION
SMtime/sb	Average time per sub-heap map adjustment
%rq SM	Percentage of requests that incurred a sub-heap map adjustment
SMtime/rq	Average time spent adjusting sub-heap map per request
#LL	Total number of times contention for heap space-lists occurred
%rq LL	Percentage of requests that incurred heap space-list contention
#LS	Total number of times contention within sub-heaps occurred
%rq LS	Percentage of requests that incurred within sub-heap contention

Table 9.1: Table of statistics collected in RHmalloc test runs.

9.9 Notes

We note that the cost of observing these tests to this detail is high, and therefore alters the program behavior – particularly performance. The interference should, however, be almost uniform across all aspects, so the relativities should remain true. In the instrumented version of RHmalloc used in the tests below, we are therefore only interested in relative, not absolute performance.

Note that, in practice not every allocation requires finding a suitable sub-heap. In many cases the sub-heap last used may be satisfactory. Also, the time for finding a suitable sub-heap potentially involves, shifting the previously used sub-heap into a different space list. This may also involve altering the heap space map bit index. Thus we do expect to see some variation in the time **f** to find a suitable sub-heap, but this variation should not be dependant on the number of sub-heaps **s**, or the size of those sub-heaps **S**.

Also note that, in practice not every allocation or deallocation within a sub-heap, involves the same number of CPU cycles. Some allocations may require block splitting. Some frees may require block coalescing. Some actions may require the sub-heap free map bit index to be altered. Thus we do expect some variation in **r** but this variation should not be a function of **B**, **b**, **S** or **s**.

9.10 Side Effects

We are also interested in observing side effects. It may be true, that under certain conditions, a significant increase in the number of objects in the system, may in fact lead to a reduction in the work to be done.

For example:

- The more sub-heaps there are in the heap, the less CPU cycles should be required to find a suitable sub-heap. Sub-heaps are always within a space list. The more sub-heaps the more likely a sub-heap of exactly or larger size is available. Searching is minimized;
- The heap only needs to have its space map changed, when the first sub-heap for a given space list becomes available, or when the last sub-heap for a given space list disappears. So with many sub-heaps, the probability of either of these two conditions occurring diminishes; and
- The same two phenomena should be true within sub-heaps. The larger the sub-heap, the less work required to keep the sub-heap's maps in order.

9.11 Test 1 – Within Sub-Heap Response Time

9.11.1 Objective

Observe if, the within sub-heap response time **r** of RHmalloc is O(1), being independent of the size of the sub-heap **S**, the block size **B** or the number of blocks in a sub-heap **b**.

9.11.2 Description

We vary (in steps which are powers of two) the size **S** of a single sub-heap, from 1KB through to 1GB, and for each sub-heap size we vary the block size **B** to be allocated, from 1B to 1KB. This yields 231 runs.

In each run, we allocate the particular block size until the single sub-heap is full, and then we free all allocated blocks. We repeat this for at least 10M allocations, in each run.

9.11.3 Results

The following diagram shows the results for Test 1.

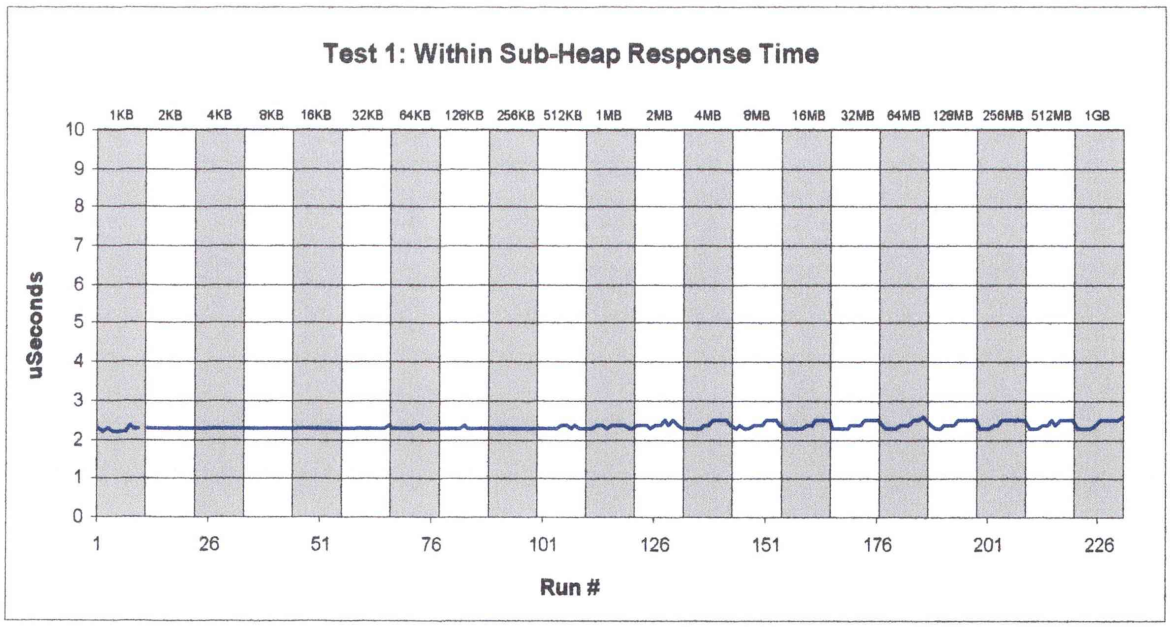


*Figure 9.1: Test 1 results showing average within sub-heap response time **r** per request (in microseconds).*

The above graph shows the response time (in microseconds) per request, for all of the runs for Test 1. The size of the heap is shown as a rectangle enclosing the results for the runs of a given heap size. In first run (run #1) the sub-heap is 1KB, while in the last run (run #231) the sub-heap is 1GB.

We note that run #11 failed – it was not possible to allocate a 1KB block within a 1KB sub-heap.

The following table presents the time to perform various operations, across all runs for Test 1.

ACTION	MINIMUM	AVERAGE	MAXIMUM
Allocate Block	2.10 µsecs	2.40 µsecs	2.70 µsecs
Free Block	2.10 µsecs	2.31 µsecs	2.50 µsecs
Split Block	0.80 µsecs	0.96 µsecs	1.20 µsecs
Coalesce Block	0.80 µsecs	0.99 µsecs	1.20 µsecs
Sub-Heap Mapping	0.20 µsecs	0.27 µsecs	0.30 µsecs

Table 9.2: Time to perform various operations in Test 1.

9.11.4 Comments

Generally, we see little variation in the within sub-heap response time **r**. In all cases, the within sub-heap response time **r** was between 2.2 and 2.6 microseconds.

In analyzing the detailed data, we observe very little variation in the time to allocate a block, the time to free a block, the time to split a block, the time to coalesce blocks, or in the time to adjust the sub-heap map.

We cannot observe any relationship between, the within sub-heap response time **r,** and the size of the sub-heap **S,** the block size **B,** or the number of blocks in the sub-heap **b**.

9.11.5 Conclusion

We conclude that **r** is O(1), being independent of **S**, **B** and **b**.

9.12 Test 2 – Find Suitable Sub-Heap Time

9.12.1 Objective

Observe if, the time **f** to find a suitable sub-heap to satisfy an allocation request in RHmalloc is O(1), being independent of the number of sub-heaps **s** in the heap, and the size **S** of the sub-heaps.

9.12.2 Description

Memory is prearranged so that each and every allocation requires shifting the previously used sub-heap to a different space list, and then finding a different suitable sub-heap.

This is achieved in the following manner. Firstly, for each run, eight blocks of different sizes are pre-allocated in each and every sub-heap. Any remaining space in each sub-heap is allocated, and subsequently never used. The largest block in each sub-heap is then freed. This puts the heap in an initial state, where each sub-heap has seven allocated blocks of seven different sizes, and only one free block – which is larger than any of the allocated blocks.

During a run, a thread starts off repeatedly allocating a block of the largest size. Given there is only one free block of that size in each sub-heap, each allocation will cause the thread to shift to a new sub-heap. After allocating one block, the thread then frees the block of the next size down in that sub-heap. This two step process repeats until no more blocks of the largest size can be allocated. This then leaves the heap in a state where, each sub-heap has seven allocated blocks of seven different sizes, and one free block which is the second largest of any block.

This sequence is then repeated for each block size, down to the smallest.

When all block sizes have been exhausted in this manner, the heap is reset as per the pre-allocation step, and the whole sequence is re-executed. This continues for 30 minutes per run.

All in all, 21 runs are tested. A fixed 1GB heap is used, but the sub-heap size is varied from 1GB down to 1KB, yielding from one to 1M sub-heaps in the heap.

9.12.3 Results

The following diagram shows the results for Test 2.

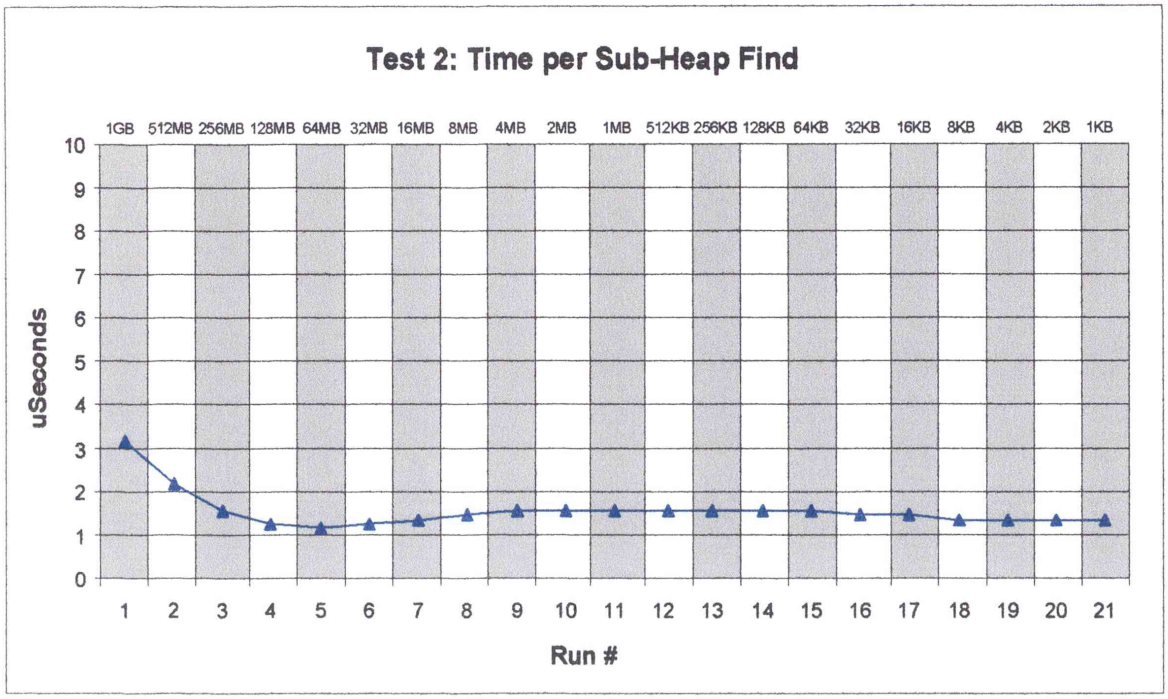


Figure 9.2: Test 2 results showing the average time f to find a suitable sub-heap to satisfy an allocation request.

The above graph, shows the time (in microseconds) to find a suitable sub-heap in order to satisfy an allocation request. There is one run per heap size. Note that in run #1 there is only one sub-heap in the heap, while by run #21 there are 1M sub-heaps.

The following table presents the time to perform various actions for Test 2.

ACTION	MINIMUM	AVERAGE	MAXIMUM
Allocate Block	3.40 μsecs	4.15 μsecs	6.20 μsecs
Free Block	1.90 μsecs	2.01 μsecs	2.10 μsecs
Testing Coalescence	0.40 μsecs	0.51 μsecs	0.57 μsecs
Shift Sub-Heap	0.80 μsecs	0.98 μsecs	2.00 μsecs
Find Sub-Heap	0.40 μsecs	0.61 μsecs	1.20 μsecs
Heap Mapping	0.00 μsecs	0.31 μsecs	0.90 μsecs
Sub-Heap Mapping	0.20 μsecs	0.28 μsecs	0.30 μsecs

Table 9.3: Time to perform various operations in Test 2.

9.12.4 Comments

We see little variation in the time to find a suitable sub-heap. In all cases the within time was between 1.4 and 3.2 microseconds.

Recall, that the time to find a sub-heap consists of two steps:

- Shifting the previous sub-heap into a new space list; and
- Looking up the space map, to select a specific sub-heap of suitable capacity, and rotating the space list within which the selected sub-heap resides (in order to minimize contention).

In examining the results in detail, we observe that as the number of sub-heaps in the heap increases from 1 to 1M, the time to perform these two steps does not increase. Interestingly, the maximum time to allocate a block occurs when there is only one-sub-heap in the heap. This is due to the fact that every second allocation fails, because of the prearrangement of memory – the algorithm is designed to retry a number of times before giving up.

We also observed that, when there are very few sub-heaps in the heap (less than four) the time to select a suitable sub-heap is slightly larger, than the remaining cases. This is

due to the fact that the heap map is relatively empty (few sub-heaps exist) so the time to search the heap map is slightly longer in those cases.

9.12.5 Conclusion

We conclude that the time to find a suitable sub-heap **f** is O(1), being independent of the number of sub-heaps **s** in the heap, and the size **S** of the sub-heaps.

9.13 Test 3 – Overall Response Time

9.13.1 Objective

Observe if sub-heaps are selected, so as to minimize contention such that over time and for most cases, **f** + **r** remains independent of the number of threads **t** in the system.

9.13.2 Description

We vary (in steps which are powers of two) the size of the heap **H** from 16MB through to 1GB. We vary the size of the sub-heap **S** from the size of the heap down to 16KB. We vary the number of threads **t** from 2 to 1,024. This produces 980 individual runs.

In each run, each thread allocates a random number of random size blocks, and then subsequently frees them. The blocks allocated by one thread are not shared or freed by another. Consequently, the blocks can be seen as a form of working set private to a thread.

9.13.3 Results

The following diagrams show the results for Test 3.

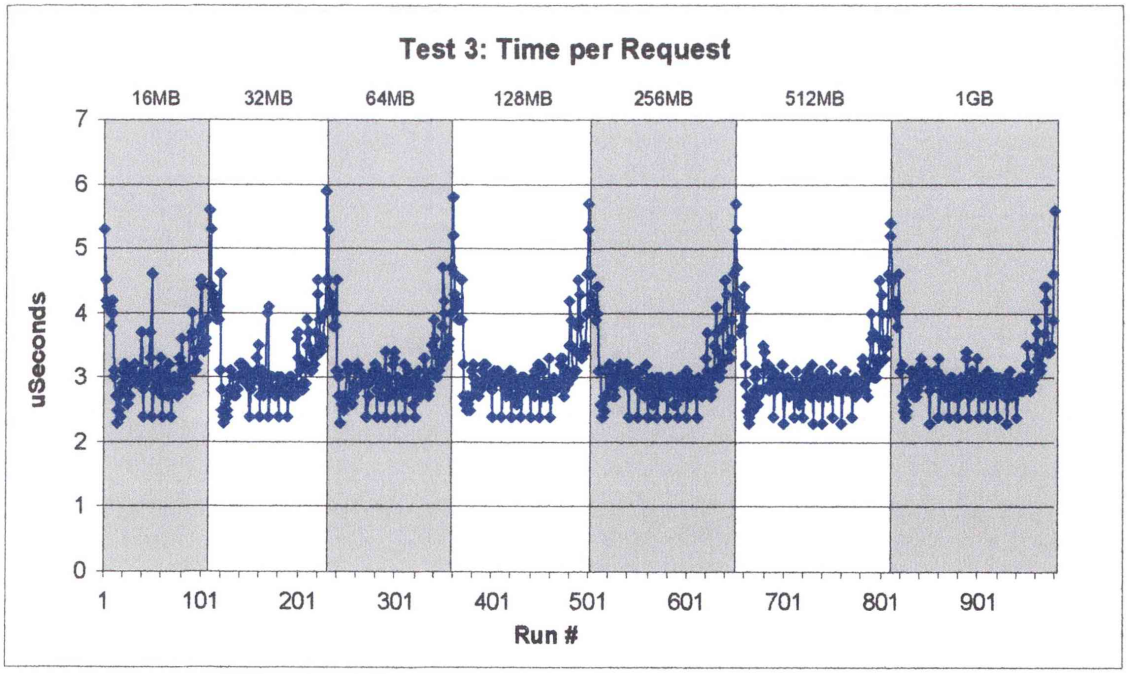


Figure 9.3: Test 3 results showing the average time per request in microseconds for each run.

The above graph, shows the time (in microseconds) to satisfy either an allocation or free request. The size of the heap is shown as a rectangle enclosing the results for the runs of a given heap size. The size of the sub-heap **S** and the number of threads **t** is not shown.

