Tracing Garbage Collection

Tracing garbage collection is the garbage collection mechanism used by the .NET CLR, the Java VM and various other managed environments —these environments do not use reference counting garbage collection in any form. Developers do not need to issue explicit memory deallocation requests; this is taken care of by the garbage collector. A tracing GC does not associate an object with a reference count, and normally doesn't incur any deallocation cost until a memory usage threshold is reached.

When a garbage collection occurs, the collector begins with the *mark phase*, during which it resolves all objects that are still referenced by the application (live objects). After constructing the set of live objects, the collector moves to the *sweep phase*, at which time it reclaims the space occupied by unused objects. Finally, the collector concludes with the *compact phase*, in which it shifts live objects so that they remain consecutive in memory.

In this chapter, we will examine in detail the various issues associated with tracing garbage collection. A general outline of these issues, however, can be provided right now:

- *Allocation cost*: The allocation cost is comparable to a stack-based allocation, because there is no maintenance associated with free objects. An allocation consists of incrementing a pointer.
- *Deallocation cost*: Incurred whenever the GC cycle occurs instead of being uniformly distributed across the application's execution profile. This has its advantages and disadvantages (specifically for low-latency scenarios) that we will discuss later in this chapter.
- *Mark phase*: Locating referenced objects requires a great deal of discipline from the managed execution environment. References to objects can be stored in static variables, local variables on thread stacks, passed as pointers to unmanaged code, etc. Tracing every possible reference path to each accessible object is everything but trivial, and often incurs a run-time cost outside the collection cycle.
- *Sweep phase*: Moving objects around in memory costs time, and might not be applicable for large objects. On the other hand, eliminating unused space between objects facilitates locality of reference, because objects that were allocated together are positioned together in memory. Additionally, this removes the need for an additional defragmentation mechanism because objects are always stored consecutively. Finally, this means that the allocation code does not need to account for "holes" between objects when looking for free space; a simple pointer-based allocation strategy can be used instead.

In the subsequent sections, we will examine the .NET GC memory management paradigm, starting from understanding the GC mark and sweep phases and moving to more significant optimizations such as generations.

Mark Phase

In the *mark phase* of the tracing garbage collection cycle, the GC traverses the graph of all objects currently referenced by the application. To successfully traverse this graph and prevent *false positives* and *false negatives* (discussed later in this chapter), the GC needs a set of starting points from which reference traversal can ensue. These starting points are termed *roots*, and as their name implies, they form the roots of the directed reference graph that the GC builds.

After having established a set of roots, the garbage collector's work in the mark phase is relatively simple to understand. It considers each internal reference field in each root, and proceeds to traverse the graph until all referenced objects have been visited. Because reference cycles are allowed in .NET applications, the GC marks visited objects so that each object is visited once and only once—hence the name of the *mark* phase.

Local Roots

One of the most obvious types of roots is *local variables*; a single local variable can form the root of an entire object graph referenced by an application. For example, consider the following code fragment within the application's Main method that creates a `System.Xml.XmlDocument` object and proceeds to call its `Load method`:

```
static void Main(string[] args) {
    XmlDocument doc = new XmlDocument();
    doc.Load("Data.xml");
    Console.WriteLine(doc.OuterXml);
}
```

We do not exercise control over the garbage collector's timing, and therefore must assume that a garbage collection might occur during the Load method call. If that happens, we would not like the XmlDocument object to be reclaimed—the local reference in the Main method is the root of the document's object graph that must be considered by the garbage collector. Therefore, every local variable that can potentially hold a reference to an object (i.e., every local variable of a reference type) can appear to be an active root while its method is on the stack.

However, we do not need the reference to remain an active root until the *end* of its enclosing method. For example, after the document was loaded and displayed, we might want to introduce additional code within the same method that no longer requires the document to be kept in memory. This code might take a long time to complete, and if a garbage collection occurs in the meantime, we would like the document's memory to be reclaimed.

Does the .NET garbage collector provide this eager collection facility? Let's examine the following code fragment, which creates a `System.Threading`.Timer and initializes it with a callback that induces a garbage collection by calling `GC.Collect` (we will examine this API later in more detail):

```
using System;
using System.Threading;

class Program {
    static void Main(string[] args) {
     Timer timer = new Timer(OnTimer, null, 0, 1000);
     Console.ReadLine();
    }
    static void OnTimer(object state) {
     Console.WriteLine(DateTime.Now.TimeOfDay);
     GC.Collect();
    }
}
```

If you run the above code in Debug mode (if compiling from the command line, without the /optimize + compiler switch), you will see that the timer callback is called every second, as expected, implying that the timer is not garbage collected. However, if you run it in Release mode (with the /optimize + compiler switch), the timer callback is only called once! In other words, the timer is collected and stops invoking the callback. This is legal (and even desired) behavior because our local reference to the timer is no longer relevant as a root once we reach the Console.ReadLine method call. Because it's no longer relevant, the timer is collected, producing a rather unexpected result if you didn't follow the discussion on local roots earlier!

EAGER ROOT COLLECTION

This eager collection facility for local roots is actually provided by the .NET Just-In-Time Compiler (JIT). The garbage collector has no way of knowing on its own whether the local variable can still potentially be used by its enclosing method. This information is embedded into special tables by the JIT when it compiles the method. For each local variable, the JIT embeds into the table the addresses of the earliest and latest instruction pointers where the variable is still relevant as a root. The GC then uses these tables when it performs its stack walk. (Note that the local variables may be stored on the stack or in CPU registers; the JIT tables must indicate this.)

```
//Original C# code:
static void Main() {
  Widget a = new Widget();
  a.Use();
  //...additional code
  Widget b = new Widget();
  b.Use();
  //...additional code
  Foo(); //static method call
}

//Compiled x86 assembly code:
     ; prologue omitted for brevity
     call 0x0a890a30;             Widget..ctor
+0x14 mov esi, eax ;              esi now contains the object's reference
     mov ecx, esi
     mov eax, dword ptr [ecx]
     ; the rest of the function call sequence
+0x24 mov dword ptr [ebp-12], eax ; ebp-12 now contains the object's reference
     mov ecx, dword ptr [ebp-12]
     mov eax, dword ptr [ecx]
     ; the rest of the function call sequence
+0x34 call 0x0a880480 ;            Foo method call
     ; method epilogue omitted for brevity

//JIT-generated tables that the GC consults:
Register or stack     Begin offset     End offset
     ESI                  0x14            0x24
  EBP - 12                0x24            0x34
```

The above discussion implies that breaking your code into smaller methods and using less local variables is not just a good design measure or a software engineering technique. With the .NET garbage collector, it can provide a performance benefit as well because you have less local roots! It means less work for the JIT when compiling the method, less space to be occupied by the root IP tables, and less work for the GC when performing its stack walk.

What if we want to explicitly extend the lifetime of our timer until the end of its enclosing method? There are multiple ways of accomplishing this. We could use a static variable (which is a different type of root, to be discussed shortly). Alternatively, we could use the timer just before the method's terminating statement (e.g., call timer.Dispose()). But the cleanest way of accomplishing this is through the use of the `GC.KeepAlive` method, which guarantees that the reference to the object will be considered as a root.

How does `GC.KeepAlive` work? It might appear to be a magical method ascending from the internals of the CLR. However, it is fairly trivial—we could write it ourselves, and we will. If we pass the object reference to any method that can't be inlined (see Chapter 3 for a discussion of inlining), the JIT must automatically assume that the object is used. Therefore, the following method can stand in for `GC.KeepAlive` if we wanted to:

```
[MethodImpl(MethodImplOptions.NoInlining)]
static void MyKeepAlive(object obj) {
  //Intentionally left blank: the method doesn't have to do anything

}
```

Static Roots

Yet another category of roots is *static variables*. Static members of types are created when the type is loaded (we have seen this process in Chapter 3), and are considered to be potential roots throughout the entire lifetime of the application domain. For example, consider this short program which continuously creates objects which in turn register to a static event:

```
class Button {
  public void OnClick(object sender, EventArgs e) {
    //Implementation omitted
  }
}

class Program {
  static event EventHandler ButtonClick;

  static void Main(string[] args) {
    while (true) {
     Button button = new Button();
     ButtonClick += button.OnClick;
    }
  }
}
```

This turns out to be a memory leak, because the static event contains a list of delegates which in turn contain a reference to the objects we created. In fact, one of the most common .NET memory leaks is having a reference to your object from a static variable!

Other Roots

The two categories of roots just described are the most common ones, but additional categories exist. For example, *GC handles* (represented by the `System.Runtime.InteropServices.GCHandle` type) are also considered by the garbage collector as roots. The *f-reachable queue* is another example of a subtle type of root—objects waiting for finalization are still considered reachable by the GC. We will consider both root types later in this chapter; understanding the other categories of roots is important when debugging memory leaks in .NET applications, because oftentimes there are no trivial (read: static or local) roots referencing your object, but it still remains alive for some other reason.

INSPECTING ROOTS USING SOS.DLL

SOS.DLL, the debugger extension we have seen in Chapter 3, can be used to inspect the chain of roots that is responsible for keeping a particular object alive. Its !gcroot command provides succinct information of the root type and reference chain. Below are some examples of its output:

```
0:004> !gcroot 02512370
HandleTable:
    001513ec (pinned handle)
    -> 03513310 System.Object[]
    -> 0251237c System.EventHandler
    -> 02512370 Employee

0:004> !gcroot 0251239c
```

```
Thread 3068:
    003df31c 002900dc Program.Main(System.String[]) [d:\...\Ch04\Program.cs @ 38]
     esi:
     -> 0251239c Employee

0:004> !gcroot 0227239c
Finalizer Queue:
    0227239c
    -> 0227239c Employee
```

The first type of root in this output is likely a static field—although ascertaining this would involve some work. One way or another, it is a pinning GC handle (GC handles are discussed later in this chapter). The second type of root is the ESI register in thread 3068, which stores a local variable in the Main method. The last type of root is the f-reachable queue.

Performance Implications

The mark phase of the garbage collection cycle is an “almost read-only” phase, at which no objects are shifted in memory or deallocated from it. Nonetheless, there are significant performance implications that arise from the work done at the mark phase:

- During a full mark, the garbage collector must touch every single referenced object. This results in page faults if the memory is no longer in the working set, and results in cache misses and cache thrashing as objects are traversed.
- On a multi-processor system, since the collector marks objects by setting a bit in their header, this causes cache invalidation for other processors that have the object in their cache.
- Unreferenced objects are less costly in this phase, and therefore the performance of the mark phase is linear in the *collection efficiency factor*: the ratio between referenced and unreferenced objects in the collection space.
- The performance of the mark phase additionally depends on the number of objects in the graph, and not the memory consumed by these objects. Large objects that do not contain many references are easier to traverse and incur less overhead. This means that the performance of the mark phase is linear in the number of live objects in the graph.

Once all referenced objects have been marked, the garbage collector has a full graph of all referenced objects and their references (see Figure 4-4). It can now move on to the sweep phase.

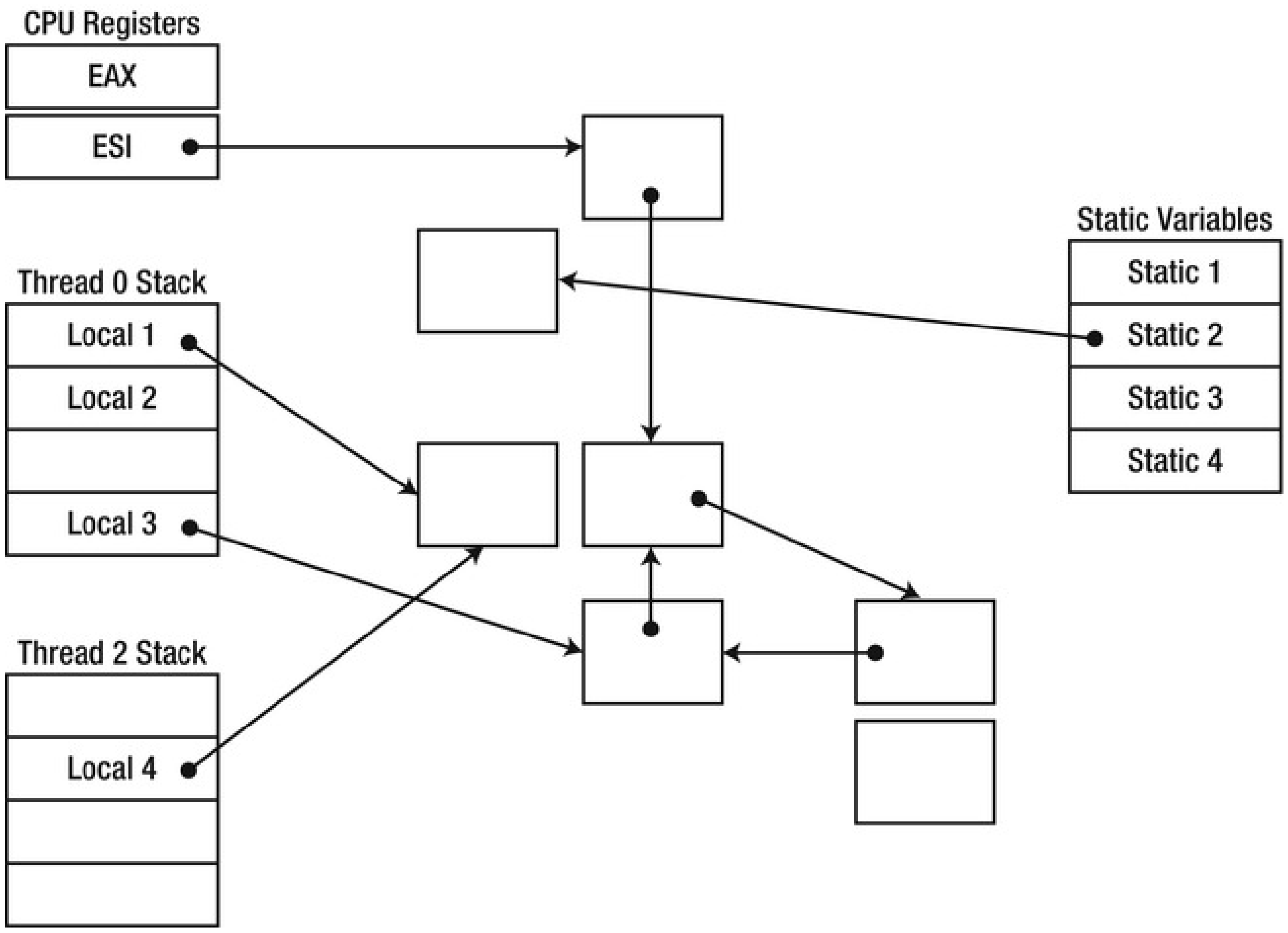

Figure 4-4. An object graph with various types of roots. Cyclic references are permitted

Sweep and Compact Phases

In the sweep and compact phases, the garbage collector reclaims memory, often by shifting live objects around so that they are placed consecutively on the heap. To understand the mechanics of shifting objects around, we must first examine the allocation mechanism which provides the motivation for the work performed in the sweep phase.

In the simple GC model we are currently examining, an allocation request from the application is satisfied by incrementing a single pointer, which always points to the next available slot of memory (see Figure 4-5). This pointer is called the *next object pointer* (or *new object pointer*), and it is initialized when the garbage-collected heap is constructed on application startup.

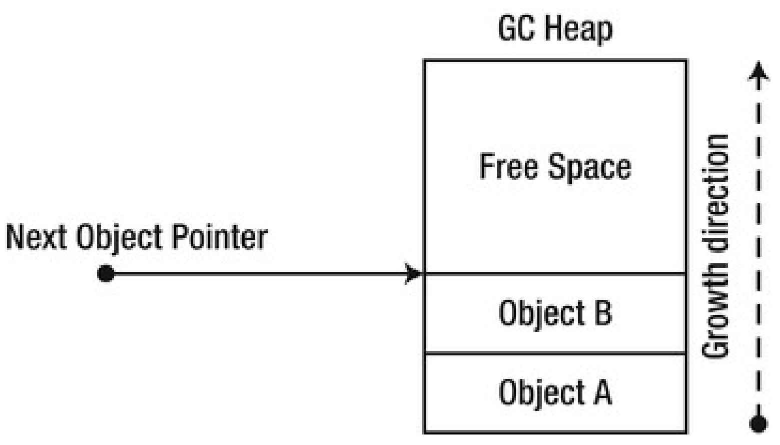

Figure 4-5 *. GC heap and the next object pointer*

Satisfying an allocation request is extremely cheap in this model: it involves only the atomic increment of a single pointer. Multi-processor systems are likely to experience contention for this pointer (a concern that will be addressed later in this chapter).

If memory were infinite, allocation requests could be satisfied indefinitely by incrementing the new object pointer. However, at some point in time we reach a threshold that triggers a garbage collection. The thresholds are dynamic and configurable, and we will look into ways of controlling them later in this chapter.

During the compact phase, the garbage collector moves live objects in memory so that they occupy a consecutive area in space (see Figure 4-6). This aids locality of reference, because objects allocated together are also likely to be used together, so it is preferable to keep them close together in memory. On the other hand, moving objects around has at least two performance pain-points:

- Moving objects around means copying memory, which is an expensive operation for large objects. Even if the copy is optimized, copying several megabytes of memory in each garbage collection cycle results in unreasonable overhead. (This is why large objects are treated differently, as we shall see later.)
- When objects are moved, references to them must be updated to reflect their new location. For objects that are frequently referenced, this scattered memory access (when references are being updated) can be costly.

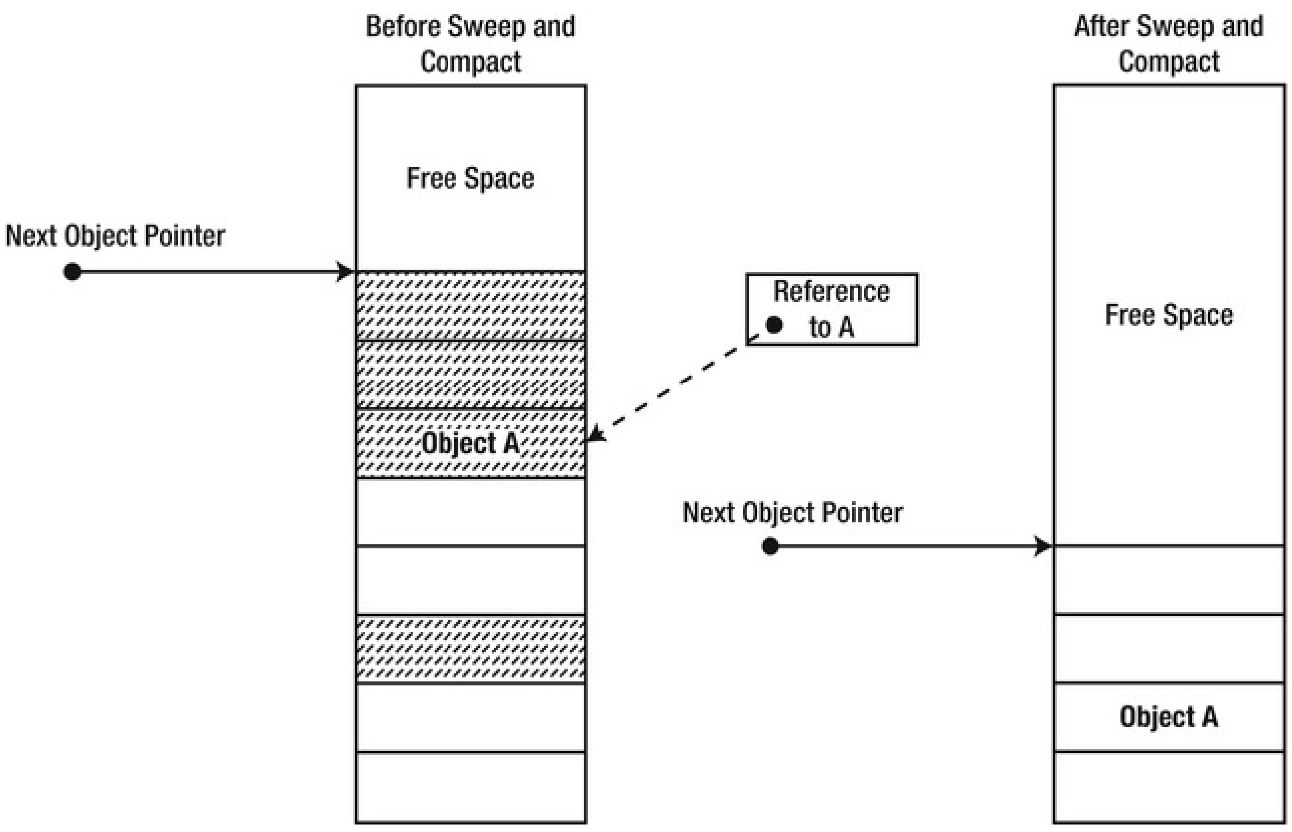

Figure 4-6 *. The shaded objects on the left survive garbage collection and are shifted around in memory. This means that the reference to A (dashed line) has to be updated. (The updated*

reference is not shown in the diagram.)

The general performance of the sweep phase is linear in the number of objects in the graph, and is especially sensitive to the collection efficiency factor. If most objects are discovered to be unreferenced, then the GC has to move only a few objects in memory. The same applies to the scenario where most objects are still referenced, as there are relatively few holes to fill. On the other hand, if every other object in the heap is unreferenced, the GC may have to move almost every live object to fill the holes.

Note Contrary to popular belief, the garbage collector does not always move objects around (i.e., there are some sweep-only collections that do not proceed into the compact phase), even if they are not pinned (see below) and even if free space is available between objects. There is an implementation-defined heuristic which determines whether moving objects to fill free space is worthwhile during the sweep phase. For example, in one test suite executed on the author's 32-bit system, the garbage collector decided to move objects around if the free space is larger than 16 bytes, consists of more than one object, and more than 16KB of allocations have been made since the last garbage collection. You can't rely on these results being reproducible, but this does demonstrate the existence of the optimization.

The mark and sweep model described in the preceding sections is subject to one significant deficiency that we will address later in this chapter as we approach the generational GC model. Whenever a collection occurs, all objects in the heap are traversed even if they can be partitioned by likelihood of collection efficiency. If we had prior knowledge that some objects are more likely to die than others, we might be able to tune our collection algorithm accordingly and obtain a lower amortized collection cost.

Pinning

The garbage collection model presented above does not address a common use case for managed objects. This use case revolves around passing managed objects for consumption by unmanaged code. Two distinct approaches can be used for solving this problem:

- Every object that should be passed to unmanaged code is marshaled by value (copied) when it's passed to unmanaged code, and marshaled back when it's returned.
- Instead of copying the object, a pointer to it is passed to unmanaged code.

Copying memory around every time we need to interact with unmanaged code is an unrealistic proposition. Consider the case of soft real-time video processing software that needs to propagate high-resolution images from unmanaged code to managed code and vice versa at 30 frames per second. Copying multiple megabytes of memory every time a minor change is made will deteriorate performance to unacceptable levels.

The .NET memory management model provides the facilities for obtaining the memory address of a managed object. However, passing this address to unmanaged code in the presence of the garbage collector raises an important concern: What happens if the object is moved by the GC while the unmanaged code is still executing and using the pointer?

This scenario can have disastrous consequences—memory can easily become corrupted. One reliable solution to this problem is *turning off* the garbage collector while unmanaged code has a pointer to a managed object. However, this approach is not granular enough if objects are frequently passed between managed and unmanaged code. It also has the potential of deadlocks or memory exhaustion if a thread enters a long wait from within unmanaged code.

Instead of turning garbage collection off, every managed object whose address can be obtained must also be *pinned* in memory. Pinning an object prevents the garbage collector from moving it around during the sweep phase until it is *unpinned*.

The pinning operation itself is not very expensive—there are multiple mechanisms that perform it rather cheaply. The most explicit way to pin an object is to create a GC handle with the `GCHandleType.Pinned flag`. Creating a GC handle creates a new root in the process' *GC handle table*, which tells the GC that the object should be retained as well as pinned in memory. Other alternatives include the magic sauce used by the P/Invoke marshaler, and the pinned pointers mechanism exposed in C# through the fixed keyword (or pin_ptr < T > in C++/CLI), which relies on marking the pinning local variable in a special way for the GC to see. (Consult Chapter 8 for more details.)

However, the performance cost around pinning becomes apparent when we consider how pinning affects the garbage collection itself. When the garbage collector encounters a pinned object during the compact phase, it must work around that object to ensure that it is not moved in memory. This complicates the collection algorithm, but the direst effect is that *fragmentation* is introduced into the managed heap. A badly fragmented heap directly invalidates the assumptions which make garbage collection viable: It causes consecutive allocations to be fragmented in memory (and the loss of locality), introduces complexity into the allocation process, and causes a waste of memory as fragments cannot be filled.

Note Pinning side-effects can be diagnosed with multiple tools, including the Microsoft CLR Profiler. The CLR profiler can display a graph of objects by address, showing free (fragmented) areas as unused white space. Alternatively, SOS.DLL (the managed debugging extension) can be used to display objects of type "Free", which are holes created due to fragmentation. Finally, the *# of Pinned Objects* performance counter (in the *.NET CLR Memory* performance counter category) can be used to determine how many objects were pinned in the last area examined by the GC.

Despite the above disadvantages, pinning is a necessity in many applications. Oftentimes we do not control pinning directly, when there is an abstraction layer (such as P/Invoke) that takes care of the fine details on our behalf. Later in this chapter, we will come up with a set of recommendations that will minimize the negative effects of pinning.

We have reviewed the basic steps the garbage collector undertakes during a collection cycle. We have also seen what happens to objects that must be passed to unmanaged code. Throughout the previous sections we have seen many areas where optimizations are in place. One thing that was mentioned frequently is that on multi-processor machines, contention and the need for synchronization might be a very significant factor for the performance of memory-intensive applications. In the subsequent sections, we will examine multiple optimizations including optimizations targeting multi-processor systems.