eheap vs. dlmalloc

by Ralph Moore

February 2016

eheap is a new heap optimized for use in embedded systems. dlmalloc has been available for a long time and is widely used. It efficiently covers a large variety of application requirements and is optimized to provide high-performance for Big applications on GPOSs.

Introduction

Although **dlmalloc** can be used for embedded applications, it is really intended for Big applications running on GPOSs such as Windows or Linux. For example, the GNU C compiler has allocation peaks of up to 2.5 MB and valleys down to 0.6 MB. During the peaks there can be as many as 25,000 small 24-byte chunks allocated. Simultaneously there can be many big chunks over 100 KB in use. Other applications may ramp up heap usage, have steady usage, or have peaks and valleys like GCC. Modern applications far exceed these statistics. [ref 1]

It has been shown to be impossible to define one allocation strategy that works well for all possible kinds of applications. dlmalloc is designed to provide reasonable performance over a wide range of requirements and to deal with gargantuan heaps.

eheap provides high performance, adaptability, and safety for embedded systems running on RTOSs or standalone. Embedded applications have the following characteristics:

- Wide range of RAM sizes from very small to large.
- Expected to run forever.
- High performance and deterministic operation are required.
- High priority tasks must be able to preempt and run quickly.
- Small code size is often necessary.
- Each embedded system has a relatively narrow range of heap requirements.
- Embedded systems have significant idle time.
- There is a growing need for self-healing

eheap and dlmalloc have many similarities. Both have similar *physical* and *logical* structures and both conform to the ANSI C Standard for malloc(), free(), realloc(), and calloc(). In addition, they have some algorithms in common and perform basic heap functions in similar manners. However, differences in intended usage do result in implementation differences. In particular, eheap is customizable, whereas dlmalloc has a fixed structure with respect to its fundamental operations.

Heap Physical Structure

The memory area allocated to a heap is divided into *chunks* of various sizes. Each chunk has a *header* and its remainder is available for use as a *data block*. Chunks are multiples of 8 bytes in size and they are aligned on 8-byte boundaries. There are two main types of chunks: *inuse* and *free*. eheap provides an additional type called a *debug* chunk. It is discussed in the debug section, below. An inuse chunk is one that has been allocated to an application via a malloc(). The application uses the data block of the chunk; it does not access the chunk header. inuse chunks have headers of 8 bytes. If the average data block is small, say 24 bytes, the header represents a significant overhead of 25%. If the block could be as small as 8-bytes, the overhead would be 50%. For embedded systems, blocks this small are probably better served from bare block pools, for which the overhead is 0.

For **dlmalloc**, the second word of the header is the *chunk size*, in bytes. The first word of the header is the *footer* of the previous chunk, if it is free. It is the size of the previous chunk and is used to facilitate merging the previous chunk with the current chunk, when it is freed. The footer of a free chunk is put into the following inuse chunk in order to facilitate 8-byte chunk alignment. The chunk before a free chunk cannot be free because free chunks are merged immediately when freed. So the first header word is not used in free chunks. It is also not used in inuse chunks that are preceded by other inuse chunks. As a consequence, a heap can be traced forward from chunk to chunk, by adding each chunk size to its address, but it cannot be traced backward, since many previous chunk sizes are missing.

For **eheap** the first word of the header is a forward link to the next chunk and second word of the header is a back link to the previous chunk. Hence the heap can be traced forward and backward. This feature is used to increase heap reliability, as discussed in the heap trace section, below.

Free Chunk Headers

Both heaps are bin type heaps. Bins hold pointers to free chunks and they are discussed in the next sections. For both heaps, free chunks have larger headers and the remaining space in the chunk is not used. The dlmalloc free chunk header for chunks in small bins consists of: footer, size + PINUSE, next pointer, and previous pointer. It occupies 16 bytes. When the chunk is allocated, the third and fourth words of the header are overwritten with data. Hence, the minimum data block is 8 bytes in a 16-byte chunk. Since the size must be a multiple of 8, its low 3 bits are always 0 and thus are not needed. This permits bit 0 of the size to be used for the INUSE flag to indicate if the chunk is inuse or free. The dlmalloc free chunk header for chunks in large bins adds: left child pointer, right child pointer, parent pointer, and bin index.

The eheap free chunk header consists of forward link (fl), backward link + flags (blf), size, free forward link, free backward link, and binx8. It occupies 24 bytes. All but fl and blf are overwritten with data when the chunk is allocated. Hence, the minimum data block is 16 bytes in a 24-byte chunk. The flags are: DEBUG (bit 1) and INUSE (bit 0). The free chunk header is the same for chunks in small and large bins.

Heap Logical Structure

Bin arrays comprise the logical structure of both heaps. Each bin stores chunks of a certain size or of a range of sizes. Once either heap has been running for a while, the bins should serve as the main source of chunks for allocations. It is helpful to look at the bin arrays as a second heap dimension, with the physical heap being the first heap dimension. Sequential heaps, which have only the first heap dimension, require sequential searching to find a big-enough chunk, and thus allocations are often very slow. Bins, on the other hand, allow chunks to be quickly plucked from anywhere in the heap with minimal or no searching.

Both heaps have *lower* and *upper* bin arrays of free chunks.

