FixedAlloc and FixedMalloc

2009-03-13 / rev 1 / lhansen@adobe.com

Goals

The purpose of this code review is to examine whether FixedAlloc and FixedMalloc are suitable for mobile systems. Primarily, this means verifying that they incorporate the following.

- Abstractions and porting interfaces that are appropriate for mobile and embedded systems with non-desktop operating systems (to the extent they are not completely portable).
- Policies (placement, allocating and releasing memory) that tend to reduce memory consumption, possibly even if this reduces performance.
- Engineering decisions that are appropriate for systems with small caches and slow memories.

The review is also looking for troublespots that should be subject to further scrutiny, primarily in the form of further experimentation. For these and other to-do items, look for **Issue**/**Action** pairs below.

In this review I see it is my job to point to design and implementation choices where (a) there may be a problem and (b) no evidence is presented that there is not a problem.

Bugs are logged at https://bugzilla.mozilla.org/show_bug.cgi?id=481683, or in separate bugs blocking that bug.

Major findings

FixedMalloc implements a "simple segregated storage" allocator (see [1], Section 3.6) with 4KB blocks, special handling of objects above (approx) 2KB, and empty-block deallocation; GCHeap is the block manager. The allocation and deallocation algorithms are loopless and fast. The allocated objects are headerless and per-block overhead is only 0.8%.

It is however possible that this allocator suffers from worse fragmentation (ie, increased memory consumption as a direct result of being unable to use free memory, usually measured as the difference between the peak heap size as allocated from the underlying allocator and the peak heap size in terms of live program objects and book-keeping storage) than competing algorithms. The following issues should therefore be addressed with some priority:

- There is potentially a large amount of internal fragmentation (wasted memory within blocks and within objects). However, there's no way to know: there is no accounting of internal fragmentation. A credible accounting of internal fragmentation must be implemented. Following that the amount of internal fragmentation must be examined experimentally.
- There is potentially a large amount of external fragmentation (the inability to use free memory for allocation requests). This must be examined experimentally. Two effects are important:
    - In general a simple segregated storage allocator has a problem with fragmentation: even when most of a block is free its free memory cannot be reused for objects in other size classes than the one to which the block is allocated.
    - The chosen block management / object placement policy within each size class may keep blocks live longer than other policies would have, exacerbating the previous problem.

General Remarks

FixedAlloc and FixedMalloc are the lower and upper layers, respectively, of the manually-controlled object management system in MMgc. The APIs are generally those of malloc and free. The allocators return uninitialized memory.

FixedAlloc is aptly named, because it manages objects of a single size only.

Issue: FixedMalloc is less well named, since it handles requests of any size and dispatches them to various underlying allocators. (I suppose "fixed" could mean "nonmoving" or "won't disappear unless you delete it" but that's stretching it :-)

Action: Perhaps rename FixedMalloc when/if we get around to a proper API scrub of the entire VM; low priority.

The general requirement for this subsystem would seem to be to provide efficient allocation and deallocation of objects of many diverse sizes from the very small to the very large (efficient in time and space).

More specific expectations are:

- fast allocation and deallocation on average, with bounded pauses (only soft requirements)
- thread safe API (?)
- low per-object overhead (low internal fragmentation)
- high degree of utilization (low external fragmentation)
- should return block memory to the underlying block allocator for recycling.
- accurate accounting of total memory, allocated memory, internal and external fragmentation

FixedAlloc APIs

FixedAlloc.h defines the FixedAlloc APIs. These APIs are not well documented. There's also comment rot in FixedAlloc.h, for example it refers to 'FixedAllocLarge' but that class does not exist.

The APIs provide allocation and deallocation as well as accounting services.

Issue: The accounting services do not distinguish between free memory and "overhead" memory (memory used for allocator data structures and internal fragmentation - memory lost at the end of a block because the object size does not divide the block size evenly).

Action: Want to fix that, the cost of doing so is negligible.

FixedMalloc APIs

FixedMalloc.h defines the FixedMalloc APIs. They are again not well documented.

There's a little API rot here, the Init and Destroy methods are empty and a comment states that they're not available, I believe this is a recent change and presumably it'll be cleaned up by and by.

