Analyzing your game performance using Event Tracing for Windows

Event Tracing for Windows (ETW) is a kernel-level tracing mechanism that logs various system events to a log file. This log can then be viewed to debug your application or determine where performance issues are happening. These events are generated by system components called providers, which make it possible to capture very fine, detailed pieces of data that can be used to analyze program performance characteristics.

Event Tracing for Windows is supported on all Windows-based platforms, and can be used to profile the Unity Editor, Standalone players and Windows Store players running on all PCs and devices.

Profiling using Event Tracing for Windows is a two-step process:

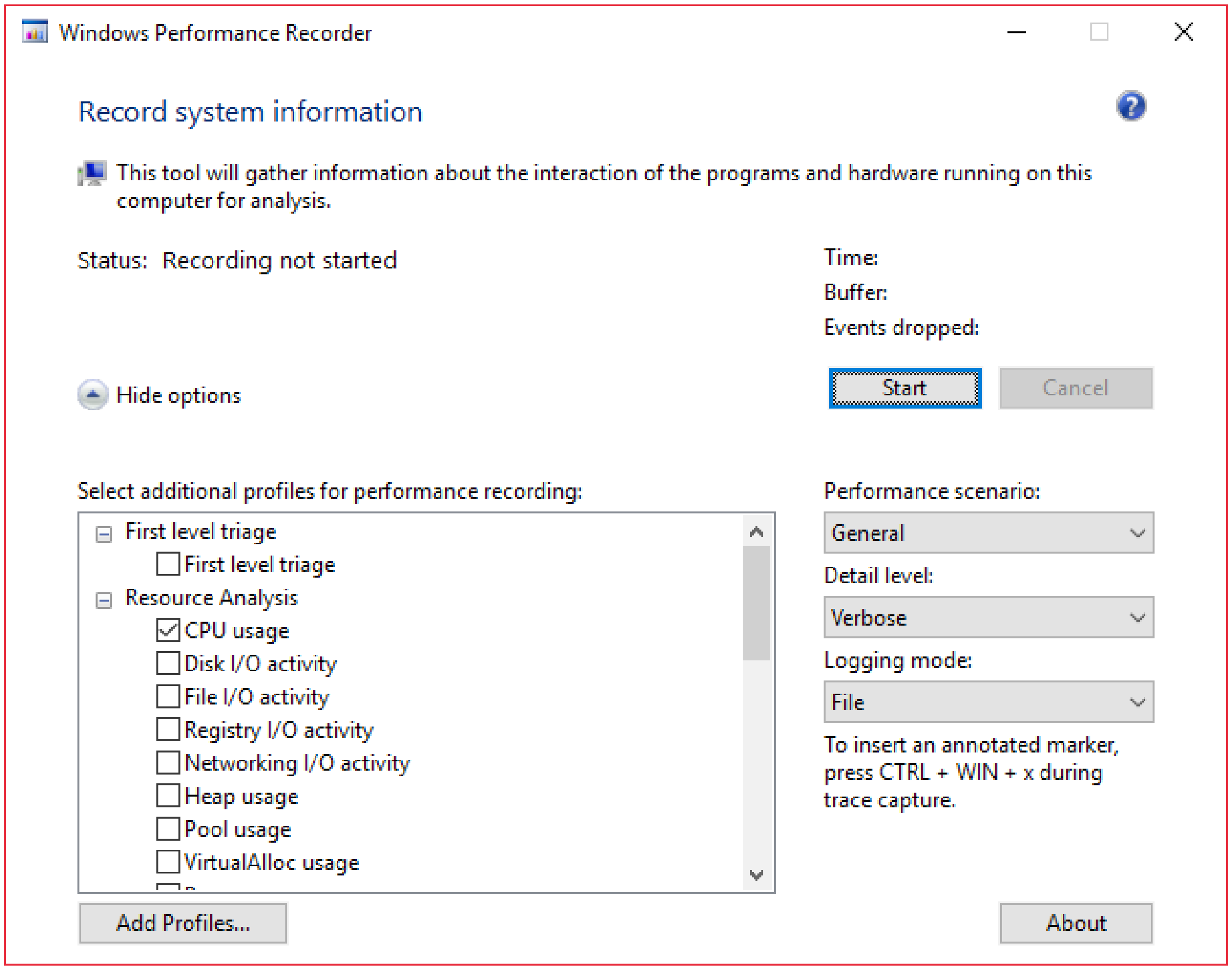
1. Run an application and record the trace log (this is carried out on the target machine)
2. Analyze the trace log (this is carried out on the developer's machine)

Running Event Tracing for Windows on a PC allows both event log capture and analysis on the same machine. The recorded event trace file can be recognized by the .etl file extension.

NOTE: It is important not to profile an application running in the Unity Editor unless it has been determined that the problem is caused by doing so, as performance characteristics will be slightly different to the final built game and may give inaccurate results.

Recording a trace on PC

The most straightforward way to capture traces on a PC is to use Windows Performance Recorder (wprui.exe).


Windows Performance Recorder

Record system information

This tool will gather information about the interaction of the programs and hardware running on this computer for analysis.

Status: Recording not started

Time:
Buffer:
Events dropped:

Select additional profiles for performance recording:

- First level triage
  - ☐ First level triage
- Resource Analysis
  - ☑ CPU usage
  - ☐ Disk I/O activity
  - ☐ File I/O activity
  - ☐ Registry I/O activity
  - ☐ Networking I/O activity
  - ☐ Heap usage
  - ☐ Pool usage
  - ☐ VirtualAlloc usage

Performance scenario: General

Detail level: Verbose

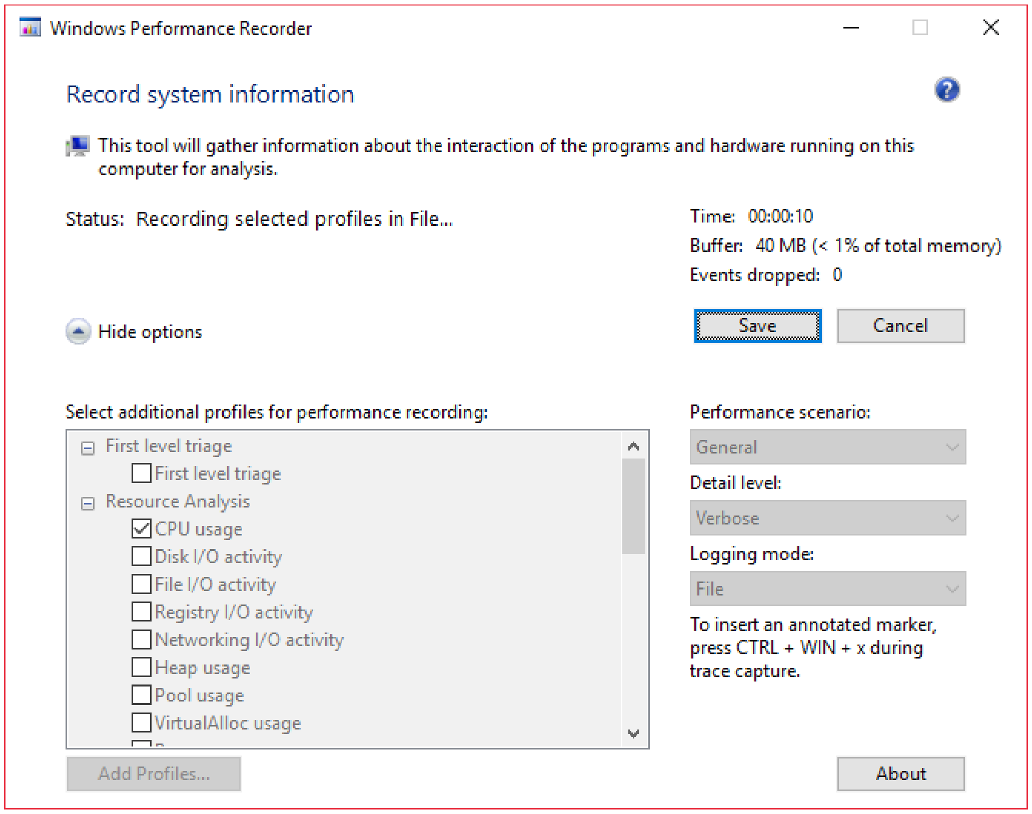
Logging mode: File

To insert an annotated marker, press CTRL + WIN + x during trace capture.

Open Windows Performance Recorder and take a look at the list of available Event Provider profiles that you can enable for the capture. You can also add custom profiles, which allow you to only enable capture for the things you are interested in (see #Custom ETW Capture Profiles).

NOTE: Profiles are resource-intensive. It is important to minimize the number of enabled profiles where possible, as this reduces the overhead for the capture, makes the captured trace log smaller, and reduces the chance that events are dropped. If you have a lot of profiles enabled, you risk a massive capture trace log - potentially several gigabytes per minute of capture.

Select what you want to profile and click the **Start** button to initiate the capture.


Windows Performance Recorder

Record system information

This tool will gather information about the interaction of the programs and hardware running on this computer for analysis.

Status: Recording selected profiles in File...

Time: 00:00:10
Buffer: 40 MB (< 1% of total memory)
Events dropped: 0

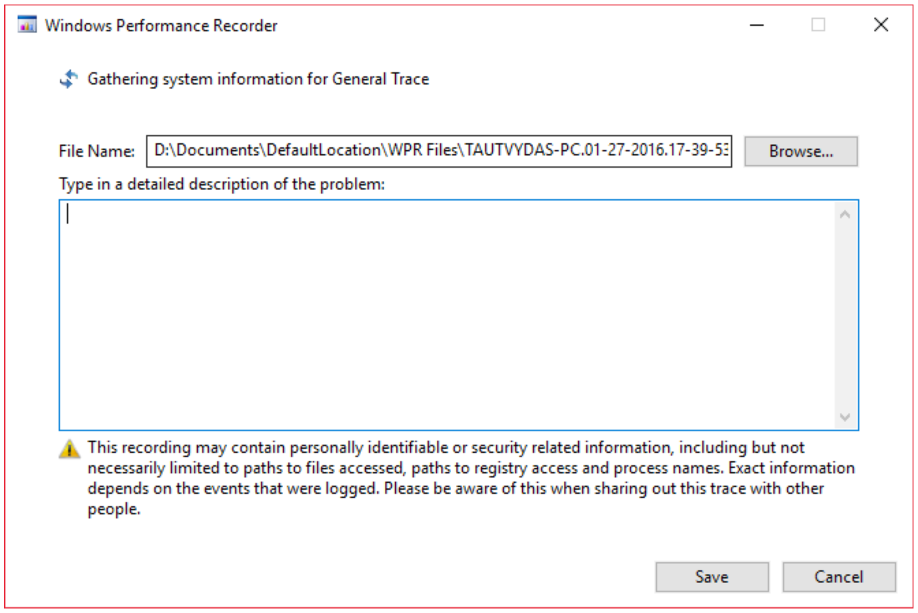
Select additional profiles for performance recording:

- First level triage
  - [ ] First level triage
- Resource Analysis
  - [x] CPU usage
  - [ ] Disk I/O activity
  - [ ] File I/O activity
  - [ ] Registry I/O activity
  - [ ] Networking I/O activity
  - [ ] Heap usage
  - [ ] Pool usage
  - [ ] VirtualAlloc usage

Performance scenario: General
Detail level: Verbose
Logging mode: File

To insert an annotated marker, press CTRL + WIN + x during trace capture.

When the capture is finished, click the Save button and, if required, fill in the detailed description box that appears. This box is optional; leave it blank if you don’t need to record a problem for future reference. Click Save again to exit this window.


Windows Performance Recorder

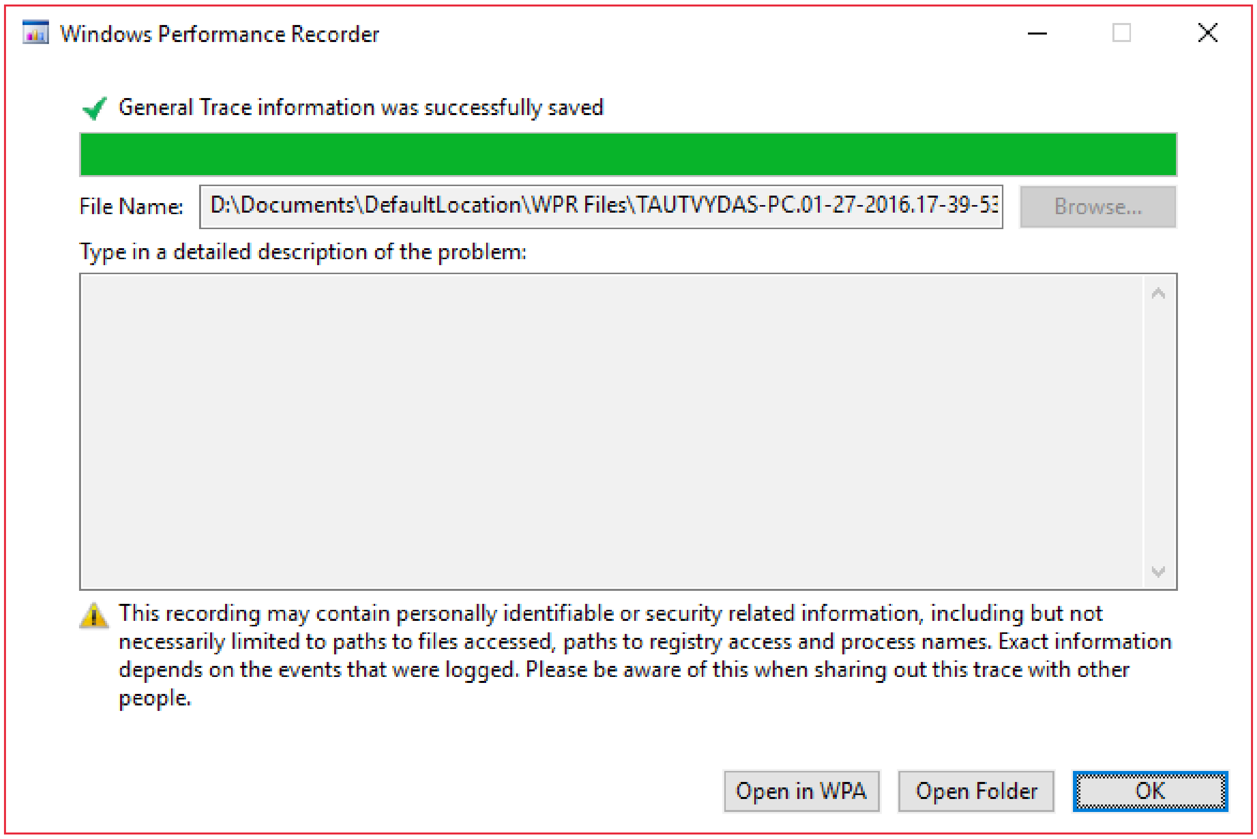
Gathering system information for General Trace

File Name: D:\Documents\DefaultLocation\WPR Files\TAUTVYDAS-PC.01-27-2016.17-39-53

Type in a detailed description of the problem:

This recording may contain personally identifiable or security related information, including but not necessarily limited to paths to files accessed, paths to registry access and process names. Exact information depends on the events that were logged. Please be aware of this when sharing out this trace with other people.

Windows Performance Recorder can take a while to save the trace log, especially if you're running it for the first time on the target machine. When the save is complete, you are asked whether you want to open it. Click "Open in WPA" to analyze the trace log in Windows Performance Analyzer.



Windows Performance Recorder

General Trace information was successfully saved

File Name: D:\Documents\DefaultLocation\WPR Files\TAUTVYDAS-PC.01-27-2016.17-39-53

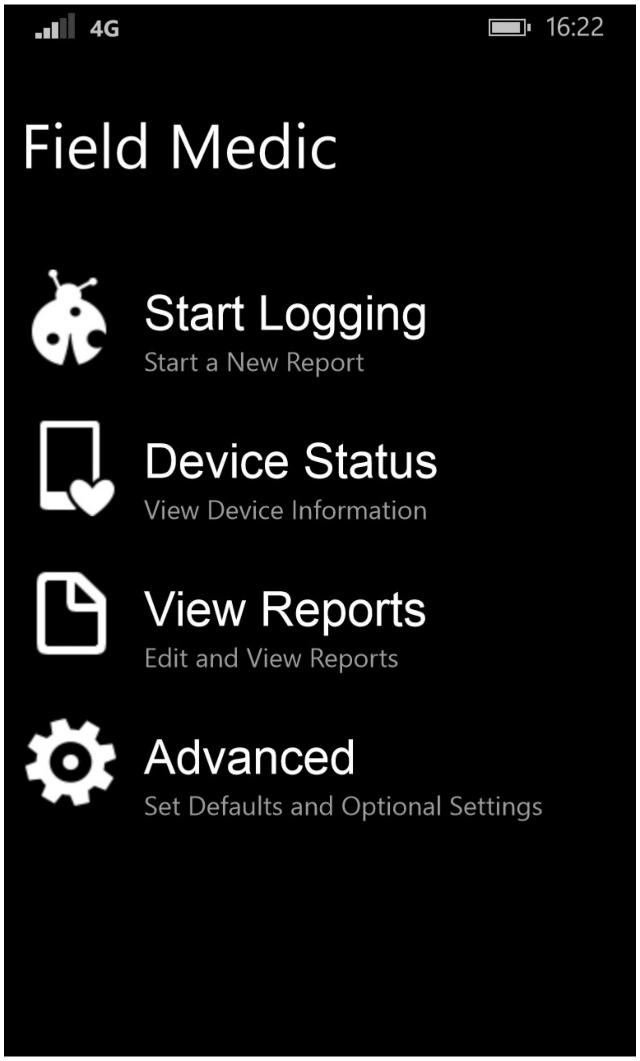
Type in a detailed description of the problem:

This recording may contain personally identifiable or security related information, including but not necessarily limited to paths to files accessed, paths to registry access and process names. Exact information depends on the events that were logged. Please be aware of this when sharing out this trace with other people.

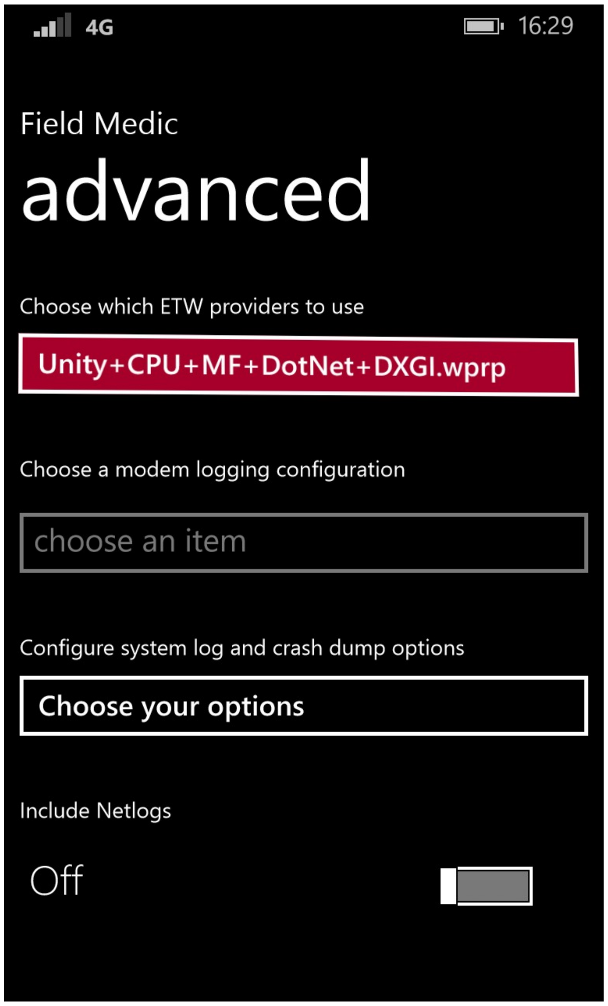
Recording a trace on a Windows phone

The most straightforward way to capture traces on a Windows phone is to use the (Field Medic) [https://www.microsoft.com/en-us/store/apps/field-medic/9wzdncrfjb82] app.

Install the app and launch it from the target device.



From the main menu, tap **Advanced** to select the ETW Event Providers to use for the capture.


4G 16:29

Field Medic

advanced

Choose which ETW providers to use

Unity+CPU+MF+DotNet+DXGI.wprp

Choose a modem logging configuration

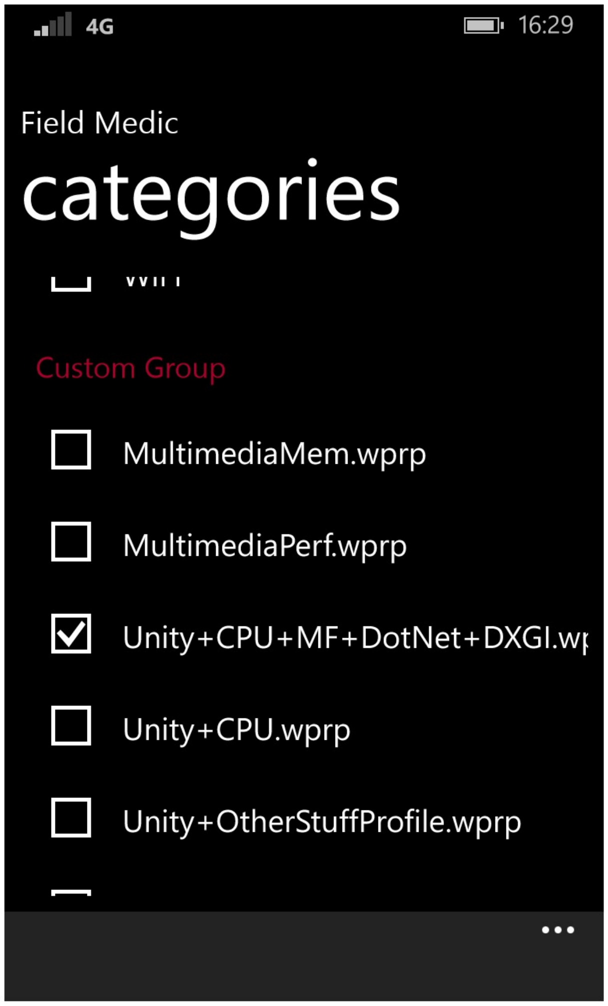
choose an item

Configure system log and crash dump options

Choose your options

Include Netlogs

Off


4G 16:29

Field Medic

categories

Custom Group

☐ MultimediaMem.wprp

☐ MultimediaPerf.wprp

☑ Unity+CPU+MF+DotNet+DXGI.wp

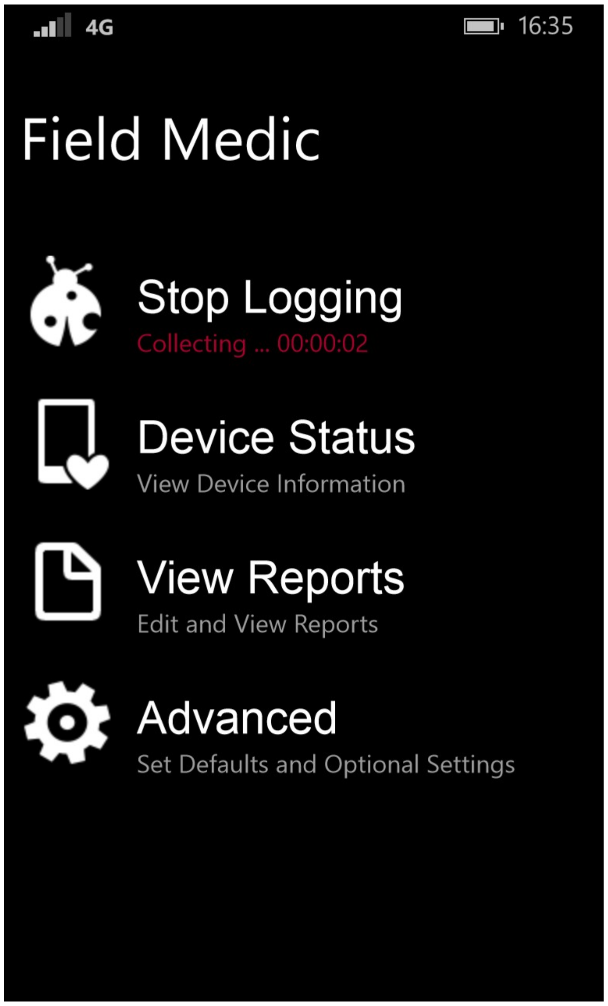
☐ Unity+CPU.wprp

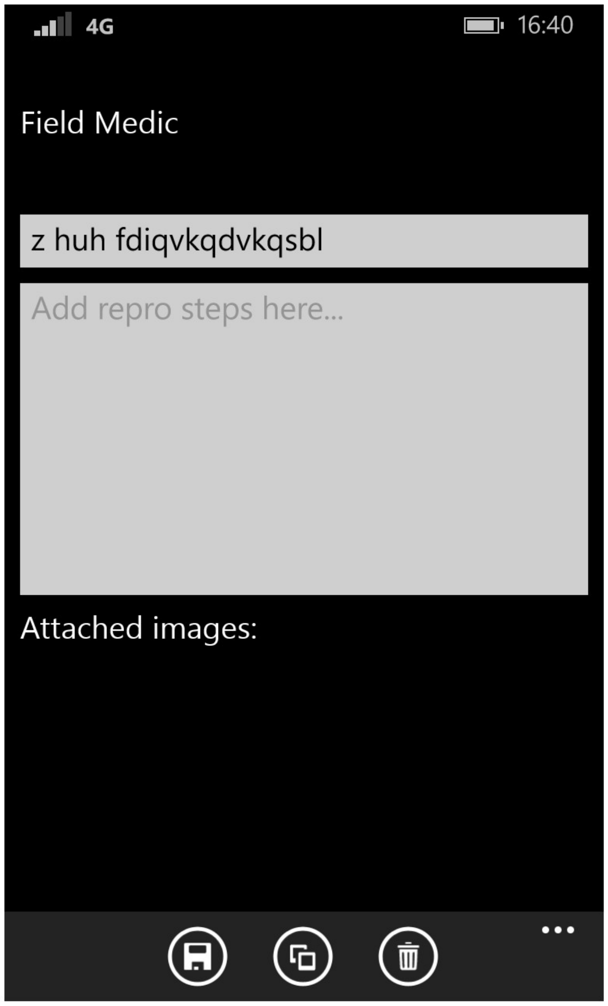
☐ Unity+OtherStuffProfile.wprp

NOTE: It is recommended that you only select one provider when using Field Medic to capture traces, as it produces one .etl file per profile, rather than one per recording.

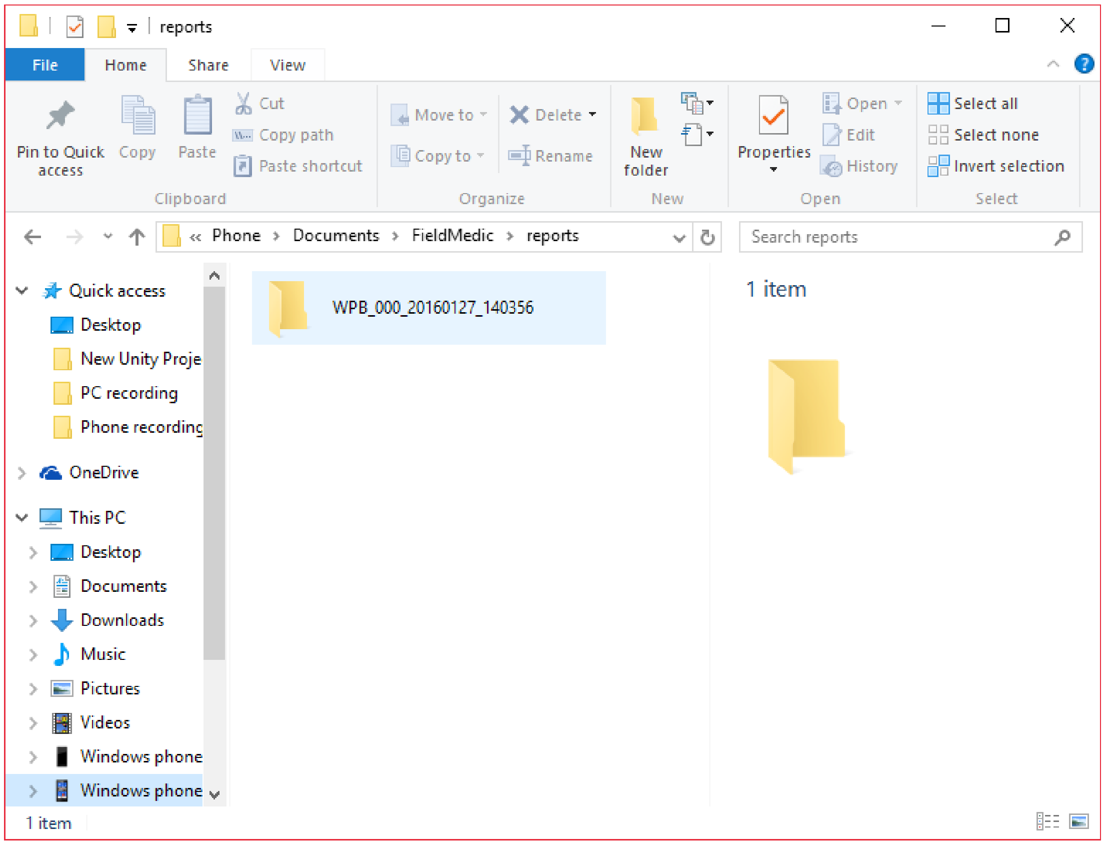
Once you have selected the Event Provider(s) to record, go back to the main menu and click **Start Logging**. When it starts collecting data, perform the actions you want to record.

When you've finished recording, go back to the Field Medic app and click Stop Logging. Enter a name for the captured trace and click Save to save the trace log to the device's storage.

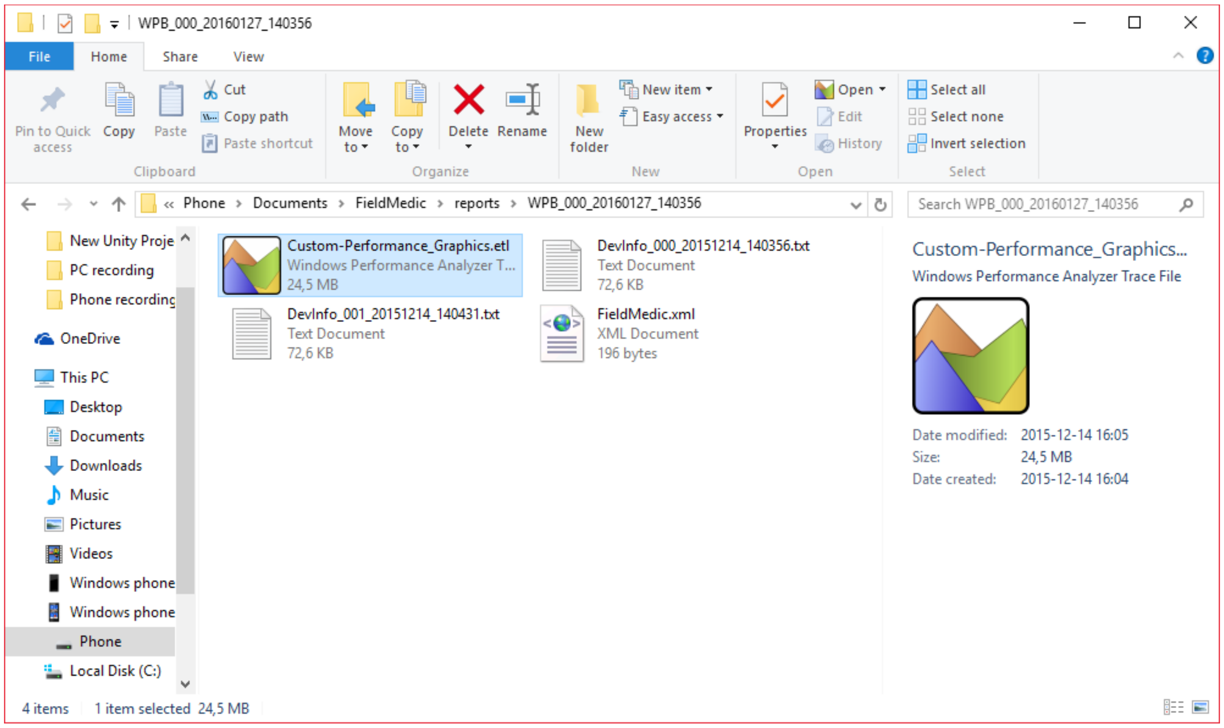




To retrieve the trace log from the device, connect the device to a PC, open its storage in Explorer and navigate to the **Documents\Field Medic\Reports** directory.



Inside that folder, locate the .etl file.



Custom-Performance_Graphics.etl
Windows Performance Analyzer T...
24,5 MB

DevInfo_000_20151214_140356.txt
Text Document
72,6 KB

DevInfo_001_20151214_140431.txt
Text Document
72,6 KB

FieldMedic.xml
XML Document
196 bytes

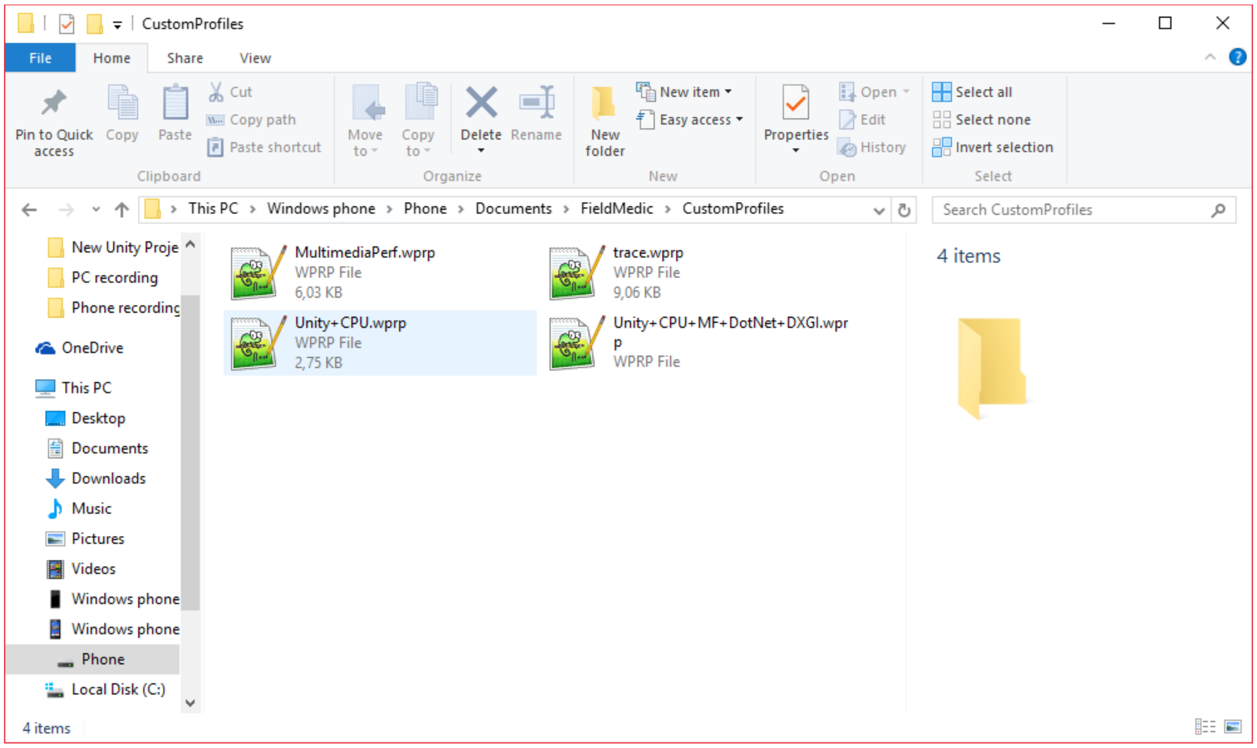
Custom-Performance_Graphics...
Windows Performance Analyzer Trace File

Date modified: 2015-12-14 16:05
Size: 24,5 MB
Date created: 2015-12-14 16:04

4 items 1 item selected 24,5 MB

Custom Event Provider profiles

To use custom profiles in Field Medic, copy the profile file to **Documents/FieldMedic/CustomProfiles** on the target Windows mobile device.



For more information on custom Event Providers, see #Custom ETW Capture Profiles.

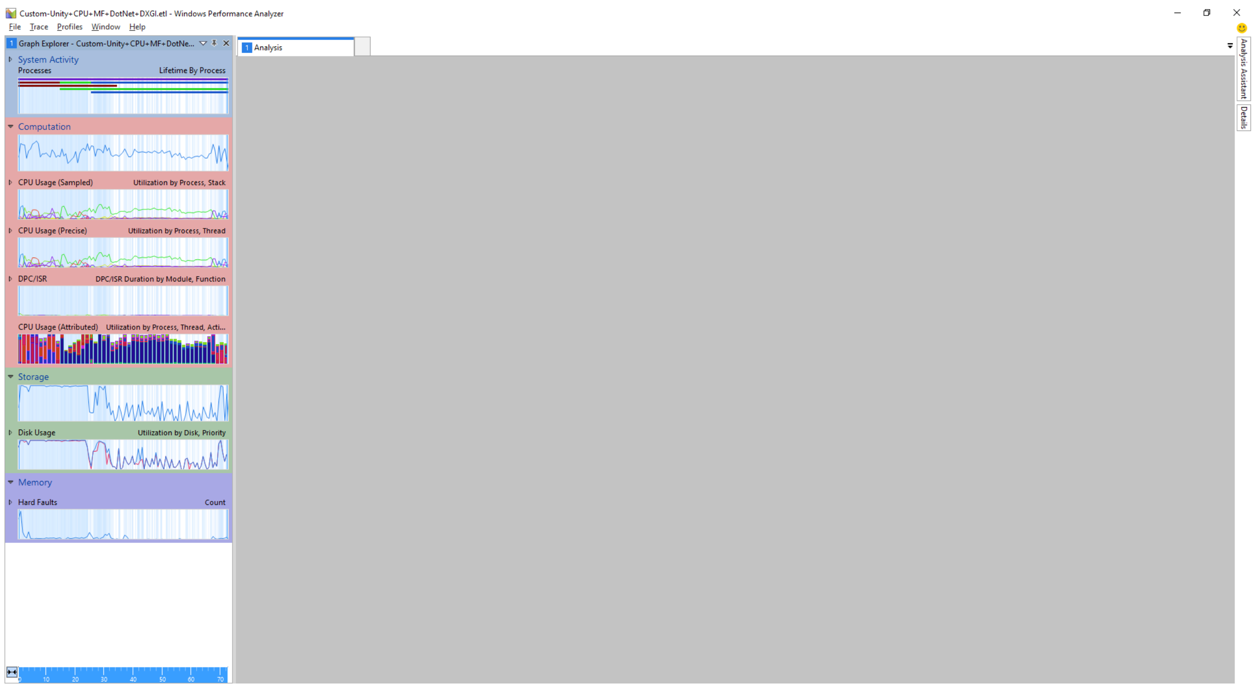
Custom ETW capture profiles

This page lists several available ETW capture profiles, which enable different Event Providers. These capture profiles are included in the zip file along with this PDF.

- Unity+CPU+MF+DXGI.wprp - Enables CPU Usage providers (sampling and precise), DXGI providers and several other MediaFoundation and Direct3D providers. Use this ETW capture profile to record CPU and frame rate providers.
- Unity+CPU+MF+DotNet+DXGI.wprp - Same as Unity+CPU+MF+DXGI.wprp, except it also captures stack traces for managed code when using .NET scripting backend.
- VirtualAlloc.wprp - Enables various memory usage providers, including VirtualAlloc Commit providers. Use this to record memory usage.

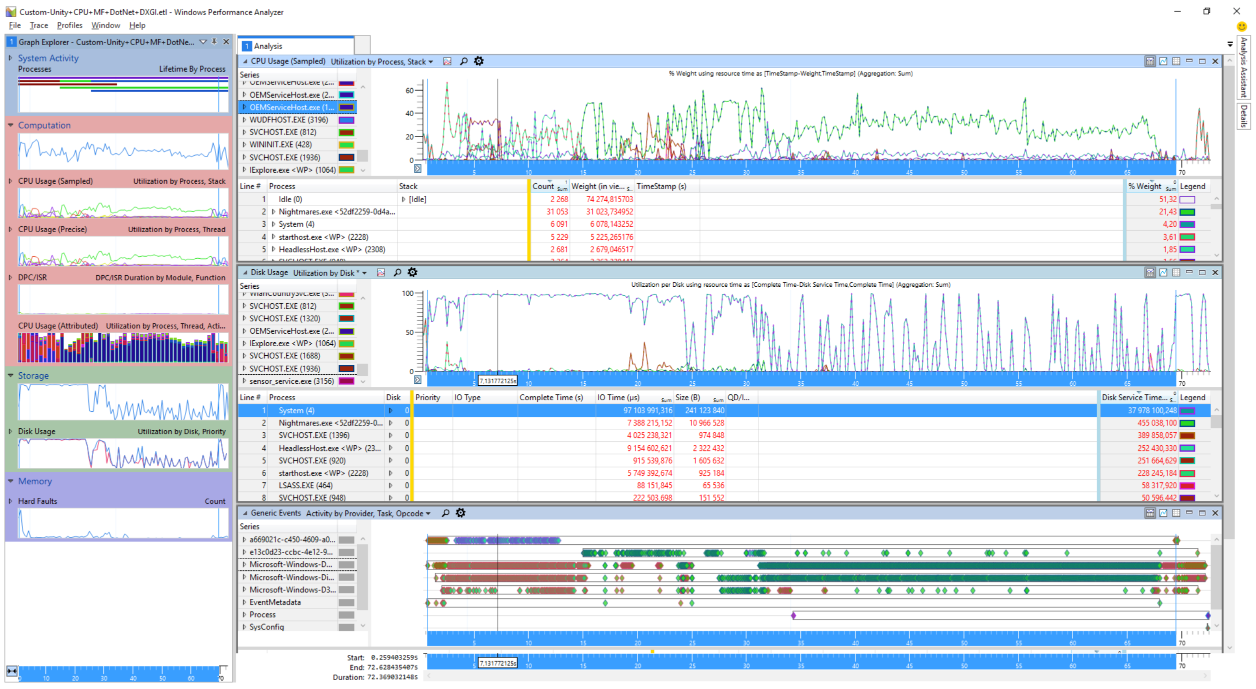
Analysing the captured trace using Windows Performance Analyzer

Windows Performance Analyzer is part of the Windows Performance toolkit, which can be installed with the [Windows SDK](https://dev.windows.com/en-us/downloads/windows-10-sdk). Open the captured trace (the .etl file) with Windows Performance Analyzer.


Custom-Unity+CPU+MF+DotNet+DXGI.etl - Windows Performance Analyzer
File Trace Profiles Window Help
Graph Explorer - Custom-Unity+CPU+MF+DotNe...
System Activity
Processes
Lifetime By Process
Computation
CPU Usage (Sampled)
Utilization by Process, Stack
CPU Usage (Precise)
Utilization by Process, Thread
DPC/ISR
DPC/ISR Duration by Module, Function
CPU Usage (Attributed)
Utilization by Process, Thread, Acti...
Storage
Disk Usage
Utilization by Disk, Priority
Memory
Hard Faults
Count
Analysis
Analysis Assistant
Details

Caption: Windows Performance Analyzer main window

The graphs on the left-hand side give you different performance metrics. The number of available graphs depends on the number of recorded Event Providers. Double-click a graph for a more detailed view in the Analysis tab. Open multiple graphs to display corresponding information across the same time range:


Custom-Unity+CPU+MF+DotNet+DXGI.etl - Windows Performance Analyzer
File Trace Profiles Window Help
Graph Explorer - Custom-Unity+CPU+MF+DotNe...
System Activity
Processes
Lifetime By Process
Computation
CPU Usage (Sampled)
Utilization by Process, Stack
CPU Usage (Precise)
Utilization by Process, Thread
DPC/ISR
DPC/ISR Duration by Module, Function
CPU Usage (Attributed)
Utilization by Process, Thread, Acti...
Storage
Disk Usage
Utilization by Disk, Priority
Memory
Hard Faults
Count
Analysis
CPU Usage (Sampled) Utilization by Process, Stack
Series
OEMServiceHost.exe (2...
OEMServiceHost.exe (1...
WUDFHOST.EXE (3196)
SVCHOST.EXE (812)
WININIT.EXE (428)
SVCHOST.EXE (1936)
IExplore.exe <WP> (1064)
% Weight using resource time as [TimeStamp-Weight,TimeStamp] (Aggregation: Sum)
Disk Usage Utilization by Disk *
Series
SVCHOST.EXE (812)
SVCHOST.EXE (1320)
OEMServiceHost.exe (2...
IExplore.exe <WP> (1064)
SVCHOST.EXE (1688)
SVCHOST.EXE (1936)
sensor_service.exe (3156)
Utilization per Disk using resource time as [Complete Time-Disk Service Time,Complete Time] (Aggregation: Sum)
7,131772125s
Generic Events Activity by Provider, Task, Opcode
Series
a669021c-c450-4609-a0...
e13c0d23-ccbc-4e12-9...
Microsoft-Windows-D...
Microsoft-Windows-Di...
Microsoft-Windows-D3...
EventMetadata
Process
SysConfig
Analysis Assistant
Details

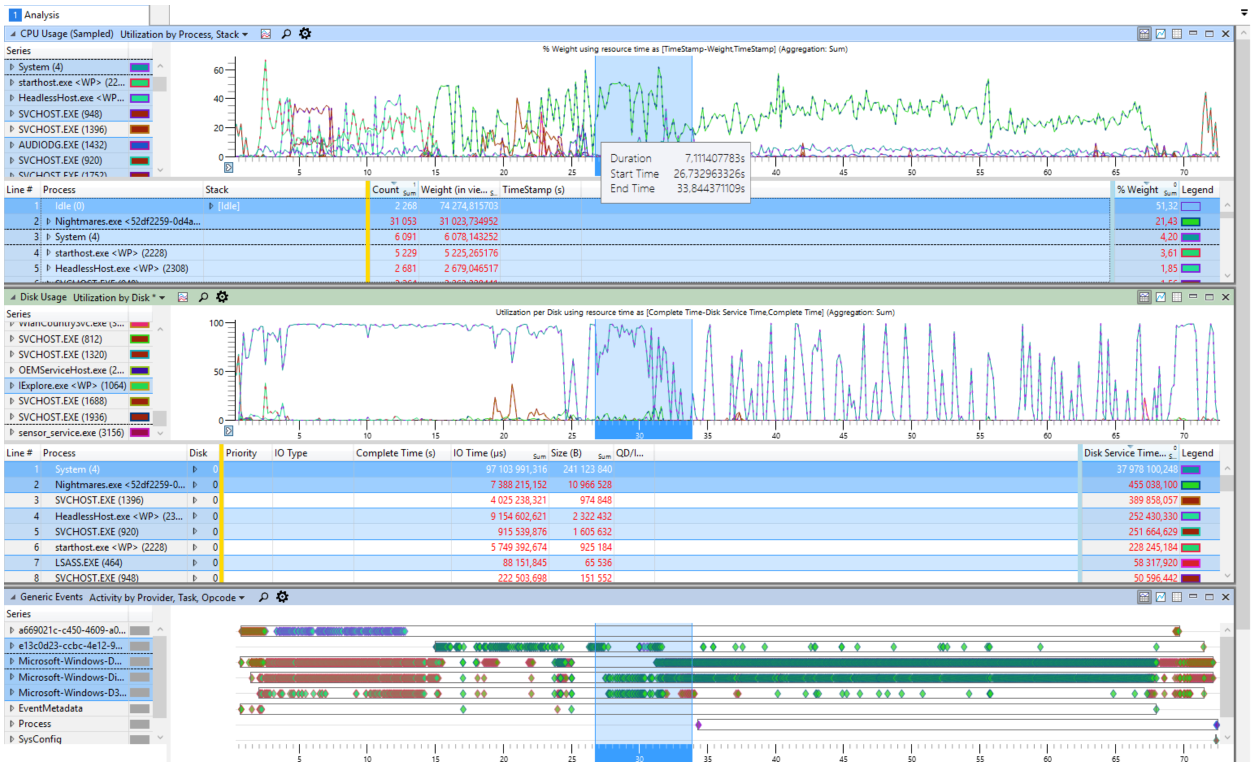
Line #	Process	Stack	Count	Weight (in vie...	TimeStamp (s)	% Weight	Legend
1	Idle (0)	[Idle]	2 268	74 274,815703		51,32	⬜
2	Nightmares.exe <52df2259-0d4a...		31 053	31 023,734952		21,43	🟩
3	System (4)		6 091	6 078,143252		4,20	🟦
4	starthost.exe <WP> (2228)		5 229	5 225,265176		3,61	🟩
5	HeadlessHost.exe <WP> (2308)		2 681	2 679,046517		1,85	🟩

Line #	Process	Disk	Priority	IO Type	Complete Time (s)	IO Time (µs)	Size (B)	QD/I...	Disk Service Time...	Legend
1	System (4)	0				97 103 991,316	241 123 840		37 978 100,248	🟦
2	Nightmares.exe <52df2259-0...	0				7 388 215,152	10 966 528		455 038,100	🟩
3	SVCHOST.EXE (1396)	0				4 025 238,321	974 848		389 858,057	🟫
4	HeadlessHost.exe <WP> (23...	0				9 154 602,621	2 322 432		252 430,330	🟩
5	SVCHOST.EXE (920)	0				915 539,876	1 605 632		251 664,629	🟥
6	starthost.exe <WP> (2228)	0				5 749 392,674	925 184		228 245,184	🟩
7	LSASS.EXE (464)	0				88 151,845	65 536		58 317,920	🟥
8	SVCHOST.EXE (948)	0				222 503,698	151 552		50 596,442	🟥

```
Start:    0.259403259s
End:      72.628435407s
Duration: 72.369032148s
```

Using the timeline

Select a time range on one graph to select it on all graphs. This also highlights the events in the selected part of the timeline:

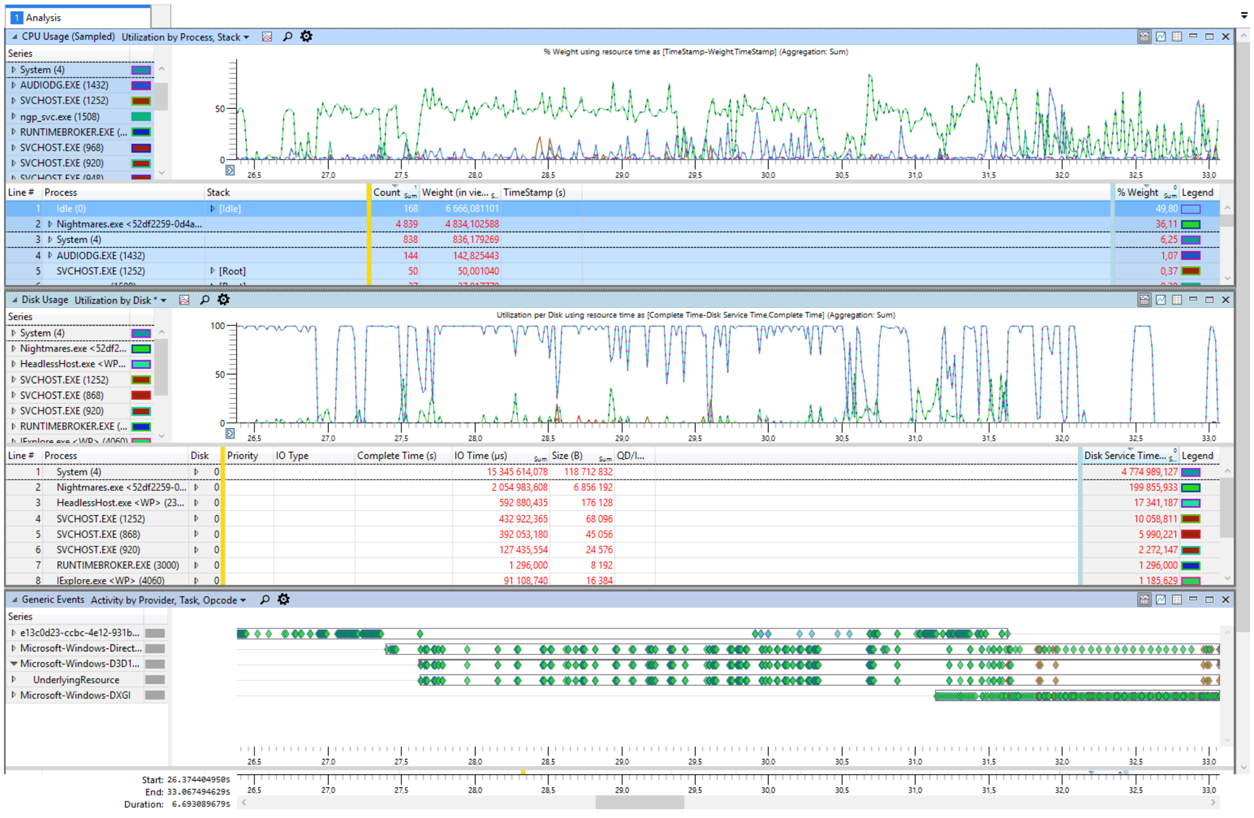


Duration	7,111407783s
Start Time	26,732963326s
End Time	33,844371109s

Line #	Process	Stack	Count Sum	Weight (in vie...	TimeStamp (s)	% Weight Sum	Legend
1	Idle (0)	[Idle]	2 268	74 274,815703		51,32	▭
2	Nightmares.exe <52df2259-0d4a...		31 053	31 023,734952		21,43	🟩
3	System (4)		6 091	6 078,143252		4,20	🟦
4	starthost.exe <WP> (2228)		5 229	5 225,265176		3,61	🟩
5	HeadlessHost.exe <WP> (2308)		2 681	2 679,046517		1,85	🟩

Line #	Process	Disk	Priority	IO Type	Complete Time (s)	IO Time (µs) Sum	Size (B) Sum	QD/I...	Disk Service Time...	Legend
1	System (4)	0				97 103 991,316	241 123 840		37 978 100,248	🟦
2	Nightmares.exe <52df2259-0...	0				7 388 215,152	10 966 528		455 038,100	🟩
3	SVCHOST.EXE (1396)	0				4 025 238,321	974 848		389 858,057	🟥
4	HeadlessHost.exe <WP> (23...	0				9 154 602,621	2 322 432		252 430,330	🟩
5	SVCHOST.EXE (920)	0				915 539,876	1 605 632		251 664,629	🟥
6	starthost.exe <WP> (2228)	0				5 749 392,674	925 184		228 245,184	🟩
7	LSASS.EXE (464)	0				88 151,845	65 536		58 317,920	🟥
8	SVCHOST.EXE (948)	0				222 503,698	151 552		50 596,442	🟥

To filter the events to the selected time range, right-click on the time range and choose Zoom:


1 Analysis
CPU Usage (Sampled) Utilization by Process, Stack
Series
System (4)
AUDIODG.EXE (1432)
SVCHOST.EXE (1252)
ngp_svc.exe (1508)
RUNTIMEBROKER.EXE (...
SVCHOST.EXE (968)
SVCHOST.EXE (920)
% Weight using resource time as [TimeStamp-Weight,TimeStamp] (Aggregation: Sum)
Disk Usage Utilization by Disk *
System (4)
Nightmares.exe <52df2...
HeadlessHost.exe <WP...
SVCHOST.EXE (1252)
SVCHOST.EXE (868)
SVCHOST.EXE (920)
RUNTIMEBROKER.EXE (...
Utilization per Disk using resource time as [Complete Time-Disk Service Time,Complete Time] (Aggregation: Sum)
Generic Events Activity by Provider, Task, Opcode
e13c0d23-ccbc-4e12-931b...
Microsoft-Windows-Direct...
Microsoft-Windows-D3D1...
UnderlyingResource
Microsoft-Windows-DXGI

Line #	Process	Stack	Count Sum	Weight (in vie...	TimeStamp (s)	% Weight Sum	Legend
1	Idle (0)	[Idle]	168	6 666,081101		49,80	▭
2	Nightmares.exe <52df2259-0d4a...		4 839	4 834,102588		36,11	🟩
3	System (4)		838	836,179269		6,25	🟦
4	AUDIODG.EXE (1432)		144	142,825443		1,07	🟦
5	SVCHOST.EXE (1252)	[Root]	50	50,001040		0,37	🟥

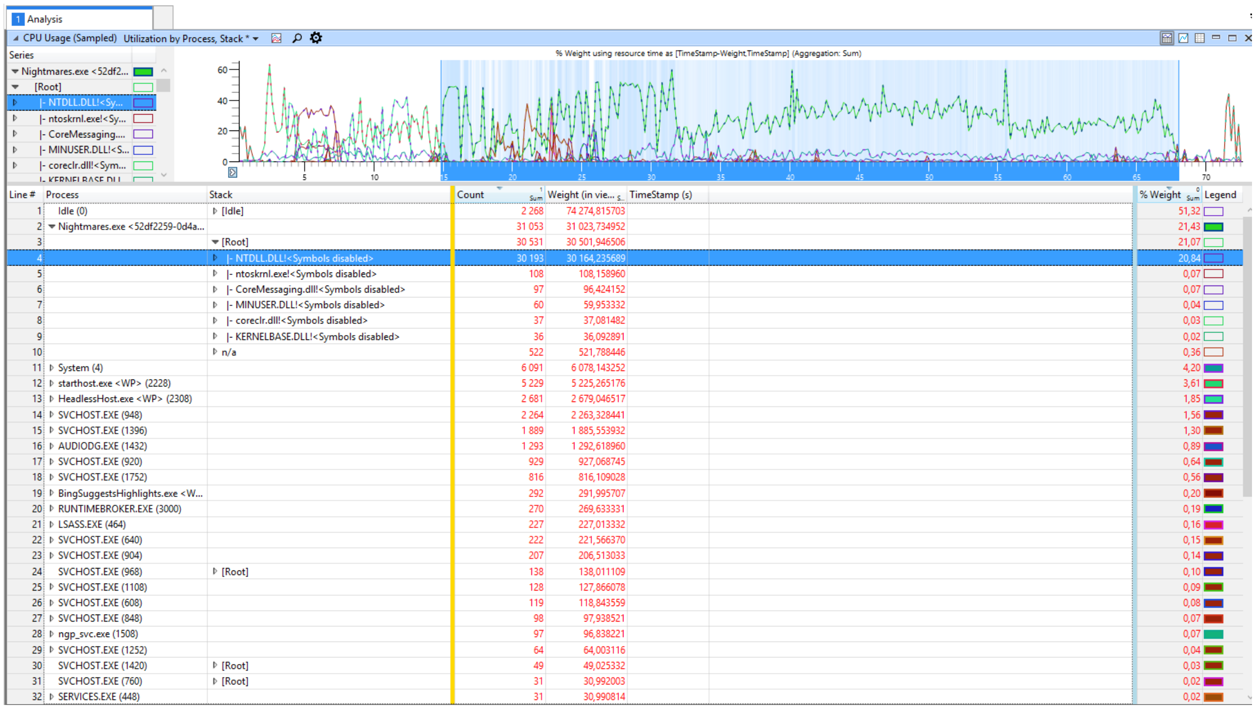
Line #	Process	Disk	Priority	IO Type	Complete Time (s)	IO Time (µs) Sum	Size (B) Sum	QD/I...	Disk Service Time...	Legend
1	System (4)	0				15 345 614,078	118 712 832		4 774 989,127	🟦
2	Nightmares.exe <52df2259-0...	0				2 054 983,608	6 856 192		199 855,933	🟩
3	HeadlessHost.exe <WP> (23...	0				592 880,435	176 128		17 341,187	🟩
4	SVCHOST.EXE (1252)	0				432 922,365	68 096		10 058,811	🟥
5	SVCHOST.EXE (868)	0				392 053,180	45 056		5 990,221	🟥
6	SVCHOST.EXE (920)	0				127 435,554	24 576		2 272,147	🟥
7	RUNTIMEBROKER.EXE (3000)	0				1 296,000	8 192		1 296,000	🟦
8	IExplore.exe <WP> (4060)	0				91 108,740	16 384		1 185,629	🟩

```
Start: 26.374404950s
  End: 33.067494629s
Duration: 6.693089679s
```

Using the Analysis tab

The Analysis tab contains event data from specific Event Providers. For each graph, the Analysis tab shows different data. Each row represents an event or an event group, while each column represents event data fields. The columns are divided into two groups:

- The columns to the left of the yellow line represent expandable event groups, grouped by column name in left-to-right priority.
- The columns to the right of the yellow line represent aggregated event data in the expandable event groups.

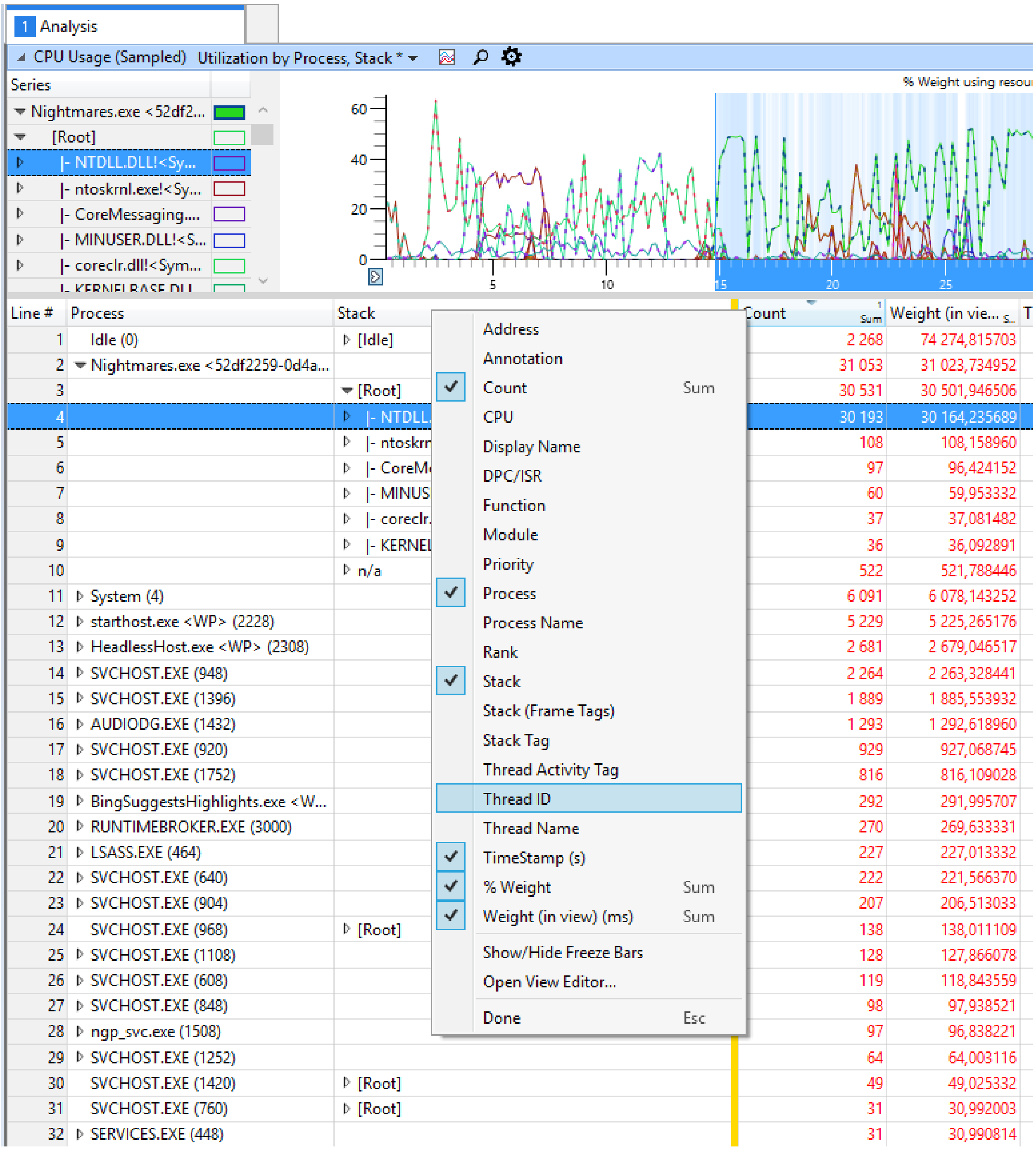


Line #	Process	Stack	Count	Weight (in vie...	TimeStamp (s)	% Weight	Legend
1	Idle (0)	[Idle]	2 268	74 274,815703		51,32	□
2	Nightmares.exe <52df2259-0d4a...		31 053	31 023,734952		21,43	■
3		[Root]	30 531	30 501,946506		21,07	□
4		|- NTDLL.DLL!<Symbols disabled>	30 193	30 164,235689		20,84	□
5		|- ntoskrnl.exe!<Symbols disabled>	108	108,158960		0,07	□
6		|- CoreMessaging.dll!<Symbols disabled>	97	96,424152		0,07	□
7		|- MINUSER.DLL!<Symbols disabled>	60	59,953332		0,04	□
8		|- coreclr.dll!<Symbols disabled>	37	37,081482		0,03	□
9		|- KERNELBASE.DLL!<Symbols disabled>	36	36,092891		0,02	□
10		n/a	522	521,788446		0,36	□
11	System (4)		6 091	6 078,143252		4,20	■
12	starthost.exe <WP> (2228)		5 229	5 225,265176		3,61	■
13	HeadlessHost.exe <WP> (2308)		2 681	2 679,046517		1,85	■
14	SVCHOST.EXE (948)		2 264	2 263,328441		1,56	■
15	SVCHOST.EXE (1396)		1 889	1 885,553932		1,30	■
16	AUDIODG.EXE (1432)		1 293	1 292,618960		0,89	■
17	SVCHOST.EXE (920)		929	927,068745		0,64	■
18	SVCHOST.EXE (1752)		816	816,109028		0,56	■
19	BingSuggestsHighlights.exe <W...		292	291,995707		0,20	■
20	RUNTIMEBROKER.EXE (3000)		270	269,633331		0,19	■
21	LSASS.EXE (464)		227	227,013332		0,16	■
22	SVCHOST.EXE (640)		222	221,566370		0,15	■
23	SVCHOST.EXE (904)		207	206,513033		0,14	■
24	SVCHOST.EXE (968)	[Root]	138	138,011109		0,10	■
25	SVCHOST.EXE (1108)		128	127,866078		0,09	■
26	SVCHOST.EXE (608)		119	118,843559		0,08	■
27	SVCHOST.EXE (848)		98	97,938521		0,07	■
28	ngp_svc.exe (1508)		97	96,838221		0,07	■
29	SVCHOST.EXE (1252)		64	64,003116		0,04	■
30	SVCHOST.EXE (1420)	[Root]	49	49,025332		0,03	■
31	SVCHOST.EXE (760)	[Root]	31	30,992003		0,02	■
32	SERVICES.EXE (448)		31	30,990814		0,02	■

Caption: Analysis tab view of a **CPU Usage (Sampled)** graph

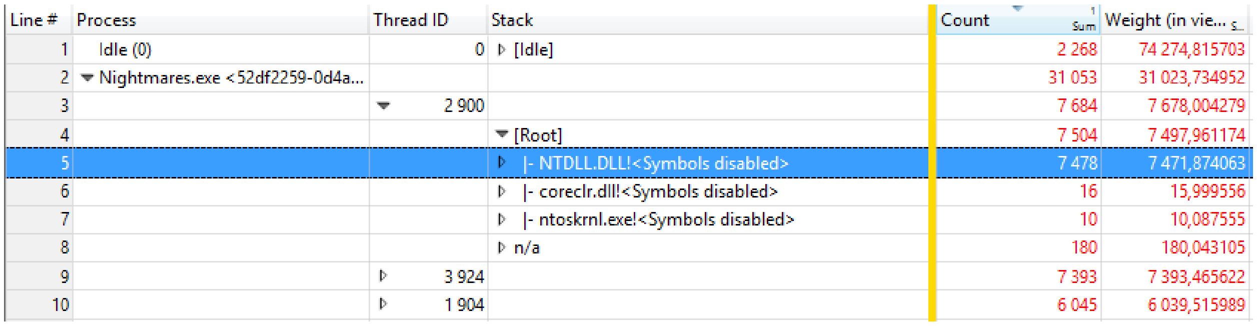
In this example screenshot, the event groups are first grouped by the Process column, then by Stack. On the right side of the yellow line is the aggregated sample count for each process, followed by the aggregated sample count for each stack frame group. For example, in the screenshot above, the spaces on the timeline highlighted in blue represent samples with a ntdll.dll function on top of their stack trace being taken from the **Nightmares.exe** process.

Add additional columns by right-clicking on the header row and selecting the desired column:



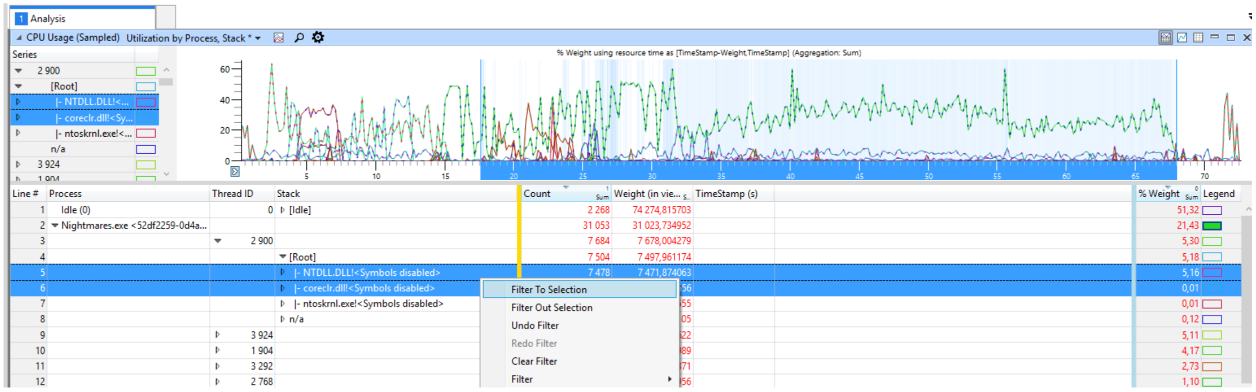
Line #	Process	Stack	Count	Weight (in vie...
1	Idle (0)	[Idle]	2 268	74 274,815703
2	Nightmares.exe <52df2259-0d4a...		31 053	31 023,734952
3		[Root]	30 531	30 501,946506
4		|- NTDLL.	30 193	30 164,235689
5		|- ntoskrn	108	108,158960
6		|- CoreM	97	96,424152
7		|- MINUS	60	59,953332
8		|- coreclr.	37	37,081482
9		|- KERNEL	36	36,092891
10		n/a	522	521,788446
11	System (4)		6 091	6 078,143252
12	starthost.exe <WP> (2228)		5 229	5 225,265176
13	HeadlessHost.exe <WP> (2308)		2 681	2 679,046517
14	SVCHOST.EXE (948)		2 264	2 263,328441
15	SVCHOST.EXE (1396)		1 889	1 885,553932
16	AUDIODG.EXE (1432)		1 293	1 292,618960
17	SVCHOST.EXE (920)		929	927,068745
18	SVCHOST.EXE (1752)		816	816,109028
19	BingSuggestsHighlights.exe <W...		292	291,995707
20	RUNTIMEBROKER.EXE (3000)		270	269,633331
21	LSASS.EXE (464)		227	227,013332
22	SVCHOST.EXE (640)		222	221,566370
23	SVCHOST.EXE (904)		207	206,513033
24	SVCHOST.EXE (968)	[Root]	138	138,011109
25	SVCHOST.EXE (1108)		128	127,866078
26	SVCHOST.EXE (608)		119	118,843559
27	SVCHOST.EXE (848)		98	97,938521
28	ngp_svc.exe (1508)		97	96,838221
29	SVCHOST.EXE (1252)		64	64,003116
30	SVCHOST.EXE (1420)	[Root]	49	49,025332
31	SVCHOST.EXE (760)	[Root]	31	30,992003
32	SERVICES.EXE (448)		31	30,990814

Dragged to the desired position as needed by left-clicking and holding as you move the column:

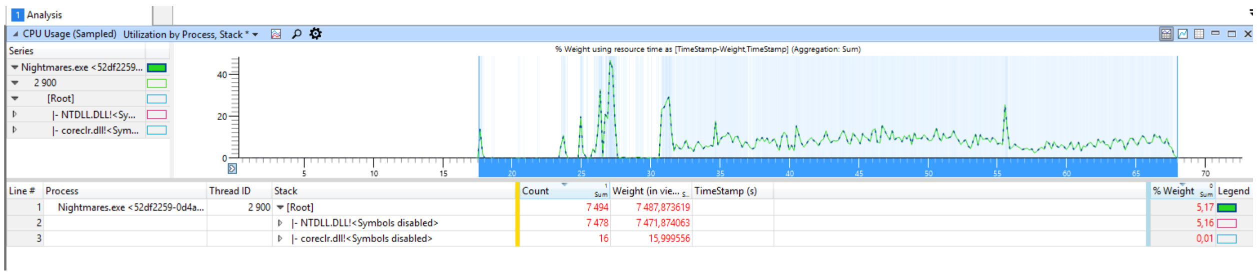


Line #	Process	Thread ID	Stack	Count	Weight (in vie...
1	Idle (0)	0	▷ [Idle]	2 268	74 274,815703
2	▼ Nightmares.exe <52df2259-0d4a...			31 053	31 023,734952
3		▼ 2 900		7 684	7 678,004279
4			▼ [Root]	7 504	7 497,961174
5			▷ |- NTDLL.DLL!<Symbols disabled>	7 478	7 471,874063
6			▷ |- coreclr.dll!<Symbols disabled>	16	15,999556
7			▷ |- ntoskrnl.exe!<Symbols disabled>	10	10,087555
8			▷ n/a	180	180,043105
9		▷ 3 924		7 393	7 393,465622
10		▷ 1 904		6 045	6 039,515989

The Analyze tab can be filtered on a very detailed basis. Select the rows you want to filter, right-click on them, and select the desired filtering option:


1 Analysis
CPU Usage (Sampled) Utilization by Process, Stack *
Series
% Weight using resource time as [TimeStamp-Weight.TimeStamp] (Aggregation: Sum)
Filter To Selection
Filter Out Selection
Undo Filter
Redo Filter
Clear Filter
Filter

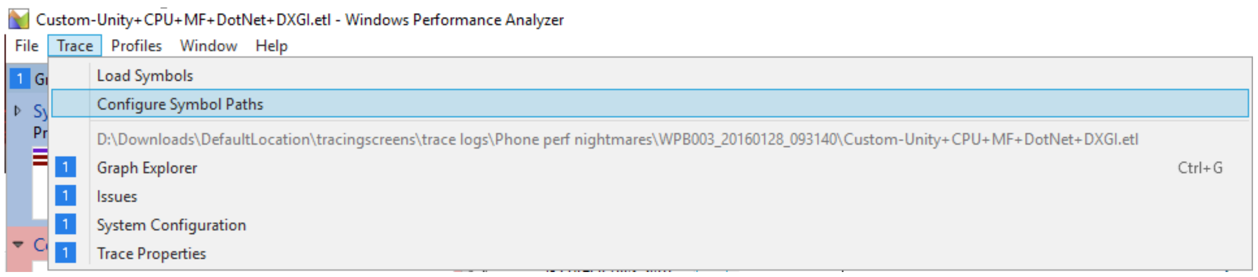
Line #	Process	Thread ID	Stack	Count	Weight (in vie...	TimeStamp (s)	% Weight	Legend
1	Idle (0)	0	▷ [Idle]	2 268	74 274,815703		51,32	▭
2	▼ Nightmares.exe <52df2259-0d4a...			31 053	31 023,734952		21,43	🟩
3		▼ 2 900		7 684	7 678,004279		5,30	▭
4			▼ [Root]	7 504	7 497,961174		5,18	▭
5			▷ |- NTDLL.DLL!<Symbols disabled>	7 478	7 471,874063		5,16	▭
6			▷ |- coreclr.dll!<Symbols disabled>		56		0,01	▭
7			▷ |- ntoskrnl.exe!<Symbols disabled>		55		0,01	▭
8			▷ n/a		05		0,12	▭
9		▷ 3 924			22		5,11	▭
10		▷ 1 904			89		4,17	▭
11		▷ 3 292			71		2,73	▭
12		▷ 2 768			56		1,10	▭


1 Analysis
CPU Usage (Sampled) Utilization by Process, Stack *
Series
Nightmares.exe <52df2259...
2 900
[Root]
|- NTDLL.DLL!<Sy...
|- coreclr.dll!<Sym...
% Weight using resource time as [TimeStamp-Weight.TimeStamp] (Aggregation: Sum)

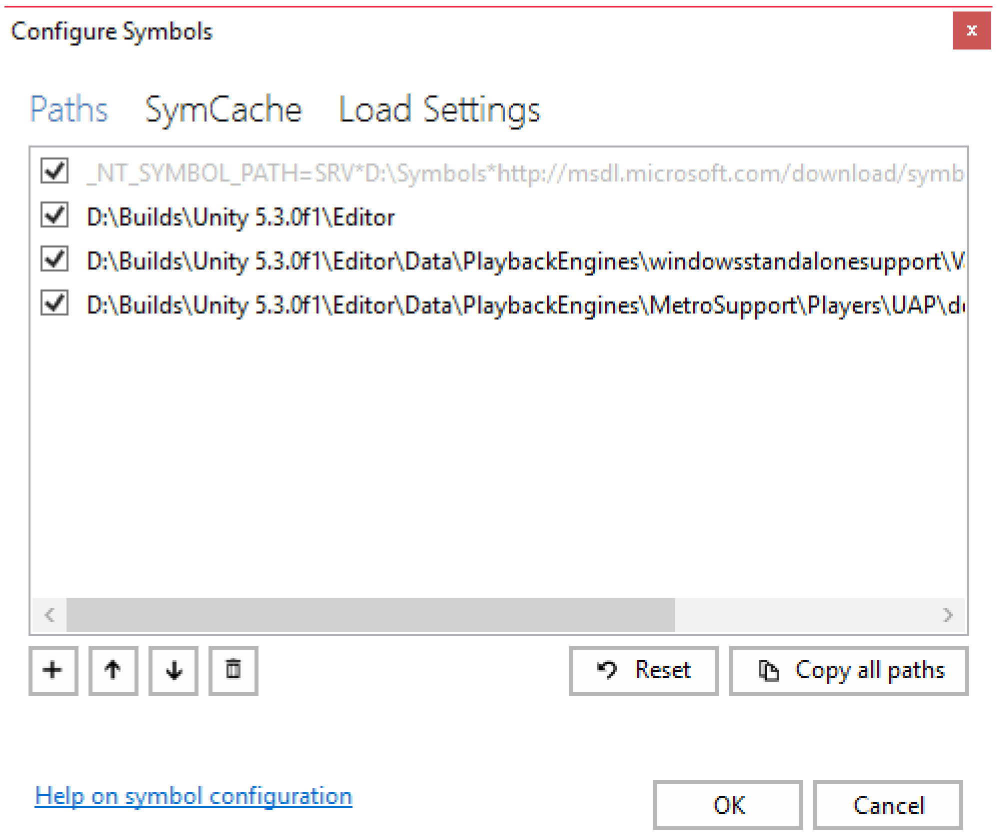
Line #	Process	Thread ID	Stack	Count	Weight (in vie...	TimeStamp (s)	% Weight	Legend
1	Nightmares.exe <52df2259-0d4a...	2 900	▼ [Root]	7 494	7 487,873619		5,17	🟩
2			▷ |- NTDLL.DLL!<Symbols disabled>	7 478	7 471,874063		5,16	▭
3			▷ |- coreclr.dll!<Symbols disabled>	16	15,999556		0,01	▭

Loading symbols

To inspect the captured stack traces, you need to load the executable symbol files (with the extension .pdb) into Windows Performance Analyzer. To do this, you first need to set the correct symbol paths. Open the Trace menu and click on Configure Symbol Paths:



D:\Downloads\DefaultLocation\tracingscreens\trace logs\Phone perf nightmares\WPB003_20160128_093140\Custom-Unity+CPU+MF+DotNet+DXGI.etl



- [✓] _NT_SYMBOL_PATH=SRV*D:\Symbols*http://msdl.microsoft.com/download/symb
- [✓] D:\Builds\Unity 5.3.0f1\Editor
- [✓] D:\Builds\Unity 5.3.0f1\Editor\Data\PlaybackEngines\windowsstandalonesupport\V
- [✓] D:\Builds\Unity 5.3.0f1\Editor\Data\PlaybackEngines\MetroSupport\Players\UAP\d

The first path in the list points to the Microsoft Symbol Servers. Windows Performance Analyzer knows how to download symbol files for OS DLLs from it. In this example, the symbol server path is SRV*D:\Symbols*http://msdl.microsoft.com/download/symbols.

