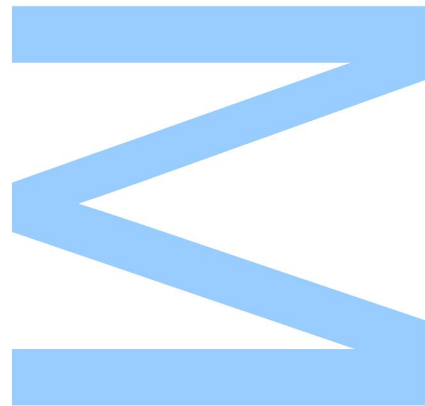
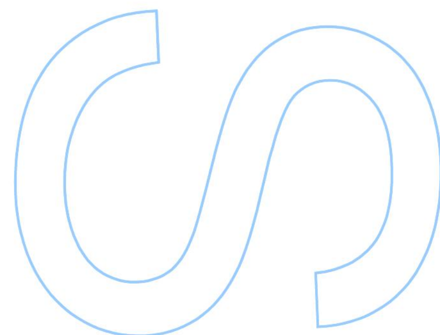
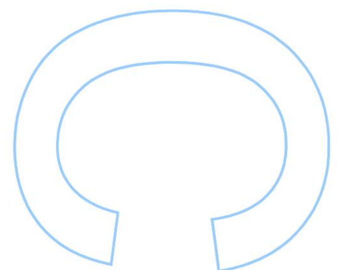
Practical Lock-Free Dynamic Memory Allocation

Ricardo Luís Pinheiro Leite
Master's degree in Computer Science
Computer Science Department
2018

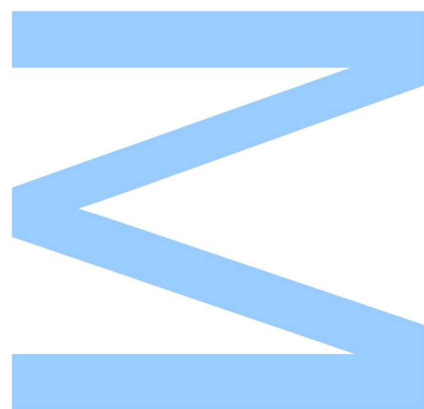
Supervisor
Ricardo Jorge Gomes Lopes da Rocha, Associate Professor,
Faculty of Sciences, University of Porto

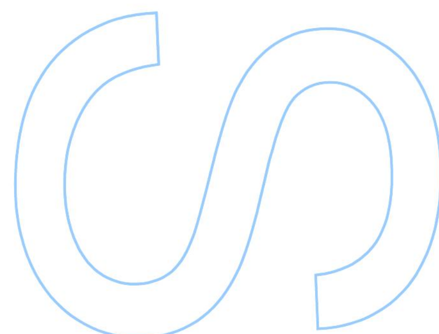




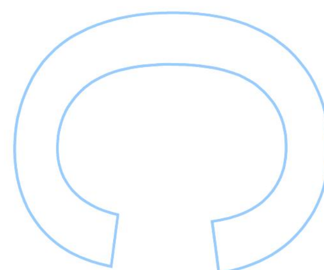
Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/_________







Abstract

Dynamic memory allocation is an important component of most applications. Nowadays, due to the use of thread-level parallelism by applications, an important requirement in a memory allocator implementation is the ability to be thread safe. To guarantee thread safety, current state-of-the-art memory allocators use *lock-based data structures*. The use of locks has implications in terms of program performance, availability, robustness and flexibility. As such, current memory allocators attempt to minimize the effects of locking by using fine-grained locks, by reducing the frequency of lock operations and by providing synchronization-free allocations as much as possible through the use of thread-specific caches. An interesting but challenging alternative is to use lock-free data structures and atomic operations to guarantee thread safety.

We present LRMalloc, a lock-free memory allocator that leverages lessons of modern memory allocators and combines them with a lock-free scheme. Current state-of-the-art memory allocators possess good performance but lack desirable lock-free properties, such as, guarantees of system-wide progress, priority inversion tolerance, kill-tolerance availability, and/or deadlock and livelock immunity. LRMalloc's purpose is to show the feasibility of lock-free memory management algorithms, without sacrificing competitiveness in comparison to commonly used state-of-the-art memory allocators, especially for concurrent multithreaded applications.

We also present a lock-free best-fit mechanism, capable of lock-free coalescing and splitting of blocks. To the best of our knowledge, our mechanism is the first lock-free general coalescing mechanism that supports coalescing and splitting of blocks with arbitrary sizes and does not put restrictions on how coalescing can be performed. Our mechanism ultimately aims to be used as a lower-level allocator for LRMalloc or other lock-free memory allocators, allowing them to manage memory without the assistance of the operating system, potentially improving performance by reducing the number of system calls.

Keywords: Memory Allocator, Memory Management, Lock-Freedom, Concurrent Data Structures, Implementation

Resumo

A alocação dinâmica de memória é uma componente chave para a maioria das aplicações. Atualmente, a utilização generalizada de paralelismo baseado em *threads* pelas aplicações leva a que um requisito importante do alocador de memória seja a sua capacidade de ser *thread safe*. Para garantir que são *thread safe*, os alocadores de memória representativos do estado da arte utilizam estruturas de dados baseadas em *locks*. O uso de *locks* tem implicações em termos do desempenho, disponibilidade, robustez e flexibilidade da aplicação. Como tal, os alocadores de memória atuais tentam minimizar esses efeitos através do uso cuidado e de grão fino de *locks*, ao reduzirem a frequência de operações de *locking* tanto quanto possível, e fornecendo alocações que não precisam de quaisquer operações de sincronização através do uso de *thread caches*. Uma alternativa interessante mas desafiadora é usar estruturas de dados *lock-free* juntamente com operações atómicas para garantir a capacidade de ser *thread safe*.

Nesta tese, apresentamos o LRMalloc, um alocador de memória *lock-free* que reutiliza conhecimento e aprendizagens conseguidos pelos alocadores de memória modernos e combina-os com um esquema de gestão de memória *lock-free*. Apesar dos alocadores de memória atuais apresentarem bom desempenho, não possuem propriedades desejáveis tais como: garantias de progresso no sistema, tolerância a inversão de prioridade, tolerância a *threads* que terminam prematuramente e/ou imunidade a *deadlocks* e *livelocks*. O propósito do LRMalloc é mostrar a viabilidade de algoritmos *lock-free* de gestão de memória, sem sacrificar a competitividade quando comparado com alocadores de memória que são o estado da arte, especialmente para aplicações *multithreaded*.

Também apresentamos um mecanismo *best-fit* que é capaz de gerir memoria de forma *lock-free*. No melhor do nosso conhecimento, o nosso mecanismo é o primeiro que é *lock-free* e capaz de *coalescing* e *splitting* de blocos de memória com qualquer tamanho, sem restrições na forma como o *coalescing* é feito. Pretendemos com que o nosso mecanismo seja usado como um alocador de baixo nível para alocadores de memória *lock-free*, de forma a que o alocador seja capaz de gerir memoria sem a assistência do sistema operativo, e de forma potencialmente mais eficiente, ao permitir a redução do número de chamadas de sistema.

Palavras-chave: Alocador de Memória, Gestão de Memória, *Lock-Freedom*, Estruturas de Dados Concorrentes, Implementação

Acknowledgements

I would like to begin by thanking my adviser, Prof. Ricardo Rocha, who has been always helpful and supportive during my research and elaboration of this work. His patience and easy-going attitude have not gone unnoticed.

I'd like to thank all the nameless colleagues and friends, some of whom I've had interesting talks with, others who I've had good fun with and that I'll remember and cherish.

At last, I would also like to thank INESC TEC for funding a good part of this research work. This work is financed by the ERDF (European Regional Development Fund) through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT (Portuguese Foundation for Science and Technology) within project POCI-01-0145-FEDER-016844.

Contents

Abstract **i**

Resumo **iii**

Acknowledgements **v**

Contents **x**

List of Tables **xi**

List of Figures **xiii**

Listings **xv**

1 Introduction **1**

1.1 Thesis Purpose and Contributions 2

1.2 Thesis Organization 3

2 Classical Memory Allocation **5**

2.1 Role of a Memory Allocator 5

2.2 Fragmentation 6

2.3 Memory Allocation Mechanisms 8

2.3.1 Sequential Fits 9

2.3.2 Segregated Free Lists 10

2.3.3 Buddy Systems 12

2.4 Program Regularities 14
2.5 Effects on Program Execution 15
2.5.1 Memory Usage Impact 16
2.5.2 Performance Impact 17
2.6 Memory Allocation in Practice 18
2.6.1 Malloc C API 18
2.6.2 Role of the Operating System 19
2.6.3 Implications of Virtual Memory 20
2.7 Chapter Summary 21

3 Principles of Multithreading **23**
3.1 Memory Access Impact 23
3.1.1 Cache Line Contention 24
3.1.2 False Sharing 25
3.2 Synchronization Primitives 26
3.2.1 Atomic Operations 27
3.2.2 Locks 29
3.3 Progress Guarantees 29
3.4 Chapter Summary 32

4 Multithreaded Memory Allocation **33**
4.1 Mitigating Synchronization Costs 33
4.1.1 Arenas 34
4.1.2 Thread Caches 35
4.2 Performance and Fragmentation Concerns 37
4.2.1 Allocator Induced False Sharing 37
4.2.2 Blowup 38
4.3 Concurrent Memory Allocators 38
4.3.1 Hoard 39

4.3.2 Ptmalloc2 39
4.3.3 Jemalloc 40
4.3.4 TCMalloc 40
4.3.5 Michael's Lock-Free Allocator 41
4.3.6 NBMalloc 42
4.4 Chapter Summary 42

5 LRMalloc Design and Implementation **43**
5.1 High-Level Overview 43
5.2 Size Classes 44
5.3 Thread Caches 46
5.4 Heap 48
5.4.1 Descriptors 48
5.4.2 Superblocks 49
5.4.3 Filling and Flushing Caches 50
5.5 Pagemap 54
5.6 Interaction with the Operating System 54
5.7 Chapter Summary 55

6 Experimental Analysis **57**
6.1 Methodology 57
6.2 Performance & Scalability 60
6.3 Active & Passive False Sharing 62
6.4 Chapter Summary 64

7 A Lock-Free Coalescing Mechanism **65**
7.1 Motivation 65
7.2 Related Work 66
7.3 Coalescing with Boundary Tags 66

7.4 Proposal 69
7.4.1 High-level Allocation and Deallocation 71
7.4.2 Coalescing 72
7.4.3 Implementation 73
7.4.4 Caveats 74
7.5 Chapter Summary 74

8 Conclusion **75**
8.1 Main Contributions 75
8.2 Future Work 76
8.3 Final Remarks 76

Bibliography **77**

List of Tables

2.1 Number of different block size requests in a percentage of all allocations 15

5.1 LRMalloc size classes 45

List of Figures

2.1 An example of memory fragmentation 7

2.2 Overview of a simple segregated storage 11

2.3 An example of a binary buddy system 14

3.1 Overview of degradation when writing simultaneously to memory 24

3.2 False sharing due to the placement of different objects in the same cache line . . 25

4.1 Multiple threads using several arenas 34

4.2 Logical overview of an allocator that uses thread caches 35

5.1 Overview of LRMalloc 44

5.2 LRMalloc thread cache 46

5.3 Example of a partial superblock 49

5.4 Superblock state machine 50

6.1 Execution time results with *Linux scalability* and *Threadtest* benchmarks 61

6.2 Throughput results with *Larson* benchmark 62

6.3 Execution time results with *cache-thrash* and *cache-scratch* benchmarks 63

7.1 A block with the information necessary to perform coalescing 66

7.2 Locating previous and next blocks for coalescing 67

7.3 Overview of backward coalescing 68

7.4 Simultaneous coalescing of two blocks with a shared neighbor 69

Listings

2.1 Malloc interface 19
2.2 Legacy malloc interface 19
3.1 Common atomic operations 27
3.2 Lock-free emulation of fetch-and-add 31
3.3 Example of a lock-free stack vunerable to the ABA problem 31
5.1 High-level allocation and deallocation routines 47
5.2 Anchor and descriptor structures 49
5.3 CacheFill routine 51
5.4 CacheFlush routine 53
7.1 Map operations and definitions 70
7.2 Tree component operations 71
7.3 High-level allocation and deallocation 72
7.4 Backward and forward coalescing 73

Chapter 1

Introduction

Dynamic memory allocation is an important component of applications written in languages that use manual memory management, such as C, C++ and Rust. Manual memory management requires the programmer to explicitly allocate and then deallocate blocks of memory. The task of a dynamic memory allocator is to provide blocks of memory for allocation requests and to internally track and manage memory, as blocks are being continuously allocated and deallocated. Nowadays, most applications are multithreaded, which requires the memory allocator to be able to correctly handle concurrent allocation and deallocation requests. In other words, the memory allocator must be *thread safe*. The requirement of thread safety naturally complicates the memory allocator's design. To guarantee thread safety, current state-of-the-art memory allocators use *lock-based data structures*. The use of locks has implications in terms of program performance, availability, robustness and flexibility. As such, current state-of-the-art memory allocators attempt to minimize the effects of locking by using fine-grained locks, by reducing the frequency of lock operations and by providing synchronization-free allocations as much as possible through the use of *thread-specific caches* [13, 16].

However, even with only infrequent and fine-grained locking, lock-based memory allocators are still subject to a number of disadvantages. They are vulnerable to deadlocks and livelocks, prune to priority inversion and delays due to preemption during locking, and incapable of dealing with unexpected thread termination. A more challenging but desirable alternative to ensure thread safety is to use *lock-free synchronization*. Lock-freedom guarantees system-wide progress whenever a thread executes some finite amount of steps, whether by the thread itself or by some other thread in the process [22]. This progress guarantee is by far the most important advantage of lock-free synchronization and it protects the memory allocator (and therefore the running application) from reaching a complete halt even in the presence of a subpar operating system scheduler, or when a very high number of application threads exist when comparing to machine cores. By definition, lock-free synchronization uses no locks and does not obtain mutual exclusive access to a resource at any point. It is therefore immune to deadlocks and livelocks. Without locks, priority inversion and delays due to preemption during locking cannot occur, and unexpected thread termination is also not problematic.

Additionally, lock-free algorithms or data structures that need to dynamically allocate memory cannot be built on top of a lock-based memory allocator without compromising the lock-freedom property. This is particularly relevant for lock-free data structures or algorithms that use memory reclamation techniques such as *epochs* or *hazard pointers* [20, 39]. As such, competitive lock-free dynamic memory allocators are essential for the complete implementation of lock-free data structures and algorithms that reclaim memory.

For lock-free dynamic memory allocators to be viable and become widely used, they have to match the performance of lock-based dynamic memory allocators. Today's lock-based dynamic memory allocators are fast, with allocation and deallocation operations in the order of dozens of nanoseconds on average, and scalable, allowing for multithreading applications to benefit from more cores even if they are bound by memory allocator performance. The importance of efficient memory allocation has been highlighted by recent work [25] by showing that the dynamic memory allocation consumes nearly 7% of all cycles in warehouse-scale and cloud computing. The competitiveness of new dynamic memory allocators is thus imperative for their viability and wide-spread use.

1.1 Thesis Purpose and Contributions

In this thesis, we present the design and implementation of LRMalloc, a lock-free dynamic memory allocator that leverages lessons of modern allocators to show the feasibility of lock-free memory management, without sacrificing performance in comparison to commonly used state-of-the-art allocators. We can identify three main components in LRMalloc's design: (i) the *thread caches*, which provide synchronization-free blocks and are used in the common allocation/deallocation case; (ii) the *heap*, a lock-free component that manages superblocks of memory pages; and (iii) the *pagemap*, a book-keeping component used to store metadata at the page-level.

Experimental results show that, unlike prior lock-free alternatives, LRMalloc is competitive with commonly used state-of-the-art memory allocators. This has been achieved by reproducing similar strategies to the ones used by lock-based memory allocators to avoid performance degradation due to usage of locks. LRMalloc uses thread caches to reduce the common allocation/deallocation case to a synchronization-free operation, thus avoiding synchronization between threads for most allocation and deallocation requests. When synchronization is necessary, to add/remove blocks to/from a thread cache, LRMalloc attempts to transfer as many blocks with as little synchronization (i.e., atomic operations) as possible, in an effort to amortize the cost of those atomic operations through the subsequent allocation and deallocation requests.

We also present a lock-free best-fit mechanism, capable of lock-free coalescing and splitting of blocks. To the best of our knowledge, our mechanism is the first lock-free general coalescing mechanism that is able to coalesce and split blocks of arbitrary size, and that does not put restrictions in how coalescing can be performed. Our mechanism ultimately aims to be used as a lower-level allocator for LRMalloc or other lock-free memory allocators, allowing them to manage

memory without the assistance of the operating system, and potentially improving performance by reducing the number of system calls.

The results and contributions of this work were already published in a national [33] and in an international [34] event. We have also received an invitation to publish in a special issue of the journal *Concurrency & Computation: Practice & Experience*.

1.2 Thesis Organization

This thesis is structured as follows:

- **Chapter 1 – Introduction**. This chapter.
- **Chapter 2 – Classic Memory Allocation**. Introduces the subject of memory allocation along with classical memory allocation research.
- **Chapter 3 – Principles of Multithreading**. Discusses multithreading and builds a strong foundation for practical considerations needed for modern concurrent memory allocators.
- **Chapter 4 – Multithreaded Memory Allocation**. Discusses challenges of multi-threaded memory allocation and prior work.
- **Chapter 5 – LRMalloc Design and Implementation**. Describes the key data structures and algorithms that support the implementation of LRMalloc.
- **Chapter 6 – Experimental Analysis**. Discusses experimental results that compare LRMalloc against other modern and relevant memory allocators.
- **Chapter 7 – A Lock-Free Coalescing Mechanism**. Describes the design of a novel lock-free best-fit mechanism capable of coalescing that can be used for managing memory.
- **Chapter 8 – Conclusion**. Outlines conclusions and discusses further work.

Chapter 2

Classical Memory Allocation

In this chapter, we introduce the subject of memory allocation. We start by discussing the role of a memory allocator. We then present a summary of memory allocation research starting in the 60s. We describe this research as *classical*, as it defines long-established mechanisms for memory allocators that are still in use today.

Throughout this chapter and in the remaining of the document, we use the terms allocator, memory allocator and dynamic memory allocator interchangeably.

2.1 Role of a Memory Allocator

The duty of a dynamic memory allocator is to keep state of the memory that has been provided for an application – to track what parts of memory are in use and what parts are free, in order to be able to fulfill memory requests for an underlying application (or program, or user). Only two kind of requests can be made: (i) to allocate a block of memory of some specified size; (ii) to free a previously allocated block. A dynamic memory allocator has no control over what the sequence, number, or properties of requests that are made by the application. However, it has control over the data structures, techniques and heuristics used to deal with the stream of memory requests.

In general, a dynamic memory allocator gives the following guarantees to the application:

1. A response to a request is irrevocable – a block supplied to the application cannot be later changed or reverted;
2. Blocks in use by the application cannot be moved or touched by the allocator;
3. Only freed blocks and unused memory can be managed and touched by the allocator.

On the other hand, the application, program or user has to guarantee that:

1. An allocated block is only freed once;

2. Once a block is freed, it can not be touched by the application.

The application's guarantees are easy to understand since, if an application is able to free an allocated block several times or touch a block after free, the memory allocator is unable to determine when a memory block can be reused and supplied back to the application. In fact, violating either of these guarantees may lead to well-known vulnerabilities – violating the first guarantee may lead to a *double-free* exploit [11], while violating the second guarantee may lead to a *use-after-free* exploit [1].

Please note that, in this document, we are discussing manual memory management and not garbage collection. Manual memory management and garbage collection are two distinct research areas. Garbage collectors have a very different set of guarantees provided to the application. Garbage collectors can, for example, move blocks that are currently in use by the application while still guaranteeing correct application behavior. In a garbage collected environment, the application does not have to explicitly deallocate blocks, and the garbage collector is left to resolve the moment from which the application can no longer access the block, in order to safely deallocate it. Compared to manual memory management, garbage collection has different design and performance characteristics [23].

Furthermore, we argue that garbage collectors are more tightly integrated with the specific language and environment they execute on – a garbage collector's design depends on the language constructs and abstract machine model in order to resolve the moment at which blocks of memory can no longer be accessed. This is bound to be different depending on the type of the programming language used (e.g., functional vs imperative). The observation that the garbage collector is an essential and irreplaceable component of the runtime system in languages such as Haskell or Java, while a dynamic memory allocator is a replaceable library in C/C++/Rust seems to support our view.

2.2 Fragmentation

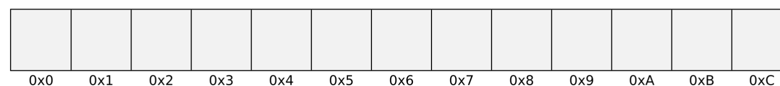
Memory allocation as a research topic has a rich history. Starting in the 1960s, and motivated by the exorbitant cost of main memory, researchers poured effort into the design of allocator strategies and algorithms that minimized *memory fragmentation* – a fundamental problem in memory management that implies needing more memory to satisfy a sequence of memory requests than the sum of the memory required for those memory requests individually. This type of research (which we describe as *classical*) assumes a sequential execution environment and concerns itself mainly with fragmentation rather than performance. A large number of memory allocation block placement algorithms (which from now on we will refer to as *mechanisms*) have been developed that exhibit different fragmentation and performance behavior [51].

Memory fragmentation is thus a fundamental problem that must be handled by any dynamic memory allocator. When an application starts, the dynamic memory allocator has a large

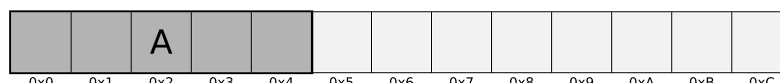
continuous region of memory. As the application requests memory allocations and deallocations, the memory allocator uses this region to fulfill the requests. Over time, this region becomes fragmented and made up of smaller continuous blocks of used and freed (available) memory. As the fragmentation increases, it becomes impossible for the memory allocator to fulfill a memory allocation request that is larger than any of the available continuous blocks, even if the total of the sum of these available continuous blocks is larger than the requested block.

Figure 2.1 demonstrates an example of a simple sequence of requests leading to a fragmented region. In this example, the allocation of blocks A and B with sizes 5 and 3, respectively, and the subsequent deallocation of A results into a fragmented region that is unsuitable to allocate a block C with size 6 – even though we can account a total of 10 free blocks available.

