

Debugging in Visual Studio 2017

Student Lab Manual

Instructor Edition

Version 0.9

**Conditions and Terms of Use**

**Microsoft Confidential - For Internal Use Only**

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2017 Microsoft Corporation. All rights reserved.

**Copyright and Trademarks**

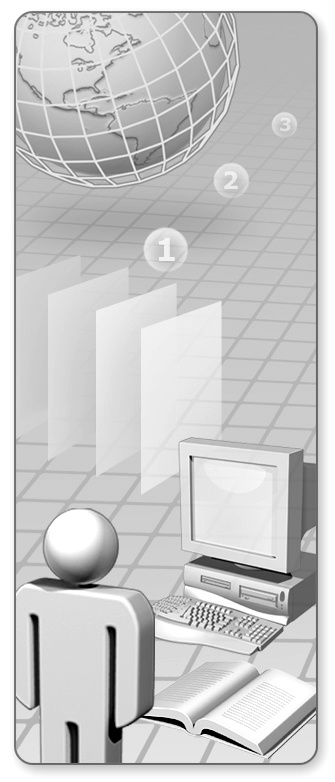
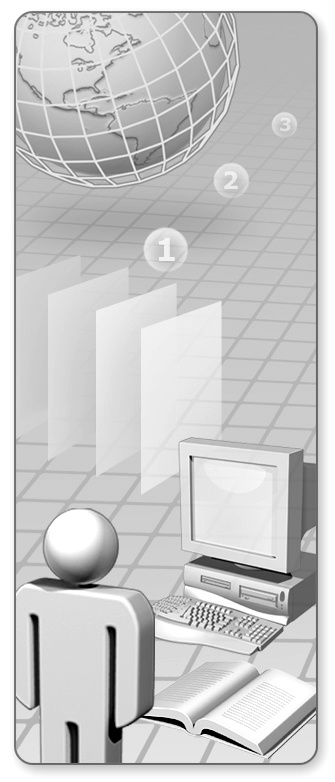
© 2017 Microsoft Corporation. All rights reserved.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see Use of Microsoft Copyrighted Content at  
[*http*://www.microsoft.com/about/legal/permissions/](http://www.microsoft.com/about/legal/permissions/)

Microsoft®, Internet Explorer®, and Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

Debugging in Visual StudioLogo_Services_black_300 2017

# Debugging in Visual Studio 2017

#### Introduction

In this lab you will go over some of the essential debugging activities in Visual Studio 2017. The lab will provide you with the basic knowledge about the Visual Studio debugger, its features, and how to use them.

#### Objectives

After completing this lab, you will be:

* Familiar with the features of Visual Studio debugger
* Able to use these features to effectively debug applications

#### Prerequisites

Visual Studio 2017 installed.

#### Estimated time to complete this lab

45 minutes

#### For more information

See the MSDN documentation: “Debugging in Visual Studio“:

<http://msdn.microsoft.com/en-us/library/sc65sadd.aspx>

## Exercise 1: Debugging an Application in Visual Studio

#### Scenario

The application, WordCount, used in this lab is a console application written in C# and it targets .NET Framework 4.5. The application simply takes an input text file and counts the number of occurrences of a search word in the file and then it displays the results. The application is written to take different approaches to achive this:

* processing each line of text from the input file one by one in a synchronous fashion and,
* using threads and tasks to take advantage of an opportunity to do parallel processing

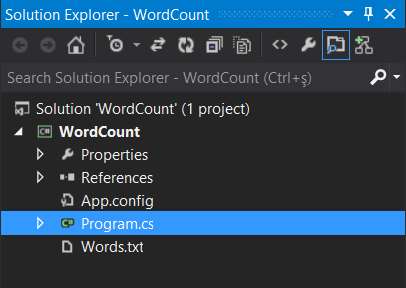
In this exercise, we will go over some of the features of Visual Studio debugger and perform common tasks to get familiar with these features and use them.

#### Prerequisites

No prerequisite steps.

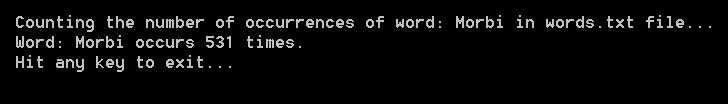
Get Familiar with the Lab Project

1. Start Visual Studio 2017.
2. Open WordCount solution (WordCount.sln) from the folder for this module.
3. WordCount solution contains only one project: WordCount which is the console application. The project contains Words.txt file which is the input file and program.cs that contains the code. Take a look at the project and the code to get yourself familiar with it.



The code in Main method in program.cs basically reads the entire contents of Words.txt into a variable called lines of type IEnumerable of string and then calls CountSearchWord method which basically iterates over each line in lines and counts the occurrences of the search word which is specified in searchWord variable in the code. You will take notice in Main method of calls to two methods that’re commented out. We will use these methods later in the lab. For now, you can leave them as is commented out.

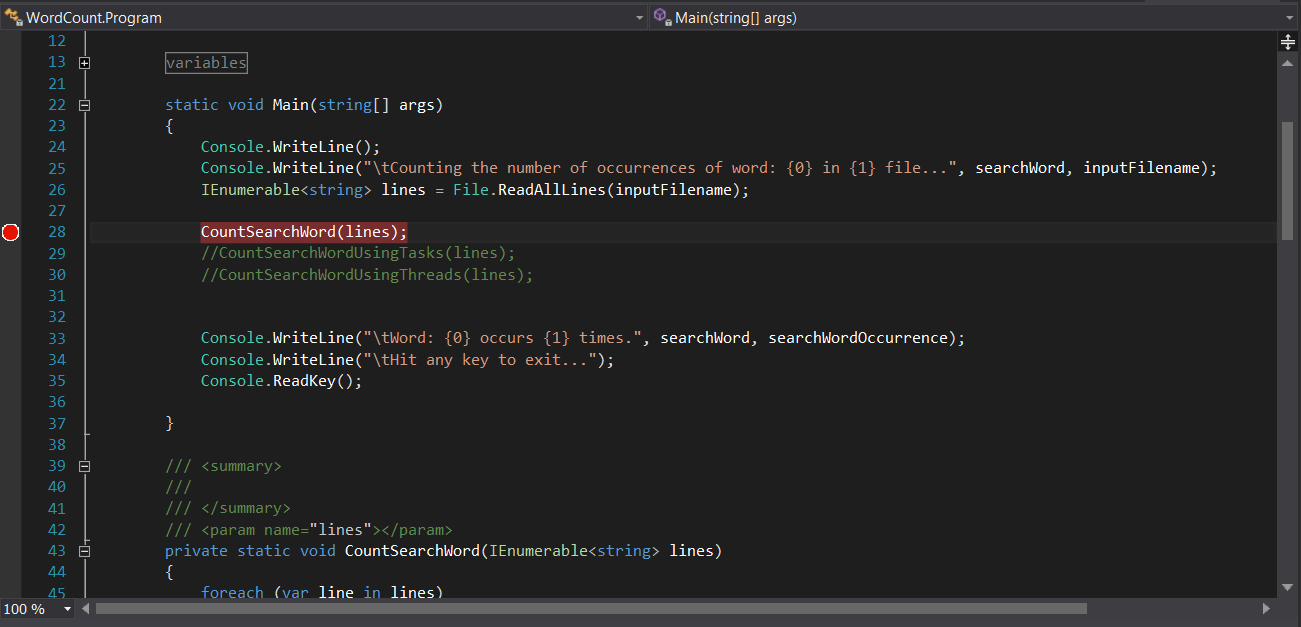
1. Compile the project and run it. The solution should compile without any errors and you should see the following output:



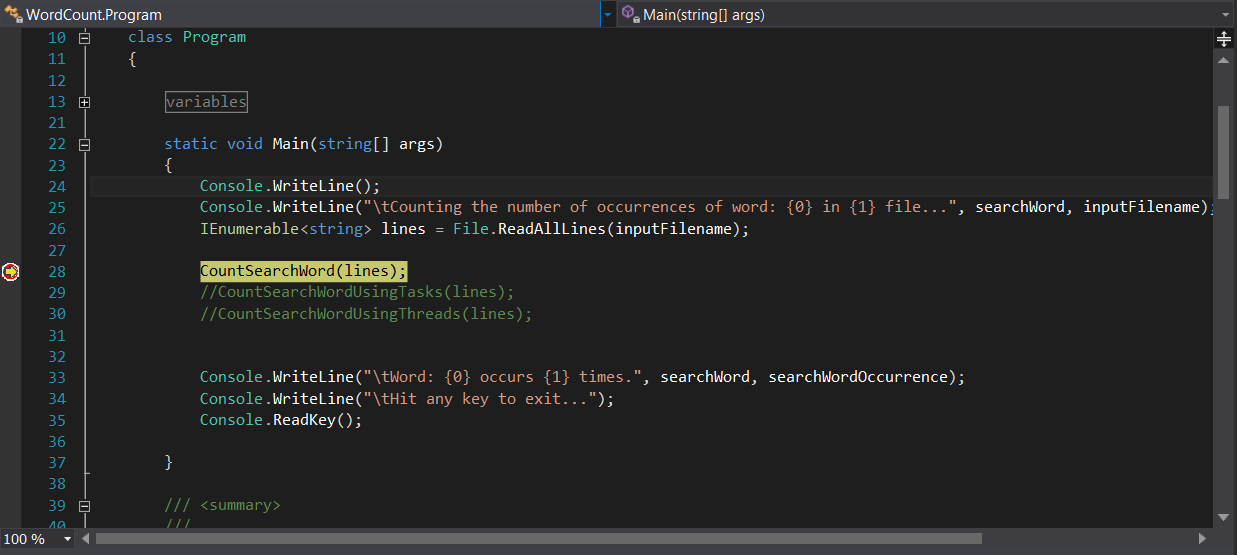
1. Hit any key to exit the program and go back to Visual Studio.

Perform Essential Debugging Activities

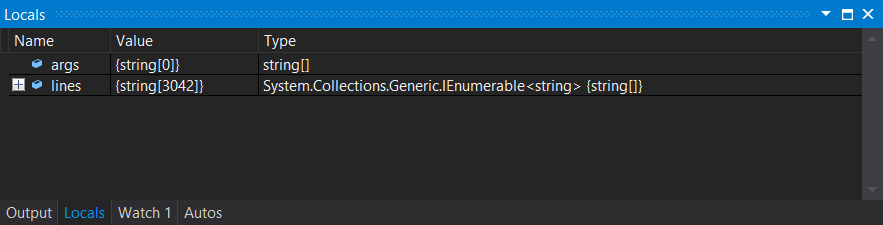
1. Set up a breakpoint on CountSearchWord method in Main method in Program.cs file. You can do this by putting the cursor anywhere on CountSearchWord line and then hitting F9. Alternatively, you can right click anywhere on that line and select Breakpoint🡪Insert Breakpoint.

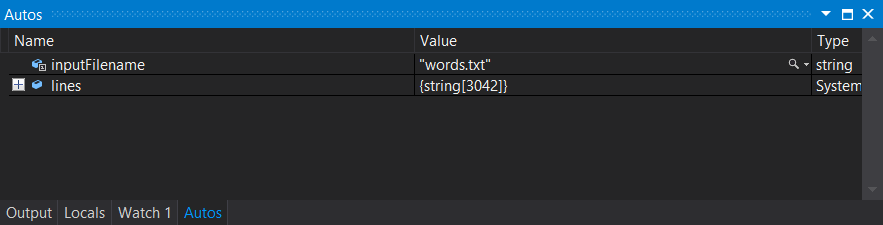


1. Hit F5 or click Start in the Visual Studio toolbar to start debugging the application.Visual Studio debugger should almost immediately break into your application:



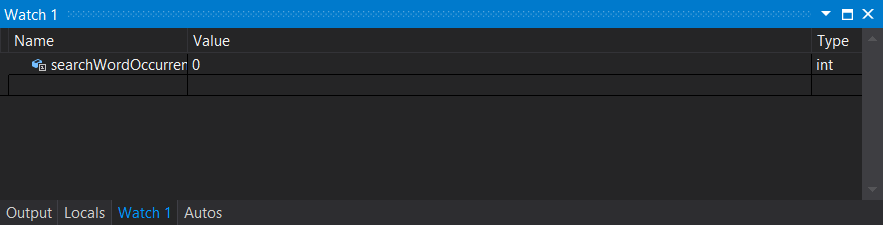
1. Take a look at the contents of locals and autos windows. If you don’t see these windows, you can bring these windows into view from Debug🡪Windows menu.





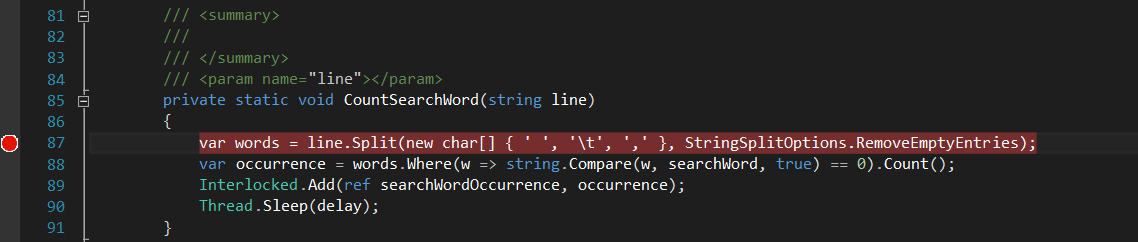
The Autos window displays variables used in the current statement and the previous statement. The locals window on the other hand displays variables local to the current context. Expand lines to inspect the data read from the input file.

1. Add a watch for searchWordOccurence variable. There are multiple ways to do this. You can go to Watch 1 windows and edit searchWordOccurrence in there or generally the name of the variable you are interested in. Alternatively, you can find the place in the code where searchWordOccurrence is declared or used, right click and select Add Watch.

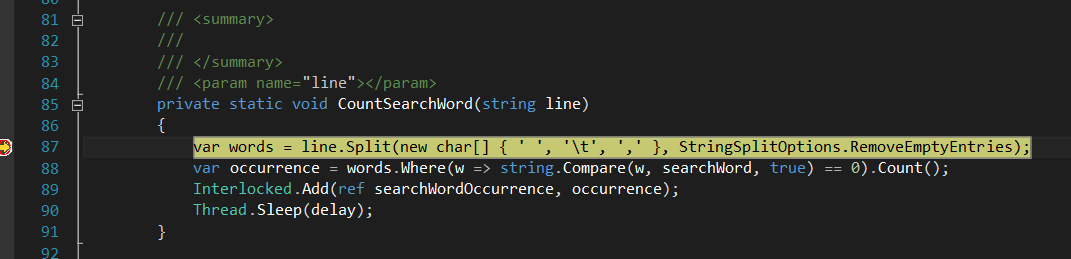


Since the program so far only loaded the input file, searchWordOccurrence is zero.

1. Add a breakpoint in CountSearchWord(string line) method. The program is still running but it is in break mode. Find CountSearchWord method and add a breakpoint on the first line of code in this method by placing the cursor on the line and hitting F9. You can also click on the gutter area of the code editor on the left corresponding to the line of code where you want to set the breakpoint.



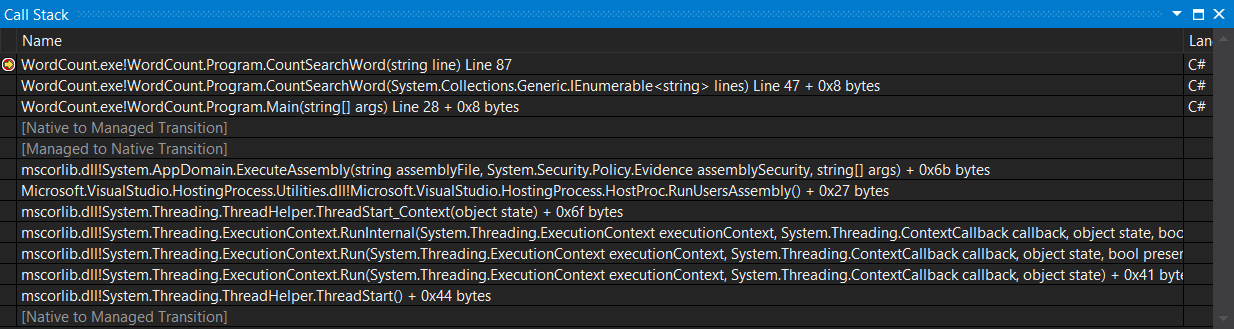
1. Hit F5 or Continue button in Visual Studio toolbar to continue execution. You will hit the breakpoint in CountSearchWord method that you just added.



1. Take a look at the locals and autos windows. Notice that the contents of these windows have changed based on the current statement and the execution content. The watch window still contains searchWordOccurrence that you added earlier.

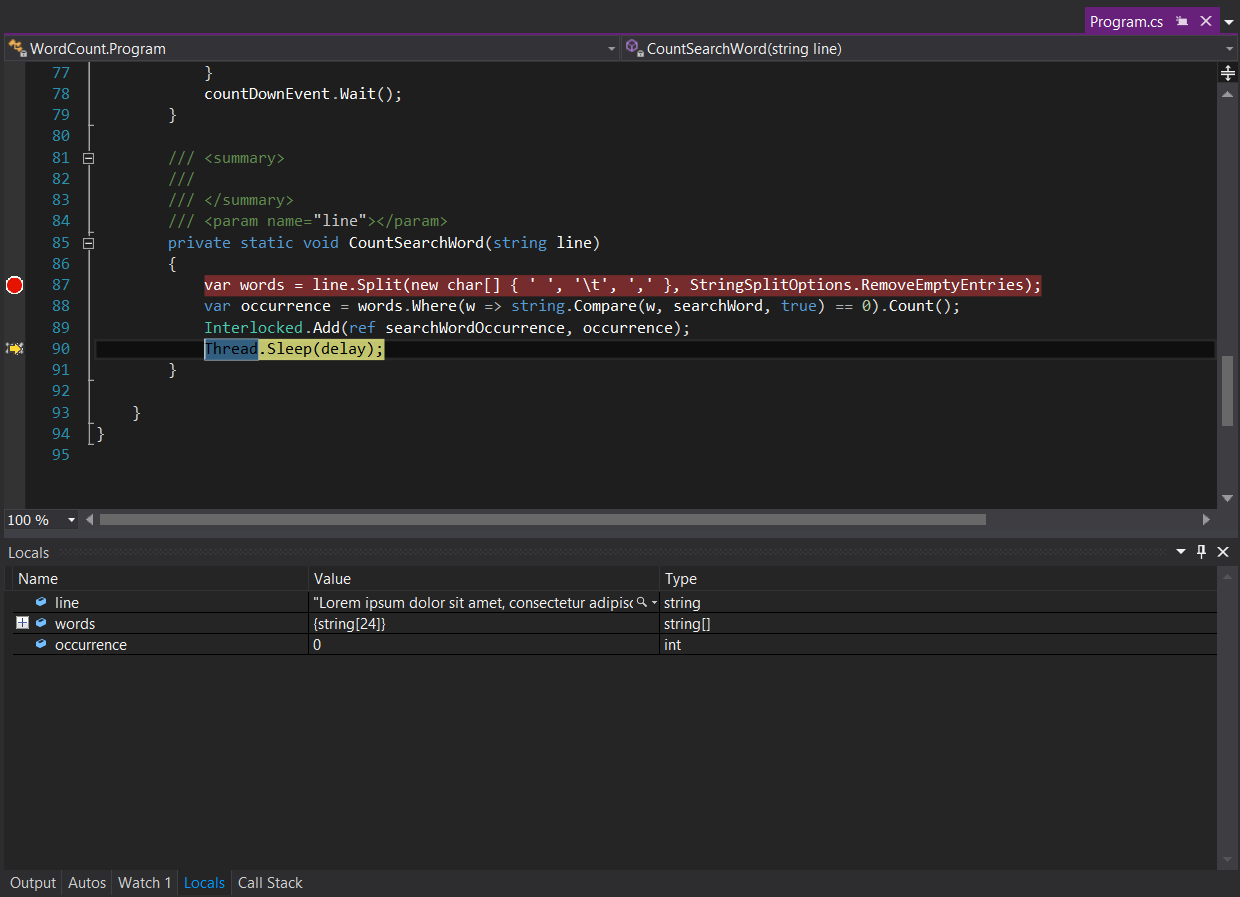
View Call Stack

1. View call stack to see the chain of method calls that led the execution to the current statement. If you don’t see the call stack window, you can bring it into view from Debug🡪Windows menu.



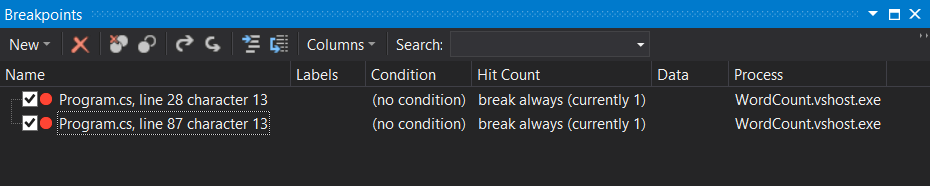
You can double click on a line in the call stack window to go to that method. When you do that, you will notice that the locals and the autos window change as well to show the variables based on the selection.

1. Click F10 to step over to execute the program line by line. Hit F10 until you reach Thread.Sleep method. If you take a look at the locals and the autos windows, you will notice that some of the variables will start to have values such as words.



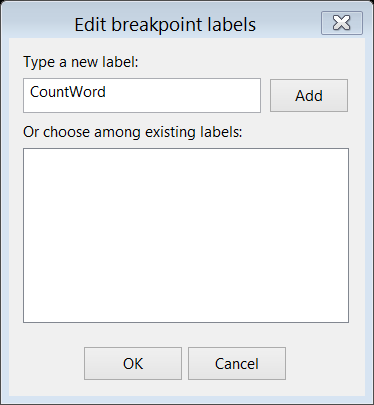
Work with Breakpoints Window

1. Display breakpoints from Debug🡪Windows menu. So far we added two breakpoints: first one before starting the program and the other in the break mode. You can manage these breakpoints by going where they are placed in the code and right clicking to bring up a context menu to see ways to interact with them. However, this way you get to see and work with only one breakpoint. If however you use breakpoints window, you can see all of your breakpoints and manage them in one convenient place.

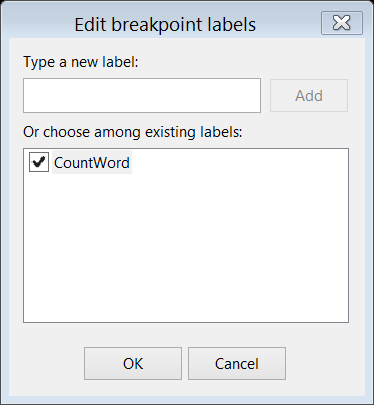


Notice you can see two breakpoints you added so far along with other information such as location, condition, hit count, etc. You can also easily enable, disable, add, delete breakpoints.

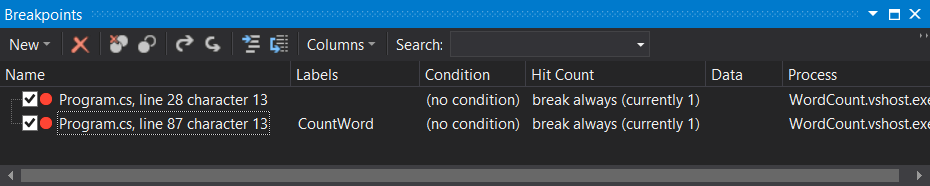
1. Right click on the last breakpoint and select Edit Labels.. from the context menu. This will bring the window shown below. Enter CountWord to add a label called CountWord to this breakpoint.



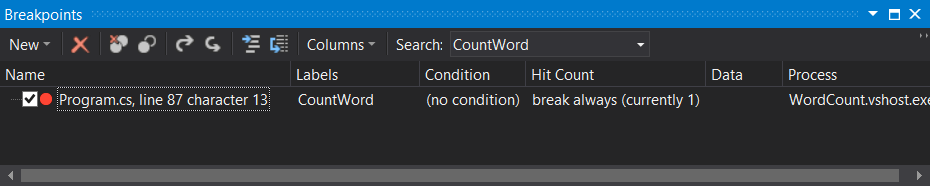
1. Enter CountWord to label the breakpoint, click Add, and then OK.



Notice in the breakpoints window that the label you added appears in the label column for the breakpoint.



1. Enter CountWord in the search box in the breakpoints window and hit enter. You should see that only the breakpoint with CountWord label is displayed. Labelling breakpoints is a good practice to organize and find your breakpoints especially if you happen to have a lot of breakpoints.

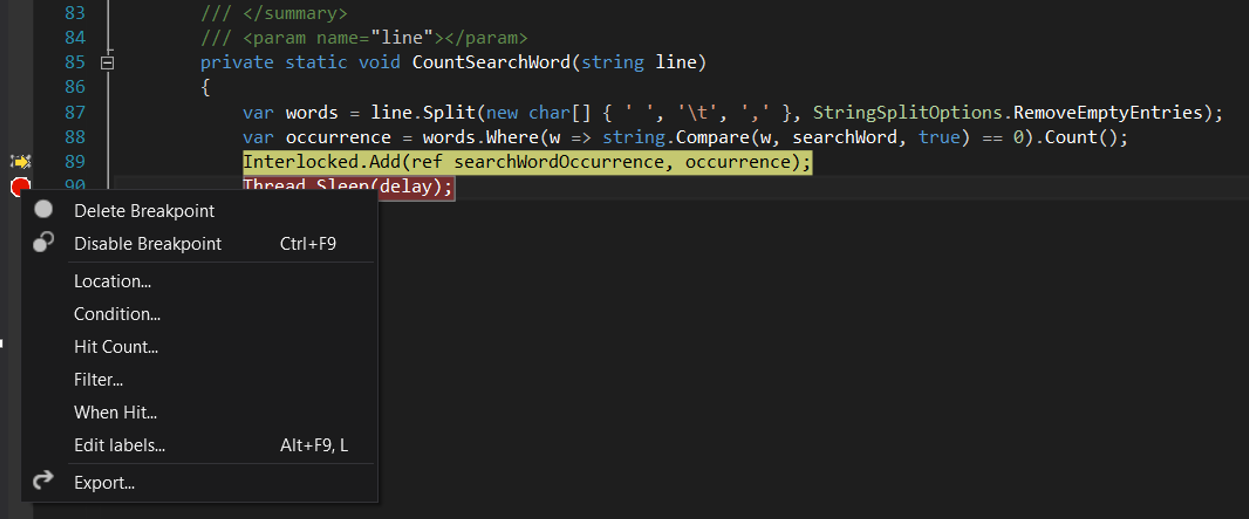


Clear the search word and hit enter to bring back the unfiltered view of breakpoints.

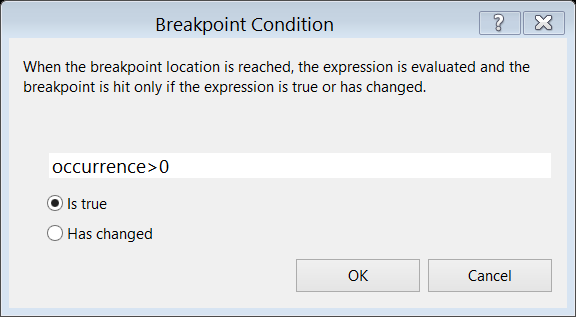
Set up a Conditional Breakpoint

If you click F5 or Continue, the program will continue execution and keep hitting the breakpoint you added in CountSearchWord method. Considering the fact that this method gets called for each line in the input file, it may be time consuming and tedious to hit F5 and break until we see the conditions that we are interested. Instead, we can specify a condition for the breakpoint so that only when that condition occurs, the Visual Studio debugger will break. The condition we are interested in is when the search word is found in a line of text from the input file. Notice that we are still debugging the application. We did not stop the debugger.

1. Remove the breakpoint you added earlier in CountSearchWord method. You can do this in the breakpoints window, placing the cursor on the line where the breakpoint is and hitting F9, or simply clicking on the breakpoint glyph in the gutter of the code editor.
2. Add a breakpoint on Thread.Sleep statement in CountSearchWord method.
3. Right click on the newly added breakpoint in the gutter and select Condition from the context menu.

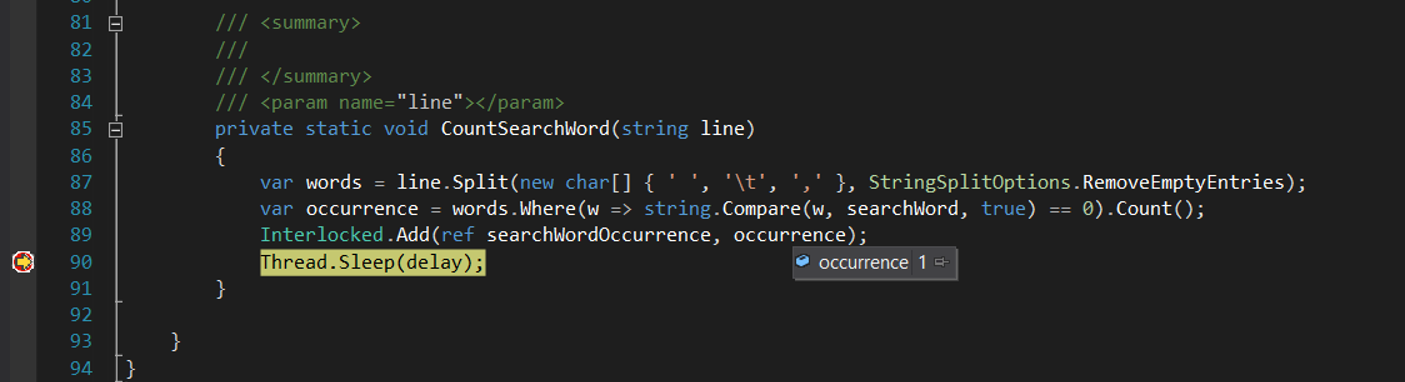


1. Fill in the Breakpoint Condition window that appears as shown in the screenshot below and then click OK.



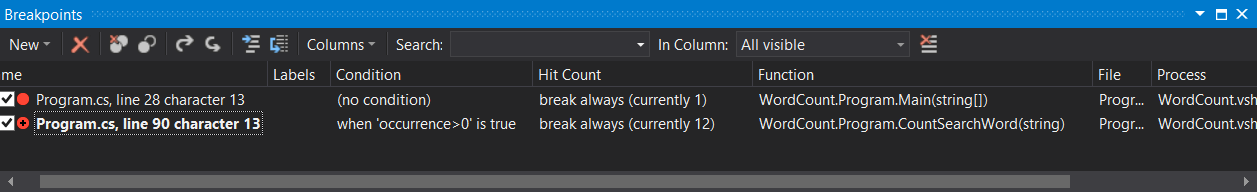
occurrence is a local variable declared in CountSearchWord method for the number of occurrences of the search word in the line of text that’s currently being processed. In the condition, we are basically instructing the debugger to break into execution only if occurrence is greater than 0. In other words, whenever the currently processed line of text contains the search word.

1. Hit F5 or Continue. The debugger should soon break into execution.
2. Hover over the variables: occurrence, searchWord, line, and words in CountSearchWord method to view the data that contain. Notice that the value occurrence contains is non zero.



Note that you can add these variable to the watch window or use the locals and the autos windows as well. There are various ways of inpecting variables.

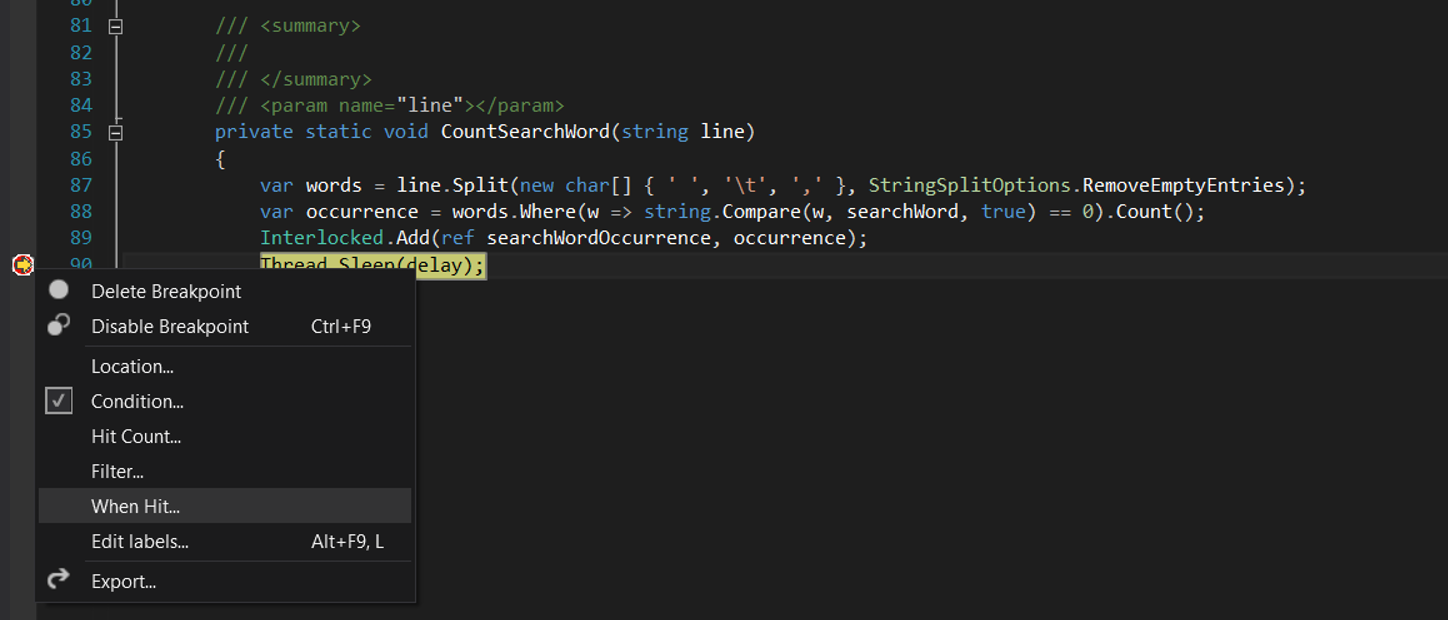
1. Hit F5 several times. When you break, take a look at the breakpoints window specifically the condition and the hit count columns. Hit Count column for the breakpoint with the condition should tell you how many times you have hit that breakpoint in the current debugging session so far.



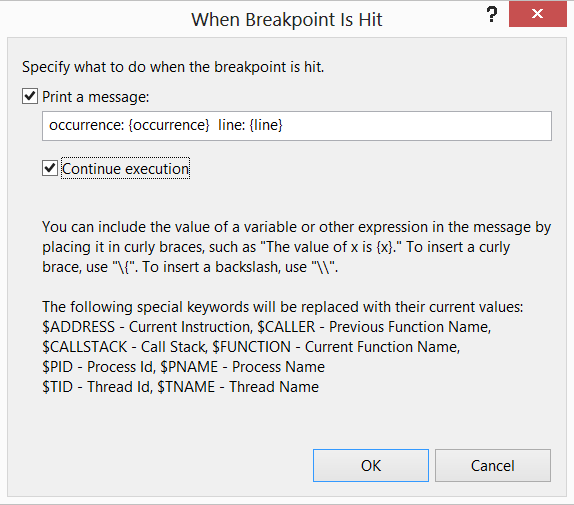
Set a Tracepoint

Sometimes instead of breaking when a breakpoint is hit, you may want to take some action and let the program continue to run. A Tracepoint, special kind of breakpoint, allows you to do this. In the previous step, you specified a breakpoint with a condition. Here you will convert that breakpoint into a tracepoint by specify an action to perform when hit instead of breaking. Notice that you are still debugging the program.

1. Right click on the breakpoint in the gutter set on Thread.Sleep statement in CountSearchWord method and select When Hit from the context menu.

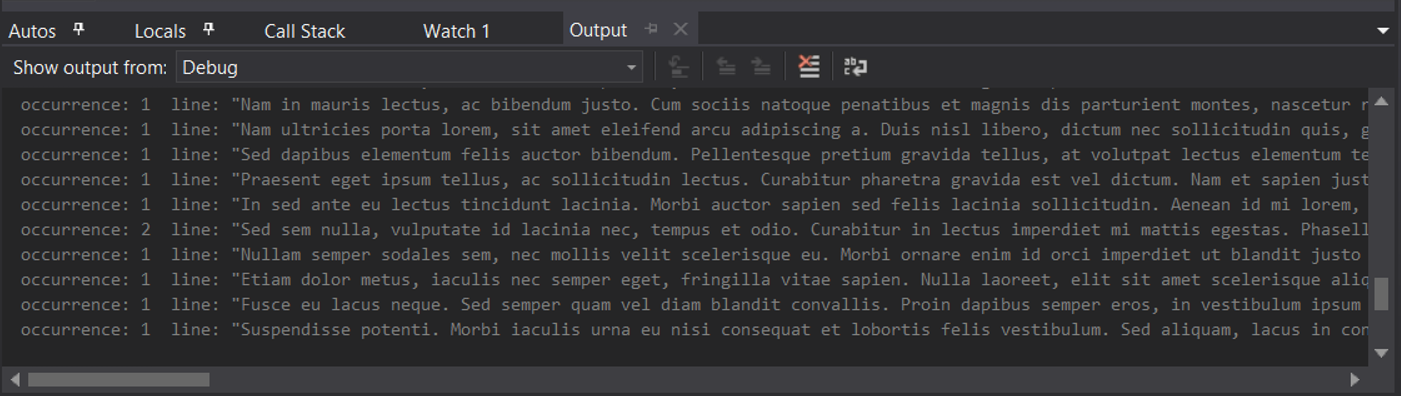


1. Fill When Breakpoint is Hit window as shown in the screenshot below and click OK. While you have this window open, read the help text to see some of the special keywords such as $PID, $TID that you can use in the message to be printed.



