About Memory Allocators

Home

Back To Tips Page

In another essay, I talk about memory damage, I also indicated that most people don't really want to know too much about what goes on inside memory allocators. But in thinking about it, I realized that you *do* need to understand what goes on in allocators.

I've spent a nontrivial part of my life writing storage allocators or working in storage allocators, and I keep forgetting how much my knowledge of the internals of allocators I actually use in everyday programming.

For those who want some deeper insight into what allocation is all about, and don't mind seeing sausage made, here's some discussion about storage allocators and performance.

What's "*whatever*-Fit"?

There are many allocation strategies. They depend upon how blocks are allocated and freed. One characterization is by how a block is chosen for allocation.

First Fit

First fit just starts at the front of the list of free storage and grabs the first block which is "big enough". If it is "too big" it "splits" the block, returning you a pointer to the front part of the block and returning the tail of the block back to the free pool.

First-fit has lousy performance. It requires that you may have to skip over blocks that are too small, thus wasting time, and it tends to split big blocks, thus increasing memory fragmentation. The original C allocators were not only first-fit, but they had to scan the entire heap from start to finish, skipping over the already-allocated blocks. In a modern machine, where you might have a gigabyte of allocation in tens of thousands of blocks, this is guaranteed to maximize your page faulting behavior. So it has truly lousy allocation performance and fragments memory badly. It is considered a truly losing algorithm.

Best Fit

"Best fit" tries to find a block that is "just right". The problem is that this requires keeping your free list in sorted order, so you can see if there is a really good fit, but you still have to skip over all the free blocks that are too small. As memory fragments, you get more and more small blocks that interfere with allocation performance, and deallocation performance requires that you do the insertion properly in the list. So again, this allocator tends to have truly lousy performance.

Quick Fit

This is an algorithm first described by Charles B. Weinstock in his Ph.D. dissertation on storage allocation. He measured the performance of this algorithm. It has stunning performance.

The idea here is that you keep a "cache" of free blocks of storage rooted in their sizes. Sizes at this level are always multiples of some basic allocation granularity, such as **DWORD**. Like most algorithms that work well, it is based on patterns of observed behavior. The L1 and L2 caches rely on what is called "locality of reference" to in effect prefetch and cache data that is very likely to be used in the near future. LRU page replacement is based on the fact that a page which hasn't been used in a long time is unlikely to be used in the near future. Working set is based on the fact that, like caches, the pages you used most recently are likely to be the pages you are most likely to use in the future. QuickFit is based on the observed premise that most programs allocate only a small number of discrete sizes of objects. You might allocate and free objects of type A hundreds of times a second, but you will actually be doing this a lot.

QuickFit relies on lazy coalescing. If you free a block of size *n*, you are very likely, in the near future, to allocate a block of size *n*, because what you *really* did was free an object of type A (sizeof(A) == *n*) and you are therefore fairly likely in the near future to allocate a new object of type A. So instead of returning the block to the general heap, and possibly coalescing it with nearby blocks, you just hang onto it, keeping it in a list of blocks of size *n*. The next time you allocate an object of size *n*, the allocator looks on the free list[*n*] to see if there are any blocks laying aroud to be reallocated, and if one is, your allocation is essentially instantaneous. Only if the list is empty do you revert to one of the slower algorithms.

In QuickFit, coalescing is done before you decide to ask the operating system for more storage. First you run a coalesce pass, then see if you now have a big enough block. If you don't, then you get more space from the system.

Note that QuickFit, unlike FirstFit and BestFit, tends to *minimize* gratuitous memory fragmentation, which improves overall system performance.

If you are using the **HeapAlloc** function (rather than the standard **new** or **malloc** allocation), you can use the **HeapSetInformation** function to select the "low-fragmentation heap", which uses a QuickFit-like allocation strategy. See more about this in the discussion of anti-fragmentation.

Split Decisions

To split or not to split? That is the question.

If you have a block of size *k* and you need a block of size *n*, when do you decide to return a block of size *n* and leave a block of size *k-n* on the heap for future allocation? (I'm ignoring the overhead bytes here, although you can't really ignore them).

Well, if *k-n* is too small to represent a block, you can't split. That's because you need some number of bytes just for overhead. If the number of bytes required for overhead is *B,* then if *k-n* < *B* there wouldn't be enough bytes left over to represent a block, even of size 0. So you allocate a block of size *k*.

But it's worse than that. A block of size *B* would represent a free block of 0 size. So we make another decision, that the minimum block size we would leave after a split is of size *L*, so if *k-n* < *B* + *L* we don't split.

Back in 1968, Jim Mitchell [now at Sun] wrote an allocator for our project where he computed, in real time, the mean and standard deviation of each allocation. He used floating point for this on a machine where floating point was not exactly fast (note that on a Pentium 4, a multiple of two **float** operands takes 1 CPU clock cycle, so floating-point multiply costs the same as integer add). But the cost of doing this was so much cheaper than the cost of having lots of little leftover storage blocks that the allocator ran *faster* after it became "slower" (and people who become overly concerned with "efficiency" where all they're looking at is a few lines of code should take note of this: by making the code run slower, the system ran *faster*. Think really hard about that!) Essentially, if he had to split a block, and the leftover block was smaller than *B* + *L*, where *L* was one sigma from the mean of allocations, then he didn't split. He computed *L* dynamically based upon program behavior.

When I implemented a version of QuickFit commercially, I spent a lot of time measuring its performance, and running experiments on it by single-stepping through it. A trick I learned was that if you couldn't find a block of size *n*, you got much better performance if you first checked the blocks at size *2n* + *B*, and split one of those. You got lower memory fragmentation.

Mark-and-Release

For some algorithms, the best allocator is the one from Pascal 1.0: Mark-and-release. In this algorithm, you issue a "mark" request which marks the current heap. Then you allocate trivially by simply incrementing a pointer into the free space. When you are done, you issue a "release" which resets the free storage pointer to the "mark", thus freeing up *everything* that was allocated after the "mark".

This is a pretty severe restriction on allocation. *Everything* allocated after the "mark" operation is implicitly freed. All of it. Not one object can live beyond the "release" operation if it was allocated after the "mark".

But the performance of this allocator is stellar. Nigel Horspool, of the University of Victoria, described a parser that could parse a million lines a minute of source code, on a relatively antique machine by modern standards. To build his parse tree, he in effect did a "mark", and after he generated the code for the syntax tree he did a "release". He pointed out that after the code generation was complete, *you didn't need the tree at all*; there was no need to retain *any* part of it. It could *all* go away. And while building it, you had The World's Fastest Storage Allocator running.

The stack

You can allocate on the stack. This is almost as fast as mark-and-release and is useful in certain contexts. In fact, if you look at the stack frame, it is *exactly* mark-and-release, except the release is implicit when you return from the function.

The **_alloca** function allocates new storage on the stack. The danger here is that if you have it in a loop, your stack keeps growing and growing and growing, and you will eventually get stack overflow if you run out of stack. But it is often convenient for the situation where you need just one object of, frequently, unknown length. The ***x*2*y*** string functions such as **A2W**, **T2W**, **W2A**, **W2T**, etc. use this technique.

Back around 1982, when I'd written a really fast allocator, one of the development projects in the company used it. Within a few days, they were in my office, proving, with a histogram, that allocator performance sucked.

I didn't believe this. I'd sweated blood to get that allocator lean, mean, and fast as possible, and the data made no sense.

But my first suspicion arose because they'd used the Unix performance tool to analyze the code. This was a typical half-assed Unix "solution" (read: inadequate hack that gave the illusion of working, like so many Unix tools). It ran the process as a child proces, and every so often (I think every clock tick, 1/60th of a second) it halted the process and recorded the program counter address. It then plotted a histogram of program counter addresses, normalized by function body ranges (rather than absolute address), and sure enough there was a massive peak at my allocator.

But I knew that allocator could allocate faster than the high-resolution 10us clock (our mainframe could execute 10 instructions in that time) on the machine I developed it on, so why was it so bad on Unix?

Well, I set a breakpoint on the equivalent of 'main', then reached in with the debugger and set a bit in the allocator control word. When the program terminated, it recorded over 4,000,000 calls on the allocator: 2,000,000 allocations and 2,000,000 free operations. I pointed out the obvious: if you call the allocator 4,000,000 times, then it is going to look like you spend a lot of time in the allocator.

The problem was tracked down to a low-level function that formatted some text. While this wasn't C, you could imagine it was something of the form

```
int n = ComputeRequiredTextSize(...);
LPCTSTR p = new TCHAR[n];
sprintf(p, ...);
...do something with formatted string
delete [] p;
```

Executed in the inner loop of a complex function, this meant that the allocator was called 4,000,000 times.

So we went to the person who had written the code, who insisted that this was absolutely necessary because when called the function had no idea how many characters would be required to represent the string, and therefore it could not possibly be changed.

So I changed it.

```
#define EXPECTED_BIG_ENOUGH_SIZE 100
int n = ComputeRequiredTextSize(...);
TCHAR buffer[EXPECTED_BIG_ENOUGH_SIZE];
LPCTSTR p = buffer;
if(n > EXPECTED_BIG_ENOUGH_SIZE)
   p = new TCHAR[n];
sprintf(p, ...);
... do something with formatted string
if(p != buffer)
    delete [] p;
```

(This wasn't quite what I did; I also added code to count how many times I actually had to do the allocation because the **EXPECTED_BIG_ENOUGH_SIZE** was too small. The number was, for most executions of the program, 0. For a few cases, it was 2. This eliminated about 1.8 million allocations and the corresponding deallocations, and we had a significant overall performance improvement).

Unsynchronized Allocation

In a heavily-multithreaded application, particularly one where you are running on a large multiprocessor, memory allocation can represent a significant performance bottleneck. If you have a lot of threads, you're going to have a lot of concurrency. If you have a lot of concurrency, and a lot of allocation/deallocation, then you have a lot of conflicts as each thread tries to get the storage allocator.

The fundamental storage allocator requires synchronization. If you have preemptive multithreading, *or* multiprocessor concurrency, such synchronization is mandatory. Windows has both, so there's no possibility that you could survive in a non-synchronized allocation context (it is worth pointing out that there actually *are* allocation strategies that can work without synchronization, and some are used in the kernel, but they are very sophisticated and outside the scope of this discussion because they don't exist in user space). Essentially, every call to the allocator has the equivalent of **EnterCriticalSection** before it starts doing anything, and upon completion executes a **LeaveCriticalSection**.

This explains why you must *never, ever* do a **SuspendThread** from any thread to another thread. If you suspend a thread while it owns the **CRITICAL_SECTION** of the storage allocator, no thread in your application will ever be able to allocate storage, at least until someone causes a **ResumeThread** to occur on that thread. This means that as each of the other threads tries to allocate, it will hang, and eventually something will hang that causes your whole application to really come to a screeching halt as far as the user sees. This is fatal.

So our problem is that we can either ignore synchronization, and watch our program crash and burn, or depend on the behavior of synchronization, and watch our application slow to a crawl. Doesn't sound like a good set of choices.

Well, there's another alternative: use unsynchronized access.

How can you do this? Well, the way it works is you create a separate heap for each thread. Each thread allocates its local storage from this heap. But no locking is needed, because no more than one thread can ever use the heap.

```
HANDLE heap = ::HeapCreate(HEAP_NO_SERIALIZE, initial_size, maximum_size);
```

The ***maximum_size*** can be 0, indicating the heap is growable indefinitely, up until all memory has been consumed.

One way to keep track of the handle (and this is only if you are in an executable, not in a DLL) is to use **__declspec(thread)**, e.g.,

```
__declspec(thread) HANDLE heap;
```

declared at the module level. This creates a variable that gives the illusion that it is a "module-level" variable, but in fact there is a unique copy of this value for each thread. This involves a cooperation between the compiler, the linker, and the C runtime environment. If you look carefully at the code to access this variable, you will find that it accesses it relative to the "thread context block", which is referenced by **FS:**, which also indicates that although segment registers are rarely of interest in Windows code, there are in fact places where segment registers are still used, and this is one of them. For those who remember 16-bit Windows, where we had to worry about **CS:**, **DS:** and **SS:** all the time, Win32 sets all these segment registers to the same values. But **FS:** is special, and is the pointer to the thread context block. This code pushes the variable **heap** declared above.

```
00052 a1 00 00 00 00       mov eax, DWORD PTR __tls_index
00057 64 8b 0d 00 00 00 00 mov ecx, DWORD PTR fs:__tls_array
0005e 8b 14 81             mov edx, DWORD PTR [ecx+eax*4]
00061 8b 82 00 00 00 00    mov eax, DWORD PTR ?heap@@3PAXA[edx]
00067 50                   push eax
```

So note that the value which is actually picked up is always relative to the thread local storage array, which is found in the **FS:** segment.

Because each thread now has its own heap, there is no need to synchronize. On the other hand, you give up the nominal ability to allocate objects and pass them across thread boundaries, such as would be done in standard cross-thread messaging (see my essay on **worker threads**, or my essay on the use of **I/O Completion Ports** as a queuing mechanism. If ;you use a thread-local heap, it becomes the responsibility of the recipient of the message to pass that pointer back to the thread so the thread can delete it. The receiving thread will be unable to free the storage, because it has no access to the heap that allocated it.

This is also why you would never, ever, under any circumstances imaginable, *ever* want to call *any* method within the thread that terminates the thread. There is one, and only one, valid way to terminate a thread, and that is return from the top-level thread function.

Why this limitation? Consider the code for a well-organized thread function:

```
static UINT ThreadFunc(LPVOID p)
   {
    heap = ::HeapCreate(HEAP_NO_SERIALIZE, 1024, 0);
    ... do thread thing
    ::HeapDestroy(heap);
    return 0;
   }
```

Should any code inside the "do thread thing" part of the code ever call some function that exits the thread (that is, **::ExitThread** or any of its wrapper disguises, such as **AfxEndThread**, **_endthread**, **_endthreadex**, or, horror of horrors, **TerminateThread**, the heap will never be freed. Even though the copy of the thread-local variable associated with that thread is destroyed, the object referenced by that variable, the heap handle in this case, is not destroyed (this is not C++, it is low-level API stuff and it wouldn't know to call a destructor or even know how to interpret the bit pattern stored in the location), and so the heap remains allocated. So we will assume that no one is ever so tasteless as to call any of these methods.

Anti-Fragmentation Techniques

Another use of application-allocated heaps is to help avoid memory fragmentation. Fragmentation is described in my **essay on memory damage**. The simplest way to avoid memory fragmentation is to not allocate chunks of different sizes. This can be handled by having size-specific heaps. All the objects of size A are allocated from heap A; all the objects of size B are allocated from heap B. The result is that whenever an object is freed in heap A, it is an A-sized object and therefore creates an A-sized hole in heap A. The next object allocated from heap A is always of size A, and therefore can be satisfied from any one of the A-sized holes. Since there are no variable-sized holes, there is no fragmentation.

One of the problems of MFC is that in the past it gave no way for the programmer to override the standard allocator, so the only way you could handle this was to create new user-defined classes and implement your own **new** operator. In VS.NET this has changed, but I have not investigated how to do it yet.

In addition to having different heaps for different kinds of objects, there are two other solutions. One is to have a single heap but to always allocate **max**(...) of all object sizes. While this gives you the illusion that it is "wasting" space, in fact, it is conserving space. This is one of the many instances where "less than optimal" local decisions produce globally optimal decisions.

Finally, if you are running on XP or Server 2003 or later, you can use the "Low Fragmentation Heap" option. By calling the **HeapSetInformation** call you can select a **QuickFit** algorithm.

```
ULONG info = 2;  // indicates the Low Fragmentation Heap (LFH) option should be enabled
BOOL b = ::HeapSetInformation(heap, HeapCompatibilityInformation, &info, sizeof(info));
```

When the LFH algorithm is enabled, the allocation is done in quanta identified in 128 "buckets" corresponding to the QuickFit caches.

MFC, when running in release mode, quantizes **CString** allocations in quanta of 64, 128, 256, and 512 bytes. That is, if you ask for a **CString** of length 1..64, you get a **CString** of length 64, no matter what size you really asked for. The result is that you don't get a lot of "little" holes in memory. This is another case of local suboptimality result in significant global performance improvement.

Is it better to use lists or arrays?

If you use linked lists, adding a new element to the list is constant time, that is *O(1)*. For an array of size *n,* adding an element is *O(2*n)*. Clearly, a list is a better representation for something that grows incrementally, because it is so much more efficient. Isn't it?

No. It can be your Worst Possible Implementation.

How could this be? It doesn't make any sense.

Did you read that earlier story where slowing down the code made the system run faster? If not, go back and **read it**.

The folks who did the Lisp Machines back in the 1980s knew this. They represented LISP lists as arrays. Why? Because it was faster.

Consider: to walk a list, you have to touch each element in the list. To insert an element in a list in sorted order gives you an algorithm that is *O(n/2)*, that is, on the average, you're going to have to look at half the elements int he list. To find the insertion point in an array, however, you can find it in $O(\log_2 n)$ which looks good, but you're going to pay *O(2n)* to do the insertion, and that's bad. But those numbers are misleading. The *real* characterization of complexity of *O(k)*, for *k* any expression, is actually *M+C*k*, where *M* is the initial overhead and *C* is the "constant of proportionality", that is, the *actual* cost of the operation. Now consider walking a list scattered throughout memory. Each element of the list might be on a different page. This will generate, potentially, *one page fault per element*, and that puts our *C* value (as **already observed**) at *six orders of magnitude slower than an access that doesn't take a page fault*.

On the other hand, it is fairly trivial to reduce the cost of array insertion from *O(2n)* to *O(n/2)* and the cost of array append to *O(1)*. Look at the **CArray::SetSize** function, for example.

```
void SetSize(
   INT_PTR nNewSize,
```

```
   INT_PTR nGrowBy = -1
);
```

If you set that ***nGrowBy*** value to something reasonable then the **CArray** gets a certain number of memory locations *pre-allocated*. To add an element takes constant time until this extra space is exhausted, at which point new space must be allocated, and a copy is done. With judicious choice of ***nGrowBy*** you can significantly reduce the cost of adding elements (in one example I did, the poster was complaining that he couldn't add more than about 6K elements/second to an array. I demonstrated that this was certainly true in the debug version of the code, but in the release version, given he was attaining sizes of 1,200,000 elements in the array, by setting ***nGrowBy*** to 500,000, I was able to, in the *release* version of the code, add about 15,000,000 elements/second to the array. The problem was that in debug mode, where each constructor did multiple kinds of checking and ***nGrowBy*** had been the default of -1, that nasty value *C* was *immense*.

If the elements had to be inserted in sorted order, on the average *n/2* elements would have to be copied, and the copy could take place within the already-allocated space.

If the nature of the data is that it eventually reaches a maximum size, doing over-allocation doesn't hurt, because you can free up the unneeded space by calling **CArray::FreeExtra** to free up the unused space.

Hybrid schemes

Some years ago I had a problem: we had a list, in sorted order of time. The list was maintained in sorted order. The data structure contained temporal loops. The problem was that when a loop occurred, it was a loop expressed by giving the actual timestamp that the loop was to return to, not a pointer to the structure. So the problem was how to find the correct place in the structure. Unfortunately, searching the structure linearly didn't work. There wasn't enough time, and we kept missing the realtime window.

The solution was to use a hybrid scheme, one based on the notion of "hints". I had a fixed size array of timestamps kept in sorted order. I could use **bsearch** to search this in $O(log_2\ n)$ order where *n* was the number of elements in the array (I chose 128, so this took at most 7 probes). The problem with **bsearch** is that it only returns a match on equality, so I had to use a **bsearch**-like mechanism which returned me a pointer to the timestamp entry *i* such that **hints[i] <= t < hints[i + 1]**. This had a pointer to the list element that was the start of a sequence of elements. But because I had 128 of these buckets, I never had to search more than 1/128 of the list, and generally this was no more than about 10 elements. What had been a linear search with average performance of $O(n/2)$ now became a search which on the average was $O(3.5) + O(5)$. Thus I had the simplicity of using the (existing) linked-list representation for data with the performance of an array implementation.

Why should I care about any of this?

Because you might care about memory fragmentation, allocation performance, page fault frequency, and similar problems. If you do, you may have to move "outside" the normal allocation strategies to use your own style of allocation. Knowing what failure modes to watch for in terms of performance, knowing that there alternative algorithms, can be a big help in figuring out what to do.

If you don't know what memory fragmentation is, what a working set is, why you take page faults, and complexity analysis, you don't have a good handle on why your program's performance sucks. Any of these factors can contribute to the problem. Improving performance is not a function of writing "small, fast code". Code that is small, fast, and memory efficient might in fact be the *worst possible solution to your problem*. Performance is impacted far more by architecture than lines of code. Lines of code and the efficiency of instructions only matter in very limited contexts, such as the inner loop of an image convolution or DSP algorithm.

[And even then is is problematic. In one case I needed to analyze a set of audio samples and render them in the frequency domain. I wanted a proof-of-concept implementation in an existing program, so I just allocated a block of floats the same length as my audio sample, did an integer-to-float conversion of the samples, called the FFT subroutine to give me the Fourier transform from time/pitch to frequency/intensity, normalized the result to 32,767=100%, and converted it back from float-to-int for plotting, then freed up the array of floats. "The performance of this algorithm sucks, but if it works, we can speed it up". We never bothered to speed it up because it reacts fast enough to in effect be "real time" from the user's viewpoint. Performance only matters when it matters, and the rest of the time it is irrelevant. See my essay on **program optimization**. Note that allocation was irrelevant to program performance].

Note also that some of these techniques can reduce memory leaks, heap fragmentation and consequent effects on working set size, and so on. So you get all kinds of improvement in your program's overall performance. And you might even improve its correctness and robustness along the way.

See also

- **An introduction to memory damage problems**
- **Inside storage allocation**
- **Optimziation: Your Worst Enemy**
- **I/O Completion Ports**
- **Worker Threads**

The views expressed in these essays are those of the author, and in no way represent, nor are they endorsed by, Microsoft.

Send mail to newcomer@flounder.com with questions or comments about this web site.
Copyright © 2006, The Joseph M. Newcomer Co. All Rights Reserved
Last modified: May 14, 2011