Garbage Collection Performance

We've covered what happens during a GC; what with building "in use lists" and heap compaction, it's no surprise that performance can become an issue for some applications.

Take heap compaction, for example. Objects are being moved around in memory so, to ensure heap integrity, the GC has to suspend the execution of threads of the executing process. Obviously, this isn't ideal for performance, but it *is* essential to ensure that the heap is managed efficiently.

The tradeoff between performance and heap efficiency has been key to the development of a number of modes under which the GC can run. As briefly mentioned earlier, the GC has two modes: **Workstation** mode, which is tuned to give maximum UI responsiveness and **Server** mode, which is tuned to give maximum request throughput. The way the GC behaves in the two modes is very different, so I will explain them separately.

Workstation GC mode

This mode is designed to give maximum possible responsiveness to the user, and cut down on any pauses due to GC. Ideally, you want to avoid any perception of pauses or jerkiness in interactive applications so, to achieve this responsiveness, Workstation GC mode limits the number of thread suspensions.

Since .NET Common Language Runtime (CLR) version 1.0, Workstation GC could run as either concurrent or non-concurrent; this simply refers to which thread the GC runs on. In non-concurrent mode, thread execution of the application code is suspended, and the GC then runs on the application thread. It was designed for uni-processor machines, where running threads concurrently wasn't an option.

As multicore/multiprocessor desktop machines are now very common, concurrent Workstation GC is now the norm and the default.

Concurrent Workstation GC

Concurrent GC has a separate thread for the GC to run on, meaning that the application can continue execution while the GC runs. Crucially, object allocation onto the ephemeral segment is also allowed as the GC is executing.

It's also worth remembering that concurrent GC only applies to full GCs, so Gen 0 and Gen 1 GCs still cause thread suspension. However, instead of just suspending *all* threads for the duration of a GC, the GC aims to only suspend threads for short periods, usually twice during execution. In contrast to non-concurrent GC, which suspends all threads for the duration of the entire GC process, concurrent GC is much less prone to latency issues.

Here's how it works. When a full concurrent GC takes places, the start and end positions of the allocated objects on the heap are determined, and garbage collection is limited to within this "GC domain." The nice thing about this is that the application can continue to allocate objects onto the heap outside of this domain.

Figure 3.3 illustrates the process whereby the GC domain is initially determined with a start and end position, and then the GC runs (bear in mind that it doesn't show the results of the collection). We can also see that while the GC is running, the application allocates additional objects to the heap (L , M , N and O).

The application can continue to allocate objects right up until the ephemeral segment limit is reached, which is the size of the segment minus a bit of space that we will call the "No Go Zone." Once this limit is reached, application execution is suspended until the full GC is finished.

