Memory Allocation Algorithms

Andrew Mengede

November 4, 2024

1 Introduction

Memory is vital for any computer to do complicated tasks, rather than operate as a simple automaton. In this paper I will examine the practice of manual memory management, specifically with regards to soft real time simulations such as videogames. I will present some common memory allocation techniques, discuss some of the relevant considerations in performance evaluation of allocators, and present a practical benchmark.

Computers operate on voltages, they consist of semiconductors which implement logic functions. Given some number of inputs, a response is produced. In this way, computers can operate without memory. These simplified computers could be purely functional (such as simple calculators), or could hard code behaviour in their circuitry and operate as "Finite State Machines", having well defined specific purposes.

Memory allows computers to extend beyond this limited (but admittedly large) set of functionality. Memory can be visualised as a long sequence of voltages. Computers can read one or more of these voltages at a time and react accordingly. Computers can also write their own output back to the sequence, either at an untouched region or over any previously used region. In this way, the sequence of voltages can encode information. The encoding and decoding of information depends entirely on the computer. The sequence of voltages can even represent instructions to the computer, making computers with memory general-purpose machines.

Such concepts have theoretically existed since Babbage's proposal of the "Analytical Engine", popularised by Ada Lovelace [2], however in practice memory has been expensive to implement. The first electronic general computers [10] had a limited number of "registers" and a small amount of temporary memory. Computer scientists working with these machines had a hard time writing machine code for the specific processor, but also had full access to all resources on the system [3]. In other words, 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 (so called "Multiprocessing"). 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. Operating Systems became increas-

ingly sophisticated and incorporated techniques like "Virtual Memory", where regions of RAM can be swapped out to disk, allowing more memory to be allocated than the system actually has available.

All of this had the effect that memory management was taken care of by the operating system. For instance, C's standard library has the *malloc*, *realloc* and *free* functions, which perform operating system calls in order to manage 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. Video games run on consoles which can have limited abilities compared to modern computers [6]. These can come in the form of less virtual memory support and increased time penalties for cache misses. Custom memory allocators can be built on top of the operating system's provided allocators which produce better performance both for allocation/free times and for access times on the memory itself. Although the gap between consoles and PCs is narrowing, mobile gaming is also a fairly large industry, which has similar limitations.

2 Terminology

All gaming systems currently use the Von Neumann architecture.

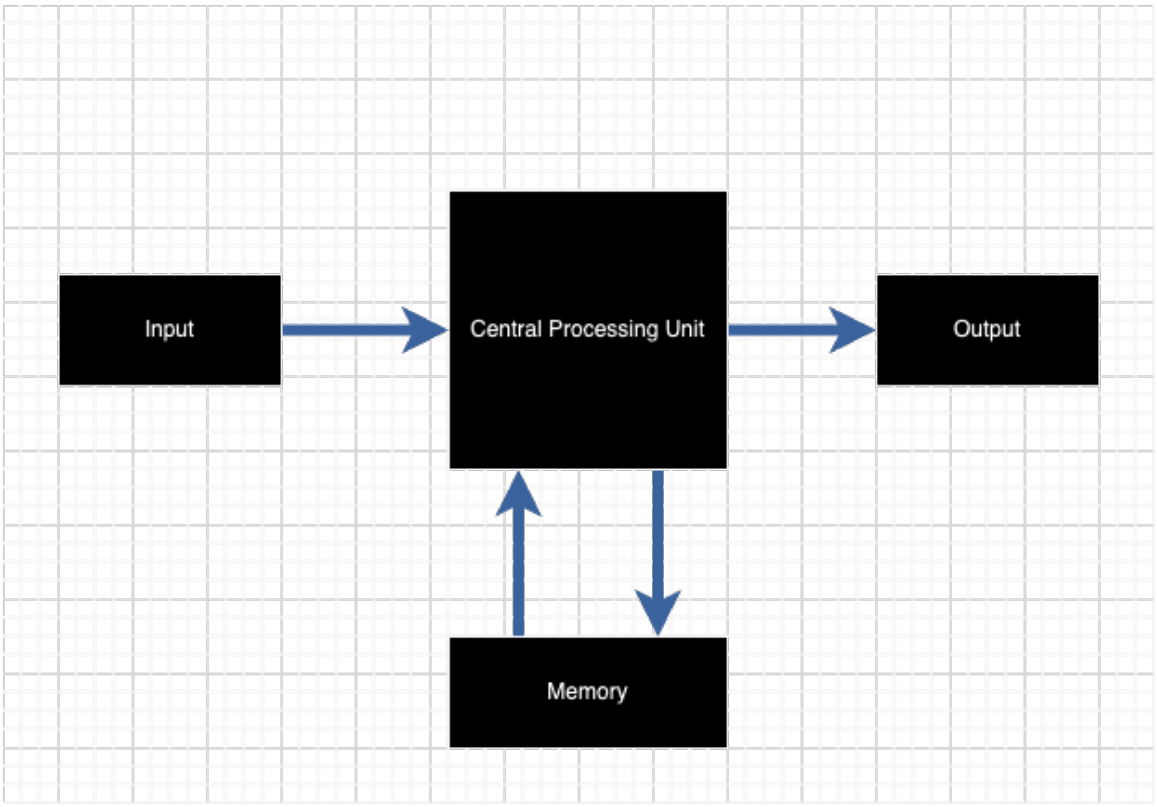

Figure 1: Von Neumann Architecture

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.

- L1 cache: ~1–2 nanoseconds
- L2 cache: ~5–10 nanoseconds
- L3 cache: ~10–20 nanoseconds
- Main memory (DRAM): ~60–100 nanoseconds
- Disk storage (HDD): ~5–10 milliseconds
- Disk storage (SSD): ~0.1–1 milliseconds

Figure 2: Approximate time to access various memory types [12]

When a program is compiled and run it is allotted a certain amount of memory automatically, this is called its stack. A common use case for the stack is function calls. When a program calls a function, local variables inside the new function need to be stored somewhere. The program also needs to know, when the function ends, which location to return back to [4].

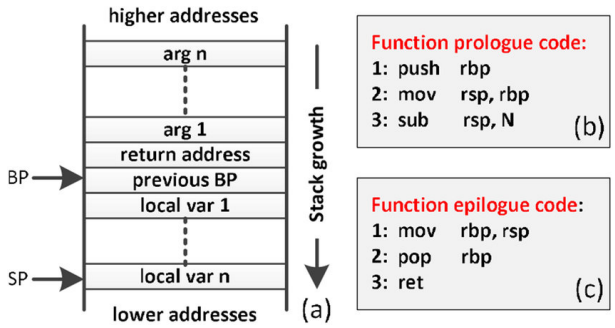

Figure 3: Example stack usage for function call. Arguments are stored in reverse order so arbitrary numbers of arguments can be handled. The function prologue and epilogue are called at the beginning and end of the function respectively in order to save and restore the return address. [22]

In addition to use cases like this, it's possible (and quite fast) to allocate memory from the stack. However, since the stack is allocated to the program at compile time, it is not intended to handle large runtime allocations and is limited to the order of a few megabytes. Excessive stack allocations can cause stack overflows and crash a program.

For general runtime allocations, the application programmer also has access to the system heap. In contrast to the heap data structure, the system's memory heap is the phrase used to describe all the memory which a system has access to [11]. The heap is the most common source from which memory is allocated at runtime. 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, possibly simultaneously using the same physical memory (thanks to virtual memory).