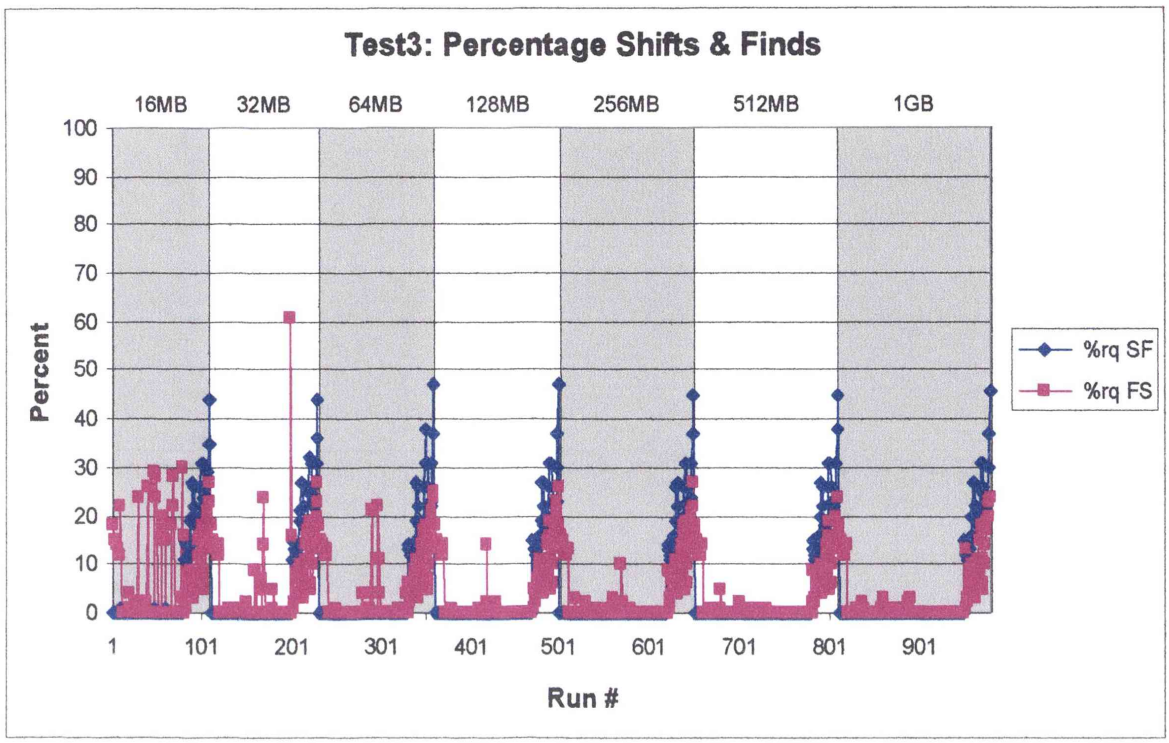


Figure 9.4: Test 3 results showing the percentage of requests which require sub-heap shifts (SF) and the percentage of requests which require sub-heap finds (FS) for each run.

The above graph shows, for each of the individual test runs, the percentage of requests which required threads to shift the previously used sub-heap to a different space list, and the percentage of requests which required threads to find a different sub-heap than that previously used.

We note that Run #201 had an unusually high percentage of requests that required sub-heap finds.

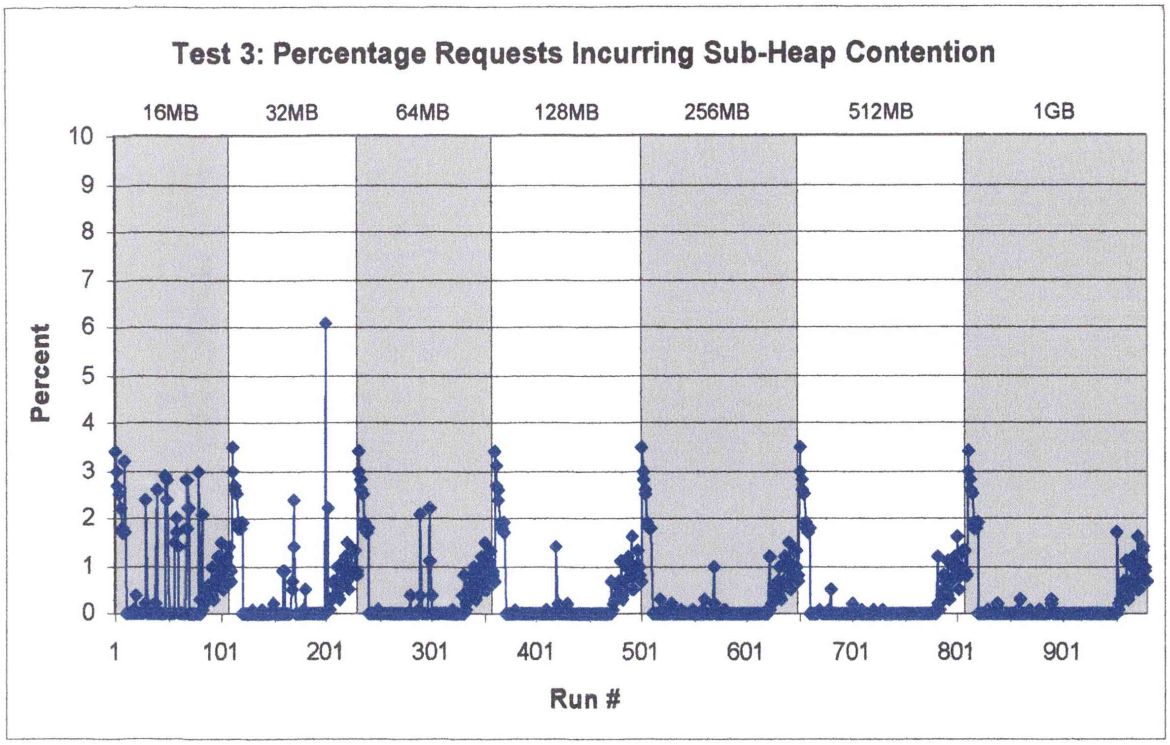


Figure 9.5: Test 3 results showing the percentage of requests which incur sub-heap contention for each run.

The above graph shows, for each of the individual test runs, the percentage of requests which incurred sub-heap contention – those runs where threads collide with another thread, while trying to use the free lists with a sub-heap.

One result, which is not presented as a graph, is the percentage of requests which incurred space-list lock contention. While there was some observed contention, it was so low that the average number of requests, which incurred space-list lock contention, was effectively zero for all runs. This is an important result, as it indicates the space-list vector is very highly scalable.

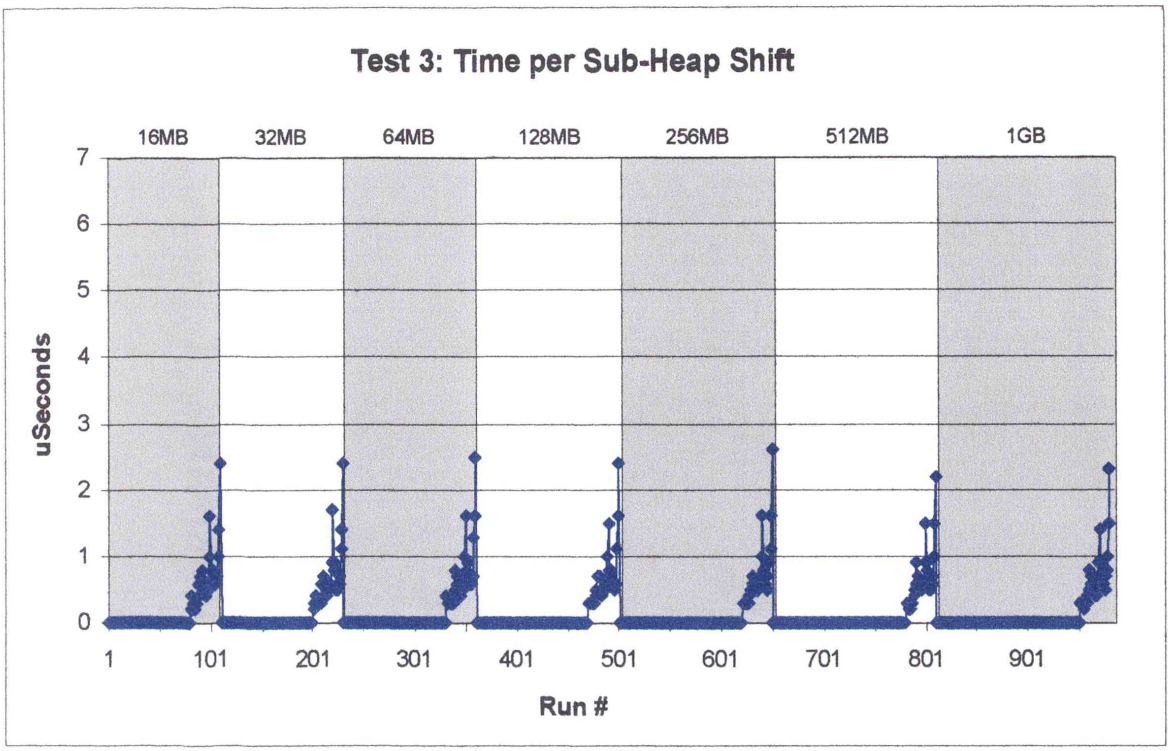


Figure 9.6: Test 3 results showing the average time per sub-heap shift (in microseconds) for each run.

The above graph shows the average time taken to shift a sub-heap from one space list to another, for each of the individual test runs.

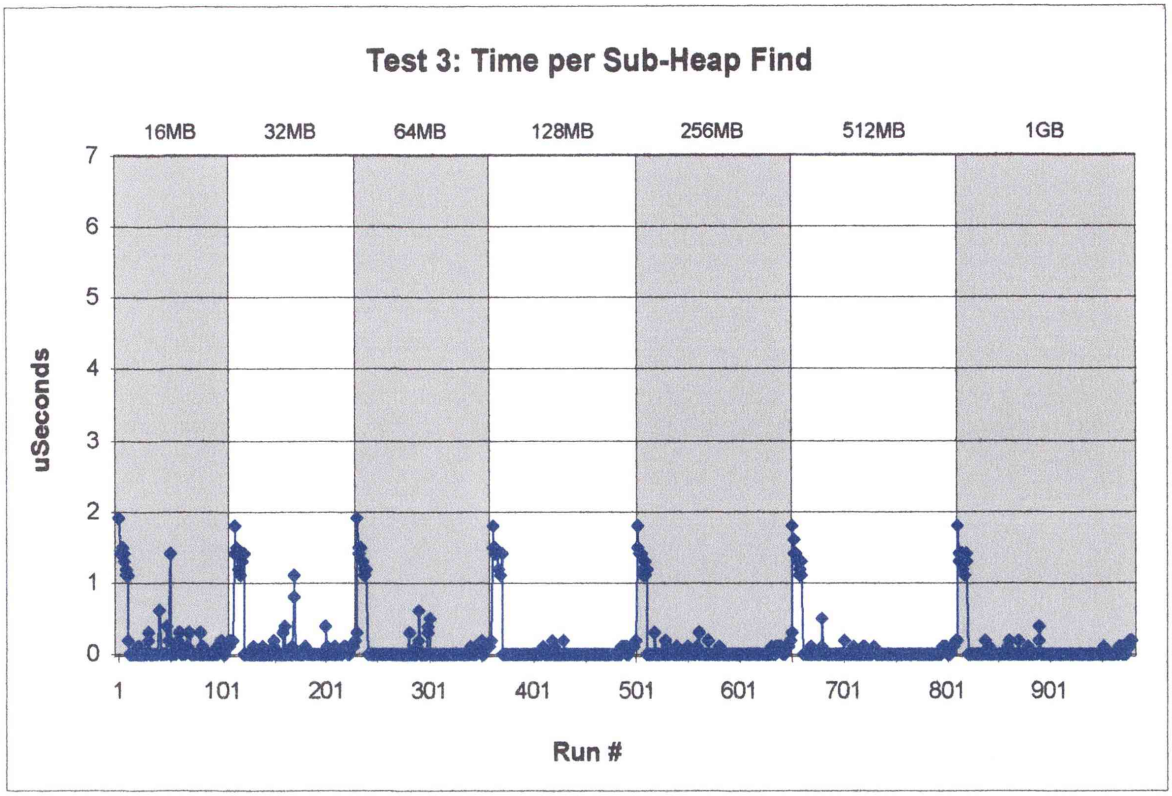


Figure 9.7: Test 3 results the time per sub-heap find (in microseconds) for each run.

The above graph shows the average time taken to find a new suitable sub-heap, for each of the individual test runs.

The following table presents the time to perform various operations for Test 3.

ACTION	MINIMUM	AVERAGE	MAXIMUM
Allocate Block	2.10 µsecs	3.27 µsecs	7.70 µsecs
Free Block	2.10 µsecs	2.93 µsecs	6.40 µsecs
Coalescing Blocks	0.30 µsecs	0.44 µsecs	0.60 µsecs
Shift Sub-Heap	0.00 µsecs	0.15 µsecs	2.60 µsecs
Find Sub-Heap	0.00 µsecs	0.12 µsecs	1.90 µsecs
Heap Mapping	0.00 µsecs	0.02 µsecs	0.20 µsecs
Sub-Heap Mapping	0.00 µsecs	0.54 µsecs	0.80 µsecs

Table 9.4: Time to perform various operations in Test 3.

9.13.4 Comments

Interestingly, Figure 9.3 shows a very similar pattern in the response times, as we vary the heap size from 16MB through to 1GB – showing an almost identical range of response time values, for each given heap size. Importantly we see good, and in many cases (see §9.14 below), flat response times across the entire range of test runs, for up to the tested limit of 1,024 threads.

Given we are varying the sub-heap size and the number of threads, in the same way for each heap size, we can immediately muse upon the hypothesis, that the response time pattern reflects some internal behavior of RHmalloc, which changes as the sub-heap size and executing threads are varied.

The question arises, why then do we see this particular pattern?

Examining the data in detail, we see the same similar pattern in the number of requests which perform either a sub-heap shift, and/or a sub-heap find. Figure 9.4 shows the percentage of requests which perform shifts and finds across the test runs.

We also see the same similar pattern in Figure 9.5, which depicts the number of requests which incur within sub-heap contention.

Figures 9.6 and 9.7 show that the time per sub-heap shift or sub-heap find does not vary significantly – in fact comparing Tables 9.1 with 9.2, the average shift sub-heap and find sub-heap times are less in Test 3 (a multi-threaded run) than Test 2 (a single-threaded run). What varies in relation to the sub-heap size **S** and number of threads **t** across the runs in Test 3, is the rate the shift and find operations are performed, and the degree of within sub-heap contention.

We explain this as follows. Recall from the design, multiple threads can be allocating from the same sub-heap. As either the number of threads increases, or the size of the sub-heap decreases (for any given heap size) the rate at which a sub-heap becomes fully allocated increases. Faster allocation exhaustion causes threads to perform sub-heap shifts and sub-heap finds at a greater rate, than when there are fewer threads and/or larger sub-heaps. This increased probability of shift and find operations alters the average response time accordingly.

Additionally, RHmalloc detects thread contention within sub-heaps, and attempts to find a new sub-heap with a lower concurrency. This also causes sub-heap shift and find activity.

Importantly, it is because of this phenomenon (which emerges out of the design of RHmalloc) that we can observe that, the overall response time **R** remains relatively consistent, for a given configuration of both the sub-heap size **s** and the number of threads **t**, across varying heap sizes. Increasing the size of the heap **H**, by being able to increase both the size of each sub-heap **S** and the number of sub-heaps **s**, is a key aspect to RHmalloc's scalability.

9.13.5 Conclusion

We conclude that the overall response time **R** is O(1), tending to remain independent of the number of threads **t**.

9.14 A More Detailed Look

As noted early, each of the seven different heaps sizes used in Test 3 produced a very similar response pattern. In this section we take a more detailed look at those patterns. While we looked in detail at all seven patterns, we choose at random, to present the findings for the 256MB heap.