Small Bin Arrays (SBA)

The lower bin array of both heaps is a *small bin array* (SBA) in which each bin holds only one size. Each bin consists of a forward link and a backward link. The SBA consists of bins of sequential chunk sizes, starting with the minimum chunk size. For dlmalloc there are 30 bins covering chunk sizes of 16, 24, 32, …, up to 248 bytes. These correspond to bins 2 thru 31. (Chunk sizes for bins 0 and 1 are too small). For eheap, the SBA may have from 0 to 31 bins. A typical eheap has bins for SBA sizes: 24, 32, …, up to 8*n. The advantage of the SBA is that initial bin selection is very fast (binno = size/8 or size/8 – 3) and the first chunk is always exactly the required size.

Average chunk sizes used by most applications, tend to be very small (e.g. 16 to 32 bytes) [ref 1], so optimizing small chunk mallocs and frees is important for performance — especially for object-oriented code, such as Java and C++. However, this is not necessarily true for embedded systems written in C.

dlmalloc Upper Bin Array

dlmalloc uses a power-of-two algorithm to structure its upper bin array. An upper bin is selected by the most-significant 1 bit in the requested chunk size. For example, 0x100 has bit 8 set and it selects the smallest large bin. This bin contains chunk sizes from 256 up to 504 bytes, in multiples of 8. The next bin is selected by bit 9 (0x200). It contains chunk sizes from 512 up to 1016 bytes. For simplicity, these bins are named *bin8, bin9*, etc., herein. The top bin is *bin24*, which covers chunk sizes 16 MB and up. Note that bin8 contains 32 chunk sizes, bin9 contains 64 chunk sizes, and so on. The fact that each successive bin has twice as many chunk sizes, as the previous bin, is a consequence of the power-of-two algorithm.

In order to speed up finding the *least-big-enough chunk*, each upper bin contains a *left binary tree* and a *right binary tree*. As a consequence, the upper bins of dlmalloc are referred to as *tree bins*. For bin n, bit n-1 selects the left tree if 0 or the right tree if 1.

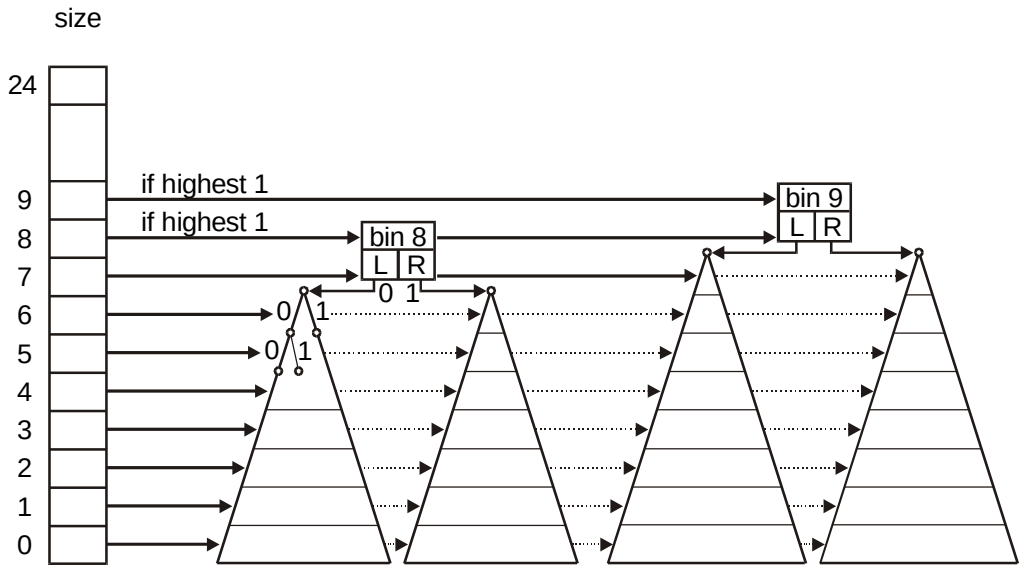

Fig 1: dlmalloc Upper Bins and Trees

Trees consist of *levels* of nodes. At each level, in each node, the next lower bit of the size is used to select the *left child* or the *right child* in the level below. Free chunks serve as tree nodes and child pointers are contained in their headers (see *free chunk headers*, above). Thus bin n may have up to n - 1 levels. However, the number of levels in an actual tree depends upon how many chunks and how many different sizes are present in the tree. For sizes that might practical in embedded systems, such as 10,000 (0x2710) bytes, bin13 would be used (0x2000 = bit 13 set) and it could have up to 12 levels.

To understand the operation of a binary tree, consider bin8.

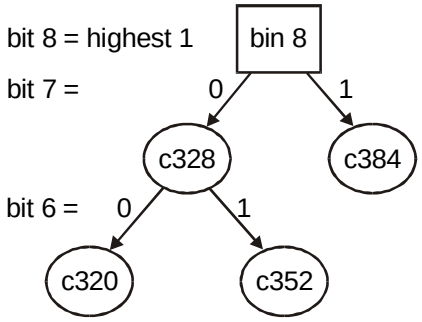

Fig 2: bin8 Example

We will examine freeing various size chunks to this bin. Starting with size = 1 0100 8000b (c328)[1], since bit 7 = 0, this chunk will be linked to the left- child pointer of the bin. Then 1 1000 0000b (c384) will be linked to the right-child pointer of the bin, since its bit 7 = 1. Next, 1 0110 0000b (c352) will be linked to the right-child pointer of c328, since its bit 6 = 1, and 1 0100 0000b (c320) will be linked to the left-child pointer of c328, since its bit 6 = 0. At this point c328 is an *internal* node and c320, c352, and c384 are *leaf* nodes.

