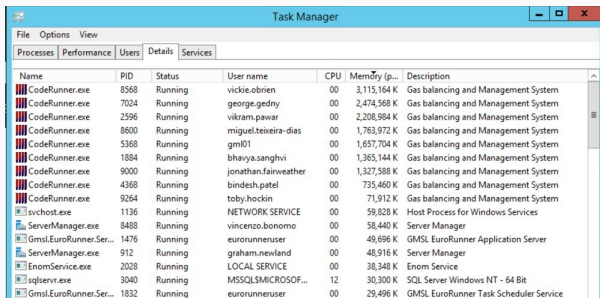


Case study: debugging memory leaks in .NET

- This tech talk is a case study of a memory leak in CodeRunner.
- It's quite gentle and reviews the fundamentals of .NET and memory management.
- We'll cover a small amount of theory interspersed with a simple example and the CodeRunner 'bug'.

Case study



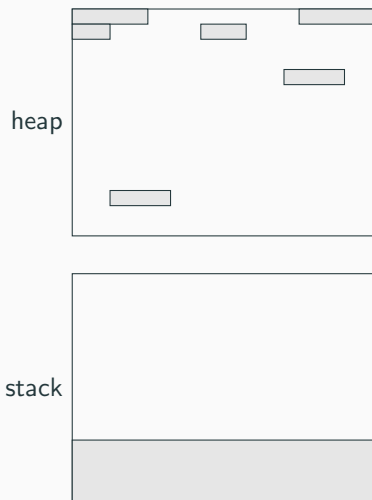
Name	PID	Status	User name	CPU	Memory (p...)	Description
CodeRunner.exe	8568	Running	vickie.obrien	00	3,115,164 K	Gas balancing and Management System
CodeRunner.exe	7024	Running	george.gedny	00	2,474,568 K	Gas balancing and Management System
CodeRunner.exe	2596	Running	vikram.pawar	00	2,208,984 K	Gas balancing and Management System
CodeRunner.exe	8600	Running	miguel.teixeira-dias	00	1,763,972 K	Gas balancing and Management System
CodeRunner.exe	5368	Running	gml01	00	1,657,704 K	Gas balancing and Management System
CodeRunner.exe	1884	Running	bhavya.sanghvi	00	1,365,144 K	Gas balancing and Management System
CodeRunner.exe	9000	Running	jonathan.fainweather	00	1,327,588 K	Gas balancing and Management System
CodeRunner.exe	4368	Running	bindesh.patel	00	735,460 K	Gas balancing and Management System
CodeRunner.exe	9264	Running	toby.hockin	00	71,912 K	Gas balancing and Management System
svchost.exe	1136	Running	NETWORK SERVICE	00	59,828 K	Host Process for Windows Services
ServerManager.exe	8488	Running	vincenzo.bonomo	00	58,440 K	Server Manager
Gmsl.EuroRunner.Ser...	1476	Running	eurorunneruser	00	49,696 K	GMSL EuroRunner Application Server
ServerManager.exe	912	Running	graham.newland	00	48,916 K	Server Manager
EnomService.exe	2028	Running	LOCAL SERVICE	00	38,348 K	Enom Service
sqlservr.exe	3040	Running	MSSQL\$MICROSOFT...	12	30,300 K	SQL Server Windows NT - 64 Bit
Gmsl.EuroRunner.Ser...	1832	Running	eurorunneruser	00	29,496 K	GMSL EuroRunner Task Scheduler Service

- The bug report said that in a recent build of CodeRunner memory usage had significantly increased and resulted in the RAM on the VM being maxed.
- As can be seen in the Task Manager window some instances were using up to 3 GB.
- Normally the first step might be to see what changes have been made inbetween the last release and this release, but it didn't seem obvious (although it is in retrospect).
- With memory leaks there's two questions we might ask: where is the leak, and why is it leaking?
- This tech talk is divided into two parts to look at each question. Let's start with where. . .

Where?

- We'll first review how memory management works in .NET.
- Then we'll look at a simple example that allocates memory.
- Finally we'll find out where the memory leak is in `CodeRunner`.

Stack & heap

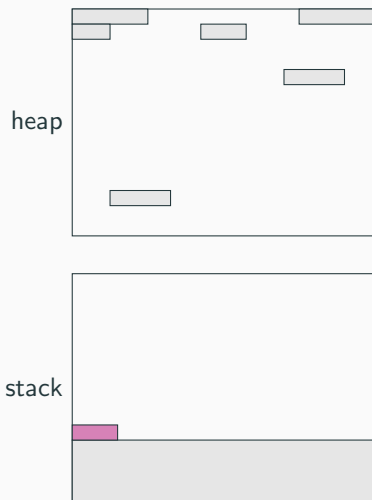


→ Allocate();

```
void Allocate()
{
    char stack;
    char[] heap = new char[1];
}
```

1. It's first useful to review how memory is allocated and unallocated ('freed').
2. We have two sections of memory: the stack and the heap. Here we have a snippet of a simple program where we are in the middle of execution. The grey boxes in the stack and the heap represent memory that's already been allocated.
3. Let's follow memory allocation in execution of this snippet.

Stack & heap



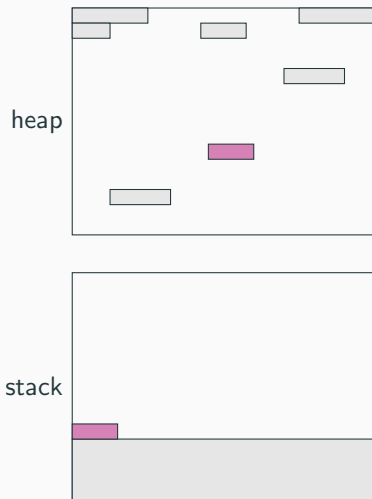
```
Allocate();
```

```
void Allocate()  
{
```

```
    → char stack;  
      char[] heap = new char[1];  
}
```

1. The first statement is something your compiler will warn you about—a variable that is declared but not used. But this variable (unless optimised away) will use memory on the stack.
2. In C# variables can be either value types or reference types. Primitive types are value types. As we'll see shortly the main effect of value types is that the memory allocation is *scoped*.
3. Value types aren't always allocated on the stack, but in the context of a function call evaluation like this they are.
4. The stack contains fixed memory allocations and like other stacks these allocations are “last in first out” (LIFO) so they are unallocated in the reverse order to their allocation.

Stack & heap



```
Allocate();
```

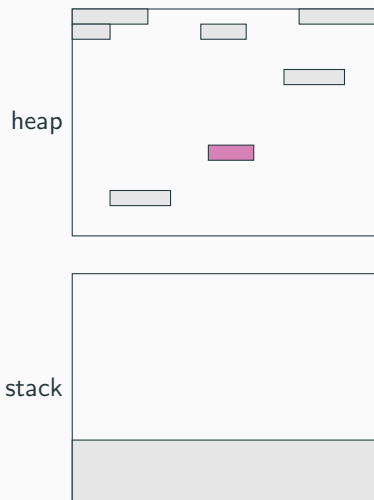
```
void Allocate()  
{
```

```
    char stack;
```

```
    → char[] heap = new char[1];  
}
```

1. This next line contains an array. Arrays are reference types. Reference types are always allocated on the heap so are easier to define than value types.
2. The key thing about reference types is that their memory allocation is not *scoped*, so their lifetime is controlled by whether anybody is using them.

Stack & heap

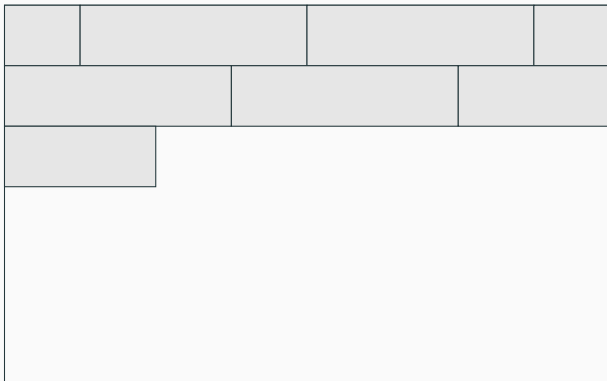


Allocate();

void Allocate()
{
 char stack;
 char[] heap = new char[1];
}

1. We can see this affect when we return from the method call. Here the stack allocation has been automatically freed, but the heap allocation is still around.
2. Stack allocations are always unallocated in the opposite order to their allocation, but heap objects may have completely different lifetimes independent to their allocation order.
3. Eventually the .NET runtime will see that this memory is no longer being used and will free it. This process is called *garbage collection*.

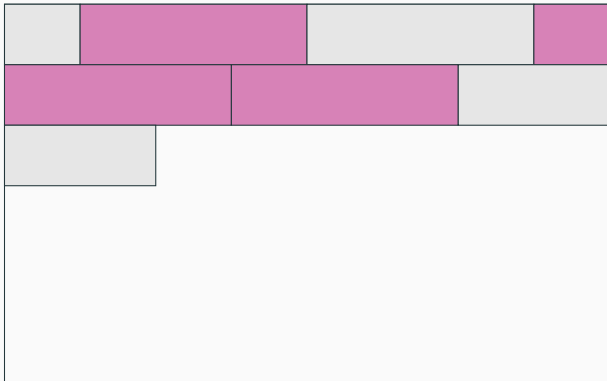
Garbage collection



1. Let's take a look at how garbage collection is performed by the .NET runtime.
2. We'll take the allocations shown as an example of the different steps in the process.
3. This is a simplification, and the real runtime has a number of differences to this, but it is a close enough model that works when debugging memory issues.

Garbage collection

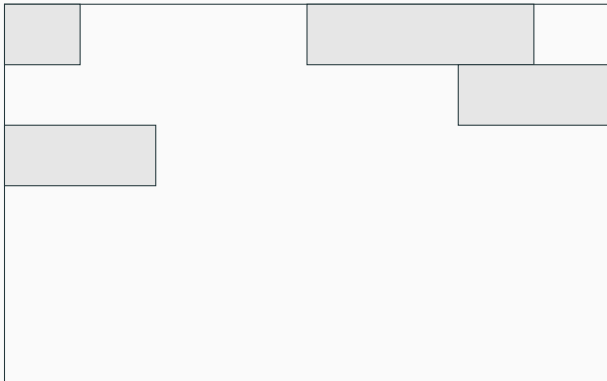
mark



1. The first step is to find all those allocations (normally objects or arrays) that aren't being used any more.
2. These objects are *marked* for deletion in this stage.

Garbage collection

sweep



1. There are two options for the second stage, depending on whether we want a quick garbage collection or a full garbage collection.
2. The quickest option is to *sweep*, which means to deallocate all those allocations that were marked in the first stage.
3. This is quite quick to perform, but has a disadvantage that the memory is now fragmented, as you can see. This makes it harder to perform other allocations in the future, because they need to fit in the gaps left by the sweep.

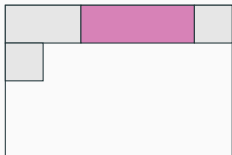
Garbage collection

compact



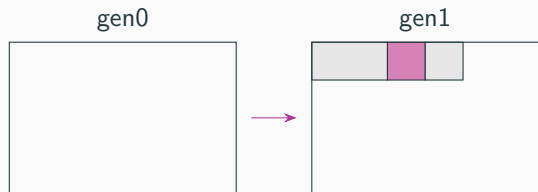
1. The alternative to sweep is to *compact*. This is where we shuffle all the used objects down so there aren't any gaps. This involves updating all references to those objects which takes a long time.
2. Compacting is slow, but leaves the memory unfragmented. In practice .NET does a mix of mark-sweep and mark-compact depending on the current memory state.

Garbage collection



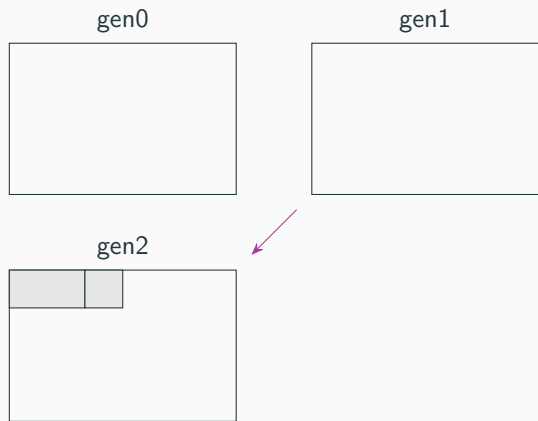
1. There's one other aspect of garbage collection that we should look called *generations*. At first this doesn't seem particularly important, but the more you learn about garbage collection the more relevant generations become.
2. Let's take the example shown here where there's a single item marked for collecting during garbage collection. For now let's not worry whether we are using mark-sweep or mark-compact.

Garbage collection



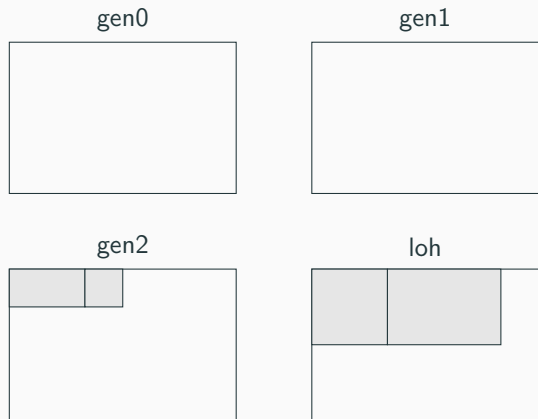
1. When the garbage collector runs, any objects in `gen0` (generation 0) that aren't garbage collected are 'moved' to `gen1`.
2. Don't worry for now about whether these are separate or shared parts of physical memory—the reality is somewhat complicated—but we can treat them as if they are physically separate.
3. `gen0` and `gen1` have a certain amount of space in them (they have their own quotas).

Garbage collection



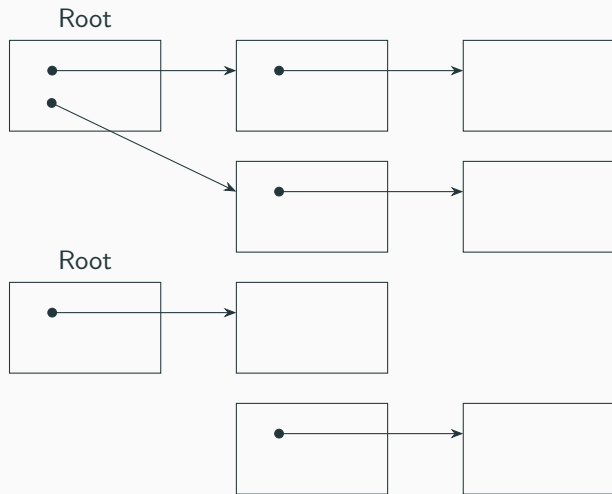
1. When garbage collection runs again, anything in `gen1` that is still alive is moved to `gen2` as you might expect. This also has its own quota.
2. Can you guess what happens if we run garbage collection again?

Garbage collection



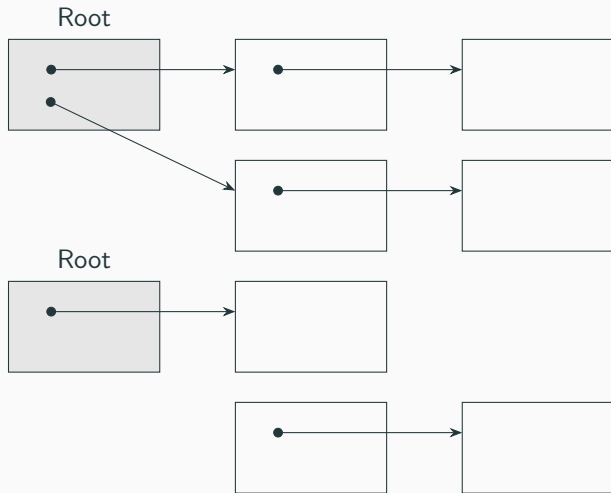
1. Actually, `gen2` is the last storage, and is used for anything that's long-lived.
2. Things that go into `gen2` are either deleted or stay in `gen2`.
3. To complicate matters slightly, you will see there is also a separate area marked `loh` which refers to the 'large object heap'. This stores all large objects allocated, regardless of how many generations they have been around for.

GC roots



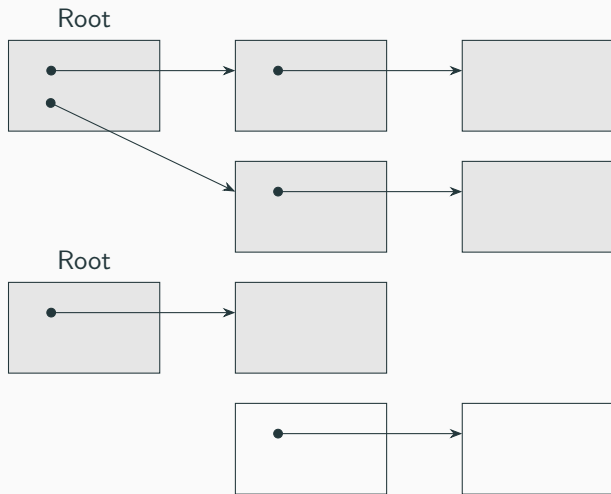
1. The last bit of theory that's worth looking at is how the 'mark' algorithm decides whether an object should be deallocated.
2. Let's imagine we have some objects that refer to other objects (that is, they have a reference in an instance variable to that object). We don't have any cycles here, but it doesn't significantly changes things if we did.
3. Some of these objects are 'root' objects. These are special objects that the garbage collector knows you definitely want. Mostly they are objects referenced in the stack somewhere, or by a static variable.

GC roots



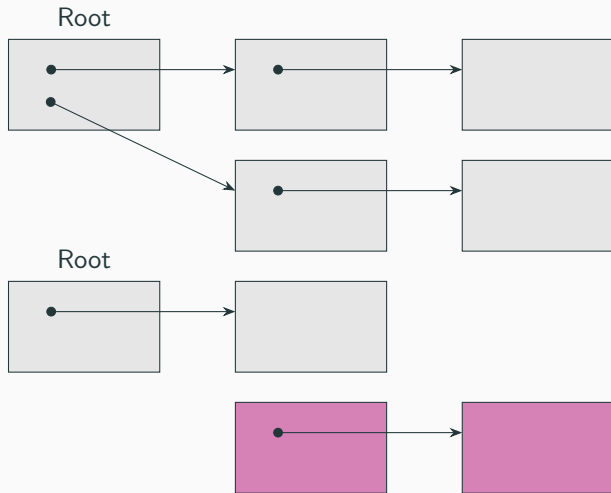
1. The garbage collector knows that you need the root objects, so it knows these shouldn't be marked.

GC roots



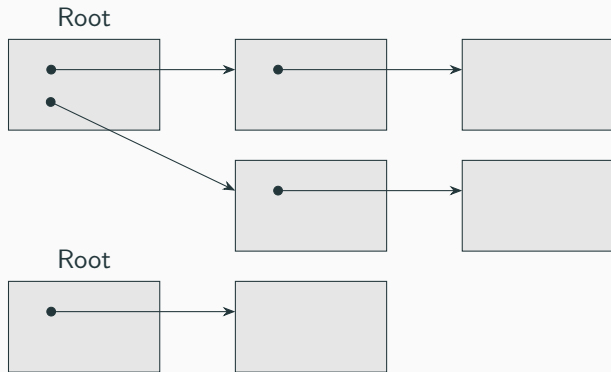
1. It can also work out that you need any objects that these root objects refer to.
2. This forms a graph, and the garbage collector just needs to explore all nodes in the graph to find all nodes that are still needed.

GC roots



1. Finally, the mark algorithm can mark anything that hasn't been identified as being needed.

GC roots



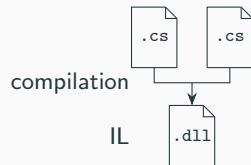
1. And then these marked objects can be deallocated in the sweep or compact stage.

- First show code for SimpleLeaker and point out in dotMemory the memory allocations over time and in the generations (including large object heap).
- After the large object heap allocation force a garbage collect and take a snapshot. Then after the two subsequent clears force a garbage collect. Point out retention in stack storage.
- Demonstrate overview, dominators, similar retention (showing GC roots).
- Next show CodeRunner. Show the dominators point to the statement cache. Open objects retained by that retention path and show the similar retention path.
- Point out that we should try and work out why `OracleInternal.Common.StatementCache` is retaining all the `CachedStatement` instances.

Why?

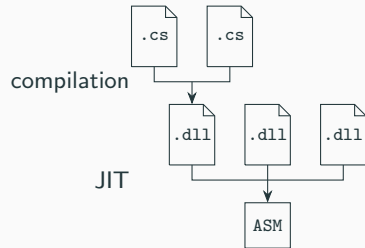
- Now that we've found where the memory leak is, we need to understand why the Oracle library is holding on to all the statements.
- To do that we'll first review how compilation happens in .NET, so we can see how we can reverse this process in order to understand third-party code we don't have the source code for.
- This process is called 'reverse engineering'.
- Again, we'll see this process for our simple application (that we do have the source for) before looking at the Oracle library used by CodeRunner.

