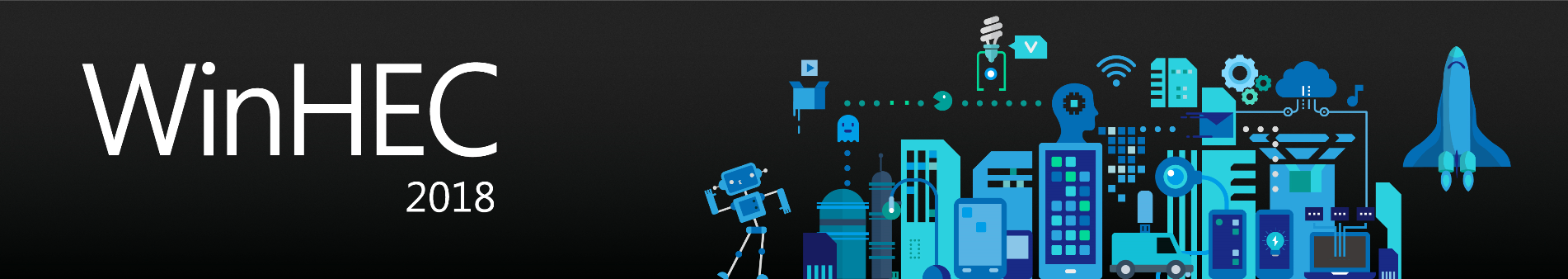
WinHEC

 Hands-on Lab

Optimizing Performance and Responsiveness for Windows Desktop

***Abstract*:** In this lab You will learn the process of measuring, analyzing, and solving performance issues the Windows Performance Toolkit and the Windows Assessment Toolkit.

This lab will allow you to achieve the following goals:

1. Gather relevant data to analyze performance problems on any system
2. Understand the analysis process to look at system resource consumption like CPU and disk
3. Identify what can impact the system responsiveness in some key Windows scenarios

Copyright (c) 2018 Microsoft. All rights reserved.

This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice.

Contents

Prerequisites 4

Exercise 1 – Evaluate Fast Startup using the Assessment Toolkit 5

Step 1: Collect data using the Assessment Toolkit 5

Step 2: Visualize the assessment results using WAC 6

Step 3: Review the Fast Startup report 7

Step 4: Examine the Resume Devices Duration metrics 8

Step 5: Determine the hiberfile size 9

Step 6: Examine the Post On/Off Duration value 10

Step 7: Open Fast Startup trace with WPA 11

Exercise 2 – Evaluate Fast Startup using WPT 12

Step 1: Open Fast Startup trace using WPA 12

Step 2: Open Fast Startup trace using WPA 12

Step 3: Visualize the activity timeline 13

Step 4: Analyze process CPU usage 15

Step 5: Analyze process disk usage 16

Exercise 3 – Understand critical path and wait analysis 19

Step 1: Capture and open a trace for a UI delay problem 20

Step 2: Identify the delayed UI thread 21

Step 3: Analyze the UI delay critical path 22

Annex I - Technical background 26

Tools 26

Fast Startup behavior 26

CPU Scheduling and Threads 27

# Prerequisites

Assessment and Deployment Toolkit can be found on MSDN.   
Windows 10 ADK (1809): <https://developer.microsoft.com/en-us/windows/hardware/windows-assessment-deployment-kit>

Make sure you have the pre-generated results and test tools on your computer (located in **C:\Performance**)

* FastStartup\_Analysis\_1.etl
* FastStartup\_Baseline.xml
* FastStartup\_OEMPreload.xml
* UIDelay.exe
* UIDelay.pdb
* UIDelay.wprp

Additional technical reference and background can be found in the Annex I at the end of this document

# Exercise 1 – Evaluate Fast Startup using the Assessment Toolkit

**Estimated Time**: 15 minutes

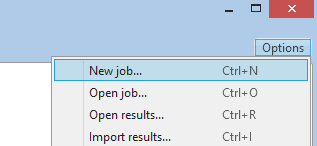
Boot time is a common benchmark that users use to measure Windows performance. Over the lifetime of their systems, longer boot times can be an indicator of system problems such as inefficient configuration, device conflicts, and malware.

## Step 1: Collect data using the Assessment Toolkit

**Note**: You can skip this step in the WinHEC lab, as the following steps use pre-generated results.

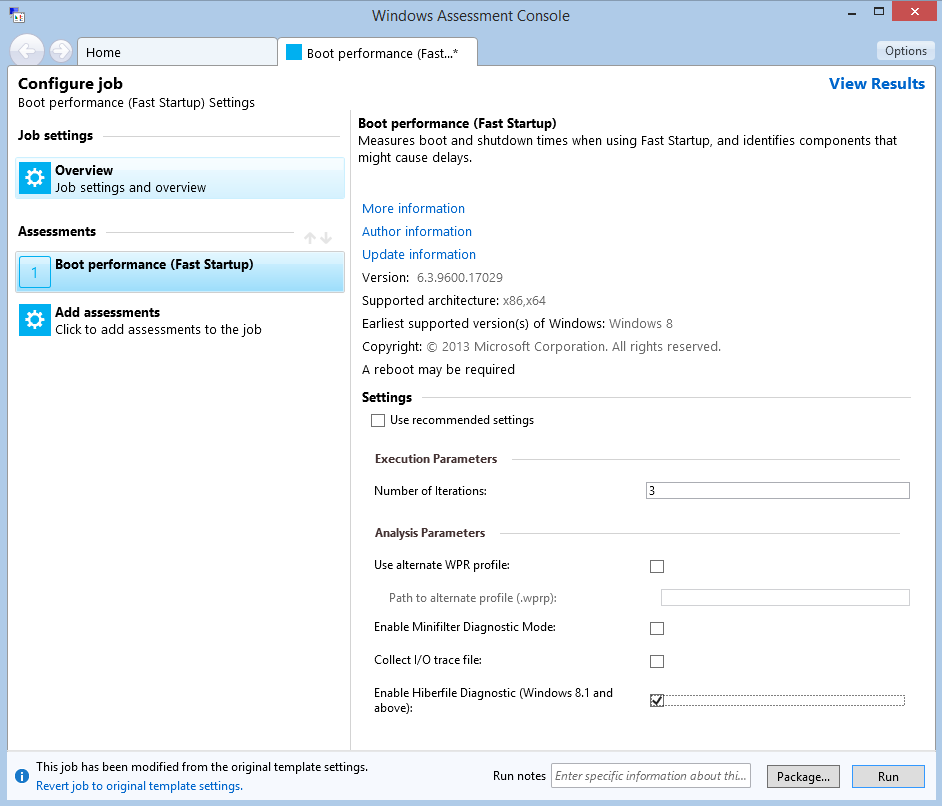
The Windows Assessment Toolkit contains a test to measure the Fast Startup time. You can use this assessment to understand the impact drivers, devices, and the software preload have on the Fast Startup time. Fast Startup time can be negatively affected by processes and services that are loaded in memory at startup, processes and services that run in the background, or resources that are used to initialize devices.

1. Open **Windows Assessment Console (WAC)** from the Start menu.
2. Open the **Options** menu and select **New Job…**



* 1. Enter **FastStartupTest** as the job name.
  2. Select **Create a custom job.**

1. Click on **Add Assessments.**
   1. Add the **Boot Performance (Fast Startup)** assessment by clicking on the “+” symbol
2. Click on the newly added **Boot Performance (Fast Startup)** assessment to enter the test configuration.
3. Unselect **Use recommended settings** and select **Enable Hiberfile Diagnostics** for the configuration.
   1. **Enable Hiberfile Diagnostics** allows you to analyze the contents of the hiberfile and identify the memory pages that contribute to its size.
4. You have two options:
   1. **Package** the job in order to create a folder with all the test resources and copy it on another test system (Click on the **Package…** button in the bottom right corner.).
   2. **Run** the job directly on the system by clicking on the **Run** button in the bottom right corner.
      1. This restarts the system to gather a trace.
      2. This test can take 30 minutes to complete.



## 

## Step 2: Visualize the assessment results using WAC

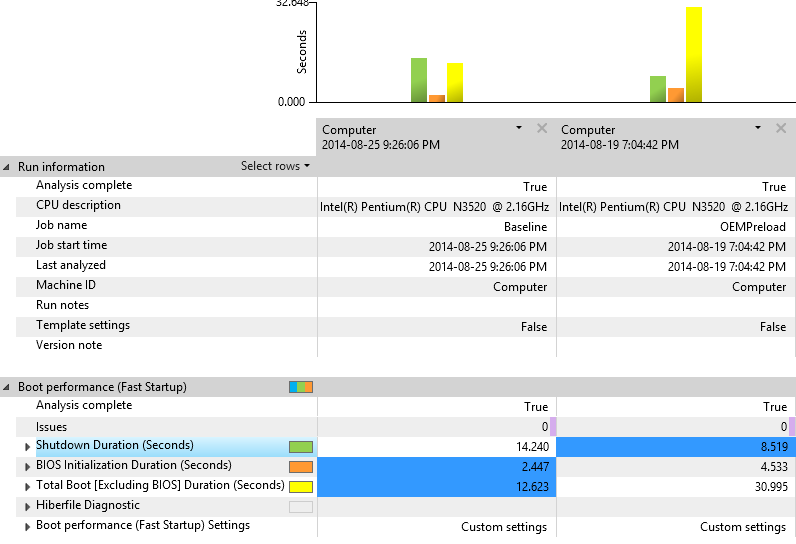
Once the assessment execution is completed, you can open the results XML report with **WAC** and start evaluating the **Fast Startup** times.

This step uses pre-generated XML reports:

* **Baseline report** (FastStartup\_Baseline.xml): The Fast Startup assessment was executed on a clean Windows retail image with a full set of drivers. A baseline enables you to understand the best case scenario for a system without any added 3rd party apps.
* **OEM image report** (FastStartup\_OEMPreload.xml): The Fast Startup assessment was executed, on the same system, but with an OEM image (also known as retail image). The delta between the baseline report and this report allows to quantify the impact that added apps have on boot times.

1. In **WAC**, in the upper-right corner, open the **Options** menu and select **Open Results…**
   1. You can also press **CTRL+R** on the keyboard.
2. Click on the **Browse…** button and navigate to **C:\Performance**
3. Select both **FastStartup\_Baseline.xml** and **FastStartup\_OEMPreload.xml** at the same time, and click Open.

The two results open side-by-side in the Windows Assessment Console as shown:



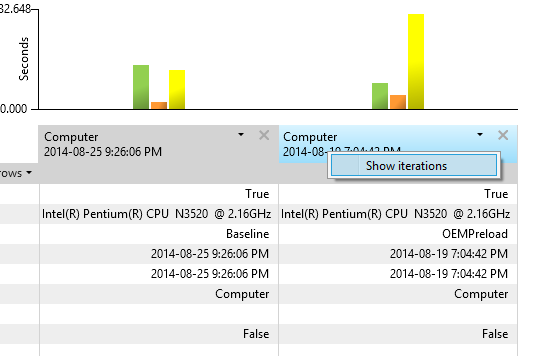
## Step 3: Review the Fast Startup report

The assessment results section provides the data you will use to understand how a system is performing and to identify issues. Most metric values are numbers you can use to compare against other metrics or computers.

|  |  |
| --- | --- |
| Phase | Description |
| **Shutdown Duration (seconds)** | The time the computer takes to shut down. You can expand this node to expose additional metrics for deeper understanding and investigation. |
| **BIOS Initialization Duration (seconds)** | The time the BIOS takes to initialize. The assessment does not offer analysis and remediation information for this metric. |
| **Total Boot (seconds)** | The time the computer takes to boot *after* the BIOS phase has completed. You can expand this node to expose additional metrics for deeper understanding and investigation. |

**User Tip**

**Assessments** often run workloads or scenarios multiple times. We refer to each of these runs as an “iteration,” and the values collected are averaged across multiple iterations. For example, by default, **Fast Startup** has three iterations. To view the individual iteration’s values (for example, to identify whether there was a variance in any of the iterations), right-click the computer name in the top column heading and then select **Show Iterations**.



The Boot Performance (Fast Startup) assessment provides boot metrics in a number of phases and components.

1. In the **WAC**, find the **Total Boot [Excluding BIOS] Duration (Seconds)** metric, and compare the baseline and test results. You should notice a large (above 18 seconds) regression time between the two.
2. Click the chevron () next to this metric to show the sub-metrics.
   * **Main Path Boot Duration:** Shows the time Windows takes to resume from the end of BIOS initialization to when the desktop is visible to the user.
   * **Post On/Off Duration:** Shows the time Windows takes to complete all startup tasks after the desktop appeared.

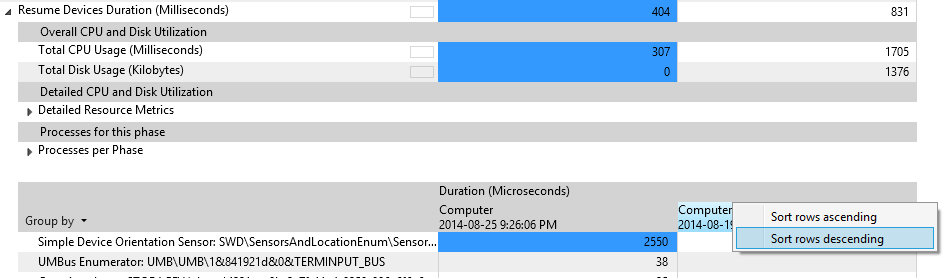
The other metric areas (**Resource Consumption** and **Resume Process Details**) provide CPU and disk usage data and will not be investigated within this lab.



## Step 4: Examine the Resume Devices Duration metrics

Device drivers may become a source of boot delays. To identify drivers that have issues, drill down into the **Resume Devices Duration** metrics to find problems.

1. Click the chevron next to the **Main Path Boot Duration** to expand it.
2. Find the **Resume Devices Duration** metric, expand the node by clicking on the chevron, and then view the sub-metrics under the **Processes Per Phase** metric.
3. Right-click the computer name column header of the test results column and then select **Sort Descending**. This sorts the data so that the largest numbers are at the top. This allows to focus on the tasks that have the longest duration.
4. Each row represent the amount of time a device took to resume to an active power state.



## 

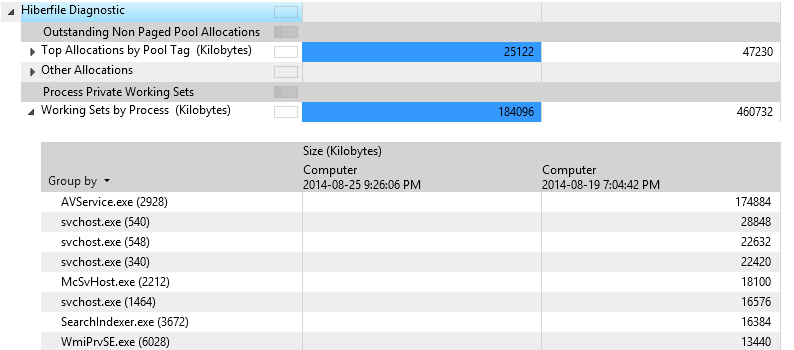
## Step 5: Determine the hiberfile size

1. Click the chevron next to the **Hiberfile Read Duration** to expand it.

The **Hiberfile size** metric represents the amount of data read from disk to restore the system context through the hibernation stack.

* + The larger the size of the file, the longer it takes for the system to boot. The size of the file is directly impacted by memory usage from services and drivers.
  + To get an estimate of the disk read throughput (MB/s), you can divide **Hiberfile size** by the **Hiberfile Read Duration** metric. If there’s a significant discrepancy between this throughput and the drive’s specification, this might indicate a problem with the driver or the BIOS storage read routines.

1. To analyze the content of the hiberfile and determine what software components contribute to its size, expand the **Hiberfile Diagnostic** metric. Two types of memory contribute to the hiberfile size:
   * Driver non-paged pool memory
   * Process working sets



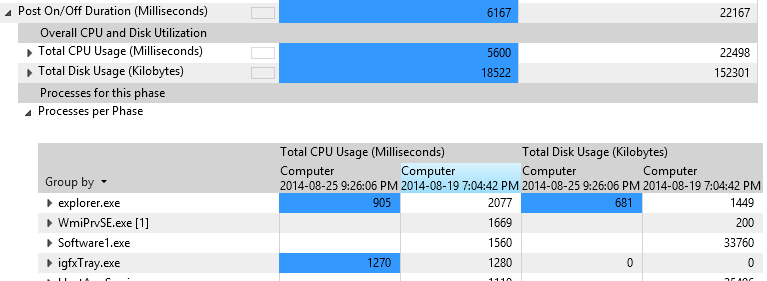
## 

## Step 6: Examine the Post On/Off Duration value

The **Post On/Off Duration** value represents the time it takes for the computer to reach an idle state after the desktop is displayed to the user. During this time, user responsiveness can be affected because system startup is completing in the background. The **Post On/Off** process completes after five seconds of low priority CPU and storage usage are accumulated.

The difference in the **Post On/Off Duration** value between the baseline and test results is 16 seconds, so it deserves further investigation.

1. Expand the **Post On/Off Duration (Milliseconds)** node.
2. Expand the **Processes Per Phase** node to display a sub-table with metrics on the individual apps and services that use CPU and storage during this phase.
3. Right-click the last column heading of that table, and then select **Sort descending**.



You can now identify processes that contribute to the phase duration. The more resources the process consumes, the more likely it affects the phase duration and should be investigated further.

## Step 7: Open Fast Startup trace with WPA

**Note**: You can skip this step in the WinHEC lab. This is for reference only.

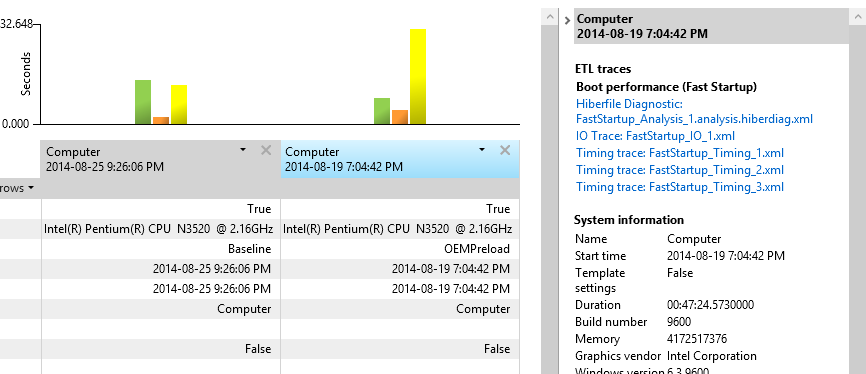
The **Fast Startup** assessment generates three types of iterations traces:

|  |  |
| --- | --- |
| Iteration type | Description |
| Training | The assessment reboots the system six times to make sure that all OS components involved in the boot process are optimized (prefetcher, superfetch, etc.) |
| Timing | Those traces (gathered) are used to compute the average measurement displayed in the XML report. The default number of iterations is three, but can be adjusted though the assessment configuration. |
| Analysis | A single trace is captured that contains detailed events and stacks in order to allow deep dive investigation into performance issues. The trace is larger in size. |

If you want to open one of the traces generated by the assessment, follow these steps:

1. Click on the report table header cell.

The right pane in the WAC UI updates and shows links to the ETL traces captured by the assessment.



1. Click on the **Analysis trace** link.

WPA automatically opens the trace so you can start your investigations. The next exercise will walk you through some analysis methodologies.

# Exercise 2 – Evaluate Fast Startup using WPT

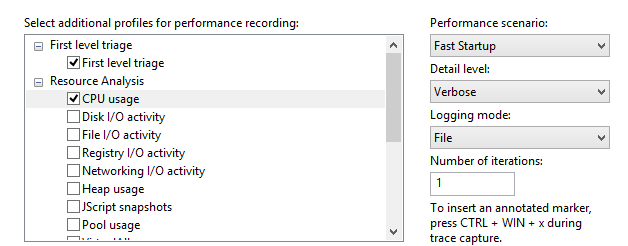
**Estimated Time**: 15 minutes

While the Fast Startup assessment is an easy way to get measurements in an easy to read report, it requires you to install the ADK, which takes some time to execute. It’s possible to quickly capture a Fast Startup trace using the Windows Performance Recorder tool.

## Step 1: Open Fast Startup trace using WPA

**Note**: You can skip this step for the WinHEC lab, as the following steps use a pre-generated trace.

1. Open **Windows Performance Recorder (WPR)** from the **Start** menu
2. Modify the tracing configuration
   1. Select the **First Level Triage** and **CPU Usage** providers.
   2. Change the **performance scenario** to **Fast Startup**.
   3. Change the **Number of iterations** to 1 in order to gather a single trace.



1. Click on **Start.**
2. Enter a path to save the resulting trace, and click on **Save**.
   1. This will force the system to reboot to gather and save the trace.
3. Once the system reboots, wait 5 minutes for tracing to end.

You now have a trace that can be analyzed with **Windows Performance Analyzer (WPA)**

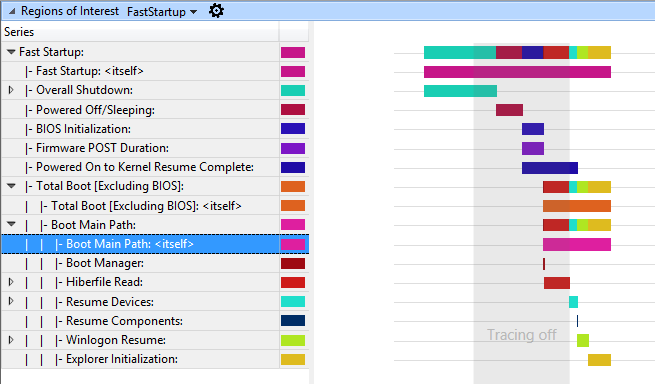
## Step 2: Open Fast Startup trace using WPA

1. Open **Windows Performance Analyzer (WPA)** from the Start menu.
2. From the **File** menu, open the **FastStartup\_Analysis\_1.etl** trace provided with the lab (**C:\Performance**).
3. Open the **Profiles** menu, and click on **Apply…**
   1. Click on **Browse Catalog…**
   2. Select **FastStartup.wpaprofile**.
   3. Click on **Open**.

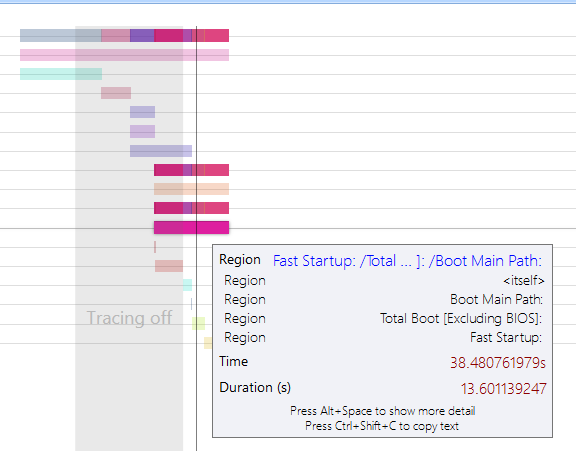
You now have applied a visualization profile to the trace in order to get some commonly used graphs (CPU, disk, etc.).

## Step 3: Visualize the activity timeline

1. Look at the **Regions of Interest** graph in the **Deep Analysis** tab
   1. This view provides a timeline overview of all the **Fast Startup** subphases mentioned in Exercise 1.



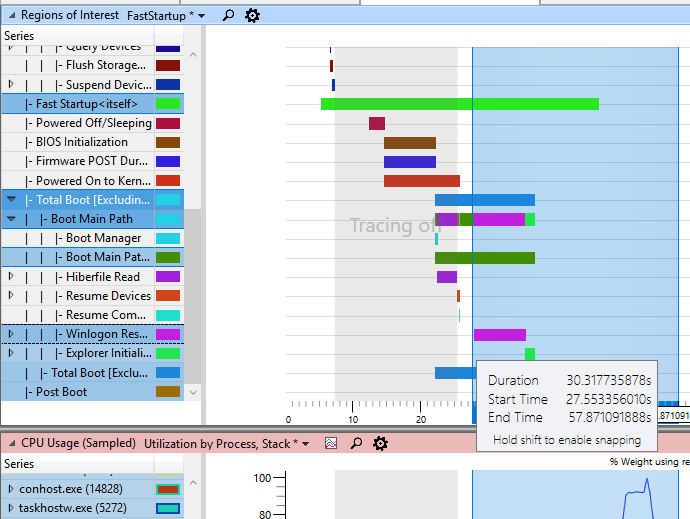
1. Hovering the mouse over a region bar causes a popup window to appear and provide more information for the region itself.
   1. If you put the mouse over the **Boot Main Path** region, you can see that it lasts 13.6 seconds.

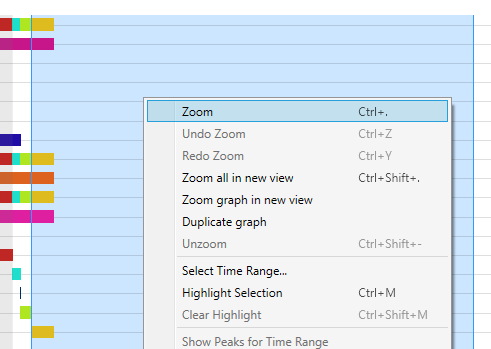


Take the time to navigate through the regions tree, and look at all the subphases to familiarize yourself with it.

You can see that **Winlogon** takes 7.5 seconds to complete. That’s the time it took to log the user in until we start initializing the shell . If you expand the region, you’ll see that most of the time was spent requesting the user crendentials.

Select a 30 seconds interval starting at the beginning of Winlogon Resume and zoom in.





Under the **Regions of Interest** graph, there are two other valuable graphs: **CPU Usage (sampled)** and **Disk usage.** They will be used to evaluate the impact that the software preload has on **post on/off** resource consumption and responsiveness.

High CPU usage by applications and services can contribute to a poor user experience, such as UI unresponsiveness and video and sound glitches. When a single process uses too much CPU, other processes can be delayed because they must compete for system resources.

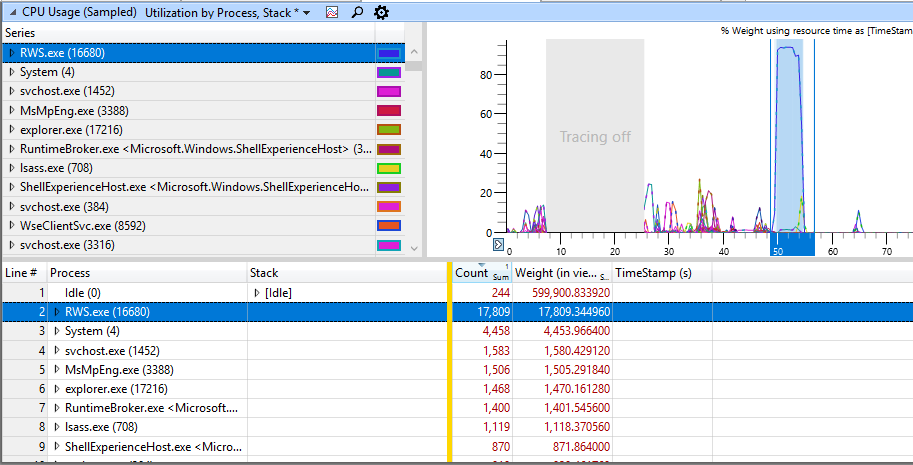
When a thread uses storage resources, it can increase the duration of the activity. When multiple threads contend for the use of storage, the resulting random disk seeks make delays more significant.

## Step 4: Analyze process CPU usage

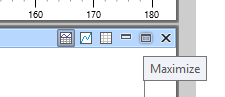
In order to evaluate how much CPU time is consumed by a process, focus on the **CPU Usage** **(sampled)** graph. The data that displays in the **CPU Usage (Sampled)** graph represents samples of CPU activity taken at a regular 1ms sampling interval. Each row in the table represents a single sample.

Any CPU activity that occurs between samples is not recorded by this sampling method. Therefore, activities of very short duration such as interrupts are not well represented in the **CPU Sampling** graph.

Review the CPU usage for each process to identify the processes that have the highest CPU usage (**Weight** and **%Weight)**. To do this, scroll down to the graph **CPU Usage (sampled)**. On the left, view the list of processes. Each active process that is selected on the left displays on the graph.



**Tip**: While using **WPA** graphs, you can change the view to display both the graph and the table. You can click the **Maximize** button to hide the other graphs displayed on the **Analysis** tab.

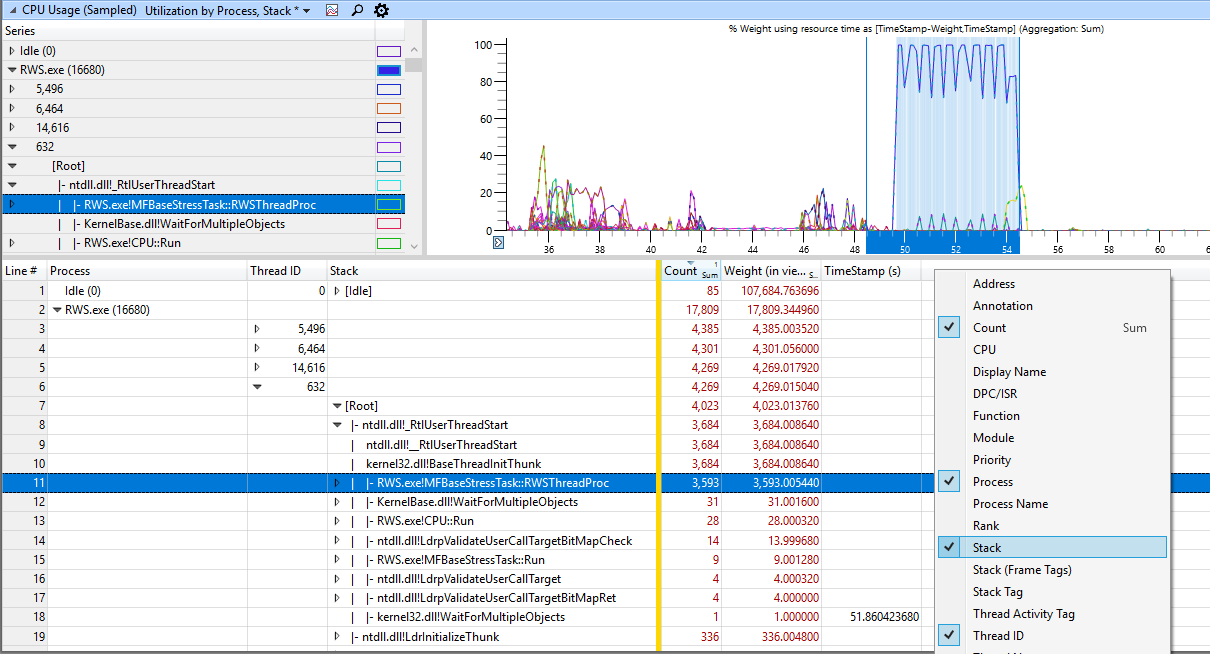


In this example, **RWS.exe** consumes 17.8 seconds of CPU time over the interval of 30 seconds currently analyzed. Since the CPU on this system has two cores, this represents a relative percentage of utilization of 12%.

Using this information, you can investigate the specific process that is causing this CPU consumption, or forward these details to the developer who owns this process.

You can add additional columns to extract more information (right-click on the table column headers):

* **Thread ID**: Identificator of the thread causing CPU usage
* **Stack**: Call stack that highlights the code paths and functions that are causing CPU usage



In the example above, there are four threads causing most of the CPU usage within the **RWS.exe** process. Each thread consumes the same amount of CPU time (4.3 seconds) and is running function MFBaseStressTask::RWSThreadProc

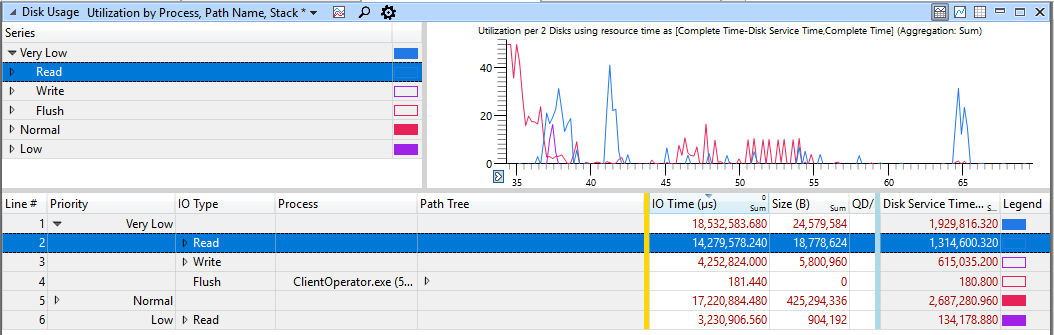
## Step 5: Analyze process disk usage

In order to evaluate how much Disk bandwidth is consumed by a process, focus on the **Disk Usage** graph.

The columns of interest are:

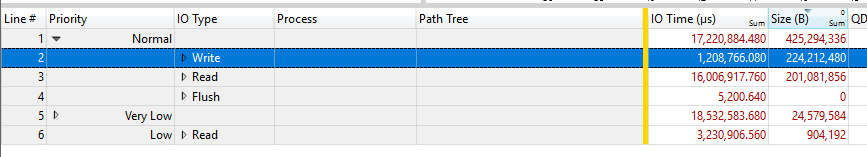
* **Pri:** Priority of the disk I/O (normal, low, very low)
* **IO Type:** Type of the I/O (read, write or flush)
* **Process:** Identificator of the process that created the disk I/O
* **Path Tree**:Structured tree representing the locations of the files accessed by the I/O
* **Size:** Size (in bytes) of the I/O
* **Disk Service time:** Amount of time it took the disk to service the I/O
* **IO Time:** Amount of time the I/O spent in the Windows I/O queue
  + **IO Time** is always longer than **Disk Service Time**, as an I/O can be queued when theirs is disk contention, or when I/O dispatcher at higher priority must be completed first.

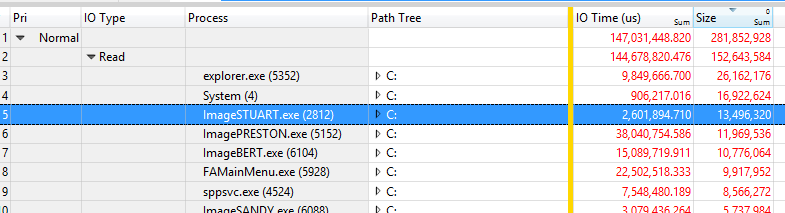
Add these columns and arrange them to obtain this view:

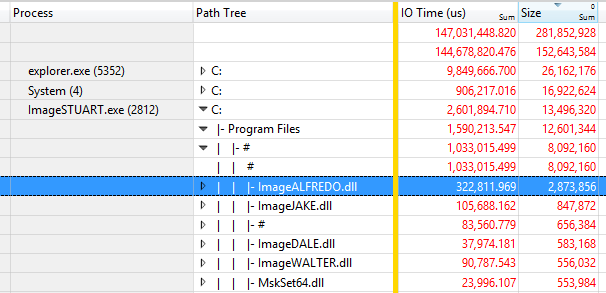


**Post on/off** only takes into account normal priority I/Os. Investigate the information about those disk reads according to process. Disk reads usually account for more disk access time than disk writes on boot, as a lot of data must be read from disk in order to launch processes and services.

1. In the table view, expand the **Normal** priority row.
2. In the table view, expand the rows for **Write, Read** and **Flush**, and then click the header for the **Size** column to sort the contents in decreasing order.
   1. 200MB of data was read from disk at normal priority
   2. 224MB of data was written to disk at normal priority
      1. Those are mainly disk writes to persist the captured ETL trace file on storage.



1. In the table view, expand the **Read IO Type** row.
   1. You should now be able to see the processes that caused the largest amount of read disk I/O during **post on/off**.
2. Identify the top process that are contributing to disk reads and that are not Windows components.
3. In the table view, expand the **Path Tree** row for **RWS.exe**, and navigate through it.



In the example above, **RWS.exe** reads 72MB of data from disk when launched during **post on/off**, and most of the accesses are made reading one rsw-read.bin file and loading required DLLs for initialization.

Using this information, a software developer should identify his components and processes, and determine if the component size can be reduced, or if the launch code path can be optimized to minimize the amount of data read from disk.

You can also use this data to identify the 3rd party processes launched on boot and causing high disk usage. If a process appears to be introducing disk contention, it can then be removed from the image or simply not started at boot time.

# Exercise 3 – Understand critical path and wait analysis

**Estimated Time**: 30 minutes

Scenarios and activities can be delayed. For example, the user launches an app but it takes longer than expected to open the UI.

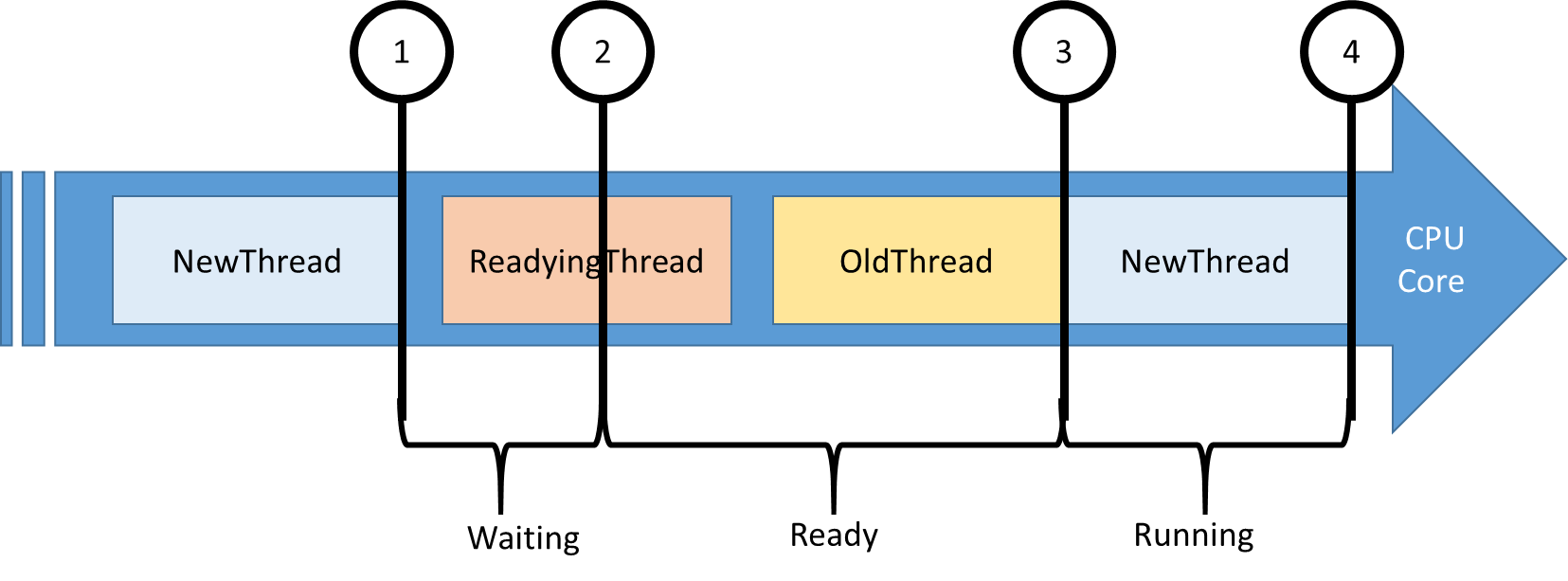
An activity is a series of operations, some sequential and some parallel, that flows from a start event to an end event. Any start/end event pair in a trace can be viewed as an activity. The longest path through this series of operations is known as the critical path. Reducing the duration of any operation on the critical path directly reduces the duration of the overall activity.

It is recommended that you identify the process and the thread that completed the activity and work backwards from the time the activity completed. Start by analyzing the thread that completed the activity to determine how that thread spent most of its time and in what state: *Waiting*, *Ready*, or *Running*.

Significant running time indicates that direct CPU usage might have contributed to the duration of the critical path. Time spent in the ready mode indicates that other threads contribute to the duration of the critical path by preventing a thread on the critical path from executing. Time spent waiting points to I/O, timers, or other threads and processes on the critical path for which the current thread was waiting.

Each thread that readied the current thread is probably another link in the critical path and can also be analyzed until the duration of the critical path is accounted for.

All the required information is recorded in the **CPU Usage (Precise)** graph and table in **WPA.** CPU usage events that are logged by the *dispatcher* are associated with context switches. This table focuses on **NewThread** which is the thread that was switched in, and each row represent a context switch. Data is collected for the following event sequence:



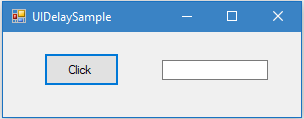
1. **NewThread** is switched out due to a blocking function call.
2. **NewThread** is made ready to run by the readying thread.
3. **NewThread** is switched in, thereby switching out an old thread.
4. **NewThread** is switched out again.

Here are the interesting columns in the **CPU Usage (Precise)** table.

|  |  |
| --- | --- |
| Column | Details |
| **% CPU Usage** | The CPU usage of the new thread after it is switched. This value is expressed as a percentage of the total CPU time over the currently visible time period. |
| **Count** | The number of context switches that are represented by the row. This is always 1 for individual rows. |
| **CPU Usage (ms)** | The CPU usage of the new thread after the context switch. |
| **NewProcess** | The process of the new thread. |
| **NewThreadId** | The thread ID of the new thread. |
| **NewThreadStack** | The stack of the new thread when it is switched in. Usually indicates what the thread was blocked or waiting on. |
| **Ready(s)** | The time that the thread spent in the Ready queue (due to pre-emption or CPU starvation) |
| **ReadyingThreadId** | The thread ID of the readying thread. |
| **ReadyingProcess** | The process that owns the readying thread. |
| **ReadyThreadStack** | The stack of the readying thread. |
| **ReadyTime (s)** | The time when the new thread was readied. |
| **SwitchInTime(s)** | The time when the new thread was switched in. |
| **Waits (s)** | The amount of time a thread waited on a logical or physical resource. The wait ends when **NewThreadId** is signaled by **ReadyingThreadId**. |

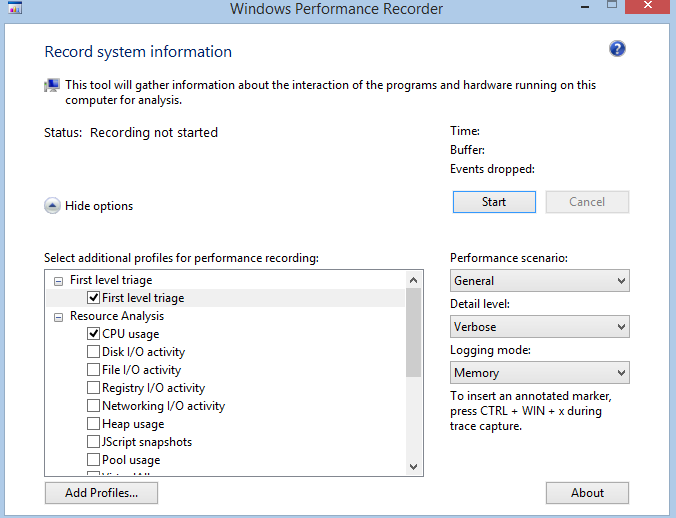
## Step 1: Capture and open a trace for a UI delay problem

This exercise will focus on a dummy process with an unresponsive UI. The process is a simple Windows Form application with a button and a text box. When the button is clicked, the UI becomes unresponsive for 20 seconds until the text box is updated. You will analyze the critical path of this operation.



You can find the process in: **C:\Performance**

1. Launch **UIDelay.exe**.
2. Open **WPR** from the **Start** menu.
3. Modify the tracing configuration.
   1. Select **First Level Triage** and **CPU Usage**.
   2. Click Add Profiles and select UIDelay.wprp
   3. Select **General** as the performance scenario.
   4. Select **Verbose** as the detail level.



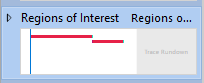
1. Click on **Start**.
2. In **UIDelay.exe** click on the **Click** button.
   1. Wait until the text box shows “*Done!*”
3. In **WPR**, save the trace and open it with **WPA**.
4. Open the **Trace** menu and select **Configure symbols path**.
   1. Specify the path of the symbol cache.
5. Open the **Trace** menu and select **Load symbols**.

Make sure to have the OS symbols present in **C:\Symbols** and add the UIDelay.pdb file to the folder

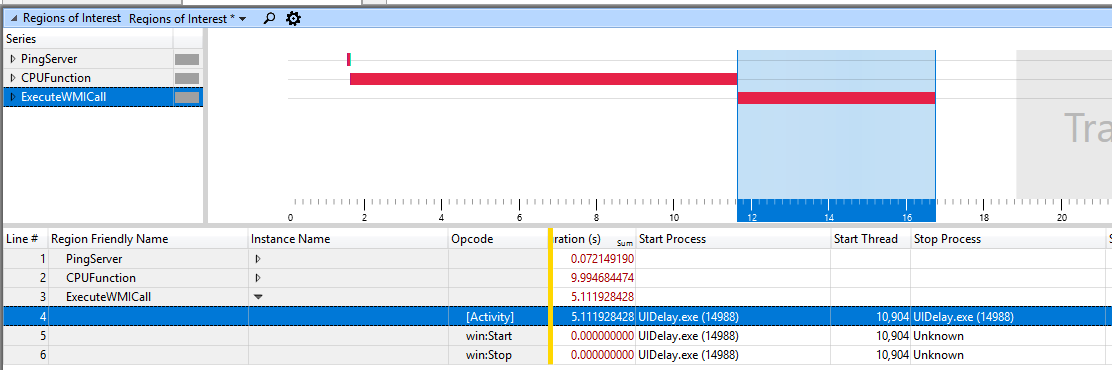
## Step 2: Identify the delayed UI thread

The first step before doing critical path analysis is to identify the activity start and stop events. Fortunately this process was instrumented using the TraceLogging API (see MSDN) and every major operation logs events that can be visualized in WPA.

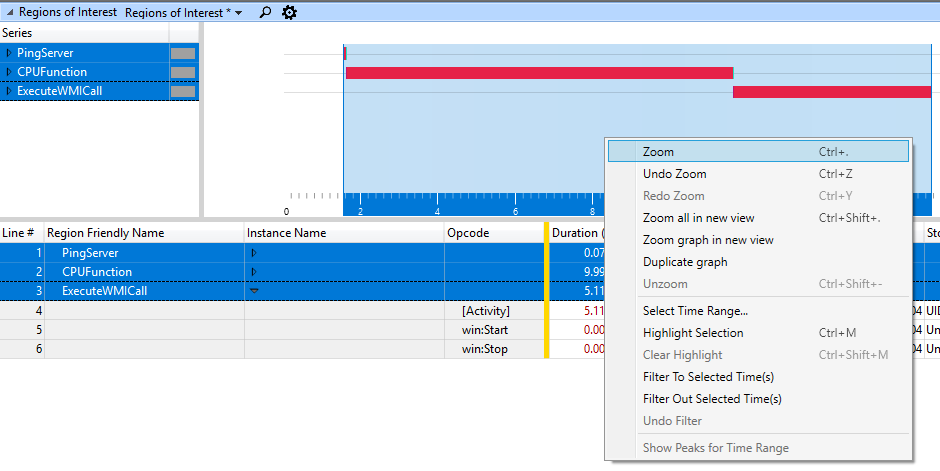
1. Find the **Regions of Interest** graph in the **System Activity** node of the **Graph Explorer**.



1. Drag and drop the **Regions of Interest** graph in the analysis tab.
2. Find the three operations performed and logged by the process.
   1. Its duration should be around 15-20 seconds which corresponds to the delay you’ve observed in the process UI.
   2. The id of the thread that logged and handled the operation is shown in the **Thread Id** column. In this example, it is 10904 for ExecuteWMICall. This value will be different in the trace you’ve capture on the lab system. Make sure to note the thread ID.



1. Select the entire time interval covered by the three operations, right-click and zoom in.



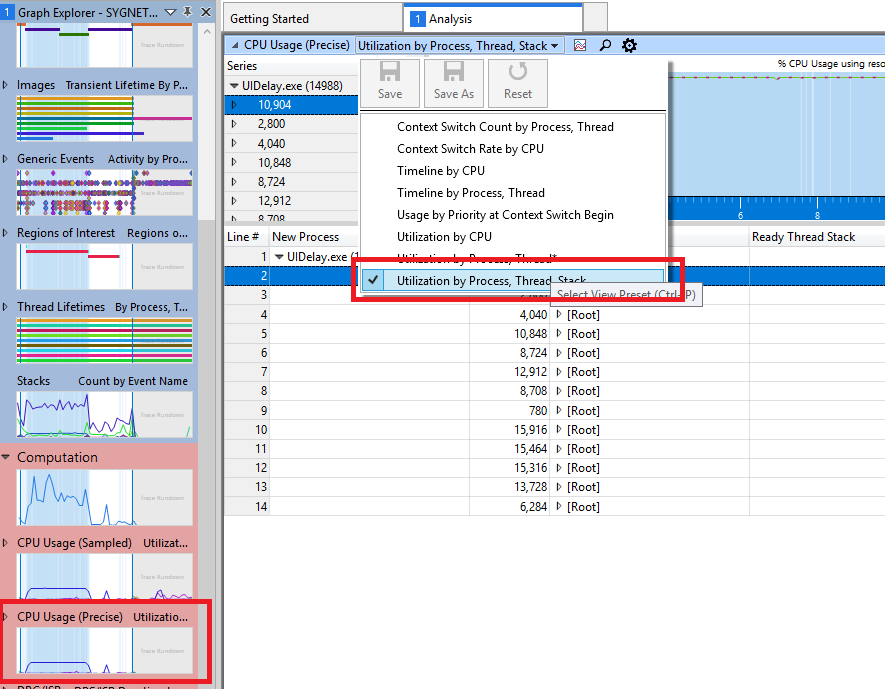
You should always zoom in the regions you’re trying to analyze. It reduces the amount of noise introduced by unrelated activities.

## Step 3: Analyze the UI delay critical path

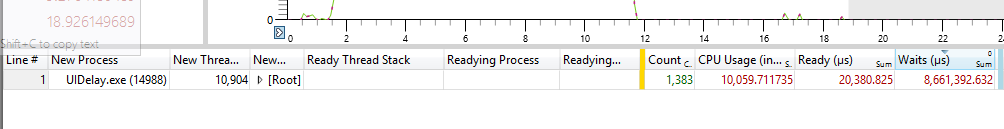
Now that you have an analysis starting point with the thread ID and the timestamps, you can start digging into the activity critical path to understand the sequence of events that lead to a 20 seconds delay on the UI thread.

**NewThreadId** for this step is the thread you’ve identified in Step 2 (example: Thread 10904 in **UIDelay.exe** process).

1. Add the **CPU Usage (Precise)** graph to the analysis tab and apply the **Utilization by Process, Thread, Stack** preset.

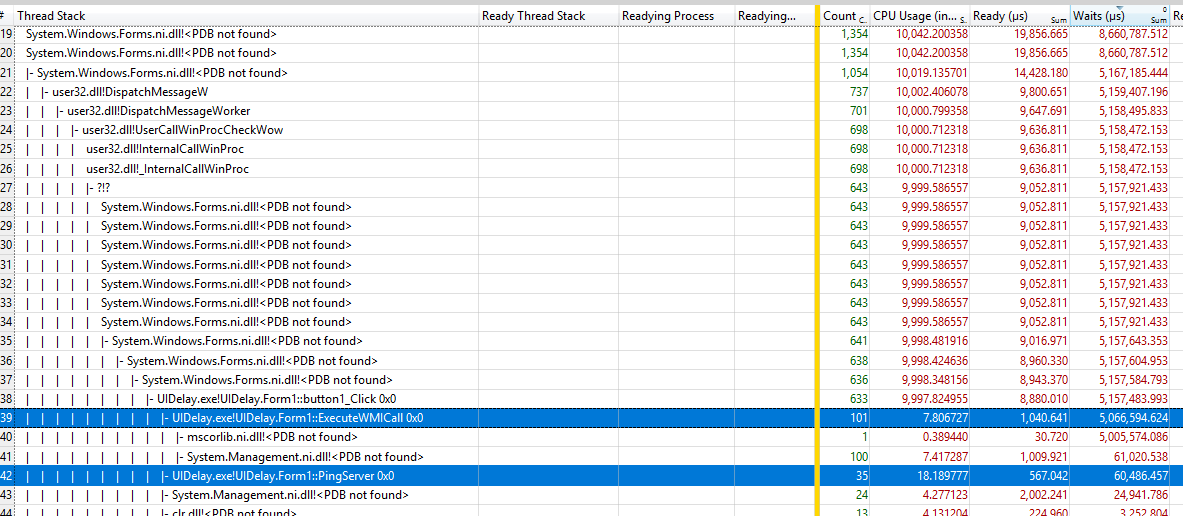


1. Find and expand the **UIDelay.exe** process in the **NewProcess** column and sort by **Waits (us) [Sum]** by clicking on the column header.
2. Search for the **NewThreadId** in the **UIDelay.exe** process, and analyze its time spent in the Running, Ready, or Waiting state.
   1. In the following example, you can find that
      1. The thread is consuming 10.025 seconds of CPU time.
      2. The thread is waiting for 8.661 seconds.
      3. The thread is in the ready state for a negligible amount of time (20ms).



You can analyze the 10 seconds of CPU activity using the same methodology described in Exercise 2, Step 4 using the **CPU Usage (sampled)** graph and looking at the **UIDelay.exe** process.

1. To discover what the **NewThreadId** was waiting for, expand the **NewThreadId** group to display the **NewThreadStack**.
2. Expand **[Root]** and identify the function calls leading to waits.

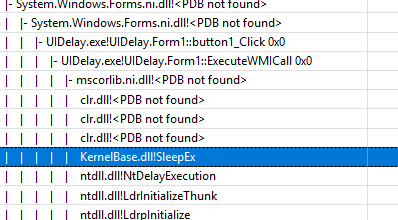


In this example, **UIDelay.exe** thread ID 24174 is waiting on underlying blocking function calls for 5.073 seconds when the button click function is triggered:

* 5 seconds are due to operations underneath the **ExecuteWMICall** function.
* 60ms are due to operations underneath the **PingServer** function.

### Step 3.1: Look at the ExecuteWMICall code path

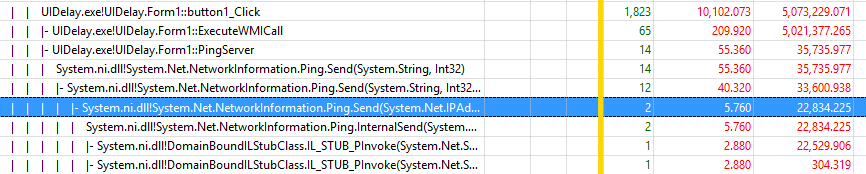
If you expand the call stack further under **ExecuteWMICall**, you’ll find that the UI thread is actually sleeping for 5 seconds by explicitly calling **Thread.Sleep**.



This kind of behavior should be avoided at all cost as it directly impacts responsiveness. If the code needs to wait for information, it should do it asynchronously on a separate thread and use an event-driven method.

### Step 3.2: Look at the PingServer code path

If you expand the call stack further under **PingServer**, you’ll find that the UI thread has I/O dependencies as it is sending **Ping** commands over the network.



While the delay is very small (35ms), it should be avoided on a UI thread. Keep in mind that the average person will notice any UI delay larger than 100ms. This operation could increase the total activity elapsed time above 100ms, resulting in users having a bad perception of responsiveness.

Those operations should happen asynchronously on a separate thread and not block the UI.

# Annex I - Technical background

## Tools

The Windows Assessment Toolkit in the ADK provides a set of performance-related tests called assessments. The assessment results are used to diagnose potential problems, so that the hardware and software that you develop are both responsive and have a minimal impact on battery life, startup performance, and shutdown time. The same assessments are available to OEM/ISV/IHV partners, enthusiasts, and other members of the community, to establish a common framework to measure, compare, and review aspects of quality.

The Windows® Performance Toolkit consists of two independent tools: Windows® Performance Recorder (WPR) and Windows® Performance Analyzer (WPA). WPR is a powerful recording tool that creates Event Tracing for Windows (ETW) recordings. You can run WPR from the user interface (UI) or from the command line (CL). WPR provides built-in profiles that you can use to select the events you want to record. WPA is a powerful analysis tool that combines a flexible UI with extensive graphing capabilities and data tables that can be pivoted and have full text search capabilities.

## Fast Startup behavior

Introduced in Windows 8, Fast Startup is the default boot behavior. The shutdown process was updated to include writing data to disk in a way that mirrors how hibernate works. During boot, the system goes through the phases that are described in the following table.

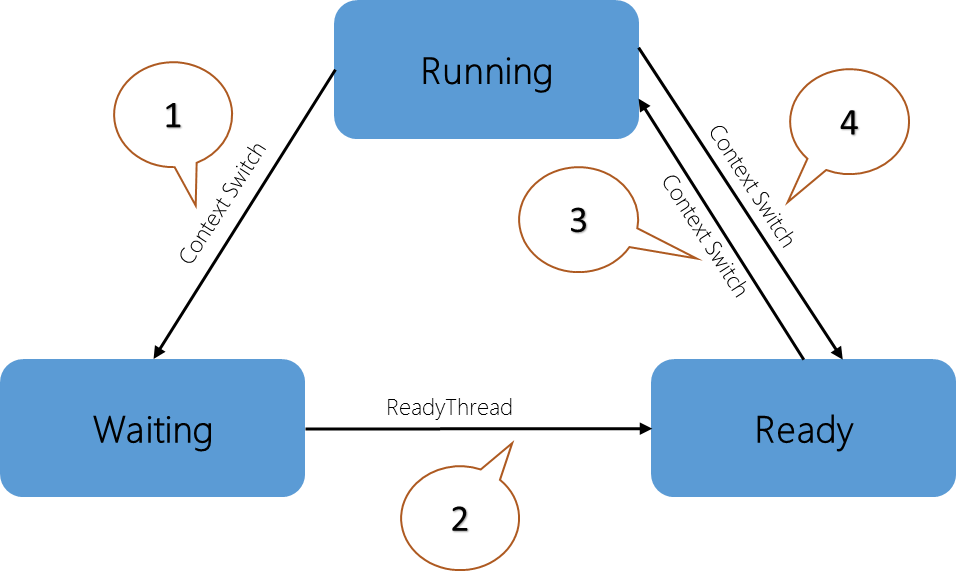
|  |  |
| --- | --- |
| Phase | Description |
| **BIOS Initialization** | The time the operating system takes to initialize the BIOS, including the Pre-Boot Execution Environment (PXE). |
| **Hiberfile Read** | The time the operating system takes to read the hiberfile from disk. The hiberfile contains all of the system context as written during shutdown. |
| **Resume Devices** | The time the operating system takes to resume devices and put them back in the active power state. |
| **WinLogon Resume** | The time the operating system takes to resume the Winlogon process. |
| **Explorer Initialization** | The time the operating system takes to initialize the Windows shell (explorer.exe). This phase ends when the desktop or Start screen is visible to the user. |
| **Post On/Off Duration** | The time Windows takes to complete all startup tasks after the desktop appears, but until CPU and disk resource become idle. |

More information is available on MSDN:  
<https://msdn.microsoft.com/en-us/library/windows/hardware/hh825330.aspx>

## CPU Scheduling and Threads

Because the number of processors in a system is limited, all threads cannot be run at the same time. Windows implements processor time-sharing, which allows a thread to run for a period of time before the processor switches to another thread. The act of switching between threads is called a context-switch and is performed by a Windows component called the dispatcher. Each thread exists in a particular execution state at any given time. Windows uses three states that are relevant to performance: *Running*, *Ready*, and *Waiting*.

Threads being executed are in the *Running* state. Threads that can execute, but are currently not running, are in the *Ready* state. Threads that cannot run (because they are waiting for a particular event) are in the *Waiting* state. The following figure illustrates the possible thread transitions.



1. A thread in the Running state initiates a transition to the Waiting state by calling a wait function such as **WaitForSingleObject** or **Sleep(> 0)**.
2. A running thread or kernel operation readies a thread in the Waiting state (for example, **SetEvent** or timer expiration).
3. A thread in the Ready state is scheduled for processing by the dispatcher when a running thread waits or reaches the end of its quantum of execution.
4. A thread in the Running state is switched out and placed into the Ready state by the dispatcher when it is preempted by a higher priority thread or when its quantum ends.

Thread state becomes an important factor in performance only when a user is waiting for a thread to complete an operation.

More details about CPU analysis are available on MSDN:  
<https://msdn.microsoft.com/en-us/library/jj679884.aspx>