



# Hands-On Lab

---

*Identify Lock Contention using the Visual Studio 2010 Parallel Performance Analyzer*

Lab version: 1.0.0

Last updated: 3/13/2010



developer & platform **evangelism**

## Contents

Introduction.....	3
Lock Contention Background .....	4
Exercise 1: Analyzing Lock Contention under Concurrency Visualizer .....	8
Part 1: Setting Environment and Creating the Reports .....	9
Part 2: Analyzing Lock Contention under Critical Sections .....	18
Part 3: Analyzing Lock Contention under Mutex .....	26
Extra Credit .....	31
Lab Summary .....	35

## Introduction

When working with parallelism we usually find that resources such as files, memory sections, devices, variables, or data-structures are often shared between threads. Usually, when all threads use these resources for read-only purposes, no thread-synchronization control is required. On the other hand, when any one thread seeks to modify a resource, then we have to synchronize resource access. One way to introduce resource synchronization is via a critical section lock.

Locks generate overhead including the cost of thread context switches. We will see how important it is to determine well-calibrated critical code sections in order to reduce the time that threads wait or contend for resources; known as lock contention.

During this lab you will learn how to use new **Visual Studio 2010 Parallel Performance Analysis (PPA) Concurrency Visualization Tools** to analyze lock contention. We will analyze the tradeoffs between decreasing lock overhead and decreasing lock contention when choosing the number and location of locks during synchronization.

## Lock Contention Background

**Resources** have always been a concern in concurrent programming. Oftentimes, a number of processes share access to system resources such as memory, processor time, or network bandwidth. Correct resource usage is essential for the overall working of a multithreaded system. When working with parallel programming we usually find that these resources require access policies in order to avoid threads using them simultaneously. Thread synchronization is required to maintain data consistency when working with these multi-threaded parallel programs.

In a multithreaded server, it is possible for shared resources to be accessed concurrently. Besides scope object attributes, shared resources include in-memory data such as instance or class variables, and external objects such as files, database connections, and network connections. When resources can be accessed concurrently, they can be used inconsistently. To prevent this, you must control the access using synchronization techniques.

The code segments that access shared resources that must not be concurrently accessed by more than one thread of execution is known as the critical section. A critical section will usually finish execution in a fixed time. A thread, task or process will only have to wait a fixed time for a locked resource to be free.

Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use. A critical section is typically used when a multithreaded program must update multiple related variables without a separate thread making conflicting changes to that data. A critical section may be used to ensure a shared resource that can only be accessed by one process at a time.

How critical sections are implemented varies between operating systems. The simplest method (brute-force approach) is to prevent any change of processor control inside the critical section. On single processor systems, this can be done by disabling interrupts on entry into the critical section, avoiding system calls that can cause a context switch while inside the section and restoring interrupts to their previous state on exit. Any thread of execution entering any critical section anywhere in the system will, with this implementation, prevent any other thread, including an interrupt, from getting the CPU and therefore from entering any other critical section or, indeed, any code whatsoever, until the original thread leaves its critical section.

One way to synchronize access to the critical section is by using semaphores. In this approach, a thread must obtain a semaphore, to enter a critical section and releases the semaphore when exiting the section. Other threads are prevented from entering the critical section at the same time as the original thread, but are free to gain control of the CPU and execute other code, including other critical sections that are protected by different semaphores.

**Locks** are synchronization mechanisms for enforcing limits on access to a resource in an environment where there are many threads of execution. Locks are one way of enforcing concurrency control policies. A (binary) semaphore is the simplest type of lock.

A lock is known as an **advisory lock** if each thread cooperates by acquiring the lock before accessing the corresponding data. Another implementation is known as a **mandatory lock**. In this latter case, threads attempt to access resources, generating exceptions if the resource was locked. Locks are classified by the way they affect the progress of the thread when trying to access the critical section

Most locking designs block the execution of the thread requesting the lock until it is allowed to access the locked resource. A spinlock is a lock where the thread simply waits ("spins") until the lock becomes available. It is very efficient if threads are only likely to be blocked for a short period of time, as it avoids the overhead of operating system process re-scheduling. It is wasteful if the lock is held for a long period of time.

Locks typically require hardware support for efficient implementation. This usually takes the form of one or more atomic instructions such as "test-and-set", "fetch-and-add" or "compare-and-swap". These instructions allow a single process to test if the lock is free, and if free, acquire the lock in a single atomic operation.

Single processor architectures have the option of using uninterruptible sequences of instructions, using special instructions or instruction prefixes to disable interrupts temporarily, but this technique does not work for multiprocessor shared-memory machines. Proper support for locks in a multiprocessor environment can require quite complex hardware and/or software support, with substantial synchronization issues.

In concurrency, as more than one task could attempt to execute the same code simultaneously, we have no guarantee that an operation like the following will give us the lock control as expected:

#### Pseudo Code

```
if (global_lock_var == 0)
    lock = myPID; // lock seems to be free, let's set it
```

The above code does not guarantee that the task has the lock because more than one task can be testing the lock at the same time. Various tasks could detect that the lock is free at the same time and could attempt to set the lock, without knowing that other tasks are also setting the lock. This kind of careless use of locks can result in **deadlock** or **livelock**.

Deadlock occurs when two tasks are waiting for locks, each holding a lock that the other is waiting for. Unless something is done, the two tasks will wait forever. A number of strategies can be used to avoid or recover from deadlocks, both at design-time and at run-time.

A livelock is similar to a deadlock, except that the states of the involved processes are constantly changing without any progress. For a better understanding, you have a livelock example when two very polite people meet in a narrow corridor, and each tries to move aside to let the other pass, but they end up moving each time to the same side of the corridor without making any progress.

Locks **granularity** is a measure of the amount of data the lock is protecting. Choosing a small number of locks, where each lock protects large segments of data, results in less lock overhead when a single process is accessing the protected data. But it results in poor performance when multiple processes are running concurrently. This scenario increases **lock contention**.

**Lock contention** takes place when a process or thread attempts to acquire a lock held by another process or thread. The more granular the locks are, the less probable one process or thread will request a lock held by the other. An example of a bad lock granularity will be to lock an entire table for accessing just one cell; this will end in having frequent lock contention between threads that require other cells from the table.

The less granular the lock, the higher the probability that the lock will stop an unrelated process from proceeding. On the other hand, when using a fine granularity, more locks (each protecting small amounts of

data) increase the overhead of the locks themselves but reduces lock contention. Extra locks also increase the risk of deadlock.

**Lock overhead** is the extra resources required for using locks: memory space allocated for locks, the CPU time to initialize and destroy locks, and the time for acquiring or releasing locks. The more locks a program uses, the more overhead associated with the usage.

There is a tradeoff between decreasing lock overhead and decreasing lock contention when choosing the number of locks in synchronization.

**Mutual exclusion** or **mutex** algorithms are used in concurrent programming to avoid the simultaneous use of common resources from critical sections. It is an object that allows multiple program threads to share the same resource, such as file access and global variables, but not simultaneously. When a program is started, a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.

In this lab we will use the following commands to protect critical sections:

- **EnterCriticalSection**: Waits for ownership of the specified critical section object. The function returns when the calling thread obtains the ownership. The threads of a single process can use a critical section object for mutual-exclusion synchronization.

**C++**

```
void WINAPI EnterCriticalSection(  
    __inout  LPCRITICAL_SECTION lpCriticalSection  
);
```

The process is responsible for allocating the memory used by a critical section object declaring a variable of type **CRITICAL\_SECTION**. Before using a critical section, a process thread must call **InitializeCriticalSection** to initialize the object.

In order to enable mutually exclusive access to a shared resource, each thread must call the **EnterCriticalSection** function to request ownership of the critical section before executing any section of code with accesses to the protected resource. For additional information please visit the **MSDN** site:

<http://msdn.microsoft.com/en-us/library/ms682608%28VS.85%29.aspx>

- **LeaveCriticalSection**: Releases ownership of the specified critical section object.

**C++**

```
void WINAPI LeaveCriticalSection(  
    __inout  LPCRITICAL_SECTION lpCriticalSection  
);
```

For additional information please visit the **MSDN** site:

<http://msdn.microsoft.com/en-us/library/ms684169%28VS.85%29.aspx>

**WaitForSingleObject:** Waits until the specified object is in the signaled state or the time-out interval elapses. The WaitForSingleObject function checks the current state of the specified object. If the object's state is non-signaled, the calling thread enters the wait state until the object is signaled or the time-out interval elapses.

**C++**

```
DWORD WINAPI WaitForSingleObject(  
    __in HANDLE hHandle,  
    __in DWORD dwMilliseconds  
);
```

For additional information please visit the **msdn** site:

<http://msdn.microsoft.com/en-us/library/ms687032%28VS.85%29.aspx>

- **ReleaseMutex:** Releases ownership of the specified **mutex** object. The ReleaseMutex function fails if the calling thread does not own the mutex object. When the thread no longer needs to own the mutex object, the ReleaseMutex function should be called so that another thread can acquire the ownership.

A thread can specify a mutex that it already owns without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call ReleaseMutex one time for each time that it obtained ownership.

**Note:** EnterCriticalSection and LeaveCriticalSection functions use variables of **CRITICAL\_SECTION** type, these methods are known as Critical Sections. The use of WaitForSingleObjec and ReleaseMutex is known as Mutex.

## Exercise 1: Analyzing Lock Contention under Concurrency Visualizer

This exercise will show you how to use **Visual Studio 2010 Beta 2** release to analyze a multithreaded application with lock contention characteristics. The code consists on creating four threads. Each thread processes numbers to determine which of them are prime numbers. Each cycle is executed five times, establishing a lock to protect the analysis as a critical section code, making the threads to share and compete for the same critical section:

**C++**

```
for( int i = 0 ; i <= 5 ; i++)
{
    #ifdef SYNCHRONIZATION_MUTEX
        WaitForSingleObject(hMutex,INFINITE);
    #endif
    #ifdef SYNCHRONIZATION_CRITICAL_SECTION
        EnterCriticalSection(&g_CriticalSection);
    #endif
    for(int i=2;i<PRIME_RANGE_BASE;i++)
    {
        isprime = 1;
        for( int j = 2 ; j <= i ; j++)
        {
            if( i == j)
                continue;
            else if( i % j == 0)
                isprime = 0;
        }
        if(isprime){
            // prime number
        }
    }
    #ifdef SYNCHRONIZATION_MUTEX
        ReleaseMutex(hMutex);
    #endif
    #ifdef SYNCHRONIZATION_CRITICAL_SECTION
        LeaveCriticalSection(&g_CriticalSection);
    #endif
}
```

Each thread in this application has a unique id from 1 to 4. If you examine the code, you will notice that there is no sharing on this code (false sharing), so there's actually no need for the critical section (lock) that is used here. But, we will use this as an example to show how this code is represented under the Parallel Performance Analysis Tools and how we can identify lock contention consequences in an application's source code under these tools..



## Part 1: Setting Environment and Creating the Reports

1. Log onto the lab machine with the credentials that were provided.

2. Navigate and expand Visual Studio 2010 folder under:

Start -> All Programs -> Microsoft Visual Studio 2010

3. **Right click** on the **Microsoft Visual Studio 2010 – ENU** executable and select **Run as Administrator**.

4. Once Visual Studio is opened, select:

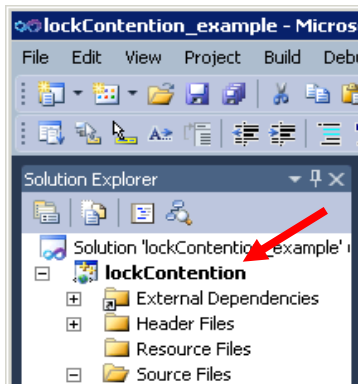
File -> Open -> Project/Solution...

5. Select the **lockContention\_example.sln** solution from the following path:

C:\Server2008 R2 Labs\Identifying Lock Contention under PPA  
Analyzer\PPA\_LockContention\_example

6. Click the **Open** button.

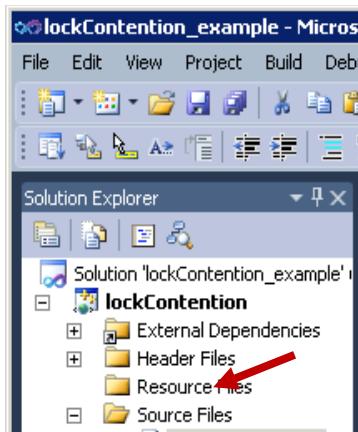
7. From the Solution Explorer, in Visual Studio 2010, click on the **lockContention** to expand the node.