Common Language Runtime



1. In C#, the compiler will take multiple source files that are part of a project, and compile them into something called an assembly. In recent versions of .NET, this always has the extension `.dll`.
2. This collection of compiled code is in a platform-independent format called *intermediate language*. In fact, the compiled code still retains much of the original structure such as classes and variable names even.
3. The purpose of the intermediate language is to make the code cross-platform so it can be run on a different operating system, and hardware. The language is higher-level than a CPU, but lower level than languages like C#.

Common Language Runtime



1. At runtime, all the different assemblies that are needed to run the program are collected together by the runtime and compiled into machine code for the particular hardware platform.
2. This second compilation step is normally done only when the code is needed. So some functions will be compiled to machine code, but other functions will be left until they are needed. This process is called *just-in-time compilation*.
3. The key to all this process is that we can take any file in the intermediate language (IL) and attempt to reverse the compilation process. Because much of the structure of the original code is retained in IL, this decompilation step is quite accurate and readable.

- Show how dotPeak works on SimpleLeaker. Demonstrate the source view when source is available, the decompiled sources, and the IL view.
- Next open up the `Oracle.ManagedDataAccess.dll` and find `OracleInternal.Common.StatementCache`.
- Find references to `m_maxCacheSize` and then the constructor for `StatementCache`. Then see where `recommendedScs` comes from and how it is set.
- In particular pay attention to the last reference in `OraclePoolManager.Initialize` and how the `ProviderConfig` gets the value for `MaxStatementCacheSize`.

Case study

The 'bug' is 'fixed' with

```
MaxStatementCacheSize = 50
```

- Setting the `MaxStatementCacheSize` to a lower value reduced the statement cache without making a significant difference to the performance of database queries since the statements weren't able to be reused in most cases.
- The bug was ultimately fixed by a simple database connection configuration change.
- One of the differences between the old version and the new was an update to the Oracle .NET package. But that by itself wouldn't easily point to what the solution was—memory dump analysis and reverse engineering are useful skills to help understand difficult bugs.

How?

I'll finish with a few suggestions on additional tools that can help as well as where to go to learn more about all of this.

Tools

Memory analysis

- dotMemory
- dotnet-dump
- dotnet-counters
- PerfView

Decompilers

- dotPeak
- ildasm
- ILSpy
- dnSpy

- dotnet-counters gives live summary statistics for the memory management.
- dotnet-dump allows you to collect dumps and analyse them similarly to dotMemory but using a command line and with more detail.
- PerfView is a tool from Microsoft that does everything dotMemory can do and a lot more. It is probably the most complete performance analysis tool available (particularly for .NET). It is unfortunately not widely used, in part because it has a steep learning curve. It does have excellent documentation, however.
- ildasm is the original disassembler for the .NET intermediate language. It is a command-line tool that is low frills and mainly useful for batch processing.
- ILSpy is similar to dotPeak but free and open source.
- dnSpy can also debug processes which can be useful to see the intermediate language during debugging. However, it isn't really an intermediate language debugger as it doesn't give information that would be useful (such as the IL stack state)—these don't seem to exist, unfortunately.

Tools

Debuggers

- Visual Studio debugger
- dnSpy
- WinDbg

Honourable mention

- SysInternals

- WinDbg is *the* Windows debugger. For .NET it combines the power of a usual debugger with the memory information from dotnet-dump, but live. It shares the same tools as dotnet-dump using what is called SOS.
- SysInternals is worth mentioning because it is a collection of tools that are immensely useful for debugging all kinds of issues on Windows. If you don't already know about these, you should learn.

[1] Tess Ferrandez-Norlander. **CSI .NET – Debugging .NET Applications**. 2021. URL:

<https://www.youtube.com/watch?v=z2B80ruMT6s> (visited on 11/07/2023).

[2] Maoni Stephens. **mem-doc repository**. 2023. URL:

<https://github.com/Maoni0/mem-doc> (visited on 11/07/2023).

- [1] is a presentation that goes over more ground than this talk but in a very accessible way. It was part of a JetBrains .NET conference a few years ago.
- [2] contains a wealth of information from the .NET GC architect. In particular there's a how-to guide on memory performance analysis that goes into an enormous amount of information.