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

```
1 void* malloc(size_t byteSize);
```

and

```
1 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".

Upon a memory request, the allocator will attempt to find a free block which is able to satisfy the request. Some allocators deal with fixed sizes and others are flexible. It's common to split blocks so that the unused space can be used for future allocations.

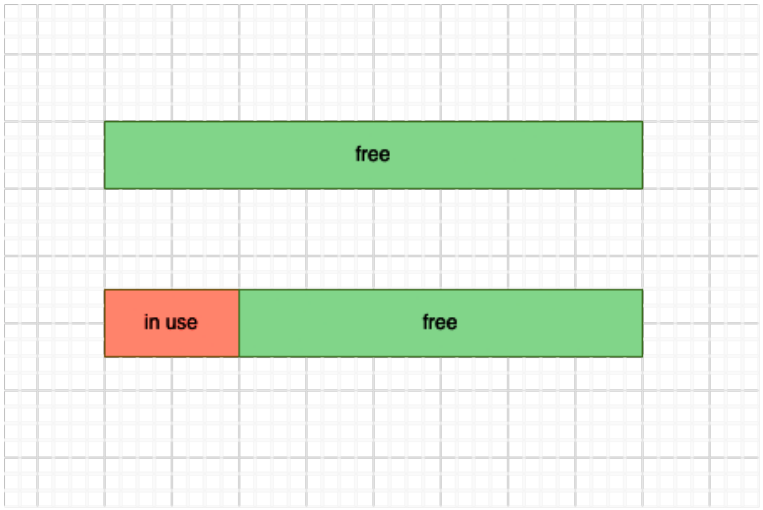

Figure 4: Splitting a block

Over the runtime of the program, memory which was previously allocated may be freed. In these cases, a custom allocator should have some mechanism to detect when neighbouring blocks can be merged together into one, larger free block. This merging process is called "coalescing".

The fundamental problem with dynamic memory allocation is it is dynamic. Over the lifetime of the program, an allocator's initial slab of memory may undergo many modifications. These modifications can make regions of memory unusable, this is known as fragmentation. Fragmentation

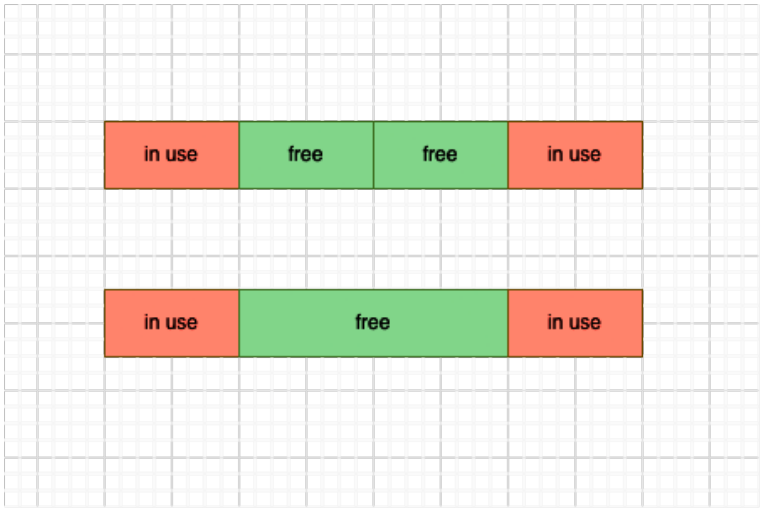

Figure 5: Memory Coalescing

comes in two forms, external fragmentation occurs when free blocks exist which cannot be coalesced together. Although the allocator has enough free memory, it may still fail to satisfy requests due to not having a large enough free block.

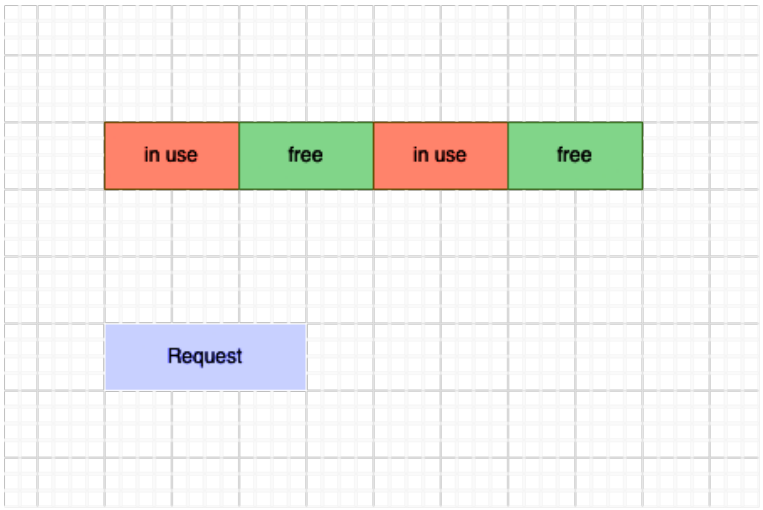

Figure 6: External fragmentation

Internal fragmentation describes the scenario where the allocator marks off a larger block than requested, does not split, and hence leaves some portion of a block unused. This may come about due to memory alignment/padding requirements, or the allocator's own mechanism. For instance fixed-size allocators exist which will not split blocks.

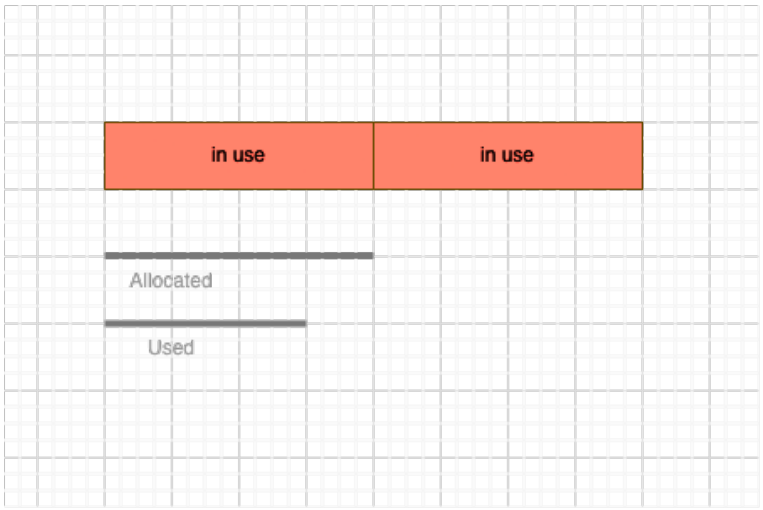

Figure 7: Internal fragmentation

Cache friendly programming can make a big difference to performance. As noted previously, in practice the Central Processing Unit attempts to fulfill fetches with its cache. Memory is not fetched a byte at a time, but by cache lines, typically 64 bytes. This means that if an application programmer is able to maximise the amount of "useful data" per cache line, the program will probably run faster. For instance, a linked list typically involves a number of nodes, each with a pointer to other nodes. If an allocator allocates space for these nodes at random locations on the heap then a lot of time is wasted traversing the list as each pointer access means fetching a new cache line. Furthermore, if the allocator allocates nodes fairly close together in memory, but incurs internal fragmentation, then the amount of useful data per cache line is reduced, and traversal may involve more cache ejections and refreshes than necessary. Though such allocators may satisfy requests quickly, the memory they allocate can still be slow to access.

Some memory requests can also request the returned pointer be a multiple of a number, usually a power of two. This practice is called memory alignment. Although modern CPUs can mitigate this problem, misalignment can lead to less efficient memory access, as the memory may be needlessly split across two cache lines. The idea of memory alignment is to introduce some necessary padding so that the address becomes a multiple of the requested alignment [1]:

$padding = (align - (address \ mod \ align)) \ mod \ align$
$aligned = address + padding$
Since the alignment is a power of two, the modulus operator can reduce to a bitwise and which

clears off all bits above and including the align bit.

$$
\begin{aligned}
padding &= (align - (address \ \& \ (align - 1))) \ \& (align - 1) \\
&= ((align \ \& (align - 1)) - (address \ \& \ (align - 1))) \ \& (align - 1) \\
&= (0 - (address \ \& \ (align - 1))) \ \& (align - 1) \\
&= (\neg (address \ \& \ (align - 1)) + 1) \ \& (align - 1) \\
&= ((\neg address \ \vee \ \neg (align - 1)) + 1) \ \& (align - 1) \\
&= (-address \ \vee \ \neg (align - 1)) \ \& (align - 1) \\
&= (-address \ \& (align - 1)) \ \vee \ (\neg (align - 1) \ \& (align - 1)) \\
&= (-address \ \& (align - 1)) \ \vee \ 0 \\
&= -address \ \& (align - 1)
\end{aligned}
$$

This padding can then be added to the requested address to ensure it is correctly aligned.

3 Previous Work

Knuth's work [11] is considered the classic introduction to memory allocation algorithms. Here, the fundamental data structures, namely free list and buddy allocators, are discussed.

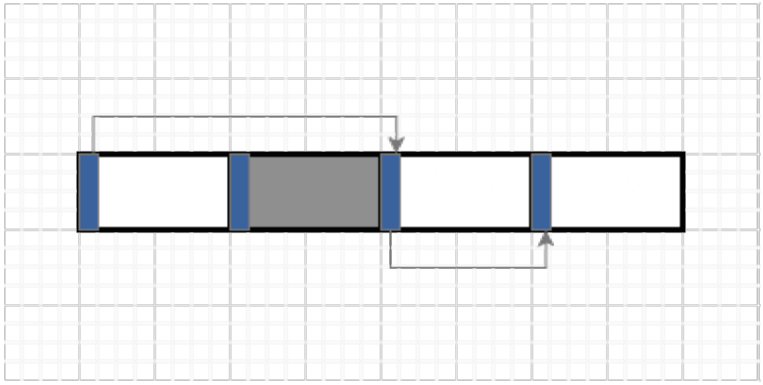

Figure 8: A linked-list representation of memory. Each header links to the header for the next free block. Neighbouring blocks can be found using block sizes.

Free list allocators are based around the idea of linked lists, each node in the linked list describes a region of memory, for instance its size, padding and whether it is currently free or in use. Two free list algorithms are presented by Knuth: first fit and best fit. First fit traverses a linked list of free nodes to find the first one which is large enough to satisfy the request. Best fit searches the whole list to find the smallest such node. Both of these strategies are linear in their worst case execution time, which may not be desired for large systems.

