.NET GC Internals

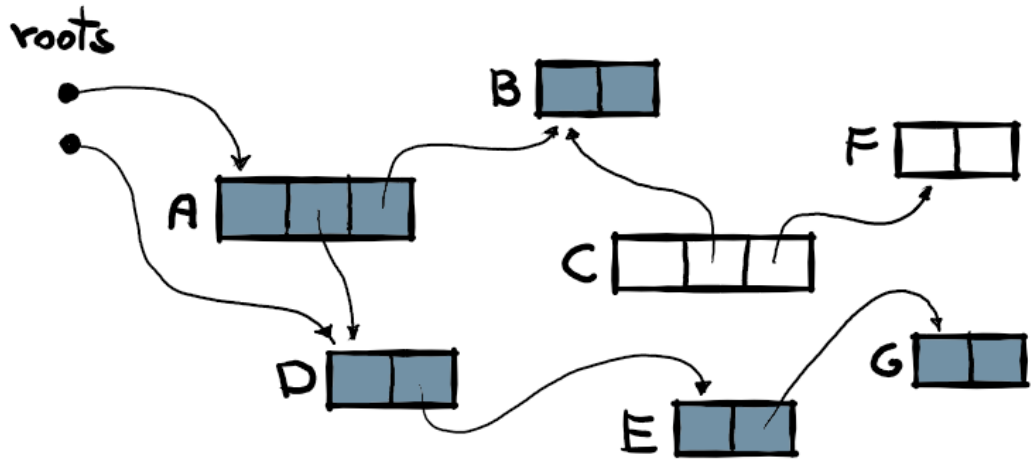
GC roots

@konradkokosa / @dotnetosorg

.NET GC Internals Agenda

- Introduction - roadmap and fundamentals, source code, ...
- **Mark** phase - roots, object graph traversal, *mark stack*, mark/pinned flag, *mark list*, ...
- **Concurrent Mark** phase - *mark array/mark word*, concurrent visiting, *floating garbage*, *write watch list*, ...
- **Plan** phase - *gap*, *plug*, *plug tree*, *brick table*, *pinned plug*, *pre/post plug*, ...
- **Sweep** phase - *free list threading*, concurrent sweep, ...
- **Compact** phase - *relocate* references, compact, ...
- **Allocations** - *bump pointer allocator*, free list allocator, *allocation context*, ...
- **Generations** - physical organization, *card tables*, demotion, ...
- **Roots internals** - stack roots, *GCInfo*, *partially/full interruptible methods*, statics, Thread-local Statics (TLS), ...
- **Q&A** - "but why can't I manually delete an object?", ...

Mark phase recap



Roots:

- local variables - stack/registers
- static/thread-local static data - handles
- finalization - Finalization & fReachable "queues"
- card tables

Local variables

Live Stack Roots vs. Lexical Scope

```
1 private int LexicalScopeExample(int value)
2 {
3     ClassOne c1 = new ClassOne();
4     if (c1.Check())
5     {
6         ClassTwo c2 = new ClassTwo();
7         int data = c2.CalculateSomething(value);
8         DoSomeLongRunningCall(data);
9         return 1;
10     }
11     return 0;
12 }
```

Lexical scope of local variables:

- **c1** - entire method
- **c2** - conditional block
- **data** - part of conditional block (primitive data)

Reachability does not span to the whole lexical scope - Eager Root Collection

- unless in Debug :)

Live Stack Roots - Debug

```
1 private int LexicalScopeExample(int value)
2 {
3     ClassOne c1 = new ClassOne();
4     if (c1.Check())
5     {
6         ClassTwo c2 = new ClassTwo();
7         int data = c2.CalculateSomething(value);
8         DoSomeLongRunningCall(data);
9         return 1;
10     }
11     return 0;
12 }
```

Fully interruptible - suspension possible at each line so each lists live stack roots:

```
1 No live slots
2 No live slots
3 Live slot of c1
4 Live slot of c1
5 Live slot of c1
6 Live slot of c1, live slot of c2
7 Live slot of c1, live slot of c2
8 Live slot of c1, live slot of c2
9 Live slot of c1, live slot of c2
10 Live slot of c1
11 Live slot of c1
```

Partially interruptible - suspension possible at *safe-points* (mostly method calls):

```
3 Live slot of c1
4 Live slot of c1
6 Live slot of c1, live slot of c2
7 Live slot of c1, live slot of c2
8 Live slot of c1, live slot of c2
```

Live Stack Roots - Debug

```
1 private int LexicalScopeExample(int value)
2 {
3     ClassOne c1 = new ClassOne();
4     if (c1.Check())
5     {
6         ClassTwo c2 = new ClassTwo();
7         int data = c2.CalculateSomething(value);
8         DoSomeLongRunningCall(data);
9         return 1;
10     }
11     return 0;
12 }
```

Fully interruptible

```
1 No live slots
2 No live slots
3 Live slots: rax
4 Live slots: rax
5 Live slots: rax
6 Live slots: rax, rbx
7 Live slots: rax, rbx
8 Live slots: rax, rbx
9 Live slots: rax, rbx
10 Live slots: rax
11 Live slots: rax
12 No live slots
```

Partially interruptible

```
3 Live slots: rax
4 Live slots: rax
6 Live slots: rax, rbx
7 Live slots: rax, rbx
8 Live slots: rax, rbx
```

Live Stack Roots - Release

```
1 private int LexicalScopeExample(int value)
2 {
3     ClassOne c1 = new ClassOne();
4     if (c1.Check())
5     {
6         ClassTwo c2 = new ClassTwo();
7         int data = c2.CalculateSomething(value);
8         DoSomeLongRunningCall(data);
9         return 1;
10     }
11     return 0;
12 }
```

Fully interruptible

```
1 No live slots
2 No live slots
3 Live slots: rax (c1)
4 Live slots: rax (c1)
5 No live slots
6 Live slots: rax (c2)
7 Live slots: rax (c2)
8 No live slots
9 No live slots
10 No live slots
11 No live slots
12 No live slots
```

Partially interruptible

```
3 Live slots: rax (c1)
4 Live slots: rax (c1)
6 Live slots: rax (c2)
7 Live slots: rax (c2)
```

Live Stack Roots

- this is so-called **GCInfo** encoded by JIT
  - rather hard to see in any tool - deep implementation detail
  - the only I know is `gcinfo` from SOS

Live Stack Roots - partially interruptible

```
> !u -gcinfo 00007ffea9948598
Normal JIT generated code
LexicalScopeExample(Int32)
Begin 00007ff81c5e3310, size 71
push rdi
push rsi
sub  rsp,28h
mov  esi,edx
mov  rcx,7FF81C69AD08h (MT: ClassOne)
call CoreCLR!JIT_New
0017 is a safepoint:
mov  rdi,rax
mov  rcx,rdi
call System_Private_CoreLib+0xc... (Object..ctor())
0022 is a safepoint:
0021 +rdi
mov  dword ptr [rdi+8],esi
mov  rcx,rdi
call 00007ff8`1c5e2bb8 (ClassOne .Check())
002d is a safepoint:
test eax,eax
je   00007ff8`1c5e3378
...
```

```
int LexicalScopeExample(int value)
{
  ClassOne c1 = new ClassOne();
  if (c1.Check())
  {
    ClassTwo c2 = new ClassTwo();
    int data = c2.CalculateSomething(value);
    DoSomeLongRunningCall(data);
    return 1;
  }
  return 0;
}
```

Live Stack Roots - partially interruptible

```
...
mov  rcx,7FF81C69AFE8h (MT: ClassTwo)
call CoreCLR!JIT_TrialAllocSFastMP_InlineGetThread
0040 is a safepoint:
mov  rdi,rax
mov  rcx,rdi
call System_Private_CoreLib+0xc... (Object..ctor())
004b is a safepoint:
004a +rdi
mov  rcx,rdi
mov  edx,esi
call 00007ff8`1c5e2be0 (CalculateSomething(Int32))
0055 is a safepoint:
mov  ecx,eax
call 00007ff8`1c5e2d70 (DoSomeLongRunningCall(Int32)
005c is a safepoint:
mov  eax,1
add  rsp,28h
pop  rsi
pop  rdi
ret
xor  eax,eax
add  rsp,28h
pop  rsi
pop  rdi
ret
```

```
int LexicalScopeExample(int value)
{
  ClassOne c1 = new ClassOne();
  if (c1.Check())
  {
    ClassTwo c2 = new ClassTwo();
    int data = c2.CalculateSomething(value);
    DoSomeLongRunningCall(data);
    return 1;
  }
  return 0;
}
```

Live Stack Roots - fully interruptible

```
> !u -gcinfo 00007fff42c18518
...
push rdi
push rsi
sub  rsp,28h
mov  esi,edx
00000008 interruptible
xor  edi,edi
mov  rcx,7FFF42DEAAC8h (MT: SomeClass)
call CoreCLR!JIT_TrialAllocSFastMP_InlineGetThread
00000019 +rax
mov  rcx,rax
0000001c +rcx
call System_Private_CoreLib+0x... (Object..ctor())
00000021 -rcx -rax
xor  eax,eax
test esi,esi
jle 00007fff`42d32f54
mov  edx,eax
imul edx,eax
add  edi,edx
inc  eax
cmp  eax,esi
jl   00007fff`42d32f47
mov  eax,edi
00000036 not interruptible
...
```

```
int RegisterMap(int value)
{
  int total = 0;
  SomeClass local = new SomeClass();
  for (int i = 0; i < value; ++i)
  {
    total += local.DoSomeStuff(i);
  }
  return total;
}

int DoSomeStuff(int value) => value * value;
```

Live Stack Roots - stack

```
1 private int LexicalScopeExample(int value)
2 {
3     ClassOne c1 = new ClassOne();
4     if (c1.Check())
5     {
6         ClassTwo c2 = new ClassTwo();
7         int data = c2.CalculateSomething(value);
8         DoSomeLongRunningCall(data);
9         return 1;
10     }
11     return 0;
12 }
```

For the real stack local variable (nor enregistered):

- **`GCInfo`** keeps also **`rsp/rbp`** (the "stack pointer") for every safe-point
- stack slot is represented as offset to it

```
3 Live slots: rax
4 Live slots: rax
6 Live slots: rax, rsp+10
7 Live slots: rax, rsp+10
8 Live slots: rax, rsp+10
```

GCInfo

- GCInfo is stored efficently as *chunks of bits* keeping initial state and incremental changes within
- to decode liveness for an address (*):
  - get a proper chunk
  - start from initial state
  - decode changes step by step until given address
  - (*) SOS's extension do that multiple times for every instruction address
- chunks remove the overhead of a method length - we always analyze single chunk
- more dense GCInfo -> **more** flexible suspension but **bigger** storage overhead
- less dense GCInfo -> **less** flexible suspension but **smaller** storage overhead

Non-interruptible :)

```
static int UseLocalVariables1(SomeClass c1, SomeClass c2, SomeClass c3,
                              SomeClass c4, SomeClass c5, SomeClass c6)
{
    return c1.Result + c2.Result + c3.Result + c4.Result + c5.Result + c6.Result;
}
```

Non-interruptible :)

```
static int UseLocalVariables1(SomeClass c1, SomeClass c2, SomeClass c3,
                              SomeClass c4, SomeClass c5, SomeClass c6)
{
    return c1.Result + c2.Result + c3.Result + c4.Result + c5.Result + c6.Result;
}
```

JIT decided method is short enough not to interrupt it at all.

Non-interruptible :)

```
static int UseLocalVariables1(SomeClass c1, SomeClass c2, SomeClass c3,
                              SomeClass c4, SomeClass c5, SomeClass c6)
{
    return c1.Result + c2.Result + c3.Result + c4.Result + c5.Result + c6.Result;
}
```

JIT decided method is short enough not to interrupt it at all. Although stack slots **+sp+30** and **+sp+28** are reported, there is no single safe point nor interruptible region:

```
Untracked: +sp+30 +sp+28
mov  eax,dword ptr [rcx+10h]
add  eax,dword ptr [rdx+10h]
add  eax,dword ptr [r8+10h]
add  eax,dword ptr [r9+10h]
mov  rdx,qword ptr [rsp+28h]
add  eax,dword ptr [rdx+10h]
mov  rdx,qword ptr [rsp+30h]
add  eax,dword ptr [rdx+10h]
ret
```

Full vs partially interruptible

Partially interruptible:

- smaller storage
- usually "good enough"
- slower suspension

Fully interruptible:

- fast suspension
- storage overhead

In summary:

- yes, it is an implementation detail
- we have only indirect control:
  - no method calls -> fully-interruptible
  - non-trivial loops with a dynamic number of iterations -> fully-interruptible
  - ...
- JIT has "heuristics" (AKA "magic") to decide

Pinned Local Variables

```
static int UsePinnedArray(byte[] array, int index)
{
    SomeClass local = new SomeClass();
    fixed (byte* pointer = array)
        local.AddSample(pointer[index]);
    return local.Result;
}
```

```
Untracked: +sp+28(pinned)
...
xor  eax,eax
mov  qword ptr [rsp+28h],rax
mov  rsi,rcx
mov  edi,edx
mov  rcx,7FFBF5ED4288h (MT: SomeClass)
call coreclr!... (JitHelp: CORINFO_HELP_NEWSFAST)
00000022 is a safepoint:
00000021 +rsi
...
mov  qword ptr [rsp+28h],rsi
test rsi,rsi
je   00007ffb`f5e0631a
...
mov  rax,qword ptr [rsp+28h]
add  rax,10h
movsxd rdx,edi
mov  eax,dword ptr [rax+rdx]    ; eax = pointer[index]
...
```

Handles

Handles

Strong handles - like normal references (keeping object reachable/alive):

```
var normal = new Normal();
GCHandle handle = GCHandle.Alloc(normal, GCHandleType.Normal);

public class Normal
{
    public long F1 = 201;
}
```

Handles

Strong handles - like normal references (keeping object reachable/alive):

```
var normal = new Normal();
GCHandle handle = GCHandle.Alloc(normal, GCHandleType.Normal);

public class Normal
{
    public long F1 = 201;
}
```

Pinned handles - strong handles with pinning behaviour (object will not be moved):

```
var blittable = new Blittable();
GCHandle handle = GCHandle.Alloc(blittable, GCHandleType.Pinned);
// handle.AddrOfPinnedObject()

[StructLayout(LayoutKind.Sequential)]
public class Blittable
{
    public long F1 = 301;
}
```

Handles

Strong handles - like normal references (keeping object reachable/alive):

```
var normal = new Normal();
GCHandle handle = GCHandle.Alloc(normal, GCHandleType.Normal);

public class Normal
{
    public long F1 = 201;
}
```

Pinned handles - strong handles with pinning behaviour (object will not be moved):

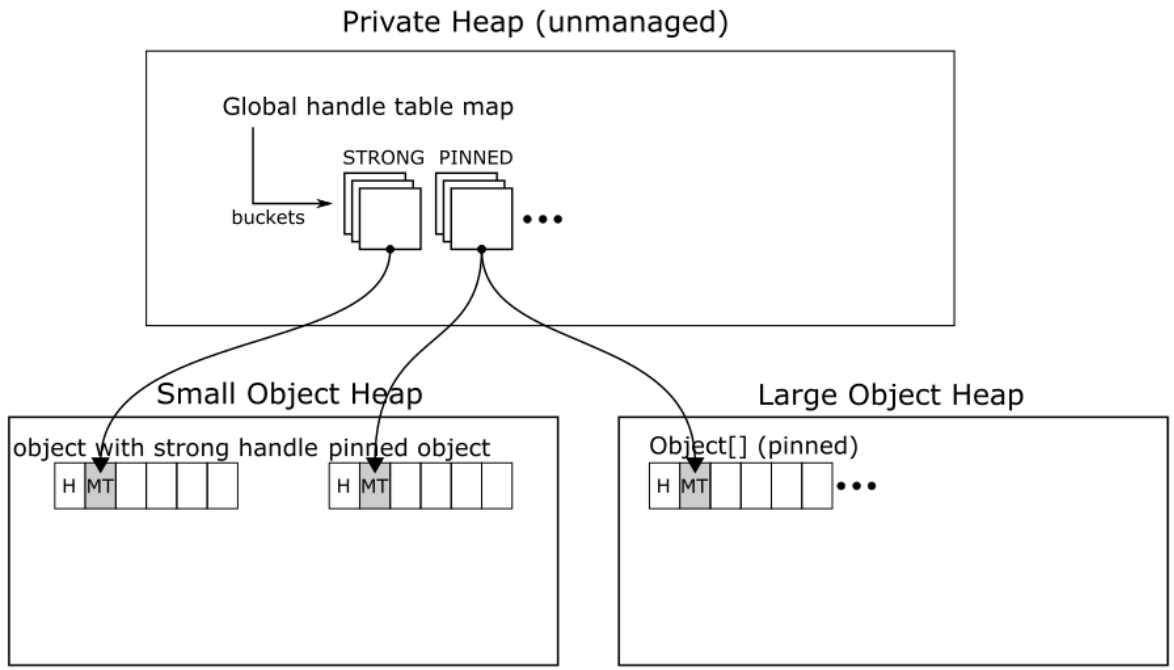
```
var blittable = new Blittable();
GCHandle handle = GCHandle.Alloc(blittable, GCHandleType.Pinned);
// handle.AddrOfPinnedObject()

[StructLayout(LayoutKind.Sequential)]
public class Blittable
{
    public long F1 = 301;
}
```

