GC modes, latency modes and other settings

GC modes

- so again, the GC trilemma problem - there is no *one-size-fits-all* GC.

GC modes

- so again, the GC trilemma problem - there is no *one-size-fits-all* GC.
- *"the GC team is always working toward having the .NET GC handle more and more perf scenarios so the users need to worry less"*

GC modes

- so again, the GC trilemma problem - there is no *one-size-fits-all* GC.
- *"the GC team is always working toward having the .NET GC handle more and more perf scenarios so the users need to worry less"*
- *"The philosophy of the .NET GC is we try to handle as much automatically as we can"* - Maoni Stephens

GC modes

- so again, the GC trilemma problem - there is no *one-size-fits-all* GC.
- *"the GC team is always working toward having the .NET GC handle more and more perf scenarios so the users need to worry less"*
- *"The philosophy of the .NET GC is we try to handle as much automatically as we can"* - Maoni Stephens
- but there always will be those edge cases... to configure

Java

Table 2: Available Options by GC Type.

Classification	Option	Remarks
Serial GC	-XX:+UseSerialGC	
Parallel GC	-XX:+UseParallelGC -XX:ParallelGCThreads=value	
Parallel Compacting GC	-XX:+UseParallelOldGC	
CMS GC	-XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=value -XX:+UseCMSInitiatingOccupancyOnly	
G1	-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC	In JDK 6, these two options must be used together.

Java

```
-server -Xms24G -Xmx24G -XX:PermSize=512m -XX:+UseG1GC
-XX:MaxGCPauseMillis=200 -XX:ParallelGCThreads=20
-XX:ConcGCThreads=5
-XX:InitiatingHeapOccupancyPercent=70
```

Java

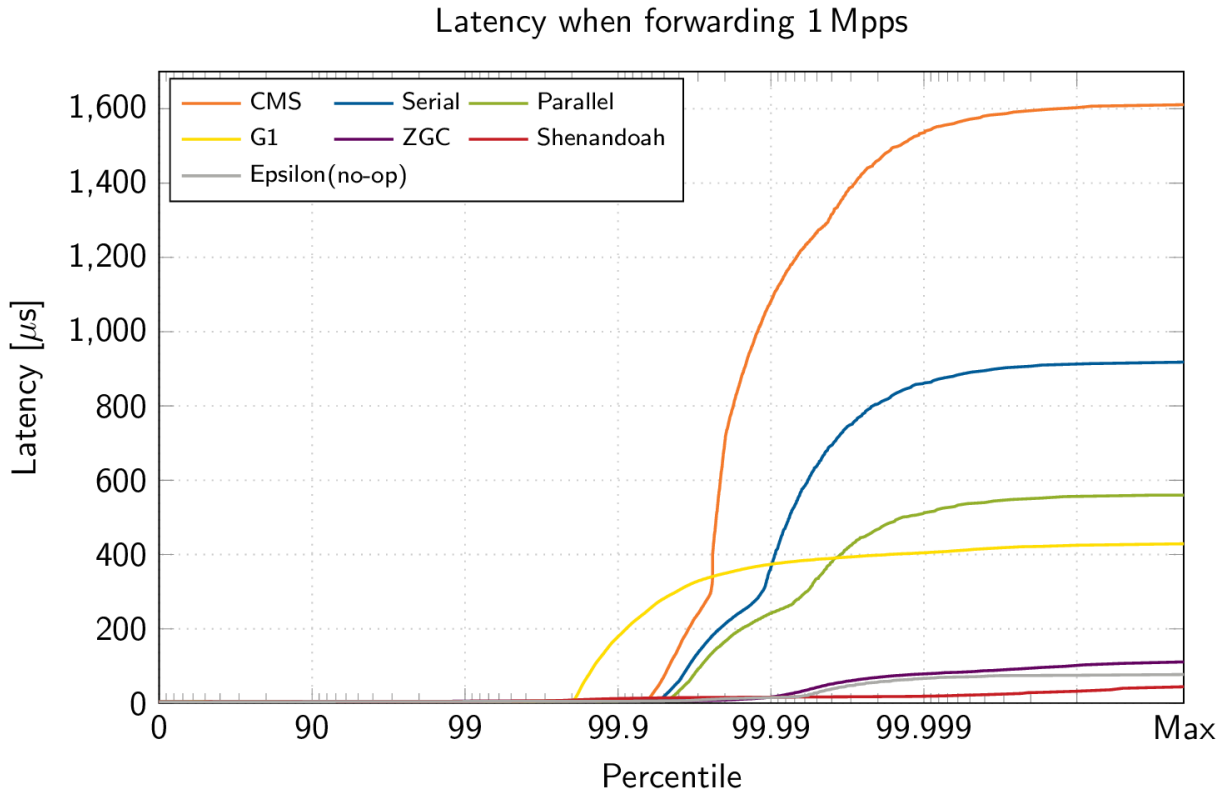
```
-server -Xms24G -Xmx24G -XX:PermSize=512m -XX:+UseG1GC
-XX:MaxGCPauseMillis=200 -XX:ParallelGCThreads=20
-XX:ConcGCThreads=5
-XX:InitiatingHeapOccupancyPercent=70
```

or...

```
-server -Xss4096k -Xms12G -Xmx12G -XX:MaxPermSize=512m
-XX:+HeapDumpOnOutOfMemoryError -verbose:gc -Xmaxf1
-XX:+UseCompressedOops -XX:+DisableExplicitGC -XX:+AggressiveOpts
-XX:+ScavengeBeforeFullGC -XX:CMSFullGCsBeforeCompaction=10
-XX:CMSInitiatingOccupancyFraction=80 -XX:+UseParNewGC
-XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode
-XX:+CMSIncrementalPacing -XX:+CMSParallelRemarkEnabled
-XX:GCTimeRatio=19 -XX:+UseAdaptiveSizePolicy
-XX:MaxGCPauseMillis=500 -XX:+PrintGCTaskTimeStamps
-XX:+PrintGCApplicationStoppedTime -XX:+PrintHeapAtGC
-XX:+PrintTenuringDistribution -XX:+PrintGCDetails
-XX:+PrintGCDateStamps -XX:+PrintGCApplicationConcurrentTime
-XX:+PrintTenuringDistribution -Xloggc:gc.log
```

Java

Java garbage collector comparison in [Ixy - user space network driver] experiment:

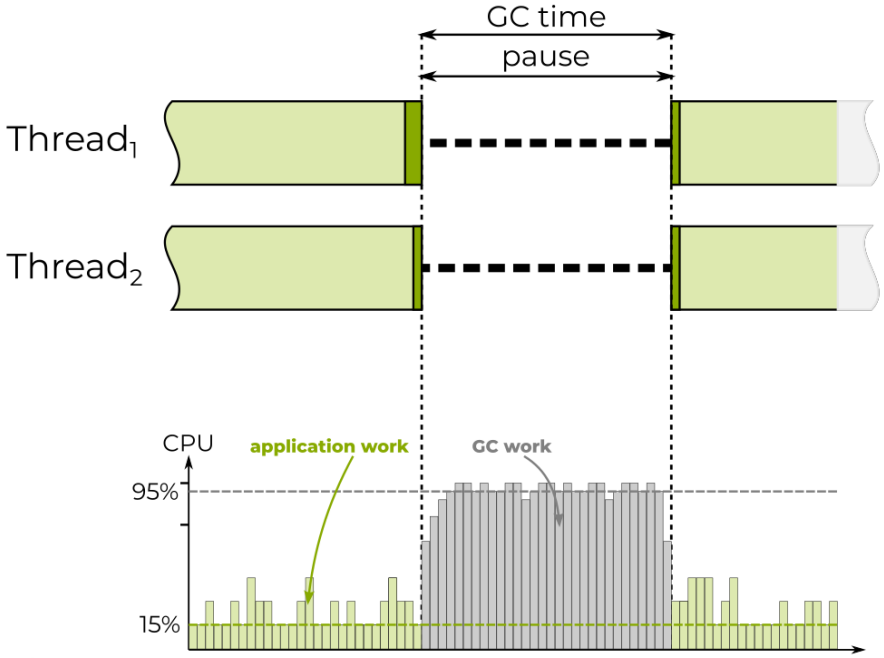


GC modes - Non-Concurrent vs. Concurrent Mode

GC modes - Non-Concurrent vs. Concurrent Mode

Non-Concurrent

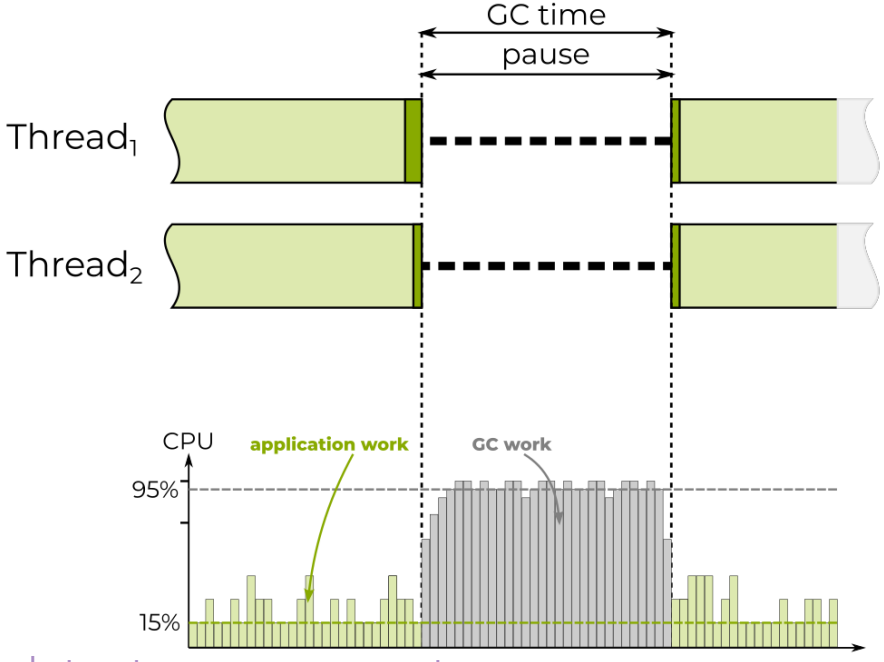
- blocking GCs - all managed threads are suspended (aka *"stop-the-world"* GC)
- pauses - no work, no allocations, no nothing...
- optimal as no floating garbage, everything collected



GC modes - Non-Concurrent vs. Concurrent Mode

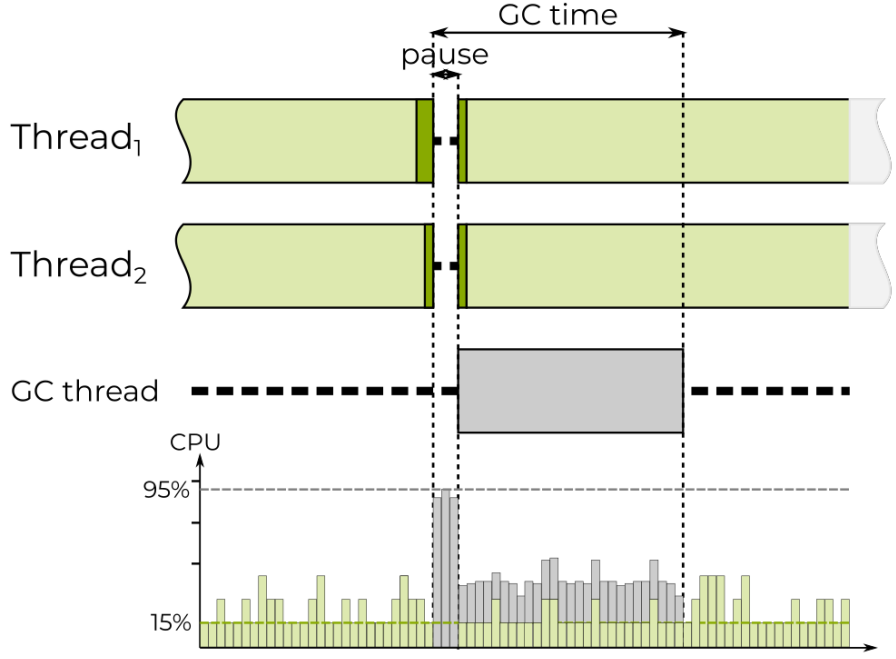
Non-Concurrent

- blocking GCs - all managed threads are suspended (aka *"stop-the-world"* GC)
- pauses - no work, no allocations, no nothing...
- optimal as no floating garbage, everything collected



Concurrent

- ***some parts*** of GC runs concurrently with managed threads
- normal work possible (mostly)
- produces some floating garbage
- not yet implemented compacting



GC modes - Workstation vs. Server Mode

GC modes - Workstation vs. Server Mode

Workstation

- designed mostly for **responsiveness** needed in interactive, UI-based applications
  - pauses as short as possible
  - good citizen in the whole interactive environment

GC modes - Workstation vs. Server Mode

Workstation

- designed mostly for **responsiveness** needed in interactive, UI-based applications
  - pauses as short as possible
  - good citizen in the whole interactive environment
- GCs will happen more frequently
  - they will have less work to do (shorter pauses)
  - lower memory usage

GC modes - Workstation vs. Server Mode

Workstation

- designed mostly for **responsiveness** needed in interactive, UI-based applications
  - pauses as short as possible
  - good citizen in the whole interactive environment
- GCs will happen more frequently
  - they will have less work to do (shorter pauses)
  - lower memory usage
- single Managed Heap
  - during the GC a single thread is used
  - does not utilize resources optimally
  - segments are smaller

GC modes - Workstation vs. Server Mode

Workstation

- designed mostly for **responsiveness** needed in interactive, UI-based applications
  - pauses as short as possible
  - good citizen in the whole interactive environment
- GCs will happen more frequently
  - they will have less work to do (shorter pauses)
  - lower memory usage
- single Managed Heap
  - during the GC a single thread is used
  - does not utilize resources optimally
  - segments are smaller

Server

- designed for simultaneous, **request-based** processing applications
  - big throughput (pauses may be unpredictable, final throughput is what matters)
  - "give me all" citizen in the system

GC modes - Workstation vs. Server Mode

Workstation

- designed mostly for **responsiveness** needed in interactive, UI-based applications
  - pauses as short as possible
  - good citizen in the whole interactive environment
- GCs will happen more frequently
  - they will have less work to do (shorter pauses)
  - lower memory usage
- single Managed Heap
  - during the GC a single thread is used
  - does not utilize resources optimally
  - segments are smaller

Server

- designed for simultaneous, **request-based** processing applications
  - big throughput (pauses may be unpredictable, final throughput is what matters)
  - "give me all" citizen in the system
- GCs will happen less frequently
  - pauses may be longer...
  - ...but because of parallelization, not necessarily
  - higher memory usage
  - higher CPU usage

GC modes - Workstation vs. Server Mode

Workstation

- designed mostly for **responsiveness** needed in interactive, UI-based applications
  - pauses as short as possible
  - good citizen in the whole interactive environment
- GCs will happen more frequently
  - they will have less work to do (shorter pauses)
  - lower memory usage
- single Managed Heap
  - during the GC a single thread is used
  - does not utilize resources optimally
  - segments are smaller

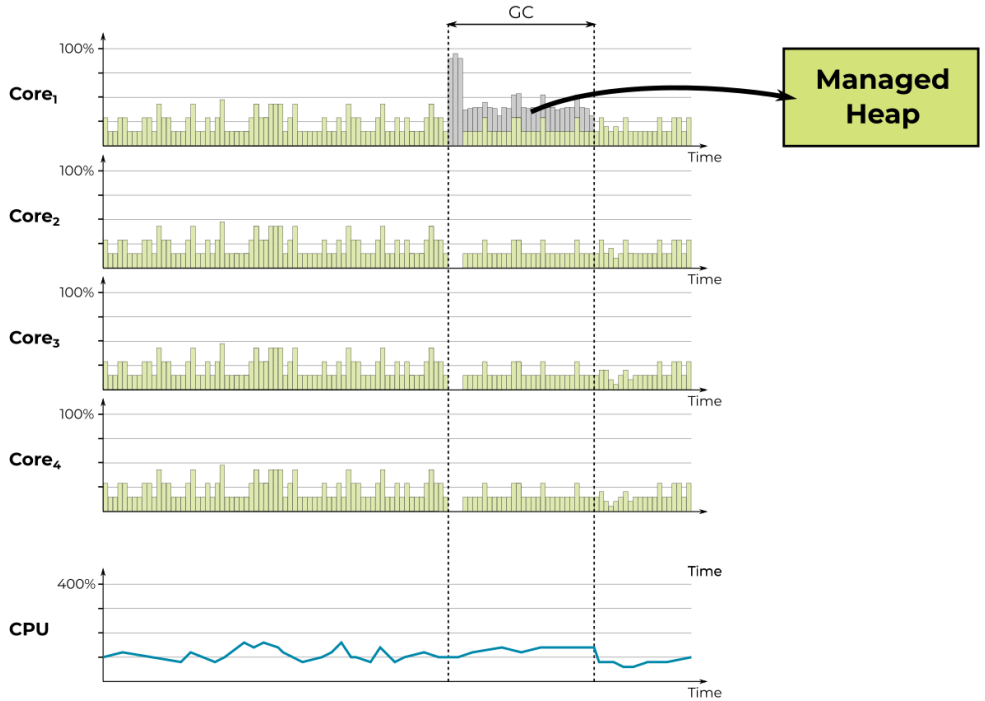
Server

- designed for simultaneous, **request-based** processing applications
  - big throughput (pauses may be unpredictable, final throughput is what matters)
  - "give me all" citizen in the system
- GCs will happen less frequently
  - pauses may be longer...
  - ...but because of parallelization, not necessarily
  - higher memory usage
  - higher CPU usage
- multiple Managed Heaps
  - there is *thread-Heap-core* affinity

    (We will see it soon...)
  - during the GC every Heap uses its own GC thread
  - maximized resources usage

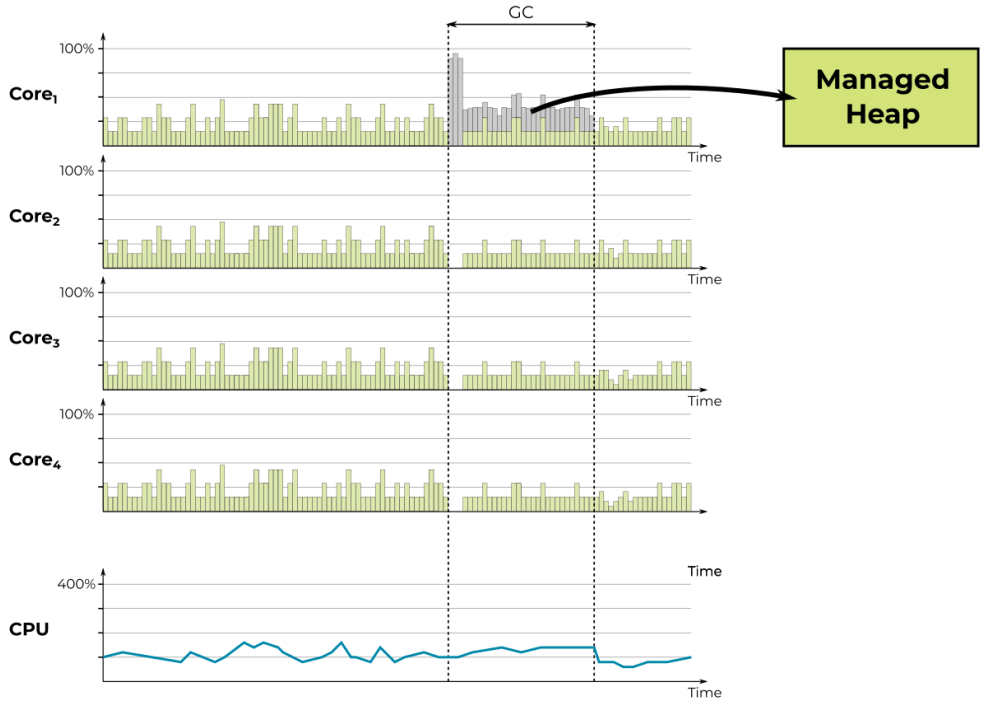
GC modes - Workstation vs. Server Mode

Workstation

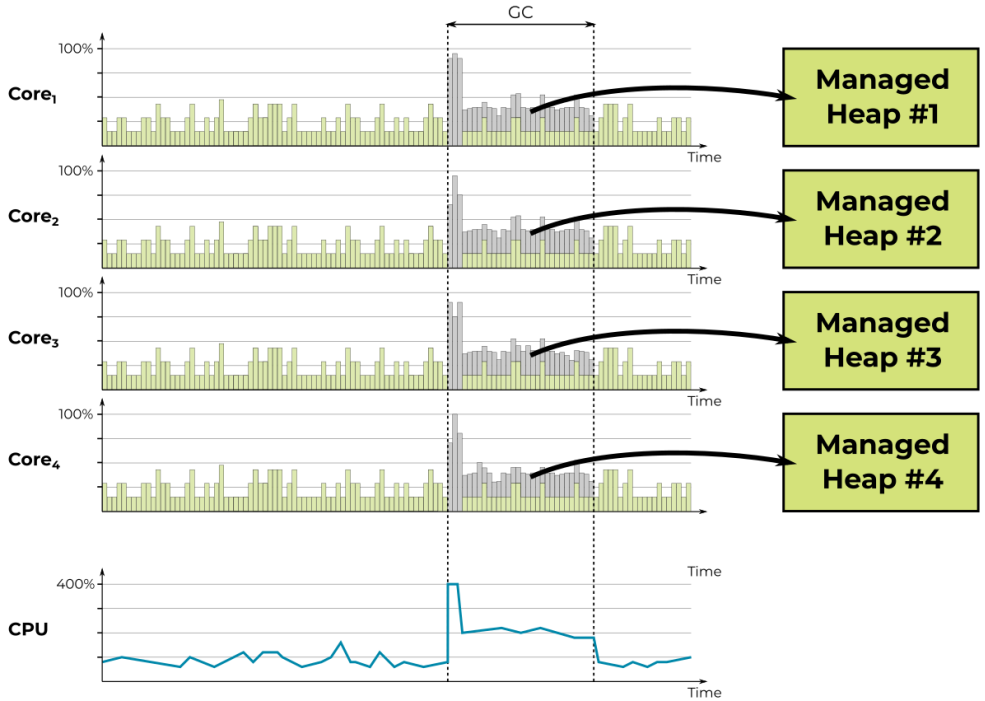


GC modes - Workstation vs. Server Mode

Workstation



Server



.NET GC from 30,000-foot view

Let's stop for a moment - we have (almost) all building blocks in place to draw an overall GC Big Picture (yet, still simplified):

.NET GC from 30,000-foot view

Let's stop for a moment - we have (almost) all building blocks in place to draw an overall GC Big Picture (yet, still simplified):

- GC is mostly triggered by allocations

.NET GC from 30,000-foot view

Let's stop for a moment - we have (almost) all building blocks in place to draw an overall GC Big Picture (yet, still simplified):

- GC is mostly triggered by allocations
- managed threads executing our code are allocating

.NET GC from 30,000-foot view

Let's stop for a moment - we have (almost) all building blocks in place to draw an overall GC Big Picture (yet, still simplified):

- GC is mostly triggered by allocations
- managed threads executing our code are allocating
- GC may be blocking or concurrent

.NET GC from 30,000-foot view

Let's stop for a moment - we have (almost) all building blocks in place to draw an overall GC Big Picture (yet, still simplified):

- GC is mostly triggered by allocations
- managed threads executing our code are allocating
- GC may be blocking or concurrent
- blocking GC may compact or sweep, concurrent GC may only sweep (as for up to .NET 6)

.NET GC from 30,000-foot view

Let's stop for a moment - we have (almost) all building blocks in place to draw an overall GC Big Picture (yet, still simplified):

- GC is mostly triggered by allocations
- managed threads executing our code are allocating
- GC may be blocking or concurrent
- blocking GC may compact or sweep, concurrent GC may only sweep (as for up to .NET 6)
- there may be multiple Managed Heaps with corresponding GC threads

.NET GC from 30,000-foot view

Let's stop for a moment - we have (almost) all building blocks in place to draw an overall GC Big Picture (yet, still simplified):

- GC is mostly triggered by allocations
- managed threads executing our code are allocating
- GC may be blocking or concurrent
- blocking GC may compact or sweep, concurrent GC may only sweep (as for up to .NET 6)
- there may be multiple Managed Heaps with corresponding GC threads
- there is not yet covered *generational* aspect - GC group objects into *young* and *old*

.NET GC from 30,000-foot view

Let's stop for a moment - we have (almost) all building blocks in place to draw an overall GC Big Picture (yet, still simplified):

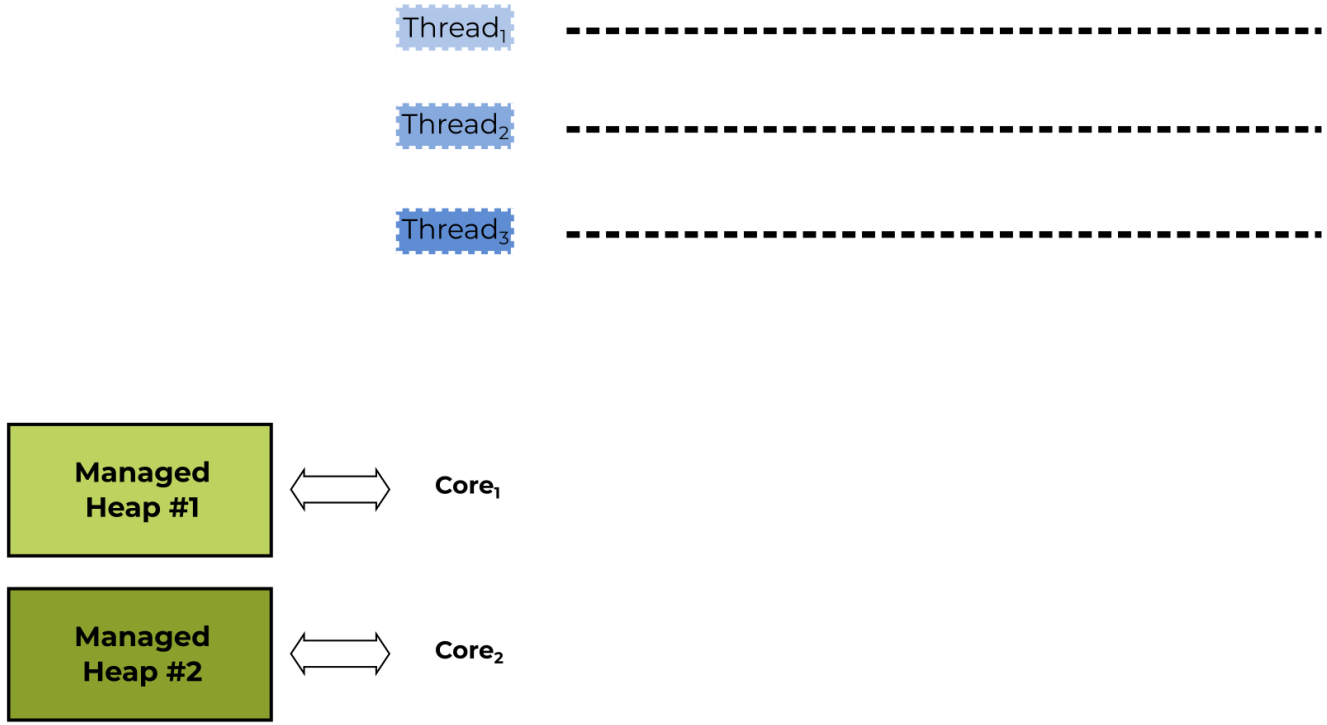
- GC is mostly triggered by allocations
- managed threads executing our code are allocating
- GC may be blocking or concurrent
- blocking GC may compact or sweep, concurrent GC may only sweep (as for up to .NET 6)
- there may be multiple Managed Heaps with corresponding GC threads
- there is not yet covered *generational* aspect - GC group objects into *young* and *old*
- *young GC* collects only *young* dead objects, so-called **Full-GC** collects all dead objects

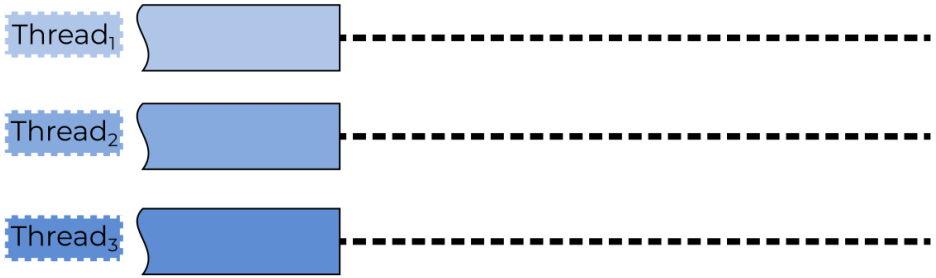
.NET GC from 30,000-foot view

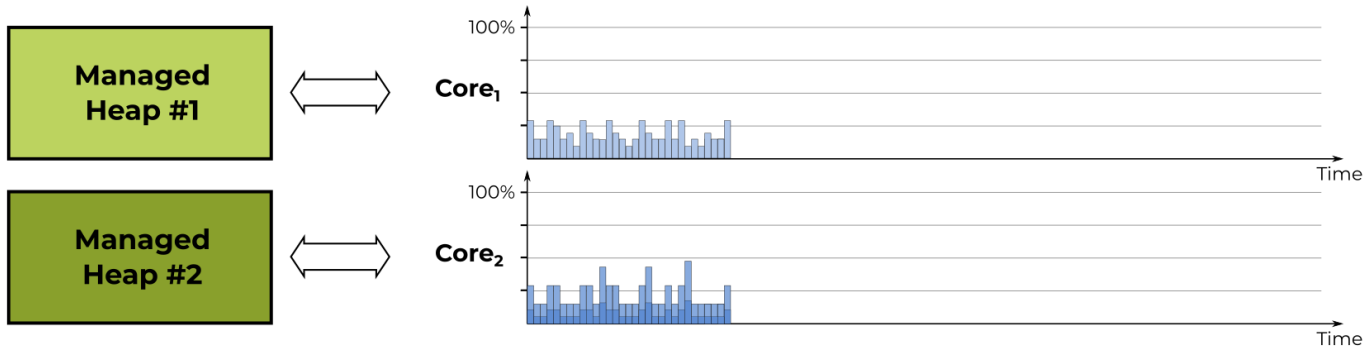
Let's stop for a moment - we have (almost) all building blocks in place to draw an overall GC Big Picture (yet, still simplified):

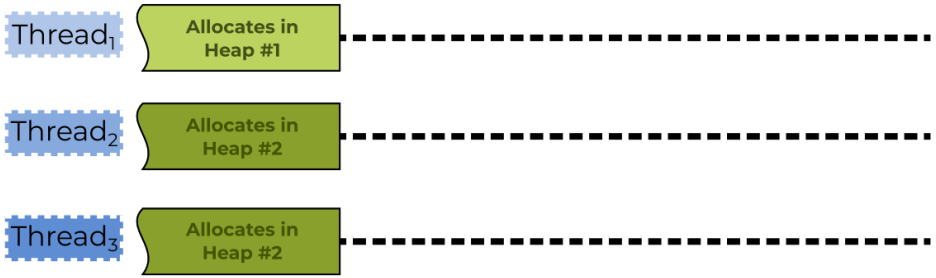
- GC is mostly triggered by allocations
- managed threads executing our code are allocating
- GC may be blocking or concurrent
- blocking GC may compact or sweep, concurrent GC may only sweep (as for up to .NET 6)
- there may be multiple Managed Heaps with corresponding GC threads
- there is not yet covered *generational* aspect - GC group objects into *young* and *old*
- *young GC* collects only *young* dead objects, so-called **Full-GC** collects all dead objects

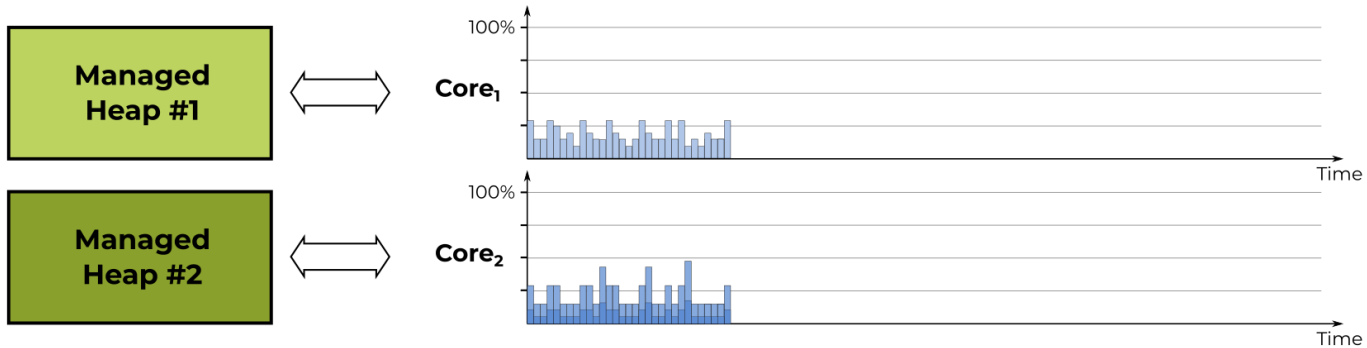
Let's visualize that!

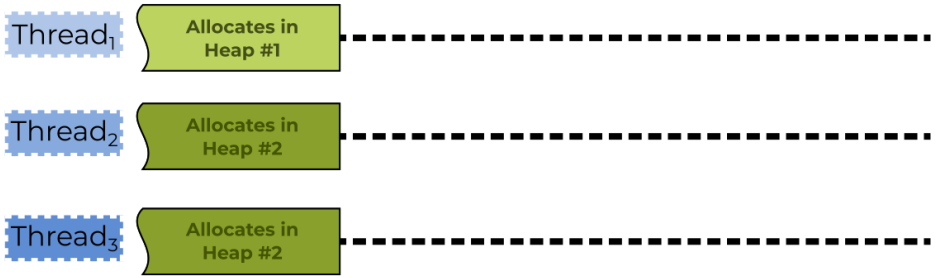

Thread1
Thread2
Thread3
Managed Heap #1
Core1
Managed Heap #2
Core2

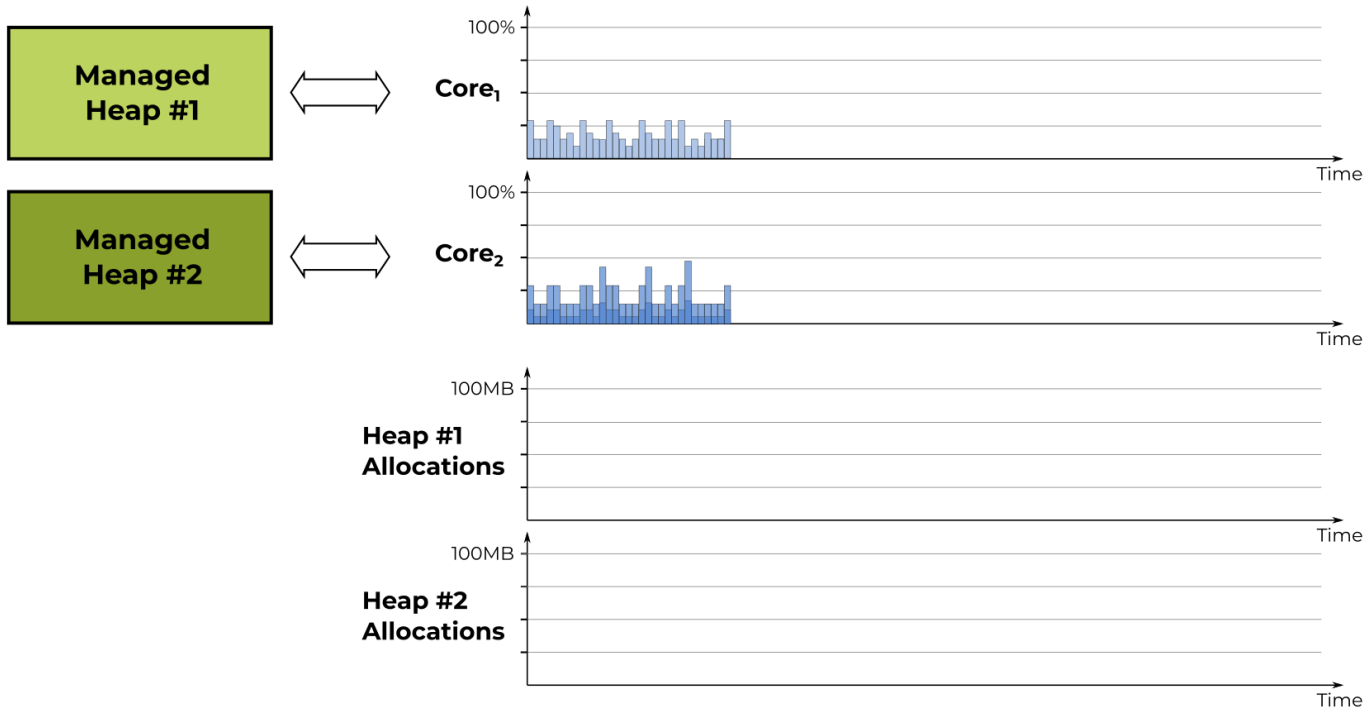

Thread1
Thread2
Thread3

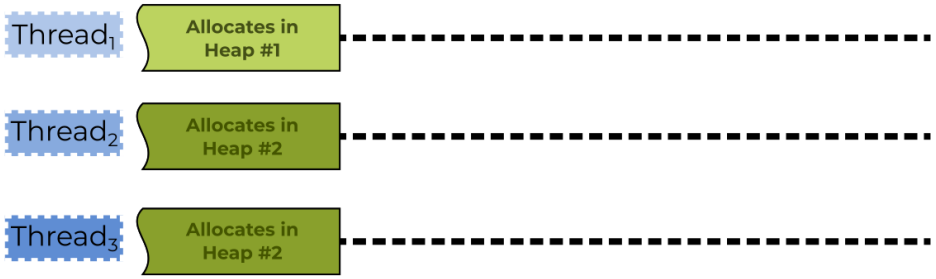

Managed Heap #1
Managed Heap #2
100%
Core1
Time
100%
Core2
Time

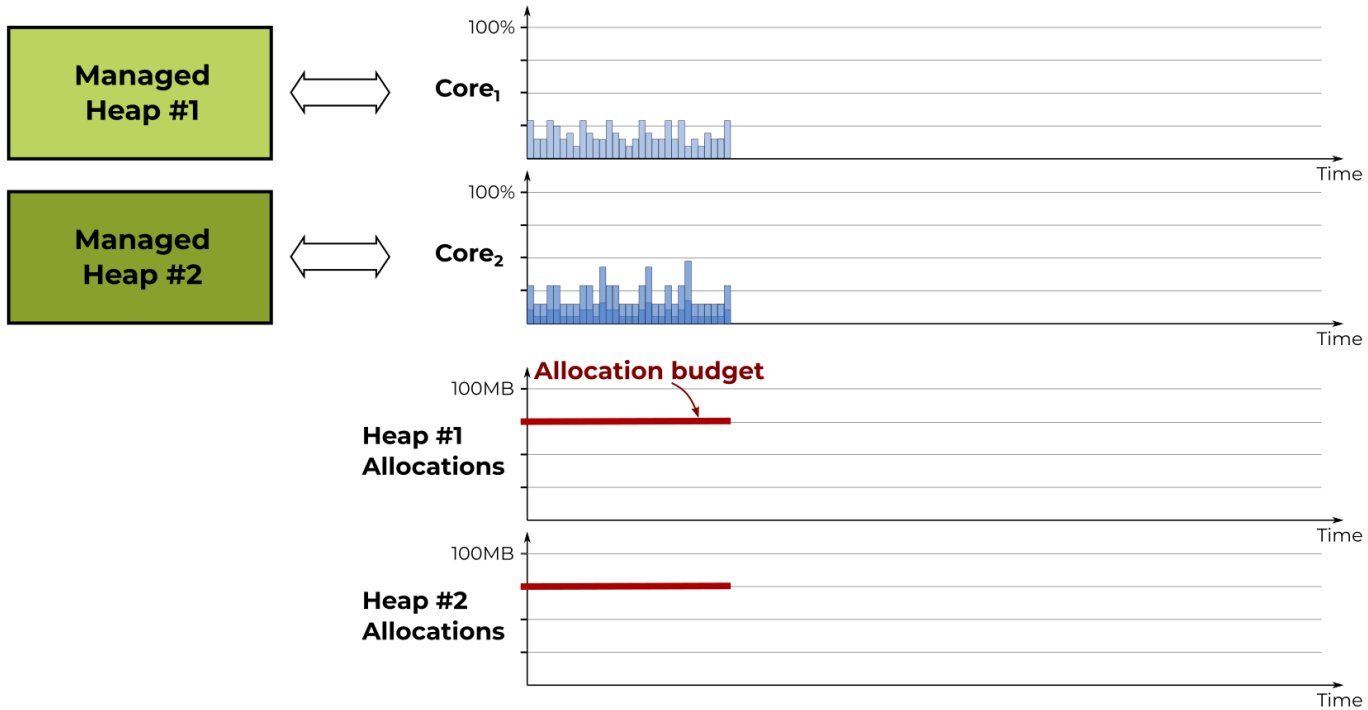

Thread1
Allocates in Heap #1
Thread2
Allocates in Heap #2
Thread3
Allocates in Heap #2

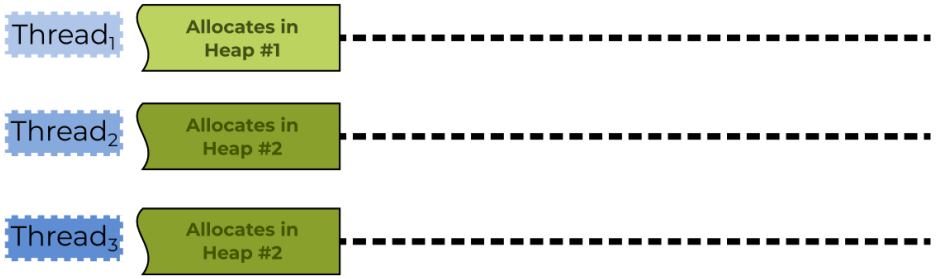

Managed Heap #1
Managed Heap #2
100%
Core1
Time
100%
Core2
Time

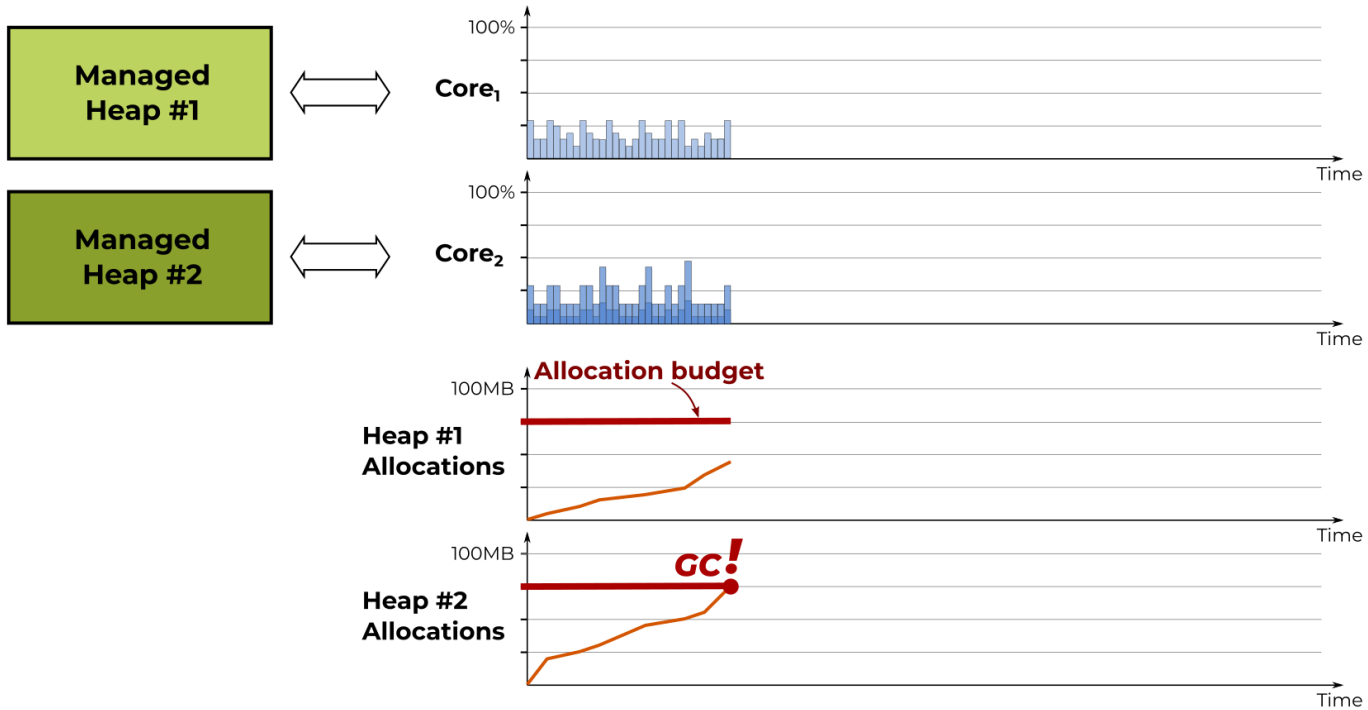

Thread1
Allocates in Heap #1
Thread2
Allocates in Heap #2
Thread3
Allocates in Heap #2

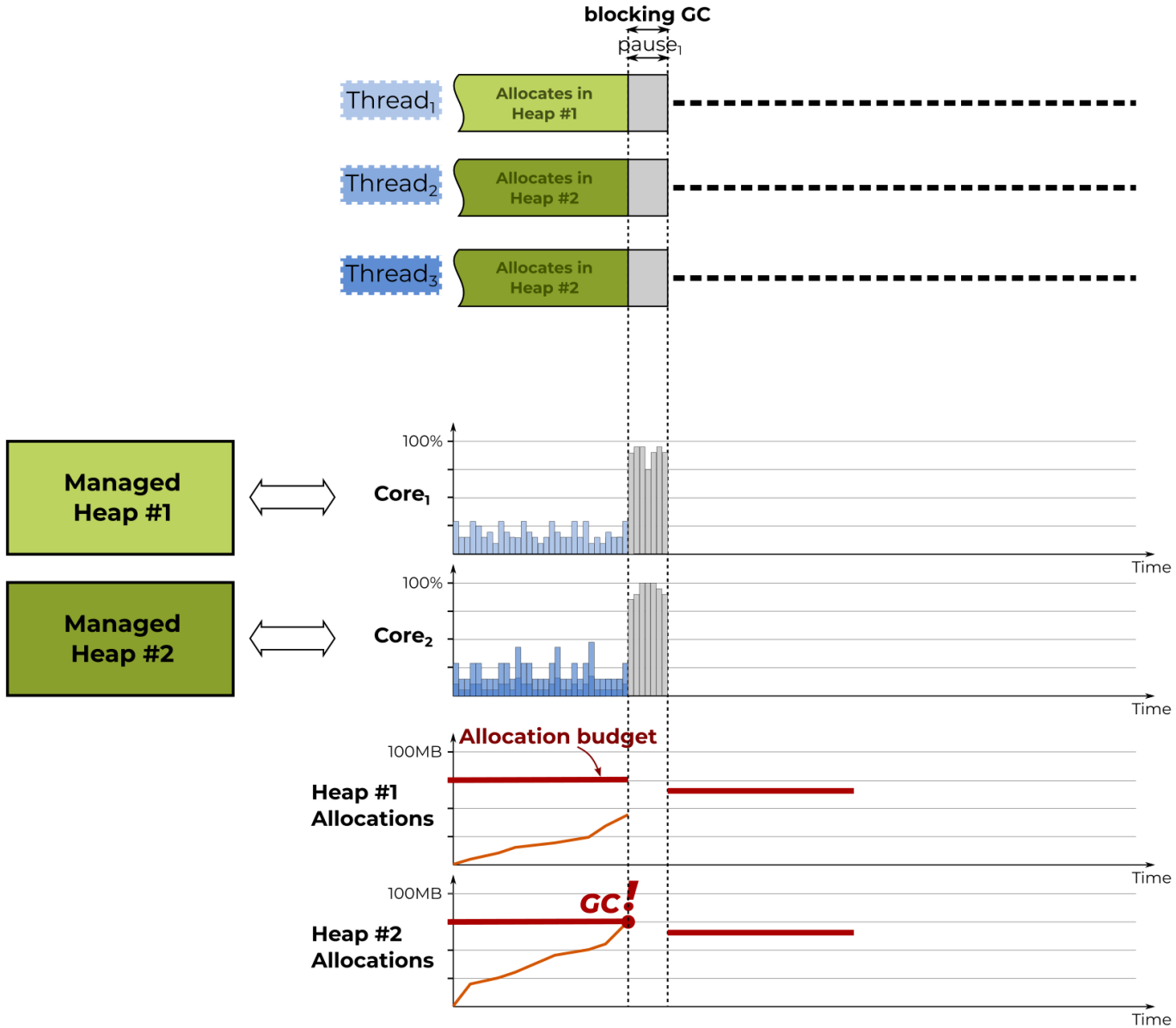

Managed Heap #1
Managed Heap #2
100%
Core1
Time
100%
Core2
Time
100MB
Heap #1 Allocations
Time
100MB
Heap #2 Allocations
Time

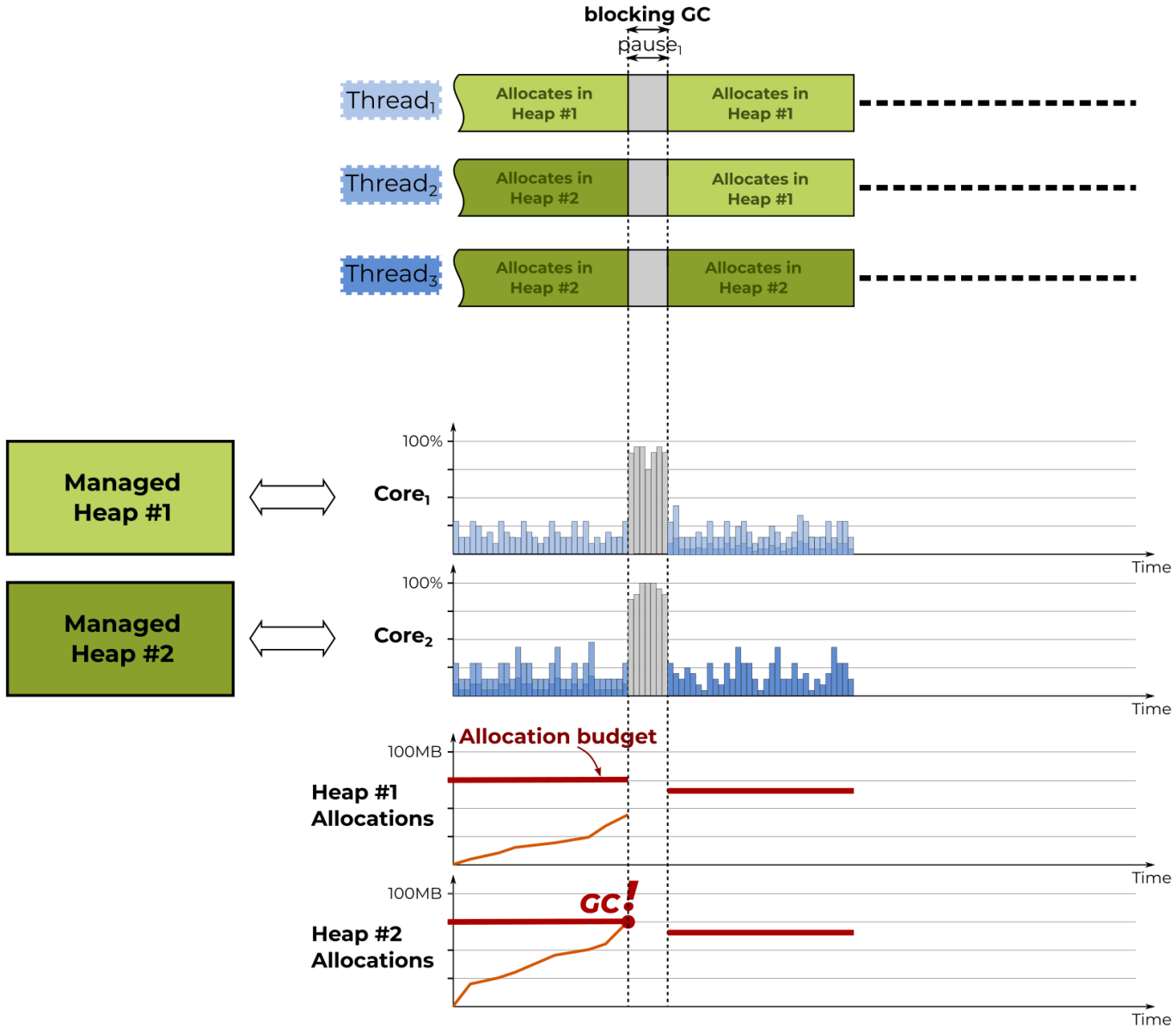

Thread1
Allocates in Heap #1
Thread2
Allocates in Heap #2
Thread3
Allocates in Heap #2

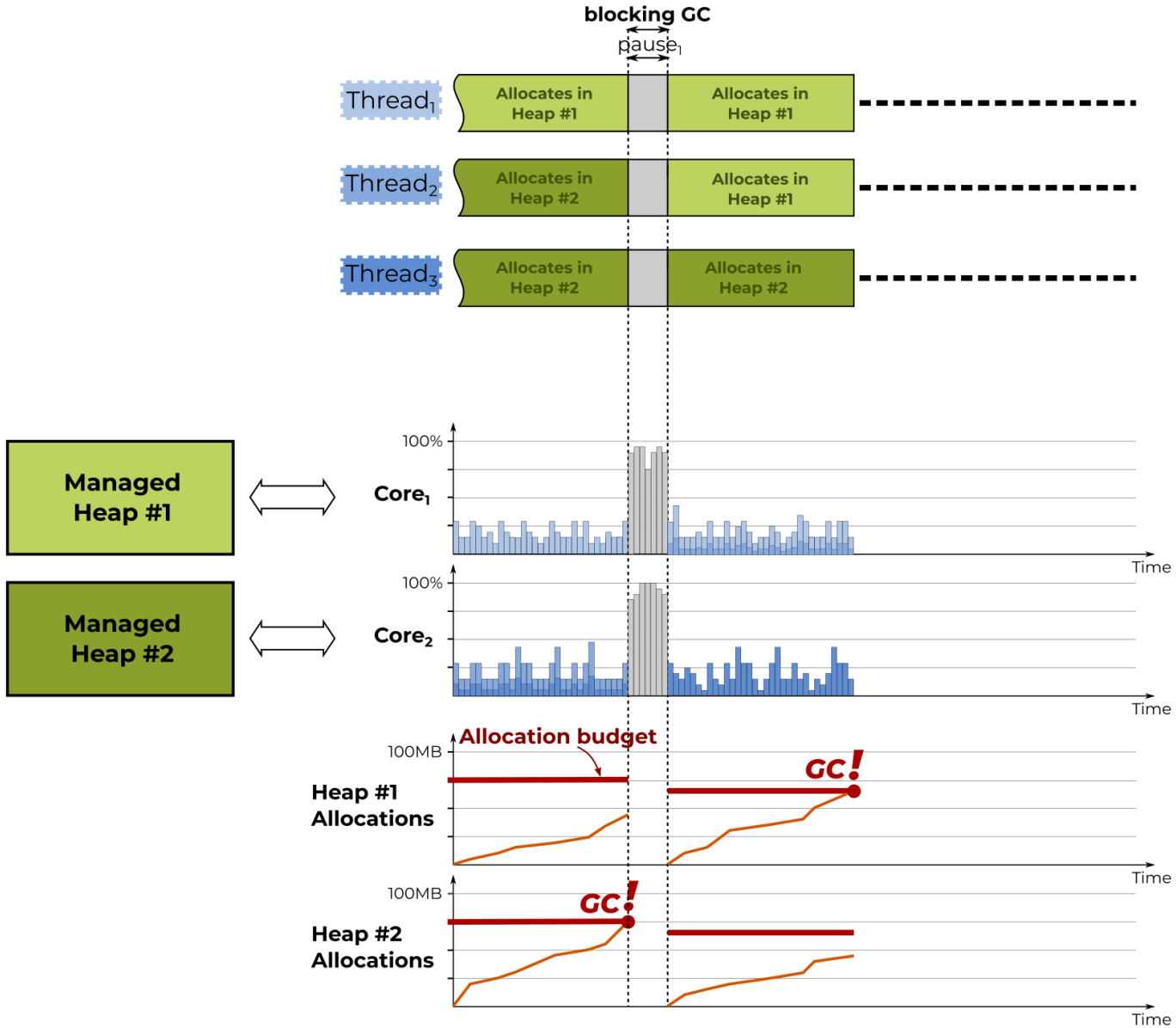

Managed Heap #1
Managed Heap #2
100%
Core1
Time
100%
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Time
100MB
Heap #2 Allocations
Time

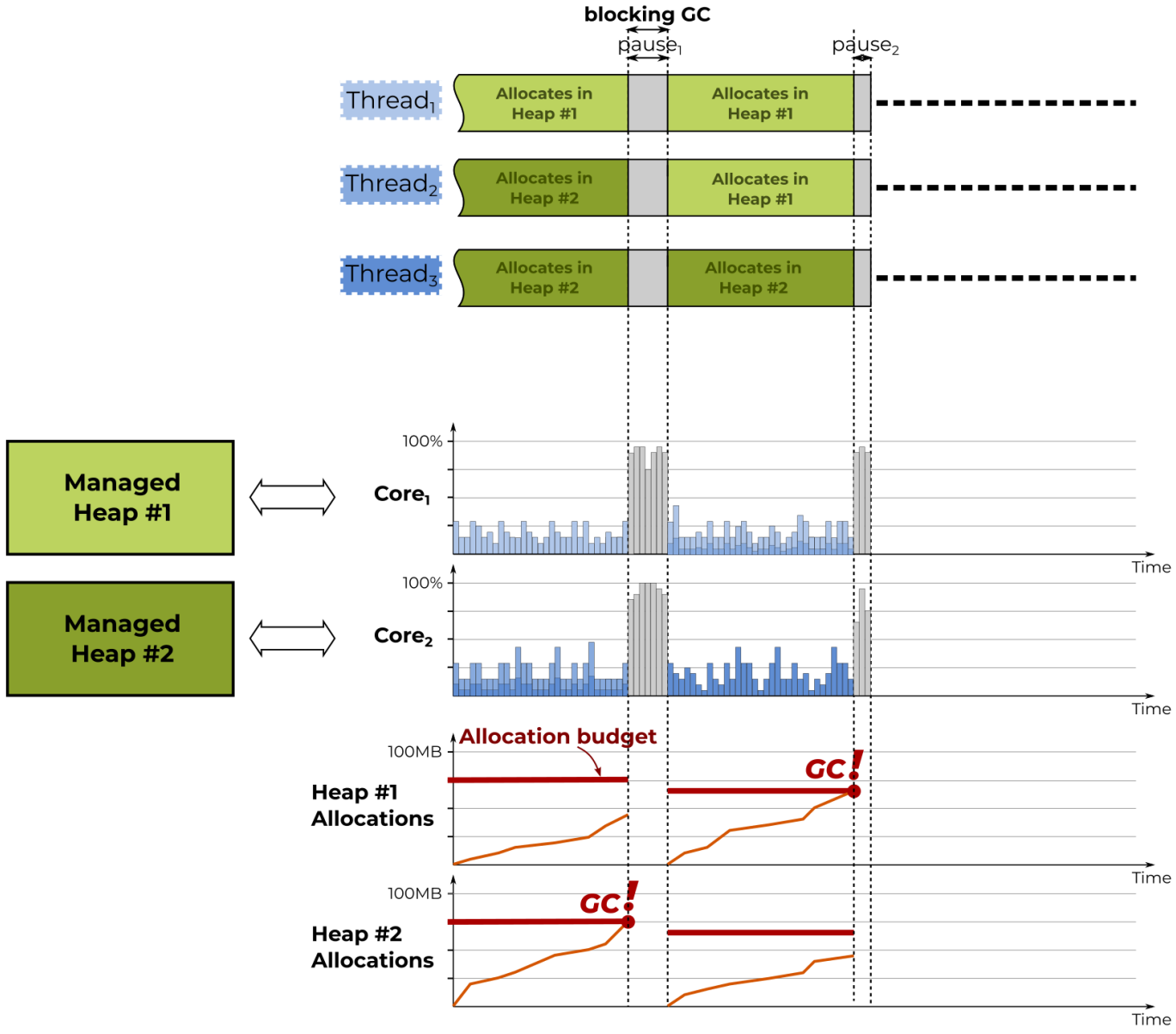

Thread1
Allocates in Heap #1
Thread2
Allocates in Heap #2
Thread3
Allocates in Heap #2

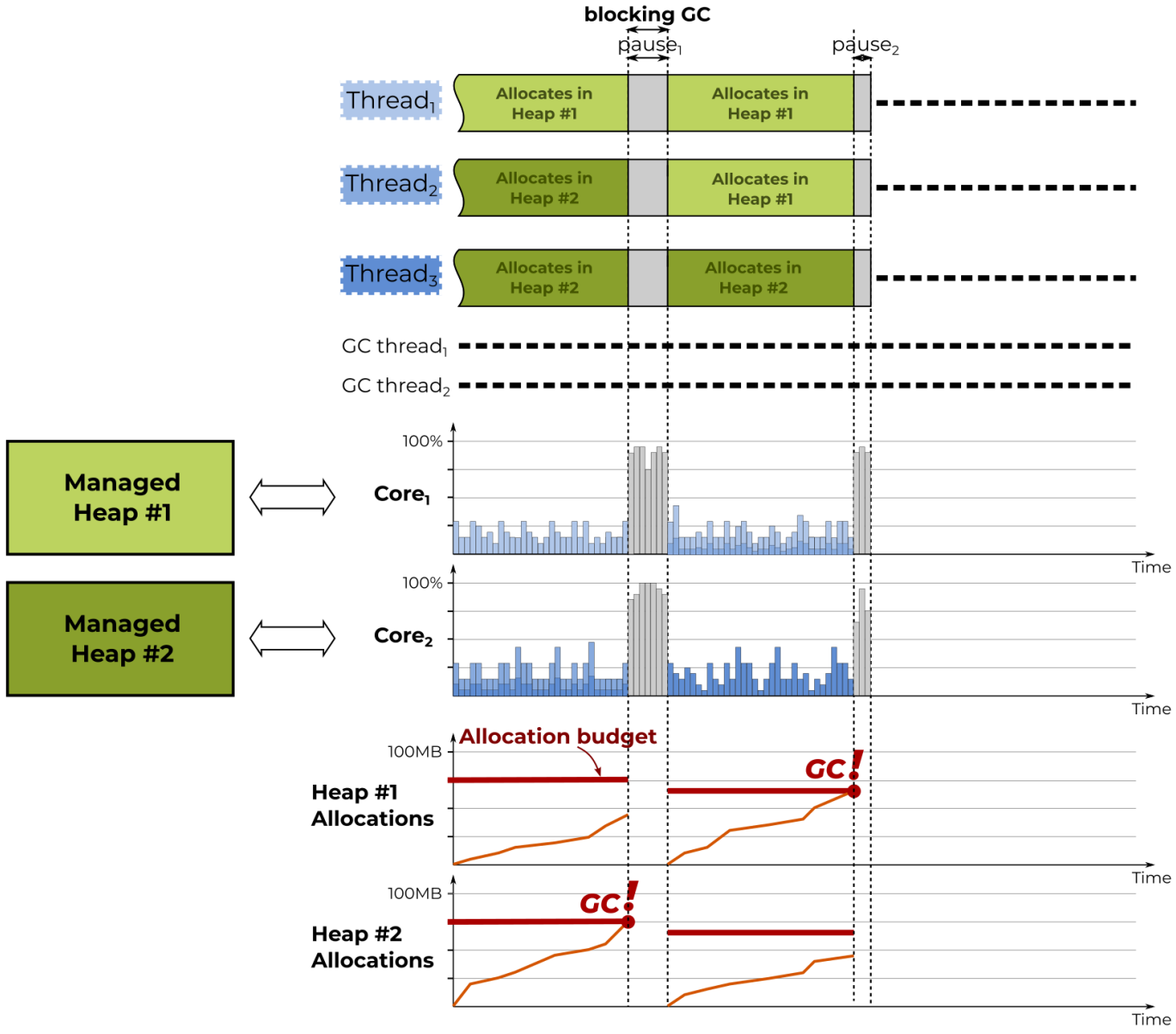

Managed Heap #1
Managed Heap #2
100%
Core1
Time
100%
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Time
100MB
GC!
Heap #2 Allocations
Time

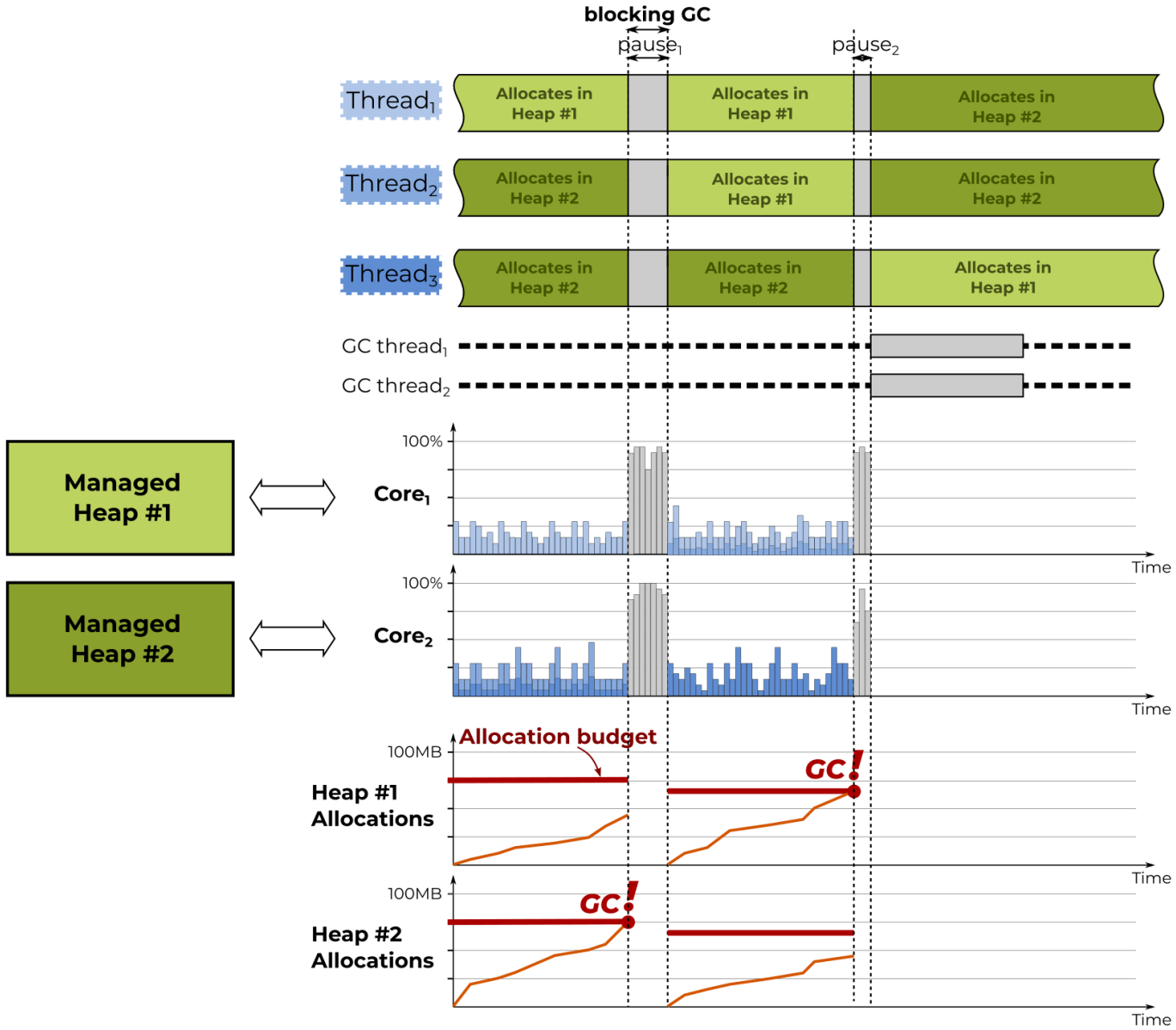

blocking GC
pause1
Thread1
Allocates in Heap #1
Thread2
Allocates in Heap #2
Thread3
Allocates in Heap #2
Managed Heap #1
Managed Heap #2
100%
Core1
Time
100%
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Time
100MB
GC!
Heap #2 Allocations
Time

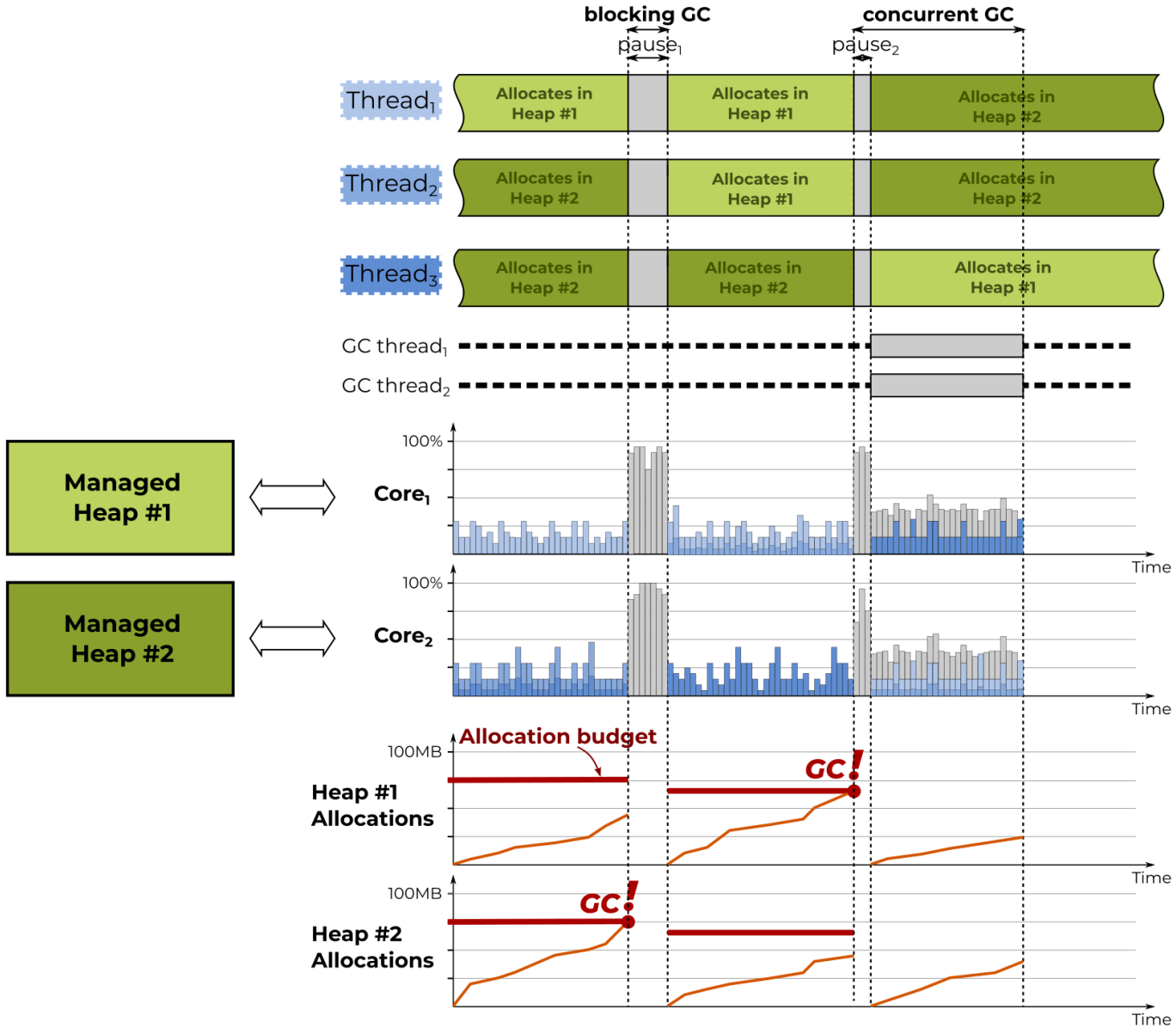

blocking GC
pause1
Thread1
Thread2
Thread3
Allocates in Heap #1
Allocates in Heap #2
Managed Heap #1
Managed Heap #2
100%
Core1
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Heap #2 Allocations
GC!

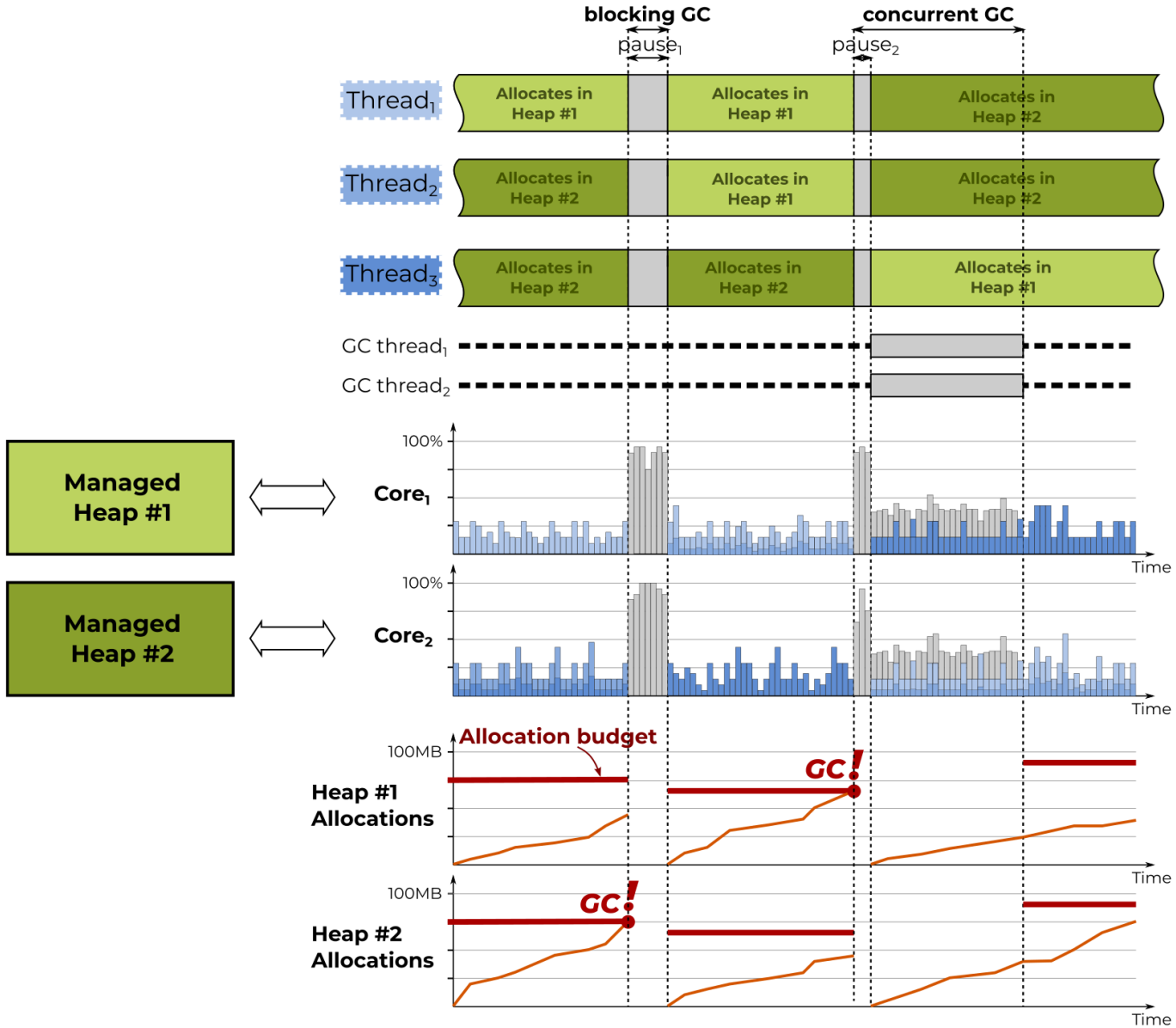

blocking GC
pause1
Thread1
Thread2
Thread3
Allocates in Heap #1
Allocates in Heap #2
Managed Heap #1
Managed Heap #2
100%
Core1
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Heap #2 Allocations
GC!


blocking GC
pause1
pause2
Thread1
Thread2
Thread3
Allocates in Heap #1
Allocates in Heap #2
Managed Heap #1
Managed Heap #2
100%
Core1
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Heap #2 Allocations
GC!

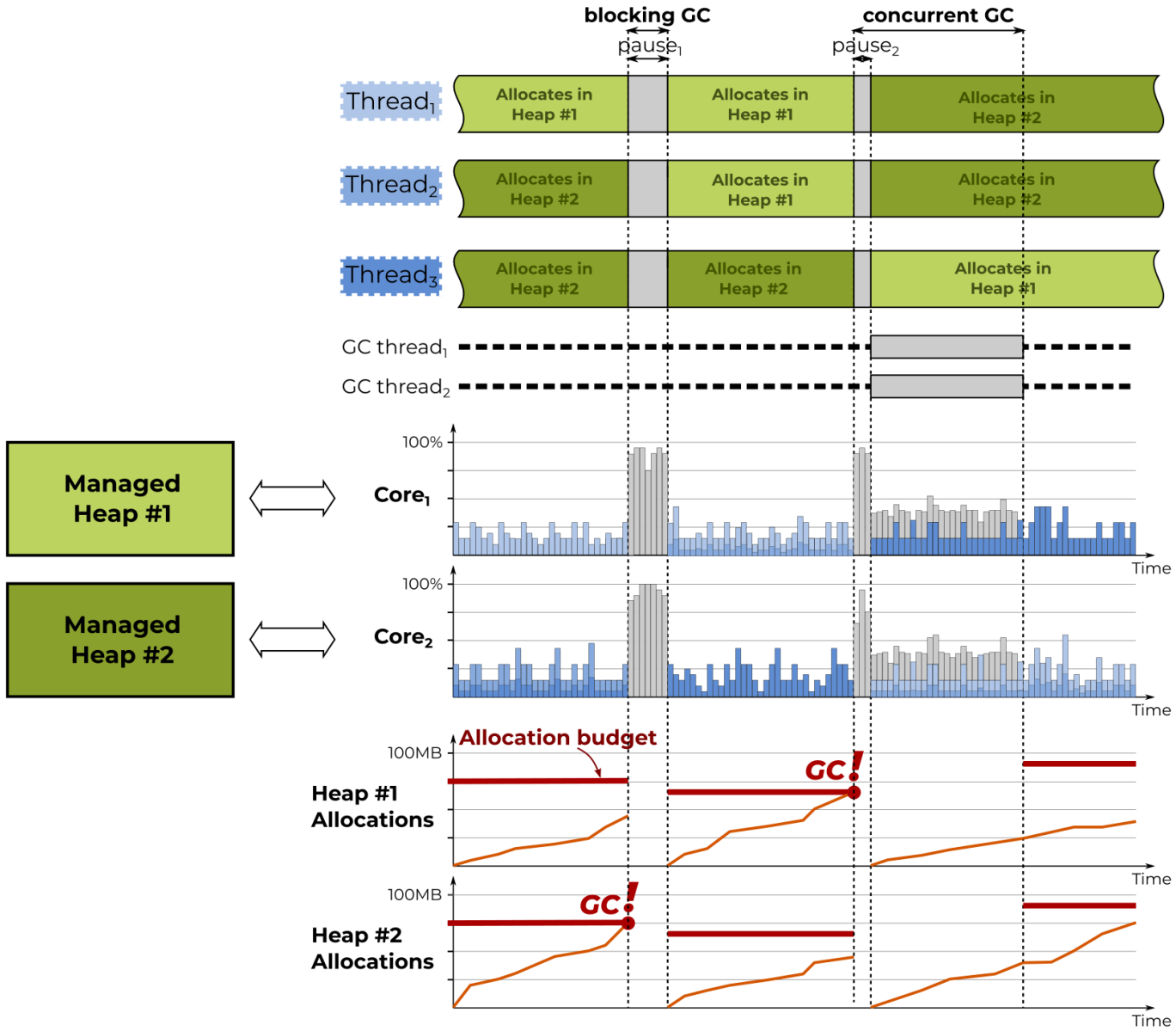

blocking GC
pause1
pause2
Thread1
Thread2
Thread3
Allocates in Heap #1
Allocates in Heap #2
GC thread1
GC thread2
Managed Heap #1
Managed Heap #2
100%
Core1
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Heap #2 Allocations
GC!

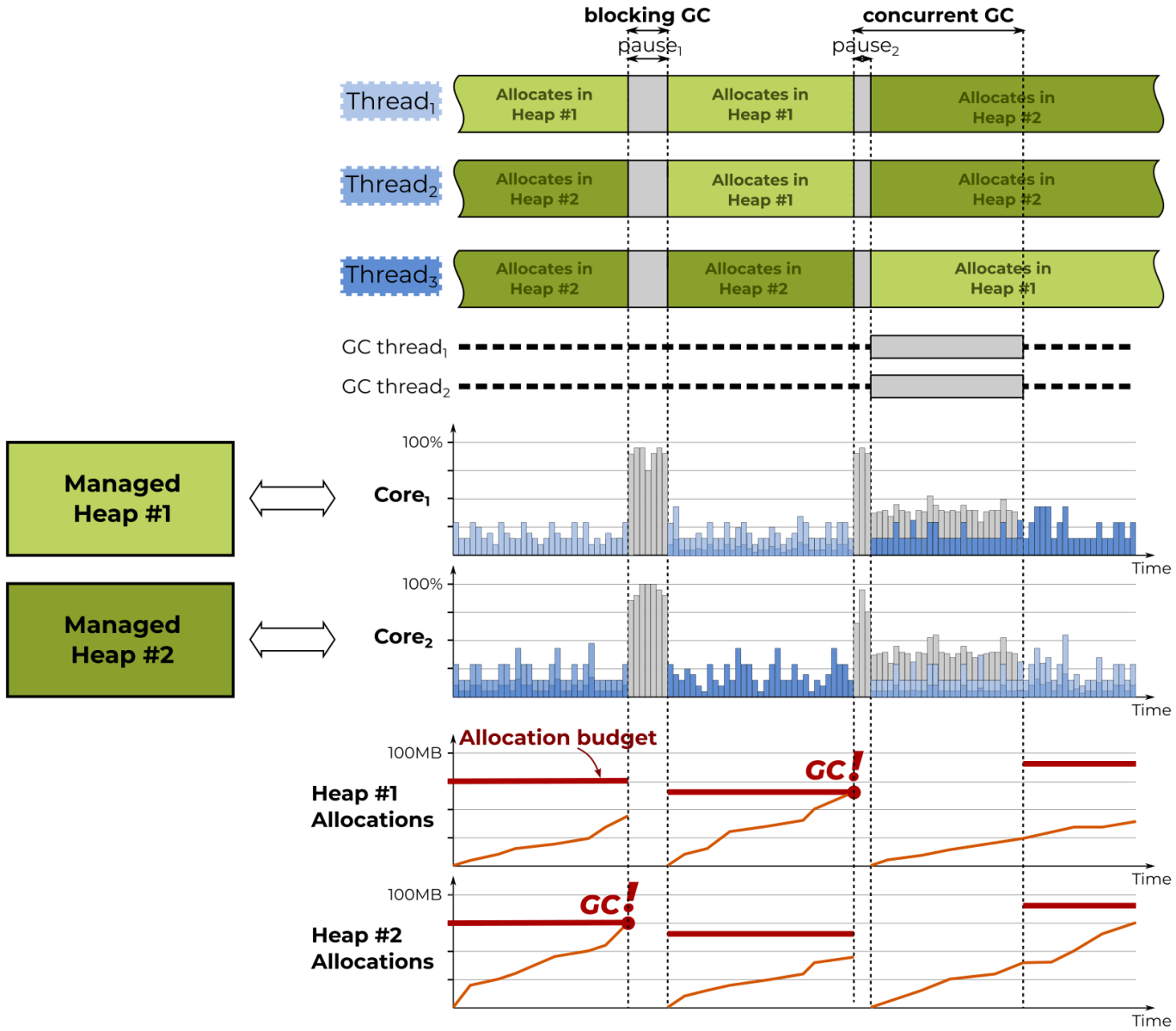

blocking GC
pause1
pause2
Thread1
Thread2
Thread3
Allocates in Heap #1
Allocates in Heap #2
GC thread1
GC thread2
Managed Heap #1
Managed Heap #2
100%
Core1
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Heap #2 Allocations
GC!


blocking GC
concurrent GC
pause1
pause2
Thread1
Thread2
Thread3
Allocates in Heap #1
Allocates in Heap #2
GC thread1
GC thread2
Managed Heap #1
Managed Heap #2
100%
Core1
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Heap #2 Allocations
GC!


blocking GC
concurrent GC
pause1
pause2
Thread1
Thread2
Thread3
Allocates in Heap #1
Allocates in Heap #2
GC thread1
GC thread2
Managed Heap #1
Managed Heap #2
100%
Core1
Core2
Time
Allocation budget
100MB
Heap #1 Allocations
Heap #2 Allocations
GC!

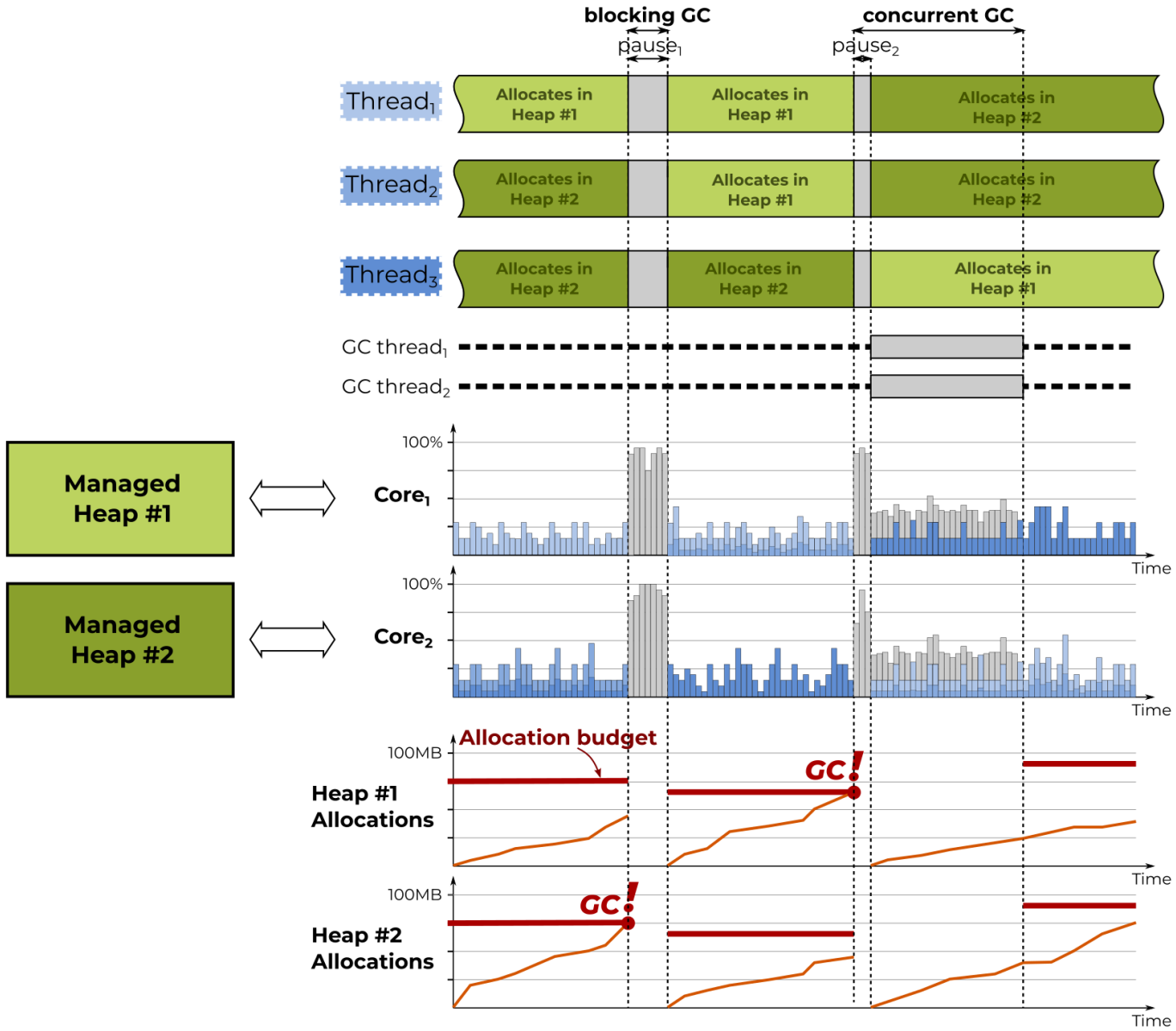
Simplifications:





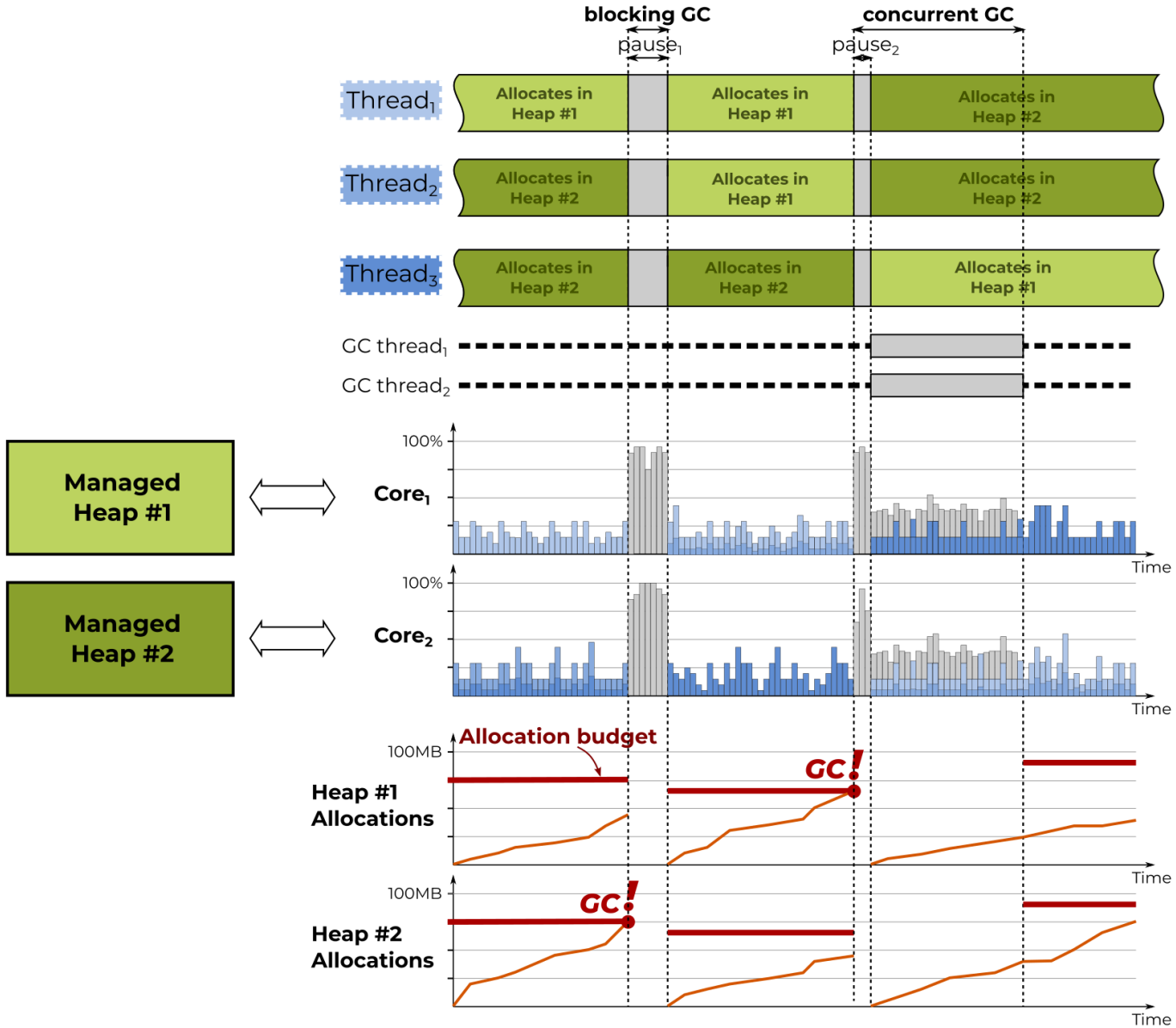
Simplifications:

- *thread-to-heap/core* assignment may change at any time



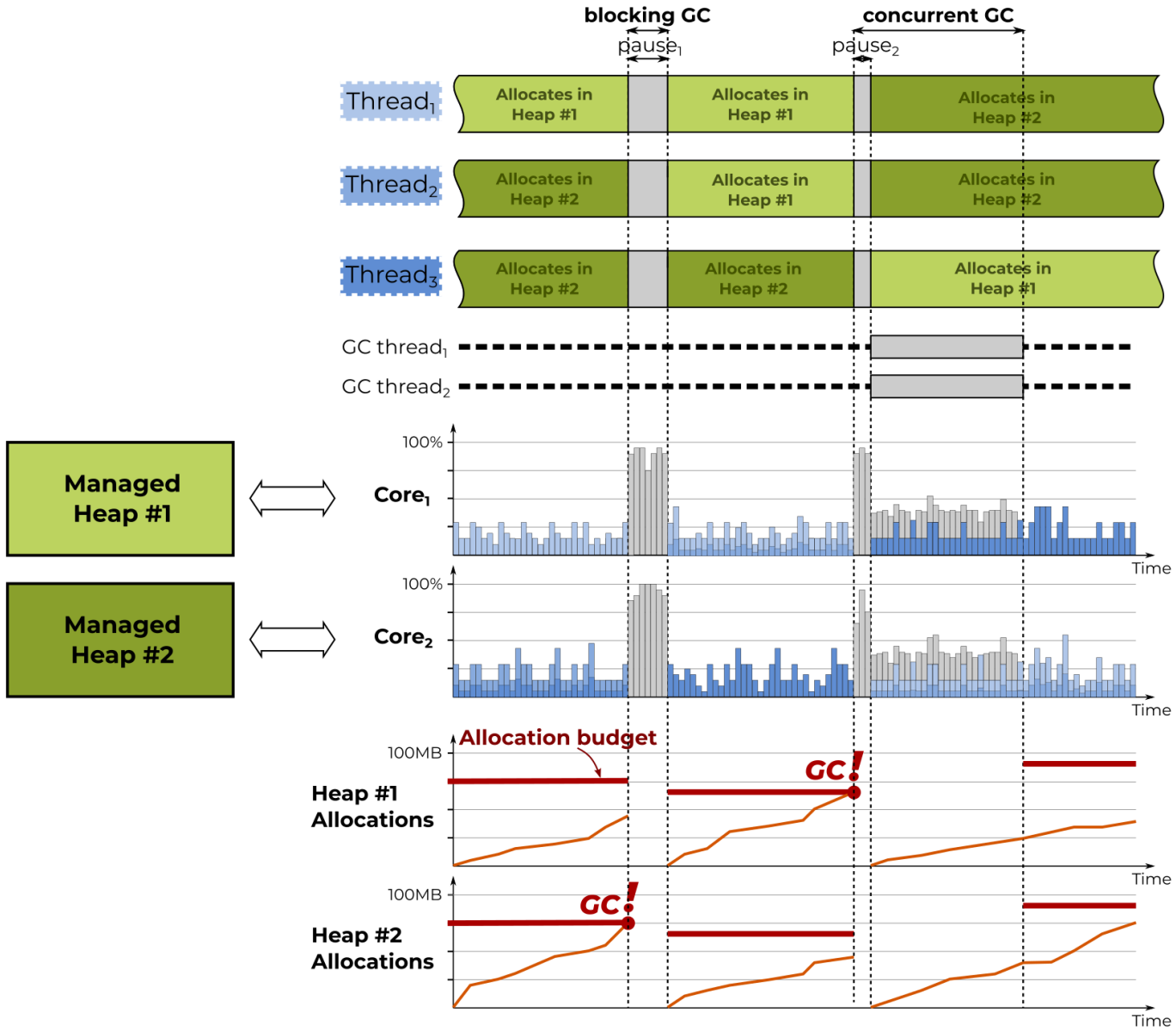
Simplifications:

- *thread-to-heap/core* assignment may change at any time
- *allocation budget* is generational



Simplifications:

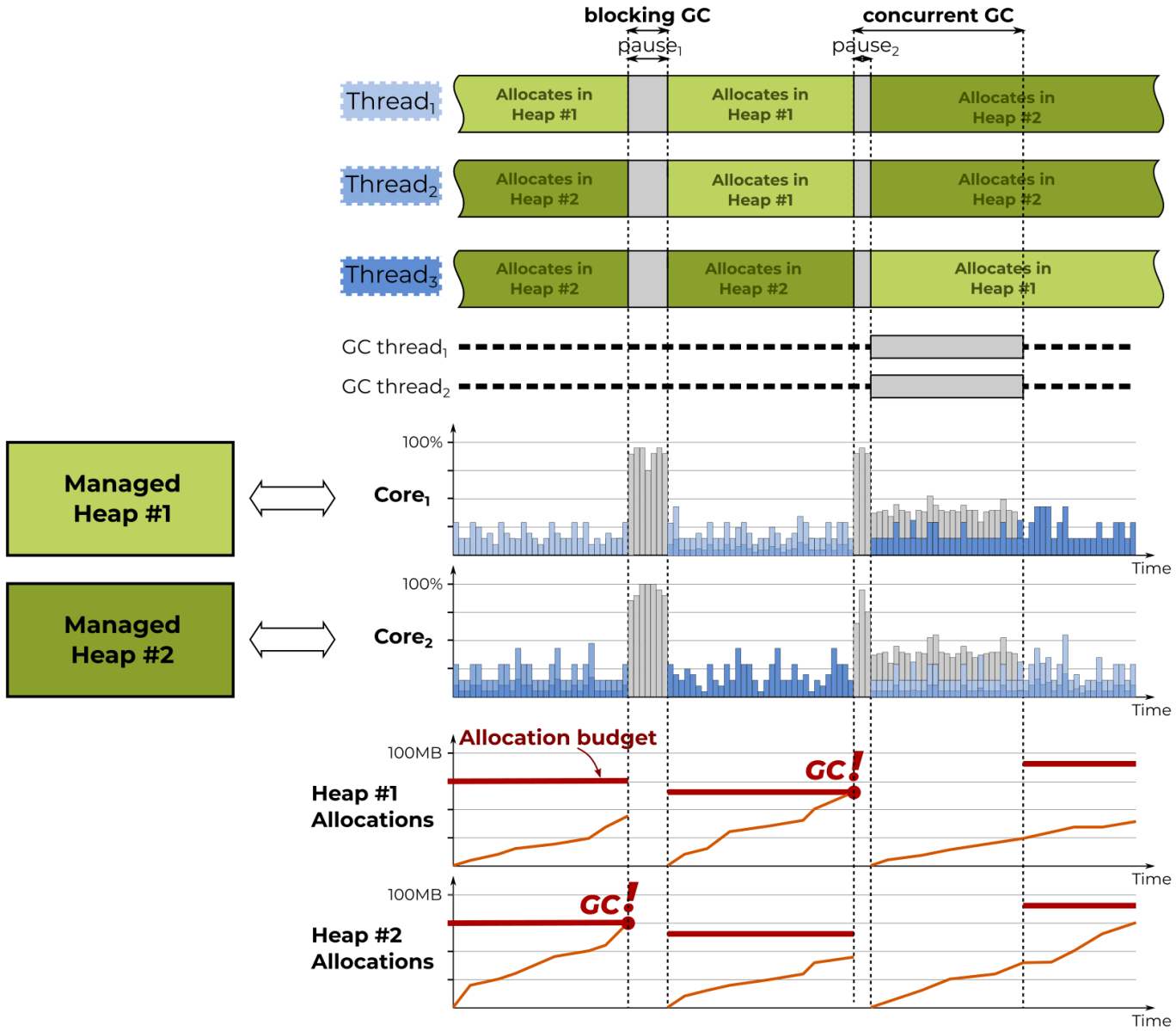
- *thread-to-heap/core* assignment may change at any time
- *allocation budget* is generational
- the very first trigger for even considering to start GC is running out of so-called *allocation context* (not covered yet)



Simplifications:

- *thread-to-heap/core* assignment may change at any time
- *allocation budget* is generational
- the very first trigger for even considering to start GC is running out of so-called *allocation context* (not covered yet)

Heaps are typically *balanced* "by nature" - e.g. in web app thread pool has rather uniformly delegated work.

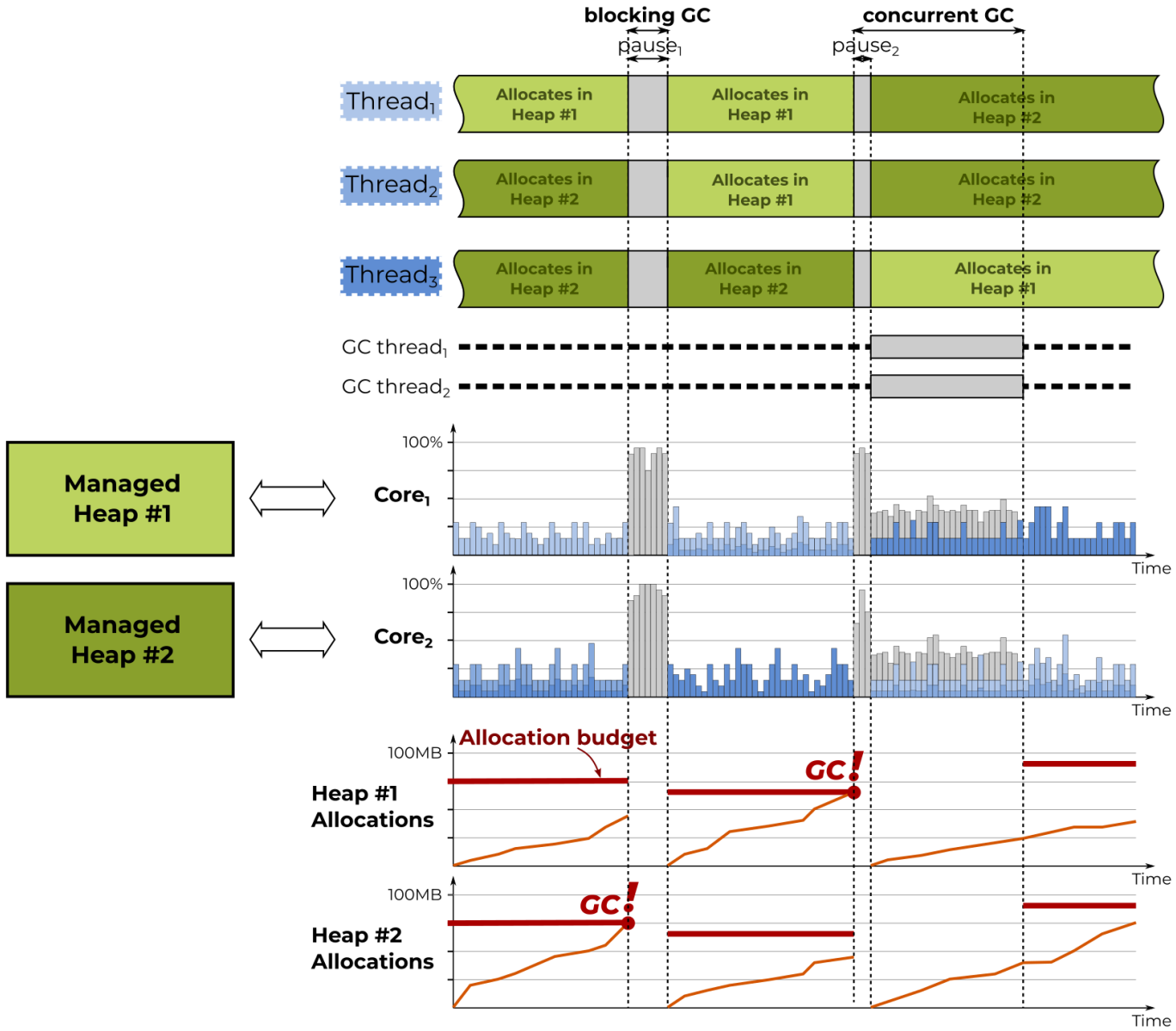


Simplifications:

- *thread-to-heap/core* assignment may change at any time
- *allocation budget* is generational
- the very first trigger for even considering to start GC is running out of so-called *allocation context* (not covered yet)

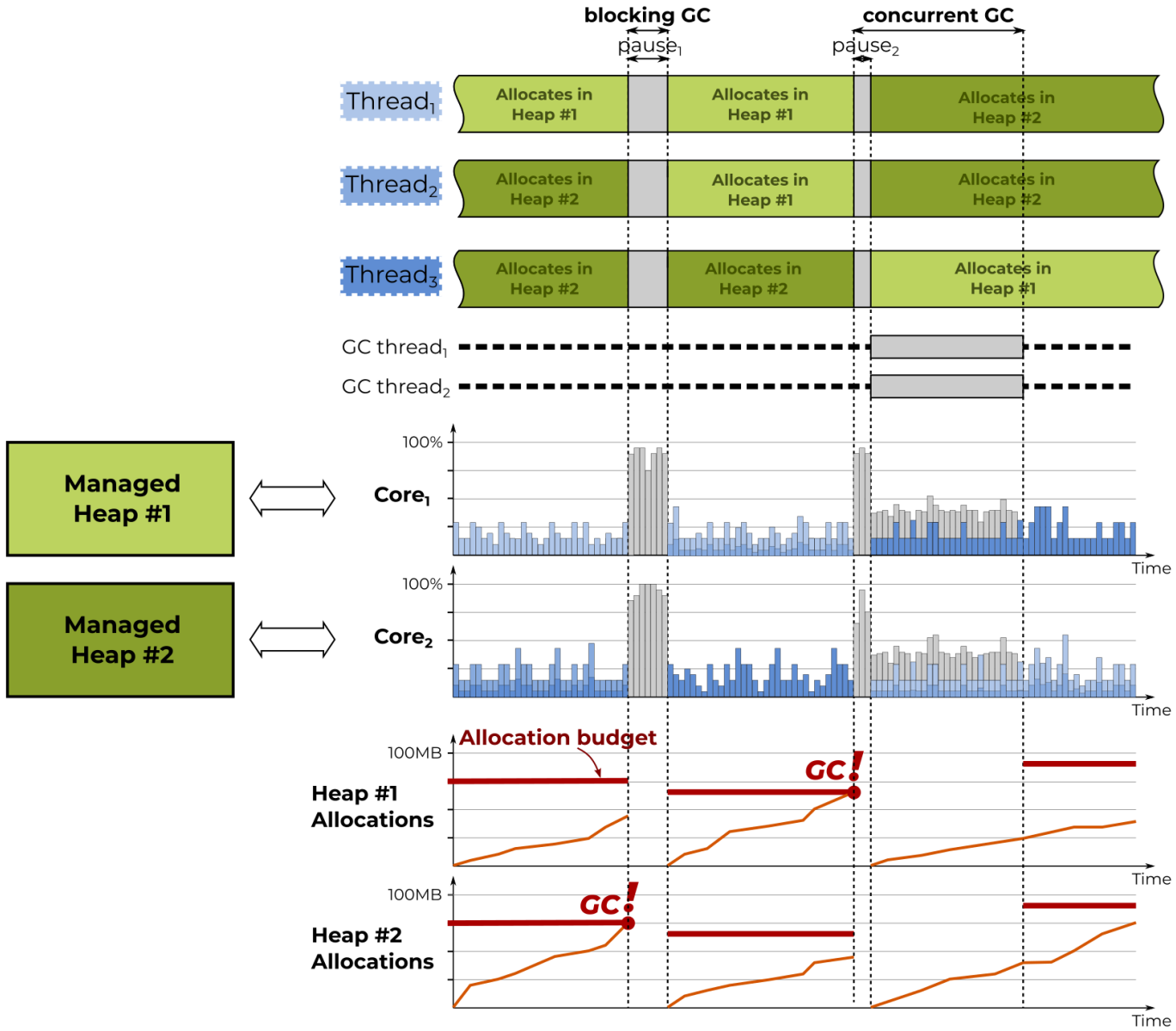
Heaps are typically *balanced* "by nature" - e.g. in web app thread pool has rather uniformly delegated work.

In case of the Workstation GC there is single Managed Heap but all the rest applies.



Maximum "allocation budget" (64-bit)

	Non-Concurrent	Concurrent
Workstation	128MB	6MB
Server	200MB	200MB



Maximum "allocation budget" (64-bit)

	Non-Concurrent	Concurrent
Workstation	128MB	6MB
Server	200MB	200MB

See how more aggressive is the Concurrent/Workstation GC comparing to the Server GC!

GC modes

So, in the end we have four main possibilities (with some defaults):

	Non-Concurrent	**Concurrent**
Workstation	Non-Concurrent Workstation	**Background Workstation**
Server	Non-Concurrent Server	**Background Server**

GC modes - project configuration

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <ServerGarbageCollection>true</ServerGarbageCollection>
    <ConcurrentGarbageCollection>true</ConcurrentGarbageCollection>
  </PropertyGroup>
</Project>
```

GC modes - runtime configuration

.NET Framework:

`[appName].exe.config`/`Web.config` file:

```
<configuration>
   ...
   <runtime>
      <gcServer enabled="true"/>
      <gcConcurrent enabled="true"/>
   </runtime>
</configuration>
```

.NET Core:

`SomeApplication.runtimeconfig.json` file:

```
{
   "runtimeOptions": {
      ...
      "configProperties": {
         "System.GC.Server": true,
         "System.GC.Concurrent": true
      }
   }
}
```

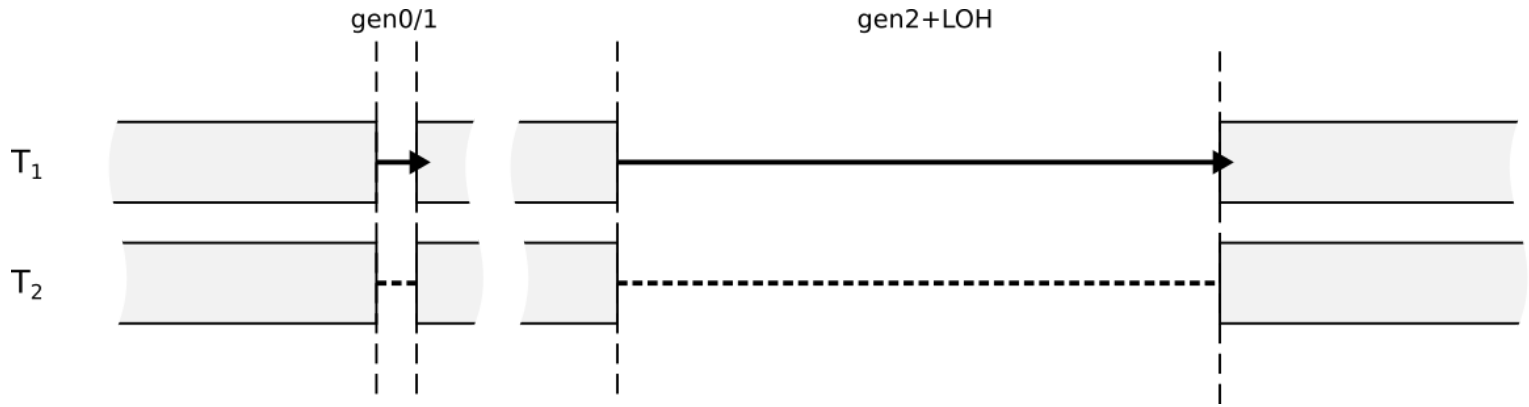
GC modes - runtime configuration

Environment variables:

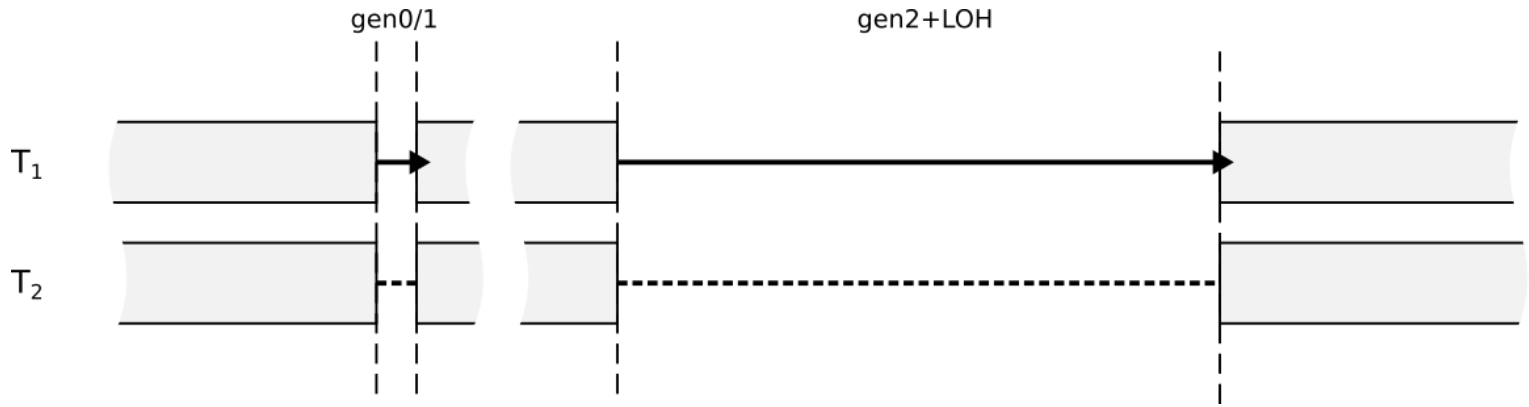
- `COMPlus_gcServer=0|1`
- `COMPlus_gcConcurrent=0|1`

  `DOTNET_gcServer` and `DOTNET_gcConcurrent` since .NET 6

GC mode - Workstation Non-concurrent

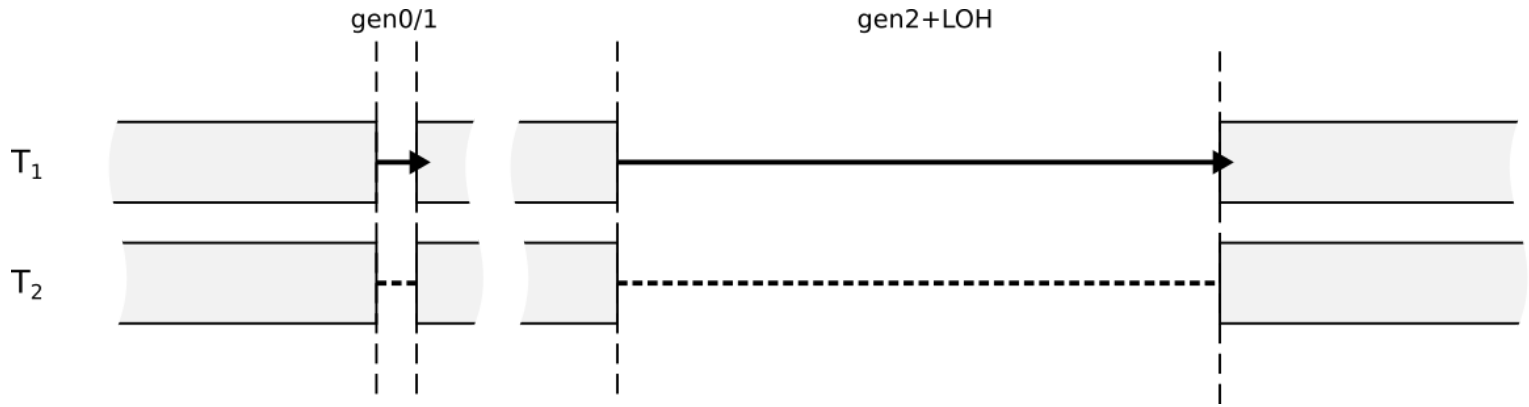


GC mode - Workstation Non-concurrent



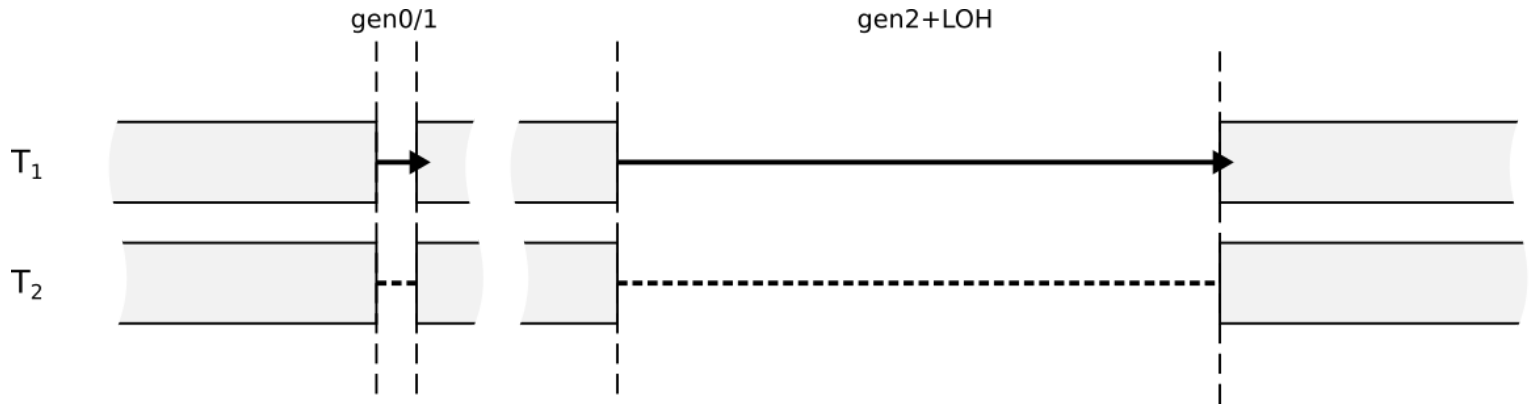
- all GCs are blocking - they typically *sweep* but occasionally *compact*

GC mode - Workstation Non-concurrent



- all GCs are blocking - they typically *sweep* but occasionally *compact*
- no additional GC threads - one of the regular threads start to do the GC work

GC mode - Workstation Non-concurrent

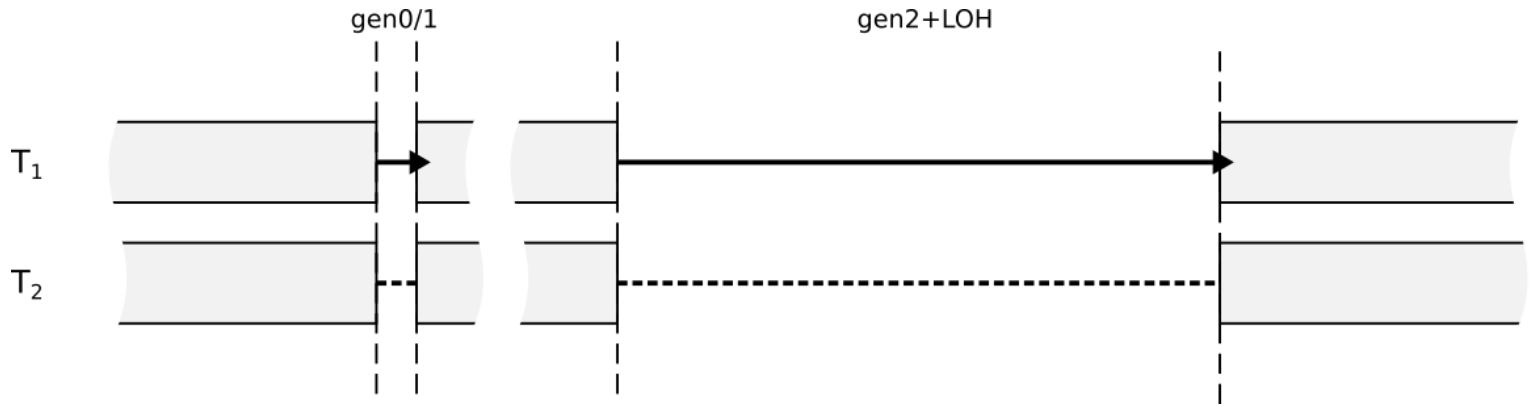


- all GCs are blocking - they typically *sweep* but occasionally *compact*
- no additional GC threads - one of the regular threads start to do the GC work

Usage:

- Never...?

GC mode - Workstation Non-concurrent

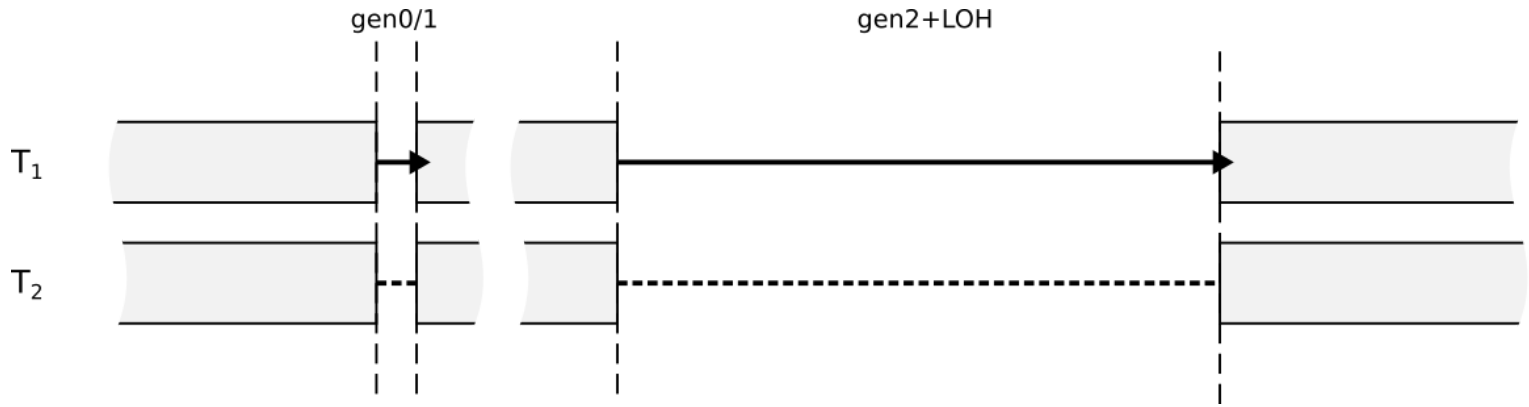


- all GCs are blocking - they typically *sweep* but occasionally *compact*
- no additional GC threads - one of the regular threads start to do the GC work

Usage:

- Never...?
- A highly saturated environment

GC mode - Workstation Non-concurrent

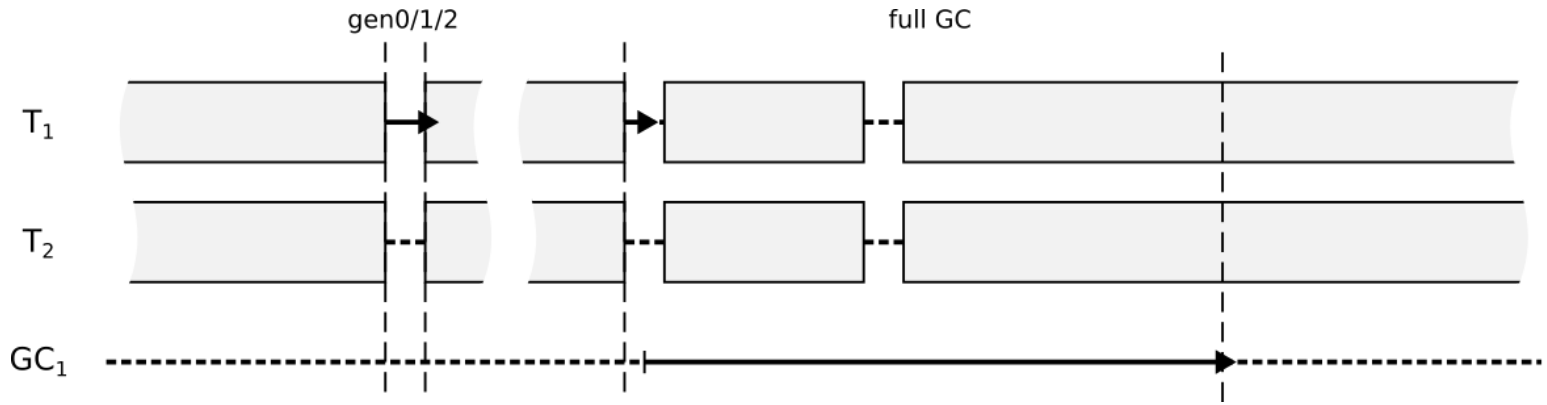


- all GCs are blocking - they typically *sweep* but occasionally *compact*
- no additional GC threads - one of the regular threads start to do the GC work

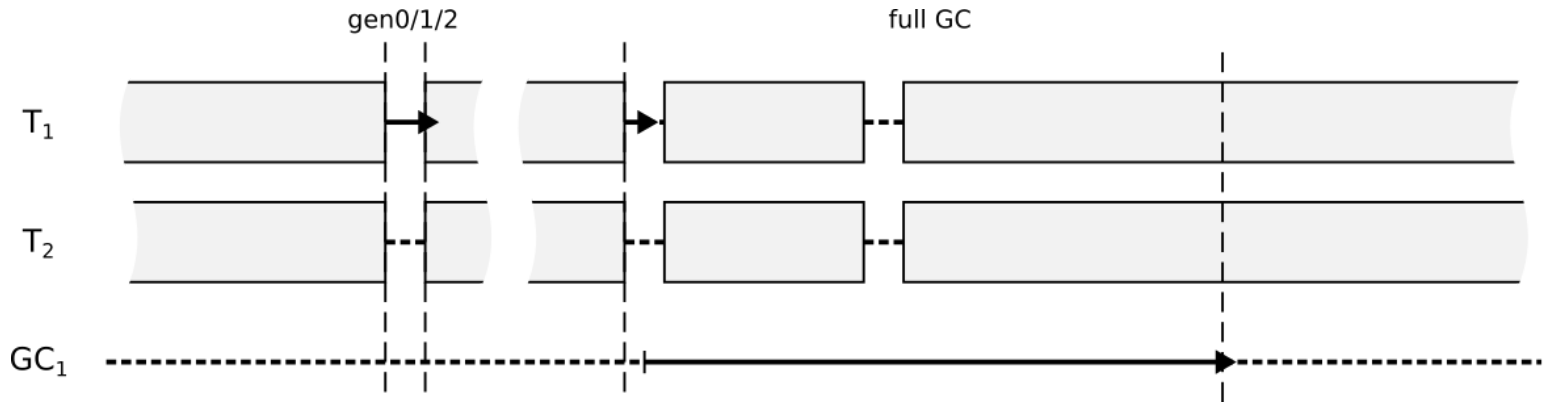
Usage:

- Never...?
- A highly saturated environment
- Environment with many lightweight web applications (like "dockerized" microservices)

GC mode - (legacy) Workstation Concurrent

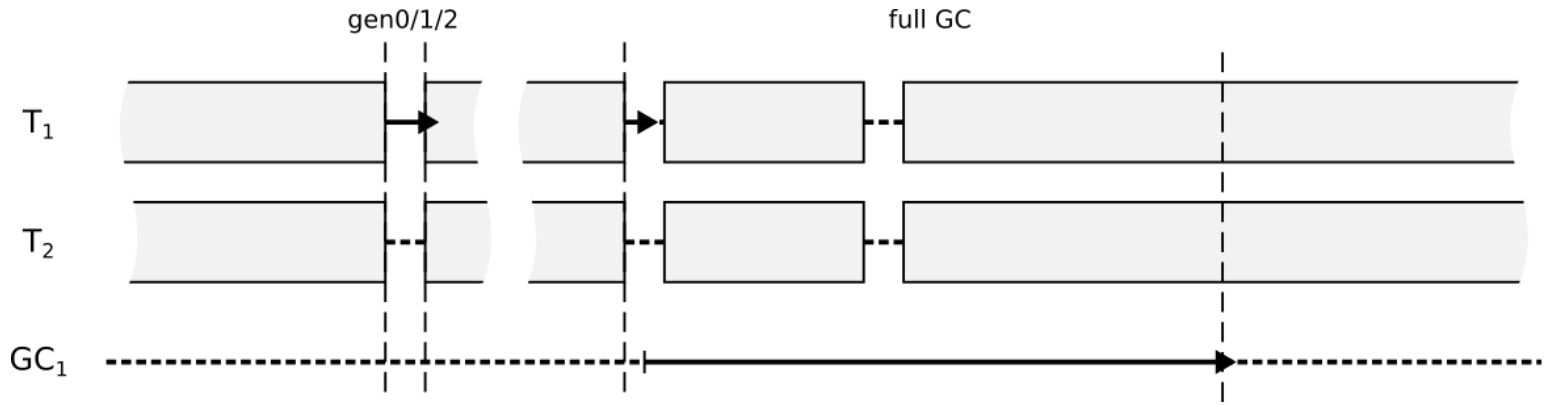


GC mode - (legacy) Workstation Concurrent



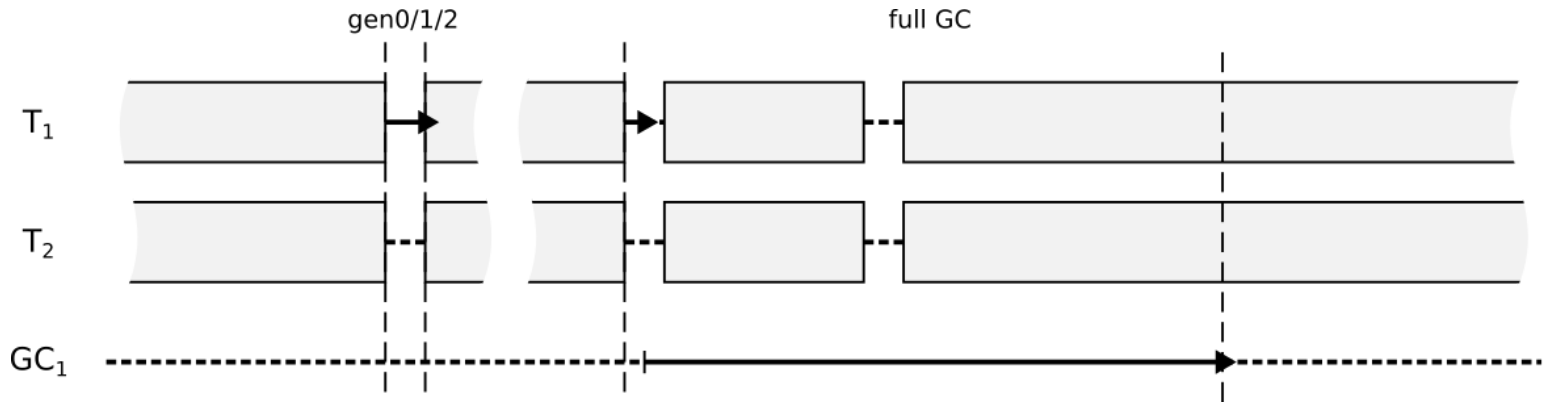
- *young* GCs are always blocking - they are fast, no need to make it complex

GC mode - (legacy) Workstation Concurrent



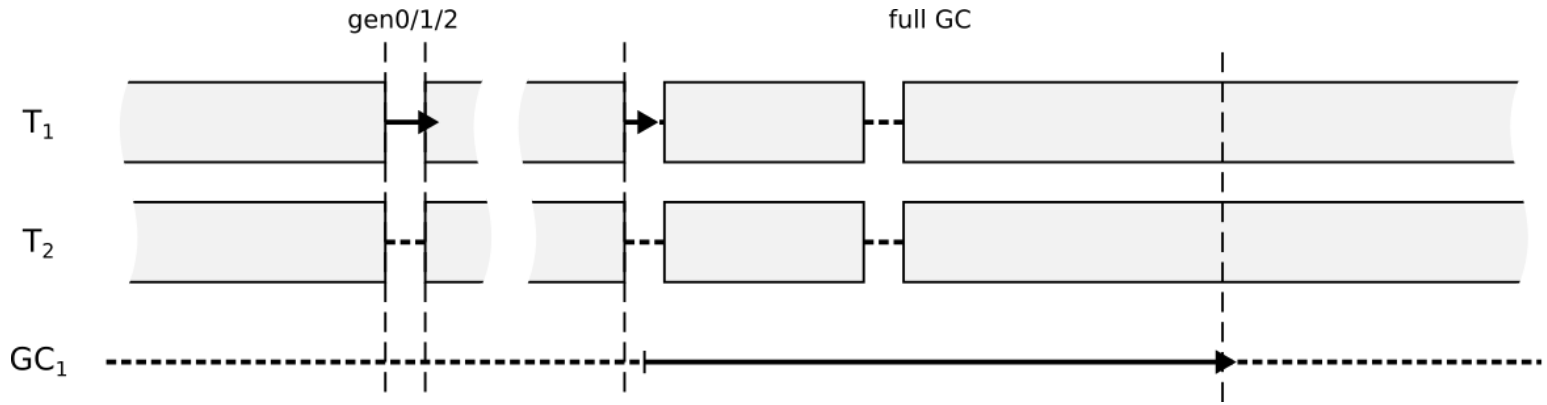
- *young* GCs are always blocking - they are fast, no need to make it complex
- most *Full GCs* are concurrent - so they only *sweep*

GC mode - (legacy) Workstation Concurrent



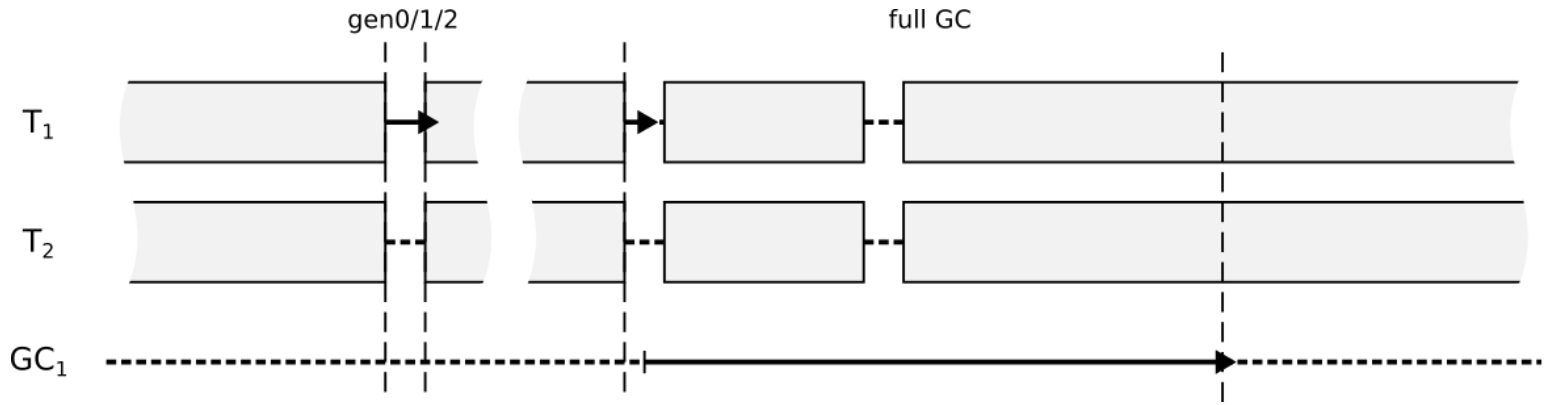
- *young* GCs are always blocking - they are fast, no need to make it complex
- most *Full GCs* are concurrent - so they only *sweep*
- they may be occasional blocking *Full GCs* - because they can *compact*

GC mode - (legacy) Workstation Concurrent



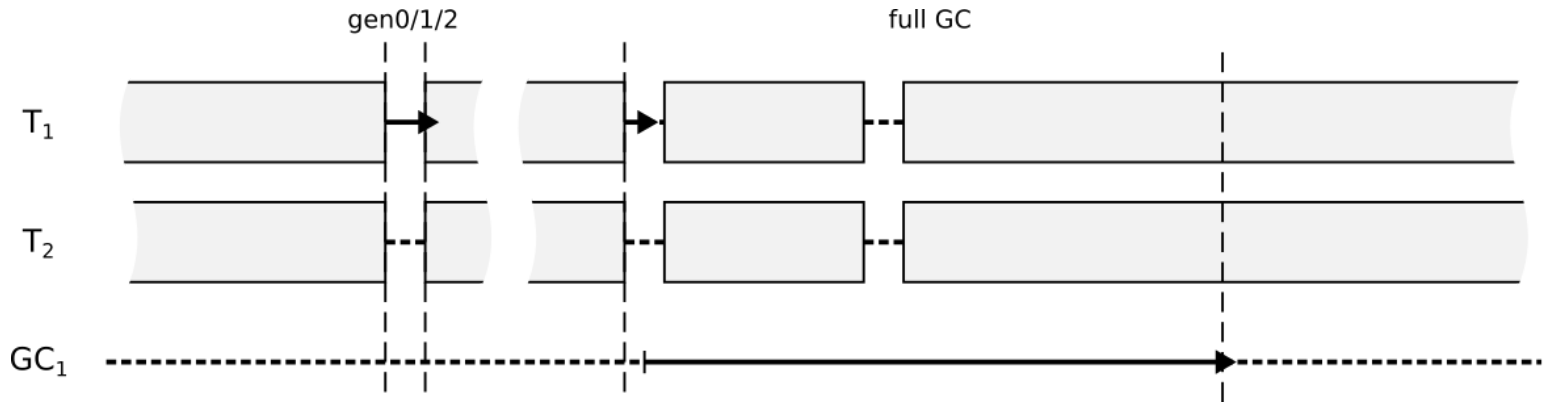
- *young* GCs are always blocking - they are fast, no need to make it complex
- most *Full GCs* are concurrent - so they only *sweep*
- they may be occasional blocking *Full GCs* - because they can *compact*
- there are two short pauses during the concurrent GC

GC mode - (legacy) Workstation Concurrent



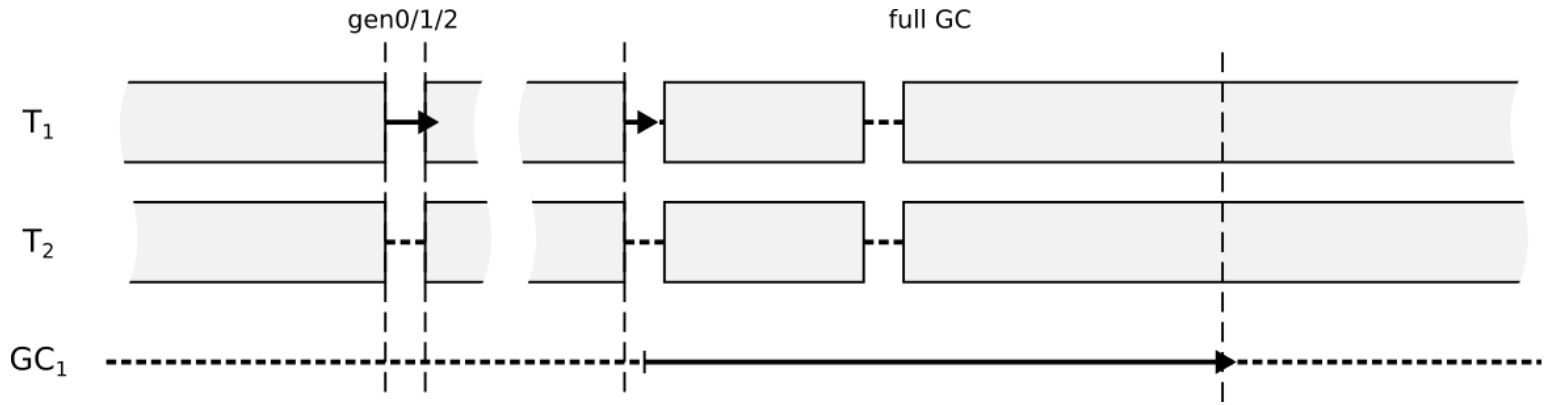
- *young* GCs are always blocking - they are fast, no need to make it complex
- most *Full GCs* are concurrent - so they only *sweep*
- they may be occasional blocking *Full GCs* - because they can *compact*
- there are two short pauses during the concurrent GC
- while the concurrent GC is running, threads allocate

GC mode - (legacy) Workstation Concurrent



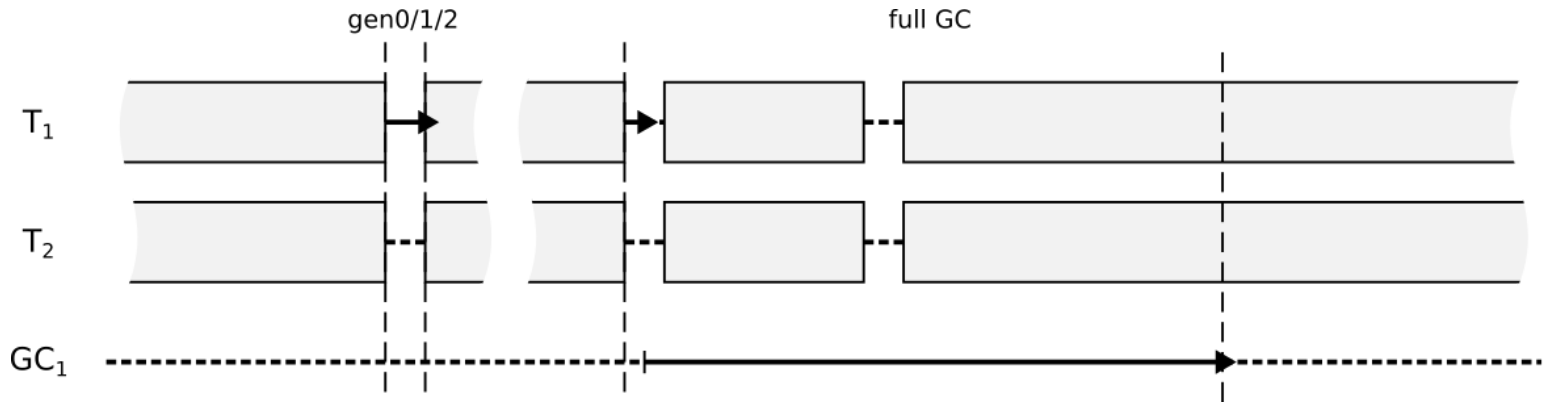
- *young* GCs are always blocking - they are fast, no need to make it complex
- most *Full GCs* are concurrent - so they only *sweep*
- they may be occasional blocking *Full GCs* - because they can *compact*
- there are two short pauses during the concurrent GC
- while the concurrent GC is running, threads allocate
  - what if we hit *allocation budget* limit?

GC mode - (legacy) Workstation Concurrent



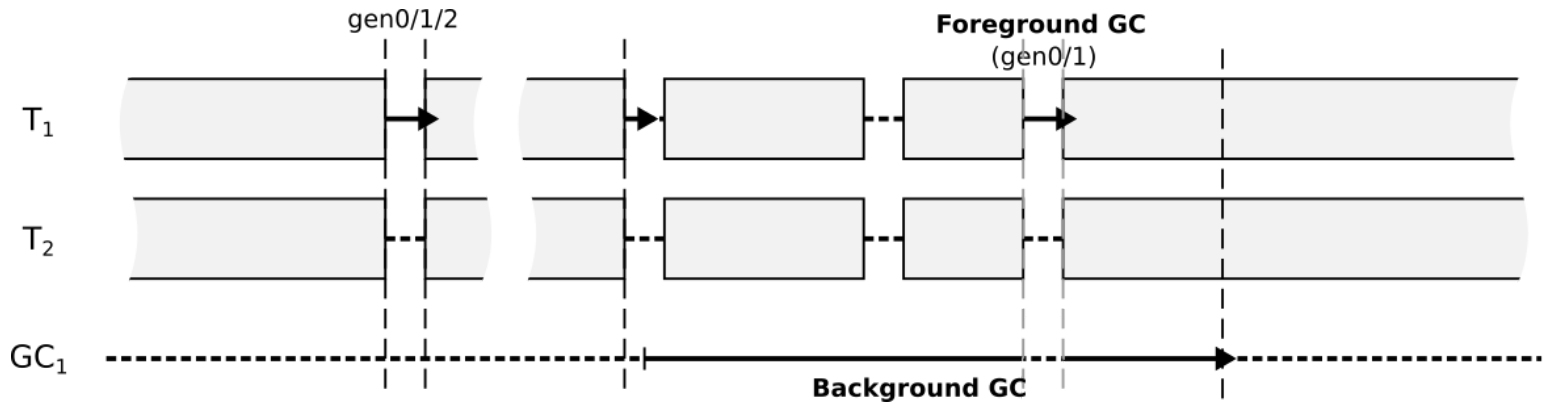
- *young* GCs are always blocking - they are fast, no need to make it complex
- most *Full GCs* are concurrent - so they only *sweep*
- they may be occasional blocking *Full GCs* - because they can *compact*
- there are two short pauses during the concurrent GC
- while the concurrent GC is running, threads allocate
  - what if we hit *allocation budget* limit?
  - we need to wait and trigger the GC.

GC mode - (legacy) Workstation Concurrent

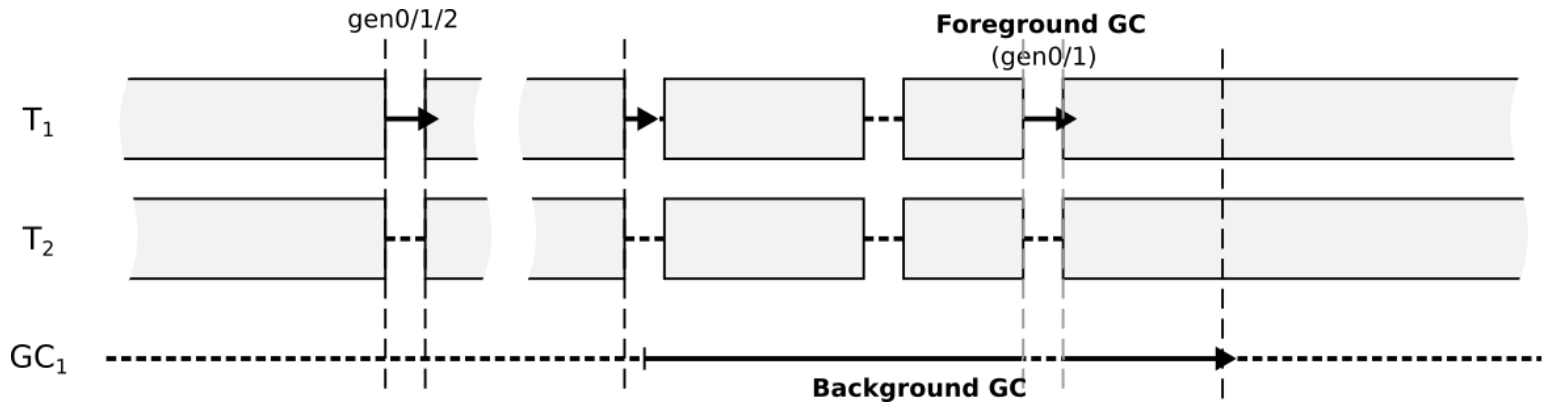


- *young* GCs are always blocking - they are fast, no need to make it complex
- most *Full GCs* are concurrent - so they only *sweep*
- they may be occasional blocking *Full GCs* - because they can *compact*
- there are two short pauses during the concurrent GC
- while the concurrent GC is running, threads allocate
  - what if we hit *allocation budget* limit?
  - we need to wait and trigger the GC.
- it was a default mode in Desktop apps up to .NET Framework 4.0 (pretty old, you must admit😇)

GC mode - Background Workstation

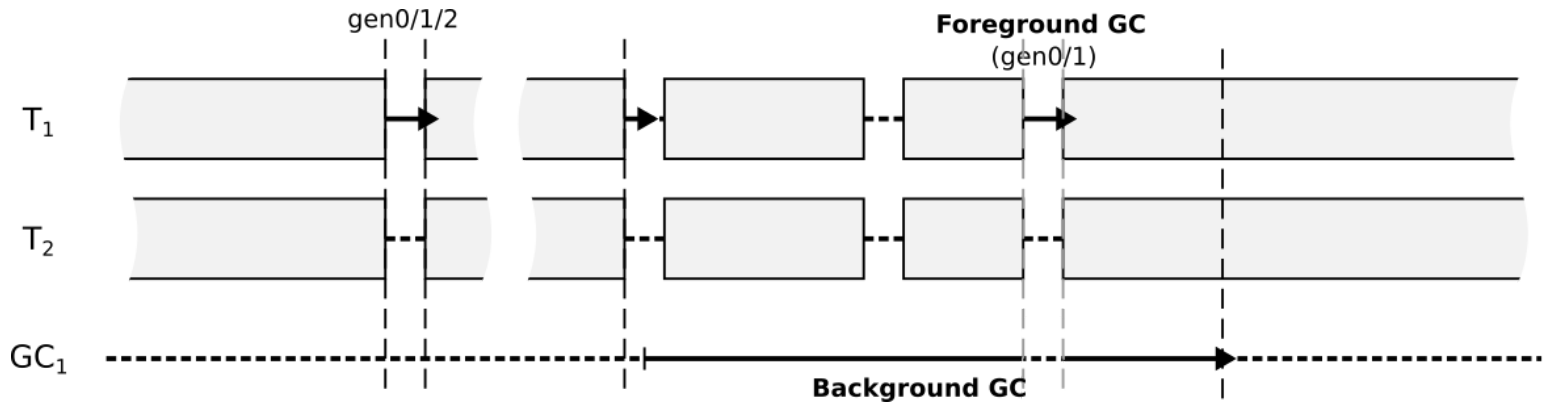


GC mode - Background Workstation



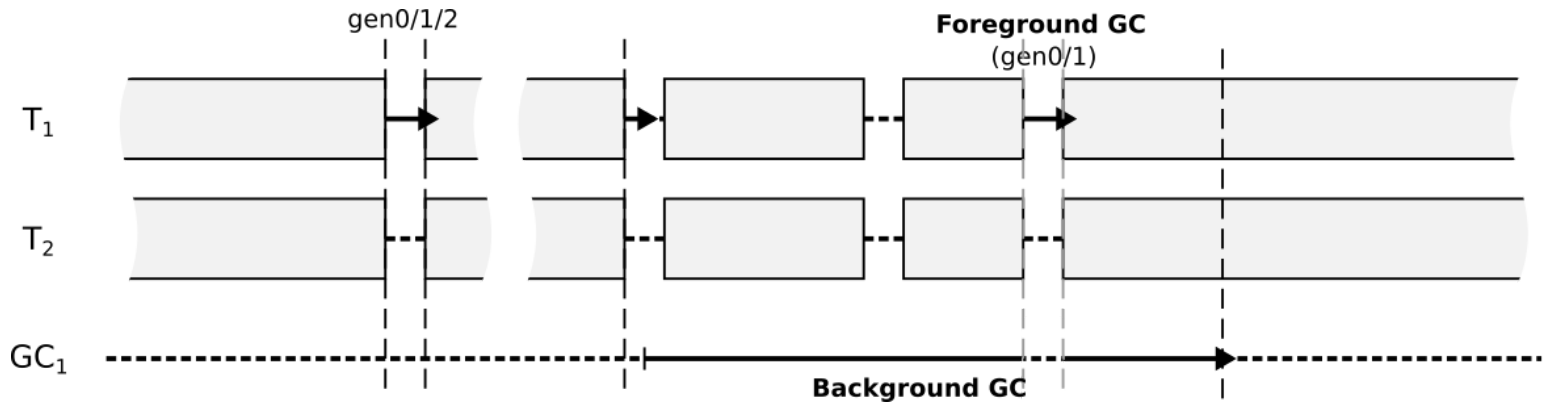
- "concurrent GC" renamed to **Background GC**

GC mode - Background Workstation



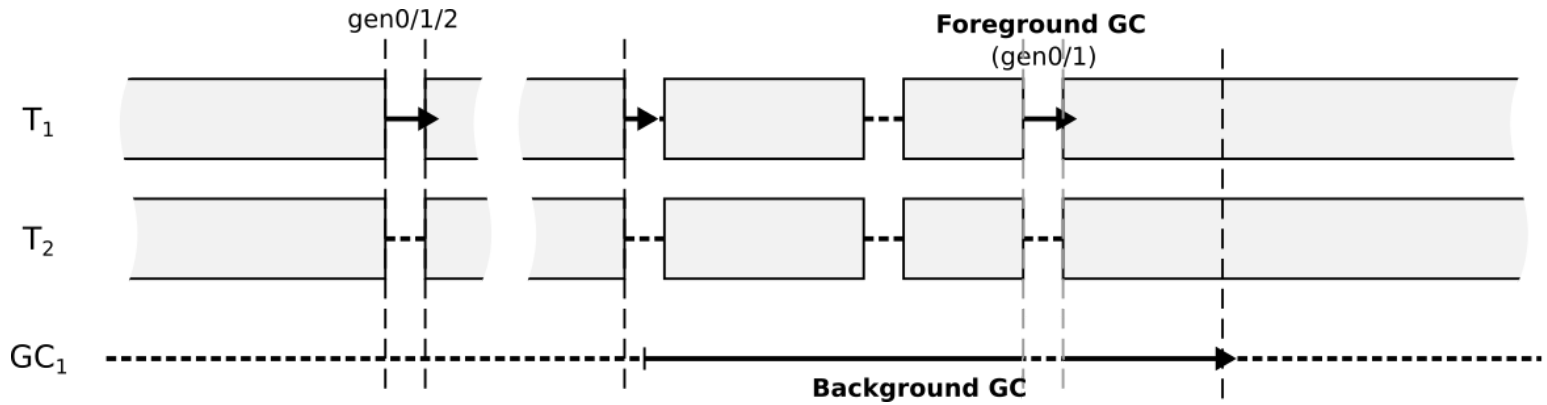
- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)

GC mode - Background Workstation



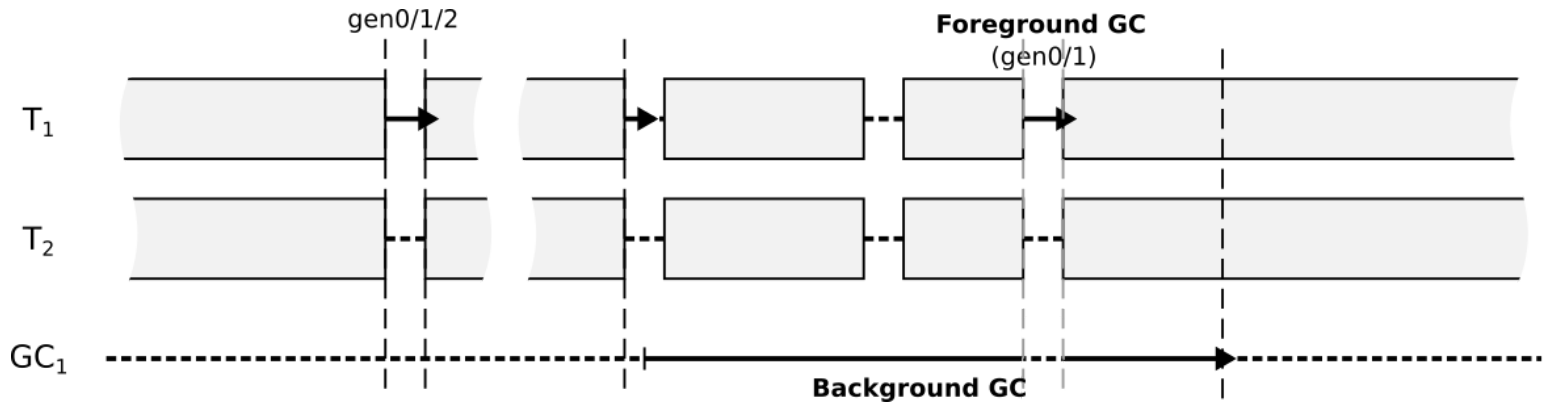
- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)
- most GCs are still concurrent

GC mode - Background Workstation



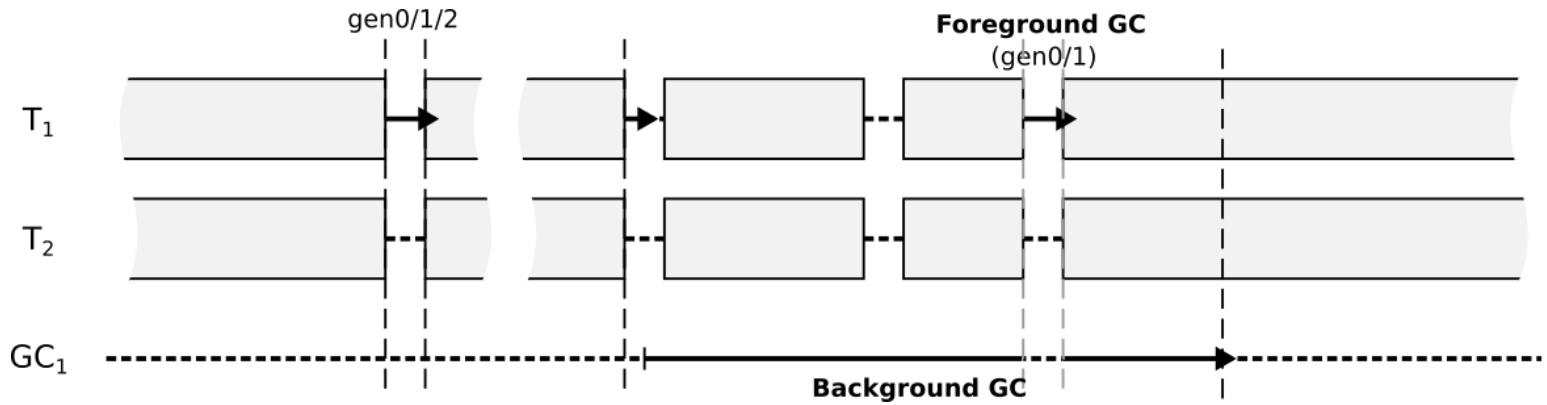
- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)
- most GCs are still concurrent
  - blocking full-GCs are very rare

GC mode - Background Workstation



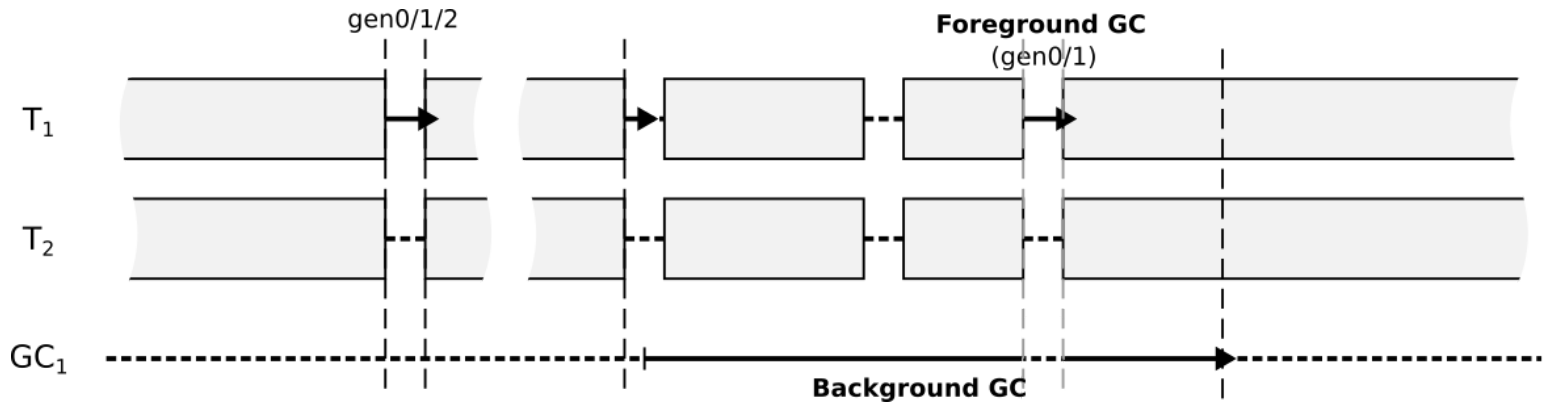
- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)
- most GCs are still concurrent
  - blocking full-GCs are very rare - bigger fragmentation (**BGC does not compact**)

GC mode - Background Workstation



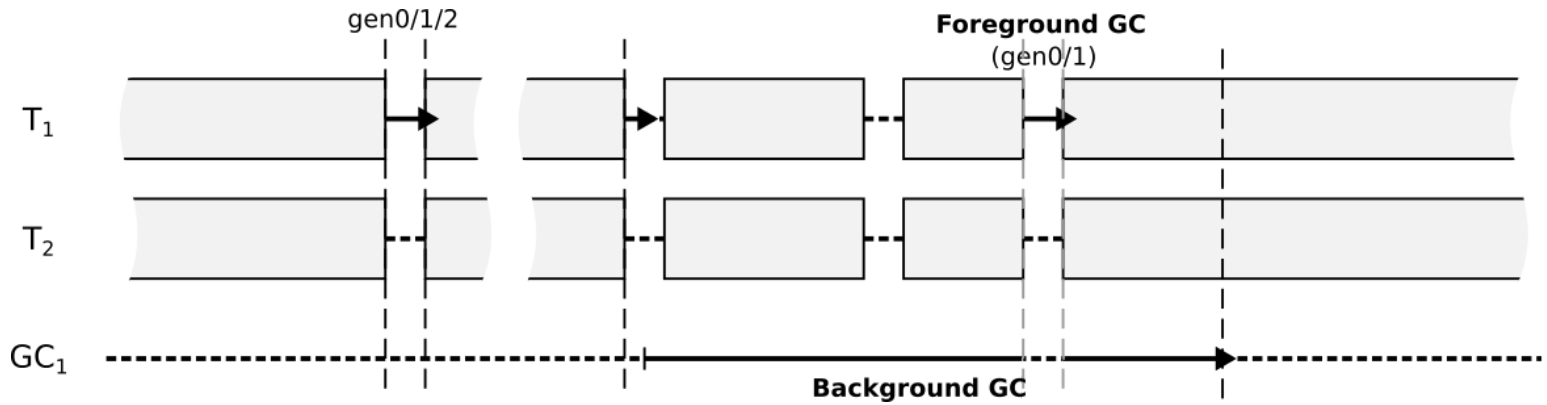
- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)
- most GCs are still concurrent
  - blocking full-GCs are very rare - bigger fragmentation (**BGC does not compact**)
  - concurrent mark - additional "floating" garbage

GC mode - Background Workstation



- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)
- most GCs are still concurrent
  - blocking full-GCs are very rare - bigger fragmentation (**BGC does not compact**)
  - concurrent mark - additional "floating" garbage
  - concurrent sweep of *old objects* - allocations are not allowed

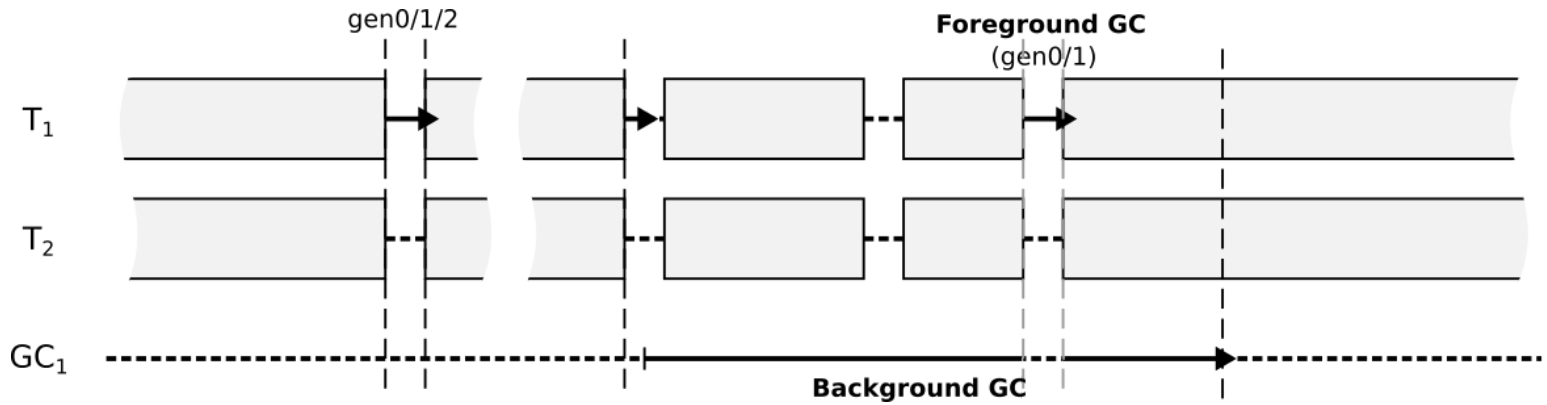
GC mode - Background Workstation



- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)
- most GCs are still concurrent
  - blocking full-GCs are very rare - bigger fragmentation (**BGC does not compact**)
  - concurrent mark - additional "floating" garbage
  - concurrent sweep of *old objects* - allocations are not allowed

Usage:

GC mode - Background Workstation

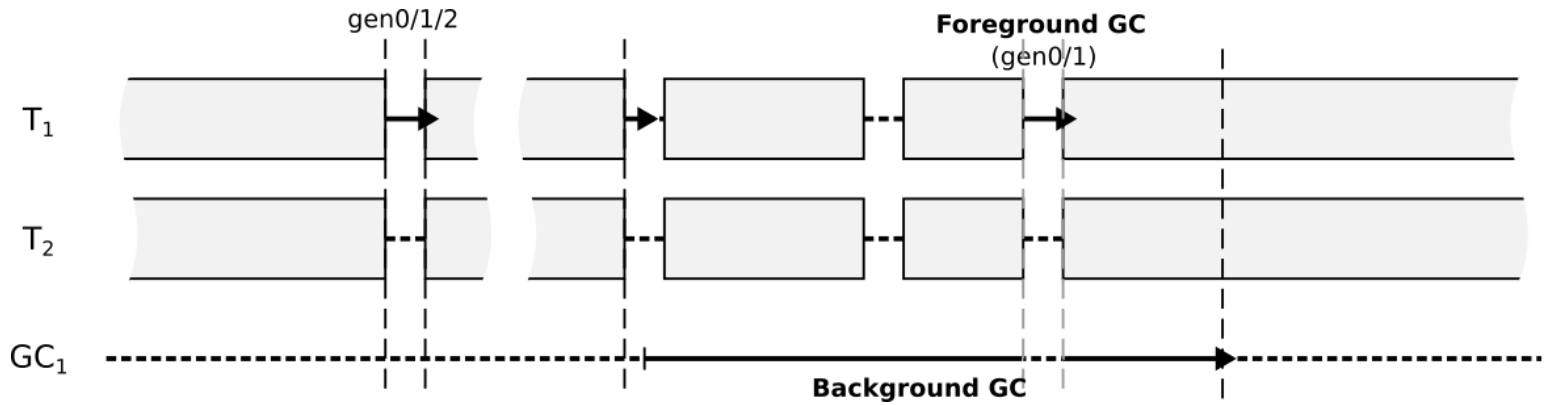


- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)
- most GCs are still concurrent
  - blocking full-GCs are very rare - bigger fragmentation (**BGC does not compact**)
  - concurrent mark - additional "floating" garbage
  - concurrent sweep of *old objects* - allocations are not allowed

Usage:

- Default since .NET 4.0 in Console/Desktop.

GC mode - Background Workstation

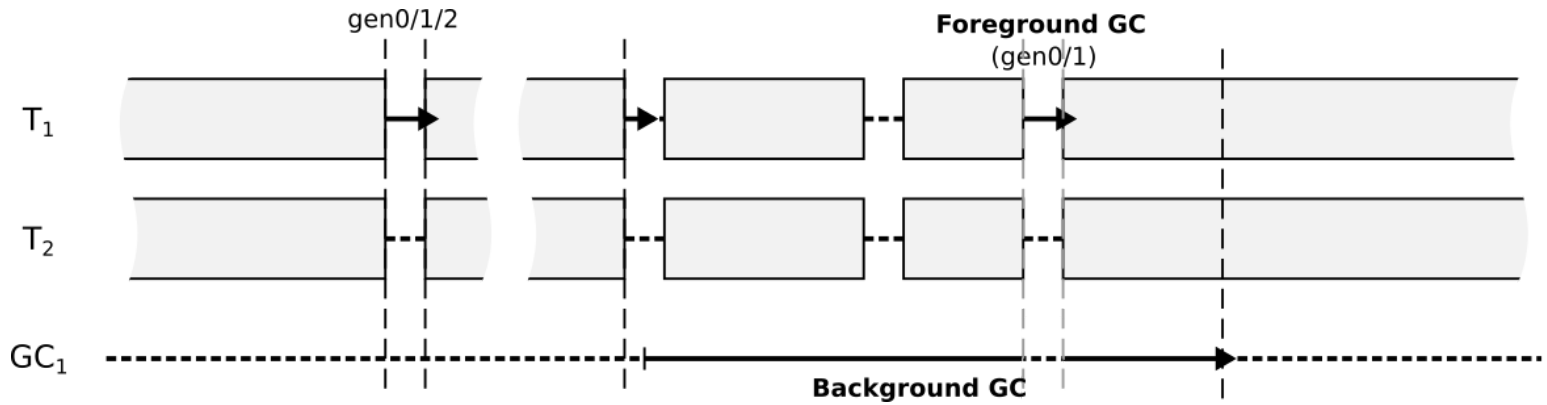


- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)
- most GCs are still concurrent
  - blocking full-GCs are very rare - bigger fragmentation (**BGC does not compact**)
  - concurrent mark - additional "floating" garbage
  - concurrent sweep of *old objects* - allocations are not allowed

Usage:

- Default since .NET 4.0 in Console/Desktop. But beware of Windows Services!

GC mode - Background Workstation

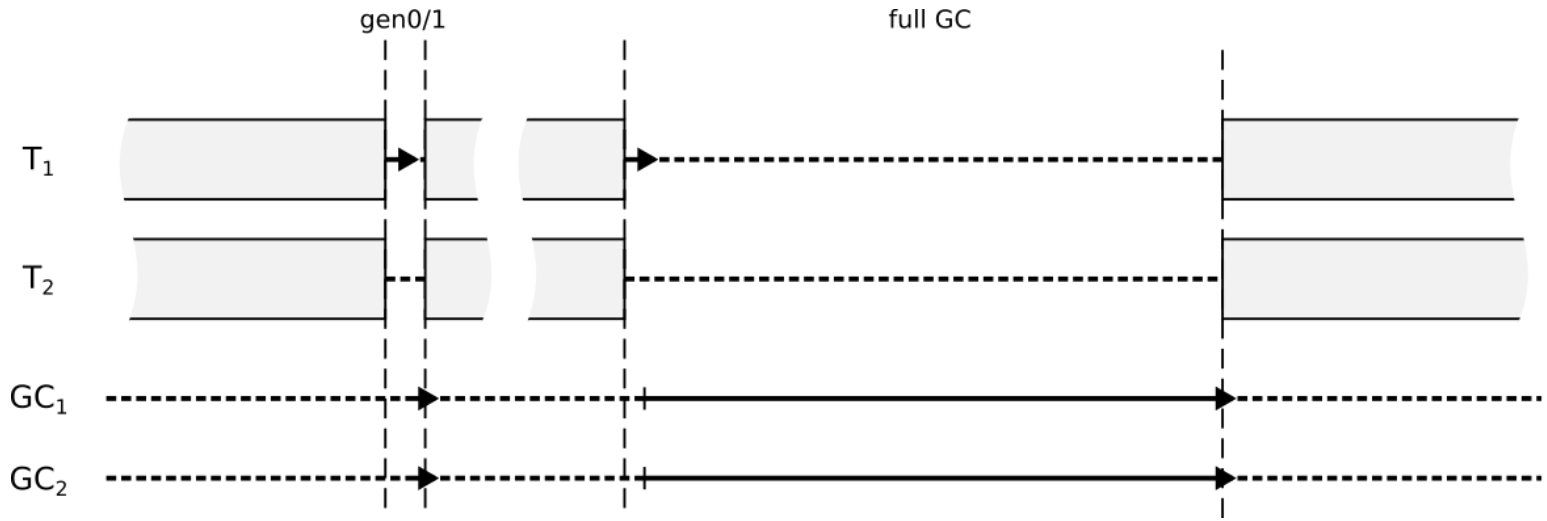


- "concurrent GC" renamed to **Background GC**
  - if *allocation budget* will be consumed during it, we don't wait but trigger short young **Foreground GC** (while pausing BGC)
- most GCs are still concurrent
  - blocking full-GCs are very rare - bigger fragmentation (**BGC does not compact**)
  - concurrent mark - additional "floating" garbage
  - concurrent sweep of *old objects* - allocations are not allowed

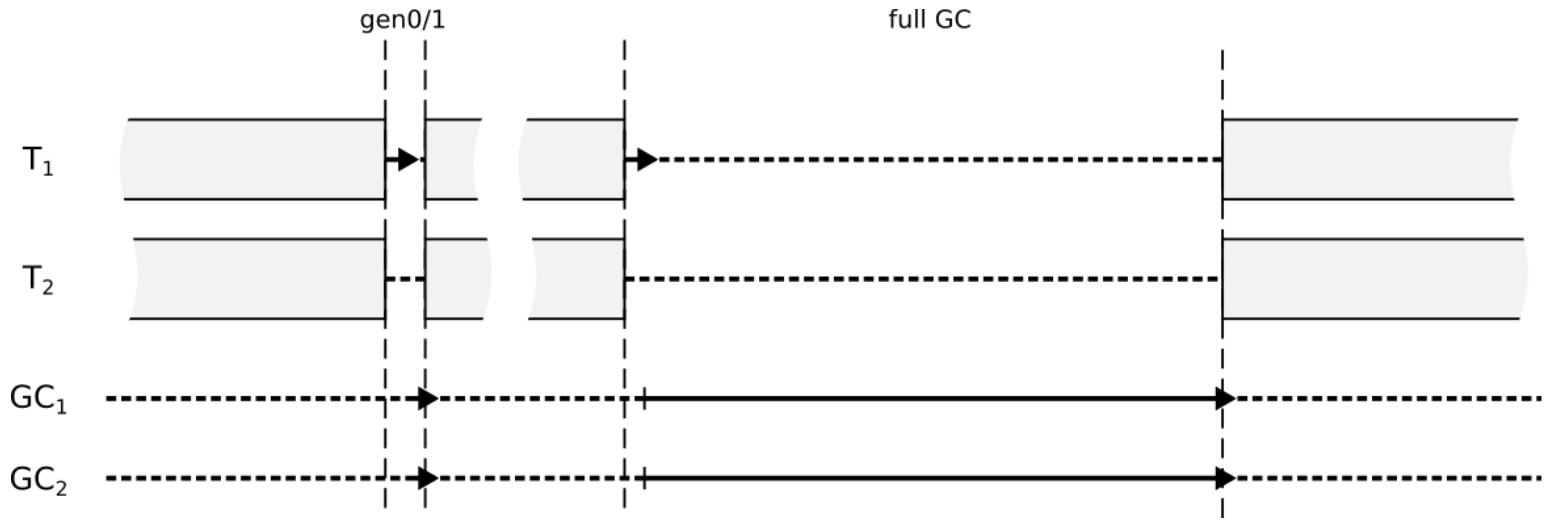
Usage:

- Default since .NET 4.0 in Console/Desktop. But beware of Windows Services!
- Saturated/dockerized environment

GC mode - Server Non-concurrent

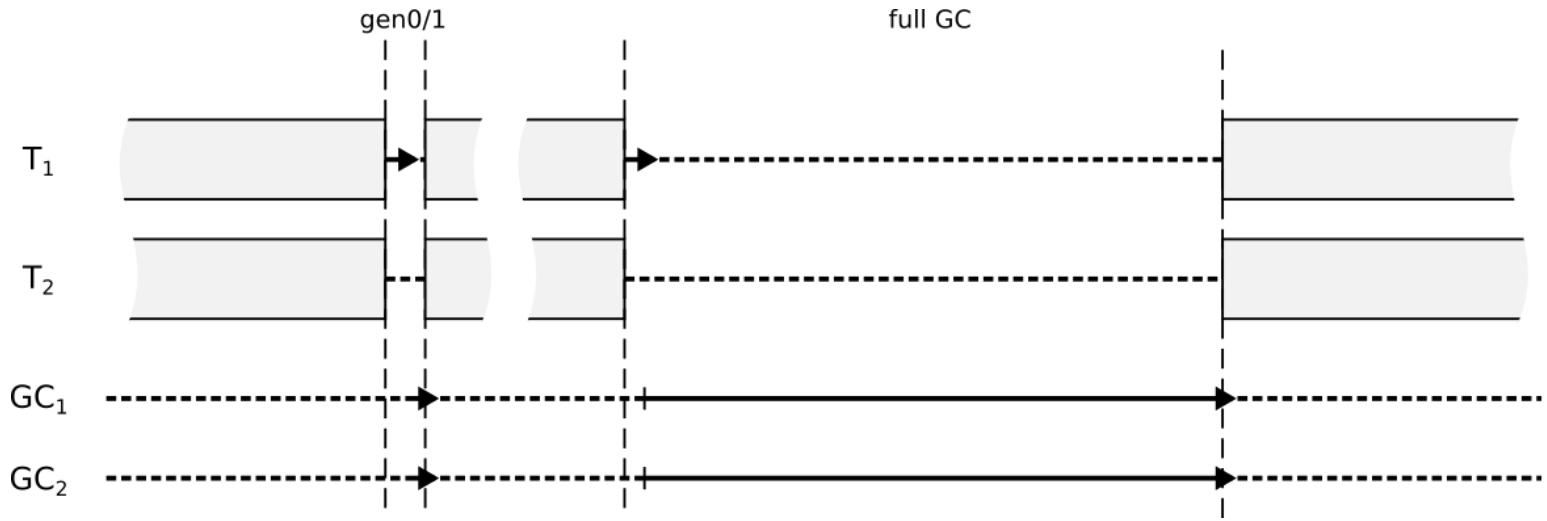


GC mode - Server Non-concurrent



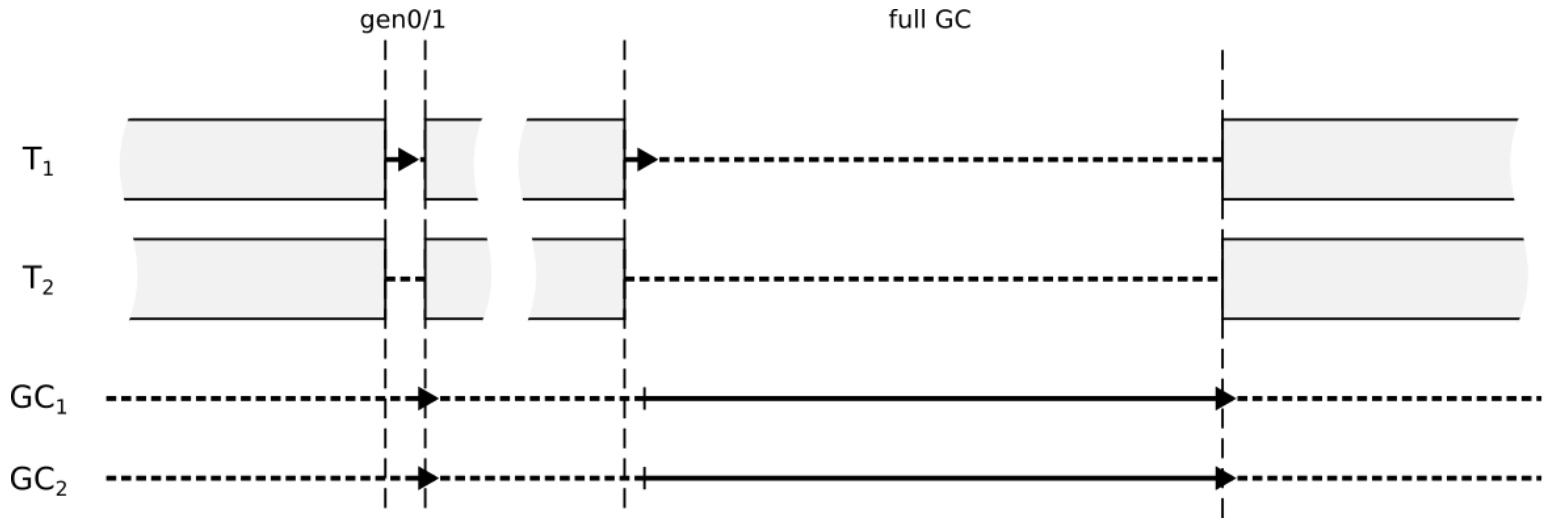
- multiple Managed Heaps and GC threads

GC mode - Server Non-concurrent



- multiple Managed Heaps and GC threads
- all GCs are blocking

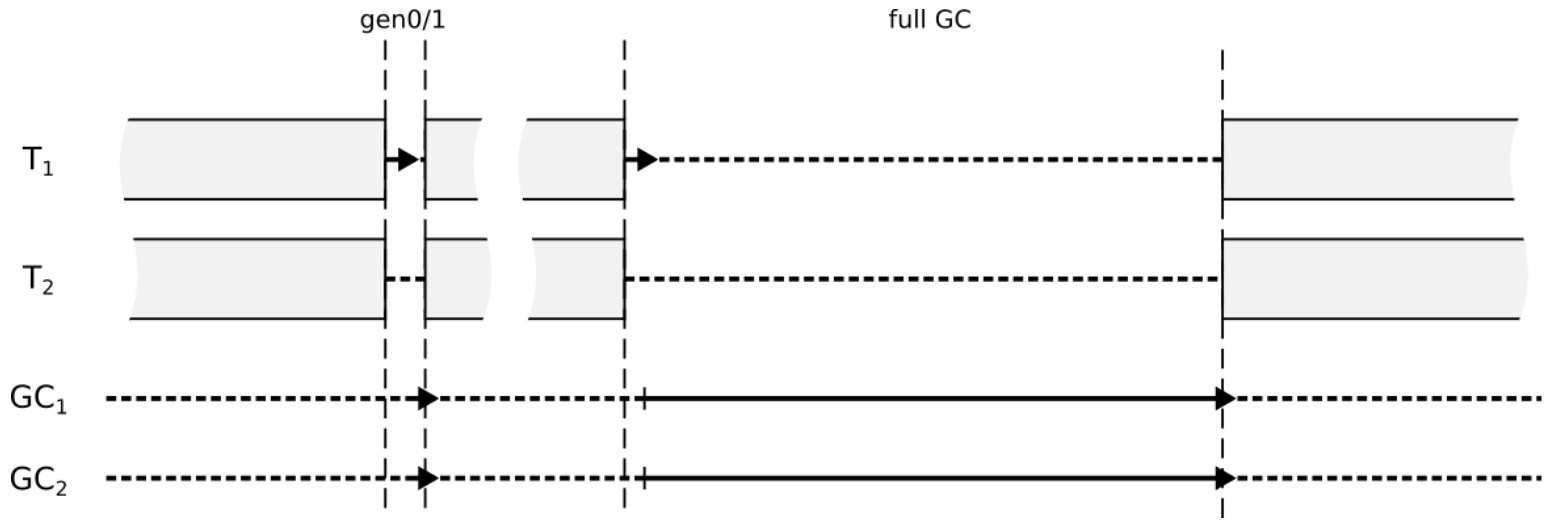
GC mode - Server Non-concurrent



- multiple Managed Heaps and GC threads
- all GCs are blocking

Usage:

GC mode - Server Non-concurrent

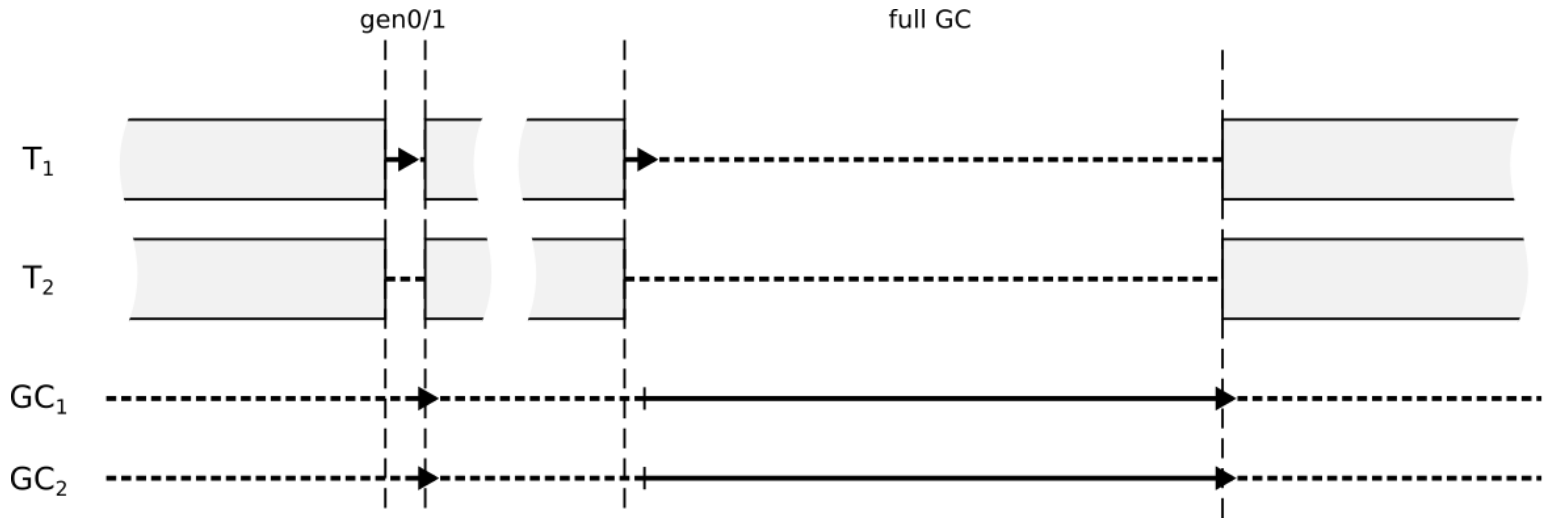


- multiple Managed Heaps and GC threads
- all GCs are blocking

Usage:

- Default in pre-.NET Framework 4.5 web apps

GC mode - Server Non-concurrent



- multiple Managed Heaps and GC threads
- all GCs are blocking

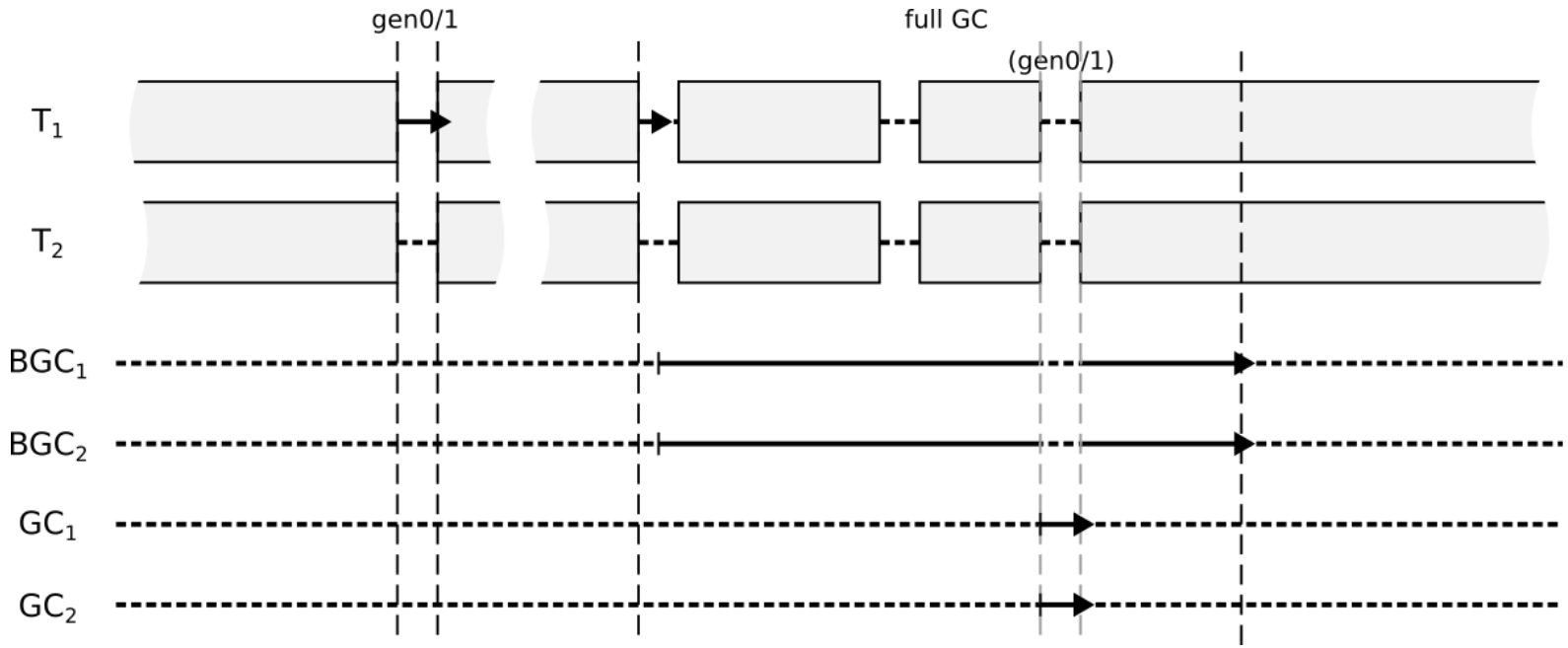
Usage:

- Default in pre-.NET Framework 4.5 web apps - GCs could be so long that *GC notifications* were invented

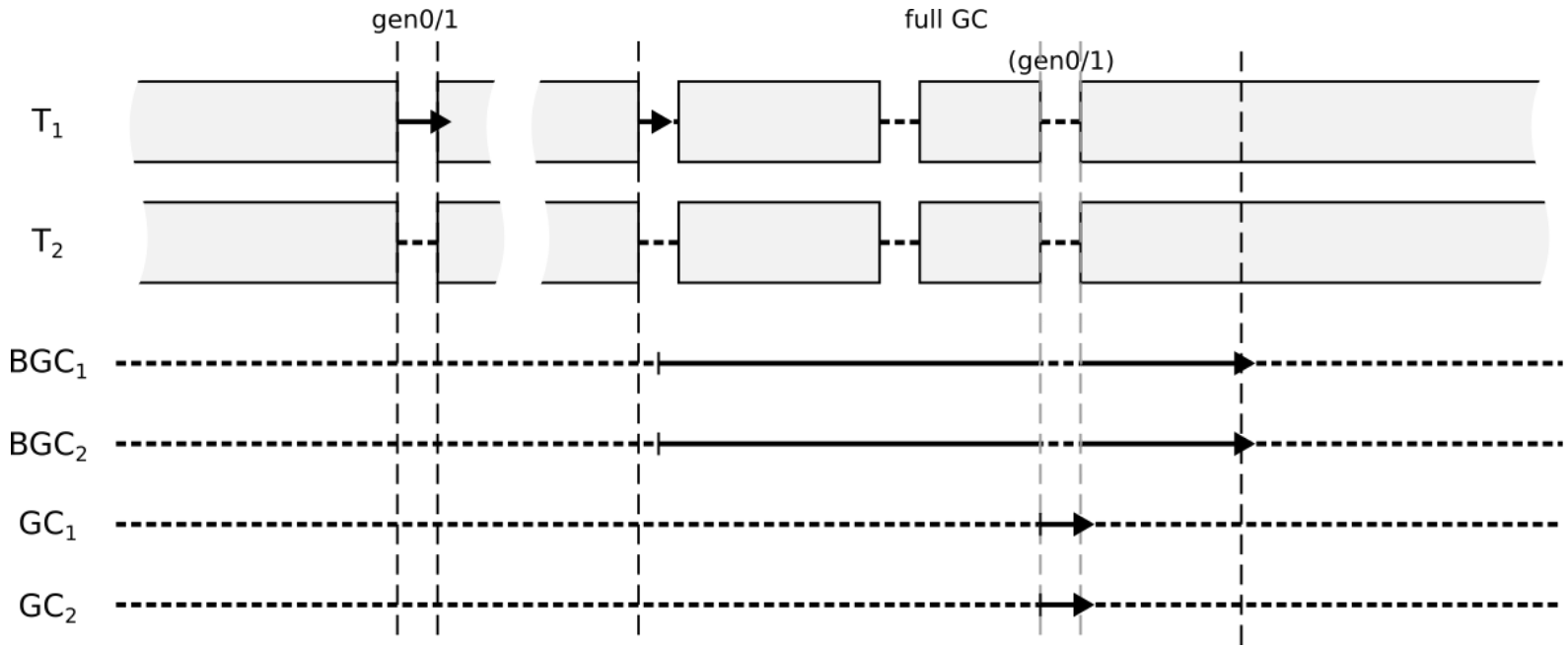
GC notifications

```
GC.RegisterForFullGCNotification(10, 10);
Thread startpolling = new Thread(() =>
{
    while (true)
    {
        GCNotificationStatus s = GC.WaitForFullGCApproach(1000);
        if (s == GCNotificationStatus.Succeeded)
        {
            Console.WriteLine("GC is about to begin");
        }
        // ...
        s = GC.WaitForFullGCComplete(1000);
        if (s == GCNotificationStatus.Succeeded)
        {
            Console.WriteLine("GC has ended");
        }
        Thread.Sleep(500);
    }
});
startpolling.Start();
GC.CancelFullGCNotification();
```

GC mode - Background Server

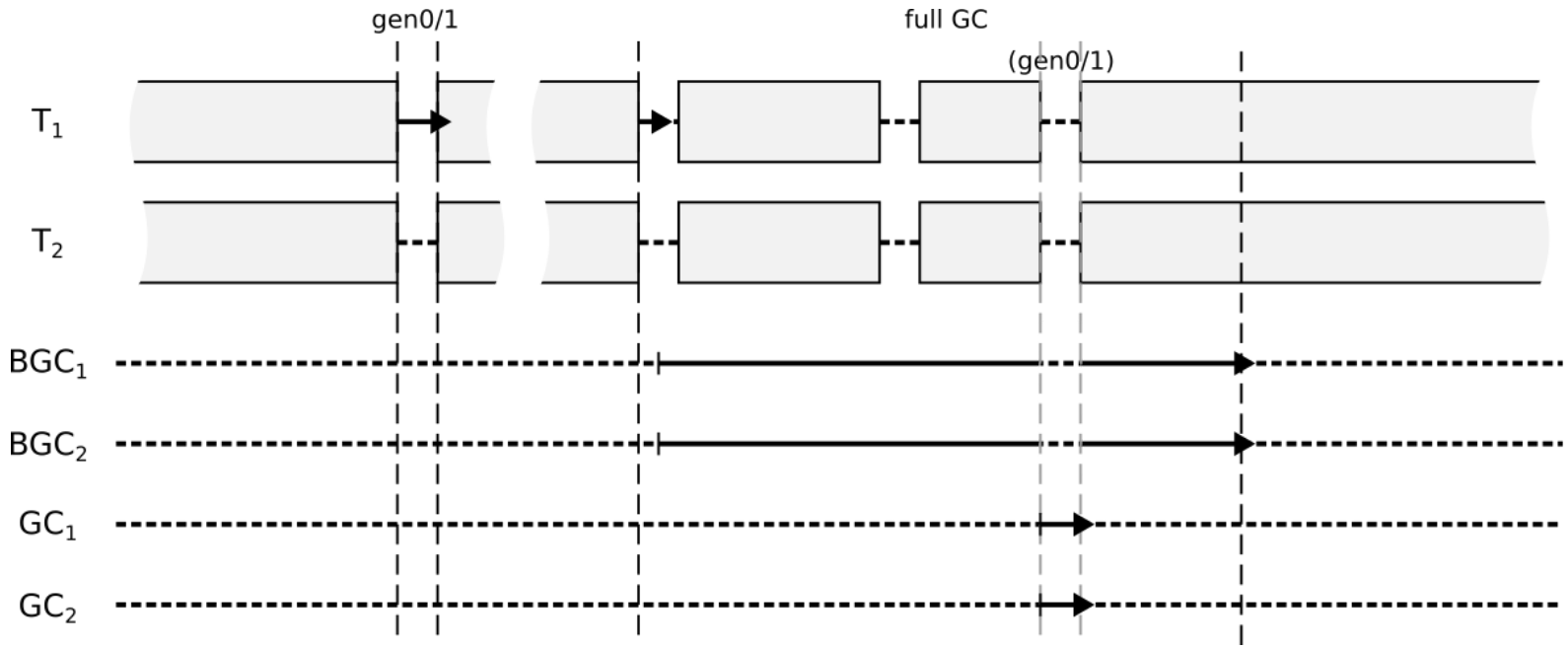


GC mode - Background Server



Usage:

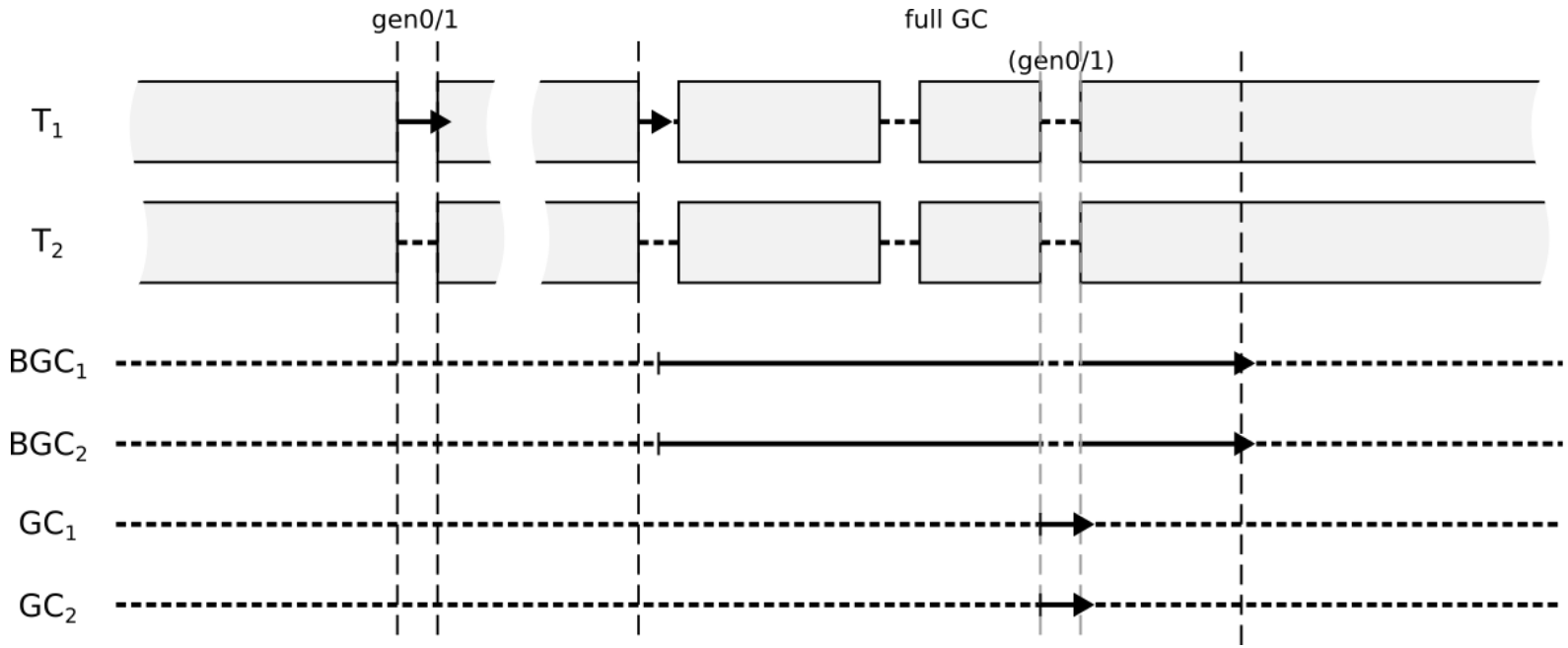
GC mode - Background Server



Usage:

- default GC for most server-based applications

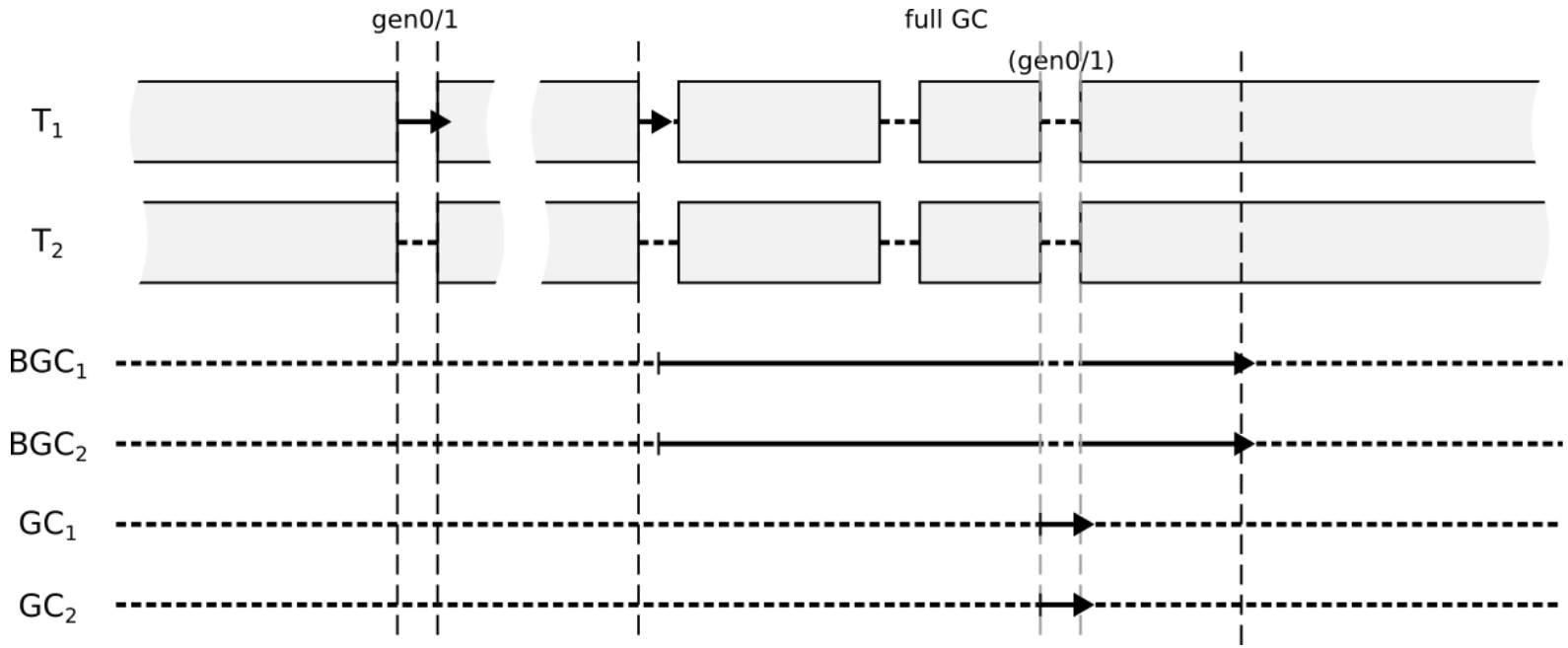
GC mode - Background Server



Usage:

- default GC for most server-based applications
- "give me all" UI clients (i.e. medical single stations)

GC mode - Background Server



Usage:

- default GC for most server-based applications
- "give me all" UI clients (i.e. medical single stations) , i.e. dnSpy (https://github.com/0xd4d/dnSpy)

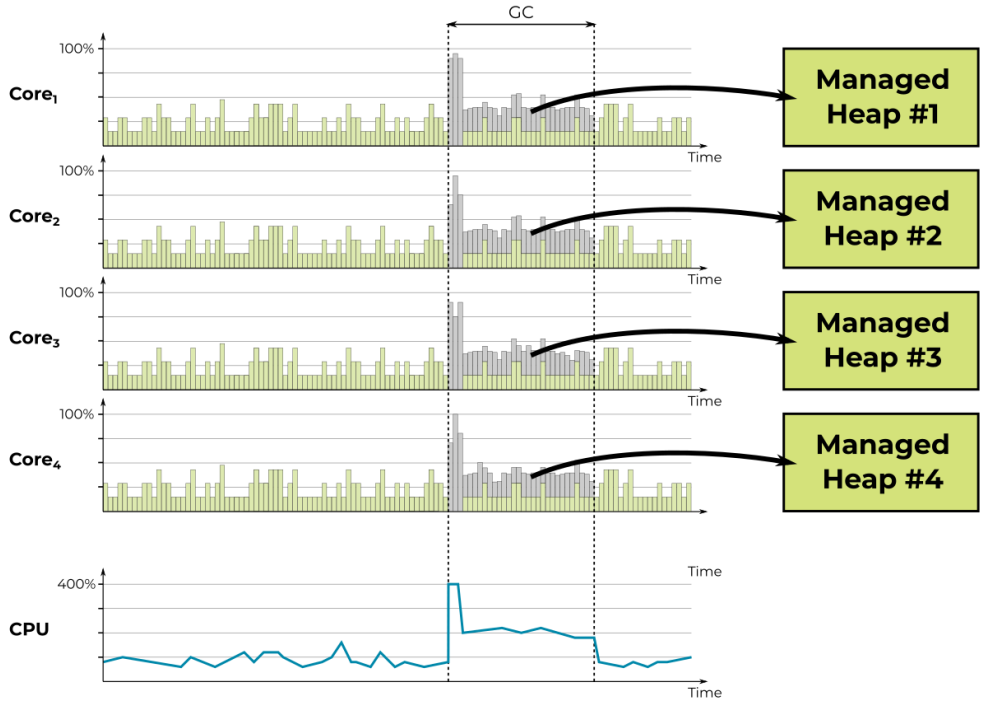
Server GC - additional configuration

GCHeapCount to limit number of Managed Heaps (and corresponsing GC threads):

```
<configuration>
   <runtime>
      <gcServer enabled="true"/>
      <GCHeapCount enabled="6"/>
   </runtime>
</configuration>
```

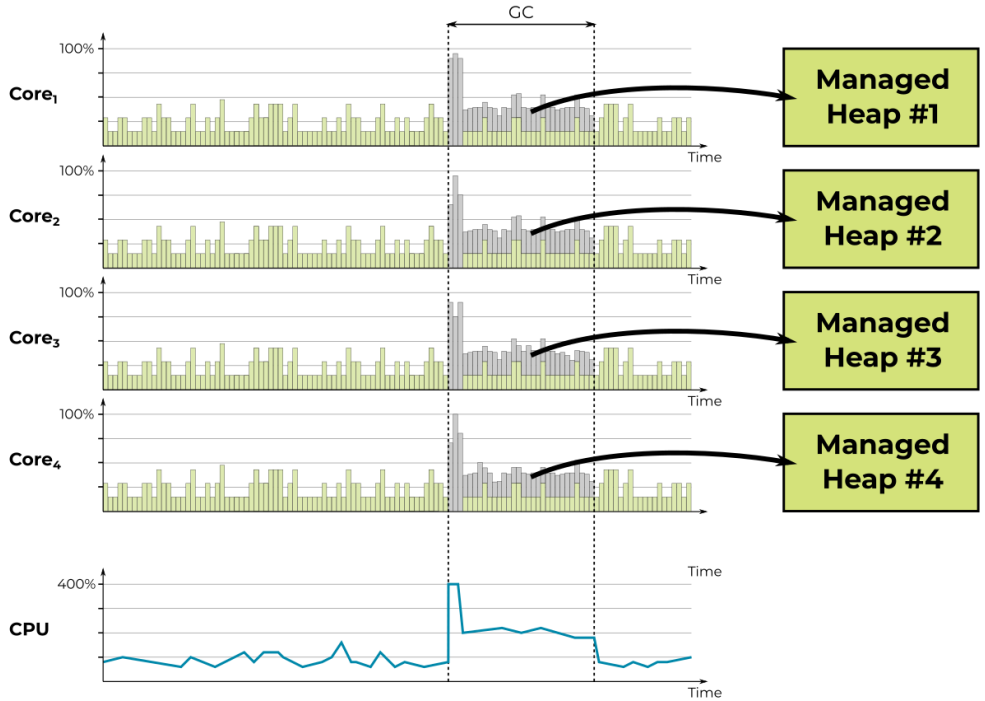
Server GC - additional configuration

`GCHeapCount=4`

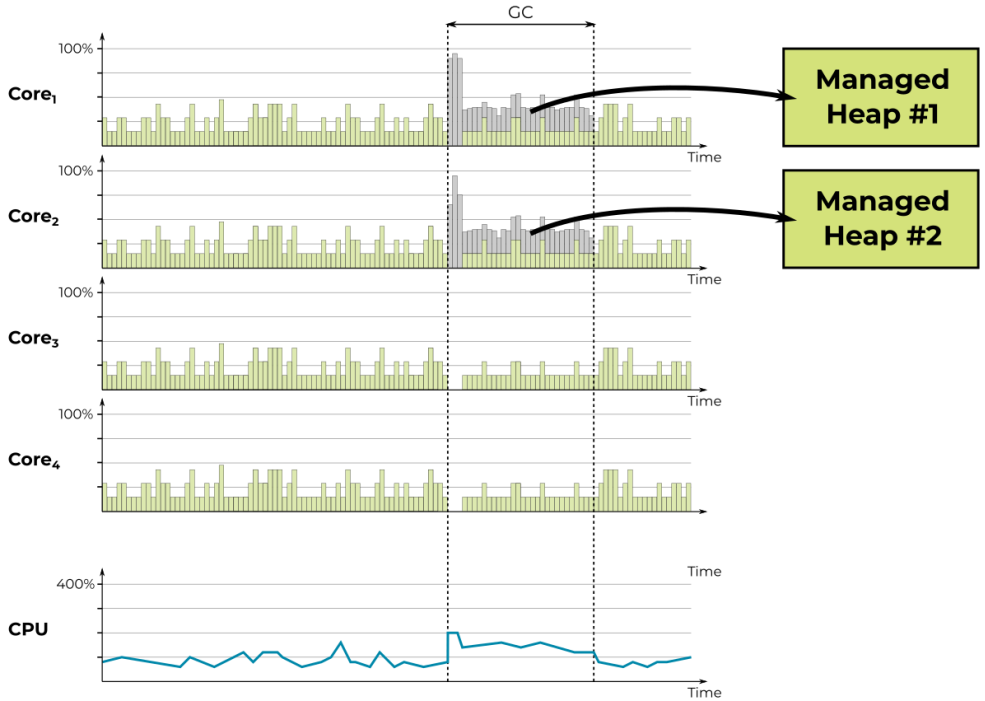


Server GC - additional configuration

`GCHeapCount=4`

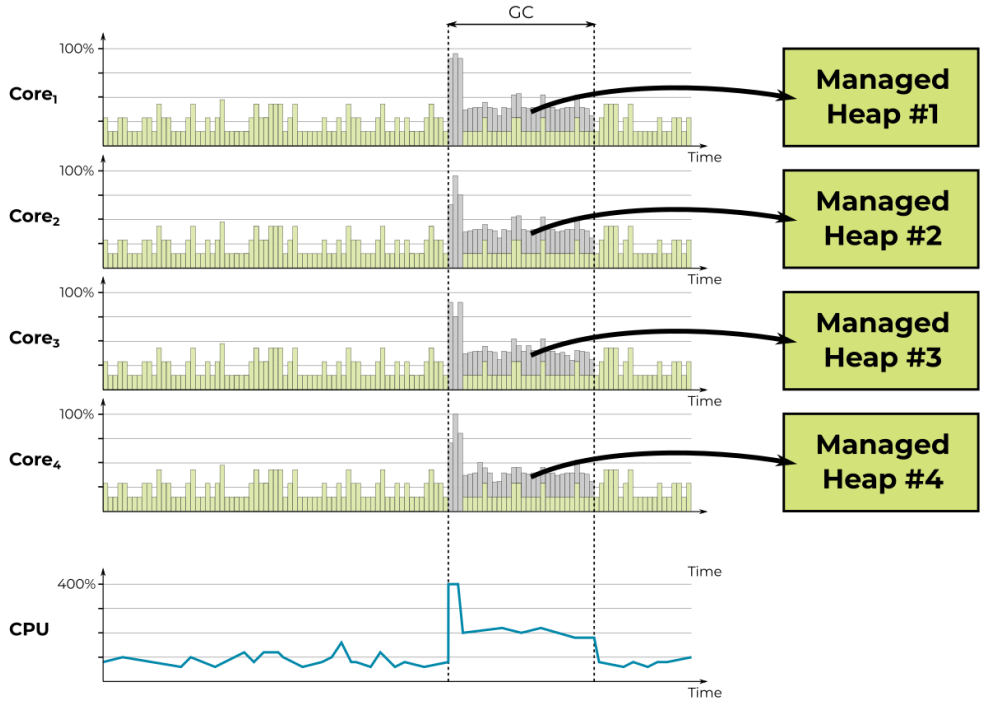


`GCHeapCount=2`

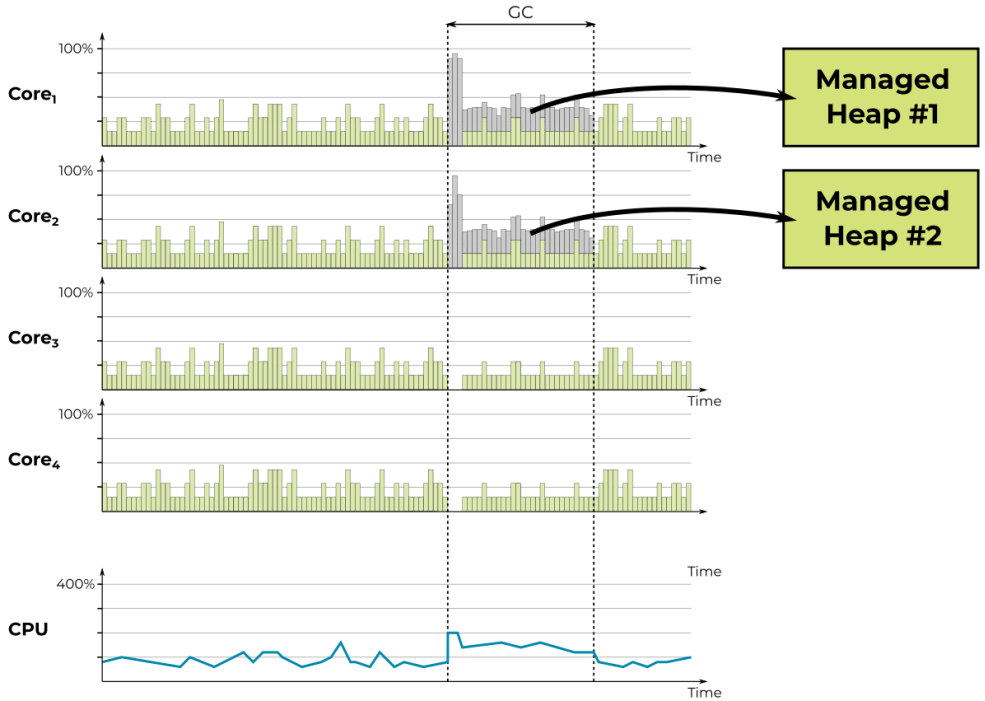


Server GC - additional configuration

`GCHeapCount=4`



`GCHeapCount=2`



- reduces memory usage - it makes the GC more aggressive
- reduces CPU usage - less CPU cores consumed during the GC
- may lengthen the pauses and influence throughput

Server GC - additional configuration

GCNoAffinitize and **GCHeapAffinitizeMask** to assign heaps to cores:

```
<configuration>
   <runtime>
      <gcServer enabled="true"/>
      <GCHeapCount enabled="6"/>
      <GCNoAffinitize enabled="true"/>
      <GCHeapAffinitizeMask enabled="144"/>
   </runtime>
</configuration>
```

Server GC - additional configuration

GCNoAffinitize and **GCHeapAffinitizeMask** to assign heaps to cores:

```
<configuration>
   <runtime>
      <gcServer enabled="true"/>
      <GCHeapCount enabled="6"/>
      <GCNoAffinitize enabled="true"/>
      <GCHeapAffinitizeMask enabled="144"/>
   </runtime>
</configuration>
```

- **GCNoAffinitize** specifies to not affinitize the Server GC threads with CPUs

Server GC - additional configuration

GCNoAffinitize and **GCHeapAffinitizeMask** to assign heaps to cores:

```
<configuration>
   <runtime>
      <gcServer enabled="true"/>
      <GCHeapCount enabled="6"/>
      <GCNoAffinitize enabled="true"/>
      <GCHeapAffinitizeMask enabled="144"/>
   </runtime>
</configuration>
```

- **GCNoAffinitize** specifies to not affinitize the Server GC threads with CPUs
- you may manually "assign" heap to the CPUs by **GCHeapAffinitizeMask**.

Server GC - additional configuration

Scenario: 32 CPUs, four apps. Creating 8 Managed Heaps per app and assigning 8 cores to each:

Server GC - additional configuration

Scenario: 32 CPUs, four apps. Creating 8 Managed Heaps per app and assigning 8 cores to each:
Process 1:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="255"/>        <!-- 00000000 00000000 00000000 11111111 -->
```

Server GC - additional configuration

Scenario: 32 CPUs, four apps. Creating 8 Managed Heaps per app and assigning 8 cores to each:
Process 1:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="255"/>        <!-- 00000000 00000000 00000000 11111111 -->
```

Process 2:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="65280"/>      <!-- 00000000 00000000 11111111 00000000 -->
```

Server GC - additional configuration

Scenario: 32 CPUs, four apps. Creating 8 Managed Heaps per app and assigning 8 cores to each:
Process 1:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="255"/>        <!-- 00000000 00000000 00000000 11111111 -->
```

Process 2:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="65280"/>      <!-- 00000000 00000000 11111111 00000000 -->
```

Process 3:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="16711680"/>   <!-- 00000000 11111111 00000000 00000000 -->
```

Server GC - additional configuration

Scenario: 32 CPUs, four apps. Creating 8 Managed Heaps per app and assigning 8 cores to each:
Process 1:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="255"/>        <!-- 00000000 00000000 00000000 11111111 -->
```

Process 2:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="65280"/>      <!-- 00000000 00000000 11111111 00000000 -->
```

Process 3:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="16711680"/>   <!-- 00000000 11111111 00000000 00000000 -->
```

Process 4:

```
<GCHeapCount enabled="8"/>
<GCHeapAffinitizeMask enabled="4278190080"/> <!-- 11111111 00000000 00000000 00000000 -->
```

Server GC - additional configuration

In **container scenarios**, with a memory limit set:

Server GC - additional configuration

In **container scenarios**, with a memory limit set:

- 75% of the limit is treated as **hard limit**

Server GC - additional configuration

In **container scenarios**, with a memory limit set:

- 75% of the limit is treated as **hard limit**
  - for example, inside container with 200MB limit, the managed memory limit will be 150MB (0.75*200MB)

Server GC - additional configuration

In **container scenarios**, with a memory limit set:

- 75% of the limit is treated as **hard limit**
  - for example, inside container with 200MB limit, the managed memory limit will be 150MB (0.75*200MB)
- since .NET Core 3.0 (PR #22180) two more configuration knobs:

Server GC - additional configuration

In **container scenarios**, with a memory limit set:

- 75% of the limit is treated as **hard limit**
  - for example, inside container with 200MB limit, the managed memory limit will be 150MB (0.75*200MB)
- since .NET Core 3.0 (PR #22180) two more configuration knobs:
  - `GCHeapHardLimit` - specifies our custom hard limit for the GC heap

Server GC - additional configuration

In **container scenarios**, with a memory limit set:

- 75% of the limit is treated as **hard limit**
  - for example, inside container with 200MB limit, the managed memory limit will be 150MB (0.75*200MB)
- since .NET Core 3.0 (PR #22180) two more configuration knobs:
  - `GCHeapHardLimit` - specifies our custom hard limit for the GC heap
  - `GCHeapHardLimitPercent` - specifies a percentage of the physical memory this process is allowed to use with the respect to the overall memory limit

Server GC - additional configuration

In **container scenarios**, with a memory limit set:

- 75% of the limit is treated as **hard limit**
  - for example, inside container with 200MB limit, the managed memory limit will be 150MB (0.75*200MB)
- since .NET Core 3.0 (PR #22180) two more configuration knobs:
  - `GCHeapHardLimit` - specifies our custom hard limit for the GC heap
  - `GCHeapHardLimitPercent` - specifies a percentage of the physical memory this process is allowed to use with the respect to the overall memory limit
- BTW you can use them outside containers, to control GC aggressiveness & memory footprint

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)
- when it happens, GC becomes more aggressive:

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)
- when it happens, GC becomes more aggressive:
  - instead of sweeping background mode it prefers...

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)
- when it happens, GC becomes more aggressive:
  - instead of sweeping background mode it prefers...
  - doing full, compacting mode

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)
- when it happens, GC becomes more aggressive:
  - instead of sweeping background mode it prefers...
  - doing full, compacting mode
- we can control this threshold with:

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)
- when it happens, GC becomes more aggressive:
  - instead of sweeping background mode it prefers...
  - doing full, compacting mode
- we can control this threshold with:
  - `COMPlus_GCHighMemPercent` environment variable

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)
- when it happens, GC becomes more aggressive:
  - instead of sweeping background mode it prefers...
  - doing full, compacting mode
- we can control this threshold with:
  - `COMPlus_GCHighMemPercent` environment variable
  - `System.GC.HighMemoryPercent` setting in `runtimeconfig.json` file (since .NET 5)

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)
- when it happens, GC becomes more aggressive:
  - instead of sweeping background mode it prefers...
  - doing full, compacting mode
- we can control this threshold with:
  - `COMPlus_GCHighMemPercent` environment variable
  - `System.GC.HighMemoryPercent` setting in `runtimeconfig.json` file (since .NET 5)
- so...

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)
- when it happens, GC becomes more aggressive:
  - instead of sweeping background mode it prefers...
  - doing full, compacting mode
- we can control this threshold with:
  - `COMPlus_GCHighMemPercent` environment variable
  - `System.GC.HighMemoryPercent` setting in `runtimeconfig.json` file (since .NET 5)
- so...
  - when your process is small (~1GB) - maybe it can perfectly life with bigger threshold like even 97%

GC - high memory configuration

- there is a **high memory load** situation detected by the runtime when the memory usage **on the machine** gets big
- by default the threshold is 90% (and 90-97% on machines with 80+ GB of RAM)
- when it happens, GC becomes more aggressive:
  - instead of sweeping background mode it prefers...
  - doing full, compacting mode
- we can control this threshold with:
  - `COMPlus_GCHighMemPercent` environment variable
  - `System.GC.HighMemoryPercent` setting in `runtimeconfig.json` file (since .NET 5)
- so...
  - when your process is small (~1GB) - maybe it can perfectly life with bigger threshold like even 97%
  - when your process is huge but you would like to limit it (make more aggressive) - maybe it can be reduced to smaller value (or use GC heap *hard limit* as absolute value)

Latency modes

Additionally, we can control pauses with the **latency mode**:

```
public enum GCLatencyMode
{
   Batch = 0,
   Interactive = 1,
   LowLatency = 2,
   SustainedLowLatency = 3,
   NoGCRegion = 4
}
```

```
GCSettings.LatencyMode = GCLatencyMode.Batch;
```

Latency modes

Batch latency mode:

Latency modes

Batch latency mode:

- not concerned about pauses length, but rather throughput or memory usage

Latency modes

Batch latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs

Latency modes

Batch latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**

Latency modes

Batch latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen

Latency modes

Batch latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:

Latency modes

Batch latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:
  - for Server GC - are infinitely suspended

Latency modes

`Batch` latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:
  - for Server GC - are infinitely suspended
  - for Workstation GC - destroyed after 20 seconds timeout (`GCTerminateConcurrentThread` event)

Latency modes

`Batch` latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:
  - for Server GC - are infinitely suspended
  - for Workstation GC - destroyed after 20 seconds timeout (`GCTerminateConcurrentThread` event)

`Interactive` latency mode:

Latency modes

`Batch` latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:
  - for Server GC - are infinitely suspended
  - for Workstation GC - destroyed after 20 seconds timeout (`GCTerminateConcurrentThread` event)

`Interactive` latency mode:

- short pauses are most desired, even in cost of memory usage

Latency modes

`Batch` latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:
  - for Server GC - are infinitely suspended
  - for Workstation GC - destroyed after 20 seconds timeout (`GCTerminateConcurrentThread` event)

`Interactive` latency mode:

- short pauses are most desired, even in cost of memory usage
- default setting for all concurrent GCs

Latency modes

`Batch` latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:
  - for Server GC - are infinitely suspended
  - for Workstation GC - destroyed after 20 seconds timeout (`GCTerminateConcurrentThread` event)

`Interactive` latency mode:

- short pauses are most desired, even in cost of memory usage
- default setting for all concurrent GCs
- enables Background GC possibility

Latency modes

`Batch` latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:
  - for Server GC - are infinitely suspended
  - for Workstation GC - destroyed after 20 seconds timeout (`GCTerminateConcurrentThread` event)

`Interactive` latency mode:

- short pauses are most desired, even in cost of memory usage
- default setting for all concurrent GCs
- enables Background GC possibility
- background GC threads:

Latency modes

`Batch` latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:
  - for Server GC - are infinitely suspended
  - for Workstation GC - destroyed after 20 seconds timeout (`GCTerminateConcurrentThread` event)

`Interactive` latency mode:

- short pauses are most desired, even in cost of memory usage
- default setting for all concurrent GCs
- enables Background GC possibility
- background GC threads:
  - will be created if they do not exist already (`GCCreateConcurrentThread` event)

Latency modes

`Batch` latency mode:

- not concerned about pauses length, but rather throughput or memory usage
- default latency setting for non-concurrent GCs
- **disables the possibility of the Background GC**
- so, only stop-the-world GCs will happen
- background GC threads:
  - for Server GC - are infinitely suspended
  - for Workstation GC - destroyed after 20 seconds timeout (`GCTerminateConcurrentThread` event)

`Interactive` latency mode:

- short pauses are most desired, even in cost of memory usage
- default setting for all concurrent GCs
- enables Background GC possibility
- background GC threads:
  - will be created if they do not exist already (`GCCreateConcurrentThread` event)

In general, we may treat them as a toggle between Concurrent and Non-concurrent GC.

Latency modes

`LowLatency` mode:

Latency modes

`LowLatency` mode:

- short as possible pauses are essential, at any cost

Latency modes

`LowLatency` mode:

- short as possible pauses are essential, at any cost
- available only in *Workstation GC* mode

Latency modes

`LowLatency` mode:

- short as possible pauses are essential, at any cost
- available only in *Workstation GC* mode
- **disables all Full GCs (both concurrent and non-concurrent)**

Latency modes

`LowLatency` mode:

- short as possible pauses are essential, at any cost
- available only in *Workstation GC* mode
- **disables all Full GCs (both concurrent and non-concurrent)**
  - Full GC is possible only with low-memory system notification or via explicit call

Latency modes

`LowLatency` mode:

- short as possible pauses are essential, at any cost
- available only in *Workstation GC* mode
- **disables all Full GCs (both concurrent and non-concurrent)**
  - Full GC is possible only with low-memory system notification or via explicit call
- results:

Latency modes

`LowLatency` mode:

- short as possible pauses are essential, at any cost
- available only in *Workstation GC* mode
- **disables all Full GCs (both concurrent and non-concurrent)**
  - Full GC is possible only with low-memory system notification or via explicit call
- results:
  - pause times will be really short

Latency modes

`LowLatency` mode:

- short as possible pauses are essential, at any cost
- available only in *Workstation GC* mode
- **disables all Full GCs (both concurrent and non-concurrent)**
  - Full GC is possible only with low-memory system notification or via explicit call
- results:
  - pause times will be really short
  - memory usage will likely grow fast

Latency modes

LowLatency mode:

```
GCLatencyMode oldMode = GCSettings.LatencyMode;
RuntimeHelpers.PrepareConstrainedRegions();
try
{
    GCSettings.LatencyMode = GCLatencyMode.LowLatency;
   //Perform time-sensitive, short work here
}
finally
{
   GCSettings.LatencyMode = oldMode;
}
```

Latency modes

`SustainedLowLatency` mode:

- short as possible pauses are essential

Latency modes

`SustainedLowLatency` mode:

- short as possible pauses are essential
- introduced in .NET Framework 4.5

Latency modes

`SustainedLowLatency` mode:

- short as possible pauses are essential
- introduced in .NET Framework 4.5
- only non-concurrent full-GCs are disabled

Latency modes

`SustainedLowLatency` mode:

- short as possible pauses are essential
- introduced in .NET Framework 4.5
- only non-concurrent full-GCs are disabled
    - possible only with low-memory system notification or via explicit trigger

Latency modes

`SustainedLowLatency` mode:

- short as possible pauses are essential
- introduced in .NET Framework 4.5
- only non-concurrent full-GCs are disabled
  - possible only with low-memory system notification or via explicit trigger
- only if runtime has started with the concurrent setting enabled (regardless of later `Batch`/`Interactive` changes)

Latency modes

`SustainedLowLatency` mode:

- short as possible pauses are essential
- introduced in .NET Framework 4.5
- only non-concurrent full-GCs are disabled
  - possible only with low-memory system notification or via explicit trigger
- only if runtime has started with the concurrent setting enabled (regardless of later `Batch`/`Interactive` changes)
- result - good and safe compromise between latency and memory usage for short period of time

Latency modes

SustainedLowLatency mode:

```
internal static class GCManager
{
/// <summary>
/// Call this method to suppress expensive blocking Gen 2 garbage GCs in scenarios where high-latency is
/// unacceptable (e.g. processing typing input).
/// </summary>
internal static void UseLowLatencyModeForProcessingUserInput() {
  var currentMode = GCSettings.LatencyMode;
  var currentDelay = s_delay;
  if (currentMode != GCLatencyMode.SustainedLowLatency)
  {
    GCSettings.LatencyMode = GCLatencyMode.SustainedLowLatency;
    // Restore the LatencyMode a short duration after the
    // last request to UseLowLatencyModeForProcessingUserInput.
    currentDelay = new ResettableDelay(s_delayMilliseconds);
    currentDelay.Task.SafeContinueWith(_ => RestoreGCLatencyMode(currentMode), ...);
    s_delay = currentDelay;
  }
  if (currentDelay != null)
  {
    currentDelay.Reset();
  }
}
```

Latency modes

`NoGCRegion` latency mode:

Latency modes

`NoGCRegion` latency mode:

- attempts to disallow garbage collection during the execution of a critical path if a specified amount of memory is available

Latency modes

`NoGCRegion` latency mode:

- attempts to disallow garbage collection during the execution of a critical path if a specified amount of memory is available
- introduced in .NET Framework 4.6

Latency modes

`NoGCRegion` latency mode:

- attempts to disallow garbage collection during the execution of a critical path if a specified amount of memory is available
- introduced in .NET Framework 4.6
- using different API:

```
bool GC.TryStartNoGCRegion(long totalSize)
bool GC.TryStartNoGCRegion(long totalSize,
                           bool disallowFullBlockingGC)
bool GC.TryStartNoGCRegion(long totalSize, long lohSize)
bool GC.TryStartNoGCRegion(long totalSize, long lohSize,
                           bool disallowFullBlockingGC)
```

Latency modes

NoGCRegion latency mode:

```
// in case of previous finally block not executed
if (GCSettings.LatencyMode == GCLatencyMode.NoGCRegion)
   GC.EndNoGCRegion();
if (GC.TryStartNoGCRegion(1024, true))
{
   try
   {
      // Do some work.
   }
   finally
   {
      try
      {
         GC.EndNoGCRegion();
      }
      catch (InvalidOperationException ex)
      {
         // Log message
      }
   }
}
```

Latency levels

CoreCLR comment:

"Latency modes required user to have specific GC knowledge (e.g., budget, full-blocking GC). We are trying to move away from them as it makes a lot more sense for users to tell us what's the most important out of the performance aspects that make sense to them"

Latency levels

CoreCLR comment:

"Latency modes required user to have specific GC knowledge (e.g., budget, full-blocking GC). We are trying to move away from them as it makes a lot more sense for users to tell us what's the most important out of the performance aspects that make sense to them"

Four such goals are planned:

Latency levels

CoreCLR comment:

"Latency modes required user to have specific GC knowledge (e.g., budget, full-blocking GC). We are trying to move away from them as it makes a lot more sense for users to tell us what's the most important out of the performance aspects that make sense to them"

Four such goals are planned:

- *memory footprint* - pauses can be long and more frequent but heap size stays small,

Latency levels

CoreCLR comment:

"Latency modes required user to have specific GC knowledge (e.g., budget, full-blocking GC). We are trying to move away from them as it makes a lot more sense for users to tell us what's the most important out of the performance aspects that make sense to them"

Four such goals are planned:

- *memory footprint* - pauses can be long and more frequent but heap size stays small,
- *throughput* - pauses are unpredictable and not very frequent, promoting throughput,

Latency levels

CoreCLR comment:

"Latency modes required user to have specific GC knowledge (e.g., budget, full-blocking GC). We are trying to move away from them as it makes a lot more sense for users to tell us what's the most important out of the performance aspects that make sense to them"

Four such goals are planned:

- *memory footprint* - pauses can be long and more frequent but heap size stays small,
- *throughput* - pauses are unpredictable and not very frequent, promoting throughput,
- *balance between pauses and throughput* (default) - pauses are more predictable and more frequent. The longest pauses are shorter than *memory footprint* pauses,

Latency levels

CoreCLR comment:

"Latency modes required user to have specific GC knowledge (e.g., budget, full-blocking GC). We are trying to move away from them as it makes a lot more sense for users to tell us what's the most important out of the performance aspects that make sense to them"

Four such goals are planned:

- *memory footprint* - pauses can be long and more frequent but heap size stays small,
- *throughput* - pauses are unpredictable and not very frequent, promoting throughput,
- *balance between pauses and throughput* (default) - pauses are more predictable and more frequent. The longest pauses are shorter than *memory footprint* pauses,
- *short pauses* - pauses are more predictable and more frequent. The longest pauses are shorter than *balanced* pauses.

Latency levels

CoreCLR comment:

"Latency modes required user to have specific GC knowledge (e.g., budget, full-blocking GC). We are trying to move away from them as it makes a lot more sense for users to tell us what's the most important out of the performance aspects that make sense to them"

Four such goals are planned:

- ***memory footprint*** **- pauses can be long and more frequent but heap size stays small,**
- *throughput* - pauses are unpredictable and not very frequent, promoting throughput,
- ***balance between pauses and throughput*** **(default) - pauses are more predictable and more frequent. The longest pauses are shorter than *memory footprint* pauses,**
- *short pauses* - pauses are more predictable and more frequent. The longest pauses are shorter than *balanced* pauses.

Latency levels

CoreCLR comment:

"Latency modes required user to have specific GC knowledge (e.g., budget, full-blocking GC). We are trying to move away from them as it makes a lot more sense for users to tell us what's the most important out of the performance aspects that make sense to them"

Four such goals are planned:

- ***memory footprint*** **- pauses can be long and more frequent but heap size stays small,**
- *throughput* - pauses are unpredictable and not very frequent, promoting throughput,
- ***balance between pauses and throughput*** **(default) - pauses are more predictable and more frequent. The longest pauses are shorter than *memory footprint* pauses,**
- *short pauses* - pauses are more predictable and more frequent. The longest pauses are shorter than *balanced* pauses.

Available via `GCLatencyLevel` Configuration Knob (i.e. `COMPlus_GCLatencyLevel` environment variable).

Materials

- Runtime configuration options for garbage collection
- Understanding different GC modes with Concurrency Visualizer
- Top 5 .NET memory management fundamentals
- The history of the GC configs

DEMO