**Figure 1**

*Solution Explorer - lockContention project*

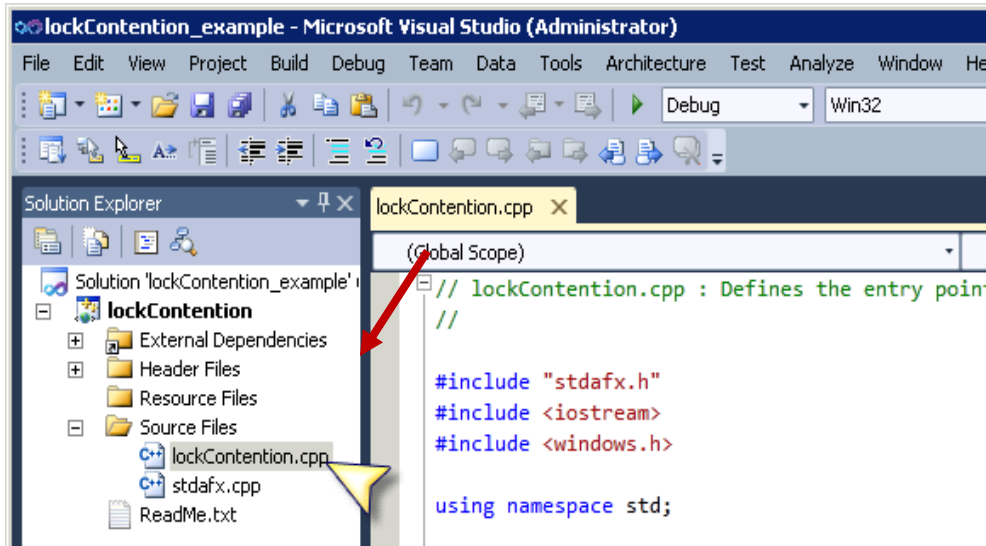
8. Click on the **Source Files** node to expand it.



**Figure 2**

*Solution Explorer - Source Files node*

9. Double click on the **lockContention.cpp** source file to open the file that will be analyzed:



**Figure 3**

*Solution Explorer - multithread\_example.cpp source file*

**Note:** The displayed code contains a main thread that creates four threads. Each thread cycles through the numbers from 2 to 10.000 checking if the number is a prime number or not. All threads start at the same time. Each thread repeats the analysis 5 times. But, for the purpose of this lab, we established the analysis as a critical section. Having this, on each of the five iterations of the analysis each thread will have to compete for the privileges of the critical zone. In the same purpose of illustration two types of Critical Section protection were implemented: **CriticalSection** and **Mutex**. Each critical section protection implementation will be activated with a **#define** preprocess command at the top of the source code.

This application logic makes the application threads need to share and compete for a highly solicited critical section.

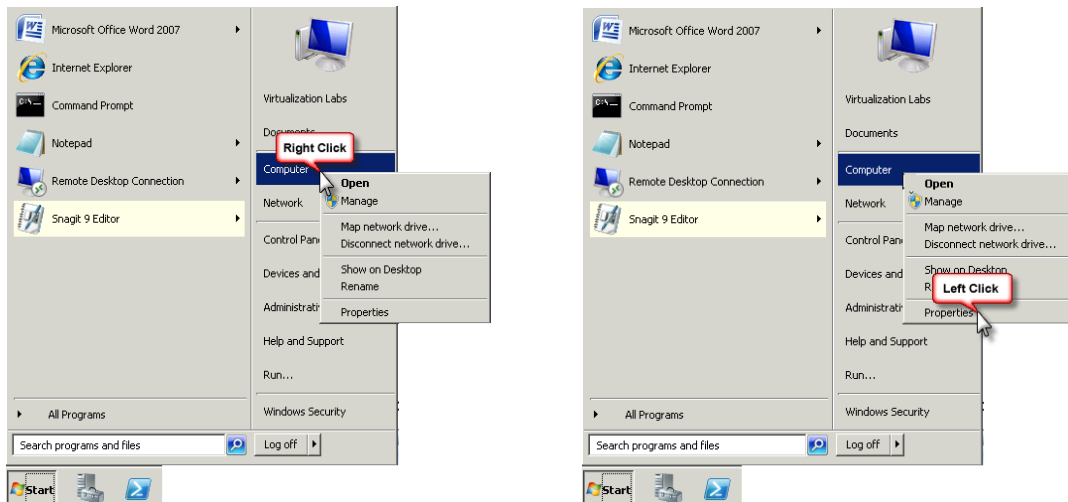
Before we start analyzing this application we need to make sure that the symbol path is well defined. Otherwise, the call stacks in the tool will not be very useful.

**SymSrv** (symsrv.dll) is a symbol server, included in the Debugging Tools for Windows package, which delivers symbol files from a centralized symbol store. This store can contain any number of symbol files, corresponding to any number of programs or operating systems. The common Microsoft debugging tools use this SymSrv technology. If you provide the correct symsrv syntax in the **\_NT\_SYMBOL\_PATH** environment variable, the debugging tools automatically include the values provided in the symbol path. You can set this variable as a system variable or as a user environment variable. For more information please visit the Microsoft Support site:

<http://support.microsoft.com/kb/311503>

Let's set a system symbol path that points to the Microsoft public symbol server on the Environment Variables.

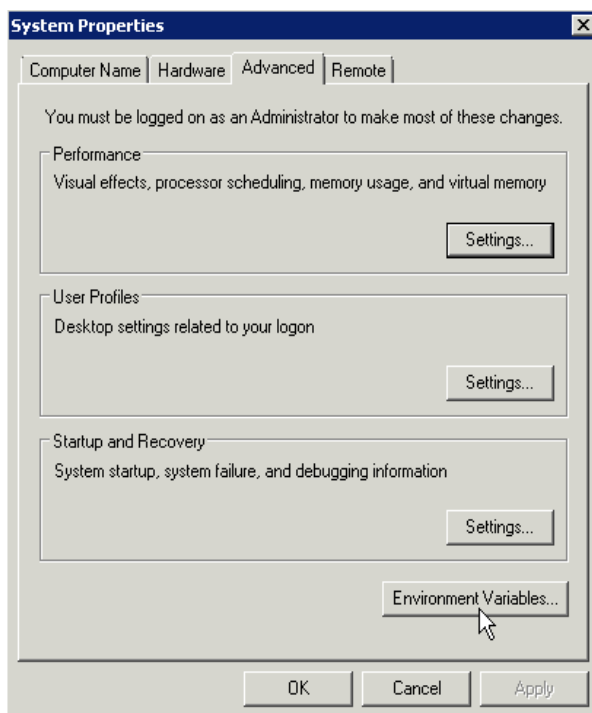
10. Go to Start->My Computer->Properties



**Figure 4**

*My Computer Properties*

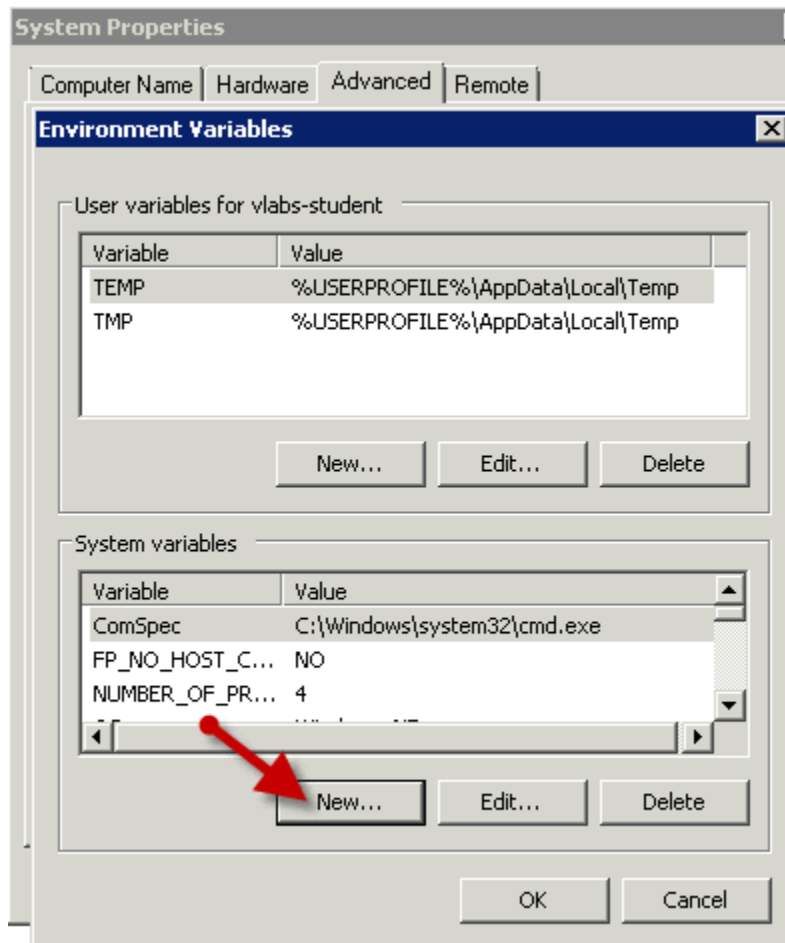
11. Click on **Advanced System Settings**.
12. Under the **Advanced tab**, click on the **Environment Variables** button.



**Figure 5**

*Environment Variables option*

13. Under **System variables** select **New**.



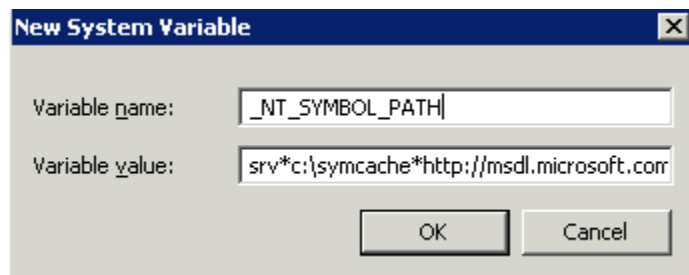
**Figure 6**

*New System Variables*

14. Type **\_NT\_SYMBOL\_PATH** as the variable name.
15. Type **srv\*c:\symcache\*http://msdl.microsoft.com/download/symbols** as the variable value

**Note:** Notice that a symbol cache is set so we won't keep retrieving symbol files from the network.

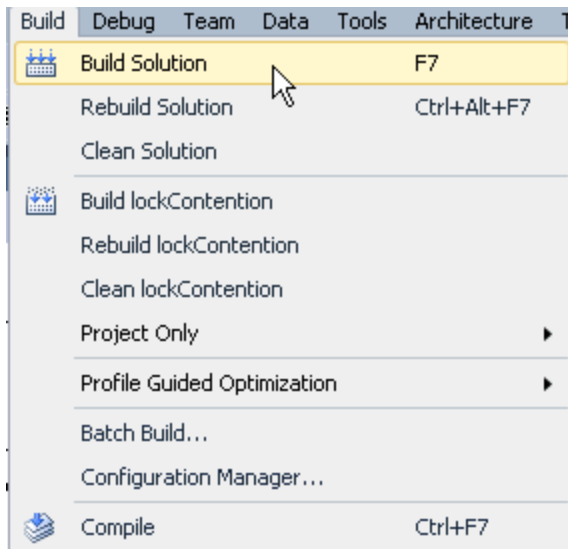
16. Press **Ok**.



**Figure 7**

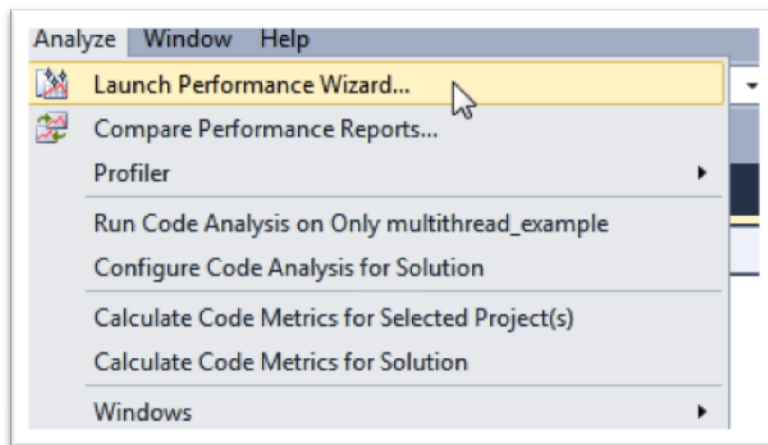
*Setting the `_NT_SYMBOL_PATH` variable*

17. On the **Environment Variables** form press **Ok**.
18. Press **ok**, on the **System Properties** form.
19. Switch back to the Microsoft Visual Studio 2010 **LockContention** project.
20. Compile the project by selecting **Build Solution** from the **Build** main menu.

**Figure 8**

*Building solution*

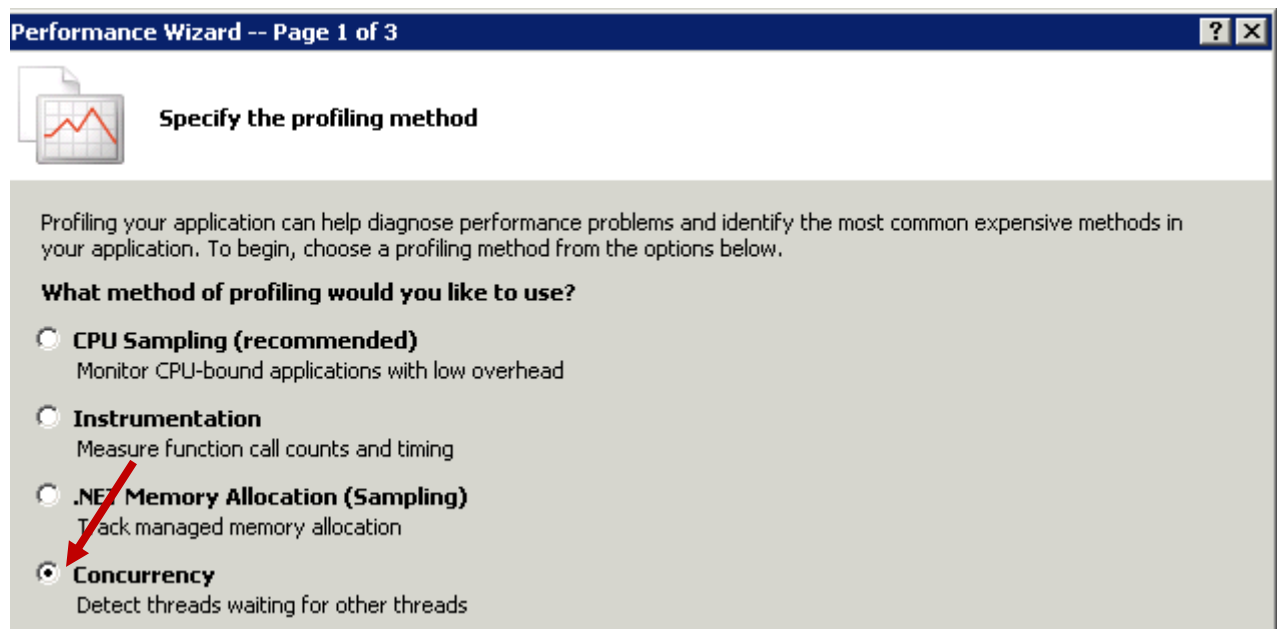
21. From the **Analyze** Menu, select **Launch Performance Wizard...**

**Figure 9**

*Analyze Menu - Launch Performance Wizard...*

**Note:** The dialog below will be displayed:

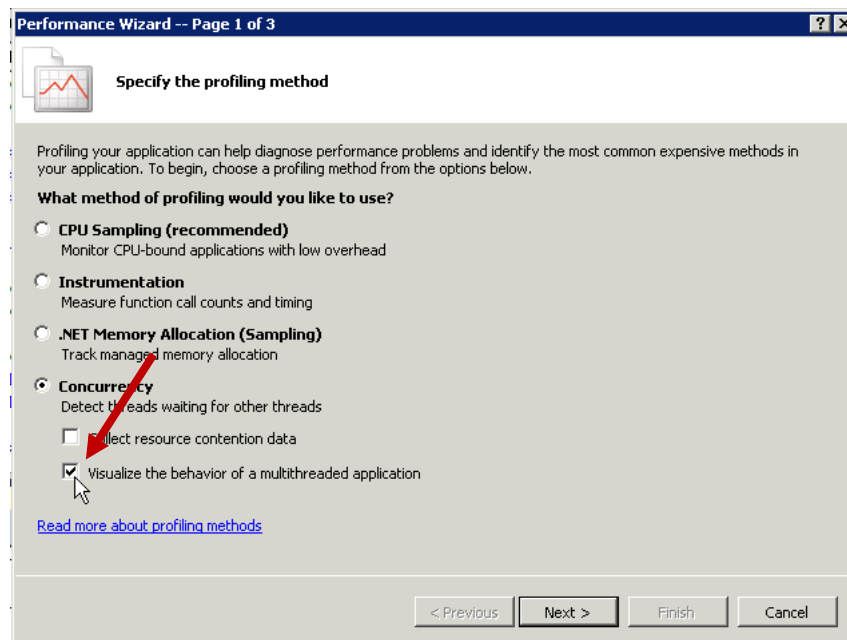
22. Select the **Concurrency** option; two checkboxes will be enabled.



**Figure 10**

*Performance Wizard - Concurrency option*

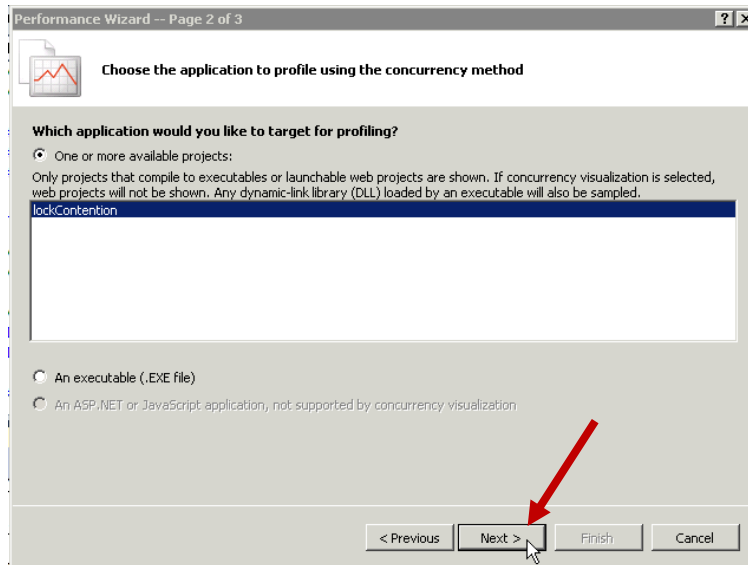
23. Select **Visualize the behavior of a multithreaded application** and click **Next**:



**Figure 11**

*Performance Wizard - Visualize the behavior of a multithreaded application option*

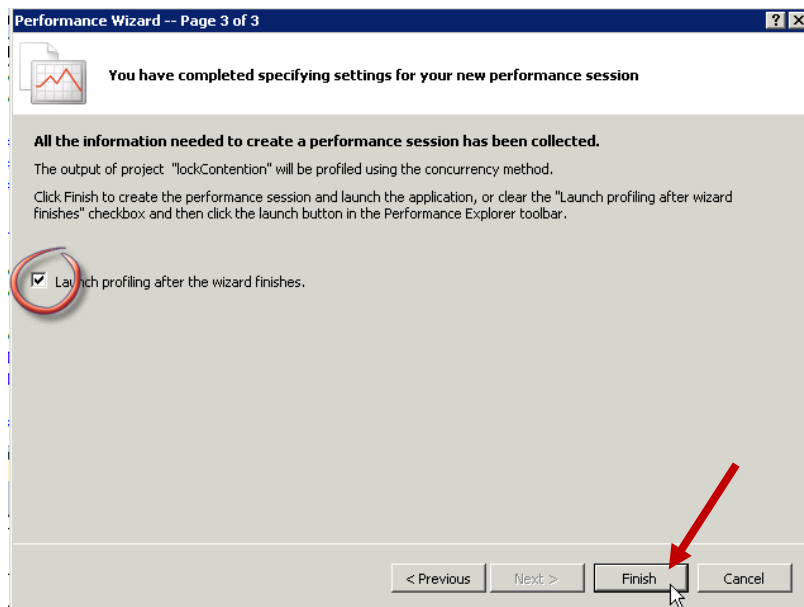
24. Leave the default option shown below and click **Next**:

**Figure 12**

*Performance Wizard – One or more available projects option*

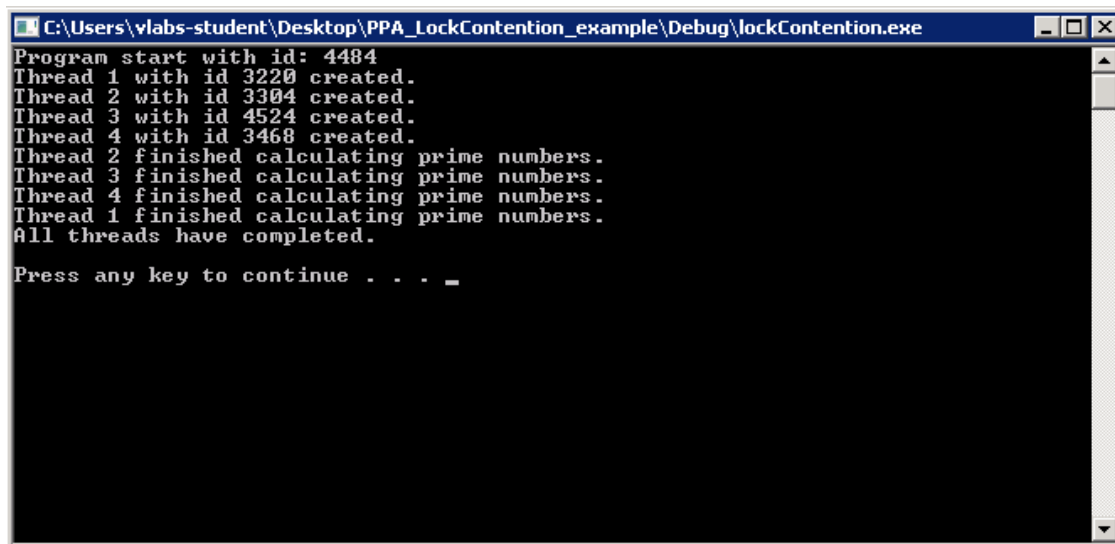
**Note:** This window allows you to: profiling executable files or profiling the code you have currently opened on the Visual Studio.

25. On the final screen, make sure the **Launch profiling after the wizard finishes** checkbox is selected and click **Finish**.

**Figure 13**

*Performance Wizard – Launch profiling after the wizard finishes option*

**Note:** The program will start executing.



```

C:\Users\vlabs-student\Desktop\PPA_LockContention_example\Debug\lockContention.exe
Program start with id: 4484
Thread 1 with id 3220 created.
Thread 2 with id 3304 created.
Thread 3 with id 4524 created.
Thread 4 with id 3468 created.
Thread 2 finished calculating prime numbers.
Thread 3 finished calculating prime numbers.
Thread 4 finished calculating prime numbers.
Thread 1 finished calculating prime numbers.
All threads have completed.

Press any key to continue . . . _

```

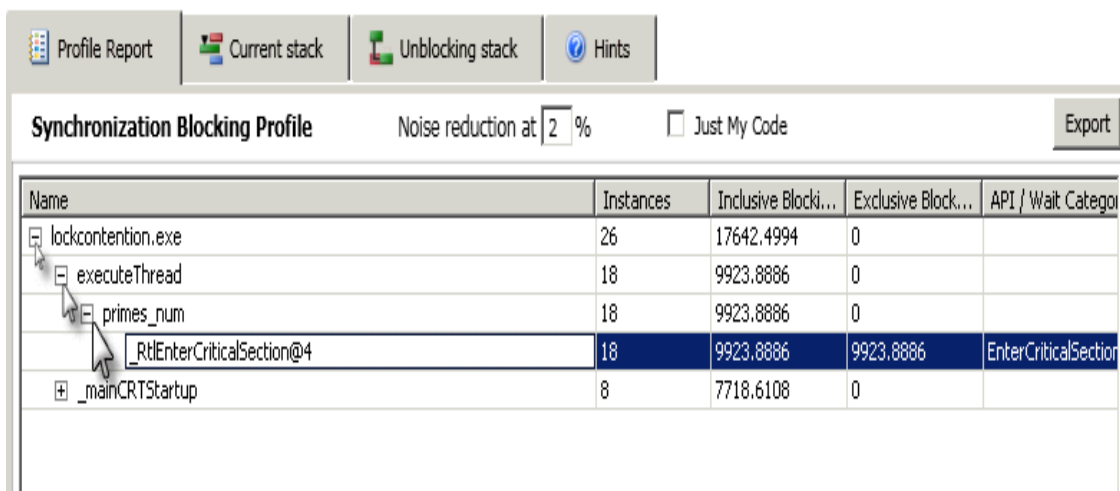
Figure 14

*Application Starts*

26. Press the **Enter** Key when prompted to **Press any key to continue**.

**Note:** During the execution of your application the PPA collects the necessary information for the reports. After the application finish running, PPA analyzes the information to create a visual representation of the application characteristics. This analysis can take a while to finish, please be patient as there are large amounts of data to be analyzed.

Once the profiling process finishes, the following visualization views will be presented:



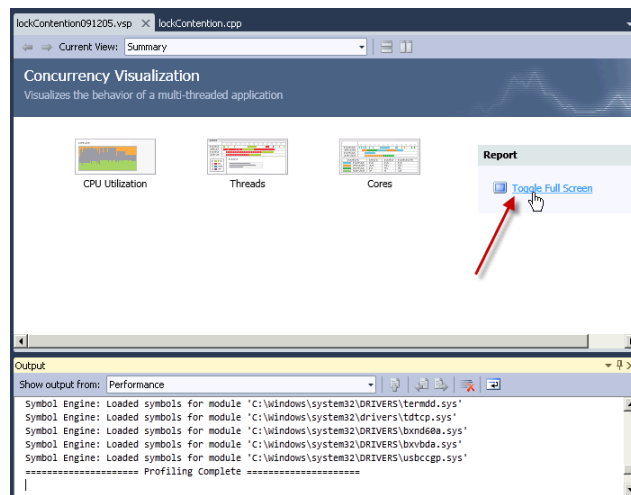
Name	Instances	Inclusive Blocki...	Exclusive Block...	API / Wait Catego
lockcontention.exe	26	17642.4994	0	
executeThread	18	9923.8886	0	
primes_num	18	9923.8886	0	
_RtlEnterCriticalSection@4	18	9923.8886	9923.8886	EnterCriticalSection
_mainCRTStartup	8	7718.6108	0	

Figure 15

*Concurrency Visualization Views*

27. Click on **Toggle Full Screen** in order to full-screen the **Concurrency Visualization** view.



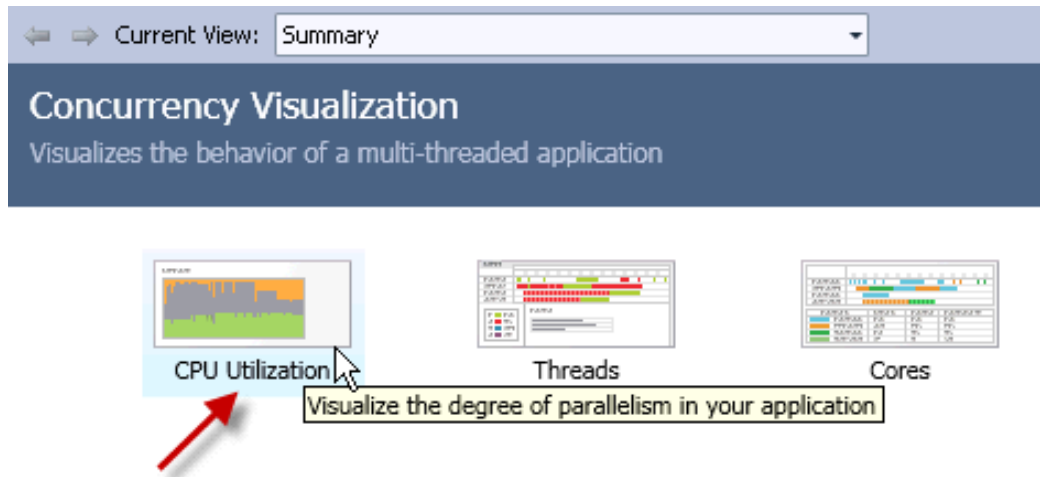
**Figure 16**

*Concurrency Visualization – Toggle Full Screen*

## Part 2: Analyzing Lock Contention under Critical Sections

**Note:** The **CPU Utilization** view is intended to tell you how well you use the CPU resources over time, and how much of the CPU available resources are being used by your application.

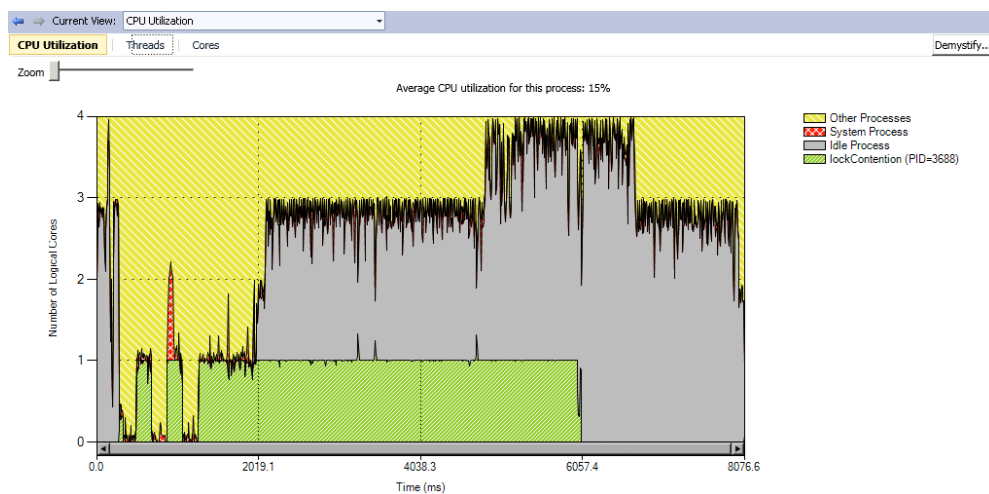
1. Click on the **CPU Utilization** icon:



**Figure 17**

*Concurrency Visualization – CPU Utilization*

**Note:** A chart of the CPU Utilization versus the time over the life of your application execution will be shown. You can use the Zoom scroll in order to better appreciate this graph



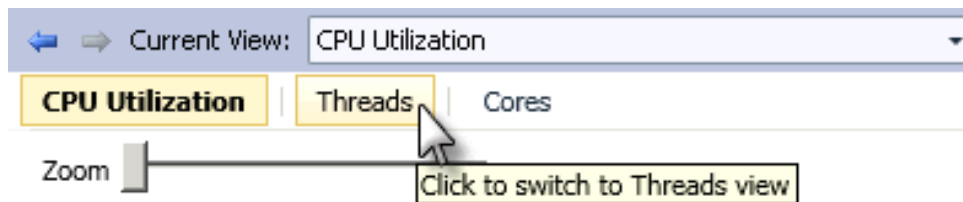
**Figure 18**

*CPU Utilization View - CPU Utilization versus time chart*

**Note:** Based on this chart, it appears that this demo application (the green area) is using only the equivalent of a single logical core in the system. Even though there are four processors in this demo machine, and even though the fact that we are creating four parallelized threads.

Our next step will consist in detecting why this application's execution is not using the whole processors power of the machine. In order to do this, let's switch to the Thread Blocking view.

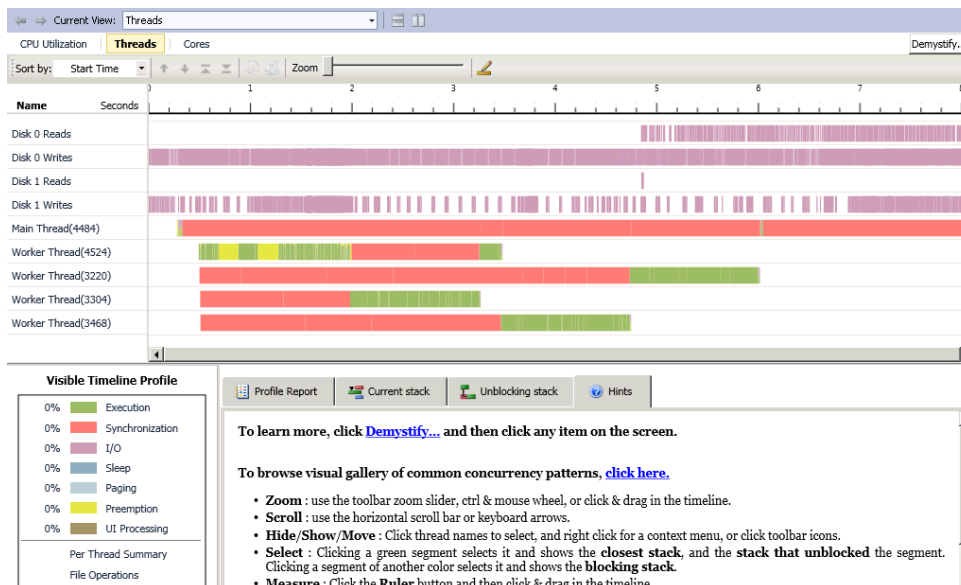
- From the current view, click on **Threads** view.



**Figure 19**

*Selecting Threads View*

**Note:** This will open the **Threads** view, as shown in the figure below.



**Figure 20**

*Threads View*

**Note:** As we see in previous steps our program consists of a main thread and four additional threads. But, there are other threads listed on the report, for example: system processes. But we are interested only in the five threads of our application so you can hide the other threads.

- Right-click over the **Disk 0 Reads** thread.

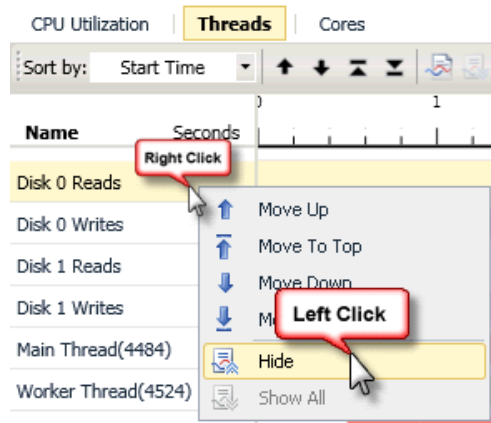


Figure 21

*Threads View – System processes*

4. Using the same approach as in the previous step, hide the **Disk 0 Writes** thread, **Disk 1 Reads** thread and **Disk 1 Writes** thread.

**Note:** If you have extra threads that not belong to our application you can also hide them.



Figure 22

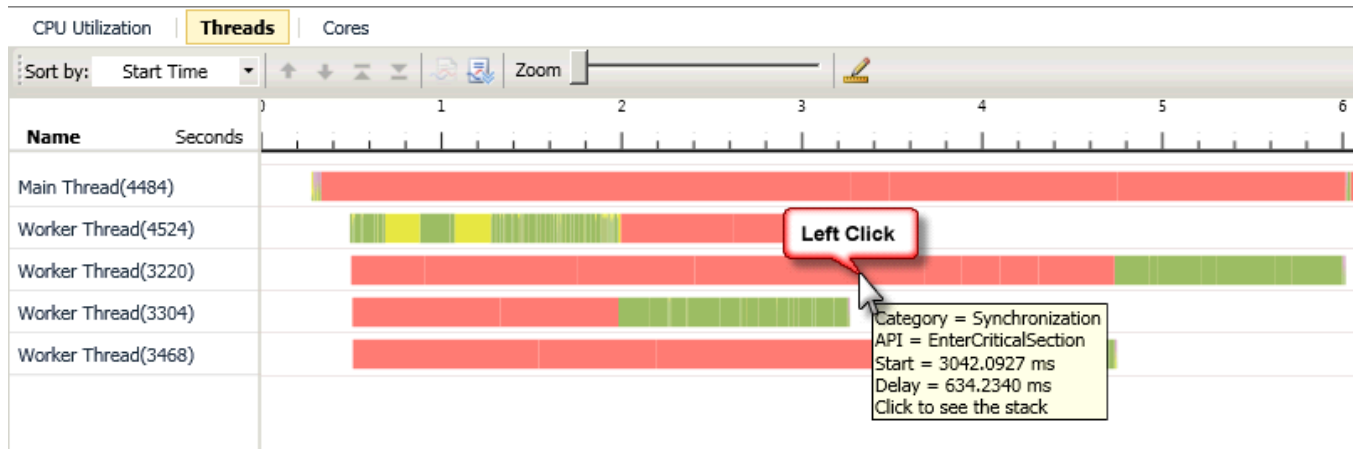
*Threads View – Demo Application threads*

**Note:** You can see in the chart that the four worker threads were created. The yellow color stands for preemption. Going back to the CPU utilization chart there could be other processes not related to our program that could result in this preemption. On this chart we can appreciate that the main thread is on synchronization (legend shows that red corresponds to synchronization delays) most of the time. We can also notice that the threads' execution is serialized (the green execution regions for the threads never overlap).

**Pay attention to this:** You can see a lot of red between the greens (synchronization). As was explained at the beginning of this lab, the application creates four threads, each of them compete for accessing a critical section five times. Then, we could expect 5 short execution periods for each thread rather than the few long execution segments that are shown on this Threads View.

Let's see why the application is behaving this way. At any time you can zoom in or out the view, using the zoom scrollbar at the upper side of the frame, to better understand the information displayed. Just remember that zooming affects the legend data and other statistics shown on the view. If you zoom in the view remember to zoom it out completely before you want to continue with the lab.

5. Click on a red region on any of the worker threads.



**Figure 23**

*Threads View – Clicking on a **Worker Thread** synchronization region.*

**Note:** The synchronization segment is now highlighted. The **Current stack** tab is activated at the bottom. The category of delay is user synchronization. An API call is shown as **EnterCriticalSection**.

Next, the call stack that resulted in the thread blocking is shown. You can see the frame showing the call to EnterCriticalSection in ntdll.dll. Having a good symbol path is important to have useful information in the stack. Now we will search for information on how expensive this particular call stack is, so that we can prioritize the performance optimization effort.

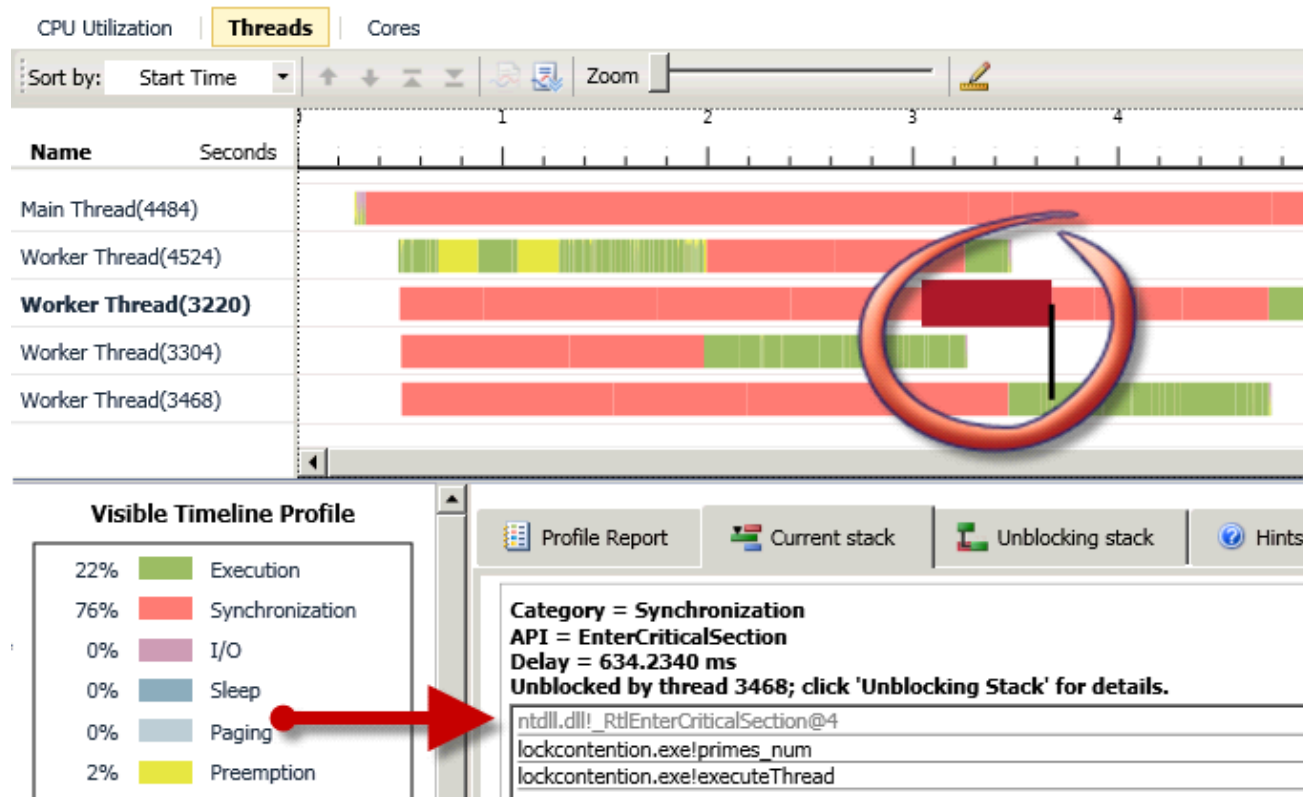


Figure 24

Threads View – Current Stack

6. Double click on the **Synchronization** label for the **Visible Timeline Profile** on the legend:

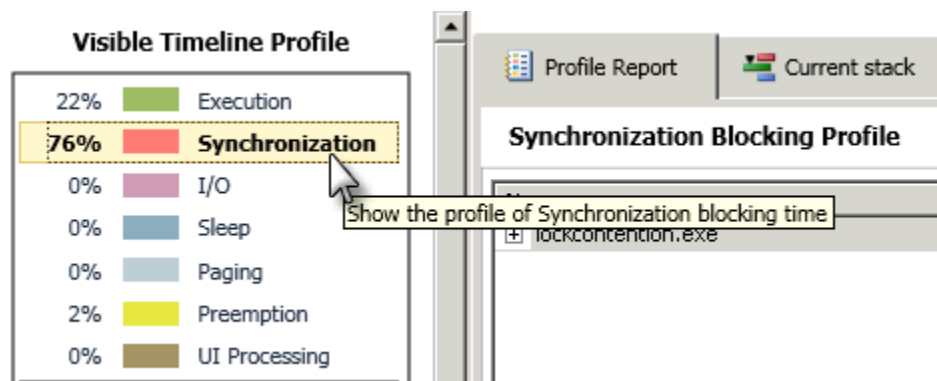


Figure 25

Synchronization Blocking Profile

**Note:** A call tree report that summarizes every synchronization call stack is shown, presenting information on how often the threads that are enabled in the current view blocked on that call stack during the displayed time period.

7. Expand the **lockcontention.exe** node under the **Synchronization Blocking Profile**:

Name	Instances	Inclusive Block...	Exclusive Block...	API / Wait Ca
lockcontention.exe	26	17642.4994	0	
executeThread	18	9923.8886	0	
primes_num	18	9923.8886	0	
_RtlEnterCriticalSection@4	18	9923.8886	9923.8886	EnterCriticalS...
_mainCRTStartup	8	7718.6108	0	

**Figure 26**

*Synchronization Blocking Profile - Expanding nodes*

**Note:** This view lets us know that there are 26 synchronization blocking events. One call stack was responsible for 18 and the other for 8 blocking events. You can also see that one of the call stacks involves the main thread (\_mainCRTStartup) and the other seems to involve the worker threads with the primes\_num function.

Now, let's examine the code that resulted in the most significant blocking stack:

8. Right click on the **\_RtlEnterCriticalSection@4** stack frame:

Name	Instances	Inclusive Block...	Exclusive Block...	API / Wait Ca
lockcontention.exe	26	17642.4994	0	
executeThread	18	9923.8886	0	
primes_num	18	9923.8886	0	
_RtlEnterCriticalSection@4	18	9923.8886	9923.8886	EnterCriticalS...
_mainCRTStartup	8	7718.6108	0	

**Figure 27**

*Examine the blocking code*

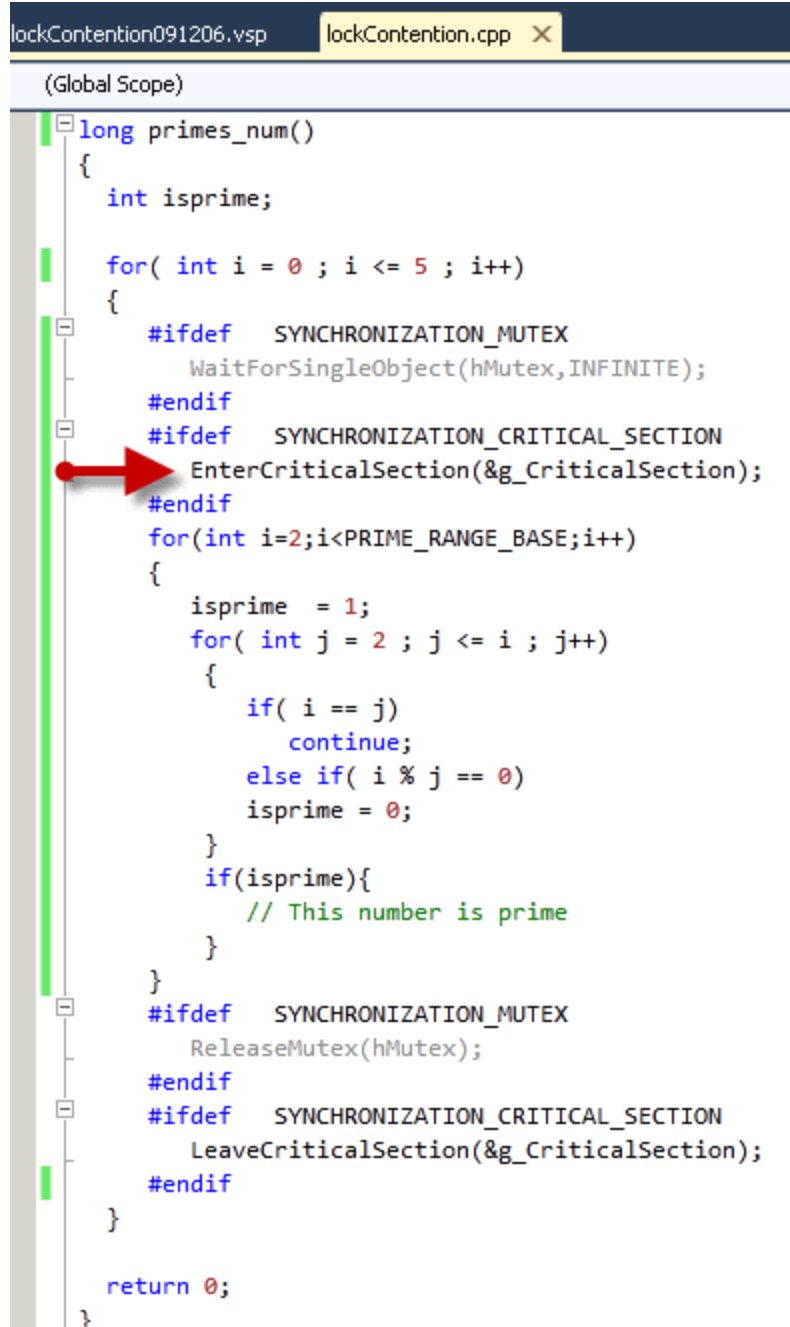
9. Select **View Call Sites** :

Name	Instances	Inclusive Block...	Exclusive Block...	API / Wait Ca
lockcontention.exe	26	17642.4994	0	
executeThread	18	9923.8886	0	
primes_num	18	9923.8886	0	
_RtlEnterCriticalSection@4	18	9923.8886	9923.8886	EnterCriticalS...
_mainCRTStartup	8	7718.6108	0	

**Figure 28**

*Selecting View Call Sites*

**Note:** **View Source** option takes you to the source code of the function specified in the frame that will corresponds to the code of the EnterCriticalSection API , but we are interested in the **View Call Sites** option, which would bring up the place in the previous function where the code is calling the blocking function:



```
lockContention091206.vsp  lockContention.cpp X
(Global Scope)

long primes_num()
{
    int isprime;

    for( int i = 0 ; i <= 5 ; i++)
    {
        #ifdef SYNCHRONIZATION_MUTEX
            WaitForSingleObject(hMutex,INFINITE);
        #endif
        #ifdef SYNCHRONIZATION_CRITICAL_SECTION
            EnterCriticalSection(&g_CriticalSection);
        #endif
        for(int i=2;i<PRIME_RANGE_BASE;i++)
        {
            isprime = 1;
            for( int j = 2 ; j <= i ; j++)
            {
                if( i == j)
                    continue;
                else if( i % j == 0)
                    isprime = 0;
            }
            if(isprime){
                // This number is prime
            }
        }
        #ifdef SYNCHRONIZATION_MUTEX
            ReleaseMutex(hMutex);
        #endif
        #ifdef SYNCHRONIZATION_CRITICAL_SECTION
            LeaveCriticalSection(&g_CriticalSection);
        #endif
    }

    return 0;
}
```

**Figure 29**

*Blocking function call on the code*



**Note:** The editor window shows us the **lockContention.cpp** code with the cursor on the call site where the call to `EnterCriticalSection()` was made. Now, you would have to spend some time determining whether this **lock** is needed and what you can do to reduce **contention** on it.

In this example, the lock is not needed at all but remember that we are doing this for the purpose of the lab.

Critical Sections do not enforce a FIFO (First in -first out) order on the threads waiting for the lock (in other words, they are not fair). The advantage of this feature is that the first thread to find the lock in a free state is allowed to acquire it even if there were other threads waiting on the lock. The disadvantage is this lack of fairness. This feature is also referred to as **anti-convoy** support. From a performance perspective, the critical section is faster because we don't have to wait for threads to wake up, removing the context switch overheads from the critical path. In this example, the application will probably be a little faster due to these characteristics. This explains the few long execution segments that are shown on the Threads View.

If we want to enforce fairness and FIFO order, we can use a synchronization primitive that does not have anti-convoy features like the Win32 Mutex.

### Part 3: Analyzing Lock Contention under Mutex

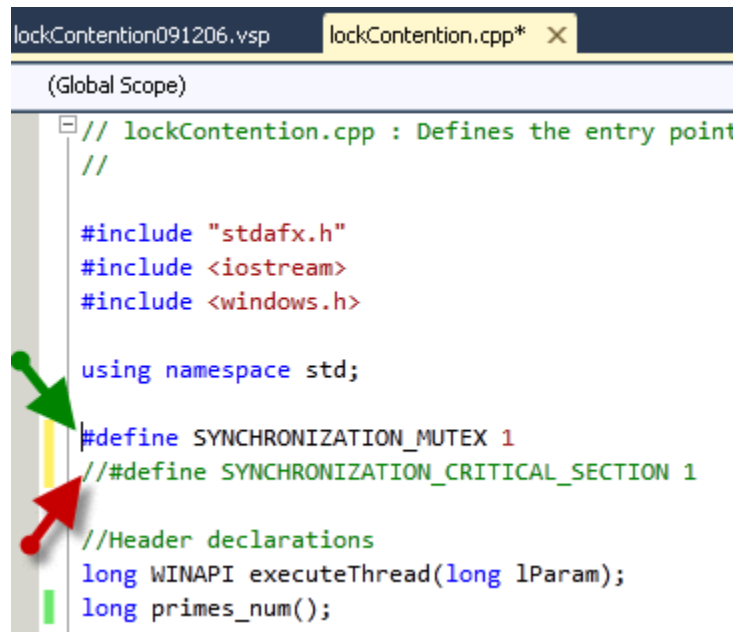
1. Scroll up to the top of the code.
2. **Comment** out the code line defining the **SYNCHRONIZATION\_CRITICAL\_SECTION** preprocessor variable:

```
C++  
// #define SYNCHRONIZATION_CRITICAL_SECTION 1
```

3. **Uncomment** the code line defining the **SYNCHRONIZATION\_MUTEX** preprocessor variable:

```
C++  
#define SYNCHRONIZATION_MUTEX 1
```

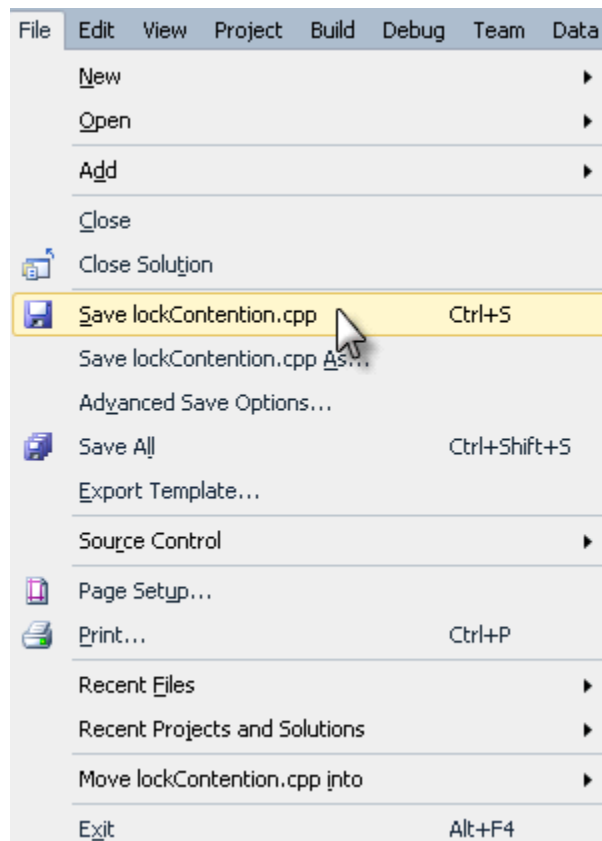
**Note:** This change makes the code use Mutex instead of Critical Section when compiling.



**Figure 30**

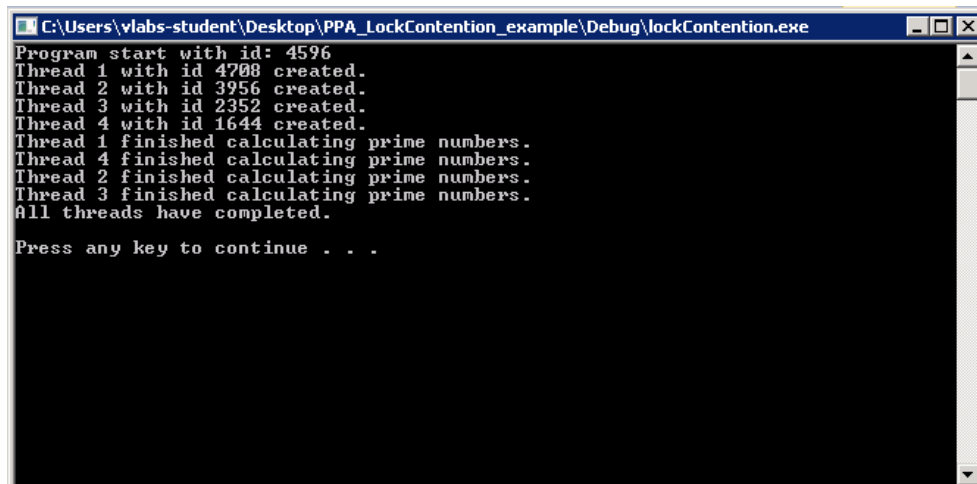
*Using Mutex*

4. Save the new changes

**Figure 31**

*Save changes*

5. **Rebuild** the solution.
6. From the **Analyze** Menu, select **Launch Performance Wizard...**
7. Select the **Concurrency** option; two checkboxes will be enabled.
8. Select **Visualize the behavior of a multithreaded application** and click **Next**.
9. Leave the default option shown below and click **Next**.
10. On the final screen, make sure the **Launch profiling after the wizard finishes** checkbox is selected and click **Finish**.



```
C:\Users\vlabs-student\Desktop\PPA_LockContention_example\Debug\lockContention.exe
Program start with id: 4596
Thread 1 with id 4708 created.
Thread 2 with id 3956 created.
Thread 3 with id 2352 created.
Thread 4 with id 1644 created.
Thread 1 finished calculating prime numbers.
Thread 4 finished calculating prime numbers.
Thread 2 finished calculating prime numbers.
Thread 3 finished calculating prime numbers.
All threads have completed.

Press any key to continue . . .
```

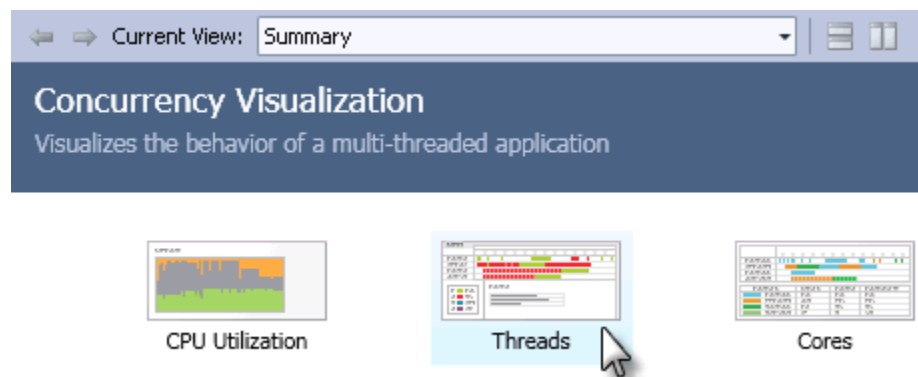
**Figure 32**

*Application Starts*

11. Press the **Enter** key to continue when prompted to **Press any key to continue**.

**Note:** After the application finishes running, PPA analyzes the information to create a visual representation of the application characteristics. This analysis can take a while to finish. Once the profiling process finishes, the following visualization views will be presented:

12. Click on the **Threads** view

**Figure 33**

*Selecting Threads view*

13. Hide the **Disk 0 Reads** thread, **Disk 0 Writes** thread, **Disk 1 Reads** thread and **Disk 1 Writes** thread and any thread that do not belong to our application.

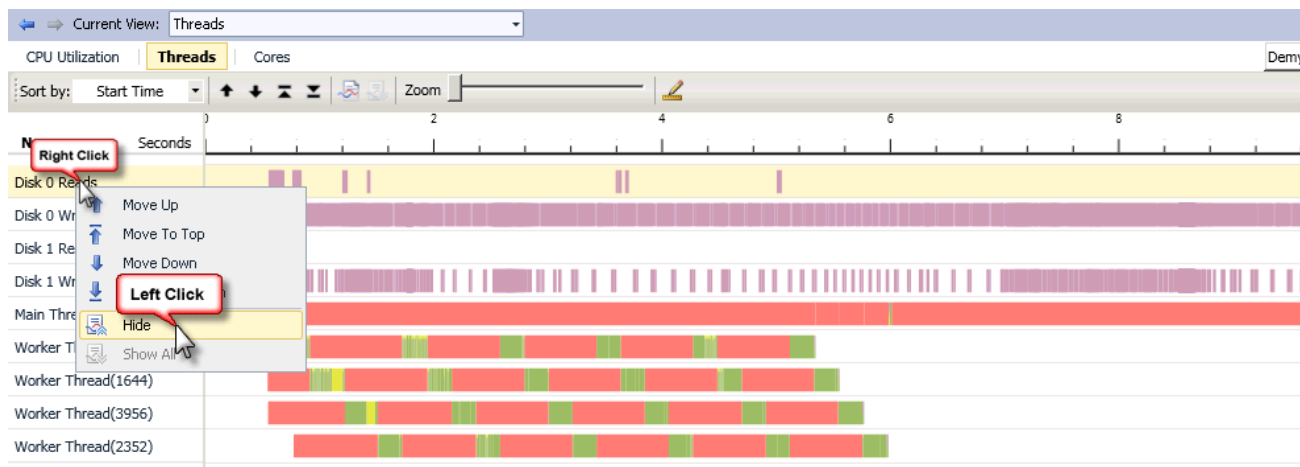


Figure 34

*Hiding irrelevant threads from the Threads View*

**Note:** On this view, you can clearly observe the convoy behavior between thread executions provided by the use of Mutexes:

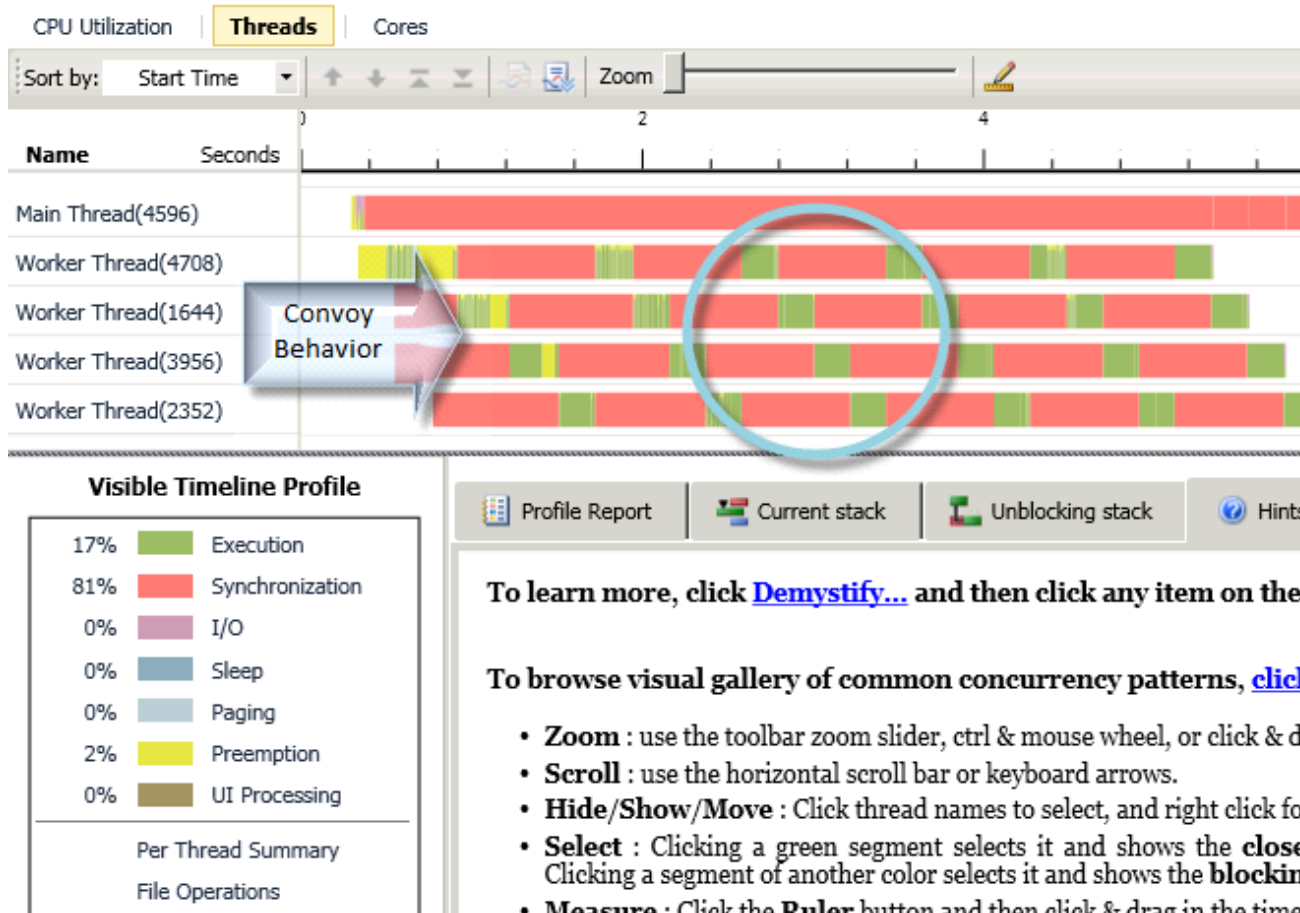


Figure 35

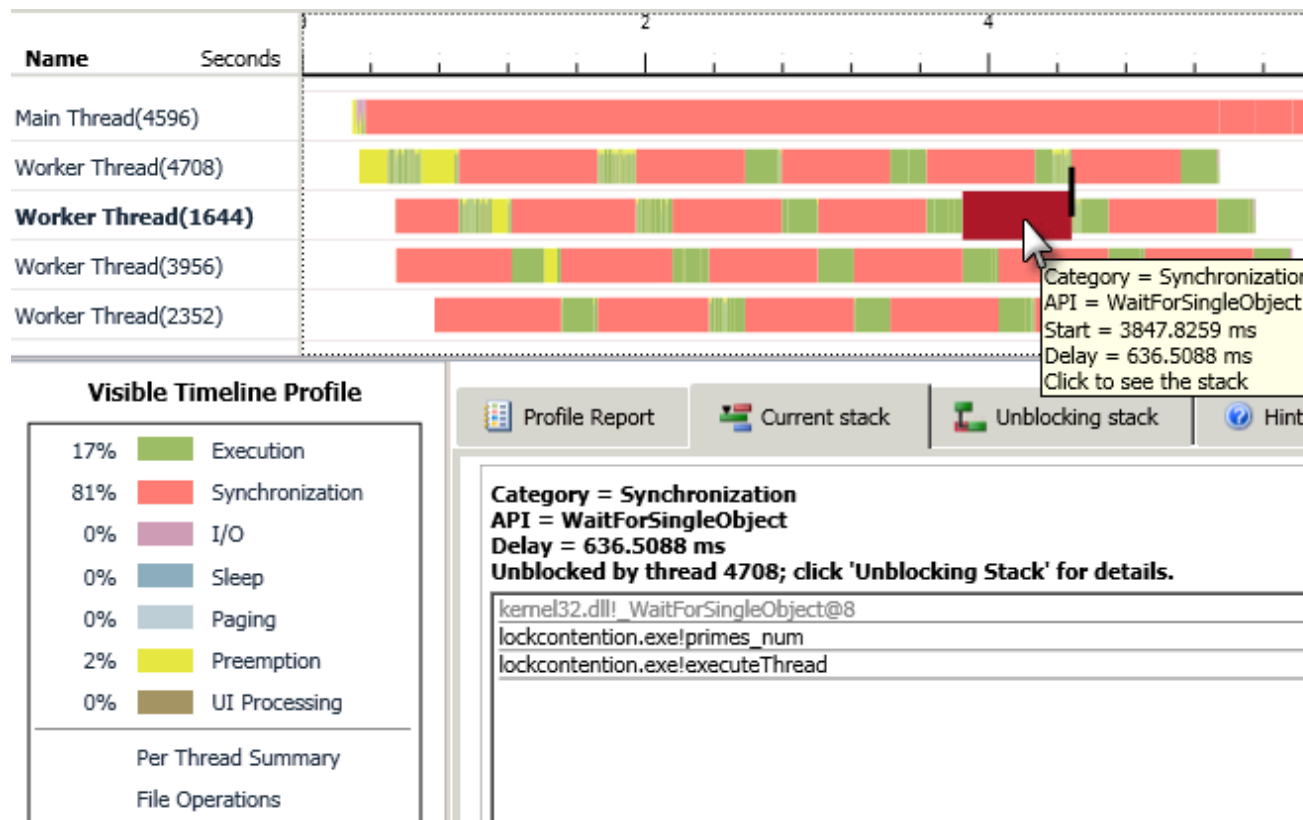
To learn more, click [Demystify...](#) and then click any item on the

To browse visual gallery of common concurrency patterns, [click](#)

- **Zoom** : use the toolbar zoom slider, ctrl & mouse wheel, or click & drag
- **Scroll** : use the horizontal scroll bar or keyboard arrows.
- **Hide/Show/Move** : Click thread names to select, and right click for
- **Select** : Clicking a green segment selects it and shows the **close** button. Clicking a segment of another color selects it and shows the **blocking** button.
- **Measure** : Click the **Ruler** button and then click & drag in the time

*Convoy Behavior*

14. Click on a red region on any of the worker threads:

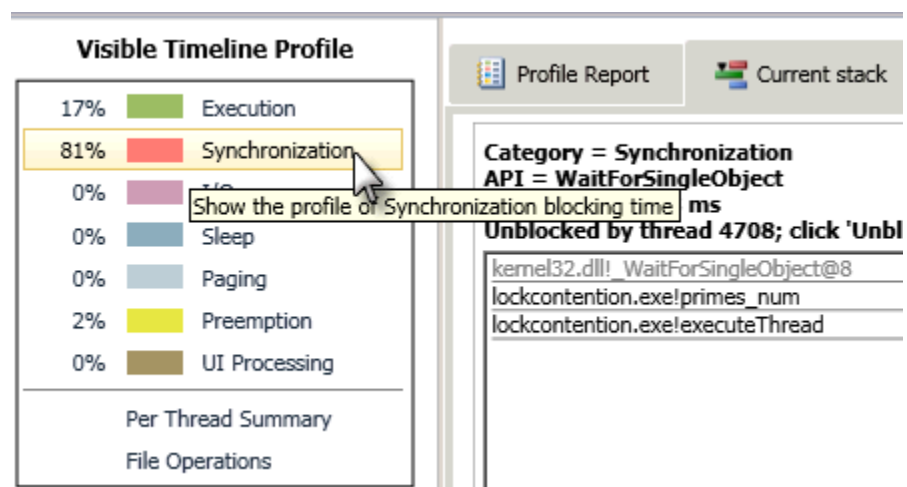


**Figure 36**

*Threads View – Clicking on a **Worker Thread** synchronization region.*

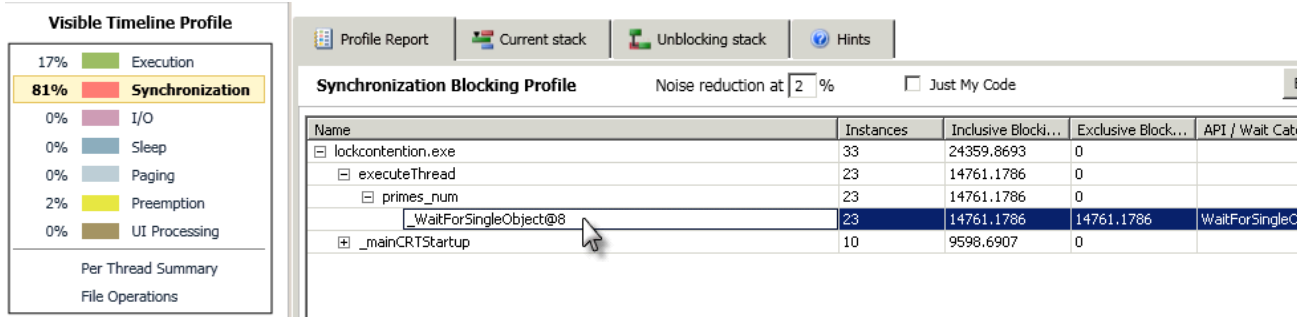
**Note:** The call stack shows us the new use of the **WaitForSingleObject** function for the Mutex implementation.

15. Double click on the **Synchronization** label for the **Visible Timeline Profile** on the lower pane:



**Figure 37**  
*Synchronization Blocking Profile*

16. Expand the tree nodes under the **Synchronization Blocking Profile**:



**Figure 38**  
*Synchronization Blocking Profile - Expanded nodes*

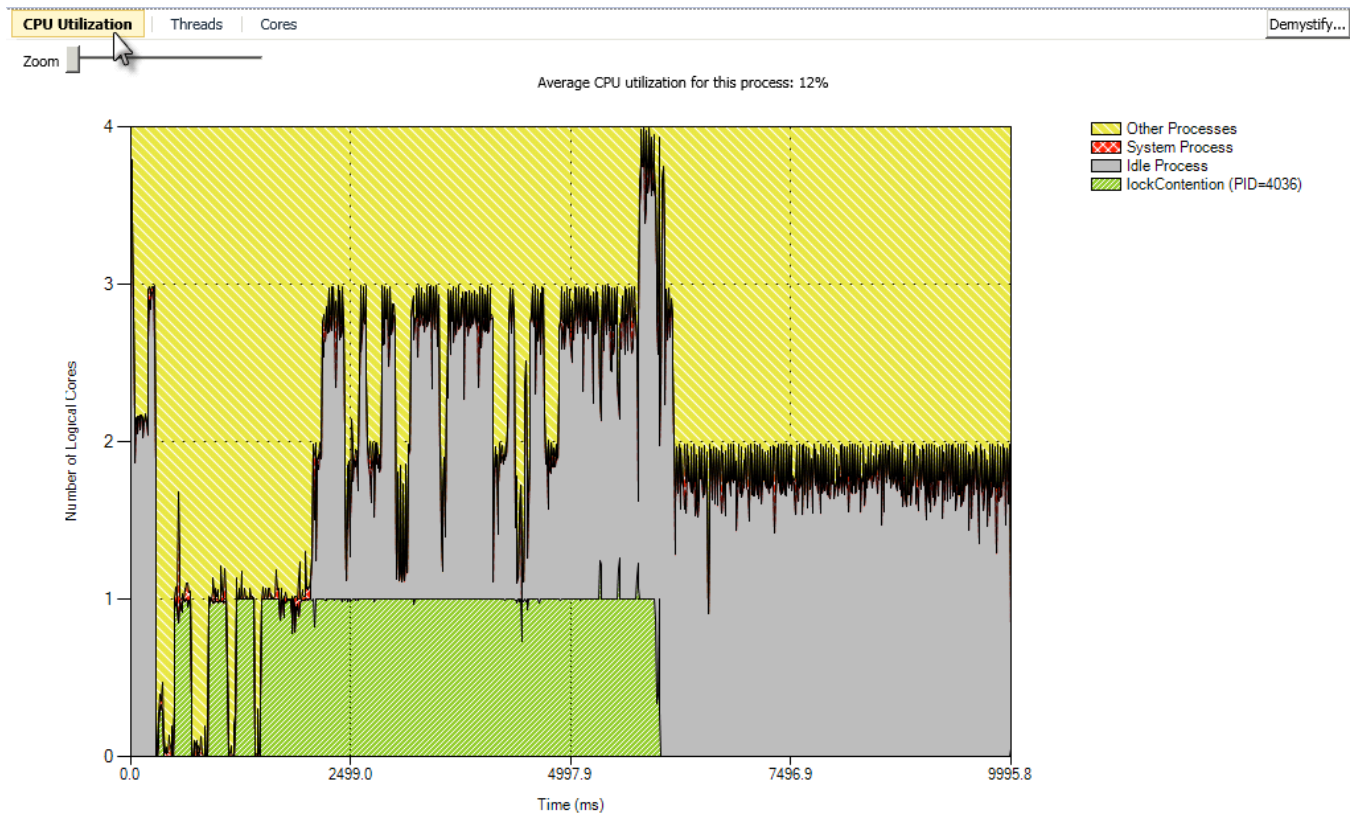
**Note:** This report establishes that there were 33 synchronization blocking events. You can see that the main thread (`_mainCRTStartup`) was responsible for 10 blocking events, while the other 23 blocking events corresponds to the slave threads **WaitForSingleObject** API call. What is important is that now you can see these 23 blocking events well distribute among the four threads execution blocks.



**Figure 39**  
*Blocking Events distribution*

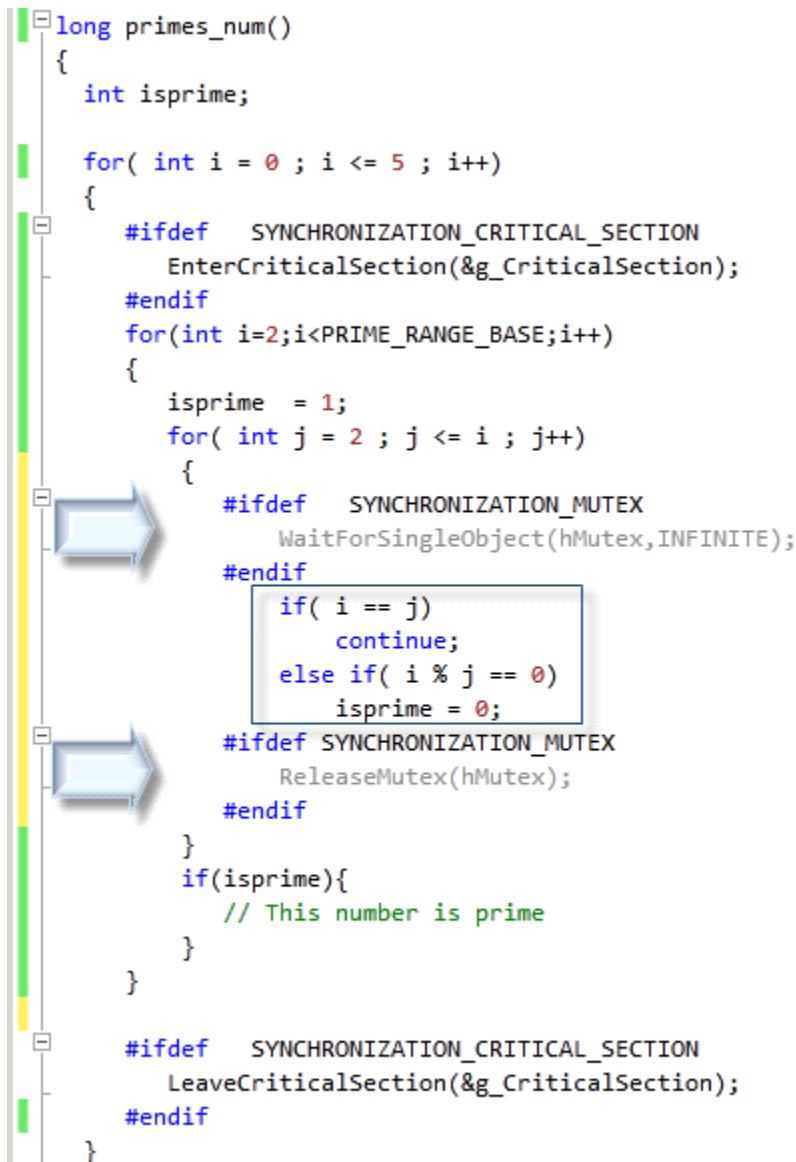
### Extra Credit

1. If you go to the **CPU Utilization** view, you will see that our application is still using the equivalent to a single logic processor. But this usage seems to be consistent.

**Figure 40***CPU Utilization view*

Go back to the code and move the use of the **mutex** to the inner part of both **for** cycles as shown in the next picture (Where the thread calculate if the number is a prime number):





```

long primes_num()
{
    int isprime;

    for( int i = 0 ; i <= 5 ; i++)
    {
        #ifdef SYNCHRONIZATION_CRITICAL_SECTION
            EnterCriticalSection(&g_CriticalSection);
        #endif
        for(int i=2;i<PRIME_RANGE_BASE;i++)
        {
            isprime = 1;
            for( int j = 2 ; j <= i ; j++)
            {
                #ifdef SYNCHRONIZATION_MUTEX
                    WaitForSingleObject(hMutex,INFINITE);
                #endif
                if( i == j)
                    continue;
                else if( i % j == 0)
                    isprime = 0;
                #ifdef SYNCHRONIZATION_MUTEX
                    ReleaseMutex(hMutex);
                #endif
            }
            if(isprime){
                // This number is prime
            }
        }

        #ifdef SYNCHRONIZATION_CRITICAL_SECTION
            LeaveCriticalSection(&g_CriticalSection);
        #endif
    }
}

```

**Figure 41***Reduce Lock Contention - Granularity*

**Note:** Analyze the effects of Lock granularity over Lock contention.

**Advice:** Reduce the amount of numbers to be analyzed by setting the PRIME\_RANGE\_BASE constant to 100. Otherwise the analysis is going to take a long time.

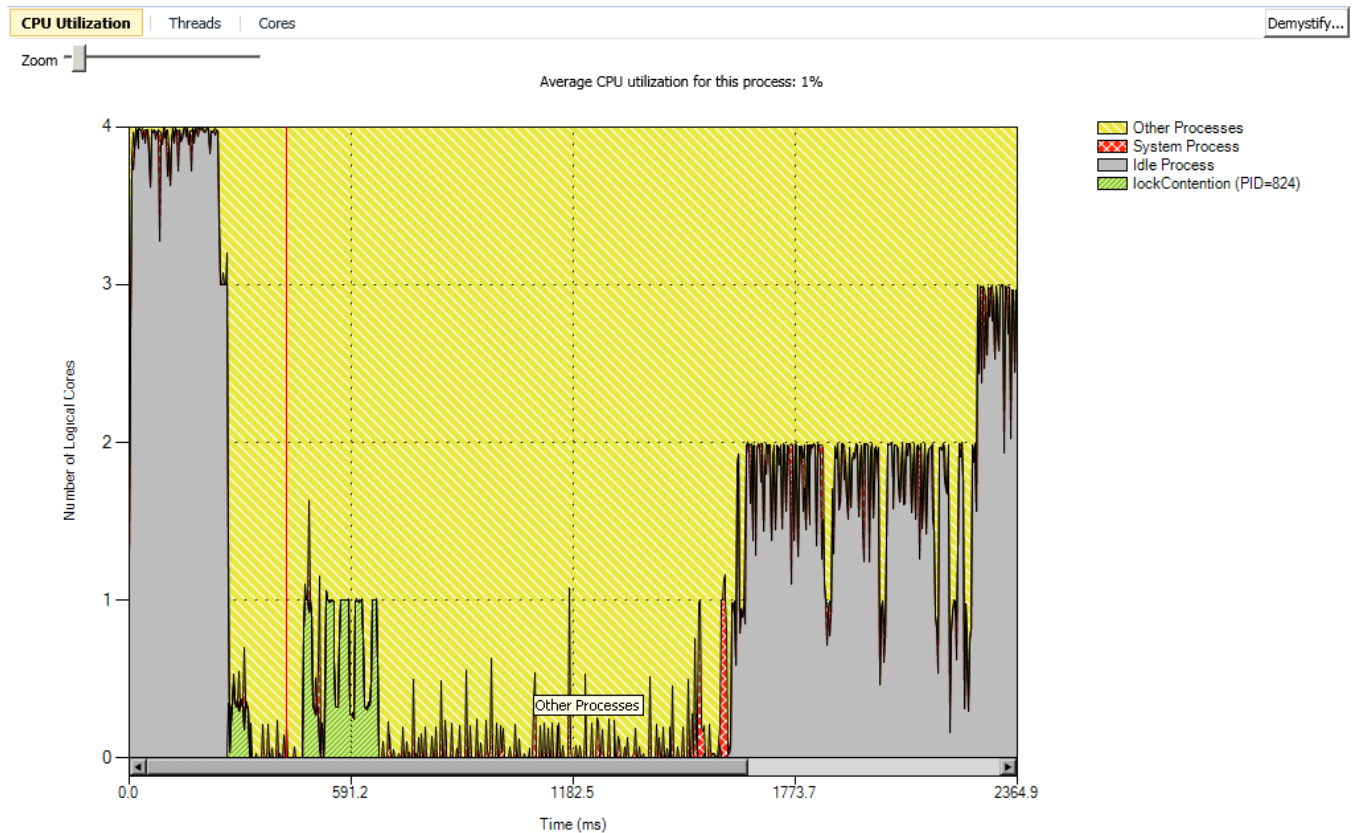


Figure 42

*Bad granularity under CPU Utilization view*



Figure 43

*Bad granularity under Threads view*

After reviewing the results, do the same using Critical Section to compare the impact of using Mutex on small fast code with highly demand, rather than taking advantage of the non-fair characteristic of the critical sections .

## Lab Summary

Critical Sections do not enforce FIFO order on the threads waiting for the lock. The first thread to find the lock in a free state is allowed to acquire it even if there are other threads waiting for the lock, known as anti-convoy support.

From a performance perspective, the critical section is faster because it reduces context switch overheads on the critical path.

Mutexes enforce fairness and FIFO order. This synchronization primitive does not have anti-convoy features.

Having a well-defined symbol path is important to have useful information in the Synchronization Blocking Profile stack.

The PPA Concurrency Visualization Tools included with Visual Studio 2010 allowed us to visualize our application making it possible for us to easily understand what was going on “under the hood” with lock contention on our application.

CPU Utilization View helped us to quickly and easily find out that our application was not making the best utilization of the available CPUs of the machine.

The threads view provided us with rich information about how our threads interact between each other. This view helped us to understand how the blocking threads waited for each other to release limited resources. Threads view helps us identify possible optimizations for the analyzed critical section code. It helped us understand lock contention and the impact of lock granularity.

During the lab, we observed the convoy behavior provided by the use of Mutexes and observed the effects of Lock granularity over Lock contention.