GCHeap and its Platform Layer

2009-03-06 / rev 1 / lhansen@adobe.com

Goals

The purpose of this code review is to examine whether GCHeap is suitable for mobile systems. Primarily, this means verifying that it incorporates

- Abstractions (and porting interfaces) that are appropriate for mobile and embedded systems with non-desktop operating systems. Some systems have MMUs and sometimes the ability to reserve memory without allocating it, but no virtual memory per se; yet others have only primitive memory management facilities.
- 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.

The general assumption is that heaps will be in the range 4MB - 40MB, possibly larger but probably not smaller.

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.

It's possible to imagine various worst-case scenarios. One is certainly a configuration that uses *malloc* as the underlying memory allocator.

Terminology: "Underlying allocator" means the allocator that GCHeap or its platform layer uses to obtain memory from the host system.

General Remarks

GCHeap is the block management subsystem for MMgc and is layered above the platform interface and below all the object managers (FixedAlloc, FixedMalloc, and the GC).

The main requirements for the block manager are:

- provide efficient allocation and deallocation of blocks (efficient in time and space)
- heuristically optimize block management so that requests for many consecutive blocks can be accomodated without requesting more memory from the operating system (ie, avoid fragmentation on the block level)
- return block memory to the operating system when possible
- be portable to many classes of operating system
- make efficient use of the various classes of operating systems (eg, use lazily committed virtual memory when it's available; make use of underlying heap zeroing functionality; avoid causing fragmentation in the underlying allocator)

GCHeap.h defines the GCHeap object structure. This file suffers from advanced comment rot; there is scant documentation and much of what is there is misleading or wrong.

(I have filed Bug 481683 in Bugzilla to track this and other issues in GCHeap for the time being.)

GCHeap API

The upward-facing API from GCHeap consists of methods for allocating and deallocating groups of blocks; setting the protection bits on the operating system's pages (to make memory executable and read-only after JIT generation); obtaining memory statistics; manipulating the heap on a gross level (expanding and contracting it); and providing an event callback interface ("hooks").

Apart from a dearth of documentation, and some misleading documentation, this API appears to be just fine.

Platform Layer / Porting API

The porting interface to the operating system consists of a handful of functions that are defined as part of GCHeap but which must be implemented on the various platforms; the semantics of these are not documented but have to be gleaned from the implementations that already exist.

More porting interfaces exist than are marked as such; for example, the configuration parameter *blocksSpanRegions* is documented as an OS-specific parameter but is set to *true* on all platforms; ditto, *searchForOldestBlock* is always set to be the opposite of *blocksSpanRegions*, but is said to be OS-specific too.

The main variation point for GCHeap is whether virtual memory is available or not. So far as I can see, no documentation is available to a platform group about what the consequences are of turning it on or off, and what ought to be true about the platform (apart from some sort of lazy commit facility being supported) for it to be turned on.

Issue: Porting should be facilitated by substantially cleaning up the porting interfaces in GCHeap.h: separating and documenting the functions that each platform must implement.

Action: Write up the porting interface thoroughly and perform a review on the resulting API. Review to include more people than myself.

Technical Details

Blocks, HeapBlocks

The fundamental unit of memory managed by GCHeap is the *Block*. Blocks are always 4K on all platforms. All the memory in a block is available to the client that allocated the block: metadata are maintained off-block.

Issue: The value 4K is usually referenced symbolically as *kBlockSize* but is occasionally hardcoded. I don't know if it's meant to be possible to change it.

Action: Clarify whether the block size is a fixed value that the block manager requires for correctness or whether it's a parameter that can in principle be changed.

Consecutive Blocks are managed in groups of varying size. A *HeapBlock* is a data structure that contains metadata about the consecutive Blocks in its block group. Every block belongs to exactly one HeapBlock. There are as many HeapBlocks as there are Blocks.

A central data structure is the *blocks* array, a linear array of HeapBlocks. It maps every Block name (an index) to a corresponding HeapBlock. Only the first block in a block group has a HeapBlock with information about that group; all other HeapBlocks for the blocks in the group have their size field set to zero.

The block name is obtained from the block address through the Region data structure (discussed in the next section); the block's address minus the region's base address gives the block index within the region, and the region provides a base index to which the index within the region is added to provide the global block name.

On a 32-bit system a HeapBlock is at least 24 bytes (provided two boolean fields are packed together by the compiler), representing a 0.6% storage overhead.

The *blocks* array is allocated and deallocated every time the number of HeapBlocks in the system increases or decreases, which is when a Region is added, extended, or removed, or when reserved memory is committed or decommitted. For a 40MB heap with 4K pages, the block array has 10K elements, or 240KB of data.

Issue: A 240KB array can be a little rough to handle:

- Repeated allocation and deallocation of a number of increasingly large odd-sized allocations for the block array will tend to lead to fragmentation in the underlying allocator; this may be bad if that allocator is also used to obtain storage for our blocks
- On small mobile platforms, which tend to have small or no caches and slow memories, the copying of 240KB of data from one array to the other can actually be slow.
- Since the old and the new arrays are both live when the array is grown, the peak memory consumption just for this array is 480KB. Such a peak may be hard to handle in the underlying allocator without growing the heap.

I hasten to add that adding or removing a region is probably a somewhat rare occurence in practice, as GCHeap takes care to avoid contracting the heap unless there's substantial free memory. However, during rapid heap expansions (eg, during startup) there may be rapid growth in the array, and this copying may add to memory management overhead in phases of growth.

Action: Look into whether the blocks array could be split so that each region carries an array for the blocks in the region. The result will be to allocate more, smaller blocks. Currently the blocks array is uses for coalescing free blocks and the algorithm may depend on the array being linear. Still, such a large array seems like a poor choice for small systems.

Action: Alternatively (and not as good), consider maintaining some slop in the array so that it can be allocated and deallocated less often.

When a block is requested by higher-level allocators it is removed from a block free list if possible. An option can be set in GCHeap to search the free lists for the oldest available block ("oldest" in the sense that it was allocated earlier) that will accommodate the result.

Issue: Finding the oldest block is potentially slow, since the search is sequential and walks all the free lists from the block size and up. The bound on the free list is the number of blocks in the system divided by two (since free blocks are merged): for a 40MB heap with 4KB pages this yields 5K blocks.

Action: Measure the number of probes per allocation for the current algorithm on reasonable benchmarks and relate it to running time; if the impact is significant, investigate other algorithms /

data structures.

Issue: Finding the oldest block is potentially not good block management. In systems where the underlying memory management is sbrk/brk (as we must assume for the smaller systems), only free memory at the top of the heap can be returned to the operating system. Thus one is more interested in the lowest addressed block (address-ordered first fit allocation), not the oldest block. Though it will often be the case that the oldest block will be the lowest-addressed block and so on, even on platforms where *malloc* is the underlying memory allocator, this is by no means guaranteed.

Action: Investigate a lowest-addressed block preference instead of the oldest-block preference.

Regions

The next larger unit of management is the *Region*. A Region is the unit of allocation and deallocation from the underlying allocator, think of it as a large swathe of contiguous, available address space. The parts of the region that are mapped to physical memory are divided into HeapBlocks.

The minimum region allocation size is *kMinHeapIncrement*, set to 32 blocks or 128KB. A larger expansion occurs only if higher-level allocators asks for more than that. So far as I can tell, almost all requests from FixedMalloc will be smaller than that, and most requests from GCAlloc (the allocator for the garbage collector) will be small - one block. So 128KB regions will be used in the absence of other effects. (The virtual memory platform interface changes this; regions start at 1MB and can grow.)

The blocks array is not sorted -- the blocks for each new region are added to the end of the array, regardless of the address ordering of regions.

Another central data structure is the list of regions, through the GCHeap element *lastRegion*. This list is unsorted; as regions are created they are pushed onto the list.

In order to map an address to its HeapBlock the address is first mapped to a region, which is then mapped to a HeapBlock by looking the region's blockId up in the *blocks* array, offset by the block number within the region. The lookup in the regions list is by linear search.

Issue: The linear search is OK only if the region list is short or searches are infrequent. A comment in the code asserts that the region list will usually be short, but that's probably true only for a virtual memory based GCHeap where regions are large (1MB by default) or may be merged. For a GCHeap that uses *malloc* or *memalign* as the underlying allocator, where regions cannot in general be merged, the list may become significantly longer, depending on the typical region size. For a 40MB heap on mobile systems, with the default 128KB allocation request size (see below), there would be 320 regions in this list.

Action: On a non-virtual memory enabled system, try to measure the number of probes performed by the region manager on a large application, and relate these to time spent.

Issue: The region list would have better locality during these searches if it were managed like the blocks array: a linear array of in-line structures, not a linked list of heap-allocated structures.

Action: Even if the number of probes is not a problem it may be useful to restructure the region list as an array. Cache locality would be better for one thing.

Platform Layer: Non-virtual Memory

Issue: The code for non-virtual memory does not compile on any platform, because GCHeap no longer include the members *m_malloc* and *m_free* that are needed.

Action: This needs to be fixed. I've submitted a preliminary patch.

A 128KB typical region size is probably OK for embedded (it's what I would have picked myself). Even so it's not unproblematic. Both much smaller and much larger regions might work better.

First an observation. At present, all the non-virtual-memory allocators use *malloc* as the underlying allocator. Since malloc is not guaranteed to return 4096-byte aligned memory, the trick is to allocate 4095 bytes per block more than needed, and then align the returned pointer appropriately. That results in a guaranteed waste of 4095 bytes per Region. In practice this will mean that about 3% of the allocated memory is lost: 4095/(128*1024+4095)=0.030. For a 40MB heap this is more than 1MB, which will presumably be sorely missed.

Issue: The current implementations of non-virtual memory allocators probably waste too much memory.

Action: Investigate whether switching to other APIs, for example *valloc* or *memalign*, will result in better heap usage.

In addition to looking for less-wasting APIs (which are not guaranteed to be available everywhere) we can consider allocating much larger regions. A 1MB region size would result in 0.4% lost memory, for example. The number of regions would also be smaller, which might reduce pressure on points made earlier in this review, like the linear search of the region list.

Issue: If the platform allocator wants to allocate larger blocks, it should allocate blocks that are a multiple of the blocks that GCHeap will usually request (ie, kBlockSize*kMinHeapIncrement). These parameters must therefore be documented; they are part of the contract between GCHeap and the platform allocator.

Action: Document these pragmatics in the porting interface documentation so that the platform-layer allocator can make larger requests.

Taking this to the logical extreme, it may be appropriate for the underlying allocator to perform exactly one allocation. Again, this decision is really made inside the platform layer but we need to ensure that GCHeap can work with it, the abstraction can't be too opaque.

It may also be interesting to allocate much smaller regions. On an operating system like Symbian, where the *RHeap* data structure provides an effectively unbounded linear array of memory on demand, allocating just a single block may be better, because it leaves more memory for other applications. (Having a single region is probably right, too.) Effectively it's a virtual memory system without *mmap*.

Issue: Is there anything to be gained by making single-page requests from the underlying allocator, and if so, under which circumstances on which platforms?

Another point is that the underlying allocator may itself use memory for various purposes and may have per-block overhead or suffer from various kinds of fragmentation, this is discussed in the section "Internal memory use" below.

Platform Layer: Virtual Memory

The "virtual memory" layer interacts with an underlying allocator that allows linearly addressed memory to be reserved and committed lazily, and to be decommitted irrespective of allocation order or whether a whole region or just part of a region is decommitted.

The virtual memory allocator reserves address space in 1MB chunks (unless larger requests from higher levels need to be satisfies), then commits address space in 128KB chunks (ditto). If MMgc is so configured regions can be extended by reserving address space adjacent to the existing last region; I expect that with virtual memory the number of regions is effectively very small (though there's still no guarantee of this, and for a plugin that runs in a shared address space with the browser it's possibly a dodgy assumption).

Issue: (noted by Tommy in the code) Memory is sometimes zeroed redundantly: it is zeroed by the operating system on mapping, and then again by the block-level allocator.

Action: Fix it.

Internal memory use

GCHeap subclasses GCAllocObject and through that it uses an underlying memory allocator, malloc/free on most systems, HeapAlloc/HeapFree on Windows. Allocations for book-keeping data for the garbage collector are therefore maintained on a competing heap, so to speak.

It's not clear to me at this point what the volume of that data might be. Basically it should be the blocks array and the HeapBlock and Region data structures, maybe some other odds and ends. Given that the blocks array may be 240KB for our worst case and one copy is allocated while another is live, and that there may be one HeapBlock per Block in the worst case and that a HeapBlock is 24 bytes, and that there may be 320 regions and a Region is also 24 bytes, we're looking at maybe

    2*240*1024 + (40*1024*1024)/(4*1024)*24 + 320*24 = 727KB

727KB is 1.8% of a 40MB heap, which is probably adequate if not exactly fabulous (provided fragmentation within the underlying heap is not too bad, see earlier discussion of the impact of having a single large block array). This figure scales with the heap it should be no more forbidding in a small heap.

However we may note that there are various other subclasses of GCAllocObject as well.

Issue: The total volume of under-the-radar data is not yet known, and it's not known if it is seen by the memory profiler.

Action: Clarify whether under-the-radar data are accounted for, and how that accounting is validated and reported.

Very small systems

Are there issues of very small systems (say, low-end smartphones with 4MB heaps, in the FlashLite/Air-FP market) that have not been discussed above? These systems have the following characteristics:

- Some have no MMU
- They have tiny caches (16KB), or no caches at all, and the caches tend to be ineffective (write-through)
- They have slow memories

- They have software libraries that often have not been subjected to large applications before

Issue: GCHeap appears to have a few issues / liabilities in this area:

- Allocating many 128KB regions may break the software libraries
- The use of the underlying allocator (instead of our own) for book-keeping data may break the software libraries
- The frequent reallocation and copying of the blocks array will tend to be a poor fit for the small caches and slow memories

Action: Investigate whether we can configure GCHeap so that it does not rely on the underlying allocator in the way it does now, for example by allocating a fixed-size heap with static data arrays.