A Survey of Heap Allocators in Multithreaded Processes

Rose Silver

April 28 2023

References

https://dl.acm.org/doi/abs/10.1145/356989.357000

https://www.bsdcan.org/2006/papers/jemalloc.pdf

http://supertech.csail.mit.edu/papers/Kuszmaul15.pdf

Introduction

Using memory in heap is crucial in a lot of programs, and thus allocators are crucial to the performance of a lot of processes. It becomes even more crucial when you have multiple threads. While the operating systems include their own functions for grabbing memory (mmap, for example), mmap is very slow. Allocators, on the other hand, have the opportunity to be smart and responsible with memory. However, since the rise of multithreaded programs, the allocator has been noted to be a bottleneck to performance. This has thus inspired a lot of work to improve allocators, especially in the implementation of multithreaded programs. In this paper, I analyze three different allocators–Hoard, JEmalloc, and Supermalloc. I detail the bottlenecks of allocators in more detail, as well as overview how each allocator is designed to tackle these bottlenecks. The paper is focused mostly on design principles, data structures, and algorithms, although for more information on experimental results, one can consult the papers. In the next section, I introduce preliminaries, then, I discuss each allocator individually, and then finally, I compare and contrast all of the allocators.

Preliminaries

Let's review the basic concepts needed to understand how memory allocation works.

Dynamic Memory Allocation

Recall the memory layout of a process in RAM. In the memory layout, there is a section for data which includes sections for data used during the program. The important section of the memory layout for this paper is the dynamic data section, often referred to as the "heap". The heap is where a process's global variables are stored. At runtime, a process is allowed to both initialize global variables in the dynamic data section ("on the heap") and remove them. Both of these actions are known as dynamic memory allocations, or simply memory allocations.

One way a user can request pages of memory is via the system call mmap(). The function mmap() takes a number of arguments, such as the address for the data to be stored, the size of the data to be stored, protections, flags, etc. The mmap() function does not do anything very clever; it simply finds a segment in memory that satisfies the requests in the argument. However, mmap has several drawbacks. For example, mmap() can be quite slow, and even if it was fast, it requires a lot of decisions from the user. It is also inefficient for allocating objects that are very small, much smaller than a page.

There is another way to request memory, via a memory allocator. In C, the user can interact with the memory allocator via the functions malloc() and free(). The function malloc(k) will request k bytes of memory on the heap and return a pointer, ptr, to the address of the memory. Once a piece of memory is allocated, it cannot be deallocated (or moved). If a user wants to remove this memory, they can use the function "free(ptr)" to free this memory. There are a few points that one can observe at this point. Note that free(ptr) does not indicate the size of the memory pointed at by ptr, so the allocator needs to determine that on its own. Under the hood, the memory allocator can interact with the operating system using mmap, but allocators are generally a lot more clever than this.

Allocator Objectives

A good allocator should strive to do/have the following:

Minimize memory usage. An allocator should strive to use as little memory as possible. In particular, the amount of memory requested from the operating system should not be too much bigger than the memory that the user needs to use. To understand memory usage, it's important to discuss memory paging and virtualization. Memory paging is when the computer stores pages in disk and tries to use it in RAM. It may use more memory than fits in RAM, and be forced to resort to using swap space. The primary function of swap space is to substitute disk space for RAM memory when real RAM fills up and more space is needed. The kernel uses a memory management program that detects blocks, aka pages, of memory in which the contents have not been used recently. The memory management program swaps enough of these relatively infrequently used pages of memory out to a special partition on the hard drive specifically designated for "paging," or swapping. This frees up RAM and makes room for

more data to be entered into your spreadsheet. Those pages of memory swapped out to the hard drive are tracked by the kernel's memory management code and can be paged back into RAM if they are needed.

Utilize virtualization. Virtualization is a very useful tool for allocators to save on memory. The operating system has the ability to create new virtual pages in the page table and to map them to arbitrary positions in physical memory (so the operating system doesn't experience any fragmentation at the page-level granularity). With virtualization, memory could be accounted for without having to touch any physical pages in RAM. Interestingly, the operating system can create a virtual page in the page table and lazily wait until later to allocate that page to physical memory (wait until it gets its first write)—one of the allocators that will be studied here ends up making use of this (Supermalloc).

