MANAGED MEMORY LEAK INVESTIGATION

Workshop

Michael Yarichuk

Possible symptoms of memory leaks

- Slowdowns
- Execution pauses (freezes)
- High resource usage (CPU/RAM)
- Obviously – constantly growing memory!

This workshop is about working in Windows!

Working with dumps on Linux requires usage of different tools, but *roughly* works the same

A word (or two) about garbage collector

Part 1

GC Algorithm Types

- Manual - allocate/release via language API
- Reference counting – allocate via language API, release via reference count

Used by the CLR →

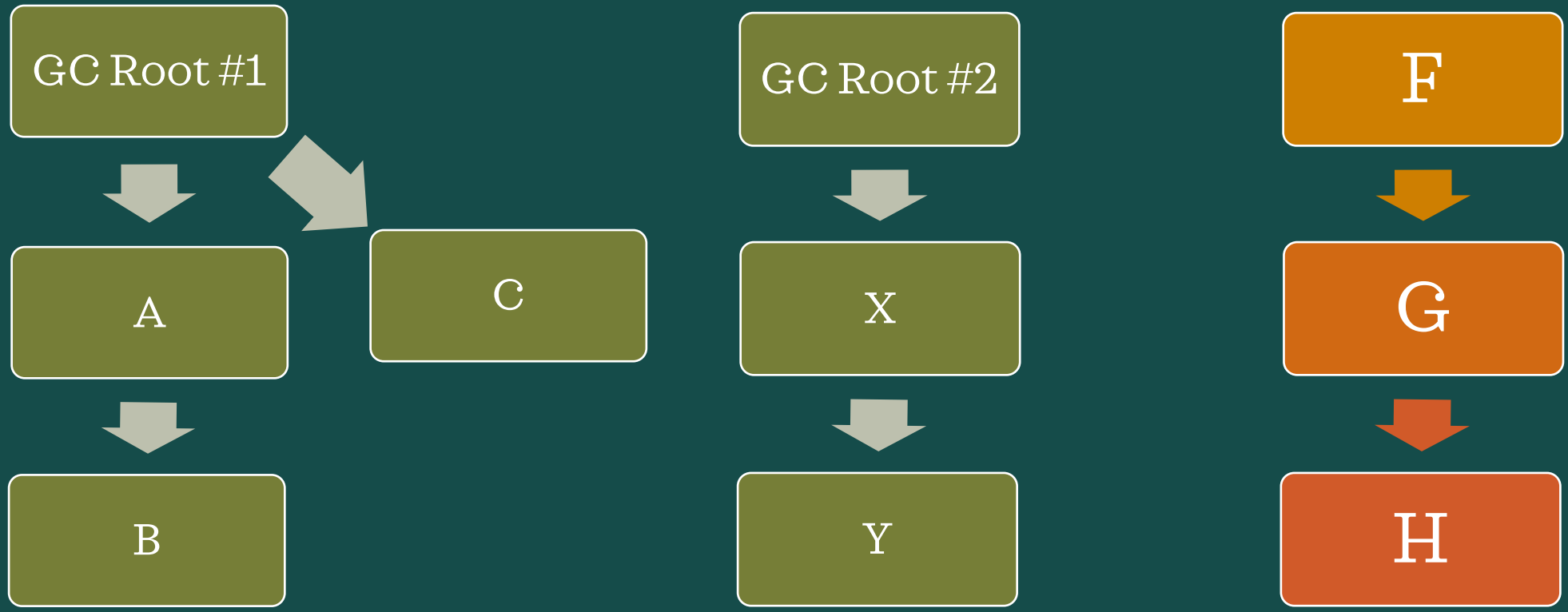
- Mark-sweep
- Others (rare!)

https://courses.cs.washington.edu/courses/csep521/07wi/prj/rick.pdf

.Net Garbage Collector

- Pause EE (Execution Engine)
- Mark
- Sweep
- Compact?