[1] c = chunk, 328 = 0x148 = 1 0100 8000b

A leaf node has no other node linked to it. Freeing another chunk to the bin may cause a leaf node to become an internal node. However, if a same-size chunk is freed to the bin, it will linked to the existing same-size node, via the next and previous pointers in it and in the node. Hence, there is a space for every possible multiple-of-8 size in the bin and there is space for an unlimited number of chunks of the same sizes.

When a chunk is allocated from a tree, the tree is traced using successively lower bits of the desired size until a leaf node is reached. Then the *least big enough* chunk along the route followed is taken. If it is an internal node, it is replaced by a leaf node below it. This clever strategy avoids what otherwise could be a complicated and time-consuming restructuring of the tree. Using the above tree for an example, if malloc() is looking for a 312-byte block, it will trace the above tree to the c320[2] leaf node and take it. But if it is looking for a 320-byte block, it will take the c328 internal node, and c328 will be replaced with an existing leaf node from below, such as c320. In that case, the bin8 left-child pointer will now point to c320, and the c320 right-child pointer will point to c352. The c320 left-child pointer will be NULL.

eheap Upper Bin Array

eheap takes a different approach. The upper bin array may consist of any combination of large bins and small bins. Large bins contain multiple sizes and small bins contain single sizes. The structure of all bins is defined by the smx_bin_size[] array which specifies the lower size limit of each bin. Unlike dlmalloc, upper bins need not have exponentially increasing sizes. All bins can contain a fixed range of sizes up to a maximum size. For example, the first 12 bins could range from 24 to 120 bytes in 8-byte increments and the next 15 bins could range from 128 to 2040 bytes in 128-byte increments. These upper bins would each contain 16 sizes. The top bin (bin28 in this example) would contain all sizes >= 2048.

The initial large bin to test is located by doing a binary search using the desired size vs smx_bin_size[] sizes. For 15 large bins, up to 4 comparisons are required. dlmalloc's method to find the initial large bin is faster. However, since smx_bin_size[] is most likely in fast local memory or in the cache, the eheap search is fast.

Small bins in the upper bin array can be defined for frequently-used sizes (e.g. Ethernet packets). Otherwise, large bins are searched for the first big-enough chunk. It is taken and split, if necessary. Least-big-enough fit is achieved by sorting the large bins during idle periods. This is not perfect, but it is a good fit for embedded systems, which must have a high percentage of idle time in order to meet deadlines during peak loads. This is different from general-purpose systems, which run at maximum speed until done, with no idle time.

A heap for a small embedded system might have a bin size array of only: 24, 32, 40, 48, 128, 256, and 512. Then the SBA would consist of bins 0, 1, and 2, with chunk sizes 24, 32, and 40, respectively; bin3 would contain sizes 48 to 120, bin4: sizes 128 to 248, bin5: sizes 256 to 504, and bin6: sizes 512 and up. If a need develops for packets of exactly 128 bytes (136-byte chunks), a small bin could be added to the bin size array as follows: 24, 32, 40, 48, 136, 144, 256, 512. The new bin4 would contain only 136-byte chunks. This eliminates search time within

[2] Remember that the chunk is at least 8 bytes larger than the block it contains.

the bin thus resulting in faster allocations. The foregoing is an example of how an eheap bin structure could be matched to the needs of a specific embedded system.

Special Chunks

In addition to bins, dlmalloc has a *designated victim* chunk *(dv)* and a *top chunk (tc)*; eheap has a *start chunk (sc), a donor chunk (dc), a top chunk (tc),* and an *end chunk (ec)*. These special chunks are never put into bins.

Following heap initialization for **dlmalloc**, the entire heap space is contained in tc. During initial operation, most mallocs come from tc and frees go to bins. Eventually mallocs start coming mostly from bins. When the first chunk from a bin is split, the remnant becomes dv. Thereafter, dv is always the latest remnant from a split chunk. It is used for small chunk allocations when the initially selected SBA bin is empty. As the heap stabilizes, most small chunk mallocs will come from SBA bins and some from the dv. tc is held in reserve as the last resort for large chunks. It can be expanded by requesting large block allocations from the OS.

Following heap initialization for **eheap**, the heap consists of sc, dc, tc, and ec and cmerge is OFF. sc and ec are permanently-allocated, 16-byte, inuse chunks that mark the heap ends. dc and tc sizes are controlled by user-defined configuration constants. dc backs up the SBA, like dv, and it is usually much smaller than tc. tc operates the same as for dlmalloc. It also can be expanded, if necessary. In this case, heap expansion is intended as an emergency measure to avoid system failure, rather than being an expected occurrence.

dv and dc, although apparently similar, operate in different ways. dv is intended to improve cache hits for small chunk allocations, since it is the remnant from a recent larger chunk allocation that required a chunk split. In heap theory, this is called *localization*. Following a larger chunk split, the new remnant becomes dv and the old dv is put into the appropriate bin for its size. dv also helps to improve small chunk allocation times since it is used instead of a larger bin, if it is big enough. This probably helps to compensate for SBA bin leakage due to chunk merging.

The rationale for dc is somewhat different. Initially, most small chunks will come from dc and most large chunks will come from tc. This serves two purposes: (1) faster chunk allocations, (2) segregation of the physical heap: small chunks will be below dc and large chunks will be above it. This helps to reduce allocation failures caused by small inuse chunks blocking the merging of larger free chunks. It also helps localization, since most small blocks will come from dc. This not only improves access to chunk headers by eheap, but also to data blocks by application code. Of course, in time, there will be cross-over chunks into the other region due to depletion of dc, merging of small chunks, and splitting of large chunks.

When dc becomes too small, it can be replenished simply by freeing it and then mallocing a larger chunk to it. This is similar to dv replenishment. The strategy for doing it is up to the user. Alternatively, use of dc can be turned off.

Heap Operations

malloc()

For **dlmalloc**, allocation of small chunks follows the path: SBA -> dv -> larger bin -> tc. Allocation of a large chunk follows the path: upper bin -> tc. **eheap** allocation is pretty much the same: SBA -> dc -> larger bin -> tc and: upper bin -> tc. Initially most bin allocations will fail and default to calving chunks from dc/dv or from tc, depending upon their size. Calving from dc/dv or tc is actually faster than a bin allocation. Over time, bin populations will increase and allocations will come primarily from the bins. At that point, dc/dv serves as SBA backup and tc serves as backup for the upper bins.

Other than the dc/dv difference noted above, the main malloc() difference is in the upper bins. As previously noted, these are organized as trees for dlmalloc. Binary trees are efficient at finding the least-big-enough chunk for a give size. Doing this, rather than taking the first big-enough chunk helps to reduce fragmentation.

Since bins are organized by size, chunks in the same bin are unlikely to be physically near to each other. And since child pointers are located in the chunks, tree tracing is likely to result in a cache miss per node. In most modern systems, external DRAM is much slower than the processor — often by a factor of 10 to 20. If 14 tree levels must be searched, for example, the search time could be very long. However, as pointed out by one author, processing times for large blocks are correspondingly long, so a slow malloc() may not increase the overhead percentage.

eheap has different large-bin allocation strategy: it takes the first big-enough chunk. Even when the bins are well-sorted by increasing size, several chunk accesses may be necessary to find a big-enough chunk. Of course, cache misses during this process are equally likely as for trees. The saving grace is that large bin sizes can be adjusted to fit specific embedded system needs. Thus, if a particular bin tends to require too many searches, it can be split into fewer sizes, all the way down to only one size.

Other strategies are also possible. For example: if a task gets many same-size chunks, then frees them all at once, turning on cmerge is likely to cause many of the chunks to be merged and put elsewhere, as they are freed.

It is important to recognize here that a given embedded system is unlikely to require a large variety of chunk sizes. Hence the eheap upper bin structure can be adjusted to efficiently accommodate the sizes that are required. If this is not so, it may be necessary to increase the number of bins supported by eheap or the specific application may need to use dlmalloc — no single heap solution solves all requirements.

free()

For **dlmalloc** free() need only add its chunk to the start of a small bin, but for a large bin, it must trace through the tree to locate the correct position for the chunk. As previously noted, this may require from 0 to n chunk accesses, depending upon the number of chunks in the bin and how many different sizes are present. In effect, a free() operation sorts the bin to facilitate least-big-

enough allocations. Due to the fact that each chunk access is likely to result in a cache miss, this can result in slow free() operations.

eheap free() to a small bin is the same as for dlmalloc. For a large bin, it much simpler: if a chunk is smaller than the first chunk in the bin, it is put ahead of that chunk, otherwise it is put after the last chunk in the bin. This requires up to two chunk accesses. Bin sorting is deferred until the eheap bin sort function runs during idle time. Hence eheap free() is likely to be faster than dlmalloc free() under normal circumstances.

Finding the Right Bin

As previously noted, for both dlmalloc and eheap, finding the initial SBA bin is a simple calculation based upon the requested size. However, what if the bin is empty and dc/dv is too small? Both heaps use bin maps to find the next larger, occupied bin. When a chunk is loaded into a bin, the bin's map bit is set. When a bin becomes empty, its map bit is cleared. A simple calculation magically converts the map into the bin number of the next larger occupied bin.

eheap has one 32-bit bin map, *bmap*, for all bins. Hence the maximum number of bins is limited to 32. This can be increased to 64, or even more, if needed. However, there is a small performance penalty, so bmap is being left at 32 for the time being, which may be enough for embedded systems. dlmalloc has *smallmap* for the SBA and *treemap* for the upper bins to support its total of 47 bins (30 small + 17 large).

Finding the correct initial large bin for dlmalloc is a size calculation using treemap. Finding the correct initial large bin for eheap is performed by doing a binary search on smx_heap_size[] vs. the requested chunk size. This may require up to n comparisons, where 2^n is the next power of 2 >= number of large bins. At most, this is 5 for a 32-bin system. Since the smx_heap_size[] can be located in on-chip RAM or ROM, this search is fast.

eheap Bin Sorting

As discussed previously, eheap operation is improved if large bins are well sorted. Then malloc() will take the least-big-enough chunk in the bin, if there is one. This is generally considered a good way to reduce fragmentation. Also, if a large bin is well-sorted, searches should be shorter, on average. Though causing suboptimal performance, unsorted bins do not cause heap failures. Hence, shabby large bins may be acceptable during periods of peak system loading.

Unlike Big applications running in the Cloud, embedded applications must have significant idle time to deal with peak loads due to asynchronous external events occurring simultaneously. Also they need spare processing time to handle new requirements likely to be added in the future. eheap provides an efficient bin sorting service, which normally runs from the idle task. Hence, in theory, large bins should almost always be perfectly sorted. If not, the bin sort service can be called from a higher priority task, or the nodes per sort can be increased.

eheap bin sort uses a *bubble sort with last turtle insertion* algorithm. The *last turtle* is the last chunk that may be smaller than a chunk ahead of it. It is called a "turtle" because it moves very slowly in a normal bubble sort. The following is an example of a bin sort:

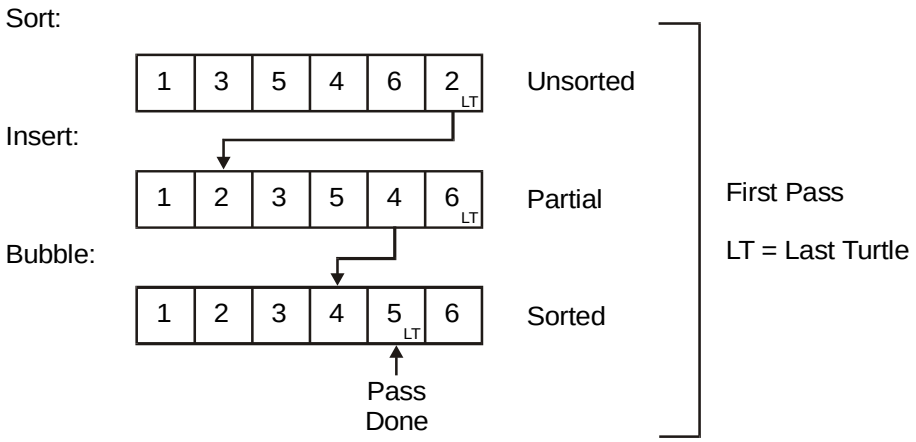

Fig 3: Bin Sort in One Pass
(Second Needed to Verify)

As each bubble sort progresses (from the front of the bin), the current last turtle is put ahead of the first larger chunk found. The chunk that was ahead of the last turtle becomes the new last turtle. This chunk could be any size that fits into the bin. After k bubble sort passes, the last k chunks are guaranteed to be sorted and larger than any preceding chunks. Hence the last turtle is k + 1 chunks from the end of the bin free list and each pass ends with the current last turtle. Note that this could be several chunks ahead of where the last turtle was at the start of the pass. If during a pass no chunk is moved, the sort is complete.

When free() puts a large chunk at the end of a sorted bin, last turtle insertion will immediately move it to its most likely correct position during the first pass. If bin was already sorted, this will complete the sort.

eheap bin sort is a *system service routine* (SSR). Hence, it cannot be preempted. In order to not interfere with higher priority tasks, bin sort runs incrementally. That is, it processes a user-controlled number of nodes per run, then returns to allow preemption by a higher priority task.

A bin sort map, *bsmap*, similar to bmap used for malloc(), determines what bins to sort. When free() puts a chunk at the end of a bin, the bsmap bit is set for that bin. The bit is never set for small bins, nor when chunks are put first in large bins. When the bin sort starts for a bin, its bsmap bit is reset. If the bsmap is set by a preempting free(), the sort will start over on the next run.

The bin sort function can be called to sort a specific bin or all bins. Either way, it returns TRUE when the job is done. Note that unless bin sort is being frequently preempted, all chunks in the bin free list are likely to be in the data cache and bin sorting should be fast. If bins are not being sufficiently sorted, bin sort can be moved to a higher priority task.

Free Chunk Merging

An important difference between dlmalloc and eheap is that dlmalloc always merges adjacent free chunks, whereas eheap permits *deferred merging*. eheap merging is controlled by the *cmerge* mode, which the application can turn ON and OFF. Following heap initialization, it is OFF. If the amerge mode is ON, cmerge will be turned OFF when heap used reaches an upper

limit, set by the user; it will be turned back on if heap used reaches a lower limit, set by the user. Other algorithms can be implemented to automatically control merging, based upon factors such as number of free chunks, average size of free chunks, total space in bins, etc. Long-run testing of these various schemes per embedded system is necessary to determine which works best and how much *heap margin* (i.e. unused / total space) is necessary to assure that heap failure does not occur. Doubtless, one strategy will fit a specific embedded system better than others.

The premise for deferred merging is that embedded systems tend to use a relatively static population of blocks. If merging is enabled, the *leaky bin syndrome* occurs. For example, suppose a 24-byte chunk is freed to bin 0 and a physically adjacent 48-byte free chunk resides in bin 3. What will happen is that these chunks will be merged into a 72-byte chunk, which will be put into bin 6. This obviously is not conducive to best performance if the application needs 24 and 48-byte chunks and not 72-byte chunks. Additionally dequeueing the two chunks and merging them requires additional time during free().

Fragmentation

When a heap becomes overly *fragmented*, small inuse chunks may prevent medium chunks from being merged in order to allow larger block allocations. When this happens, *heap failure* is the result. This is not necessarily catastrophic, especially in embedded systems. The task requesting the large block could be rescheduled to run at a later time, then try again. Another approach is for less important tasks to free their heap blocks, with merging enabled. Obviously, mission-critical tasks should not use the heap or they should only request small blocks, which are always available.

Both dlmalloc and eheap try to hold reserve space in tc for large chunk allocations. However, over time, tc is likely to be eaten up. Both heaps allow tc to be extended into reserve memory. For **dlmalloc**, heap extension is a frequent occurrence with Big applications, and dlmalloc is designed to obtain large blocks from the OS. It can also release such blocks back to the OS, when they are no longer needed.

Deferred merging, which is implemented in **eheap**, is generally viewed as increasing fragmentation, thus making heap failure more likely. If this is a problem, eheap permits merging to be enabled at all times, like dlmalloc. However, as noted above, this may reduce performance due to leaky bins. Hence, merge control, as discussed above, may provide better performance.

If merge control fails to stem heap failure, an eheap recovery service is provided to search the heap in order to find and merge adjacent free chunks to satisfy a failed allocation. If this fails, an eheap extension service can be used to extend the heap into reserve memory, such as a slower RAM area. Unfortunately, reserve memory may not be available in a particular embedded system. If this and other strategies, such as those suggested above, do not work, then in the worst case it may be necessary to shut down tasks using the heap in a controlled manner. This might solve the problem and these tasks might be resumed later. If not, eheap permits reinitializing the heap, then restarting all tasks using it.

Reliable Operation

Debugging

Embedded systems are typically designed by small teams of engineers who have too much to do. However, delivering bug-free software is important because bugs in embedded systems can have serious, if not tragic, consequences. Debugging often makes or breaks projects and the heap may frequently be the crux of the problem. Understanding how the heap works and having good visibility into it is important, but heap operations can be complicated and difficult to understand.

To aid debugging, eheap has a *debug mode*. When debug is ON, allocations create *debug* chunks instead of *inuse* chunks. A debug chunk is a special form of an allocated chunk (in fact, its INUSE flag is set). Since debug chunks are typically much larger than inuse chunks it may not be possible for all allocated chunks to be debug chunks — especially in small heaps. For that reason they can be selectively controlled by turning debug mode ON to create debug chunks and OFF to create inuse chunks. A heap may contain any mixture of debug, inuse, and free chunks. As a system grows, debug chunks may be used for new code and inuse chunks used for debugged code.

A debug chunk header is the same size as a free chunk header. It consists of: fl, blf, size, time, onr, and fence. Time is when the chunk was created, and onr is the task or LSR that created it. These can be used to track down the sources of heap leaks (e.g. a chunk owned by a task that has been deleted or a chunk that should have been freed long ago). The fence is a known pattern such as 0xAAAAAAA3, which is specified at compile time. It can be any pattern, as long as bits 0 and 1 are set. In addition, any number of fences can be put before and after the data block. Fences serve two purposes:

(1) to allow normal operation to continue after a block overflow or underflow has occurred, and
(2) to permit seeing the overflow pattern. This feature, by itself, is valuable.

eheap also has a fill mode, which fills unique patterns into data blocks when allocated, free blocks when freed, and dc and tc when they are initialized or increased in size. These are very helpful to view the heap in a memory window in order to figure out what is happening. It is easier to see free, inuse, and debug chunks and to see how much of an inuse data block has been used. It is also helpful to see how much of dc and tc have been used.

In addition, heap service parameters are checked, and invalid parameters are reported. In most cases, services abort if an invalid parameter is found. These checks can be disabled for malloc() and free() if higher performance is desired, however this is not recommended, even for released systems.

Self-Healing Heaps

Simultaneous progress toward ever smaller semiconductor feature sizes, massive deployment of Things and remote routers, and moving more processing and intelligence into them from the Cloud spells Trouble. As fewer electrons differentiate a 1 from a 0, environmental conditions, such as energetic neutrons from cosmic ray collisions in the upper atmosphere and electromagnetic radiation from severe sunspot activity, to name a few, will cause more and more bit flips. These problems increase with altitude and latitude and are already a problem for high-

flying aircraft. Ironically, the “Cloud” can be put under thick slabs of concrete, but Things and remote routers cannot. [ref 2] Malware and cyber-attacks make self-healing even more important.

A data block may contain valuable information, but each data item can be subjected to reasonableness checks and discarded if the checks fail. Data being transmitted or received is generally protected by error codes and other means. However, one bad heap pointer can puta system “into the weeds”. Heaps are particularly susceptible to bit flips. Considering just inuse chunks, which should predominate in a running system, if the average chunk size is 32 bytes, as suggested in [ref 1], then the target density is 25% (i.e. 100 x pointer space / total space). If 64 bytes is more reasonable for an average chunk in an embedded system, then the target density is 12.5% and adding 2.5% for the free chunks, which have more pointers, gives 15% — still a large target for random particles and disturbances.

Implementing self-healing requires making reasonable assumptions. For eheap, it is assumed that only one word can be damaged at a time. It is further assumed that a heap has thousands of chunks and that billions of normal heap operations will occur between events that damage the heap. Hence, the odds are good that there is sufficient time to fix the heap before malloc() or free() step on a rotten pointer. This little extra bit of protection could be vital if as many systems are running in unprotected environments, as currently projected for the IoT.

Self-healing is implemented in eheap by heap scanning and bin scanning, which are discussed next.

Heap Scanning

The eheap heap scan service is intended to be run from the idle task so that it will not consume valuable processing time. It scans each chunk from the start of the heap to the end of the heap, fixing broken links, flags, sizes, and fences, as it goes. In a debug version, the scan stops on a broken fence so that the fence can be studied for clues to what happened. In a release version, the fence is fixed. Fixes are reported and saved in the event buffer for later analysis of systems in the field. It is important to note that during scans, all pointers are heap-range tested before use, in order to avoid *data abort exceptions*, that might stop the system.

If a broken forward link is found and the chunk is either a free chunk or a debug chunk, the size field is used to attempt a fix. (Size is effectively a redundant forward link.) Otherwise, a backward scan is started from the end of the heap. It proceeds until the break is found and the forward link is fixed. While back scanning, other broken forward links are also fixed, if encountered. However a broken backward link stops the backward scan and a *bridge* is formed between the chunk with the broken forward link and the chunk with the broken backward link. When this happens, many chunks of all kinds may be bridged over. This serves to allow the scan to complete and may give the system a chance to limp along and to possibly fix itself, but generally it does not fix the problem and thus **heap broken** is reported.

Note: two broken links violates the basic assumption of only one damaged word at a time. However, bridging is easy to implement, so it was done.

Heap broken can be dealt with by reinitializing the heap and restarting all tasks that use it. If high-priority, mission-critical tasks use only block pools, this can be a workable, worst-case

solution — valuable data might be lost, but the system will continue performing its vital mission. By taking action before malloc() or free() encounter a broken link the system is saved from going off into the weeds and failing.

As noted in the introduction, it is absolutely essential that high-priority tasks not be blocked from running for too long. Since heap scan is a system service, it cannot be preempted. Consequently it has been designed to operate in an incremental fashion. When called it performs a *run* of num chunks per call. Thus it must be called repetitively until it is done. Heap scan permits specifying different numbers of forward scan chunks, fnum, and backward scan chunks, bnum. This is because backward scan is much faster and also more urgent, once a break has been found. So, for example, fnum might be 1 or 2, but bnum might be 100 or more.

The scan operation will go the specified number of chunks then return FALSE, unless it is done or an error is encountered. It can be called repetitively with

```
while(!heap_scan()) {}
```

until done or an error is encountered or it can be called once per pass through idle_main(). Since heap errors are likely to occur very infrequently, heap scan normally runs as a slow, steady patrol looking for trouble, and fixing what it can.

Bin Scanning

Whereas bins are in local memory and may be less susceptible to bit flips, most bin free list pointers are in the free chunks out in the heap and thus are just as susceptible to damage as are heap links. Thus bin free list scanning is necessary to provide a consistent level of protection. Bin scan is similar to heap scan. It is incremental, scans doubly-linked free chunk lists, and fixes broken links, if it can. It has three parameters: bin number, fnum, and bnum. Like heap scan, it returns FALSE until it is done with a bin or an error is found. If a preempting malloc() uses a bin being scanned, the bin scan is restarted.

Bin scan scans backward to fix broken forward links. Fixes are reported. Double breaks are bridged, like heap scan. In this case, the bridged chunks are no longer available to be allocated, but heap operation can continue normally, if cmerge is OFF. In this case, bridging is a partial solution, but heap reported to allow a better solution to be implemented. If merging is ON, free may fail due to encountering a bad pointer when it attempts to merge a bridged chunk. In some cases, a merge will proceed ok.

Peeks and Sets

Heap modes can be checked via heap peeks. These include: auto merge, bin scan forward, debug, fill, heap initialized, heap scan forward, cmerge, and use dc. The following heap modes can be directly set: auto merge, debug, fill, cmerge, and use dc. The ability to view heap modes helps to diagnose heap problems. This combined with the capability to change heap modes permits dynamic heap control to do things such as automatic merge control, turning debug mode on or off, enabling or disabling fill, etc. Bin peeks allow obtaining information on bins, such as: number of chunks in the bin, first chunk, last chunk, bin size, and total size of chunks in the bin. This information can be used to improve performance.

Performance

Localization

is intended to maximize cache hits for systems where the heap is in external memory, which is usually case for large heaps. The strategy is to allocate blocks that are together in time to be together in space. dlmalloc does this by allocating from the designated victim, dv, whenever a small bin miss occurs. Since dv is the remnant of the last chunk split, it may often be too small and allocation will come from a larger bin or tc.

eheap does a similar thing by using the donor chunk, dc. Initially, dc is a large piece of the heap and most SBA-size allocations come from it, since the bins are empty. Hence, good localization is achieved, during this period. When nearly all allocations are coming from bins, localization could still be good if the data cache is large and related small chunks where allocated together (i.e.” together in time”). Since these chunks would have come from the same section of dc, cache operation could be very good — not only for chunk headers by eheap, but also for data blocks by application code. In systems where very large numbers of small blocks are needed, allocating a large portion of heap space to dc may greatly improve performance.