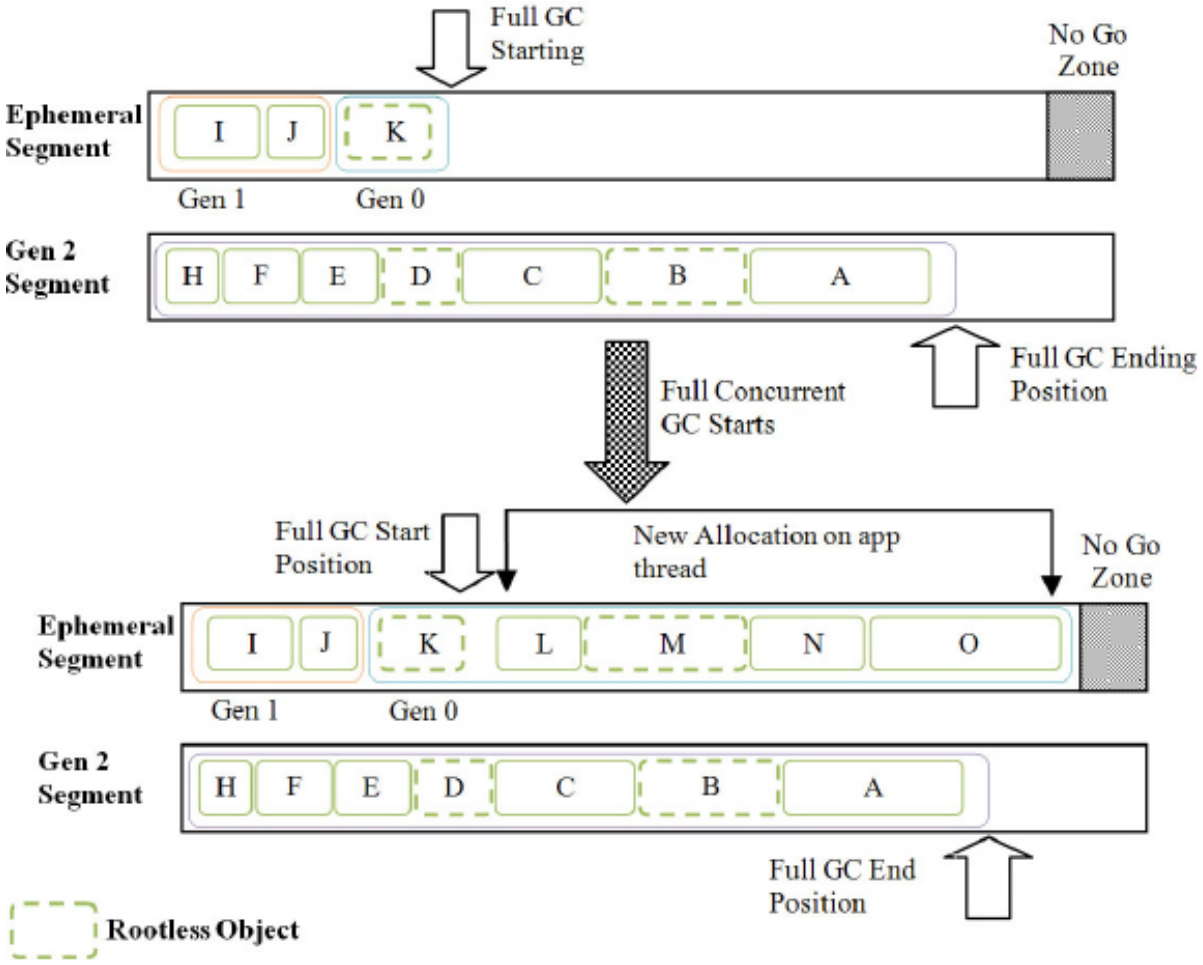

Figure 3.3: Concurrent GC, which would result in the collection of objects K , D and B.

In Figure 3.3, Object M has become rootless, but it is outside of the current GC domain so its space can't be collected. It now contributes to the ephemeral segment usage even though it's no longer being used, and it's going to make thread suspension more likely to happen. That's annoying!

Ideally, we need a way of collecting garbage from both the current GC domain **and** the newly allocated objects, **and** to be able expand the heap if necessary so that thread suspension could be entirely avoided. Well, guess what those crafty Microsoft developers have done...

Background Workstation GC mode (.NET 4.0)

You guessed it! .NET 4.0 has introduced background GC, and its aim is to do all of the above.

With background GC, a Gen 0 or Gen 1 GC can be triggered for the newly allocated objects while a full Gen 2 GC is in progress.

Gen 0 and Gen 1 now have tunable allocation thresholds which fire a background collection when exceeded, and allow rootless objects to be compacted and their space to be reclaimed. At the very least, this delays the inevitable reaching of the ephemeral segment boundary "No Go Zone."

It gets better: a background Gen 1 collection can now also create a new segment and copy Gen 0 objects into it just like in classic GC. That means there is no segment limit, which means no eventual thread suspensions due to exhaustion.

There is, of course, a price to pay for all this, and that is the fact that application and foreground GC threads are suspended while background GC executes. However, due to the speed of Gen 0 and Gen 1 collections, this is usually only a small price.

Server GC mode

It's probably no surprise to learn that Server GC is designed to give maximum throughput, scalability and performance for server environments. It achieves this by making the GC multithreaded, with multiple threads running the GC in parallel on multiprocessors/cores.

That sounds good in theory, but how does the GC prevent all of these threads from interfering with each other? The answer is that, for each process, it allocates a separate SOH and LOH per logical processor (a processor with four cores has four logical processors).

This may seem a bit confusing until you think about what a server is typically doing, which is providing a request/response service. Request/response, by its nature, simply doesn't require state to be held on a directly shared heap; state persistence is specifically provided for as a separate mechanism. As a result, most objects created in a request/response go out of scope immediately, and so can be collected.

In this kind of scenario, maximizing the number of heaps that can be allocated to reduces allocation bottlenecks and, of course, allows garbage collection to use multiple GC threads executing on multiple processors.

On the other side of the fence, an application can certainly have threads allocating objects to multiple heaps, and there is a cross-referencing mechanism in place in the .NET framework which means that objects can still reference each other across the heaps. However, as application responsiveness isn't a direct goal of the Server GC, when Server GC runs, all application threads are suspended for the duration of the GC.

Finally, segment sizes and generation thresholds are typically much larger in Server GC (unsurprising, given the higher demands placed on those sorts of machines).

Configuring the GC

All you need to do to configure the GC is alter your `config` file (ASP.NET applications use `aspnet.config` for your framework version). To enable Server GC, just set `gcServer="true"` in the runtime section. Obviously, for Workstation, set it to `false.`

```
<configuration>
   <runtime>
      <gcServer enabled="true | false"/>
   </runtime>
</configuration>
```

Listing 3.2: Configuring Server GC.

Alternatively, if you have configured Workstation GC, then you can switch on concurrency by setting the `gcConcurrent enabled` flag to `true.`

```
<configuration>
   <runtime>
      <gcConcurrent enabled="true | false"/>
   </runtime>
</configuration>
```

Listing 3.3: Configuring Workstation concurrency.

It's worth bearing in mind that setting the concurrency of Server GC has no effect.