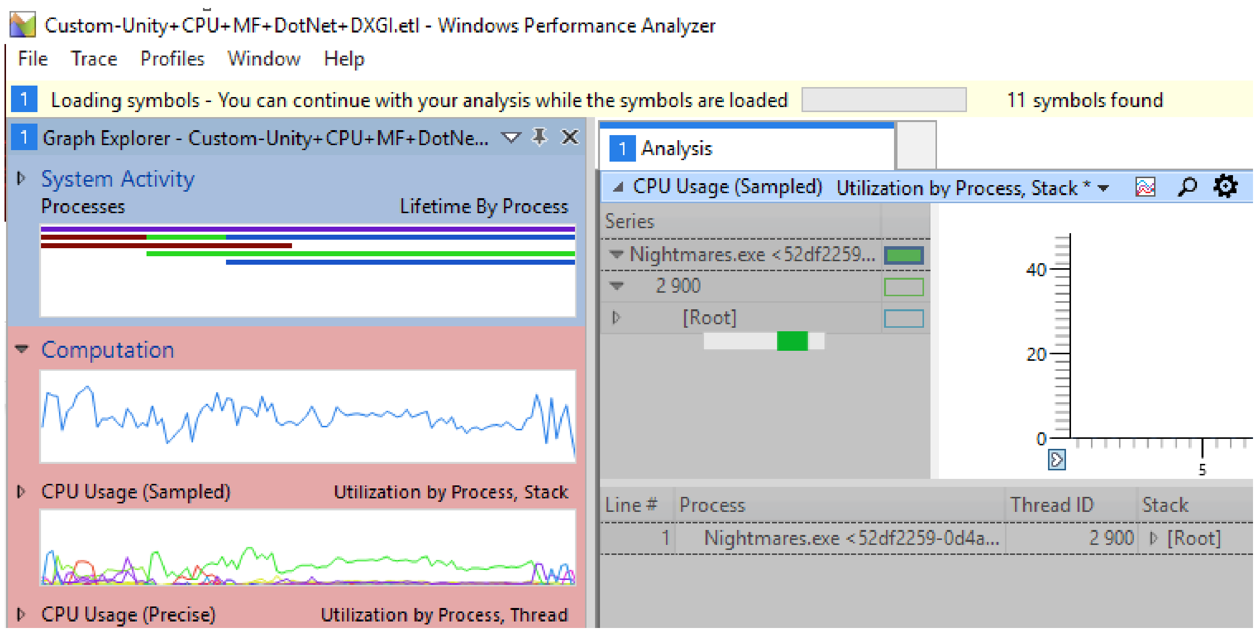
Note: The first part of the server path, **SRV**, indicates that the path points to a symbol server. The second part of the path, **D:\Symbols**, indicates which directory the symbols are downloaded to from the server. The third part of the path, http://msdl.microsoft.com/download/symbols, is the URL to the Microsoft Symbol Servers.

The symbol server path can also be set automatically by setting a **_NT_SYMBOL_PATH** environment variable to the path string. This means that Windows Performance Analyzer (and many other tools) can use this without you needing to manually configure it.

When you profile your game, you also need to add paths to Unity symbols and any plugins you might use. Unity symbols are automatically installed with Unity. You can find them next to the executables:

- **Windows Editor:** <UnityInstallDir>\Editor
- **Windows Standalone Player:** <UnityInstallDir>\Editor\Data\PlaybackEngines\windowsstandalonesupport\Variations\<PlayerType>
- **Windows Store Player with .NET scripting backend:**
  - <UnityInstallDir>\Editor\Data\PlaybackEngines\metrosupport\Players\<SDK>\<CPU Architecture>\<Configuration>\
  - <GeneratedVSSolutionDir>\<ProjectName>\bin\<CPU Architecture>\<Configuration>
- **Windows Store Player with IL2CPP scripting backend:** <GeneratedVSSolutionDir>\build\<CPU Architecture>\<Configuration>\

After adding all desired symbol paths, go to **Trace > Load symbols**:


Custom-Unity+CPU+MF+DotNet+DXGI.etl - Windows Performance Analyzer
File
Trace
Profiles
Window
Help
Graph Explorer - Custom-Unity+CPU+MF+DotNe...
System Activity
Processes
Lifetime By Process
Computation
CPU Usage (Sampled)
Utilization by Process, Stack
CPU Usage (Precise)
Utilization by Process, Thread
Analysis
CPU Usage (Sampled)
Utilization by Process, Stack *
Series
Nightmares.exe <52df2259...
2 900
[Root]
40
20
0
5

1 Loading symbols - You can continue with your analysis while the symbols are loaded   11 symbols found

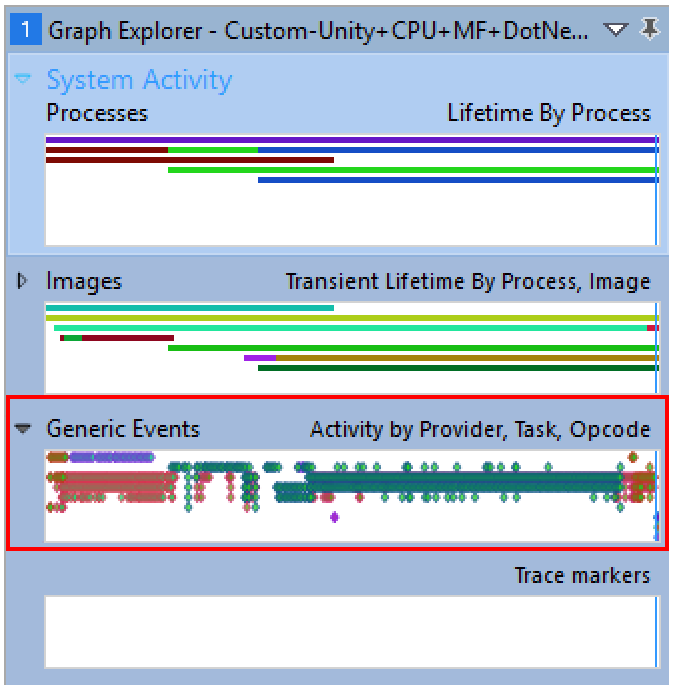
Line #	Process	Thread ID	Stack
1	Nightmares.exe <52df2259-0d4a...	2 900	[Root]

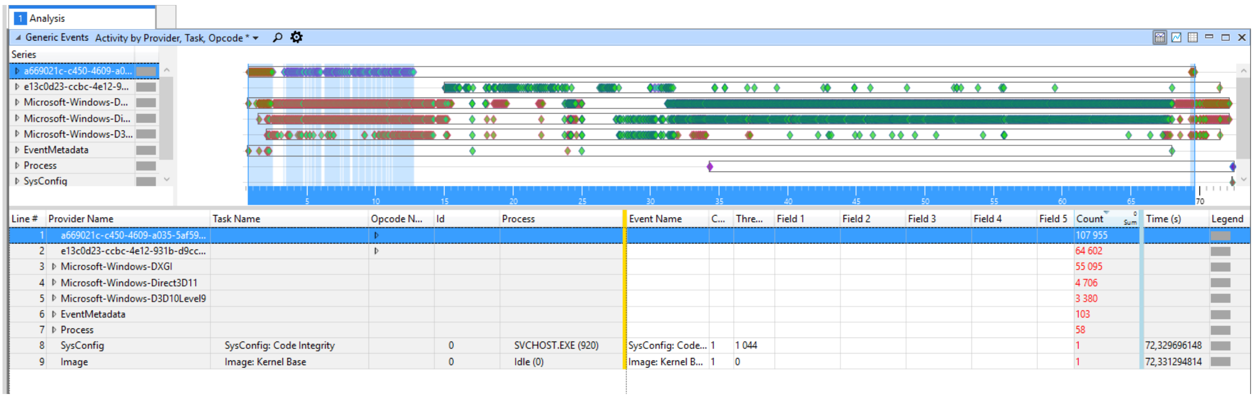
Loading symbols for the first time can take a while, especially if you're on a slow internet connection. Subsequent loads will be faster, as the symbols are cached on your machine.

Frame rate provider

Every time DirectX presents a frame to the screen, it logs an ETW event named **IDXGISwapChain_Present**. It outputs two such events per frame: one when the presentation starts, and one when it finishes. The time between finish events indicates how long each frame is.

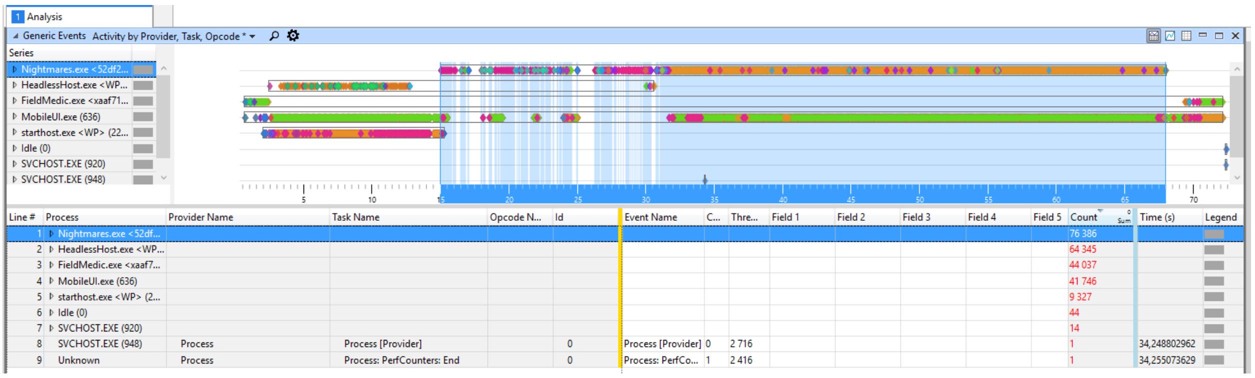
To see this event in Windows Performance Analyzer, you need to capture your trace with the DXGI Event Provider enabled. To view the data, expand the **System Activity** graph in the Graph Explorer and then double-click on the **Generic Events** graph:


1 Graph Explorer - Custom-Unity+CPU+MF+DotNe...
System Activity
Processes
Lifetime By Process
Images
Transient Lifetime By Process, Image
Generic Events
Activity by Provider, Task, Opcode
Trace markers



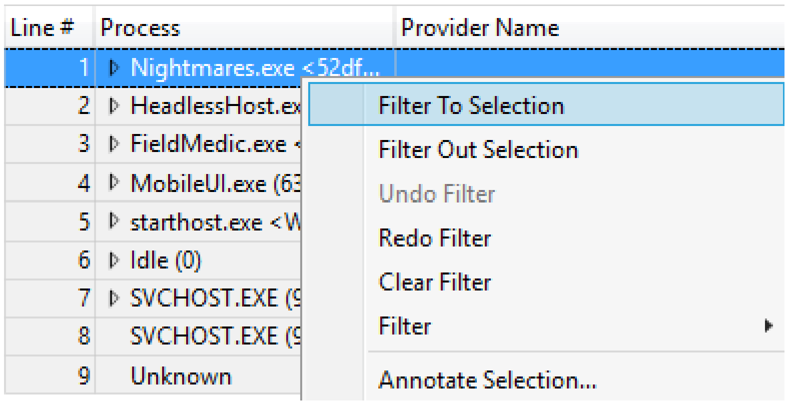
Line #	Provider Name	Task Name	Opcode N...	Id	Process	Event Name	C...	Thre...	Field 1	Field 2	Field 3	Field 4	Field 5	Count	Time (s)	Legend
1	a669021c-c450-4609-a035-5af59...													107 955		
2	e13c0d23-ccbc-4e12-931b-d9cc...													64 602		
3	Microsoft-Windows-DXGI													55 095		
4	Microsoft-Windows-Direct3D11													4 706		
5	Microsoft-Windows-D3D10Level9													3 380		
6	EventMetadata													103		
7	Process													58		
8	SysConfig	SysConfig: Code Integrity		0	SVCHOST.EXE (920)	SysConfig: Code...	1	1 044						1	72,329696148	
9	Image	Image: Kernel Base		0	Idle (0)	Image: Kernel B...	1	0						1	72,331294814	

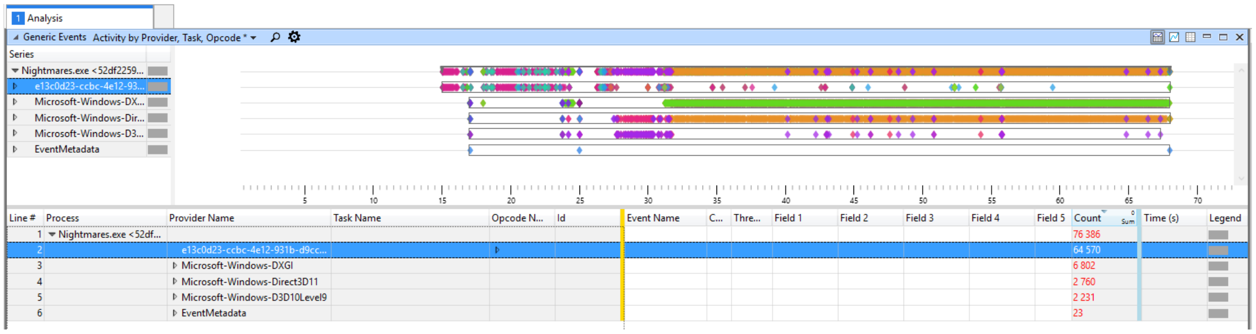
To see the frame rate for your process only, drag the **Process** column to the left so that it becomes the most significant grouping column. This groups all the events by process:



Line #	Process	Provider Name	Task Name	Opcode N...	Id	Event Name	C...	Thre...	Field 1	Field 2	Field 3	Field 4	Field 5	Count	Time (s)	Legend
1	Nightmares.exe <52df...													76 386		▨
2	HeadlessHost.exe <WP...													64 345		▨
3	FieldMedic.exe <xaaf7...													44 037		▨
4	MobileUI.exe (636)													41 746		▨
5	starthost.exe <WP> (2...													9 327		▨
6	Idle (0)													44		▨
7	SVCHOST.EXE (920)													14		▨
8	SVCHOST.EXE (948)	Process	Process [Provider]		0	Process [Provider]	0	2 716						1	34,248802962	▨
9	Unknown	Process	Process: PerfCounters: End		0	Process: PerfCo...	1	2 416						1	34,255073629	▨

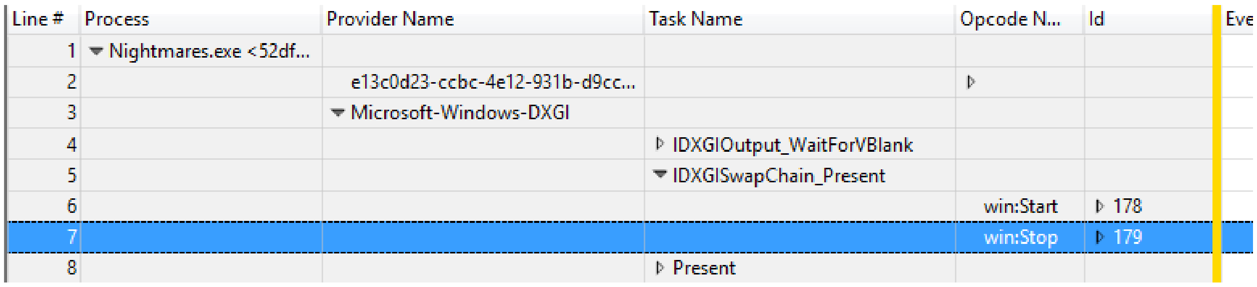
Right-click on the process you want to see and click **Filter To Selection** so that only the selected process is visible:





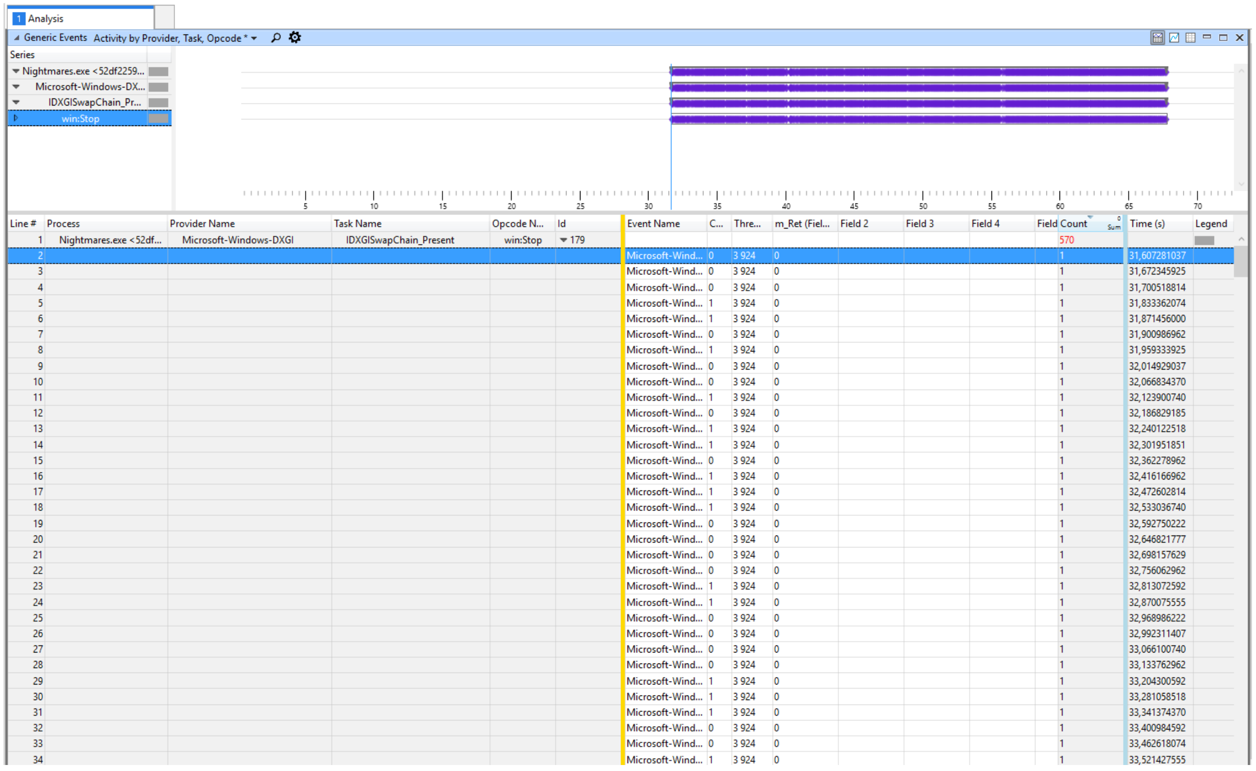
Line #	Process	Provider Name	Task Name	Opcode N...	Id	Event Name	C...	Thre...	Field 1	Field 2	Field 3	Field 4	Field 5	Count	Time (s)	Legend
1	Nightmares.exe <52df...													76 386		▨
2		e13c0d23-ccbc-4e12-931b-d9cc...												64 570		▨
3		Microsoft-Windows-DXGI												6 802		▨
4		Microsoft-Windows-Direct3D11												2 760		▨
5		Microsoft-Windows-D3D10Level9												2 231		▨
6		EventMetadata												23		▨

In the Provider Name column, open **Microsoft-Windows-DXGI**, then locate the row that contains **win.Stop** in the Opcode Name column. Click this row to filter to the events traced at the end of each **IDXGISwapChain_Present** presentation.



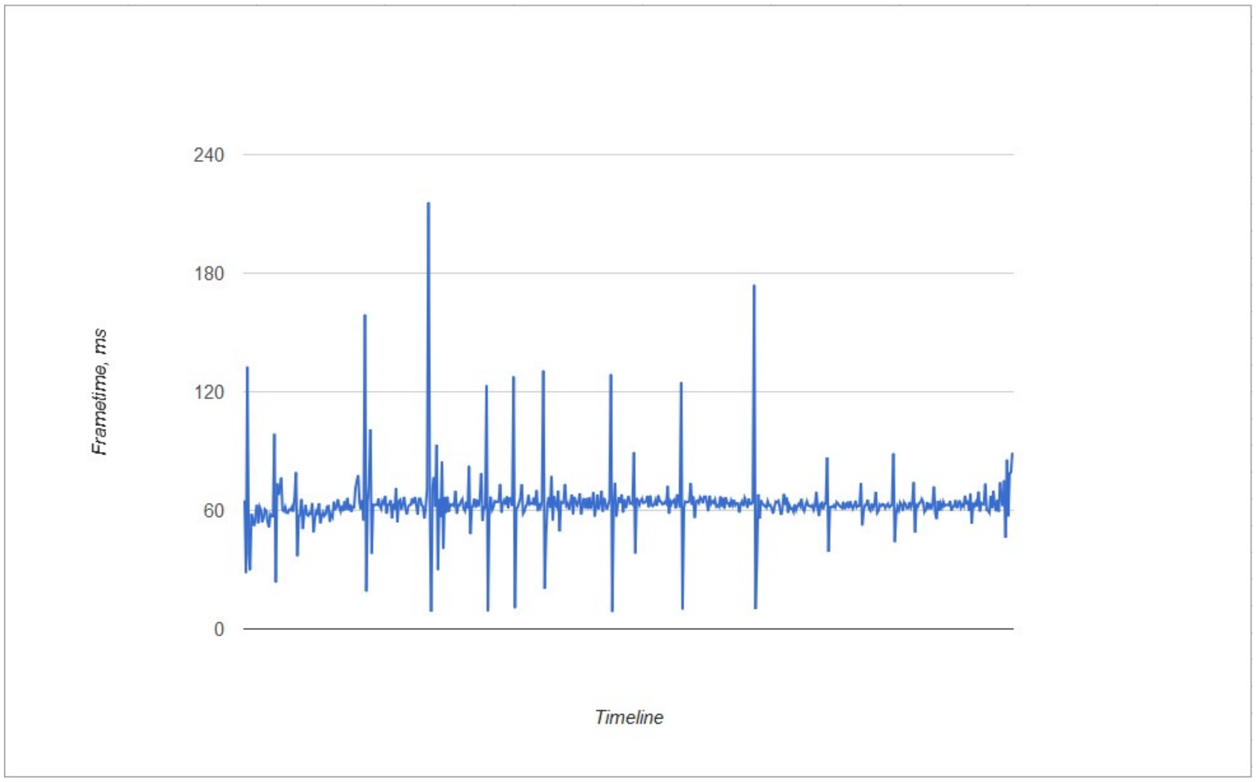
Line #	Process	Provider Name	Task Name	Opcode N...	Id	Eve
1	▼ Nightmares.exe <52df...					
2		e13c0d23-ccbc-4e12-931b-d9cc...		▷		
3		▼ Microsoft-Windows-DXGI				
4			▷ IDXGIOutput_WaitForVBlank			
5			▼ IDXGISwapChain_Present			
6				win:Start	▷ 178	
7				win:Stop	▷ 179	
8			▷ Present			

This filter allows you to see all the distinct frames that happened during the capture:

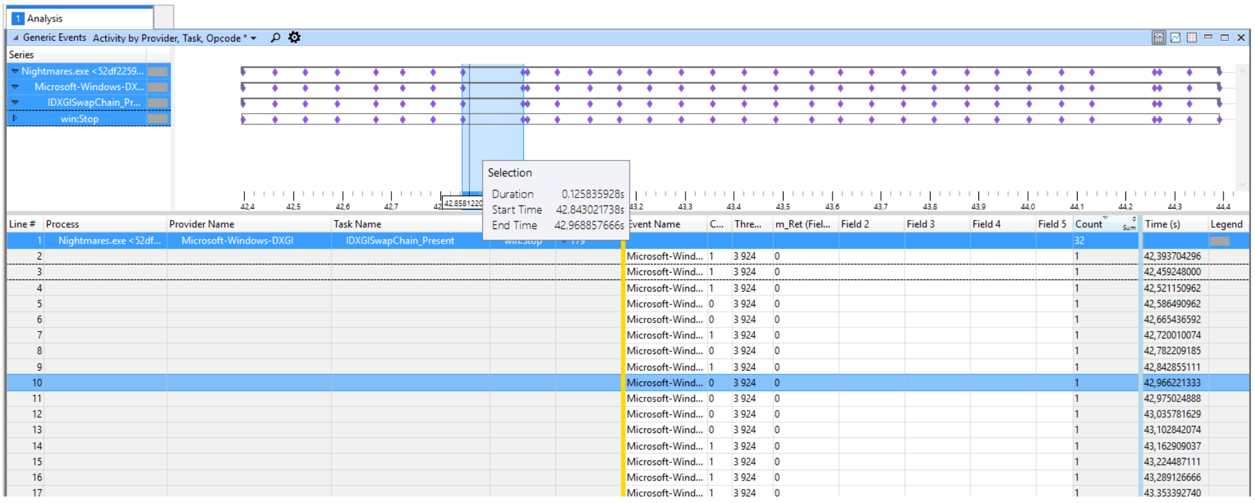


Line #	Process	Provider Name	Task Name	Opcode N...	Id	Event Name	C...	Thre...	m_Ret (Fiel...	Field 2	Field 3	Field 4	Field	Count	Time (s)	Legend
1	Nightmares.exe <52df...	Microsoft-Windows-DXGI	IDXGISwapChain_Present	win:Stop	▼ 179									570		▨
2						Microsoft-Wind...	0	3 924	0					1	31,607281037	
3						Microsoft-Wind...	0	3 924	0					1	31,672345925	
4						Microsoft-Wind...	0	3 924	0					1	31,700518814	
5						Microsoft-Wind...	1	3 924	0					1	31,833362074	
6						Microsoft-Wind...	1	3 924	0					1	31,871456000	
7						Microsoft-Wind...	0	3 924	0					1	31,900986962	
8						Microsoft-Wind...	1	3 924	0					1	31,959333925	
9						Microsoft-Wind...	0	3 924	0					1	32,014929037	
10						Microsoft-Wind...	0	3 924	0					1	32,066834370	
11						Microsoft-Wind...	1	3 924	0					1	32,123900740	
12						Microsoft-Wind...	0	3 924	0					1	32,186829185	
13						Microsoft-Wind...	1	3 924	0					1	32,240122518	
14						Microsoft-Wind...	1	3 924	0					1	32,301951851	
15						Microsoft-Wind...	0	3 924	0					1	32,362278962	
16						Microsoft-Wind...	1	3 924	0					1	32,416166962	
17						Microsoft-Wind...	1	3 924	0					1	32,472602814	
18						Microsoft-Wind...	1	3 924	0					1	32,533036740	
19						Microsoft-Wind...	0	3 924	0					1	32,592750222	
20						Microsoft-Wind...	0	3 924	0					1	32,646821777	
21						Microsoft-Wind...	0	3 924	0					1	32,698157629	
22						Microsoft-Wind...	0	3 924	0					1	32,756062962	
23						Microsoft-Wind...	1	3 924	0					1	32,813072592	
24						Microsoft-Wind...	1	3 924	0					1	32,870075555	
25						Microsoft-Wind...	0	3 924	0					1	32,968986222	
26						Microsoft-Wind...	0	3 924	0					1	32,992311407	
27						Microsoft-Wind...	0	3 924	0					1	33,066100740	
28						Microsoft-Wind...	0	3 924	0					1	33,133762962	
29						Microsoft-Wind...	1	3 924	0					1	33,204300592	
30						Microsoft-Wind...	1	3 924	0					1	33,281058518	
31						Microsoft-Wind...	1	3 924	0					1	33,341374370	
32						Microsoft-Wind...	0	3 924	0					1	33,400984592	
33						Microsoft-Wind...	0	3 924	0					1	33,462618074	
34						Microsoft-Wind...	1	3 924	0					1	33,521427555	

An easy way to see frame rate spikes is to copy and paste these frame times into a spreadsheet program and convert them into a line graph:



You can also inspect individual frames by zooming into the timeline. Usually when investigating low performance, the problem is either constant low frame rate or frame rate spikes. A zoomed-in view helps you find time periods to focus the analysis on when focusing on the whole trace is impractical:



Selection

Duration 0,125835928s
Start Time 42,843021738s
End Time 42,968857666s

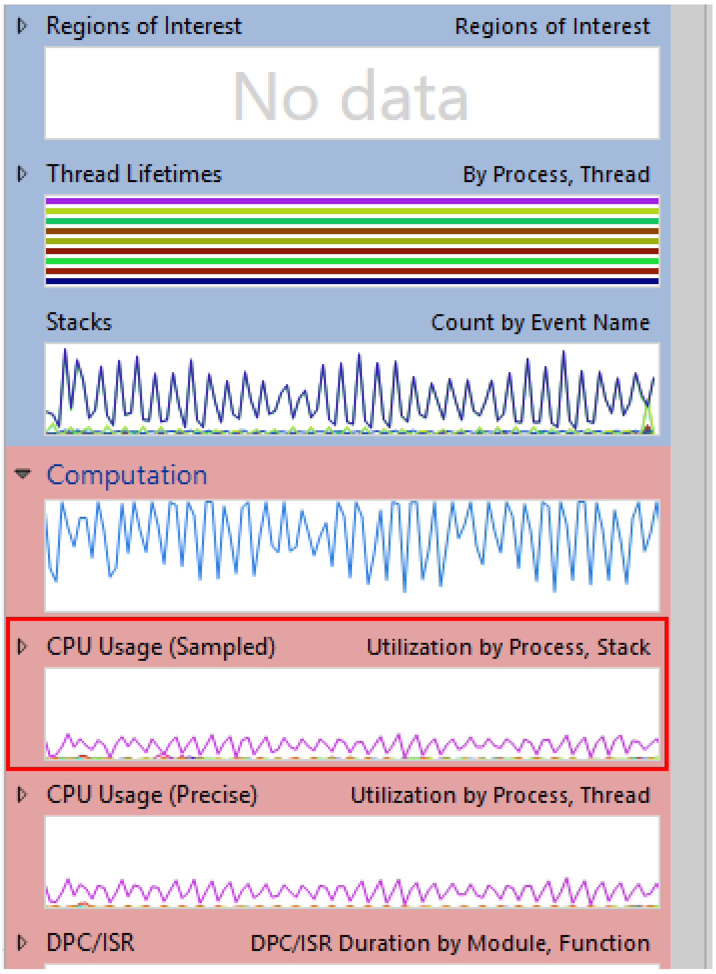
Line #	Process	Provider Name	Task Name			Event Name	C...	Thre...	m_Ret (Fiel...	Field 2	Field 3	Field 4	Field 5	Count	Time (s)	Legend
1	Nightmares.exe <52df...	Microsoft-Windows-DXGI	IDXGISwapChain_Present	win:Stop	179									32		▬
2						Microsoft-Wind...	1	3 924	0					1	42,393704296	
3						Microsoft-Wind...	1	3 924	0					1	42,459248000	
4						Microsoft-Wind...	1	3 924	0					1	42,521150962	
5						Microsoft-Wind...	0	3 924	0					1	42,586490962	
6						Microsoft-Wind...	0	3 924	0					1	42,665436592	
7						Microsoft-Wind...	1	3 924	0					1	42,720010074	
8						Microsoft-Wind...	0	3 924	0					1	42,782209185	
9						Microsoft-Wind...	1	3 924	0					1	42,842855111	
10						Microsoft-Wind...	0	3 924	0					1	42,966221333	
11						Microsoft-Wind...	0	3 924	0					1	42,975024888	
12						Microsoft-Wind...	0	3 924	0					1	43,035781629	
13						Microsoft-Wind...	0	3 924	0					1	43,102842074	
14						Microsoft-Wind...	1	3 924	0					1	43,162909037	
15						Microsoft-Wind...	1	3 924	0					1	43,224487111	
16						Microsoft-Wind...	1	3 924	0					1	43,289126666	
17						Microsoft-Wind...	1	3 924	0					1	43,353392740	

CPU Usage (Sampled) provider

The CPU Usage (Sampled) profiler logs what every CPU core is doing every millisecond - that's 1000 samples per second per CPU core. The accuracy of this provider is not 100%; it doesn't know how long each particular function has taken, just that it was executing when the program was sampled.

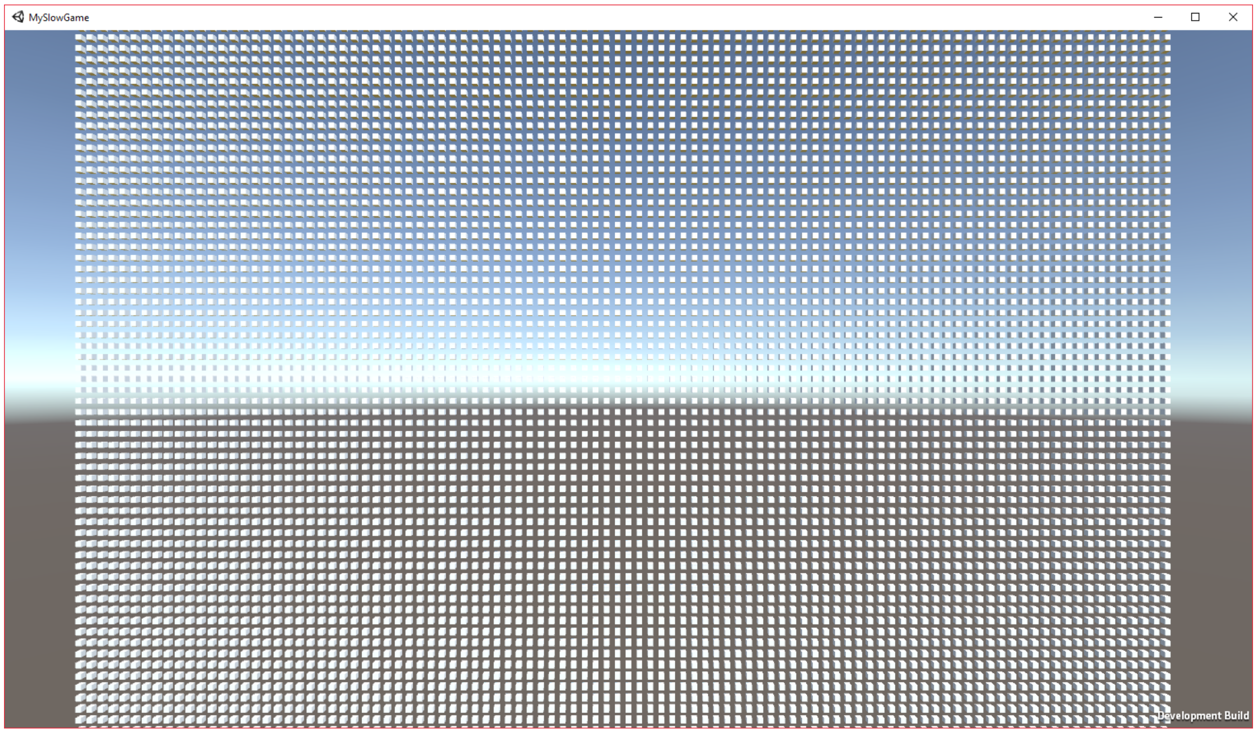
The provider is useful for investigating a program's CPU usage over the length of a capture. Statistically, the more samples it takes, the more accurate it becomes, so it's recommended to use this provider when the profiling time is at least 100 ms. Using it to analyze shorter periods of time can be inaccurate.

To bring the CPU Usage (Sampled) event provider into the Analysis tab, double-click on it in the Graph Explorer:



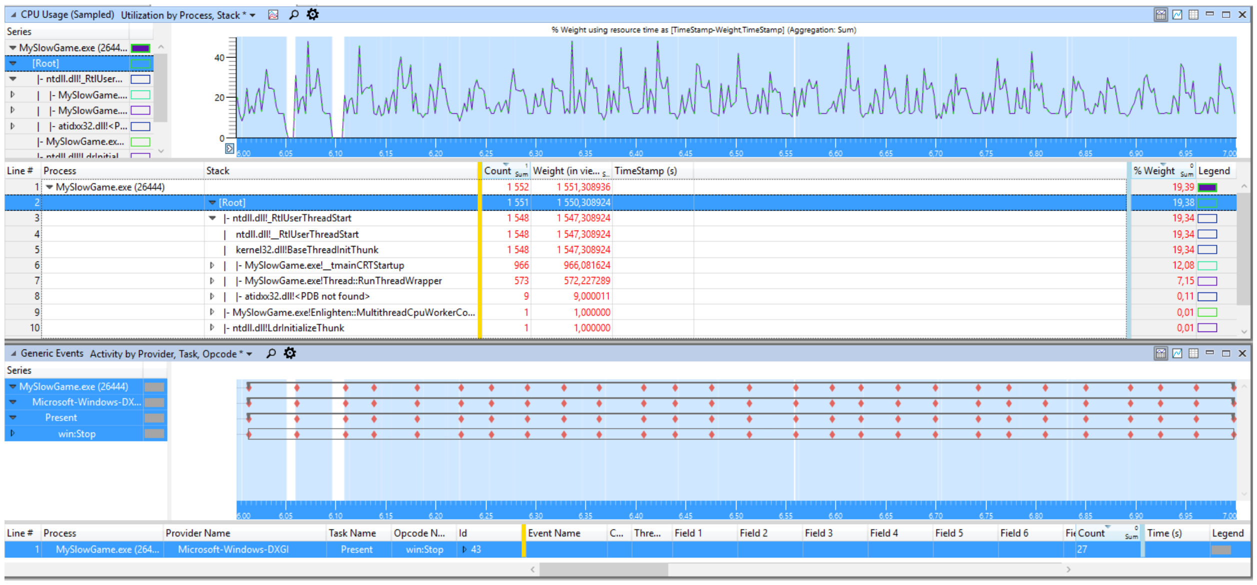
Example walkthrough

In this example, we have a game that has lower performance than we'd like.