It should also be noted that frequent task switches, which are typical of embedded systems, tend to undermine any localization strategy — both for dlmalloc and for eheap. Hence a high degree of localization may be difficult to achieve unless special measures are taken in the application. Block pools may be a better choice if localization is very important.

Upper Bin Performance

For **dlmalloc**, finding the least-big-enough chunk may require as little as no searches or as many as n searches, depending upon how many chunks are in bin n and how their sizes relate. Similarly **eheap** may require no searches or potentially unlimited searches, depending upon how many chunks are in bin n, how their sizes relate, and how well they are sorted. Excessive eheap search times, can be eliminated by splitting the bin or by enabling merging, when large numbers of chunks are freed, so they will coalesce into larger chunks, or even into dc or tc. Also, bins that have too many or too few chunks can be adjusted during idle time. In addition, care can be taken minimize this problem in the application design.

Unlike dlmalloc, which must deal efficiently with all possible chunk sizes, eheap need deal only with the sizes used by a specific embedded application. This allows adjusting the sizes of upper bins to fit the requirements of a specific system. As a consequence, some bins may cover large size ranges, but contain few chunks, whereas other bins may cover small size ranges, even a single size, yet contain many chunks. In addition, the SBA can be adjusted to allow more or fewer large bins. In principle, it should be possible to achieve very fast large bin mallocs for a specific embedded application.

Performance Measurements

Performing meaningful performance comparisons of eheap to dlmalloc is more difficult than might be expected due to the high degree of optimization of dlmalloc. For example, one might think that bp1 = malloc(32), bp2 = malloc(40), and free(bp1) would put cp1 (the bp1 chunk) into bin 5, then free(bp2) would dequeue cp1 from bin 5, merge it with cp2, and put cp1+cp2 into bin

11. Almost correct — bin 5 still points to cp1. Then bp1 = malloc(32) would dequeue cp1+cp2 from bin 11, split out cp1, and put the remant into bin 6. Wrong — the remant is put into dv and bin 5 and bin 11 still point to cp1, even though it is inuse. Then free(32) does not result in cp1+cp2 being put back into bin11; instead it is put into dv. However, bin 5 & 11 still point to it. bp1 malloc(32) results in bp1 being calved from dv.

Hence, the above sequence of operations does not test dlmalloc small bin operation, as would be expected. Instead, it tests calving and merging with dv. However, in actual operation, bp1 and bp2 are likely to be separated from dv by one or more inuse chunks. Furthermore, dv is likely to already have a chunk which must be enqueued in a bin in order to replace it. Hence the test must be more sophisticated to get the expected result.

The foregoing example illustrates that a good understanding of dlmalloc is necessary in order create tests that work as expected. Unfortunately, tracing dlmalloc code is notas simple as one might hope. For example if smallmap indicates that a chunk is in bin 5, it is actually in bin 10. However since the bins are defined as chunks, bin 10 contains *prev_foot*, bin 11 contains *head*, bin 12 contains *fd* = forward chunk pointer, and bin 13 contains *bk* = backward chunk pointer. Thus bin 12, which is for the next chunk size, points to the chunk in bin 10, however the chunk itself points to the bin 10. In addition, dlmalloc does not change bin pointers unless it has to. So if the chunk in bin 10 is allocated, bin 10 still points to it. To make matters even more difficult, much of the dlmalloc code consists of macros and thus can be traced only in assembly language — good luck!

For typical SBA operation, malloc() is slightly faster for dlmalloc than for eheap and free() is slightly faster for eheap than for dlmalloc, if merging is not enabled in eheap. Hence SBA operation is a push. Large bin coperations depend upon the variety of large chunk sizes being used and the number of free large chunks, at a given time. Generally eheap free() will be faster because eheap defers bin sorting to idle time, whereas dlmalloc free() does tree building at the time. Tree building is susceptible to cache misses when tree links are accessed for tracing and performance can be pretty bad. eheap malloc() times depend largely upon how well upper bins have been adjusted to the requirements of a system. For example, if a large bin size has been set equal to a frequently-used chunk size, then malloc() for that size is likely to be much faster for eheap than for dlmalloc. If there is a large variety of chunk sizes and a large number of free large chunks, then malloc() for dlmalloc is likely to be faster than for eheap. (Note: this may be a good reason to increase the eheap bin limit.)

Obviously, sophisticated simulations are needed to arrive at meaningful large bin comparisons. However, it has been shown that such simulations are pretty much a futile exercise [ref 1] and that more meaningful results come from running actual applications with the heaps to be compared. With eheap customized to an actual embedded system, its overall performance is likely to be better than dlmalloc. If not, the difference may be insignificant or within the bounds of required performance. In these cases, the extra features of eheap may justify using it.

In order to facilitate actual on-system comparisons of eheap to other heaps, a free source-code download of eheap is available at www.smxrtos.com/eheap. eheap will run on systems using any RTOS or even on standalone systems. Full source code permits tuning eheap to specific system requirements. Also, debug, self-healing, and other unique features of eheap can be tested.

References

1. Wilson, Paul R, “Dynamic Storage Allocation: A Survey and Critical Review”
2. Moore, Ralph C, “Feature Sizes and the IoT”