How to find memory leaks in a .NET program

Memory leaks are a nightmares for any developer. A memory leak increases the memory of an entire application slowly, and gradually eats up the entire process memory and eventually the entire system memory. The memory leak becomes the biggest problem when the application is deployed to the server and needs to run day and night as a service.

Memory leaks can occur either in a stack, an unmanaged heap, or managed heaps. There are many ways to detect memory leaks in a program. Tools such as **Windbg**, **PerfMon** , **DebugDiag**, or even Visual Studio can be used to detect memory leaks in a program. The most important area of memory that is used by the process is represented by private bytes.

> **Note**
>
> It is important to note that one should always avoid using the task manager to confirm memory leaks in a program. A task manager memory usage can be misleading most of the times because it gives information about the working set memory and not the actual memory used. Some memory of the working set is even shared between multiple processes. Hence, the task manager memory usage is not exact.

Getting ready

Memory leak in a program can create a lot of symptoms. For instance, when a program is leaking memory, it can throw `OutOfMemoryException` or it may be sluggish in responding to user input because it has started swapping virtual memory to disk or maybe the memory is gradually increasing in the task manager. When you are certain that there is a memory leak in the program, the first thing that you need to do is to detect exactly where the memory leak is occurring. In this recipe, we will try to find the memory leak of the program.

Memory leak can happen either in managed resources or in unmanaged resources. Let us try to put unmanaged resources using `Marshal.AllocHGlobal(8000)` in a timer, such that whenever the timer becomes elapsed, memory gets created.

Similarly, for managed memory leaks we use a type with one member with more than 85,000 bytes of data, so that it is created in LOH, rather than SOH. As LOH is not been compacted, it produces a large memory gap between objects which can essentially produce memory leaks:

```
public  class LargeObjectHeap
{
  private byte[] buffer  = new byte[90000];
}
```

This code produces a large heap allocation, and eventually invoking this multiple times will produce a memory leak in the program.

How to do it...

To deal with a memory leak, we are going to use **PerfMon**. PerfMon is an interesting tool that can be used to examine counters on process. We can detect the private bytes allocated by the process, the .NET CLR memory, the bytes on all heaps, and .NET CLR `LocksAndThreads` . Based on the counters we can determine exactly where the problem is.

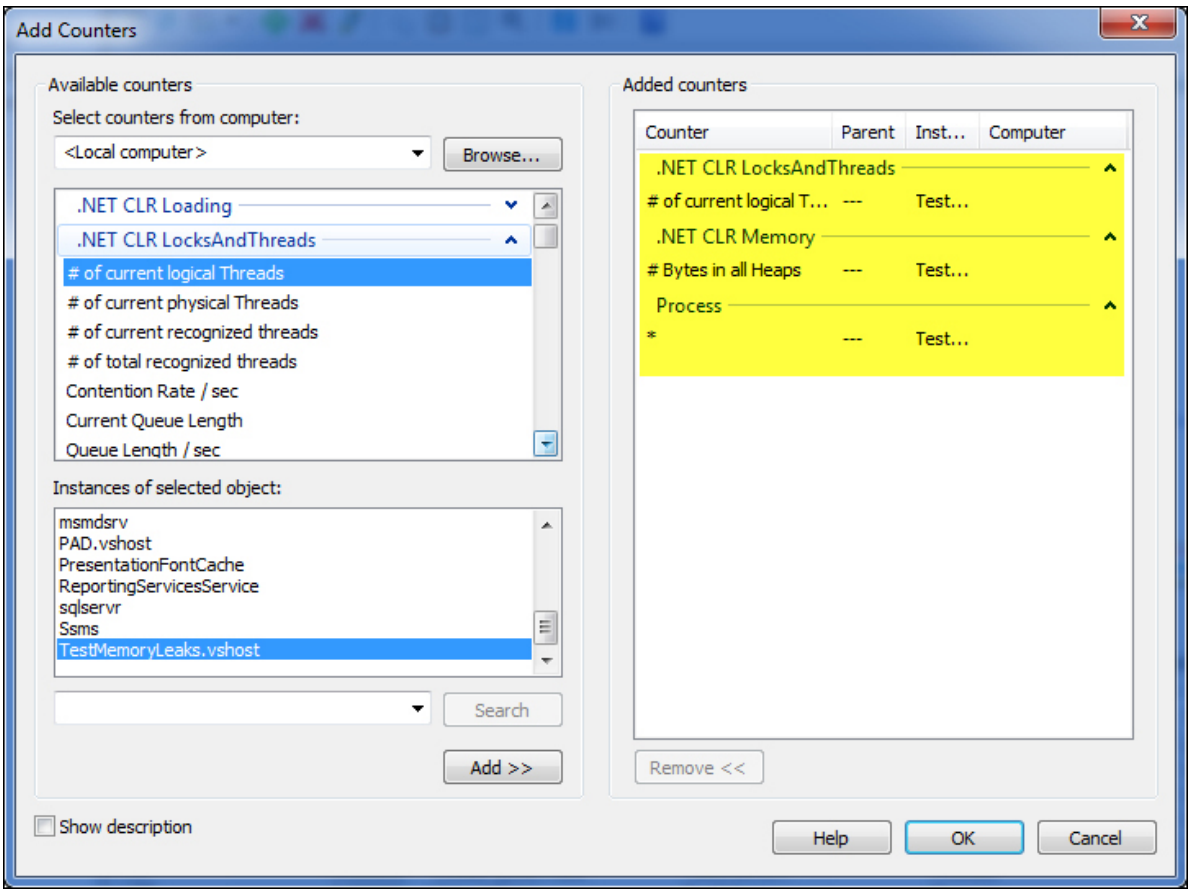
If CLR `LocksAndThreads` **(# of current logical Threads**) is increasing then the thread stack is leaking.

If only private bytes are increasing but .NET CLR memory is not increasing, then we have an unmanaged memory leak.

If both are increasing then it is a managed memory leak.

To work with the PerfMon tool, let's follow these steps:

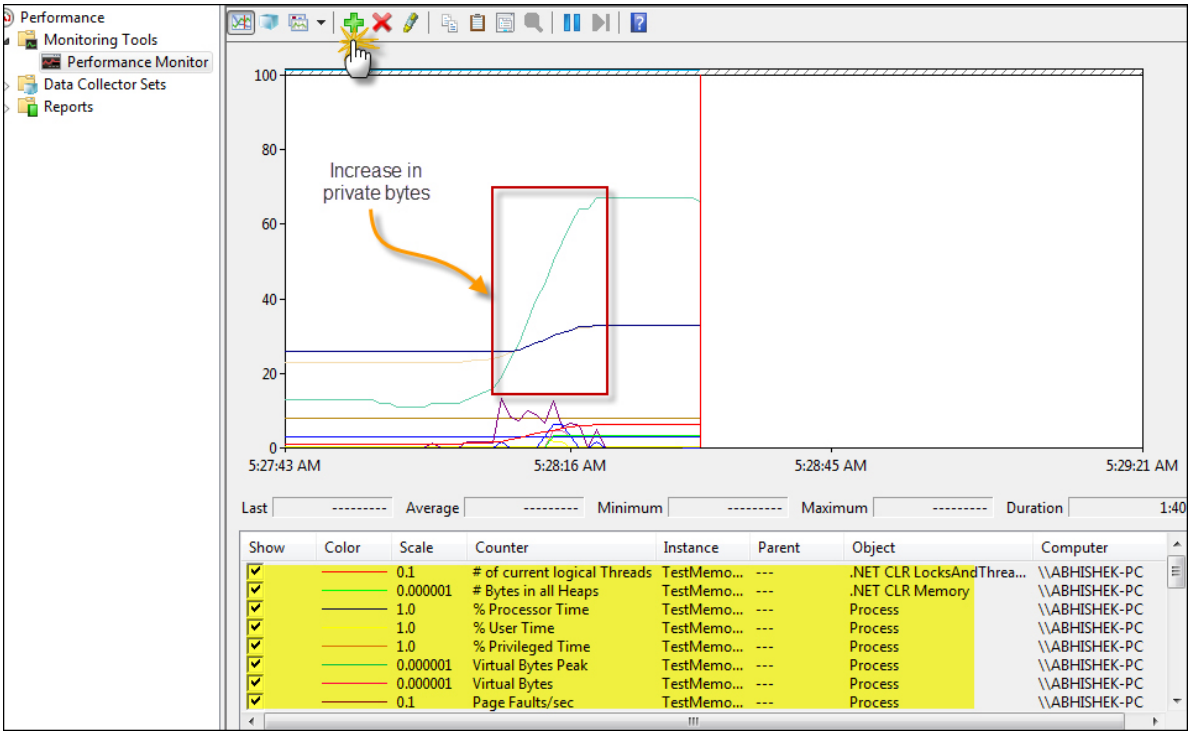
1. Start the application that has memory leaks.
2. Run `PerfMon` from the **Run** command to open the **Performance Monitor** tool.
3. Delete all current performance counters that are already added using the **Delete** button on top and select **Add** to add counters:



4. Select **Process** from **Performance** object, and select **Private bytes** from the list of counters. Select the appropriate process instance that is leaking memory.

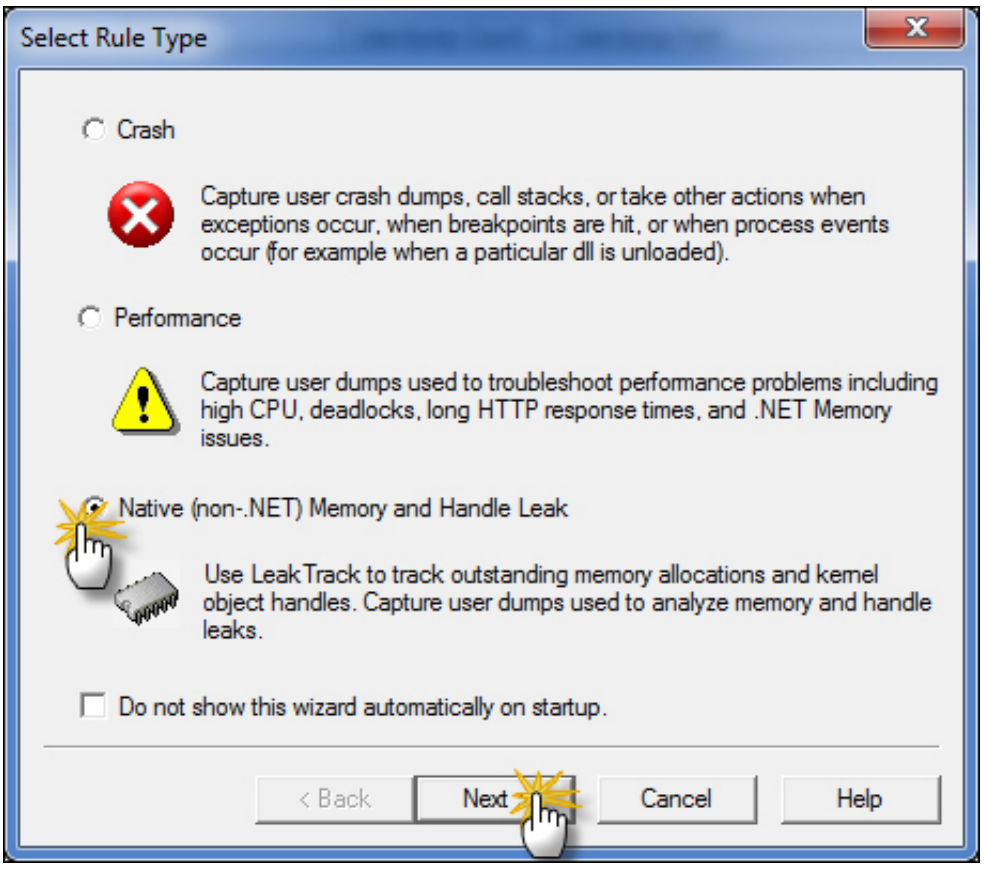
5. Again, select **.NET CLR Memory** from the **Performance** object again and select **# bytes in all heaps**.

6. Finally, select **.NET CLR LocksAndThreads** in the **Performance** object and select **#of current logical Threads** from the list of counters. In the preceding figure you can see how it looks when you are adding these performance counters. I have already added the counters, so click on **OK** when complete:



7. The performance counters in the graph will show the continuous update on these counters with different colors. You can see the list of performance counters in the screenshot marked as yellow and clearly, you can identify the hike in private bytes in the picture.

8. Download the **DebugDiag** tool from the Microsoft site, start the tool and select **Native (non-.NET) Memory and Handle Leak**. The tool actually gives you three options. It allows you to check either for a crash of the application, or performance, or memory leaks. As our application probably has a memory leak, we are going to choose the one that deals with memory. You can also check performance or crash checking for your process using this tool:



9. Next, select the process that you need to check for leakage. The tool will give a list of all the processes that are running on the computer. You can select the one that is appropriate and choose **Next**.
10. Finally, select **Activate Rule** to start the monitoring.
11. After a certain amount of time has elapsed, select **Memory Pressure Analysis** from the tools and select **Start Analysis**.
12. From the HTML report you can identify how memory has been allocated in managed or unmanaged resources. It shows you a warning on the code that has been creating memory leaks.
13. Based on the report we need to go to the source code to actually fix the memory leak on the program.
14. There is no fixed rule to remove memory leak in the program, but you can use DebugDiag's analysis report to actually investigate the problem and fix it from the pointers you get from it.

How it works...

Memory leaks are generally caused by a managed application when:

- **Holding references to a managed object**: This is a situation where the variable never goes out of active scope and hence never gets exposed to GC
- **Failing to release unmanaged resources**: When dealing with unmanaged memory, loaded memory needs to be freed manually and hence releasing unmanaged memory using disposable pattern is important
- Failing to Dispose drawing objects: Drawing objects holds unmanaged resources, hence we need to dispose explicitly

Memory management is one of the most important considerations for any application.

There's more...

Let's talk some more about the problems and solutions that can help you to deal with memory leaks.

What are weak references in .NET

While doing actual code, sometimes it is important to know how to allocate a memory and when to create a memory location. When GC kicks in, it tries to find all the reachable types that have strong references from the application. It goes in and traces all the bits and pieces of the program, to find out if the object is somehow linked to the program, that is, it has some reference from the program that can still use this object. The GC uses this algorithm to determine whether the memory is actually garbage or not.

If we can call an object from our program, GC treats the object as "important" and marks it to move to the next generation (if not, the highest generation is reached). But it is important to remember, that when we are talking about references from the program, we only speak about strong references. A weak reference is another concept that works in contrast with the strong references.

A weak reference is an exception to GC, such that when GC kicks in, it will always thread a memory that is held by a weak reference as garbage. The garbage collection's algorithm puts a weak reference as an exception to it, and collects it when GC tries to find the reference from the program.

The .NET class library exposes a special type called `WeakReference` that actually implements the weak reference concept just stated. Sometimes it is important to create an object that can exists from the application, but when the memory pressure is high we do not want that memory to stay. In such cases, we create a weak reference and forget about the object. When we need the object again, we first try to find the object from the weak reference, if found we use it, otherwise we recreate the object again.

Hence, weak reference acts as an exception to the existing GC algorithm, which lets the GC reclaim the memory even though the memory is still accessible from the program.

Weak references can be of two types:

- **Short**: They lose the reference when GC is collected. In a general case, when we create an object of the `WeakReference` class in our program, we create this type of memory.
- **Long**: They retain their object even after the `Finalize` method has been called. The object state cannot be determined in such cases. We can pass `trackResurrection` to `true` on the constructor of `WeakReference` to create a `Long` weak reference:

```
WeakReference weakref = null;
    private SomeBigClass _somebigobject = null;
    public SomeBigClass SomeBigObject
    {
         get
         {
             SomeBigClass sbo = null;
             if (weakref == null)  //When it is first time or
object weakref is collected.
             {
                 sbo = new SomeBigClass();
                 this.weakref = new WeakReference(sbo);
                 this.OnCallBack("Object created for first
time");
             }
             else if (weakref.Target == null) // when target
object is collected
             {
                 sbo = new SomeBigClass();
                 weakref.Target = sbo;
                 this.OnCallBack("Object is collected by GC, so
new object is created");
             }
             else // when target object is not collected.
             {
                 sbo = weakref.Target as SomeBigClass;
                 this.OnCallBack("Object is not yet collected,
so reusing the old object");
             }
             this._somebigobject = sbo; //gets you a strong
reference
             return this._somebigobject;
         }
         set
         {
             this._somebigobject = null;
         }
     }
```

Let us suppose `SomeBigObject` is a class that creates a large amount of data inside it. So while creating an instance of that class we

use the weak reference implementation of .NET and pass the object to it. We can find the object from the `Target` property only when the object has not been collected by GC. In the preceding code, you can see that we check whether the object has already been collected or not and depending on the same we create the object again or pass the existing object.

The `Target` property gets you a strong reference from `WeakReference`. When you are holding the value of `Target` to some reference in your program, the GC will not collect the object. But as soon as the reference is nulled and no reference to the `Target` object has been stored, the object inside `WeakReference` is exposed for collection.

Here in the preceding code, we have also created a callback just to print the message to the caller whether the object is recreated or not.

What are lazy objects in .NET

In our previous code we see how we can create a weak reference that can be used while dealing with very big objects. The concept of lazy objects on the other hand is a pattern that defers the initialization of the object when it is really required. A lazy object in .NET actually allows you to specify a callback that will automatically be called whenever the object is invoked. The idea is to pass the callback which creates the object, in such a way that when the object actually needs to be created, the lazy object will invoke the handler and create the object.

There are three types in .NET that support lazy Initialization:

- **Lazy**: This is just a wrapper class that supports lazy initialization
- **ThreadLocal**: This is the same as `Lazy` but the only difference is that it stores data on thread local basis
- **LazyInitializer**: This provides static implementation of lazy initializer that eliminates the overhead of creation of lazy objects

Lazy

`Lazy` creates a thread-safe lazy initialization of objects. Say for instance, when we need to create a large number of objects and each of these objects by itself create a lots of objects, it is easy to initialize the master object using `Lazy` blocks. For instance, say there are a large number of customers and for each customer, there are a large number of payments, if `Customer` is an entity and `Payment` is also an entity, `Customer` will contain an array of `Payment` objects. Thus, each entity requires a large number of database calls to ensure that the data is retrieved, which doesn't makes sense. Using the `Lazy` class you can eliminate this problem.

Let us look at how to use it:

```
public class Customer
 {
        public string Name { get; set; }
        public Lazy<IList<Payment>> Payments
        {
            get
            {
                return new Lazy<IList<Payment>>(() =>
this.FetchPayments());
            }
        }

        private IList<Payment> FetchPayments()
        {
            List<Payment> payments = new List<Payment>();
            payments.Add(new Payment { BillNumber = 1, BillDate =
DateTime.Now, PaymentAmount = 200 });
            payments.Add(new Payment { BillNumber = 2, BillDate =
DateTime.Now.AddDays(-1), PaymentAmount = 540 });
            payments.Add(new Payment { BillNumber = 3, BillDate =
DateTime.Now.AddDays(-2), PaymentAmount = 700 });
            payments.Add(new Payment { BillNumber = 4, BillDate =
DateTime.Now, PaymentAmount = 500 });
            //Load all the payments here from database
            return payments;
        }
        public Payment GetPayment(int billno)
```

```
        {
            if (this.Orders.IsValueCreated)
            {
                var payments = this.Payments.Value;
                Payment p = payments.FirstOrDefault(pay =>
pay.BillNumber.Equals(billno));
                return p;
            }
            else
                throw new NotImplementedException("Object is not
initialized");
        }

    }

    public class Payment
    {
        public int BillNumber {get;set;}
        public DateTime BillDate { get; set; }
        public double PaymentAmount { get; set; }
    }
```

Here I have created a class called `Payment` that has a few properties. Each customer has a list of payments. You can see the `Customer` class that uses Lazy implementation. This calls `FetchPayments` to create objects whenever `this.Payments.Value` is called. The `IsValueCreated` property will evaluate to `true` when the list is created.

Similar to the list, you can use `Lazy` binding to any other CLR objects as well.

> **Note**
>
> `System.Lazy` creates a `ThreadSafe` object by default. The default constructor creates an object with `LazyThreadSafetyMode.ExecutionAndPublication`. Thus, once an object is created by one thread, the object will be accessible to all other concurrent threads.

ThreadLocal

Similar to `Lazy`, `ThreadLocal` creates an object local to one thread, so each individual thread will have its own `Lazy` initializer object and hence, will create the object multiple times once for each thread. You can create objects that are local to one thread using the `ThreadStatic` attribute. But sometimes `ThreadStatic` fails to create a true `ThreadLocal` object. Basic static initializer is initialized for once, in case of the `ThreadStatic` class.

`ThreadLocal` creates a wrapper of `Lazy` and creates a true `ThreadLocal` object:

```
    public void CreateThreadLocal()
        {
            ThreadLocal<List<float>> local = new ThreadLocal<List<float>>
(() => this.GetNumberList(Thread.CurrentThread.ManagedThreadId));
            Thread.Sleep(5000);
            List<float> numbers = local.Value;
            foreach (float num in numbers)
                Console.WriteLine(num);


        }
        private List<float> GetNumberList(int p)
        {
            Random rand = new Random(p);
```

```
            List<float> items = new List<float>();
            for(int i = 0; i<10;i++)
                items.Add(rand.Next());
            return items;
        }
```

In the preceding methods, `CreateThreadLocal` creates a local thread and takes the lazy object `GetNumberList` when the value is called for (just like normal lazy implementation).

Now, if you call `CreateThreadLocal` using the following code:

```
Thread newThread = new Thread(new ThreadStart(this.CreateThreadLocal));
            newThread.Start();
            Thread newThread2 = new Thread(new
ThreadStart(this.CreateThreadLocal));
            newThread2.Start();
```

Each thread `newThread` and `newThread2` will contain its own list of lists.

LazyInitializer

Finally, coming to `LazyInitializer`, you can create the same implementation of the lazy objects without creating the object of `Lazy`. `LazyInitializer` handles the lazy implementation internally giving you static interfaces from outside that enable you to use it without much heck.

The `LazyInitializer.EnsureInitialized` method takes two arguments, in general. The first one is the `ref` parameter, where you have to pass the value of the variable. Here, you want the target to be generated and the delegate that generates the output:

```
public void MyLazyInitializer()
{
   List<Payment> items = new List<Payment>();
   for (int i = 0; i < 10; i++)
   {
      Payment paymentobj = new Payment();
      LazyInitializer.EnsureInitialized<Payment>(ref paymentobj, () =>
      {
          return this.GetPayment(i);
      });
      items.Add(paymentobj);
   }
}
```

In the preceding scenario, you can see that I have used `LazyInitializer` that fetches each `Payment` object when it is required. The `ref` parameter takes a class type object explicitly and returns the object to the variable.

> **Tip**
>
> **When should we use Lazy?**
>
> If you are using an object base which is resource consuming and you are sure that every object will not be required for the application to run, you can go for lazy initializers, otherwise it is not recommended. Another important consideration is that `Lazy` doesn't work very well with value types, and it is better to avoid it for these.

How to use Visual Studio to create memory dump files

Visual Studio allows you to store and create dump files during debugging sessions. These dump files allow you to save the entire dump information of a program when a program crashes. With the use of dump files, you can debug it later whether on a build computer or another computer that has the source code and debugging symbols.

Visual Studio 2012 debugger can save mini dump files for either managed or native code. To create a dump, select **Debug** | **Save Dump file**. You can either select **Minidump** or **Minidump with heap**.

Once the dump has been saved, you can use this dump to open in Visual Studio from **File** | **Open** to get the entire report of the dump. From the

Action section, you can debug the program either with native only or with mixed.

How to isolate code using AppDomain

In .NET, the primary execution unit of an application is not a process but rather an AppDomain. AppDomain is a separate unit that exists in the .NET environment that loads memory and runs user code. Such that the memory that has been allocated in one `AppDomain` instance is totally isolated from another one. Unlike threads, an `AppDomain` instance does not share the same memory, and hence any corruption of memory in one domain cannot affect the other.

While doing memory management of an application, it is good to use separate `AppDomain` instances to load the assembly that leaks memory. This will ensure that the assembly executes in a separate and isolated memory.

You can use the following code to create an `AppDomain` instance:

```
AppDomain newDomain = AppDomain.CreateDomain("domainName");
newDomain.ExecuteAssembly(yourassembly);
```

Even though we do not create an `AppDomain` instance during the execution of any application, the CLR creates a default `AppDomain` instance for us. You can use `AppDomain.CurrentDomain` to get information about the domain where the user code is executing.