Diagnosing Memory Leaks

Maoni Stephens

.NET GC Architect

02/25/2022

Definition of a managed memory leak

- From mem-doc

A managed memory leak means you have at least one user root that refers to, directly or indirectly, more and more objects as the process runs.

It’s a leak because the GC by definition cannot reclaim memory of these objects so even if the GC tried the hardest (ie, doing a full blocking GC) the heap size still ends up growing.

Measuring memory in an environment with a GC

- The memory area seems to have many, many confusing terms
  - Committed, reserved, virtual, physical, working set…
  - Tools aren't consistent with the terminology
- Virtual memory fundamentals are explained here
  - This talk (slides) explains how the hardware, the OS and the GC are connected (and clarifies many memory terms)
- What do you need to care about wrt memory leaks?
  - Most of the time, committed (of the GC heap)
  - Committed is the amount of physical storage used to store your data
  - Most of the time, this correlates with working set

When to measure?

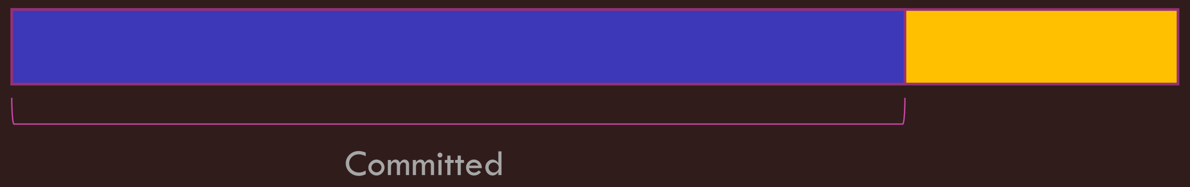
- If you are not asking this question, you should! 😃
  - If you happen to measure on entry of a GC, that's when the heap size is at the largest
  - If you happen to measure on exit of a GC, that's when the heap size is at the smallest
  - If you happen to measure on entry and exit of a GC that compacts the whole heap, there could be a huge difference!
- Note that heap size <= committed size!
- We don't decommit space we know we will be using right away!

What the heap looks like

- Allocation – commit as needed

What the heap looks like

- Allocation – commit as needed

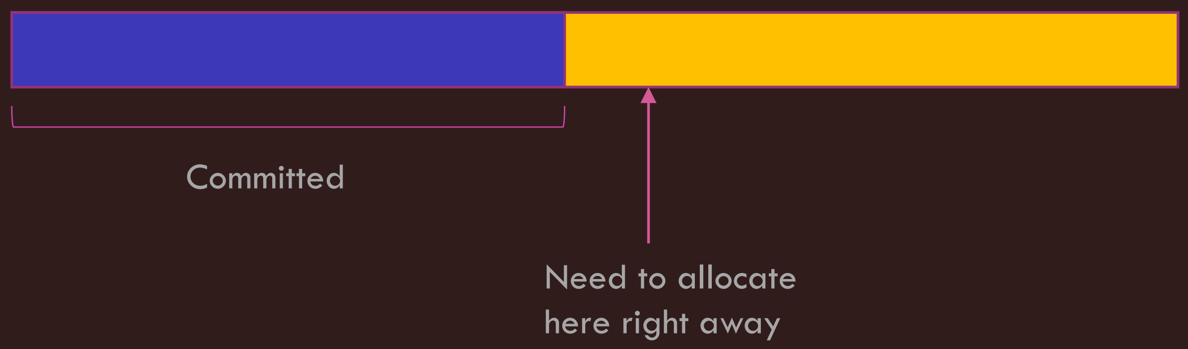


GC happens

- Allocate enough, GC#1 happens, compacted
- If we decommitted all the extra space, it looks like

GC happens

- Allocate enough, GC#1 happens, compacted
- If we decommitted all the extra space, it looks like