Mark Phase



Sweep phase

- All objects marked as unreachable are freed
- Objects with finalizers are moved to finalization queue

Compact


Allocated
Free
Allocated
Allocated
Allocated
Allocated
Free
Allocated

Compact


Allocated
Allocated
Allocated
Free
Allocated
Allocated
Allocated
Free

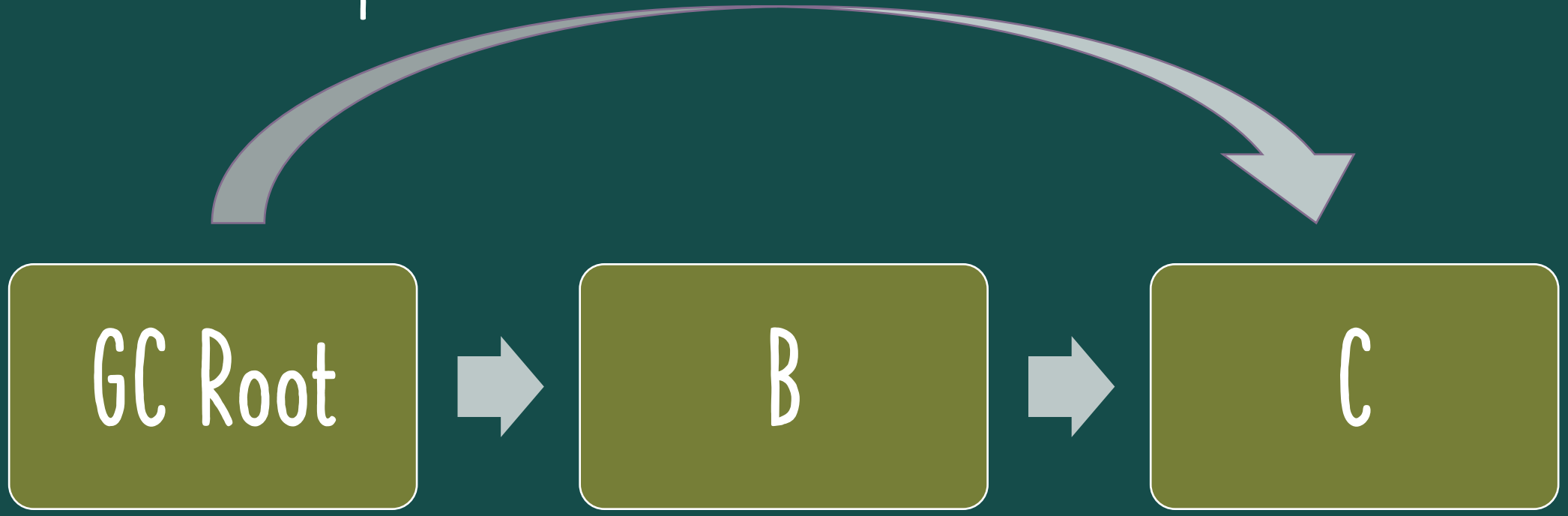
GC Roots

- Starting point for *Mark Phase* iteration
- GC Roots and any objects they reference are **not** released!

GC Roots

- Static references
- Stack references (roots for duration of method execution!)
- Handles (for p/invokes & unmanaged)
- F-reachable queue references (more on this later!)

Shortest retention path

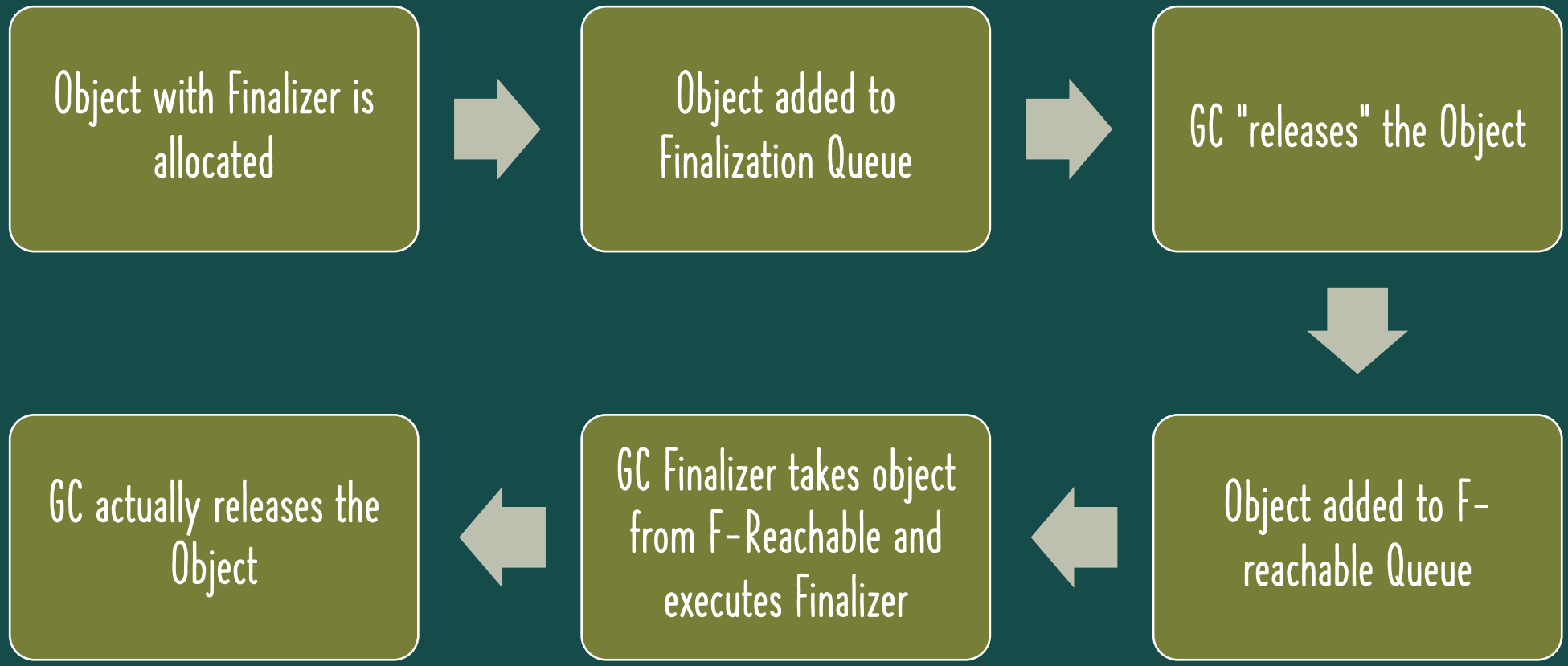


GC Generations

GC Generations

- Newly allocated objects start at Gen 0
- Objects surviving a GC are promoted to next Gen
- Bigger Gen = less frequent GC

Finalization & F-reachable queues



Memory Fragmentation

- When CLR GC frees memory, it leaves "holes"
- It may decide to compact memory and "close" the "holes"
- Big fragmentation = more frequent GC cycles
- LOH is not compacted automatically!
  - Except one specific scenario – read more in the Github link

https://github.com/dotnet/runtime/issues/8392
https://github.com/dotnet/docs/blob/master/docs/standard/garbage-collection/large-object-heap.md

Memory Fragmentation

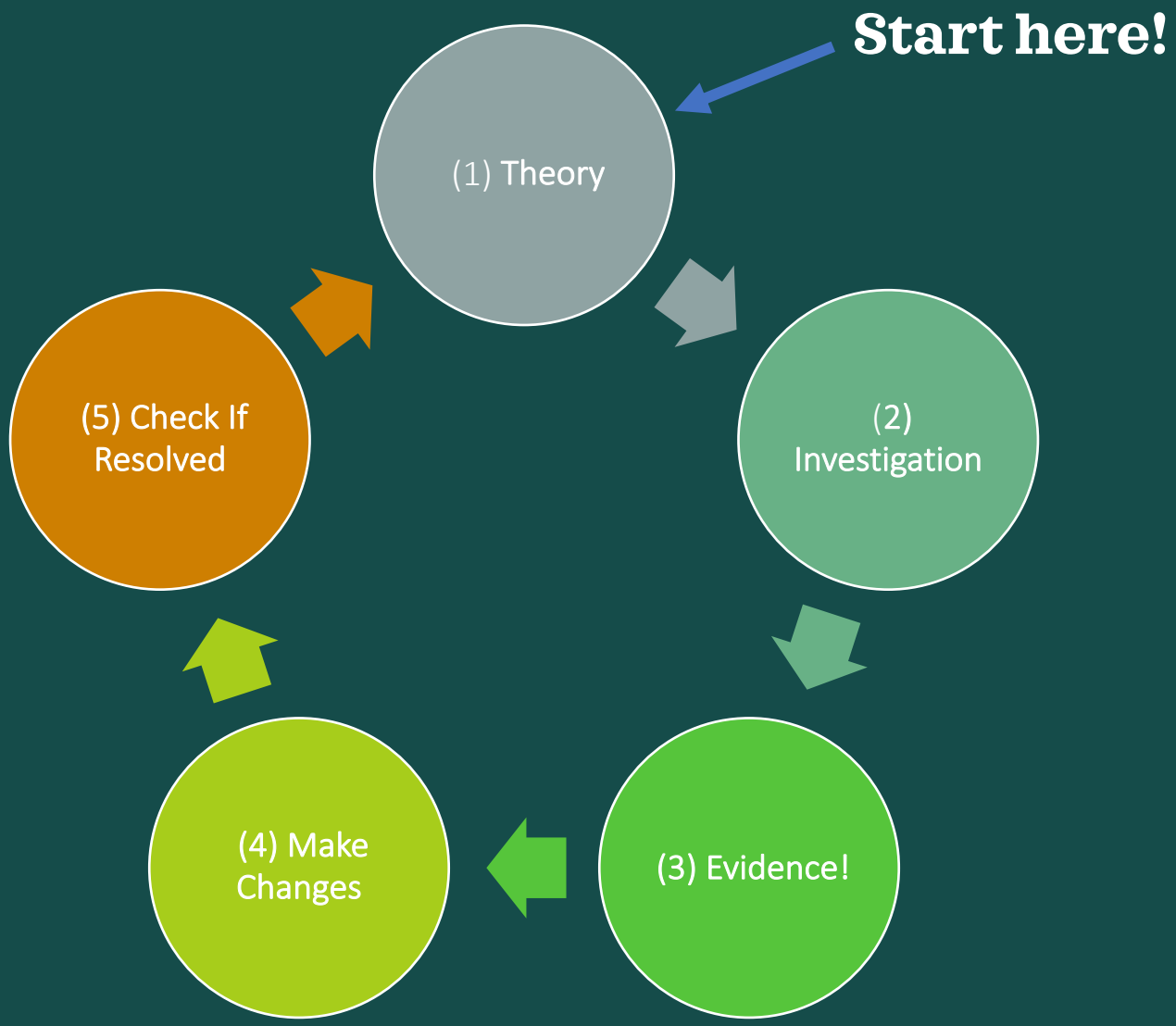
- High fragmentation = more GC cycles
- More GC cycles = More CPU

A word (or three!) about the investigation process

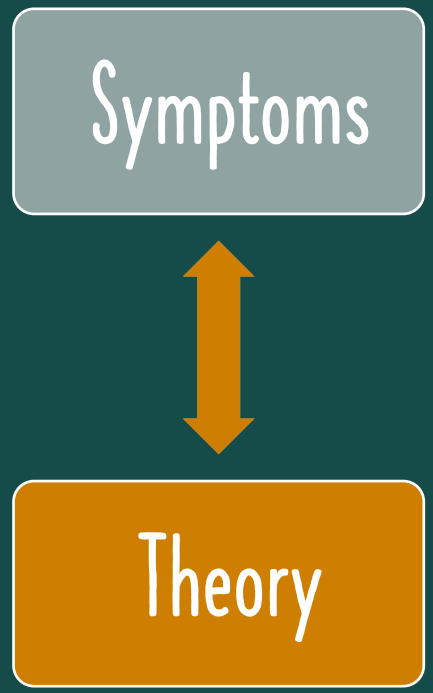
Part 2


WHAT IF I TOLD YOU
THERE IS NO SILVER BULLET?

Investigation process



1) Theory?



Note: resource leaks as a symptom (not only memory leaks!)

```
public static async Task<string> GetResponseFromGet(string url)
{
  using (var httpClient = new HttpClient())
  {
    var result = await httpClient.GetAsync(url);
    return await result.Content.ReadAsStringAsync();
  }
}
```

Dispose will "leak" connections

https://www.nimaara.com/beware-of-the-net-httpclient/

2) Investigation! (gather information)

- WinDBG
- Perfview
- Memory Profilers
- ClrMD
- Dashboards/monitoring

https://github.com/microsoft/clrmd
https://github.com/microsoft/perfview

3) Evidence (find a smoking gun...)

Examples:

- Unreasonable amount of object instances
- The amount of HUGE object instances
- Very high memory fragmentation
- Too much connections (resource leak?)
- GC Root s pointing to certain objects

Example of evidence

```
Size:        168(0xa8) bytes
File:
Fields:
              MT    Field   Offset                 Type VT     Attr            Value Name
00007ffc76ea8b68  40002d5        8 ...s.Logging.ILogger  0 instance 0000018589019f68 log
00007ffc7a058c18  40002d6       10 ...eansTaskScheduler  0 instance 00000185890197c8 masterScheduler
00007ffc7a4124f8  40002d7       78         System.Int32  1 instance                0 state
00007ffccef49dd8  40002d8       18        System.Object  0 instance 000001858c0365e8 lockable
00007ffc7a570528  40002d9       20 ....Task, mscorlib]]  0 instance 000001858c0365b8 workItems
00007ffccef330f8  40002da       50         System.Int64  1 instance 180578 totalItemsEnQueued
00007ffccef330f8  40002db       58         System.Int64  1 instance 180578 totalItemsProcessed
00007ffccef34258  40002dc       80      System.TimeSpan  1 instance 000001858c036590 totalQueuingDelay
00007ffccef3ccc8  40002dd       28 ...eading.Tasks.Task  0 instance 0000000000000000 currentTask
00007ffccef344e0  40002de       88      System.DateTime  1 instance 000001858c036598 currentTaskStarted
00007ffccef330f8  40002df       60         System.Int64  1 instance 0 shutdownSinceTimestamp
00007ffccef330f8  40002e0       68         System.Int64  1 instance 0 lastShutdownWarningTimestamp
00007ffc7a5707c8  40002e1       30 ...TrackingStatistic  0 instance 0000000000000000 queueTracking
00007ffccef330f8  40002e2       70         System.Int64  1 instance 0 quantumExpirations
00007ffccef4c148  40002e3       7c         System.Int32  1 instance                0 workItemGroupStatisticsNumber
00007ffccef3e300  40002e4       90 ...CancellationToken  1 instance 000001858c0365a0 cancellationToken
00007ffc7a0575c8  40002e5       38 ...erStatisticsGroup  0 instance 00000185890194f0 schedulerStatistics
00007ffc7a4128d8  40002e6       40 ...tionTaskScheduler  0 instance 000001858c036600 <TaskScheduler>k__BackingField
00007ffccef344e0  40002e7       98      System.DateTime  1 instance 000001858c0365a8 <TimeQueued>k__BackingField
00007ffc798766e0  40002e9       48 ...ime.IGrainContext  0 instance 000001858c0362a8 <GrainContext>k__BackingField
00007ffccef38238  40002d4      238 ...ding.WaitCallback  0   static 0000018588621020 ExecuteWorkItemCallback
0:000> !DumpObj /d 00000185886ec3e8
```

Example of evidence

Statistics:

State	# of Tasks
TASK STATE DELEGATE INVOKED	7
TASK STATE FAULTED	32
TASK STATE CANCELED	2
TASK STATE RAN TO COMPLETION	24059
TASK STATE WAITINGFORACTIVATION	123844

State # of Tasks

4 & 5) Make changes & check if resolved

- Make relevant changes (less allocations, remove references etc.)
- It is possible that the original theory wasn't correct!
- Rinse and repeat as needed

F-Reachable queue

- Elements in *f-reachable*
  - roots during **mark** phase
  - "live" in Gen2 while in *f-reachable*
  - *Cease* to become roots after GC

T he practical side of things...

Part 3

Memory dumps, when?

- High resource usage (CPU/RAM)
- Process hangs/crashes
- Inefficiencies/slowdowns

Note: memory dumps are a last resort!

Not only WinDbg!

Any memory profiler

PerfView

VMMap, Process Monitor, ProcDump (SysInternals)

Debug Diagnostic Tool

Visual Studio (parallel stacks!)

.Net CLI tools

Linux - where to start?

Valgrind

LLDB

GDB

.Net CLI tools

vmstat, htop,
other bash tools

https://github.com/microsoft/ProcDump-for-Linux

Investigating in Linux

SOS plugin exists for LLDB debugger

Working in LLDB is similar to WinDbg

Install SOS plugin in Linux using **dotnet-sos** CLI tool

https://github.com/dotnet/diagnostics/blob/master/documentation/sos-debugging-extension.md

.Net CLI Tools

Performance monitoring	https://docs.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-counters
Downloads PDBs and DAC files	https://docs.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-symbol
Collect and analyze Windows and Linux dumps.	https://docs.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-dump

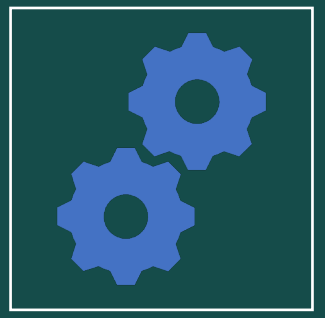
dotnet tool install --global [tool name]

WinD bg

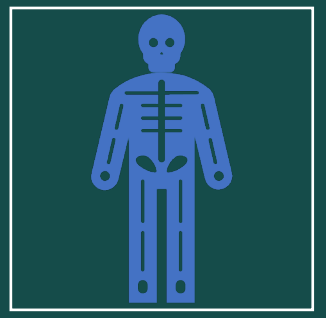
How and what?

WinDbg?

Windows-only debugger



Can debug kernel and user-mode



Not only post-mortem! Can do live debugging

SOS debugging extension (Son of Strike)

- WinDBG extension that allows debugging .Net processes
- SOS depends on Windows version!
  - Especially relevant for dumps from another machine
  - New WinDbg will fetch SOS automatically
  - Use **dotnet-sos** for Linux or old WinDbg

Load SOS with those commands →

```
.loadby sos clr
.loadby sos coreclr
```

https://docs.microsoft.com/en-us/dotnet/framework/tools/sos-dll-sos-debugging-extension

Common commands

- !dumpheap [-stat] [-mt <>] [-type <>] [-strings] [-min] [-max]
- !dumpgen <genNum> [-free] [-stat] [-type <>] [-nostrings] (SOSEX)
- !gcroot <objectAddr> [-nostacks]
- !refs <objectAddr> [-target|-source] (SOSEX)
- !finalizequeue
- !finq [genNum] [-stat] (SOSEX)
- !frq [-stat]
- !objsize <addr>
- !do <address>
- !mdt [typename | MT] [addr] [-r[:level]] [-e[:level]] (SOSEX)

Useful extensions

- SOSEX - http://www.stevestechspot.com/downloads/sosex_64.zip
- **GSOSE** (**G**rand **S**on **O**f **S**trike) - https://github.com/chrisnas/DebuggingExtensions
- MEX Debugging Extension - https://github.com/REhints/WinDbg/tree/master/MEX

.load [path to extension dll]

Note: GSOSE and MEX don't work with .Net Core

Useful reading materials

- .Net Book of Runtime
- https://blog.reverberate.org/2013/05/deep-wizardry-stack-unwinding.html
- https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/
- https://github.com/dotnet/docs/blob/master/docs/standard/garbage-collection/index.md
- https://mattwarren.org/2017/07/10/Memory-Usage-Inside-the-CLR/

Hands-on practice

Part 4

Hands-on flow

1. Take a look at the process, it's usage of system resources. Do tracing.
2. Use Process Explorer to take "Full Dump" and investigate
3. Once theory is established, take a look at code
   - If needed, take a look at dump again
4. Conclusions, perhaps a fix

Summary

- Dumps are most useful to investigate issues in production
- WinDbg can and should be used together with other tools
- Memory leaks in .Net can have multiple possible symptoms
- Dump investigation: common sense + technical knowledge + code behavior
- Dump investigation is unpredictable
- More often than not, managed memory leak = reference leak

Questions?

- michael.yarichuk@gmail.com
- @myarichuk
- https://github.com/myarichuk
- http://www.graymatterdeveloper.com/