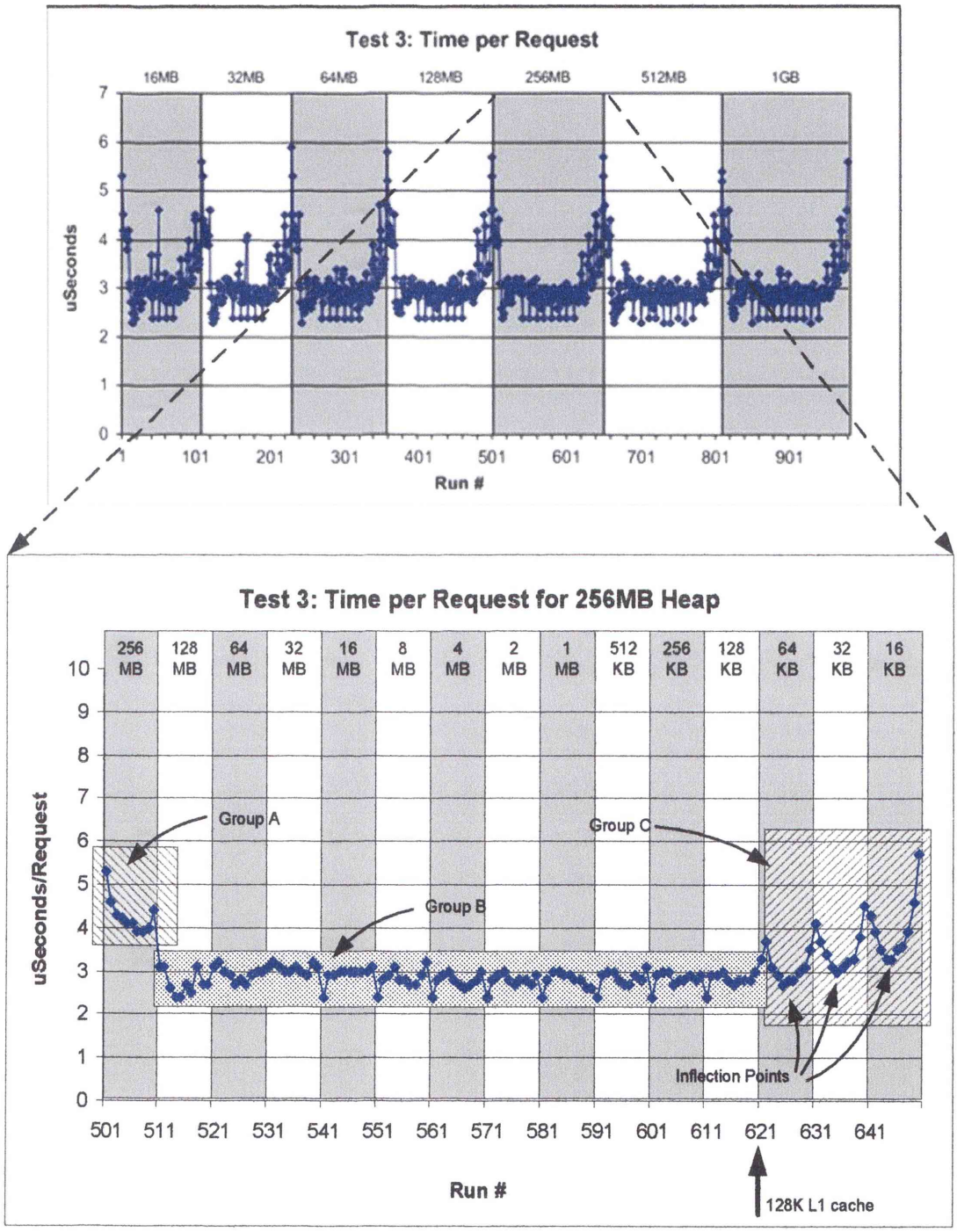


Figure 9.8: The 256MB heap response pattern – rectangles show sub-heap size.

The previous diagram shows the results for the 256MB heap test in more detail. The rectangles show the size of the sub-heap. In this test there is 1 × 256MB sub-heap (for runs 501 – 510) through to 16,384 × 16KB sub-heaps (for runs 641 – 650).

The heavy vertical arrow, between the 128KB and 64KB sub-heaps, is a visual reminder that the machine these tests are being run on has a 128KB L1 cache.

Our first observation is that there seem to be three distinct groups[114]. We will call these groups: Group A – Single Sub-Heap, Group B – Independent Working Sets and Group C – Interwoven Working Sets.

9.14.1 Group A – Single Sub-Heap

Generally Group A – Single Sub-Heap can be characterized as:

- Consisting of one and only one sub-heap;
- No sub-heap shift activity occurs;
- No heap mapping activity occurs;
- Moderate find sub-heap activity (between 12-18% of requests involved a find sub-heap for the 256MB case) ;
- Rare space list contention (much less than 0.1% of requests encountered contention for a space list for the 256MB case); and
- Medium sub-heap contention (between 18-35% of requests encountered contention within a sub-heap for the 256MB case).

Even though there is only one sub-heap, most, if not all of the test runs yielded good response times.

9.14.2 Group B – Independent Working Sets

Generally Group B – Independent Working Sets can be characterized as:

---

[114] In each of the heap sizes tested we observed these same three distinct response groupings.

- Consisting of two or more, medium to large sub-heaps (between 256KB - 128MB for the 256MB case);
- Very little sub-heap shift activity, if any;
- Very little heap mapping activity, if any;
- Light find sub-heap activity (10% or less of all requests for the 256MB case);
- Rare space list contention (much less than 0.1 % for the 256MB case); and
- Very little sub-heap contention, if any.

It is important to note that there are 110 test runs, with between 2 and 1,024 threads, which all have excellent response times – and similar results can be seen across the other six different heap sizes.

9.14.3 Group C – Interwoven Working Sets

Generally Group C – Interwoven Working Sets can be characterized as:

- A large number of small sub-heaps (4,096 × 65KB to 16,384 ×16KB sub-heaps for the 256MB case);
- Significant shift sub-heap activity (11-45% of requests for the 256MB case);
- Significant heap mapping activity (22-62% of requests for the 256MB case);
- Significant find sub-heap activity (0-27% of requests for the 256MB case);
- Rare space list contention (much less than 0.1 % for the 256MB case); and
- Moderate sub-heap contention (1-13% of requests for the 256MB case).

We observe many of the test runs yielded good response times.

9.14.4 The Three Sub-Curves

In Group C for the 256MB case, we see three sub-curves, where for each of these sub-curves the response time tends to fall away till an inflection point, whereupon the response time then rises. These three sub-curves correspond to three different sub-heap sizes: being 64KB, 32KB and 16KB respectively.

The following graph shows the percentage of requests that involve a sub-heap find. A similar pattern is seen for sub-heap shifts and heap mapping.

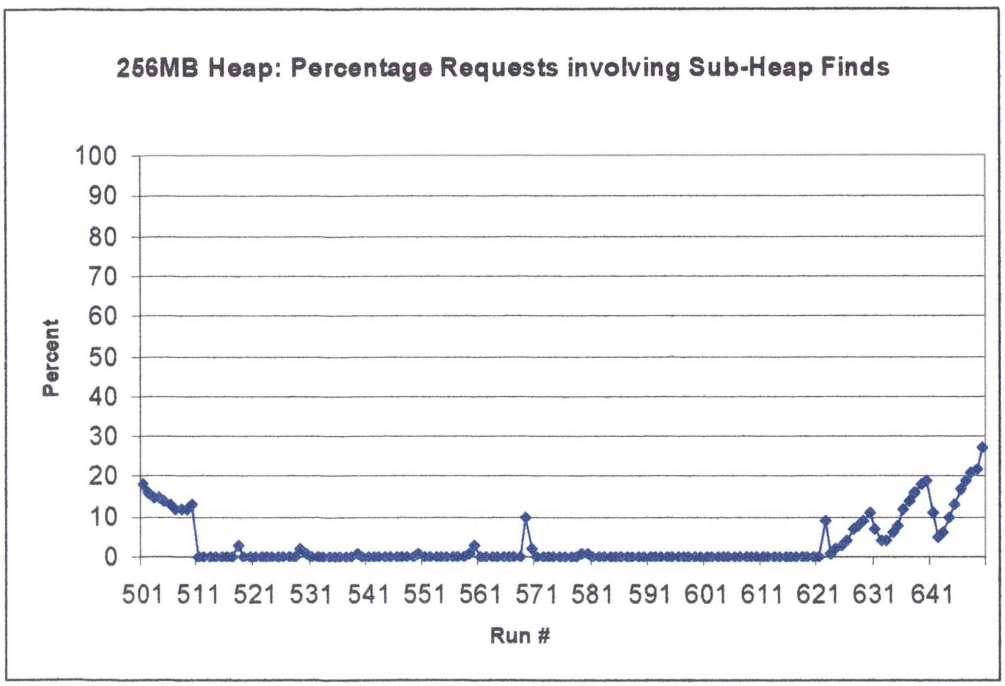


Figure 9.9: Percentage of requests that involve sub-heap finds for the 256MB Heap.

Examining the data for Group C, the response time appears to fall away in each of three cases, because of a reduction in the frequency of sub-heap shifts, sub-heap finds and heap mapping. After the inflection point, there is a combined increase in sub-heap shifts, sub-heap finds and consequently heap mapping, with a corresponding increase in response times.

The question is, why then do we see such a change in sub-heap shifts, sub-heap finds and heap mapping?

9.14.5 Long Running Steady State Systems

In any non-static long running system, the number of block frees, must over time, equal the number of block allocations – otherwise the system would exhaust all available memory. Any given shorter interval may exhibit a pattern showing an imbalance between these two rates, but over time, these rates always tend towards each other.

For example, if we map the number of block allocations and frees for the 256MB test runs, we get the following graph.

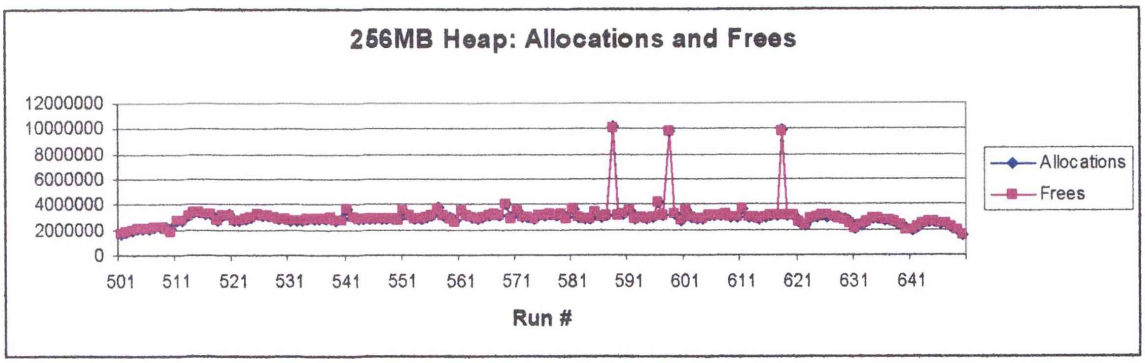


Figure 9.10: Number of allocations and frees for the 256MB Heap.

There are two curves plotted, but both go through the same points – so they appear as a single line.

9.14.6 Block Coalescing

We observed that block coalescing, however, is not uniform in the system. Plotting the percentage of frees which perform a block coalesce, we see that the curve appears as a mirror image of the response time.

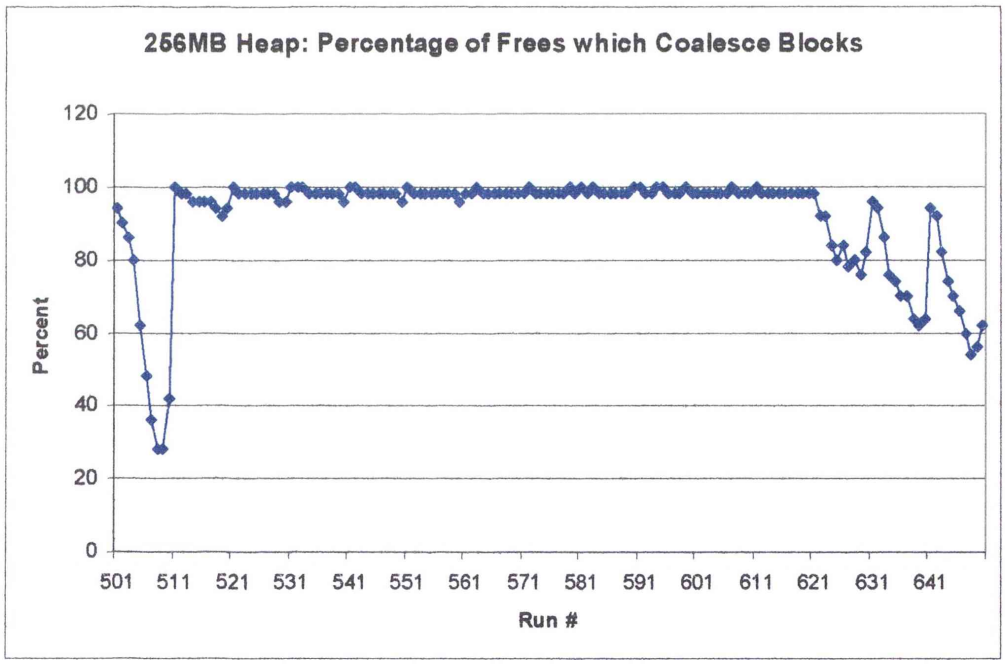


Figure 9.11: Percentage of frees which coalesce blocks for the 256MB heap.

We explain this as follows. For the test runs in Group B, there is little or no sub-heap shift or find activity. This is because each sub-heap is sufficiently large enough to

satisfy the full allocation request issued by a thread, and there are more sub-heaps than threads. Consequently, threads tend to have a single sub-heap to themselves.

Things tend to be very orderly in the Group B test runs. The allocations by any given thread are serviced from one sub-heap. And the blocks are freed in the reverse order they are allocated, so each free involves a coalesce action.

In Group C, things tend to get a bit more chaotic. The sub-heaps are not large enough to satisfy the full allocation request, so threads must use multiple sub-heaps. Recall we observed significant sub-heap shifts and finds – threads are searching, and to a degree are competing, for suitable sub-heaps. Given threads share sub-heaps, but not the blocks themselves, the block allocations tend to become randomly scatted throughout sub-heaps. Consequently, frees are statistically less likely to perform coalescence.

In Group A, there is only one sub-heap shared by up to 1,024 threads – so things tend to be very chaotic. Consequently, we can see in Figure 9.11 that there are times when very little coalescing occurs.

9.14.7 Cache Related Effects

Refer again to Figure 9.8. In Group C we have an increased sub-heap find and shift activity, compared to Group B. Also recall that the test machine has a 128KB L1 cache.

When a thread is forced to switch to a different sub-heap, the pertinent information relating to that sub-heap has to make its way into the cache – if it is not already there. Cache misses cause the CPU to stall, thereby increasing response times.

Sub-heap find activity may also have the additional side effect of invalidating cache, to some degree. While we minimize invalidation effects, by using a bitmap to hold condensed information, we should expect the act of searching the heap space map and vector, to have some effect on the contents of the cache – particularly in a multi-processor environment.

We conclude that cache related effects are probably a contributing factor, in the observed minor increase in response times.

Please note however, that caching is one of the least observable behaviors on a machine and in sophisticated cache architectures our suppositions may not hold.

9.14.8 A 1PB Heap on a 1,024 CPU Machine

Given this insight into the behavior of RHmalloc, it is interesting to muse on what we would observe on a very large machine. From our results, we should expect a similar response time pattern for a 1PB heap on a 1,024 CPU machine – with some notable differences due to the number of CPU/cache's, and the amount of memory available.

Note that the 1PB heap test case would have 370 test runs, across 37 different sub-heap sizes – from 1PB down to 16KB. Ultimately, a 1PB heap could be configured with 68,719,476,736 × 16KB sub-heaps.

Firstly, we would expect multiple sub-curves in Group A, initially with exaggerated peaks and troughs. Since there are a small number of sub-heaps, that are being shared between a large number of CPU/cache's, there would be increased inter-cache traffic. In the case of the single 1PB sub-heap, having 1,024 CPU's simultaneously trying to access the same cache lines, could induce such high levels of thrashing, that very little real work would actually get done.

The number of such sub-curves in Group A would likely be a function of the number of CPU/cache's; and the response time, a function of the probability and degree of simultaneous cache line access. As we increase the number of sub-heaps, we would expect the response time peaks and troughs to reduce, due to a corresponding reduction in inter-cache traffic; the shape of the curves therefore morphing, as we increase the number of sub-heaps, into the relatively flat response time pattern of Group B.

Secondly, we'd expect Group B to be longer – potentially almost double in size. There are a large number of configurations which would have substantially more sub-heaps than CPU/cache's, and said sub-heaps have adequate space for a threads working set. Consequently, and most importantly, we would continue to observe good response times in this group, due to little or no thread contention within sub-heaps, or inter-cache traffic.

Thirdly, we probably would observe a similar number of sub-curves, with similar shape, in Group C as we have for the 256MB heap case. For Test 3, sub-heaps of 64KB in size, and less, are not large enough for a thread's working set, so there would be similar shift and find activity. Shift and find activity is a function of the sub-heap size; not the overall heap size.

9.14.9 The Inflection Points

We are interested in understanding why we see the inflection points in Group C. Is there an explanation to why we see this change? The following diagram plots the time to allocate a block, versus the time to free a block for Group C.