PS. There are also internal *async pinned handles* - with as-fast-as-possible release feature.

Handles

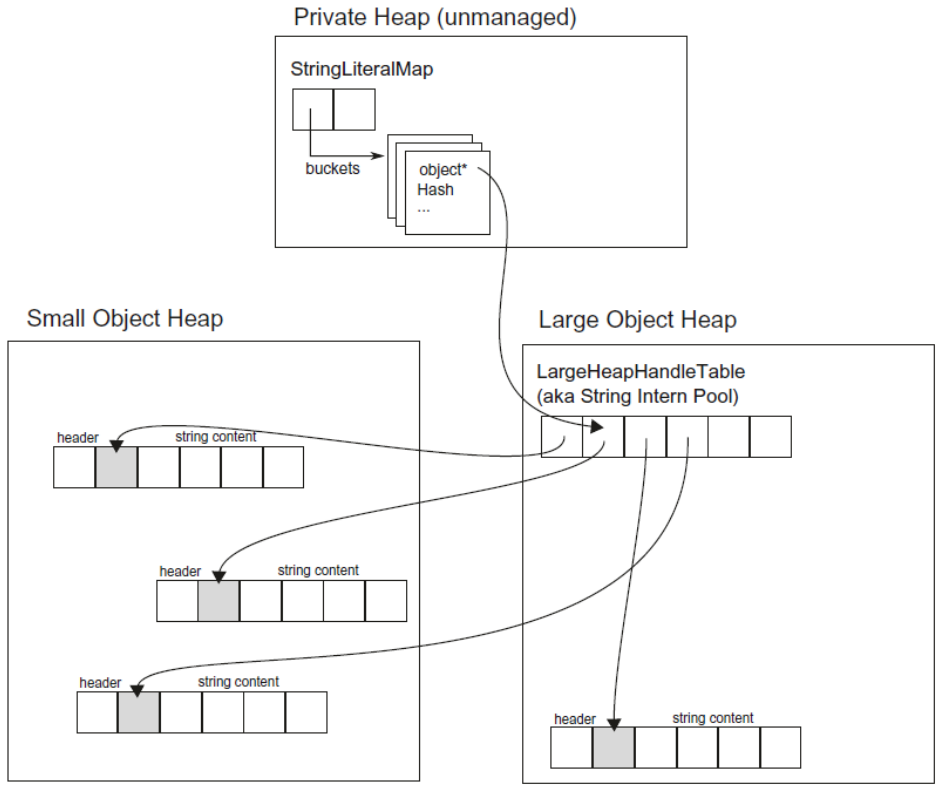
Internally they are bucketed arrays of pointers - treated as roots during Mark phase:



Handles - string interning example

```
public static void Method5()
{
    Console.WriteLine("Hello world!");
}
```

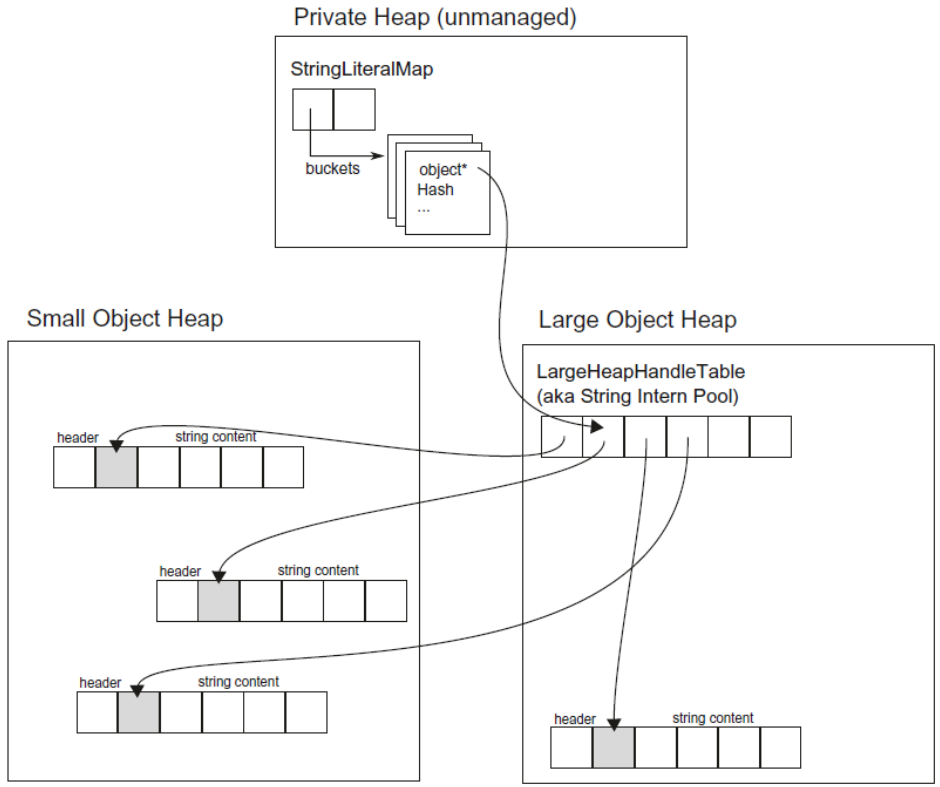
```
sub  rsp, 0x28
mov  rcx, 0x22ca9f99d98
mov  rcx, [rcx]
call System.Console.WriteLine(System.String)
nop
add  rsp, 0x28
ret
```



Handles - string interning example

```
public static void Method5()
{
    Console.WriteLine("Hello world!");
}
```

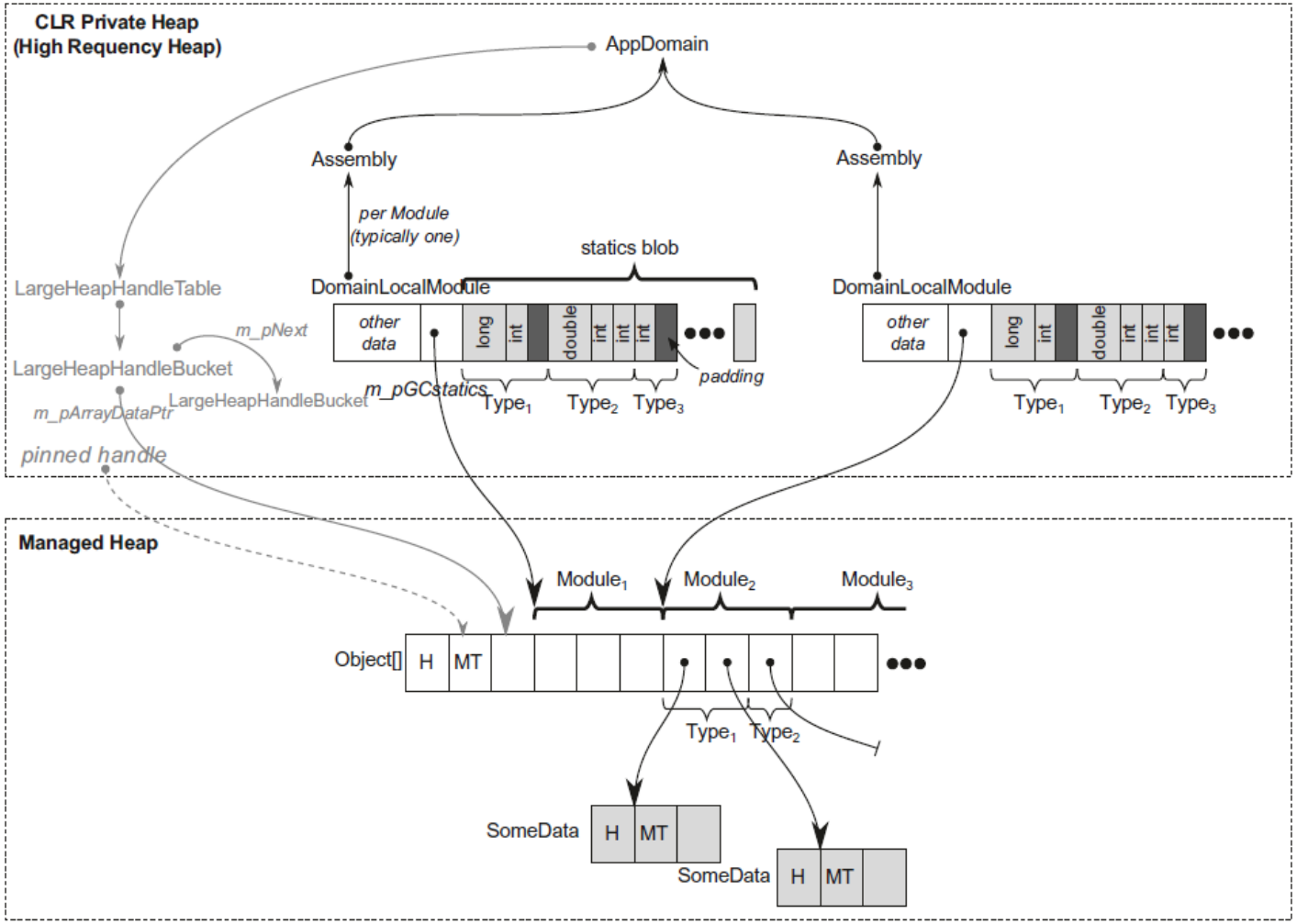
```
sub  rsp, 0x28
mov  rcx, 0x22ca9f99d98
mov  rcx, [rcx]
call System.Console.WriteLine(System.String)
nop
add  rsp, 0x28
ret
```



Who keeps **LargeHeapHandleTable** alive...?

Statics & Thread Local Statics

Static roots



Static roots

```
public class Type1
{
    public static int StaticPrimitive;
    public static R SomeData = new R();
}

public class R
{
    public int Value;
}

public static void Method()
{
    Console.WriteLine(Type1.SomeData.Value);
}
```

Static roots

```
public class Type1
{
    public static int StaticPrimitive;
    public static R SomeData = new R();
}

public class R
{
    public int Value;
}

public static void Method()
{
    Console.WriteLine(Type1.SomeData.Value);
}
```

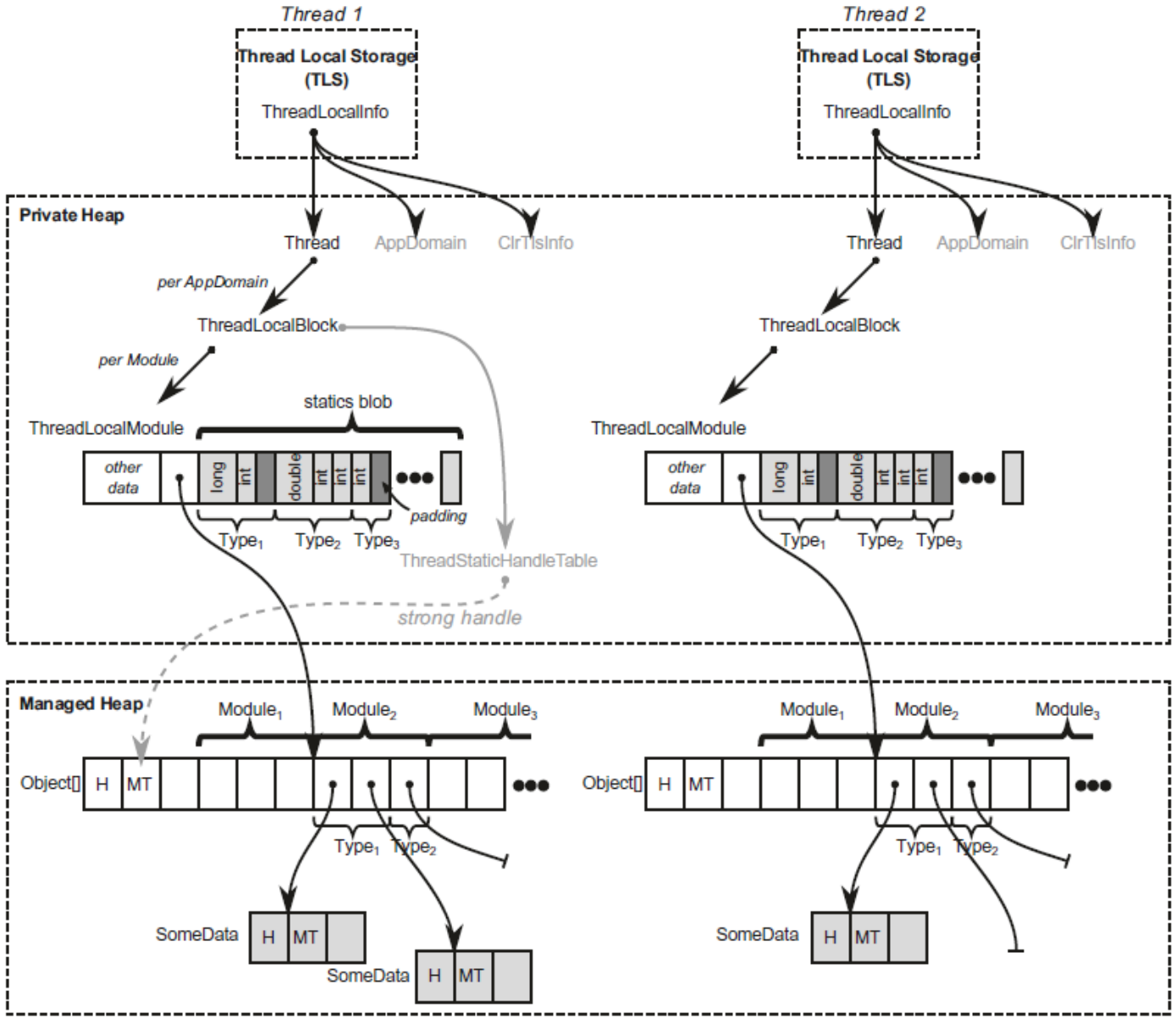
```
sub  rsp, 28h
mov  rcx, 154F1222C88h        ; addres in LOH - LargeHeapHandleBucket (pinned by handle)
mov  rcx, qword ptr [rcx]
mov  ecx, dword ptr [rcx+8]
call System.Console.WriteLine(Int32)
...
add     rsp,28h
ret
```

Threal-local Static roots

```
class SomeData
{
    public int Field;
}

class SomeClass
{
    [ThreadStatic]
    private static int threadStaticValueData;
    [ThreadStatic]
    private static SomeData threadStaticReferenceData;
}
```

- uses per-thread storage: *Thread Local Storage (TLS)* (Windows) and *Thread-specific data* (Linux)
- storage is super small - 64-1088 pointer-sized *slots* on Windows
- we don't keep **data** there - we keep pointer to thread-specific data


Thread 1
Thread Local Storage (TLS)
ThreadLocalInfo
Thread 2
Thread Local Storage (TLS)
ThreadLocalInfo
Private Heap
Thread
AppDomain
ClrTlsInfo
Thread
AppDomain
ClrTlsInfo
per AppDomain
ThreadLocalBlock
ThreadLocalBlock
per Module
statics blob
ThreadLocalModule
ThreadLocalModule
other data
long
int
double
int
int
int
padding
other data
long
int
double
int
int
int
Type1
Type2
Type3
Type1
Type2
Type3
ThreadStaticHandleTable
strong handle
Managed Heap
Module1
Module2
Module3
Module1
Module2
Module3
Object[]
H
MT
Object[]
H
MT
Type1
Type2
Type1
Type2
SomeData
H
MT
SomeData
H
MT
SomeData
H
MT

Finalization

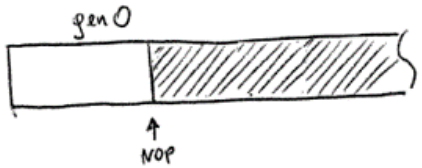
Finalization

```
class SomeClass
{
    ~SomeClass()
    {
    }
}
```

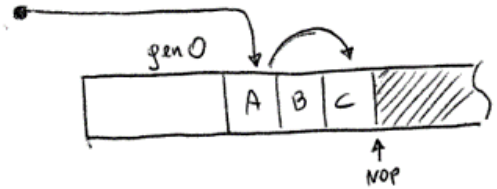
Finalization

```
class SomeClass
{
    ~SomeClass()
    {
        Thread.Sleep(100_000_000);
    }
}
```

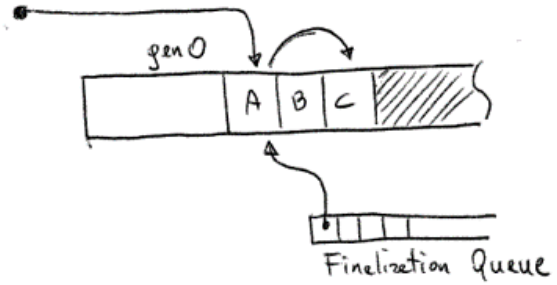
Finalization - Finalization queue, fReachable queue



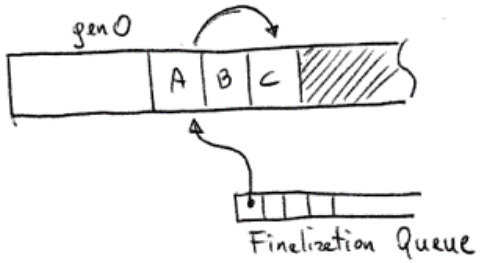
Finalization - Finalization queue, fReachable queue



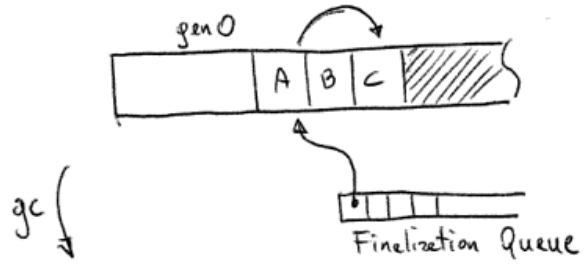
Finalization - Finalization queue, fReachable queue



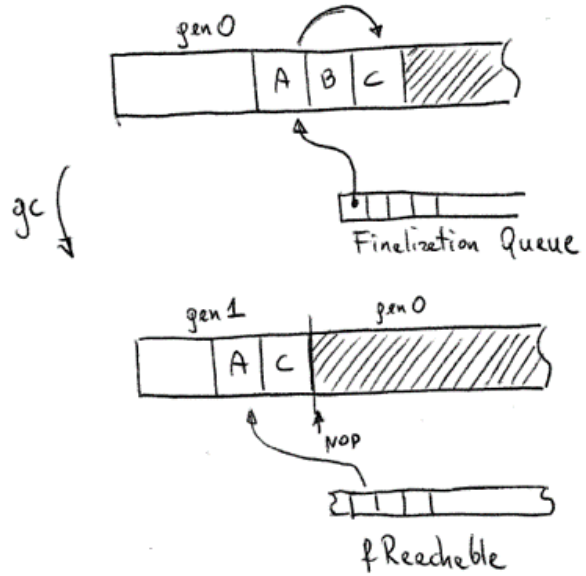
Finalization - Finalization queue, fReachable queue



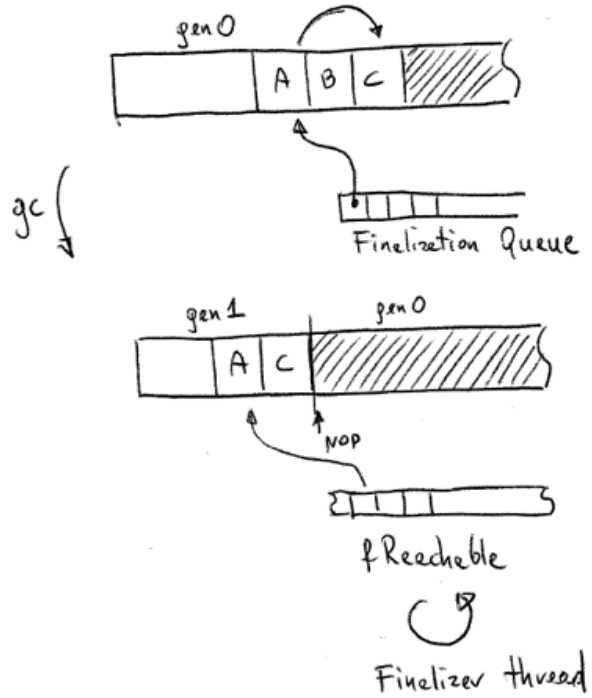
Finalization - Finalization queue, fReachable queue



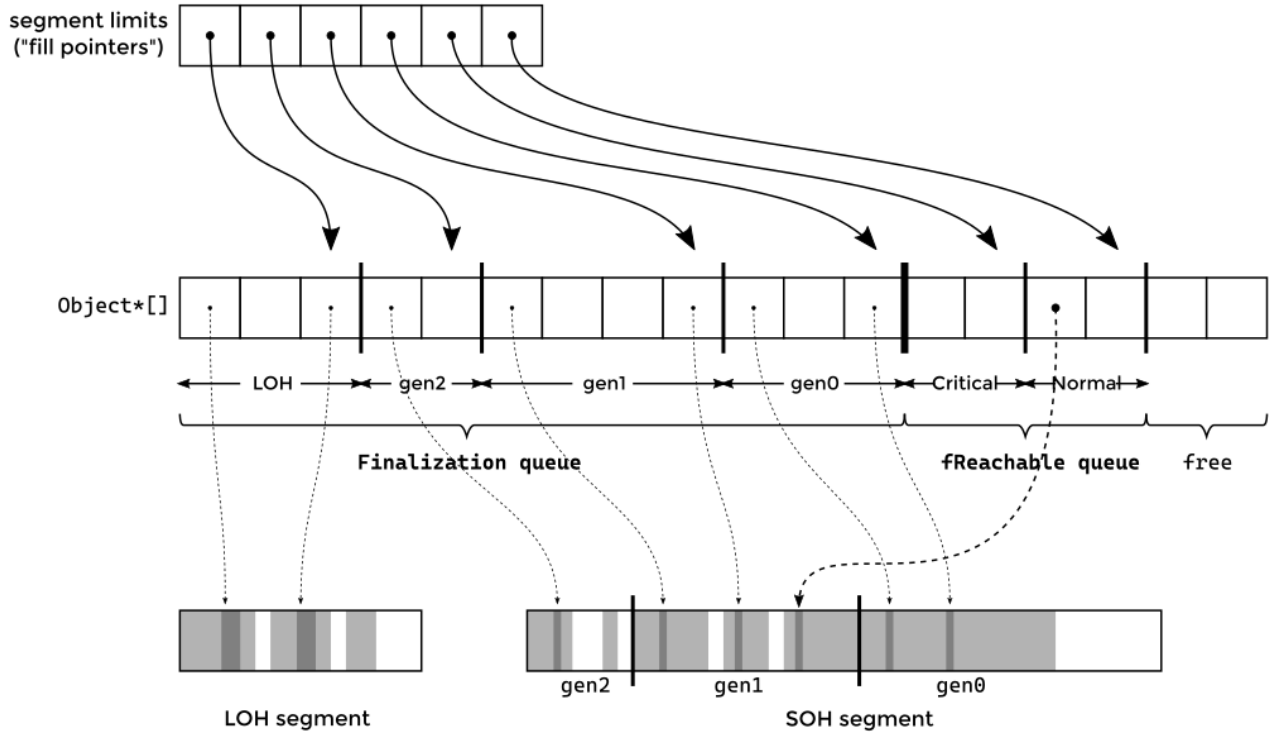
Finalization - Finalization queue, fReachable queue



Finalization - Finalization queue, fReachable queue



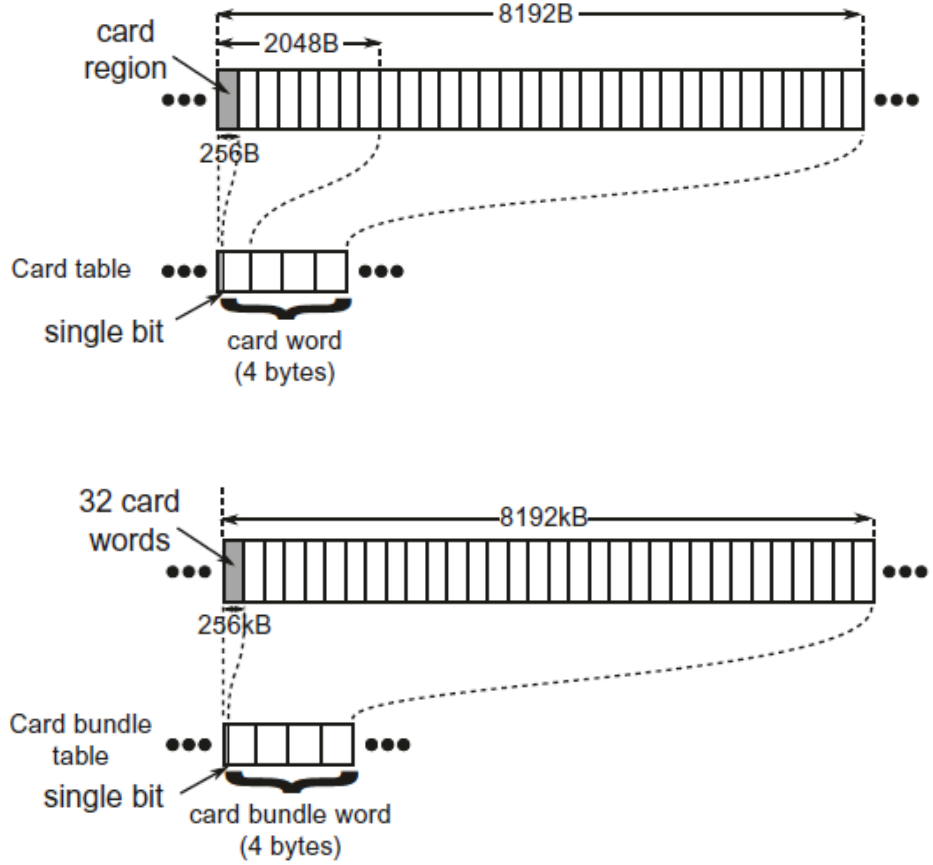
Finalization - internals



- they aren't queues but a single array (with a lock)!
- "register for finalization" - insert pointer to *gen0* (and... shift *Critical* & *Normal*)
- promote/demote an object - update *fill pointers* and/or copy pointer (and yes, shift other elements)

Card tables

Card tables


8192B
card region
2048B
256B
Card table
single bit
card word
(4 bytes)
32 card words
8192kB
256kB
Card bundle table
single bit
card bundle word
(4 bytes)