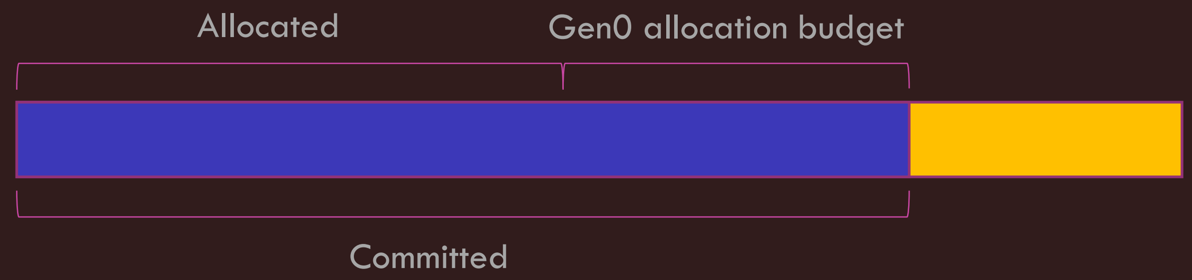
What the heap looks like

- At the end of GC#1

- Allocated is what we call the heap size, because it's occupied by objects
- FAQ: how come my heap size is quite a bit smaller than the memory usage of the process?

What the heap looks like

- At the end of GC#1



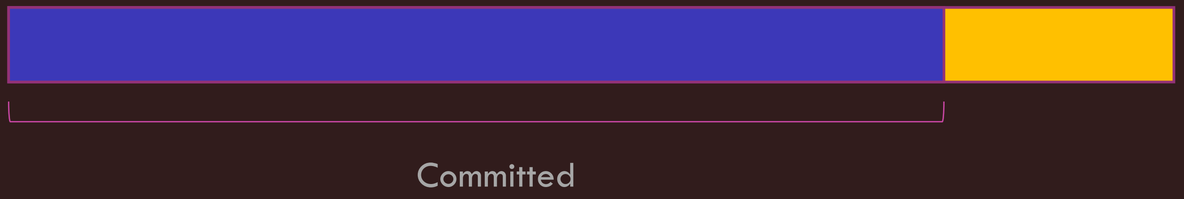
- Allocated is what we call the heap size, because it’s occupied by objects
- FAQ: how come my heap size is quite a bit smaller than the memory usage of the process?
  - The space we leave committed after Allocated is the gen0 allocation budget
  - This could especially be true for Server GC with many heaps

However, there's also the generation aspect

- The .NET GC is a generational GC
  - 3 generations
  - Gen0/1/2 – SOH (Small Object Heap)
    - Gen0 and Gen1 are ephemeral generations
  - Gen3/4 – UOH (User Old heap)
    - Gen3 – LOH (Large Object Heap)
    - Gen4 – POH (Pinned Object Heap, added in .NET 5)
- GC chooses to do a gen0, gen1 or gen2 GC
  - Gen2 GCs collect the whole heap, also called full GCs
  - UOH is only collected during gen2 GCs
  - What doesn't get collected will act as "internal roots" to the portion that gets collected
- Gen2/UOH can get very big, as big as needed by the app

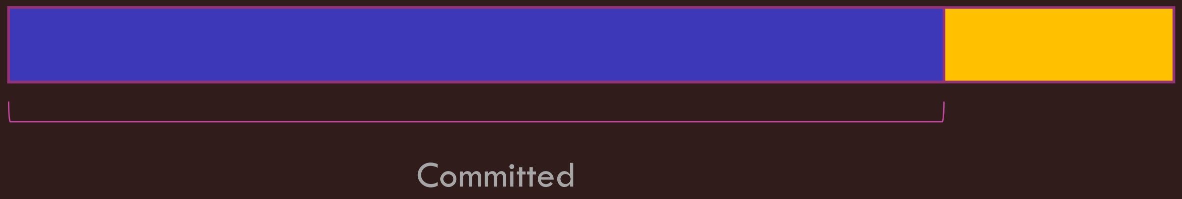
Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn't fluctuate much
- If we do a full GC, commit size *may* fluctuate very much if we compact away a lot of dead space
- Full GCs don't happen nearly as often as ephemeral GCs
- What you might see -
- Gen0 GC - begin



