

FRAGMENTATION CAN BE A STICKY PROBLEM. HOW MEMORY ALLOCATION OCCURS DETERMINES WHETHER, WHEN, AND HOW MEMORY FRAGMENTATION BECOMES AN ISSUE.

Handling memory fragmentation

ULTIMATELY, MEMORY FRAGMENTATION leads to out-of-memory conditions, even when plenty of free memory may still in fact exist in the system. Given enough time, a system that constantly generates memory fragmentation, regardless of how little, will run out of memory. This situation is unacceptable in many embedded systems, especially in the high-availability market. Some software environments, such as OSE real-time operating systems, already provide good tools for avoiding memory fragmentation, but choices that the individual programmer makes can still influence the outcome.

“Fragmented memory” describes all of a system’s unusable free memory. These resources remain unused because the memory allocator responsible for allocating them cannot make the memory available. This problem usually occurs because free memory is scattered at separate locations in small, discontinuous portions. Because the allocation method determines whether memory fragmentation becomes a problem, the memory allocator plays the central role in ensuring the availability of free resources.

COMPILING TIME AND RUNTIME

Memory allocation occurs in many contexts. A programmer, by way of the compiler and linker, can allocate memory for data in structures, unions, arrays, and scalars as local, static, or global variables. The programmer can also dynamically allocate memory at runtime, using calls such as malloc(). When the compiler and the linker perform the memory-allocation function, memory fragmentation does not occur, because the compiler understands the data lifetime. Having the data lifetime available offers the advantage of making the data stackable in a last-in/first-out arrangement. This fact makes it possible for the memory allocator to work efficiently and without fragmentation. Generally, memory allocation performed during runtime is not stackable. Memory allocations are independent in time, which makes the fragmentation problem difficult to resolve.

Memory allocators waste memory in three basic ways: overhead, internal fragmentation, and external fragmentation (Figure 1). The memory allocator needs to store some data describing the state of its allocations. It stores information about the location, size, and ownership of any free blocks, as well as other internal status details. A runtime allocator typically has no better place to store this overhead information than in the memory it man-

ages. A memory allocator needs to adhere to some basic memory-allocation rules. For example, all memory allocations must start at an address divisible by four, eight, or 16, depending on the processor architecture. There may also be other reasons that the memory allocator assigns blocks of only certain predefined sizes to clients. When a client requests a block of 43 bytes, it may well get 44, 48, or even more bytes. The extra space that results from rounding the requested size upward is called internal fragmentation.

External fragmentation occurs when unused gaps arise between blocks of allocated memory. This situation can happen, for example, when an application allocates three blocks in succession and then frees the one in the middle. The memory allocator might reuse the middle block for future allocations, but it is no longer possible to allocate a block as large as all free memory. Provided that the memory allocator does not change its implementation or rounding policy during runtime, overhead and internal fragmentation remain constant throughout an application’s lifetime. Although overhead and internal fragmentation may be undesirable because they waste memory, external fragmentation is the embedded-system developer’s real enemy; it is the allocation problem that kills systems.

Several alternative approaches exist to define memory fragmentation; the most common is:

$$\text{FRAGMENTATION} = 1 - \frac{\text{LARGEST_free_block}}{\text{ALL_free_memory}}$$

This technique applies to external fragmentation, but you can modify it to comprehend internal fragmentation by including internal fragmentation in the denominator. Fragmentation is a fraction between zero and one. A system in which fragmentation is one (100%) is completely out of memory. With all free memory in a single block (the largest), fragmentation is 0%. With one-quarter of all free memory in the largest block, fragmentation is 75%. Here’s an example: Fragmentation is 99% in a system with 5 Mbytes of free memory, when the largest block available for

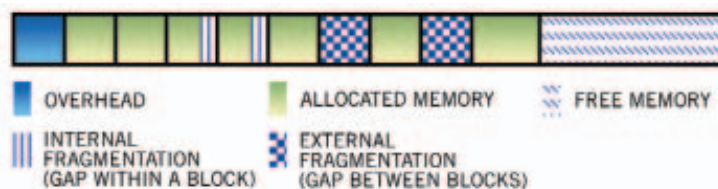


Figure 1 Memory fragmentation comes in several forms.

allocation is 50 kbytes.

The 99%-fragmentation example comes from a real-life situation that arose during the development of an embedded, soft-real-time application. This fragmentation level happened one second before the system crashed. The system had been running continuously in field tests for about two weeks before fragmentation

reached 99%. How could this situation occur, and why was it discovered so late in the project? Naturally, the system had been tested, but few of the tests lasted more than two hours. The final stress test before delivery lasted a weekend. The problem occurred because the results of fragmentation do not necessarily arise during such a short period.

The question of how long fragmentation can take to reach a critical level is difficult to answer. For some applications, under some conditions, the system reaches a steady state before memory runs out. For other applications, the system does not reach steady state in time (Figure 2). By removing the elements of uncertainty and risk, nonfragmenting memory allocators (Figure 3), which rapidly reach a steady state, help developers sleep at night. When developing long-running applications that are intended to not restart or reboot for months or years, rapid steady-state convergence is an important factor. It's essential that you be able to properly test the applications in less time than the period over which they are intended to run without interruption.

It's hard to say which memory allocation algorithms are better than others, because their merits vary with the application (Table 1). The first-fit memory-allocation algorithm is among the most common. It employs four pointers:

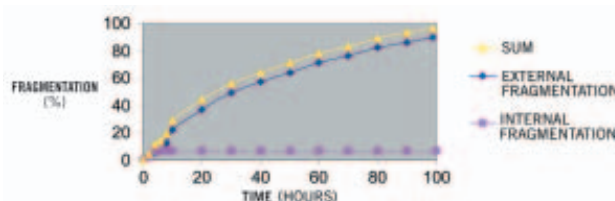


Figure 2 This case study used a first-fit memory allocator in an embedded project. The system ran continuously in field tests for about two weeks before fragmentation reached 99%.

MSTART points to the beginning of the managed memory; MEND points to the end of the managed memory; MBREAK points to the end of the used memory between MSTART and MEND; and PFREE points to the first free memory block, if one exists.

In the beginning of system operation, PFREE is NULL, and MBREAK points at MSTART. When an allocation request comes in, the allocator first checks PFREE for any free blocks. Because PFREE is NULL, a block of the requested size plus an administration header is broken off at MBREAK, and MBREAK is updated. This process repeats until the system frees a block, at which point the administration header contains the size of the block. At this time, PFREE is updated to point at the block via linked-list insertion at the head, and the block itself is updated with a pointer to the old content of PFREE to create a linked list. The next time an allocation request occurs, the system searches the linked list of free blocks for the first block that fits the requested size. Once it finds the right block, it splits the block into one part it returns to the application and another that it puts back into the free list.

First-fit is simple to implement and works reasonably well in the beginning. Nonetheless, after a while, the following situation develops: When the system releases blocks to the free list, it removes

large blocks from the beginning of the list and inserts small, leftover pieces at the head. First-fit effectively becomes a sorting algorithm that places all small memory fragments at the beginning of the free list. The free list can therefore become long, with hundreds or even thousands of elements. As a result, allocation becomes long and

unpredictable, and large allocations take longer than smaller ones. Also, the unlimited splitting of blocks creates a high degree of fragmentation. When freeing memory, some implementations join adjacent blocks if they are free. That approach helps a bit, but first-fit does nothing to improve the chances of adjacent blocks being freed at approximately the same time, unlike time and spatial co-location algorithms.

BEST- AND WORST-FIT ALLOCATORS

Best-fit functions just like first-fit, except that, when allocating a block, the system searches the entire free list for the block that is closest to the requested size. This search takes a lot longer than first-fit, but the difference in the time requirements for allocating small and large blocks disappears. Best-fit causes more fragmentation than first-fit, because the sorting tendency of placing tiny unusable fragments of blocks at the head of the list is stronger. Because of its negative characteristics, best-fit is almost never used.

Worst-fit is also seldom used. Worst-fit functions just like best-fit, except that, when allocating a block, the system searches the entire free list for the block that fits the worst match for the requested size. This approach is faster than best-fit, because it has a weaker tendency to produce tiny, unusable blocks. Consistently selecting the largest free block for

TABLE 1—COMMON ALLOCATION ALGORITHMS

Allocation algorithm	Method	Upside	Downside
First-fit	Searches for the first block that offers the requested size.	Easy to implement Works well at the beginning	Long and unpredictable High degree of fragmentation
Best-fit (almost never used)	Searches for the block that is closest to the requested size	Uniform time for allocating blocks regardless of size	Slow High fragmentation
Worst-fit (rarely used)	Searches for the block that is the worst match for the requested size	Faster than best-fit Less fragmentation than best-fit	Not enough benefits to justify the cost
Buddy	Blocks are split from and joined to "buddy blocks" based on data structure	Info on data Limits fragmentation to a certain degree	Can be difficult to write Properties may vary
Fixed size	Usually searches multiple lists of identically sized blocks	Easy to implement Counteracts fragmentation when there are few block sizes Consistently fast	Large amounts of internal fragmentation

splitting increases the chance that the remaining part will be large enough for something useful.

Buddy allocators, unlike the other allocators described in this article, don't carve out new blocks as needed from the beginning of the managed memory. The defining commonality is that blocks are split and joined, but not arbitrarily.

Each block has a friend, or "buddy," from which it can be split and to which it can be joined. Buddy allocators store blocks in data structures more advanced than linked lists. Often, the structures are combinations or variations of buckets, trees, and heaps. It is hard to describe in general how buddy allocators work, because the technique varies with the selected data structure. Buddy allocators find use because of the availability of a variety of data structures with known properties. Some are even available in source code. Buddy allocators are often complicated to write, and their properties may vary. Usually, they limit fragmentation to some degree.

Fixed-size allocators are somewhat like first-free algorithms. There is usually more than one free list, and, most important, all blocks in the same free list are identical in size. There are at least four pointers: MSTART points to the beginning of the managed memory, MEND points to the end of the managed memory, MBREAK points to the end of the used memory between MSTART and MEND, and PFREE[n] is an array of pointers to any free memory blocks. In the beginning, PFREE[*] is NULL, and MBREAK points at MSTART. When an allocation request comes in, the system augments the requested size to one of the available sizes. The system then checks PFREE[*augmented size*] for free blocks. Because FREE[*augmented size*] is NULL, a block of that size plus an administration header breaks off at MBREAK, and MBREAK is updated.

These steps repeat until the system frees a block, at which point the administration header contains the size of the block. When a block is freed, PFREE[*the corresponding size*] is updated to point at the block via linked-list insertion at the head, and the block itself is updated with a pointer to the old content of PFREE[*the corresponding size*] to create a linked list. The next time an allocation request

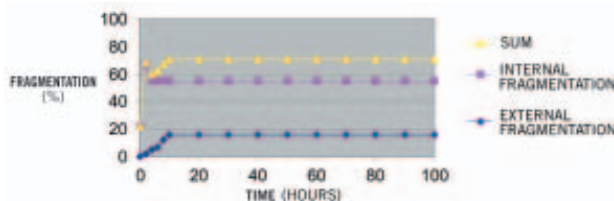


Figure 3 A nonfragmenting memory allocator reaches steady state as soon as it exercises all the parts of the application.

comes in, the system gives the first block of the PFREE[*augmented requested size*] linked list to the application. There is no reason to search the linked list, because all linked blocks are identical in size.

Fixed-size allocators are easy to implement and good at countering fragmentation, at least if the number of block sizes is relatively low. They are limited by the fact that there is a maximum size that they can allocate. Fixed-size allocators are fast and retain their speed under all conditions. Such allocators may produce large amounts of internal fragmentation, but, for certain applications, their benefits outweigh this shortcoming.

REDUCING MEMORY FRAGMENTATION

Memory fragmentation comes about as the result of allocating and freeing a memory block but not returning the freed memory to the largest block. The last step is critical. If the memory allocator is to be of any use, you cannot prevent applications from allocating and freeing blocks. Even if a memory allocator cannot make sure that returned memory is joined to the largest block, an approach that would completely avoid memory fragmentation, you can do a lot to control and limit such fragmentation. All of these actions involve splitting blocks. You make improvements whenever the system reduces the number of splits and ensures that the split blocks remain as large as possible.

The goal is to reuse blocks as often as possible without breaking them up to fit exactly each time. Breaking up memory produces an abundance of small fragments that are like grains of sand. It is difficult to later glue these grains back together with the rest of the memory. Instead, it is better to allow a few unused bytes within each block. How many depends on how badly your application needs to avoid memory fragmentation. Adding a few bytes of internal fragmentation for small allocations is a step in the

right direction. When an application asks for 1 byte, the amount you allocate depends on the application's behavior.

If a substantial part of the application's allocations is 1 to 16 bytes, it might be wise to allocate 16 bytes for small allocations. You can also obtain substantial savings by limiting the largest block

that can be allocated. This approach, however, leads to the drawback that applications may cease to function when they keep trying to allocate blocks that are larger than the limit. Reducing the number of sizes between these extremes also helps. Employing sizes that increase logarithmically saves a lot of fragmentation. For example, each size could be 20% larger than the previous size. "One size fits all" might not be true for memory allocators in embedded system. This approach would be incredibly expensive in terms of internal fragmentation, but the system would be completely free of external fragmentation up to the maximum supported size.

Joining adjacent free blocks is an obvious technique for reducing fragmentation. Certain allocation algorithms, such as first-fit, simply cannot do without this approach. Nevertheless, success is limited. Joining adjacent blocks can only ease the pain caused by the allocation algorithm; it cannot cure the underlying problem. But, joining may be hard to implement when the block sizes are limited.

Some memory allocators are advanced enough to collect statistics on an application's allocation habits during runtime. They then categorize all allocations by size: small, medium, and large, for example. The system directs each allocation to an area of the managed memory that contains such block sizes. Smaller ranges are allocated from the larger sizes. This scenario represents an interesting hybrid between first-fit and a limited set of fixed sizes, but it moves away from real time.

It is usually difficult to efficiently use temporal locality, but it is worth noting that allocators that spread temporally collocated allocations around in memory are more prone to memory fragmentation. Although other techniques may offer some relief, limiting the number of different-sized memory blocks remains the key technique for reducing memory fragmentation.

Modern software environments have already implemented tools for avoiding memory fragmentation. For example, the OSE real-time operating system, which has especially been developed for fault-tolerant, distributed, and high-availability systems, offers three runtime memory allocators: the kernel `alloc()`, which allocates from the system or block pools; the heap `malloc()`, which allocates from the program heaps; and the OSE memory-manager `alloc_region`, which allocates from the memory-manager memory.

`Alloc` is in many ways the ultimate memory allocator. It produces little memory fragmentation, is fast, and is deterministic. You can tune and even remove memory fragmentation. External fragmentation occurs only when allocating one size, freeing and not allocating that size again. Internal fragmentation occurs constantly but remains constant for a given application and set of eight sizes over time.

`Alloc` is an implementation of a fixed-size memory allocator with as many as eight free lists. The system programmer can configure each size and can decide to

use fewer sizes to further reduce fragmentation. Allocation and freeing, except initially, are constant time operations. First, the system must round the requested block size upward to the next available size. For eight sizes, this goal is achievable using three if-statements. Second, the system always inserts and removes blocks at the head of each of the eight free lists. Initially, allocating unused memory takes a few cycles longer but is still extremely fast and still constant in time.

Heap `malloc()` has a lower memory overhead (8 to 16 bytes/alloc) than `alloc`, and you can disable private ownership of memory. The `malloc()` allocator is reasonably fast, on average. It has less internal fragmentation but more external fragmentation than `alloc()`. It has a maximum size of allocations, but, for most practical systems, the limit is high enough. The optional shared ownership and low overhead make `malloc()` ideal for C++ applications with many small and shared objects. Heap is an implementation of a buddy system with an internal heap-data structure. In OSE, 28 distinct sizes are available; each size is the

sum of the two previous sizes and forms a Fibonacci sequence. The actual block sizes are the sequence numbers times 16 bytes, including the allocator overhead, or eight (or 16 with file and line information enabled) bytes/allocation.

The OSE memory manager fits best when you rarely need large chunks of memory. Typical applications would be to allocate space for entire applications, heaps, or pools. On systems with an MMU, some implementations use the MMU's translation capabilities to significantly reduce or even remove the memory fragmentation. Otherwise, the OSE memory manager is highly fragmenting. It has no maximum allocation size and is an implementation of a first-fit memory allocator. Memory allocations are rounded to an even number of pages—typically, 4 kbytes. □

AUTHOR'S BIOGRAPHY

Jan Lindblad is system architect at Enea Embedded Technology. He holds a master's of science in computer science from the Royal Institute of Technology (Stockholm, Sweden).