Efficiency. It's important that allocators do not become a time bottleneck to a running process. The hope is that an allocator should be orders of magnitude more efficient than calling mmap(). The tricky thing is guaranteeing that concurrent allocators run about as fast as their state-of-the-art serial allocators.

Good spatial locality. Allocators should want to maintain good data locality. Spatial locality is the probability of accessing some data (or a storage location) soon after some nearby data (or a storage location) on the same medium has been accessed. This is related to false sharing, which will be discussed later. One wants each thread's allocations to be in a local region specific to that thread. This is very helpful for caching.

Core Problems Allocators Face

Before continuing to the next section, it's important to understand some of the core problems that allocator designers are up against.

Fragmentation and memory overhead. Fragmentation, a phenomenon when memory is not utilized efficiently, affects both serial and concurrent allocators. In particular, a chunk of memory could have multiple unused fragments that are too small to put any objects in. There are two kinds of fragmentation: internal fragmentation and external fragmentation. Internal fragmentation is space wastage caused by padding an object to a larger size. External fragmentation is space wastage due to unallocated space between objects. One famous result from the theory literature, which does not appear to be covered in these papers, is that fragmentation is unavoidable in the sense that there exists a workload that forces any allocator to allocate t bytes of space using $\Theta(t \log t)$ memory. Finally, related to fragmentation is space consumption of metadata: the allocator needs data structures to keep track of where the free space is, how big each allocator is, etc. This metadata itself can be space inefficient, especially if the allocations in the system are small.

The following problems are central to concurrent allocators.

Memory blowup. One common thing that allocators do is allocate on a per

processor level. But then if processor x allocates, and processor y deallocates, this can cause an issue where the memory never actually gets freed. In some implementations of malloc (not the ones here) this can cause an asymptotic blowup in memory usage as a function of the number of processors. Unbounded blowup refers to when memory consumption grows without bound while the memory required is fixed. Better than unbounded blowup is blowup that scales with P, the number of processors, but unbounded blowup is the worse of two evils.

Synchronization costs and contention. When talking about concurrency, one cares a lot about synchronization costs. Synchronization costs refer to the overhead or additional resources required to coordinate and synchronize the actions of different processes or threads in a computer system. When multiple threads or processes are running concurrently and need to access shared resources, such as memory or I/O devices, they must coordinate their actions to ensure that they do not interfere with each other. Synchronization costs can arise due to various factors, such as the need for locking mechanisms to ensure mutual exclusion and prevent race conditions, the overhead of context switching between threads or processes, the cost of communication and message passing between processes or threads, and the delay introduced by waiting for synchronization primitives like semaphores, monitors, or condition variables. In general, minimizing synchronization costs is important for improving the overall performance and scalability of multi-threaded or multi-process systems. However, achieving efficient synchronization requires careful design and implementation, as excessive synchronization can lead to contention and performance degradation, while inadequate synchronization can result in data inconsistency and program errors.

False Sharing. Lastly, it's important that allocators don't induce false sharing of cache lines. False sharing of cache lines is a performance issue that can occur in multi-threaded programs when two or more threads access different variables that happen to be located on the same cache line in memory. A cache line is a fixed-size block of memory that is loaded into the processor's cache when a variable is accessed. If two or more threads access different variables that happen to share the same cache line, each time one thread modifies its variable, it will cause the cache line to be invalidated or updated, and the other thread(s) will have to reload the cache line from memory, even though they didn't modify the shared variable. This can result in unnecessary cache thrashing and reduced performance, especially if the affected cache line is frequently accessed by different threads. False sharing of cache lines is called "false" because it is not caused by the threads actually sharing the same variable, but by the way the variables are laid out in memory and how they are mapped to cache lines. To avoid false sharing of cache lines, allocators can pad the variables to ensure that they are located on separate cache lines, using thread-local variables instead of shared variables where possible, or using specialized libraries or tools that can detect and mitigate false sharing issues. The problem is that padding induces an unacceptable amount of internal fragmentation.

Hoard

The Algorithm

In this section, I will describe the Hoard allocator and demonstrate how this allocator is able to (mostly) solve all of the four problems above.

Before an allocator can begin allocating, it must establish some tools. At the highest level, every processor gets a bundle of its own per-processor heap, and then every processor shares a global heap. In particular, Hoard allocates $2P + 1$ heaps, where P is the number of processors. Heap 0 is the global heap, then heaps $1, \ldots, 2P$ are for the processors (every processor gets two heaps). Every thread is then assigned its own thread-local heap. Note that, in general, the number of threads is not equal to the number of processors. Thus, Hoard uses a hash function to assign every thread its own heap in an attempt to avoid multiple threads sharing a single heap. Going a level deeper, every heap contains two kinds of content: superblocks and statistics. A superblock is a fixed-sized chunk of memory (of size S). One can think of S as being a multiple of a page size, and one can think of each heap as having many superblocks. Note that these superblocks are given to Hoard by the operating system via system calls such as mmap. Furthermore, a superblock can be divided up into smaller chunks called blocks, whose sizes are constant within each superblock (but may be different between different superblocks, more on this later). In addition to a set of superblocks, each heap has two heap statistics: u_i, the amount of memory actively in use, and a_i, the total amount of memory available in the heap.

Hoard uses these tools to allocate memory in the following way. When a thread calls malloc, Hoard has two properties it must maintain at any time. The first property is that a thread can only allocate memory from the heap to which it was prescribed. This means that, if a thread calls malloc, but the thread's local heap has run out of free memory, then Hoard will bring in a superblock from the global heap to the thread local heap to fulfill the thread's request. If the global heap does not have any superblocks to donate to the thread's local heap (i.e. the global heap has run out of memory), then at this point Hoard will simply ask the operating system to fetch a superblock of memory. The second property is that each object is allocated into the best-fitting block in the thread-local memory, i.e. the minimum-sized block such that it has space to fit the request, in order to reduce internal fragmentation.

I want to dive deeper here in how the second property is implemented. It's first important to note that objects are partitioned into size classes. Within each heap, each size class has its own designated set of superblocks. For a set of superblocks within the same size class, the super blocks are partitioned into fullness groups, where the superblocks in a given fullness group have similar fullnesses to each other. Each fullness group is then stored as a linked list. This makes it so that in $O(1)$ time one can (1) tell whether a free causes a super block to become less than an f-fraction full; and (2) find the fullest superblock out of

those available. The reason that one would want to find the fullest super block out of those available is that, heuristically, this strategy allows for less-full super blocks to eventually empty out. The point of this strategy is to mitigate external fragmentation, by hopefully allowing for the entirety of some superblocks to be freed, so that those superblocks can be repurposed. A few things to note is that, if the thread wants to allocate a really large item (larger than $S/2$), then Hoard will call mmap directly. Also, the superblocks contain metadata at the beginning of each superblock that keep track of the size of the objects in the superblock.

I now describe how the allocator uses these tools to perform a free. When a thread calls free, the object's block within the superblock is freed. If the corresponding superblock after the free is nearly empty, then the superblock will be donated to the global heap. In particular, if a local heap is using less than an f-fraction of available memory in its heap (f is a parameter of the system) and also the available memory in the heap is larger than KS for some K (also a parameter of the system), Hoard will donate an f-empty superblock from the heap to the global heap. Hoard can find f-empty superblocks in constant time using the concept of "fullness groups".

Analysis

I am now going to take a step back to talk about how this strategy addresses the four problems addressed above. The first observation is that, when Hoard donates superblocks from the local heaps to the global heaps, this reduces blowup. If Hoard did not do this, then consider the malicious allocation pattern where one thread only allocates blocks of memory while a different thread only frees. Since the first thread can only use memory within its own heap, it has to consistently ask for more memory. If a global heap did not exist, then Hoard would have to fetch more memory from the system, even though the second thread was sitting on a large batch of empty memory. As a result, the blowup is decreased. The second observation is that this strategy reduces false sharing. Note that, the only way any active/passive false sharing can occur is if a thread allocates memory to a superblock, the superblock gets passed to the global heap (because there has been a free or the block is just really empty), and then the superblock gets passed from the global heap to another thread's heap, and then that thread uses that superblock. However, this should seldom happen due to the allocate-fullest strategy. Finally, note that this algorithm reduces external fragmentation. When Hoard partitions elements into size classes, and then uses the fullest available superblock within a given size class (and within a given thread-local heap), the probability that random accesses hit the same page is increased, and this also is great for speed as this means that there is less pages, and thus a higher chance of using the TLB instead of having to look in the page table.

The authors prove a theoretical result bounding their memory blowup as a function of the number of processors. Let $U(t)$ denote the maximum amount of memory allocated and in use by the program after memory operation t. Theorem 1: blowup is linear in the number P of processors. If $P << U(t)$, then blowup is

constant.

Unfortunately, this theorem is impossible, due to a lower bound that was proven in Robson's 1971 paper "An Estimate of the Store Size Necessary for Dynamic Storage Allocation", which shows that any allocator must incur $\Omega(\log n)$ blowup, where n is the size of memory. The theorem is true for each individual size class, but is false across all the size classes. The authors dodge this issue by claiming that the number of size classes is a constant. But it is actually logarithmic, and by ignoring this the authors are able to ignore a logarithmic factor in their space blowup.

Finally, it is worth noting that in terms of handling scalability issues, there are two important ideas that they use: Idea 1: Grabbing a lock for a per-processor heap should scale well because these should be low contention. Idea 2: Grabbing a lock for the global heap does have scalability concerns, but the event seldom happens, and that consequently should also typically have low contention.

JEmalloc

A Brief Overview

JEmalloc is a memory allocator that was introduced to the world six years after the birth of Hoard. In some ways, it can be thought of as a version of Hoard that is more technical. The advancements that JEmalloc made on Hoard came from a few interesting insights about allocators that the author, Jason Evans, had. The first insight is that, in practice, most memory allocations are being performed on small objects. Having an efficient memory allocator for small objects is not only important due to the sheer number of small objects, but also because the overhead incurred is a much larger ratio compared to the amount of energy it takes to write the object. The second insight that Evans had is that most users care about the performance of the memory allocators over long periods of time. It was thus important to Evans to improve allocators in such a way that their memory footprint over time is minimized. In particular, JEmalloc first tries to minimize memory usage, and tries to allocate contiguously only when it doesn't conflict with the first goal. The hope was also that, by making this the primary goal, other secondary and tertiary goals would fall into place; one can begin to see how this helps with other goals such as increasing data locality, reducing lock contention, reducing false sharing, and reducing fragmentation.

Evans was interested in observing the empirical performance of his allocator. He claims that it is actually difficult to measure allocation performance empirically, and thus it's hard to tell how well an allocator does in practice. For example, there could be pathological allocation patterns that cause the allocator to perform poorly, but it's not easy to find such patterns in practice. JEmalloc tries to handle this by doing various experiments.

Data Structures and Algorithms

In this section, I will define a few data structures and concepts used by the JEmalloc algorithm. I hope to peel back the layers of the data structures, depicting first the larger structures at play and then the smaller ones.

At a high level, just like Hoard, JEmalloc will divide the landscape into arenas, where each thread is assigned to an arena. It's worth mentioning now that, while Hoard uses a hash function to assign threads to arenas, JEmalloc uses a round-robin algorithm to assign threads to arenas, and these assignments are made at runtime (during a thread's first malloc or free). This requires that each thread has its own thread-local storage where it can keep track of which arena it is assigned to, but luckily most systems include thread-local storage. Every arena contains chunks (the superblocks of JEmalloc), which are our next objects of study.

The next smaller level has to do with the chunks in an arena. A chunk is a contiguous 2MB piece of memory. All chunks are dished out by the operating system via system calls like mmap; any chunk-sized memory allocated by the operating system will be chunk-aligned and aligned with the 0 address. Note that the large size of the chunk implies that it contains multiple contiguous pages. There are a lot of similarities between the chunks of JEmalloc and the superblocks of Hoard. Every arena gets a set of chunks, and these chunks can be passed around to the global arena or other thread's arenas. One additional thing that a chunk has is metadata to keep track of the information within the chunk; while superblocks only hold objects of one class size, a chunk can contain multiple different size classes. While multiple size classes per chunk makes chunks more complicated, this process is simplified due to what is known as page runs.

At the next smaller level are page runs. A run is a set of consecutive pages within a chunk. Runs are one of the concepts that massively separates Hoard from JEmalloc. Runs are carved into chunks via the buddy algorithm. This means that a chunk is halved into two "buddy" runs, and then each buddy is halved again and again, until the run is the appropriate size. When a "buddy" is freed (and the conditions are right) it can meld back together with its buddy into a larger piece again. Turning our attention back to chunks for a second, note that the metadata at the beginning of the chunk is to keep track of these runs. The smallest a run could be is just a few pages. Every run also has a bitmap to keep track of the free regions within the run.

In order to make sense of the run, one has to understand what size classes are in the context of JEmalloc. Size classes are just the allowable units of memory partitioned by the allocator. There are dozens of size classes, and each can be categorized as small, large, or huge. When memory of size m is malloc-ed by the program, JEmalloc will ignore the value m and instead pretend the user wrote m', where m' m rounded up to the nearest size class. A size class categorized as small is anything smaller than a page. A size class categorized as large is one

that is a few pages large. Finally, a size class categorized as huge is going to be memory on the order of the chunk size.

A basic principle for JEmalloc is that each run can only contain one class size, but a chunk (containing multiple runs) can contain multiple class sizes. Every chunk has its own local red-black tree for the runs in the chunk. These red-black trees are somewhat complicated, and one of the things that supermalloc will do is replace them with simpler data structures (but more on that later). Then every run has a red-black tree to keep track of the free regions in the run.

Putting the Pieces Together

I will now describe how all of these pieces (arenas, chunks, runs, class sizes, and metadata) fit together. When a thread calls malloc(m), the allocator assigns a thread an arena if it has not done so already. Then, the allocator rounds the object of size m up to the nearest size class m'. If the object is assigned a huge size class, then the allocator hands off this allocation to the operating system. The allocator manages this allocation using a red-black tree, a designated data structure just for huge allocations. Recall that all memory is handled in chunk sizes, so ultimately the huge object will be assigned chunks. In particular, it is assigned contiguous chunks. If the object instead falls into the large or small size class, then the object is assigned to one chunk. Each chunk has metadata about the runs and the size class of each run. If there is a "current" run with the object's size class, and this object can fit in the run, then the object goes in the first available region in this run. If there are multiple "current" runs with the object's size class, the object will go into the fullest of the runs. If there is a "current" run with the object's size class, but the object can't fit in the run, then another run is used (although the paper does not specify its behavior). If a run is emptied, and its "buddy" is also emptied, these can be globbed together to form a bigger run. If a chunk is emptied, it will be returned to the global arena. Some loose ends not described in the paper include: what happens if there are multiple chunks that JEmalloc can choose to allocate to. For example, if there is no "current" run in any chunk what happens? If there is a "current" run in every chunk, which do you choose?

Further Analysis

It's clearer to see that when threads are assigned to arenas in a round robin fashion, instead of by hashing as in Hoard, that the arenas are assigned the same number of threads, things are just spread out more equally. Also hashing is difficult. It's also noted that one could dynamically rebalance but this seems like there would be a lot of overhead to manage this.

The two important features of JEmalloc that differ from Hoard are as follows. First, the allocator tries to always use the "current" run if it can, which is good both as a method to try to reduce external fragmentation (it gives other runs some time to empty out) and also as a method to create good locality (items that

are allocated at nearby points in time are hopefully given nearby allocations). Second, the allocator has many arenas in order to avoid lock contention, and allocates threads to these arenas in a very low-overhead fashion (using round robin instead of hashing).

Note that the original version of JEmalloc used the binary buddy system to perform allocations for smaller objects. This means that memory is split into a binary tree, and whenever an object is allocated memory, it is given a full (power-of-two-size) subtree. The buddy system has the advantage that it can be implemented using very only 1 bit of metadata per leaf of the tree. Basically, the important thing is that if two sibling subtrees ever both become free, then they get merged into being a single larger free subtree. These types of merges ensure that as frees occur, the freed memory eventually gets chunked back together into (roughly) the largest chunks that it can be in. The binary buddy system worked well for small object because the objects were typically power-of-two sized.

The binary buddy system appears to have been removed in later versions (at least if I am understanding the supermalloc paper correctly), and replaced with a red-black tree that keeps track of where the freed memory is. Previously, the red-black tree was used for huge allocations, but now I believe it is used for all allocations.

Drawbacks: "Evans found that, al- though individual groups of equal-sized regions might ben- efit from fullest fit, chunks as a whole did not drain as well because a single "fullest" page could keep a chunk mapped indefinitely"

Supermalloc

The last allocator of focus is Supermalloc, introduced by Kuszmaul in 2015. Supermalloc is designed to be an allocator for x86 hardware transactional memory on a 64-bit machine. The biggest principle about 64-bit machines that drives the design of supermalloc is that, while physical memory is precious, virtual memory is abundant. This insight drives supermalloc to use virtual memory lavishly while relying on the fact that not all of virtual memory will be committed to physical memory at all times. This paper aims to give an overview of the design principles of the algorithm, but one can read the paper more to understand the full scope and all corner cases.