Caption: Example game screenshot

The first step was to take a look at the frame rate, which in this instance seemed to be pretty stable. I picked a particular time period, between the 6th and 7th second of the capture. In this particular second, it had just over 27 frames. This is a very good case for using a **CPU usage (Sampled) provider** to investigate, as we have enough data points for it to be useful:


CPU Usage (Sampled) Utilization by Process, Stack *

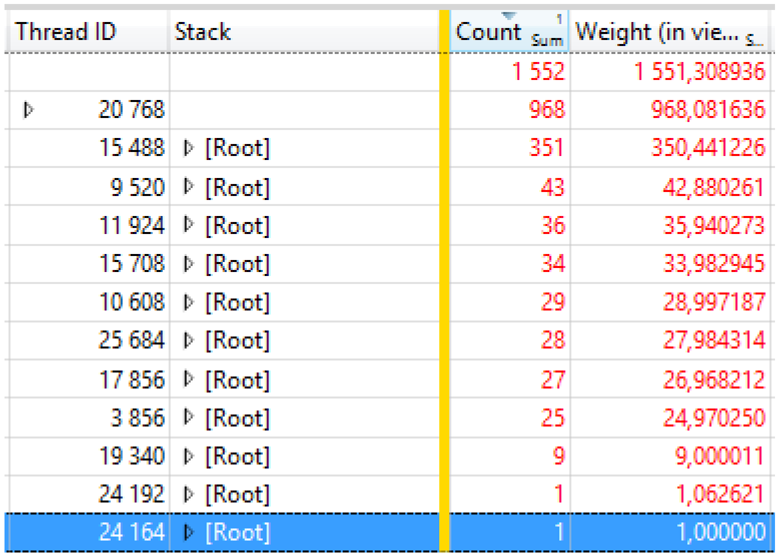
% Weight using resource time as [TimeStamp-Weight,TimeStamp] (Aggregation: Sum)

Line #	Process	Stack	Count Sum	Weight (in vie...	TimeStamp (s)	% Weight Sum	Legend
1	MySlowGame.exe (26444)		1 552	1 551,308936		19,39	■
2		[Root]	1 551	1 550,308924		19,38	▭
3		|- ntdll.dll!_RtlUserThreadStart	1 548	1 547,308924		19,34	▭
4		| ntdll.dll!__RtlUserThreadStart	1 548	1 547,308924		19,34	▭
5		| kernel32.dll!BaseThreadInitThunk	1 548	1 547,308924		19,34	▭
6		| |- MySlowGame.exe!__tmainCRTStartup	966	966,081624		12,08	▭
7		| |- MySlowGame.exe!Thread::RunThreadWrapper	573	572,227289		7,15	▭
8		| |- atidxx32.dll!<PDB not found>	9	9,000011		0,11	▭
9		|- MySlowGame.exe!Enlighten::MultithreadCpuWorkerCo...	1	1,000000		0,01	▭
10		|- ntdll.dll!LdrInitializeThunk	1	1,000000		0,01	▭

Generic Events Activity by Provider, Task, Opcode *

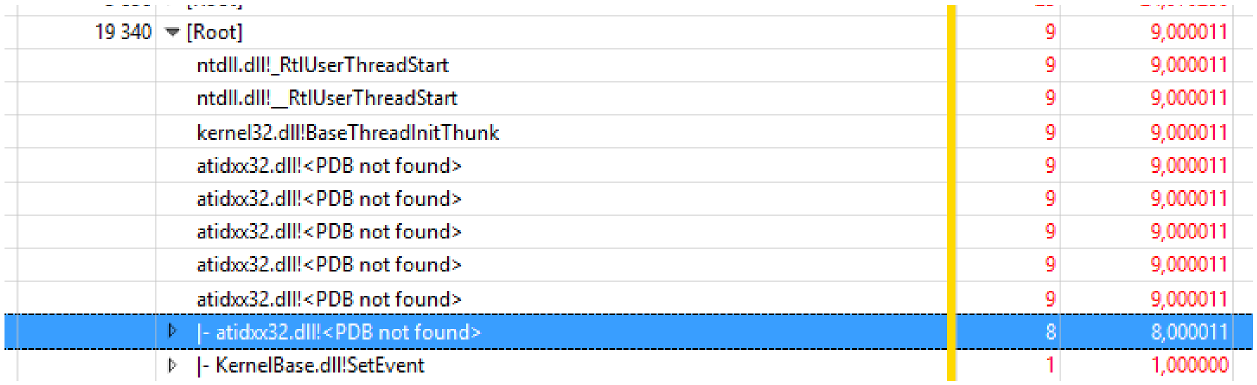
Line #	Process	Provider Name	Task Name	Opcode N...	Id	Event Name	C...	Thre...	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6	Fi	Count Sum	Time (s)	Legend
1	MySlowGame.exe (264...	Microsoft-Windows-DXGI	Present	win:Stop	43											27		■

Generally when investigating performance issues in Unity games, you only need to focus on a subset of Thread IDs, as most threads do not impact frame rate at all (for example, audio threads and COM message loops). Drag the **Thread ID** column to the left-hand side, to group samples by the thread they were taken from:



Thread ID	Stack	Count Sum	Weight (in vie...
		1 552	1 551,308936
20 768		968	968,081636
15 488	[Root]	351	350,441226
9 520	[Root]	43	42,880261
11 924	[Root]	36	35,940273
15 708	[Root]	34	33,982945
10 608	[Root]	29	28,997187
25 684	[Root]	28	27,984314
17 856	[Root]	27	26,968212
3 856	[Root]	25	24,970250
19 340	[Root]	9	9,000011
24 192	[Root]	1	1,062621
24 164	[Root]	1	1,000000

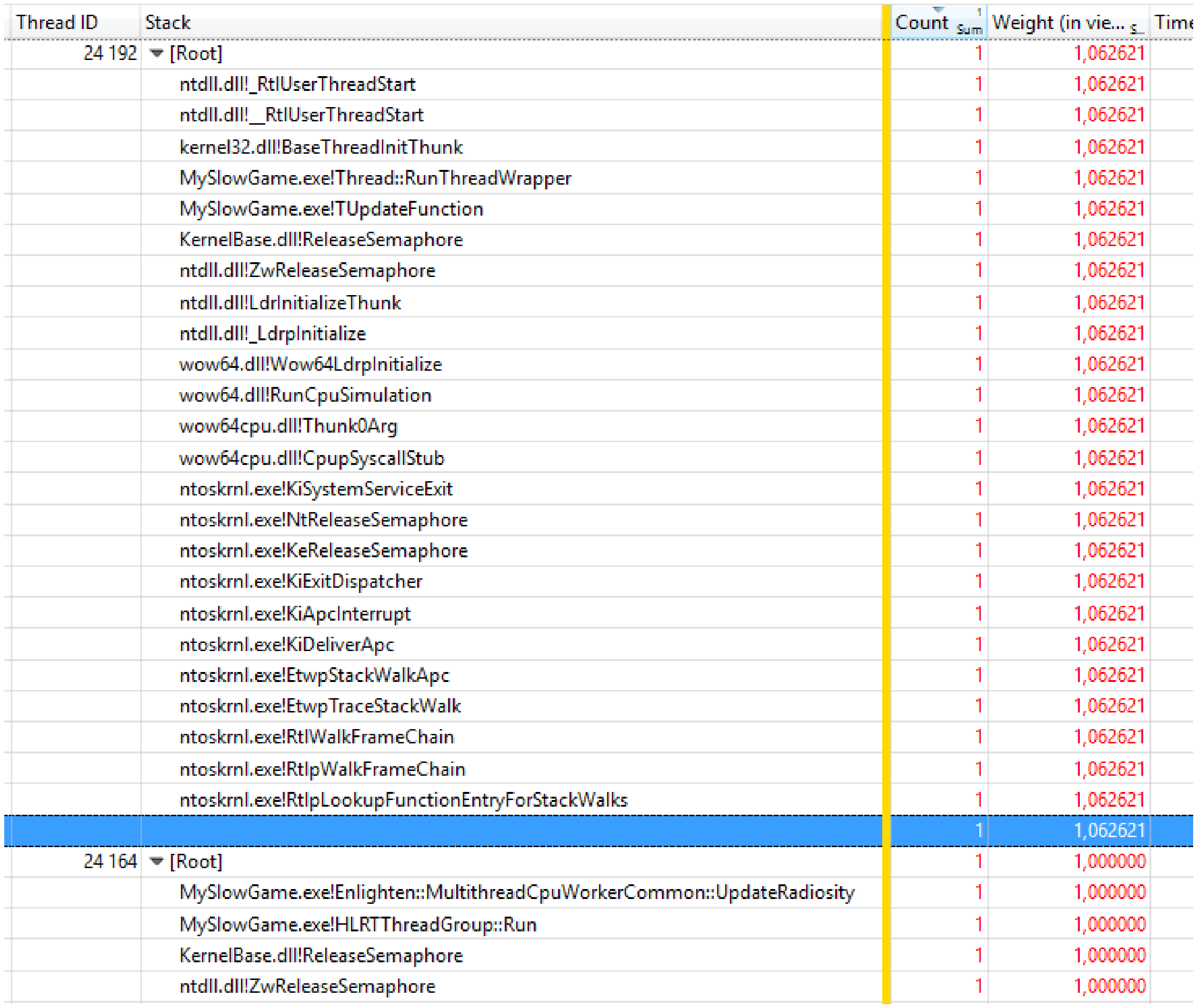
In this example, the three bottom threads aren't relevant to the frame rate problem. Thread **19340** is an internal AMD graphics driver thread (we can determine this by googling the name of the DLL if it isn't already known and can't be worked out from the name).



19 340	[Root]	9	9,000011
	ntdll.dll!_RtlUserThreadStart	9	9,000011
	ntdll.dll!__RtlUserThreadStart	9	9,000011
	kernel32.dll!BaseThreadInitThunk	9	9,000011
	atidxx32.dll!<PDB not found>	9	9,000011
	atidxx32.dll!<PDB not found>	9	9,000011
	atidxx32.dll!<PDB not found>	9	9,000011
	atidxx32.dll!<PDB not found>	9	9,000011
	atidxx32.dll!<PDB not found>	9	9,000011
	|- atidxx32.dll!<PDB not found>	8	8,000011
	|- KernelBase.dll!SetEvent	1	1,000000

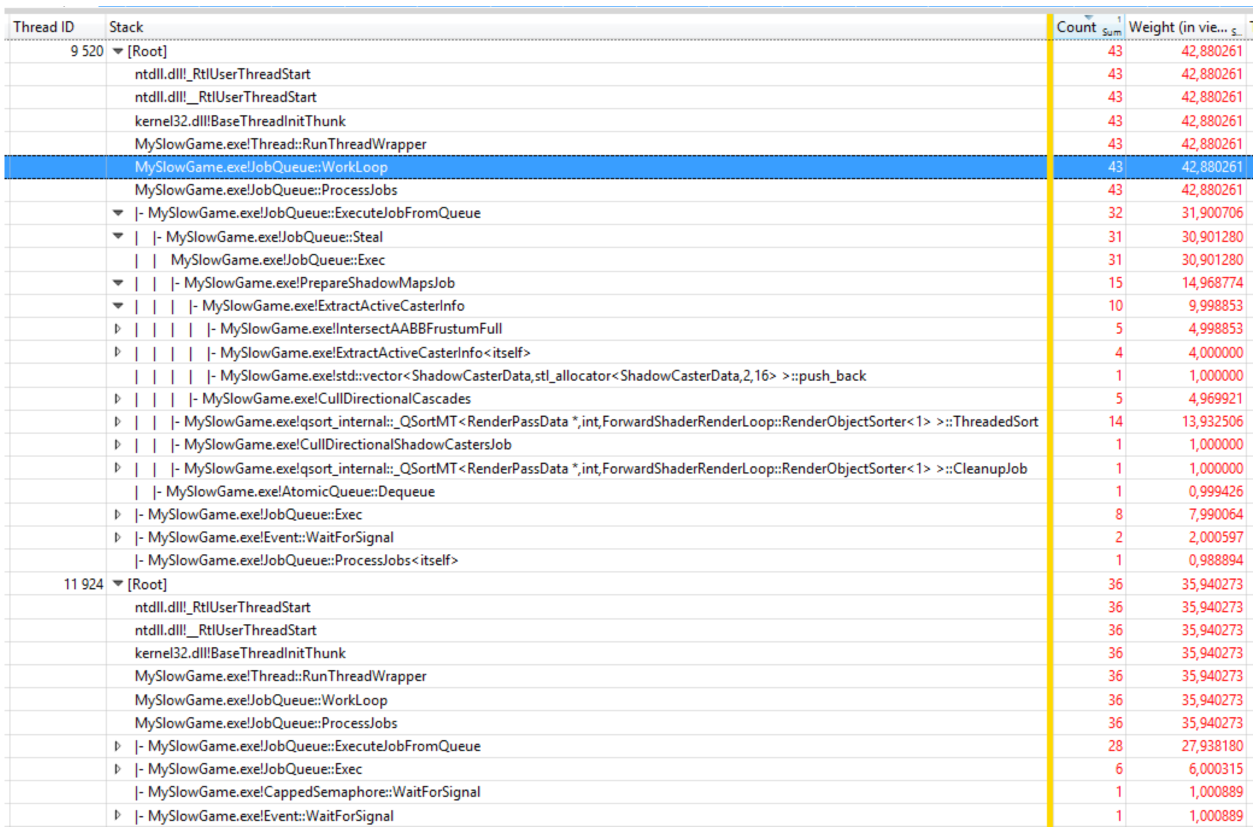
It might have been a candidate, being graphics-related, but since it took up so little time we can remove it from our list of suspects.

Threads **24192** and **24164** are Enlighten global illumination worker threads (our example game doesn't use global illumination, so these are mostly idle):



Thread ID	Stack	Count Sum	Weight (in vie... S...	Time
24 192	[Root]	1	1,062621	
	ntdll.dll!_RtlUserThreadStart	1	1,062621	
	ntdll.dll!__RtlUserThreadStart	1	1,062621	
	kernel32.dll!BaseThreadInitThunk	1	1,062621	
	MySlowGame.exe!Thread::RunThreadWrapper	1	1,062621	
	MySlowGame.exe!TUpdateFunction	1	1,062621	
	KernelBase.dll!ReleaseSemaphore	1	1,062621	
	ntdll.dll!ZwReleaseSemaphore	1	1,062621	
	ntdll.dll!LdrInitializeThunk	1	1,062621	
	ntdll.dll!_LdrpInitialize	1	1,062621	
	wow64.dll!Wow64LdrpInitialize	1	1,062621	
	wow64.dll!RunCpuSimulation	1	1,062621	
	wow64cpu.dll!Thunk0Arg	1	1,062621	
	wow64cpu.dll!CpupSyscallStub	1	1,062621	
	ntoskrnl.exe!KiSystemServiceExit	1	1,062621	
	ntoskrnl.exe!NtReleaseSemaphore	1	1,062621	
	ntoskrnl.exe!KeReleaseSemaphore	1	1,062621	
	ntoskrnl.exe!KiExitDispatcher	1	1,062621	
	ntoskrnl.exe!KiApcInterrupt	1	1,062621	
	ntoskrnl.exe!KiDeliverApc	1	1,062621	
	ntoskrnl.exe!EtwpStackWalkApc	1	1,062621	
	ntoskrnl.exe!EtwpTraceStackWalk	1	1,062621	
	ntoskrnl.exe!RtlWalkFrameChain	1	1,062621	
	ntoskrnl.exe!RtlpWalkFrameChain	1	1,062621	
	ntoskrnl.exe!RtlpLookupFunctionEntryForStackWalks	1	1,062621	
		1	1,062621	
24 164	[Root]	1	1,000000	
	MySlowGame.exe!Enlighten::MultithreadCpuWorkerCommon::UpdateRadiosity	1	1,000000	
	MySlowGame.exe!HLRTThreadGroup::Run	1	1,000000	
	KernelBase.dll!ReleaseSemaphore	1	1,000000	
	ntdll.dll!ZwReleaseSemaphore	1	1,000000	

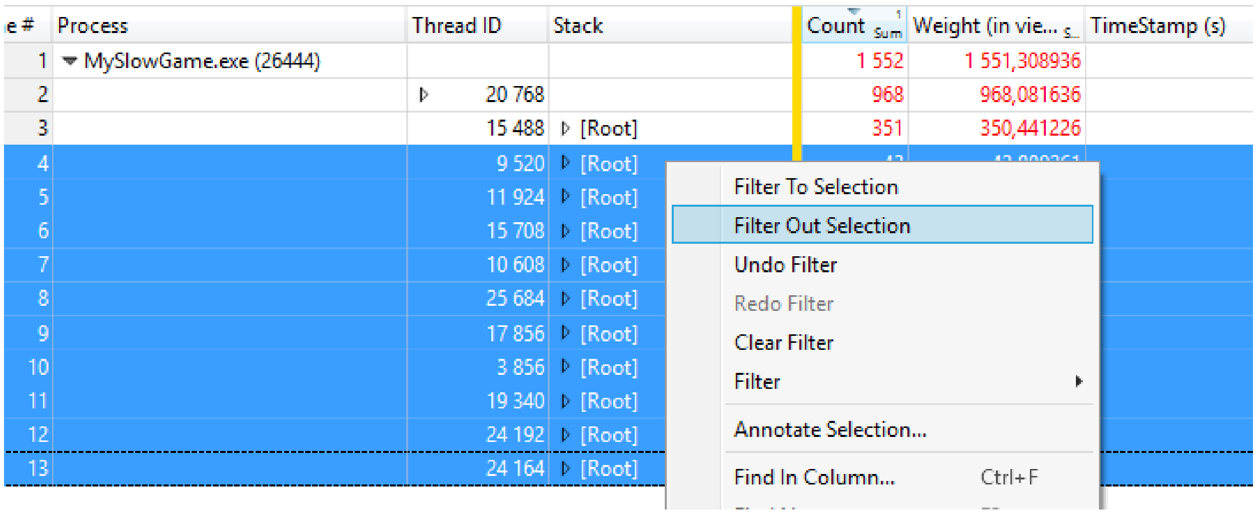
The following 7 threads (**9520**, **11924**, **15708**, **10608**, **25684**, **17856** and **3856**) are Unity JobQueue threads. They all share similar stacktraces, all starting with “**JobQueue::WorkLoop**”:



```
Thread ID  Stack                                                                                                                   Count  Weight (in vie...
    9 520  [Root]                                                                                                                     43  42,880261
             ntdll.dll!_RtlUserThreadStart                                                                                            43  42,880261
             ntdll.dll!__RtlUserThreadStart                                                                                           43  42,880261
             kernel32.dll!BaseThreadInitThunk                                                                                         43  42,880261
             MySlowGame.exe!Thread::RunThreadWrapper                                                                                  43  42,880261
             MySlowGame.exe!JobQueue::WorkLoop                                                                                        43  42,880261
             MySlowGame.exe!JobQueue::ProcessJobs                                                                                     43  42,880261
             |- MySlowGame.exe!JobQueue::ExecuteJobFromQueue                                                                          32  31,900706
             |  |- MySlowGame.exe!JobQueue::Steal                                                                                     31  30,901280
             |  |  MySlowGame.exe!JobQueue::Exec                                                                                      31  30,901280
             |  |  |- MySlowGame.exe!PrepareShadowMapsJob                                                                             15  14,968774
             |  |  |  |- MySlowGame.exe!ExtractActiveCasterInfo                                                                       10   9,998853
             |  |  |  |  |- MySlowGame.exe!IntersectAABBFrustumFull                                                                    5   4,998853
             |  |  |  |  |- MySlowGame.exe!ExtractActiveCasterInfo<itself>                                                             4   4,000000
             |  |  |  |  |- MySlowGame.exe!std::vector<ShadowCasterData,stl_allocator<ShadowCasterData,2,16> >::push_back               1   1,000000
             |  |  |  |- MySlowGame.exe!CullDirectionalCascades                                                                        5   4,969921
             |  |  |- MySlowGame.exe!qsort_internal::_QSortMT<RenderPassData *,int,ForwardShaderRenderLoop::RenderObjectSorter<1> >::ThreadedSort  14  13,932506
             |  |  |- MySlowGame.exe!CullDirectionalShadowCastersJob                                                                   1   1,000000
             |  |  |- MySlowGame.exe!qsort_internal::_QSortMT<RenderPassData *,int,ForwardShaderRenderLoop::RenderObjectSorter<1> >::CleanupJob     1   1,000000
             |  |- MySlowGame.exe!AtomicQueue::Dequeue                                                                                 1   0,999426
             |- MySlowGame.exe!JobQueue::Exec                                                                                          8   7,990064
             |- MySlowGame.exe!Event::WaitForSignal                                                                                    2   2,000597
             |- MySlowGame.exe!JobQueue::ProcessJobs<itself>                                                                           1   0,988894
   11 924  [Root]                                                                                                                     36  35,940273
             ntdll.dll!_RtlUserThreadStart                                                                                            36  35,940273
             ntdll.dll!__RtlUserThreadStart                                                                                           36  35,940273
             kernel32.dll!BaseThreadInitThunk                                                                                         36  35,940273
             MySlowGame.exe!Thread::RunThreadWrapper                                                                                  36  35,940273
             MySlowGame.exe!JobQueue::WorkLoop                                                                                        36  35,940273
             MySlowGame.exe!JobQueue::ProcessJobs                                                                                     36  35,940273
             |- MySlowGame.exe!JobQueue::ExecuteJobFromQueue                                                                          28  27,938180
             |- MySlowGame.exe!JobQueue::Exec                                                                                          6   6,000315
             |- MySlowGame.exe!CappedSemaphore::WaitForSignal                                                                          1   1,000889
             |- MySlowGame.exe!Event::WaitForSignal                                                                                    1   1,000889
```

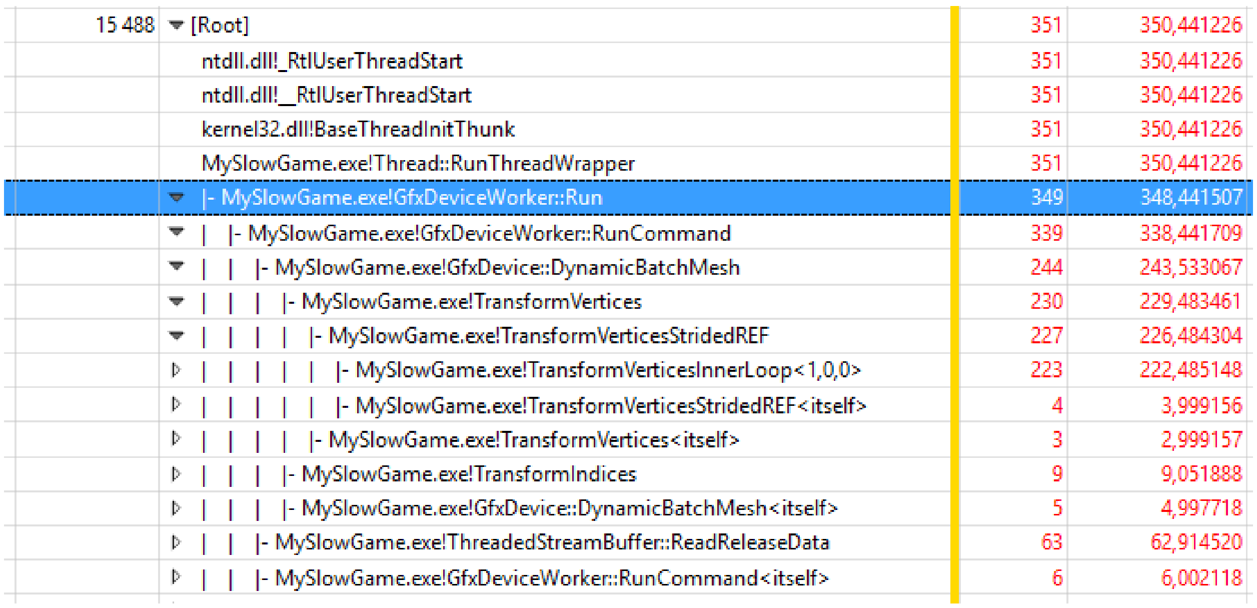
The number of JobQueue threads depends on the machine that the game is running on. This trace was captured on an i7 machine with 4 physical cores and hyperthreading (8 logical cores in total), so Unity decided to make 7 such threads. These worker threads do multithreaded work that can be moved from the main thread safely, such as culling, preparing to render into shadow maps, sorting objects for rendering, and physics calculations. In our case these threads have a relatively small sample count, so they’re not affecting the frame rate.

You can hide irrelevant threads by selecting them, right-clicking on them and pressing “filter out selection”:



e #	Process	Thread ID	Stack	Count Sum	Weight (in vie... Sum	TimeStamp (s)
1	MySlowGame.exe (26444)			1 552	1 551,308936	
2		20 768		968	968,081636	
3		15 488	[Root]	351	350,441226	
4		9 520	[Root]			
5		11 924	[Root]			
6		15 708	[Root]			
7		10 608	[Root]			
8		25 684	[Root]			
9		17 856	[Root]			
10		3 856	[Root]			
11		19 340	[Root]			
12		24 192	[Root]			
13		24 164	[Root]			

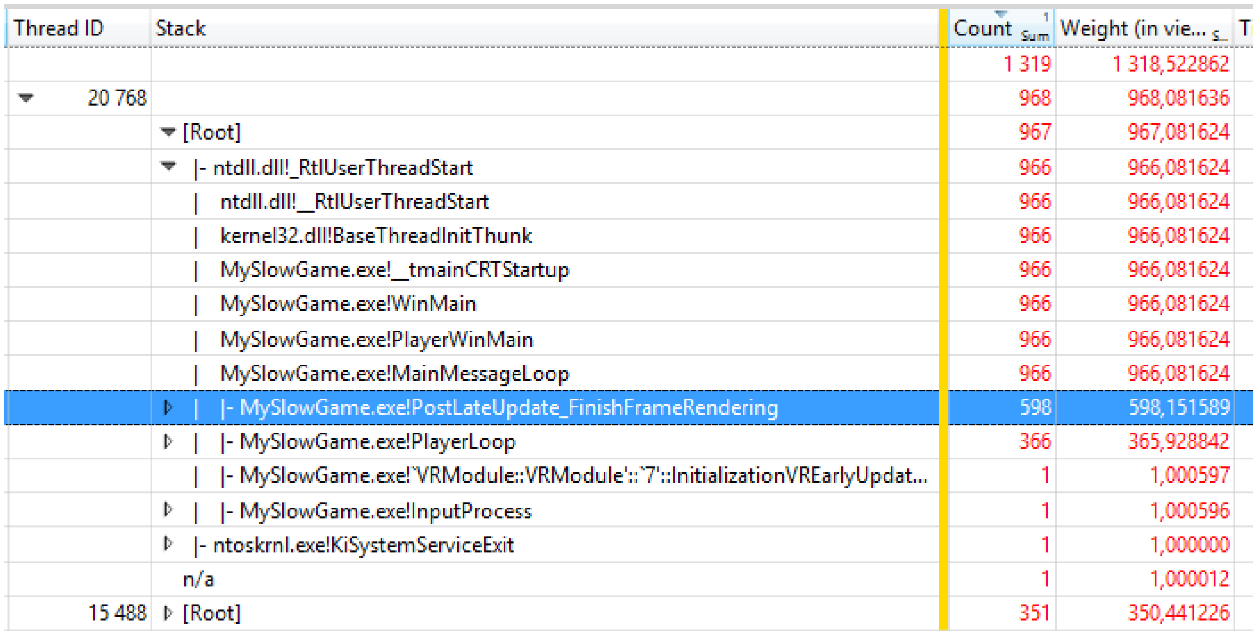
We are left with only two threads. Let’s take a look at the bottom one:



15 488	[Root]	351	350,441226
	ntdll.dll!_RtlUserThreadStart	351	350,441226
	ntdll.dll!__RtlUserThreadStart	351	350,441226
	kernel32.dll!BaseThreadInitThunk	351	350,441226
	MySlowGame.exe!Thread::RunThreadWrapper	351	350,441226
	|- MySlowGame.exe!GfxDeviceWorker::Run	349	348,441507
	| |- MySlowGame.exe!GfxDeviceWorker::RunCommand	339	338,441709
	| | |- MySlowGame.exe!GfxDevice::DynamicBatchMesh	244	243,533067
	| | | |- MySlowGame.exe!TransformVertices	230	229,483461
	| | | | |- MySlowGame.exe!TransformVerticesStridedREF	227	226,484304
	| | | | | |- MySlowGame.exe!TransformVerticesInnerLoop<1,0,0>	223	222,485148
	| | | | | |- MySlowGame.exe!TransformVerticesStridedREF<itself>	4	3,999156
	| | | | |- MySlowGame.exe!TransformVertices<itself>	3	2,999157
	| | | |- MySlowGame.exe!TransformIndices	9	9,051888
	| | | |- MySlowGame.exe!GfxDevice::DynamicBatchMesh<itself>	5	4,997718
	| | |- MySlowGame.exe!ThreadedStreamBuffer::ReadReleaseData	63	62,914520
	| | |- MySlowGame.exe!GfxDeviceWorker::RunCommand<itself>	6	6,002118

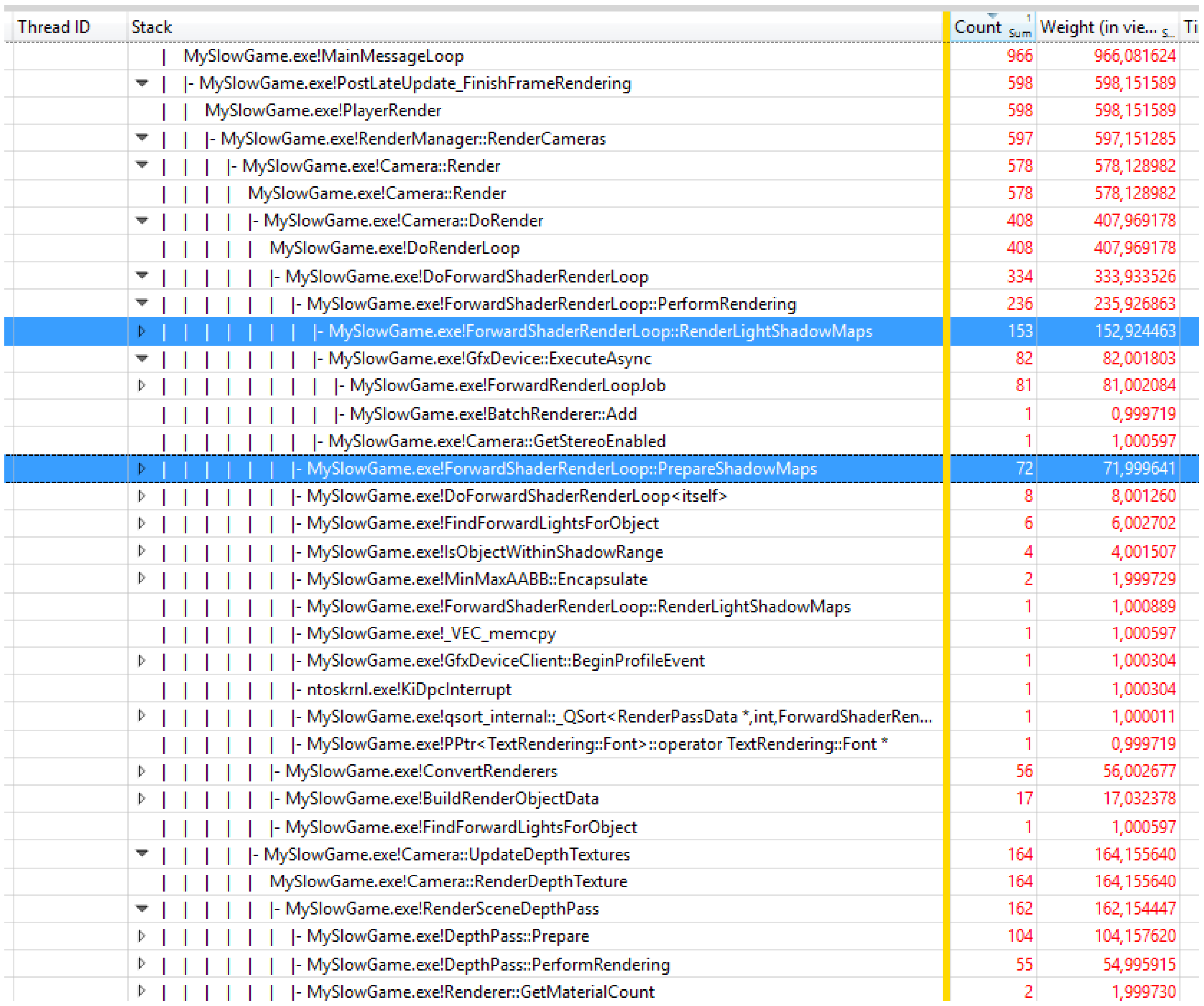
This is Unity’s rendering thread. You can recognize it because it starts with the “**GfxDeviceWorker::Run**” function. In this example it looks like it spends most of its time doing dynamic batching (transforming vertices of each object so that objects can be drawn together with fewer draw calls). This can be expensive in cases where there are many tiny dynamic objects that don’t get statically batched.

Let’s look at the last remaining thread. This is Unity’s main thread:



Thread ID	Stack	Count Sum	Weight (in vie... S	T
		1 319	1 318,522862	
▼ 20 768		968	968,081636	
	▼ [Root]	967	967,081624	
	▼ |- ntdll.dll!_RtlUserThreadStart	966	966,081624	
	| ntdll.dll!__RtlUserThreadStart	966	966,081624	
	| kernel32.dll!BaseThreadInitThunk	966	966,081624	
	| MySlowGame.exe!__tmainCRTStartup	966	966,081624	
	| MySlowGame.exe!WinMain	966	966,081624	
	| MySlowGame.exe!PlayerWinMain	966	966,081624	
	| MySlowGame.exe!MainMessageLoop	966	966,081624	
	▷ | |- MySlowGame.exe!PostLateUpdate_FinishFrameRendering	598	598,151589	
	▷ | |- MySlowGame.exe!PlayerLoop	366	365,928842	
	| |- MySlowGame.exe!`VRModule::VRModule'::`7'::InitializationVREarlyUpdat...	1	1,000597	
	▷ | |- MySlowGame.exe!InputProcess	1	1,000596	
	▷ |- ntoskrnl.exe!KiSystemServiceExit	1	1,000000	
	n/a	1	1,000012	
15 488	▷ [Root]	351	350,441226	

It seems that most of work here is divided between “**PostLateUpdate_FinishFrameRendering**” and “**PlayerLoop**”. Let’s take a look at the former one first.

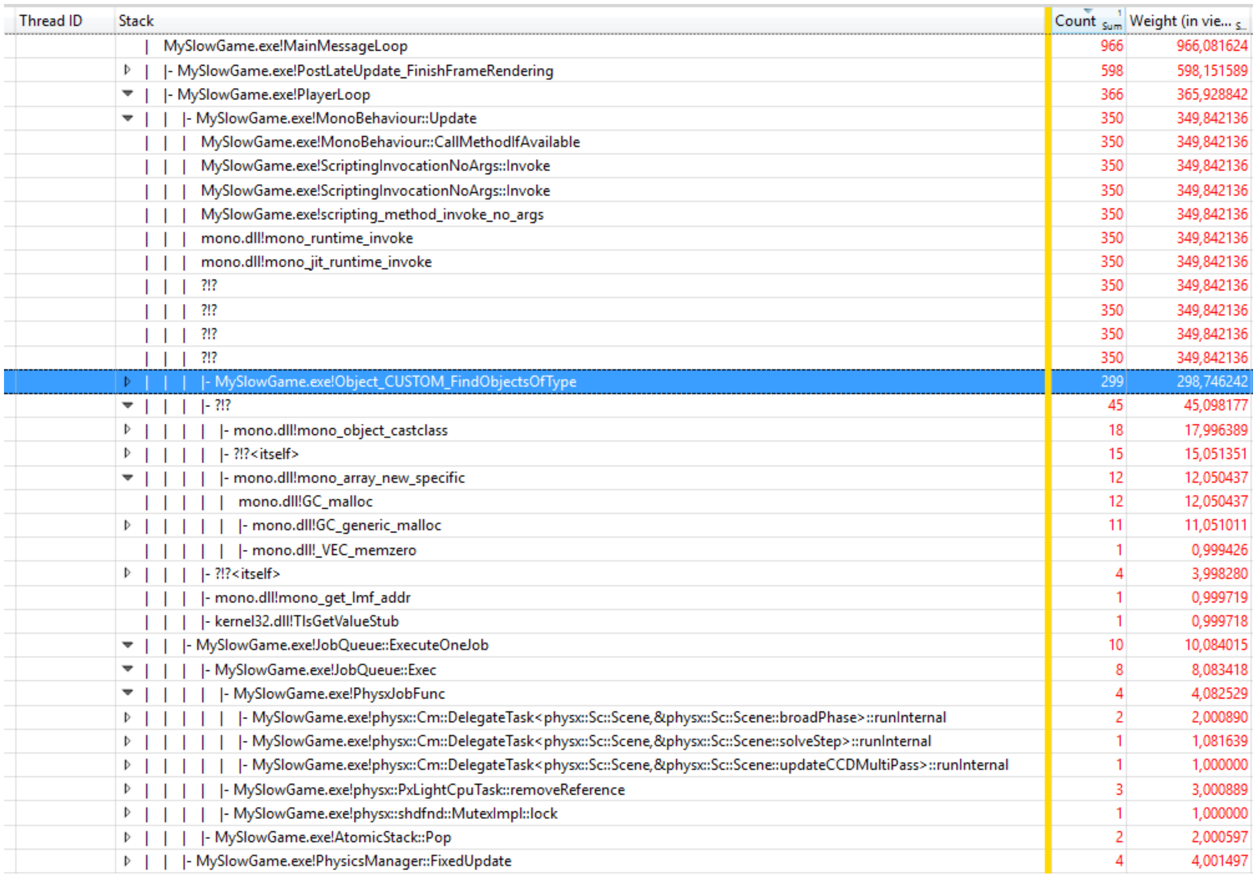


```
Thread ID  Stack                                                                                                  Count Sum   Weight (in vie... S..  Ti
              |  MySlowGame.exe!MainMessageLoop                                                                      966    966,081624
           ▼  | |- MySlowGame.exe!PostLateUpdate_FinishFrameRendering                                                598    598,151589
              | |  MySlowGame.exe!PlayerRender                                                                       598    598,151589
           ▼  | | |- MySlowGame.exe!RenderManager::RenderCameras                                                     597    597,151285
           ▼  | | | |- MySlowGame.exe!Camera::Render                                                               578    578,128982
              | | | |  MySlowGame.exe!Camera::Render                                                               578    578,128982
           ▼  | | | | |- MySlowGame.exe!Camera::DoRender                                                           408    407,969178
              | | | | |  MySlowGame.exe!DoRenderLoop                                                              408    407,969178
           ▼  | | | | | |- MySlowGame.exe!DoForwardShaderRenderLoop                                              334    333,933526
           ▼  | | | | | | |- MySlowGame.exe!ForwardShaderRenderLoop::PerformRendering                          236    235,926863
           ▷  | | | | | | | |- MySlowGame.exe!ForwardShaderRenderLoop::RenderLightShadowMaps                   153    152,924463
           ▼  | | | | | | | |- MySlowGame.exe!GfxDevice::ExecuteAsync                                          82     82,001803
           ▷  | | | | | | | | |- MySlowGame.exe!ForwardRenderLoopJob                                          81     81,002084
              | | | | | | | | |- MySlowGame.exe!BatchRenderer::Add                                            1      0,999719
              | | | | | | | |- MySlowGame.exe!Camera::GetStereoEnabled                                         1      1,000597
           ▷  | | | | | | |- MySlowGame.exe!ForwardShaderRenderLoop::PrepareShadowMaps                         72     71,999641
           ▷  | | | | | | |- MySlowGame.exe!DoForwardShaderRenderLoop<itself>                                 8      8,001260
           ▷  | | | | | | |- MySlowGame.exe!FindForwardLightsForObject                                       6      6,002702
           ▷  | | | | | | |- MySlowGame.exe!IsObjectWithinShadowRange                                       4      4,001507
           ▷  | | | | | | |- MySlowGame.exe!MinMaxAABB::Encapsulate                                          2      1,999729
              | | | | | | |- MySlowGame.exe!ForwardShaderRenderLoop::RenderLightShadowMaps                  1      1,000889
              | | | | | | |- MySlowGame.exe!_VEC_memcpy                                                     1      1,000597
           ▷  | | | | | | |- MySlowGame.exe!GfxDeviceClient::BeginProfileEvent                              1      1,000304
              | | | | | | |- ntoskrnl.exe!KiDpcInterrupt                                                   1      1,000304
           ▷  | | | | | | |- MySlowGame.exe!qsort_internal::_QSort<RenderPassData *,int,ForwardShaderRen...  1      1,000011
              | | | | | | |- MySlowGame.exe!PPtr<TextRendering::Font>::operator TextRendering::Font *      1      0,999719
           ▷  | | | | | |- MySlowGame.exe!ConvertRenderers                                                  56     56,002677
           ▷  | | | | | |- MySlowGame.exe!BuildRenderObjectData                                             17     17,032378
              | | | | | |- MySlowGame.exe!FindForwardLightsForObject                                        1      1,000597
           ▼  | | | | |- MySlowGame.exe!Camera::UpdateDepthTextures                                          164    164,155640
              | | | | |  MySlowGame.exe!Camera::RenderDepthTexture                                           164    164,155640
           ▼  | | | | | |- MySlowGame.exe!RenderSceneDepthPass                                              162    162,154447
           ▷  | | | | | | |- MySlowGame.exe!DepthPass::Prepare                                            104    104,157620
           ▷  | | | | | | |- MySlowGame.exe!DepthPass::PerformRendering                                   55     54,995915
           ▷  | | | | | | |- MySlowGame.exe!Renderer::GetMaterialCount                                    2      1,999730
```

Over a third of the whole rendering time in our game is spent rendering shadow maps - but if you look at the screenshot of the game, you can see that the shadows aren't even visible. What a waste! Disabling shadows in this case helps performance a lot, with no visual degradation whatsoever.

The rest of the time during rendering is spent running the forward render loop, building render queues, and preparing to render. The only way to reduce this cost is to reduce the number of meshes in the scene.