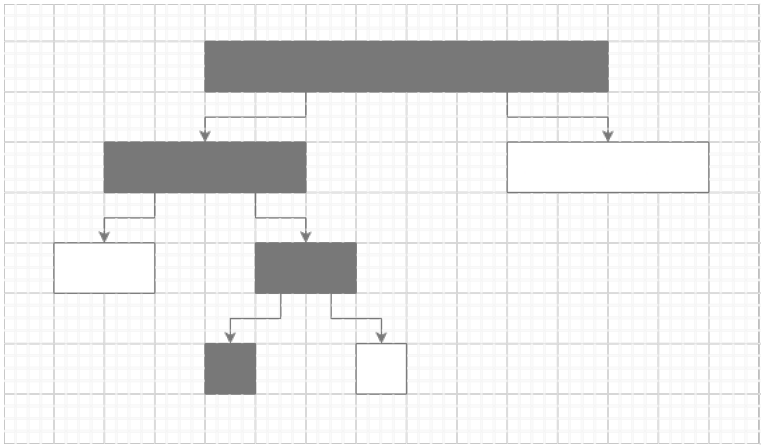

Figure 9: A buddy allocation system, represented here with a binary tree.

Knuth also discusses buddy systems, these are similar except that blocks are split in half until a suitable size is allocated. Buddy systems can be made faster than free lists by imposing a binary search tree structure, giving them logarithmic search time. Knuth also mentions the benefit of immediately freeing memory over garbage collection.

In the following years some incremental progress has been made on these fundamental algorithms. Hosking, Moss and Jones [9] summarise some of these in the Garbage Collection Handbook. The concepts of size classes are discussed, along with the internal fragmentation problem of buddy allocators. Some more advanced data structures such as splay trees, Cartesian trees and bitmap fits are presented. The authors also briefly discuss considerations for multi threaded allocation, however such a topic is out of scope for this project. At any rate the authors conclude that most of the gains of multi threading can be realised by having well designed single thread allocators run independently.

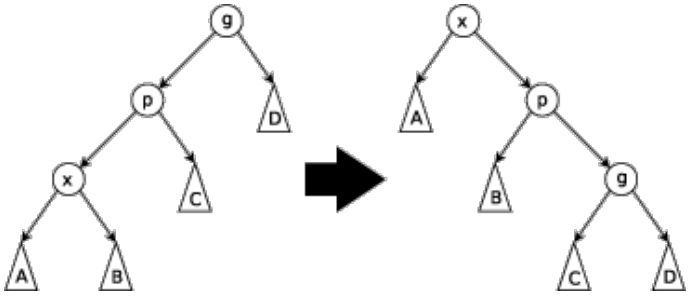

Figure 10: A splay tree, before and after searching for node x.

Splay Trees, first proposed by Sleator and Tarjan [17], are binary search trees where upon search, the result is promoted to the root of the tree. This has the benefit that frequently searched nodes are "hot" in memory and require less work to retrieve in future searches.

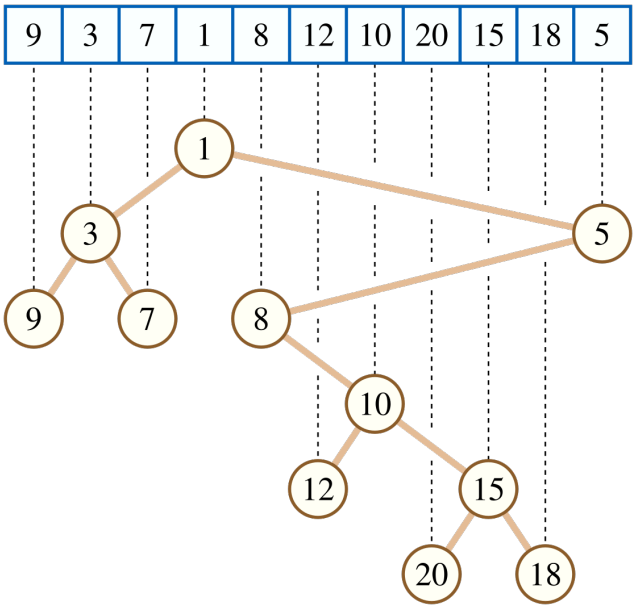

Figure 11: A Cartesian tree, x axis represents memory location, y axis represents value.

Cartesian trees, presented by Vuillemin [21], are a two dimensional generalization of search trees. In a Cartesian tree, addresses are ordered from left to right and sizes are ordered 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. A notable constraint of Cartesian trees is the inability to be re-balanced, and so the probablilty of highly unbalanced trees with linear search time can increase. This makes the worst case execution time difficult to predict.
Stephenson proposes a memory allocator based on Cartesian Trees in [19], named "Fast Fits". Stephenson found that Fast Fits performs somewhere between Free Lists with Buddy Allocators. Although neither Cartesian nor Splay trees were investigated in this report, they are novel approaches which could be examined in future work.

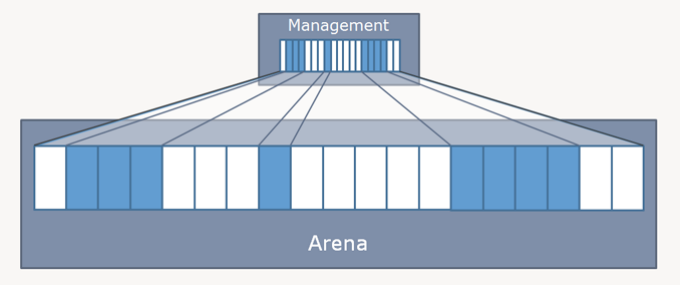

Figure 12: Separating headers from blocks

As mentioned previously, maximising the amount of useful data per cache line will tend to make programs run faster, the same principle holds for the data structure used to track blocks. As such, a number of allocators have been developed which separate the block headers from the blocks they manage.

Matani and Menghali [14] propose such an allocator which takes this concept even further. A fast bitmap fit allocator encodes 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, all searches, allocations and frees can be done with bit level operations on a single unsigned integer, making the search structure very fast. As a trade-off, 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 for fixed size object pools, but may be limited in its use cases.

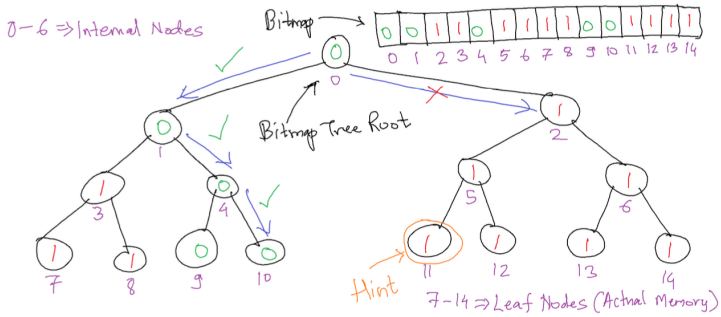

Figure 13: Bitmap allocator, leaf nodes represent actual memory blocks. Algorithm chooses search directions at each level in order to minimise the distance to the allocation hint.

A big limitation with the allocators discussed so far is not necessarily their execution time, but the predictability of their worst case execution times. Real time systems need some guarantee that a task will complete within a bounded interval of time, and so a number of allocators have been proposed which guarantee this.
Ogasawara proposed a constant time allocator in [15] which maintains an array of free lists, each corresponding to a size class in the range $\left[2^i, 2^{i+1}\right)$. This keeps size classes flexible and enforces 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 or binary tree, provided the free list has a block, that block will be appropriate. Upon allocation, blocks can be split, reducing internal fragmentation. While internal fragmentation is reduced, this can still have the effect of scattering memory allocations randomly through memory.

Ogasawara's constant time allocator is based on "power of two" size classes, this covers a wide range of sizes but is fairly coarse. Linear size classes can fill in gaps but are difficult to effectively search for large ranges of sizes. Knuth had experimented with multi-level allocators but did not achieve satisfactory results, nonetheless commenting that multi level allocators were theoretically possible.

Masmano, Ripoli, Crespo and Real [13] developed a two level structure which they called the Two Level Segregated Fit (TLSF). This allocator manages a first level which holds "power of two" size classes. Each first level size class links to a second level, an array of linear size classes. The power of this structure is not in its concept, but its execution. Every memory size can be mapped to an underlying free list via the coordinates (f, s):