Data Structures and Algorithms

Like the other mallocs, Supermalloc allocates chunks of fixed size, and all memory requests to the system are processed in multiples of the chunk size. However, unlike mallocs like JEmalloc, Supermalloc will dedicate a chunk to only one size of objects. Multiple objects of (the same) smaller size will fit into one chunk, while larger objects will use multiple contiguous chunks. The allocator defers allocations of very large (huge) size to the operating system using mmap. Similar to JEmalloc and Hoard, supermalloc has size classes, but these are called small,

medium, large, and huge. For every object size, there is a corresponding bin number which makes lookups into tables a lot easier.

Every chunk has metadata stored at the very beginning. This metadata includes a heap, which sorts pages by how in-use they are at any moment, and a bitmap, which, for each object frame, stores a 1 or 0 for whether that frame is storing an object or not. To select which page to use for a given allocation in a given size class, supermalloc uses a fullest-fit algorithm within each size class, assigning objects to the fullest page. Thus, supermalloc needs to keep track of how full each page is. There are also larger data structures to organize the chunks. There is a large chunk table, where the entries are the bin numbers, and the value of each entry is the head of a linked list storing chunks with free space that can store objects with that bin number. While JEmalloc uses a tree to keep track of the free spots within chunks, supermalloc just uses an array of bits to keep track of the free spots within each page. A key observation here is that the bit array for a given page can be traversed super quickly because one can look at 64 bits at a time. So the overhead of having a red black tree is (arguably) not necessary.

Within the non-huge size classes, supermalloc does fine-grained size classes. One of the tricks it does in some size classes is to round objects to have sizes that are a prime number of cache lines. This avoids associativity conflicts in the cache. There is a problem where, perhaps these medium-sized objects are not multiples of the page size. In this case, supermalloc introduces folios, which are a group of pages together. These folios act as a unit. This is done to reduce internal fragmentation in this setting and to make uncomitting easier. Finally, there are large objects. Large objects can range from four pages to half the size of a chunk.

One interesting optimization that supermalloc makes is that it allocates memory in very large chunks (2MB) from the OS, and then gradually brings the memory from that chunk into use. The insight is that, even after the OS has allocated a 2MB virtual chunk of memory, the individual pages don't actually get created in the page table until they get touched, so the 2MB chunk allocations do not actually create bad external fragmentation. One final note to make is that Supermalloc is optimized to have small critical sections so that transactions will be able to most likely succeed without concurrency issues causing them to abort.

Comparison of Experiments

All of the allocators evaluate both time and space efficiency. Hoard designs microbenchmarks that are intended to induce active and passive false sharing. This allows it to demonstrate how well it avoids false sharing issues. Everyone claims to be faster and better than the prior work.

Although JEmalloc does not compare itself to Hoard, supermalloc compares itself to both of them. Supermalloc can in some cases be 2-4 times faster than Hoarde and JEmalloc. On the other hand, neither Hoard nor JEmalloc seems to

be strictly better than the other.

One drawback of supermalloc is that it does not try to solve false sharing and it does not replicate Hoard's microbenchmarks intended to evaluate false sharing. This would be an interesting direction for future work.

Conclusion

One question to be investigated is, what are some different problems the allocators try to solve? The next is, what are the strengths and weaknesses of each allocator? In this paper, I also investigate the similarities and differences in the data structures. Here is a sneak peak of some of those differences: Hoard and JEmalloc both use multiple arenas for allocation; Hoard never returns memory back to OS but JEmalloc does; All use O/S memory for large allocations; Hoard cares about false sharing for small objects. JEmalloc and supermalloc, not so much. Their stance is that, if a user really cares about this then it is their responsibility to make sure that they implement and allocate responsibly. "This decision stands in contrast to allocators, such as Hoard, that try to use temporal locality to induce spatial locality"; Supermalloc uses hardware transactional memory instead of locks.; Jemalloc uses red-black trees all over the place, but supermalloc argues that you can often just replace those with some well-designed bit maps.