VMem

By Stewart Lynch.

Contents

Introduction

The goal of VMem was to create an allocator that reduced fragmentation and performed well in a multi-threaded environment without compromising speed or efficiency. VMem had to be a direct replacement of the standard malloc.

VMem achieves these goals using a number of techniques, specifically related to aggressive de-committing of virtual memory, biasing, and careful placement of spinlocks. These will all be discussed in detail.

The kind of environment that VMem was designed for is real-time applications such as games. The typical load of this sort of system can be tens of thousands of allocations per second and a runtime of many hours. Fragmentation is often a real issue in these complex applications and so time spent reducing fragmentation in an allocator can be of real benefit.

Memory corruptions are probably the hardest type of bug to track down. VMem has a significant amount of internal checking that often catches these bugs close to the source of the problem. This checking has a negligible overhead in both speed and memory. Additionally, VMem has more heavy weight checking that can be enabled when needed, that will catch more problems.

VMem has detailed stats reporting. Each heap is broken down into used, unused, overhead, committed, reserved. This allows you to keep an eye on sizes by allocation, VMem's internal overhead and fragmentation.

Credentials

I have been writing malloc replacement allocators since 2001 for large and complex AAA console and PC games. VMem is a culmination of many years of experience in writing allocators and fixing fragmentation and memory corruption bugs. VMem has so far out performed all other allocators that have been tried in terms of speed, overhead and fragmentation. Detailed analysis and comparisons have been made but due to intellectual property rights these cannot be shared here. The best way to evaluate an allocator is to drop it into your app and profile the results. Please see the website for a list of applications that have made use of VMem.

Overview

VMem is similar in structure to most malloc replacements. Its main heap consists of multiple sub heaps each dealing with a different size of allocation. VMem has two sorts of heaps, the fixed size heap (FSA) and the coalesce heap.

The FSA Heap

A fixed size allocator (FSA) can allocate only one size of allocation which is setup at creation time. The FSA allocates a page of memory, partitions it up into allocation slots of the desired size, and links them all together into a free list. Allocating is as simple as removing from the free list.

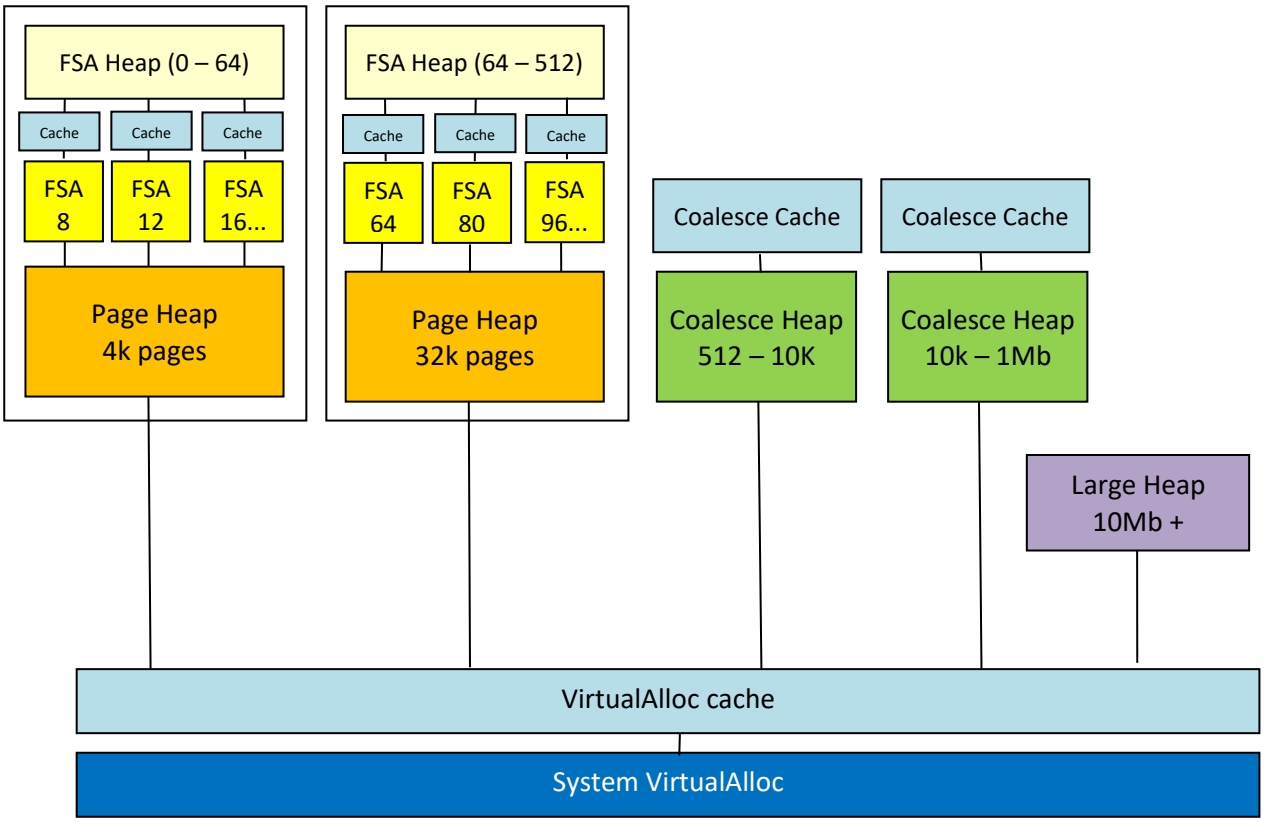
An FSA Heap is a collection of FSA's. Typically there will be an FSA for each allocation size from 0 – 512 bytes.

The Coalesce Heap

The coalesce heap can hold allocations of varying size. Each block in memory points to the next free or allocated block of memory. When a block is freed it attempts to merge (or coalesce) with any joining free blocks. When allocating the coalesce heap much find the best free block to fit the allocation.

VMem also has a large heap for allocations that go directly to the OS VirtualAlloc. It keeps a list of all large allocations so that it can tell if it allocated an allocation.

These heaps can be combined in any way to for the main heap, and they can also be used separately for individual heaps. VMem contains a default heap VMemHeap that is well balanced for a standard malloc replacement. The default setup is to have 2 FSA heaps, and 2 coalesce heaps and one large heap. The reason for having two FSA heaps is that smaller allocations do better with smaller page sizes. The reason for two coalesce heaps is that segregating allocations of different sizes helps reduce fragmentation.



Additionally VMem has a basic FSA and a basic coalesce allocator. These allocators get the memory directly from the system and have no internal overhead. Overhead for the other allocators and heaps is allocated using these basic allocators.

So far it's all pretty standard, most allocators have a similar structure. Next I'll discuss the concepts behind what makes VMem different. How is reduces fragmentation deals with multi-threading.

Getting Started

The simplest way to use VMem is to simply include VMemNew.hpp into the main cpp file of your app. This will override the new and delete operators to use the default VMemHeap. In more cases this is all you need to do.

If you need to change the values of the default allocator it is recommended that you create a new heap class in your app. Simply take a copy of the VMemHeap.cpp, rename it, override new/delete , and tweak it as desired. This will make integrating any updates to VMem much easier.

You can have multiple heaps setup in any way that is needed, use VMemHeap as a template. You can also use the allocators directly in your code as an alternative to using new/delete. The FSA allocator is particularly useful for cases where you need lots of one type of object and don't want to fragment the main heap. Create a local FSA object and call Alloc and Free on that directly.

VMemDefs.hpp is where all the defines live. This where the integrity checking options are enabled and disabled. It's where the system page sizes is set. And it defines all of the different guard values.

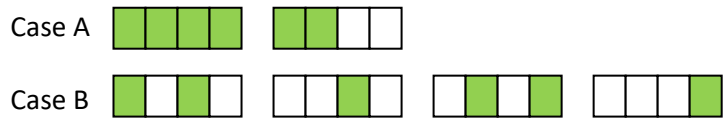
VMemSys.cpp defines all system functions such as VirtualAlloc that VMem uses. This is currently only implemented for Win32. If you wish to use VMem on a different platform this file is the only file in VMem that you should need to modify.

Fragmentation

Before discussing VMem in detail we need to define what is meant by fragmentation. Fragmentation in the most general sense is a measure of the memory that is committed but not in use by app. It is a measure of wasted memory.

There are two different sorts of fragmentation. One sort of fragmentation will be termed temporal and the other spatial.

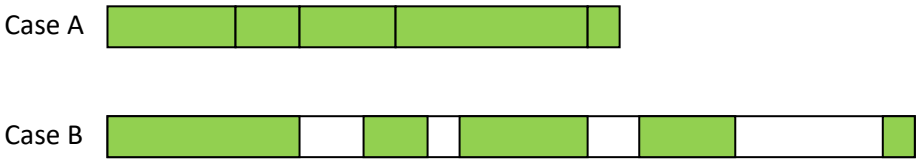
Temporal fragmentation typically happens in a fixed size allocator. The allocator allocates pages of memory and divides them up into equally sizes slots. In the diagram below both case A and B have the same number of slots allocated, but because case B is more fragmented it is using up twice the number of pages.