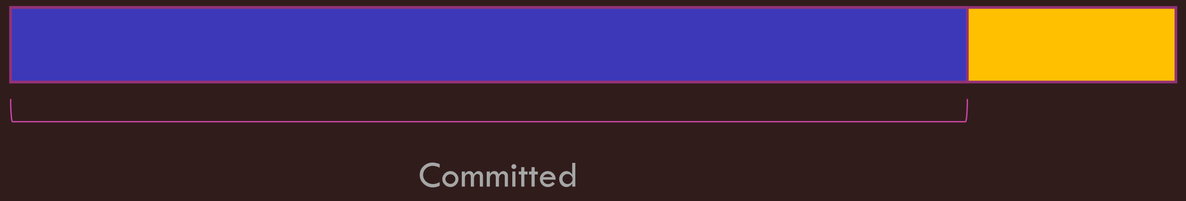
Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn't fluctuate much
- If we do a full GC, commit size may fluctuate very much if we compact away a lot of dead space
- Full GCs don't happen nearly as often as ephemeral GCs
- What you might see -
- Gen0 GC - end



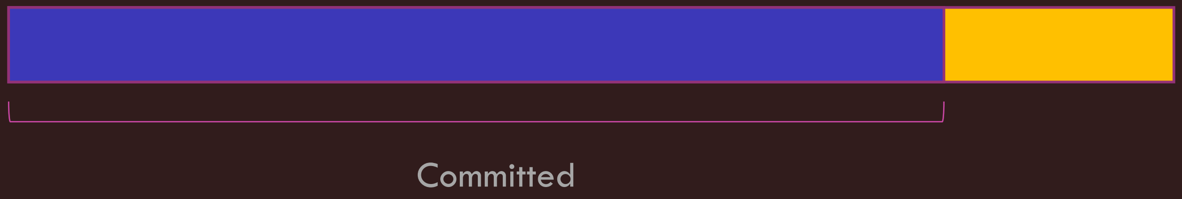
Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn't fluctuate much
- If we do a full GC, commit size may fluctuate very much if we compact away a lot of dead space
- Full GCs don't happen nearly as often as ephemeral GCs
- What you might see -
- Gen0 GC - begin



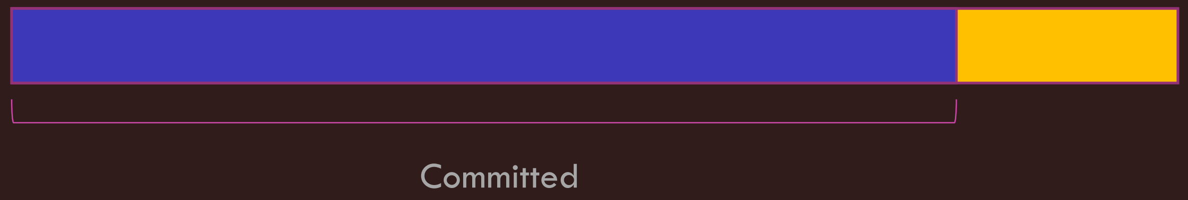
Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn’t fluctuate much
- If we do a full GC, commit size may fluctuate very much if we compact away a lot of dead space
- Full GCs don’t happen nearly as often as ephemeral GCs
- What you might see -
- Gen0 GC - end



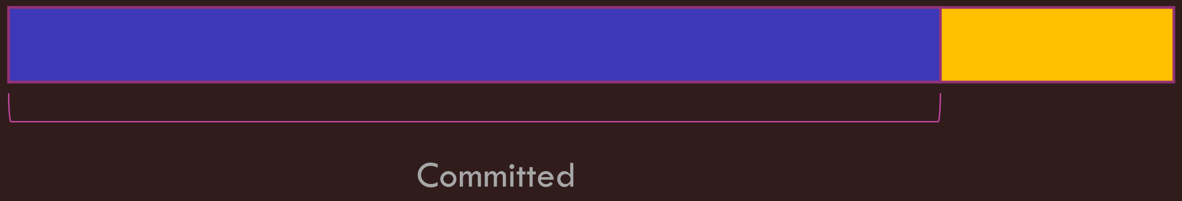
Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn’t fluctuate much
- If we do a full GC, commit size may fluctuate very much if we compact away a lot of dead space
- Full GCs don’t happen nearly as often as ephemeral GCs
- What you might see -
- Gen1 GC - begin