Now let's look at the "**PlayerLoop**" part of the main thread.



```
Thread ID  Stack                                                                                                   Count   Weight (in vie...
             |  MySlowGame.exe!MainMessageLoop                                                                       966   966,081624
         ▷   |  |- MySlowGame.exe!PostLateUpdate_FinishFrameRendering                                                598   598,151589
         ▼   |  |- MySlowGame.exe!PlayerLoop                                                                         366   365,928842
         ▼   |  |  |- MySlowGame.exe!MonoBehaviour::Update                                                           350   349,842136
             |  |  |  MySlowGame.exe!MonoBehaviour::CallMethodIfAvailable                                            350   349,842136
             |  |  |  MySlowGame.exe!ScriptingInvocationNoArgs::Invoke                                               350   349,842136
             |  |  |  MySlowGame.exe!ScriptingInvocationNoArgs::Invoke                                               350   349,842136
             |  |  |  MySlowGame.exe!scripting_method_invoke_no_args                                                 350   349,842136
             |  |  |  mono.dll!mono_runtime_invoke                                                                   350   349,842136
             |  |  |  mono.dll!mono_jit_runtime_invoke                                                               350   349,842136
             |  |  |  ?!?                                                                                            350   349,842136
             |  |  |  ?!?                                                                                            350   349,842136
             |  |  |  ?!?                                                                                            350   349,842136
             |  |  |  ?!?                                                                                            350   349,842136
         ▷   |  |  |  |- MySlowGame.exe!Object_CUSTOM_FindObjectsOfType                                              299   298,746242
         ▼   |  |  |  |- ?!?                                                                                          45    45,098177
         ▷   |  |  |  |  |- mono.dll!mono_object_castclass                                                            18    17,996389
         ▷   |  |  |  |  |- ?!?<itself>                                                                               15    15,051351
         ▼   |  |  |  |  |- mono.dll!mono_array_new_specific                                                          12    12,050437
             |  |  |  |  |  mono.dll!GC_malloc                                                                        12    12,050437
         ▷   |  |  |  |  |  |- mono.dll!GC_generic_malloc                                                             11    11,051011
             |  |  |  |  |  |- mono.dll!_VEC_memzero                                                                   1     0,999426
         ▷   |  |  |  |- ?!?<itself>                                                                                   4     3,998280
             |  |  |  |- mono.dll!mono_get_lmf_addr                                                                    1     0,999719
             |  |  |  |- kernel32.dll!TlsGetValueStub                                                                  1     0,999718
         ▼   |  |  |- MySlowGame.exe!JobQueue::ExecuteOneJob                                                          10    10,084015
         ▼   |  |  |  |- MySlowGame.exe!JobQueue::Exec                                                                 8     8,083418
         ▼   |  |  |  |  |- MySlowGame.exe!PhysxJobFunc                                                                4     4,082529
         ▷   |  |  |  |  |  |- MySlowGame.exe!physx::Cm::DelegateTask<physx::Sc::Scene,&physx::Sc::Scene::broadPhase>::runInternal          2     2,000890
         ▷   |  |  |  |  |  |- MySlowGame.exe!physx::Cm::DelegateTask<physx::Sc::Scene,&physx::Sc::Scene::solveStep>::runInternal           1     1,081639
         ▷   |  |  |  |  |  |- MySlowGame.exe!physx::Cm::DelegateTask<physx::Sc::Scene,&physx::Sc::Scene::updateCCDMultiPass>::runInternal  1     1,000000
         ▷   |  |  |  |  |- MySlowGame.exe!physx::PxLightCpuTask::removeReference                                       3     3,000889
         ▷   |  |  |  |  |- MySlowGame.exe!physx::shdfnd::MutexImpl::lock                                               1     1,000000
         ▷   |  |  |  |- MySlowGame.exe!AtomicStack::Pop                                                                2     2,000597
         ▷   |  |  |- MySlowGame.exe!PhysicsManager::FixedUpdate                                                       4     4,001497
```

The first items to address here are the mysterious “**?!?**” stack frames. These frames are Mono JIT-ed code, which Windows Performance Analyzer cannot decode. Therefore, managed stack frames cannot be shown when using Mono scripting backend. With .NET scripting backend, they can be decoded as long as the trace was recorded with .NET ETW provider enabled, while with IL2CPP scripting backend they can be decoded as long as there is a matching PDB file.

It looks like most of the “**PlayerLoop**” time is taken by a MonoBehaviour update. It calls the “**Object.FindObjectsOfType()**” function multiple times, which is very resource-intensive as evident in the sample count attributed to it. There’s also some managed code taking a little time, and finally some physics calculations. The main course of action in this case is to eliminate these “**FindObjectsOfType()**” calls every frame, and perhaps cache the results in the “**Start()**” function.

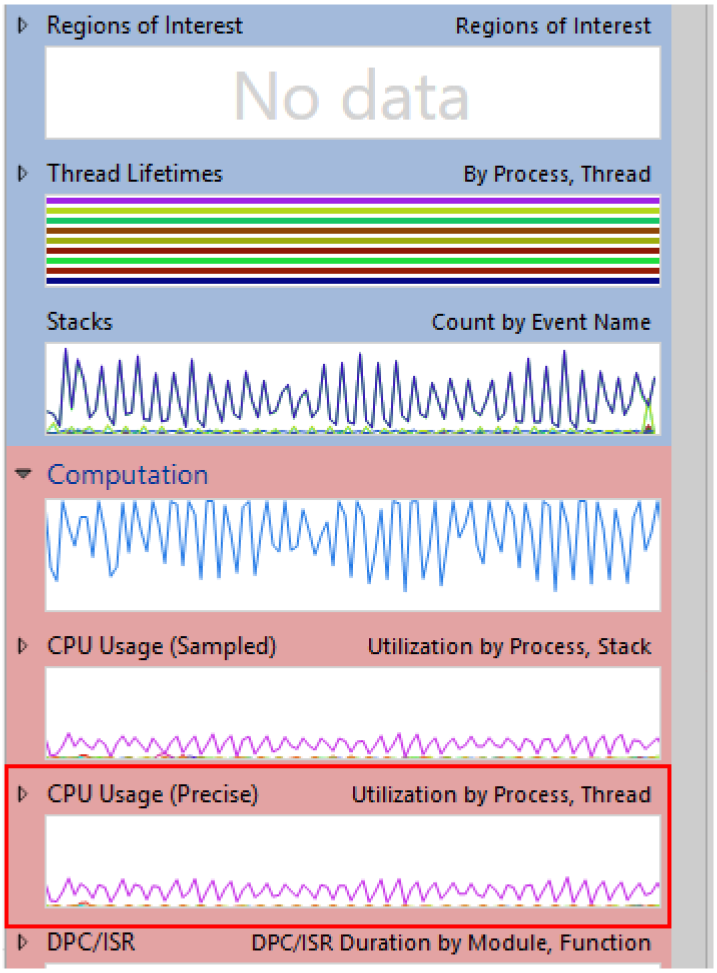
CPU Usage (Precise) provider

The CPU Usage (Precise) provider shows the precise CPU usage of all threads running in the system, by logging every single context switch that the operating system executes. A context switch is the switching execution on the CPU from one thread to another.

Unlike the CPU Usage (Sampled) provider, CPU Usage (Precise) only indicates which threads are executing at a particular point in time, not what those threads are doing. The only stack trace it gives is the stack trace from where the thread was when it started its execution during the context switch. This provider is usually used for wait analysis, to investigate why a thread isn't running.

The CPU Usage (Precise) provider shows very different data compared to the CPU Usage (Sampled) provider. The CPU Usage (Precise) provider only logs OS context switches, so these events make up the rows of the Analysis tab for this provider.

To bring the CPU Usage (Sampled) event provider into the Analysis tab, double-click on it in the Graph Explorer:



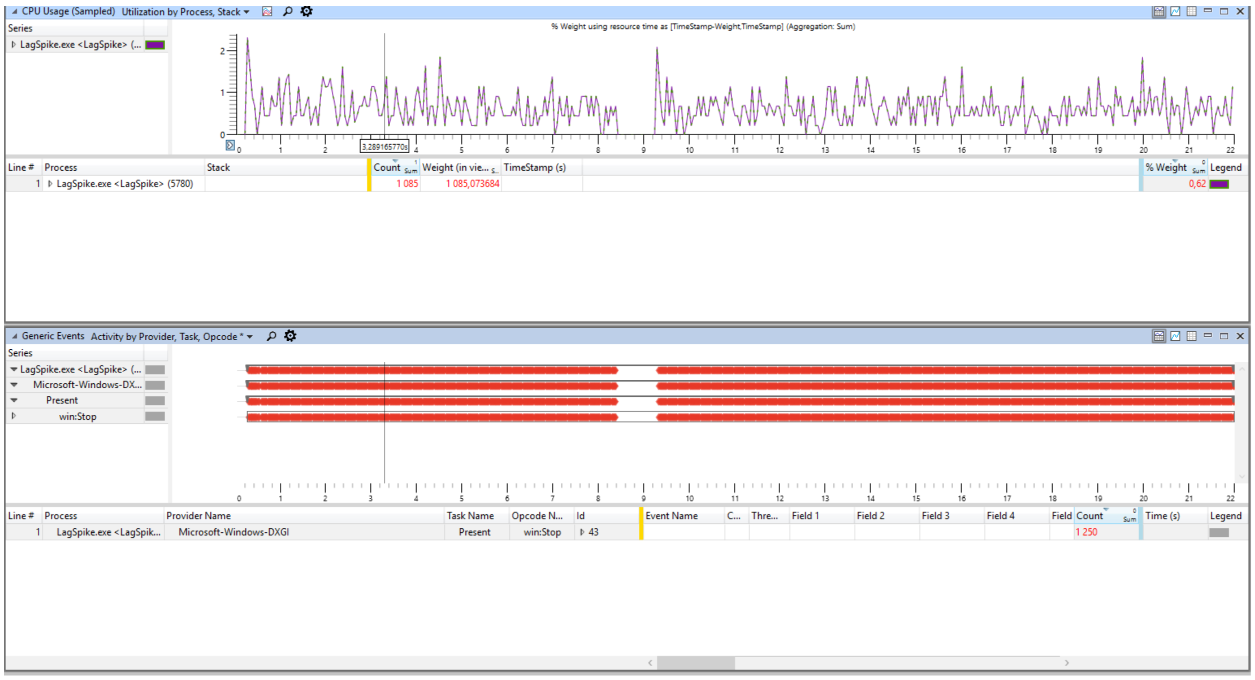
These are the most significant columns in the Analysis tab:

12. **New Process**: The process that owns the new thread.
13. **New Thread ID**: The thread to which the context was switched.
14. **New Thread Stack**: The stack trace of the new thread when it was switched in (note: this matches the stack for when it was last switched out).
15. **Readying Process**: The process that owns the readying thread.
16. **Readying Thread ID**: The thread that caused the new thread to wake up. This is equal to -1 in cases where the new thread wasn't waiting for anything, and was swapped out because its quantum had run out.
17. **Count**: Total context switch count for that row.
18. **Ready**: The moment in time when the new thread became ready to be switched in.

19. **Waits**: The amount of time the new thread waited before it became ready to be switched in.
20. **Switch-In Time**: The moment in time when the new thread was switched in.

CPU Usage (Precise) provider: Example walkthrough

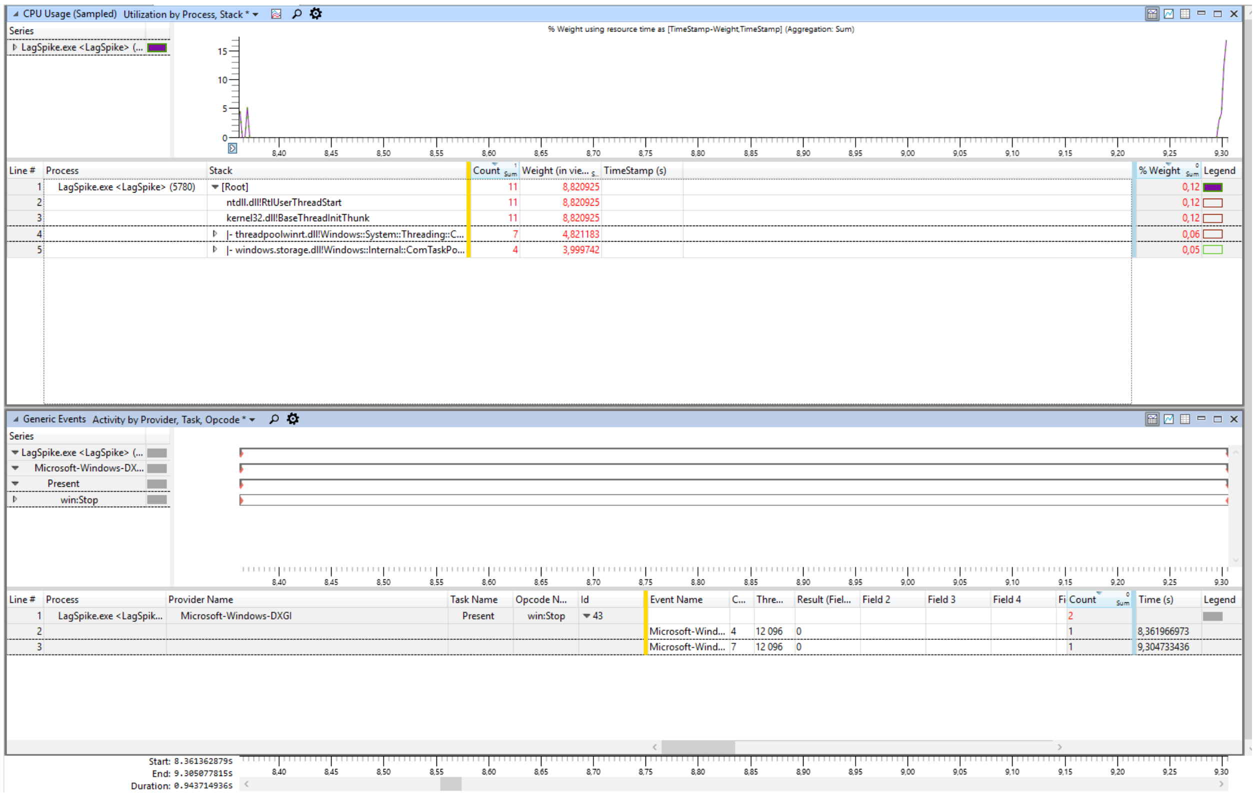
We have a game that runs nicely most of the time, but sometimes a frame rate spike occurs:


CPU Usage (Sampled) Utilization by Process, Stack
Series
LagSpike.exe <LagSpike> (...
% Weight using resource time as [TimeStamp-Weight.TimeStamp] (Aggregation: Sum)
3,289165770s
Generic Events Activity by Provider, Task, Opcode *
Series
LagSpike.exe <LagSpike> (...
Microsoft-Windows-DX...
Present
win:Stop

Line #	Process	Stack	Count	Weight (in vie...	TimeStamp (s)	% Weight	Legend
1	LagSpike.exe <LagSpike> (5780)		1 085	1 085,073684		0,62	🟪

Line #	Process	Provider Name	Task Name	Opcode N...	Id	Event Name	C...	Thre...	Field 1	Field 2	Field 3	Field 4	Field	Count	Time (s)	Legend
1	LagSpike.exe <LagSpik...	Microsoft-Windows-DXGI	Present	win:Stop	43									1 250		▨

Upon zooming into the selected region, we can see that the **CPU Usage (Sampled) provider** had no data of what was happening during the spike (only 11 samples were captured), even though the spike took a massive 942.766 ms:

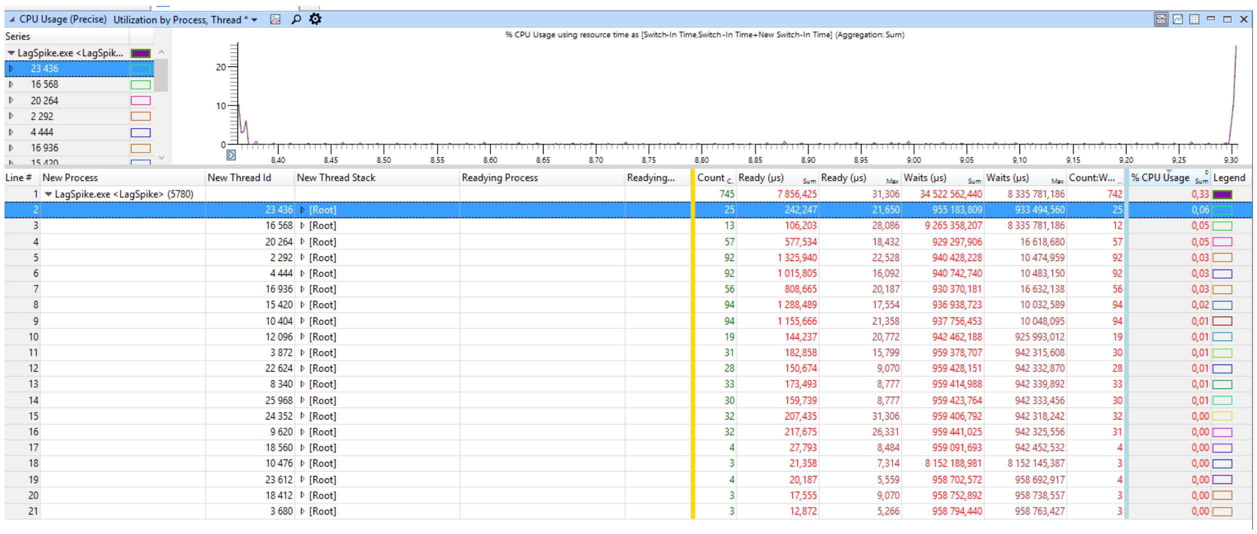

CPU Usage (Sampled) Utilization by Process, Stack *
% Weight using resource time as [TimeStamp-Weight.TimeStamp] (Aggregation: Sum)
Series
LagSpike.exe <LagSpike> (...
Generic Events Activity by Provider, Task, Opcode *
Series
LagSpike.exe <LagSpike> (...
Microsoft-Windows-DX...
Present
win:Stop

Line #	Process	Stack	Count (Sum)	Weight (in vie... (Sum)	TimeStamp (s)	% Weight (Sum)	Legend
1	LagSpike.exe <LagSpike> (5780)	[Root]	11	8,820925		0,12	🟪
2		ntdll.dll!RtlUserThreadStart	11	8,820925		0,12	▭
3		kernel32.dll!BaseThreadInitThunk	11	8,820925		0,12	▭
4		|- threadpoolwinrt.dll!Windows::System::Threading::C...	7	4,821183		0,06	▭
5		|- windows.storage.dll!Windows::Internal::ComTaskPo...	4	3,999742		0,05	▭

Line #	Process	Provider Name	Task Name	Opcode N...	Id	Event Name	C...	Thre...	Result (Fiel...	Field 2	Field 3	Field 4	Fi	Count (Sum)	Time (s)	Legend
1	LagSpike.exe <LagSpik...	Microsoft-Windows-DXGI	Present	win:Stop	43									2		▨
2						Microsoft-Wind...	4	12 096	0					1	8,361966973	
3						Microsoft-Wind...	7	12 096	0					1	9,304733436	

```
   Start: 8.361362879s
     End: 9.305077815s
Duration: 0.943714936s
```

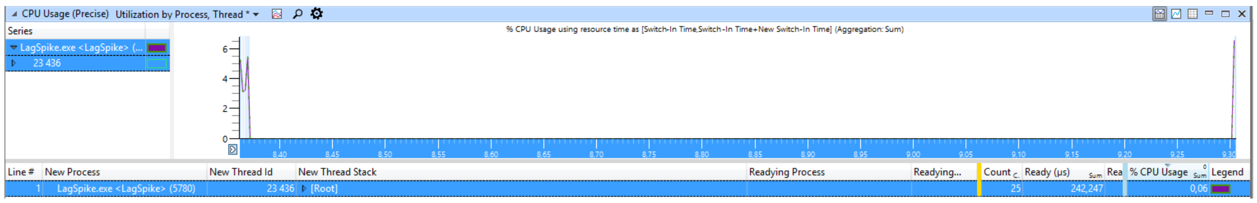
This usually indicates that the game was not actually executing. Let’s look at the **CPU Usage (Precise) provider** data to confirm our theory:


CPU Usage (Precise) Utilization by Process, Thread *
% CPU Usage using resource time as [Switch-In Time,Switch-In Time+New Switch-In Time] (Aggregation: Sum)
Series
LagSpike.exe <LagSpik...
23 436
16 568
20 264
2 292
4 444
16 936
15 420

Line #	New Process	New Thread Id	New Thread Stack	Readying Process	Readying...	Count	Ready (µs) Sum	Ready (µs) Max	Waits (µs) Sum	Waits (µs) Max	Count:W...	% CPU Usage Sum	Legend
1	LagSpike.exe <LagSpike> (5780)					745	7 856,425	31,306	34 522 562,440	8 335 781,186	742	0,33	🟪
2		23 436	[Root]			25	242,247	21,650	955 183,809	933 494,560	25	0,06	▭
3		16 568	[Root]			13	106,203	28,086	9 265 358,207	8 335 781,186	12	0,05	▭
4		20 264	[Root]			57	577,534	18,432	929 297,906	16 618,680	57	0,05	▭
5		2 292	[Root]			92	1 325,940	22,528	940 428,228	10 474,959	92	0,03	▭
6		4 444	[Root]			92	1 015,805	16,092	940 742,740	10 483,150	92	0,03	▭
7		16 936	[Root]			56	808,665	20,187	930 370,181	16 632,138	56	0,03	▭
8		15 420	[Root]			94	1 288,489	17,554	936 938,723	10 032,589	94	0,02	▭
9		10 404	[Root]			94	1 155,666	21,358	937 756,453	10 048,095	94	0,01	▭
10		12 096	[Root]			19	144,237	20,772	942 462,188	925 993,012	19	0,01	▭
11		3 872	[Root]			31	182,858	15,799	959 378,707	942 315,608	30	0,01	▭
12		22 624	[Root]			28	150,674	9,070	959 428,151	942 332,870	28	0,01	▭
13		8 340	[Root]			33	173,493	8,777	959 414,988	942 339,892	33	0,01	▭
14		25 968	[Root]			30	159,739	8,777	959 423,764	942 333,456	30	0,01	▭
15		24 352	[Root]			32	207,435	31,306	959 406,792	942 318,242	32	0,00	▭
16		9 620	[Root]			32	217,675	26,331	959 441,025	942 325,556	31	0,00	▭
17		18 560	[Root]			4	27,793	8,484	959 091,693	942 452,532	4	0,00	▭
18		10 476	[Root]			3	21,358	7,314	8 152 188,981	8 152 145,387	3	0,00	▭
19		23 612	[Root]			4	20,187	5,559	958 702,572	958 692,917	4	0,00	▭
20		18 412	[Root]			3	17,555	9,070	958 752,892	958 738,557	3	0,00	▭
21		3 680	[Root]			3	12,872	5,266	958 794,440	958 763,427	3	0,00	▭

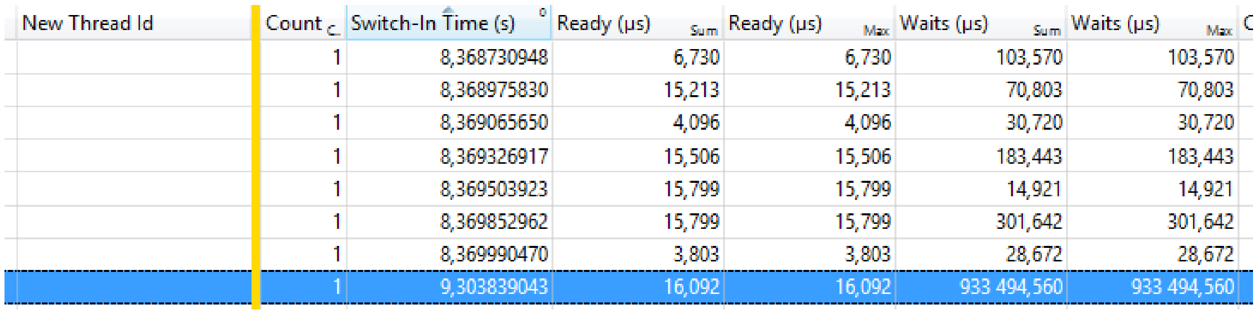
CPU Usage (Precise) provider shows what we already knew: the CPU was not used a lot by our process during that spike. However, it still detected that our process used the CPU for brief moments of time - it had its thread switched in a total of 745 times. Even though there are so many active threads, most of them will not be very relevant to our investigation: there will be Unity’s JobQueue threads, Enlighten global illumination threads, OS thread pool threads, and others that we can't affect. We're really only interested in figuring out why Unity's main thread wasn't doing anything: after all, that's what affects our framerate.

Our main thread in this case is thread **23436**, which has a total of 25 context switch-ins recorded between the two frames. If we filter to only this thread, we can see that it was doing nothing (or waiting for something) almost the whole time:


CPU Usage (Precise) Utilization by Process, Thread *
Series
LagSpike.exe <LagSpike> (...
23 436
% CPU Usage using resource time as [Switch-In Time,Switch-In Time+New Switch-In Time] (Aggregation: Sum)
6
4
2
0
8,40
8,45
8,50
8,55
8,60
8,65
8,70
8,75
8,80
8,85
8,90
8,95
9,00
9,05
9,10
9,15
9,20
9,25
9,30

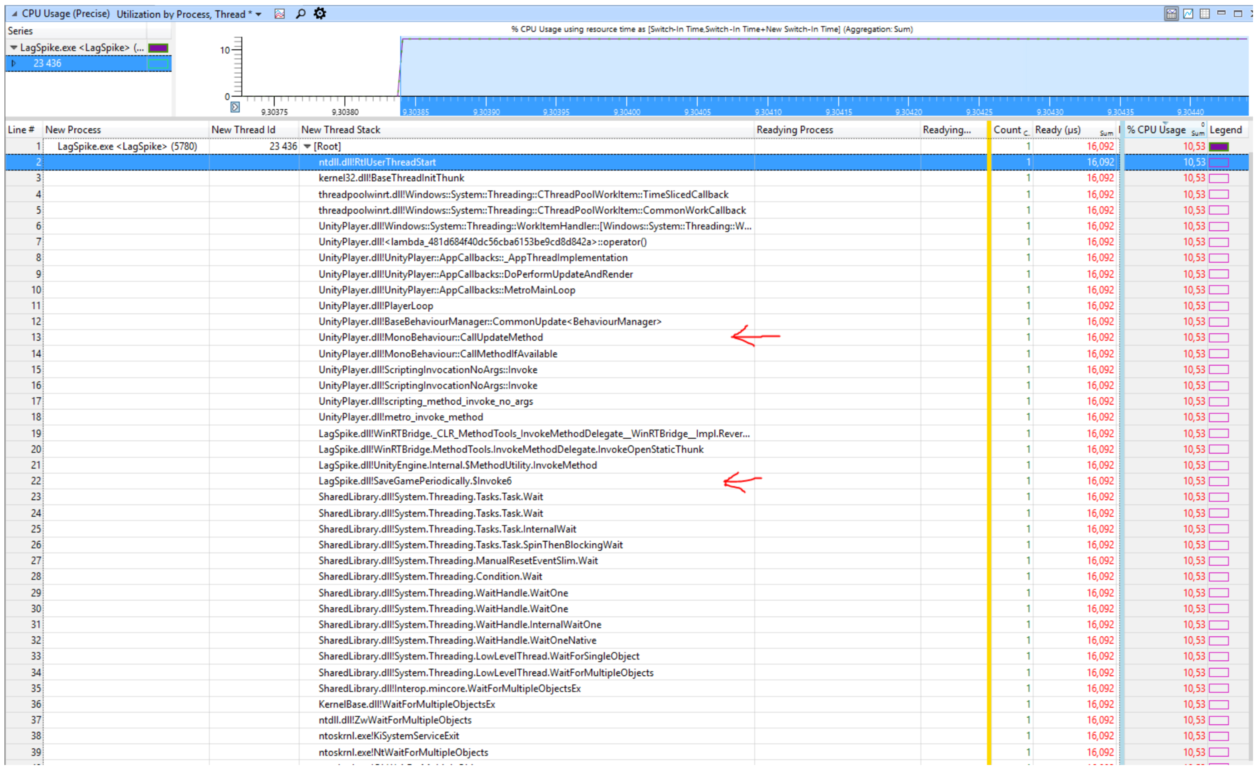
Line #	New Process	New Thread Id	New Thread Stack	Readying Process	Readying...	Count	Ready (µs) Sum	Rea	% CPU Usage Sum	Legend
1	LagSpike.exe <LagSpike> (5780)	23 436	[Root]			25	242,247		0,06	🟪

To understand what the thread was doing, we need to take a look at its stack during the first switch-in after the spike. To do this, sort the context switches by the **Switch-In Time (s)** column, so we can see which context switch was the first one after the spike:



New Thread Id	Count	Switch-In Time (s)	Ready (µs) Sum	Ready (µs) Max	Waits (µs) Sum	Waits (µs) Max	C
	1	8,368730948	6,730	6,730	103,570	103,570	
	1	8,368975830	15,213	15,213	70,803	70,803	
	1	8,369065650	4,096	4,096	30,720	30,720	
	1	8,369326917	15,506	15,506	183,443	183,443	
	1	8,369503923	15,799	15,799	14,921	14,921	
	1	8,369852962	15,799	15,799	301,642	301,642	
	1	8,369990470	3,803	3,803	28,672	28,672	
	1	9,303839043	16,092	16,092	933 494,560	933 494,560	

We can clearly see which context switch happened just after the spike, and after filtering our view to it, we can see where our thread was switched in:



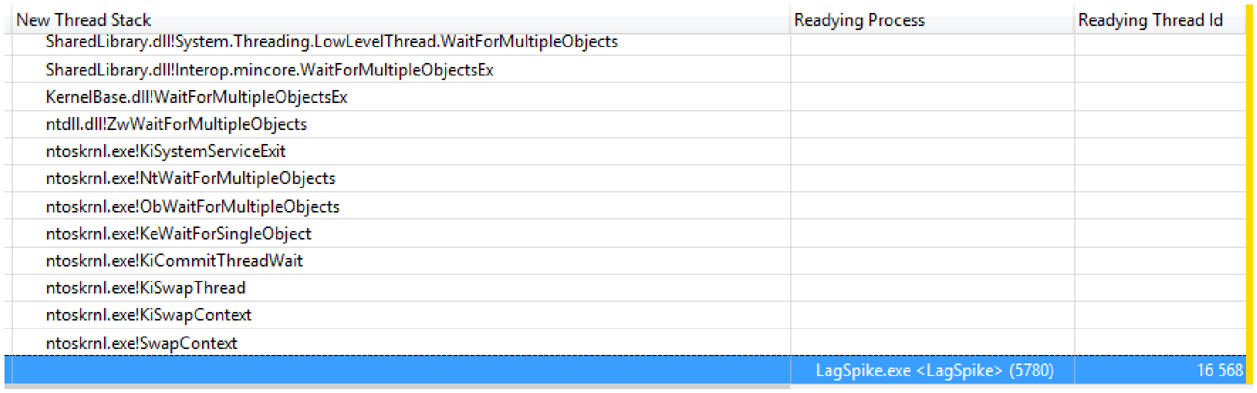
CPU Usage (Precise) Utilization by Process, Thread *

% CPU Usage using resource time as [Switch-In Time,Switch-In Time+New Switch-In Time] (Aggregation: Sum)

Line #	New Process	New Thread Id	New Thread Stack	Readying Process	Readying...	Count	Ready (µs) Sum	% CPU Usage Sum	Legend
1	LagSpike.exe <LagSpike> (5780)	23 436	[Root]			1	16,092	10,53	■
2			ntdll.dll!RtlUserThreadStart			1	16,092	10,53	□
3			kernel32.dll!BaseThreadInitThunk			1	16,092	10,53	□
4			threadpoolwinrt.dll!Windows::System::Threading::CThreadPoolWorkItem::TimeSlicedCallback			1	16,092	10,53	□
5			threadpoolwinrt.dll!Windows::System::Threading::CThreadPoolWorkItem::CommonWorkCallback			1	16,092	10,53	□
6			UnityPlayer.dll!Windows::System::Threading::WorkItemHandler::[Windows::System::Threading::W...			1	16,092	10,53	□
7			UnityPlayer.dll!<lambda_481d684f40dc56cba6153be9cd8d842a>::operator()			1	16,092	10,53	□
8			UnityPlayer.dll!UnityPlayer::AppCallbacks::_AppThreadImplementation			1	16,092	10,53	□
9			UnityPlayer.dll!UnityPlayer::AppCallbacks::DoPerformUpdateAndRender			1	16,092	10,53	□
10			UnityPlayer.dll!UnityPlayer::AppCallbacks::MetroMainLoop			1	16,092	10,53	□
11			UnityPlayer.dll!PlayerLoop			1	16,092	10,53	□
12			UnityPlayer.dll!BaseBehaviourManager::CommonUpdate<BehaviourManager>			1	16,092	10,53	□
13			UnityPlayer.dll!MonoBehaviour::CallUpdateMethod ←			1	16,092	10,53	□
14			UnityPlayer.dll!MonoBehaviour::CallMethodIfAvailable			1	16,092	10,53	□
15			UnityPlayer.dll!ScriptingInvocationNoArgs::Invoke			1	16,092	10,53	□
16			UnityPlayer.dll!ScriptingInvocationNoArgs::Invoke			1	16,092	10,53	□
17			UnityPlayer.dll!scripting_method_invoke_no_args			1	16,092	10,53	□
18			UnityPlayer.dll!metro_invoke_method			1	16,092	10,53	□
19			LagSpike.dll!WinRTBridge._CLR_MethodTools_InvokeMethodDelegate__WinRTBridge__Impl.Rever...			1	16,092	10,53	□
20			LagSpike.dll!WinRTBridge.MethodTools.InvokeMethodDelegate.InvokeOpenStaticThunk			1	16,092	10,53	□
21			LagSpike.dll!UnityEngine.Internal.$MethodUtility.InvokeMethod			1	16,092	10,53	□
22			LagSpike.dll!SaveGamePeriodically.$Invoke6 ←			1	16,092	10,53	□
23			SharedLibrary.dll!System.Threading.Tasks.Task.Wait			1	16,092	10,53	□
24			SharedLibrary.dll!System.Threading.Tasks.Task.Wait			1	16,092	10,53	□
25			SharedLibrary.dll!System.Threading.Tasks.Task.InternalWait			1	16,092	10,53	□
26			SharedLibrary.dll!System.Threading.Tasks.Task.SpinThenBlockingWait			1	16,092	10,53	□
27			SharedLibrary.dll!System.Threading.ManualResetEventSlim.Wait			1	16,092	10,53	□
28			SharedLibrary.dll!System.Threading.Condition.Wait			1	16,092	10,53	□
29			SharedLibrary.dll!System.Threading.WaitHandle.WaitOne			1	16,092	10,53	□
30			SharedLibrary.dll!System.Threading.WaitHandle.WaitOne			1	16,092	10,53	□
31			SharedLibrary.dll!System.Threading.WaitHandle.InternalWaitOne			1	16,092	10,53	□
32			SharedLibrary.dll!System.Threading.WaitHandle.WaitOneNative			1	16,092	10,53	□
33			SharedLibrary.dll!System.Threading.LowLevelThread.WaitForSingleObject			1	16,092	10,53	□
34			SharedLibrary.dll!System.Threading.LowLevelThread.WaitForMultipleObjects			1	16,092	10,53	□
35			SharedLibrary.dll!Interop.mincore.WaitForMultipleObjectsEx			1	16,092	10,53	□
36			KernelBase.dll!WaitForMultipleObjectsEx			1	16,092	10,53	□
37			ntdll.dll!ZwWaitForMultipleObjects			1	16,092	10,53	□
38			ntoskrnl.exe!KiSystemServiceExit			1	16,092	10,53	□
39			ntoskrnl.exe!NtWaitForMultipleObjects			1	16,092	10,53	□

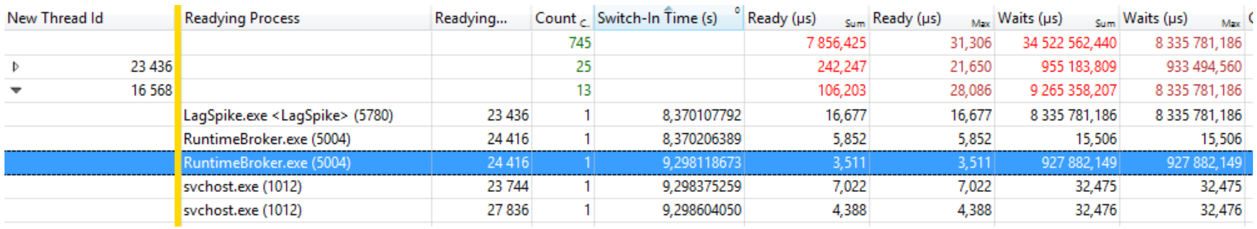
So, it seems that the culprit is a script named "**SaveGamePeriodically**". It seems to be doing something expensive in its **Update()** function (note: we cannot see "**Update()**" function in the callstack because the JIT most likely inlined it, but we can tell the script name from its invoker method "**SaveGamePeriodically.$Invoke6**", and we can tell that it is an update method because higher in the call stack we can see the Unity's function "**CallUpdateMethod**").

We identified where the time is spent waiting, but we still don't know what it is waiting **for**. To do that, we can check the "**Readying Process**" and "**Readying Thread Id**" columns, which tell us which thread was responsible for bring our thread out of the wait:



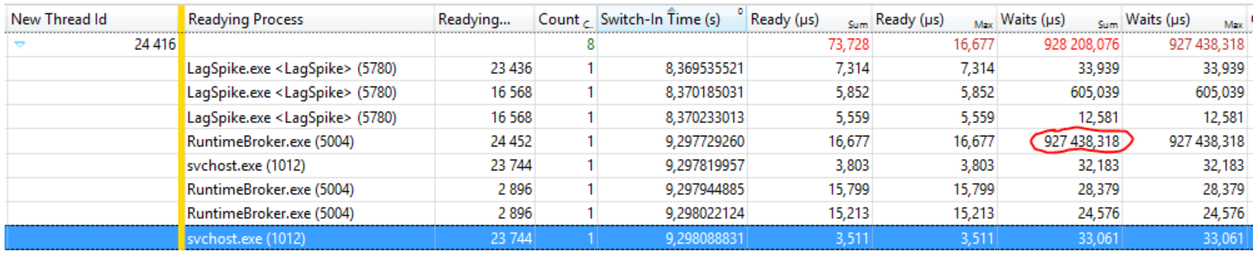
New Thread Stack	Readying Process	Readying Thread Id
SharedLibrary.dll!System.Threading.LowLevelThread.WaitForMultipleObjects		
SharedLibrary.dll!Interop.mincore.WaitForMultipleObjectsEx		
KernelBase.dll!WaitForMultipleObjectsEx		
ntdll.dll!ZwWaitForMultipleObjects		
ntoskrnl.exe!KiSystemServiceExit		
ntoskrnl.exe!NtWaitForMultipleObjects		
ntoskrnl.exe!ObWaitForMultipleObjects		
ntoskrnl.exe!KeWaitForSingleObject		
ntoskrnl.exe!KiCommitThreadWait		
ntoskrnl.exe!KiSwapThread		
ntoskrnl.exe!KiSwapContext		
ntoskrnl.exe!SwapContext		
	LagSpike.exe <LagSpike> (5780)	16 568

Our thread was waiting for thread **16568**. But what was thread **16568** doing?



