Debugging Memory Leaks in .NET

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

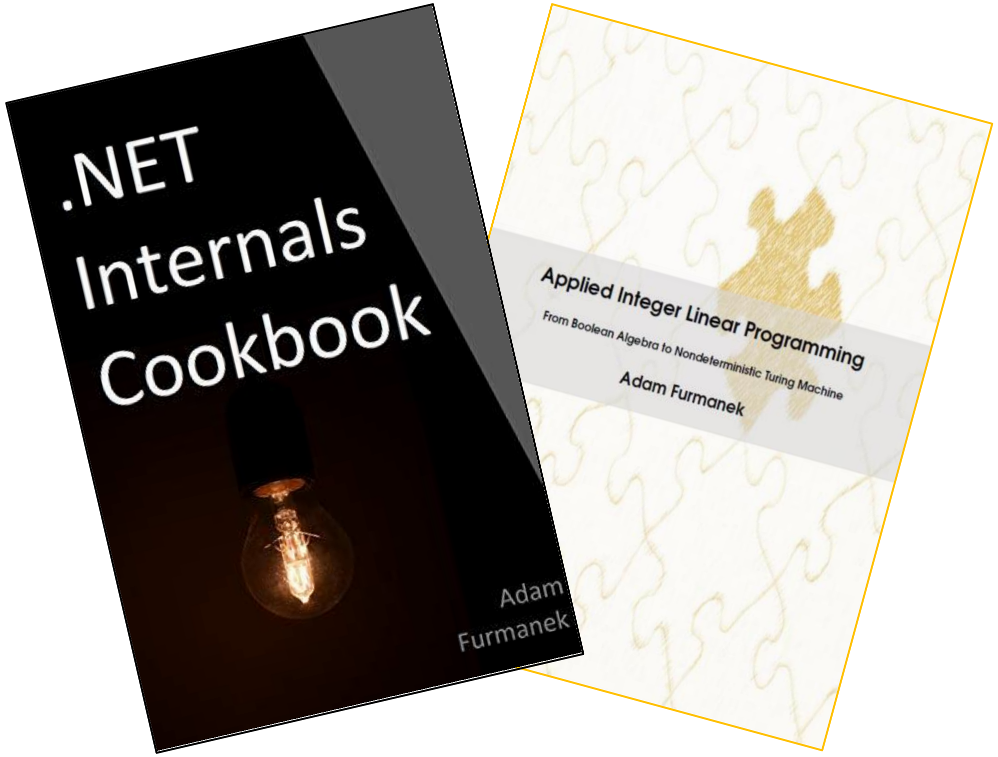
About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of ***Applied Integer Linear Programming*** and ***.NET Internals Cookbook***.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam



Random IT Utensils

IT, operating systems, maths, and more.

Agenda

Garbage Collection:

- Reference counting
- Mark and Swep, Stopping the world, Mark and Sweep and Compact
- Generational hypothesis, card tables

.NET GC:

- Roots, types
- SOH and LOH
- Finalization queue, IDisposable, Resurrection

Demos:

- WinDBG
- Event handlers
- XML Generation
- WCF

Theory

Reference counting

Each object has counter of references pointing to it.

On each assignment the counter is incremented, when variable goes out of scope the counter is decremented.

Can be implemented automatically by compiler.

Fast and easy to implement.

Cannot detect cycles.

Used in COMs.

Used in CPython and Swift.

Mark and Sweep

At various moments GC looks for all living objects and releases dead ones.

Release means *mark memory as free.* There is no list of all alocated objects! GC doesn’t know whether there is an object (or objects) or not.

If object needs to be released with special care (e.g., contains destructor), GC must know about it so it is rememberd during allocation.

Stop the world

GC stops all running threads.

SuspendThread*: This function is primarily designed for use by debuggers. It is not intended to be used for thread synchronization. Calling* ***SuspendThread*** *on a thread that owns a synchronization object, such as a mutex or critical section, can lead to a deadlock if the calling thread tries to obtain a synchronization object owned by a suspended thread. To avoid this situation, a thread within an application that is not a debugger should signal the other thread to suspend itself. The target thread must be designed to watch for this signal and respond appropriately.*

How does GC know whether it is safe to pause the thread? Safepoints.

What if the thread doesn't want to go to the safepoint? Thread hijacking.

Mark and Sweep

Can be executed without stopping the world:

- If we mark object as alive and in fact it is not (false positive), it will be released next time
- If we allocate new object during GC phase, GC needs to know about it (so GC hijacks allocation process)
- Finding roots might be a bit difficult (since they can move to and from registers and be optimized away)

Mark and Sweep and Compact

When Mark and Swep is done (e.g., memory is ready to be released), objects are compacted.

Compaction might take significant amount of time so there are heuristics to avoid it (e.g., LOH).

Objects are copied from one place to another and all references are updated.

Can be executed without stopping the world:

- Memory page with object is marked as read-only
- When thread tries to access it, GC handles page fault and redirects read to other place

Generational hypothesis

Reality shows that objects can be divided in two groups:

- Those dying very quickly after allocation
- Those living very long (e.g., throught whole application execution)

We can come up with hypothesis: if object survives first GC phase, it will live long.

Idea: let’s divide objects into generations (0, 1 and 2 in .NET, eden and tenured in CMS, eden, survivor and tenured in G1).

Benefits:

- We can run GC more often and focus only on newly allocated objects
- We don’t need to scan whole memory (since allocations occur in small address space)

Bonus chatter: back references

```
using System;

public class C1{
        public C2 Field {get; set;}
}

public class C2{
}

public class Program
{

        public static void Main()
        {
                var c1 = new C1();
                GC.Collect();
                // c1 should be in generation 1 now

                c1.Field = new C2();
                GC.Collect(0);
                // How does GC find the C2 instance now?
        }
}
```

Card tables

Card table is a set of bits representing whole memory.

Each bit says whether particular region of memory (typically 256B) was modified.

When we perform allocation of any time, it is not executed directly (e.g., as mov in machine code) but is redirected to .NET helper method.

This method assigns the variable and stores the bit in card table.

GC then uses card tables to avoid scanning whole memory.

Interesting things not covered

Tri-color marking.

Types of weak references.

Internal pointers.

Differentiating pointers from value types.

Tagged pointers.

Mark and don't sweep.

Hard realtime GC, Metronome algorithm.

GC without stop the world.

GC and structures like XOR list.

.NET

GC in general

GC:

- checks JIT compiler, stack, handles table, finalizer queue, static variables and registers
- might not stop the threads running native code
- leaves cookies on the stack to find out transitions between native and managed code
- doesn’t release once allocated blocks, this is called VM_HOARDING
- can execute finalizer even when there is other object’s method running
- can pin non-movable objects
- can be turned off
- supports weak references
- uses three generations (0, 1, and 2)

.NET doesn’t use Frame Pointer Omission.

GC phases

Marking, usually requires stop the world for generation 0 or 1.

Relocating (updating pointers).

Compacting.

GC Types

Workstation

- Can be concurrent (default on client machines)
- Used always on uniprocessor machine
- Collection is performer on calling thread
- GC has the same priority
- Doesn’t stop threads running native code

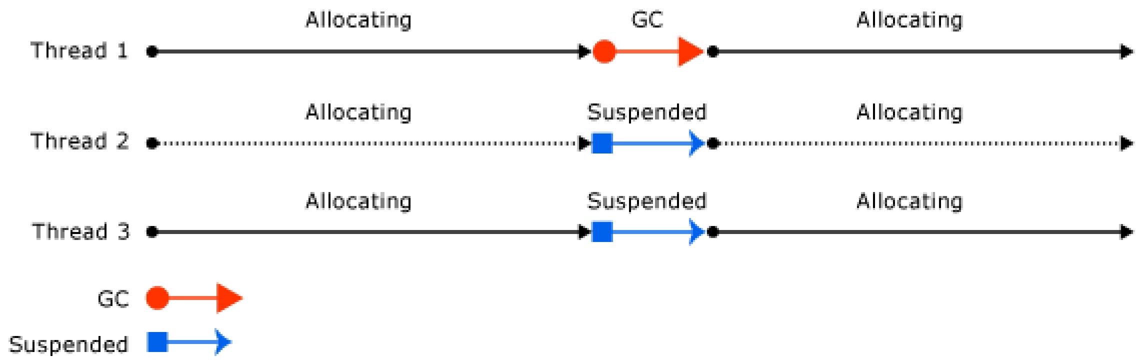
Server

- Works on mulitple dedicated threads with priority THREAD_PRIORITY_HIGHEST
- Each procesor has separate stack and steap
- Stops all threads

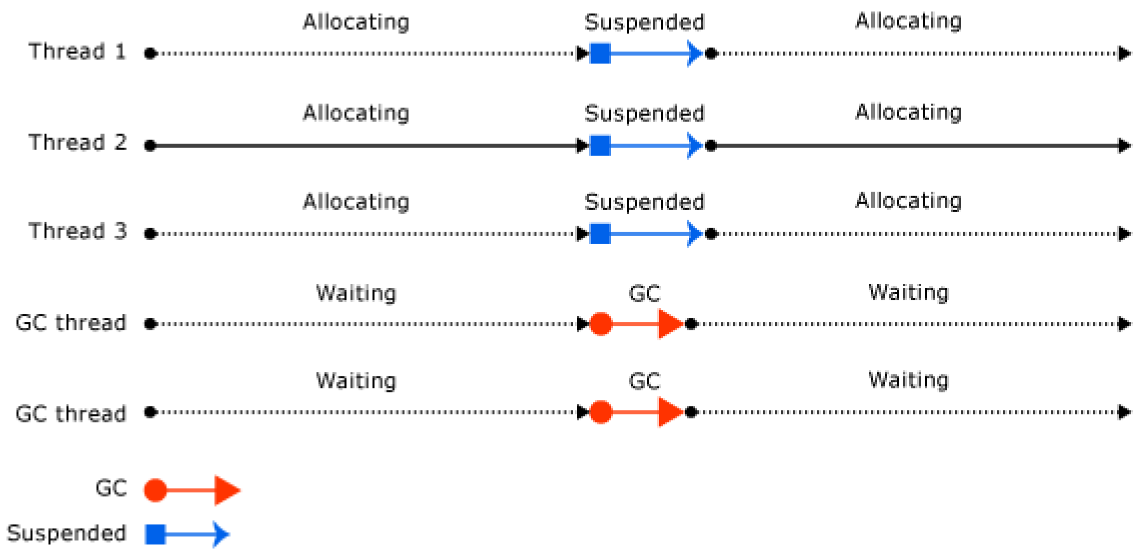
Background GC

- Works in Workstation and Server
- Collects only generation 2

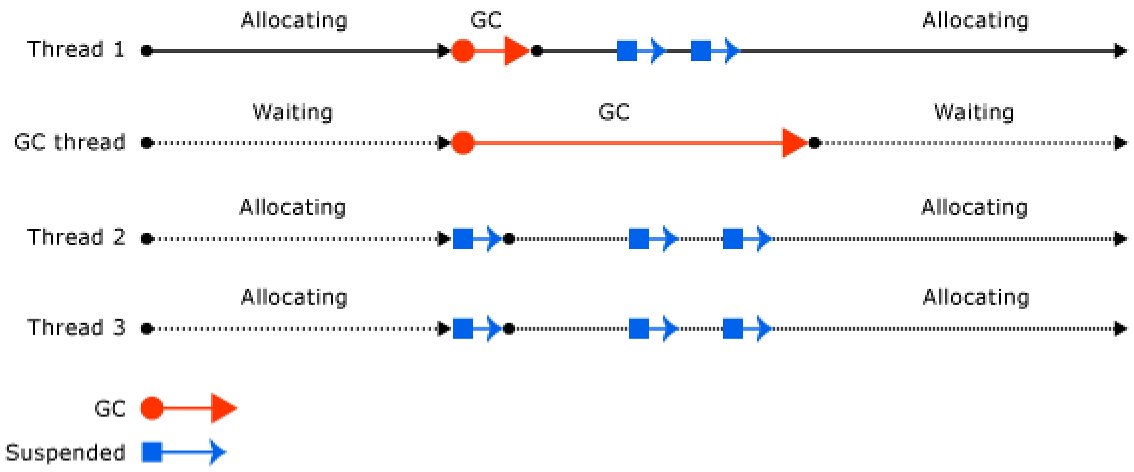
GC Types – Workstation non-concurrent



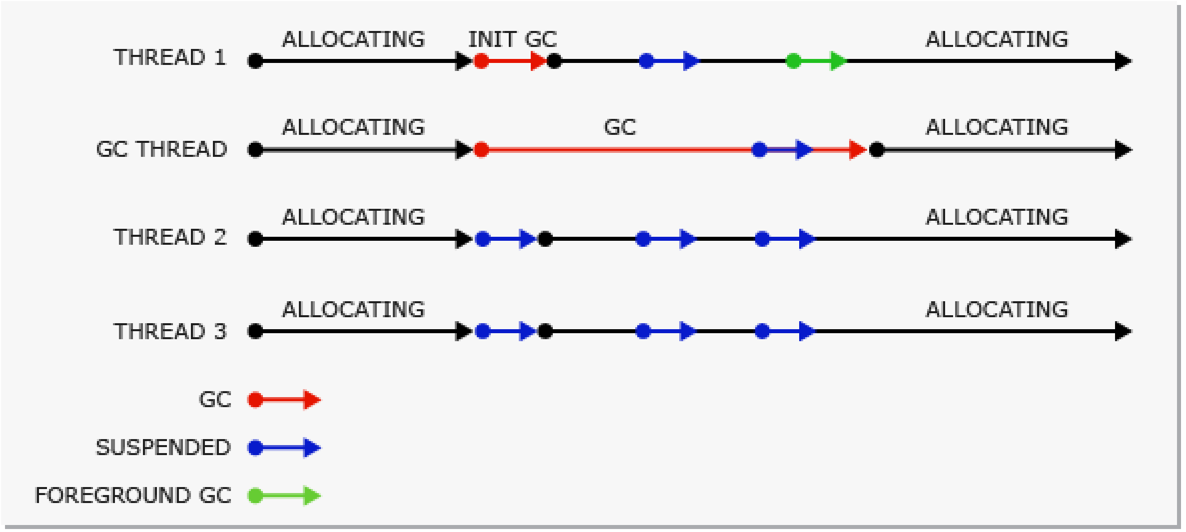
GC Types – Server non-concurrent



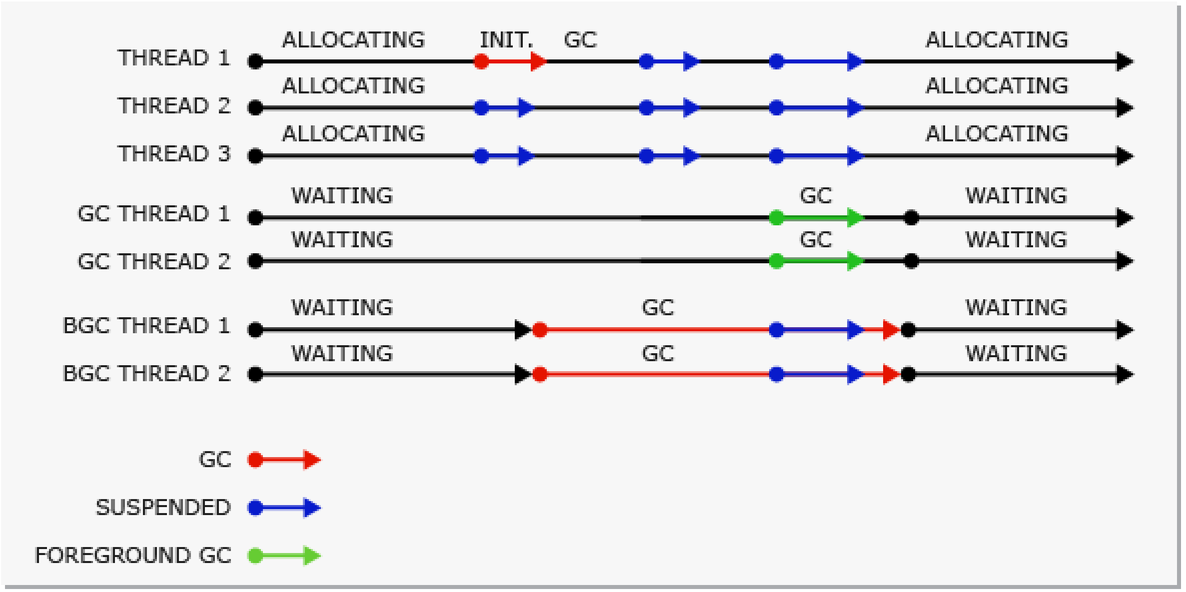
GC Types - Concurrent



GC Types — Workstation background



GC Types — Server background



SOH and LOH

Compacting big objects might take a lot of time.

Objects bigger than 85000 bytes are allocated directly in generation 2 (sometimes incorrectly called generation 3) on the special area called Large Object Heap.

They are not compacted automatically, can be compacted on demand since 4.5.1.

Fun fact: arrays of 1000+ doubles are stored on LOH in 32-bit .NET Framework / Core.

These are all undocumented features and might change anytime.

Small Object Heap contains ephemeral segment for generations 0 and 1. Each new segment is ephemeral, old ephemeral segment becomes generation 2 segment.

Ephemeral segment can include generation 2 objects.

GC can either copy objects to other generations or move whole segment to other generation.

Generations

There are three generations: 0, 1, and 2. **This can change**!

Initally object is allocated in generation 0 or 2 (LOH).

Object is copied to generation 1 after GC.

Generations are calculated using addresses. Stack is in generation 2 because it doesn’t fit in any other generation ranges.

It is possible to allocated reference object on a stack.

Write barrier

```
using System;

namespace ConsoleApplication11
{
    1 reference
    public class C1
    {
        public C2 Field;
    }

    2 references
    public class C2
    {
    }

    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            var c1 = new C1();
            GC.Collect();

            c1.Field = new C2();
            Console.ReadLine();
        }
    }
}
```

```
.method private hidebysig static
    void Main (
        string[] args
    ) cil managed
{
    // Method begins at RVA 0x2058
    // Code size 27 (0x1b)
    .maxstack 8
    .entrypoint

    IL_0000: newobj instance void ConsoleApplication11.C1::.ctor()
    IL_0005: call void [mscorlib]System.GC::Collect()
    IL_000a: newobj instance void ConsoleApplication11.C2::.ctor()
    IL_000f: stfld class ConsoleApplication11.C2 ConsoleApplication11.C1::Field
    IL_0014: call string [mscorlib]System.Console::ReadLine()
    IL_0019: pop
    IL_001a: ret
} // end of method Program::Main
```

```
.method private hidebysig static
    void Main (
        string[] args
    ) cil managed
{
    // Method begins at RVA 0x2058
    // Code size 27 (0x1b)
    .maxstack 8
    .entrypoint

    IL_0000: newobj instance void ConsoleApplication11.C1::.ctor()
    IL_0005: call void [mscorlib]System.GC::Collect()
    IL_000a: newobj instance void ConsoleApplication11.C2::.ctor()
    IL_000f: stfld class ConsoleApplication11.C2 ConsoleApplication11.C1::Field
    IL_0014: call string [mscorlib]System.Console::ReadLine()
    IL_0019: pop
    IL_001a: ret
} // end of method Program::Main
```

```
>>> 00ea0448 55            push    ebp
00ea0449 8bec            mov     ebp,esp
00ea044b 56              push    esi
00ea044c b9904de400      mov     ecx,0E44D90h (MT: ConsoleApplication11.C1)
00ea0451 e8722cf9ff      call    00e330c8 (JitHelp: CORINFO_HELP_NEWSFAST)
00ea0456 8bf0            mov     esi,eax

c:\users\adam.furmanek\documents\visual studio 2015\Projects\ConsoleApplication11\ConsoleApplication11\Program.cs @ 19:
00ea0458 83c9ff          or      ecx,0FFFFFFFFh
00ea045b 8d5103          lea     edx,[ecx+3]
00ea045e e8ad79d150      call    mscorlib_ni+0x317e10 (51bb7e10) (System.GC._Collect(Int32, Int32), mdToken: 06000e6c)

c:\users\adam.furmanek\documents\visual studio 2015\Projects\ConsoleApplication11\ConsoleApplication11\Program.cs @ 21:
00ea0463 b9ec4de400      mov     ecx,0E44DECh (MT: ConsoleApplication11.C2)
00ea0468 e85b2cf9ff      call    00e330c8 (JitHelp: CORINFO_HELP_NEWSFAST)
00ea046d 8d5604          lea     edx,[esi+4]
00ea0470 e84be23271      call    clr!JIT_WriteBarrierEAX (721ce6c0)

c:\users\adam.furmanek\documents\visual studio 2015\Projects\ConsoleApplication11\ConsoleApplication11\Program.cs @ 22:
00ea0475 e8d2705451      call    mscorlib_ni+0xb4754c (523e754c) (System.Console.ReadLine(), mdToken: 06000b40)

c:\users\adam.furmanek\documents\visual studio 2015\Projects\ConsoleApplication11\ConsoleApplication11\Program.cs @ 23:
00ea047a 5e              pop     esi
00ea047b 5d              pop     ebp
00ea047c c3              ret
```

Pinning

.NET moves objects in memory which might cause problems (e.g., P/Invoke).

We can *pin* object in memory using *fixed* keyword or *GCHandle.Alloc* with type *Pinned*.

Problems:

- GC cannot move objects — fragmentation
- Ephemeral segment might become full

Weak references

Weak reference must be known to .NET and GC. It cannot be a simple pointer because:

- Objects are moved in memory (compaction) so GC needs to update the pointer — so weak reference cannot be an IntPtr
- GC needs to be able to free the memory — so weak reference cannot be a typed reference

Weak reference ist stored as an IntPtr registered in GC.

Every access to weak reference requires asking GC whether the object is still there.

Important: we first need to copy weak reference to strong reference and after that ask wheter it is still alive. Otherwise we might be evicted by GC.

Important 2: Dictionary<TKey, WeakReference> is not good as a cache. The proper way is to use ConditionalWeakTable<TKey, TValue>

Finalization queue

Objects with finalizers are remembered by GC during allocation.

They are stored in finalization queue.

After mark phase, they are moved to f-reachable queue.

There is one separate thread for running finalizers. It can be blocked.

When closing application there is a 2 seconds limit for all finalizers to run.

Bonus chatter: which thread is responsible for closing the application?

IDisposable, Resurrection

When implementing IDisposable interface, object should be removed from finalization queue in Dispose method.

When implementing object pooling, object should be registered for finalization in finalizer.

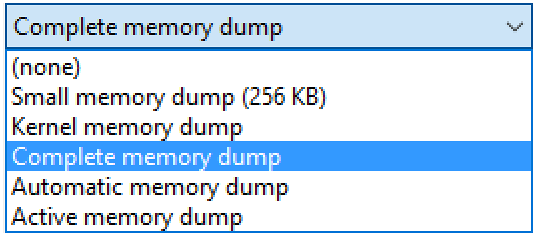
These are ordinary cases in .NET, not some black magic stuff.

Demos

Memory dump types

Created by Windows:

- Complete memory dump
  - Contains absolutely everything
- Kernel memory dump
- Small memory dump
  - 256 kB
  - Contains loaded drivers, bugcheck (BSOD) code and critical kernel structures
- Automatic == kernel memory dump
- Active memory dump
  - Ignores data for virtual machines



Memory dump is created in pagefile by default. Can be changed.

Memory dump types

Created by developer:

- Full dump
  - **<u>Does not contain all informations</u>**
- Minidump
  - Can be configured to contain everything

```
typedef enum _MINIDUMP_TYPE {
  MiniDumpNormal                          = 0x00000000,
  MiniDumpWithDataSegs                    = 0x00000001,
  MiniDumpWithFullMemory                  = 0x00000002,
  MiniDumpWithHandleData                  = 0x00000004,
  MiniDumpFilterMemory                    = 0x00000008,
  MiniDumpScanMemory                      = 0x00000010,
  MiniDumpWithUnloadedModules             = 0x00000020,
  MiniDumpWithIndirectlyReferencedMemory  = 0x00000040,
  MiniDumpFilterModulePaths               = 0x00000080,
  MiniDumpWithProcessThreadData           = 0x00000100,
  MiniDumpWithPrivateReadWriteMemory      = 0x00000200,
  MiniDumpWithoutOptionalData             = 0x00000400,
  MiniDumpWithFullMemoryInfo              = 0x00000800,
  MiniDumpWithThreadInfo                  = 0x00001000,
  MiniDumpWithCodeSegs                    = 0x00002000,
  MiniDumpWithoutAuxiliaryState           = 0x00004000,
  MiniDumpWithFullAuxiliaryState          = 0x00008000,
  MiniDumpWithPrivateWriteCopyMemory      = 0x00010000,
  MiniDumpIgnoreInaccessibleMemory        = 0x00020000,
  MiniDumpWithTokenInformation            = 0x00040000,
  MiniDumpWithModuleHeaders               = 0x00080000,
  MiniDumpFilterTriage                    = 0x00100000,
  MiniDumpValidTypeFlags                  = 0x001fffff
} MINIDUMP_TYPE;
```

Creating memory dump in Windows

Memory dump can be created using:

- Task manager
  - Only full dump (?)
- Process Explorer
  - Minidumps and full dumps
- ADPlus
  - Minidumps and full dumps
- WinDBG
  - Any type of dump
  - .dump /mf <path>

WCF

```
var type = typeof (ClientBase<IBooking>);

var field = type.GetField("factoryRefCache", BindingFlags.Static | BindingFlags.NonPublic);

var cache = field.GetValue(null);

cache.GetType().GetMethod("Clear").Invoke(cache, new object[0]);
```

Q&A

References

Jeffrey Richter - „CLR via C#”

Jeffrey Richter, Christophe Nasarre - „Windows via C/C++”

Mark Russinovich, David A. Solomon, Alex Ionescu - „Windows Internals”

Penny Orwick – „Developing drivers with the Microsoft Windows Driver Foundation”

Mario Hewardt, Daniel Pravat - „Advanced Windows Debugging”

Mario Hewardt - „Advanced .NET Debugging”

Steven Pratschner - „Customizing the Microsoft .NET Framework Common Language Runtime”

Serge Lidin - „Expert .NET 2.0 IL Assembler”

Joel Pobar, Ted Neward — „Shared Source CLI 2.0 Internals”

Adam Furmanek – „.NET Internals Cookbook”

https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md — „Book of the Runtime”

https://blogs.msdn.microsoft.com/oldnewthing/ — Raymond Chen „The Old New Thing”

References

https://youtu.be/K1N9-9O6PrE — Adam Furmanek about DLL Injection

http://blog.adamfurmanek.pl/2016/03/26/dll-injection-part-1/ — the same as before

https://blog.adamfurmanek.pl/2017/04/15/debugging-wcf-high-memory-usage/ — memory dump debugging

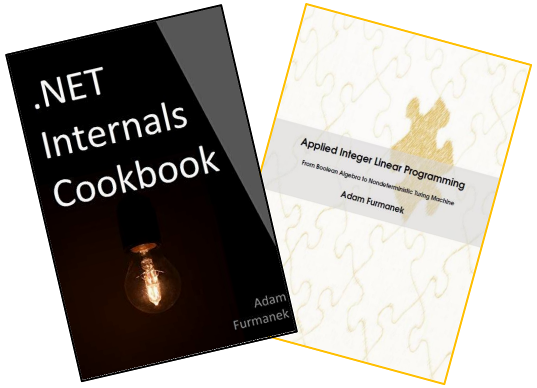
https://blog.adamfurmanek.pl/2016/04/23/custom-memory-allocation-in-c-part-1/ — allocating object on a stack

References

https://channel9.msdn.com/Shows/Defrag-Tools — Defrag Tools on Channel 9

https://www.azul.com/files/c4_paper_acm1.pdf — C4 — Collector without stop the world on x86

https://en.wikipedia.org/wiki/Tracing_garbage_collection — GC overview



Random IT Utensils

IT, operating systems, maths, and more.

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM