During the meeting I will show how to allocate memory directly by hand using TypedReferences. We will see that it is possible to allocate reference type on a stack, how to implement List faster than default implementation provided by .NET Framework, how to implement custom memory allocator (with any fragmentation strategy we like), and finally, how to hack new operator provided by the platform. We will see memory dumps with WinDBG, emit IL code dynamically, and generate direct machine code for x86 in runtime.

Adam Furmanek – Speaker – Scala first choice, but not that proficient. Side projects in .NET

Running query 4-5 hours

.NET developer for 5 years

Scala developer at Amazon

I am Adam Furmanek and I recently moved to Seattle from Europe. I have been working as a .NET developer for 5 years and right now I am at Amazon working with Scala and Spark. I am always interested in digging deeper, exploring machine code and going through implementation details. In my free time I play ping-pong, watch Woody Allen's movies and blog stuff at <http://blog.adamfurmanek.pl>

Scala - Piles down to Java byte code. originally a compiler to CLR and …

CLR has more power but can’t do stuff.

Vijay Subramani [vsubramani@google.com](mailto:vsubramani@google.com) - Google Cloud representative

Aims to run .NET on Google Cloud Platform (GCP)

Some of Googles app is made in .NET

Next month meetup .NET Core on Google Cloud Platform by Ian Talarico – Google Kirkland

**Manual memory management**

C# programmer:

“Value types are on the stack, reference types are on the heap.”

Not entire truth, value types are also stored on the heap.

Reference type can also be stored on the stack

Use the unsafe keyword to do manual memory management

World of unsafe

Agenda

.NET internals

Managed object on a stack

Unsafe list

Custom memory allocator

Hijacking new summary

Whenever we create a new object, it will use OUR custom memory allocator

Useful for debugging memory leaks, etc. for .NET application / CLI process

ECMA Standard ISO/IEC 23270

If M is an instance function member declared in a reference-type for an instance constructor this evaluation consists of allocating (typically from a garbage-collected heap) the storage of the new object.

If M is an instance function member declared in a value type for an instance constructor this evaluation consists of allocating (typically from an execution stack) for the new object.

“Erm, What?” – answering precise interview question

**.NET Internals**

Object structure (how do we get pointer? *Object should be clean and reliable*)

Allocation

\_refvalue + \_makeref + history

Object Structure

var o = new object();

o is just a pointer

object is only a data – not more

manager pointer

-> RTTI address - > Interface Map Table - > Interface1 Address

Memory Laid out for Object or Reference Types

0x0050

|  |
| --- |
| Managed pointer (0x0020) |

Size: (depends)

|  |
| --- |
| Sync block address (0x0000) (stored before the object) |
| RTTI address (0x0020) (use reflection) (another pointer 4/8 bytes) |
| Field 1 |
| Field 2 |
| Field N |

Reference Type instance in GC Heap

Method table structure for a Runtime Type

|  |
| --- |
| Interface Map Table Address (0x04C3E) |
| Inherited virtual method |
| New virtual method address |
| Static method address |
| Static Field 1 N |
| Interface Method Address |

Appdomain …..

|  |
| --- |
| Interface address1 |
| Interface address2 |

**\_makeref**

The mkrefany instruction supports the passing of dynamically typed references. The pointer must be of type &, \* or native int, and hold the valid address of a piece of data

Class is the class token describing the type of the data referenced by the pointer.

Mkrefany pushes a typed reference on the stack, providing an opaque descriptor of the pointer and the type class

“First read the object, write it in the stack, then point, point”

“We cannot get the pointer to the object”

“Before .NET 2, there were no generics, we have to accept what objects, boxing is slow”

“through levels of indirection, can access.”

\_refvalue

The refanyval instruction retrieves the address embedded in the a typed reference.

The type embedded in the typed reference supplied on the stack must match ……

**indirection**

65 (5000) <- 5000 (6000) <- 6000 (7000) <- 7000 (8000)

Actual data reference typed reference pointer

New object() object o = … \_makeref(o)

GC “Facts”

Facts  
there are three generations: 0, 1, and 2

Large object heap contains objects having at least 85000 bytes (only?)

Large object heap is in generation 2

Questions:

Which generation holds a stack?

Managed object on a stack *demo 1*

Class

Virtual method

Destructor

New object

Create reference to object

Reference pointing to original object

Allocate memory to stack

4 levels of indirection

Subtract 4 for sync bytes

Check in WinDebug App – can use in production house, make a heap dump/memory dump

“Whenever you want to debug something, you want to have a support on the platform you are using”

“the only thing the visual studio does is just a UI for debugging”

“Son of Strike? (SOS)” only available in VS 2010

“compiled against the … internals”

“can rely on .NET support instead”

“Code name of CLR before Is lightning, early plugin named “Strike””

“wrote another plugin Son of Strike”

“wrote a new plugin son of strike extended”

!clrstack – a -i

Output:  
Local:  
makeref.poco heap:Poco @ 0x3282848

Makeref.poco OriginalPoco @ 0x3282848

“same memory location”

“put fields in array, put array in stack”

Output:  
0 for sync block

“rewriting thread”

“dumping memory”

“Set reference to stack”

“with pointers, we gone to the reference, and not the object”

4 bytes later to the sync block

makeref.poco heap:Poco @ 0x10fee58 (now points to the stack)

Makeref.poco OriginalPoco @ 0x3282848

“new reference, pointing to the object”

“call a method that is nonvirtual, prints the field”

Output:  
Heap: 5

Stack: 5

“modifying values using references”

“display type of object”

“object in stack is in generation 2” – guess is whatever address that doesn’t fit in generation 1 could be placed in generation 2

“Garbage collection starts with generation 0 and 1”

“check if we can perform locking”  
“lock can be release”

“tell GC, if have this object, please clean it up, not a good idea as it is on the stack, could crash”

“Copying heap to the stack, pointed to the stack”

Unsafe list (demo 2)

“Copied the object to any place we choose”

Ordinary list – does not store the objects

PTR pTR pTR - > object member 1,2,3

L

V

Object memb 1, 2, 3

Unsafe list

“copies the whole list in the internals”

UnsafeList == std::vector<object>

Object - > objec

Mem 1 mem 1

Mem 2 mem2

“Poco class with 16 integers (64 bytes, and 4 byes for RTTI, and 4 bytes sync block)

“create stopwatch for test”

“stop the stopwatch, print the result”  
“pre allocate the list so that it does not expand everytime”

“we provide the size of the list”

“1 is an array of integers”

“when adding the item”

“get the reference of the array”

“update some bookkeeping to update the index data”

“adding to the array is slower than iterating“

“for unsafe list is 3x faster, than adding to the list of T”

“unsafe list is faster why? don’t know, benchmark your code”

“addition should be slower and iteration should be faster”  
“iterations are faster because of the CPU cache lines, why? don’t know”

“copied the objects, but unsafe is still faster. It could have optimized the code”

**IL allocator demo 3**

“what we don’t like is copying the object”

“allocate the object completely from scratch”

“use lambda not as a func, but as an expression of the func of T”

“will not allocate the object”

“how to get RTTI address? Use reflection”

“how to collect constructor? 2 ways, either reflection (call constructor then invoke), serializers (without calling the constructor) or get aisle code on the fly

**Custom new operator demo 4**

“we don’t like providing the lambda”

Allocation – newobj

The newobj instruction:

Allocates a new instance of the class associated with ctor

initialize all the fields in the new instance to 0 (of the proper type) or null references as appropriate

Calls the constructor ctor

After calling the constructor the initialize object reference (type 0) is pushed on the stack

“create an allocator”

“then allocate the object using the allocator”  
“then allocate new object using new built in operator”  
“hijack the new object”

Output of non hijack

Generation 0

Output of hijack

Generation 2

“check what happens if code the new operator”

“we done 2 things, code the method with CORINFO\_HELP\_NEWSFAST, then we call the constructor”

“hey you have space in Page? Yes? Allocate space”

“if instance method, needs 2 parameters, since we don’t need pointer, it must be static”

“pointer to pointer to pointer to reference”

“either our method or .net method”

“modify .net method to jump to our method”

“Output of generation 2 means we actually modify the method”

Jeffrey richter – clr via C#

Jeffrey richter – windows via c/C++

Mark Russinovich – windows internals

Penny orwick – developing drivers with the Microsoft Windows Driver Foundation

Mario Hewardt – advanced windows debugging

Mario hewardt – advanced .NET debugging

Steven Pratschner – customizing the Microsoft .NET Framework CLR

**Serge Lidin – Expert .NET 2.0 IL Assembler**

Joel Pobar – Shared source CLI 2.0 internals

<https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md> - book of the runtime

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

channel9.msdn.com/Shows/Defrag-Tools