Make your custom .NET GC

"whys" and "hows"

Konrad Kokosa

Welcome To The World Of Custom GCs!

Welcome To The World Of Custom GCs!

which in .NET does not exist so much...

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…

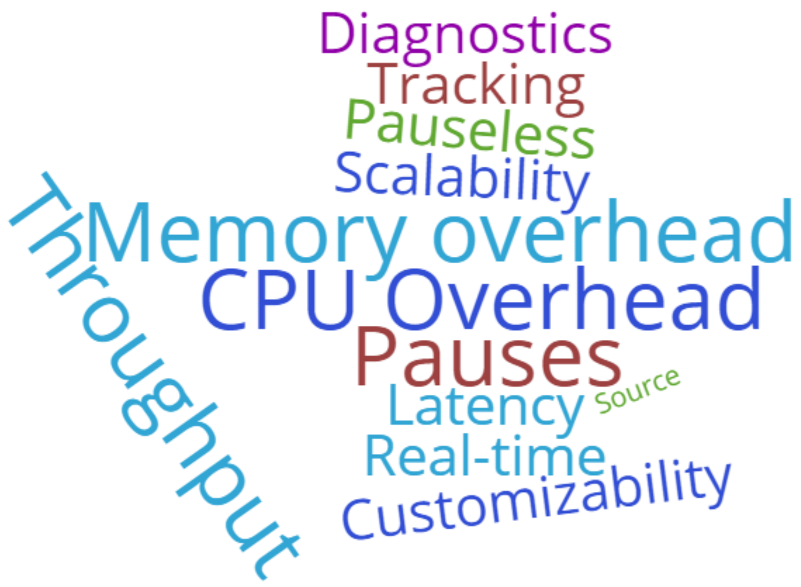
```
-server -Xss4096k -Xms12G -Xmx12G -XX:MaxPermSize=512m
-XX:+HeapDumpOnOutOfMemoryError -verbose:gc -Xmaxf1
-XX:+UseCompressedOops -XX:+DisableExplicitGC -XX:+AggressiveOpts
-XX:+ScavengeBeforeFullGC -XX:CMSFullGCsBeforeCompaction=10
-XX:CMSInitiatingOccupancyFraction=80 -XX:+UseParNewGC
-XX:+UseConcMarkSweepGC -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
```

Cargo cult ~~programming~~ configuring.

But why different/custom GCs at all?!

Jack of all trades is master of none.

Different workloads, different applications, different expectations...


Diagnostics
Tracking
Pauseless
Scalability
Throughput
Memory overhead
CPU Overhead
Pauses
Latency
Source
Real-time
Customizability

"Simple" knobs

"Simple" knobs

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

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.cpp` has <40 kLOC of C++

`.\src\gc\gcsvr.cpp` defines `SERVER_GC` constant and `SVR` namespace:

```
#define SERVER_GC 1
namespace SVR {
#include "gcimpl.h"    // <-- defines MULTIPLE_HEAPS
#include "gc.cpp"
}
```

`.\src\gc\gcwks.cpp` defines `WKS` namespace:

```

namespace WKS {
#include "gcimpl.h"
#include "gc.cpp"
}
```

gc.cpp has <40 kLOC of C++

.\src\gc\gcsvr.cpp defines SERVER_GC constant and SVR namespace:

```
#define SERVER_GC 1
namespace SVR {
#include "gcimpl.h"    // <-- defines MULTIPLE_HEAPS
#include "gc.cpp"
}
```

.\src\gc\gcwks.cpp defines WKS namespace:

```

namespace WKS {
#include "gcimpl.h"
#include "gc.cpp"
}
```

and then the whole gc.cpp begins…

```
heap_segment* gc_heap::get_segment_for_loh (size_t size
  #ifdef MULTIPLE_HEAPS
                                            , gc_heap* hp
#endif //MULTIPLE_HEAPS
                                            )
{
  #ifndef MULTIPLE_HEAPS
    gc_heap* hp = 0;
#endif //MULTIPLE_HEAPS
    heap_segment* res = hp->get_segment (size, TRUE);
```

Non-Concurrent vs. Concurrent Mode

Non-Concurrent

- "stop the world" - all managed threads are suspended
- 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
- no concurrent compacting

- `.\src\gc\gc.cpp` consumes `BACKGROUND_GC` constant
- always defined in both SVR and WKS versions
- dynamic flag checked

```
void GCStatistics::AddGCStats(const gc_mechanisms& settings, size_t timeInMSec)
{
  #ifdef BACKGROUND_GC
     if (settings.concurrent)
    {
        bgc.Accumulate((uint32_t)timeInMSec*1000);
        cntBGC++;
    }
    else if (settings.background_p)
    {
        // ...
```

	Concurrent (false)	**Concurrent (true)**
Workstation	Non-Concurrent Workstation	Background Workstation
Server	Non-Concurrent Server	Background Server

	Concurrent (false)	Concurrent (true)
Workstation	Non-Concurrent Workstation	Background Workstation
Server	Non-Concurrent Server	Background Server

Additional GC knobs:

- `GCNoAffinitize` and `GCHeapAffinitizeMask`:

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

- Latency Modes

- Latency Optimization Goals

  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"*

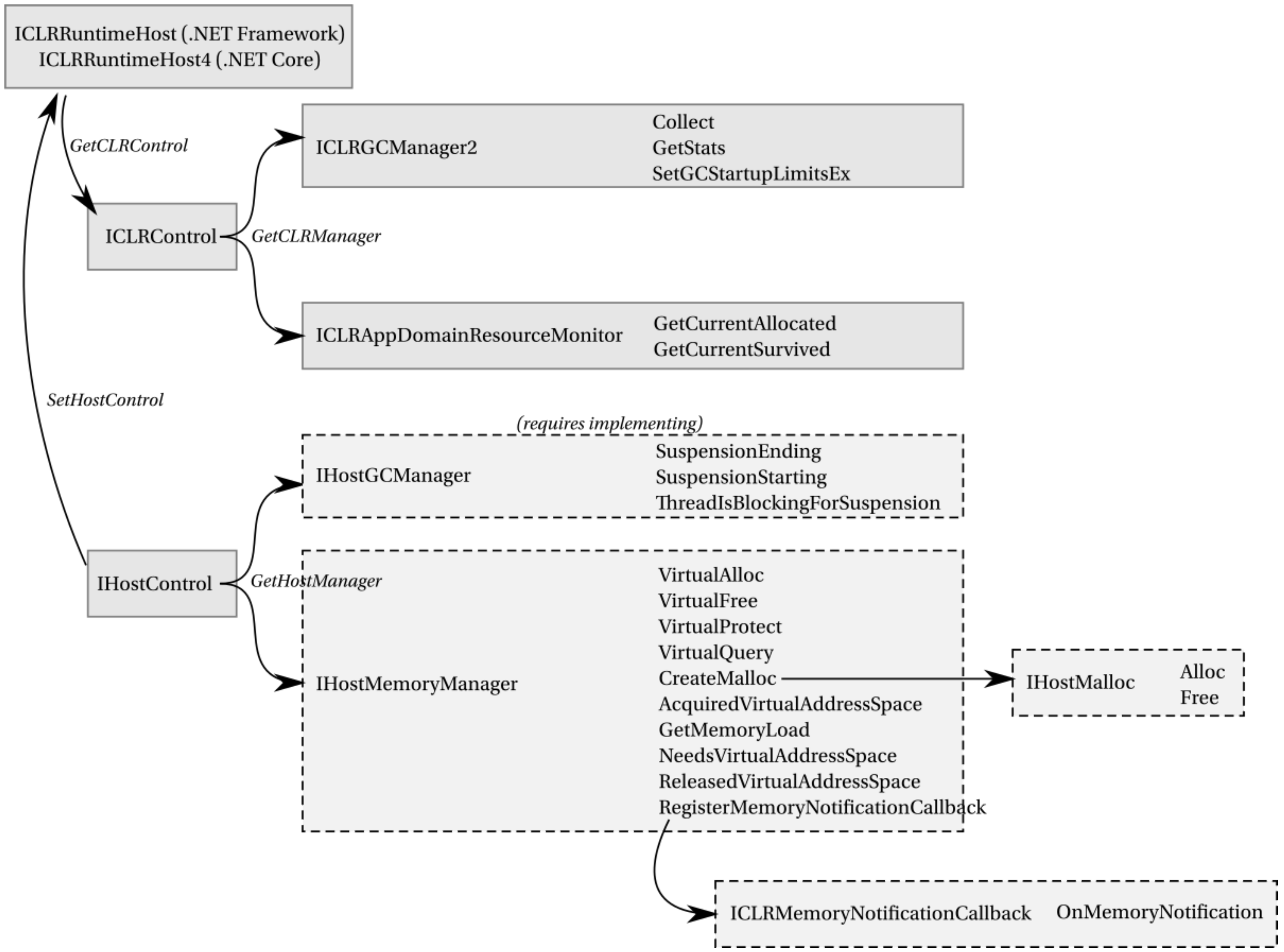
- VM Hoarding

- `GCSettings.LargeObjectHeapCompactionMode`

CLR Hosting

CLR Hosting

- host your own .NET inside a process:
  - to be able to call managed code inside - i.e. SQL Server
  - to customize CLR runtime (including **memory management**)


ICLRRuntimeHost (.NET Framework)
ICLRRuntimeHost4 (.NET Core)
GetCLRControl
ICLRControl
GetCLRManager
ICLRGCManager2
Collect
GetStats
SetGCStartupLimitsEx
ICLRAppDomainResourceMonitor
GetCurrentAllocated
GetCurrentSurvived
SetHostControl
(requires implementing)
IHostGCManager
SuspensionEnding
SuspensionStarting
ThreadIsBlockingForSuspension
IHostControl
GetHostManager
IHostMemoryManager
VirtualAlloc
VirtualFree
VirtualProtect
VirtualQuery
CreateMalloc
AcquiredVirtualAddressSpace
GetMemoryLoad
NeedsVirtualAddressSpace
ReleasedVirtualAddressSpace
RegisterMemoryNotificationCallback
IHostMalloc
Alloc
Free
ICLRMemoryNotificationCallback
OnMemoryNotification

Most interesting for us:

- `ICLRGCManager2`:
  - `SetGCStartupLimitsEx` - sets the size of GC segments and the maximum size of the gen0
- `IHostMemoryManager`:
  - `VirtualAlloc`, `VirtualFree`, `VirtualProtect`, `VirtualQuery` - how CLR operates on **virtual memory**
- `IHostMalloc`:
  - `Alloc`/`DebugAlloc`, `Free` - **native** heap allocations

CLR Hosting 101

```
ICLRRuntimeHost* runtimeHost;
ICLRMetaHost *pMetaHost = nullptr;
ICLRRuntimeInfo *pRuntimeInfo = nullptr;
hr = CLRCreateInstance(CLSID_CLRMetaHost, IID_ICLRMetaHost,
                      (LPVOID*)&pMetaHost);
hr = pMetaHost->GetRuntime(L"v4.0.30319", IID_PPV_ARGS(&pRuntimeInfo));
hr = pRuntimeInfo->GetInterface(CLSID_CLRRuntimeHost, IID_ICLRRuntimeHost,
                                (LPVOID*)&runtimeHost);
ICLRControl* clrControl;
hr = runtimeHost->GetCLRControl(&clrControl);

DWORD dwReturn;
hr = runtimeHost->Start();
hr = runtimeHost->ExecuteInDefaultAppDomain(targetApp,
                                            L"HelloWorld.Program",
                                            L"Test", L"", &dwReturn);
```

CLR Hosting 101

```
ICLRGCManager2* clrGCManager;
hr = clrControl->GetCLRManager(IID_ICLRGCManager2, (void**)&clrGCManager);
SIZE_T segmentSize = 4 * 1024 * 1024 * 1024;
SIZE_T maxGen0Size = 4 * 1024 * 1024 * 1024;
hr = clrGCManager->SetGCStartupLimitsEx(segmentSize, maxGen0Size);
```

CLR Hosting 101

```
CustomHostControl customHostControl;
hr = runtimeHost->SetHostControl(&customHostControl);

...

class CustomHostControl : public IHostControl
{
   virtual HRESULT GetHostManager(REFIID riid, void ** ppObject) override
   {
      if (riid == IID_IHostMemoryManager)
      {
         IHostMemoryManager *pMemoryManager = new CustomHostMemoryManager();
         *ppObject = pMemoryManager;
         return S_OK;
      }
      *ppObject = NULL;
      return E_NOINTERFACE;
   }
   ...
```

I.e. page locking manager:

```
class CustomHostMemoryManager : public IHostMemoryManager
{
   virtual HRESULT VirtualAlloc(void * pAddress, SIZE_T dwSize, DWORD
      flAllocationType, DWORD flProtect, EMemoryCriticalLevel eCriticalLevel,
      void ** ppMem) override
   {
      void* result = ::VirtualAlloc(pAddress,
                                    dwSize,
                                    flAllocationType,
                                    flProtect);
      *ppMem = result;
      BOOL locked = false;
      if (flAllocationType & MEM_COMMIT)
      {
           locked = ::VirtualLock(*ppMem, dwSize);
      }
      return S_OK;
   }
   ...
}
```

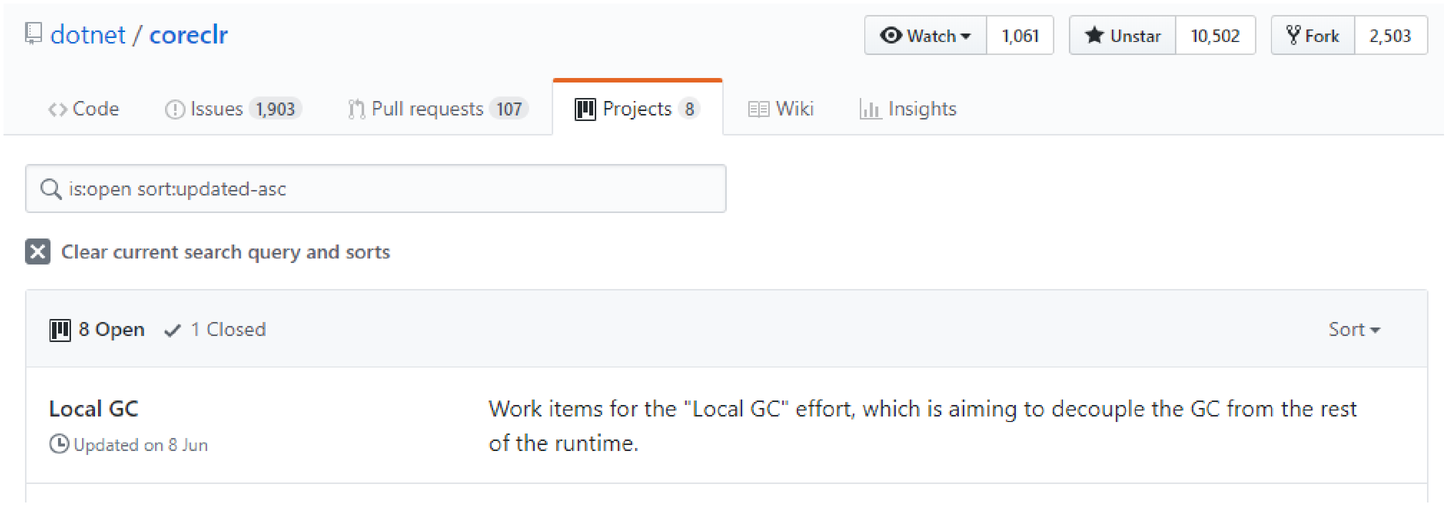
I.e. page locking manager:

```
class CustomHostMemoryManager : public IHostMemoryManager
{
   virtual HRESULT VirtualAlloc(void * pAddress, SIZE_T dwSize, DWORD
      flAllocationType, DWORD flProtect, EMemoryCriticalLevel eCriticalLevel,
      void ** ppMem) override
   {
      void* result = ::VirtualAlloc(pAddress,
                                    dwSize,
                                    flAllocationType,
                                    flProtect);
      *ppMem = result;
      BOOL locked = false;
      if (flAllocationType & MEM_COMMIT)
      {
           locked = ::VirtualLock(*ppMem, dwSize);
      }
      return S_OK;
   }
   ...
}
```

See: Non-paged CLR host project by Sasha Goldshtein and Alon Fliess at https://archive.codeplex.com/?p=nonpagedclrhost

Custom GC

(aka Local GC)


dotnet / coreclr

Watch 1,061 | Unstar 10,502 | Fork 2,503

Code | Issues 1,903 | Pull requests 107 | Projects 8 | Wiki | Insights

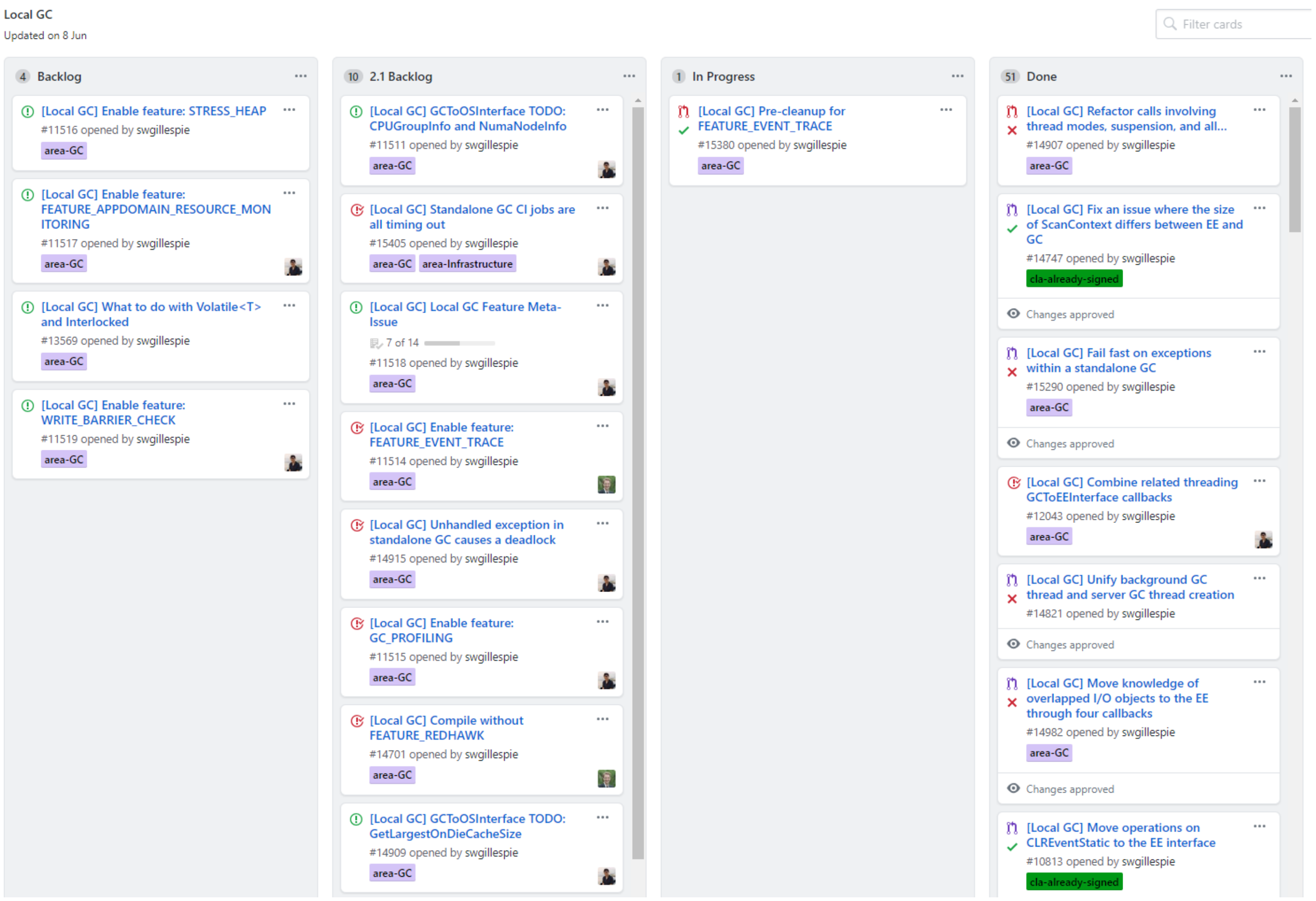
is:open sort:updated-asc

Clear current search query and sorts

8 Open ✓ 1 Closed Sort

Local GC
Updated on 8 Jun

Work items for the "Local GC" effort, which is aiming to decouple the GC from the rest of the runtime.



Local GC
Updated on 8 Jun

Filter cards

4 Backlog

[Local GC] Enable feature: STRESS_HEAP
#11516 opened by swgillespie
area-GC

[Local GC] Enable feature: FEATURE_APPDOMAIN_RESOURCE_MONITORING
#11517 opened by swgillespie
area-GC

[Local GC] What to do with Volatile<T> and Interlocked
#13569 opened by swgillespie
area-GC

[Local GC] Enable feature: WRITE_BARRIER_CHECK
#11519 opened by swgillespie
area-GC

10 2.1 Backlog

[Local GC] GCToOSInterface TODO: CPUGroupInfo and NumaNodeInfo
#11511 opened by swgillespie
area-GC

[Local GC] Standalone GC CI jobs are all timing out
#15405 opened by swgillespie
area-GC area-Infrastructure

[Local GC] Local GC Feature Meta-Issue
7 of 14
#11518 opened by swgillespie
area-GC

[Local GC] Enable feature: FEATURE_EVENT_TRACE
#11514 opened by swgillespie
area-GC

[Local GC] Unhandled exception in standalone GC causes a deadlock
#14915 opened by swgillespie
area-GC

[Local GC] Enable feature: GC_PROFILING
#11515 opened by swgillespie
area-GC

[Local GC] Compile without FEATURE_REDHAWK
#14701 opened by swgillespie
area-GC

[Local GC] GCToOSInterface TODO: GetLargestOnDieCacheSize
#14909 opened by swgillespie
area-GC

1 In Progress

[Local GC] Pre-cleanup for FEATURE_EVENT_TRACE
#15380 opened by swgillespie
area-GC

51 Done

[Local GC] Refactor calls involving thread modes, suspension, and all...
#14907 opened by swgillespie
area-GC

[Local GC] Fix an issue where the size of ScanContext differs between EE and GC
#14747 opened by swgillespie
cla-already-signed
Changes approved

[Local GC] Fail fast on exceptions within a standalone GC
#15290 opened by swgillespie
area-GC
Changes approved

[Local GC] Combine related threading GCToEEInterface callbacks
#12043 opened by swgillespie
area-GC

[Local GC] Unify background GC thread and server GC thread creation
#14821 opened by swgillespie
Changes approved

[Local GC] Move knowledge of overlapped I/O objects to the EE through four callbacks
#14982 opened by swgillespie
area-GC
Changes approved

[Local GC] Move operations on CLREventStatic to the EE interface
#10813 opened by swgillespie
cla-already-signed

What can be done with it?

What can be done with it?

Everything!

What can be done with it?

Everything!

Well... almost

Usage

Since .NET Core 2.1:

- `set COMPlus_GCName=f:\CoreCLR.ZeroGC\x64\Release\ZeroGC.dll`

In .NET Core 2.0 (preview):

- additionally required recompiling runtime with `FEATURE_STANDALONE_GC` feature enabled:

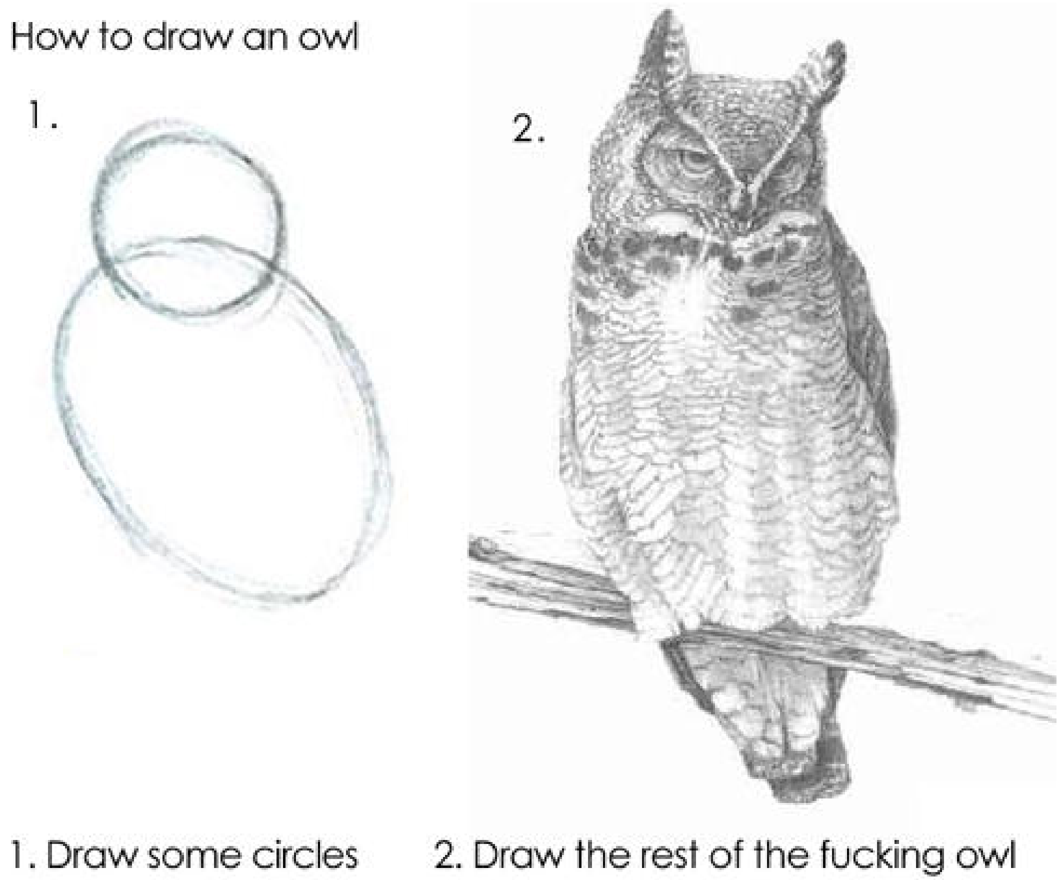
  ```
  > build.cmd -buildstandalonegc
  ```

Implementing

- regular C++ library (i.e. created in Visual Studio)
- include only three files from CoreCLR:

```
#include "debugmacros.h"
#include "gcenv.base.h"
#include "gcinterface.h"
```

- implement two exported simple methods
  - `GC_Initialize`
  - `GC_VersionInfo`
- implement the rest of the GC:
  - `IGCHeap` - responsible for… everything
  - `IGCHandleManager` and `IGCHandleStore` - responsible for handling… handles


How to draw an owl
1.
2.
1. Draw some circles
2. Draw the rest of the fucking owl

Is it difficult?

Is it difficult?

No but it requires very deep knowledge about the runtime and... the GC

Implementing - cont.

```
extern "C" DLLEXPORT void
GC_VersionInfo(
/* Out */ VersionInfo* result
)
{
   result->MajorVersion = GC_INTERFACE_MAJOR_VERSION;
   result->MinorVersion = GC_INTERFACE_MINOR_VERSION;
   result->BuildVersion = 0;
}
```

Specifying which GC API version our custom GC supports.

Implementing - cont.

```
extern "C" DLLEXPORT HRESULT
GC_Initialize(
   /* In */ IGCToCLR* clrToGC,
   /* Out */ IGCHeap** gcHeap,
   /* Out */ IGCHandleManager** gcHandleManager,
   /* Out */ GcDacVars* gcDacVars
)
{
   IGCHeap* heap = new ZeroGCHeap(clrToGC);
   IGCHandleManager* handleManager = new ZeroGCHandleManager();
   *gcHeap = heap;
   *gcHandleManager = handleManager;
   return S_OK;
}
```

Implementing - cont.

```
extern "C" DLLEXPORT HRESULT
GC_Initialize(
   /* In */ IGCToCLR* clrToGC,
   /* Out */ IGCHeap** gcHeap,
   /* Out */ IGCHandleManager** gcHandleManager,
   /* Out */ GcDacVars* gcDacVars
)
{
   IGCHeap* heap = new ZeroGCHeap(clrToGC);
   IGCHandleManager* handleManager = new ZeroGCHandleManager();
   *gcHeap = heap;
   *gcHandleManager = handleManager;
   return S_OK;
}
```

Specifying pointers to our custom `IGCHeap` and `IGCHandleManager` implementations.

Implementing - cont.

```
extern "C" DLLEXPORT HRESULT
GC_Initialize(
   /* In */ IGCToCLR* clrToGC,
   /* Out */ IGCHeap** gcHeap,
   /* Out */ IGCHandleManager** gcHandleManager,
   /* Out */ GcDacVars* gcDacVars
)
{
   IGCHeap* heap = new ZeroGCHeap(clrToGC);
   IGCHandleManager* handleManager = new ZeroGCHandleManager();
   *gcHeap = heap;
   *gcHandleManager = handleManager;
   return S_OK;
}
```

Remembering `IGCToCLR` as it provides so convenient API as:

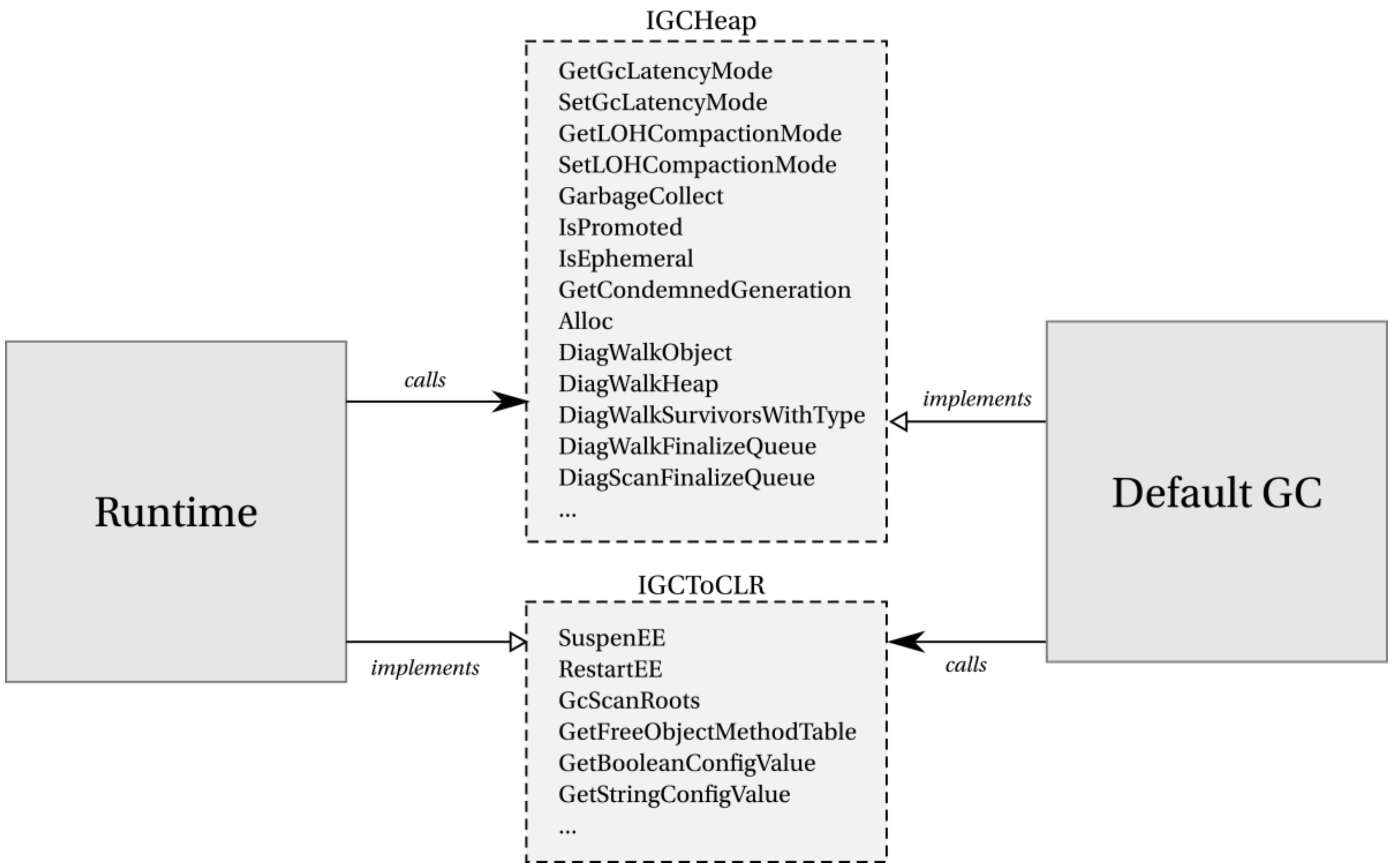
- `SuspendEE` and `RestartEE` methods for thread suspensions
- `GcScanRoots` for methods root scanning
- `GcStartWork` and `GcDone` to inform the runtime

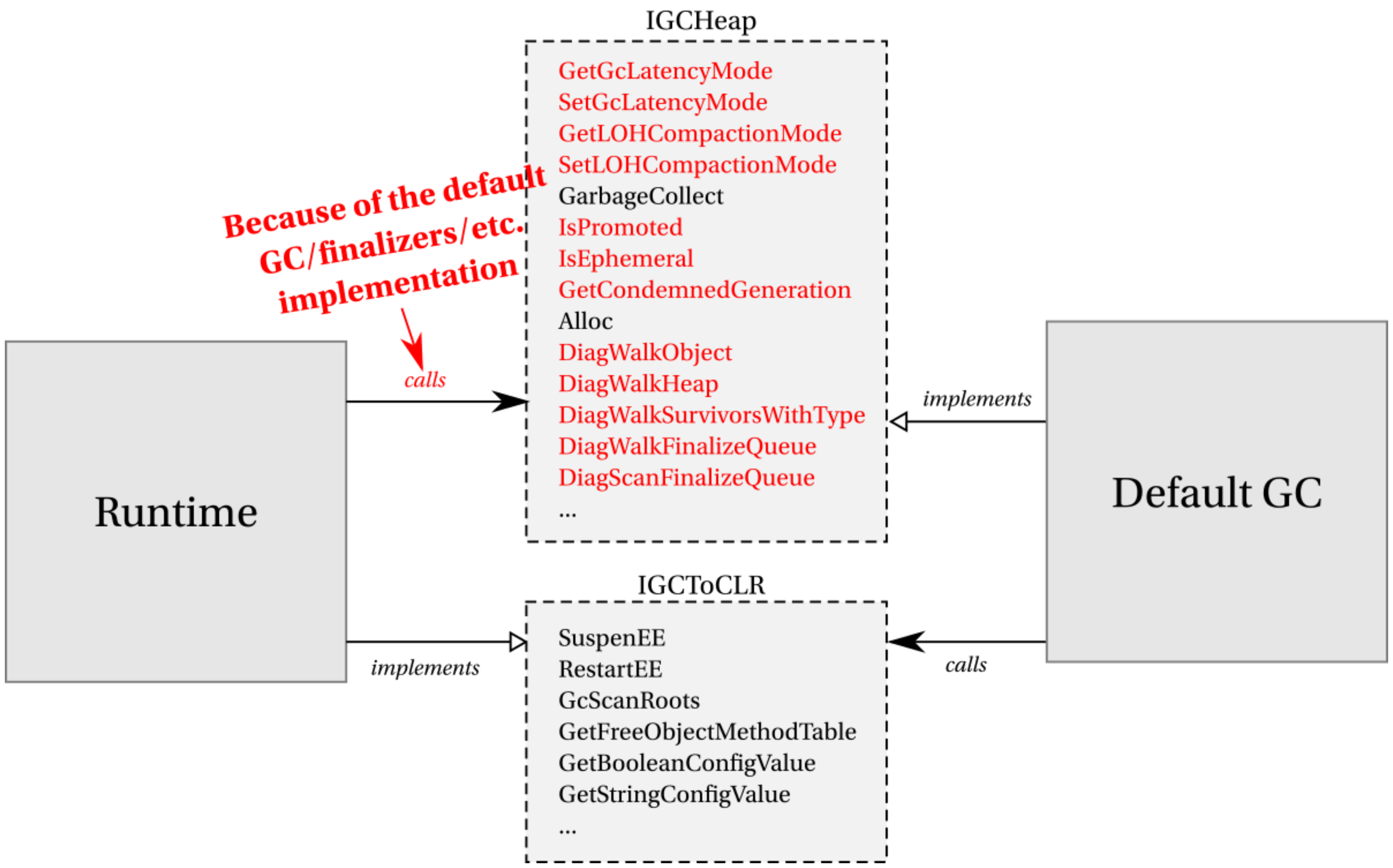
IGCHeap

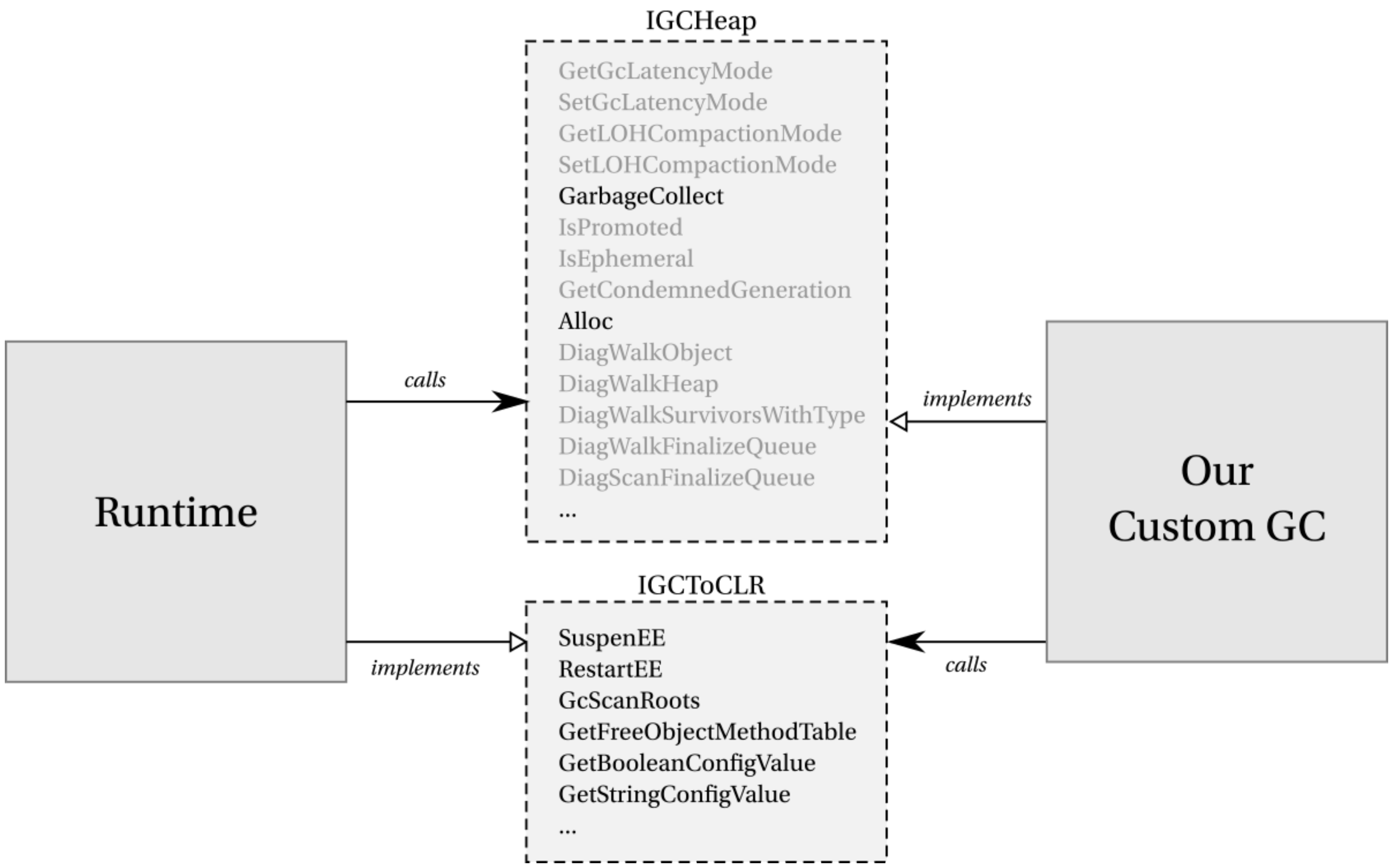
```
class ZeroGCHeap : public IGCHeap
{
private:
   IGCToCLR* gcToCLR;
public:
   ZeroGCHeap(IGCToCLR* gcToCLR)
   {
      this->gcToCLR = gcToCLR;
   }
   // Inherited via IGCHeap
   ...
   75 methods!
}
```

```
    // Inherited via IGCHeap
    virtual bool IsValidSegmentSize(size_t size) override;
    virtual bool IsValidGen0MaxSize(size_t size) override;
    virtual size_t GetValidSegmentSize(bool large_seg = false) override;
    virtual void SetReservedVMLimit(size_t vmlimit) override;
    virtual void WaitUntilConcurrentGCComplete() override;
    virtual bool IsConcurrentGCInProgress() override;
    virtual void TemporaryEnableConcurrentGC() override;
    virtual void TemporaryDisableConcurrentGC() override;
    virtual bool IsConcurrentGCEnabled() override;
    virtual HRESULT WaitUntilConcurrentGCCompleteAsync(int millisecondsTimeout) ov
    virtual bool FinalizeAppDomain(void* pDomain, bool fRunFinalizers) override;
    virtual void SetFinalizeQueueForShutdown(bool fHasLock) override;
    virtual size_t GetNumberOfFinalizable() override;
    virtual bool ShouldRestartFinalizerWatchDog() override;
    virtual Object* GetNextFinalizable() override;
    virtual void SetFinalizeRunOnShutdown(bool value) override;
    virtual int GetGcLatencyMode() override;
    virtual int SetGcLatencyMode(int newLatencyMode) override;
    virtual int GetLOHCompactionMode() override;
    virtual void SetLOHCompactionMode(int newLOHCompactionMode) override;
    virtual bool RegisterForFullGCNotification(uint32_t gen2Percentage, uint32_t l
    virtual bool CancelFullGCNotification() override;
    virtual int WaitForFullGCApproach(int millisecondsTimeout) override;
    virtual int WaitForFullGCComplete(int millisecondsTimeout) override;
    virtual unsigned WhichGeneration(Object* obj) override;
    virtual int CollectionCount(int generation, int get_bgc_fgc_coutn = 0) overrid
    virtual int StartNoGCRegion(uint64_t totalSize, bool lohSizeKnown, uint64_t lo
    virtual int EndNoGCRegion() override;
    virtual size_t GetTotalBytesInUse() override;
    virtual HRESULT GarbageCollect(int generation = -1, bool low_memory_p = false,
    virtual unsigned GetMaxGeneration() override;
    virtual void SetFinalizationRun(Object* obj) override;
    // Inherited via IGCHeap
    virtual bool IsValidSegmentSize(size_t size) override;
    virtual bool IsValidGen0MaxSize(size_t size) override;
    virtual size_t GetValidSegmentSize(bool large_seg = false) override;
    virtual void SetReservedVMLimit(size_t vmlimit) override;
    virtual void WaitUntilConcurrentGCComplete() override;
    virtual bool IsConcurrentGCInProgress() override;
    virtual void TemporaryEnableConcurrentGC() override;
    virtual void TemporaryDisableConcurrentGC() override;
    virtual bool IsConcurrentGCEnabled() override;
    virtual HRESULT WaitUntilConcurrentGCCompleteAsync(int millisecondsTimeout) ov
    virtual bool FinalizeAppDomain(void* pDomain, bool fRunFinalizers) override;
    virtual void SetFinalizeQueueForShutdown(bool fHasLock) override;
    virtual size_t GetNumberOfFinalizable() override;
    virtual bool ShouldRestartFinalizerWatchDog() override;
    virtual Object* GetNextFinalizable() override;
    virtual void SetFinalizeRunOnShutdown(bool value) override;
    virtual int GetGcLatencyMode() override;
    virtual int SetGcLatencyMode(int newLatencyMode) override;
    virtual int GetLOHCompactionMode() override;
    virtual void SetLOHCompactionMode(int newLOHCompactionMode) override;
    virtual bool RegisterForFullGCNotification(uint32_t gen2Percentage, uint32_t l
    virtual bool CancelFullGCNotification() override;
    virtual int WaitForFullGCApproach(int millisecondsTimeout) override;
    virtual int WaitForFullGCComplete(int millisecondsTimeout) override;
    virtual unsigned WhichGeneration(Object* obj) override;
    virtual int CollectionCount(int generation, int get_bgc_fgc_coutn = 0) overrid
    virtual int StartNoGCRegion(uint64_t totalSize, bool lohSizeKnown, uint64_t lo
    virtual int EndNoGCRegion() override;
    virtual size_t GetTotalBytesInUse() override;
    virtual HRESULT GarbageCollect(int generation = -1, bool low_memory_p = false,
    virtual unsigned GetMaxGeneration() override;
    virtual void SetFinalizationRun(Object* obj) override;
    virtual bool RegisterForFinalization(int gen, Object* obj) override;
    virtual HRESULT Initialize() override;
    virtual bool IsPromoted(Object* object) override;
    virtual bool IsHeapPointer(void* object, bool small_heap_only = false) overrid
    virtual unsigned GetCondemnedGeneration() override;
    virtual bool IsGCInProgressHelper(bool bConsiderGCStart = false) override;
    virtual unsigned GetGcCount() override;
    virtual bool IsThreadUsingAllocationContextHeap(gc_alloc_context* acontext, in
    virtual bool IsEphemeral(Object* object) override;
    virtual uint32_t WaitUntilGCComplete(bool bConsiderGCStart = false) override;
    virtual void FixAllocContext(gc_alloc_context* acontext, bool lockp, void* arg
    virtual size_t GetCurrentObjSize() override;
    virtual void SetGCInProgress(bool fInProgress) override;
    virtual bool RuntimeStructuresValid() override;
    virtual size_t GetLastGCStartTime(int generation) override;
    virtual size_t GetLastGCDuration(int generation) override;
    virtual size_t GetNow() override;
    virtual Object* Alloc(gc_alloc_context* acontext, size_t size, uint32_t flags)
    virtual Object* AllocLHeap(size_t size, uint32_t flags) override;
    virtual Object* AllocAlign8(gc_alloc_context* acontext, size_t size, uint32_t
    virtual void PublishObject(uint8_t* obj) override;
    virtual void SetWaitForGCEvent() override;
    virtual void ResetWaitForGCEvent() override;
    virtual bool IsObjectInFixedHeap(Object* pObj) override;
    virtual void ValidateObjectMember(Object* obj) override;
    virtual Object* NextObj(Object* object) override;
    virtual Object* GetContainingObject(void* pInteriorPtr, bool fCollectedGenOnly
    virtual void DiagWalkObject(Object* obj, walk_fn fn, void* context) override;
    virtual void DiagWalkHeap(walk_fn fn, void* context, int gen_number, bool walk
    virtual void DiagWalkSurvivorsWithType(void* gc_context, record_surv_fn fn, vo
    virtual void DiagWalkFinalizeQueue(void* gc_context, fq_walk_fn fn) override;
    virtual void DiagScanFinalizeQueue(fq_scan_fn fn, ScanContext* context) overri
    virtual bool RegisterForFinalization(int gen, Object* obj) override;
    virtual HRESULT Initialize() override;
    virtual bool IsPromoted(Object* object) override;
    virtual bool IsHeapPointer(void* object, bool small_heap_only = false) overrid
    virtual unsigned GetCondemnedGeneration() override;
    virtual bool IsGCInProgressHelper(bool bConsiderGCStart = false) override;
    virtual unsigned GetGcCount() override;
    virtual bool IsThreadUsingAllocationContextHeap(gc_alloc_context* acontext, in
    virtual bool IsEphemeral(Object* object) override;
    virtual uint32_t WaitUntilGCComplete(bool bConsiderGCStart = false) override;
    virtual void FixAllocContext(gc_alloc_context* acontext, bool lockp, void* arg
    virtual size_t GetCurrentObjSize() override;
    virtual void SetGCInProgress(bool fInProgress) override;
    virtual bool RuntimeStructuresValid() override;
    virtual size_t GetLastGCStartTime(int generation) override;
    virtual size_t GetLastGCDuration(int generation) override;
    virtual size_t GetNow() override;
    virtual Object* Alloc(gc_alloc_context* acontext, size_t size, uint32_t flags)
    virtual Object* AllocLHeap(size_t size, uint32_t flags) override;
    virtual Object* AllocAlign8(gc_alloc_context* acontext, size_t size, uint32_t
    virtual void PublishObject(uint8_t* obj) override;
    virtual void SetWaitForGCEvent() override;
    virtual void ResetWaitForGCEvent() override;
    virtual bool IsObjectInFixedHeap(Object* pObj) override;
    virtual void ValidateObjectMember(Object* obj) override;
    virtual Object* NextObj(Object* object) override;
    virtual Object* GetContainingObject(void* pInteriorPtr, bool fCollectedGenOnly
    virtual void DiagWalkObject(Object* obj, walk_fn fn, void* context) override;
    virtual void DiagWalkHeap(walk_fn fn, void* context, int gen_number, bool walk
    virtual void DiagWalkSurvivorsWithType(void* gc_context, record_surv_fn fn, vo
    virtual void DiagWalkFinalizeQueue(void* gc_context, fq_walk_fn fn) override;
    virtual void DiagScanFinalizeQueue(fq_scan_fn fn, ScanContext* context) overri
    virtual void DiagScanHandles(handle_scan_fn fn, int gen_number, ScanContext* c
    virtual void DiagScanDependentHandles(handle_scan_fn fn, int gen_number, ScanC
    virtual void DiagDescrGenerations(gen_walk_fn fn, void* context) override;
    virtual void DiagTraceGCSegments() override;
    virtual bool StressHeap(gc_alloc_context* acontext) override;
    virtual segment_handle RegisterFrozenSegment(segment_info *pseginfo) override;
    virtual void UnregisterFrozenSegment(segment_handle seg) override;
    virtual void ControlEvents(GCEventKeyword keyword, GCEventLevel level) overrid
    virtual void ControlPrivateEvents(GCEventKeyword keyword, GCEventLevel level)
    virtual void GetMemoryInfo(uint32_t * highMemLoadThreshold, uint64_t * totalPh
    virtual void SetSuspensionPending(bool fSuspensionPending) override;
    virtual void SetYieldProcessorScalingFactor(uint32_t yieldProcessorScalingFact
```

So, what we MUST implement?


IGCHeap
GetGcLatencyMode
SetGcLatencyMode
GetLOHCompactionMode
SetLOHCompactionMode
GarbageCollect
IsPromoted
IsEphemeral
GetCondemnedGeneration
Alloc
DiagWalkObject
DiagWalkHeap
DiagWalkSurvivorsWithType
DiagWalkFinalizeQueue
DiagScanFinalizeQueue
...
Runtime
calls
implements
Default GC
implements
calls
IGCToCLR
SuspenEE
RestartEE
GcScanRoots
GetFreeObjectMethodTable
GetBooleanConfigValue
GetStringConfigValue
...


IGCHeap
GetGcLatencyMode
SetGcLatencyMode
GetLOHCompactionMode
SetLOHCompactionMode
GarbageCollect
IsPromoted
IsEphemeral
GetCondemnedGeneration
Alloc
DiagWalkObject
DiagWalkHeap
DiagWalkSurvivorsWithType
DiagWalkFinalizeQueue
DiagScanFinalizeQueue
...
Because of the default GC/finalizers/etc. implementation
calls
Runtime
implements
Default GC
IGCToCLR
SuspenEE
RestartEE
GcScanRoots
GetFreeObjectMethodTable
GetBooleanConfigValue
GetStringConfigValue
...
implements
calls


IGCHeap
GetGcLatencyMode
SetGcLatencyMode
GetLOHCompactionMode
SetLOHCompactionMode
GarbageCollect
IsPromoted
IsEphemeral
GetCondemnedGeneration
Alloc
DiagWalkObject
DiagWalkHeap
DiagWalkSurvivorsWithType
DiagWalkFinalizeQueue
DiagScanFinalizeQueue
...
Runtime
calls
implements
Our Custom GC
implements
calls
IGCToCLR
SuspenEE
RestartEE
GcScanRoots
GetFreeObjectMethodTable
GetBooleanConfigValue
GetStringConfigValue
...

Let's write Minimum Valuable Product - Zero GC

- only allocating
- no Garbage Collection at all

Zero GC

Most `IGCHeap` methods may be dummy:

```
bool CustomGCHeap::RuntimeStructuresValid()
{
    return true;
}

bool ZeroGCHeap::IsPromoted(Object * object)
{
    return false;
}

unsigned ZeroGCHeap::GetCondemnedGeneration()
{
    return 0;
}
```

Zero GC

`IGCHeap::GarbageCollect`

- called by the runtime in rare cases:
  - `GC.Collect`
  - low-memory notification
- not called by the GC itself

Zero GC

`IGCHeap::GarbageCollect`

- called by the runtime in rare cases:
    - `GC.Collect`
    - low-memory notification
- not called by the GC itself

Trivial implementation:

```
HRESULT ZeroGCHeap::GarbageCollect(int generation, bool low_memory_p, int mode)
{
    return NOERROR;
}
```

Zero GC

`IGCHeap` - allocations:

```
Object* ZeroGCHeap::Alloc(gc_alloc_context * acontext, size_t size, uint32_t flags
{
    // return address of a new object
    // trigger GC if necessary
}

Object* ZeroGCHeap::AllocLHeap(size_t size, uint32_t flags)
{
    // return address of a new object
    // trigger GC if necessary
}
```

Zero GC

`IGCHeap` - allocations:

```
Object* ZeroGCHeap::Alloc(gc_alloc_context * acontext, size_t size, uint32_t flags
{
    // return address of a new object
    // trigger GC if necessary
}

Object* ZeroGCHeap::AllocLHeap(size_t size, uint32_t flags)
{
    // return address of a new object
    // trigger GC if necessary
}
```

Zero GC

`IGCHeap` - allocations:

```
Object* ZeroGCHeap::Alloc(gc_alloc_context * acontext, size_t size, uint32_t flags
{
    // return address of a new object
}

Object* ZeroGCHeap::AllocLHeap(size_t size, uint32_t flags)
{
    // return address of a new object
}
```

Zero GC

`IGCHeap` - allocations:

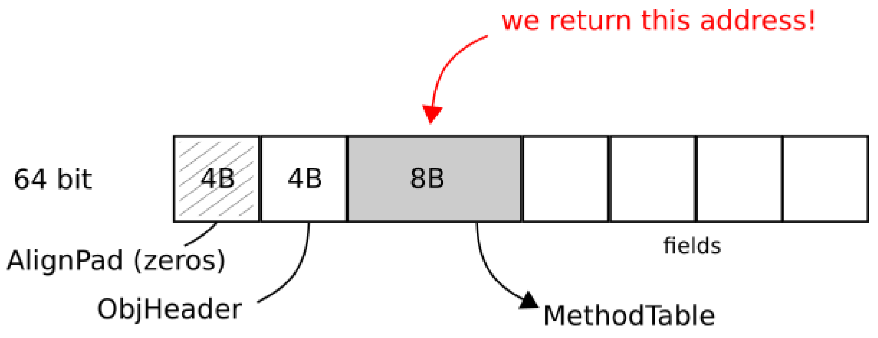
```
Object* ZeroGCHeap::Alloc(gc_alloc_context * acontext, size_t size, uint32_t flags
{
    int sizeWithHeader = size + sizeof(ObjHeader);
    ObjHeader* address = (ObjHeader*)calloc(sizeWithHeader, sizeof(char));
    return (Object*)(address + 1);
}

Object* ZeroGCHeap::AllocLHeap(size_t size, uint32_t flags)
{
    int sizeWithHeader = size + sizeof(ObjHeader);
    ObjHeader* address = (ObjHeader*)calloc(sizeWithHeader, sizeof(char));
    return (Object*)(address + 1);
}
```

Zero GC

`IGCHeap` - allocations:

```
Object* ZeroGCHeap::Alloc(gc_alloc_context * acontext, size_t size, uint32_t flags
{
    int sizeWithHeader = size + sizeof(ObjHeader);
    ObjHeader* address = (ObjHeader*)calloc(sizeWithHeader, sizeof(char));
    return (Object*)(address + 1);
}
```



Zero GC

`IGCHeap` - creating handles (pinning, strong, ...):

```
bool ZeroGCHandleManager::Initialize()
{
    g_gcGlobalHandleStore = new ZeroGCHandleStore();
    return true;
}

OBJECTHANDLE
ZeroGCHandleManager::CreateGlobalHandleOfType(Object * object, HandleType type)
{
    return g_gcGlobalHandleStore->CreateHandleOfType(object, type);
}
```

```
int handlesCount = 0;
OBJECTHANDLE handles[65535];

OBJECTHANDLE
ZeroGCHandleStore::CreateHandleOfType(Object * object, HandleType type)
{
    handles[handlesCount] = (OBJECTHANDLE__*)object;
    return (OBJECTHANDLE)&handles[handlesCount++];
}
```

Zero GC

`IGCHandleManager` - storing handles:

```
void
ZeroGCHandleManager::StoreObjectInHandle(OBJECTHANDLE handle, Object * object)
{
    Object** handleObj = (Object**)handle;
    *handleObj = object;
}

bool
ZeroGCHandleManager::StoreObjectInHandleIfNull(OBJECTHANDLE handle, Object* object
{
    Object** handleObj = (Object**)handle;
    if (*handleObj == NULL)
    {
        *handleObj = object;
        return true;
    }
    return false;
}
```

And that's mostly all!

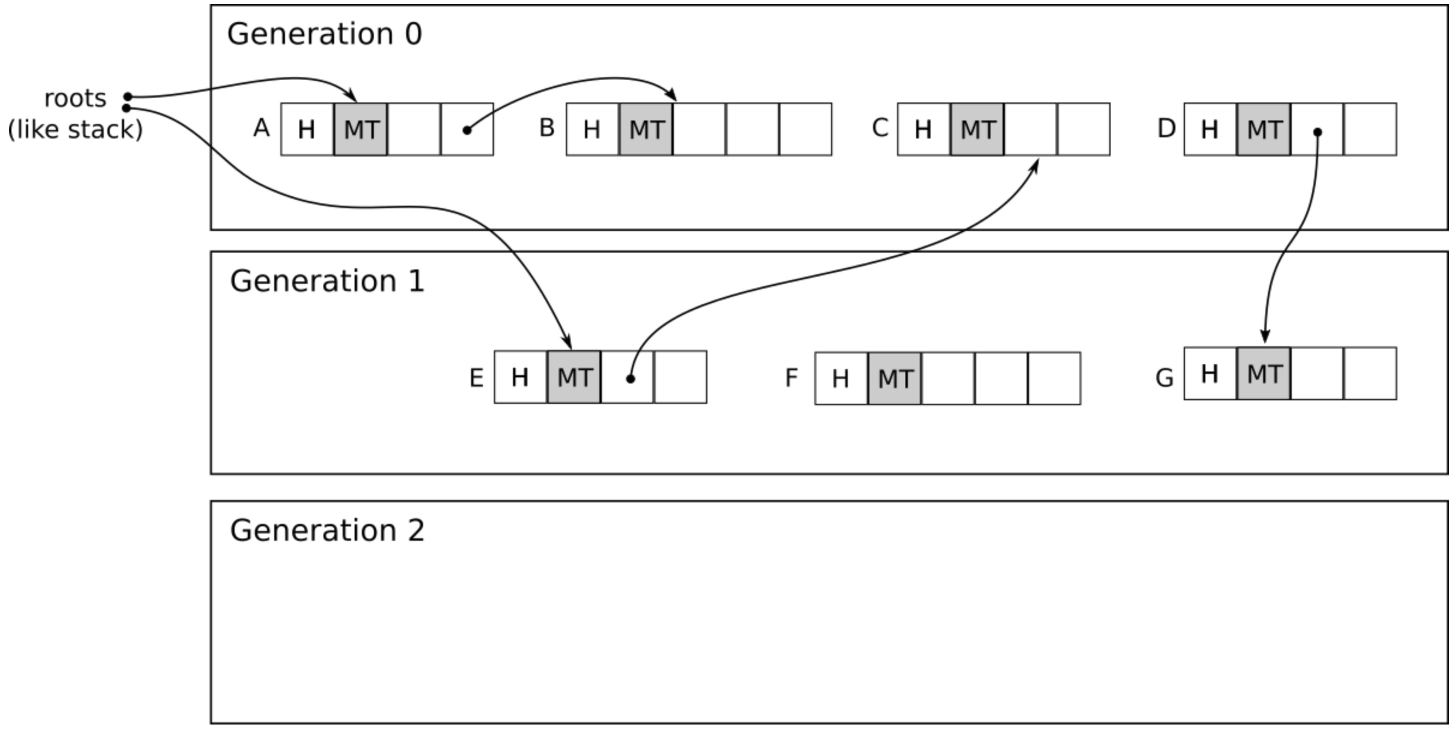
Complete *Calloc-based* implementation:

https://github.com/kkokosa/CoreCLR.ZeroGC

"Mostly"

Caveat #1 - write barriers

Remembered sets (card tables)



```
LEAF_ENTRY JIT_WriteBarrier_PostGrow64, _TEXT
        align 8
        mov     [rcx], rdx
        NOP_3_BYTE ; padding for alignment of constant
PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Lower
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Check the lower and upper ephemeral region bounds
        cmp     rdx, rax
        jb      Exit
        nop ; padding for alignment of constant

PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Upper
        mov     r8, 0F0F0F0F0F0F0F0F0h
        cmp     rdx, r8
        jae     Exit
        nop ; padding for alignment of constant

PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_CardTable
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Touch the card table entry, if not already dirty.
        shr     rcx, 0Bh
        cmp     byte ptr [rcx + rax], 0FFh
        jne     UpdateCardTable
        REPRET
    UpdateCardTable:
        mov     byte ptr [rcx + rax], 0FFh
        ret
    align 16
    Exit:
        REPRET
LEAF_END_MARKED JIT_WriteBarrier_PostGrow64, _TEXT
```

```
LEAF_ENTRY JIT_WriteBarrier_PostGrow64, _TEXT
        align 8
        mov     [rcx], rdx
        NOP_3_BYTE ; padding for alignment of constant
PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Lower
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Check the lower and upper ephemeral region bounds
        cmp     rdx, rax
        jb      Exit
        nop ; padding for alignment of constant

PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Upper
        mov     r8, 0F0F0F0F0F0F0F0F0h
        cmp     rdx, r8
        jae     Exit
        nop ; padding for alignment of constant

PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_CardTable
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Touch the card table entry, if not already dirty.
        shr     rcx, 0Bh
        cmp     byte ptr [rcx + rax], 0FFh
        jne     UpdateCardTable
        REPRET
    UpdateCardTable:
        mov     byte ptr [rcx + rax], 0FFh
        ret
    align 16
    Exit:
        REPRET
LEAF_END_MARKED JIT_WriteBarrier_PostGrow64, _TEXT
```

```
LEAF_ENTRY JIT_WriteBarrier_PostGrow64, _TEXT
        align 8
        mov     [rcx], rdx
        NOP_3_BYTE ; padding for alignment of constant
PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Lower
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Check the lower and upper ephemeral region bounds
        cmp     rdx, rax
        jb      Exit
        nop ; padding for alignment of constant

PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Upper
        mov     r8, 0F0F0F0F0F0F0F0F0h
        cmp     rdx, r8
        jae     Exit
        nop ; padding for alignment of constant

PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_CardTable
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Touch the card table entry, if not already dirty.
        shr     rcx, 0Bh
        cmp     byte ptr [rcx + rax], 0FFh
        jne     UpdateCardTable
        REPRET
    UpdateCardTable:
        mov     byte ptr [rcx + rax], 0FFh
        ret
    align 16
    Exit:
        REPRET
LEAF_END_MARKED JIT_WriteBarrier_PostGrow64, _TEXT
```

```
LEAF_ENTRY JIT_WriteBarrier_PostGrow64, _TEXT
        align 8
        mov     [rcx], rdx
        NOP_3_BYTE ; padding for alignment of constant
PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Lower
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Check the lower and upper ephemeral region bounds
        cmp     rdx, rax
        jb      Exit
        nop ; padding for alignment of constant

PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Upper
        mov     r8, 0F0F0F0F0F0F0F0F0h
        cmp     rdx, r8
        jae     Exit
        nop ; padding for alignment of constant

PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_CardTable
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Touch the card table entry, if not already dirty.
        shr     rcx, 0Bh
        cmp     byte ptr [rcx + rax], 0FFh
        jne     UpdateCardTable
        REPRET
    UpdateCardTable:
        mov     byte ptr [rcx + rax], 0FFh
        ret
    align 16
    Exit:
        REPRET
LEAF_END_MARKED JIT_WriteBarrier_PostGrow64, _TEXT
```

Zero GC

`IGCHeap` - fooling write barriers:

```
HRESULT ZeroGCHeap::Initialize()
{
    // Not used currently
    MethodTable* freeObjectMethodTable = gcToCLR->GetFreeObjectMethodTable();

    WriteBarrierParameters args = {};
    args.operation = WriteBarrierOp::Initialize;
    args.is_runtime_suspended = true;
    args.requires_upper_bounds_check = false;
    args.card_table = new uint32_t[1];
    args.lowest_address = reinterpret_cast<uint8_t*>(~0);;
    args.highest_address = reinterpret_cast<uint8_t*>(1);
    args.ephemeral_low = reinterpret_cast<uint8_t*>(~0);
    args.ephemeral_high = reinterpret_cast<uint8_t*>(1);
    gcToCLR->StompWriteBarrier(&args);

    return NOERROR;
}
```

Zero GC

`IGCHeap` - fooling write barriers:

```
HRESULT ZeroGCHeap::Initialize()
{
    // Not used currently
    MethodTable* freeObjectMethodTable = gcToCLR->GetFreeObjectMethodTable();

    WriteBarrierParameters args = {};
    args.operation = WriteBarrierOp::Initialize;
    args.is_runtime_suspended = true;
    args.requires_upper_bounds_check = false;
    args.card_table = new uint32_t[1];
    args.lowest_address = reinterpret_cast<uint8_t*>(~0);;
    args.highest_address = reinterpret_cast<uint8_t*>(1);
    args.ephemeral_low = reinterpret_cast<uint8_t*>(~0);
    args.ephemeral_high = reinterpret_cast<uint8_t*>(1);
    gcToCLR->StompWriteBarrier(&args);

    return NOERROR;
}
```

Zero GC

`IGCHeap` - fooling write barriers:

```
HRESULT ZeroGCHeap::Initialize()
{
    // Not used currently
    MethodTable* freeObjectMethodTable = gcToCLR->GetFreeObjectMethodTable();

    WriteBarrierParameters args = {};
    args.operation = WriteBarrierOp::Initialize;
    args.is_runtime_suspended = true;
    args.requires_upper_bounds_check = false;
    args.card_table = new uint32_t[1];
    args.lowest_address = reinterpret_cast<uint8_t*>(~0);;
    args.highest_address = reinterpret_cast<uint8_t*>(1);
    args.ephemeral_low = reinterpret_cast<uint8_t*>(~0);
    args.ephemeral_high = reinterpret_cast<uint8_t*>(1);
    gcToCLR->StompWriteBarrier(&args);

    return NOERROR;
}
```

Still:

- requires Workstation GC mode - Server GC injects `JIT_WriteBarrier_SVR64` that omits ephemeral checks and crashes the runtime :(

Zero GC - *Calloc-based* - applied

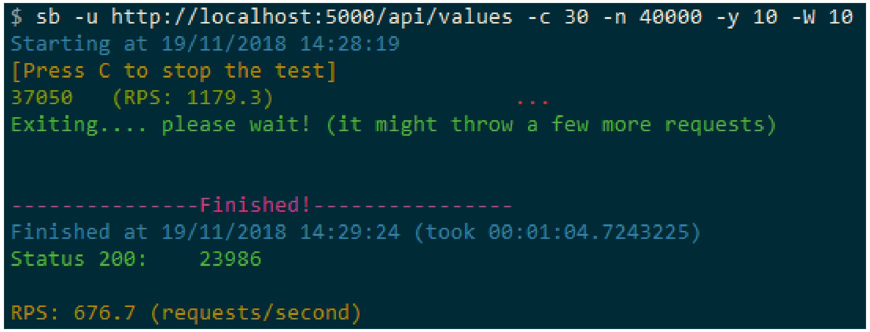
```
> dotnet new webapi -o CoreCLR.WebApi
```

```
[HttpGet]
public IEnumerable<string> Get()
{
   return new string[] { DateTime.Now.ToLongTimeString(), "value2" };
}
```

```
> dotnet build -c Release
> set COMPlus_GCName=f:\CoreCLR.ZeroGC\x64\Release\ZeroGC.dll
> dotnet run -c Release
```

Zero GC applied - results

.NET Core 2.1 with Zero GC:

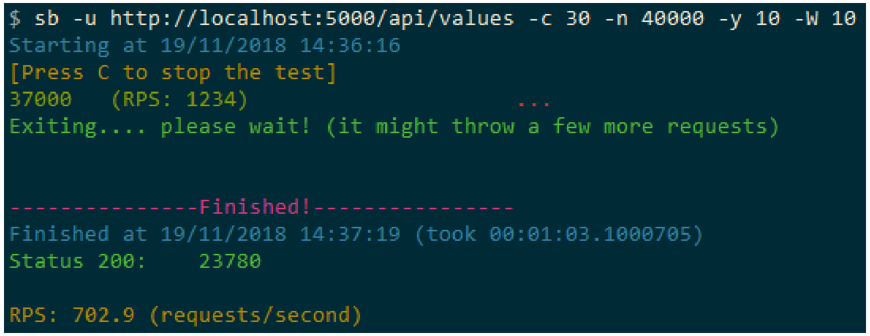


```
$ sb -u http://localhost:5000/api/values -c 30 -n 40000 -y 10 -W 10
Starting at 19/11/2018 14:28:19
[Press C to stop the test]
37050   (RPS: 1179.3)                   ...
Exiting.... please wait! (it might throw a few more requests)


---------------Finished!----------------
Finished at 19/11/2018 14:29:24 (took 00:01:04.7243225)
Status 200:    23986

RPS: 676.7 (requests/second)
```

.NET Core 2.1:



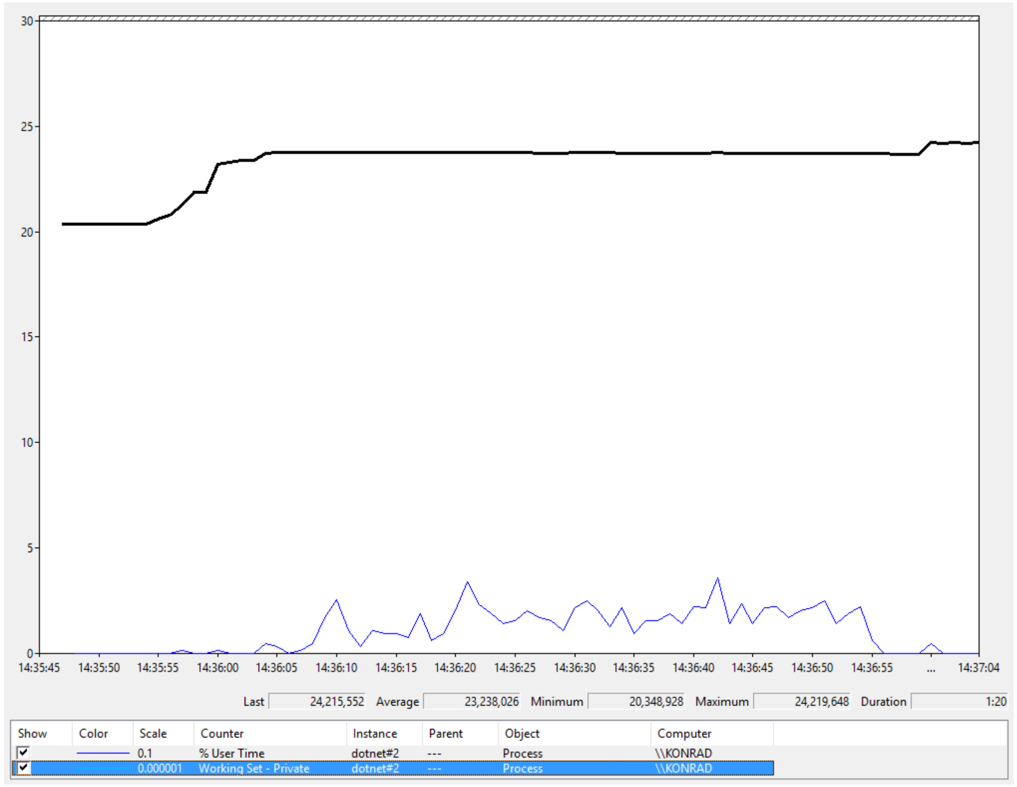
```
$ sb -u http://localhost:5000/api/values -c 30 -n 40000 -y 10 -W 10
Starting at 19/11/2018 14:36:16
[Press C to stop the test]
37000   (RPS: 1234)                     ...
Exiting.... please wait! (it might throw a few more requests)


---------------Finished!----------------
Finished at 19/11/2018 14:37:19 (took 00:01:03.1000705)
Status 200:    23780

RPS: 702.9 (requests/second)
```

Zero GC applied - results

.NET Core 2.1:

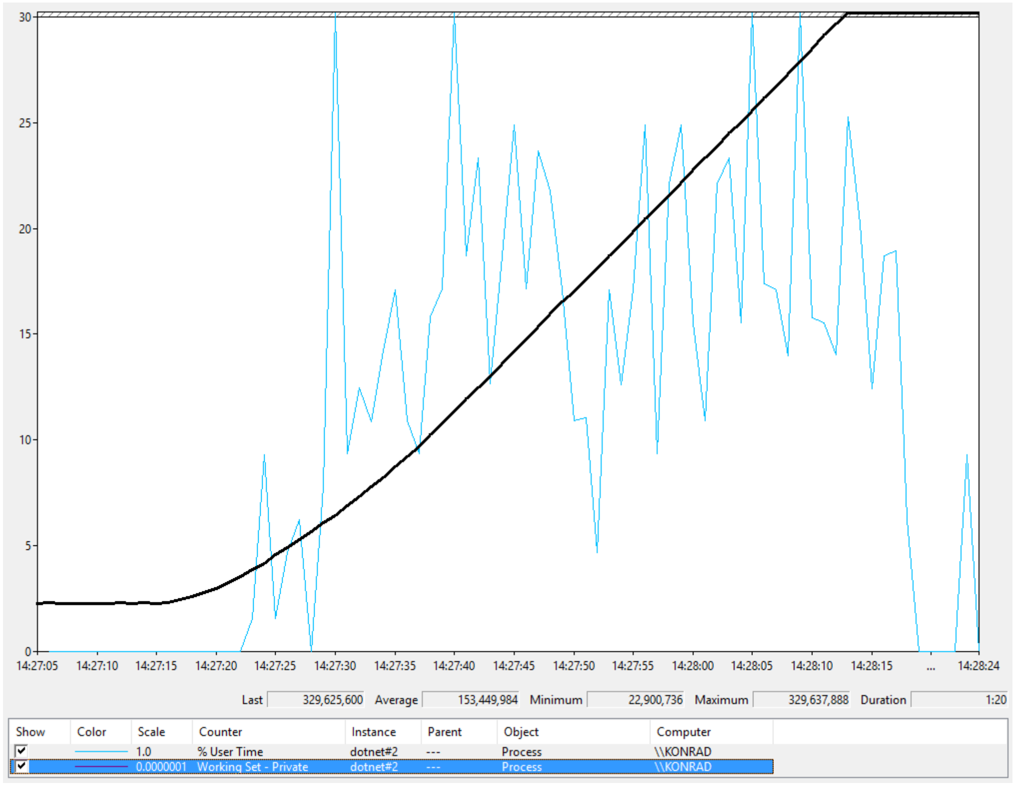


Last 24,215,552 Average 23,238,026 Minimum 20,348,928 Maximum 24,219,648 Duration 1:20

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	──	0.1	% User Time	dotnet#2	---	Process	\\KONRAD
✓		0.000001	Working Set - Private	dotnet#2	---	Process	\\KONRAD

Zero GC applied - results

.NET Core 2.1 with Zero GC:



Last	Average	Minimum	Maximum	Duration
329,625,600	153,449,984	22,900,736	329,637,888	1:20

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	―	1.0	% User Time	dotnet#2	---	Process	\\KONRAD
✓	―	0.0000001	Working Set - Private	dotnet#2	---	Process	\\KONRAD

~314 MB after 24k requests (~11kB/request)

What's next?

