Contents

Memory Management and Garbage Collection in .NET

2 minutes to read • Edit Online

This section of the documentation provides information about managing memory in .NET.

In This Section

Cleaning Up Unmanaged Resources
Describes how to properly manage and clean up unmanaged resources..

Garbage Collection
Provides information about the .NET garbage collector.

Related Sections

Development Guide

minutes to read • Edit Online

For the majority of the objects that your app creates, you can rely on .NET's garbage collector to handle memory management. However, when you create objects that include unmanaged resources, you must explicitly release those resources when you finish using them in your app. The most common types of unmanaged resource are objects that wrap operating system resources, such as files, windows, network connections, or database connections. Although the garbage collector is able to track the lifetime of an object that encapsulates an unmanaged resource, it doesn't know how to release and clean up the unmanaged resource.

If your types use unmanaged resources, you should do the following:

- Implement the dispose pattern. This requires that you provide an IDisposable.Dispose implementation to enable the deterministic release of unmanaged resources. A consumer of your type calls Dispose when the object (and the resources it uses) is no longer needed. The Dispose method immediately releases the unmanaged resources.
- Provide for your unmanaged resources to be released in the event that a consumer of your type forgets to call Dispose. There are two ways to do this:
  - Use a safe handle to wrap your unmanaged resource. This is the recommended technique. Safe handles are derived from the System.Runtime.InteropServices.SafeHandle class and include a robust Finalize method. When you use a safe handle, you simply implement the IDisposable interface and call your safe handle's Dispose method in your IDisposable.Dispose implementation. The safe handle's finalizer is called automatically by the garbage collector if its Dispose method is not called.

    —or—

  - Override the Object.Finalize method. Finalization enables the non-deterministic release of unmanaged resources when the consumer of a type fails to call IDisposable.Dispose to dispose of them deterministically. However, because object finalization can be a complex and error-prone operation, we recommend that you use a safe handle instead of providing your own finalizer.

Consumers of your type can then call your IDisposable.Dispose implementation directly to free memory used by unmanaged resources. When you properly implement a Dispose method, either your safe handle's Finalize method or your own override of the Object.Finalize method becomes a safeguard to clean up resources in the event that the Dispose method is not called.

In This Section

Implementing a Dispose Method Describes how to implement the dispose pattern for releasing unmanaged resources.

Using Objects That Implement IDisposable Describes how consumers of a type ensure that its Dispose implementation is called. We recommend using the C# `using` statement or the Visual Basic `Using` statement to do this.

Reference

System.IDisposable
Defines the Dispose method for releasing unmanaged resources.

Object.Finalize

Provides for object finalization if unmanaged resources are not released by the Dispose method.

GC.SuppressFinalize

Suppresses finalization. This method is customarily called from a `Dispose` method to prevent a finalizer from executing.

Implementing a Dispose method

14 minutes to read • Edit Online

You implement a Dispose method to release unmanaged resources used by your application. The .NET garbage collector does not allocate or release unmanaged memory.

The pattern for disposing an object, referred to as a dispose pattern, imposes order on the lifetime of an object. The dispose pattern is used only for objects that access unmanaged resources, such as file and pipe handles, registry handles, wait handles, or pointers to blocks of unmanaged memory. This is because the garbage collector is very efficient at reclaiming unused managed objects, but it is unable to reclaim unmanaged objects.

The dispose pattern has two variations:

- You wrap each unmanaged resource that a type uses in a safe handle (that is, in a class derived from System.Runtime.InteropServices.SafeHandle). In this case, you implement the IDisposable interface and an additional `Dispose(Boolean)` method. This is the recommended variation and doesn't require overriding the Object.Finalize method.

  > **NOTE**
  > The Microsoft.Win32.SafeHandles namespace provides a set of classes derived from SafeHandle, which are listed in the Using safe handles section. If you can't find a class that is suitable for releasing your unmanaged resource, you can implement your own subclass of SafeHandle.

- You implement the IDisposable interface and an additional `Dispose(Boolean)` method, and you also override the Object.Finalize method. You must override Finalize to ensure that unmanaged resources are disposed of if your IDisposable.Dispose implementation is not called by a consumer of your type. If you use the recommended technique discussed in the previous bullet, the System.Runtime.InteropServices.SafeHandle class does this on your behalf.

To help ensure that resources are always cleaned up appropriately, a Dispose method should be callable multiple times without throwing an exception.