The API consists of methods for allocating and deallocating objects of any size (up to 4GB), getting the size of an object given its pointer, and accounting methods: retrieving the amount of memory controlled and the amount of memory live.

Issue: As for FixedAlloc there is no distinction between memory that's free and memory lost to overhead and internal fragmentation.

Action: Want to fix that; discussed in more detail later.

Technical Details

FixedAlloc

Overhead, allocation, deallocation

A *FixedAlloc* is a simple allocator for headerless objects of a given size. The size is set when the FixedAlloc is created and cannot exceed the size of a page minus some space used for internal purposes by FixedAlloc. The theoretical size limit is 4096-32=4064 bytes on a 32-bit system (less on a 64-bit system). Since the available space is divided among equal sized objects the practical limit is one that does not waste too much space on the page; a comment in FixedAlloc.h suggests that roughly 400 bytes is the practical limit.

The book-keeping space is allocated at the beginning of the object and that fact is part of the API: FixedMalloc depends on that fact to be able to tell large objects from small; only large objects start on a block boundary. FixedMalloc also knows how large the book-keeping space is, but without obtaining it from FixedAlloc - it just "knows". And it knows that the book-keeping space is larger than 2 words.

Issue: The three items just mentioned are legitimate needs, but the API design is not ideal. (For example, FixedMalloc loses if a C++ compiler does not pack the two shorts in the page header together in a single word.)

Action: Clean up the API. Low priority.

The divisibility should not be ignored; for an object size of 372 bytes, 344 bytes goes to waste at the end of the block - an overhead of 8.5%. This overhead is related to the size class, not to the utilization, and represents the *least* amount of memory held by a FixedAlloc instance that can't be utilized for other size classes.

On the other hand, the amount of memory actually wasted depends on the distribution of size classes at run-time and the number of size classes. As we will see, FixedMalloc is careful to create only non-wasting size classes, but that introduces other problems.

Issue: It is possible that waste can be substantial.

Action: As noted earlier, we want an API to account for waste, but otherwise we should not do anything here at the moment.

The FixedAlloc instance maintains a list of blocks that it owns. These are obtained singly from the underlying GCHeap.

An object is mapped to its block efficiently by masking off the low 12 bits (so FixedAlloc truly depends on a block being 4096 bytes; presumably it can be adapted to any power-of-two block size).

Each block has a list of free objects and a pointer past the last object carved from the block. The latter pointer reduces latency though not throughput, at the cost of one word - reasonable IMO, as that word would otherwise just be used for padding. In addition there is a list of blocks with free objects.

There is a "current block" for the allocator, it always has space for at least one allocation. (The first block for the allocator is created on the first Alloc call, so unused size classes cost very little.)

When an object is allocated it is taken off the free list of the current block, otherwise carved off the end of the current block. If either action makes the block have no free objects then the current block pointer is updated: if possible a block is removed from the list of blocks with free objects; otherwise, a fresh block is allocated.

Issue: Strangely for a fixed-object-size allocator the *Alloc* method takes the size of the object to allocate. The value is used for debugging but is redundant in release builds.

Action: Consider whether passing the request size is really necessary. Low priority.

All out-of-memory handling takes place on the block allocation level, allocation failures are not propagated through FixedAlloc.

When an object is freed, it is pushed onto the free list of its block; this block will then be added to the list of blocks with free objects if the block was previously full. If freeing the object makes the block have no live objects then the block is returned to the GCHeap.

Block management

Clearly, as long as there is a free object in any block owned by the FixedAlloc it will be prefered over allocating a fresh block, so with that narrow view the allocator is optimal.

When a block is placed on the list of blocks with free objects, it is made the *first* free block, and when a block is taken off that list to be used for allocation, the *first* free block is taken. Ergo, Free and Alloc implement a block management strategy where the block that most recently transitioned from full to non-full is favored for further allocation.

Issue: The block management policy just outlined *may* keep more blocks in play than necessary, and it *may* scatter allocations across more blocks than necessary. Either way, it may reduce the number of globally free blocks or make ranges of globally free blocks smaller. In either case the result may be an increase in heap size; ie, we may have increased fragmentation.

(When I say "may" it is because the effects will depend crucially on liveness patterns in the programs. However, research has shown that block and object management strategies that pack allocations at one end of the address range will *tend to* reduce fragmentation.)

Action: Alternative management strategies need to be experimentally tested, to see if they have effects on heap size. One simple alternative to the current strategy would be to keep the free list in strict address order so that blocks in the least active address range (high or low) will tend to be freed, as objects in them have more time to die.

Performance

The speed of FixedAlloc would seem to be excellent in general; there are no loops on the critical path and the slowest operations are going to be allocating and freeing blocks.

Issue: Since FixedMalloc does not call FixedAlloc but instead FixedAllocSafe for allocation and deallocation, the path to the allocator is two calls deep in the common case.

Action: If there's evidence that FixedAlloc is on the critical path for performance we should consider inlining the fast paths of FixedAlloc::Alloc and FixedAlloc::Free. With slight tweaking, the fast paths (take an object off the current block's free list; put an object on its block's free list) could be made inlineable, with book-keeping and block freeing postponed until the out-of-line path is taken. (In this case CreateChunk would probably divide the block memory into objects and populate the free list with those objects, and the first chunk would not be created by Alloc, but when the FixedAlloc is created - or could be initialized to point to a dummy block with an empty free list, say.)

FixedAllocSafe

(Part of the FixedAlloc API.)

The class *FixedAllocSafe* extends FixedAlloc and provides thread-safe Alloc and Free methods.

Issue: Since the Alloc and Free methods in FixedAlloc are not virtual, the client must not cast a FixedAllocSafe to FixedAlloc and then call Alloc and Free in the hope that they are thread-safe; they will not be. The client must be cognizant of what's going on.

Action: No action; give 'em rope. But it would be good to document it.

Issue: The FixedAllocSafe class is not thread safe for the rest of the API; for example, if one thread calls Free (which may free an arbitrary block) while another thread calls GetBytesInUse (which traverses the block list internal to FixedAlloc) there will be a race in accessing and updating the block list and the memory of the block that's being freed. (That memory may become unmapped when the block is returned to GCHeap, so we may in principle see a crash.)

Action: (Too much rope?) At a minimum document this for FixedAllocSafe.

Issue: I don't know what the intent of the OOM handling strategy is, but it is possible for the FixedAllocSafe's internal lock to be held when Abort() is called; as Abort longjumps to the top level, the lock will not be released. This matters only if there is going to be some magic after the longjump that clears out the heap and runs some shutdown code in the host, and that shutdown code again enters the FixedAllocSafe path for the same object size. At that point the system will deadlock.

Action: Check that this can't happen.

FastAllocator

(Part of the FixedAlloc API.)

The class *FastAllocator* is a very simple base class that provides *new* and *delete* functionality with a FixedAlloc (not a FixedAllocSafe) as the underlying memory manager.

Issue: This class is very, very strange. Its *new* operator uses the provided FixedAlloc instance, but the *new[]* operator does not, instead going to the FixedMalloc (!) instance that hangs off GCHeap. This sort of makes sense, as FixedAlloc can't handle variable sized items and therefore new[] couldn't use it, but I find the behavior bewildering.

Action: At a minimum document the behavior.

Issue: FastAllocator is not used in the VM code (not even in selftests). I assume it is intended to be used by the Player.

Action: Verify that it's used by the Player. Perhaps protect it with a feature so that clients that don't use it won't see it, and nuke it if it isn't used at all. Or perhaps just move it into the Player code.

Action: Provide test cases for it if it remains in AVM code.

FixedMalloc

FixedMalloc is a general-purpose memory allocators. FixedMalloc creates 41 instances of FixedAllocSafe, for various size classes (on 32-bit systems there's a 4-byte allocator, then allocators spaced 8 apart up to 128, then allocators spaced unevenly apart up to 2032. The unevenness is due to FixedMalloc's attempting to use the space of FixedAlloc blocks as best possible: the size classes 448, 504, 576, 672, 808, 1016, 1352, and 2032 all divide 4064 (the available space in the block) nicely, if not always perfectly.

Issue: The documentation of the size classes is stale and/or missing.

Action: Fix the documentation.

Issue: Making the size classes divide the block size nicely means there won't be internal fragmentation in the block (see earlier discussion about FixedAlloc), but the trick just shifts that fragmentation into the object, thereby making it harder to account for. We don't know whether these size classes actually fit the application very well.

Action: We really want to know how much internal fragmentation there is, because that will tell us whether we want more size classes or different size classes.

The Alloc, Free, and Size methods are inlined, but call out to more substantial methods.

Issue: There is some brittleness in FixedMalloc: The largest non-large block size on 32-bit systems is 2032. The overhead for the FixedBlock structure is (in the best case) 8 words (if two shorts are packed together), ie 32 bytes. 2032*2 = 4064. A block size is 4096. 4096-4064 = 32, well and good. But if the shorts are not packed, then only one of the larger blocks fit on the page and we will have a lot of block-internal fragmentation. This is true for most of the large size classes, because the sizes have been chosen to pack the block tightly. There should be an assert somewhere that checks that the header size is what we expect it to be, but there isn't. Some compilers are not good about the alignment, so for the sake of porting we should check.

Action: Add that assert.

There is also a large-block allocator for requests larger than 2032 bytes. This allocates an integral number of blocks directly from GCHeap and returns a pointer to a location very close to the beginning of those blocks (in debug builds there may be some debugging information at the beginning of the block).

User code that has a fondness for page-sized allocations will work well with this scheme, but for arbitrary large objects there could be significant wasted memory (internal fragmentation) from rounding the rounding up.

The GetBytesInUse API contains the comment "not entirely accurate" when it accounts for the size of large objects. In fact the accounting entirely ignores internal fragmentation for large blocks, so any memory reporting will not get a correct view of internal fragmentation. Since internal fragmentation in large objects is expected to be 2KB *per object on the average*, and since objects are allocated "large" starting at a size less than 2KB, and since even large objects will probably tend to have an inverse exponential size distribution, the internal fragmentation for large objects may be very significant. (Assuming a distribution where half the objects are 4KB, a quarter 8KB, an eight 12KB, and so on, with 2KB wasted per object on the average, the total waste is 33%.)

Issue: Internal fragmentation in the large-object allocator needs to be accounted for so that we can determine how well this allocator serves us.

Action: Investigate options for tracking object size exactly without perturbing memory consumption unreasonably. (A strawman proposal would be that objects that almost or completely fill out their pages have no exact-size field; objects that occupy no more than some number of pages minus five words have a distinguished four-word signature at the end of the last page preceded by the true size of the object. It's better than what we have now.)

Memory utilization in segregated allocators

Free memory in one FixedAlloc cannot be reused in another; this is a direct consequence of the simple segregation by size. Wilson ([1], section 3.10) reports that simple segregated storage is subject to serious external fragmentation for this reason.

However, whether this is a problem or not depends hugely on the behavior of the host program. Clearly it can be awful: In principle, it is possible to have an almost empty heap yet for it to be impossible to allocate objects of almost any size. (Fill the heap with objects of size n; then carefully free all but one from every block. Now try to allocate an object of size m != n.) But if the host program has allocation and deallocation behavior that fits simple segregated storage well, then fragmentation could be very low.

Issue: A segregated fit allocator is *probably* not an optimal choice for small-memory systems.

Action: We need to experiment and measure internal and external fragmentation and memory consumption across many benchmark programs.

Large object placement

As FixedMalloc relies directly on the underlying block manager to handle allocations of multi-block memory for large objects, FixedMalloc depends directly on the algorithms in GCHeap for object placement. (The GCHeap API does not allow for placement hints, but it's not clear that FixedMalloc could provide any either.)

GCHeap uses a hybrid best-fit/first-fit strategy to search for a suitable block, except when configured to search for the oldest block; as noted in the audit of GCHeap, searching for the oldest block is potentially both slow and detrimental to good block management.

But at this time nothing suggests that relying on GCHeap for large object management is a bad idea.

References

[1] Wilson, Johnstone, Neely, and Boles, "Dynamic Storage Allocation: A Survey and Critical Review". Originally in *Proceedings of the 1995 International Workshop on Memory Management*, Springer Verlag. Later revised, see ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps.