
Summary

Through the course of this chapter, we reviewed the motivation and implementation of the .NET garbage collector, the entity responsible for automatically reclaiming unused memory. We examined alternatives to tracing garbage collection, including reference counting garbage collection and manual free list management.

At the heart of the .NET garbage collector lie the following coordinating concepts, which we examined in detail:

- *Roots* provide the starting point for constructing a graph of all reachable objects.
- *Mark* is the stage at which the garbage collector constructs a graph of all reachable objects and marks them as used. The mark phase can be performed concurrently with application thread execution.
- *Sweep* is the stage at which the garbage collector shifts reachable objects around and updates references to these objects. The sweep phase requires stopping all application threads before proceeding.
- *Pinning* is a mechanism which locks an object in place so that the garbage collector can't move it. Used in conjunction with unmanaged code requiring a pointer to a managed object, and can cause fragmentation.
- *GC flavors* provide static tuning for the behavior of the garbage collector in a specific application to better suit its memory allocation and deallocation patterns.
- *The generational model* describes the life expectancy of an object based on its current age. Younger objects are expected to die fast; old objects are expected to live longer.
- *Generations* are conceptual regions of memory that partition objects by their life expectancy. Generations facilitate frequently performing cheap partial collections with higher collection likelihood, and rarely performing full collections which are expensive and less efficient.
- *Large object heap* is an area of memory reserved for large objects. The LOH can become fragmented, but objects do not move around, thereby reducing the cost of the sweep phase.
- *Segments* are regions of virtual memory allocated by the CLR. Virtual memory can become fragmented because segment size is fixed.
- *Finalization* is a backup mechanism for automatically releasing unmanaged resources in a non-deterministic fashion. Prefer deterministic finalization to automatic finalization whenever possible, but offer clients both alternatives.

The common pitfalls associated with garbage collection are often related to the virtues of its most powerful optimizations:

- The generational model provides performance benefits for well-behaved applications, but can exhibit the mid-life crisis phenomenon which direly affects performance.
- Pinning is a necessity whenever managed objects are passed by reference to unmanaged code, but can introduce internal fragmentation into the GC heap, including the lower generations.
- Segments ensure that virtual memory is allocated in large chunks, but can exhibit external fragmentation of virtual memory space.
- Automatic finalization provides convenient disposal of unmanaged resources, but is associated with a high performance cost and often leads to mid-life crisis, high-pressure memory leaks and race conditions.

The following is a list of some of the best practices for getting the most of the .NET garbage collector:

- Allocate temporary objects so that they die fast, but keep old objects alive for the entire lifetime of your application.
- Pin large arrays at application initialization, and break them into small buffers as needed.
- Manage memory using pooling or unmanaged allocation where the GC fails.
- Implement deterministic finalization and rely on automatic finalization as a backup strategy only.
- Tune your application using GC flavors to find which one works best on various types of hardware and software configurations.

Some of the tools that can be used to diagnose memory-related issues and to examine the behavior of your application from the memory management perspective are the following:

- *CLR Profiler* can be used to diagnose internal fragmentation, determine the heavy memory allocation paths in an application, see which objects are reclaimed at each garbage collection and obtain general statistics on the size and age of retained objects.
- *SOS.DLL* can be used to diagnose memory leaks, analyze external and internal fragmentation, obtain garbage collection timing, list the objects in the managed heap, inspect the finalization queue and view the status of the GC threads and the finalizer thread.

- *CLR performance counters* can be used to obtain general statistics on garbage collection, including the size of each generation, the allocation rate, finalization information, pinned object count and more.
- *CLR hosting* can be used as a diagnostic utility to analyze segment allocations, garbage collection frequencies, threads causing garbage collections and non-GC related memory allocation requests originating at the CLR.

Armed by the theory of garbage collection, the subtleties of all related mechanisms, common pitfalls and best performance practices, and diagnostic tools and scenarios, you are now ready to begin the quest to optimize memory management in your applications and to design them with a proper memory management strategy.