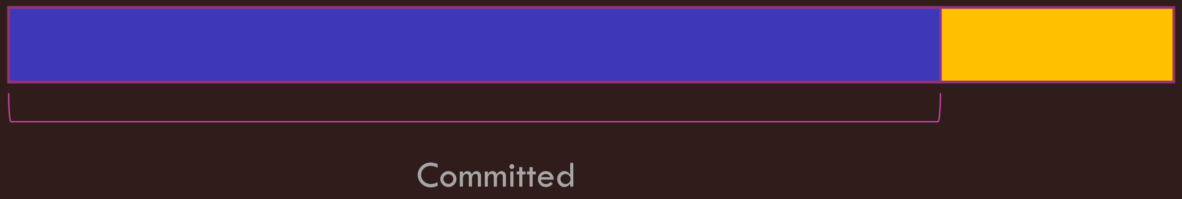
Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn't fluctuate much
- If we do a full GC, commit size may fluctuate very much if we compact away a lot of dead space
- Full GCs don't happen nearly as often as ephemeral GCs
- What you might see -
- Gen1 GC - end



Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn’t fluctuate much
- If we do a full GC, commit size may fluctuate very much if we compact away a lot of dead space
- Full GCs don’t happen nearly as often as ephemeral GCs
- What you might see -
- Gen2 GC - begin



Commit size fluctuation

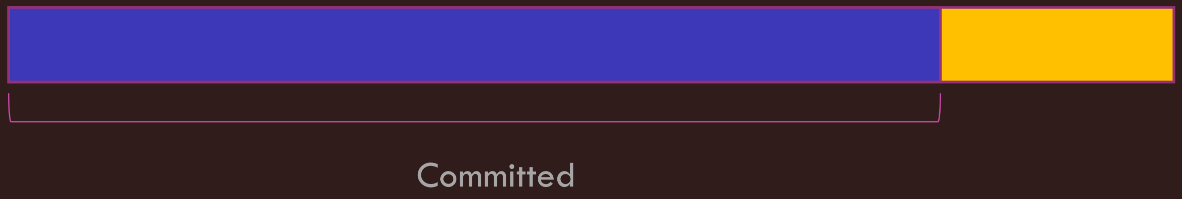
- If we only collect ephemeral generations, commit size doesn't fluctuate much
- If we do a full GC, commit size may fluctuate very much if we compact away a lot of dead space
- Full GCs don't happen nearly as often as ephemeral GCs
- What you might see -
- Gen2 GC - end

STW (Stop-The-World) vs Concurrent

- STW does all its GC work while managed threads are paused
  - Called Blocking GC or Non Concurrent GC (NGC)
- Concurrent does most of its GC work concurrently with managed threads
  - Called Background GC (BGC)
  - Background GC is only for full GCs
  - BGCs do not compact! So the heap size does not change much
  - BGC’s job is to build up free lists that will accommodate survivors from gen1 GCs

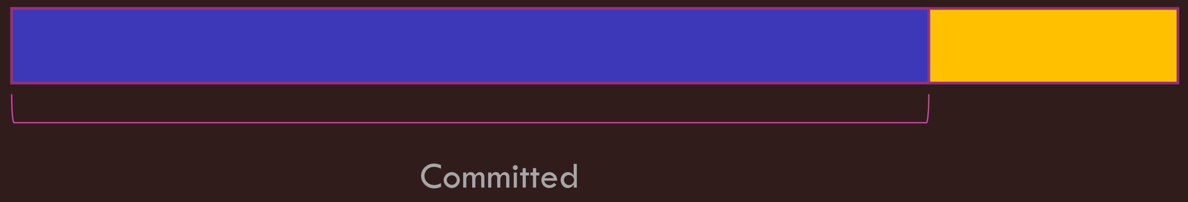
Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn't fluctuate much
- If we do a full GC that's a BGC
- BGC does not compact
- What you might see -
- BGC - begin