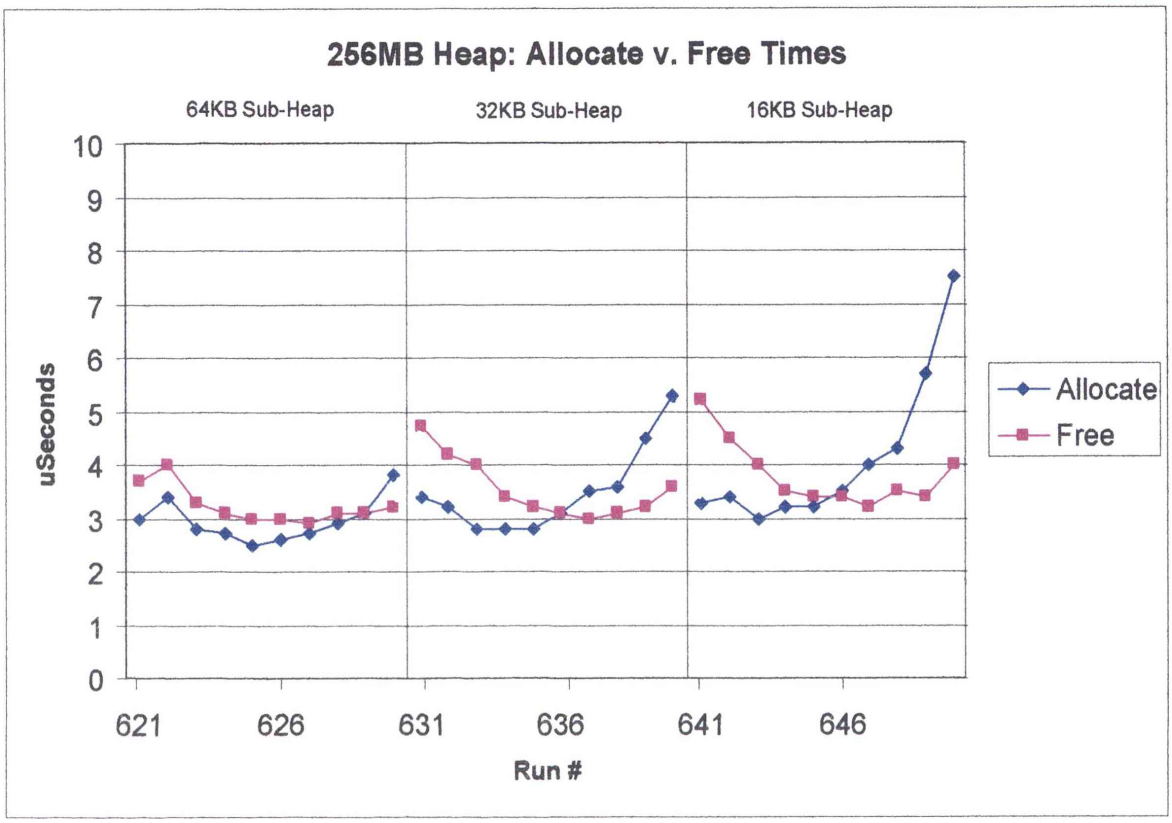


Figure 9.12: Time to allocate a block versus time to free for 256MB heap.

Note that there are 4,096 × 64KB sub-heaps, 8,192 × 32KB sub-heaps, and 16,384 × 16KB sub-heaps tested progressively within the 256MB heap. Importantly, we see relatively flat response times in the 64KB sub-heap for 2 to 1,024 threads. We see that in Group C, for each sub-heap size, the time to free a block starts out greater than the time to allocate, and then progressively the time to allocate becomes the greater. We see these values cross once, in each of the sub-heap sizes tested – at points which correlate with the inflection points in Group C. This pattern is exhibited in the test runs for the other six heap sizes. This may be the basic cause of the inflection point.

We explain this as follows. As the time to free a block reduces, the rate of block frees increases. The probability that a thread will find a suitable block, in its current sub-heap, therefore increases. This reduces the frequency of sub-heap exhaustion,

consequently reducing the rate of sub-heap shifts, along with sub-heap finds and heap mapping.

As the time to allocate a block increases, the probability that threads will be allocating increases. This increases sub-heap contention and consequently increases sub-heap shift, sub-heap finds and heap mapping.

We conclude that the response time pattern, exhibited in Group C, probably emerges from the complex interplay between resource availability and resource contention.

9.14.10 Possible Inherent Imbalance

We examined the time spent splitting and coalescing blocks for Group C. The following diagram plots the total time spent splitting blocks, and the total time spent coalescing blocks.

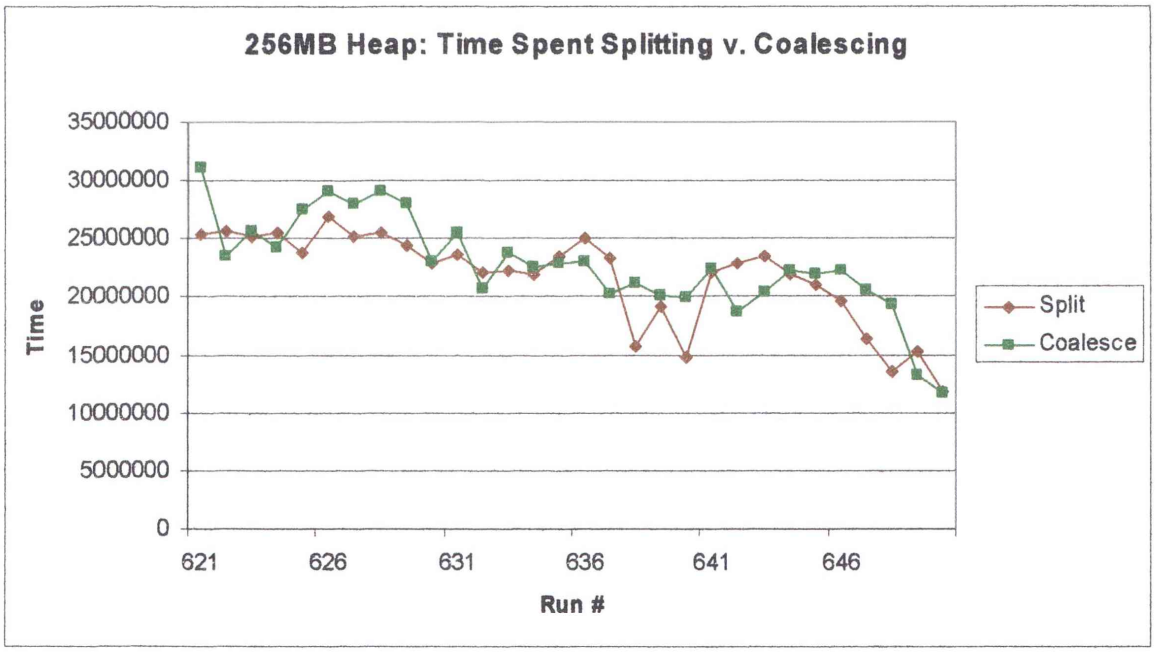


Figure 9.13: Time spent splitting blocks versus time spent coalescing blocks for 256MB heap.

In the 30 test runs we observed that, 20 of the test runs spent more time (in aggregate) attempting to coalesce blocks, than splitting blocks.

This gives rise to the following hypothesis: coalescing blocks is slightly more expensive than splitting blocks, and potentially inherently so.

9.14.11 Malloc - Free State Transition Diagram

The following diagram shows the state transition diagram for block allocation and freeing.

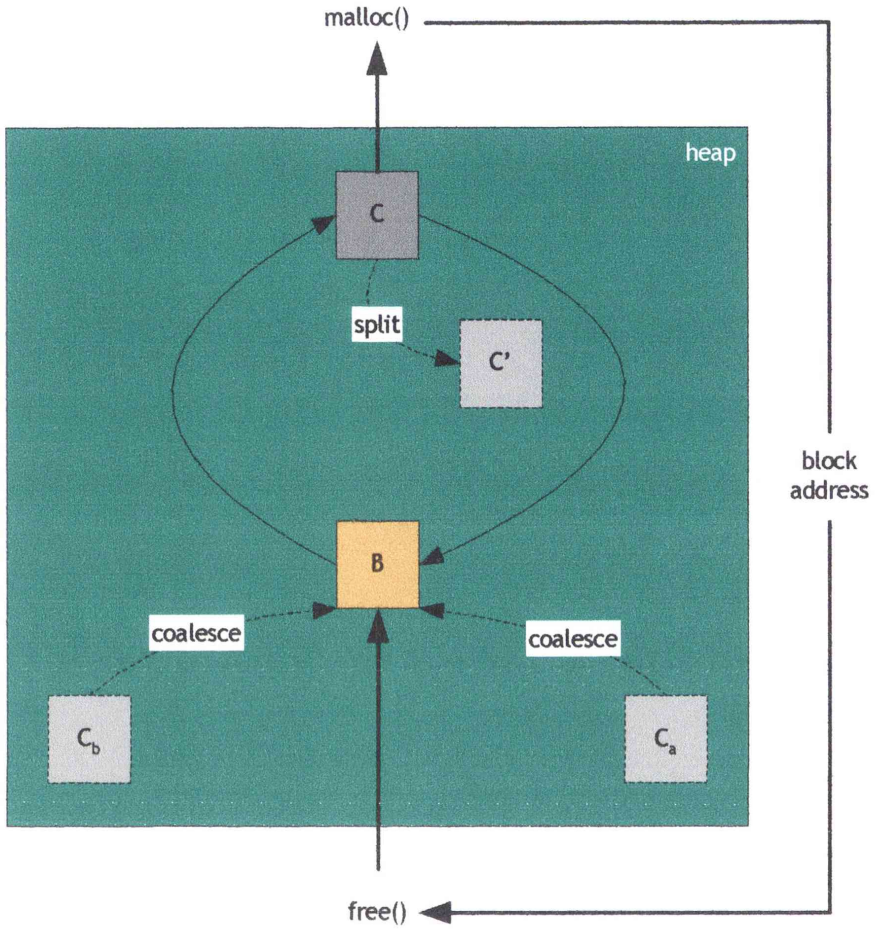


Figure 9.14: Malloc – Free state transition diagram.

When an allocation request is received (a malloc() is issued), the system finds a suitable free chunk **C** and tests if it can be split. This produces block **B**, and may produce a new free chunk **C'**. Thus an allocation has one test for split, and a possible action.

Freeing on the other hand (**B** → **C**) requires two tests for coalescing: the memory piece $\mathbf{C_b}$ before the block being freed needs to be tested, to see if it is a free chunk which can be coalesced, with the block being freed, and the memory piece $\mathbf{C_a}$ after the block being freed has to be tested as well.

Any algorithm must rightly assume that, there is no correlation in the order of frees, with respect to the order of allocation. Consequently, each and every free requires

testing the memory piece before and after the block being freed. Thus a free has two tests for coalescence, and two possible actions. The system cannot coalesce more than it splits, but nevertheless the two tests must be done for each free.

Any heap manager which performs continuous coalescing cannot escape from this fact.

9.14.12 A Fourth Maxim of Scalability

This observation seems to have gone unnoticed in the literature. Allocation costs are discussed in virtually every paper, but there is little to no discussion on the time/cost to free. The only paper we have found which reports on the number of instructions to free blocks is by Detlefs et al. [DDZ1993]. In this paper there are times when the cost of freeing is greater than the cost for allocation, but the authors make no explicit comment. There is discussion [WJNB1995] about wasting CPU time in coalescing blocks which are only subsequently split again, but there does not seem to be any discussion anywhere on the relative costs of splitting versus coalescing.

The reason for this may be historical – the emergence of heaps predates multithreaded multiprocessor systems. In a single threaded program on a single processor machine, any subtle imbalance between the CPU cost to allocate and that to free is, of no real consequence and is likely to go unnoticed – while a block is being freed in such systems the system cannot be allocating a block. On a multiprocessor machine, however, where parallelism occurs, if the cost of freeing a block is, or at times becomes, greater than that of allocation, then there may be times when the heap becomes starved of its memory resource – the rate of demand can outstrip the rate of supply.

Consequently, we propose a fourth maxim of scalability[115]:

- Inverse operations cannot take more time-space than the original operation.

A system, such as a heap, can be viewed as a set of *do* operations, and a set of inverse *undo* operations. Free(), for example, can be viewed as the inverse of malloc(). Within such a system, if the inverse operations are computational more expensive than the original operations, then the system would not be scalable.

---

[115] Refer to Chapter 8 for the other three maxims of scalability.

9.14.13 A Comment on Entropy

Under what conditions could we see, or expect to see, the cost for freeing become more than the cost of allocation? The simplest answer is when there is a higher degree of entropy.

When blocks are allocated and freed in a pattern, such that coalescing tends to occur less frequently (such as we see in Group C), then the time spent testing for possible coalescence is wasted: the tests tend to fail, because the neighboring blocks are not free – they are in a working set which belongs to a different thread. Block coalescing is anti-entropic – the system expends energy (in the form of CPU cycles) to reduce randomness, to reabsorb the information which was published at the time of allocation, striving to ultimately return to its original state. It is quite likely, however, that most long running systems tend never to return to their original state.

The simple observation is that it generally takes less effort to create a mess than it does to clean it up[116]. This correlates with the Second Law of Thermodynamics in physics.

We therefore propose that block coalescence rate may be a measure of entropy in a heap.

9.14.14 Entropy, Cache and the Cost to Free

When a block is freed, if either neighboring blocks are not in cache, then those blocks have to be brought into cache, before the coalescence test can be performed. Two things happen when this occurs: the CPU stalls and other blocks have to be evicted from the cache.

An increase in entropy would tend to disturb the spatial locality, afforded by the fact that sub-heaps are a region or interval of memory. Bringing adjacent blocks into cache is of little use if they belong to different working sets – there is no systemic reason why they will be accessed again in the short term.

---

[116] The author has built a parallel concurrent garbage collector [Hue2001]. The CPU cycles required to identify and remove garbage in parallel to user threads creating garbage became inordinately large, ultimately causing that particular system not to be able to scale much beyond 10GB.

Bringing blocks into cache from different working sets would also tend to disturb the temporal locality of all threads. Some blocks, a subset of which may be accessed again in the short term, have to be evicted from cache to make way for the adjacent blocks.

A heap that has a higher degree of entropy (coalescing tends not to happen) should therefore tend to have a higher cost of freeing, due to such cache effects.

Refer to Figure 9.12. Observe that the free curves shift progressively upwards (from left to right). We see an increasing cost to free, which corresponds with an increasing degree of entropy, within the cache itself – refer Figure 9.11 as a measure of entropy.

What we have just observed then, in microscopic detail, is the time-space tradeoff as it occurs dynamically within the cache of a machine. Entropy of the heap in main memory naturally manifests as entropy in cache, with causal affects on performance.

9.14.15 Systematic v. Stochastic Processes

In 1977, John Shore [Sho1977] observed anomalous behavior in Knuth's Fifty-Percent Rule [Knu1973]. Mathematical analysis and simulation experiments lead Shore to conclude that, not only was fragmentation the root cause of the unexpected behavior, but that fragmentation is fundamentally caused by the difference in the order that blocks are freed, compared to the order the blocks were first allocated:

> *"...it is hypothesized that the anomalous behavior results from the combined affects of systematic placement and the statistics of the release process. Additional simulations support this hypothesis. Systematic placement, which refers to the natural convention of always allocating storage requests against the same end of the free block selected by the allocation strategy, tends to order blocks within contiguous groups according to their allocation time. The degree of anomalous behavior depends on the extent to which allocated blocks are released in the order of their allocation. For non-Markovian release processes, the extent of the correlation between allocation order and release order varies approximately inversely with the coefficient of variation of the memory residence time distribution."*

What is appealing about Shore's perspective is that he clearly observes a) the fundamental difference in the nature of the two processes: allocation is systematic while

deallocation is stochastic, and b) that the difference in these sequences has a causal and measurable effect. We believe it is worth pausing briefly to model this.

The following state transition diagram shows a heap in some continuous dynamic steady state **h**.

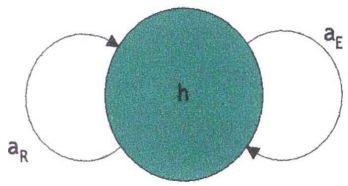


Figure 9.15: A heap in a dynamic steady state.

An allocation request stimulates the heap to emit an address $\mathbf{a_E}$. Releasing (freeing) the block returns an address $\mathbf{a_R}$ to the heap which absorbs it. Note that in a continuously executing system, an absorbed address is a candidate for reemission.

In the following diagram, we show the continuous interaction of a heap emitting addresses to, and absorbing addresses from, a multithreaded system.

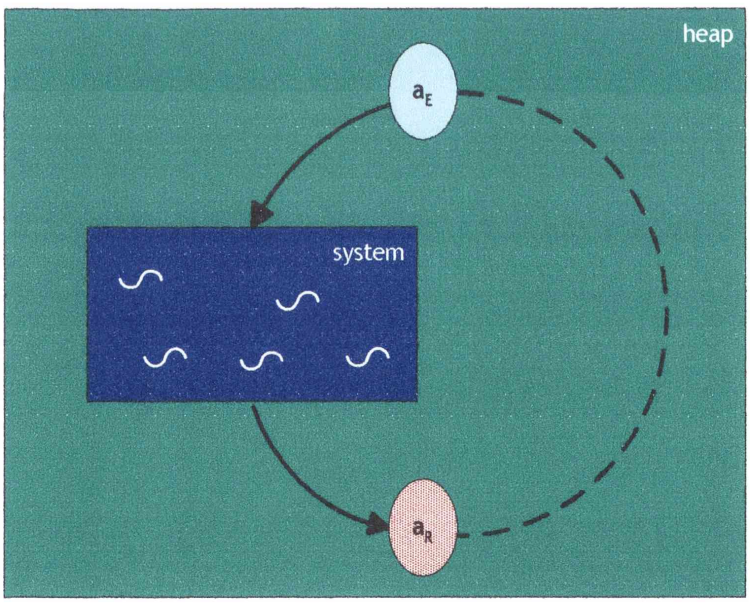


Figure 9.16: The continuous interaction between a heap and a multithreaded system.

Over any particular time interval, a heap will emit a sequence of addresses $[\ \mathbf{a_E}\ ]$, and it will absorb a sequence of addresses $[\ \mathbf{a_R}\ ]$. As described by Shore, the allocation process is systematic, while the release process is stochastic. Consequently, $[\ \mathbf{a_R}\ ]$ may or may not be exactly the same as, or equivalent to $[\ \mathbf{a_E}\ ]$ over a particular interval.

What has piqued our interest is that in SMP scalable heaps, because of parallelism, there are multiple focal points associated with the systematic placement – threads in RHmalloc attempt to allocate out of the same bounded memory region that they last allocated from.

It is beyond the scope of this thesis but it is worth conjecturing that, depending upon the geometry of the heap, the hardware configuration and the specific sequences involved, dynamic patterns [Kel1995] must emerge – some leading to stable behavior with lower degrees of fragmentation, others leading to stable behavior with higher degrees of fragmentation, while others leading to unstable patterns (such as those leading to exhaustion). Triggered by changes in a programs' request/release behavior, phase changes in these dynamic patterns within the heap itself should also be observable[117].

Loosely speaking, what Shore observed in his experiments was that, as $[\ \mathbf{a}_R\ ]$ *deviates* from $[\ \mathbf{a}_E\ ]$ fragmentation increases. The immediate question is: what do we mean by deviate? This is a complex question. For example: any number of release sequences may be equivalent (strongly or weakly) to a particular allocation sequence, if they ultimately produce the same result. How long then is a valid sequence? Do we measure such deviation during the sequence or at the end? Are there time delays or phase shifts which need to be considered in comparing such sequences in a multithreaded SMP environment?

Clearly, one cannot readily compare an unbounded systematic sequence, directly with an unbounded stochastic sequence, in such an environment. But we may be able to observe their interaction/interference and the effects thereof. So rather than trying to define *deviates* quantitatively, it may be more fruitful to define it qualitatively.

Firstly, let us define two types of interaction sequences:

- **Totally-ordered** – the sequence of returned addresses, exactly matches the sequence of allocated addresses (i.e. $[\ \mathbf{a}_R\ ] = [\ \mathbf{a}_E\ ]$); and
- **Totally-random** – the sequence, in which addresses are being returned, is completely random with respect to any sub-sequence of emission – there is no

[117] And probably visualizable given the right tools.

information which can be used to determine the next likely address to be returned (i.e. $[\mathbf{a_R}] \neq [\mathbf{a_E}]$).

In a totally-ordered interaction sequence, the calling application has effectively recorded the allocation sequence. The information has not been lost, since it is reabsorbed by the allocator when the blocks are freed, leading to lower overall (potentially zero) fragmentation.

In a totally-random interaction sequence, the calling application either has not recorded the allocation sequence, or because of application issues/idiosyncrasies, the addresses are not returned in any equivalence of allocation order. Since the information is effectively lost, this has the potential to lead to higher overall fragmentation.

With this perspective, we can now define three practical differences between a given request sequence $[\mathbf{a_E}]$ and a given release sequence $[\mathbf{a_R}]$, namely:

- **Cluttering** – there is a practical difference between the two sequences, observable as the tendency to increase entropy;
- **Stable** – there is no practical difference between the two sequences, observable as no real change in entropy; and
- **Tidying** – there is a practical difference between the two sequences, observable as the tendency to decrease entropy.

This brings us to state the three following hypotheses:

1. Entropy changes in any continuously executing system, when the sequence of undo operations practically differs from the sequence of do operations;
2. Entropy manifests itself in the dynamic patterns of the system, and said patterns will undergo phase changes when there are practical changes in the difference between such sequences; and
3. Entropy, and the changes to entropy, will affect the performance of the system.

These observations and hypotheses are strong indicators for future work.

9.15 A Final Comment on Our Testing Approach

There is much discussion in the literature about what constitutes a valid or solid test for dynamic memory management. We chose three relatively simple tests, but exhaustively varied the configuration of the heap, and the degree of concurrency. This produced a wealth of data, which could be examined and understood in detail – producing fruitful results.

We cannot find a similar approach in the literature, and suggest that such a test approach should be considered by others, so as to supplement their methodology.

9.16 Overall Conclusions

We proposed a series of tests on a 32-bit machine, and showed by extraploation, that RHmalloc would scale to at least to 1PB on a 1,024 CPU machine[118] because:

- The within sub-heap response time **r** is O(1), being independent of the number of blocks **b** and the block size **B**;
- The time **f** to find a suitable sub-heap to satisfy a request is O(1), being independent of the number of sub-heaps **s** and the sub-heap size **S**;
- Sub-heaps are arranged and selected so as to minimize within sub-heap contention, such that over time and for most cases **f** + **r** remains independent of the number of threads **t** in the system; and
- Test cases showed a max(**S**) of 1GB and max(**s**) of 1M could be supported with consistent overall response times **R** for up to 1,024 threads.

Furthermore, since we have shown that within RHmalloc **R** = **f** + **r** is independent of **B**, **b**, **S**, **s** and tends to remain independent of **t,** then we have shown theoretically that dynamic memory management scales indefinitely.

We also observed that the rate of block coalescence may be a measure of entropy in a heap.

In the next and final chapter, we state our overall conclusions on dynamic memory scalability and discuss possible future work. We also provide a chapter-based summary of this thesis.

[118] Assuming the machine linearly scaled to this size.

10 Conclusion and Summary

We present our final conclusions on the scalability of heaps, as well as describe the various directions future work may take. We also supply a chapter based summary of this thesis.

10.1 Overall Perspective

In order to get a sense of the scale of our result, observe the following diagram, which shows a logarithmic graph of memory plotted against CPU's.

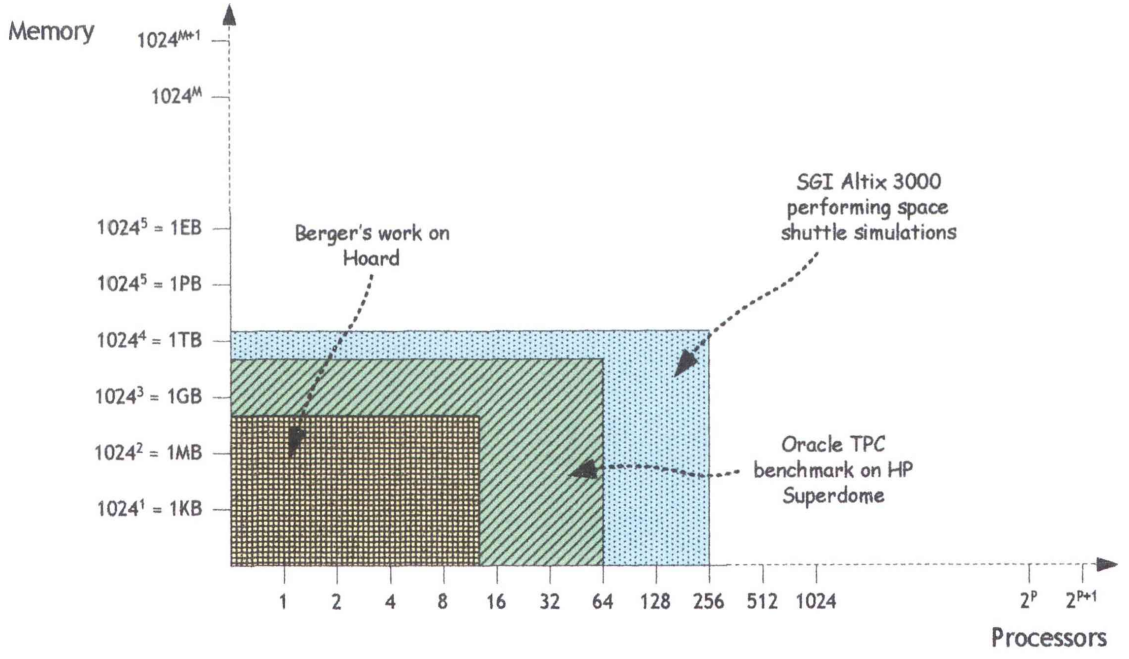


Figure 10.1: CPU and memory plot indicating relative size of various activities.

The diagram depicts three boxes. The outermost box represents one of the largest general purpose SMP machines available today – the Altix 3000 by Silicon Graphics. This particular machine may have up to 256 Itanium 2 processors and 4TB of memory, and is capable of running a single instance of Linux. The three largest of these machines in existence are currently at NASA performing space shuttle simulations in order to identify systemic faults in the craft. The middle box represents the machine upon which Oracle recently set the world's fastest TPC-C benchmark – a HP

Superdome with 64 Itanium 2 processors and 750GB of memory. The innermost box represents the largest heap reported in the literature to date – 600MB on a 14 processor Sun Enterprise 5000 by Emery Berger using a DMM known as Hoard, circa 2000.

Our extrapolation result encompasses all three.

10.1.1 Heaps are Provably Scalable

As shown in this thesis heaps are scalable with respect to CPU and memory:

(a) A heap can be divided into an indefinite number of sub-heaps each of finite size **S**, arranged such that within any given sub-heap the response time **r** is O(1), being independent of the number of blocks **b** and the block size **B**;

(b) A heap can be constructed as an indefinite number of sub-heaps **s**, arranged such that the time **f** to find a suitable sub-heap to satisfy a request is O(1), being independent of the number of sub-heaps **s** and the sub-heap size **S**; and

(c) Sub-heaps can be arranged and selected so as to minimize contention, such that over time and for most cases **f** + **r** remains independent of the number of threads **t** in the system.

Note, this does not mean that a system which uses a heap is automatically or by definition scalable. Dynamic memory management enables a system to create and destroy objects during execution. The constituent structures and algorithms of those objects are logically separate from those used by a heap; and those structures and algorithms may, or may not, scale as we increase the multiplicity of underlying hardware resources.

10.1.2 Why are Heaps Scalable?

Heaps are scalable on SMP equipment because of two fundamental reasons:

1. Structurally a heap can be constructed such that it has two degree's of freedom: it is simultaneously a divisible and an expandable entity. A heap can be divided into an indefinite number of sub-heaps, while each sub-heap can be expanded to any size; and

2. Algorithms exist which are O(1) in locating a sub-heap (being independent of the number of such resources in the system) and O(1) in using the sub-heap (being independent of the size of the individual resource).

By divisible entity, we mean a resource (such as a data structure) that can be divided into two or more separate entities, such that there is no information lost in doing so, nor is there information shared between the resulting new entities. The division is a perfect separation of concerns: the new entities are completely independent of each other. From the perspective of a thread/CPU dealing with one such entity, it is irrelevant how many other entities exist in the system.

By expandable entity, we mean that it is a resource that can, in principle, always be made larger irrespective of the number of such resources in the system.

The following diagram is a stylized depiction of a heap, and how it can be divided and expanded into an arbitrary number of sub-heaps of arbitrary size, as required.

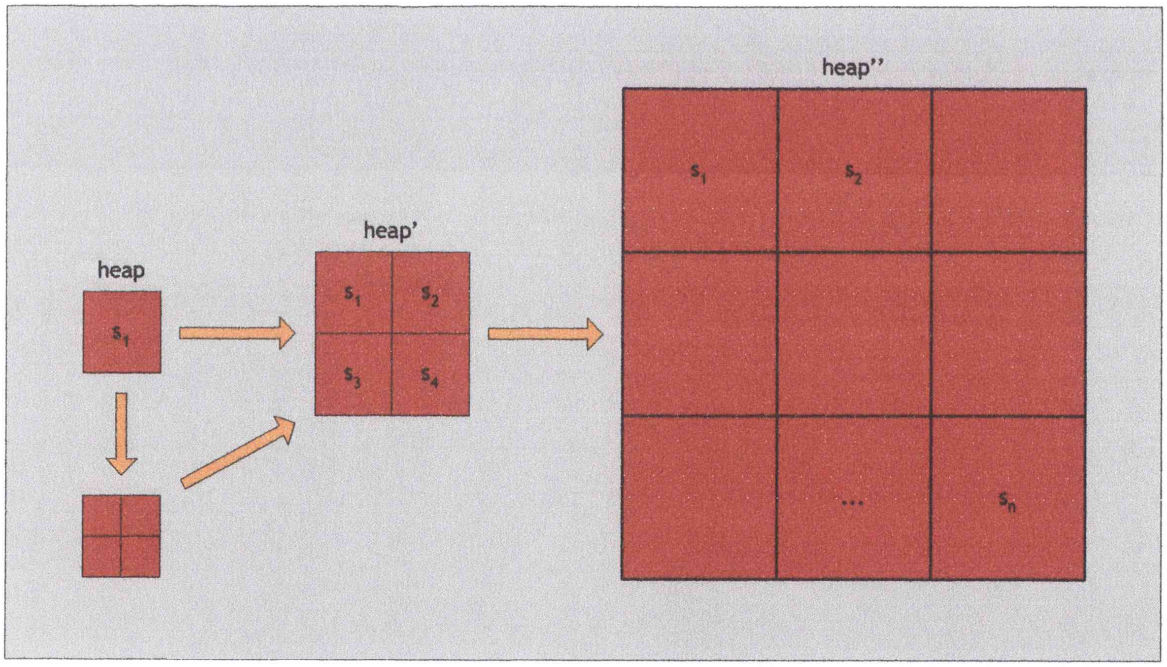


Figure 10.2: A heap is simultaneously divisible and expandable.

Consequently, as we add CPU's into the model, a heap can be continuously divided into more sub-heaps in order to handle more degrees of parallelism, while as we add memory into the model more sub-heaps of the same size can be added, the existing sub-heaps can be made larger, or both.

10.1.3 Why is any Scalable System Scalable?

Having explored the scalability of heaps at length, it may be reasonable to hypothesize that any system, which is scalable, will probably be found to have at least two degree's of freedom in its architecture, that are concordant with the freedom to vary CPU and memory.

Therefore, an indefinitely scalable system (or a sub-class thereof) potentially may be one, whose architecture divides its computational time-space into three concerns:

1. One or more finite but unbounded resources whose response time or usage is O(1);
2. Managing an indefinite number of such resources in a manner that finding a suitable resource is O(1); and
3. Resources can be configured (size and quantity), and selected in a manner such that over time, any affect of resource contention on response time is effectively zero.

10.1.4 Four Maxims of Scalability

Finally, in Chapter 8 we explored the space of potential solutions for highly scalable dynamic memory manager. Because the space was large, we derived four guiding principles or maxims about scalable systems, to assist our design process. As way of closure to this thesis, we believe it is worth restating them in more general terms.

In a scalable system one might expect:

1. Each operation to be an action which is finite in some manner, which does not change as we increase the underlying hardware capacity;
2. There must be an increasing multiplicity of shared resources, as we increase the underlying hardware parallelism;
3. That resources in the system will tend to remain balanced over time; and
4. Inverse operations not to take more time-space than the original operation.

10.2 Future Directions

10.2.1 Heaps in General

There are a number of research areas which remain open for very large heaps – as well as heaps in general. These questions tend to stem from the fact that, the testing and exploration in the literature to date, has revolved around a very small number of programs, which are not representative of the gamut of contemporary applications.

The following questions are of interest:

1. Does memory fragmentation change as significantly more memory is used? Most of the discussion on fragmentation has been prompted by observations on a restrictive set of programs, whose memory consumption is best measured in megabytes. It is interesting to muse on the behavior of heaps at the other end of the 64-bit spectrum. On a future machine with one exabyte of memory[119], will there be so many placement choices, that fragmentation has trended to zero? Refer to §4.3.13 and Coffman et al.'s conjecture in [CKS1985];

2. Do the characteristics of dynamic memory managers remain constant/consistent over very long periods? Again, most of the discussion on performance and fragmentation has been prompted by observations on a restrictive set of programs, whose total execution time is best measured in minutes. The properties of long-running systems remain unexamined – and not just from the heap perspective; and

3. Of particular interest is how, or to what degree, does program phase behavior affect a very large or very long running heap? In a multithreaded application different threads may be exhibiting different, potentially conflicting, phase behaviors at the same time. Phase behavior of programs may in fact be key to

[119] RHmalloc running on a machine with 1EB of memory would have 16 million 16MB sub-heaps.

understanding the nature of very large/very long running systems, and guaranteeing certain confidence levels of safety, correctness and robustness[120].

10.2.2 Scalability in General

In researching the scalability of heaps, we made an observation in passing that there appears to be no formally accepted definition of SMP scalability. If this is true, we believe it may be of significant interest to:

- To formally define SMP scalability – possibly from first principles;
- Potentially identify a classification taxonomy of scalability;
- Try to identify any inherent properties in those systems, which by definition, would characterize them as belonging to a particular class of scalability;
- For those systems identified as indefinitely scalable, examine their architecture to see if they have a formal separation of concerns as we have postulated; and
- Examine standard algorithms and data structures, and show how they can be transformed into SMP scalable algorithms.

---

[120] It may be possible for a *'phase behavioral algebra'* to be developed. A first step may be to examine the behavior of traditional data structures from the perspective of the heap when those structures are manipulated under certain conditions in order to see if there are basic usage patterns. A second step may be to examine the behavior of transactional systems [BN1997, GR1993] – the transaction unit can be viewed as a time-work quantum.

10.3 Conclusion

In relation to DMM, in this thesis we have a) explained the basic concepts and fundamental techniques; b) discussed the melting pot of engineering tradeoffs, c) surveyed the history and dissected the state of the art; d) concluded that no allocator has been designed and built to scale with respect to CPU and memory; and e) presented our technical requirements for such an allocator, one which should exhibit efficient and predictable, linearly scalable O(1) time-space behavior.

Observing that the solution was non-obvious, we were motivated to explore the solution space via a set of 14 thought experiments. Our solution, known as RHmalloc, is based on the perspective that a highly scalable heap can be viewed as an unbounded set of finite sized sub-heaps, where each sub-heap is concurrently shared by any number of threads; such that a suitable sub-heap can be found in O(1) time and an allocation from a suitable sub-heap is also O(1).

Testing the design properties of RHmalloc, we were able to show by extrapolation that RHmalloc will scale to at least 1,024 CPU's and 1PB; and we have theoretically proven that dynamic memory scales indefinitely, with respect to both CPU and memory.

10.4 Summary

We present a chapter based summary of this thesis.

10.4.1 Chapter 1 – Introduction

Chapter 1 of this thesis introduced the topic of dynamic memory management. We used a hypothetical example of a hand held device as the context to discuss the fundamental issues of: allocation, freeing, coalescing, splitting, exhaustion, fragmentation, compaction and expansion. We also discussed the typical constraints placed upon a conventional allocator – such as the lack of *a priori* information and the inability to compact memory.

10.4.2 Chapter 2 – Basic Techniques

Chapter 2 presented a catalog of basic dynamic memory management techniques. We discussed the general or specific applicability of allocators; whether allocators have a free form or patterned view of memory; whether they partition memory and have mixed or segregated pools. Allocators can track free space via a number of techniques such as: free lists, vectors of free lists based on exact or banded sizes, trees of free lists or bitmaps. When using lists an allocator can maintain those lists in FIFO, LIFO, size order or address order.

Allocators implement an allocation policy to determine how to satisfy an allocation request. Known allocation policies include: first fit, next fit, best fit, worst fit and exact fit. Having chosen a particular block an allocator may always split the block, if it is larger than requested; may only split the block, if it is larger by some pre-determined threshold; or the block may never be split. When a block is freed by an application, the allocator may always attempt to coalesce that block with neighboring free blocks, in order to create larger units of free space; it may defer the act of coalescing until some other trigger occurs (such as exhaustion); or it may never coalesce.

Some allocators (particularly those which are in a handle based environment) have the liberty to compact memory, while other allocators (notably those in an address based environment) can not compact memory.

When an allocator requests more memory from the operating system (such as requesting more virtual memory pages) the allocator may be designed to preserve the newly available space as long possible, or to immediately carve it up into blocks of particular sizes. Conversely, some allocators are designed to reduce overall memory consumption, by returning virtual memory back to the operating system whenever possible, while other allocators are not so designed.

In order to track individual memory blocks, some allocators use header and/or footer boundary tags on each and every block. This is generally the case when the allocator has a free form or mixed pool view of memory. Allocators are generally designed to cater for minimum and maximum block sizes, as well as only allocate blocks in multiples of a discrete unit (in order to cater for word alignment issues).

Finally some allocators are single threaded, in that they only allow one thread at any given moment, to be allocating or freeing, while others are multithreaded allowing multiple threads to allocate or free at any given moment.

10.4.3 Chapter 3 – Engineering Issues

Chapter 3 discussed the engineering aspects involved with the time-space tradeoffs of allocator design. In designing an allocator, one must take into account that programs exhibit patterned behavior (in terms of the specific sizes of blocks they request, the relative and absolute quantities of those block sizes, and the patterns in allocating and freeing), as well as the fact that program behavior can also go through phase changes (where the size and quantity patterns associated with allocation and freeing change).

The act of freeing, in a non-compacting heap leads to fragmentation – unusable free space scattered throughout memory. Fragmentation occurs in two forms: external (where there is recorded free space in the heap but none of it is of a size which is usable) versus internal (where due to implementation issues more space is allocated per request than needed). Fragmentation is insoluble, due to the discrete nature of digital machines and is caused by isolated freeing exacerbated by time-varying program behavior.

Fragmentation however can be managed, and there are a number of different allocation policies which yield acceptably low degrees of fragmentation. In designing an

allocator, one must also take into account spatial locality (memory locations near each other tend to be referenced close together in time) as well as temporal locality (memory locations which have been accessed more recently tend to be accessed again).

An allocator which implemented an address order allocation policy may provide very good spatial locality to the associated program. An allocator which implemented a least recently used allocation policy (LIFO) may perform well because the memory blocks may still be in cache – leading to fewer cache misses/CPU stalls.

Performance is an important design issue. Simple algorithms may provide acceptable performance in constrained, single threaded environments but typically fail to scale to very large environments. Due to the discrete nature of digital machines, there are alignment issues relating to where data values can placed. Boundary constraints of this kind must be taken into consideration otherwise bus errors occur; typically meaning that the size of blocks actually allocated must be in multiples of the largest constraint. Consequently, alignment is a systemic cause of internal fragmentation. Within an allocator there will always be some degree of overhead in order to create and track the heap itself.

Scalability has two aspects: CPU scalability (how does request throughput change as we change the number of CPU's in the system) and memory scalability (how does response time change as we change the size of the heap).

Concurrency has two aspects: degree of concurrency (single or multithreaded) as well as fineness of concurrency (a coarsely grained allocator does not permit concurrent access to internal data structures, while a finely grained allocator permits concurrent access to internal data structures).

Dynamic memory management is a pervasive technique used by virtually all programs and systems. Consequently, the robustness and reliability of an allocator affects the robustness and reliability of all programs and systems using it. There are two schools of thought on allocator evaluation: synthetic tests (where a test harness uses one or more mathematical distributions to drive the allocator) or real tests (where real world programs are used to drive the allocator).

10.4.4 Chapter 4 – History of the Art

Chapter 4 presented the fifty year history of dynamic memory management and its association with data structures and operating systems[121].

In the 1950's the hardware and software environments were significantly different from that of today: machines had no explicit program counter, programming environments had no support for strings or data structures. Dynamic memory management had not yet emerged. FORTRAN I did not address any of these issues. IPL-II had the first embryonic support for dynamic memory management – a list of free cells. LISP in 1959 provided sophisticated symbolic list manipulation capabilities, dynamic memory management and is credited with being the first system that had garbage collection.

In the 1960's operating systems, languages and data structures were in their early stages of evolution. DMM was explored in very early operating systems on machines such as the IBM 7090 and the Burroughs B5000. In 1961 Ross published a paper on a technique called PLEX – effectively the data structure as we know it today. In 1962 Iliffe and Jodeit presented a DMM interface which included the notion of a memory block. In 1963 Weizenbaum presented SLIP – a FORTRAN library which implemented the list data structure. In 1964 Comfort first uses a vector to track free space. In 1969 Randell made the distinction between external and internal fragmentation.

In the 1970's many researchers focused on developing new DMM techniques, while others examined the efficiency and fragmentation of those techniques – some doing so in a theoretical manner, others empirically. The research in this decade identified, among other things, the patterned behavior of programs, the buddy class of allocators, as well as the idea of using bitmaps to track and/or summarize free memory,

In the 1980's there was a significant convergence leading to the form of the DMM sub-system we known today. Significant work was done in trying to improve buddy systems, only to show their memory efficiency was inherently poor. This was coupled with continued research into more sophisticated techniques focused on reducing fragmentation and improving performance through improved cache efficiency.

---

[121] We also refer the reader to Appendix C for a timeline on the history of DMM.

In the 1990's the first early examples of scalable DMM work were done. Researchers also explored the idea of analyzing program structure and behavior, in order to provide predicative assistance in improving cache efficiency, and further minimize fragmentation; this lead to a number of customizable, as well as specialized allocators, being developed. Discussions on the validity and appropriateness of synthetic versus real traces/testing emerged.

In 2000 Berger et al. again observed that very little work has been done on concurrent allocators and presented a concurrent, CPU scalable allocator called *Hoard*. Also in 2000 Seidl discussed a memory allocation system called *Avon* – the objective was to improve execution performance by identifying and capitalizing on the skew in reference density between objects. In 2001 Printezis and Jones presented an adaptable heap visualization framework known as GCSPY. In 2004 Michael presented a lock-free CPU scalable allocator which used atomic test-and-set operations instead of spin locks.

10.4.5 Chapter 5 – State of the Art

Chapter 5 examined the state of the art in dynamic memory management. We began by defining our terminology. While there have been many allocators built over the years we chose DLmalloc, PTmalloc, LKmalloc and Hoard as a representative sub-set to examine.

We recounted Berger's taxonomy of serial single heap, concurrent single heap, pure private sub-heaps, private sub-heaps with ownership and private sub-heaps with thresholds to establish a framework for comparison. We then systematically dissected and compared DLmalloc, PTmalloc, LKmalloc and Hoard using this taxonomy.

In summary DLmalloc is classified as a serial single heap, and is based on a dual vector of lists approach known as binning with deferred coalescing. DLmalloc does not scale with respect to CPU and memory.

PTmalloc is a derivative of DLmalloc, and is characterized as private sub-heaps with ownerships, where each thread-safe sub-heap is an entire DLmalloc heap. Searching for a sub-heap is O(n), with simple pathological cases leading to premature memory exhaustion.

LKmalloc is classified as private sub-heaps with ownerships, where each sub-heap is also dual vector of lists with each list guarded by a lock. Threads are permanently assigned to a specific sub-heap, which can lead to premature memory exhaustion.

Hoard is classified as private sub-heaps with thresholds. Hoard views a heap as a set of sub-heaps, where each sub-heap manages only one fixed size. Hoard moves sub-heaps from private to common pools based on threshold limits, in order to bound blow-up. Hoard trades memory utilization to achieve concurrency.

We ended this chapter by observing that work on heap scalability to date, has been with respect to CPU only, and that no work has been done on scalability with respect to both CPU and memory.

10.4.6 Chapter 6 – Technical Issues

Chapter 6 discussed the issues relating to very large, highly concurrent dynamic memory management by first defining the terms *very large* and *highly concurrent*. We then presented a first order calculation, showing time to allocate memory given CPU speed, in order to convey a sense of scale.

We then argued that, for a very large, highly concurrent dynamic memory manager to be feasible, it would have to show O(1) performance with respect to both CPU and memory. We then discussed practical engineering considerations such as time and space efficiency, and their tradeoff. We then formalized our requirements.

With consideration to these requirements we defined a very large, highly concurrent dynamic memory manager as one which exhibits efficient and predictable, linearly scalable, O(1) time-space behavior.

10.4.7 Chapter 7 – Design Discussion

Chapter 7 conducted a design discussion, which used a set of thought experiments to explore the solution space. We described the philosophy of our approach as one which acknowledged the valuable lessons in the literature, and that we were looking for a solution based on combining or extending existing techniques.

We restated our fundamental objective, and then broke that objective into two separate concerns: namely the requirement for service times to be independent of memory, as

well as the requirement for request throughput to be directly proportional to the number of CPU's in an SMP environment. From this we derived two guiding principles: each operation must be an action which is finite in some manner, which does not change as we add more memory into the model; and secondly, that there must be an increasing multiplicity of shared resources, in order to maintain throughput as we add more CPU's into the model.

We then proceeded through 14 different thought experiments, during which we arrived at a third guiding principle: resources in the system will tend to remain balanced over time. A critical insight was gained when it was realized that, sub-heaps could be indexed in an O(1) manner by the size of the largest free block they contain.

Those 14 thought experiments considered solutions based on: a single locked list, a vector of locked lists, a dual vector of unsorted locked lists, a dual vector of locked lists with one vector of lists kept sorted by size, one very large vector of locked lists covering all possible sizes, one private sub-heap per thread, private sub-heaps with ownership, multiple concurrent sub-heaps with ownership, common sub-heap accessible by all threads plus a sub-heap private to each thread, multiple common sub-heaps accessible by all threads plus a sub-heap private to each thread, a consisting of only a set of common sub-heaps, using two vectors to track sub-heaps, one single vector to track sub-heaps, and finally a single vector of sub-heaps with a two level bit index.

We explored how to detect excessive thread contention, and how to efficiently move sub-heaps between lists; as well as noting how the virtual memory region being managed by a sub-heap, can be returned to the operating system, when all blocks in that sub-heap become free.

We concluded by presenting a detailed first order sketch of how a very large, highly dynamic memory manager can be viewed as a unbounded set of finite sized sub-heaps, commonly accessible to all threads, in an arrangement such that, a suitable sub-heap can be found in constant time, and a suitable block can be found within a sub-heap in constant time.

10.4.8 Chapter 8 – Formal Specification

Chapter 8 presented a formal specification for RHmalloc – a very large, highly concurrent dynamic memory manager. We restated our view that, a very large heap is an unbounded set of finite sized sub-heaps. Two data structures emerged: the heap control block and the sub-heap.

In RHmalloc sub-heaps are bounded, non-overlapping regions of virtual memory. Sub-heaps are linked together to form a list maintained in address-order. Although sub-heaps are fixed in size, they can be made any particular size. RHmalloc only directly manages up to a pre-determined block size, above which such requests are handles by other operating system techniques.

Within a sub-heap a small amount of space is reserved for control information, the rest being available as block space. A sub-heap manages its space using a vector of free lists, where the length of the vector reflects the word size of the machine. A one word bit vector is used to summarize the free list vector. A vector of locks (the free lock vector) is used to individually lock each free list during access. Half of the free list vector manages exact size free lists; the other half manages banded size free lists. The banded free lists are evenly spaced. Blocks are allocated in multiples of two words. So on a 64-bit machine RHmalloc manages 65,536 discrete block sizes.

Application threads attempt to allocate blocks from a given sub-heap, until that sub-heap cannot be used to satisfy the threads current request. The application thread searches for a different sub-heap with sufficient space. If no such sub-heap exists, a new sub-heap is created. In order to find sub-heaps in constant time, RHmalloc uses a vector to organize the sub-heaps into lists according to the largest block they have – the space list vector.

There is a list for each possible, largest discrete free space in a sub-heap, and said list is a circular doubly-linked list. A multi-level space bitmap summarizes this space list vector. There is a vector of locks guarding each individual sub-heap list. After selecting a new sub-heap, threads rotate the sub-heap list to reduce the probability of contention, and allow sub-heaps to age. Sub-heaps are shifted from one list to another only when they cannot service a threads request, or when freeing produces a block which is bigger than the previous largest.

Algorithms were presented for: allocate block, free block, pop chunk, push chunk, split chunk, coalesce chunk, find sub-heap, shift sub-heap, create sub-heap, destroy sub-heap, and initialize heap.

10.4.9 Chapter 9 – Proving Scalability

Chapter 9 formally explored the CPU and memory scalability of RHmalloc. We proposed and ran three tests, which systematically examined the response time behavior of RHmalloc, as we varied both CPU and memory.

The tests were run using a version of RHmalloc instrumented to collect a variety of detailed statistics. In particular, we scrutinized a cross section of one of the tests in great detail – which yielded valuable insights into the behavior of RHmalloc, and dynamic memory managers in general.

Using the results of the tests, we showed by extraploation that RHmalloc will scale up to 1,024 CPU's and 1PB of memory. Further our mathematical approach enabled us to show theoretically that dynamic memory management can scale indefinitely.

10.4.10 Chapter 10 – Summary and Conclusion

The background research into dynamic memory management, coupled with the rigorous design and construction of RHmalloc, has shown that dynamic memory management can be made to scale to very large SMP environments – very large numbers of CPU's and very large amounts of memory. This thesis has shown that, one possible solution is to view a heap as an unbounded set of finite sized sub-heaps, commonly accessible to all threads in an arrangement such that, a suitable sub-heap can be found in constant time, and a suitable block within a sub-heap can also be found in constant time. Other solutions may exist.

Consequently, we are certain that stable and efficient dynamic memory management systems, can and will indeed, scale with respect to CPU and memory, well beyond the capabilities of 64-bit SMP architectures, into the indefinite future.

Appendix A – Notation and Conventions

Numbers

Numeric Value	Prefix	Abbreviation	Meaning
10^{18} (or 2^{60})	exa	E	quintillion
10^{15} (or 2^{50})	peta	P	quadrillion
10^{12} (or 2^{40})	tera	T	trillion
10^{9} (or 2^{30})	giga	G	billion
10^{6} (or 2^{20})	mega	M	million
10^{3} (or 2^{10})	kilo	K	thousand
10^{-3}	milli	m	one thousandth
10^{-6}	micro	μ	one millionth
10^{-9}	nano	n	one billionth
10^{-12}	pico	p	one trillionth
10^{-15}	femto	f	one quadrillionth
10^{-18}	atta	a	one quintillionth

Bits and Bytes

In this thesis we use:

- An uppercase 'B' to indicate bytes. E.g. 1TB means one terabyte; and
- A lowercase 'b' to indicate bits. E.g. 1Kb means one kilobit.

Appendix B – Reported Heap Sizes

The following table summarizes the heap sizes reported in the literature over the last decade.

YEAR	MINIMUM	MAXIMUM	REFERENCE
1993	35KB	2.1MB	BZ1993
1993	60KB	4.1MB	GZH1993
1993	21KB	3.8MB	DDZ1993
1994	17KB	2.7MB	ZG1994
1996	20KB	6.8MB	Vo1996
1996	1.3MB	104MB	Joh1997
2000	400KB	614MB	BMBW2000
2000	10KB	86MB	Sei2000

Appendix C

Timeline of Dynamic Memory Management

1950

1952

Grace Murray Hopper of Remington Rand describes the programming model of the UNIVAC.

1954

John Backus leads FORTRAN team. No support for data structures or dynamic memory.

1954

1955

1956

Newell, Shaw and Simon develop IPL-II for the JOHNNIAC.
First example of dynamic memory management.

1959

John McCarthy develops LISP. Has dynamic memory management and garbage collection.

1959

1960

1960

Perlis+Thornton develop threaded lists technique for non-recursive environments.
Work on ALGOL starts.

1961

Collins: identifies allocator as a sub-system.
Maher: describes allocator for Burroughs B5000.
Ingerman: memory pseudo-vector for Algol.
Ross: plex — the first generic data structure.

1962

Weizenbaum: develops knotted lists.
Salton: describes generic trees.
Iliffe & Jodeit: concept of memory block has emerged.

1963

Weizenbaum creates SLIP, and with it, the list data structure.

1964

Comfort generalizes the list data structure.

1964

1965

1965

Knowlton describes first binary buddy system.
Totschek performs first investigation into block size distribution.

1966

Hoare and Wirth's proposal for records in ALGOL gets rejected.

1967

Ross describes the AED Free Storage Package.
IBM enhances PL/I with DMM facilities.

1969

Randell makes the distinction between external and internal fragmentation.

1969

1970

1970

Batson examines block size distributions on Burroughs B5500.
Purdon+Stigler analyze buddy systems.

1971

Margolin researches algorithms for IBM CP-67 v3.
Robson mathematical analysis of dynamic storage allocation in general.
Isoda* use bitmap techniques.

1972

Garey* perform mathematical worst-case analysis of first fit and best fit.

1973

Knuth's classic text book is reprinted.
Hirschberg builds the fibonacci system — a variant of the buddy system.

1974

Shen+Peterson propose weighted buddy system.
Robson explores amount of storage required to guarantee operation.

1974

1975

1975

Hinds explores fibonacci buddy coalescing.
Cranston+Thomas discuss time-space problems for coalescing in fibonacci buddy system.
Shore conducts first simulation experiments.

1976

Burton uses variation of fibonacci buddy to allocate disk storage for geometrical data.

1977

Nielsen investigates performance of 35 algorithms against 18 test cases.
Shore reports anomalies in Knuth's 50% rule.

1979

McIlroy establishes the malloc interface.

1979

1980

1980

Kingsley writes a binary buddy allocator which gets distributed with BSD UNIX.

1982

Beck observes fragmentation is caused by differences in release times blocks.
Stone uses FETCH-AND-ADD instruction to implement parallel first fit.

1983

Stephenson introduces fast fits technique.
Aoki+Adams implements Stephenson's allocator.
Vo implements a best fit allocator using a bottom up splay tree.

1984

Kaufman introduces tailored list buddy system and recombination buddy system.
Bozman proposes software lookaside buffer to improve cache hit ratio.

1984

1985

1985

Oldehoeft+Allan explore an adaptive list working set technique to improve cache hit ratio.
Baker explores memory compaction algorithms.

1986

Page+Haggins propose the dual buddy system.

1987

Doug Lea begins work on DLmalloc.

1989

Brent proposes technique for implementing address ordered first fit using a balanced binary tree. Discusses separating strategy from algorithm.

1989

1990

1991

Hoskings discusses notion of a persistent heap.
Furuson* explore reducing synchronization costs between allocator and garbage collector.

1993

Iyengar first explores parallelism. Barret+Zorn explore using lifetime prediction to improve allocation performance. Grunwald explore cache locality. Detlets* profile allocation.

1994

Zorn and Grunwald question simulations which use actual versus real traces.
Haertel develops a segregated allocator, gets distributed with GNU C.

1994

1995

1995

Wilson* present a comprehensive survey and critical review of dynamic storage allocation.

1996

Iyengar tests his parallel algorithms to 130 processors.
Vo develops Vmalloc — a customizable allocator.

1997

Johnstone presents a comprehensive exploration into fragmentation and locality of 53 algorithms.
Concludes low fragmentation with good locality is achievable.

1998

Larson+Krishnan observe little work in long-running systems. Build LKmalloc which emphasizes CPU scalability.

1999

Vee+Hsu build VHmalloc designed for time-critical parallel programs.

1999

2000

2000

Berger* build HOARD — a concurrent, CPU scalable allocator.
Seidl builds AVON — improve performance by utilizing skew in reference density.

2001

Serrano+Boehm develop the KPROF profiler and the KBDB interactive heap inspector.

2002

Printezis+Jones develop GCSPY — an adaptable heap visualization framework.

2004

Huetter proves DMM scales indefinitely wrt. CPU and memory.

2004

Appendix D – Huetter's DMM Taxonomy

Having explored the nature of heaps we present a taxonomy which we believe categorizes the space of all possible dynamic memory managers.

First Order – The view of a heap

We view a heap as consisting of one or more pools of memory resources. A pool can be single or multi-threaded:

1. **Single Serial Pool** – the heap is viewed as a single pool of memory resources, within which only one thread may perform any given operation within that pool at any given point in time;
2. **Single Concurrent Pool** – the heap is viewed as a single pool of memory resources, within which multiple threads may perform various operations within that pool at any given point in time;
3. **Multiple Serial Pools** – the heap is viewed as multiple pools of memory resources, within which only one thread per pool may perform any given operation within that pool at any given point in time; and
4. **Multiple Concurrent Pools** – the heap is viewed as multiple pools of memory resources, within which multiple threads per pool may perform various operations at any given point in time.

Second Order – The nature of pools

A pool can be viewed as a programming artifact used either primarily to, first and foremost, manage a region of virtual memory (blocks over a contiguous interval of addresses) or to primarily manage a set of free blocks (the block addresses have no correlation).

Additionally, a pool can be private (permanently associated with one or more threads for their exclusive use only), personal (temporarily associated with one or more threads) or public (there is no specific thread association):

1. **Private Mapped Pool** – a pool is permanently associated with a set of threads. The pool is a programming artifact used to manage a region of virtual memory, so all blocks must be returned to the pool from whence they were allocated;
2. **Private Abstract Pool** – a pool is permanently associated with a set of threads. The pool is a programming artifact used to manage a set of free blocks, so blocks maybe returned a pool different to the one they were allocated from;
3. **Personal Mapped Pool** – a pool is temporarily associated with a set of threads. The pool is a programming artifact used to manage a region of virtual memory, so all blocks must be returned to the pool from whence they were allocated;
4. **Personal Abstract Pool** – a pool is temporarily associated with a set of threads. The pool is a programming artifact used to manage a set of free blocks, so blocks maybe returned a pool different to the one they were allocated from;
5. **Public Mapped Pool** – a pool is not permanently associated with any given threads. The pool is a programming artifact used to manage a region of virtual memory, so all blocks must be returned to the pool from whence they were allocated; and
6. **Public Abstract Pool** – a pool is not permanently associated with any given threads. The pool is a programming artifact used to manage a set of free blocks, so blocks maybe returned a pool different to the one they were allocated from.

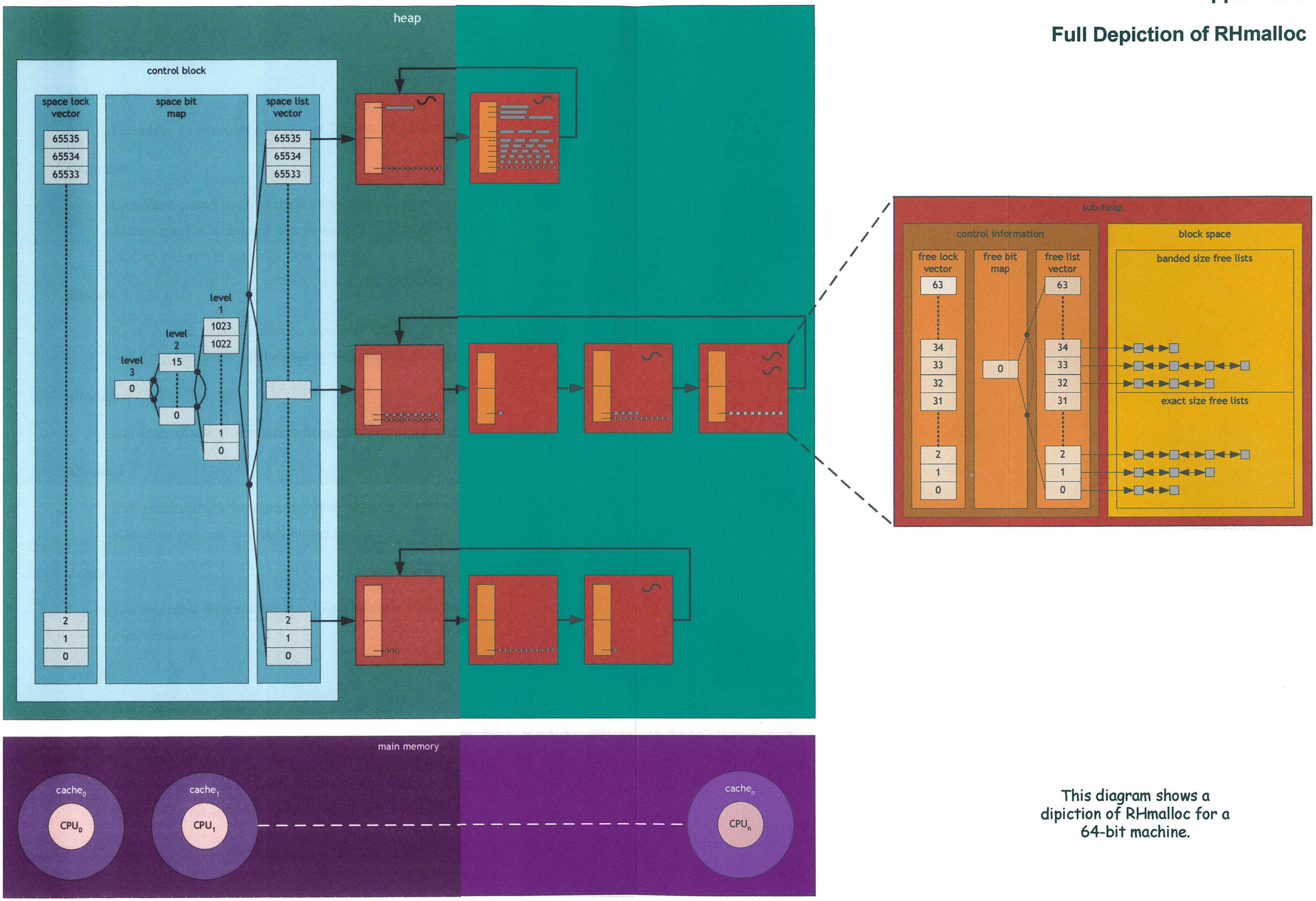
Examples

By way of example:

1. DLmalloc is a single serial pool;
2. PTmalloc is multiple serial public mapped pools;
3. LKmalloc is multiple concurrent private mapped pools;
4. Hoard is multiple serial personal mapped pools; and
5. RHmalloc is multiple concurrent public mapped pools.

Appendix E

Full Depiction of RHmalloc



This diagram shows a dipiction of RHmalloc for a 64-bit machine.

Glossary

address-ordered

Technique by which free blocks are kept (in a list) sorted by their address.

alignment

Constraint placed on the address of an object in memory – usually an object's address must be a multiple of a power of two. The bus hardware of many modern processors cannot access multi-byte objects at any arbitrary address.

allocate

Request made to an allocator to supply the address of a block of memory of certain length which can be used by the caller to store data in.

allocation

Act of an allocator satisfying a programs request for more memory.

allocator

Sub-system which manages one or more pools of memory and satisfies allocation requests to supply blocks from those pool(s).

arena

Another name for a memory pool – particularly when the allocator is layered over another.

best fit

Allocation policy that always allocates the smallest suitable free block.

binary buddy

Most common type of buddy system – all blocks sizes are a power of two. Calculating the address of a block's buddy is a simple address calculation.

binning

Vector of free lists in DLmalloc. Bins are either exact or sorted.

bitmap

A table of bits used to indicate the allocated/free state of a unit of memory or data structure.

block

A block is a unit of contiguous memory which is identified and tracked by an allocator.

blowup

Phenomena where an allocator fails to be able to reallocate previously freed blocks, consequently consuming an ever increasing amount of memory over time, ultimately leading to premature memory exhaustion.

buddy system

Buddy systems are a class of allocators which place restrictions on the size of blocks which can be allocated. Buddy systems make splitting and coalescing fast by pairing each block with a unique adjacent 'buddy' block. Buddies can be determined by a simple address calculation.

bus error

A fault or interrupt generated by a hardware bus – such as when an invalid address is issued.

caching

Act of holding blocks of commonly requested sizes in a data structure for quick allocation. Blocks held in the cache structure are typically not coalesced.

chunk

A free block; one which is available for allocation.

coalesce

The act of merging two adjacent free blocks into a single block.

compaction

The process of moving allocated blocks to eliminate free space between them.

deallocate

A request by a program to an allocator to make a block of memory available for reuse.

deferred coalescing

A policy where free blocks are coalesced some time after the blocks become free, as opposed to coalescing free blocks immediately as they are freed.

double buddies

A class of allocator which uses a pair of binary buddy systems with staggered size classes. For example one size class may be a strict powers-of-two (2,4,8,...) while the other some fixed multiple of powers-of-two (3,6,12,...).

dynamic memory allocation

The allocation and deallocation of memory according to requests made by the calling program in some arbitrary order. Required when the size, quantity or life time of objects cannot be determined *a priori* at compile time.

dynamic memory manager

Subsystem which provides dynamic memory allocation services enabling a program to create/extend it's workspace during execution, catering for the situation where memory requirements are not known *a priori* or objects have indefinite lifetimes.

exact fit

The allocation policy that uses the first suitable free block it discovers which exactly fits the size requested.

external fragmentation

The inability to use free memory to satisfy an allocation request because memory has been divided into many small blocks.

false sharing

Phenomena where multiple processors share words in the same cache line without actually sharing data. Can lead to cache thrashing.

fastbins

Name of the block caching structure in DLmalloc.

FIFO

Acronym standing for: first in, first out. Technique by which free blocks are kept in a list in the order they are added to the list.

first fit

The allocation policy that uses the first suitable free block it discovers.

fragmentation

The inability to use free memory because of the arrangement of memory already in use.

free$^{(1)}$

A request by a program to an allocator to make a block of memory available for reuse.

free$^{(2)}$

A block of memory which is available to be used to satisfy an allocation request.

free list

A single or doubly linked list used by an allocator to keep track of free blocks.

header

A fixed amount of memory (usually at the beginning of a block) used to store information about a block such as its size and state.

heap

The area of memory managed by an allocator.

heap allocation

The allocation and deallocation of dynamic memory according to requests made by the calling program in some arbitrary order.

heap memory manager

Subsystem which provides fine-grained dynamic memory allocation services enabling a program to create/extend it's workspace during execution, catering for the situation where memory requirements are not known *a priori* or objects have indefinite lifetimes.

internal fragmentation

The loss of usable memory because an allocator allocates blocks larger than that actually requested.

LIFO

Acronym standing for: last in, first out. Technique by which free blocks are kept in a list in the reverse order they are added to the list.

locality

Locality is the extent to which successive access of nearby memory locations are nearby in time.

lookaside

A (smaller) data structure which summarizes another (larger) data structure. The objective is to improve cache hit ratio by searching the smaller data structure rather than the larger.

next fit

A variant of first fit which uses a roving pointer to find a suitable fit by starting its search from the point where the last search concluded.

object lifetime

The total number of bytes allocated between the time an object is allocated when it is deallocated.

policy

A decision procedure for placing blocks in memory. Examples include first fit, next fit, best fit, worst fit and exact fit.

pre-allocation

The act of creating a set of blocks all at once for commonly requested sizes.

quick fit

Allocation technique which uses a set of free lists with one list per block size.

release

A request by a program to a DMM to make a memory block available for reuse.

scalability

The ability of a system to increase its throughput as CPU and/or memory are added to machine.

size ordered

Technique by which free blocks are kept in a list sorted by their size.

spatial locality

Memory locations whose addresses are near one another tend to be referenced close together in time.

split

The act of dividing a block into two smaller blocks.

superblock

A region of virtual memory from which blocks are allocated from.

sub-heap

A program artifact which represents a subset of the overall heap.

strategy

The underlying philosophy of an allocator which attempts to exploit regularities in the request stream in order to minimize fragmentation.

temporal locality

Memory locations which have been accessed more recently tend to be accessed again.

trimming

The act of releasing a region of virtual memory back to the operating system – thereby reducing the overall virtual memory consumed by a heap.

vector

An array of addresses; an array of addresses pointing to a set of lists.

wilderness

The name of the chunk representing the space bordering the topmost virtual memory address allocated from the operating system. So named by Kiem-Phong Vo.

wilderness preservation

Act of using the wilderness as conservatively as possible. The importance is that the wilderness should be used slowly in order to dampen the rate by which memory is extended.

worst fit

An allocation policy that always allocates from the largest free block.

Bibliography

[Abr1967] J. Abramowich. Storage allocation in a certain iterative process. In *Communications of the ACM,* 10(6):368-370, June 1967.

[AF1994] G. Attardi and T. Flagella. A customizable memory management framework. In *Proceedings of the USENIX C++ Conference,* Cambridge, Massachusetts, 1994.

[AFI1998] G. Attardi, T. Flagella and P. Iglio. A customizable memory management framework for C++. In *Practice and Experience,* 28(11):1143-1184, 1998.

[Bac1978] J. W. Backus. The history of FORTRAN I, II and III. In *History of Programming Languages*, The first ACM SIGPLAN conference on History of Programming Languages Pages 165 – 180, 1987.

[Bay1977] C. Bays. A comparison of next fit, first fit and best fit. In *Communications of the ACM,* 20(3):191-192, March 1977.

[BB1977] A. P. Batson and R. E. Brundage. Segment sizes and lifetimes in ALGOL 60 programs. In *Communications of the ACM,* 20(1):36-44, January 1977.

[BBB1964] M. J. Bailey, M. P. Barnett and P. B. Burleson. Symbolic manipulation in FORTRAN – SASP I subroutines. In *Communications of the ACM,* 7(6):339-346, June 1964.

[BBDT1984] G. Bozman, W. Buco, T. P. Daly and W. H. Tetzlaff. Analysis of free storage algorithms – revisted. In *IBM Systems Journal,* 23(1):44-64, 1984.

[BC1979] D. G. Bobrow and D. W. Clark. Compact encodings of list structures. In *ACM Transactions on Programming Languages and Systems,* 1(2):266-286, October 1979.

[BCW1985] B. S. Baker, E. G. Coffman, Jr. and D. E. Willard. Algorithms for resolving conflicts in dynamic storage allocation. In *Journal of the ACM*, 32(2):327-343, April 1985.

[Bec1982] L. L. Beck. A dynamic storage allocation technique based on memory residence time. In *Communications of the ACM,* 25(10):714-724, October 1982.

[Ber1965] A. T. Berztiss. A note on storage of strings. In *Communications of the ACM*, 8(8):512, August 1965.

[Ber1971] D. M. Berry. Introduction to Oregano. In *SIGPLAN Notices*, 6(2):171-190, February 1971.

[Ber2002] E. D. Berger. Memory management for high-performance applications. PhD thesis, University of Texas at Austin, August 2002.

[BJW1970] A. P. Batson, S. M. Ju and D. C. Wood. Measurements of segment size. In *Communications of the ACM,* 13(3):155-159, March 1970.

[BL1992] T. Ball and J. R. Larus. Optimal profiling and tracking of programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages,* pages 59-70. ACM Press, January 1992.

[BMBW2000] E. D. Berger, K. S. McKinley, R. D. Blumofe and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, November 2000), ACM, pp. 117-128.

[BN1997] P. A. Bernstein and E. Newcomer. Principles of transaction processing. Morgan Kaufmann Publishers, Inc., San Francisco California, 1997.

[Bow1963] H. J. Bowlden. A list-type storage technique for alphameric information. In *Communications of the ACM*, 6(8):433-434, 1963.

[Boz1984] G. Bozman. The software lookaside buffer reduces search overhead with linked lists. In *Communications of the ACM*, 27(3):222-227, March 1984.

[BR1964] D. G. Bobrow and B. Raphael. A comparison of list-processing computer languages. In *Communications of the ACM,* 7(4):231-240, April 1964.

[Bre1989] R. Brent. Efficient implementation of the first fit strategy for dynamic storage allocation. In *ACM Transactions on Programming Languages and Systems,* July 1989.

[Bur1976] W. Burton. A buddy system variation for disk storage allocation. In *Communications of the ACM,* 19(7):416-417, July 1976.

[BZ1993] D. A. Barret and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation,* pages 187-196.

[Cam1971] J. A. Campbell. A note on an optimal fit method for dynamic allocation of storage. In *Computer Journal*, 14(1):7-9, February 1971.

[Car1959] J. W. Carr III. Recursive subscripting compilers and list-type memories. In *Communications of the ACM*, 2(2):4-6, February 1959.

[CKS1985] E. G. Coffman Jr., T. T. Kadota and L. A. Shepp. On the asymptotic optimality of first fit storage allocation. In *IEEE Transactions on Software Engineering,* SE-11(2):235-239, February 1985.

[Cod1960] E. F. Codd. Multiprogram scheduling. Part 1 and 2. Introduction and theory. In *Communications of the ACM*, 3(6):347-350, June 1960.

[Cod1970] E. F. Codd. A relational model of data for large shared data banks. In *Communications of the ACM*, 13(6):377-387, June 1970.

[Coh1961] L. J. Cohen. Stochastic evaluation of a static storage allocation. In *Communications of the ACM*, 4(10):460-464, October 1961.

[Col1960] G. E. Collins. A method for overlapping and erasure of lists. In *Communications of the ACM*, 3(12):655-657, December 1960.

[Col1961] G. O. Collins. Experience in automatic storage allocation. In *Communications of the ACM,* 4(10):436-440, October 1961.

[Com1964] W. T. Comfort. Multiword list items. In *Communications of the ACM,* 7(6), June 1964.

[CT1975] B. Cranston and R. Thomas. A simplified recombination scheme for the Fibonacci buddy system. In *Communications of the ACM*, 18(6):331-332, July 1975.

[DDZ1993] D. Detlefs, A. Dosser and B. Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado at Boulder, Dept. Of Computer Science, Boulder, Colorado, August 1993.

[Den1970] P. J. Denning. Virtual memory. In *ACM Computing Surveys,* 3(2):153-189, September 1970.

[EPV1961] A. Evans Jr., A. J. Perlis and H. Van Zoeren. The use of threaded lists in constructing a combined ALGOL and machine-like assembly processor. In *Communications of the ACM*, 4(1):36-41, January 1961.

[FMY1991] S. Furusou, S. Matsuoka and A. Yonezawa. Parallel conservative garbage collection with fast object allocation. In *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[Fot1961] J. Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. In *Communications of the ACM*, 4(10):435-436, October 1961.

[GC1993] E. F. Gehringer and J. M. Chang. Hardware-assisted memory management. In *Proceedings OOPSLA'93 Workshop on Memory Management and Garbage Collection*, September 1993.

[GGU1972] M. R. Garey, R. L. Graham and J.D. Ullman. Worst-case analysis of memory allocation algorithms. In *Fourth Annual ACM Symposium on the Theory of Computing,* 1972.

[GHG1960] H. Gelernter, J. R. Hansen and C. L. Gerberich. A Fortran-compiled list-processing language. In *Journal of the ACM*, 7(2):87-101, April 1960.

[Glo2002] W. Gloger. Dynamic memory allocator implementations in Linux system libraries. http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html. Site visited 2002.

[GR1993] J. Gray and A. Reuter. Transaction processing: concepts and techniques. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1993.

[GZ1993] D. Grunwald and B. Zorn. CustoMalloc: efficient synthesized memory allocators. In *Software Practice and Experience,* 23(8):851-869, August 1993.

[GZH1993] D. Grunwald, B. Zorn and R. Henderson. Improving the cache locality of memory allocation. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation,* pages 177-186.

[Hin1975] J. A. Hinds. An algorithm for locating adjacent storage blocks in the buddy system. In *Communications of the ACM,* 18(4):221-222, April 1975.

[Hir1973] D. S. Hirschberg. A class of dynamic memory allocation algorithms. In *Communications of the ACM,* 16(10):615-618, October 1973.

[HK1954] F. E. Hamilton and E. C. Kubie. The IBM magnetic drum calculator type 650. In *Journal of the ACM*, 1(1):13-20, January 1954.

[HL1961] W. P. Heising and R. A. Larner. A semi-automatic storage allocation system at loading time. In *Communications of the ACM*, 4(10):446-449, October 1961.

[Hof1985] D. R. Hofstdater. Metamagical themas: questing for the essence of mind and pattern. BasicBooks New York NY, 1985.

[Hol1961] A. W. Holt. Program organization and record keeping for dynamic storage allocation. In *Communications of the ACM*, 4(10):422-431, October 1961.

[Hop1952] G. M. Hopper. The education of a computer. In *Proceedings of the 1952 ACM national meeting (Pittsburgh)*, pages 243-249, 1952.

[Hos1991] A. L. Hosking. Main memory management for persistence. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991.

[Hue2001] R. J. Huetter. United States Patent No. 6,324,550. *Data object identification and removal system.* November 2001.

[IGK1971] S. Isoda, E. Goto and I. Kimura. An efficient bit table technique for dynamic storage allocation of 2^n word blocks. In *Communications of the ACM,* 14(9):589-592, September 1971.

[IJ1962] J. K. Iliffe and J. G. Jodeit. A dynamic storage allocation scheme. In *Computer Journal,* 5(3):200-209, October 1962.

[Ing1961] P. Z. Ingerman. Dynamic declarations. In *Communications of the ACM*, 4(1):59-60, January 1961.

[Iye1993] A. K. Iyengar. Parallel dynamic storage allocation algorithms. In *Fifth IEEE Symposium on Parallel and Distributed Processing,* 1993.

[Iye1996] A. Iyengar. Scalability of dynamic storage allocation algorithms. In *Proceedings of the 6th Symposium on Frontiers of Massively Parallel Computing*, 1996.

[JMN1961] J. Jensen, P. Mondrup and P. Naur. A storage allocation scheme for ALGOL 60. In *Communications of the ACM*, 4(10):441-445, October 1961.

[Joh1971] J. B. Johnston. The contour model of block-structured languages. In *ACM Symposium on Data Structures in Programming Languages, SIGPLAN Notices*, February 1971.

[Joh1997] M. S. Johnstone. Non-Compacting memory allocation and real-time garbage collection. PhD thesis, University of Texas at Austin, December 1997.

[JW1998] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: solved? In *Proceedings of the first international symposium on memory management,* pages 26-36, 1998.

[Kau1984] A. Kaufman. Tailored-list and recombination-delaying buddy systems. In *ACM Transactions on Programming Languages and Systems*, 6(1):118-125, January 1984.

[Kel1961] J. E. Kelley Jr. Techniques for storage allocation algorithms. In *Communications of the ACM*, 4(10):449-454, October 1961.

[Kel1995] J. A. Kelso. Dynamic patterns: the self-organization of brain and behavior. MIT Press, Cambridge, Massachusetts, 1995.

[KK1983] LOOM – Large object-oriented memory for Smalltalk-80 systems. In G. Krasner, editor, Smalltalk-80: bits of history, words of advice. Chapter 14, pages 251-270. Addison-Wesley, 1983.

[Kno1965] K. C. Knowlton. A fast storage allocator. In *Communications of the ACM*, 8(1):623-625, October 1965.

[Knu1960] D. E. Knuth. Minimizing drum latency time. In *Journal of the ACM*, 8(2):119-150, April 1961.

[Knu1973] D. E. Knuth. The art of computer programming. Vol 1: Fundamental Algorithms. (2nd edition) Addison-Wesley, Reading Massachusetts, 1973. Section 2.5.

[Law1967] H. W. Lawson Jr. PL/I list processing. In *Communications of the ACM*, 10(6):358-367, June 1967.

[Lea2000] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html

[LH1982] B. W. Leverett and P. G. Hibbard. An adaptive system for dynamic storage allocation. In *Software Practice and Experience,* 12(6):543-556, June 1982.

[LK1998] P. Larson and M. Krishnan. Memory allocation for long-running server applications. In *Proceedings of the first international symposium on memory management*, pages 176-185, 1998.

[Lov2003] R. Love. Linux kernel development. Sams Publishing, Indianapolis, Indiana, 2003.

[Mah1961] R. J. Maher. Problems of storage allocation in a multiprocessor multiprogrammed system. In *Communications of the ACM,* 4(10):421-422, October 1961.

[McC1960] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. In *Communications of the ACM*, 3(4):184-195, April 1960.

[McC1999] J. McCarthy. Lisp – notes on its past and future – 1980. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, 1980.

[Mic2002] M. M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73-82, August 2002.

[Mic2004] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation,* pages 35-46, June 2004.

[MPS1971] B. H. Margolin, R. P. Parmelee and M. Schatzoff. Analysis of free-storage algorithms. In *IBM Systems Journal,* 10(4):283-304, 1971.

[Nau1960] P. Naur (editor). Revised report on the algorithmic language ALGOL 60. In *Communications of the ACM*, 3(5):299-314, May 1960.

[Nie1977] N. R. Nielson. Dynamic memory allocation in computer simulation. In *Communications of the ACM,* 20(11):864-873, November 1977.

[NT1960] A. Newell and F. M. Tonge. An introduction to information processing language V. In *Communications of the ACM*, 3(4):205-211, April 1960.

[OA1985] R. R. Oldehoeft and S. J. Allan. Adaptive exact fit storage management. In *Communications of the ACM,* 28(5):506-511, May 1985.

[One1961] R. W. O'Neill. A preplanned approach to a storage allocating compiler. In *Communications of the ACM*, 4(10):417, October 1961.

[Pag1982] I. P. Page. Optimal fit of arbitrary sized segments. In *Computer Journal* , 25(1):32-36, January 1982.

[PH1986] I. P. Page and J. Hagins. Improving the performance of buddy systems. In *IEEE Transactions on Computers,* 35(5):441-447, May 1986.

[PJ2002] T. Printezis and R. Jones. GCspy: An adaptable heap visualization framework. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 343-358, 2002.

[PN1977] J. L. Peterson and T. A. Norman. Buddy systems. In *Communications of the ACM,* 20(6):421-431, June 1977.

[PS1970] P. W. Purdom and S. M. Stigler. Statistical properties of the buddy system. In *Journal of the ACM,* 17(4):683-697, October 1970.

[PT1960] A. J. Perlis and C. Thornton. Symbol manipulation by threaded lists. In *Communications of the ACM,* 3(4):195-204, April 1960.

[Ran1969] B. Randell. A note on storage fragmentation and program segmentation. In *Communications of the ACM*, 12(7):365-372, July 1969.

[Ree1979] C. M. Reeves. Free store distribution under random fit allocation. In *Computer Journal,* 22(4):346-351, November 1979.

[Ree1980] C. M. Reeves. Free store distribution under random fit allocation: Part 2. In *Computer Journal,* 23(4):298-306, November 1980.

[Ree1982] C. M. Reeves. A lumped-state model of clustering in dynamic storage allocation. In *Computer Journal,* 27(2):135-142, 1982.

[Ree1983] C. M. Reeves. Free store distribution under random fit allocation: Part 3. In *Computer Journal,* 26(1):25-35, February 1983.

[Ris1961] B. N. Riskin. Core allocation based on probability. In *Communications of the ACM*, 4(10):454-459, October 1961.

[RK1968] B. Randell and C. J. Kuehner. Dynamic storage allocation systems. In *Communications of the ACM,* 11(5):297-306, May 1968.

[Rob1961] A. E. Roberts Jr. A general formulation of storage allocation. In *Communications of the ACM,* 4(10):419-420, October 1961.

[Rob1971] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. In *Journal of the ACM*, 18(3):416-423, July 1971.

[Rob1974] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. In *Journal of the ACM*, 21(3):491-499, July 1974.

[Rob1977] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. In *Computer Journal,* 20(3):242-244, August 1977.

[Ros1961] D. T. Ross. A generalized technique for symbol manipulation and numerical calculation. In *Communications of the ACM,* 4(3):147-150, March 1961.

[Ros1967] D. T. Ross. The AED free storage package. In *Communications of the ACM,* 10(8):481-492, August 1967.

[Ros1969] S. Rosen. Electronic computers: a historical survey. In *ACM Computing Surveys*, 1(1):7-36, March 1969.

[Ros1986] D. T. Ross. A personal view of the personal work station. Some firsts in the fifties. *In Proceedings of the ACM Conference on The history of personal workstations*, pages 19-48, 1986.

[RR1996] C. Runciman and N. Röjemo. Two-pass heap profiling: a matter of life and death. In *Selected papers from the Eight Internal Workshop on the Implementation of Functional Language,* pages 222-232, 1996.

[Sal1962] G. Salton. Manipulation of trees in information retrieval. In *Communications of the ACM*, 5(2):103-114, February 1962.

[Sam1961a] B. H. Sams. The case for dynamic storage allocation. In *Communications of the ACM*, 4(10):417-418, October 1961.

[Sam1961b] B. H. Sams. Dynamic storage allocation for an information retrieval system. In *Communications of the ACM*, 4(10), 431-435, October 1961.

[Sat1961] K. Sattley. Allocation of storage for arrays in ALGOL 60. In *Communications of the ACM*, 4(1):60-65, January 1961.

[SB2000] M. Serrano and H. Boehm. Understanding memory allocation of Scheme programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 245-256, 2000.

[Sei2000] M. L. Seidl. Increasing the reference locality of dynamically allocated objects. PhD thesis, University of Colorado at Boulder, 2000.

[SGI2003] Silicon Graphics, Inc. SGI – Products: Servers and Supercomputers: SGI Altix 3000 Family. http://www.sgi.com/servers/altix