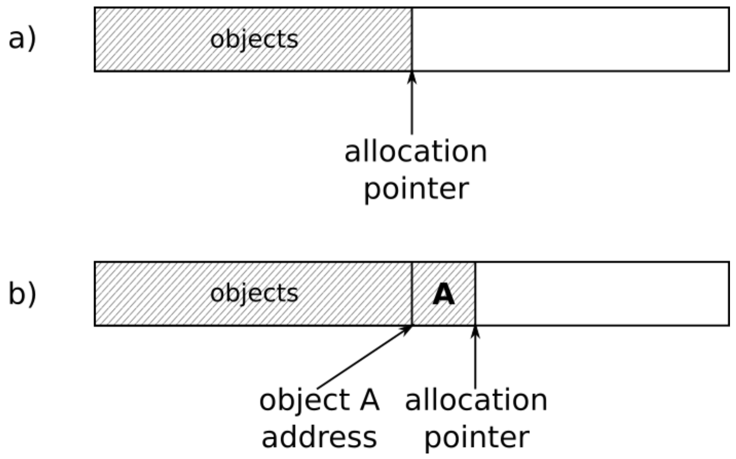
What's next?

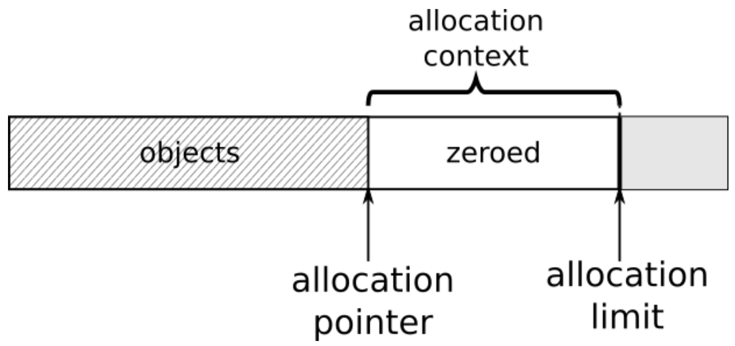
Calloc-based allocator is slow (each object triggers OS call and memory zeroying)

What's next?

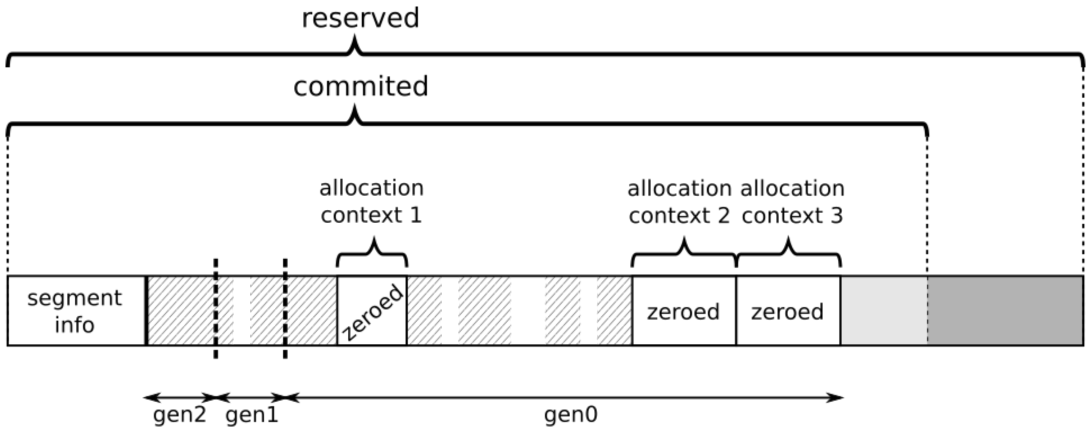
Calloc-based allocator is slow (each object triggers OS call and memory zeroying)

Bump-pointer allocator instead of slooow `calloc`


a)
objects
allocation pointer
b)
objects
A
object A address
allocation pointer



```
Allocator::Allocate(amount)
{
   if (alloc_ptr + amount <= alloc_limit)
   {
      // This is the fast path - we have enough memory to bump the pointer
      PTR result = alloc_ptr;
      alloc_ptr += amount;
      return result;
   }
   else
   {
      // This is the slow path - new allocation context will be created
      ...
   }
}
```



Thread-affinity of **the allocation context structure** - ensured by the runtime

Bump-pointer GC allocator - step #1:

```
// Normally both SOH and LOH allocations go through there
Object * ZeroGCHeap::Alloc(
   gc_alloc_context * acontext,
   size_t size,
   uint32_t flags)
{
   // Per thread acontext...
   // acontext->alloc_ptr
   // acontext->alloc_limit
}
```

Bump-pointer GC allocator - step #2:

```
// Normally both SOH and LOH allocations go through there
Object * ZeroGCHeap::Alloc(
   gc_alloc_context * acontext,
   size_t size,
   uint32_t flags)
{
   uint8_t* result = acontext->alloc_ptr;
   uint8_t* advance = result + size;
   if (advance <= acontext->alloc_limit)
   {
      acontext->alloc_ptr = advance;
      return (Object* )result;
   }
   ...
}
```

Bump-pointer GC allocator - step #3:

```
// Normally both SOH and LOH allocations go through there
Object * ZeroGCHeap::Alloc(
   gc_alloc_context * acontext,
   size_t size,
   uint32_t flags)
{
   uint8_t* result = acontext->alloc_ptr;
   uint8_t* advance = result + size;
   if (advance <= acontext->alloc_limit)
   {
      acontext->alloc_ptr = advance;
      return (Object* )result;
   }
   int growthSize = 16 * 1024 * 1024;
   uint8_t* newPages = (uint8_t*)VirtualAlloc(NULL, growthSize,
                                              MEM_RESERVE | MEM_COMMIT,
                                              PAGE_READWRITE);
   uint8_t* allocationStart = newPages;
   acontext->alloc_ptr = allocationStart + size;
   acontext->alloc_limit = newPages + growthSize;
   return (Object*)(allocationStart);
}
```

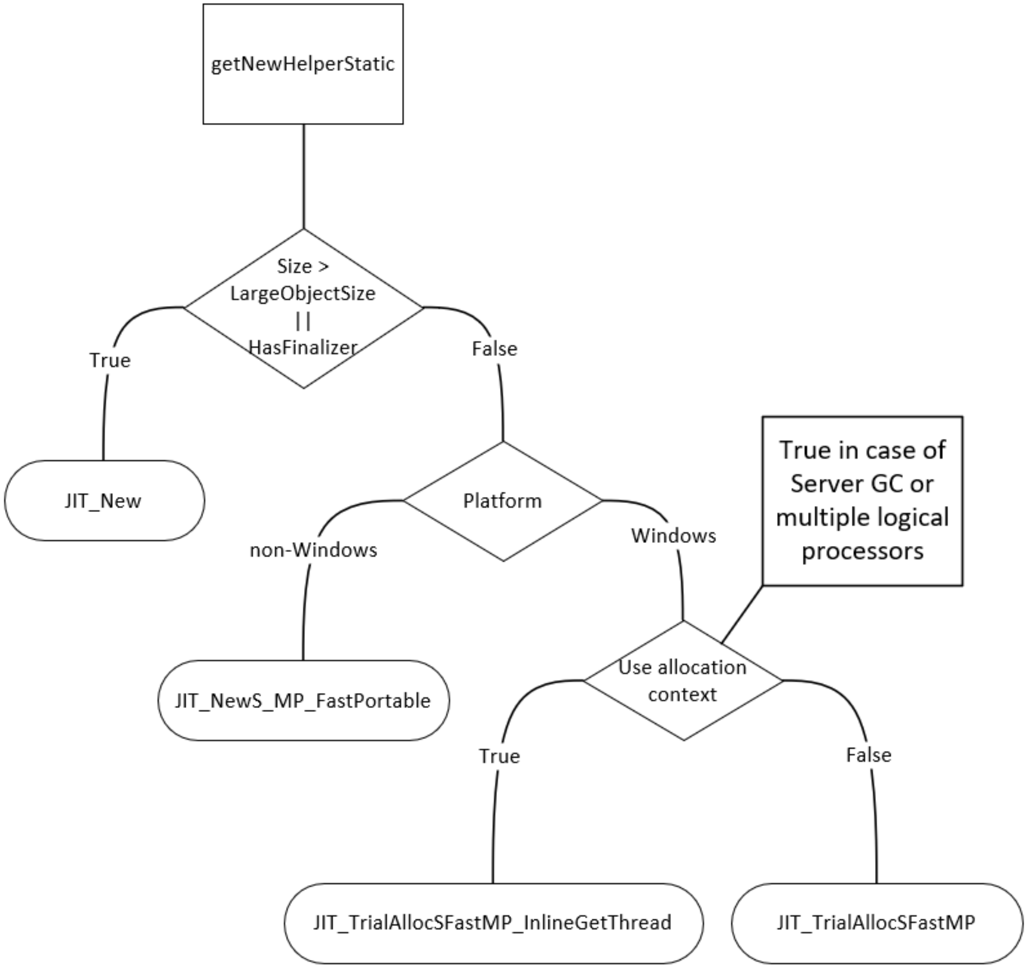
Bump-pointer GC allocator - step #4:

```
// Normally both SOH and LOH allocations go through there
Object * ZeroGCHeap::Alloc(
   gc_alloc_context * acontext,
   size_t size,
   uint32_t flags)
{
   uint8_t* result = acontext->alloc_ptr;
   uint8_t* advance = result + size;
   if (advance <= acontext->alloc_limit)
   {
      acontext->alloc_ptr = advance;
      return (Object* )result;
   }
   int beginGap = 24;
   int growthSize = 16 * 1024 * 1024;
   uint8_t* newPages = (uint8_t*)VirtualAlloc(NULL, growthSize,
                                              MEM_RESERVE | MEM_COMMIT,
                                              PAGE_READWRITE);
   uint8_t* allocationStart = newPages + beginGap;
   acontext->alloc_ptr = allocationStart + size;
   acontext->alloc_limit = newPages + growthSize;
   return (Object*)(allocationStart);
}
```

Bump-pointer GC allocator - let's ignore those LOHs (thread-safety!):

```
// This variation is used in the rare circumstance when you want to allocate
// an object on the large object heap but the object is not big enough to
// naturally go there.
Object * ZeroGCHeap::AllocLHeap(size_t size, uint32_t flags)
{
    int sizeWithHeader = size + sizeof(ObjHeader);
    ObjHeader* address = (ObjHeader*)calloc(sizeWithHeader, sizeof(char*));
    return (Object*)(address + 1);
}
```

Caveat #2 - allocation context is reused by the runtime (JIT!)


getNewHelperStatic
Size > LargeObjectSize || HasFinalizer
True
JIT_New
False
Platform
non-Windows
JIT_NewS_MP_FastPortable
Windows
True in case of Server GC or multiple logical processors
Use allocation context
True
JIT_TrialAllocSFastMP_InlineGetThread
False
JIT_TrialAllocSFastMP

Fast path in EE (not changeable)

```
; IN: rcx: MethodTable*
; OUT: rax: new object
LEAF_ENTRY JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
        mov     edx, [rcx + OFFSET__MethodTable__m_BaseSize]

        ; m_BaseSize is guaranteed to be a multiple of 8.

        INLINE_GETTHREAD r11
        mov     r10, [r11 + OFFSET__Thread__m_alloc_context__alloc_limit]
        mov     rax, [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr]

        add     rdx, rax

        cmp     rdx, r10
        ja      AllocFailed

        mov     [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr], rdx
        mov     [rax], rcx

        ret

    AllocFailed:
        jmp     JIT_NEW
LEAF_END JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
```

Fast path in EE (not changeable)

```
; IN: rcx: MethodTable*
; OUT: rax: new object
LEAF_ENTRY JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
        mov     edx, [rcx + OFFSET__MethodTable__m_BaseSize]

        ; m_BaseSize is guaranteed to be a multiple of 8.

        INLINE_GETTHREAD r11
        mov     r10, [r11 + OFFSET__Thread__m_alloc_context__alloc_limit]
        mov     rax, [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr]

        add     rdx, rax

        cmp     rdx, r10
        ja      AllocFailed

        mov     [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr], rdx
        mov     [rax], rcx

        ret

    AllocFailed:
        jmp     JIT_NEW
LEAF_END JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
```

So why Calloc-based approach works?

Fast path in EE (not changeable)

```
; IN: rcx: MethodTable*
; OUT: rax: new object
LEAF_ENTRY JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
        mov     edx, [rcx + OFFSET__MethodTable__m_BaseSize]

        ; m_BaseSize is guaranteed to be a multiple of 8.

        INLINE_GETTHREAD r11
        mov     r10, [r11 + OFFSET__Thread__m_alloc_context__alloc_limit]
        mov     rax, [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr]

        add     rdx, rax

        cmp     rdx, r10
        ja      AllocFailed

        mov     [r11 + OFFSET__Thread__m_alloc_context__alloc_ptr], rdx
        mov     [rax], rcx

        ret

    AllocFailed:
        jmp     JIT_NEW
LEAF_END JIT_TrialAllocSFastMP_InlineGetThread, _TEXT
```

JIT_NEW fall-back:

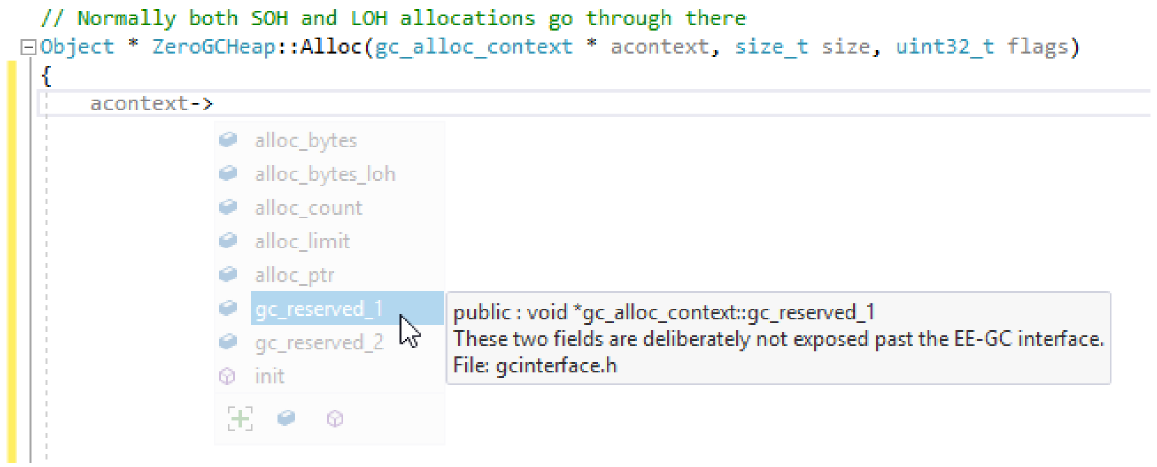
```
HCIMPL1(Object*, JIT_New, CORINFO_CLASS_HANDLE typeHnd_)
{
    ...
    TypeHandle typeHnd(typeHnd_);
    MethodTable *pMT = typeHnd.AsMethodTable();
    newobj = AllocateObject(pMT);
    return(OBJECTREFToObject(newobj));
}
HCIMPLEND
```

```
Object * AllocateObject(MethodTable * pMT)
{
    alloc_context * acontext = GetThread()->GetAllocContext();
    Object * pObject;
    size_t size = pMT->GetBaseSize();
    uint8_t* result = acontext->alloc_ptr;
    uint8_t* advance = result + size;
    if (advance <= acontext->alloc_limit)
    {
        acontext->alloc_ptr = advance;
        pObject = (Object *)result;
    }
    else
    {
        pObject = g_theGCHeap->Alloc(acontext, size, 0);
        if (pObject == NULL) return NULL;
    }
    pObject->RawSetMethodTable(pMT);
    return pObject;
}
```

Bump-pointer GC allocator - step #4 repeated:

```
// Normally both SOH and LOH allocations go through there
Object * ZeroGCHeap::Alloc(
   gc_alloc_context * acontext,
   size_t size,
   uint32_t flags)
{
   uint8_t* result = acontext->alloc_ptr;
   uint8_t* advance = result + size;
   if (advance <= acontext->alloc_limit)
   {
      acontext->alloc_ptr = advance;
      return (Object* )result;
   }
   int beginGap = 24;
   int growthSize = 16 * 1024 * 1024;
   uint8_t* newPages = (uint8_t*)VirtualAlloc(NULL, growthSize,
                                              MEM_RESERVE | MEM_COMMIT,
                                              PAGE_READWRITE);
   uint8_t* allocationStart = newPages + beginGap;
   acontext->alloc_ptr = allocationStart + size;
   acontext->alloc_limit = newPages + growthSize;
   return (Object*)(allocationStart);
}
```

Or ignore `alloc_ptr` and `alloc_limit` by using custom fields

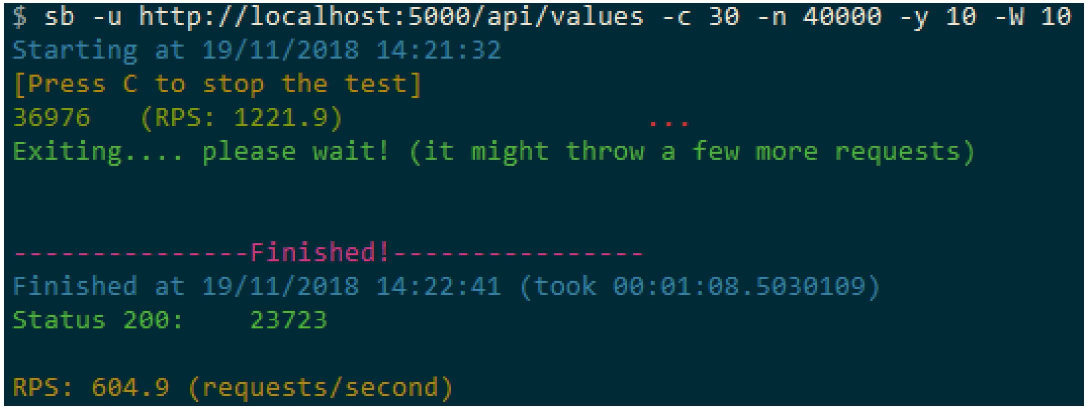


```
// Normally both SOH and LOH allocations go through there
Object * ZeroGCHeap::Alloc(gc_alloc_context * acontext, size_t size, uint32_t flags)
{
    acontext->
```

public : void *gc_alloc_context::gc_reserved_1
These two fields are deliberately not exposed past the EE-GC interface.
File: gcinterface.h

Zero GC bump pointer applied - results

.NET Core 2.1:

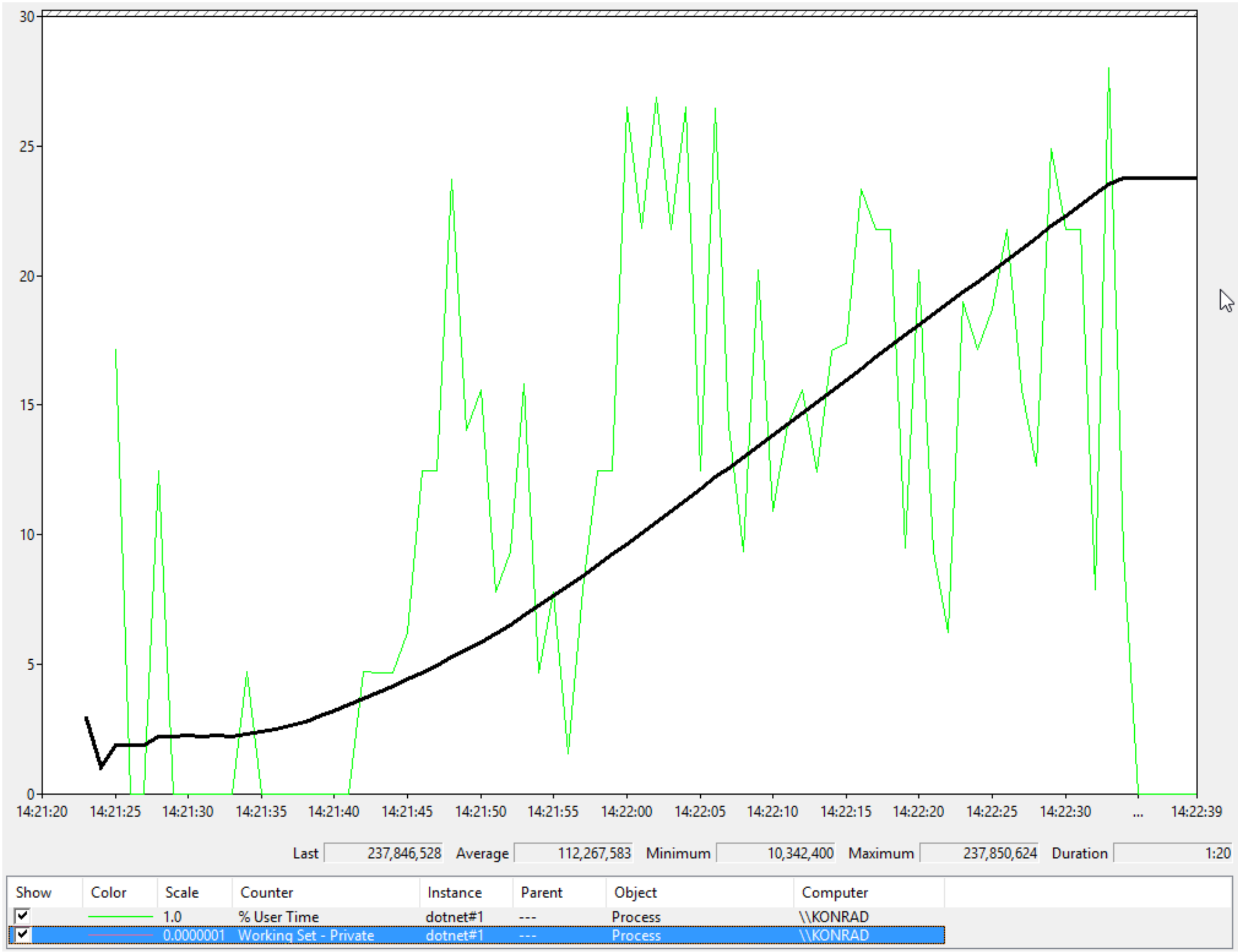


```
$ sb -u http://localhost:5000/api/values -c 30 -n 40000 -y 10 -W 10
Starting at 19/11/2018 14:21:32
[Press C to stop the test]
36976   (RPS: 1221.9)                   ...
Exiting.... please wait! (it might throw a few more requests)


---------------Finished!----------------
Finished at 19/11/2018 14:22:41 (took 00:01:08.5030109)
Status 200:    23723

RPS: 604.9 (requests/second)
```

Zero GC applied - results



Last 237,846,528 Average 112,267,583 Minimum 10,342,400 Maximum 237,850,624 Duration 1:20

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟩	1.0	% User Time	dotnet#1	---	Process	\\KONRAD
✓	🟥	0.0000001	Working Set - Private	dotnet#1	---	Process	\\KONRAD

Important runtime support

- Events

```
gcToCLR->EventSink()->FireGCCreateSegment_V1(newPages, growthSize, 0);
```

- Threading:

```
CreateThread(void (*threadStart)(void*), void* arg, bool is_suspendable,
             const char* name)
SuspendEE(SUSPEND_REASON reason)
RestartEE(bool bFinishedGC)
GcScanRoots(promote_func* fn, int condemned, int max_gen, ScanContext* sc)
```

- Configuration:

```
GetBooleanConfigValue(const char* key, bool* value)
GetIntConfigValue(const char* key, int64_t* value)
GetStringConfigValue(const char* key, const char** value)
```

What's next?

What's next?

... just draw f** owl!

Question: What one should even care?

Question: What one should even care?

- learning A LOT
- having a GREAT FUN
- creating customized, specialized GC
    - **or awaited concurrent compacting GC (yeah, simple...)**

Question: What about finalizers?

Question: What about finalizers?

- Currently ignored!
- Runtime still creates and maintains **finalization thread**
- Hmm ... AFAIK currently no API to communicate with it...

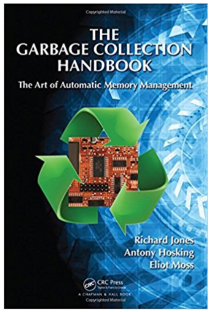
Question: What about multiple GC heaps (like in Server GC)?

Question: What about multiple GC heaps (like in Server GC)?

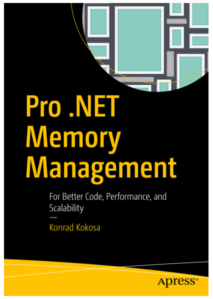
- Currently ignored!
- One would need to implement it - core/heap affinity, heap balancing, ...

Literature:

- The Garbage Collection Handbook (http://gchandbook.org) - Richard Jones, Antony Hosking, Eliot Moss


THE GARBAGE COLLECTION HANDBOOK
The Art of Automatic Memory Management
Richard Jones
Antony Hosking
Eliot Moss
CRC Press

- Pro .NET Memory Management Management (https://prodotnetmemory.com) - Konrad Kokosa


Pro .NET Memory Management
For Better Code, Performance, and Scalability
Konrad Kokosa
Apress

- http://tooslowexception.com/zero-garbage-collector-for-net-core/
- http://tooslowexception.com/zero-garbage-collector-for-net-core-2-1-and-asp-net-core-2-1/

That's all! Thank **you**! Any questions?!