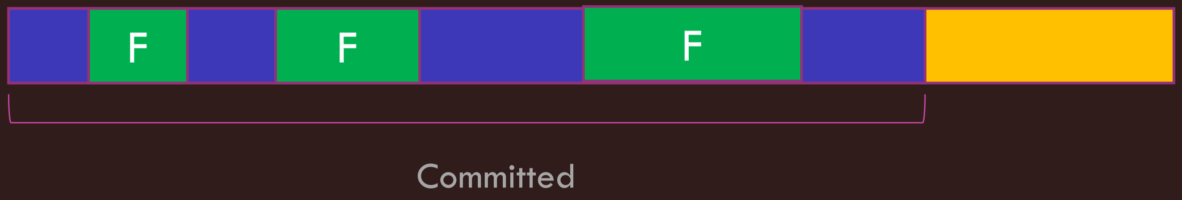
Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn’t fluctuate much
- If we do a full GC that’s a BGC
- BGC does not compact
- What you might see -
- BGC - end



Commit size fluctuation

- If we only collect ephemeral generations, commit size doesn't fluctuate much
- If we do a full GC that's a BGC
- BGC does not compact
- What you might see -
- BGC - end



How do we decide which generation to collect?

- Most significant factors
  - Each generation maintains its own allocation budget
  - When physical memory load is high (>= 90%) it means gen2 GCs will much more likely be compacting
- Perf characteristics
  - If it's not under memory pressure, you'll likely get ephemeral GCs (gen0/gen1 GCs) and BGCs
    - And you could see a significant amount of fragmentation in gen2
    - Again – depending on when you measure
  - Else you will likely get full compacting GCs and the pauses could be long
- How do you draw conclusions whether you have a memory leak?
  - If LDS (Live Data Size) is increasing, it means there's a leak
  - Only gen2 GCs give you LDS (called "Promoted bytes" in tooling)
  - LDS = total heap size – fragmentation, ie, space occupied by your objects
    - And GC cannot reclaim these objects because they are held live by your code (meaning code you wrote and libraries you use)

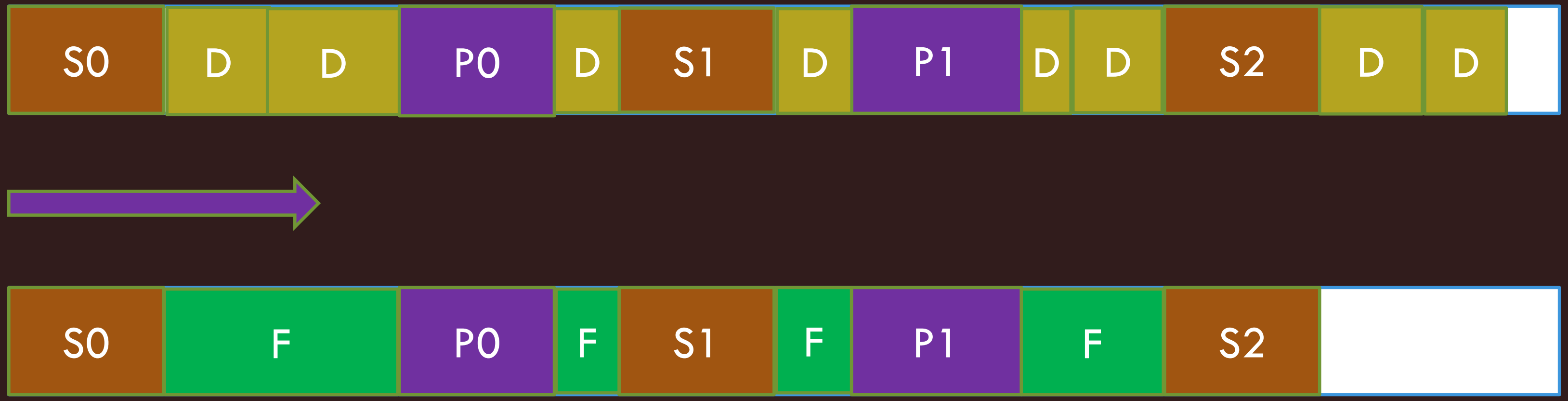
Pinning

- Pinning is pretty prevalent when you have a Server app due to network IO
- Pinning without thinking about the implications should absolutely be avoided
- When an object cannot be moved it causes a lot of stress on the GC
  - This is the only reason fragmentation still exists after a full compacting GC
  - It can also make the heap size larger and larger in BGC if there's more and more pins that "stretch" out the heap

Sn - Survived object; Pn - Pinned object; D - Dead object; F – Free object

Compacting GC – more expensive and can reduce heap size dramatically

Sweeping GC – cheaper but barely reduces heap size



Fragmentation – the sum of free objects

- Used to accommodate survivors from the lower generation (or user allocations for gen0)

When the problem is simple

- Simple means it's easy to repro, overhead of profiling is irrelevant
- Pretty much any tool will do
- Common approach – take a few dumps with some time in between

Counters that give you committed size

- .NET Framework
  - Perf counter: .NET Memory\# of total committed bytes
- .NET 6.0
  - Dotnet counter: committed bytes

Finding out if you have a memory leak

- PerfView uses GC events to show you a rich set of info in GCStats
  - [How to collect top level GC events]()
  - GCStats includes info that tells you when you have a managed memory leak – Promoted MB for gen2 GCs
  - Example

GC Index	Trigger Reason	Gen	Peak MB	After MB	Ratio Peak/After	Promoted MB
547	AllocSmall	2B	36,571.01	32,122.75	1.14	17,837.62
586	AllocLarge	2B	35,063.13	32,970.99	1.06	17,822.01
624	AllocSmall	2B	35,948.02	32,084.92	1.12	17,831.78
662	AllocLarge	2B	34,204.18	33,955.79	0.98	17,832.79

What if I don't want as much fragmentation?

- .NET 6 added a new config (also available in .NET Framework 4.8)

  `COMPlus_GCConserveMemory/DOTNET_GCConserveMemory`

  - Integer value of 1-9 that tells the GC how conservative you want it to be

- Good for capacity planning (because it's more predictable)

