Chapter 1

Introduction

In the early days of computing memory was fairly expensive and users more or less had free reign of the whole system. Computers were designed for simple tasks like numeric computations, memory was limited to the order of kilobytes, and the user was guaranteed that their program would be the only task running at execution time. Besides a few clever tricks to partition memory within parts of their own program, memory management was not particularly complicated.

As computers became more sophisticated they also became more general, to the point where multiple programs could be running at the same time. Now the guarantee of system resource supremacy was gone. In its place was something called an operating system, a lower level program, always running, which allowed multiple programs to use the system's resources, but in turn gave each process the impression that it had total access.

You may be familiar with this story, standard libraries have functions like C's malloc and free for allocating and deallocating memory. And sure, sometimes someone forgets to use these properly, and some other people take that personally and go away and write languages like Rust. But otherwise the memory allocation problem is solved, right?

Yes, and no. In the early days computers had a lot of limitations, and those limitations bred a lot of creativity in the engineers who squared off against them. Fast forward to today and although hardware has advanced a lot, the diversity of game engines has reduced quite a bit, and the technical know-how (for lack of a better term, the "street fighting programming") has too. This is evident in the number of AAA titles releasing now with worse performance than games ten years ago. The purpose of this paper is to survey the field of memory management, present some common algorithms, and discuss use cases. The paper advocates for a varied approach to memory management rather than a single black box allocator.

Chapter 2

Terminology

All gaming systems currently use the Von Neumann architecture.
(Figure: Von Neumann Architecture)
That is, they take input, process it and produce output. Input can come either directly from the user through peripheral devices, or from data stored in the system's main memory. The system can also store and retrieve data in memory. In order to speed up memory access, systems have a memory hierarchy called a cache. Data immediately being used by the system is stored in CPU registers. Beyond this, the CPU has L1, L2 and sometimes L3 cache, and further beyond this the system has its main memory (RAM), and even the hard drive. The memory stores of this hierarchy become progressively large, but also progressively slower to access.
(Table: Approximate clock cycles to access)

At runtime, memory can either be allocated on the stack or the heap. A program's stack is a region of memory reserved for any local variables and context, it's quite fast to allocate from the stack, but the stack is limited to the order of a few megabytes. Excessive stack allocations can cause stack overflows and crash a program. The heap is the phrase used to describe all the memory which a system has access to. With the advent of virtual memory this is no longer limited to the amount of RAM available. Heap allocation is more flexible, but involves a call to the operating system and higher overheads, as the heap is shared by many other programs running simultaneously on the system.

A memory allocation can be visualized as a pointer to the beginning of a region of memory, with the guarantee that only this pointer will have exclusive ownership over that region. Many allocators have the same public interface as C's standard library, namely the functions
$void*\ malloc(size\_t\ byteSize)$
and
$void\ free(void*\ location)$,
which allocate and free memory respectively. Internally, many allocators also share similar mechanisms. An allocator will typically request some region of memory from the operating system at the beginning of the program, and then use some sort of data structure to partition that region to satisfy runtime allocations. It is common to name these sub regions "blocks".

Over the runtime of the program, memory may become unused, this is called fragmentation. Fragmentation can either be external, when a block is freed but cannot be merged (coalesced) with its neighbors, or internal, when a block is allocated but not all of it is used. External fragmentation can cause situations where an allocator has enough free memory to satisfy a request, but cannot as that memory is not contiguous in a single block. Internal fragmentation can reduce cache hit rates by spreading allocated blocks from each other spatially, this has the effect that the memory allocated can be slow to work with.

Chapter 3

Literature Review

3.1 Fundamental Algorithms

Highly influential volume. A little dated, but all modern allocators (specifically free list and buddy) have their roots in these algorithms. Great to understand the common DNA of most allocators and includes some exercises for the motivated reader. Most other research papers will cite this in some way. [6]

3.2 The Garbage Collection Handbook: The Art of Automatic Memory Management

Has a very comprehensive chapter on memory allocation, explains a wide variety of standard allocators and the principles behind them. Presents the core terminology and problems of memory management, with a good variety of allocators and variants. This chapter is a good starting point, a little more modern than Knuth's treatment. [5]

3.3 Inside Memory Management

Briefly discusses the history of memory management.
Gives a simple example of how one might completely reimplement malloc using Linux Operating System calls (sbrk). Interesting but a little too in the weeds for this project. [1]

3.4 Memory allocation costs in large C and C++ programs

Critical in the evaluation of an allocator is the measurement of how quickly it allocates and frees memory. In the past it has been common practice to asymptotically analyze algorithms, however with the advent of instruction-level profiling tools it has become possible to measure the actual number of clock cycles an algorithm takes.
This paper evaluates various memory allocators for a variety of real malloc-heavy applications, by counting the number of instructions used for allocation and free operations.
While a good starting point, clock cycles alone are not a comprehensive measure of performance, for example they don't take into account theoretical bounds on access speed or the speed with which allocated memory can be used. [2]

3.5 The Memory Fragmentation Problem: Solved?

Fragmentation (both internal and external) can have an effect on system performance. Both by wasting space and by scattering useful data. Previously, fragmentation was measured by synthetic trace analysis. In some cases great efforts were taken to ensure traces had statistically significant properties. This paper re-examines the implicit assumption: that synthetic trace analysis is an indicator of fragmentation in real programs, and finds this not to be true.