$f := \lfloor \log_2 (size) \rfloor$
$s := \left(size - 2^f\right) \frac{2^{SLI}}{2^f}$
Where SLI, the second level index, is a parameter which describes how many second levels each first level size class is split into. It turns out that (f, s) can be computed with bit-level operations. The authors describe some further optimisations, chief among them: by restricting the number of first and second level size classes to 32 at most, integers can be used to track the free status of each size class, reducing the work to search for available free lists able to satisfy (f, s). Using integers also restricts the worst case search time.
TLSF is considered a good general allocator, especially in real time systems. Elegant in its design and fairly flexible, perhaps its only shortcoming is a lack of multi threading support, which is out of scope for this project.

While the development of allocation algorithms is important, the ability to benchmark allocators is equally so. Traditionally this was done with asymptotic analysis. While a good tool, this is not the only important metric. With the development of instruction-level profiling tools, it became possible to measure exactly how much work an allocator was doing. Detlefs, Dosser and Zorn [5] published one such paper where various memory allocators were compared for a variety of real malloc-heavy applications, by counting the number of instructions used for allocation and free operations.
One of the chief conclusions of the paper was that a well designed garbage collected allocator could compete with manual memory managers. Although the authors acknowledged that the programs themselves took significantly longer to run with garbage collection, possibly due to a trade off between Garbage Collection overhead (running more frequently) and external fragmentation (running less frequently). Nevertheless, the concept of measuring clock cycles for allocators was an important step forwards.

Fragmentation (both internal and external) can have an effect on system performance. Both by wasting space and by scattering useful data. Therefore it is important to have some means of estimating an allocator's fragmentation rate. Previously, fragmentation was measured by synthetic trace analysis. That is, randomly generating a series of malloc and free calls, then replaying that. In some cases great efforts were taken to ensure traces had statistically significant properties. Johnstone and Wilson [8] re-examined the implicit assumption in these benchmarks: that synthetic trace analysis is an indicator of fragmentation in real programs, and found this not to be true.

The authors used a journal replay method. Real programs were executed which logged requests to file, this gave a real trace to be analysed. This "Memory Journal" could then be replayed, and at any point an allocator's total unusable memory and requested memory could be queried and compared as a means of estimating fragmentation. It was found that in real use cases, fragmentation rates were lower than previously predicted, and acceptable for most allocators. In addition to challenging synthetic trace analysis, another goal of the study was to evaluate allocator policies independent of implementation. The authors concluded that immediate coalescing on free and reuse of most frequently freed blocks gives best performance.

Previous research by Grunwald and Zorn had profiled allocators on the basis of instruction counts,

but this neglects the possibility that an allocator's allocation or free time may be independent of how fast the allocated memory is to use. Grunwald, Zorn and Henderson wrote a follow up paper [7] with the aim of investigating the relationship between allocator and execution time for a number of real programs. More specifically, they attempted to measure cache occupancy. This sort of research into cache hit rates had only previously been undertaken for Garbage Collected languages, where its effect was obvious. It was concluded that the choice of allocator effects both cache hit rate and total execution time for a number of allocation-heavy programs, but that there isn't a single implementation which is universally best for every program.

4 In Practice

In order to see how some of this theory might work in practice I decided to recreate a blog post by Forrest Smith [18]. Smith wanted to challenge the notion that the standard C malloc routine is slow by profiling it on a real, open source program, Doom 3. The experimental design went as follows:

- Modify the allocate and free functions in Doom 3 source code to write a description of the request to a memory journal.
- For a number of different allocators, replay the journal, timing each allocation or free operation
- Plot the results, do some statistical analysis

I downloaded the same source port as Smith and got it running with the proper assets. I also identified the point at which to log info out, though unfortunately some further modification would be needed to enable the program to write out to files. Determining that my goal was to profile allocators rather than produce an original trace, I downloaded Smith's memory journal and used that. Journal lines had the format:

```
1     //a size pointer threadID timestamp
2     a 2048 000002C453C48560 7360 329200
```

for allocations, or

```
1     //f pointer threadID timestamp
2     f 000002C453C17290 7360 911600
```

for frees.
I designed an abstract allocator class with the following interface:

```
1  class Allocator{
2      public:
3      virtual void* custom_malloc(size_t size, size_t alignment) = 0;
4      virtual void* custom_realloc(size_t size,
5                      size_t alignment,
6                      void* oldAddress) = 0;
7      virtual void custom_free(void* data, bool debug) = 0;
8      std::string name;
```

```
};
```

This could then be used for journal replay (full listing in Appendix A)

```
void benchmark(Allocator& allocator) {

    // Local variables to track active allocations, allocation sizes
        etc.
    // ...

    // pre read and count the number of allocations and frees
    std::ifstream journal;
    journal.open("../data/doom3_journal.txt");
    int64_t malloc_count = 0, free_count = 0;
    while (std::getline(journal, line)) {

        // ...

    }

    // data sets for results.
    // x axis: timestamp
    // y axis: duration
    // color: size
    // ...

    while (std::getline(journal, line)) {
        std::vector<std::string> words = split(line, "␣");

        if (words[0].compare("a") == 0) {
            //allocate
            //...

            //---- time this ----//
            auto start = std::chrono::high_resolution_clock::now();
            void* data = allocator.custom_malloc(size, 16);
            auto end = std::chrono::high_resolution_clock::now();
            float duration = std::chrono::
            duration_cast<std::chrono::nanoseconds>(end - start).
               count();
            //--------------------//

            //---- log ----//
            // ...
            //------------//

        }

        if (words[0].compare("f") == 0) {
```

```
44              //free
45              //...
46
47              //---- time this ----//
48              auto start = std::chrono::high_resolution_clock::now();
49              if (data) {
50                  allocator.custom_free(data);
51              }
52              auto end = std::chrono::high_resolution_clock::now();
53              float duration = std::chrono::
54              duration_cast<std::chrono::nanoseconds>(end - start).
                    count();
55              //--------------------//
56
57              //---- log ----//
58              //...
59              //------------//
60          }
61      }
62
63      // Write back results
64      std::ofstream file;
65      std::stringstream filename_builder;
66
67      filename_builder << "../data/" << allocator.name << "_allocate.
            txt";
68      file.open(filename_builder.str(), std::ofstream::out);
69      for (size_t j = 0; j < malloc_count; ++j) {
70          file << ...;
71      }
72      file.close();
73
74      filename_builder.str("");
75      filename_builder << "../data/" << allocator.name << "_free.txt";
76      file.open(filename_builder.str(), std::ofstream::out);
77      for (size_t j = 0; j < free_count; ++j) {
78          file << ...;
79      }
80      file.close();
81  }
82
83  int main() {
84      FreeListAllocator bestFitList(FreeListAllocator::PlacementPolicy
            ::FIND_BEST);
85      benchmark(bestFitList);
86      return 0;
87  }
```

I implemented an allocator based on C's standard library (See Appendix B), and found two more online, a TLSF [16] and linear free list allocator [20]. Some adaptation was necessary in order for them to comply with the abstract allocator.

I then plotted the results in python.

```
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt("data/TLSF_allocate.txt", delimiter="␣")

timestamp = vanilla_alloc_data[:,0]
time = vanilla_alloc_data[:,1]
size = vanilla_alloc_data[:,2]

valid_indices = time != 0

x = timestamp[valid_indices]
y = np.log10(time[valid_indices])
c = np.log2(size[valid_indices])

y_ticks = [1, 2, 3, 4, 5, 6]
y_labels = ["10ns", "100ns", "1us", "10us", "100us", "1ms"]

c_ticks = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
    21, 22, 23, 24, 25, 26, 27, 28, 29, 30,]
c_labels = ["", "4␣b", "", "16␣b", "", "64␣b", "", "256␣b", "", "1␣
   kb",
    "", "4␣kb", "", "16␣kb", "", "64␣kb", "", "256␣kb", "", "1␣mb",
    "", "4␣mb", "", "16␣mb", "", "64␣mb", "", "256␣mb", "", "1␣gb",]

thing = plt.scatter(x = x, y = y, c = c, s = 0.2, cmap="gist_ncar")
plt.yticks(ticks=y_ticks, labels=y_labels)
cbar = plt.colorbar(ticks=c_ticks)
cbar.set_ticklabels(c_labels)
plt.tight_layout()
plt.show()
```

To produce the following graphs:

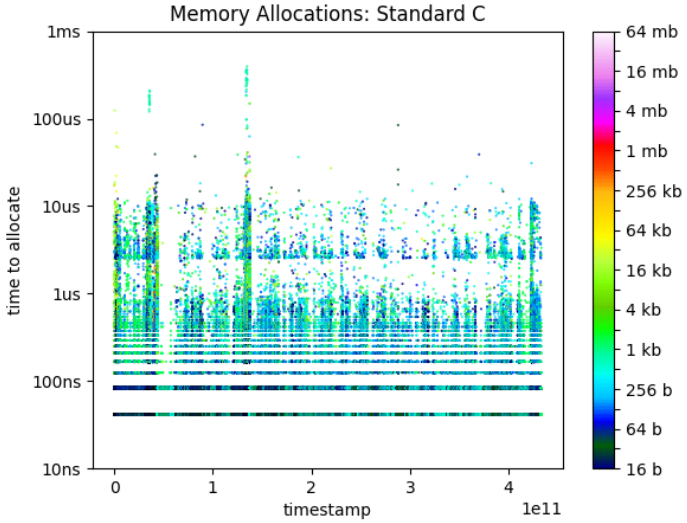

Figure 14: Standard C allocator

From these it seems that allocation and freeing have similar behaviour, Free Lists take an exceptional amount of time, and TLSF performs similarly to C's standard library, but with less variance. This can be further investigated with box and whisker plots.

From these distributions we can see that TLSF is incredibly consistent for allocations, and otherwise more or less matches C's standard library allocator. As seen in the initial graphs, First fit is an extremely slow allocator for a real time system.

5 Strategies

We could look at the results above and conclude something along the lines of: the standard library does a pretty good job. We could use a TLSF allocator if performance is critical and we have time to spend on debugging and tuning. Such a conclusion would be perfectly reasonable, but it skips a big question: do we need a dynamic memory allocator?

As mentioned previously, the problem with dynamic memory allocation is that it is dynamic. Hence, the solution to dynamic memory allocation could be not to allocate memory dynamically. Consider the distribution of sizes for all the allocation requests in the Doom 3 memory journal.

During its short runtime, the program has made a very large number of small allocations, most commonly 16 bytes. While a dynamic allocator can make this approach possible, it runs the very real risk of scattering data randomly in memory and needlessly driving up overheads on allocation

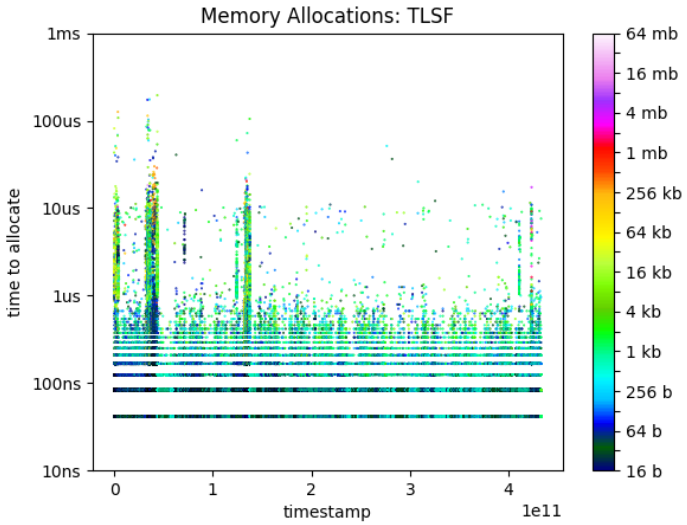

Figure 15: Two Level Segregated Fit allocator

calls. Before a fully dynamic allocation is made, there are a number of strategies which can be considered.
For short lived memory, in the order of a few kilobytes or less, stack allocation is viable. C's alloca and free can achieve this. Allocating on the stack ensures that the memory locations are fairly close together. Also since it involves incrementing the stack pointer rather than searching a data structure, it is fairly fast.

For objects such as linked lists, it can be viable to allocate some amount of backing memory to hold the nodes, reallocating if needed. Not only does this reduce the number of allocation calls, but it also ensures fast traversal. For small collections of objects, a pool or fixed size allocator may be beneficial.

Rather than individual objects, abstract collection objects can be used to group resources. These go by a few names such as "Managed Arrays", "Vectors", "Component Sets" or "Arena Allocators". In all cases the idea is the same, some object creates and manages backing memory. Upon resize, doubling the size of the backing memory further reduces the number of reallocations, at the expense of allocating more memory than needed.

Rethinking the way the program decomposes into subsystems, each subsystem can manage its backing memory. This also has the benefit of making a program's memory footprint more predictable.

6 Conclusion

In this project I have presented the problem of dynamic memory allocation, along with the related concepts and terminology. I've discussed some of the work which has been undertaken in the field,

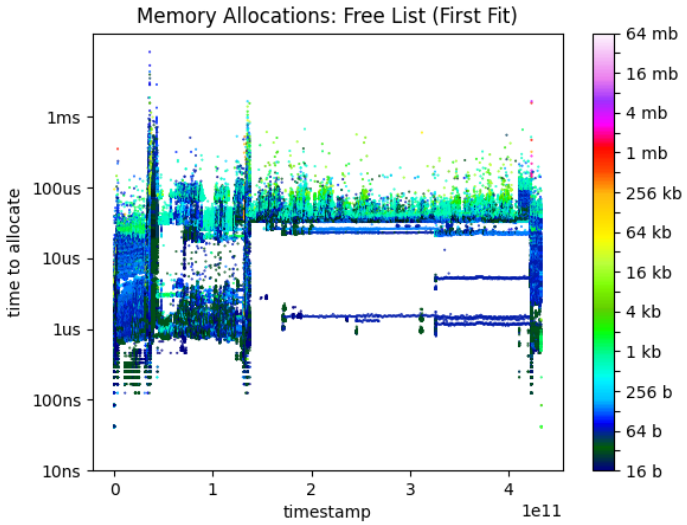

Figure 16: Free List allocator (first fit strategy)

and benchmarked some allocators in a practical scenario. I've also presented a number of recommendations for strategies to try before reaching for a dynamic memory allocator. As many other studies have likewise concluded, there is no universal solution to all memory allocation problems. In fact, many more benefits can be realised by designing systems so as to rely less on dynamic allocation. As a final, motivating example, here is a simple Python Raytracer I made in order to stress test my CPU.

Rendering one scene, with 2048 samples per ray, and every ray rebounding up to 2048 times takes 55 minutes on a Mac M1 Pro. Python does not have a concept of stack allocation, so each local variable is a small heap allocation, incurring all the associated pitfalls. Replacing local variables with a single list of floating point numbers, which is created once at the start of the program and then passed around in arguments as a sort of "scratchpad" reduces execution time to 45 minutes. It is not monumental, but it is enough to prove that memory matters.

References

[1] Data structure alignment. `https://en.wikipedia.org/wiki/Data_structure_alignment`, 2024. [accessed 02-11-24].

[2] Sarah Baldwin. Ada lovelace and the analytical engine. `https://blogs.bodleian.ox.ac.uk/adalovelace/2018/07/26/ada-lovelace-and-the-analytical-engine/`, 2018. [accessed 28-10-24].

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

[4] Michael C. Daconta. *C++ pointers and dynamic memory management*. Wiley-QED Publishing, USA, 1995.

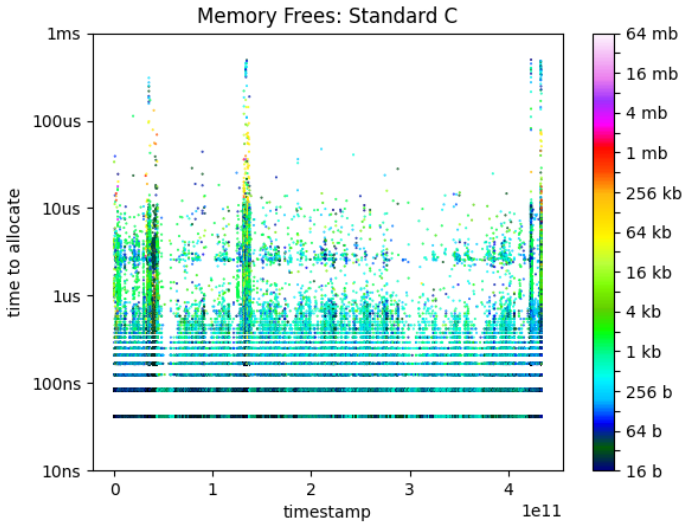

Figure 17: Standard C allocator

[5] 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.

[6] Jason Gregory. *Game Engine Architecture, Second Edition*. A. K. Peters, Ltd., USA, 2nd edition, 2014.

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

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

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

[10] Stephen Kaisler. *Second Generation Mainframes: The IBM 7000 Series*. 06 2019.

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

[12] Bharat Kumar Makhija. Learn how to elevate your system's performance with multi-level caching. `https://medium.com/@bharatmakhija/learn-how-to-elevate-your-systems-performance-with-multi-level-caching-511efb774a6e`, 2023. [accessed 28-10-24].

[13] 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.

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

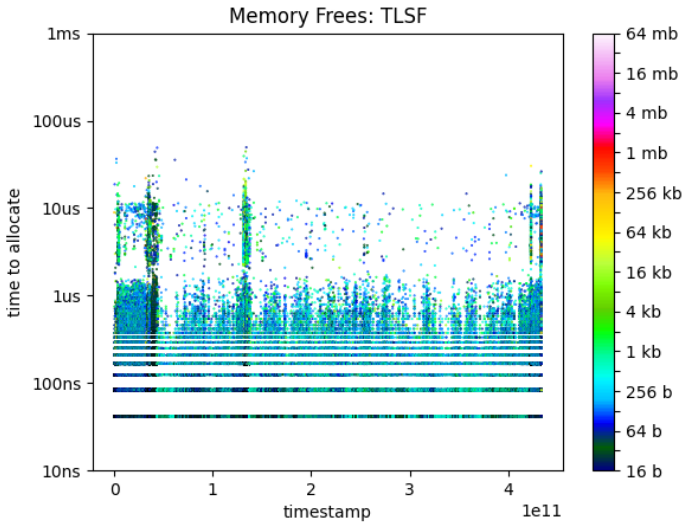

Figure 18: Two Level Segregated Fit allocator

[15] 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.

[16] Mindaugas Rasiukevicius. Tlsf: two-level segregated fit o(1) allocator. `https://github.com/rmind/tlsf/tree/master`, 2022. [accessed 12-10-24].

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

[18] Forest Smith. Benchmarking malloc with doom 3. `https://www.forrestthewoods.com/blog/benchmarking-malloc-with-doom3/`, 2022. [accessed 01-08-24].

[19] 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.

[20] Mariano Trebino. memory-allocators. `https://github.com/mtrebi/memory-allocators/tree/master`, 2020. [accessed 14-10-24].

[21] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, apr 1980.

[22] Jun Zhang. Stack frame and related operations. `https://www.researchgate.net/figure/Stack-frame-and-related-operations_fig1_329007575`, 2018. [accessed 30-10-24].

A Source Code for Journal Replay

```
1 #include "allocators/allocator.h"
2 #include "allocators/vanilla/vanilla.h"
3 #include "allocators/free_list/free_list.h"
```

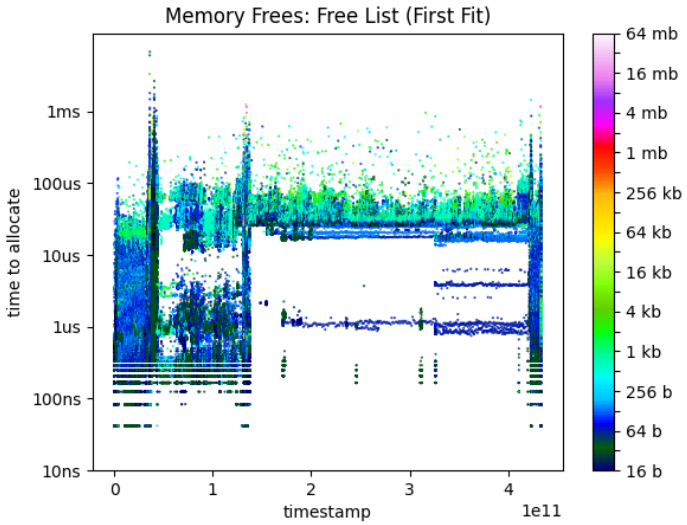

Figure 19: Free List allocator (first fit strategy)

```
4  #include "allocators/tlsf/tlsf.h"
5  #include <fstream>
6  #include <vector>
7  #include <unordered_map>
8  #include <cstdint>
9  #include <sstream>
10 #include <iostream>
11 #include <chrono>
12
13 std::vector<std::string> split(std::string line, std::string
       delimiter) {
14
15     std::vector<std::string> words;
16
17     size_t pos = 0;
18     std::string token;
19     while((pos = line.find(delimiter)) != std::string::npos) {
20         token = line.substr(0,pos);
21         words.push_back(token);
22         line.erase(0, pos + delimiter.length());
23     }
24     words.push_back(line);
25
26     return words;
27 }
28
29 void benchmark(Allocator& allocator) {
```

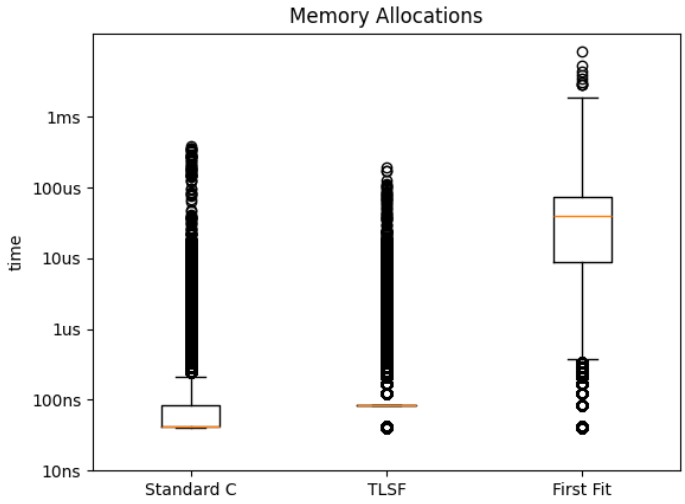

Figure 20: Memory Allocation times

```
30
31      std::cout << "----␣Benchmark␣" << allocator.name << "␣Allocator␣
            ----" << std::endl;
32      std::string line;
33      std::unordered_map<std::string, void*> allocations;
34      std::unordered_map<std::string, size_t> sizes;
35      size_t size;
36      std::string ptr;
37      std::string threadID;
38      std::string timestamp;
39      float malloc_average = 0;
40      float free_average = 0;
41
42      // pre read in order to allocate
43      std::ifstream journal;
44      journal.open("../data/doom3_journal.txt");
45      int64_t malloc_count = 0, free_count = 0;
46      while (std::getline(journal, line)) {
47
48          std::vector<std::string> words = split(line, "␣");
49
50          if (words[0].compare("a") == 0) {
51              malloc_count++;
52          }
53          if (words[0].compare("f") == 0) {
54              free_count++;
55          }
```

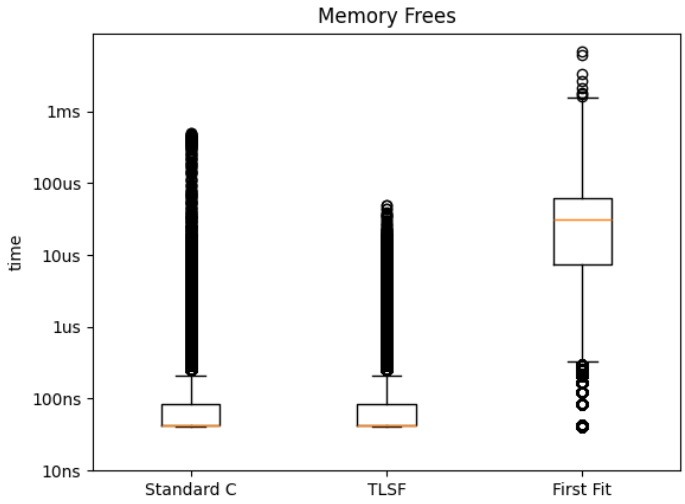

Figure 21: Memory Free times

```
56
57      }
58      journal.close();
59      journal.open("../data/doom3_journal.txt");
60
61      float malloc_coefficient = 1.0f / malloc_count;
62      float free_coefficient = 1.0f / free_count;
63
64      // x axis: timestamp
65      // y axis: duration
66      // color: size
67      std::vector<float> x_a(malloc_count);
68      std::vector<float> y_a(malloc_count);
69      std::vector<float> c_a(malloc_count);
70      std::vector<float> x_f(free_count);
71      std::vector<float> y_f(free_count);
72      std::vector<float> c_f(free_count);
73
74      while (std::getline(journal, line)) {
75          std::vector<std::string> words = split(line, "␣");
76
77          if (words[0].compare("a") == 0) {
78              //allocate
79              //a size ptr threadID timestamp
80              std::stringstream converter(words[1]);
81              converter >> size;
82              ptr = words[2];
```

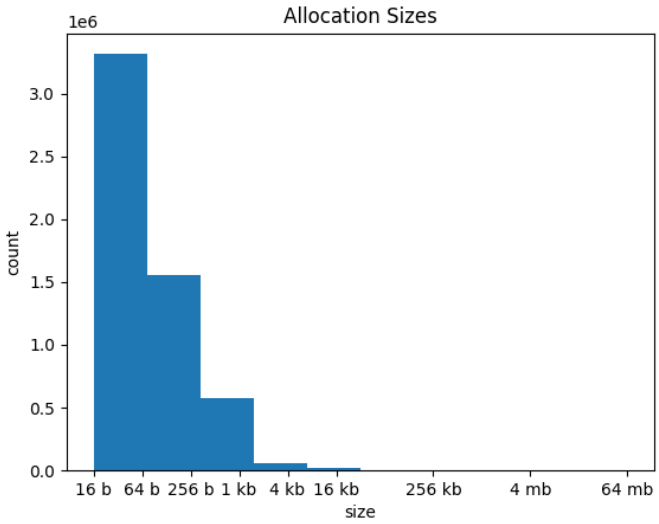

Figure 22: Distribution of requested sizes

```
83              threadID = words[3];
84              timestamp = words[4];
85
86              //---- time this ----//
87              auto start = std::chrono::high_resolution_clock::now();
88              void* data = allocator.custom_malloc(size, 16);
89              auto end = std::chrono::high_resolution_clock::now();
90              float duration = std::chrono::
91              duration_cast<std::chrono::nanoseconds>(end - start).
                    count();
92              malloc_average += malloc_coefficient * duration;
93              //--------------------//
94
95              //---- log ----//
96              x_a[i] = std::stof(timestamp);
97              y_a[i] = duration;
98              c_a[i++] = static_cast<float>(size);
99              //------------//
100
101             allocations[ptr] = data;
102             sizes[ptr] = size;
103
104         }
105
106         if (words[0].compare("f") == 0) {
107             //free
108             //f ptr threadID timestamp
```

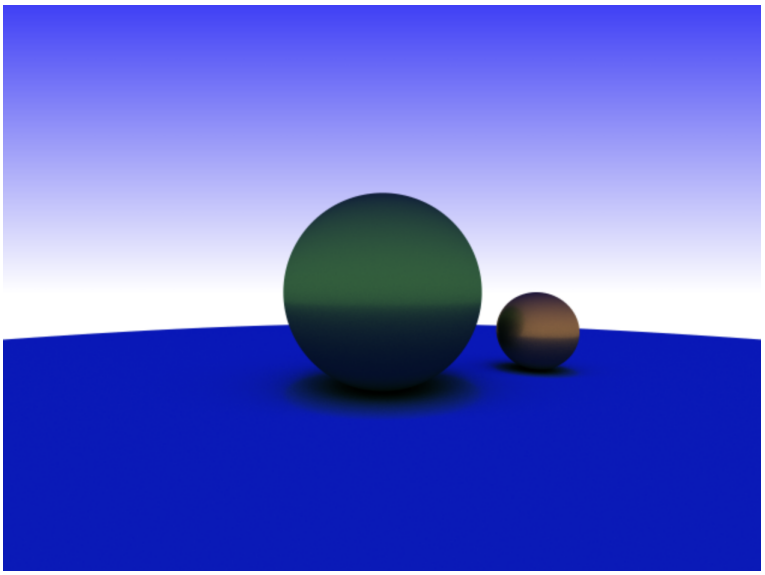

Figure 23: A CPU raytracer.

```
109             ptr = words[1];
110             threadID = words[2];
111             timestamp = words[3];
112 
113             void* data = allocations[ptr];
114             //std::cout << data << std::endl;
115             allocations.erase(ptr);
116             //---- time this ----//
117             auto start = std::chrono::high_resolution_clock::now();
118             if (data) {
119                 allocator.custom_free(data);
120             }
121             auto end = std::chrono::high_resolution_clock::now();
122             float duration = std::chrono::
123             duration_cast<std::chrono::nanoseconds>(end - start).
                    count();
124             free_average += free_coefficient * duration;
125             //--------------------//
126 
127             //---- log ----//
128             x_f[i2] = std::stof(timestamp);
129             y_f[i2] = duration;
130             c_f[i2++] = static_cast<float>(sizes[ptr]);
131             //------------//
132         }
133     }
134 
```

```
135     std::cout << "Allocation␣time␣(average):␣"
136         << malloc_average << "␣ns." << std::endl;
137     std::cout << "Free␣time␣(average):␣"
138         << free_average << "␣ns." << std::endl;
139 
140     // Write back results
141     std::ofstream file;
142     std::stringstream filename_builder;
143 
144     filename_builder << "../data/" << allocator.name << "_allocate.
            txt";
145     file.open(filename_builder.str(), std::ofstream::out);
146     for (size_t j = 0; j < malloc_count; ++j) {
147         file << x_a[j] << "␣" << y_a[j] << "␣" << c_a[j] << std::
                endl;
148     }
149     file.close();
150 
151     filename_builder.str("");
152     filename_builder << "../data/" << allocator.name << "_free.txt";
153     file.open(filename_builder.str(), std::ofstream::out);
154     for (size_t j = 0; j < free_count; ++j) {
155         file << x_f[j] << "␣" << y_f[j] << "␣" << c_f[j] << std::
                endl;
156     }
157     file.close();
158 }
159 
160 int main() {
161     //VanillaAllocator vanillaAllocator;
162     //TLSFAllocator tlsfAllocator;
163     FreeListAllocator bestFitList(FreeListAllocator::PlacementPolicy
            ::FIND_BEST);
164     benchmark(bestFitList);
165     return 0;
166 }
```

B Source Code for Standard Allocator

```
#pragma once
#include "../allocator.h"

class VanillaAllocator: public Allocator {
public:
    VanillaAllocator() {
        name = "Vanilla";
    }

    void* custom_malloc(size_t size, size_t alignment) {
        return aligned_alloc(alignment, size);
    }

    void* custom_realloc(size_t size, size_t alignment,
        void* oldAddress) {

        void* newAddress = aligned_alloc(alignment, size);
        memcpy(newAddress, oldAddress, size);
        free(oldAddress);
        return newAddress;
    }

    void custom_free(void* data) {
        free(data);
    }
};
```