
Survey of Garbage Collection Flavors

Depending on your perspective, the GC is either the best feature in the .NET framework, or the bane of modern application development. Throughout the various .NET versions, we have received multiple implementations, each trying to improve on earlier ones, and thereby subtly changing the rules.

In the beginning, there were just two flavors: **Server** and **Workstation** (renamed to Concurrent GC at some stage, but we'll stick with "Workstation" for now). Workstation GC is the default and is the only option available on a single processor computer, and Server GC is available whenever you have more than one processor in the machine. As has been touched upon in Chapter 4, Server GC is optimized for throughput, meaning you get a GC thread for each available processor, each of which runs in parallel.

There are several key points to consider when deciding between Server and Workstation garbage collection. The main thing to bear in mind is that Workstation GC favors responsiveness and Server GC favors throughput: you get fewer things done over a long period of time with Workstation GC, but the application will respond more quickly to any individual action.

Workstation GC runs on the same thread that triggered the garbage collection. This thread will typically run at normal priority, and must compete with all the other running threads for processing time. Since this thread is running at normal priority, it will easily be preempted by the other threads that the operating system is running. This can mean that garbage collecting takes longer than if it had a dedicated thread that was set at a higher priority. On the other hand, Server GC uses multiple threads, each running at the highest priority, making it significantly faster than Workstation GC.

However, Server GC can be resource intensive. With Server GC, a separate high priority thread is spawned for each processor. Each of these threads will be given the highest priority, so they will finish substantially faster than under a workstation equivalent, but they will also preempt all other threads, so all that speed is not free. Consider a web server with 4 processors, and running 4 web applications. If all of these web applications are configured to run with Server GC, there will be 16 threads running at the highest priority, all dedicated to garbage collection.

This leads to the "freeze" that is often experienced during garbage collection in server mode. As the number of processors or the number of applications is increased, the number of threads involved can lead to excessive time context switching, which may affect overall performance. In extreme circumstances, the extra threads created by the Server mode GC may have the opposite effect and degrade performance. Thus, if you are running multiple applications on the same computer or have a shared hosted web server, you *may* want to consider Workstation GC over Server GC. I stress "may," because this *does* go against conventional wisdom and will not necessarily be the correct choice in all scenarios.

Every environment is different. You need to test and profile your application under load, and repeat these tests as your configuration or your load profile changes. Upgrading the number of processors on a server may change the ideal configuration. Adding a new instance of the application, may change the recommendation.

In most cases, you will probably want to use Server GC whenever you are running server processes on a server OS but, on the high end of scaling, this may not always be the case. Essentially, while Server GC is faster than Workstation GC on the same size heap, there can be problems if you're running hundreds of instances of an app because of excessive context switching between threads

Added to this mix, we have **Concurrent GC**, which splits the garbage collection process in two and handles the two cases differently. Concurrent GC is available only under Workstation mode. Generation 0 and 1 collections are handled normally, since they will typically finish quickly anyway, but Generation 2 GCs are handled a little differently. This is where the concurrent part comes into play because, to a certain extent, your program can continue running while the Generation 2 garbage is collected.

Without Concurrent GC, all threads in your application are blocked until the garbage collection is complete. The concurrent part comes into play by allowing your code to continue executing during the Generation 2 GC, albeit within certain strict limitations. Your program can continue running *until* the active thread's ephemeral segment is full (the ephemeral segment is the segment that was most recently allocated by the GC). When you run out of room on this segment, you will be back to being completely blocked until garbage collection is finished.

While the total time to run garbage collection will be longer in Concurrent mode, the amount of lag time or halted processing will be reduced, from a user's perspective. In fact, for a large portion of the time, you may not be blocked at all. The default is to enable Concurrent GC whenever you are running under Workstation mode. However, if you explicitly switch from Server mode to Workstation mode as mentioned earlier (in a shared hosting environment), you should also explicitly disable Concurrent GC. With that in mind, you can easily disable/enable Concurrent GC in the `config` file, as in Listing 6.1.

```
<configuration>
  <runtime>
```

```
    <gcConcurrent enabled=
“true”/>
  </runtime>
</configuration>
```

Listing 6.1: Configuration settings to enable/disable Concurrent GC.

Concurrent GC is intended to minimize screen freezes during garbage collection and is best suited to application scenarios where a responsive UI is the most critical design goal. Unless you are running a server process on a server OS, you will want to keep Concurrent GC enabled.

.NET 4.0 introduced a new flavor to the table: **Background GC.** Technically this isn't really a new option, as it replaces Concurrent GC, although there have been improvements. In particular, Background GC works just like Concurrent GC *except* that your application isn't blocked when the ephemeral segment is full. Instead we can run a foreground Generation 0 and Generation 1 collection against the ephemeral segment. While this happens, instead of blocking the foreground threads because we are garbage collecting, the background thread running the garbage collection is suspended until the foreground is finished. Because this is strictly a Generation 0 and Generation 1 collection against a single segment, this collection will be very fast. This further reduces the latency on the foreground threads.

There are no new configuration settings for Background GC, so when you target the 4.0 version of the framework and enable Concurrent GC, you will get Background GC.