The important thing to note about temporal fragmentation is that the wasted space can still be used. If we do lots of allocations and fill in all of the empty slots we eliminate the fragmentation. The fragmentation is caused by a few allocations in each page having a longer lifetime and stopping the page from being freed. The fragmentation can be temporary and not necessarily a cause for concern. The whole reason for these style of allocators is that they eliminate spatial fragmentation which is a much more insidious problem.

Spatial fragmentation typically happens in coalesce heaps. When a coalesce heap starts to fill up each allocation is adjacent to the previous allocation and there are no gaps, no fragmentation. After many allocs and frees, due to allocations not always fitting exactly into free blocks small gaps start to appear. These gaps are often too small to be re-used and the best we can hope for is that they will eventually be coalesced into larger blocks. Coalesce heaps suffer from temporal fragmentation as well as spatial fragmentation.

Again we have case A that has no fragmentation, and case B is what the same heap may look like after many frees and allocs. They both have the same amount of memory allocated but case B uses up much more system memory.



VMem has techniques that reduce both sorts of fragmentation. These include biasing, aggressive decommitting and immediate coalesce. These are discussed in detail in later sections.

Virtual Regions

Memory managers typically allocate memory from the system in large chunks and then divide it up into smaller chunks for the individual allocators. In VMem these are called Regions. When freeing an allocation it is very quick to find out which heap it belongs by checking which region contains the address. Typical memory managers have large regions, and this can lead to a lot of wastage.

In some memory managers fixed size allocators get their memory from the system one page at a time. While this gets around the wastage due to large regions it can lead to very bad virtual address space fragmentation. I've seen apps that mix a garbage collected language that allocates in 32Mb regions, with an allocator that allocates single pages at a time, and the result was not pretty.

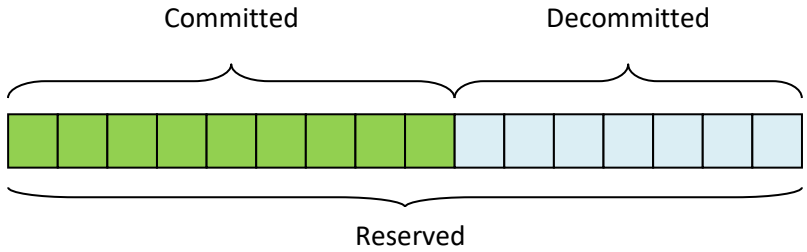
Where VMem differs is that when regions are created the memory is only *reserved*, nothing is committed. A region only commits pages as they are needed, and de-commits then when they are not needed. This means that a region only uses up as much physical memory as it currently needs.

This lead to a self balancing system, heaps can dynamically resize themselves as demand changes. It also helps reduce fragmentation, any holes that appear, assuming they are big enough, can be decommitted back to the system. This, combined with Biasing which is discussed later, radically reduce memory lost to fragmentation and overhead.

When freeing an allocation we want to find out which heap it came from as quickly as possible. Ideally we make the region size big enough that there will only be one region per heap. This depends on the availability of virtual memory. If virtual memory is limited, regions can be made smaller, the speed hit is not significant.

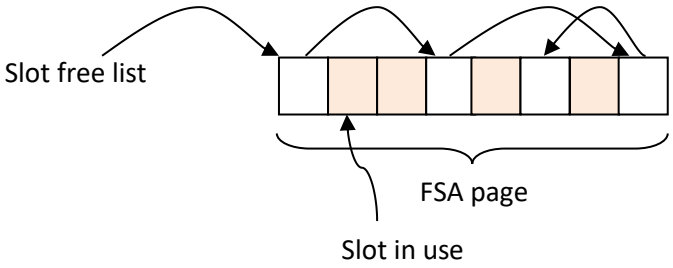
Having resizable regions also means that much less memory is used overall. If each region was committed up front we would always be dealing with the worst possible case, which in reality might never happen. Having the regions resize themselves means that VMem can automatically cope with many different situations, always only using what it needs, down to the granularity of a page size.

This technique of using virtual memory is not just useful in memory managers. It can also be used for creating a resizable array that doesn't need to move in memory. Assuming that it's a pretty big array (in terms of the system page size) we can set it up in the same way, we reserved the entire range on creation, and commit and de-commit pages are the array resizes. The array will only use up as much memory as necessary, and assuming we have enough virtual address space we can reserve as much as we like.



The FSA

The FSA gets its memory from a page heap. When a page is allocated by the FSA it divides the page up into equal sizes slots and links them all together into a free list. To allocate, a slot it is taken off the head of the free list. Freeing a slot puts it back into the head of the free list. The pointer to the next free slot is embedded into the slot so there is no overhead. Note that the free slot list is not sorted by address so the free slots can be in any order.

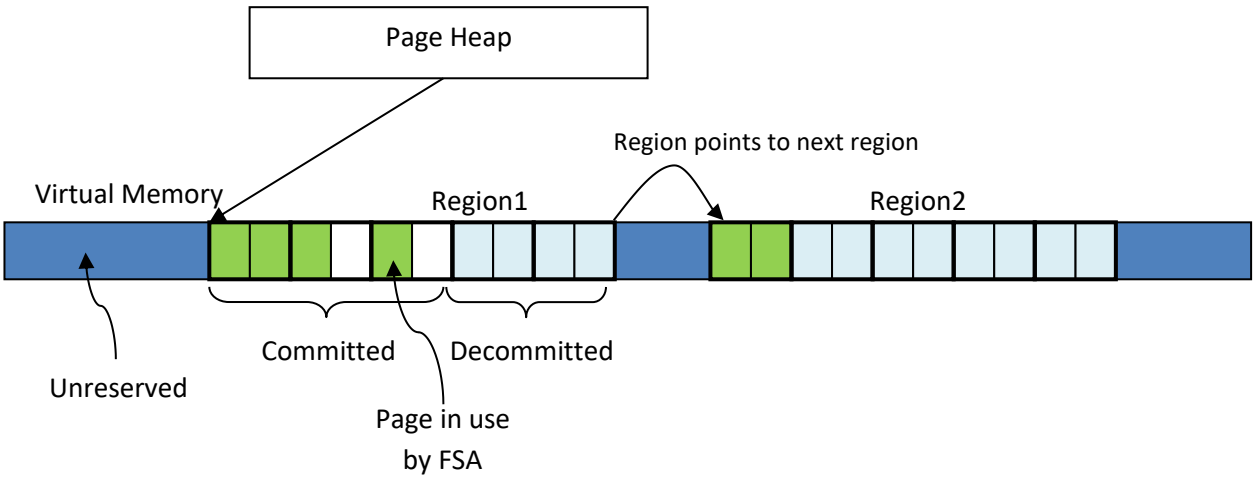


The FSA has a page free list which contains pages that are partially full. When a page is full it is taken off the free list and when a page is empty it is given back to the page heap. Each page has its own slot free list. The reason that the FSA has a page free list and free slot list per page rather than on single free slot list that spans all pages is because of the biasing.

An FSA heap contains a page heap. The page heap is setup to allocate pages of a specified size, for example 4k or 32k. The page size must be a multiple or whole fraction of the system page size. Whenever an FSA allocator in the FSA heap needs some memory it gets it from the page heap.

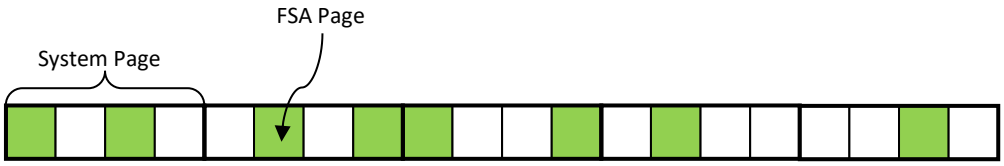
The page heap has a list of regions. These regions are reserved ranges of memory. Typically if virtual memory is not a constraint the region size is made to be big enough that the page heap will only have one region. Allowing for multiple regions give us flexibility and a guarantee that we won't run out of space while there is still system memory available.

Typically a region is 32Mb, the system page size is 4K and the page size is 32k. The example illustrates how it would look if the sub FSA page size was half the system page size with two regions. As can be seen when no sub pages in a system page are in use the system page is decommitted back to the system. Committed pages are biased towards low memory.



Biasing

All empty system pages are decommitted back to the system, but there is a problem with this technique. The way that many allocators work is they give back the most recently freed allocation. A freed allocation gets pushed onto a list, and then immediately popped back off when another allocation of that size is requested. This is good for the CPU cache but it has a drawback. If we assume that the freeing pattern is random this also makes the allocation pattern random. Over time the allocations will diffuse over the entire range and we will have lots of free space, but no system page will be empty so we can't decommit anything.



The way to get around this problem is biasing. Speed of allocation is of prime importance, so we can't do anything too fancy, but it turns out that simply biasing all allocations to the lowest address gives very good results. Here, biasing simply means always allocating the free slot that is lowest in memory. Because all the used memory is squashed up to one end of the range it becomes more likely that pages can be decommitted at the other end of the range.

An FSA has a page free list, a list of pages that have at least one empty slot. To bias the allocations we simply need to keep this list sorted by page address, lowest to highest. Allocations within a page do not need to be biased because that won't have any effect on whether we can free the system page.

All FSA allocators in an FSA heap share the same page allocator. Each FSA is always allocating from the lowest pages that it has. This means that the page heap is much more likely to be able to decommit system pages from the high end of the range.

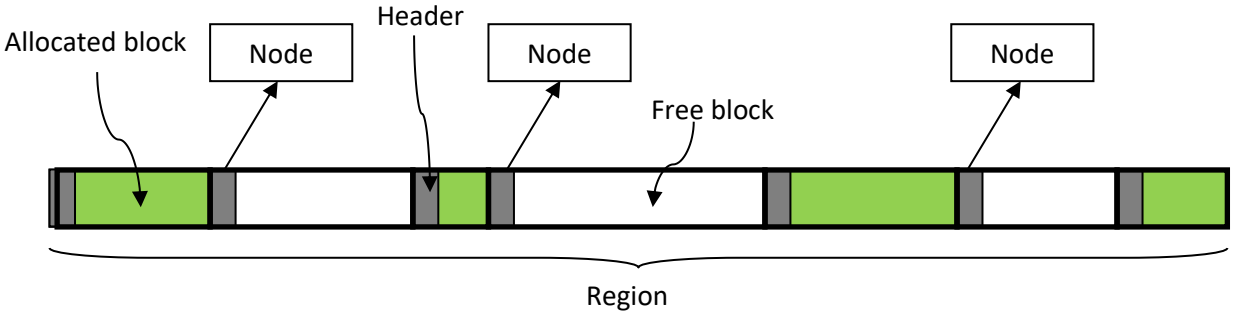
The Coalesce Allocator

The coalesce allocator uses the ‘best fit’ and ‘immediate coalesce’ algorithms.

‘Best fit’ in terms of a coalesce allocator means that it always allocates the smallest block that is big enough for the allocation. Immediate coalesce means that when freeing an allocation the free block will be coalesce immediately with any adjacient free blocks.

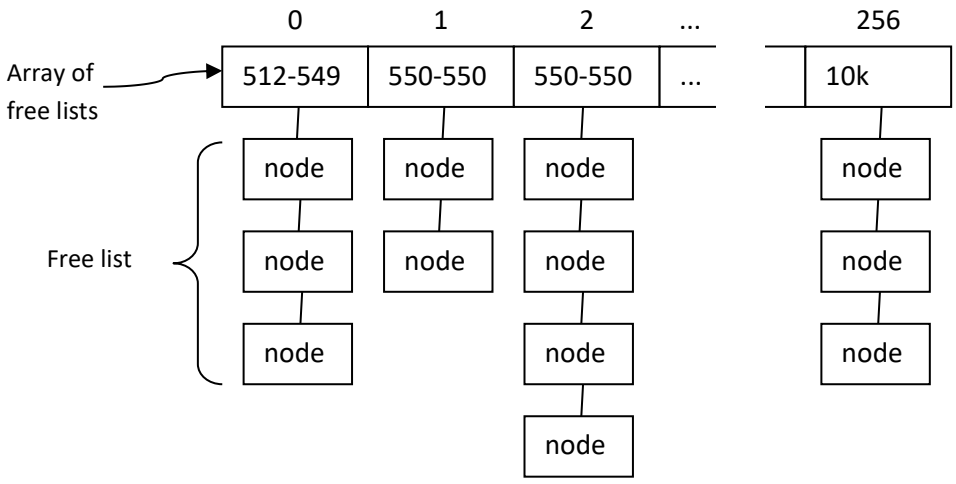
This is not the fastest coalesce algorithms, but they have been found to reduce fragmentation. Because we have the FSA heaps to deal with the majority of allocations per frame, speed of the coalesce allocator is not so much of an issue. Real-time applications typically allocate an order of magnitude more small allocations per frame than large ones. Because coalesce heaps tend to have the worst fragmentation it is worth spending the extra cycles.

The coalesce allocator has a list of regions. Each region contains free and allocated blocks. Each block has a header. Free block headers point to a free node. Each free node is linked into a free list. The header also stores the sizes of the allocation and the previous allocation, these are used for coalescing.



The reason that the free nodes are not embedded into the header is speed up iteration and sorting of the free lists. The nodes are all allocated using an internal FSA which makes iterating over the nodes very cache friendly.

A Coalesce allocator has a minimum and maximum of size it can allocate. There are 256 free lists, one for each size range. Shown below is what it might look like for a coalesce heap with minimum 512 bytes and maximum 10k.



To allocate a specific size it first finds the free list, and then iterates over the nodes to find one that is big enough. If no nodes are big enough it tries the next free list. When a region is created it creates a single free block of the entire region size and adds it to the free list at index 256.

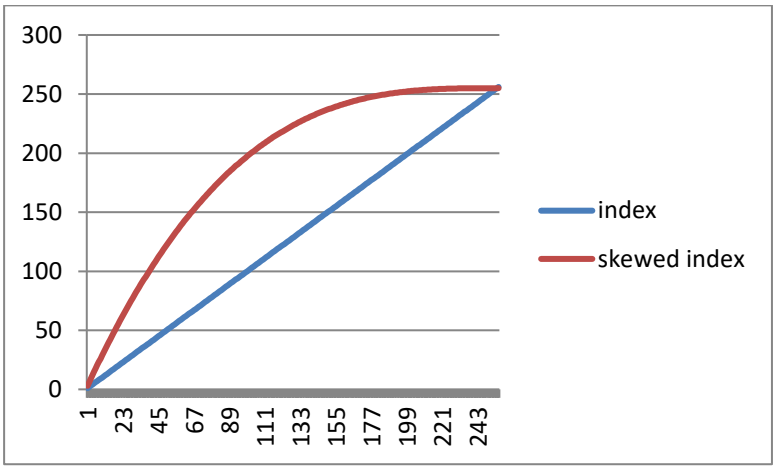
When allocating sometimes the allocation won't quite fit into the smallest free block and there will be a small leftover free block, In VMem this is called a fragment. If this free block is smaller than the minimum allocation size for this allocator there is no point putting it into the node free list, it can't be reallocated and would just slow things down. These free blocks are marked as fragments. When freeing an allocation any adjoining fragments will be coalesced ensuring that these small holes can be re-used.

Skewing indices

It was found that there are typically many more allocations of smaller sizes than large sizes. This unbalanced the free lists, with the first few having lots of free nodes and the last having less, this made iterating and sorting the small free lists quite slow. To counter this effect a function is applied to the free list index to skew the sizes to the end of the free list. This makes the distribution of sizes between all free lists more even.

The index is worked out like so:

r = max – min
s = max – alloc_size
skew = m - (s*s*s) / (r*r)
index = (skew * 255) / r

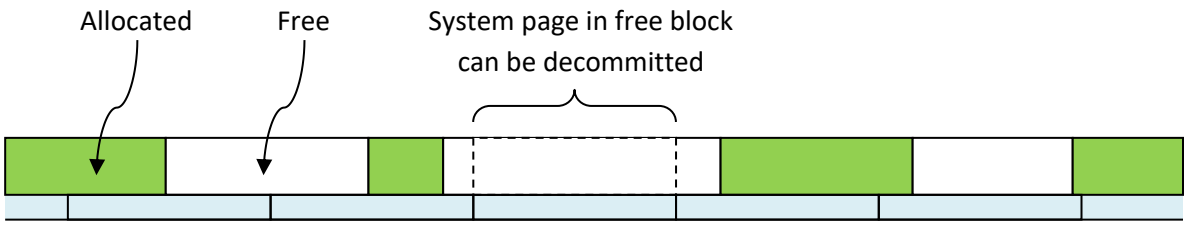


De-committing free blocks

The problem with coalesce heaps is that they can become very fragmented. This is somewhat unavoidable, because they have to deal with different sizes of allocations, and because of the inherent randomness of allocation patterns fragmentation is inevitable. However, we can use the same techniques to reduce the problem.

When an allocation is freed, after it has coalesced with any adjacent free blocks we can decommit any system pages that lie entirely within the freed block. The coalesce allocator uses the 'best fit' and 'immediate coalesce' algorithms in order to maximise the size of the free blocks and increase the chances of being able to decommit pages.

The coalesce allocator also uses biasing to low memory, which has similar benefits as has been described in the FSA allocator. If there are equally good free blocks the lowest one in memory is always used. This is only a small bias, but it has been found that 'best fit' has a bigger impact than the bias. However, the bias can still make a significant difference.



Multi-threading

Multi-threading is increasingly becoming a requirement for all applications, especially real-time applications. Therefore a malloc replacement not only needs to be thread save, but needs to perform well under high contention. Simply putting a critical section around the entire heap can lead to very poor performance.

One thing to note is that in most applications small allocations vastly outnumber larger allocations. For example a typical game might allocate up to 40,000 allocations per second from 0 – 64 bytes, and maybe only around 100 larger allocations per frame. This means that the coalesce heaps would have virtually no contention so simply putting a lock around each larger heap will suffice. The FSA heaps is where we need to focus our attention.

The first incarnation of VMem used a lock free algorithm for the FSA allocators, but this has since been dropped in favour of multiple spinlocks. Lock free lists are a solved problem, so at first sight implementing a lock free FSA seems simple. The problem comes when deciding when to release an empty page. Guaranteeing that no other thread comes along and allocates from the empty page before it is released is currently an unsolved problem. Steps were taking in the VMem allocator to ensure that this didn't happen, but there was still a window of opportunity. The crash only showed up once in two weeks of rigorous testing in a real-world application. Other allocators that claim to be lock free have a very similar problem, although the one that I looked at had a tendency to leak pages instead of crashing. To my knowledge a lock free allocator is currently an unsolved problem, despite what certain papers may lead you to believe.

The current solution VMem uses is for each FSA to have its own spinlock.

Interestingly, in practice using the current spinlock solution was significantly faster than the (admittedly broken) lock free algorithm. The lock free algorithm did perform better if two threads were in high contention both continuously allocating from the same FSA, but in reality this is quite a rare condition. Most large applications have fairly random allocation patterns, two threads allocating the same size at the same time is usually quite rare. Under such low contention spinlocks were out performing the lock free algorithm. The reason for this is that because two CAS instructions were needed, two memory barriers were also needed, and memory barriers are not cheap. The two memory barriers were a constant cost even under no contention. A spinlock only has one memory barrier. To make the lock free FSA totally safe would almost certainly require adding more complexity, and slow it down even more, so even if it was possible it would still be slower than using spin locks.

Integrity Checking

VMem has a significant amount of integrity checking. This checking is reason alone for using VMem. Tracking down memory corruptions, caused by things like buffer overruns and writing to deleted objects are a class of the hardest type of bug. These bugs can take up a huge amount of time at the end of a project and can make the released app buggy if they are not all found. Catching them early is the best form of defence, and VMem has a few tricks to help.

Each integrity checking feature can be enabled or disabled independently, although there are a few that are recommended to be always on except in the very final build. Other more heavy weight options can be turned on after a problem has been detected.

The CheckIntegrity function can be called at any time to run a full check of all heaps. This is quite a slow function, but is useful for narrowing down a bug. Alternatiively incremental checking can be turned on which spreads the checking of the heaps across a few frames.

All of the defines listed below can be found in MemSysDefs.hpp, along with all of the guard values and what they mean.

VMEM_ASSERTS
VMem asserts on pretty much everything that it can assert on. Asserts can be turned off, but in my experience leaving them on can save a lot of time in the long run, with memory corruptions being caught much earlier. The speed hit for these asserts is minimal, they are all cheap checks taking a few instructions.

VMEM_MEMSET
VMem will set all uninitialized memory to a non-zero value and clear all freed memory to a different value. This is very useful for checking for uninitialized variables and writing to deleted objects. It is recommended to always have this enabled except for final builds.

VMEM_FSA_GUARDS
In FSA allocators guards are put at the start and or end of each slot. Typically only the end guard is used, adding just 4 bytes to each allocation. Again, the extra memory usage is not large, so this can always be enabled for debug builds.

VMEM_COALESCE_GUARDS
In Coalesce allocators guards are put at the start and end of each allocation.

VMEM_ALTERNATE_PAGES
This is for FSA page heaps. Only alternate pages are ever committed in each range. This means that every other page is decommitted, and if anything tries to write to that page there will be a system exception. This is particularly useful for catching buffer overruns and the worst sort of memory corruption where random sections of memory are written to. The great benefit of this is that the program will halt immediately anything tries to write to a decommitted page, allowing you to see what the cause is. The other benefit of this is that it only uses up virtual memory, no more memory

is committed, and there is no CPU overhead. If you have virtual memory to space this can be left on in debug builds.

VMEM_COALESCE_GUARD_PAGES
This is similar to the alternate pages for the FSA page heap. When the coalesce heap reserves a range it allocates guard allocations, without committing the memory, these allocations are never freed. If something writes to these allocations the system will immediately throw an exception. Again, this uses up no extra memory because the memory is not committed and there is no CPU overhead.

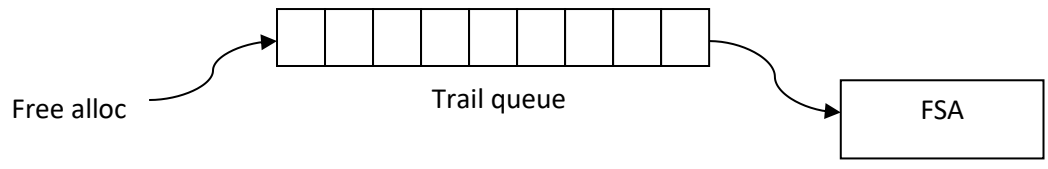
VMEM_INC_INTEG_CHECK
Incremental integrity check. This define is set to a value, such as 1000. Every 1000 allocations VMem performs a full integrity check on one of the heaps, cycling through the heaps on each integrity check. Full integrity checks can be quite slow so usually only turn this in if there is a problem. This can be very useful in catching memory corruptions closer to the source of the problem.