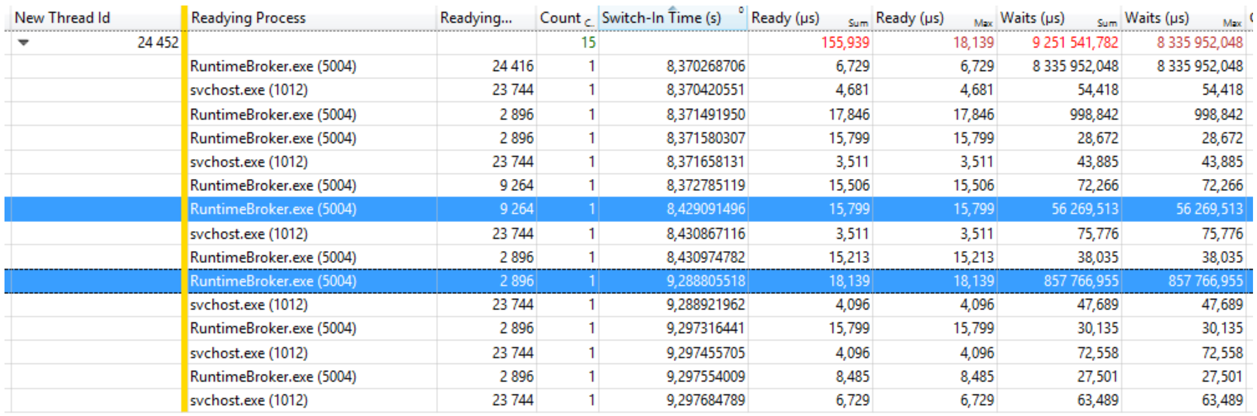
New Thread Id	Readying Process	Readying...	Count	Switch-In Time (s)	Ready (µs) Sum	Ready (µs) Max	Waits (µs) Sum	Waits (µs) Max
			745		7 856,425	31,306	34 522 562,440	8 335 781,186
23 436			25		242,247	21,650	955 183,809	933 494,560
16 568			13		106,203	28,086	9 265 358,207	8 335 781,186
	LagSpike.exe <LagSpike> (5780)	23 436	1	8,370107792	16,677	16,677	8 335 781,186	8 335 781,186
	RuntimeBroker.exe (5004)	24 416	1	8,370206389	5,852	5,852	15,506	15,506
	RuntimeBroker.exe (5004)	24 416	1	9,298118673	3,511	3,511	927 882,149	927 882,149
	svchost.exe (1012)	23 744	1	9,298375259	7,022	7,022	32,475	32,475
	svchost.exe (1012)	27 836	1	9,298604050	4,388	4,388	32,476	32,476

It turns out that thread **16568** was waiting for yet another thread for over 927 ms, and that other thread is from another process! Let's look at the CPU usage of thread **24416**:



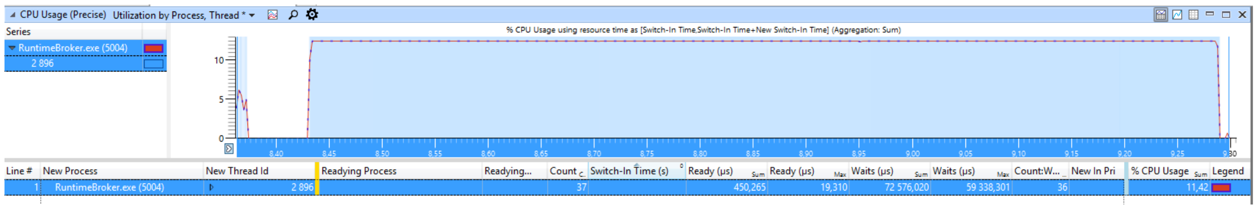
New Thread Id	Readying Process	Readying...	Count	Switch-In Time (s)	Ready (µs) Sum	Ready (µs) Max	Waits (µs) Sum	Waits (µs) Max
24 416			8		73,728	16,677	928 208,076	927 438,318
	LagSpike.exe <LagSpike> (5780)	23 436	1	8,369535521	7,314	7,314	33,939	33,939
	LagSpike.exe <LagSpike> (5780)	16 568	1	8,370185031	5,852	5,852	605,039	605,039
	LagSpike.exe <LagSpike> (5780)	16 568	1	8,370233013	5,559	5,559	12,581	12,581
	RuntimeBroker.exe (5004)	24 452	1	9,297729260	16,677	16,677	927 438,318	927 438,318
	svchost.exe (1012)	23 744	1	9,297819957	3,803	3,803	32,183	32,183
	RuntimeBroker.exe (5004)	2 896	1	9,297944885	15,799	15,799	28,379	28,379
	RuntimeBroker.exe (5004)	2 896	1	9,298022124	15,213	15,213	24,576	24,576
	svchost.exe (1012)	23 744	1	9,298088831	3,511	3,511	33,061	33,061

Apparently, thread **24416** was waiting for 927 ms for thread **24452**! Let's keep following the chain:



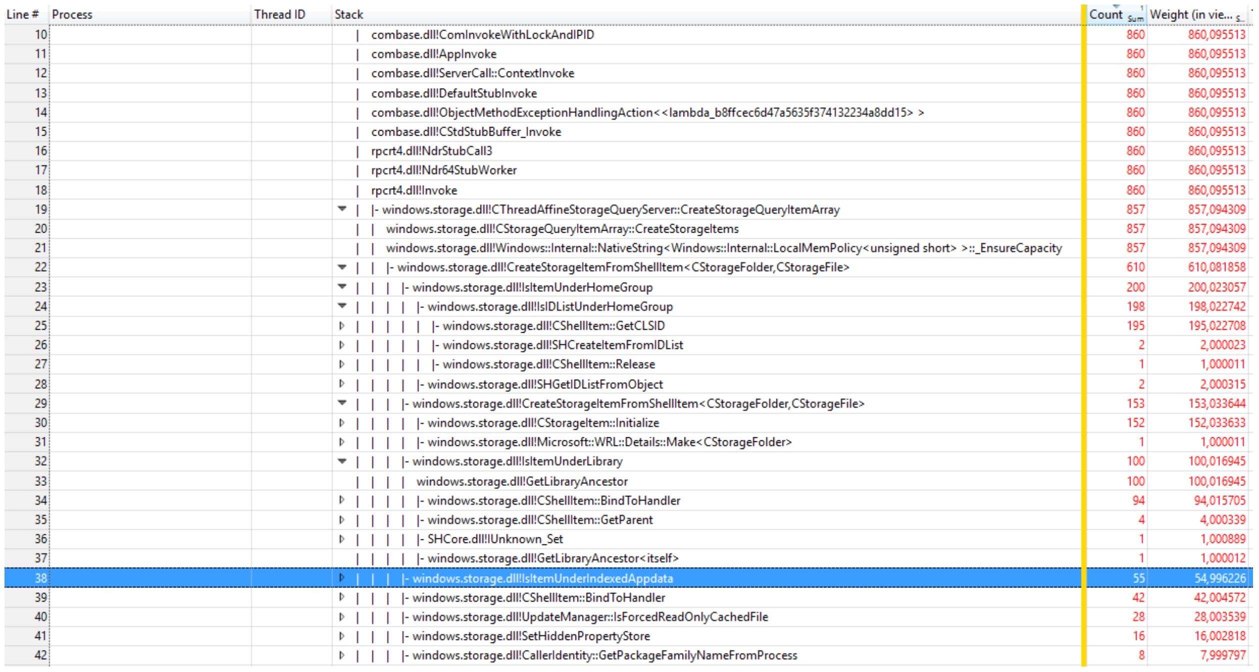
New Thread Id	Readying Process	Readying...	Count	Switch-In Time (s)	Ready (µs) Sum	Ready (µs) Max	Waits (µs) Sum	Waits (µs) Max
24 452			15		155,939	18,139	9 251 541,782	8 335 952,048
	RuntimeBroker.exe (5004)	24 416	1	8,370268706	6,729	6,729	8 335 952,048	8 335 952,048
	svchost.exe (1012)	23 744	1	8,370420551	4,681	4,681	54,418	54,418
	RuntimeBroker.exe (5004)	2 896	1	8,371491950	17,846	17,846	998,842	998,842
	RuntimeBroker.exe (5004)	2 896	1	8,371580307	15,799	15,799	28,672	28,672
	svchost.exe (1012)	23 744	1	8,371658131	3,511	3,511	43,885	43,885
	RuntimeBroker.exe (5004)	9 264	1	8,372785119	15,506	15,506	72,266	72,266
	RuntimeBroker.exe (5004)	9 264	1	8,429091496	15,799	15,799	56 269,513	56 269,513
	svchost.exe (1012)	23 744	1	8,430867116	3,511	3,511	75,776	75,776
	RuntimeBroker.exe (5004)	2 896	1	8,430974782	15,213	15,213	38,035	38,035
	RuntimeBroker.exe (5004)	2 896	1	9,288805518	18,139	18,139	857 766,955	857 766,955
	svchost.exe (1012)	23 744	1	9,288921962	4,096	4,096	47,689	47,689
	RuntimeBroker.exe (5004)	2 896	1	9,297316441	15,799	15,799	30,135	30,135
	svchost.exe (1012)	23 744	1	9,297455705	4,096	4,096	72,558	72,558
	RuntimeBroker.exe (5004)	2 896	1	9,297554009	8,485	8,485	27,501	27,501
	svchost.exe (1012)	23 744	1	9,297684789	6,729	6,729	63,489	63,489

This thread has actually been waiting twice: for thread **9264** for 56 ms, and for thread **2896** for 857 ms. Let's ignore the shorter wait and focus on the longer one. So, what was thread **2896** doing?


CPU Usage (Precise) Utilization by Process, Thread *
Series
RuntimeBroker.exe (5004)
2 896
% CPU Usage using resource time as [Switch-In Time,Switch-In Time+New Switch-In Time] (Aggregation: Sum)
10
5
0
8.40
8.45
8.50
8.55
8.60
8.65
8.70
8.75
8.80
8.85
8.90
8.95
9.00
9.05
9.10
9.15
9.20
9.25
9.30

Line #	New Process	New Thread Id	Readying Process	Readying...	Count	Switch-In Time (s)	Ready (µs) Sum	Ready (µs) Max	Waits (µs) Sum	Waits (µs) Max	Count:W...	New In Pri	% CPU Usage Sum	Legend
1	RuntimeBroker.exe (5004)	2 896			37		450,265	19,310	72 576,020	59 338,301	36		11,42	🟥

Thread **2896** has been busy during the whole frame rate spike. This is the final thread in the chain that our **Update()** function was waiting for. Let's find out what it was actually doing this whole time, using the **CPU Usage (Sampled) provider**:



```
Line # Process  Thread ID  Stack                                                                                                             Count  Weight (in vie...
    10                       |  combase.dll!ComInvokeWithLockAndIPID                                                                            860    860,095513
    11                       |  combase.dll!AppInvoke                                                                                           860    860,095513
    12                       |  combase.dll!ServerCall::ContextInvoke                                                                           860    860,095513
    13                       |  combase.dll!DefaultStubInvoke                                                                                   860    860,095513
    14                       |  combase.dll!ObjectMethodExceptionHandlingAction<<lambda_b8ffcec6d47a5635f374132234a8dd15> >                     860    860,095513
    15                       |  combase.dll!CStdStubBuffer_Invoke                                                                               860    860,095513
    16                       |  rpcrt4.dll!NdrStubCall3                                                                                         860    860,095513
    17                       |  rpcrt4.dll!Ndr64StubWorker                                                                                      860    860,095513
    18                       |  rpcrt4.dll!Invoke                                                                                               860    860,095513
    19                    ▼  |  |- windows.storage.dll!CThreadAffineStorageQueryServer::CreateStorageQueryItemArray                              857    857,094309
    20                       |  |  windows.storage.dll!CStorageQueryItemArray::CreateStorageItems                                               857    857,094309
    21                       |  |  windows.storage.dll!Windows::Internal::NativeString<Windows::Internal::LocalMemPolicy<unsigned short> >::_EnsureCapacity  857    857,094309
    22                    ▼  |  |  |- windows.storage.dll!CreateStorageItemFromShellItem<CStorageFolder,CStorageFile>                            610    610,081858
    23                    ▼  |  |  |  |- windows.storage.dll!IsItemUnderHomeGroup                                                               200    200,023057
    24                    ▼  |  |  |  |  |- windows.storage.dll!IsIDListUnderHomeGroup                                                          198    198,022742
    25                    ▷  |  |  |  |  |  |- windows.storage.dll!CShellItem::GetCLSID                                                         195    195,022708
    26                    ▷  |  |  |  |  |  |- windows.storage.dll!SHCreateItemFromIDList                                                         2      2,000023
    27                    ▷  |  |  |  |  |  |- windows.storage.dll!CShellItem::Release                                                            1      1,000011
    28                    ▷  |  |  |  |  |- windows.storage.dll!SHGetIDListFromObject                                                             2      2,000315
    29                    ▼  |  |  |  |- windows.storage.dll!CreateStorageItemFromShellItem<CStorageFolder,CStorageFile>                      153    153,033644
    30                    ▷  |  |  |  |  |- windows.storage.dll!CStorageItem::Initialize                                                       152    152,033633
    31                    ▷  |  |  |  |  |- windows.storage.dll!Microsoft::WRL::Details::Make<CStorageFolder>                                    1      1,000011
    32                    ▼  |  |  |  |- windows.storage.dll!IsItemUnderLibrary                                                                 100    100,016945
    33                       |  |  |  |  windows.storage.dll!GetLibraryAncestor                                                                 100    100,016945
    34                    ▷  |  |  |  |  |- windows.storage.dll!CShellItem::BindToHandler                                                        94     94,015705
    35                    ▷  |  |  |  |  |- windows.storage.dll!CShellItem::GetParent                                                             4      4,000339
    36                    ▷  |  |  |  |  |- SHCore.dll!IUnknown_Set                                                                              1      1,000889
    37                       |  |  |  |  |- windows.storage.dll!GetLibraryAncestor<itself>                                                        1      1,000012
    38                    ▷  |  |  |  |- windows.storage.dll!IsItemUnderIndexedAppdata                                                           55     54,996226
    39                    ▷  |  |  |  |- windows.storage.dll!CShellItem::BindToHandler                                                           42     42,004572
    40                    ▷  |  |  |  |- windows.storage.dll!UpdateManager::IsForcedReadOnlyCachedFile                                           28     28,003539
    41                    ▷  |  |  |  |- windows.storage.dll!SetHiddenPropertyStore                                                              16     16,002818
    42                    ▷  |  |  |  |- windows.storage.dll!CallerIdentity::GetPackageFamilyNameFromProcess                                      8      7,999797
```

At first sight, thread **2896** looks like it's creating some kind of shell storage item array, and is querying its various properties. Unfortunately, we cannot really find out why it's taking so long - the whole code in the stacktrace is part of Windows OS, and is not available publicly.

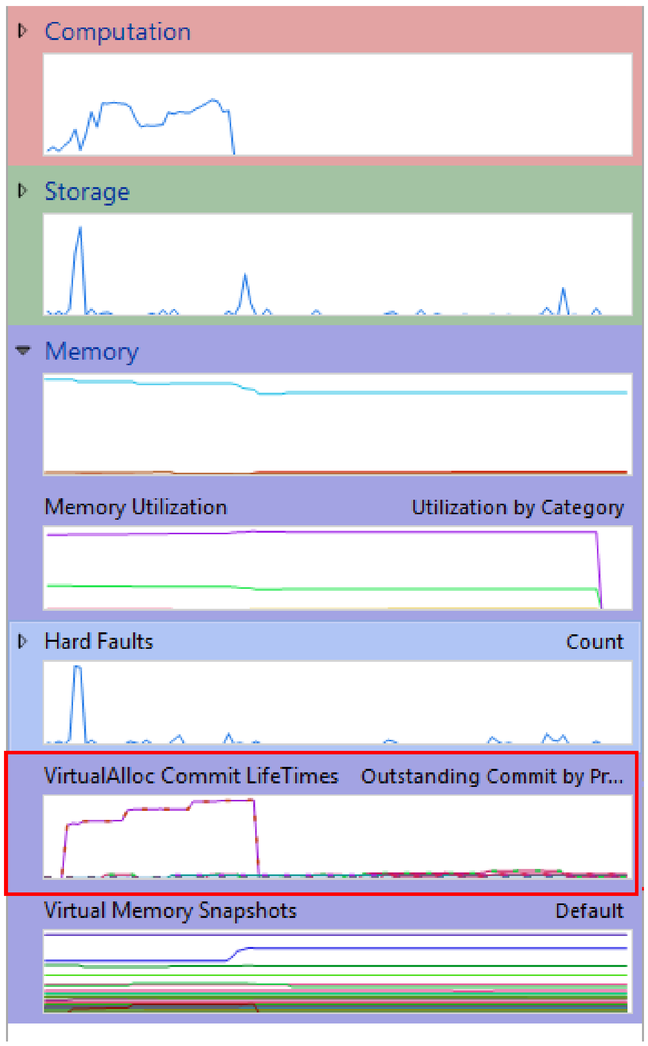
The only actionable thing to do here is to not do the call that causes the spike. Apparently, our **Update()** function was trying to enumerate AppData directory when it wanted to save the game - a process that can be very resource-intensive. It would be wise to do this enumeration at times when the delay caused by doing so would not impact the user (e.g. between screens, or when loading a scene), or find some other less resource-intensive way of getting the data we need to save the game.

VirtualAlloc Commit provider

Unity’s own built-in memory usage profiler can show you why the engine is using a certain amount of memory. However, the real memory usage is usually higher than the Unity memory profiler might suggest. The difference usually comes from the fact that Unity only knows how much it has itself allocated, and estimates the memory usage outside of its control (such as loaded executable images, graphics drivers, and plug-ins).

The VirtualAlloc Commit provider shows the allocated memory detected by the operating system. It is different to Unity’s built-in memory usage profiler, as it counts every single low-level instance of operating system virtual memory allocation, and allows you to investigate what led to the allocation.

To bring the VirtualAlloc Commit provider into the Analysis tab, expand the memory graph in the Graph Explorer and double-click the **VirtualAlloc Commit LifeTimes** graph:

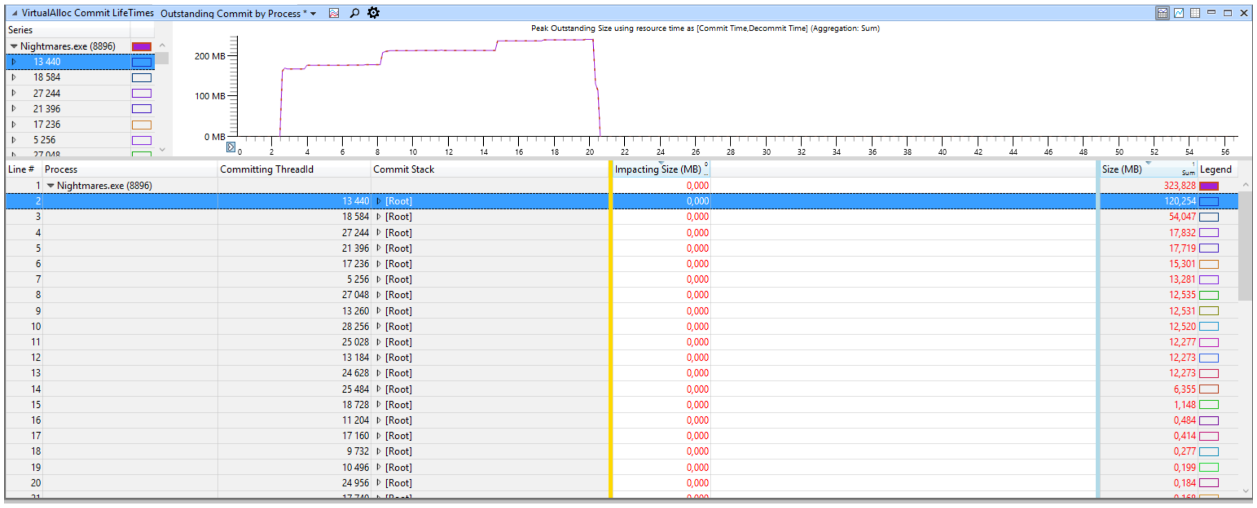


Each row in the VirtualAlloc Commit provider’s Analysis tab represents a virtual memory allocation. These are the most important columns of this provider:

21. **Process**: The process allocating the memory.
22. **Committing Thread Id**: The ID number of the thread that is allocating the memory.
23. **Commit Stack**: The stack trace at the time of memory allocation.
24. **Decommit Stack**: The stack trace at the time of freeing the allocated memory.
25. **Commit Time**: The point in time that the memory was allocated.
26. **Decommit Time**: The point in time that the memory was freed.
27. **Address**: The start address of the allocated memory region.
28. **Count**: The total allocation count in the row.
29. **Impacting Size**: The size of the allocated memory region, if the memory was not freed before the end of the zoomed region. If the memory was freed in the same region, the impact size is given as 0.

When investigating high memory usage, you'll mainly be focusing on allocated memory that was not freed. The **Impacting Size** column is the main focus point.

After capturing a trace, open the VirtualAlloc Commit provider Details view and filter the view to your own process to see something like this:

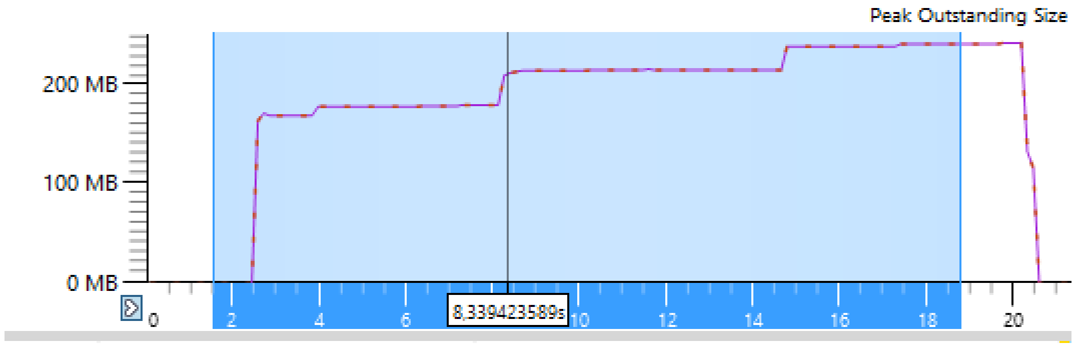

VirtualAlloc Commit LifeTimes
Outstanding Commit by Process *
Series
Nightmares.exe (8896)
13 440
18 584
27 244
21 396
17 236
5 256
Peak Outstanding Size using resource time as [Commit Time,Decommit Time] (Aggregation: Sum)
200 MB
100 MB
0 MB

Line #	Process	Committing ThreadId	Commit Stack	Impacting Size (MB)	Size (MB) Sum	Legend
1	Nightmares.exe (8896)			0,000	323,828	■
2		13 440	[Root]	0,000	120,254	□
3		18 584	[Root]	0,000	54,047	□
4		27 244	[Root]	0,000	17,832	□
5		21 396	[Root]	0,000	17,719	□
6		17 236	[Root]	0,000	15,301	□
7		5 256	[Root]	0,000	13,281	□
8		27 048	[Root]	0,000	12,535	□
9		13 260	[Root]	0,000	12,531	□
10		28 256	[Root]	0,000	12,520	□
11		25 028	[Root]	0,000	12,277	□
12		13 184	[Root]	0,000	12,273	□
13		24 628	[Root]	0,000	12,273	□
14		25 484	[Root]	0,000	6,355	□
15		18 728	[Root]	0,000	1,148	□
16		11 204	[Root]	0,000	0,484	□
17		17 160	[Root]	0,000	0,414	□
18		9 732	[Root]	0,000	0,277	□
19		10 496	[Root]	0,000	0,199	□
20		24 956	[Root]	0,000	0,184	□

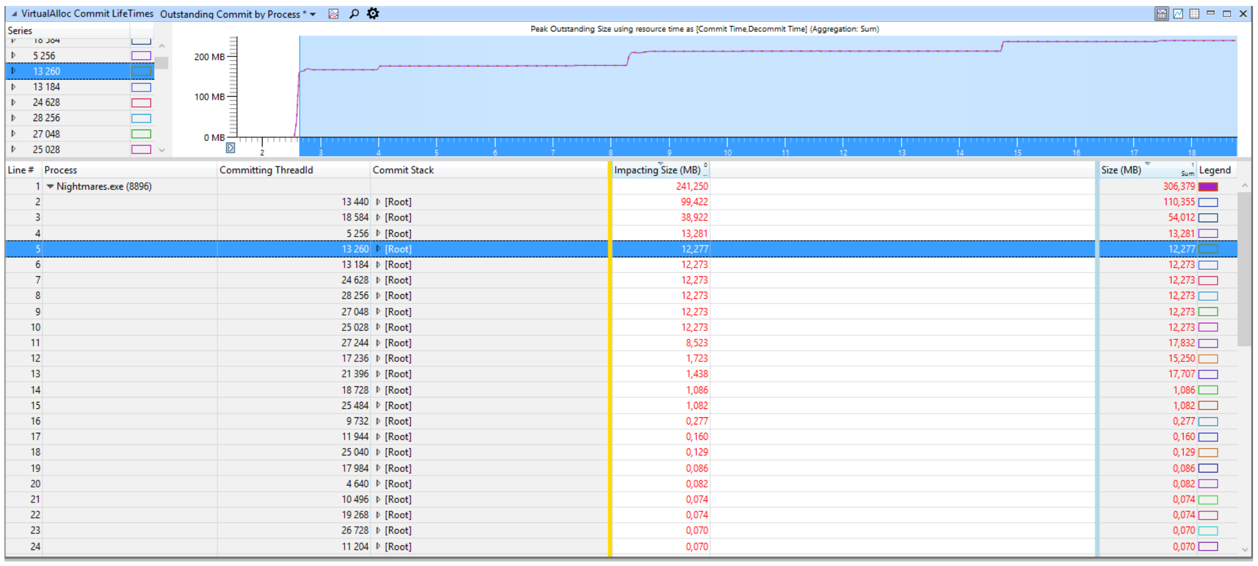
As you can see, the **Impacting Size** is reported to be 0MB on every single thread. This is due to the fact that the process both starts and exits while the stack is being recorded.

VirtualAlloc Commit provider: Example Walkthrough

Let’s say we’re interested in looking at memory usage at its peak. To do so, we need to select a time range which starts before the process was started, and ends at its memory usage peak:



After zooming into the selected region on the graph, we can now see the **Impacting Size (MB)** column telling us how much memory was used at the end of the selection:

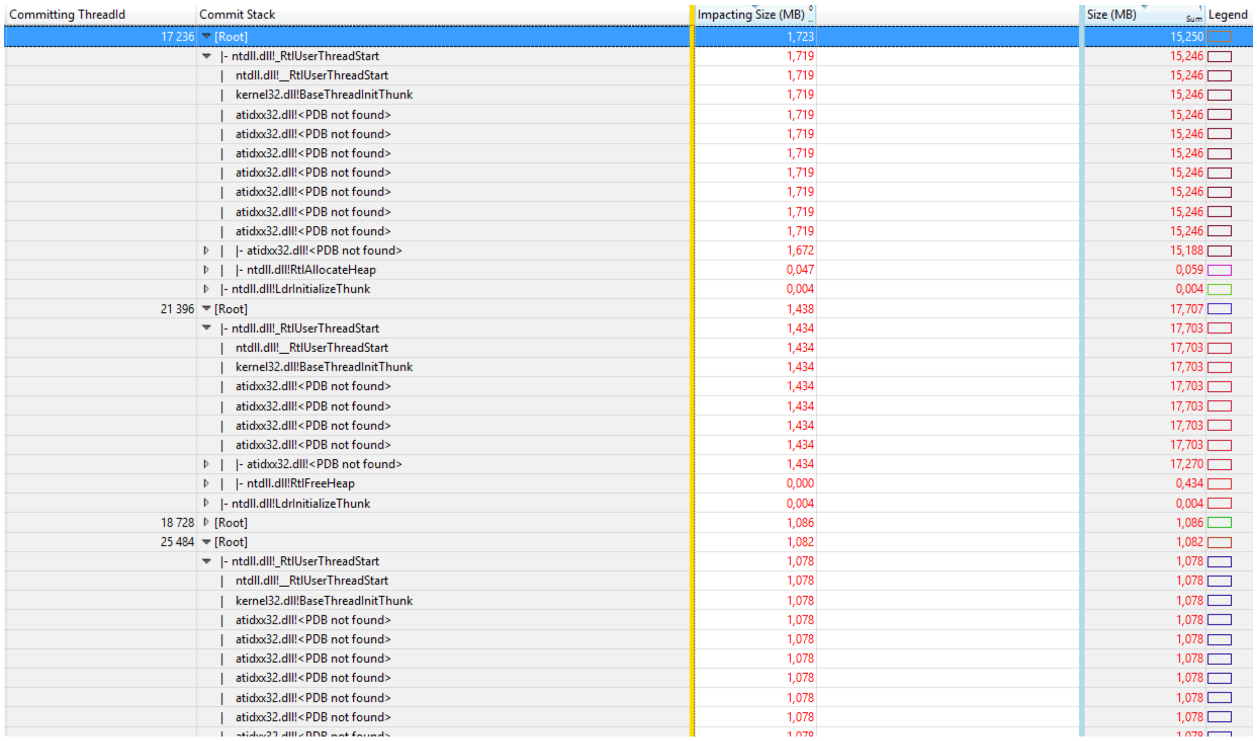

VirtualAlloc Commit LifeTimes
Outstanding Commit by Process *
Series
Peak Outstanding Size using resource time as [Commit Time,Decommit Time] (Aggregation: Sum)
200 MB
100 MB
0 MB

Series	Legend
5 256	🟪
13 260	🟩
13 184	🟦
24 628	🟥
28 256	🟦
27 048	🟩
25 028	🟪

Line #	Process	Committing ThreadId	Commit Stack	Impacting Size (MB)	Size (MB) Sum	Legend
1	Nightmares.exe (8896)			241,250	306,379	🟪
2		13 440	[Root]	99,422	110,355	🟦
3		18 584	[Root]	38,922	54,012	🟦
4		5 256	[Root]	13,281	13,281	🟪
5		13 260	[Root]	12,277	12,277	🟩
6		13 184	[Root]	12,273	12,273	🟦
7		24 628	[Root]	12,273	12,273	🟥
8		28 256	[Root]	12,273	12,273	🟦
9		27 048	[Root]	12,273	12,273	🟩
10		25 028	[Root]	12,273	12,273	🟪
11		27 244	[Root]	8,523	17,832	🟪
12		17 236	[Root]	1,723	15,250	🟧
13		21 396	[Root]	1,438	17,707	🟪
14		18 728	[Root]	1,086	1,086	🟩
15		25 484	[Root]	1,082	1,082	🟥
16		9 732	[Root]	0,277	0,277	🟦
17		11 944	[Root]	0,160	0,160	🟦
18		25 040	[Root]	0,129	0,129	🟧
19		17 984	[Root]	0,086	0,086	🟦
20		4 640	[Root]	0,082	0,082	🟪
21		10 496	[Root]	0,074	0,074	🟩
22		19 268	[Root]	0,074	0,074	🟥
23		26 728	[Root]	0,070	0,070	🟦
24		11 204	[Root]	0,070	0,070	🟪

Let’s investigate the memory investigations of each thread, starting with the ones that allocated at least a megabyte. Anything less than that is probably insignificant.

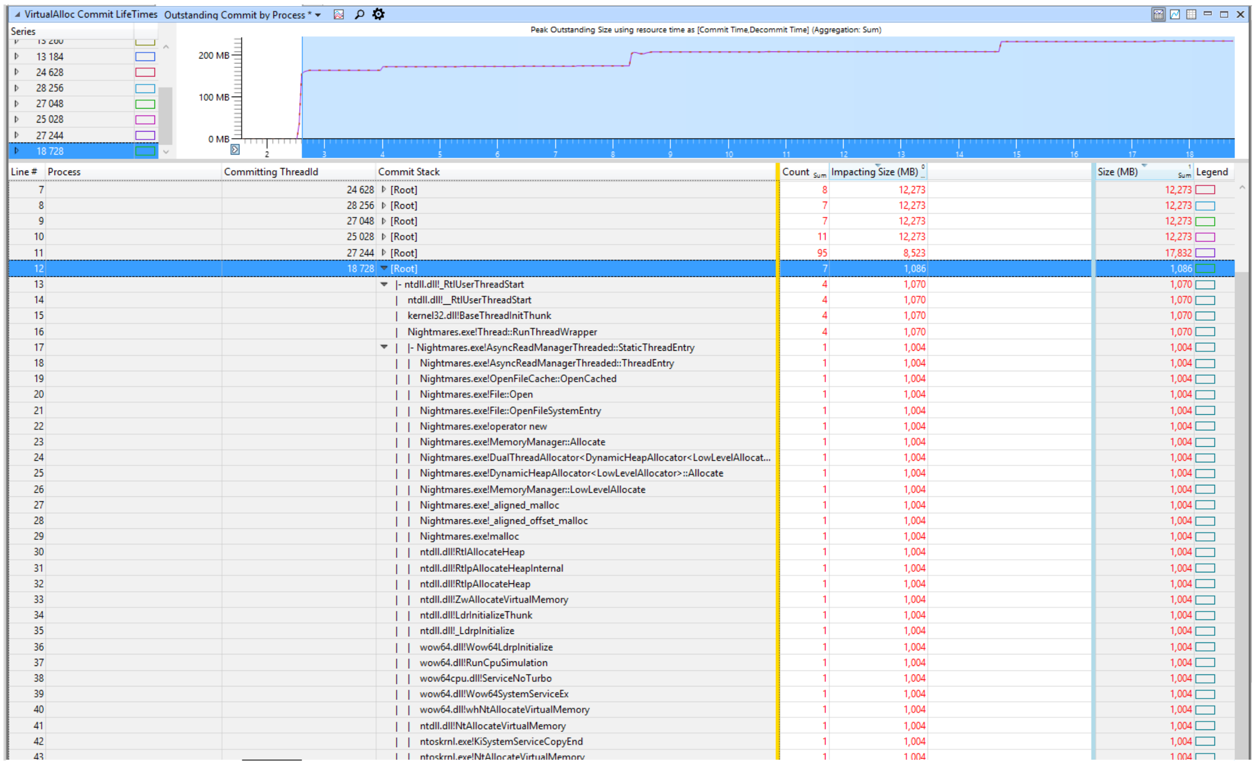
Straight away, we can identify that threads **17236**, **21396** and **25484** allocate memory for the graphics driver:



Committing ThreadId	Commit Stack	Impacting Size (MB)	Size (MB) Sum	Legend
17 236	[Root]	1,723	15,250	🟥
	|- ntdll.dll!_RtlUserThreadStart	1,719	15,246	🟥
	| ntdll.dll!__RtlUserThreadStart	1,719	15,246	🟥
	| kernel32.dll!BaseThreadInitThunk	1,719	15,246	🟥
	| atidxx32.dll!<PDB not found>	1,719	15,246	🟥
	| atidxx32.dll!<PDB not found>	1,719	15,246	🟥
	| atidxx32.dll!<PDB not found>	1,719	15,246	🟥
	| atidxx32.dll!<PDB not found>	1,719	15,246	🟥
	| atidxx32.dll!<PDB not found>	1,719	15,246	🟥
	| atidxx32.dll!<PDB not found>	1,719	15,246	🟥
	| atidxx32.dll!<PDB not found>	1,719	15,246	🟥
	| |- atidxx32.dll!<PDB not found>	1,672	15,188	🟥
	| |- ntdll.dll!RtlAllocateHeap	0,047	0,059	🟪
	|- ntdll.dll!LdrInitializeThunk	0,004	0,004	🟩
21 396	[Root]	1,438	17,707	🟦
	|- ntdll.dll!_RtlUserThreadStart	1,434	17,703	🟥
	| ntdll.dll!__RtlUserThreadStart	1,434	17,703	🟥
	| kernel32.dll!BaseThreadInitThunk	1,434	17,703	🟥
	| atidxx32.dll!<PDB not found>	1,434	17,703	🟥
	| atidxx32.dll!<PDB not found>	1,434	17,703	🟥
	| atidxx32.dll!<PDB not found>	1,434	17,703	🟥
	| atidxx32.dll!<PDB not found>	1,434	17,703	🟥
	| |- atidxx32.dll!<PDB not found>	1,434	17,270	🟥
	| |- ntdll.dll!RtlFreeHeap	0,000	0,434	🟥
	|- ntdll.dll!LdrInitializeThunk	0,004	0,004	🟥
18 728	[Root]	1,086	1,086	🟩
25 484	[Root]	1,082	1,082	🟥
	|- ntdll.dll!_RtlUserThreadStart	1,078	1,078	🟦
	| ntdll.dll!__RtlUserThreadStart	1,078	1,078	🟦
	| kernel32.dll!BaseThreadInitThunk	1,078	1,078	🟦
	| atidxx32.dll!<PDB not found>	1,078	1,078	🟦
	| atidxx32.dll!<PDB not found>	1,078	1,078	🟦
	| atidxx32.dll!<PDB not found>	1,078	1,078	🟦
	| atidxx32.dll!<PDB not found>	1,078	1,078	🟦
	| atidxx32.dll!<PDB not found>	1,078	1,078	🟦
	| atidxx32.dll!<PDB not found>	1,078	1,078	🟦
	| atidxx32.dll!<PDB not found>	1,078	1,078	🟦

You can't really influence what the driver does, and since we don't have symbols for the driver either, we can't tell why it's allocating the memory. Let's move on.

Thread 18728 is Unity's **AsyncReadManager** thread:


VirtualAlloc Commit LifeTimes
Outstanding Commit by Process *
Series
Peak Outstanding Size using resource time as [Commit Time,Decommit Time] (Aggregation: Sum)
200 MB
100 MB
0 MB
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Series	
13 184	🟦
24 628	🟥
28 256	🟦
27 048	🟩
25 028	🟪
27 244	🟪
18 728	🟩

Line #	Process	Committing ThreadId	Commit Stack	Count Sum	Impacting Size (MB)	Size (MB) Sum	Legend
7		24 628	[Root]	8	12,273	12,273	🟥
8		28 256	[Root]	7	12,273	12,273	🟦
9		27 048	[Root]	7	12,273	12,273	🟩
10		25 028	[Root]	11	12,273	12,273	🟪
11		27 244	[Root]	95	8,523	17,832	🟪
12		18 728	[Root]	7	1,086	1,086	🟩
13			|- ntdll.dll!_RtlUserThreadStart	4	1,070	1,070	🟦
14			| ntdll.dll!__RtlUserThreadStart	4	1,070	1,070	🟦
15			| kernel32.dll!BaseThreadInitThunk	4	1,070	1,070	🟦
16			| Nightmares.exe!Thread::RunThreadWrapper	4	1,070	1,070	🟦
17			| |- Nightmares.exe!AsyncReadManagerThreaded::StaticThreadEntry	1	1,004	1,004	🟦
18			| | Nightmares.exe!AsyncReadManagerThreaded::ThreadEntry	1	1,004	1,004	🟦
19			| | Nightmares.exe!OpenFileCache::OpenCached	1	1,004	1,004	🟦
20			| | Nightmares.exe!File::Open	1	1,004	1,004	🟦
21			| | Nightmares.exe!File::OpenFileSystemEntry	1	1,004	1,004	🟦
22			| | Nightmares.exe!operator new	1	1,004	1,004	🟦
23			| | Nightmares.exe!MemoryManager::Allocate	1	1,004	1,004	🟦
24			| | Nightmares.exe!DualThreadAllocator<DynamicHeapAllocator<LowLevelAllocat...	1	1,004	1,004	🟦
25			| | Nightmares.exe!DynamicHeapAllocator<LowLevelAllocator>::Allocate	1	1,004	1,004	🟦
26			| | Nightmares.exe!MemoryManager::LowLevelAllocate	1	1,004	1,004	🟦
27			| | Nightmares.exe!_aligned_malloc	1	1,004	1,004	🟦
28			| | Nightmares.exe!_aligned_offset_malloc	1	1,004	1,004	🟦
29			| | Nightmares.exe!malloc	1	1,004	1,004	🟦
30			| | ntdll.dll!RtlAllocateHeap	1	1,004	1,004	🟦
31			| | ntdll.dll!RtlpAllocateHeapInternal	1	1,004	1,004	🟦
32			| | ntdll.dll!RtlpAllocateHeap	1	1,004	1,004	🟦
33			| | ntdll.dll!ZwAllocateVirtualMemory	1	1,004	1,004	🟦
34			| | ntdll.dll!LdrInitializeThunk	1	1,004	1,004	🟦
35			| | ntdll.dll!_LdrpInitialize	1	1,004	1,004	🟦
36			| | wow64.dll!Wow64LdrpInitialize	1	1,004	1,004	🟦
37			| | wow64.dll!RunCpuSimulation	1	1,004	1,004	🟦
38			| | wow64cpu.dll!ServiceNoTurbo	1	1,004	1,004	🟦
39			| | wow64.dll!Wow64SystemServiceEx	1	1,004	1,004	🟦
40			| | wow64.dll!whNtAllocateVirtualMemory	1	1,004	1,004	🟦
41			| | ntdll.dll!NtAllocateVirtualMemory	1	1,004	1,004	🟦
42			| | ntoskrnl.exe!KiSystemServiceCopyEnd	1	1,004	1,004	🟦
43			| | ntoskrnl.exe!NtAllocateVirtualMemory	1	1,004	1,004	🟦

AsyncReadManager is a dedicated thread for reading files from disk. You may have noticed two peculiarities with this stack trace:

- It allocated an incredibly round number of bytes during one of its allocations: 1.004 MB.
- That allocation comprises the majority of the total memory used by the thread, and was allocated by the “**File::Open**” function.

