Everybody thinks about garbage collection the wrong way

devblogs.microsoft.com/oldnewthing/20100809-00

August 9, 2010

Raymond Chen

Welcome to CLR Week 2010. This year, CLR Week is going to be more philosophical than usual.

When you ask somebody what garbage collection is, the answer you get is probably going to be something along the lines of "Garbage collection is when the operating environment automatically reclaims memory that is no longer being used by the program. It does this by tracing memory starting from roots to identify which objects are accessible."

This description confuses the mechanism with the goal. It's like saying the job of a firefighter is "driving a red truck and spraying water." That's a description of what a firefighter does, but it misses the point of the job (namely, putting out fires and, more generally, fire safety).

Garbage collection is *simulating a computer with an infinite amount of memory*. The rest is mechanism. And naturally, the mechanism is "reclaiming memory that the program wouldn't notice went missing." It's one giant application of the *as-if* rule.[1]

Now, with this view of the true definition of garbage collection, one result immediately follows:

> If the amount of RAM available to the runtime is greater than the amount of memory required by a program, then a memory manager which employs the null garbage collector (which never collects anything) is a valid memory manager.

This is true because the memory manager can just allocate more RAM whenever the program needs it, and by assumption, this allocation will always succeed. A computer with more RAM than the memory requirements of a program has effectively infinite RAM, and therefore no simulation is needed.

Sure, the statement may be obvious, but it's also useful, because the null garbage collector is both very easy to analyze yet very different from garbage collectors you're more accustomed to seeing. You can therefore use it to produce results like this:

> A correctly-written program cannot assume that finalizers will ever run at any point prior to program termination.

The proof of this is simple: Run the program on a machine with more RAM than the amount of memory required by program. Under these circumstances, the null garbage collector is a valid garbage collector, and the null garbage collector never runs finalizers since it never collects anything.

Garbage collection simulates infinite memory, but there are things you can do even if you have infinite memory that have visible effects on other programs (and possibly even on your program). If you open a file in exclusive mode, then the file will not be accessible to other programs (or even to other parts of your own program) until you close it. A connection that you open to a SQL server consumes resources in the server until you close it. Have too many of these connections outstanding, and you may run into a connection limit which blocks further connections. If you don't explicitly close these resources, then when your program is run on a machine with "infinite" memory, those resources will accumulate and never be released.

What this means for you: Your programs cannot rely on finalizers keeping things tidy. Finalizers are a safety net, not a primary means for resource reclamation. When you are finished with a resource, you need to release it by calling `Close` or `Disconnect` or whatever cleanup method is available on the object. (The `IDisposable` interface codifies this convention.)

Furthermore, it turns out that not only can a correctly-written program not assume that finalizers will run during the execution of a program, it cannot even assume that finalizers will run when the program terminates: Although the .NET Framework will try to run them all, a bad finalizer will cause the .NET Framework to give up and abandon running finalizers. This can happen through no fault of your own: There might be a handle to a network resource that the finalizer is trying to release, but network connectivity problems result in the operation taking longer than two seconds, at which point the .NET Framework will just terminate the process. Therefore, the above result can be strengthened in the specific case of the .NET Framework:

> A correctly-written program cannot assume that finalizers will ever run.

Armed with this knowledge, you can solve this customer's problem. (Confusing terminology is preserved from the original.)

> I have a class that uses `XmlDocument`. After the class is out of scope, I want to delete the file, but I get the exception `System.IO.Exception: The process cannot access the file 'C:\path\to\file.xml' because it is being used by another process.` Once the progam exits, then the lock goes away. Is there any way to avoid locking the file?

This follow-up might or might not help:

> A colleague suggested setting the `XmlDocument` variables to null when we're done with them, but shouldn't leaving the class scope have the same behavior?

Bonus chatter: Finalizers are weird, since they operate "behind the GC." There are also lots of classes which operate "at the GC level", such as `WeakReference` `GCHandle` and of course `System.GC` itself. Using these classes properly requires understanding how they interact with the GC. We'll see more on this later.

Related reading

- The stack is an implementation detail. Another case where people confuse the mechanism with the goal.
- Back to basics: Series on dynamic memory management. When I wrote a garbage collector for a toy project, I used the twospace algorithm for its simplicity.

Unrelated reading: Precedence vs. Associativity Vs. Order.

Footnote

[1] Note that by definition, the simulation extends only to garbage-collected resources. If your program allocates external resources those external resources continue to remain subject to whatever rules apply to them.

Raymond Chen

Follow