Finding out if you have a memory leak

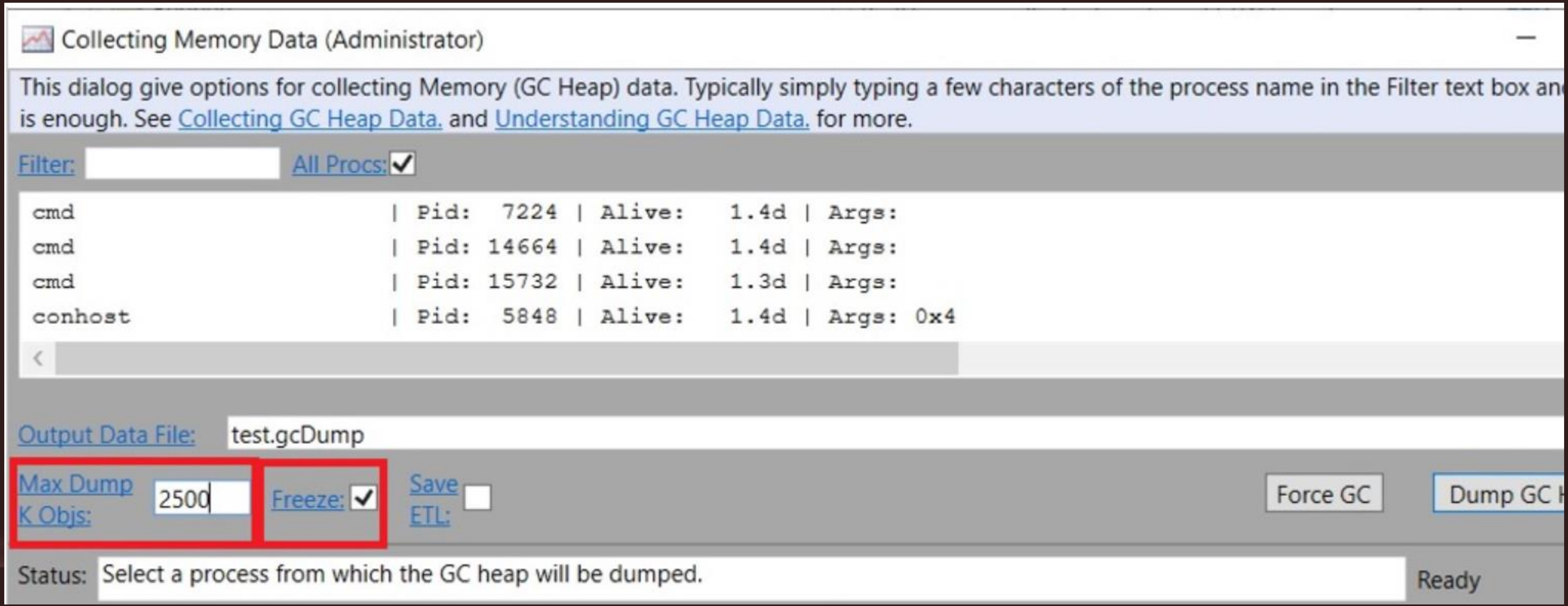
- .NET 5 introduced a new API to allow rich and easy in-proc monitoring

```
public static GCMemoryInfo GetGCMemoryInfo(GCKind kind);

public enum GCKind
{
    FullBlocking = 2,
    Background = 3
};
public readonly struct GCGenerationInfo
{
    public long SizeAfterBytes;
    public long FragmentationAfterBytes;
}
```

```
GCMemoryInfo memoryInfoLastNGC2 = GC.GetGCMemoryInfo(GCKind.FullBlocking);
long lastNGC2Index = memoryInfoLastNGC2.Index;
long LDS = 0;
int numGenerations = memoryInfoLastNGC2.GenerationInfo.Length;

for (int i = 0; i < numGenerations; i++)
{
    long genSize = memoryInfoLastNGC2.GenerationInfo[i].SizeAfterBytes -
                   memoryInfoLastNGC2.GenerationInfo[i].FragmentationAfterBytes;
    LDS += genSize;
}
```

If you do have a memory leak

- PerfView allows you to collect a heap snapshot that shows you object types/sizes and connectivity
  - Uncheck the Freeze option to avoid having to pause user threads for long
  - If you already captured a process dump, you can also load that into PerfView



Generation aware analysis in .NET 5

- Very different from “being able to look at different generations of a heap”!!!
- You can tell the runtime when you’d like to capture a trace by these filters
  - generation of the GC (thus generation aware)
  - min survived bytes observed in this GC
  - min GC index (to avoid certain phases)
- It was added mostly for debugging ephemeral GC problems
- But also used for debugging memory leaks
  - Especially useful if the OOMs only occur on some machines

Generation aware analysis in .NET 5

- Example for debugging memory leaks

  ```
  set COMPlus_GCGenAnalysisGen=2
  set COMPlus_GCGenAnalysisBytes=40000000
  set COMPlus_GCGenAnalysisIndex=3E8
  ```

  During a full blocking GC, if it survives at least 1GiB and GC index is at least 1000, capture a trace

- .NET 6 added an option to capture a dump

  ```
  set COMPlus_GCGenAnalysisDump=1
  ```

“Why isn’t the GC collecting my object??”

```
public static int Main()
{
    MaoniType o = new MaoniType(128, 256);
    GCHandle h = GCHandle.Alloc(o, GCHandleType.Weak);
    GC.Collect();
    Console.WriteLine("Collect called, h.Target is {0}",
        (h.Target == null) ? "collected" : "not collected");
    return 0;
}
```

Output - Collect called, h.Target is not collected

```csharp
[MethodImpl(MethodImplOptions.NoInlining)]
public static void TestLifeTime()
{
    MaoniType o = new MaoniType(128, 256);
    h = GCHandle.Alloc(o, GCHandleType.Weak);
}
public static int Main()
{
    TestLifeTime();
    GC.Collect();
    Console.WriteLine("Collect called, h.Target is {0}",
        (h.Target == null) ? "collected" : "not collected");
    return 0;
}
```

Output: Collect called, h.Target is collected