VMEM_TRAIL_GUARDS
Enabling trail guards ensures that the last *n* free allocations will not be re-allocated. When an allocation is freed is pushed onto a guard queue. If the guard queue has reached its maximum size an allocation is popped off the end of the queue and freed. When an allocation goes onto the guard queue the memory is cleared to a specific value, and when it is popped off it checks that the allocation still has the same value. Trail guards are a very powerful tool for catching people using objects that have been deleted which would otherwise lead to memory corruptions.

The speed overhead for trail guards is minimal and the memory overhead can be set to whatever you want. By default turning on this define will put small trail guards on each FSA. However, sometimes it is useful to turn trail guards on for just one FSA and give the trail as much memory as you can spare. For example if the memory corruption is always hitting allocations of the same size trail gaurds can be enabled for that FSA. The more memory the trail is given the better it will work because the longer it will take for it to come off the trail.

Trail guards can be enabled for FSA allocators and Coalesce allocators. In my experience most memory corruption bugs are caused by deleted objects being written to, and trail guards are the single most effective way of tracking these down. Once the overwrite has been caught a memory profiler that has a history option that can show the previous owners of the memory will usually show the cause of the problem.

Conceptually when an allocation is freed it goes onto the trail guard queue and an allocation from the other end of the queue is pushed off and given back to the FSA.
Anything writing to the allocation while it's on the queue is caught.



Stats

VMem keeps track of stats for all of its heaps. This is controlled by the VMEM_STATS define. It is recommended to keep it enabled in all but the final build. The overheads are negligible in terms of both speed and memory.

Note that VMem only stores general stats for each heap, it does not record all allocations or call stacks. Storing detailed information on allocations should be done by an external memory profiler, writing it into an allocator adds unnecessary overhead and complexity.

Below is an example of the output of the stats for the main heap.

		Used	Unused	Overhead	Total	Reserved
FSA1:	56%	15936396 (15.2Mb)	3474104 (3.3Mb)	8773684 (8.4Mb)	28184184 (26.9Mb)	67108864 (64.0Mb)
FSA2:	81%	33447268 (31.9Mb)	5007056 (4.8Mb)	2386272 (2.3Mb)	40840596 (38.9Mb)	100663296 (96.0Mb)
Coalesce1:	77%	32258160 (30.8Mb)	9090808 (8.7Mb)	495768 (0.5Mb)	41844736 (39.9Mb)	67166208 (64.1Mb)
Coalesce2:	98%	42814784 (40.8Mb)	636452 (0.6Mb)	23708 (0.0Mb)	43474944 (41.5Mb)	100667392 (96.0Mb)
Large:	99%	269308158 (256.8Mb)	101582 (0.1Mb)	564 (0.0Mb)	269410304 (256.9Mb)	269410304 (256.9Mb)
Internal:	0%	0 (0.0Mb)	5052 (0.0Mb)	12344 (0.0Mb)	17396 (0.0Mb)	32768 (0.0Mb)
TOTAL:	92%	393764766 (375.5Mb)	18315054 (17.5Mb)	11692340 (11.2Mb)	423772160 (404.1Mb)	605048832 (577.0Mb)

Each row is a different heap. The percent in the first column is the percent of used to committed memory. Each size is shown in bytes and Mb. The columns are defines as follows:

Used: Amount of memory in use by the app.

Unused: Amount of memory allocated by VMem but not in use by the app. Usually an indicator of fragmentation.

Overhead: Memory allocated by the allocator classes for tracking purposes. This is usually negligible.

Total: The total amount of memory committed by VMem. Used + Unused + Overhead

Reserved: The amount of memory that has been reserved. Includes both committed and non-committed memory.

Terminology

Allocator:	Class that allocates memory. For example a ‘fixed size allocator’ or a coalesce allocator.
Heap:	Contains one or more allocators for allocating memory. For example an FSA heap contains an FSA for each allocation size.
FSA:	Fixed Size Allocator. Allocates slots of a single size. Slots have zero overhead.
Slot:	Single item in a fixed size allocator. A slot can be in a used or unused state.
PageHeap:	An allocator that is setup to allocate pages of a specified size. A page heap has a linked list of regions.
Coalesce:	To join two adjacient free blocks of memory into one single block.
Block:	Block of memory is either an allocated or free section of memory in a coalesce heap.
Region:	A contiguous range of reserved pages. For example a page heap has a list of regions it can commit pages from.
Spinlock:	Can be locked or released. When locked stops other threads entering the code block. Other threads spin continuously checking the state of the lock.
Biasing:	Make allocations more likely to be allocated in a specific place, usually low memory.
Guard:	Memory that should not be written to by the application and that is set to a specific value. This value is then checked at part of the integrity checking to ensure nothing has written to it. Guards are often put immediately after the end of an allocation.