GARBAGE COLLECTION IN .NET

Aastha Trehan*

Ritika Grover*

Sakshi Kohli*

Abstract

This paper highlights how garbage collection works in .NET. It explains how resources are allocated and managed, then gives a detailed step-by-step description of how the garbage collection algorithm works. It also discusses the way resources can clean up properly when the garbage collector decides to free a resource's memory and how to force an object to clean up when it is freed.The .NET Framework's garbage collector manages the allocation and release of memory for your application . Memory is not infinite, eventually the garbage collector must perform a collection in order to free some memory. The garbage collector's optimizing engine determines the best time to perform a collection, based upon the allocations being made. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory.

* Department of Computer Science, Dronacharya College of Engineering,Gurgaon-123506,India

1. What is Garbage Collection?

Garbage Collection is an automatic memory management technique which dealloactes the memory when the object goes out of the scope or is no longer referenced. Garbage Collection is one of the features provided by .NET Framework CLR.In applications with significant memory requirements, you can force garbage collection by invoking the Garbage Collection and then collect method from the program. This is not recommended and should be used only in extreme cases. Almost every program uses resources such as database connection, file system objects etc.

First we allocate a block of memory in the managed memory by using the new keyword. (This will emit the newobj instruction in the MSIL code). Then use the constructor of the class to set the initial state of the object and use the resources by accessing the type's members. At last clear the memory.

2. Why we need Garbage Collection?

Garbage Collection frees the developer from tracking memory usage and knowing when to free memory. Every program uses resources of one sort or another i.e. memory buffers, screen space, network connections, database resources, file resources, and so on. The two bugs cause resource leaks (memory consumption) and object corruption (destabilization), making your application perform in unpredictable ways at unpredictable times. These leaks eventually consumed a process's entire memory space and caused it to crash.

There are two places the .NET framework stores items in memory as your code executes- Stack and Heap.

2.1 Stack

The Stack is more or less responsible for keeping track of what's executing in our code (or what's been "called"). Stack is a series of boxes stacked one on top of the next. We keep track of what's going on in our application by stacking another box on top every time we call a method (called a Frame). We can only use what's in the top box on the stack. When we're done with the top box we throw it away and proceed to use the stuff in the previous box on the top of the stack. The

Stack is self-maintaining, meaning that it basically takes care of its own memory management. When the top box is no longer used, it's thrown out.

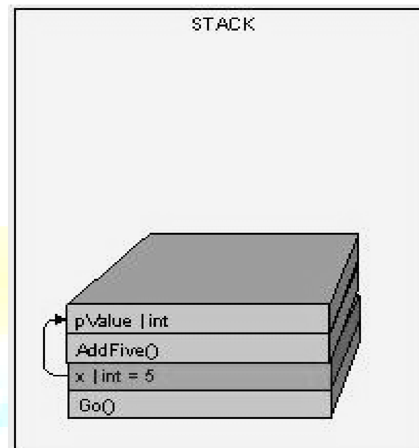

Fig.1 Stack

2.2 Heap

Heap is similar to stack except that its purpose is to hold information (not keep track of execution most of the time) so anything in our Heap can be accessed at any time. With the Heap, there are no constraints as to what can be accessed like in the stack. The Heap is like the heap of clean laundry on our bed that we have not taken the time to put away yet - we can grab what we need quickly. The Stack is like the stack of shoe boxes in the closet where we have to take off the top one to get to the one underneath it. The Heap, on the other hand, has to worry about Garbage collection (GC) - which deals with how to keep the Heap.

3. Garbage Collection – Marking Memory

When the garbage collector starts running, it makes the assumption that all objects in the heap are garbage. The garbage collector starts walking the roots and building a graph of all objects reachable from the roots. As the garbage collector walks from object to object, if it attempts to add an object to the graph that it previously added, then the garbage collector can stop walking down that path. Once all the roots have been checked, the garbage collector's graph contains the

set of all objects that are somehow reachable from the application's roots; any objects that are not in the graph are not accessible by the application, and are therefore considered garbage.

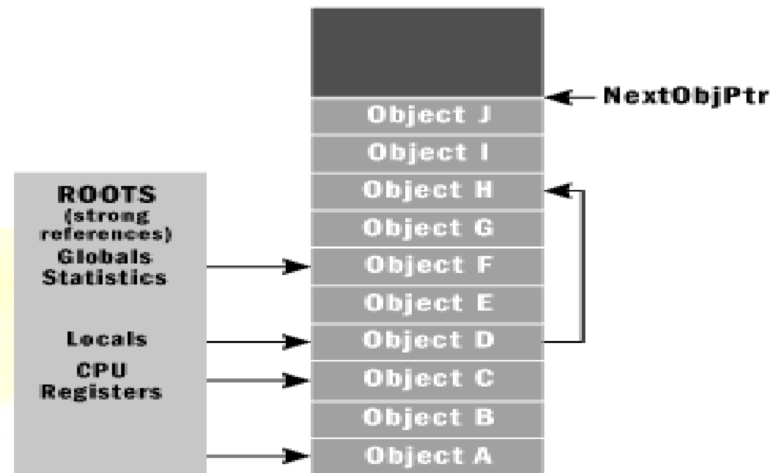

Fig.2 Marking Memory

4. Garbage Collection-Finalization

By using finalization, a resource representing a file or network connection is able to clean itself up properly when the garbage collector decides to free the resource's memory. When the garbage collector detects that an object is garbage, the garbage collector calls the object's Finalize method (if it exists) and then the object's memory is reclaimed.

```
public class BaseObj
{
 public BaseObj()
{
}
protected override void Finalize()
{
// Perform resource cleanup code here...
 // Example: Close file/Close network connection
 Console.WriteLine("In Finalize.");
}
}
```

```
class MyObject

{

 ~MyObject() {    }

}
```

is equivalent to

```
class MyObject
{
 protected override void Finalize()
{
Base .Finalize();
 }
}
```

4.1-Finalization Internals

When an application creates a new object, the new operator allocates the memory from the heap. If the object's type contains a Finalize method, then a pointer to the object is placed on the finalization queue.

The finalization queue is an internal data structure controlled by the garbage collector.

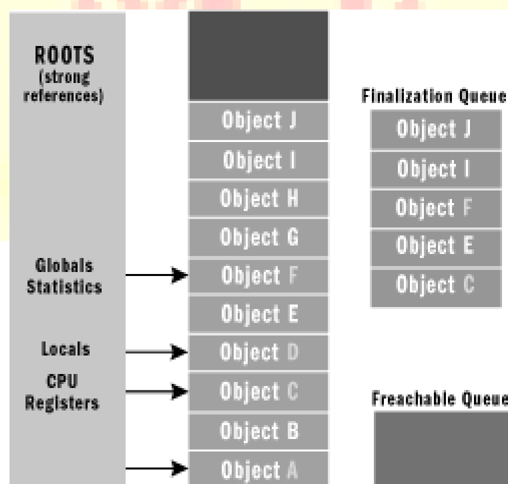

Fig-3 Heap with Many Objects

Some of these objects are reachable from the application's roots, and some are not. When objects C, E, F, I, and J were created, the system detected that these objects had Finalize methods and pointers to these objects were added to the finalization queue.

When a GC occurs, objects B, E, G, H, I, and J are determined to be garbage. The garbage collector scans the finalization queue looking for pointers to these objects. When a pointer is found, the pointer is removed from the finalization queue and appended to the freachable queue (pronounced "F-reachable"). The freachable queue is another internal data structure controlled by the garbage collector. Each pointer in the freachable queue identifies an object that is ready to have its Finalize method called.

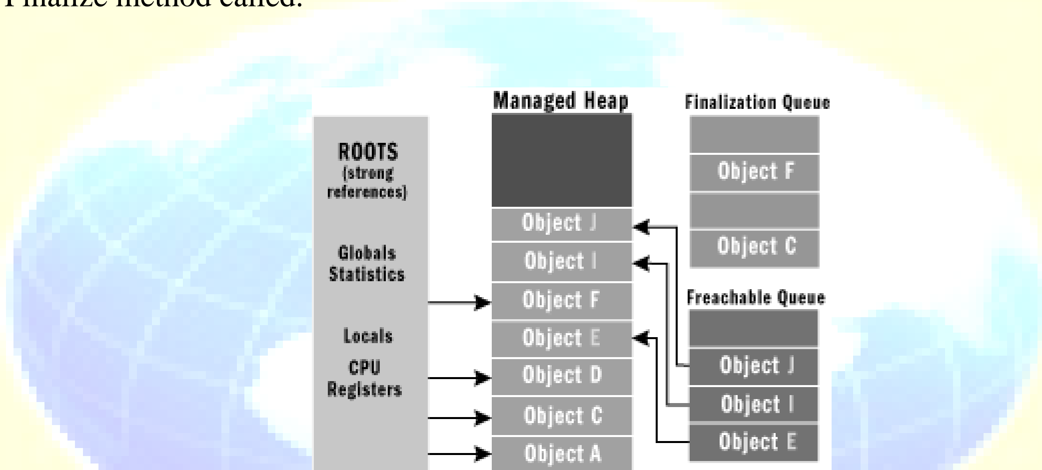

Fig.4 Managed Heap after Garbage Collection

After the collection, the managed heap looks like Figure 4. Here, you see that the memory occupied by objects B, G, and H has been reclaimed because these objects did not have a Finalize method that needed to be called. However, the memory occupied by objects E, I, and J could not be reclaimed because their Finalize method has not been called yet. There is a special runtime thread dedicated to calling Finalize methods. When the freachable queue is empty (which is usually the case), this thread sleeps. But when entries appear, this thread wakes, removes each entry from the queue, and calls each object's Finalize method. The freachable queue is considered to be a root just like global and static variables are roots. Therefore, if an object is on the freachable queue, then the object is reachable and is not garbage. When an object is not reachable, the garbage collector considers the object garbage. Then, when the garbage collector moves an object's entry from the finalization queue to the freachable queue, the object is no longer considered garbage and its memory is not reclaimed. At this point, the garbage

collector has finished identifying garbage. Some of the objects identified as garbage have been reclassified as not garbage. The garbage collector compacts the reclaimable memory and the special runtime thread empties the freachable queue, executing each object's Finalize method. The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage, since the application's roots don't point to it and the freachable queue no longer points to it. Now the memory for the object is simply reclaimed.

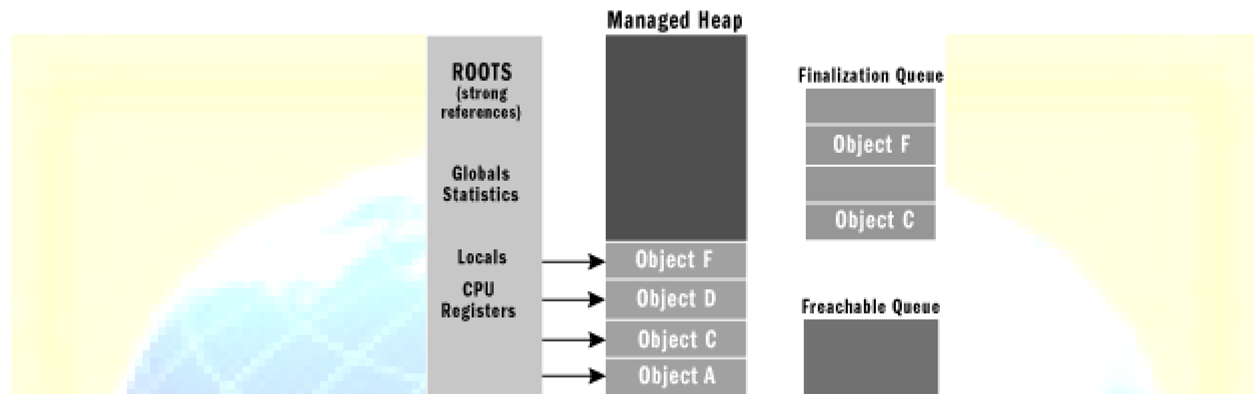

Fig.5 Managed Heap after Second Garbage Collection

5. Garbage Collection – Myths

- GC is necessarily slower than manual memory management
- Modern GC's appear to run as quickly as manual storage allocators. Customized memory allocator are faster but the extra code required to make manual memory management work properly is often more expensive than a garbage collector would be.
- GC will necessarily make my program pause
- Since GC's usually stop the entire program while seeking and collecting garbage objects, they cause pauses long enough to be noticed by the users. But with the advent of modern optimization techniques, these noticeable pauses can be eliminated.
- Manual memory management won't cause pauses
- Manual memory management does not guarantee performance. It may cause pauses for considerable periods either on allocation or deallocation.

6. Conclusion

The motivation for garbage-collected environments is to simplify memory management for the developer. Garbage collection can become a bottleneck in different applications depending on the requirements of the applications. By understanding the requirements of the application and the garbage collection options, it is possible to minimize the impact of garbage collection. Garbage collection in the Microsoft .NET common language runtime environment completely absolves the developer from tracking memory usage and knowing when to free memory. However programs with Garbage Collection are huge and bloated; it isn't suitable for small programs or systems. Though using Garbage Collection is advantageous in complex systems, there is no reason for garbage collection to introduce any significant overhead at any scale. It uses twice as much memory. The data structures used for Garbage Collection need not be larger than those for manual memory management.

References

[1] http://msdn.microsoft.com/en-us/magazine/bb985010.aspx

[2] The Garbage Collection Book by Jones & Lins, ISBN 0471941484

[3] The Garbage Collection FAQ - http://www.iecc.com/gclist/GC-faq.html

[4] The Garbage Collection Page - http://www.cs.kent.ac.uk/people/staff/rej/gc.html

[5] Uniprocessor Garbage Collection Techniques - ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps