

Re: A re-announce on GC's defects

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2007-01/msg02793.html>

- *From:* Andre Kaufmann <andre.kaufmann_re_move_@xxxxxxxxxxxxx>
 - *Date:* Thu, 18 Jan 2007 18:35:56 +0100
-

Born wrote:

GC is really garbage itself

[...]

Negative effects by the destruction delay:

1) Efficiency issue

It's bad for CPU/Resource intensive but memory cheap objects.

It's also bad allocating / deallocating permanently small objects on a native heap. It's rather time consuming, compared to the managed one and it doesn't scale that well if you have a multi core CPU.

>[...]

2) Logic issue

[...]

Don't tell me the IDisposable pattern is for that. There may be more than one strong references and you don't know when and where to call Dispose.

You could use reference counting as well for a managed object. Instead of calling the destructor you call Dispose if the reference counter is 0.

[...]

An example:

Suppose we're doing a 3D game. A radar is monitoring a target. Obviously, the radar should hold a weak reference to the target. When the target is killed, logical confusion is immediately brought to the radar watcher (the gamer). Is the target destroyed or not? You can not tell him, hey, it's killed but still shown on the radar because you've got to wait for the GC to make it.

What has the state of an object (resource) to do with the memory it has allocated ? That's the only thing GC is supposed to do – manage memory.

Re: A re-announce on GC's defects

Reason 2:

[...]

Fairly speaking, GC itself is not garbage. However, when java and C# integrate it and prevent the user from manually managing memory, it becomes garbage. Note GC in java and C# is not really an additive as someone would argue since there is no way to do real memory management like delete obj in C++.

In C++ you have many classes which handle the memory by them self, e.g. most STL classes to ensure that not permanently memory is allocated or that the memory is consecutive. GC handles this automatically.

Better memory management in my mind is reference counting + smart pointer, which makes things automatic and correct. You have deterministic destructions while no need to manually call Dispose.

As I wrote, why not also implementing reference counting for managed objects, which are calling Dispose if the reference count is 0 ?
Though there is a performance impact, because for thread safe reference counting you have to use Interlocked functions.

You need not to manually change the reference count as smart pointers help you achieve it.

Agreed, you would have to call them manually in C#, because there's no RAII. Which I'm really missing in C#.

The only problem with this approach is cyclic reference. However, even if not theoretically proven, the problem generally can be solved by replacing some strong references with weak references.

Yes, but it's sometimes tricky and in complex object hierarchies you have a very high chance to build cyclic references, which are hard to deal with.

I believe the restriction by GC is one of the main reasons why in some field (the gaming industry, for example), java or C# is rarely used in serious products who face real computing challenges.

Hm, perhaps because one can't see that Java or C# is used. E.g. the game Chrome is written in Java (not 100% but many parts of it)

Re: A re-announce on GC's defects

Solution

1) The ideal solution is to convince the language providers to give us back the ability of managing memory by our own. GC can still be there, and it becomes a real addictive in that situation.

GC doesn't solve resource allocation problems. They are different as in C++ and so are the problems you have to face. It's the same with memory handling. In C++ you still have to think over and over again, how the memory is handled and if it's better to use an object cache. Otherwise you will face performance problems too.

2) Transfer the burden to the user. We can ask the user to always take special cautions (for example, always use "using" in C# to have Dispose correctly called even exception occurs). Things can work around if the user do them right. However, that's at risk in nature and not a robust solution.

Isn't that the case ? The developer has to use "using" e.g. for file objects, which shall release the file handles directly after their usage.

I admit that sometimes I'm missing reference counting, when I'm dealing with objects stored in multiple lists. How shall I know when to call dispose ?

E.g. if a file object is stored in 2 or more lists and has to be removed from one of the lists. How do I know if I have to call Dispose ? Only performant solution for me would be to use reference counting.

Though you can't have smart pointers, which are automatically destroyed and will decrease the reference count of an object automatically. You have to do it manually in C#. :(– perhaps there's a better solution in C# that I don't know yet ? (any comments and solutions would be highly appreciated)

Andre