In Print a message box, local variables, occurrence and line, defined in CountSearchWord method are printed. You can see them in the output window when you hit F5 or continue execution. If you don’t have the output window in view, you can select Output from View of Visual Studio.

1. Bring Output window into view and hit F5 or continue. You should see the messages as the tracepoint is hit.



Note that you will probably notice of a slowdown. Conditional breakpoints and tracepoints slow the execution down. You don’t need to wait until the program completes. You can stop it after a while.

1. Stop debugging the program.
2. Delete all breakpoints. You can use the breakpoints window to do this.
3. End of exercise.

## Exercise 2: Debugging a Multithreaded Application

#### Scenario

Application code runs on threads. As you debug your application, sooner or later, you will need to deal with threads that are created either by you explicitly or by the runtime implicitly. Although debugging a multithreaded application can be challenging, Visual Studio provides rich specialized tools to make this process easy. In this exersice, you will work on the same project from Exercise 1. To remind you again, this project contains a console application that counts the number of times a search word appears in an input text file. Originally, the console application reads the whole input file and processes each line sequentially. However, this calculation can be implemented by taking advantage of data parallelism. Each line of text from the input line is independent of each other. So each line can be processed concurrently. To achieve this, two approaches are taken: one that uses the .NET thread pool to create worker threads and the other one uses the Tasks that are introduced in .NET 4.

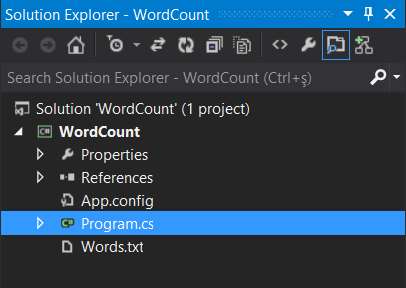
#### Prerequisites

Visual Studio 2017 installed.

Get Familiar with the Lab Project

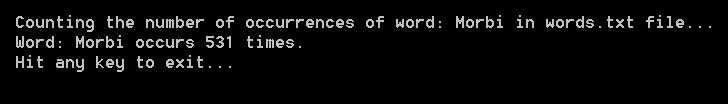
If you have already completed the previous exercise, you can skip this part and go to Use Threads part.

1. Start Visual Studio 2017.
2. Open WordCount solution (WordCount.sln) from the folder for this module.
3. WordCount solution contains only one project: WordCount which is the console application. The project contains Words.txt file which is the input file and program.cs that contains the code. Take a look at the project and the code to get yourself familiar with it.



The code in Main method in program.cs basically reads the entire contents of Words.txt into a variable called lines of type IEnumerable of string and then calls CountSearchWord method which basically iterates over each line in lines and counts the occurrences of the search word which is specified in searchWord variable in the code. You will take notice in Main method of calls to two methods that’re commented out. We will use these methods later in the lab. For now, you can leave them as is commented out.

1. Compile the project and run it. The solution should compile without any errors and you should see the following output:

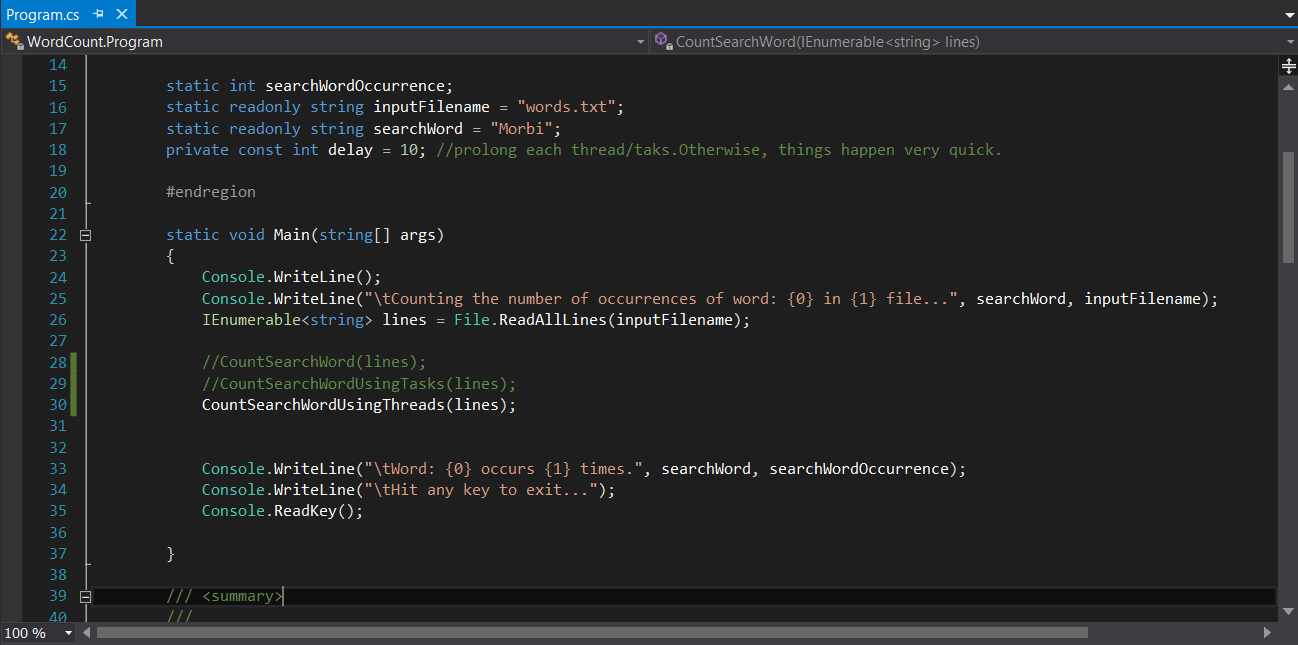


1. Hit any key to exit the program and go back to Visual Studio.

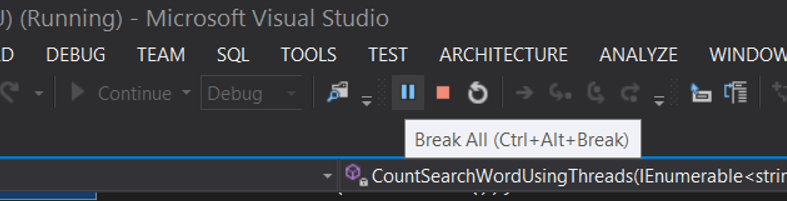
Use Threads

**IMPORTANT:** **Due to the nature of multithreading, some of screenshots in this portion of the lab specifically the ones taken during debugging may be different than the ones you see. So don’t be alarmed. Understand what the lab step is trying to show you and perform that action in your debugging session. If you have any issues or are not so sure, please ask the instructor for assistance.**

1. Make code changes to use threads. By default the application uses a sequential implementation by calling CountSearchWord method in Main method in program.cs. You need to make changes to call CountSearchWordUsingThreads method. Comment out the call to CountSearchWord and uncomment the call to CountSearchWordUsingThreads. When you are done, Main method should look like as shown below:



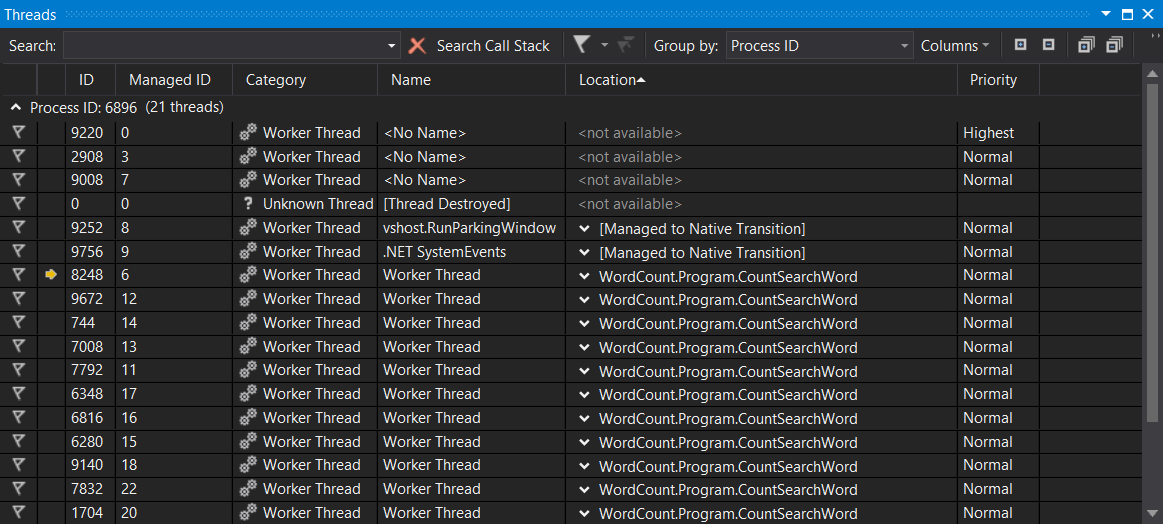
1. Take a look at CountSearchWordUsingThreads method to understand what the code is doing. This method basically queues a work for each line of text to the .NET Thread pool. The work is specified as an anonymous method in lambda expression and it calls CountSearchWord. Also note the use of CountdownEvent to wait for all worker threads to complete their work.
2. Build the project and debug it by hitting F5 or Start. Notice that you have not set up a breakpoint yet.
3. Let the application run for a little while.
4. While the application is still running, switch back to Visual Studio and click on Break All button in the toolbar of Visual Studio.



This is another way to break into execution. If you are debugging an application in Visual Studio and experience an unexpected strange behavior such as a hang or delay, you can go to Visual Studio and break all to break the execution to see what the application is doing.

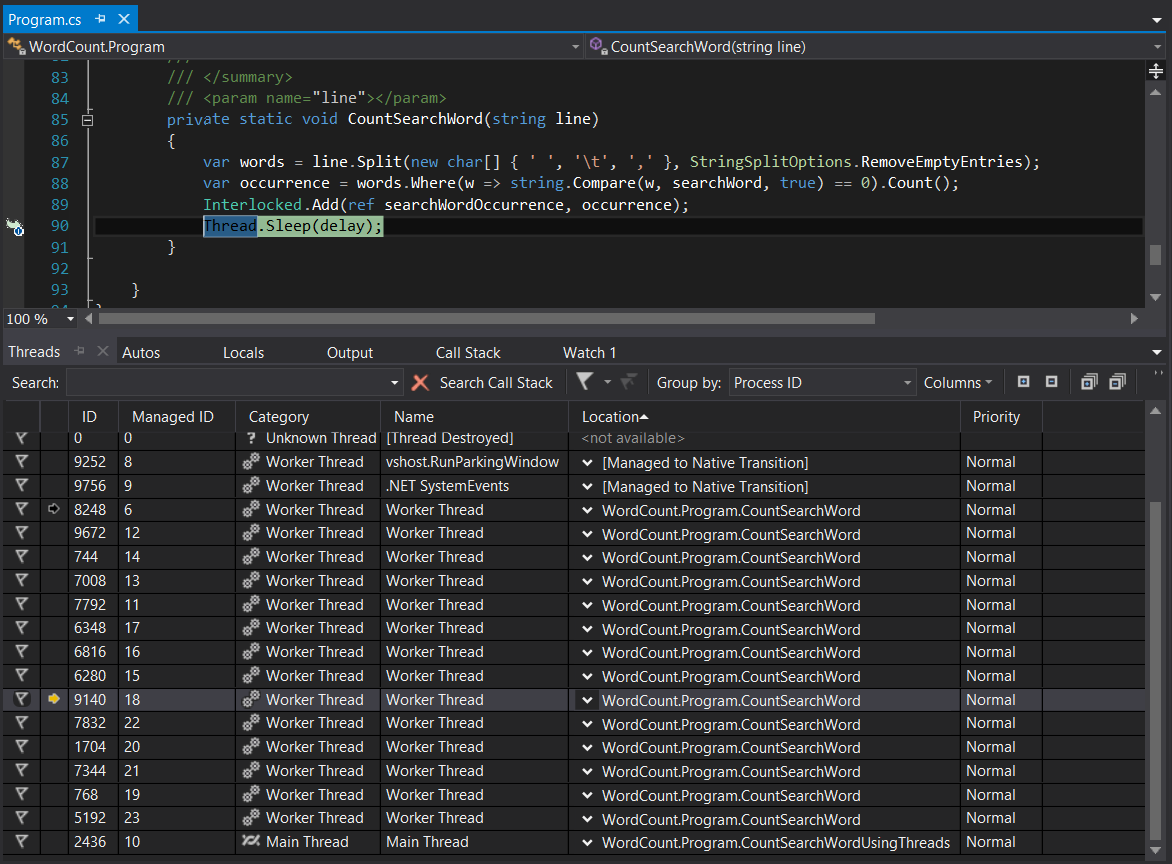
Note that if you are slow to break the execution and the application completed. Simply repeat the steps to run the application and break execution somewhere in the middle. Alternatively, there is a variable called delay in Program class. The value of this variable is in milliseconds and is used to put each thread to sleep to introduce a delay. You can increase the current value to a larger one.

1. Bring up the threads window from Debug🡪Windows menu.



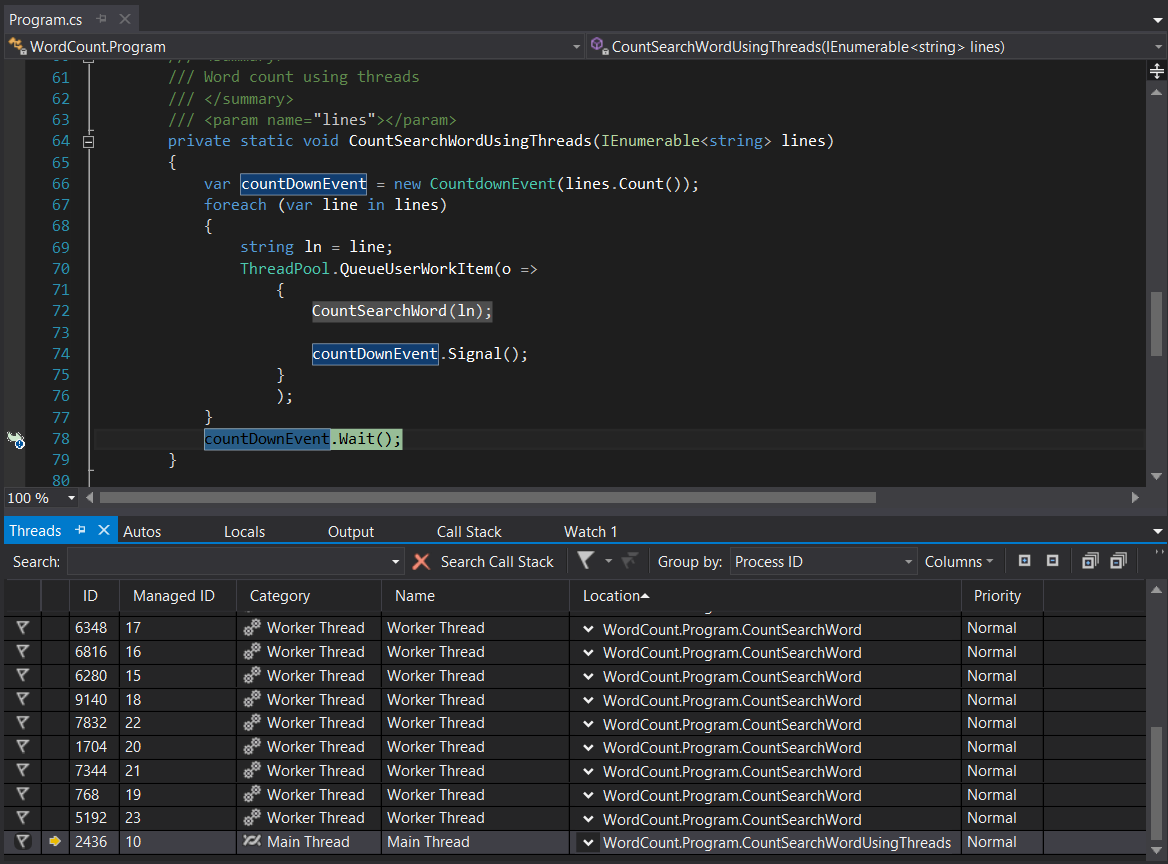
Note that you will most likely see something different than the above screenshot. However, you should still see a bunch of threads running and doing work.

1. Spend some time to understand the threads window. This window shows you all the treads running in your application along with a lot of information such thread id, category, name, location, priority, etc. Worker threads are the ones that we queued to count the search word.
2. Double click on a worker thread to see what it is executing.



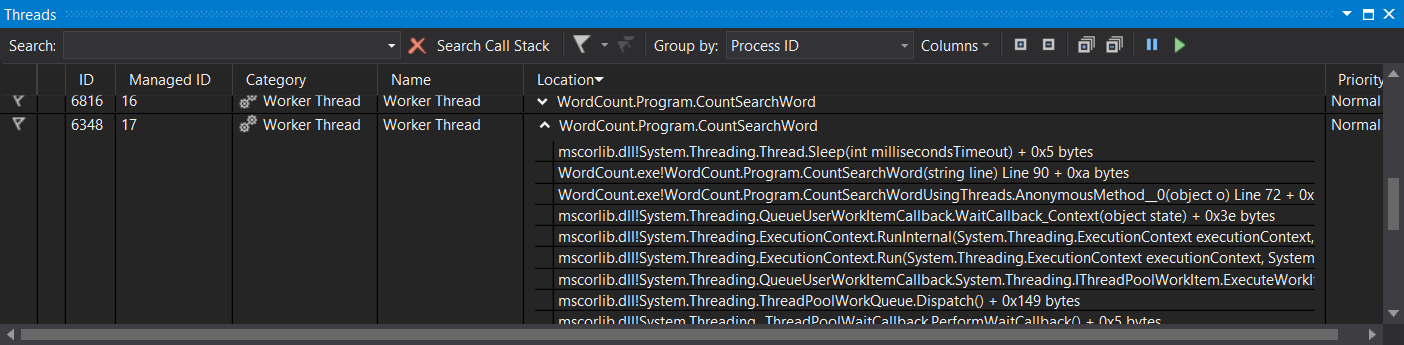
The worker thread will likely be waiting on a call to sleep to return.

1. Double click on the thread named Main Thread to see what this thread is doing. This is the main application thread.

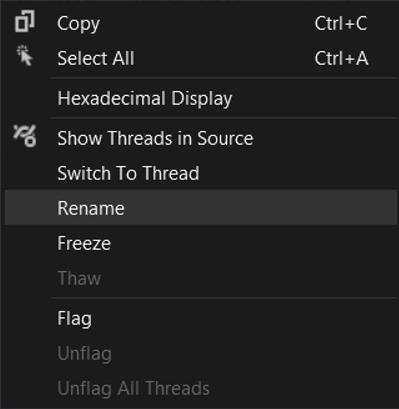


As you can see, the application is waiting for all of the worker threads to complete their work.

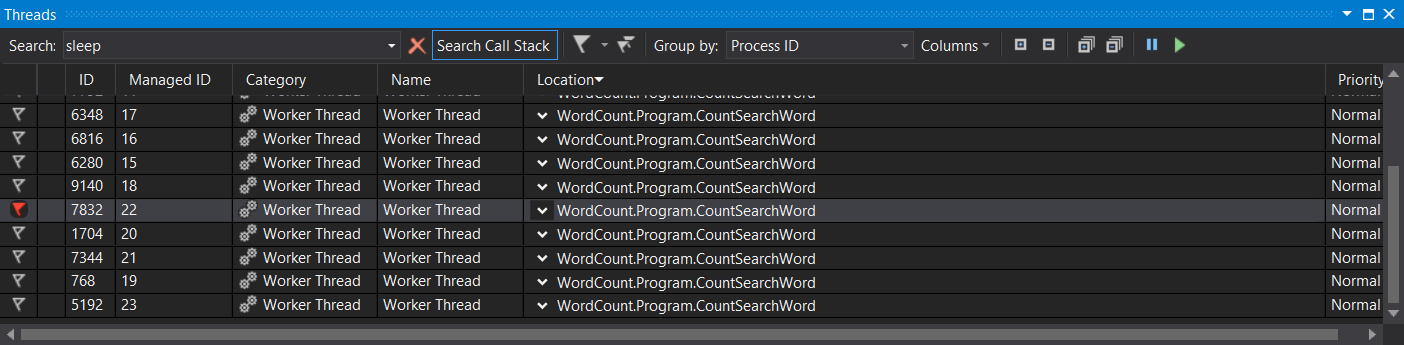
1. Hover over the location data for a worker thread or click on down arrow to see the call stack for that thread.



1. Right click on a thread to see the other actions that you can take.

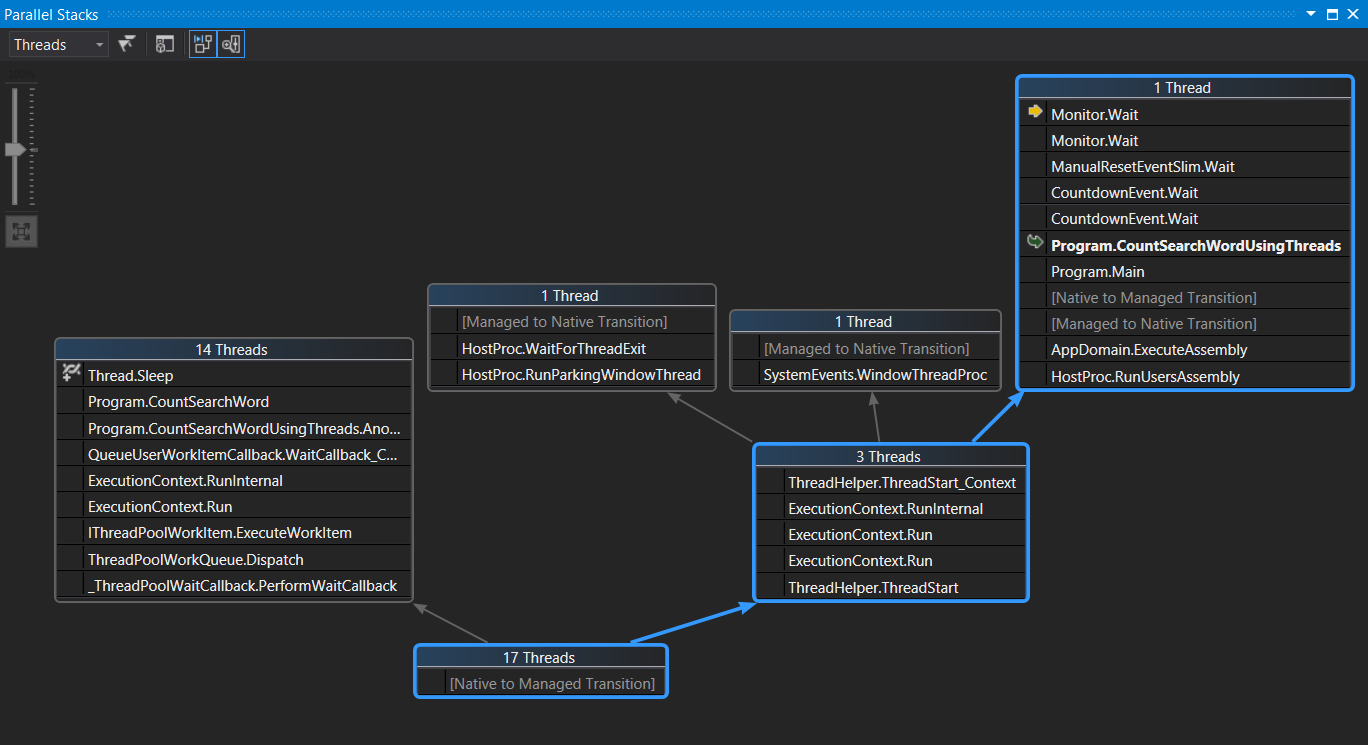


1. Flag a thread. You can flag a thread in the threads window to give that thread a special attention because it may be of importance to you. To do this simply click on the flag icon in the threads window to flag it.



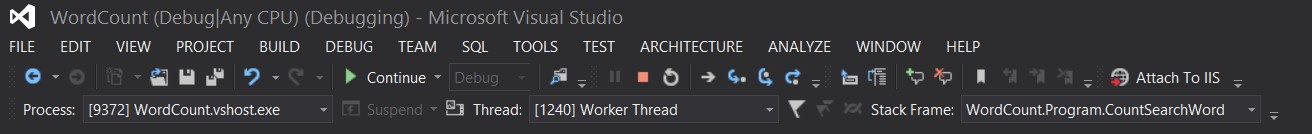
When you flag a thread, the flag icon turns read. To unflag it, simply click on the flag icon again and the flag icon will become blank.

1. Bring up parallel stacks window from Debug🡪Windows. This is a very useful interactive view that visualizes very nicely the running threads.



This window shows different threads that are currently running. You get to see the number of threads, the code path (arrows), call stack, etc. This is an interactive window. You can click on thread block to select it, hover over to see the threads in the tooltip, double click on a line in the call stack to go to that place in the source code. There are also commands available in the toolbar of the parallel stacks window and also menus that come when you right click on threads. Take some time to explore the parallel stacks window to get familiar with its features.

1. Bring up Debug Location toolbar. Right click somewhere on the Visual Studio toolbar to bring up a list of toolbars that you can add to the Visual Studio Toolbar. Select Debug Location from that list.



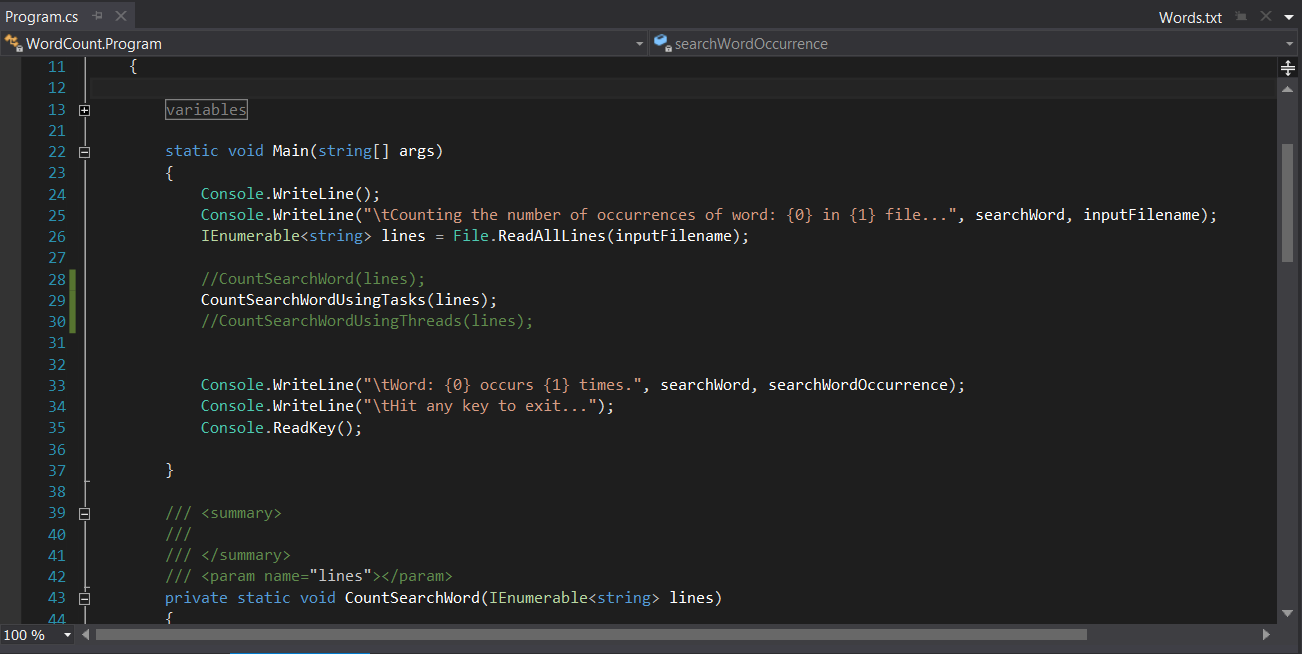
The debug location toolbar allows you to conveniently access process, thread, and call stack information during debugging. You can switch thread, view call stacks, flag/unflag, and filter threads.

1. Stop debugging.

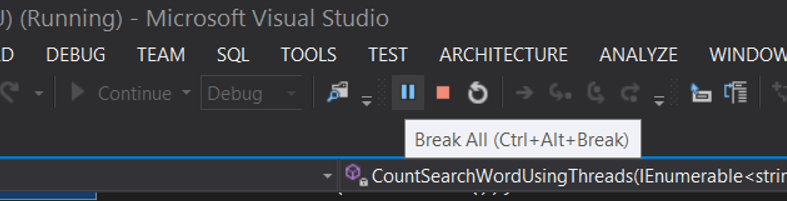
Use Tasks

**IMPORTANT:** **Due to the nature of multithreading, some of screenshots in this portion of the lab specifically the ones taken during debugging may be different than the ones you see. So don’t be alarmed. Understand what the lab step is trying to show you and perform that action in your debugging session. If you have any issues or are not so sure, please ask the instructor for assistance.**

1. Make code changes to use tasks. CountSearchWordUsingTasks method uses a task-based implementation to count the number of occurrences of the search word. Comment out the call to CountSearchWordUsingThreads and uncomment the call to CountSearchWordUsingTasks. When you are done, Main method should look like as shown below:



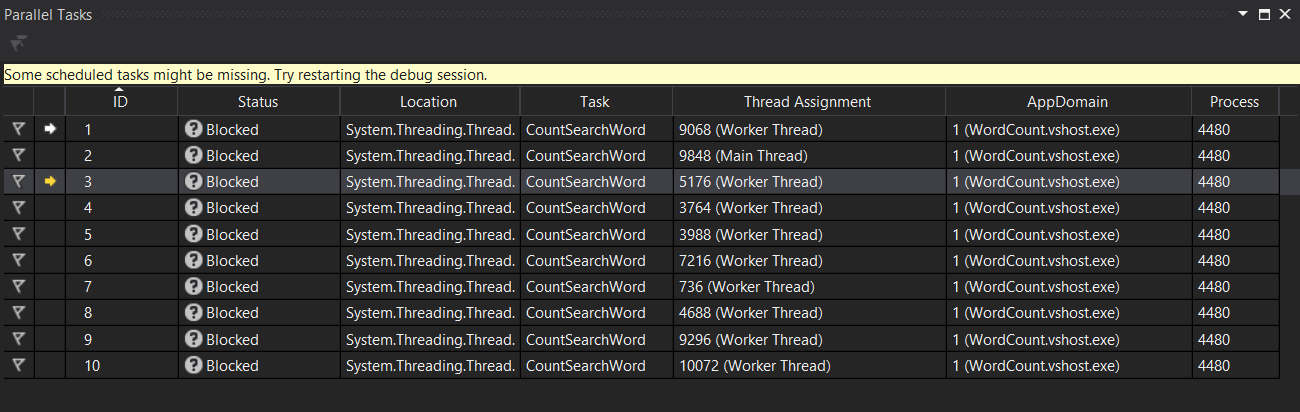
1. Take a look at CountSearchWordUsingTasks method to understand what the code is doing. This method basically makes use of Parallel.ForEach to process each line in parallel. It uses CountSearchWord method for the work and line as the parameter.
2. Build the project and debug it by hitting F5 or Start. Notice that you have not set up a breakpoint yet.
3. Let the application run for a little while.
4. While the application is still running, switch back to Visual Studio and click on Break All button in the toolbar of Visual Studio.



This is another way to break into execution. If you are debugging an application in Visual Studio and experience an unexpected strange behavior such as a hang or delay, you can go to Visual Studio and break all to break the execution to see what the application is doing.

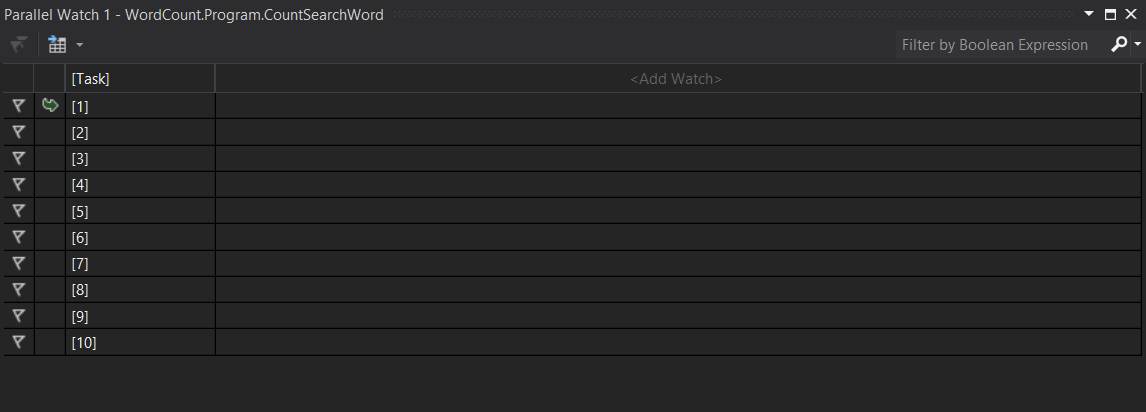
Note that if you are slow to break the execution and the application completed. Simply repeat the steps to run the application and break execution somewhere in the middle. Alternatively, there is a variable called delay in Program class. The value of this variable is in milliseconds and is used to put each thread to sleep to introduce a delay. You can increase the current value to a larger one.

1. View Parallel Tasks window from Debug🡪Windows menu.

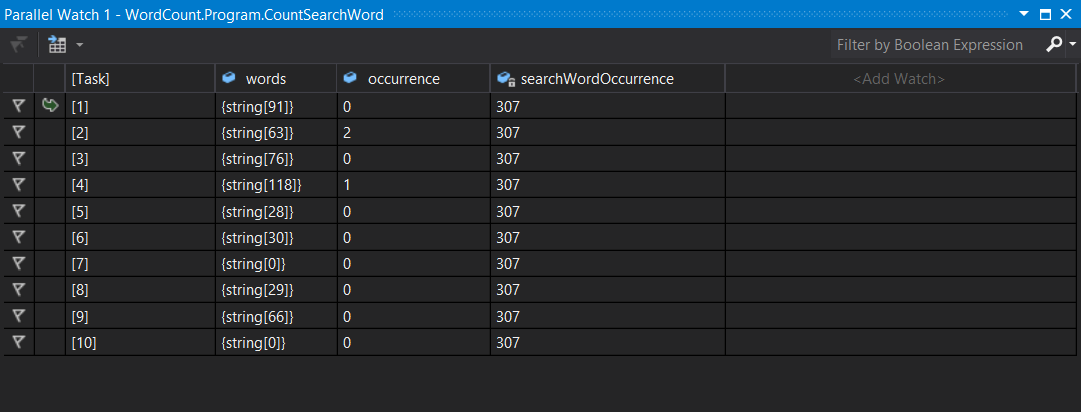


Parallel tasks window shows the tasks in the application and various information about each task such as location, status, thread, and id. You can double click on a task to go to the source and see what the task is doing. You can flag a task just like you could with a thread. You can right click on a task to bring the context menu to see the other actions you can take.

1. Add a parallel watch for task local variables words, occurrence, and searchWordOccurrence. To do this, bring up one of the parallel watch window from Debug🡪Windows🡪Parallel Watch menu.

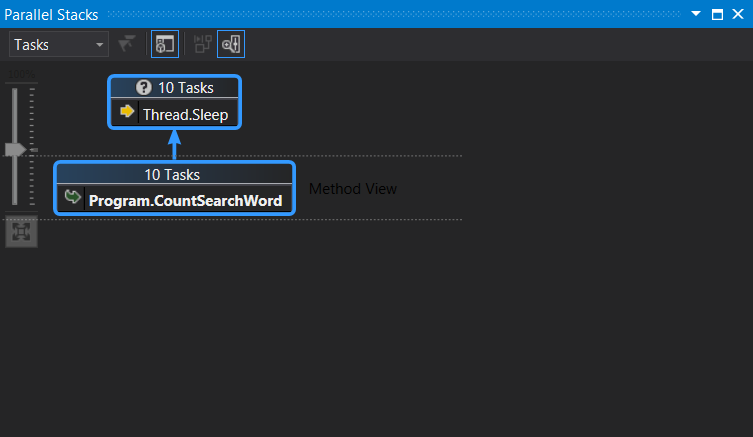


Add watch for words, occurrence, and searchWordOccurrence by clicking in <Add Watch> box and typing in the name of variable.



As you can see, using the parallel watch window we can easily inspect the value of variable across different tasks.

1. Bring up the parallel stacks window from Debug🡪Windows menu. The parallel stacks window gives you an nice interactive visualization of your tasks.



1. End of exercise.