5665 Oberlin Drive, Suite 202, San Diego, California 92121
t 858.587.8876 f 858.587.8907 www.pacificmindworks.com

: .NET Won't Slow You Down

Presented by:

Kirk Fertitta
Chief Technical Officer
Pacific MindWorks

MEMORY MANAGEMENT

Techniques for managing application program memory have always involved tradeoffs between numerous optimization possibilities. Different memory management schemes are characterized by the programming burden imposed on the developer, the CPU overhead required, the impact on working set size, the level of determinism, cache coherency, paging performance, and pause times. Generally speaking, these techniques fall into two camps – *deterministic* methods and *heuristic* methods. Deterministic methods, such as manual memory management and reference counting, rate very differently in the above categories when compared with heuristic methods, such as the various forms of modern garbage collection. Often, the nature of the application under development has dictated which memory management technique was desired. Real-time applications may have very different requirements than interactive applications which themselves may have different requirements than middleware programs. Increasingly, however, automated memory management techniques like garbage collection have found broad acceptance across a wide range of applications. The dramatically simplified programming model offered by garbage-collected systems is arguably its most compelling characteristic. Memory management issues can consume an extraordinarily large proportion of development time, and they are very often the source of numerous and hard-to-find bugs. Garbage collectors offer the application developer relief from many of these concerns. Yet, while the advantages of garbage-collected systems have been well understood for some time, they have earned a reputation for negatively impacting performance. With the introduction of Microsoft's .NET Framework and its garbage collection scheme, the importance of understanding more fully the performance characteristics of garbage collectors is paramount. To that end, this article will examine the detailed operation of the .NET garbage collector and discuss how it actually offers several important performance benefits when compared to alternative schemes such as reference counting.

MEMORY LEAKS

The principle challenge the application developer faces with programs that use dynamic memory is properly releasing memory once it is no longer needed. In the simplest scenarios, an object acquires memory resources when it is created and releases those resources when it is destroyed. However, few real-world systems have such a simple in-memory object topology to allow such a rudimentary memory management policy. More often it is the case that memory acquired by one object may be referenced by multiple other objects. The result is an in-memory object graph with objects holding references to each other. An object that created a memory resource may not have sufficient knowledge or context to ascertain when that memory is no longer needed. The object itself does not logically "own" the memory, as other objects maintain references to it and potentially perform operations on the memory. Thus, the object cannot naively release the memory when the object itself is deleted. Indeed, none of the other objects may logically "own" the memory.

Without automated memory management systems like garbage collectors, the programmer is left with the burden of implementing some sort of distributed ownership policy in the application code. Not only is this an enormous development burden, but it is extremely error prone. Memory management errors are some of the easiest errors to make and some of the most difficult to find and debug. If the developer's memory management scheme fails, then two types of errors are likely. One result might be that memory is accessed after it has been released and returned to the heap. This usually results in abrupt termination of the application program. The other likely result is that some memory is never reclaimed. This is known as a memory leak and it is such a prevalent torment to developers that an entire industry of memory diagnostic tools has evolved over the years. Programs with memory leaks suffer from increased memory pressure as the program executes. This leads to poor virtual memory paging performance as the heap grows and spreads out across an increasing number of virtual pages. It also reduces the efficiency of conventional heap managers, such as the C-runtime heap manager, as it becomes increasingly difficult to find contiguous regions of unused memory available to satisfy allocation requests. In programs that run for any considerable period of time, these effects will range from annoying to catastrophic.

Garbage collectors were invented principally to deal with the problem of releasing memory resources at the appropriate time without requiring any assistance from the application program. With garbage collectors, the application developer is completely absolved of any responsibility for freeing memory used by the application. Not only does this totally eliminate a common and costly source of bugs, but it helps to dramatically reduce development time. While many other memory management technologies can be found in modern computing systems, each comes with its own set of costs and disadvantages. The characteristics of one such popular scheme known as *reference counting* will be examined later in this article.

GARBAGE COLLECTION FUNDAMENTALS

When a process first starts, the .NET common language runtime (CLR) reserves a contiguous region of memory from which it allocates all objects created by a .NET application. This region of space is known as the *managed heap*. The CLR maintains a pointer to the next available memory location in the managed heap. When an object allocation request arrives from a .NET application, the CLR satisfies the request by simply returning the value of the heap pointer and then incrementing this pointer by the size of the requested object so that it subsequently points to the next available memory location. This simple arrangement is depicted in Figure 1.

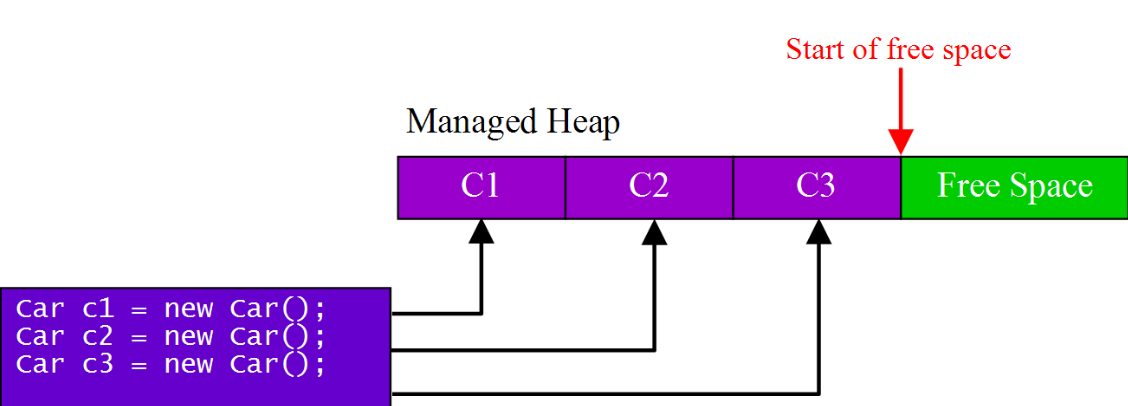

Figure 1 -- CLR memory allocation on the managed heap

When contrasted with how the C-runtime heap allocates objects, object allocation in .NET is extraordinarily fast and simple. Recall that the C-runtime heap allocates objects by walking through a linked list of data structures looking for a block of memory large enough to accommodate the requested object. Then, the memory block must be split and the linked list updated. As the program continues to execute and memory pressure increases, the heap becomes fragmented and the fit-finding performed by the C-runtime heap manager can dramatically impact performance. By comparison, object allocation in the .NET managed heap involves simply incrementing a pointer and so executes extremely quickly. In fact, object allocation from the managed heap is almost as fast as allocating memory from the program stack!

Obviously, the process described above cannot continue indefinitely as the heap pointer would eventually reach the end of the managed region. This is where the garbage collector "kicks-in". When a certain threshold of managed heap memory is reached, the garbage collector will suspend the application program, look for unused memory on the managed heap, and make that memory available for new allocation requests. Determining the appropriate threshold for performing a garbage collection is a science unto itself and, thus, the heuristic used by the .NET garbage collector remains a closely guarded secret of Microsoft. The interested reader is directed to [1] for detailed information on various algorithms. The garbage collector employed in .NET is known as a *mark-compact collector*, as it proceeds to reclaim memory in two phases – marking memory in use and then compacting the managed heap by "squeezing out the holes" created by unused memory cells. The mark phase proceeds by starting at each of the live references held by the application program. These references may reside in local stack variables, in CPU registers, or in static variables. Collectively, they are known as the *root set*. The garbage collector recursively chases references starting at the root set and marks objects it finds along the way as reachable. At the end of the mark phase, any objects on the heap that were not marked are no longer reachable from the application program, and thus, they are garbage and may be reclaimed. The compact phase proceeds by relocating reachable objects to the bottom of the managed heap and then decrementing the heap pointer so that it points to the end of the reachable objects. All of the previously unreachable objects become part of the free space at the end of the heap.

Figure 2 depicts the managed heap before the garbage collector has performed its compaction. Note that the figure shows that the difficult problem of cyclic references is automatically handled by the garbage collector. This benefit will be discussed further shortly. Figure 3 shows what the managed heap looks like after a compaction has completed. All of the unused space has been reclaimed and the live objects have been compressed to the bottom of the heap.

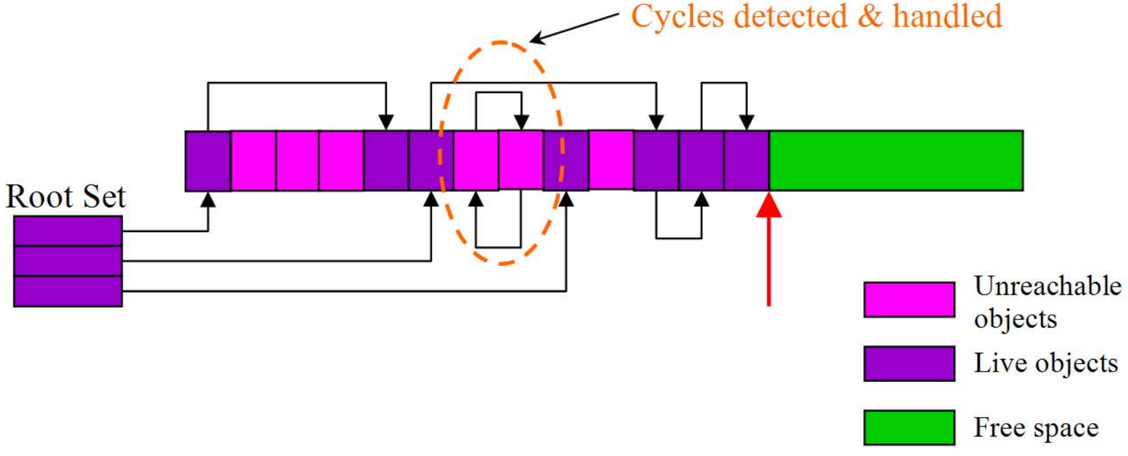

Figure 2 -- Managed heap before compaction

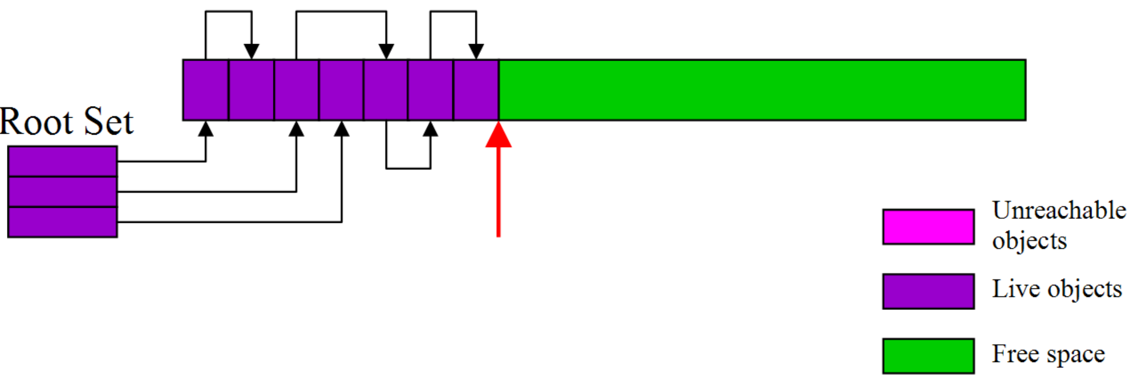

Figure 3 -- Managed heap after compaction

REFERENCE COUNTING

Many non-garbage collected systems employ a technique known as reference counting for managing memory resources. With reference counting, each object maintains an additional field known as the reference count. Each object that acquires a reference to another object increments the target object's reference count upon acquiring the resource and decrements the target's reference count when the reference is released. As long as the object is in use, its reference count will be non-zero. When the reference count drops to zero, then no other objects are referring to the resource and it may safely be deleted. Microsoft's pervasive component technology, COM, is one of the better known systems that use reference counting. Though fairly simple to implement and understand, reference counting has several disadvantages when compared with garbage collection. Reference-counted systems impose a larger per-object memory overhead than do garbage collectors. Each object must maintain an additional field to store the reference count itself, and typically, this field must be pointer-sized, which means each object is a full 32 bits larger (or 64 bits as the industry shift to a 64-bit address space). Garbage collectors, on the other hand, require only a single bit to serve as the mark flag during tracing (assuming the target hardware is capable of addressing a single bit). It is sometimes possible to even eliminate this single bit if it can be smuggled into an unused bit of some other portion of the object. Larger objects result in increased memory pressure and if the target object is only sized on the order of a pointer or less, then the reference count memory overhead actually becomes the dominant component of the object's memory usage.

An often overlooked disadvantage of reference counting is the CPU overhead involved in constantly incrementing and decrementing the reference count. In order to properly manage reference-counted objects, a function call must be made on the object to increment the reference count *each time the reference is copied*. Similarly, a function call must be made on the object to decrement the reference count each time a copy of the reference is no longer needed. These increment and decrement operations occur much more often than many developers realize. When object references are passed into or out of functions, then the reference count is often incremented and decremented. In situations such as functions running inside of tight loops, the performance hit associated with calling increment and decrement functions can be considerable. Further adding to the difficulty is the need to maintain thread-safe reference counts when operating in multi-threaded environments. Not only does this further complicate the reference counting infrastructure itself, but it also increases the per-object memory footprint because each object may now need to contain an additional field for storing a synchronization lock, such as a critical section or a mutex. Moreover, the CPU overhead associated with calling increment and decrement function is increased as each call must successfully acquire the synchronization lock before modifying the reference count.

Perhaps most fundamentally, reference counted systems do not provide the chief advantage that garbage collectors offer – *they remove neither the programming burden of tracking resources nor the errors that accompany that burden*. The application developer must carefully follow a disciplined regime for balancing reference count increment and decrement operations or memory leaks and other bugs will result.

Each application contains explicit program code for using reference counted objects. Even the most well-intentioned developer can easily fall into reference counting traps. One such infamous trap is the menacing *cyclic reference*.

CYCLIC REFERENCES

One of the most insidious problems with deterministic memory management schemes such as reference counting is the problem of cyclic references. A cyclic reference occurs when two objects maintain a counted reference to one another. Consider the arrangement shown in Figure 4. The application program has created object **A** and incremented its reference count by 1. Object **A** contains a child object, shown here as object **B**. This could be the parent-child relationship that exists between, say, folders and subfolders in a TreeView depiction of files on a user's hard drive. In order to easily navigate from child folders to parent folders, object **B** maintains a counted reference to its parent, object **A**. This results in **A**'s reference count being incremented to 2. However, the application program is completely unaware (as it should be) of the "extra" reference count on **A**. When the application is finished with **A**, it will release its reference and only decrement the reference count on **A** by 1. Object **A** will not decrement the ref count on **B** until **A**'s ref count falls to zero, which cannot occur until **B**'s ref count is zero. Thus, the objects **A** and **B** are locked in a "deadly embrace" and will never destroy one another. The result is that all of the memory and resources associated with **A** and **B** can never be reclaimed. As the program continues to execute, the total amount of leaked memory grows, increasing memory pressure and potentially degrading performance as cache misses and virtual page fault frequency goes up.

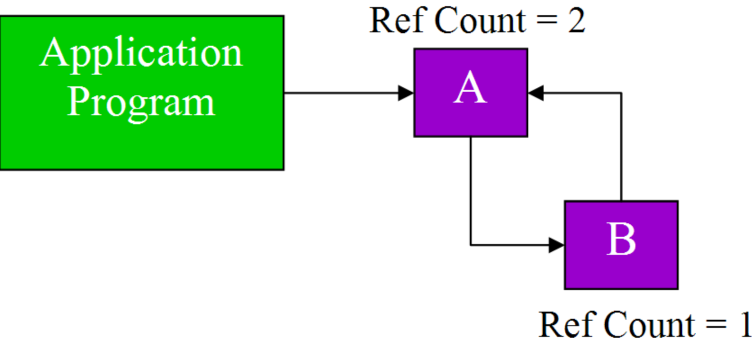

Figure 4 -- Classic reference counting cycle

Program errors associated with reference counting cycles are common, often extremely difficult to find, and potentially very difficult to fix. In COM, special techniques such as weak references, weak identity, and split identity are often employed to combat the problem, but each of these techniques either works only within a limited context or is burdensome and inelegant to implement (or both). The .NET garbage collector completely eliminates the application developer's concern over reference counting cycles. The

algorithm it uses when tracing live references during the mark phase properly detects reference cycles and marks the associated objects as unreachable so that they will subsequently be returned to the free space region of the managed heap. Indeed, the ability of the .NET garbage collector to properly deal with reference cycles and the resulting reduction in application bugs and improvement in memory performance is likely the single most important factor influencing this author's belief in the efficacy of garbage collection.

LOCALITY OF REFERENCE

Non-garbage collected systems, such as those employing a conventional C-runtime heap produce highly fragmented memory. Objects are potentially spread out across a large range of virtual address space allocated to the process, as the heap manager constantly traverses the heap looking for a contiguous region large enough to accommodate allocation requests. As a program executes, this fragmentation gets progressively worse. This is known as poor *locality of reference* and has two very important performance consequences. First, object allocation is slow because CPU resources are consumed during the heap manager's fit-finding operations. Second, the application program is constantly accessing objects that live in different virtual memory pages, resulting in poor cache performance and an increase in the number of page faults. By contrast, the .NET garbage collector compacts all live objects into a contiguous region of memory at the bottom of the managed heap, as shown previously in Figure 3. This produces greatly improved locality of reference when compared with traditional heaps. Moreover, a growing body of empirical evidence supports the assertion that objects that are allocated closely together in time are most often accessed together in a running application. The simple sequential allocation scheme employed in .NET ensures that objects allocated closely together in time do indeed reside closely together in memory. Consequently, a running program may increasingly find the objects it accesses in the data cache, which can dramatically improve object access times. Additionally, the collective body of active objects is spread across many fewer virtual pages than with traditional heaps, resulting in better virtual paging performance.

PAUSE TIMES

The chief component of overhead associated with garbage collection in the .NET platform is the time it takes to perform its two-phase mark-compact operation. During this time, the application program threads must be halted. This is due to the fact that the garbage collector is physically relocating objects and must fix-up any pointers to these relocated objects so that they point to the proper place after the compaction. Once the compaction starts, all of the application program's references are immediately invalid, and therefore no operations can be allowed until the garbage collector has completed its work. The result is that a pause time is introduced into the running application. Some classes of application may not be able to tolerate these non-deterministic pause times, as they occur according to the internal heuristic of the .NET garbage collector. Yet, this pause time is not unlike the numerous sources of non-

deterministic behavior that have characterized Windows applications for years. While hard real-time applications may find this intractable, Windows has never purported to be a real-time operating system, and it is likely that the vast majority of Windows applications already exhibit larger sources of non-deterministic behavior from elements in both the operating system and in their application code. Nevertheless, improving the performance of the garbage collector so that pause times are minimized will be an aggressive, ongoing area of work for Microsoft – just as improving the performance of the C-runtime heap algorithms has been for years. In fact, with the public release version of the .NET platform, Microsoft claims to have already achieved garbage collection times that are no longer than an ordinary page fault. The next section discusses one of the optimizations Microsoft is using to achieve this goal – *generations*.

GENERATIONS

Traversing the *entire* managed heap during a garbage collection in order to locate unreachable objects might be prohibitively expensive. Rather than scanning the entire heap at each collection, the .NET garbage collector divides the heap into *generations* of objects. The youngest generation is called generation 0 and contains the youngest objects, specifically, those that have not yet "survived" a garbage collection. Objects in generation 1 have survived a single garbage collection, and objects in generation 2 have survived 2 or more collections. The present Microsoft implementation uses three generations. Figure 5 shows a simple depiction of the managed heap with generations. The .NET garbage collector subscribes to two fundamental principles that have been demonstrated [1] to be true in a very large class of applications. The first principle is known as the *weak generational hypothesis*, which states that most objects die young. The second principle is called the *strong generational hypothesis*, which says that the older an object is, the longer it is likely to live. This means that the garbage collector can focus its attention on the youngest objects and scan only that portion of the managed heap in order to retrieve the largest proportion of unused memory. Indeed, this is what the .NET garbage collector does. It first scans generation 0 objects and if "enough" storage is not reclaimed, then it moves on to scanning generation 1 objects as well. If enough storage has still not been reclaimed, then generation 2 objects will be collected. In this way, pause times can be greatly reduced because the garbage collector doesn't waste time scanning regions of the heap that are likely to contain live objects.

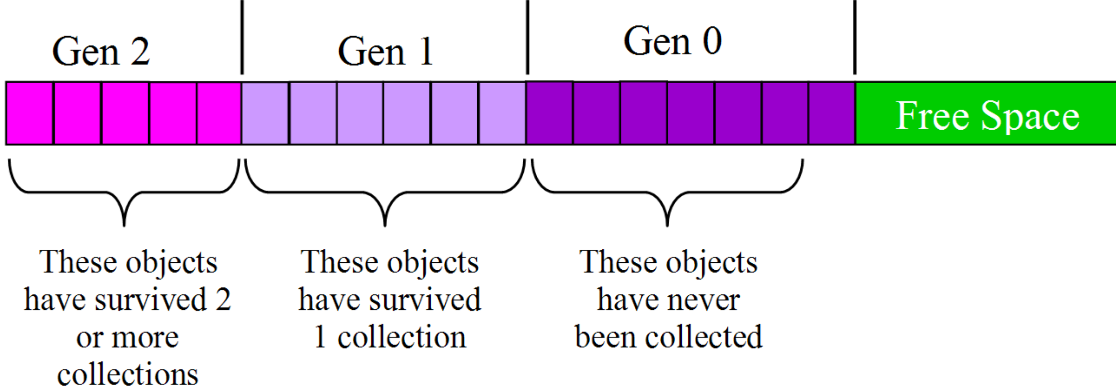

Figure 5 -- Managed heap with generations

LARGE OBJECTS

The compaction of all live objects to the bottom of the managed heap would impose a performance burden when large objects are involved. Repeatedly copying large objects from one region of memory to another would potentially lengthen pause times noticeably. Microsoft's employs an additional optimization specifically to deal with this issue. Large objects (those that are 20,000 bytes or larger) are allocated on a special large object heap (LOH). This heap operates more like a traditional C-runtime heap, so that the performance hit associated with moving large objects is not incurred, thereby keeping pause times for garbage collection short.

CONCLUSION

This article has described the operation of the garbage collector employed by Microsoft's new .NET runtime environment. Garbage collection was shown to offer considerable benefit in terms of reduced programming burden and eliminating memory leaks that are all too common in manually managed memory systems. Additionally, the performance characteristics of the garbage collector were examined and compared to traditionally managed heaps, like the C-runtime heap. Garbage collection was shown to offer better cache coherency, virtual paging performance, as well as extremely fast and simple object allocation. The article concluded with a discussion of optimizations used to reduce pause times associated with garbage collection.

REFERENCES

[1] *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, by Richard Jones and Rafael Lins (John Wiley & Sons, 1996)

[2] *Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework*, by Jeffrey Richter (MSDN Magazine, Nov 2000)

[3] *Garbage Collection – Part 2: Automatic Memory Management in the Microsoft .NET Framework*, by Jeffrey Richter (MSDN Magazine, Dec 2000)