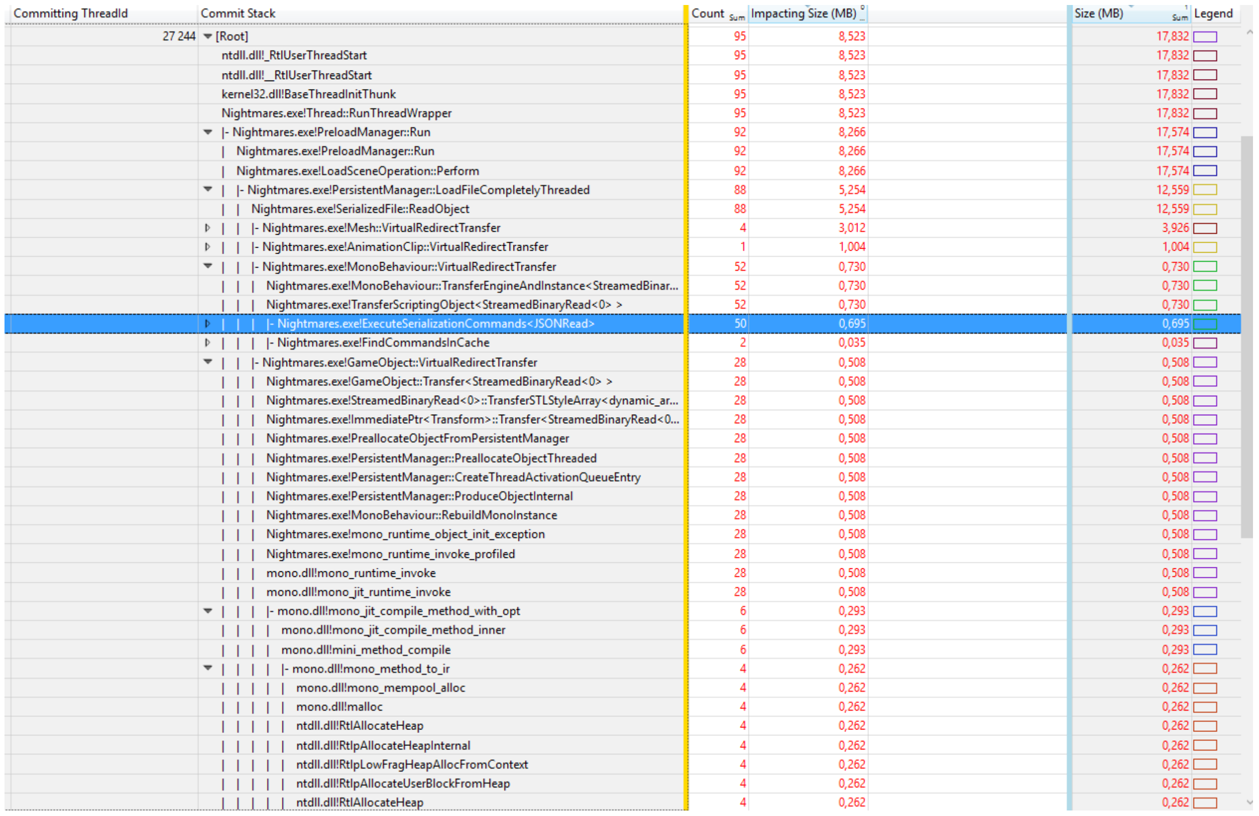
It would seem unlikely that opening a single file would cost so much memory, and for this to be the reason for the majority of thread memory usage. However, this isn’t a coincidence - it’s a consequence of how Unity allocates memory internally.

Unlike many applications, Unity doesn’t request memory from the OS and then return it once it’s no longer needed. Instead, it requests memory in chunks of 1 MB, and reuses these chunks without giving them back to the OS. You can see the difference between Unity actually using the memory versus requesting it from the OS by looking at the used and reserved memory in Unity’s internal memory profiler.

Depending on the target platform, Unity either keeps separate memory reservations per thread (on desktops/consoles), or has a shared pool for all the worker threads (on mobile devices). On Windows, shared pool is used only on ARM devices.

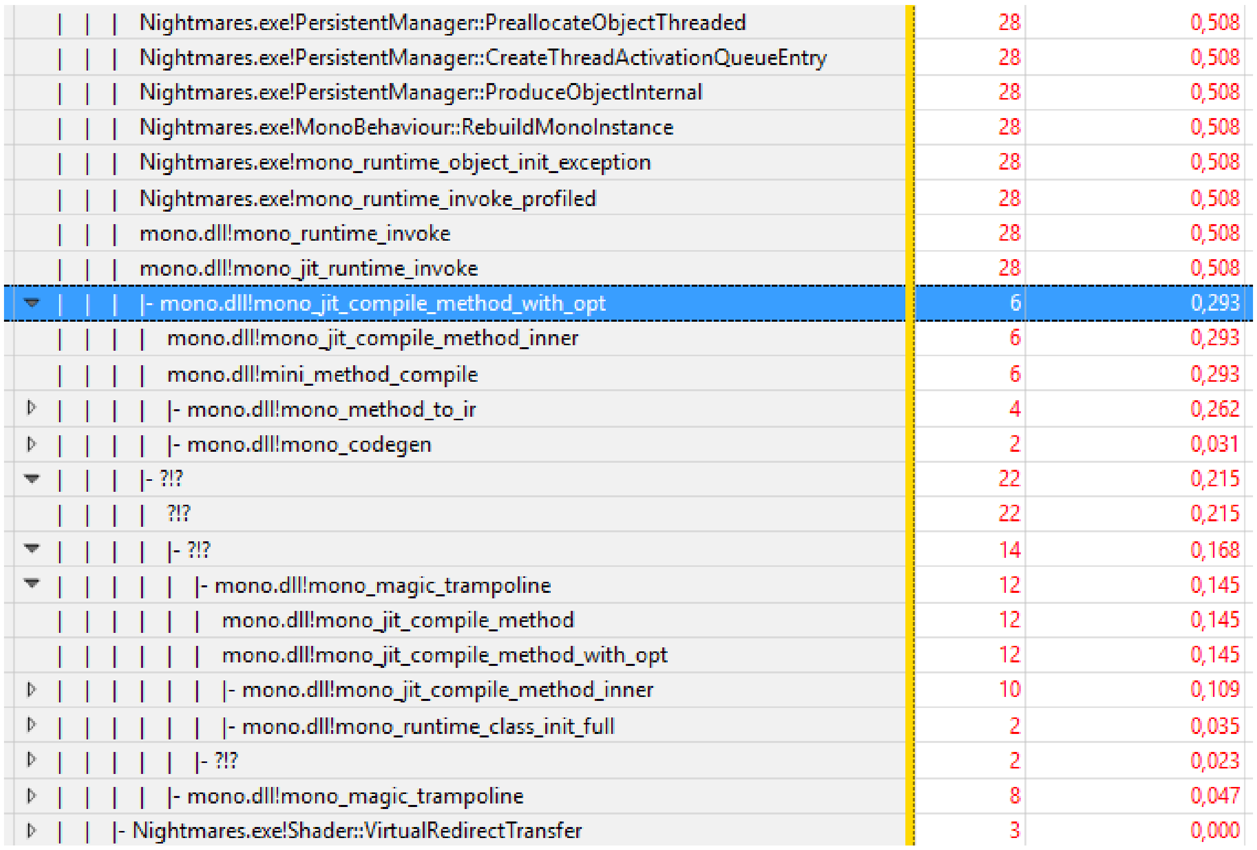
Due to the way Unity manages its memory, the VirtualAlloc Commit provider can’t really show what Unity is doing with it - if you want to investigate that, you’ll have to use Unity’s internal memory profiler. You can recognize Unity-managed allocation by looking at the callstack and seeing whether “**MemoryManager::Allocate**” is part of it. You will almost never want to look at these allocations with Windows Performance Analysis.

The next thread in the list is thread **27244**:



Committing ThreadId	Commit Stack	Count Sum	Impacting Size (MB)	Size (MB) Sum	Legend
27 244	▼ [Root]	95	8,523	17,832	🟪
	ntdll.dll!_RtlUserThreadStart	95	8,523	17,832	🟥
	ntdll.dll!__RtlUserThreadStart	95	8,523	17,832	🟥
	kernel32.dll!BaseThreadInitThunk	95	8,523	17,832	🟥
	Nightmares.exe!Thread::RunThreadWrapper	95	8,523	17,832	🟥
	▼ |- Nightmares.exe!PreloadManager::Run	92	8,266	17,574	🟦
	| Nightmares.exe!PreloadManager::Run	92	8,266	17,574	🟦
	| Nightmares.exe!LoadSceneOperation::Perform	92	8,266	17,574	🟦
	▼ | |- Nightmares.exe!PersistentManager::LoadFileCompletelyThreaded	88	5,254	12,559	🟨
	| | Nightmares.exe!SerializedFile::ReadObject	88	5,254	12,559	🟨
	▷ | | |- Nightmares.exe!Mesh::VirtualRedirectTransfer	4	3,012	3,926	🟥
	▷ | | |- Nightmares.exe!AnimationClip::VirtualRedirectTransfer	1	1,004	1,004	🟨
	▼ | | |- Nightmares.exe!MonoBehaviour::VirtualRedirectTransfer	52	0,730	0,730	🟩
	| | | Nightmares.exe!MonoBehaviour::TransferEngineAndInstance<StreamedBinar...	52	0,730	0,730	🟩
	| | | Nightmares.exe!TransferScriptingObject<StreamedBinaryRead<0> >	52	0,730	0,730	🟩
	▷ | | | |- Nightmares.exe!ExecuteSerializationCommands<JSONRead>	50	0,695	0,695	🟩
	▷ | | | |- Nightmares.exe!FindCommandsInCache	2	0,035	0,035	🟪
	▼ | | |- Nightmares.exe!GameObject::VirtualRedirectTransfer	28	0,508	0,508	🟪
	| | | Nightmares.exe!GameObject::Transfer<StreamedBinaryRead<0> >	28	0,508	0,508	🟪
	| | | Nightmares.exe!StreamedBinaryRead<0>::TransferSTLStyleArray<dynamic_ar...	28	0,508	0,508	🟪
	| | | Nightmares.exe!ImmediatePtr<Transform>::Transfer<StreamedBinaryRead<0...	28	0,508	0,508	🟪
	| | | Nightmares.exe!PreallocateObjectFromPersistentManager	28	0,508	0,508	🟪
	| | | Nightmares.exe!PersistentManager::PreallocateObjectThreaded	28	0,508	0,508	🟪
	| | | Nightmares.exe!PersistentManager::CreateThreadActivationQueueEntry	28	0,508	0,508	🟪
	| | | Nightmares.exe!PersistentManager::ProduceObjectInternal	28	0,508	0,508	🟪
	| | | Nightmares.exe!MonoBehaviour::RebuildMonoInstance	28	0,508	0,508	🟪
	| | | Nightmares.exe!mono_runtime_object_init_exception	28	0,508	0,508	🟪
	| | | Nightmares.exe!mono_runtime_invoke_profiled	28	0,508	0,508	🟪
	| | | mono.dll!mono_runtime_invoke	28	0,508	0,508	🟪
	| | | mono.dll!mono_jit_runtime_invoke	28	0,508	0,508	🟪
	▼ | | | |- mono.dll!mono_jit_compile_method_with_opt	6	0,293	0,293	🟦
	| | | | mono.dll!mono_jit_compile_method_inner	6	0,293	0,293	🟦
	| | | | mono.dll!mini_method_compile	6	0,293	0,293	🟦
	▼ | | | | |- mono.dll!mono_method_to_ir	4	0,262	0,262	🟧
	| | | | | mono.dll!mono_mempool_alloc	4	0,262	0,262	🟧
	| | | | | mono.dll!malloc	4	0,262	0,262	🟧
	| | | | | ntdll.dll!RtlAllocateHeap	4	0,262	0,262	🟧
	| | | | | ntdll.dll!RtlpAllocateHeapInternal	4	0,262	0,262	🟧
	| | | | | ntdll.dll!RtlpLowFragHeapAllocFromContext	4	0,262	0,262	🟧
	| | | | | ntdll.dll!RtlpAllocateUserBlockFromHeap	4	0,262	0,262	🟧
	| | | | | ntdll.dll!RtlAllocateHeap	4	0,262	0,262	🟧

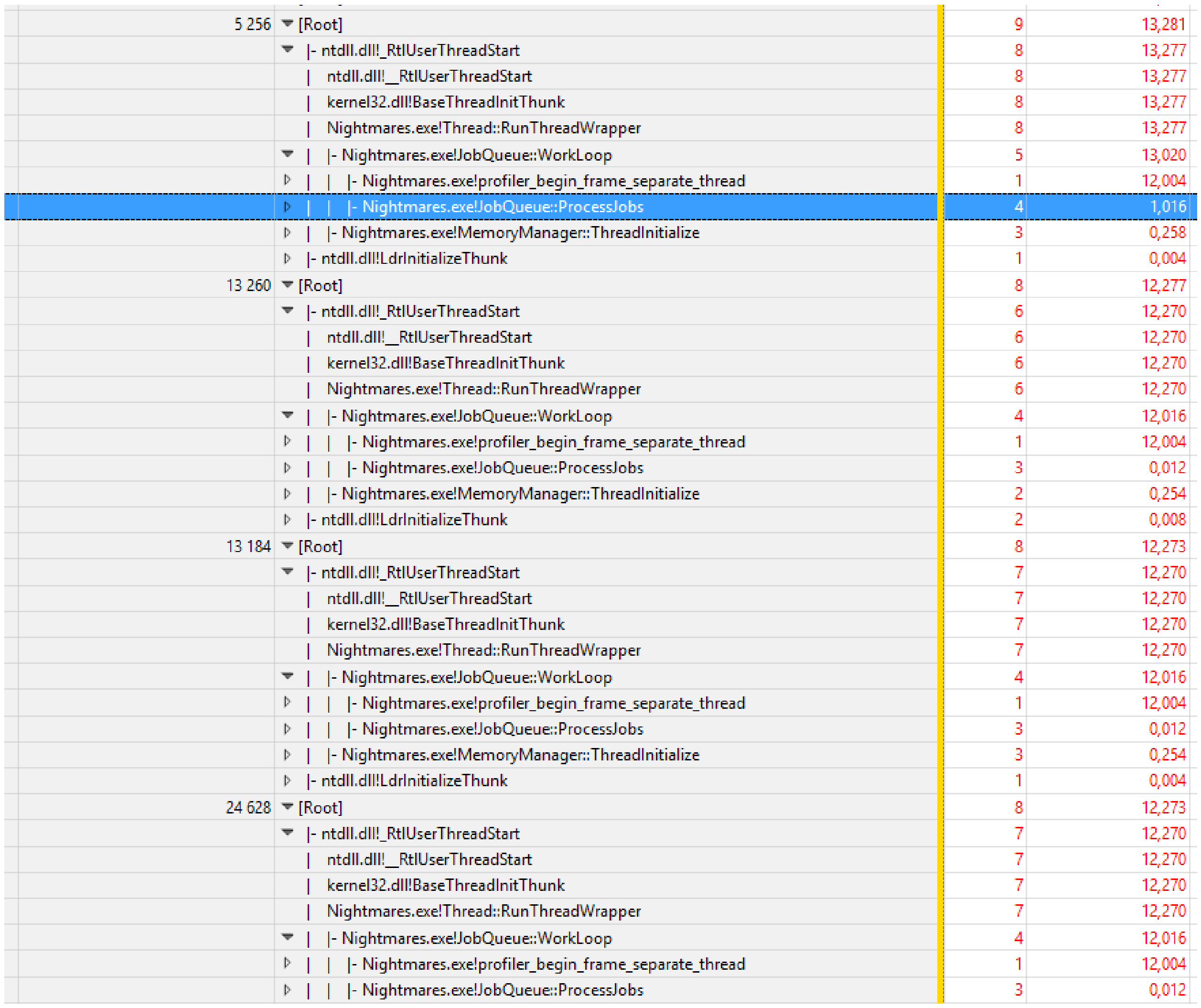
This is Unity’s loading thread. It can be recognized by the fact that it starts with the “**PreloadManager::Run**” function. Most of the allocations in this thread go through Unity’s MemoryManager, so we will not look at that. However, there are also other allocations coming from code JIT-ing:



```
|  |  |  Nightmares.exe!PersistentManager::PreallocateObjectThreaded          28   0,508
|  |  |  Nightmares.exe!PersistentManager::CreateThreadActivationQueueEntry   28   0,508
|  |  |  Nightmares.exe!PersistentManager::ProduceObjectInternal             28   0,508
|  |  |  Nightmares.exe!MonoBehaviour::RebuildMonoInstance                   28   0,508
|  |  |  Nightmares.exe!mono_runtime_object_init_exception                   28   0,508
|  |  |  Nightmares.exe!mono_runtime_invoke_profiled                         28   0,508
|  |  |  mono.dll!mono_runtime_invoke                                        28   0,508
|  |  |  mono.dll!mono_jit_runtime_invoke                                    28   0,508
|  |  |  |- mono.dll!mono_jit_compile_method_with_opt                         6   0,293
|  |  |  |  mono.dll!mono_jit_compile_method_inner                            6   0,293
|  |  |  |  mono.dll!mini_method_compile                                      6   0,293
|  |  |  |  |- mono.dll!mono_method_to_ir                                     4   0,262
|  |  |  |  |- mono.dll!mono_codegen                                         2   0,031
|  |  |  |- ?!?                                                             22   0,215
|  |  |  |  ?!?                                                             22   0,215
|  |  |  |  |- ?!?                                                          14   0,168
|  |  |  |  |  |- mono.dll!mono_magic_trampoline                           12   0,145
|  |  |  |  |  |  mono.dll!mono_jit_compile_method                         12   0,145
|  |  |  |  |  |  mono.dll!mono_jit_compile_method_with_opt                12   0,145
|  |  |  |  |  |  |- mono.dll!mono_jit_compile_method_inner               10   0,109
|  |  |  |  |  |  |- mono.dll!mono_runtime_class_init_full                 2   0,035
|  |  |  |  |  |- ?!?                                                       2   0,023
|  |  |  |  |- mono.dll!mono_magic_trampoline                               8   0,047
|  |  |- Nightmares.exe!Shader::VirtualRedirectTransfer                      3   0,000
```

The more scripting code you have, the more memory will be allocated by Mono and .NET JITs at runtime. Note that there's no JIT when using IL2CPP scripting backend or .NET scripting backend with .NET Native enabled.

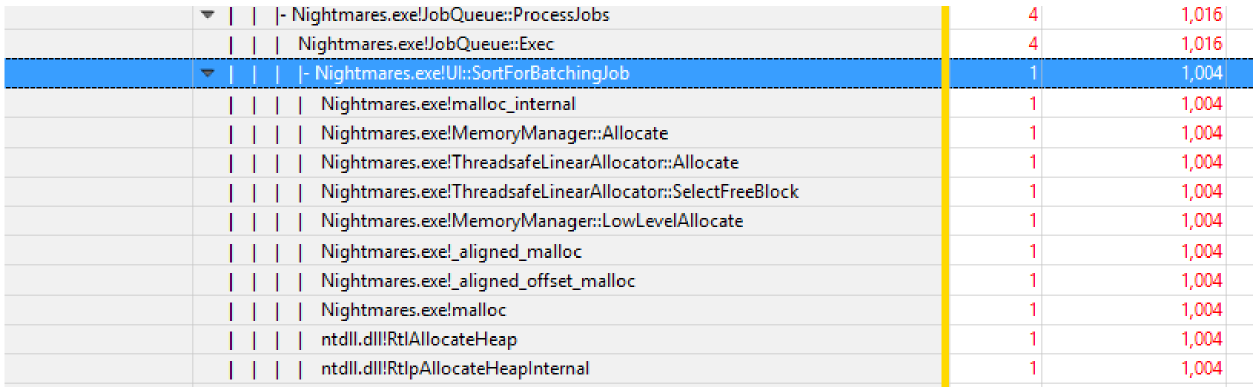
Let's continue our investigation. The next 7 threads are Unity's JobQueue threads:



```
 5 256  [Root]                                                              9   13,281
          |- ntdll.dll!_RtlUserThreadStart                                  8   13,277
          |   ntdll.dll!__RtlUserThreadStart                                8   13,277
          |   kernel32.dll!BaseThreadInitThunk                              8   13,277
          |   Nightmares.exe!Thread::RunThreadWrapper                       8   13,277
          |   |- Nightmares.exe!JobQueue::WorkLoop                          5   13,020
          |   |   |- Nightmares.exe!profiler_begin_frame_separate_thread    1   12,004
          |   |   |- Nightmares.exe!JobQueue::ProcessJobs                   4    1,016
          |   |- Nightmares.exe!MemoryManager::ThreadInitialize             3    0,258
          |- ntdll.dll!LdrInitializeThunk                                   1    0,004
13 260  [Root]                                                              8   12,277
          |- ntdll.dll!_RtlUserThreadStart                                  6   12,270
          |   ntdll.dll!__RtlUserThreadStart                                6   12,270
          |   kernel32.dll!BaseThreadInitThunk                              6   12,270
          |   Nightmares.exe!Thread::RunThreadWrapper                       6   12,270
          |   |- Nightmares.exe!JobQueue::WorkLoop                          4   12,016
          |   |   |- Nightmares.exe!profiler_begin_frame_separate_thread    1   12,004
          |   |   |- Nightmares.exe!JobQueue::ProcessJobs                   3    0,012
          |   |- Nightmares.exe!MemoryManager::ThreadInitialize             2    0,254
          |- ntdll.dll!LdrInitializeThunk                                   2    0,008
13 184  [Root]                                                              8   12,273
          |- ntdll.dll!_RtlUserThreadStart                                  7   12,270
          |   ntdll.dll!__RtlUserThreadStart                                7   12,270
          |   kernel32.dll!BaseThreadInitThunk                              7   12,270
          |   Nightmares.exe!Thread::RunThreadWrapper                       7   12,270
          |   |- Nightmares.exe!JobQueue::WorkLoop                          4   12,016
          |   |   |- Nightmares.exe!profiler_begin_frame_separate_thread    1   12,004
          |   |   |- Nightmares.exe!JobQueue::ProcessJobs                   3    0,012
          |   |- Nightmares.exe!MemoryManager::ThreadInitialize             3    0,254
          |- ntdll.dll!LdrInitializeThunk                                   1    0,004
24 628  [Root]                                                              8   12,273
          |- ntdll.dll!_RtlUserThreadStart                                  7   12,270
          |   ntdll.dll!__RtlUserThreadStart                                7   12,270
          |   kernel32.dll!BaseThreadInitThunk                              7   12,270
          |   Nightmares.exe!Thread::RunThreadWrapper                       7   12,270
          |   |- Nightmares.exe!JobQueue::WorkLoop                          4   12,016
          |   |   |- Nightmares.exe!profiler_begin_frame_separate_thread    1   12,004
          |   |   |- Nightmares.exe!JobQueue::ProcessJobs                   3    0,012
```

As seen from the screenshot, almost all memory allocated for these threads is coming from "**profiler_begin_frame_separate_thread**", which allocates 12MB on each thread. Unity pre-allocates 12 MB per JobQueue thread for the profiler events in development builds.

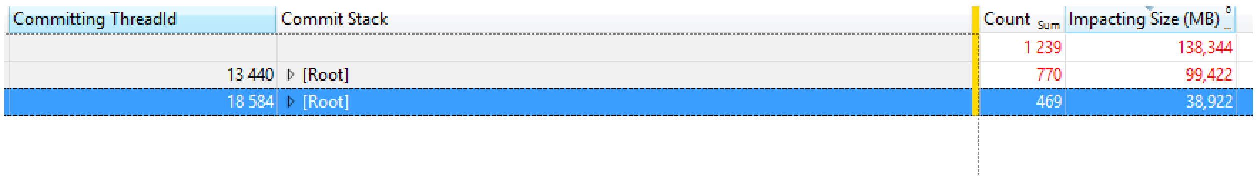
The only other allocation by these threads comes from "**UI::SortForBatchingJob**":



```
| | |- Nightmares.exe!JobQueue::ProcessJobs                      4   1,016
| | |   Nightmares.exe!JobQueue::Exec                            4   1,016
| | |   |- Nightmares.exe!UI::SortForBatchingJob                 1   1,004
| | |   |   Nightmares.exe!malloc_internal                       1   1,004
| | |   |   Nightmares.exe!MemoryManager::Allocate               1   1,004
| | |   |   Nightmares.exe!ThreadsafeLinearAllocator::Allocate   1   1,004
| | |   |   Nightmares.exe!ThreadsafeLinearAllocator::SelectFreeBlock   1   1,004
| | |   |   Nightmares.exe!MemoryManager::LowLevelAllocate       1   1,004
| | |   |   Nightmares.exe!_aligned_malloc                       1   1,004
| | |   |   Nightmares.exe!_aligned_offset_malloc                1   1,004
| | |   |   Nightmares.exe!malloc                                1   1,004
| | |   |   ntdll.dll!RtlAllocateHeap                            1   1,004
| | |   |   ntdll.dll!RtlpAllocateHeapInternal                   1   1,004
```

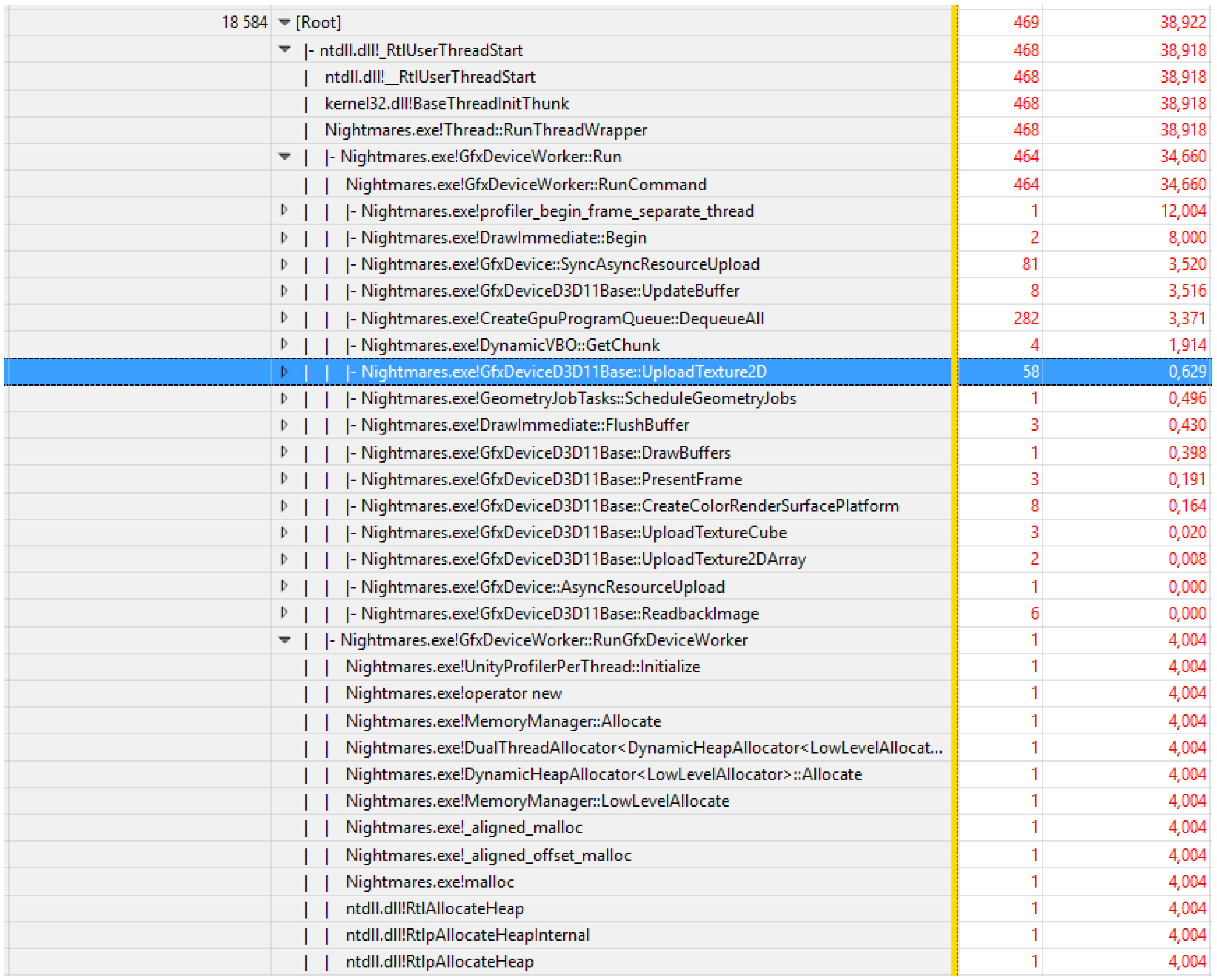
However, since this is allocated through MemoryManager, we cannot be sure whether that is what is using the memory.

We have two threads to go:



Committing ThreadId	Commit Stack	Count Sum	Impacting Size (MB)
		1 239	138,344
13 440	▷ [Root]	770	99,422
18 584	▷ [Root]	469	38,922

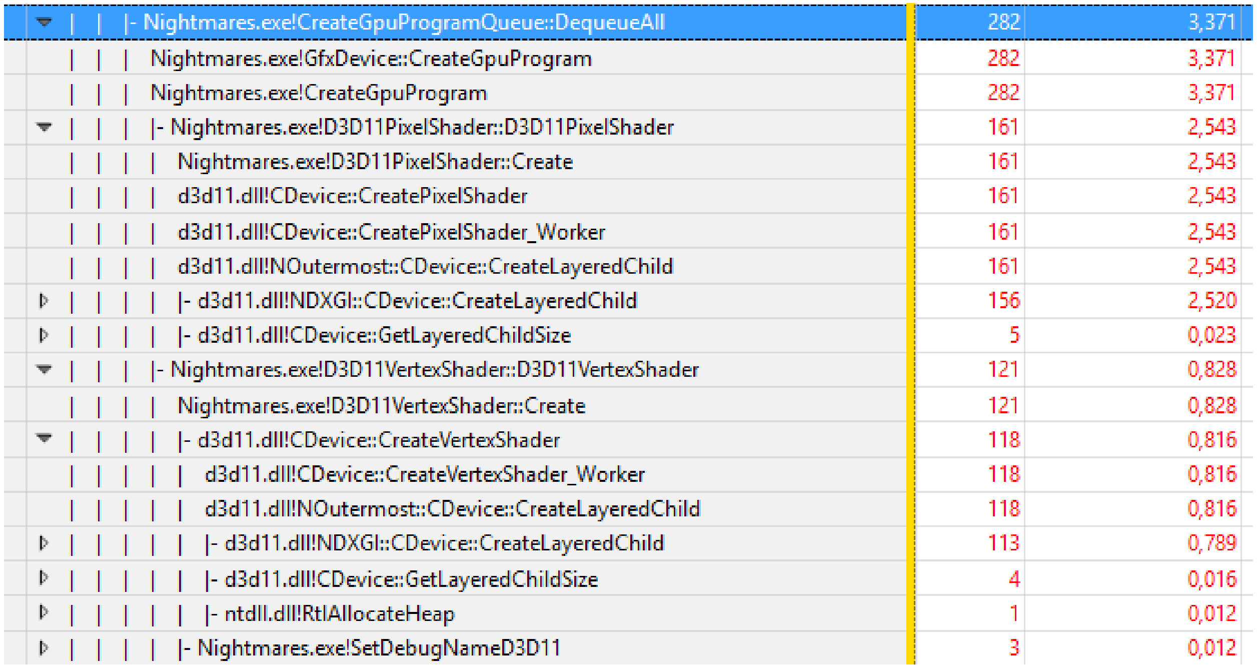
The bottom thread, **18584**, is Unity’s rendering thread:



18 584	▼ [Root]	469	38,922
	▼ |- ntdll.dll!_RtlUserThreadStart	468	38,918
	| ntdll.dll!__RtlUserThreadStart	468	38,918
	| kernel32.dll!BaseThreadInitThunk	468	38,918
	| Nightmares.exe!Thread::RunThreadWrapper	468	38,918
	▼ | |- Nightmares.exe!GfxDeviceWorker::Run	464	34,660
	| | Nightmares.exe!GfxDeviceWorker::RunCommand	464	34,660
	▷ | | |- Nightmares.exe!profiler_begin_frame_separate_thread	1	12,004
	▷ | | |- Nightmares.exe!DrawImmediate::Begin	2	8,000
	▷ | | |- Nightmares.exe!GfxDevice::SyncAsyncResourceUpload	81	3,520
	▷ | | |- Nightmares.exe!GfxDeviceD3D11Base::UpdateBuffer	8	3,516
	▷ | | |- Nightmares.exe!CreateGpuProgramQueue::DequeueAll	282	3,371
	▷ | | |- Nightmares.exe!DynamicVBO::GetChunk	4	1,914
	▷ | | |- Nightmares.exe!GfxDeviceD3D11Base::UploadTexture2D	58	0,629
	▷ | | |- Nightmares.exe!GeometryJobTasks::ScheduleGeometryJobs	1	0,496
	▷ | | |- Nightmares.exe!DrawImmediate::FlushBuffer	3	0,430
	▷ | | |- Nightmares.exe!GfxDeviceD3D11Base::DrawBuffers	1	0,398
	▷ | | |- Nightmares.exe!GfxDeviceD3D11Base::PresentFrame	3	0,191
	▷ | | |- Nightmares.exe!GfxDeviceD3D11Base::CreateColorRenderSurfacePlatform	8	0,164
	▷ | | |- Nightmares.exe!GfxDeviceD3D11Base::UploadTextureCube	3	0,020
	▷ | | |- Nightmares.exe!GfxDeviceD3D11Base::UploadTexture2DArray	2	0,008
	▷ | | |- Nightmares.exe!GfxDevice::AsyncResourceUpload	1	0,000
	▷ | | |- Nightmares.exe!GfxDeviceD3D11Base::ReadbackImage	6	0,000
	▼ | |- Nightmares.exe!GfxDeviceWorker::RunGfxDeviceWorker	1	4,004
	| | Nightmares.exe!UnityProfilerPerThread::Initialize	1	4,004
	| | Nightmares.exe!operator new	1	4,004
	| | Nightmares.exe!MemoryManager::Allocate	1	4,004
	| | Nightmares.exe!DualThreadAllocator<DynamicHeapAllocator<LowLevelAllocat...	1	4,004
	| | Nightmares.exe!DynamicHeapAllocator<LowLevelAllocator>::Allocate	1	4,004
	| | Nightmares.exe!MemoryManager::LowLevelAllocate	1	4,004
	| | Nightmares.exe!_aligned_malloc	1	4,004
	| | Nightmares.exe!_aligned_offset_malloc	1	4,004
	| | Nightmares.exe!malloc	1	4,004
	| | ntdll.dll!RtlAllocateHeap	1	4,004
	| | ntdll.dll!RtlpAllocateHeapInternal	1	4,004
	| | ntdll.dll!RtlpAllocateHeap	1	4,004

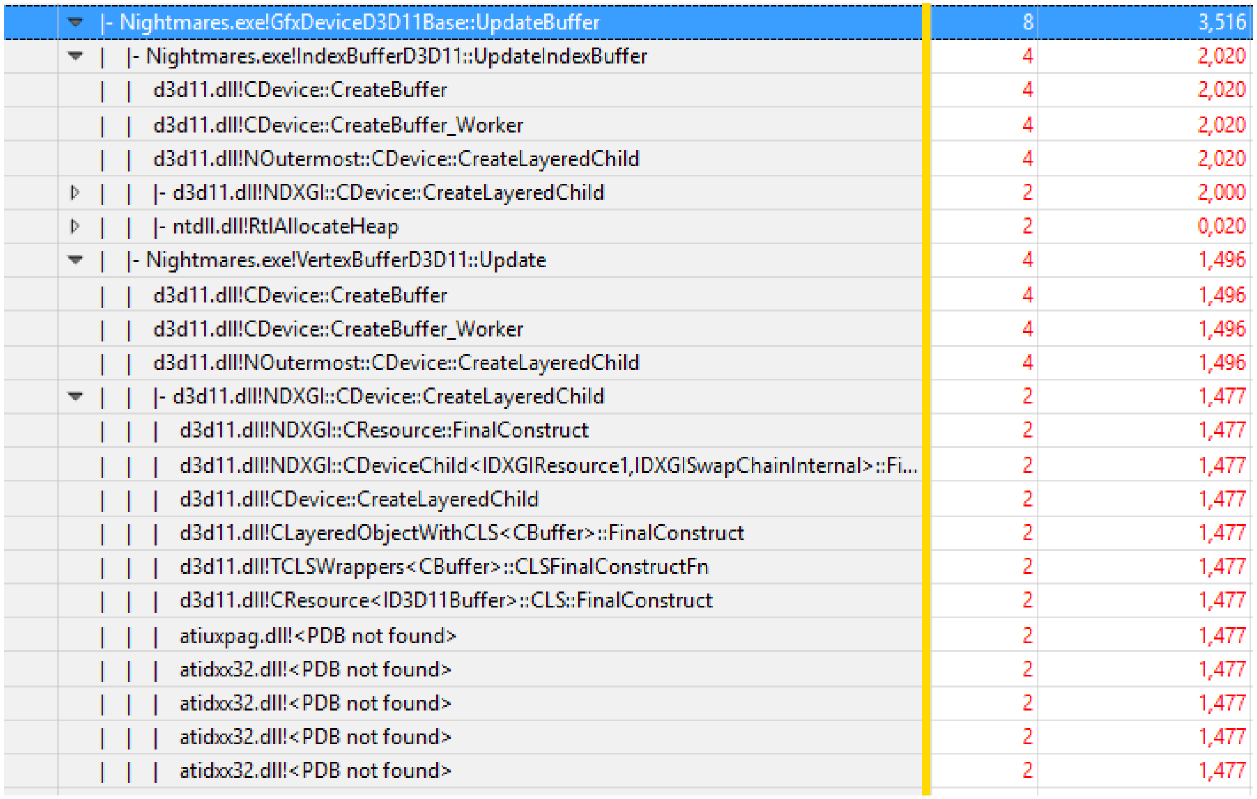
We can see that Unity allocates the same 12MB for the profiler in this thread. It also allocates 4MB through the MemoryManager - but only a small portion of this goes to the profiler.

The rest of the allocations on this thread are far more interesting. For example, graphics drivers/Direct3D 11 runtime allocate a total of 3.3 MB for shaders:



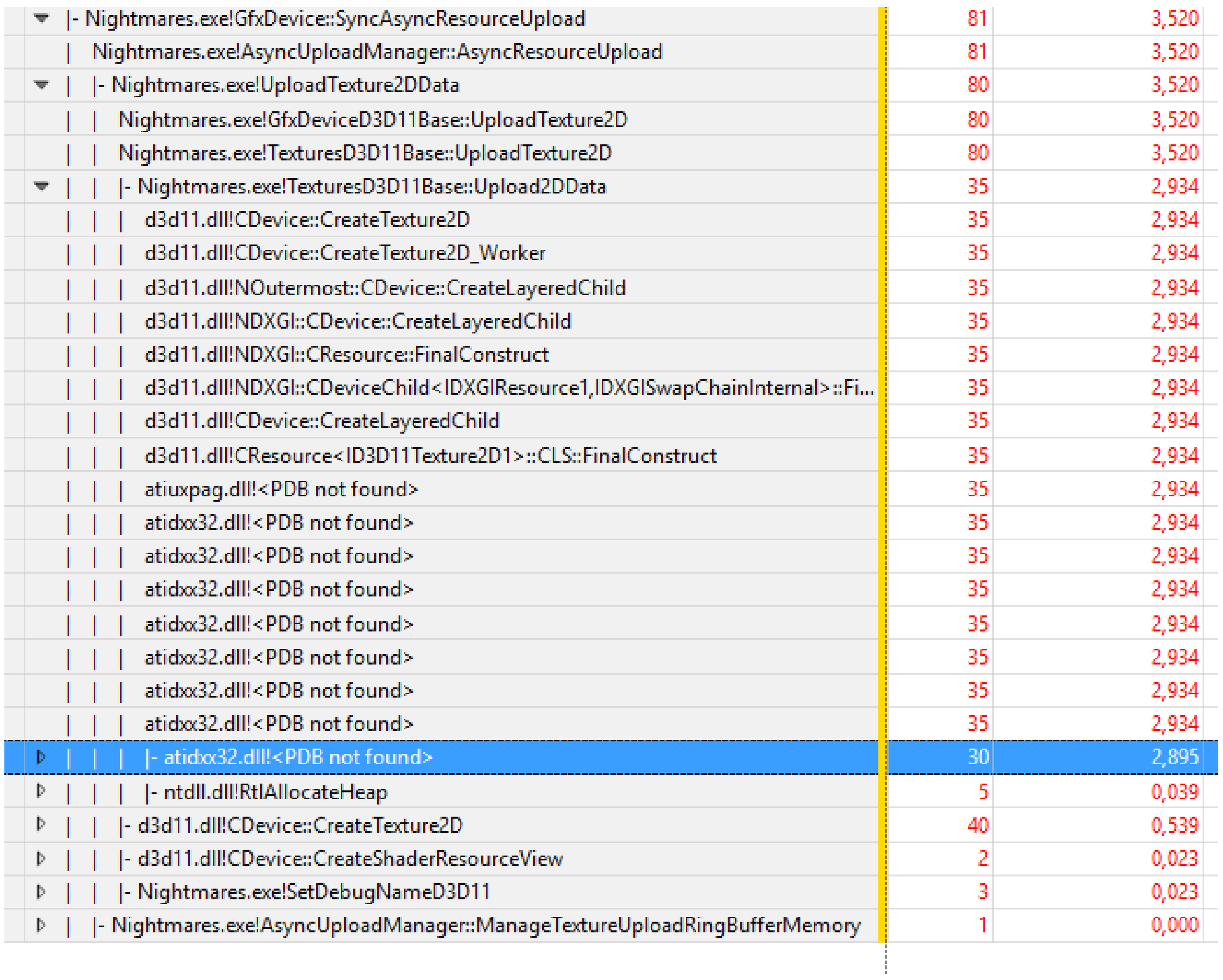
```
▾ | | |- Nightmares.exe!CreateGpuProgramQueue::DequeueAll                   282   3,371
  | | |  Nightmares.exe!GfxDevice::CreateGpuProgram                         282   3,371
  | | |  Nightmares.exe!CreateGpuProgram                                    282   3,371
▾ | | |  |- Nightmares.exe!D3D11PixelShader::D3D11PixelShader               161   2,543
  | | |  |  Nightmares.exe!D3D11PixelShader::Create                         161   2,543
  | | |  |  d3d11.dll!CDevice::CreatePixelShader                            161   2,543
  | | |  |  d3d11.dll!CDevice::CreatePixelShader_Worker                     161   2,543
  | | |  |  d3d11.dll!NOutermost::CDevice::CreateLayeredChild               161   2,543
▹ | | |  |  |- d3d11.dll!NDXGI::CDevice::CreateLayeredChild                156   2,520
▹ | | |  |  |- d3d11.dll!CDevice::GetLayeredChildSize                        5   0,023
▾ | | |  |- Nightmares.exe!D3D11VertexShader::D3D11VertexShader             121   0,828
  | | |  |  Nightmares.exe!D3D11VertexShader::Create                        121   0,828
▾ | | |  |  |- d3d11.dll!CDevice::CreateVertexShader                       118   0,816
  | | |  |  |  d3d11.dll!CDevice::CreateVertexShader_Worker                118   0,816
  | | |  |  |  d3d11.dll!NOutermost::CDevice::CreateLayeredChild          118   0,816
▹ | | |  |  |  |- d3d11.dll!NDXGI::CDevice::CreateLayeredChild           113   0,789
▹ | | |  |  |  |- d3d11.dll!CDevice::GetLayeredChildSize                   4   0,016
▹ | | |  |  |  |- ntdll.dll!RtlAllocateHeap                               1   0,012
▹ | | |  |  |- Nightmares.exe!SetDebugNameD3D11                             3   0,012
```

We can also see the memory used by vertex and index buffers:



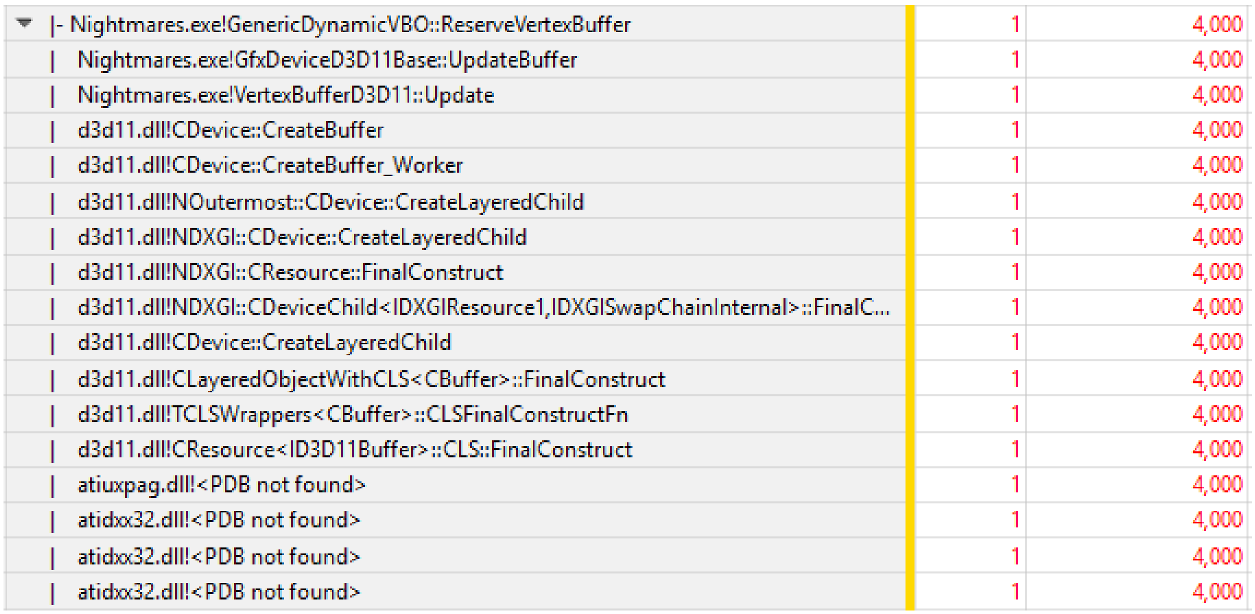
```
▾ |- Nightmares.exe!GfxDeviceD3D11Base::UpdateBuffer                             8   3,516
▾ |  |- Nightmares.exe!IndexBufferD3D11::UpdateIndexBuffer                        4   2,020
  |  |  d3d11.dll!CDevice::CreateBuffer                                          4   2,020
  |  |  d3d11.dll!CDevice::CreateBuffer_Worker                                   4   2,020
  |  |  d3d11.dll!NOutermost::CDevice::CreateLayeredChild                        4   2,020
▹ |  |  |- d3d11.dll!NDXGI::CDevice::CreateLayeredChild                          2   2,000
▹ |  |  |- ntdll.dll!RtlAllocateHeap                                             2   0,020
▾ |  |- Nightmares.exe!VertexBufferD3D11::Update                                 4   1,496
  |  |  d3d11.dll!CDevice::CreateBuffer                                          4   1,496
  |  |  d3d11.dll!CDevice::CreateBuffer_Worker                                   4   1,496
  |  |  d3d11.dll!NOutermost::CDevice::CreateLayeredChild                        4   1,496
▾ |  |  |- d3d11.dll!NDXGI::CDevice::CreateLayeredChild                          2   1,477
  |  |  |  d3d11.dll!NDXGI::CResource::FinalConstruct                           2   1,477
  |  |  |  d3d11.dll!NDXGI::CDeviceChild<IDXGIResource1,IDXGISwapChainInternal>::Fi...   2   1,477
  |  |  |  d3d11.dll!CDevice::CreateLayeredChild                                2   1,477
  |  |  |  d3d11.dll!CLayeredObjectWithCLS<CBuffer>::FinalConstruct             2   1,477
  |  |  |  d3d11.dll!TCLSWrappers<CBuffer>::CLSFinalConstructFn                  2   1,477
  |  |  |  d3d11.dll!CResource<ID3D11Buffer>::CLS::FinalConstruct               2   1,477
  |  |  |  atiuxpag.dll!<PDB not found>                                         2   1,477
  |  |  |  atidxx32.dll!<PDB not found>                                         2   1,477
  |  |  |  atidxx32.dll!<PDB not found>                                         2   1,477
  |  |  |  atidxx32.dll!<PDB not found>                                         2   1,477
  |  |  |  atidxx32.dll!<PDB not found>                                         2   1,477
```

Uploading textures generally doesn't allocate much memory if the device has dedicated VRAM. At 80 allocations for a total of 3.5 MB, graphics drivers/Direct3D 11 allocated an average of 45 KB per texture:

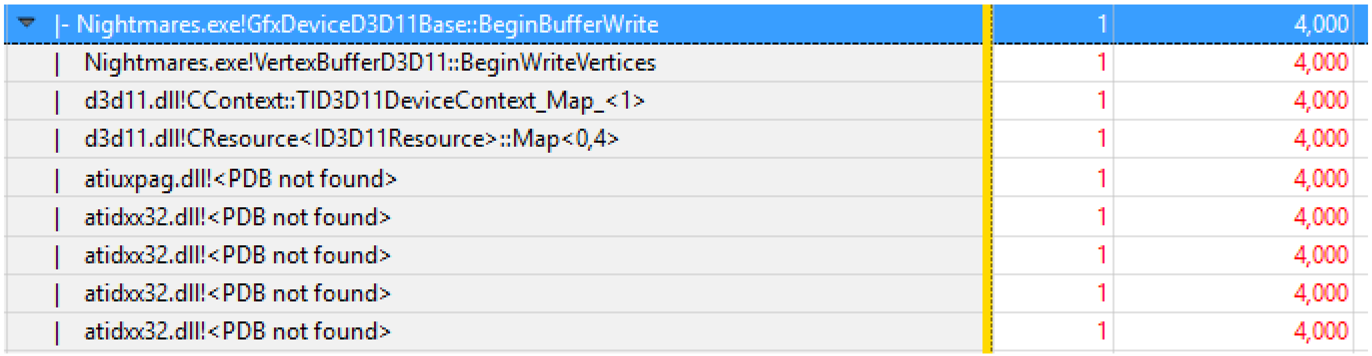


```
|- Nightmares.exe!GfxDevice::SyncAsyncResourceUpload                                   81   3,520
|  Nightmares.exe!AsyncUploadManager::AsyncResourceUpload                              81   3,520
|  |- Nightmares.exe!UploadTexture2DData                                               80   3,520
|  |  Nightmares.exe!GfxDeviceD3D11Base::UploadTexture2D                               80   3,520
|  |  Nightmares.exe!TexturesD3D11Base::UploadTexture2D                                80   3,520
|  |  |- Nightmares.exe!TexturesD3D11Base::Upload2DData                                35   2,934
|  |  |  d3d11.dll!CDevice::CreateTexture2D                                            35   2,934
|  |  |  d3d11.dll!CDevice::CreateTexture2D_Worker                                     35   2,934
|  |  |  d3d11.dll!NOutermost::CDevice::CreateLayeredChild                             35   2,934
|  |  |  d3d11.dll!NDXGI::CDevice::CreateLayeredChild                                  35   2,934
|  |  |  d3d11.dll!NDXGI::CResource::FinalConstruct                                    35   2,934
|  |  |  d3d11.dll!NDXGI::CDeviceChild<IDXGIResource1,IDXGISwapChainInternal>::Fi...   35   2,934
|  |  |  d3d11.dll!CDevice::CreateLayeredChild                                         35   2,934
|  |  |  d3d11.dll!CResource<ID3D11Texture2D1>::CLS::FinalConstruct                    35   2,934
|  |  |  atiuxpag.dll!<PDB not found>                                                  35   2,934
|  |  |  atidxx32.dll!<PDB not found>                                                  35   2,934
|  |  |  atidxx32.dll!<PDB not found>                                                  35   2,934
|  |  |  atidxx32.dll!<PDB not found>                                                  35   2,934
|  |  |  atidxx32.dll!<PDB not found>                                                  35   2,934
|  |  |  atidxx32.dll!<PDB not found>                                                  35   2,934
|  |  |  atidxx32.dll!<PDB not found>                                                  35   2,934
|  |  |  atidxx32.dll!<PDB not found>                                                  35   2,934
|  |  |  |- atidxx32.dll!<PDB not found>                                               30   2,895
|  |  |  |- ntdll.dll!RtlAllocateHeap                                                   5   0,039
|  |  |- d3d11.dll!CDevice::CreateTexture2D                                            40   0,539
|  |  |- d3d11.dll!CDevice::CreateShaderResourceView                                    2   0,023
|  |  |- Nightmares.exe!SetDebugNameD3D11                                               3   0,023
|  |- Nightmares.exe!AsyncUploadManager::ManageTextureUploadRingBufferMemory            1   0,000
```

Finally, 8MB is allocated for dynamic vertex buffers:



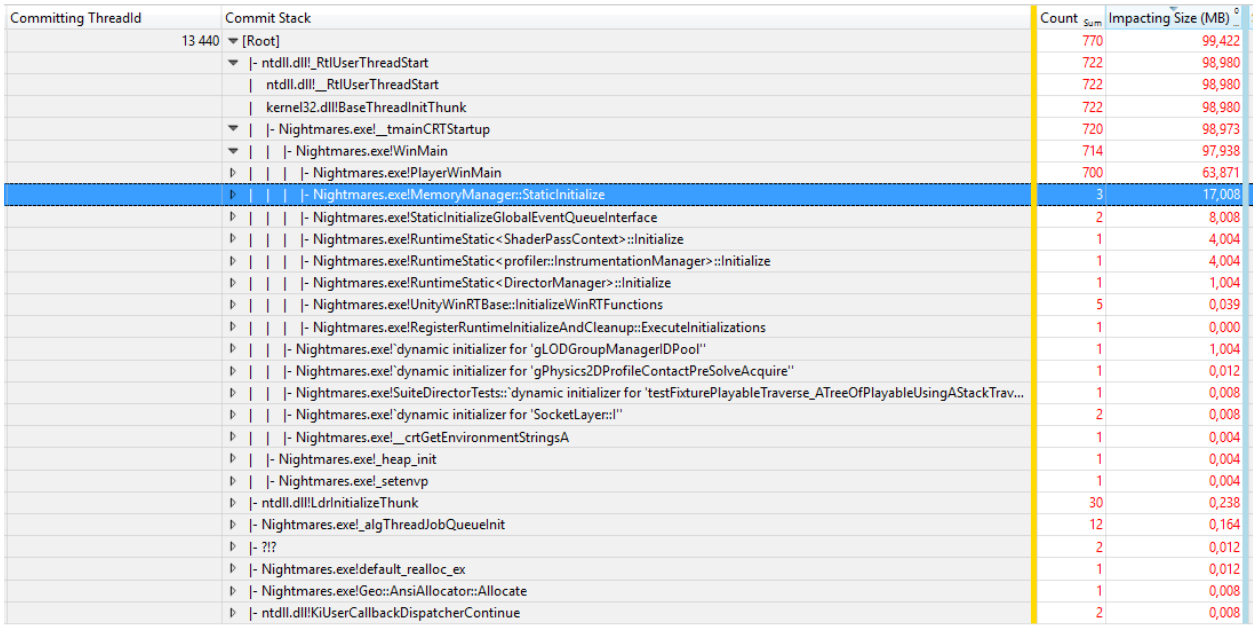
```
|- Nightmares.exe!GenericDynamicVBO::ReserveVertexBuffer                              1      4,000
|  Nightmares.exe!GfxDeviceD3D11Base::UpdateBuffer                                    1      4,000
|  Nightmares.exe!VertexBufferD3D11::Update                                           1      4,000
|  d3d11.dll!CDevice::CreateBuffer                                                    1      4,000
|  d3d11.dll!CDevice::CreateBuffer_Worker                                             1      4,000
|  d3d11.dll!NOutermost::CDevice::CreateLayeredChild                                  1      4,000
|  d3d11.dll!NDXGI::CDevice::CreateLayeredChild                                       1      4,000
|  d3d11.dll!NDXGI::CResource::FinalConstruct                                         1      4,000
|  d3d11.dll!NDXGI::CDeviceChild<IDXGIResource1,IDXGISwapChainInternal>::FinalC...   1      4,000
|  d3d11.dll!CDevice::CreateLayeredChild                                              1      4,000
|  d3d11.dll!CLayeredObjectWithCLS<CBuffer>::FinalConstruct                           1      4,000
|  d3d11.dll!TCLSWrappers<CBuffer>::CLSFinalConstructFn                               1      4,000
|  d3d11.dll!CResource<ID3D11Buffer>::CLS::FinalConstruct                             1      4,000
|  atiuxpag.dll!<PDB not found>                                                       1      4,000
|  atidxx32.dll!<PDB not found>                                                       1      4,000
|  atidxx32.dll!<PDB not found>                                                       1      4,000
|  atidxx32.dll!<PDB not found>                                                       1      4,000
```



```
|- Nightmares.exe!GfxDeviceD3D11Base::BeginBufferWrite              1      4,000
|  Nightmares.exe!VertexBufferD3D11::BeginWriteVertices             1      4,000
|  d3d11.dll!CContext::TID3D11DeviceContext_Map_<1>                 1      4,000
|  d3d11.dll!CResource<ID3D11Resource>::Map<0,4>                    1      4,000
|  atiuxpag.dll!<PDB not found>                                     1      4,000
|  atidxx32.dll!<PDB not found>                                     1      4,000
|  atidxx32.dll!<PDB not found>                                     1      4,000
|  atidxx32.dll!<PDB not found>                                     1      4,000
|  atidxx32.dll!<PDB not found>                                     1      4,000
```

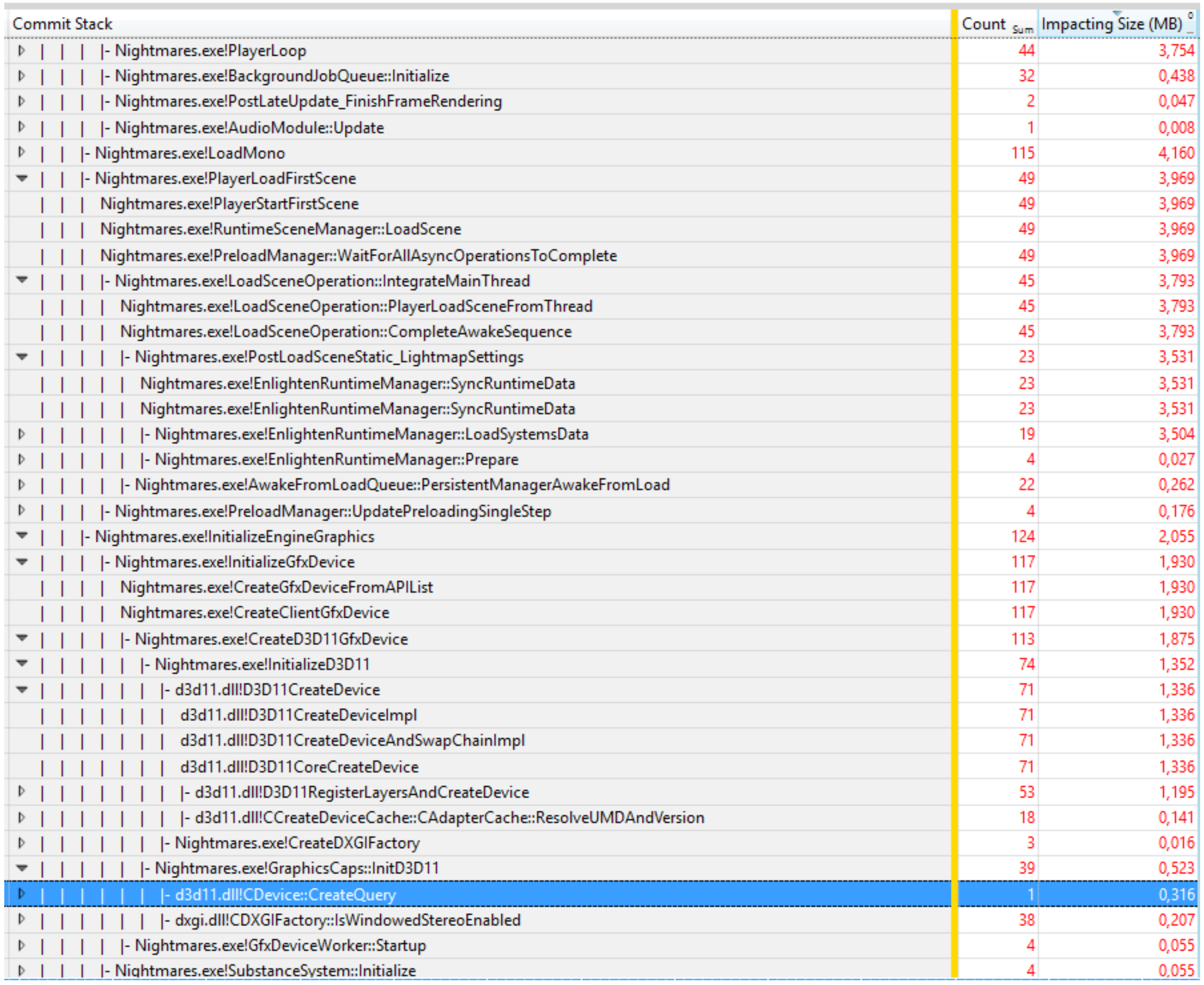
Those were all of the outstanding allocations that the rendering thread did in our game.

Now let’s move to Unity’s main thread:



```
Committing ThreadId   Commit Stack                                                                                                    Count Sum   Impacting Size (MB)
             13 440   [Root]                                                                                                                770        99,422
                        |- ntdll.dll!_RtlUserThreadStart                                                                                    722        98,980
                        |   ntdll.dll!__RtlUserThreadStart                                                                                  722        98,980
                        |   kernel32.dll!BaseThreadInitThunk                                                                                722        98,980
                        |   |- Nightmares.exe!__tmainCRTStartup                                                                             720        98,973
                        |   |   |- Nightmares.exe!WinMain                                                                                   714        97,938
                        |   |   |   |- Nightmares.exe!PlayerWinMain                                                                         700        63,871
                        |   |   |   |- Nightmares.exe!MemoryManager::StaticInitialize                                                         3        17,008
                        |   |   |   |- Nightmares.exe!StaticInitializeGlobalEventQueueInterface                                               2         8,008
                        |   |   |   |- Nightmares.exe!RuntimeStatic<ShaderPassContext>::Initialize                                            1         4,004
                        |   |   |   |- Nightmares.exe!RuntimeStatic<profiler::InstrumentationManager>::Initialize                             1         4,004
                        |   |   |   |- Nightmares.exe!RuntimeStatic<DirectorManager>::Initialize                                              1         1,004
                        |   |   |   |- Nightmares.exe!UnityWinRTBase::InitializeWinRTFunctions                                                5         0,039
                        |   |   |   |- Nightmares.exe!RegisterRuntimeInitializeAndCleanup::ExecuteInitializations                             1         0,000
                        |   |   |- Nightmares.exe!`dynamic initializer for 'gLODGroupManagerIDPool''                                          1         1,004
                        |   |   |- Nightmares.exe!`dynamic initializer for 'gPhysics2DProfileContactPreSolveAcquire''                         1         0,012
                        |   |   |- Nightmares.exe!SuiteDirectorTests::`dynamic initializer for 'testFixturePlayableTraverse_ATreeOfPlayableUsingAStackTrav...   1   0,008
                        |   |   |- Nightmares.exe!`dynamic initializer for 'SocketLayer::l''                                                  2         0,008
                        |   |   |- Nightmares.exe!__crtGetEnvironmentStringsA                                                                 1         0,004
                        |   |- Nightmares.exe!_heap_init                                                                                      1         0,004
                        |   |- Nightmares.exe!_setenvp                                                                                        1         0,004
                        |- ntdll.dll!LdrInitializeThunk                                                                                      30         0,238
                        |- Nightmares.exe!_algThreadJobQueueInit                                                                             12         0,164
                        |- ?!?                                                                                                                2         0,012
                        |- Nightmares.exe!default_realloc_ex                                                                                  1         0,012
                        |- Nightmares.exe!Geo::AnsiAllocator::Allocate                                                                        1         0,008
                        |- ntdll.dll!KiUserCallbackDispatcherContinue                                                                         2         0,008
```

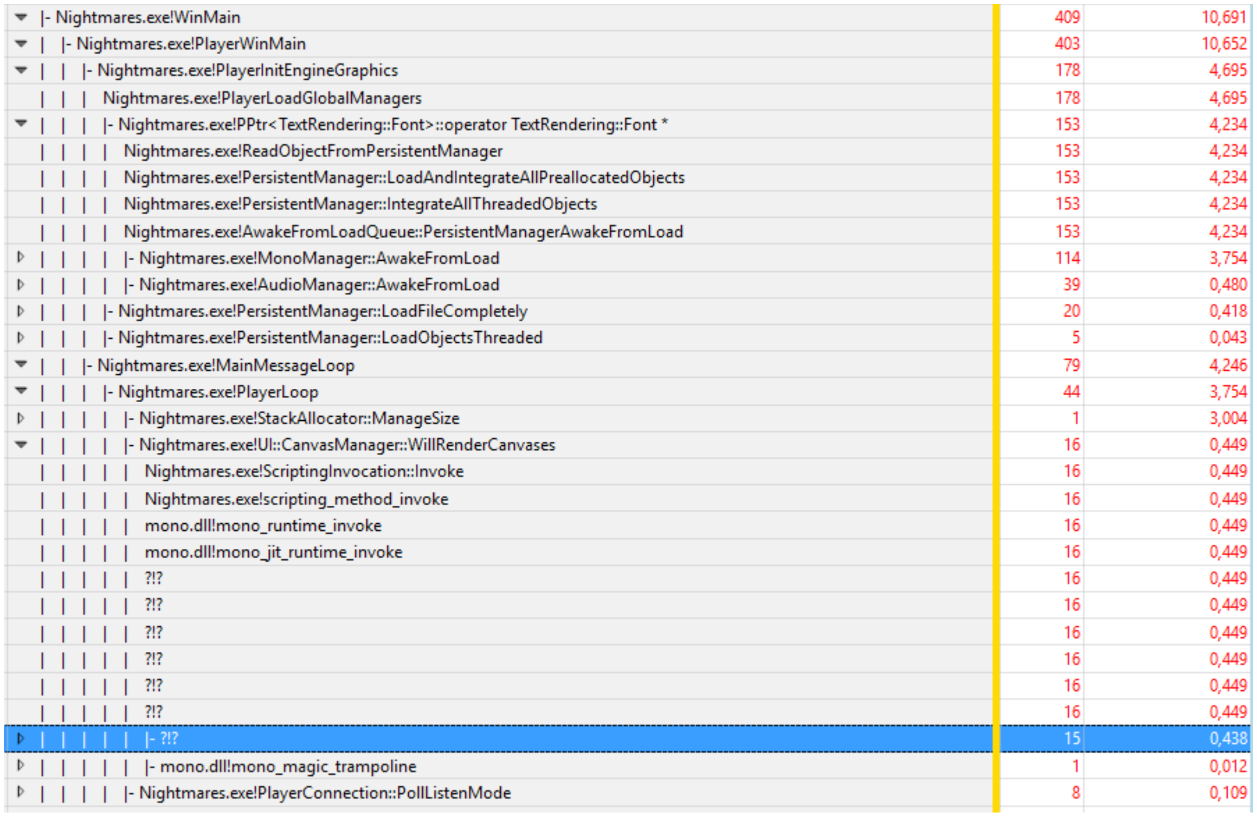
Most of the allocations made on Unity’s main thread will go through the MemoryManager. We don’t need to pay attention to these, as we can see all of their details in Unity’s internal memory profiler. Let’s filter these allocations out:



```
Commit Stack                                                                                      Count Sum   Impacting Size (MB)
▷ | | | |- Nightmares.exe!PlayerLoop                                                                     44    3,754
▷ | | | |- Nightmares.exe!BackgroundJobQueue::Initialize                                                 32    0,438
▷ | | | |- Nightmares.exe!PostLateUpdate_FinishFrameRendering                                           2    0,047
▷ | | | |- Nightmares.exe!AudioModule::Update                                                            1    0,008
▷ | | |- Nightmares.exe!LoadMono                                                                        115    4,160
▼ | | |- Nightmares.exe!PlayerLoadFirstScene                                                            49    3,969
  | | |  Nightmares.exe!PlayerStartFirstScene                                                           49    3,969
  | | |  Nightmares.exe!RuntimeSceneManager::LoadScene                                                  49    3,969
  | | |  Nightmares.exe!PreloadManager::WaitForAllAsyncOperationsToComplete                             49    3,969
▼ | | | |- Nightmares.exe!LoadSceneOperation::IntegrateMainThread                                        45    3,793
  | | | |  Nightmares.exe!LoadSceneOperation::PlayerLoadSceneFromThread                                  45    3,793
  | | | |  Nightmares.exe!LoadSceneOperation::CompleteAwakeSequence                                      45    3,793
▼ | | | | |- Nightmares.exe!PostLoadSceneStatic_LightmapSettings                                       23    3,531
  | | | | |  Nightmares.exe!EnlightenRuntimeManager::SyncRuntimeData                                     23    3,531
  | | | | |  Nightmares.exe!EnlightenRuntimeManager::SyncRuntimeData                                     23    3,531
▷ | | | | | |- Nightmares.exe!EnlightenRuntimeManager::LoadSystemsData                                 19    3,504
▷ | | | | | |- Nightmares.exe!EnlightenRuntimeManager::Prepare                                          4    0,027
▷ | | | | |- Nightmares.exe!AwakeFromLoadQueue::PersistentManagerAwakeFromLoad                         22    0,262
▷ | | | |- Nightmares.exe!PreloadManager::UpdatePreloadingSingleStep                                     4    0,176
▼ | | |- Nightmares.exe!InitializeEngineGraphics                                                       124    2,055
▼ | | | |- Nightmares.exe!InitializeGfxDevice                                                          117    1,930
  | | | |  Nightmares.exe!CreateGfxDeviceFromAPIList                                                   117    1,930
  | | | |  Nightmares.exe!CreateClientGfxDevice                                                       117    1,930
▼ | | | | |- Nightmares.exe!CreateD3D11GfxDevice                                                      113    1,875
▼ | | | | | |- Nightmares.exe!InitializeD3D11                                                          74    1,352
▼ | | | | | | |- d3d11.dll!D3D11CreateDevice                                                          71    1,336
  | | | | | | |  d3d11.dll!D3D11CreateDeviceImpl                                                      71    1,336
  | | | | | | |  d3d11.dll!D3D11CreateDeviceAndSwapChainImpl                                         71    1,336
  | | | | | | |  d3d11.dll!D3D11CoreCreateDevice                                                      71    1,336
▷ | | | | | | | |- d3d11.dll!D3D11RegisterLayersAndCreateDevice                                     53    1,195
▷ | | | | | | | |- d3d11.dll!CCreateDeviceCache::CAdapterCache::ResolveUMDAndVersion               18    0,141
▷ | | | | | | |- Nightmares.exe!CreateDXGIFactory                                                    3    0,016
▼ | | | | | |- Nightmares.exe!GraphicsCaps::InitD3D11                                                39    0,523
▷ | | | | | | |- d3d11.dll!CDevice::CreateQuery                                                      1    0,316
▷ | | | | | | |- dxgi.dll!CDXGIFactory::IsWindowedStereoEnabled                                      38    0,207
▷ | | | | |- Nightmares.exe!GfxDeviceWorker::Startup                                                  4    0,055
▷ | | | |- Nightmares.exe!SubstanceSystem::Initialize                                                  4    0,055
```

1.3MB of memory is allocated by creating a Direct3D 11 device; 300 KB of memory is allocated by creating a Direct3D 11 query; 200KB of memory is allocated because Unity asked DXGIFactory whether it supports stereoscopic rendering; 3.5 MB of memory is allocated by Enlighten global illumination initialization; 4MB is allocated by loading Mono.

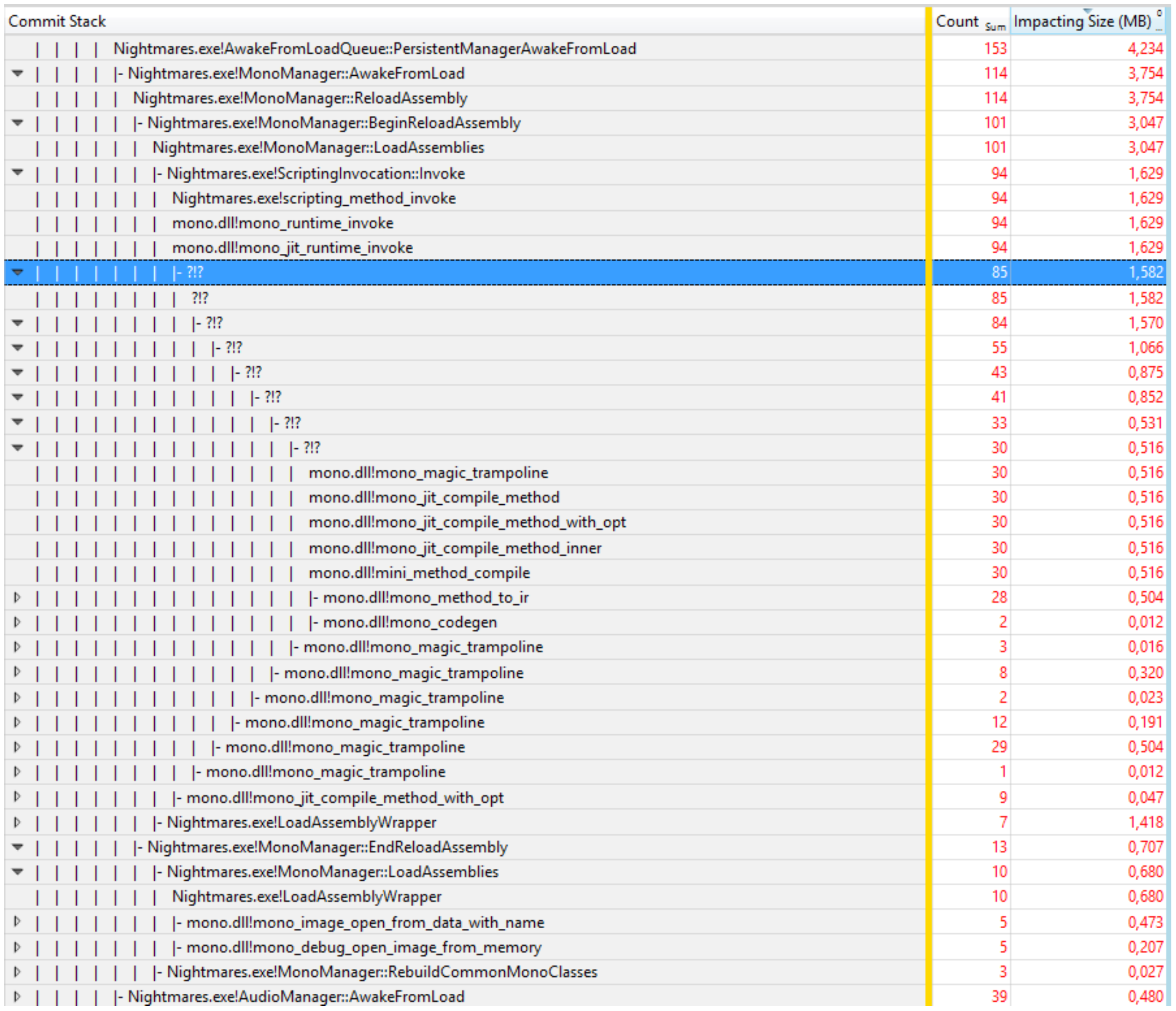
Let’s look at the rest:



```
|- Nightmares.exe!WinMain                                                   409   10,691
|  |- Nightmares.exe!PlayerWinMain                                          403   10,652
|  |  |- Nightmares.exe!PlayerInitEngineGraphics                            178    4,695
|  |  |  Nightmares.exe!PlayerLoadGlobalManagers                            178    4,695
|  |  |  |- Nightmares.exe!PPtr<TextRendering::Font>::operator TextRendering::Font *   153    4,234
|  |  |  |  Nightmares.exe!ReadObjectFromPersistentManager                  153    4,234
|  |  |  |  Nightmares.exe!PersistentManager::LoadAndIntegrateAllPreallocatedObjects   153    4,234
|  |  |  |  Nightmares.exe!PersistentManager::IntegrateAllThreadedObjects    153    4,234
|  |  |  |  Nightmares.exe!AwakeFromLoadQueue::PersistentManagerAwakeFromLoad   153    4,234
|  |  |  |  |- Nightmares.exe!MonoManager::AwakeFromLoad                    114    3,754
|  |  |  |  |- Nightmares.exe!AudioManager::AwakeFromLoad                    39    0,480
|  |  |  |- Nightmares.exe!PersistentManager::LoadFileCompletely             20    0,418
|  |  |  |- Nightmares.exe!PersistentManager::LoadObjectsThreaded              5    0,043
|  |  |- Nightmares.exe!MainMessageLoop                                      79    4,246
|  |  |  |- Nightmares.exe!PlayerLoop                                        44    3,754
|  |  |  |  |- Nightmares.exe!StackAllocator::ManageSize                      1    3,004
|  |  |  |  |- Nightmares.exe!UI::CanvasManager::WillRenderCanvases          16    0,449
|  |  |  |  |  Nightmares.exe!ScriptingInvocation::Invoke                    16    0,449
|  |  |  |  |  Nightmares.exe!scripting_method_invoke                        16    0,449
|  |  |  |  |  mono.dll!mono_runtime_invoke                                 16    0,449
|  |  |  |  |  mono.dll!mono_jit_runtime_invoke                             16    0,449
|  |  |  |  |  ?!?                                                          16    0,449
|  |  |  |  |  ?!?                                                          16    0,449
|  |  |  |  |  ?!?                                                          16    0,449
|  |  |  |  |  ?!?                                                          16    0,449
|  |  |  |  |  ?!?                                                          16    0,449
|  |  |  |  |  ?!?                                                          16    0,449
|  |  |  |  |  |- ?!?                                                       15    0,438
|  |  |  |  |  |- mono.dll!mono_magic_trampoline                             1    0,012
|  |  |  |  |- Nightmares.exe!PlayerConnection::PollListenMode               8    0,109
```

Almost 500KB is allocated by scripting code; 3MB goes to Unity's main thread stack allocator (Unity uses it for various temporary data storage); almost 500KB is used by audio initialization, and the rest (around 3.7 MB) is used by **MonoManager::AwakeFromLoad**.

Let's take a look inside **MonoManager::AwakeFromLoad** and see what it does:



```
Commit Stack                                                                          Count Sum  Impacting Size (MB)
  | | | | Nightmares.exe!AwakeFromLoadQueue::PersistentManagerAwakeFromLoad                153      4,234
▼ | | | | |- Nightmares.exe!MonoManager::AwakeFromLoad                                     114      3,754
  | | | | | Nightmares.exe!MonoManager::ReloadAssembly                                      114      3,754
▼ | | | | | |- Nightmares.exe!MonoManager::BeginReloadAssembly                              101      3,047
  | | | | | | Nightmares.exe!MonoManager::LoadAssemblies                                     101      3,047
▼ | | | | | | |- Nightmares.exe!ScriptingInvocation::Invoke                                 94      1,629
  | | | | | | | Nightmares.exe!scripting_method_invoke                                        94      1,629
  | | | | | | | mono.dll!mono_runtime_invoke                                                  94      1,629
  | | | | | | | mono.dll!mono_jit_runtime_invoke                                              94      1,629
▼ | | | | | | | |- ?!?                                                                    85      1,582
  | | | | | | | | ?!?                                                                     85      1,582
▼ | | | | | | | | |- ?!?                                                                  84      1,570
▼ | | | | | | | | | |- ?!?                                                                55      1,066
▼ | | | | | | | | | | |- ?!?                                                              43      0,875
▼ | | | | | | | | | | | |- ?!?                                                            41      0,852
▼ | | | | | | | | | | | | |- ?!?                                                          33      0,531
▼ | | | | | | | | | | | | | |- ?!?                                                        30      0,516
  | | | | | | | | | | | | | | mono.dll!mono_magic_trampoline                              30      0,516
  | | | | | | | | | | | | | | mono.dll!mono_jit_compile_method                             30      0,516
  | | | | | | | | | | | | | | mono.dll!mono_jit_compile_method_with_opt                    30      0,516
  | | | | | | | | | | | | | | mono.dll!mono_jit_compile_method_inner                       30      0,516
  | | | | | | | | | | | | | | mono.dll!mini_method_compile                                 30      0,516
▷ | | | | | | | | | | | | | | |- mono.dll!mono_method_to_ir                              28      0,504
▷ | | | | | | | | | | | | | | |- mono.dll!mono_codegen                                    2      0,012
▷ | | | | | | | | | | | | | |- mono.dll!mono_magic_trampoline                            3      0,016
▷ | | | | | | | | | | | | |- mono.dll!mono_magic_trampoline                              8      0,320
▷ | | | | | | | | | | | |- mono.dll!mono_magic_trampoline                                2      0,023
▷ | | | | | | | | | | |- mono.dll!mono_magic_trampoline                                 12      0,191
▷ | | | | | | | | | |- mono.dll!mono_magic_trampoline                                   29      0,504
▷ | | | | | | | | |- mono.dll!mono_magic_trampoline                                      1      0,012
▷ | | | | | | | |- mono.dll!mono_jit_compile_method_with_opt                              9      0,047
▷ | | | | | | |- Nightmares.exe!LoadAssemblyWrapper                                       7      1,418
▼ | | | | | |- Nightmares.exe!MonoManager::EndReloadAssembly                              13      0,707
▼ | | | | | | |- Nightmares.exe!MonoManager::LoadAssemblies                               10      0,680
  | | | | | | | Nightmares.exe!LoadAssemblyWrapper                                          10      0,680
▷ | | | | | | | |- mono.dll!mono_image_open_from_data_with_name                            5      0,473
▷ | | | | | | | |- mono.dll!mono_debug_open_image_from_memory                             5      0,207
▷ | | | | | | |- Nightmares.exe!MonoManager::RebuildCommonMonoClasses                      3      0,027
▷ | | | | |- Nightmares.exe!AudioManager::AwakeFromLoad                                    39      0,480
```

Those 3.7MB consist of 1.6 MB for code JIT-ing and 2.1MB for managed assembly loading.

The more you know about your application's memory allocation, the easier it will be to solve any related problems that are occurring. An application that is unexpectedly using very large amounts of memory can be analysed in this way to narrow down where the problem might be occurring.