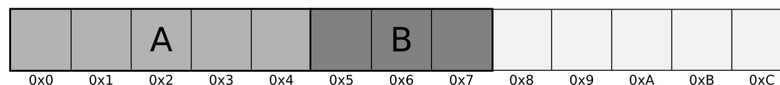
1. Initial free region



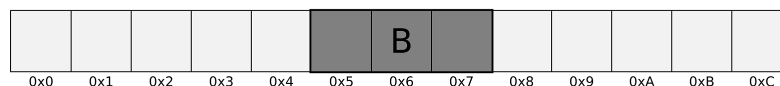
2. Allocation of A, size 5



3. Allocation of B, size 3



4. Deallocation of A



5. Allocation of C, size 6 **fails**

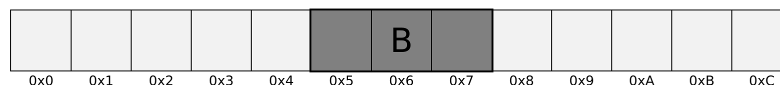


Figure 2.1: An example of memory fragmentation

Memory allocation can be seen as a combinatorial problem. In computational complexity terms, an offline version of memory allocation can be modeled as a modified bin packing problem, which is known to be NP-hard [12]. Therefore, even an offline algorithm, that has access to the entire request stream, cannot guarantee minimal or no fragmentation for a non-trivial stream of requests. A dynamic memory allocator is in essence an online algorithm for the same problem. An online algorithm has no information about future requests in the request stream and has to irrevocably fulfill the request. That lack of information inevitably increases the difficulty of minimizing fragmentation.

Fragmentation is ultimately a result of the block placement decisions of the memory allocator. In Fig. 2.1, the memory allocator chooses to place a block in the first position where it can be

placed. A different set of decisions could have resulted in a different fragmentation behavior.

Note however that the stream of requests that a memory allocator has to deal with is not random by nature – it is a consequence of program behavior. Computer programs exhibit regularities which can be exploited. Modern CPU architectures exploit temporal and spatial locality through the use of caches, and specific cache designs further exploit program regularities. Classical memory allocation research has given origin to a number of memory allocation mechanisms with different performance characteristic that exploit these regularities with different effectiveness.

2.3 Memory Allocation Mechanisms

Wilson *et al.* [51] present a survey that summarizes most memory allocation research prior to 1995. In it, they discuss the importance of the task of *block placement*. Wilson *et al.* argue that a dynamic memory allocator's functioning can be thought of three parts: *strategy*, *policy* and *mechanism*. The strategy attempts to exploit regularities in the request stream – the chosen strategy allows a range of acceptable policies that can be used. A policy is an *implementable decision procedure for placing blocks in memory* or, in other words, an action to choose a block. Finally, a mechanism is an algorithm that implements the desired policy. Wilson *et al.* [51] describe and categorize the following memory allocator mechanisms:

- *Sequential fits*, such as, first-fit, next-fit and best-fit.
- *Segregated free lists*, such as, simple segregated storage, segregated fits and slabs.
- *Buddy systems*, most importantly binary buddy systems.

Two techniques for supporting placement policies are *splitting* and *coalescing* of free blocks. A dynamic memory allocator may have to split a large block into two smaller ones in order to fulfill a memory request. Similarly, a dynamic memory allocator may coalesce (or merge) neighboring free blocks together in order to fulfill a request for a larger block. Ignoring performance concerns, coalescing frequently (e.g., after every deallocation) is desirable – a larger block is more useful than many smaller blocks when fulfilling block requests. One may even view fragmentation as an overabundance of small blocks that cannot be coalesced. As will be seen later, memory allocator mechanisms restrict the way in which blocks can be split and coalesced.

Some mechanisms may fulfill allocation requests by providing blocks that are larger than the requested block size. Fragmentation that is due to the wasted space inside a block that is too large is called *internal fragmentation* [45]. All remaining fragmentation is called *external fragmentation*. This distinction is important, as will be seen later in this chapter, some mechanisms potentiate one type of fragmentation but not the other.

2.3.1 Sequential Fits

Some of the most basic mechanisms used in early memory allocators are called *sequential fits*, where the memory allocator uses a single list of free blocks (called a *free list*) to determine what block is going to be provided for the next memory allocation request. Note that this list might be replaced by a more appropriate data structure that allows the allocator to have better performance characteristics, while behaving fundamentally the same in terms of fragmentation. Sequential fits include *first-fit*, *next-fit*, *best-fit* and *worst-fit*.

First-fit

First-fit searches the free list and provides the first found block that meets the size requirements. If the block found is not an exact match, it is split, and the smaller remaining block is re-inserted back into the list. Blocks that have been deallocated or are obtained as a side effect from splitting may be put: (i) in the front of the free list, commonly called a LIFO (last-in-first-out) first-fit; (ii) in the back of the free list, called a FIFO (first-in-first-out) first-fit or (iii) in an address-ordered position in the free list, which is referred to as an address-ordered first-fit. Address-ordered first-fit encodes the adjacency of free blocks, which can be used to implement faster coalescing. Worst-case fragmentation for first-fit is well-known [46] and FIFO and address-ordered first-fit are known to have good fragmentation performance [52].

Next-fit

Next-fit is in essence an attempt to optimize first-fit where the search in the free list begins at the point at which the search stopped in the last request, in an attempt to decrease average search times when the free list is implemented as a linear list. However, next-fit ends up changing block placement significantly in a way that is well distinct from any type of first-fit. Next-fit has been found to cause more fragmentation than first-fit or best-fit [43].

Best-fit

Best-fit uses the free list to do an exhaustive search looking for the smallest block that can fulfill the request. The goal of a best-fit mechanism is to minimize the size of resulting fragments from a split and thus reduce the amount of wasted space. However, this may backfire if best-fit fails to find perfect matches and instead generates fragments that are too small to be used by future allocation requests. Best-fit has been found to exhibit good fragmentation performance [52] but undesirable worst-case fragmentation [46]. Similarly to first-fit, best-fit has FIFO, LIFO and address-ordered variants. These variants define the chosen free block in case there are several equal-sized blocks that match a memory allocation request. Address-ordered best-fit has been found to have particularly good performance [24].

Worst-fit

Worst-fit is conceptually the opposite of best-fit – it uses the free list to search and use the largest free block. The reasoning behind worst-fit is that by always using the largest free block, the remaining block is as large as possible, which should reduce the number of unusable fragments. In practice, worst-fit tends to work quite badly, as it tends to reduce the number of large blocks available [48].

2.3.2 Segregated Free Lists

Segregated free lists use several distinct free lists of blocks (often an array of free lists), usually segregated by size. Allocation requests are serviced by selecting the appropriate free list. To limit the required number of distinct free lists, various segregated free list schemes aggregate various block sizes into a *size class*. Size classes are commonly composed by powers-of-two or by sums of powers of two. If size classes are used, requests are serviced by rounding up the requested block size to the nearest size class and then using the appropriate free list. Size classes are a particularly important concept for modern memory allocators, and the use of size classes has implications in how the allocator has to deal with lower level details such as *alignment* and implications on internal fragmentation, since a block given to the application may include wasted memory, if the provided block is larger than the requested size.

Note that segregated free lists have distinct free lists, but the blocks in these lists are not physically segregated. Coalescing and splitting is still possible for adjacent blocks that are in different free lists, if additional mechanisms are in place. Furthermore, a block of memory (or any specific memory address) can only belong to a single free list. Segregated free lists include *simple segregated storage*, *segregated fits* and *slab allocation*.

Simple segregated storage

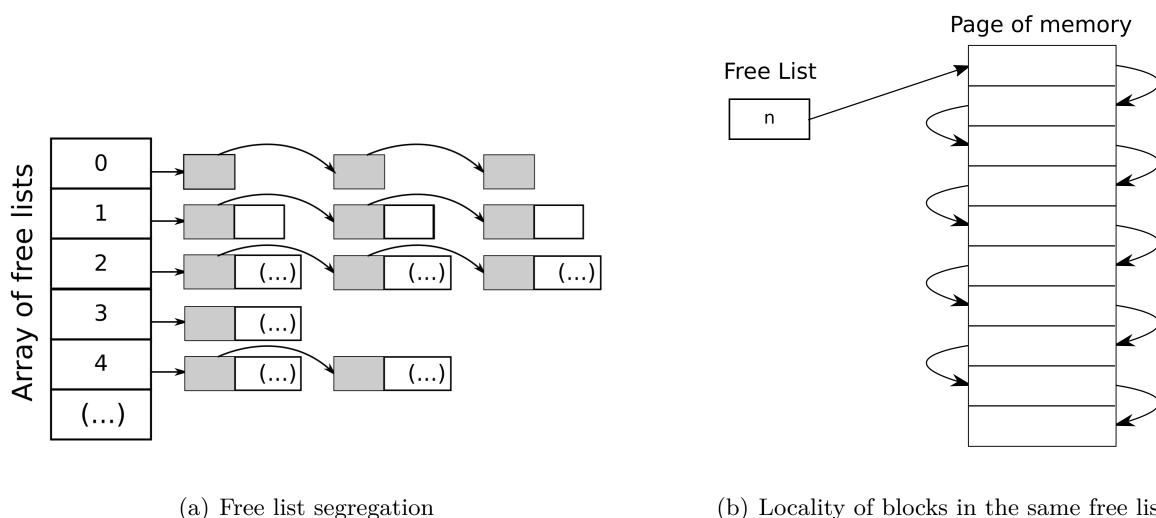
In simple segregated storage, requests can only be serviced by a single free list. On a allocation request, if the appropriate free list is empty, there is no attempt to obtain a larger block from another free list in order to split it. Instead, more memory is obtained from the operating system in order to assemble blocks to be added to the empty free list. On deallocations, there is no coalescing of blocks, the block is just added to the corresponding free list or returned along other free blocks to the operating system (or to a more specific lower-level component).

This lack of splitting and coalescing mechanisms has some severe implications in terms of fragmentation – since allocation requests can only be serviced by a single free list, some care must be taken in the management of the amount of free blocks that are kept in these free lists. The upside of this fragmentation disadvantage is that this scheme requires less memory to do memory bookkeeping – a single number can be used to track equal-sized blocks in a region (e.g., a page), instead of more complicated and costly data structures such as bitmaps or trees that

are needed when blocks can be coalesced and thus no assumption of the size of blocks in a given region can be made. Late classical research [24] suggests that, without some care in the chosen size classes, the increased fragmentation by far exceeds the reduced internal memory overhead.

As will be seen later in Ch. 4, simple segregated storage is by far the most common scheme used in modern dynamic memory allocators. It possesses good locality properties, as blocks with the same size are necessarily grouped together, and this ties up nicely with certain properties that can be explored in regular programs. It offers desirable speed performance, as an allocation becomes just an array lookup to obtain the correct free list (with the necessary size class computation operations), followed by an element pop (e.g., the block) from the list.

Figure 2.2 shows an overview of a simple segregated storage. Figure 2.2(a) exemplifies the free list segregation, where each free list contains blocks of a single size. The free list itself can be composed using the free blocks' memory (just enough to contain a pointer). The memory used to compose the list does not count as actual memory allocator overhead, since it is used to fulfill a memory allocation request. Therefore, the free list itself can be represented with a single pointer that acts as the list's head. Figure 2.2(b) highlights the actual memory layout when filling a particular free list. Pages (or an equivalent region of memory) will be obtained from the operating system and then split into several equal-sized blocks. These equal-sized blocks will be provided sequentially to fulfill memory allocations, with a high degree of spatial locality.



(a) Free list segregation

(b) Locality of blocks in the same free list

Figure 2.2: Overview of a simple segregated storage

Segregated fits

Segregated fits differ from simple segregated storage in that blocks can be split and coalesced, and a search for an appropriate-sized block can visit several free lists. In the allocation case, and similarly to simple segregated storage, there is a lookup for the free list that corresponds to the wanted size (if there is a separate list for every possible size, this is called an *exact list segregated*

fit). If the free list is not empty, a free block is removed. If empty, an iterative search is made on the lists with larger sizes until a block is found. The found block is then split if it is large enough to be split (i.e., such that the remaining block can be added to a list), and the remaining block is added to the corresponding free list.

When a block is deallocated in a segregated fits scheme, it can be coalesced with neighboring free blocks. Those neighboring free blocks are removed from their free lists, and the resulting larger coalesced block is added to the corresponding list. Not all segregated fits schemes choose to coalesce immediately on block deallocation, some may choose to defer it to a later time in an effort to improve the deallocation case performance.

Segregated fits can be seen as a form of best-fit or *good-fit* (a kind of weaker best-fit) with better performance characteristics because of the free list segregation. Similarly to first-fit and best-fit, it is known to achieve good performance in practice [24].

Slabs

Slabs (also commonly known as slab allocation) is a particular type of simple segregated storage that uses *object types* instead of block sizes to determine which free list is used. Slabs require a non-standard memory allocator interface that allows allocation of an object of a specific type, rather than just a size. In slab allocation, there is a free list per object type rather than size or size class. There can be multiple free lists that store blocks of the same size, as long as they are providing blocks for objects of different types.

Slab allocation is most prominently used for operating system object allocation [6, 7], but it can also be used as custom memory allocation for some applications [2, 15]. Slab allocation offers better locality characteristics than simple segregated storage, as blocks representing objects with the same type are more closely put together than blocks representing different objects types, and objects of the same type have more closely tied lifetimes. Variations of slab allocators that are deeply integrated with the program/application (as in the case in [6]) can also offer object construction and destruction optimizations.

Slab allocation is less general and program-agnostic than other memory allocation schemes described so far, but is still generic enough to be called a generic allocation scheme. It cannot be used with the standard memory allocation interface (which only allows allocation of blocks of a specific size, not type), but it can be used in strongly typed languages that have extended memory allocation interfaces that include type information, such as C++.

2.3.3 Buddy Systems

Buddy systems impose a hierarchical structure over an entire region of memory to determine how it can be logically split/coalesced. Initially, the entire heap is a block that can be split into two *buddies*. Each buddy is a block that can be further split into another two buddies, and so

forth. Each block always has a corresponding buddy – the other block that was produced by the splitting of the block that is one level up in the hierarchy. Since each block can easily calculate the corresponding buddy, this offers a very fast and low memory overhead coalescing mechanism. A block can quickly find its buddy through a simple address computation, and only a single bit of information is required to determine whether the buddy is free and available for coalescing or is being used.

The buddy system imposes a structure where blocks can only be split in a deterministic way, regardless of the requested block size. To find free blocks in the hierarchy, buddy systems use a segregated fits mechanism, containing one free list per each possible block size. For example, in a binary buddy system, there is a free list per each power of two. To allocate a block, the segregated fits mechanism is used to quickly find a block. Allocations are then fulfilled by either using the found block, if it has the appropriate size, or by recursively splitting it into two buddy blocks – the first is checked for the appropriate size, the second is added as a free block to the segregated fits mechanism. Deallocations are fulfilled by finding the buddy of a block and then by performing recursive coalescing – a block is recursively coalesced with its buddy until a not free buddy is found. Blocks being coalesced are removed from the corresponding free list in the segregated fits mechanism, and the resulting coalesced block is added.

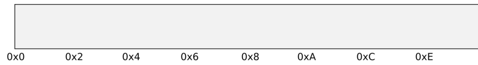
The most common and well-known form of buddy systems are binary buddies, where blocks in the hierarchy use powers of two – a small variation of this type of system is used for managing pages in the Linux Kernel [37]. There are other more exotic forms of buddy systems that use Fibonacci series (called Fibonacci buddies) and forms that use specific size class series (called Weighted buddies). These have been well documented in [51].

In general, buddy systems show good performance characteristics, allowing for fast coalescing with only a minimal amount of memory needed. However, buddy systems have been found to have undesirable fragmentation in practice [24], both internal fragmentation due to the limited number of different block sizes and external fragmentation due to the restricted coalescing that can be done.

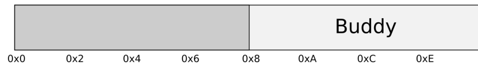
Figure 2.3 shows a allocation and a deallocation example in a binary buddy system that uses a heap of size 16 (2^4). Figure 2.3(a) shows the allocation of a block A of size 3. The initial heap size is too large for this allocation, thus it is split into two blocks (buddies), each one of size 8. The resulting first block is still too large for the allocation, so it is split once more, into two buddies of size 4, which are suitable for an allocation of size 3. The resulting first block of size 4 is then used for the allocation of block A. For a requested allocation of size 3, no better match could have been made, although the remaining space is unusable and accounted as internal fragmentation.

Figure 2.3(b) shows the deallocation of a block A in a heap where two blocks A and B have been allocated. Block A is first deallocated, and immediately after A's buddy is located and an attempt is made for coalescing. Next, the buddy of this coalesced block is also located, but coalescing fails because the buddy is being used to hold block B.

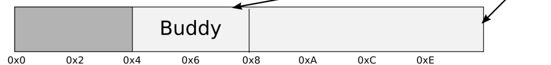
1. Initial heap state (heap size = 2^4)



2. Split into two buddies



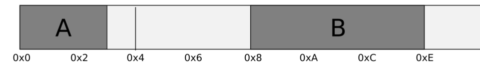
3. Further split



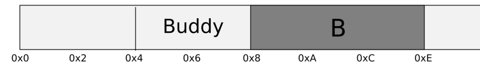
4. Block has adequate size, allocate A

(a) Allocation of block A with size 3 in an empty heap

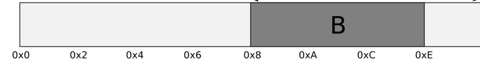
1. Heap state after allocating A and B



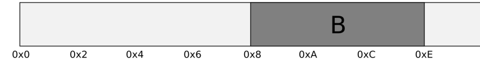
2. Deallocate A, find buddy



3. Coalesce, find buddy



4. Coalescing fails, buddy is in use



(b) Deallocation of block A in a heap with a block B also allocated

Figure 2.3: An example of a binary buddy system

2.4 Program Regularities

Computer programs exhibit patterns of behavior and the regularities they exhibit have been exploited to design effective hardware. Memory caches, prefetching and branch prediction are all examples of hardware design features that assume and exploit program regularity. The same is true for memory allocation – programs exhibit regularities in the stream of requests done to the memory allocator and these regularities can be exploited in order to obtain more effective memory allocators, both in terms of speed and fragmentation.

The fact that there are regularities in the stream of requests is well accepted, but most of earlier classical memory allocation research does not directly address it – mechanisms were not explicitly designed to exploit specific regularities, some just happen to do so. Wilson *et al.* [51] point out that best-fit and address-ordered first-fit mechanisms work well in practice and have good fragmentation, but the reasons for that were not clear at the time. They argue that past evaluation of memory allocators has led to this state of affairs, because the performance of memory allocators and related mechanisms was measured with streams of requests that were randomly generated, often with unrealistic distributions. A random stream of requests has no regularities that can be exploited, and thus memory allocators that perform best in this type of evaluation are not necessarily the same memory allocators that also perform well in practice. In what follows, we present some regularities that are known and can be exploited by memory allocators.

Programs allocate only a few block sizes

Most memory allocation requests that a program does are for the same few types of block size.

Table 2.1, by Johnstone *et al.* [24], shows the number of distinct block size requests in a given percentage of all allocations for a set of 8 different programs. On average, 90% of all allocations are for 6 distinct block sizes, which shows that the majority of the allocations being done is for a limited number of object types, likely the essential types in the data structures used by the program.

Table 2.1: Number of different block size requests in a percentage of all allocations

Program	**90%**	**99%**	**99.9%**	**100%**
GCC	5	12	254	641
Espresso	9	95	308	758
Ghostscript	7	85	344	589
Grobner	12	55	100	139
Hyper	1	2	2	6
LRUsim	1	1	5	21
P2C	4	26	58	92
Perl	10	27	60	96
Average	6	36	141	293

This regularity has some severe implications for mechanisms such as simple segregated storage, which use size classes and therefore round up allocation sizes, originating some internal fragmentation. Furthermore, simple segregated storage obtains some memory from the operating system to fill an empty free list in order to fill it with several blocks – that number of blocks may end up unused and count as external fragmentation for a good amount of time, especially if a given size is scarcely used. Therefore, careful consideration is needed to decide how many blocks must be used to fill an empty free list with memory allocators that use simple segregated storage.

Blocks allocated near in time are also deallocated at similar times

Johnstone *et al.* [24] have made the observation that blocks that are allocated at similar times (or near in time) are also deallocated at similar times, often in the same or in reverse order. This regularity explains in part the low fragmentation of address-ordered first-fit and best-fit, both of which tend to place blocks that are allocated successively in continuous memory. Johnstone *et al.* argue that these address-ordered variants have better fragmentation because new blocks tend to be allocated from one end of memory, which gives time for blocks that were allocated closer to the other end of memory to be deallocated and thus allow coalescing to succeed.

2.5 Effects on Program Execution

A program that uses manual memory management requires a dynamic memory allocator to fulfill the program's stream of requests. Thus, it is important to know what overhead we can associate with the dynamic memory allocator and what impact it has on program execution. As will see, a dynamic memory allocator has impact not only on the program memory usage, but also on the overall program performance.

2.5.1 Memory Usage Impact

The most obvious effect of a dynamic memory allocator, and which is the most significant subject of evaluation in classical memory allocation research, is memory usage, with a focus on fragmentation. In terms of memory usage, a good memory allocator is one that reduces the amount of memory required to execute a program. Despite the focus on fragmentation by earlier memory allocation research, it is not the only source of increased memory usage. Johnstone *et al.* [24] have shown that, in practice, if a good memory allocator mechanism is used, fragmentation is minimal and increased program memory usage can be mainly attributed to allocator implementation overhead, such as headers and alignment. In summary, we can accurately account for increased program memory usage in these components:

- Internal fragmentation
- External fragmentation
- Implementation overhead

Internal fragmentation

Internal fragmentation accounts for wasted space in allocated blocks that are larger than the requested size. This mainly happens when the memory allocator uses size classes, as is the case of mechanisms like simple segregated storage, segregated fits and buddy systems. This type of fragmentation can be decreased by increasing the granularity of the size classes, i.e., using size classes that are closer to each other. The worst-case scenario of this type of fragmentation is obvious when the available size class are known. For example, the worst-case internal fragmentation for a mechanism that is using powers-of-two size classes (as is the case of a binary buddy system) is 50% of the total memory usage.

External fragmentation

In essence, external fragmentation accounts for free blocks of memory that cannot be used for future requests because of being too small or logically inaccessible. Some memory allocator mechanisms such as best-fit and first-fit only have external fragmentation in the form of free blocks that are too small to be used.

Unlike internal fragmentation, worst-case external fragmentation is not known without careful analysis of the memory allocator mechanism in question. Robson [47] shows that the lower bound for worst-case fragmentation in any algorithm is $M \log_2 n$, where M is the amount of memory being currently used and n is the ratio between the smallest and the largest block sizes in the requests.

Implementation overhead

Dynamic memory allocators need to keep some internal state in order to fulfill their function. The amount of memory required for a dynamic memory allocator to work depends on the mechanisms and implementation of the memory allocator. There are data structures that can be used almost for free, such as the lists of free blocks that use the block's free memory to assemble the actual list. However, the pointer to the first free block cannot be avoided and that accounts as implementation overhead. Mechanisms such as simple segregated lists or segregated fits must take this into account, as each size class amounts to an instance of a free list that must exist somewhere in memory.

Coalescing is another source of implementation overhead for memory allocators. To be able to coalesce blocks, memory allocators have to store information about what blocks are neighbors and whether those blocks are free and thus available for coalescing. Buddy systems, which impose a strict hierarchy on the heap address space and severely restrict the way coalescing can be done, can escape this by only storing a single bit of information per block. Other mechanisms, such as boundary tags [28], require more memory-intensive methods like having a header field per block. The header field will occupy at least one processor word (e.g., 8 bytes in a 64 bit machine) and contains enough information to track prior and next blocks.

Another source of implementation overhead is alignment. Modern CPU architectures such as x86-64 or ARMv8 require memory operants to be aligned to processor word size. Therefore, the memory allocator must provide blocks that are aligned at least to processor word size, to ensure correct program behavior. These alignment requirements force smaller allocations such as 1-byte to actually occupy at least 8 bytes, because no new block can be placed in the following bytes.

2.5.2 Performance Impact

Unlike memory usage, it is not obvious what impact, if any, a dynamic memory allocator has in program performance. Ultimately, the memory allocator controls the placement of blocks in the heap and provides memory to the program so that it can be accessed – and memory access times may improve or degrade depending on the proximity of the blocks being accessed.

As discussed in Sec. 2.4, programs exhibit regularities that can be exploited not only by hardware but also by the memory allocator, and such regularities influence fragmentation and thus program memory usage. But, there are others that can influence the memory allocator's impact on program performance. Blocks that were allocated near in time are likely to be accessed later at similar times. Data structures that use nodes can benefit from improved performance if nodes are placed together in memory. A memory allocator can thus heuristically improve performance of such data structures by placing blocks that are of the same size as close to each other as possible. This is one of the reasons why slab allocation [6] is particularly effective – slabs group up blocks not only by size, but also by object type. This effectively improves memory access times.

On multithreaded programs, the dynamic memory allocator's placement decisions can also have severe effects on performance if different blocks are placed in the same cache line and then given to different threads, on what is called *false sharing* [4]. This will be discussed further in Ch. 4.

2.6 Memory Allocation in Practice

To understand the requirements and limitations of a dynamic memory allocator, it is important to understand how exactly it is used in practice. In particular, manual memory management in the operating system is fundamentally different from manual memory management in a program or application. A dynamic memory allocator operating in a program can always fall back to the operating system to obtain more memory, or even to manage memory.

In practice, a dynamic memory allocator operates in user-space and has access to the operating system. Unlike what the name suggests, a dynamic memory allocator does not actually have to manage memory in the real sense of the word, since it does not have to be able to start with the entire available memory and then manage it to fulfill requests – that duty is left to the operating system. In fact, some of the memory allocation mechanisms we have described explicitly assume that there is an operating system (or lower-level allocator) they can resort to, namely simple segregated storage.

2.6.1 Malloc C API

The most widely used memory allocation interface is the malloc programming interface, which is a group of functions in the C standard library. A dynamic memory allocator is thus a library that implements the malloc interface. This interface defines how the program is able to allocate and deallocate memory. The C standard specifies a group of functions that allow interaction with the dynamic memory allocator. Listing 2.1 summarizes the malloc interface. There are also other legacy functions that the memory allocator must support, shown in Listing 2.2.

The semantics and arguments of these functions have a deep impact on how memory allocators are implemented. The fact that **free()** does not take in a size argument implies that the memory allocator must somehow be able to infer the size of the allocation from just the pointer – this requires the memory allocator to store this information somewhere, which necessarily increases implementation overhead. Memory allocators that use techniques such as boundary tags can use the same information for both coalescing and to find the size of a memory allocation. Mechanisms such as simple segregated storage that do not use coalescing have to explicitly store size information, often by assuming all blocks in a page (or equivalent region of memory) are equal-sized and then storing metadata for that page.

```
// allocate a block of memory with at least 'size' bytes
void* malloc(size_t size);

// free a previously allocated block of memory
void free(void* ptr);

// allocate a block of memory with at least 'n' * 'size' bytes
void* calloc(size_t n, size_t size);

// allocate 'size' bytes, try to use a prior allocation
void* realloc(void* ptr, size_t size);

// allocate a block of memory with specified alignment
// alignment must be a power of two
void* aligned_alloc(size_t alignment, size_t size);

// obtain the number of usable bytes of block pointed to by ptr
size_t malloc_usable_size(void* ptr);
```

Listing 2.1: Malloc interface

```
// alignment must be power of two
int posix_memalign(void** memptr, size_t alignment, size_t size);

// aligned to a page
void* valloc(size_t size);

// alignment must be power of two
void* memalign(size_t alignment, size_t alignment);

// same as valloc but rounds size of allocation to next multiple of page size
void* pvalloc(size_t size);
```

Listing 2.2: Legacy malloc interface

As discussed earlier in Sec. 2.5.1, alignment is another important aspect that affects the implementation of memory allocators. Not only there are functions that require the memory allocator to return a memory block that is explicitly aligned, but there are also minimum alignment requirements, as specified by the C standard [9].

2.6.2 Role of the Operating System

The operating system has to manage physical memory in order for it to be used by applications and by the operating system itself. Every operating system does so through manual memory management and so there is at least one memory allocator. However, the fundamental unit of memory that the operating system works with is a *page*, rather than a byte. The Linux Kernel for example uses a buddy system to manage pages [37]. Additionally, the operating system may have other memory allocators to fulfill smaller allocations for kernel data structures, as is the case of the slab allocator [6] in the Linux Kernel, and these use the default page allocator to obtain their own memory to manage.

The operating system provides ways for programs to obtain memory. In Linux, pages of

memory can be obtained by using the **sbrk()** or **mmap()** system calls. The **srbk()** system call changes a single value that controls the end of the program's heap. When more memory is needed, the program heap is extended. When memory that is at the end of the program heap is freed, the program heap is shrunk, and thus memory is freed back to the operating system. Nomenclatures such as *wilderness* or *wilderness block* [29] are used to describe memory that is at the end of the program heap and are mostly used by dynamic memory allocators that use **srbk()** to obtain memory from the operating system, such as dlmalloc [31]. The **mmap()** system call, on the other hand, allows to obtain continuous set of pages that can be flexibly returned to the operating system when they are no longer needed, even partially, with the use of **unmmap()**. Another way of returning memory to the operating system is through the use of **madvice()**, which can inform the operating system that a set of pages is no longer needed and can be released, while keeping the page's address space still valid.

At first sight, it might seem that there is no need to duplicate the role of memory allocation in both the program and the operating system, with the operating system's memory allocators being capable of providing memory needed by programs that use manual memory management. And while in theory, the operating system could completely handle dynamic memory allocator, not doing so has a number of benefits. First, interaction with the operating system is done through system calls, which are slow. A user-space dynamic memory allocator can fulfill memory requests much more quickly than a system call can come to completion. Second, a system-wide memory allocator would have to interweave allocations from different programs in memory, and as we have seen in Sec. 2.5, the dynamic memory allocator has an effect on program performance, and this interweaving would be detrimental to block locality.

2.6.3 Implications of Virtual Memory

Virtual memory is a memory management technique that allows flexible mapping of physical memory to a program's address space. The program is exposed to an entirely virtual address space, which on memory access is address-translated into the corresponding physical address through the CPU's Memory Management Unit (MMU). In practice, pages of physical memory are mapped to pages of virtual memory, instead of arbitrary byte-to-byte mapping.

It is common to say that "all problems in computer science can be solved by another level of indirection"[1]. Virtual memory is a layer of indirection that solves the problem of fragmentation, as non-continuous pages of physical memory can be transformed into continuous pages of virtual memory. This is why the Linux Kernel can use a buddy system [37], which is relatively fragmentation-prune, to manage pages of physical memory. With virtual memory, fragmentation is not a problem since fragmented pages of physical memory can be made into large continuous segments of virtual memory to be given to applications or used by the operating system itself.

An important aspect is that pages that have been given to a program by the operating system do not have to be necessarily backed up by physical memory. When such a page is first accessed

[1] By David Wheeler, inventor of the subroutine.

by the program, a page fault (an hardware exception) is raised, which has to be handled by the operating system. The operating system then maps the page to physical memory, and the memory access can then continue normally. Pages that maps to physical memory are called *committed*. Pages that have no backing of physical memory are called *uncommitted*. The fact that the operating system can provide pages that are not mapped yet to physical memory can complicate the memory allocator design, as pages that have already been used for prior block allocations become preferable to pages that have never been used before, in other to reduce real memory usage. Moreover, an operating system providing uncommitted pages has implications in how the dynamic memory allocator may choose to organize and place internal data structures. Techniques such as boundary tags lead to interweaving of internal data structures with blocks, which cause the memory allocator to touch pages that will need to be backed up by physical memory.

Another important observation is that the operating system's memory allocator is much better suited to fulfill large allocations of pages than the program's dynamic memory allocator, as the operating system is capable of transforming non-continuous physical memory into continuous virtual memory. We can understand then that the role of a user-space dynamic memory allocator can be reduced to managing smaller allocations, that are no larger than a few pages, while larger allocations can be deferred to the operating system.

The common saying "all problems in computer science can be solved by another level of indirection" is concluded with "... but that usually will create another problem". Unfortunately, this also applies to virtual memory. Besides the operating system complexity and slower overall memory access due to this extra layer of indirection, the operating system now has an additional resource to manage – the virtual address space of each program. Managing the virtual address space of a program is an analogous task to managing memory in a physical device, i.e., it is a memory allocation problem, and thus must be an additional memory allocator to fulfill this task. In 64-bit architectures, such as x86-64 or ARMv8, address space is plentiful and thus there are no particular concerns for managing it. In legacy 32-bit architectures however, there is a real concern of the scarcity of the address space, and a user-space dynamic memory allocator must be aware to not induce fragmentation of the address space.

2.7 Chapter Summary

In this chapter, we gave an introduction to the subject of memory allocation, including an overview of classical research, notions of fragmentation and memory allocation mechanisms. We also discussed practical issues such as the effects of a memory allocator on program execution, and the impact of virtual memory in the relationship of the memory allocator with the operating system.

Chapter 3

Principles of Multithreading

In this chapter, we introduce and discuss multithreading and multiprocessor systems. This chapter serves to build a strong foundation for Ch. 4, where we discuss multithreaded memory allocation, which is much more focused on performance than classical memory allocation, and thus we will spend some time describing important details of memory behavior in modern multiprocessor systems.

The advent of multiprocessor systems with higher and higher processor counts came at the same time as memory increased in size and decreased in cost. Both of these changes had deep implications on memory allocation, as focus shifted to performance and scalability, often in detriment of fragmentation and memory usage. Multithreaded memory allocation requires not only dealing with techniques to ensure the correct functioning of the allocator for concurrent memory requests, but also having a deep understanding of how memory behaves in a multithreaded environment in order to achieve decent performance.

Multithreaded applications make use of several units of execution, or *threads*. Threads are scheduled by the operating system scheduler and can do work in parallel if run on different processors or cores simultaneously. All threads share the same address space in the application. In particular, the heap and the data program segments, which contain dynamically and statically allocated memory, respectively, can be used by all threads in an application. The scheduler can preempt (e.i., interrupt) any thread at any time and thus interrupt any ongoing work by a thread. Threads that operate on shared data or data structures that can be modified by other threads need to employ *synchronization primitives* in order to ensure correct program behavior since, otherwise, the use of shared memory without synchronization can lead to problems such as race conditions [42].

3.1 Memory Access Impact

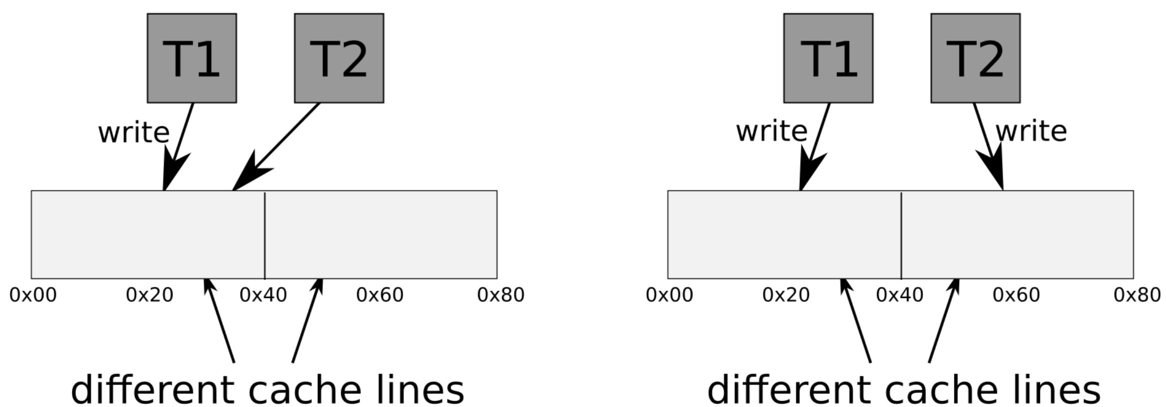
Modern multiprocessor systems contain several processors (or cores), each capable of running a thread at a time. Each processor has access to a set of caches, ranging from the small but very

fast L1 and L2 caches, that are private and used exclusively by the processor, to the slower but larger L3 cache, which is shared between multiple processors. These caches are used to decrease the average access time to memory, by fulfilling a large portion of the memory access requests, as access times to main memory can be an order of magnitude higher than access times to any of the caches. When the main memory is accessed, data is copied to caches in fixed size blocks called *cache lines*, which are 64-byte blocks in most modern multiprocessor systems.

Multiple processors may contain copies of the same data in memory in their private caches. When one processor updates that data (e.g., writes to memory), the copies in the private caches of other the processors must be *invalidated*, or otherwise processors will have conflicting, incoherent views of memory. Cache coherence protocols [5] ensure that different processors view memory in a coherent way, at some expense in terms of system complexity and degradation of memory access times for some memory access patterns. Cache coherence protocols use cache lines as the fundamental unit of data where coherence is maintained.

3.1.1 Cache Line Contention

A well known memory access pattern that causes performance degradation occurs when simultaneous writes to the same cache line are done by several processors, causing the cache coherence mechanisms to invalidate data in each processor's private cache and forcing both processors to communicate. We will refer to this type of performance degradation as *cache line contention*, for lack of an existing term that describes this type of cache contention. Figure 3.1 summarizes how cache line contention occurs. Figure 3.1(a) shows a case where degradation occurs if two threads running in parallel, T1 and T2, write to the same cache line. The cache coherence mechanism must order those two writes and invalidate the caches of the processors running T1 and T2. However, if writes are done to different cache lines, as is the case in Fig. 3.1(b), then degradation no longer occurs.



(a) Degradation occurs if threads T1 and T2 write to the same cache line simultaneously

(b) No performance degradation occurs if threads T1 and T2 write to distinct cache lines simultaneously

Figure 3.1: Overview of degradation when writing simultaneously to memory

3.1.2 False Sharing

A phenomena related to cache line contention is *false sharing* [49]. False sharing can be described as performance degradation that occurs due to seemingly independent objects being unintentionally placed in the same cache line, causing cache line contention if updates are done to those objects. Figure 3.2 has an example of false sharing occurring due to the placement of two objects A and B in the same cache line. Simultaneous writes to A and B by different threads will cause cache line contention.

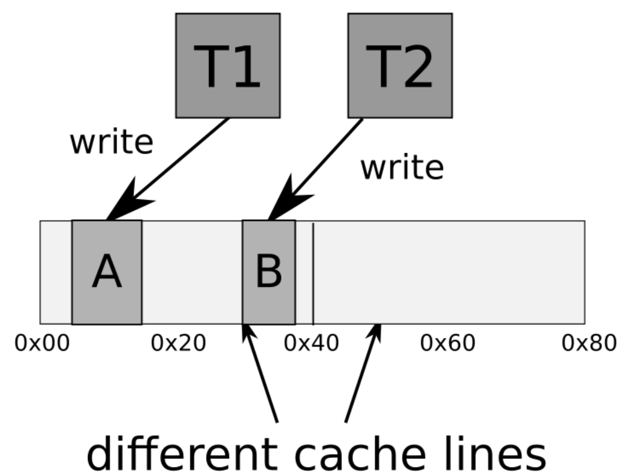


Figure 3.2: False sharing due to the placement of different objects in the same cache line

Placement of objects in the same cache line is only problematic for shared memory – memory which can be accessed and written to by multiple threads in the application. False sharing does not occur when objects are placed in the same cache for read-only data, such as code (which resides on the text segment), or local data, such as local function variables (which reside on the stack segment). Therefore, the issue of placement can be reduced to statically and dynamically allocated objects, on the data and heap segments.

Generally speaking, prevention of false sharing in statically allocated data relies on the programmer, who has to specify the layout in memory, size and alignment requirements of objects. False sharing in dynamically allocated data is less clear. The dynamic memory allocator is responsible for allocating dynamic blocks of data, and may place blocks in the same cache line and then distribute them to different threads, which will lead to false sharing. An ideal solution would be for the dynamic memory allocator to never place two blocks on the same cache line, but as will be seen in Ch. 4, that may come with an unacceptable increase in memory usage. Another alternative is for the dynamic memory allocator to only give blocks in the same cache line to the same thread, but false sharing may still occur if the application then has several threads operating on those blocks, at which point the programmer is once again responsible for explicitly allocating blocks which will not share a cache line by using memory allocator functions such as **aligned_alloc()**.

3.2 Synchronization Primitives

Threads need to use shared data and data structures in order to communicate and coordinate with other threads. However, the use of shared data presents a challenge, as modern processors and compilers are optimized for single-threaded applications, and many of these optimizations manifest themselves by effectively reordering program access to memory. In a multithreaded environment, due to memory access reordering, an application may no longer work as intended. Various factors in modern systems may reorder access to memory:

- Compiler
  - Register allocation
  - Subexpression elimination
  - Optimization heuristics
- Processor
  - Prefetching
  - Speculative execution
  - Out-of-order execution
- Cache
  - Store buffers
  - Private processor caches

Synchronization primitives are in essence operations that stop the reordering of accesses to memory in order to allow synchronization of data that can be accessed by multiple processors and that can thus be used to write and reason about multithreaded applications. Synchronization primitives are fundamentally slower than other native operations, as they prevent memory access reordering and require inter-processor communication in order to achieve synchronization of what is stored in the main memory and in the caches of the other processors. There are several types of synchronization primitives:

- Atomic operations
- Locks (i.e., mutual exclusion)
- Transactional memory

In this document, we will not cover transactional memory [21], as it is not yet a widely available feature in multiprocessor systems, and software transactional memory is not a sufficiently high-performance alternative [8].

3.2.1 Atomic Operations

Atomic operations are instructions that cannot be interrupted by the operating system's scheduler and that appear to run instantaneously to the other processors. The available atomic operations depend on the architecture used by the CPU and not all atomic operations are universally available. The following instructions are commonly available:

- Compare-and-swap
- Test-and-set
- Fetch-and-add
- Load-link and store-conditional

All the instructions above are atomic read-modify-write operations, i.e., operations which read a value from memory and write a new value atomically. They operate on a single processor word. Listing 3.1 shows the pseudo-code for all atomic operations.

```
bool CompareAndSwap(int* address, int expected, int desired) {
  if (*address == expected) {
    *address = desired;
    return true;
  }
  return false;
}

bool TestAndSet(bool* address) {
  bool value = *address;
  *address = true;
  return value;
}

int FetchAndAdd(int* address, int increment) {
  int value = *address;
  *address = value + increment;
  return value;
}

int LoadLinked(int* address) {
  return *address;
}

bool StoreConditional(int* address, int desired) {
  if (address not written to since last LoadLinked) {
    *address = desired;
    return true;
  }
  return false;
}
```

Listing 3.1: Common atomic operations

All the atomic operations in Listing 3.1 can be used to implement higher level synchronization primitives such as locks. Here, we will focus on the description of the compare-and-swap (CAS)

and the load-link/store-conditional (LL/SC) instructions, as they are of particular relevance for non-blocking algorithms and lock-free programming. Furthermore, the other atomic operations can be efficiently emulated with CAS or LL/SC.

The most relevant atomic instruction is CAS, which is widely supported in just about every modern architecture either directly or by efficient emulation by using load-link and store-conditional. CAS takes in three arguments – a memory address, an expected value for that address and a desired value. CAS updates the address to the desired value and returns true if upon reading, the address contains the expected value; if the address does not contain the expected value, CAS returns false and does nothing else. Some instructions implement or expose a weaker form of CAS, called a *weak CAS*, which may return false and do nothing even if the provided address contains the expected value. When a weak CAS fails in this fashion, it is said to have failed spuriously. CAS is capable of efficiently emulating all other atomic operations, with the exception of load-link/store-conditional.

LL/SC together make up an atomic read-modify-write operation that can be used to emulate all of the other atomic operations, including CAS. LL/SC is particularly important for lock-free constructs, as it solves the ABA problem [10] – a fundamental and challenging problem which complicates the implementation and design of lock-free data structures and algorithms. An ideal version of LL/SC works as follows:

1. LL takes a single argument – the address to be read and returned.
2. SC takes two arguments – the address previously provided to LL and a value to be stored.
3. LL/SC work in pairs – a LL must be followed by a SC, and a SC must be preceded by a LL.
4. SC updates the address to the desired value if the address has not been written to since the corresponding LL, in which case it returns true; otherwise, it returns false.
5. An arbitrary number of other instructions can be executed between the LL and the SC.

In practice, no computer architecture implements the ideal version of LL/SC described above, and some architectures do not implement LL/SC at all. No form of LL/SC is available in widely-used architectures, such as x86 or x86-64, and other architectures only implement weak or restricted forms of LL/SC, which can fail spuriously or that have restrictions on what or how many instructions can be executed between LL and SC.

Excessive usage of any atomic operation on a memory address by multiple threads leads to cache line contention, as described earlier in Sec. 3.1. This type of performance degradation is added to the overhead that atomic operations have due to the consistency and ordering guarantees they provide.

3.2.2 Locks

Mutual exclusion locks are a popular synchronization primitive used in multithreaded applications. Locks can be used for synchronization by ensuring mutually exclusive access to a resource. A thread that wants to use the resource protected by a lock has to *acquire* the lock before being able to access the resource. Once it has used the resource and no longer needs it, it can then *release* the lock, making it available for other threads to acquire it. While the lock is acquired, no other threads can access the resource. Threads that seek to acquire the lock will *block* until the lock can be acquired. How a thread blocks depends on the lock implementation. A *spinlock* is a lock that blocks by simply waiting in a loop (spinning) while trying to acquire the lock. Other lock implementations may instead choose to suspend the acquiring thread.

Locks are a simple and easy-to-use mechanism to ensure synchronized access to data. However, it is a *blocking* construct, as threads that wait to acquire a lock will make no progress. The use of locks, or of any other blocking construct has implications in terms of program performance, availability, robustness and flexibility. Careless use of locks may lead to deadlocks or livelocks [27], putting threads permanently waiting to be able to acquire a resource. Deadlocks occur when threads are simultaneously waiting on each other for access to a resource. Consider the following situation with two locks L1 and L2 and two threads T1 and T2. T1 and T2 will deadlock if: T1 has acquired L1, T2 has acquired L2, and T1 and T2 are also trying to acquire L2 and L1, respectively. Livelocks are similar to deadlocks, but rather than permanently waiting for the same resource, a thread in livelock undergoes changes after failing to acquire a lock to a resource.

The use of locks can also lead to priority inversion, which are situations where threads that have been assigned a higher priority can be preempted due to a lower priority thread. If a lower priority thread holds a lock that a higher priority thread has to acquire, the higher priority thread will have no progress until the lower priority thread releases the lock. This effectively subdues the priorities that have been assigned to these two threads. In a similar fashion, threads that are waiting on a lock held by other thread may suffer arbitrarily delays, until the thread that is holding the lock releases it. The higher the use of a lock, the more likely are threads to suffer delays before being able to acquire the lock. This is called *lock contention*.

Lastly, progress may halt for the entire application if locks are used and a thread unexpectedly terminates. A thread that is holding a lock and unexpectedly terminates will not release the lock, and all other threads that later try to acquire it will permanently block.

3.3 Progress Guarantees

Algorithms designed to work in multithreaded environments may be described in terms of progress guarantees – what guarantees in terms of program progress are given to individual threads or to the system as a whole when threads are given a time slot to execute by the operating system's scheduler. Herlihy *et al.* [21] list and categorize progress guarantees. We identify progress

guarantees that we consider important and that can be given by concurrent algorithms:

- Blocking
- Non-blocking
  - Lock-free
  - Wait-free

Blocking can be described as a lack of progress guarantee rather than a progress guarantee in itself. Algorithms that use locks are generally blocking. All of the disadvantages associated with locking that have been described previously in Sec. 3.2 are generalized to blocking algorithms.

Lock-freedom and wait-freedom are both non-blocking forms of progress guarantee. Lock-freedom algorithms guarantee that there is system-wide progress if at least one thread is executing. Individual threads may not progress, but only if there is another thread that is actively progressing. In practice, a large number of efficient lock-free versions of various data structures have been proposed [3, 38, 41, 50].

Wait-freedom is the strongest form of progress guarantee. A wait-free algorithm has a bounded number of steps required to achieve completion, i.e., a thread that executes a sufficient number of steps will always progress. With wait-freedom, individual threads always progress, regardless of the state of the other threads. Wait-free algorithms, despite owning desirable properties, are typically inefficiently and not used in practice [14].

Non-blocking algorithms are implemented with atomic operations, but the atomic operations and data structures used by the algorithm must themselves be non-blocking. A wait-free algorithm, for example, requires the use of wait-free primitives. In general, atomic operations that are supported by the processor architecture are wait-free. However, not all atomic operations are available on all processor architectures. Wait-free algorithms may not be able to use emulated atomic operations, as they may be lock-free rather than wait free.

Listing 3.2 shows a simple non-blocking function, **CountCalls()**, which simply counts how many times it has been called. **CountCalls()** would be wait-free if the fetch-and-add operation it uses is wait-free. However, this particular fetch-and-add is emulated using CAS. In the emulated fetch-and-add, CAS will fail if another thread has modified the counter variable while the currently executing thread has already read the counter value in line 11 but not yet executed the CAS on line 13. CAS may fail in this fashion an infinite number of times, in which case the currently executing thread will never progress. Thus, the emulated fetch-and-add is not wait-free. But, it is still lock-free, because CAS only fails if another thread in the application has made progress, thus ensuring system-wide progress.

```
void CountCalls()
{
  static int count = 0;                                        // global variable
  FetchAndAdd(&count, 1);                                      // increment counter by 1
}

int FetchAndAdd(int* address, int increment) {
  // emulate fetch-and-add with compare-and-swap
  int expected, desired;
  do {
    expected = *address;                                       // atomically read old value
    desired = expected + increment;                            // prepare new value
  } while (!CompareAndSwap(address, expected, desired));       // try to swap value
  return expected;
}
```

Listing 3.2: Lock-free emulation of fetch-and-add

Some non-blocking algorithms require the use of load-link/store-conditional, or are otherwise vulnerable to the ABA problem [10]. In a nutshell, the ABA problem occurs when an algorithm assumes that when an address is read twice and the value is the same for both reads, then no changes were made to that address. Situations that can violate this assumption might cause an ABA problem. Listing 3.3 shows one such example, in an implementation of a lock-free stack where **Push()** works correctly, but **Pop()** is vulnerable to ABA. ABA will occur if a thread T1 halts just before executing CAS in line 24 and another thread T2 removes the nodes represented by the variables *head* and *next* from the stack with two **Push()** and then adds *head* back to the stack with **Pop()**. When T1 resumes, the CAS at line 24 will succeed, unintentionally adding the removed node *next* back to the stack, thus compromising the consistency of this data structure.

```
struct Node {
  Node* next;
};

struct Stack {
  Node* head;
};

void Push(Stack* stack, Node* node)
{
  Node* head;
  do {
    head = stack->head;
    node->next = head;
  } while (!CompareAndSwap(&stack->head, head, node));
}

Node* Pop(Stack* stack) {
  while (1) {
    Node* head = stack->head;
    Node* next = head->next;
    // an ABA can occur here if head and next are popped by another thread
    // and head is then pushed back into the stack
    if (CompareAndSwap(&stack->head, head, next))
      return head;
  }
}
```

Listing 3.3: Example of a lock-free stack vunerable to the ABA problem

ABA prevention in algorithms that are prune to ABA can be done with the use of LL/SC instead of CAS, but in practice, few processor architectures have support for LL/SC and none with the ideal LL/SC semantics. Other techniques are used instead, such as version tags [18] (which store a version counter and data in a single processor word) or hazard pointers [39].

3.4 Chapter Summary

In this chapter, we discussed various multithreading topics that are relevant to memory allocation, such as, synchronization primitives, disadvantages of locking and various forms of performance degradation. We also discussed progress guarantees, and how lock-freedom offers desirable properties and is more feasible to be achieved in practice than wait-freedom.

Using the concepts introduced in this chapter, the following chapter introduces and discusses multithreaded memory allocation.

Chapter 4

Multithreaded Memory Allocation

In this chapter, we discuss memory allocation in multithreaded environments. We start by discussing how modern memory allocators reduce the cost of synchronization, and then describe concerns that are specific to multithreaded memory allocation. Next, we list prior work, and describe some of the concurrent memory allocators that are used in practice.

Multithreaded applications require memory allocators to be capable of dealing with concurrent memory requests without inconsistency or failure. In particular, memory allocators have to be *thread safe*. Compared to serial memory allocators that operate in a single-threaded environment and can use any of the mechanisms described in Ch. 2, a concurrent memory allocator must employ the use of synchronization primitives in order to function correctly in the presence of concurrent memory requests.

Furthermore, modern memory allocators have to prioritize application performance, both in terms of block placement and in terms of average memory application latency when fulfilling memory requests. Concurrent memory allocators are expected to be able to scale, e.g., achieve a higher rate of memory requests as more threads are used by an application. These performance requirements often come at a cost in memory usage, either through higher fragmentation, or due to additional memory required by the memory allocator's internal structures.

4.1 Mitigating Synchronization Costs

A modern memory allocator has to be able to operate in a multithreaded environment and thus it has to be able to fulfill memory requests concurrently. Serial memory allocators such as dlmalloc [31] were initially adapted to multithreading environments by simply wrapping the entire memory allocation logic with a single lock, which preserves the memory allocator's fragmentation behavior, but ultimately degrades performance and limits scalability, as lock acquire and release have to be done on each allocation and deallocation request. With a single coarse lock over the entire memory allocation mechanism, all the memory management is serialized and no requests can be fulfilled in parallel.

Some of the memory allocation mechanisms described in Ch. 2 allow multiple fine-grained locks to be used, each of which can protect a specific part of the mechanism. Classes of mechanisms such as segregated free lists can use a lock per list. In general, fine-grained locking reduces lock contention and allows tasks to be done in parallel. In the context of a memory allocator mechanism, fine-grained lock has somewhat more limited benefits, as applications use only a few distinct block sizes, which leads to the same data structures in the memory allocator (or free lists in the case of a segregated free lists mechanisms) being used, thus leading to the same lock contention and scalability problems described earlier. Nevertheless, use of fine-grain locking allows for some memory requests to be truly fulfilled in parallel, particularly when different threads are requesting different block sizes.

An idea explored by some early memory allocators in order to avoid synchronization is to have a private memory allocator for each thread in an application However, because memory is shared across all threads, threads can *remote free* blocks. A remote free is a memory deallocation that is called by a different thread than the one to which the block was originally allocated to. Remote frees complicate designs that use a private memory allocator per thread, as it forces some synchronization on deallocation requests. Berger *et al.* [4] name this type of memory allocators *pure private heaps* and they discuss how some of these designs can cause unbounded memory usage.

Modern concurrent memory allocators use strategies to decrease the impact of synchronization, at some expense in terms of memory usage and fragmentation, namely *arenas* and *thread caches*.

4.1.1 Arenas

Arenas are a technique to reduce lock contention, at the expense of increased memory overhead and fragmentation. The term arena was introduced by Evans [13], but earlier works [4, 30] often refer to it as the use of several heaps. Each arena is conceptually a fully fledged memory allocator that can handle concurrent memory allocation and deallocation requests done by multiple threads, which distinguishes it from the notion of a private memory allocator per thread as described earlier.

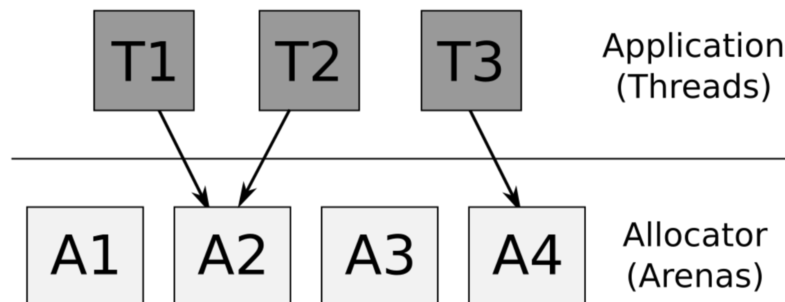


Figure 4.1: Multiple threads using several arenas

The number of threads in an application is not known in advance, and may vary over the course of the application. As threads are created and destroyed in the application, more or less threads may be using the same arena simultaneously. The memory allocator must assign arenas to threads in a way that minimizes lock contention. Various strategies can be used to determine the arena to be used in an allocation request. We identify some strategies used for arena assignment:

- The arena identified by hashing the thread's ID [4, 30];
- The arena with fewer threads assigned to it [13];
- An arena not locked yet [19].

In general, on deallocation requests, there is no arena assignment, as blocks are ideally returned to the arena they were originally allocated from. This requires the memory allocator to store and use information that allows it to associate an arena to every block given to the application.

4.1.2 Thread Caches

Thread caches enable memory allocation and deallocation requests to be serviced without the use of synchronization. Conceptually, each thread maintains a private list of blocks that it can use to fulfill requests. Synchronization is only required when transferring blocks from and to the private list, in cases where the private list is too full or has no block that can fulfill a request. Thread caches cannot function in isolation, and require the use of a lower-level memory allocation mechanism that the thread caches can use to obtain and return lists of blocks. Figure 4.2 shows a logical overview of the use of thread caches. Note that there is a single thread cache per thread, and that interactions between thread caches and individual threads result in transfers of a single block, while interaction between thread caches and the rest of the allocator result in transfers of lists of blocks.

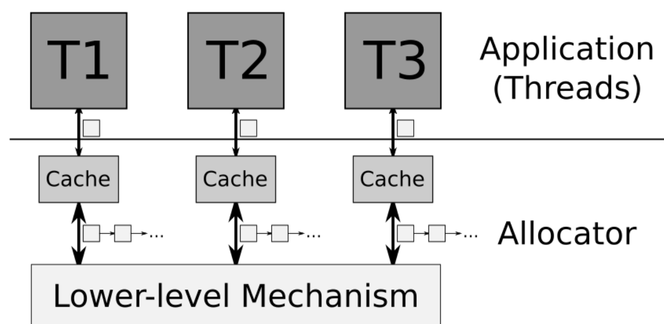


Figure 4.2: Logical overview of an allocator that uses thread caches

The use of thread caches is largely motivated by the high cost of synchronization primitives. Even in ideal conditions, without degradation due to cache line contention, synchronization primitives are much slower than regular instructions and are thus the most significant cost of servicing a memory request. This leads to the conclusion that synchronization primitives are too expensive to be used in every memory request, and that is why most allocations need to be fulfilled without synchronization at all. Thread caches meet these requirements, at some obvious trade-off in terms of memory usage. Modern concurrent memory allocators make extensive use of thread caches, as it allows an average allocation and deallocation case that is extremely fast since it requires no synchronization.

In practice, thread caches are implemented with simple segregated storage, a mechanism described in Ch. 2. Simple segregated storage is ideal for this purpose, as it presents excellent performance and locality characteristics and does not support coalescing, which cannot be done anyway without synchronization due to remote frees. In the context of simple segregated storage, memory allocations are fulfilled by finding the appropriate free list and then removing the first block. Deallocations are serviced by adding a block to the appropriate free list. Both of these operations can be implemented with very few instructions.

Trade-offs can be made in thread caches between performance and memory usage. Higher performance can be achieved by allowing the thread cache to contain more blocks and by transferring a large number of blocks whenever synchronization is required. However, this comes at a cost in terms of memory. Some modern memory allocators such as TCMalloc [16] use adaptive algorithms to dynamically adjust how many blocks are kept, and some work has been done in improving these adaptive algorithms [32].

Other trade-offs can result from how free lists in the simple segregated storage mechanism are implemented. Most commonly, they are implemented as a singly-linked list, by storing a single processor word which is used as the pointer to the list head, while the rest of the list is composed by using the blocks' own memory. This requires minimal memory usage, and allows for adaptive algorithms to be used without list size limitations. However, it forces the allocator to touch the blocks of memory beforehand, which leads the operating system to allocate physical memory for all the pages being used. Furthermore, the push and pop operations used to add and remove blocks to the singly-linked list have memory dependencies which can stall instruction execution and affect instruction-level parallelism done by modern CPUs. A recent work [26] suggests memory hardware acceleration to alleviate this issue and improve throughput. An alternative by Jemalloc [13] is to use a fixed-size array of pointers to compose the list. This limits adaptive algorithms, as the array has a maximum size, and requires more memory than a single pointer. However, it allows the allocator to manage blocks without touching them (thus not forcing the operating system to allocate physical memory), and removing/adding a block from the array has no memory dependency, which represents a slight performance gain.

4.2 Performance and Fragmentation Concerns

Memory allocation in a multithreaded environment has to deal with some additional concerns when compared to a serial execution. Berger *et al.* [4] describe and highlight a couple of problems that are unique to memory allocation in multithreaded environments, namely *false sharing* and *blowup*.

4.2.1 Allocator Induced False Sharing

As detailed in Ch. 3, false sharing is a performance degradation phenomena that occurs in multiprocessor systems when seemingly independent objects used by different processors are unintentionally placed on the same cache line, causing cache line contention.

A concurrent memory allocator may cause false sharing by interleaving memory allocations that are given to different threads. Berger *et al.* classify false sharing induced by memory allocators into two categorizes: *active false sharing* and *passive false sharing*. Memory allocators are said to actively induce false sharing if they distribute blocks that are in the same cache line to different threads, regardless of properties in the program's request stream. Active false sharing was common among early concurrent memory allocators which were built using a serial memory allocator (such as dlmalloc [31]) with a synchronization primitive on top.

Concurrent memory allocators can passively induce false sharing if they distribute blocks that are in the same cache line to the same thread and, then, the program performs remote frees. In modern concurrent memory allocators, passive false sharing generally happens because of caching mechanisms. For example, consider an allocator with a cache per thread that can store a limited number of blocks, where allocation is done by removing a block from the cache, and deallocation is done by adding a block to the cache. Suppose now that a thread T1 allocates two blocks B1 and B2, which are on the same cache line. Later, if thread T1 passes block B1 to a thread T2, there will be false sharing, albeit not induced by the allocator. Moreover, if thread T2 also deallocates block B1, the next allocation by T2 may be fulfilled with block B1 due to the cache, which will cause false sharing, this time induced by the memory allocator.

In general, multithreaded applications can always induce false sharing by moving blocks between threads. Concurrent memory allocators can only prevent overall false sharing by ensuring that no two blocks are ever in the same cache line, effectively increasing the minimum block size to a cache line. However, this is an unacceptable cost in terms of fragmentation, and may increase memory usage dramatically, with small 8 and 16 byte blocks increasing to 64 bytes, the most common cache line size. To the best of our knowledge, there is no concurrent memory allocator that deals with false sharing in this manner, only custom memory allocation tools such as **cache_aligned_allocator<T>** in Intel TBB [44],

4.2.2 Blowup

Blowup is a particular type of fragmentation only present in concurrent memory allocators. Blowup is defined as the ratio between the maximum memory usage by a given concurrent memory allocator in a multithreaded environment divided by the maximum memory usage by an ideal serial memory allocator. Berger *et al.* [4] have shown that some early concurrent memory allocators have blowup that range from P (the number of processors) to unbounded memory consumption.

Unbounded blowup occurs in schemes where each thread has a private memory allocator. This type of scheme can be used to service allocations and deallocations with no synchronization. However, deallocations of blocks originally allocated by other threads is troublesome, since such blocks need to be returned to the correct private memory allocator, which uses no synchronization. A solution is to *defer* the deallocation of a block by placing it in a temporary list place from where the original thread can remove, and deallocate it. However, this may lead to unbounded memory consumption in scenarios where a thread T1 repeatedly allocates blocks and passes them to thread T2, which deallocates them, adding ever more blocks to the temporary list. This type of scenario is known to happen in practice with programming patterns such as producer-consumer.

To prevent unbounded blowup, concurrent memory allocators must ensure that a bounded maximum number of free blocks can be kept in any component of the memory allocator at all times.

4.3 Concurrent Memory Allocators

This section provides short summaries on some of the existing concurrent memory allocators. For some memory allocators, especially those used in practice, we base ourselves on the analysis of the source code, due to the lack of available documentation on any academic work. However, memory allocators are non-trivial pieces of software, and thus we cannot guarantee the correctness of our analysis. Memory allocators are implemented as a library, and aspects such as the use of thread-specific storage, dynamic loading and cross-platform portability (not only different hardware architectures but also different operating systems) complicate not only the implementation but also the overall comprehension.

Furthermore, we present no taxonomy of concurrent memory allocators, as none exists that appropriately describes existing mechanisms and techniques. We do not present our own taxonomy either, as creating one that better describes the current state-of-the-art is a work in and of itself. There are some taxonomies presented in prior works [4, 17], but we argue that they fail to describe advances made in concurrent memory allocation in the last decade.

4.3.1 Hoard

Hoard is a concurrent memory allocator, introduced by Berger *et al.* [4], which highlights the importance of avoiding blowup and false sharing in a setup that attempts to reduce contention costs.

Hoard uses a number of *heaps*. There is a global heap, from which all memory is obtained from, and then there is one heap per processor. Heaps manage and own *superblocks*, which are continuous sets of pages used to carve out blocks, to be given to the application to satisfy an allocation request. Superblocks are created in the global heap, and transferred between the global heap and the processor heaps. Superblocks contain equal-sized blocks, belonging to a single size class. Memory allocations are fulfilled by accessing a heap (by hashing the current thread's ID) and finding a free block with the appropriate size in a superblock. Deallocations are fulfilled by finding the superblock that the provided block belongs to, and adding it to the superblock's free block list. Hoard maintains some statistics to ensure that no heap has too much free memory, thus bounding blowup to $O(1)$.

Hoard achieves decreased synchronization costs by employing an arena strategy, where each heap is an independent concurrent memory allocator. Although not described in the original paper, recent versions of Hoard make use of thread caches[1].

4.3.2 Ptmalloc2

Ptmalloc2 [19] is the standard memory allocator distributed with glibc, and thus the most commonly used memory allocator in Linux systems. It is a modified thread-safe version of Doug Lea's memory allocator [31], a fragmentation-focused memory allocator that uses a best-fit mechanism with a deferred coalescing scheme. For coalescing, ptmalloc2 uses boundary tags to store metadata per block, causing every memory allocation to cost at least one additional processor word in memory overhead.

Ptmalloc2 has been updated over time with features to improve its multithreaded performance, such as the use of arenas and thread caches[2]. Each arena has an associated lock that must be held before performing any operation on the arena. Ptmalloc2 has an unique arena selection strategy – on allocation, an arena that is not yet locked is chosen, and if none is found, a new arena is created so that it may be used to allocate memory. With this strategy, ptmalloc2 can have a varying number of arenas at runtime.

Overall, ptmalloc2 is a concurrent memory allocator that prioritizes low memory usage and fragmentation along with portability (hence why it is used as the default glibc memory allocator), rather than performance and scalability.

[1] Thread caches in Hoard are called Thread Local Allocation Buffers (TLABs). Curiously, this term is also used to describe a mechanism akin to thread caches in the JVM's garbage collector.

[2] In ptmalloc2, thread caches are disabled by default, so whether thread caches are actually used will depend on the glibc compilation options.

4.3.3 Jemalloc

Jemalloc [13] is a state-of-the-art memory allocator that focuses on fragmentation avoidance and scalable concurrency support. It is the default memory allocator on the FreeBSD operating system and is widely used in numerous multithreaded applications. Jemalloc was initially designed with the use of arenas, and more recent versions also use thread caches.

At application start, Jemalloc creates a number of arenas equal to 4 times the number of CPUs in the system in an effort to minimize lock contention. Threads are assigned to arenas in round-robin fashion, i.e., new threads are assigned to the arena that has the lowest number of assigned threads. Jemalloc's thread caches are a simple segregated storage mechanism that uses size classes. To obtain blocks for the thread caches, Jemalloc carves out *slabs* (a set of pages used for equal-sized blocks) from *extents* (a region of memory that can hold several slabs or larger allocations). Arenas manage extents, and extents contain logical blocks that can be coalesced.

Jemalloc achieves very good segregation between application data and memory allocator data. In fact, Jemalloc is the only concurrent memory allocator that implements thread caches with arrays of pointers to blocks instead of assembling a linked list with the blocks' free memory itself, presumably to avoid touching the free blocks' memory. This ties up nicely with the rest of the allocator, which also does not touch memory, and thus Jemalloc is particularly well suited for applications which request memory that they do not use. Furthermore, Jemalloc also explicitly tracks regions of memory that are being reused, and leverages that information to reuse existing physical memory rather than to give blocks in pages that might still be uncommitted.

In practice, Jemalloc achieves state-of-the-art performance with good memory usage characteristics, and can be considered an ideal concurrent memory allocator for multithreaded applications.

4.3.4 TCMalloc

TCMalloc [16] is a state-of-the-art concurrent memory allocator with a focus on thread caching. Unlike other allocators, it does not use arenas to decrease contention on shared data structures. Instead, it uses a strategy of fine-grained locking to decrease contention, and employs a number of adaptive cache management algorithms.

TCMalloc has three main components: (i) the *thread caches*, a synchronization-free component implemented with simple segregated storage and using the free blocks' memory to compose singly-linked lists that make up each free list; (ii) the *central cache*, a synchronized component that keeps lists of blocks that can be quickly transferred between individual thread caches and the central cache; and (iii) the *page heap*, which maintains *spans*, TCMalloc's nomenclature for continuous pages in memory that can be used to carve lists of blocks. The page heap is implemented with a segregated fits mechanism, capable of coalescing and is the component that obtains and returns memory to the operating system.

TCMalloc's implementation of thread caches have been subject to further study in academic works. Improved adaptive cache management algorithms [32] as well as hardware acceleration [26] have been proposed to further improve cache performance.

4.3.5 Michael's Lock-Free Allocator

Michael's Lock-Free Allocator (MLFA) [40] is an academic lock-free memory allocator, which only uses widely available and portable CAS operations for lock-free synchronization. MLFA presents a simple lock-free memory management mechanism somewhat akin to simple segregated storage.

MLFA uses *superblocks*, continuous set of pages which can be used to carve out equal-sized blocks. Superblocks only contain blocks of a specific size class, and contain no metadata. Metadata for superblocks is instead stored on *descriptors*. Descriptors are unreclaimable objects (i.e., once allocated, they can never be returned to the operating system) that contain superblock metadata and that compose lock-free shared data structures used in MLFA. Each descriptor contains an *anchor*, an object small enough to fit inside a processor word that can describe the superblock's state, namely the number of free blocks, the first free block available (the remaining of the list is assembled using the free blocks' memory), and a state.

In MLFA, superblocks can be in one of four states: (i) *full*, if the superblock has no free block available; (ii) *empty*, if all of the blocks in the superblock are free, at which point the superblock can be returned to the operating system; (iii) *partial*, if the superblock has at least one free block; and (iv) *active*, if the superblock is not empty and is currently used by a processor heap to satisfy memory requests. Superblocks are managed by the *processor heap*, a component that contain one active superblock and a lock-free list of partial blocks (which we will refer to as *partial list*) per each size class. The partial list is composed by descriptors.

Allocations are fulfilled by removing a block from the currently active superblock, which is done by updating the anchor with a CAS in the superblock's descriptor. If that superblock becomes empty, it is removed as the active superblock, and one partial superblock is removed from the partial list, and installed as active. If there is no partial superblock that can be installed, a new superblock is obtained from the operating system. Deallocations are fulfilled by finding the block's superblock and associated descriptor, and adding it to the free list of blocks.

Note that unlike other concurrent memory allocators described so far, MLFA requires the operating system to do some of the memory management, as MLFA itself cannot do coalescing and splitting in order to allocate and deallocate superblocks.

Overall, MLFA is predictably slow as it needs to use at least one CAS per allocation and deallocation. However, it is lock-free, and the author highlights that a lock-free memory allocator provides better guarantees to user applications than lock-based memory allocators, namely the guarantees of system-wide progress, immunity to deadlocks, livelocks, priority inversion and delays due to preemption during locking, as well as tolerance to arbitrary thread termination.

4.3.6 NBMalloc

NBMalloc [17] is an academic lock-free memory allocator inspired by Hoard's architecture. It uses *flat sets*, a novel lock-free data structure that allows the lock-free transfer of blocks between different flat sets.

Similar to Hoard, NBMalloc uses *superblocks*, which are continuous sets of pages from which equal-sized blocks can be obtained. Superblocks contain a free list of blocks, and a header that contains metadata. To manage superblocks, NBMalloc uses per-processor heaps, which stores superblocks in flat sets, and a global heap from which all memory is obtained. The use of flat sets allows the lock-free transfers of superblocks between per-processor heaps and the global heap. On allocation, a processor heap is selected and a superblock with some free blocks is obtained, so that a block may be removed from it. If the block obtained was the last free block in the superblock, the superblock is removed from the corresponding flat set in the processor heap. On deallocation, the block is added to the free list of its superblock, and if the superblock is completely free, it is returned to the operating system.

NBMalloc uses a substantial number of lock-free operations per allocation and deallocation, making it particularly slow when compared to modern concurrent memory allocators.

4.4 Chapter Summary

This chapter discussed memory allocation in a multithreaded environment. We described various techniques used to mitigate synchronization costs, such as arenas and thread caches, and concerns specific to multithreaded memory allocation, such as blowup. We then described various concurrent memory allocators, both lock-based and lock-free.

Chapter 5

LRMalloc Design and Implementation

This chapter describes the design and implementation of LRMalloc, a lock-free memory allocator that we presented in [33, 34]. LRMalloc leverages lessons of modern memory allocators and combines them with a lock-free scheme. We assume implementation on a 64-bit system architecture (e.g., x86-64) and, for the sake of brevity, we only show C-like code for the key implementation routines.

LRMalloc aims to show the feasibility of lock-free memory allocators for multithreaded applications, by emphasizing that a lock-free memory can be competitive with state-of-the-art lock-based memory allocators that are currently used in practice. Lock-based memory allocators are subject to a number of disadvantages: they are vulnerable to deadlocks and livelocks, prune to priority inversion and delays due to preemption during locking, and incapable of dealing with unexpected thread termination. Lock-free algorithms deal with those disadvantages, while also providing a guarantee that there is progress in the system if there is at least one thread executing. A lock-free memory allocator therefore has properties that are desirable in a modern memory allocator. Moreover, LRMalloc is highly portable, as it only uses CAS operations for synchronization, rather than LL/SC or transactional memory.

5.1 High-Level Overview

From a high-level perspective, LRMalloc is divided into three main components: (i) the *thread caches*, one per thread; (ii) the *heap*; and (iii) the *pagemap*. Figure 5.1 shows the relationship between these three components, the application and the operating system.

Similarly to other memory allocators, the thread caches are a thread-specific component that allows memory allocations and deallocations to be done without synchronization. One thread cache exists per thread. Our implementation of thread caches is a simple segregated storage mechanism that uses stacks to store a finite number of free blocks that are available for each size class. The thread cache interacts with the heap to obtain lists of blocks. When the thread cache is empty, lists of blocks are transferred from the heap to fill it. When it is full, lists of blocks are

transferred back to the heap, so that they may be re-used by other threads or used to return memory to the operating system.

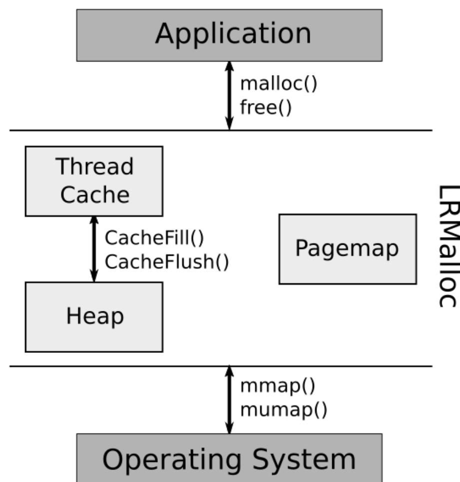


Figure 5.1: Overview of LRMalloc

The heap and the pagemap are lock-free components that can be accessed concurrently by multiple threads. The heap manages *superblocks* from which it carves lists of blocks to be used by thread caches. Superblocks are continuous set of pages, which may either be used to carve up blocks of the same size class or for a *single large allocation*. Similarly to other memory allocators, in LRMalloc, large allocations are allocations larger than the largest size class. As previously discussed in Ch. 2, large allocations are better handled by the operating system due to the operating system's control over virtual memory. The pagemap stores metadata for pages used by superblocks. Its raison d'être is to find out metadata of a given block, such as the size class and superblock the block belongs to. Therefore, the pagemap is merely a bookkeeping component, kept up to date by the heap. The pagemap stores metadata on a per-page basis instead of a per-block basis. This reduces the amount of memory used for bookkeeping, avoids interleaving of allocator data with application data in superblocks and increases the locality of blocks provided to the application.

5.2 Size Classes

LRMalloc, like many other state-of-the-art allocators, uses simple segregated storage for its thread caches and there is one free list for each possible block size. The number of distinct block sizes guides some of the memory usage characteristics of the memory allocator. A large number of distinct block sizes will increase the amount of memory needed for internal data structures, but reduce internal fragmentation. A small number of block sizes has the opposite effect, increasing internal fragmentation and reducing memory overhead for internal data structures. The different block sizes chosen by the memory allocator are called *size classes*. In LRMalloc, size classes are

generated according to the following series (series also adopted by Jemalloc [13]):

1. 2^X
2. $2^X + 2^{(X-2)}$
3. $2^X + 2^{(X-1)}$
4. $2^X + 2^{(X-1)} + 2^{(X-2)}$
5. $2^{(X+1)}$ (repeat, same as first case)

Not all values generated by the series are valid due to alignment requirements on the C standard [9], and thus those are removed. Without those cases, the above series limits internal fragmentation to a maximum of 25% of the allocated memory.

Table 5.1: LRMalloc size classes

Size Class	**Formula**	**Block Size (in bytes)**
1	2^3	8
0	$2^3 + 2^1$	10
0	$2^3 + 2^2$	12
0	$2^3 + 2^2 + 2^1$	14
2	2^4	16
0	$2^4 + 2^2$	20
3	$2^4 + 2^3$	24
0	$2^4 + 2^3 + 2^2$	28
4	2^5	32
5	$2^5 + 2^3$	40
6	$2^5 + 2^4$	48
7	$2^5 + 2^4 + 2^3$	56
8	2^6	64
9	$2^6 + 2^4$	80
10	$2^6 + 2^5$	96
11	$2^6 + 2^5 + 2^4$	112
12	2^7	128
⋮	⋮	⋮
37	2^{13}	8192
38	$2^{13} + 2^{11}$	10240
39	$2^{13} + 2^{12}$	12288
40	$2^{13} + 2^{12} + 2^{11}$	14336
41	2^{14}	16384

Table 5.1 shows the size classes used by LRMalloc. Size classes in gray-filled rows are invalid size classes due to alignment requirements – block sizes must be multiple of word size (in this case, 64 bits, or 8 bytes). The largest size class is equal to 16 kilobytes, or 4 pages. Larger size classes make little sense since the operating system can naturally provide allocations that are a multiple of a page, which guarantees that allocations larger than 4 pages will have, at most, 25%

internal fragmentation, same value as the smaller allocations that use size classes. Furthermore, the larger the size of a block given to the application, the more it makes sense to simply defer that work to the operating system, given that the operating system is fundamentally better suited to answer large block requests, as was described earlier in Ch. 2.

In LRMalloc, *small allocations* are allocations small enough to use a size class and are fulfilled by the thread cache, while *large allocations* do not use a size class, are rounded up to a multiple of page size and are fulfilled by the operating system.

5.3 Thread Caches

Thread caches are the component that allows memory allocation and deallocation to be serviced without the use of synchronization. A thread cache exists per each thread in the application. Thread caches are implemented using simple segregated storage, which has excellent performance and locality properties. Simple segregated storage uses several free lists, as many as there are size classes. Each free list is conceptually a stack, where blocks are added through a push operation and removed through a pop operation. In modern memory allocators, these stacks are implemented in two ways: (i) singly-linked list that uses the memory of the free blocks, thus only needing a single pointer per singly-linked list to store the head; or (ii) as a fixed-size array of pointers to blocks. Each of these implementations has advantages and disadvantages in terms of memory usage and performance, as described earlier in Ch. 4.

LRMalloc implements thread caches with singly-linked lists, since despite performance disadvantages, singly-linked lists allow the transfer between the thread cache and the heap of any number of blocks with a constant number of atomic operations. Furthermore, the use of singly-linked lists is consistent with the remaining components, namely the heap, which uses singly-linked lists to store free blocks. Figure 5.2 shows an overview of thread caches in LRMalloc and the layout of free lists.

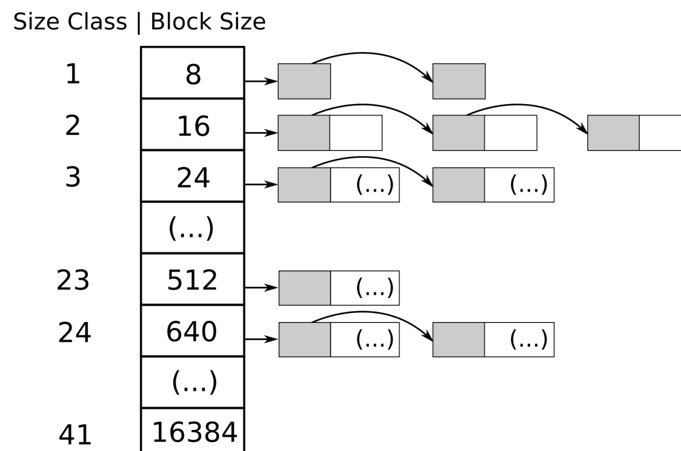


Figure 5.2: LRMalloc thread cache

Ultimately, the thread caches' goal is to provide extremely fast, synchronization-free memory allocations and deallocations. It does so by ensuring that most **malloc()** calls are just simple stack pops, and that most **free()** calls are stack pushes. This simple and speed-efficient average case is needed to amortize the costs of synchronization that is necessary to transfer blocks to and from the thread caches, when a stack is empty and has no blocks, or is full and has too many blocks.

Note that the stacks used by the thread caches will only store a limited number of blocks, and when a **free()** adds a block that exceeds that limit, a list of blocks will be transferred back to the heap. This is needed to avoid unbounded blowup, as otherwise a producer-consumer pattern could cause a thread cache to store an unlimited number of blocks that cannot be used by other threads, eventually leading to expected program termination

Listing 5.1 shows the **malloc()** (allocation) and **free()** (deallocation) high-level routines. These make up the interactions between the application and the memory allocator, and are in essence single-block transfers that occur between the application and the memory allocator.

```
1  void* malloc(size_t size) {
2    size_t scIdx = ComputeSizeClass(size);
3    if (scIdx == 0)                                        // large allocation
4      return AllocateLargeBlock(size);
5    Cache* cache = GetCache(scIdx);
6    if (CacheIsEmpty(cache))
7      CacheFill(scIdx, cache);
8    return CachePopBlock(cache);
9  }
10
11 void free(void* ptr) {
12   size_t scIdx = GetSizeClassFromPageMap(ptr);            // get metadata
13   if (scIdx == 0)                                        // large allocation
14     return DeallocateLargeBlock();
15   Cache* cache = GetCache(scIdx);
16   if (CacheIsFull(cache))
17     CacheFlush(scIdx, cache);
18   CachePushBlock(cache, ptr);
19 }
```

Listing 5.1: High-level allocation and deallocation routines

Memory allocation starts by computing the size class corresponding to the requested size. For large allocations, LRMalloc does not use the thread cache and instead creates a superblock with the appropriate size in number of pages through **mmap()**. When a large allocation is **free()**'d, the corresponding superblock is **munmap()** and thus returned to the operating system. Large allocations are identified with a size class equal to 0. If the allocation is not large, the thread cache is used and the stack corresponding to the size class is accessed and checked. In the common case, the cache will not be empty and thus a block will be fetched from the cache with a pop operation. Note that the common case does not require synchronization, and it consists essentially in popping a block from a stack, a simple and fast operation. In the uncommon case, the cache is empty and thus it must be filled with **CacheFill()**, which requires interaction with the heap and thus some synchronization.

Memory deallocation also starts by finding out the size class of the provided allocation. This is the step where the pagemap component becomes relevant, as it keeps metadata about all blocks. As before, blocks obtained from large allocations are handled differently, in that they are dealt directly by the operating system, and for blocks obtained from small allocations the corresponding cache is accessed. In the common case, the cache will not be full and thus the block will just be added to the cache. Once more, the common case does not require synchronization and consists of pushing a block to a stack, a simple and fast operation. In the uncommon case, the cache is full and thus **CacheFlush()** must be called to reduce the number of blocks in the cache, before the deallocated block can be added to the cache.

Caches are thread-specific objects and thus all operations using the cache need no synchronization. In the allocation and deallocation algorithms, only the **CacheFill()** and **CacheFlush()** routines require synchronization. Both routines are described in more detail next. For the sake of brevity, we omit the **GetCache()**, **CacheIsEmpty()**, **CacheIsFull()**, **CachePopBlock()** and **CachePushBlock()** subroutines, which are trivially implemented.

5.4 Heap

Due to its lock-free nature, the LRMalloc's heap component is by far the most complex component of the allocator. It is based on Michael's lock-free memory allocation algorithm [40] but adapted with a number of improvements to work with the presence of thread caches. The heap manages *superblocks* through *descriptors*. Superblocks are continuous sets of pages that are cut into equal-sized blocks that can be used by the application. Superblocks contain no metadata of their own. Instead, metadata is stored in descriptors.

5.4.1 Descriptors

Descriptors are unreclaimable but reusable objects used by the heap to track superblocks. Descriptors contain the superblock's metadata, such as, where the superblock begins, the size class of its blocks and the number of blocks it contains. It also includes an *anchor*, an inner structure that describes the superblock's state and is small enough to fit inside a single processor word (e.g., 64 bits) to be atomically updated with CAS instructions. Listing 5.2 presents the anchor and descriptor struct definitions. The **Anchor.avail** field refers to the first available block in the superblock. A free block then points to the next free block on the chain.

The exact number of bits used by the **Anchor.avail** and **Anchor.count** fields (31 in the current implementation) are implementation dependent and can be adjusted to how large superblocks can be and to how many blocks at most they can have. In our proposal, and unlike Michael's original algorithm, the anchor does not need a ABA prevention tag.

```
1  struct Anchor {
2    size_t state : 2;           // may be FULL = 0, PARTIAL = 1 or EMPTY = 2
3    size_t avail : 31;                           // index of first free block
4    size_t count : 31;                             // number of free blocks
5  };
6
7  struct Descriptor {
8    Anchor anchor;
9    char* superblock;                              // pointer to superblock
10   size_t blocksize;                   // size of each block in superblock
11   size_t maxcount;                                    // number of blocks
12   size_t scIdx;                       // size class of blocks in superblock
13 };
```

Listing 5.2: Anchor and descriptor structures

5.4.2 Superblocks

Superblocks are continuous sets of pages that are cut into equal-sized blocks. Individual blocks in a superblock are in one of 3 possible situations: (i) in use by the application; (ii) in use in a thread cache; (iii) in the superblock's free list. A superblock can be in 3 different states: (i) *full*, if every block in the superblock is being used by a thread cache or by the application, in which case the superblock is said to be *full* of used blocks; (ii) *partial*, if a superblock has some used blocks and some free blocks remaining; (iii) *empty*, if the superblock has only free blocks, in which case the superblock can just be returned back to the operating system.

Figure 5.3 shows an example of a partial superblock. The associated descriptor contains a pointer to the superblock, and the descriptor's anchor describes the superblock's state. The presented superblock contains a list of 3 free blocks, whereas all other blocks are in use either by the application or by a thread cache – neither descriptor nor superblock contain enough information to distinguish either case.

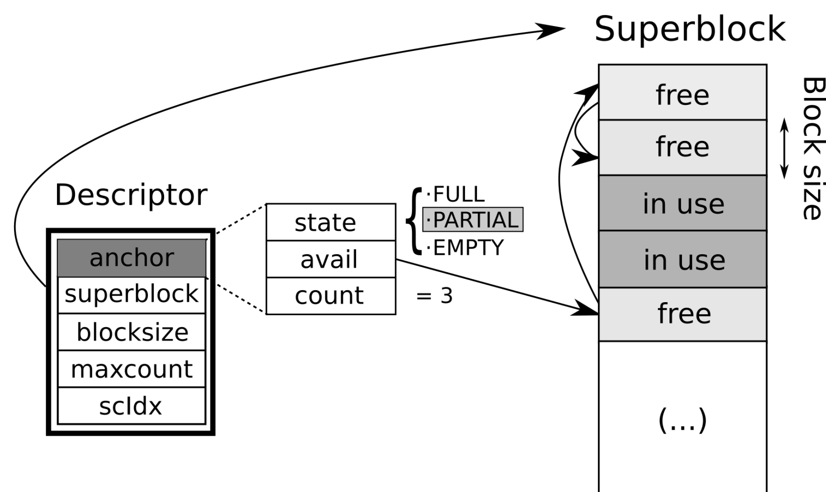


Figure 5.3: Example of a partial superblock

The superblock's states can be used to describe the superblock's lifetime and interaction with the rest of the heap, as highlighted in Fig. 5.4. Superblocks are constructed with memory obtained from the operating system, and are immediately used to fill a thread cache with all of its blocks, at which point the superblock becomes full. When full, a superblock may become partial if some blocks become free, or empty if all blocks become free. When partial, a superblock has some free blocks. If more blocks are added to the superblock's free list, the superblock either remains partial, if there are still used blocks, or empty otherwise. The superblock may transition from partial to full if used to fill a thread cache, in which case all blocks are removed from the free list. When empty, a superblock is returned to the operating system.

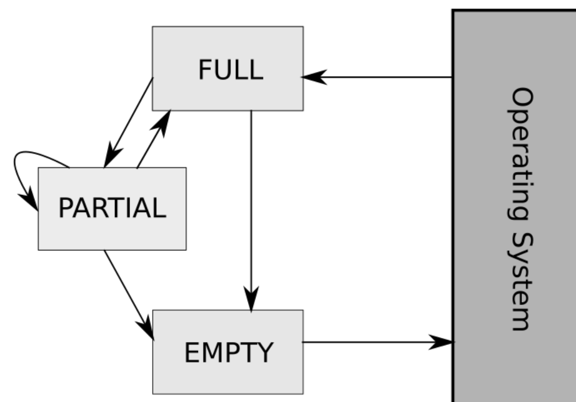


Figure 5.4: Superblock state machine

5.4.3 Filling and Flushing Caches

The heap interacts with the operating system to obtain memory with which it constructs superblocks and descriptors. The descriptors are used to manage superblocks. The superblocks contain lists of free blocks that can be transferred from and to thread caches, through the use of **CacheFill()** and **CacheFlush()**.

Cache Fill

Listing 5.3 describes the **CacheFill()** algorithm. By default, the algorithm tries to reuse a partially free superblock by calling **CacheFillFromPartialSB()**, which corresponds to start by trying to get a descriptor from a lock-free stack (**HeapGetPartialSB()** in line 10). The resulting descriptor cannot be immediately used and has to be checked, as it may refer to a superblock where all blocks have since then been freed, in which case the superblock has been returned to the operating system and is no longer usable. In this case, the descriptor can be safely reused and is put into a global recycle list by calling (**DescriptorRetire()** in line 17) and the algorithm is repeated. Otherwise, all available blocks in the superblock are reserved, with a CAS that updates the anchor (line 23), and then added to the thread's cache (lines 24–29).

Note that in the absence of contention, this single CAS along the other atomic operations used to remove the descriptor from the lock-free stack can transfer up to $N - 1$ blocks to the thread caches, where N is the number of blocks the superblock can contain. Furthermore, no ABA-prevention tag is needed on this CAS, because the only change that can happen to the underlying superblock is that more blocks become available, thus updating **Anchor.avail** and **Anchor.count**, which would fail the CAS. This is opposed to Michael's original algorithm, where blocks in the superblock can concurrently become used and free due to *active* superblocks. This is important for our bit math described earlier, as having a ABA counter can limit the maximum number of blocks that a superblock may have.

```
1  void CacheFill(size_t scIdx, Cache* cache) {
2    // try to fill cache from a single partial superblock ...
3    bool res = CacheFillFromPartialSB(scIdx, cache);
4    // ... and if that fails, create a new superblock
5    if (!res)
6      CacheFillFromNewSB(scIdx, cache);
7  }
8
9  bool CacheFillFromPartialSB(size_t scIdx, Cache* cache) {
10   Descriptor* desc = HeapGetPartialSB(scIdx);
11   if (!desc)                                   // no partial superblock available
12     return false;
13   Anchor newAnc, oldAnc;
14   do {
15     oldAnc = desc->anchor;
16     if (oldAnc.state == EMPTY) {
17       DescriptorRetire(desc);
18       return CacheFillFromPartialSB(scIdx, cache);                    // retry
19     }
20     newAnc.state = FULL;
21     newAnc.avail = desc->maxcount;
22     newAnc.count = 0;
23   } while (!CAS(&desc->anchor, oldAnc, newAnc));
24   char* block = desc->superblock + oldAnc.avail * desc->blockSize;
25   size_t blockCount = oldAnc.count;
26   while (blockCount-- > 0) {
27     CachePushBlock(cache, block);
28     block = *(char**)block;
29   }
30   return true;
31 }
32
33 void CacheFillFromNewSB(size_t scIdx, Cache* cache) {
34   Descriptor* desc = DescriptorAlloc();
35   DescriptorInit(desc, scIdx);                 // initialize with size class info
36   Anchor anc;
37   anc.state = FULL;
38   anc.avail = desc->maxcount;
39   anc.count = 0;
40   desc->anchor = anc;
41   desc->superblock = mmap(...);
42   for (size_t idx = 0; idx < desc->maxcount; ++idx) {
43     char* block = desc->superblock + idx * desc->blockSize;
44     CachePushBlock(cache, block);
45   }
46   PageMapRegisterDescriptor(desc);                           // update pagemap
47 }
```

Listing 5.3: CacheFill routine

If there are no available partial superblocks for the size class at hand, a new superblock must

be allocated and initialized. This is done in **CacheFillFromNewSB()**, which allocates a new superblock from the operating system and assigns a descriptor to it. The assigned descriptor is provided by **DescriptorAlloc()**, which accesses a global lock-free list of recycled descriptors. All blocks in the newly allocated superblock are then added to the thread's cache (lines 42–45 in Listing 5.3). Because all of the blocks in the superblocks are immediately given to a thread cache, a superblock that was just allocated from the operating system is full, and will only become partial/empty as blocks are flushed from thread caches.

Cache Flush

Flushing a cache is a less straightforward procedure. When a cache is full, several blocks must be returned to their respective superblocks, which requires updating the superblocks' associated descriptors. If blocks were returned to their respective superblocks one at a time, at least a CAS per block would be required, which is not only inefficient but would be a source of contention. Instead, it is best to group up blocks according to descriptor as best as possible, in order to be able to return several blocks back to the same superblock in a single CAS. Listing 5.4 shows LRMalloc's cache flushing algorithm.

CacheFlush() starts by popping a block from the cache and by forming an ad-hoc singly-linked list with all next blocks in cache that belong to the same superblock (lines 4 –16). Recall that blocks that belong to the same superblock share the same descriptor. The pagemap stores not only the size class, but also descriptors and thus it can be used to find the descriptor of the superblock that the block belongs to. We use **CachePeekBlock()** to inspect the first block in the cache without popping it. The ad-hoc list is then appended to the corresponding superblock's free block list and the anchor updated accordingly (lines 19–30). At this stage, the CAS only fails if blocks were added to the free block list due to other blocks are being simultaneously flushed from other caches, or if the free block list just became empty due to all free blocks removed for a cache fill. At the end, if these are the first blocks to be freed, the superblock is added to the lock-free stack of partial superblocks (line 33). Otherwise, if all blocks are made free, the pagemap is updated to reflect the change and the superblock returned to the operating system (lines 35–38). The pagemap must be updated before the superblock is returned to the operating system, because otherwise the pagemap update might affect a new superblock that was just allocated on the same memory.

While our cache flush algorithm is simple and has an obvious worst-case scenario (interleaved blocks from different superblocks will lead to a CAS operation being needed per each block), we find that in practice this algorithm works well and succeeds in transferring a large number of blocks with a single CAS from the thread cache to the heap. As described earlier in Ch. 2, real program behavior exhibit regularities exploitable by memory allocators. One regularity observed is that blocks that are allocated close in the request stream tend to have similar lifetimes, and thus they tend to be deallocated close in the deallocation request stream. This is especially true when those allocations are close or equal in size. In our case, as we provide blocks from the same

superblock (e.g., close in the request stream), we can expect those blocks to have similar expected lifetimes and thus to return to the thread cache at similar times, grouped together. As such, interleaved blocks from different superblocks becomes an unlikely and uncommon occurrence.

```
1  void CacheFlush(size_t scIdx, Cache* cache) {
2    while (!CacheIsEmpty(cache)) {
3      // form a list of blocks to return to a common superblock
4      char* head, tail;
5      head = tail = CachePopBlock(cache);
6      Descriptor* desc = PageMapGetDescriptor(head);
7      size_t blockCount = 1;
8      while (!CacheIsEmpty(cache)) {
9        char* block = CachePeekBlock(cache);
10       if (PageMapGetDescriptor(block) != desc)
11         break;
12       CachePopBlock(cache);
13       ++blockCount;
14       *(char**)tail = block;
15       tail = block;
16     }
17
18     // add list to descriptor and update anchor
19     char* superblock = desc->superblock;
20     size_t idx = ComputeIdx(superblock, head);
21     Anchor oldAnc, newAnc;
22     do {
23       newAnc = oldAnc = desc->anchor;
24       *(char**)tail = superblock + oldAnc.avail * desc->blockSize;
25       newAnc.state = PARTIAL;
26       newAnc.avail = idx;
27       newAnc.count += blockCount;
28       if (newAnc.count == desc->maxcount)                // can free superblock
29         newAnc.state = EMPTY;
30     } while (!CAS(&desc, oldAnc, newAnc));
31
32     if (oldAnc.state == FULL)
33       HeapPutPartialSB(desc);
34     else if (newAnc.state == EMPTY) {
35       // unregister metadata from pagemap ...
36       PageMapUnregisterDescriptor(superblock, scIdx);
37       // ... and release superblock back to OS
38       munmap(superblock, ...);
39     }
40   }
41 }
```

Listing 5.4: CacheFlush routine

When a cache flush completely empties a superblock (line 34), note that the corresponding descriptor cannot be returned to the operating system since other threads can be potentially holding a reference to it. Moreover, it cannot be recycled also as there may still be a reference to it in the list of partial superblocks, and guaranteeing its correct removal is a non-trivial task. In our approach, a descriptor is only recycled when a thread removes it from the list of partial superblocks (line 17 in Listing 5.3).

5.5 Pagemap

The pagemap component stores *allocation metadata per page* based on the observation that most allocations are much smaller than a page, and that blocks in the same page are in the same superblock, thus sharing the same size class and descriptor. Storing metadata per page instead of per allocation has several advantages. First, it is more memory efficient, as equal metadata is not kept for every distinct block in a page. Second, it helps separating memory allocator memory from application memory, which ultimately improves application data locality, as memory allocator and application memory are no longer interleaved.

The pagemap can be implemented in a number of different ways. With an operating system that allows memory overcommitting, it can be a simple array, where the size depends on the size of the valid address space and how much memory is needed for each page's metadata. For example, assuming that a single processor word is enough for metadata, and that the address space can only have 2^{48} bytes (common for 64 bit architectures), then this array requires 2^{48-12} words, or about 512GB. Of course, the actual physical memory is bounded by the number of pages required for user applications. LRMalloc uses this type of implementation. A cheaper solution in terms of virtual memory would be to use a lock-free radix tree, at the cost of some performance and additional complexity.

Unlike the heap, the pagemap is a mostly read-only component, only used to track block size and corresponding superblock. It is kept up to date by the heap, when a new superblock is created in **CacheFillFromNewSB()** or a superblock becomes empty in **CacheFlush()** and is thus given back to the operating system. Between these two points, data on the pagemap is read-only, and used whenever a block metadata lookup is required.

5.6 Interaction with the Operating System

Dynamic memory allocators operate in user-space and have access to the operating system. A user-space memory allocator cannot obtain memory to manage and satisfy requests from anywhere other than the operating system, so interaction with the operating system is mandatory. System calls are notoriously slow, more than synchronization primitives, so there is value in discussing how often we need to interact with the operating system.

The heap obtains memory from the operating system to construct superblocks that are needed to carve out lists of blocks. Memory used for superblocks is returned to the operating system once all the blocks in a superblock are free. The frequency of requests to the operating system is going to depend directly on the size of superblocks. Larger superblocks will satisfy more memory allocation requests, and allow larger caches to be filled. In our implementation, LRMalloc uses 2MB superblocks for all size classes, an ad-hoc value large enough to reduce operating system contention but small enough to avoid excessive memory usage. Memory is obtained and returned to the operating system through the use of **mmap()** and **munmap()**.

One interesting thing to note is that LRMalloc cannot operate without the presence of an operating system, and the same is true with all the other lock-free memory allocators described before [17, 40]. They all require the operating system to obtain and release superblocks or the equivalent to superblocks, and the operating system is effectively a synchronization layer that allows memory to be reused by different threads. In all these cases, the operating system could be replaced by another lower-level allocator that is capable of directly managing memory.

In Ch. 2, we have described early memory allocation schemes that are capable of managing memory without an operating system by being able to coalesce and split blocks of memory. A lock-free lower-level mechanism that is able of coalesce and split blocks could be used to reduce the amount of requests done to the operating system.

5.7 Chapter Summary

In this chapter, we have presented the components, data structures and techniques used by LRMalloc, a lock-free memory allocator that aims to be competitive with current state-of-the-art memory allocators.

The following chapter will present our experimental analysis of LRMalloc, comparing it against other memory allocators using standard benchmarks.

Chapter 6

Experimental Analysis

In this chapter, we present an experimental analysis of our work. We compare LRMalloc against other modern and relevant memory allocators using a variety of well-established benchmarks that have been used to evaluate prior work.

Our experimental analysis focuses on memory allocator latency and scalability, rather than memory usage and fragmentation. We highlight a few reasons for this. First, for multithreaded applications, performance and scalability has a higher value than memory usage. All modern concurrent memory allocators use techniques that explicitly trade-off better performance and scalability at the expense of memory usage, such as arenas and thread caches. Second, there are plenty of standard benchmarks for measuring performance and scalability, but none for measuring memory usage as far as we know. Methodologies used by authors to measure memory usage are not always clear. Often, authors do not state if they are measuring physical memory that is being used by an application, or are otherwise accounting for memory that has been reserved but is not actually yet in use (e.g., uncommitted pages). Very little is also said about how the memory usage data is collected, so it is not clear if memory usage was collected by periodically polling the operating system for process statistics, or if applications themselves were modified to collect memory usage at precise moments.

Of course, that is not enough to argue that memory usage is irrelevant for concurrent memory allocators. At least, they must have bounded blowup, or otherwise can cause unexpected application termination. In any case, the development of an effective methodology to measure memory usage and fragmentation, the selection of sample applications to be used as benchmarks, and the thorough analysis of memory usage for existing concurrent memory allocators a very is demanding task, one that we have not done in this document.

6.1 Methodology

To measure memory allocator performance and scalability, we used standard benchmarks commonly used in the literature [4, 40], namely the *Linux scalability* [35], *Threadtest* and

Larson [30] benchmarks. To measure active false sharing and passive sharing, we use the *cache-thrash* and *cache-scratch* benchmarks [4]. All of these benchmark applications take in a number of threads as an argument along some other parameters, and report execution time or throughput at completion. Data was collected by running these benchmark applications using different memory allocators with a varying number of threads. To run a benchmark with a specific memory allocator, we use the *LD_PRELOAD* mechanism to dynamically load the library corresponding to the desired memory allocator at execution time. None of the benchmarks are statically linked with any memory allocator at compilation.

The environment for our experiments was a dedicated x86-64 multiprocessor system with four AMD SixCore Opteron TM 8425 HE @ 2.1 GHz (24 cores in total) and 128 GBytes of main memory, running Ubuntu 16.04 with kernel 4.4.0-104 64 bits.

Benchmarks

Linux scalability is a benchmark used to measure memory allocator latency and scalability. It launches a given number of independent threads, each of which runs a batch of 10 million **malloc()** requests allocating 16-byte blocks followed by a batch of identical **free()** requests. Blocks are deallocated in the same order that they were allocated, and they are not touched after being allocated. This benchmark measures memory allocator throughput in a best case scenario, with each thread behaving independently and without remote frees. Because this benchmark does not actually touch the blocks it allocates, memory allocators which do not touch blocks may have an advantage, as fewer physical pages are needed, thus reducing the amount of work done by the operating system.

Threadtest is similar to *Linux scalability* with a slightly different allocation profile. It also launches a given number of independent threads, each of which runs batches of 100 thousand **malloc()** requests followed by 100 thousand **free()** requests. Each thread runs 100 batches in total. Blocks in this benchmark are deallocated in the same order that they were allocated. This benchmark measures memory allocator throughput in a scenario with alternating allocations and deallocations.

Larson simulates the behavior of a long-running server process. It repeatedly creates threads that work on a slice of a shared array. Each thread runs a batch of 10 million **malloc()** requests with sizes uniformly distributed between 16 and 128 bytes and the resulting allocations are stored in random slots in the corresponding slice of the shared array (each thread's slice includes 1000 slots). When a slot is occupied, a **free()** request is first done to release it. At any given time, there is a maximum limit of threads running simultaneously. Only one thread has access to a given slice of the shared array at any given time, and slices are recycled as threads are created and destroyed. To summarize, this benchmark does memory allocations with varying sizes, memory deallocations that are done in random order, and allocations interleaved with deallocations. There are some remote frees, as slices are recycled and a new thread can call **free()** on blocks that were allocated by another thread. Like *Linux scalability* and *Threadtest*,

this benchmark does not touch allocated blocks.

Cache-thrash measures whether the memory allocator induces active false sharing. It creates a number of independent threads, each of which allocates a 16 byte block, followed by 100 thousand writes to that block, followed by a deallocation. Each thread does a total of 600 memory allocations. By having all threads allocating blocks smaller than a cache line, this benchmark tests whether the allocator induces active false sharing.

Cache-scratch measures whether the memory allocator induces passive false sharing. It starts by making several 16 byte block allocations in the main thread, which are to be passed to the other threads. It then creates a number of independent threads, each of which deallocates a 16 byte block received from the main thread. Next, each thread allocates a 16 byte block, followed by 100 thousand writes to that block, followed by a deallocation. Each thread does a total of 600 memory allocations and deallocations. This benchmark tests passive false sharing by performing a remote free on each thread and by checking whether that thread re-uses that block in a later allocation.

Memory Allocators

We compared LRMalloc against two groups of memory allocators: (i) previous lock-free memory allocators, namely Michael's allocator [40] and NBMalloc [17]; (ii) lock-based state-of-the-art memory allocators, namely Hoard [4], Ptmalloc2 [19], Jemalloc-5.0 [13] and TCMalloc [16]. Source code for all the memory allocators used is publicly available. Since some of these memory allocators are ongoing work, when possible, we state version and git commit:

- Michael's allocator is available at https://github.com/ricleite/lrmichael. This is our own implementation of Michael's allocator, significantly more efficient than the existing implementation at https://github.com/scotts/michael. We used the version at commit 184650bfacd0f80f8dc06a1aed258fb87182e4e4.
- NBMalloc is available at http://www.cse.chalmers.se/research/group/dcs/nbmalloc.html. We used version 0.6.0, the latest available at the time of writing.
- Hoard is available at http://hoard.org. We used version 3.12.0, specifically commit a0e46aa1b7719c9a84a5fc66beb7286811c4aedc.
- Ptmalloc2 is available at http://www.malloc.de/en. We used the version distributed with glibc in Ubuntu 16.04, which is the default memory allocator in the system.
- Jemalloc is available at http://jemalloc.net. We used version 5.0, specifically commit c834912aa9503d470c3dae2b2b7840607f0d6e34.
- TCMalloc is available at https://github.com/gperftools/gperftools. We used version 2.7, commit 9608fa3bcf8020d35f59fbf70cd3cbe4b015b972.

- Lastly, our memory allocator, LRMalloc is available at https://github.com/ricleite/lrmalloc. The version used in this document is at commit bec52c8cfce6284255735a21faf0d58e495622b7.

6.2 Performance & Scalability

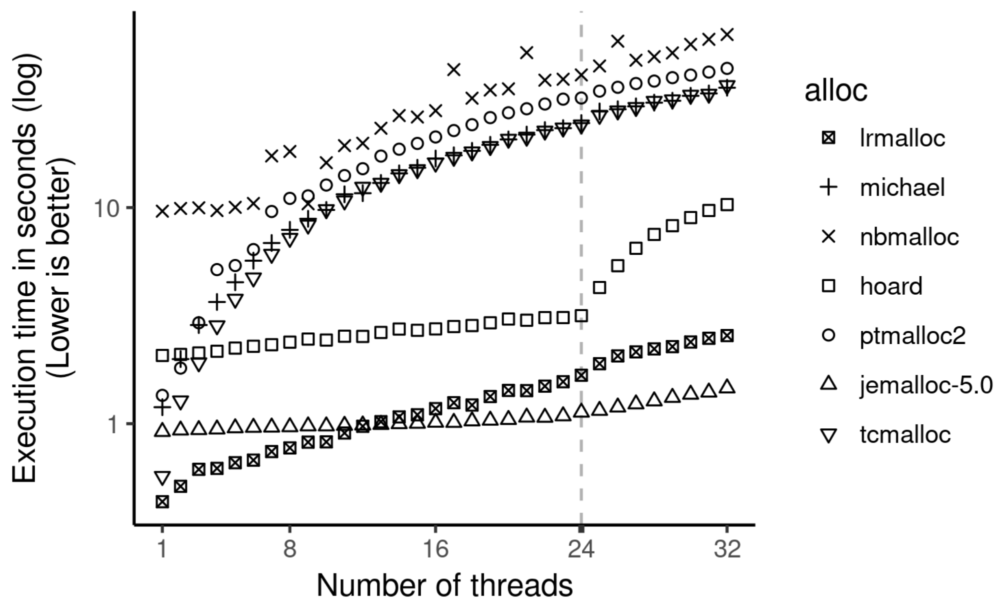
Figures 6.1 and 6.2 present experimental results for the *Linux scalability*, *Threadtest* and *Larson* benchmarks when using the different memory allocators with configurations from 1 to 32 threads. The results presented are the average of 5 runs.

Figure 6.1(a) shows the execution time, in seconds (log scale), for running the *Linux scalability* benchmark. In general, the results show that LRMalloc is very competitive, being only overtaken by Jemalloc as the number of threads increases. Jemalloc's behavior can be explained by the fact that it creates a number of arenas equal to four times the number of cores in the system. Multiple threads may be mapped to the same arena but, with so many arenas, collisions are unlikely and thus little contention happens on shared data structures. On the other hand, LRMalloc's heap has no arena-like multiplexing, and thus there is a greater potential for contention as the number of threads increases. In particular, shared data structures such as the descriptor recycle list and the lists containing partial descriptors become bottlenecks are the number of threads increases.

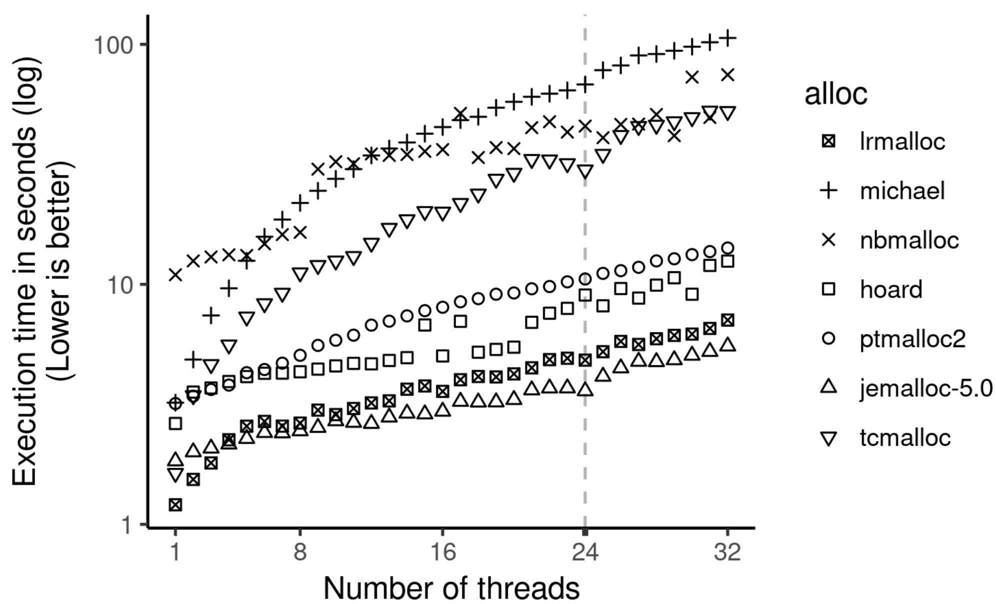
Figure 6.1(b) shows the execution time, in seconds (log scale), for running the *Threadtest* benchmark. Again, the results show that LRMalloc is very competitive and only surpassed by Jemalloc by a small margin. In particular, for a small number of threads, LRMalloc is slightly faster than Jemalloc and, as the number of threads increases, LRMalloc keeps an identical tendency as Jemalloc. Note that in both these benchmarks, the other lock-free memory allocators, Michael's and NBMalloc are an order-of-magnitude worse than LRMalloc.

Figure 6.2 shows the number of operations per second (log scale), for running the *Larson* benchmark. Unlike prior lock-free memory allocators, LRMalloc consistently achieves increased throughput as the number of threads approaches the number of cores available, while carrying out over a order-of-magnitude more throughput. LRMalloc also performs similarly to all other state-of-the-art allocators, with a small performance degradation as the number of threads increases. Presumably, this degradation has a couple of factors. First, the cache flushing mechanism is triggered every-time a thread exits, which lowers memory fragmentation but, for the kind of programs which create a huge number of threads during its lifetime (as is the case with this benchmark), it can incur in some extra overhead. This can be improved by recycling cache structures when a thread exits, in order to allow them to be reused by new spawning threads. Second, the *Larson* benchmark does deallocations in a random order. This may cause interweaving of blocks that belong to different superblocks in the thread caches, thus leading to some additional contention on cache flush due to the nature of our cache flush algorithm.

In general, the three benchmarks show that LRMalloc is in toe with the other state-of-the-art concurrent memory allocators, and far superior to prior lock-free memory allocators. We highlight however that LRMalloc has some flaws, namely in the lack of multiplexing that is done



(a) Linux scalability



(b) Threadtest

Figure 6.1: Execution time results in log scale comparing LRMalloc, Michael's allocator, NBMalloc, Hoard, Ptmalloc2, Jemalloc-5.0 and TCMalloc for the *Linux scalability* and *Threadtest* benchmarks with configurations from 1 to 32 threads

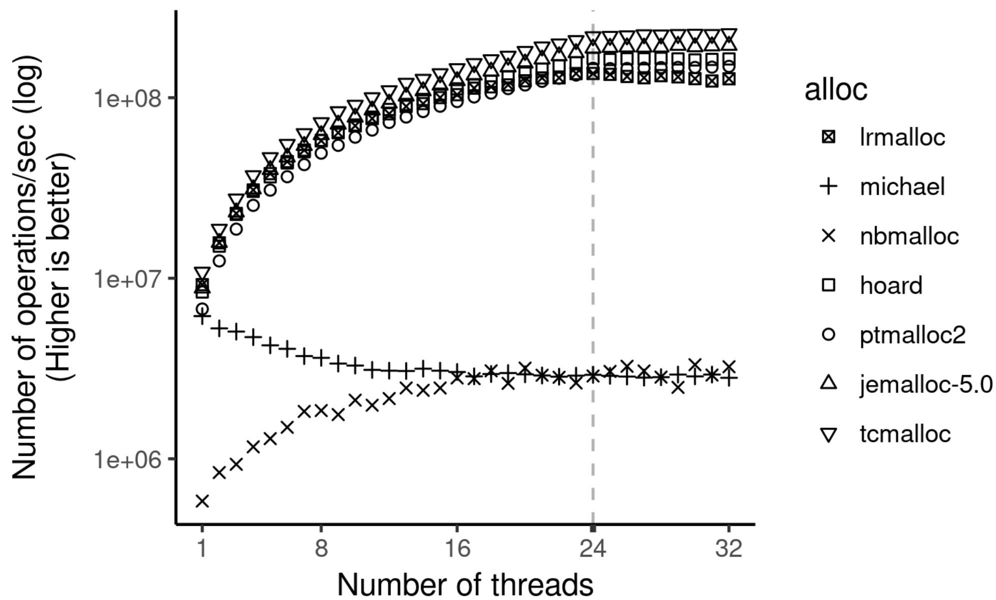


Figure 6.2: Throughput results in log scale comparing LRMalloc, Michael's allocator, NBMalloc, Hoard, Ptmalloc2, Jemalloc-5.0 and TCMalloc for the *Larson* benchmark with configurations from 1 to 32 threads

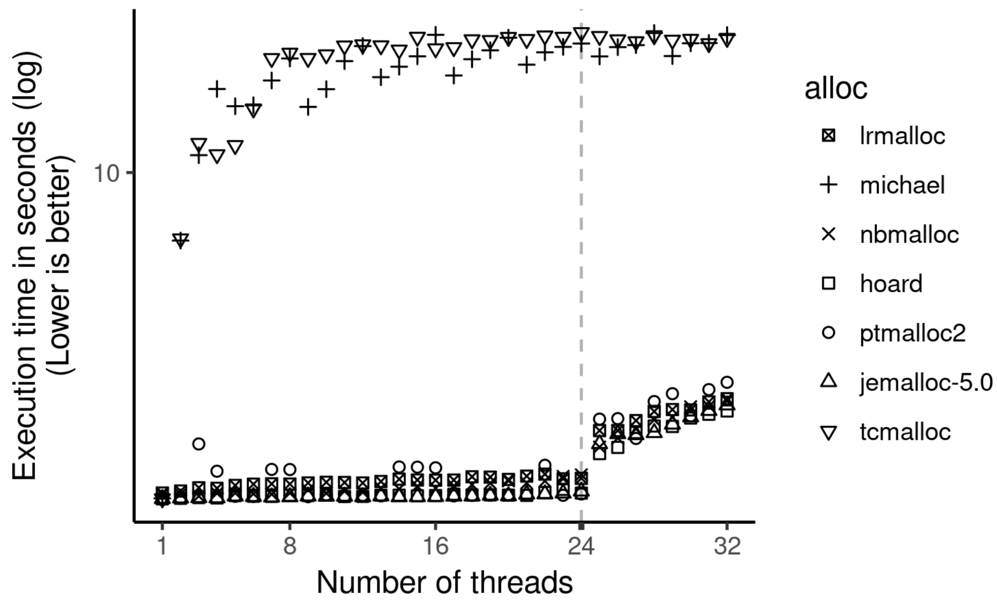
with shared data structures, namely the descriptor recycle list and the lists containing partial descriptors. Multiplexing these structures will lead to lower overall contention, thus improving LRMalloc's scalability.

Another interesting observation in the *Linux scalability* and *Threadtest* benchmarks is that TCMalloc performs quite badly. Since we have tout TCMalloc as a modern concurrent memory allocator used in practice, we investigated further what caused this behavior. We have found that the version of TCMalloc used does an overwhelming amount of calls to the operating system, asking only a few pages of memory at a time, which causes a significant amount of contention at the operating system level, thus explaining its low performance.

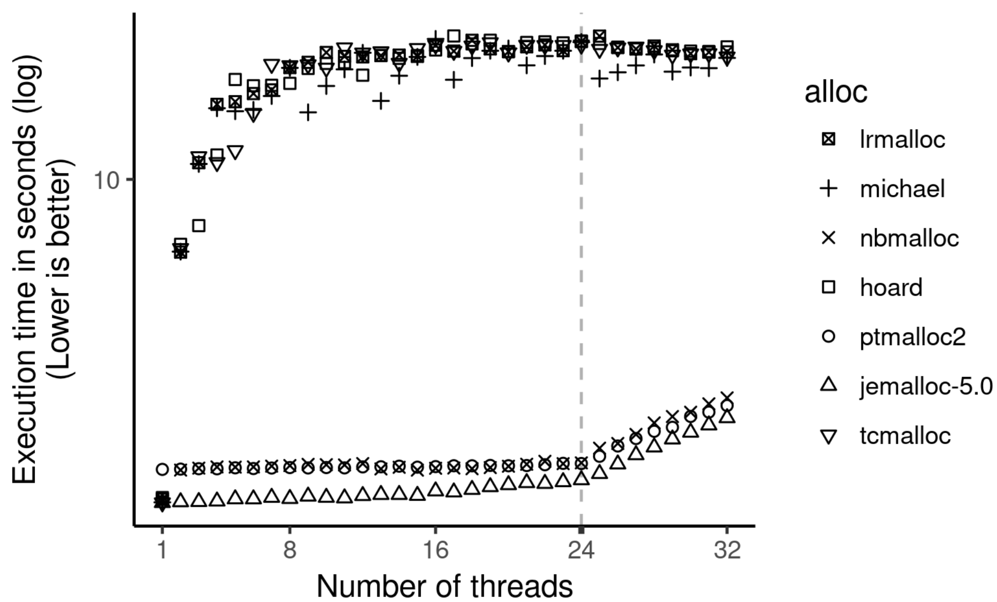
6.3 Active & Passive False Sharing

Figure 6.3 presents experimental results for the *cache-thrash* and *cache-scratch* benchmarks when using the different memory allocators with configurations from 1 to 32 threads. Again, the results presented are the average of 5 runs.

Figure 6.3(a) shows the execution time, in seconds (log scale), for running the *cache-thrash* benchmark. We remind that this benchmark attempts to detect whether the memory allocator induces active false sharing. Results clear show that Michael's allocator and TCMalloc suffer from performance degradation for more than 1 thread, and thus we can conclude that they exhibit active false sharing. All the remaining allocators, including LRMalloc, behave normally and seem to not induce active false sharing.



(a) Cache-thrash



(b) Cache-scratch

Figure 6.3: Execution time results in log scale comparing LRMalloc, Michael's allocator, NBMalloc, Hoard, Ptmalloc2, Jemalloc-5.0 and TCMalloc for the *cache-scratch* and *cache-thrash* benchmarks with configurations from 1 to 32 threads

Figure 6.3(b) shows the execution time, in seconds (log scale), for running the *cache-scratch* benchmark, which attempts to measure whether the memory allocator induces passive false sharing. Results show that a number of existing allocators suffer from performance degradation and thus induce passive false sharing. These include LRMalloc, Michael's allocator, Hoard and TCMalloc. Three memory allocators do not seem to suffer from performance degradation, Jemalloc, Ptmalloc2 and NBMalloc. We found these results particularly surprising for Jemalloc, since it uses thread caches, but must somehow invalidate them or track individual blocks that were obtained from other threads.

6.4 Chapter Summary

In this chapter, we have presented an experimental analysis of LRMalloc, a lock-free memory allocator that we have developed. We have compared LRMalloc against other concurrent memory allocators using established benchmarks, and found it to be competitive against state-of-the-art memory allocators, and clearly superior to prior lock-free memory allocators.

Chapter 7

A Lock-Free Coalescing Mechanism

In this chapter, we present a lock-free address-ordered best-fit mechanism. Our mechanism supports lock-free coalescing and splitting of blocks with arbitrary sizes, unlike existing lock-free buddy systems, and is equivalent to the address-ordered best-fit mechanism presented in Ch. 2, which has very desirable low fragmentation characteristics.

7.1 Motivation

Existing lock-free memory allocators assume and require the presence of the operating system to perform memory management. This is true for prior lock-free allocators, as the ones presented in Ch. 4, but also for LRMalloc, the lock-free memory allocator we have presented in Ch. 5. All the lock-free memory allocators discussed so far require a lower-level allocator that is capable of providing regions that contain an arbitrary number of continuous pages, so that those regions may be used to construct superblocks or other similar objects. In order to be effective, this lower-level allocator has to be capable of coalescing and splitting to fulfill requests with low fragmentation, either through the use of a mechanism, as discussed in Ch. 2, or through the use of virtual memory, as is the case in the operating system. At the moment, the operating system fills the role of the lower-level allocator we have described, usually through the use of the **mmap()** and **munmap()** system calls. As such, existing lock-free memory allocators are fundamentally incapable of operating in a environment where no operating system exists, or where the operating system only provides a single region of memory that has to be managed.

In practice, memory allocators can assume the existence of an operating system, and as such the fact that the lock-free memory have no mechanism to split and coalesce sets of pages is not problematic. However, interaction with the operating system is slow, and a lock-free lower-level allocator capable of providing arbitrarily-sized blocks is worth pursuing if it allows avoiding some of that cost.

To summarize, we argue that a lock-free mechanism that is capable of managing a region of memory by coalescing/splitting blocks has value in the context of user-space lock-free memory

allocators, and academic interest for usage in embedded systems and in environments where a single region of memory has to be managed.

7.2 Related Work

There is little research regarding lock-free coalescing mechanisms. There are some lock-free buddy systems, one used by the Linux Kernel to manage pages [37], and a lock-free binary system in a recent work [36]. However, buddy systems are known to cause high fragmentation in practice [24]. This occurs in part due to: (i) the use of size classes, that leads to internal fragmentation; and (ii) the hierarchical structure that buddy systems impose and that restricts how coalescing can be performed.

To the best of our knowledge, there are no lock-free versions of *general coalescing mechanisms* – mechanisms that do not restrict the way in which coalescing can performed nor cause internal fragmentation due to the use of size classes, such as various sequential fits and segregated fits mechanisms that have been described in Ch. 2.

7.3 Coalescing with Boundary Tags

Consider that we have a block that has been deallocated and we want to coalesce it with its neighbors in a lock-free manner. To perform coalescing, the block needs to provide information allowing to locate its neighbors and their state. For this, let us assume we are using boundary tags [28], such that all blocks have a header and a footer that can be used to find neighbors and navigate, as Fig 7.1 shows. The header contains information about the block, namely its size and its state (whether it is currently allocated or free). The footer contains a back pointer to the header of the same block that it belongs to.

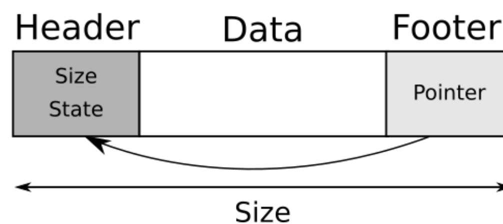
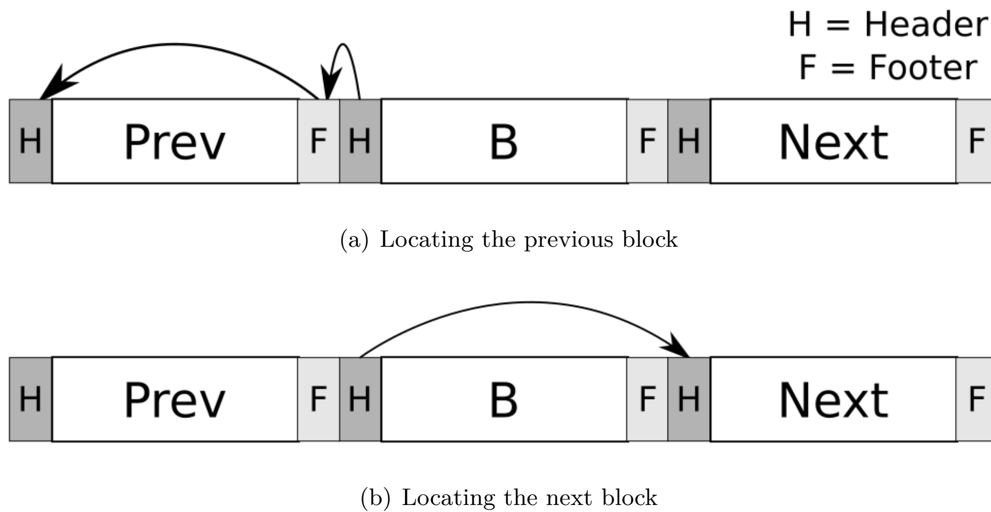


Figure 7.1: A block with the information necessary to perform coalescing

Every block has two neighbors that it might be able to coalesce with, the one that precedes it in memory, the *previous* block, and the one that follows it in memory, the *next* block. As Fig. 7.2 shows, given a block B, we can locate the previous block by reading the memory that immediately precedes B's header, which corresponds to the previous block's footer. We can then follow the footer's back pointer, which gives us the previous block's header, where state

information is stored. Similarly, we can locate the next block by using B's size to locate B's footer, and then read the memory that follows it, which corresponds to the next block's header.



(a) Locating the previous block

(b) Locating the next block

Figure 7.2: Locating previous and next blocks for coalescing

Once we have located a block, we can begin to attempt to coalesce. Coalescing is the merging of two continuous blocks to form a single (larger) block. With boundary tags, that corresponds to constructing a new block with a header and a footer. We name the process of a block coalescing with the previous and the next block, as *backward coalescing* and *forward coalescing*, respectively. In the case of backward coalescing, as shown in Fig. 7.3, with a block B being coalesced with a previous block P to form a new coalesced block C, P's header becomes C's header, B's footer becomes C's footer, and P's footer along B's header become irrelevant. Analogously, in forward coalescing, where a block B coalesces with a next block N to form a new coalesced block C, B's header becomes C's header, N's footer becomes C's footer, and B's footer along N's header become irrelevant. In both backward and forward coalescing, C's header and footer have to be updated and adjusted to C's size, and thus they store different information than the original blocks.

Performing coalescing in a sequential environment with the boundary tag mechanism presented above is trivial. It is similarly trivial in a multithreaded environment, if a lock is used to serialize coalescing operations. Current state-of-the-art lock-based memory allocators mentioned in Ch. 4 perform coalescing of blocks in a given region by employing a single lock. Concurrent coalescing can occur through the use of multiple regions.

Compared to lock-based alternatives, lock-free coalescing presents a number of challenges. First, we cannot guarantee that no other threads changes the state of neighbor blocks, that we are trying to coalesce with, at some given point. Second, using boundary tags, we have two pieces of information that ideally would have to be kept coherent – the header and the footer of each block. However, as seen previously in the coalescing procedure, the physical location of the header and the footer is important, so we cannot use a single processor word to store both, and thus we cannot use a single CAS to update both the header and the footer of a block

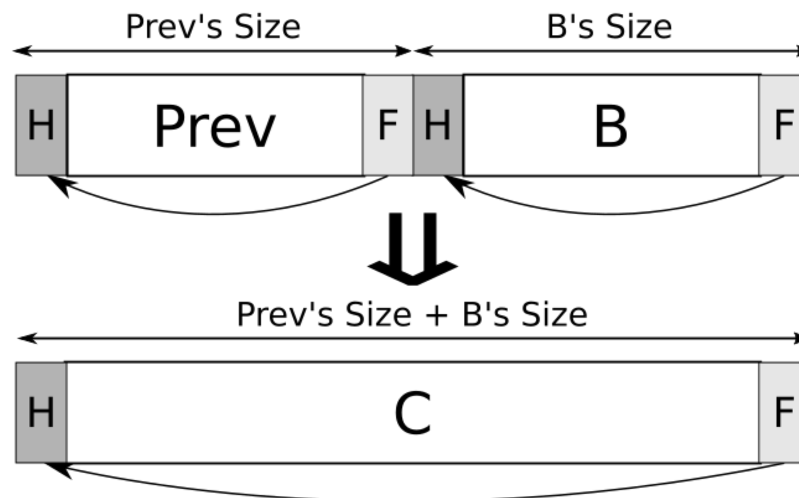
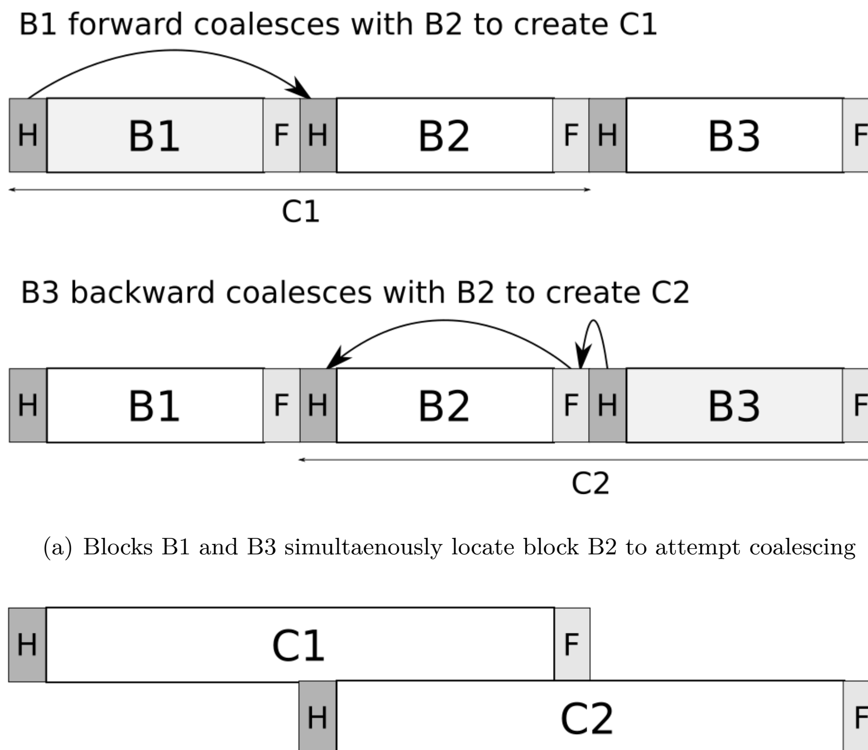


Figure 7.3: Overview of backward coalescing

to keep them coherent. Lastly, the region of memory we are managing is finite, thus not all blocks have neighbors in both directions. Furthermore, such region might mutate over time, increasing and decreasing in size. When we read the memory that precedes a block, with the intention of reading the previous block's footer, we might be reading invalid memory that we are not managing. In a sequential or a serialized multithreaded execution it is trivial to check the current dimension of a region to assert whether we can read memory that precedes a block. However, in a lock-free coalescing mechanism, nothing ensures that the dimensions of a region do not change between the moment at which we read those dimensions, and the moment at which we read the memory that precedes a block, thus potentially reading invalid memory.