The paper also attempts to evaluate allocator policies independent of implementation and concludes that most fragmentation problems are actually poor implementations. The paper also examines which common features are shared by low fragmentation algorithms, and finds that immediate coalescing on free and reuse of most frequently freed blocks gives best performance. The experiment design is well documented. [4]

3.6 Improving the cache locality of memory allocation

Previous research by Zorn just profiled allocators on the basis of instruction counts, but this neglects the possibility that an allocator may take more time to allocate, but allocate memory which is faster to use. This paper investigates the relationship between underlying allocator and execution time for a number of real programs. This sort of research into cache hit rates had only previously been undertaken for Garbage Collected languages, where its effect was obvious. The paper concludes that implementation rate effects both cache hit rate and total execution time for a number of allocation-heavy programs, while noting that there isn't a single implementation which is universally best for every program. [3]

3.7 TLSF: a new dynamic memory allocator for real-time systems

Proposes a new data structure, the Two Level segregated Free List. In Realtime Systems, scheduling algorithms require that performance be predictable and bounded. Traditional Segmented Free Lists can either cover a wide range of size classes or be very granular to reduce fragmentation. TLSF can do both and has constant access time. The indices for size classes at both levels can be calculated with bit-level operations, making them fast. A potential downside to the the method is that two pointer accesses are performed, and that the metadata for blocks is stored directly in blocks themselves, reducing the cache locality when accessing a block's neighbours (eg, for memory coalescing.) [7]

3.8 An algorithm with constant execution time for dynamic storage allocation.

Proposes a one level variant of TLSF (this paper was published before TLSF was proposed), based around "power of two" size classes. Each size class holds blocks in the size range $[2^i, 2^{i+1})$, making them flexible but still enforcing the condition that every block on the list must be able to fulfill a request. This means that memory requests do not require searching a free list. This has the advantage of one table lookup rather than two. Internal fragmentation is higher due to power of two size classes, but it's yet to be determined whether this matters in practice. [9]

3.9 Fast Bitmap Fit: A CPU Cache Line friendly memory allocator for single object allocations

Proposes encoding a small binary search tree directly into the bits of an integer. The proposed allocator can quickly search, allocate and free as the whole structure is cache-local. The allocator can also take a "hint" to attempt adjacent allocations. In this system, the search structure is independent of the memory block, and all search, allocation and free can be done with bit level operations on a single unsigned integer, making the search structure very fast. As a tradeoff, the allocator is more limited. A certain maximum number of allocations can be made (at least in the naive form), and allocations must all be fixed size. This approach is interesting but may be limited in its use cases. [8]

3.10 New methods for dynamic storage allocation (Fast Fits)

Compares Free Lists with Buddy Allocators and proposes a new data structure which performs somewhere between the two. Fast Fits are based around a novel data structure called a Cartesian tree. A cartesian tree is a two dimensional generalisation of a search tree where addresses are ascending from left to right and sizes are descending from top to bottom. This has the advantage that the root node has the highest block size, and requests which are too large can be quickly rejected. The paper also has a lack of practical results, so could be a novel approach to investigate. However the constraints of the Cartesian tree mean that the tree can't be rebalanced, and so the worst case performance is linear in highly unbalanced scenarios. This makes the worst case execution time difficult to predict. Being a fairly old paper it also benchmarks itself against outdated free list allocators. [11]

3.11 Self-adjusting binary search trees

3.11.1 Summary

Introduces Splay Trees. A splay tree is a binary search tree where the most recent search result is "splayed" or rotated to the root. This gives the standard binary search tree performance for searching, with the added benefit that frequently accessed nodes will be quicker to retrieve. This would fulfill Johnstone and Wilson's optimal strategy for reduced fragmentation, namely that memory is coalesced immediately upon free, and recently freed blocks have a tendency to be reused first. [10]

Bibliography

[1] Jonathan Bartlett. Ibm developer — developer.ibm.com. `https://developer.ibm.com/tutorials/l-memory/`, 2004. [Accessed 10-09-2024].

[2] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large c and c++ programs. *Softw. Pract. Exper.*, 24(6):527–542, jun 1994.

[3] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. *SIGPLAN Not.*, 28(6):177–186, jun 1993.

[4] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? *SIGPLAN Not.*, 34(3):26–36, oct 1998.

[5] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st edition, 2011.

[6] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., third edition, 1997.

[7] M. Masmano, I. Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. volume 16, pages 79– 88, 01 2004.

[8] Dhruv Matani and Gaurav Menghani. Fast bitmap fit: A cpu cache line friendly memory allocator for single object allocations, 2021.

[9] T. Ogasawara. An algorithm with constant execution time for dynamic storage allocation. In *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*, pages 21–25, 1995.

[10] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, jul 1985.

[11] C. J. Stephenson. New methods for dynamic storage allocation (fast fits). In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, page 30–32, New York, NY, USA, 1983. Association for Computing Machinery.