Consider the following example in Fig. 7.4 to better understand the difficulty of lock-free coalescing. In a region with three adjacent blocks, B1, B2 and B3, blocks B1 and B3 are deallocated simultaneously and attempt to coalesce with B2, which sits between B1 and B3 and is not allocated, thus available for coalescing. Suppose that this coalescing mechanism uses boundary tags, and that the header and footer can be updated with a single atomic operation. Coalescing is done by locating the header of the neighbor block, checking whether the neighbor is free for coalescing, and then by atomically updating the header, then the footer, for the new coalesced block. As seen in Fig. 7.4(a), both B1 and B3 can reach B2's header to attempt coalescing. However, because constructing a new coalesced block is a two-step process, B1 can locate B2, begin coalescing, update the header for the new coalesced block C1 (which is B1's header), and then B3 can locate B2 (through B2's footer) and successfully coalesce and create block C2 before B1 has a chance to update B2's footer. The result of this simultaneous coalescing are two incorrect overlapping blocks C1 and C2, as shown in Fig. 7.4(b).

Note that so far, we have only discussed how coalescing is done once we have a block that has been deallocated and that we wish to coalesce with its neighbors. We have not discussed how free blocks are tracked nor how blocks are allocated in the first place. The use of additional



(a) Blocks B1 and B3 simultaenously locate block B2 to attempt coalescing

(b) Blocks B1 and B3 can both succeed with a specific order of operations, creating blocks C1 and C2, which overlap

Figure 7.4: Simultaneous coalescing of two blocks with a shared neighbor

data structures needed to find and allocate blocks naturally complicate the goal of obtaining a lock-free coalescing mechanism.

7.4 Proposal

Our proposal for a lock-free best-fit mechanisms consists of two logical data structures: (i) the *map*, which represents the physical layout in memory and is needed in order to perform coalescing, and (ii) the *tree*, a ordered set-like data structure that will be used to obtain blocks in a best-fit fashion and that is needed for block allocation.

In our proposal, all blocks include a header and footer, to be managed by the map. The header contains the block's size and the footer contains a pointer to the block. Neither header nor the footer contain information on whether the block is allocated or free. Instead, the tree is the authority used to assert whether a block is allocated or free. The tree is simultaneously used to satisfy allocations, but also to determine whether coalescing can be done. Before using a block for coalescing or to fulfill an allocation, the tree is used to ensure that a single thread obtains the block's ownership, Indeed, the main insight of our proposal, is that the data structure used to find free blocks must also be used to coordinate coalescing, since otherwise if state is kept elsewhere (i.e., in the header or footer or other auxiliary data structures) the mechanism to

determine whether a block is free for coalescing inevitably becomes unable to keep both the data structure and the referred state coherent with each other as coalescing occurs.

The Map

The *map* represents the physical layout in memory, and allows a block to find its neighbors in order to perform coalescing. Each block logically has a footer and a header that can be accessed by neighboring blocks. However, to allow the mechanism to manage varying amounts of memory and to avoid invalid accesses to memory as discussed earlier, a block does not physically contain its header and its footer. In essence, header and footer become akin to *descriptors*, and are objects whose memory cannot be reclaimed, only recycled and reused. The operations that the map needs to support are shown in Listing 7.1.

```
1  struct Header {
2    size_t size;
3  };
4
5  struct Footer {
6    void* ptr;
7  };
8
9  // used as an auxiliary structure that describes a block
10 struct Block {
11   size_t size;
12   void* ptr;
13 };
14
15 // update header for provided block
16 void UpdateBlockHeader(Block b, Header h);
17
18 // update footer for provided block
19 void UpdateBlockFooter(Block b, Footer f);
20
21 // get block header corresponding to provided address
22 Header GetBlockHeader(void* ptr);
23
24 // get block footer corresponding to provided address
25 Footer GetBlockFooter(void* ptr);
```

Listing 7.1: Map operations and definitions

Both the header and footer are small enough to fit inside a single processor word, and thus they can be atomically updated through CAS or other atomic operations. Therefore, both **UpdateBlockHeader()** and **UpdateBlockFooter()** are equivalent to an atomic write operation, while **GetBlockHeader()** and **GetBlockFooter()** are equivalent to atomic reads.

The map can be implemented using an array that uses uncommitted memory (similar to LRMalloc's pagemap), or using a lock-free radix tree. It needs to be able to store two processor words (header and footer) per page (if the smallest block being managed is a page), so that we can have one page blocks.

The Tree

The *tree* is an indexing data structure that contains free blocks. In order to implement address-ordered best-fit, it sorts blocks by size, and blocks with the same size by address. The tree has to support the operations shown in Listing 7.2.

```
1 // insert a block in the tree
2 void InsertBlock(Block b);
3
4 // remove a block from the tree
5 bool RemoveBlock(Block b);
6
7 // remove the least-ordered block from the tree that satisfies requested size
8 Block RemoveBlockWithAtLeast(size_t size);
```

Listing 7.2: Tree component operations

The tree is implemented with a lock-free indexed data structure. Because it relies on a lock-free data structure, we felt the need to describe each operation, its expected behavior and requirements, without which our coalescing scheme will not work as intended:

- **InsertBlock()** – adds a block to the tree, that must be available for removal when the function returns. This function can assume that the block does not exist in the tree.
- **RemoveBlock()** – if the provided block is in the tree, it removes it and returns true; otherwise returns false. The removal must be complete when the function returns, such that the same block can be added by **InsertBlock()**.
- **RemoveBlockWithSize()** – removes and returns the least-ordered block from the tree that satisfies the requested size. This function has the same removal requirements as **RemoveBlock()**. If no block has been found, it returns an invalid block starting at address 0 and with 0 size.

7.4.1 High-level Allocation and Deallocation

Listing 7.3 shows the high-level allocation and deallocation routines. Allocation starts by removing a block with sufficient size from the tree with **RemoveBlockWithAtLeast()** in line 2. If no adequate block is found, more memory has to be obtained from the operating system and a block constructed with it. Otherwise, if a block is found, it can be used to fulfill the memory request. To eliminate internal fragmentation, the block is split if it is larger than what has been requested. Splitting (shown in lines 10–18) is simple, and consists of updating header and footer for the original and remaining block, and then adding the remaining block to the tree, so that it may be used to fulfill further memory allocation requests or for coalescing.

Deallocation starts by finding out the size of the provided block. Since the header stores the block's size, the map can be used for this purpose. Then, we can attempt to coalesce this

block with both neighbors (lines 29–30). Coalescing is further detailed in **CoalesceBackward()** and **CoalesceForward()**, explained next. After we have attempted to coalesce, we can add the resulting block to the tree (line 33).

```
1  void* Alloc(size_t size) {
2    Block b = RemoveBlockWithAtLeast(size);
3    if (b.ptr == NULL) {                                     // no block has been found
4      ...                             // obtain more memory from the operating system
5    }
6
7    if (b.size > size) {                                  // block is too large, split
8      // block has been removed from the tree
9      // so that it is owned by us and can be manipulated safely
10     Block s = { b.ptr, size };
11     SetBlockHeader(s, { s.size });
12     SetBlockFooter(s, { s.ptr });
13     // remaining block
14     Block r = { b.ptr + size, b.size - size };
15     SetBlockHeader(r, { r.size });
16     SetBlockFooter(r, { r.ptr });
17     // insert remaining block in the tree, so that it becomes available
18     InsertBlock(r);
19   }
20
21   return b.ptr;
22 }
23
24 void* Dealloc(void* ptr) {
25   Header h = GetBlockHeader(ptr);
26   // at this point we own the block and it is not in the tree
27   Block b = { h.size, ptr };
28
29   b = CoalesceBackward(b);                           // attempt backward coalescing
30   b = CoalesceForward(b);                             // attempt forward coalescing
31
32   // add the coalesced block to the tree
33   InsertBlock(b);
34 }
```

Listing 7.3: High-level allocation and deallocation

7.4.2 Coalescing

Coalescing only occurs when a block has been deallocated, which means that the block is not yet present in the tree. To coalesce a deallocated block, we have to locate its neighbors with the map, and then attempt to remove each of those neighbors from the tree. If we can remove a neighboring block from the tree, we have acquired ownership of it, and thus we can use it for coalescing. Both backward and forward coalescing operate in the same manner, by attempting to remove the previous/next block from the tree, and then performing coalescing.

Listing 7.4 shows the backward and forward coalescing routines. Backward coalescing begins by locating the previous block through its footer (line 2). A noteworthy detail, shown in line 3, is that the footer is sufficient to obtain the size of the previous block, thus avoiding the need to read the previous block's header. Once the previous block has been located, we can attempt to remove it from the tree with **RemoveBlock()**, shown in line 5. Note that the tree stores blocks

not only by address, but also by size. This is important, as we could otherwise be removing a block that starts at the previous block's address, but has a different size (thus not sharing a boundary with our block). If the removal of the previous block is successful, we can finally perform coalescing, and update the headers and footers stored in the map (lines 8–11).

The forward coalescing routine starts on line 15. The next block is located through its header (line 16), and then we attempt to remove it from the tree. If the removal succeeds, the provided block can be coalesced with the next block (lines 22–25).

```
1  Block CoalesceBackward(Block b) {
2    Footer f = GetBlockFooter(b.ptr - 1);         // get footer of previous block
3    Block p = { f.ptr - b.ptr, f.ptr};            // compute size using block addresses
4
5    if (!RemoveBlock(p))
6      return;                                     // previous block is not free
7
8    // can coalesce, update map to reflect change
9    b = { b.size + p.size, p.ptr };
10   SetBlockHeader(b, b.size);
11   SetBlockFooter(b, b.ptr);
12   return b;
13 }
14
15 Block CoalesceForward(Block b) {
16   Header h = GetBlockHeader(b.ptr + b.size);    // get header of next block
17   Block n = { h.size, b.ptr + b.size };         // compute block address using size
18
19   if (!RemoveBlock(n))
20     return;                                     // next block is not free
21
22   // can coalesce, update map to reflect change
23   b = { b.size + n.size, b.ptr };
24   SetBlockHeader(b, b.size);
25   SetBlockFooter(b, b.ptr);
26   return b;
27 }
```

Listing 7.4: Backward and forward coalescing

7.4.3 Implementation

For testing and development purposes, we have implemented a simple concurrent memory allocator that allocates pages using our lock-free best-fit design. In our implementation, the map is implemented with a simple array made of uncommitted pages (similar to LRMalloc's pagemap), and the tree is implemented using a lock-free binary tree as proposed by Natarajan *et al.* [41].

Integrating our coalescing mechanism into a lock-free memory allocator, such as LRMalloc, is on-going work, as is the performance evaluation of this mechanism, which is important to assert whether this mechanism can be used in practice to replace the operating system as a lower-level allocator.

7.4.4 Caveats

There are a number of caveats with our proposal that we would like to point out.

Our design cannot guarantee that coalescing always occur. Consider an example with two neighboring blocks, B1 and B2, operated by thread T1 and T2, respectively. If T1 and T2 deallocate B1 and B2 simultaneously, T1 will try to remove B2 from the tree, and T2 will try to remove T1, and both will fail. Then T1 will add B1 to the tree and T2 will add B2 to the tree. We thus end up with two blocks, B1 and B2, which are free and adjacent to each other, but not coalesced. This can be mitigated by performing *recursive coalescing*, e.g., continue coalescing forward or backward until it fails. In the context of the example, that ensures that both B1 and B2 will be coalesced when another neighbor block is deallocated and attempts to coalesce.

The failure of coalescing reveals an interesting property – as contention increases and more threads operate on the same space with a small number of blocks, the more likely it is that coalescing fails. We leave as an open problem whether this property can be formally described, and what guarantees our solution of recursive coalescing offers.

Another caveat is that the tree is implemented by a lock-free data structure that likely uses internal nodes that have to be dynamically allocated. In particular, the data structure we have chosen in our implementation is vulnerable to the ABA problem and requires the use of a memory reclamation method such as epochs or hazard pointers [20, 39]. Dynamically allocating memory for a data structure in a memory allocator is naturally tricky, as is the use of memory reclamation techniques. Both of these requirements are problematic and complicate the design of the memory allocator. For our design to be used as a lower-level allocator in a production memory allocator such as TCMalloc or Jemalloc, the tree component would preferably not need to dynamically allocate memory. We leave it as another open problem whether there is a lock-free data structure that is better suited to this context.

7.5 Chapter Summary

This chapter presented the first known lock-free address ordered best-fit mechanism, capable of coalescing and splitting of blocks with arbitrary sizes. We have discussed the mechanism's utility, quirks, implementation difficulties and challenges.

Chapter 8

Conclusion

This chapter, summarizes the main contributions of this work, proposes directions for future work and wraps up with some final remarks.

8.1 Main Contributions

We have presented LRMalloc, a lock-free memory allocator designed to fulfill important and desirable properties in a memory allocator, such as, immunity to deadlocks and livelocks, tolerance to arbitrary thread termination and priority inversion, and a strong guarantee that there is progress in the memory allocator if there is at least one thread executing, regardless of scheduling. It is our belief that future progress in memory allocators would involve the adoption of lock-free strategies as a way to provide these behavior properties to user applications.

Our experiments showed that LRMalloc's current implementation is already quite competitive and comparable to other modern state-of-the-art memory allocators. In particular, it has much better performance characteristics than prior lock-free memory allocators such as Michael's allocator or NBMalloc. We believe that our work clearly shows that a next iteration of state-of-the-art memory allocators can not only keep current performance characteristics but also acquire lock-freedom, in order to provide better properties to user applications. To achieve such, future lock-free memory allocators must effectively employ thread caches in order to amortize the cost of atomic operations throughout several allocation and deallocation requests, as well as a lock-free component that can effectively transfer lists of blocks to and from threads caches with a small number of atomic operations.

We have also devised a lock-free best-fit mechanism that is capable of lock-free coalescing and splitting. To the best of our knowledge, this is the first lock-free general coalescing mechanism – it does not need to use size classes, nor puts restrictions on how coalescing can be performed. Our mechanism ultimately aims to be used as a lower-level allocator for lock-free memory allocators, allowing them to manage memory without the assistance of the operating system, and potentially improving performance by reducing the number of system calls.

8.2 Future Work

Nevertheless, there are a number of possible extensions that we plan to study which could further improve LRMalloc's performance. A first example is the support for multiple arenas as a way to reduce allocation contention as the number of threads increases, thus improving scalability. Another good example is the usage of improved cache management algorithms, as the ones implemented by TCMalloc [32], as a way to reduce the average memory allocator latency and reduce overall memory usage. LRMalloc currently also induces passive false sharing, which can lead to performance degradation in multithreaded applications.

LRMalloc's current implementation uses a lock-free scheme which relies on the operating system to provide continuous pages of memory, which we use to construct superblocks. We plan to integrate our best-fit coalescing mechanism with LRMalloc, in order to replace the roll of the operating system as a lower-level allocator.

We also have a number of open problems regarding our lock-free best-fit coalescing mechanism. A sample implementation we have made uses a lock-free binary tree for our mechanism, which we have found is not ideal due to concerns with memory reclamation and ABA prevention. We intend to find a lock-free data structure that is better suited for our mechanism. Furthermore, our mechanism fails to perform coalescing if several threads attempt simultaneously attempt to coalesce neighboring blocks. The failure of coalescing reveals an interesting properly – as contention increases, the more likely it is that coalescing fails. We would like to formally describe this property, and what guarantees the solution we have proposed (recursive coalescing) provides.

Lastly, the subject of concurrent memory allocation currently lacks a comprehensive survey that describes the existing mechanisms and techniques. Chapters 3 and 4 in this document aim to fill part of that gap, but it still lacks a through and up-to-date taxonomy of concurrent memory allocators.

8.3 Final Remarks

LRMalloc is available from https://github.com/ricleite/lrmalloc, licensed under MIT license. It fully implements the C11 and POSIX malloc interface.

Bibliography

[1] Jonathan Afek and Adi Sharabani. Dangling Pointer: Smashing the Pointer for Fun and Profit. *Black Hat USA*, 2007.

[2] Miguel Areias and Ricardo Rocha. An Efficient and Scalable Memory Allocator for Multithreaded Tabled Evaluation of Logic Programs. In *Parallel and Distributed Systems, 2012 IEEE 18th International Conference on*, pages 636–643. IEEE, 2012.

[3] Miguel Areias and Ricardo Rocha. A Simple and Efficient Lock-Free Hash Trie Design for Concurrent Tabling. In *Technical Communications of the 30th International Conference on Logic Programming*, 2014.

[4] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 117–128. ACM, 2000.

[5] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6), 2009.

[6] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[7] Jeff Bonwick and Jonathan Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX Annual Technical Conference, General Track*, pages 15–33, 2001.

[8] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why is it only a Research Toy? *Queue*, 6(5):40, 2008.

[9] C++ Standard Committee. Information Technology – Programming Languages – C. Standard, International Organization for Standardization, 2011.

[10] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs. In *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192. IEEE, 2010.

[11] Igor Dobrovitski. Exploit for CVS double free() for linux pserver, 2003.

[12] Leah Epstein and Rob van Stee. Improved results for a memory allocation problem. *Theory of Computing Systems*, 48(1):79–92, 2011.

[13] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conference*, 2006.

[14] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-Free Algorithms can be Practically Wait-Free. In *International Symposium on Distributed Computing*, pages 78–92. Springer, 2005.

[15] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[16] Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-caching malloc, 2009. http://goog-perftools.sourceforge.net/doc/tcmalloc.html (read on June 14, 2018).

[17] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. NBmalloc: Allocating Memory in a Lock-Free Manner. *Algorithmica*, 58(2):304–338, 2010.

[18] David Gifford and Alfred Spector. Case Study: IBM's System/360-370 Architecture. *Communications of the ACM*, 30(4):291–307, 1987.

[19] Wolfram Gloger. Ptmalloc, 2006. http://www.malloc.de/en (read on June 14, 2018).

[20] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[21] Maurice Herlihy and J Eliot B Moss. *Transactional Memory: Architectural Support for Lock-Free Data Structures*, volume 21. ACM, 1993.

[22] Maurice Herlihy and Nir Shavit. On the Nature of Progress. In *International Conference On Principles Of Distributed Systems*, pages 313–328. Springer, 2011.

[23] Matthew Hertz and Emery D Berger. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In *ACM SIGPLAN Notices*, volume 40, pages 313–326. ACM, 2005.

[24] Mark S Johnstone and Paul R Wilson. The Memory Fragmentation Problem: Solved? In *ACM SIGPLAN Notices*, volume 34, pages 26–36. ACM, 1998.

[25] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 158–169. ACM, 2015.

[26] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating Memory Allocation. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 33–45. ACM, 2017.

[27] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with threads.* Sun Soft Press Mountain View, 1996.

[28] Donald E Knuth. *The Art of Computer Programming; Volume 2: Seminumeral Algorithms.* 1981.

[29] David G Korn and Kiem-Phong Vo. In search of a better malloc. In *Proceedings of the Summer 1985 USENIX Conference*, pages 489–506, 1985.

[30] Per-Åke Larson and Murali Krishnan. Memory Allocation for Long-Running Server Applications. *ACM SIGPLAN Notices*, 34(3):176–185, 1998.

[31] Doug Lea. A Memory Allocator Called Doug Lea's Malloc or dlmalloc for Short, 1996. http://g.oswego.edu/dl/html/malloc.html (read on June 14, 2018).

[32] Sangho Lee, Teresa Johnson, and Easwaran Raman. Feedback Directed Optimization of TCMalloc. In *Workshop on Memory Systems Performance and Correctness*, page 3. ACM, 2014.

[33] Ricardo Leite and Ricardo Rocha. A Modern and Competitive Lock-Free Dynamic Memory Allocator. In *10th INForum - Simpósio de Informática (INForum 2018)*, 2018.

[34] Ricardo Leite and Ricardo Rocha. LRMalloc: a Modern and Competitive Lock-Free Dynamic Memory Allocator. In *13th International Meeting on High Performance Computing for Computational Science (VECPAR 2018)*, 2018.

[35] Chuck Lever and David Boreham. malloc() Performance in a Multithreaded Linux Environment. In *USENIX Annual Technical Conference*, pages 301–311. USENIX, 2000.

[36] Romolo Marotta, Mauro Ianni, Andrea Scarselli, Alessandro Pellegrini, and Francesco Quaglia. A non-blocking buddy system for scalable memory allocation on multi-core machines. In *2018 IEEE International Conference on Cluster Computing*, pages 164–165. IEEE, 2018.

[37] Scott Andrew Maxwell. *Linux Core Kernel Commentary.* Coriolis Group Books, 2001.

[38] Maged 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. ACM, 2002.

[39] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel & Distributed Systems*, 15(6):491–504, 2004.

[40] Maged M Michael. Scalable Lock-Free Dynamic Memory Allocation. *ACM Sigplan Notices*, 39(6):35–46, 2004.

[41] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 49, pages 317–328. ACM, 2014.

[42] Robert HB Netzer and Barton P Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1): 74–88, 1992.

[43] Ivor P Page. Analysis of a cyclic placement scheme. *The Computer Journal*, 27(1):18–26, 1984.

[44] Chuck Pheatt. Intel Threading Building Blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[45] Brian Randell. A note on storage fragmentation and program segmentation. *Communications of the ACM*, 12(7):365–372, 1969.

[46] John M Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 1977.

[47] John Michael Robson. Bounds for Some Functions Concerning Dynamic Storage Allocation. *Journal of the ACM*, 21(3):491–499, 1974.

[48] John E Shore. On the External Storage Fragmentation Produced by First-fit and Best-fit Allocation Strategies. *Communications of the ACM*, 18(8):433–440, 1975.

[49] Josep Torrellas, HS Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[50] John David Valois. Lock-free Data Structures. 1996.

[51] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Memory Management*, pages 1–116. Springer, 1995.

[52] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. Memory Allocation Policies Reconsidered. Technical report, Technical report, University of Texas at Austin Department of Computer Sciences, 1995.