The code example provided for the GC.KeepAlive method shows how garbage collection can cause a finalizer to run, while an unmanaged reference to the object or its members is still in use. It may make sense to utilize GC.KeepAlive to make the object ineligible for garbage collection from the start of the current routine to the point where this method is called.

Dispose() and Dispose(Boolean)

The IDisposable interface requires the implementation of a single parameterless method, Dispose. However, the dispose pattern requires two `Dispose` methods to be implemented:

- A public non-virtual (`NonInheritable` in Visual Basic) IDisposable.Dispose implementation that has no parameters.
- A protected virtual (`Overridable` in Visual Basic) `Dispose` method whose signature is:

```
protected virtual void Dispose(bool disposing)
```

```
Protected Overridable Sub Dispose(disposing As Boolean)
```

The Dispose() overload

Because the public, non-virtual (`NonInheritable` in Visual Basic), parameterless `Dispose` method is called by a consumer of the type, its purpose is to free unmanaged resources and to indicate that the finalizer, if one is present, doesn't have to run. Because of this, it has a standard implementation:

```
public void Dispose()
{
   // Dispose of unmanaged resources.
   Dispose(true);
   // Suppress finalization.
   GC.SuppressFinalize(this);
}
```

```
Public Sub Dispose() _
           Implements IDisposable.Dispose
   ' Dispose of unmanaged resources.
   Dispose(True)
   ' Suppress finalization.
   GC.SuppressFinalize(Me)
End Sub
```

The `Dispose` method performs all object cleanup, so the garbage collector no longer needs to call the objects' Object.Finalize override. Therefore, the call to the SuppressFinalize method prevents the garbage collector from running the finalizer. If the type has no finalizer, the call to GC.SuppressFinalize has no effect. Note that the actual work of releasing unmanaged resources is performed by the second overload of the `Dispose` method.

The Dispose(Boolean) overload

In the second overload, the *disposing* parameter is a Boolean that indicates whether the method call comes from a Dispose method (its value is `true`) or from a finalizer (its value is `false`).

The body of the method consists of two blocks of code:

- A block that frees unmanaged resources. This block executes regardless of the value of the `disposing` parameter.
- A conditional block that frees managed resources. This block executes if the value of `disposing` is `true`. The managed resources that it frees can include:

  Managed objects that implement IDisposable. The conditional block can be used to call their Dispose implementation. If you have used a safe handle to wrap your unmanaged resource, you should call the SafeHandle.Dispose(Boolean) implementation here.

  Managed objects that consume large amounts of memory or consume scarce resources. Freeing these objects explicitly in the `Dispose` method releases them faster than if they were reclaimed non-deterministically by the garbage collector.

If the method call comes from a finalizer (that is, if *disposing* is `false`), only the code that frees unmanaged resources executes. Because the order in which the garbage collector destroys managed objects during finalization is not defined, calling this `Dispose` overload with a value of `false` prevents the finalizer from trying to release managed resources that may have already been reclaimed.

Implementing the dispose pattern for a base class

If you implement the dispose pattern for a base class, you must provide the following:

> **IMPORTANT**
> You should implement this pattern for all base classes that implement Dispose() and are not `sealed` (`NotInheritable` in Visual Basic).

- A Dispose implementation that calls the `Dispose(Boolean)` method.
- A `Dispose(Boolean)` method that performs the actual work of releasing resources.
- Either a class derived from SafeHandle that wraps your unmanaged resource (recommended), or an override to the Object.Finalize method. The SafeHandle class provides a finalizer that frees you from having to code one.

Here's the general pattern for implementing the dispose pattern for a base class that uses a safe handle.

```
using Microsoft.Win32.SafeHandles;
using System;
using System.Runtime.InteropServices;

class BaseClass : IDisposable
{
   // Flag: Has Dispose already been called?
   bool disposed = false;
   // Instantiate a SafeHandle instance.
   SafeHandle handle = new SafeFileHandle(IntPtr.Zero, true);

   // Public implementation of Dispose pattern callable by consumers.
   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }

   // Protected implementation of Dispose pattern.
   protected virtual void Dispose(bool disposing)
   {
      if (disposed)
         return;

      if (disposing) {
         handle.Dispose();
         // Free any other managed objects here.
         //
      }

      disposed = true;
   }
}
```

```
Imports Microsoft.Win32.SafeHandles
Imports System.Runtime.InteropServices

Class BaseClass : Implements IDisposable
   ' Flag: Has Dispose already been called?
   Dim disposed As Boolean = False
   ' Instantiate a SafeHandle instance.
   Dim handle As SafeHandle = New SafeFileHandle(IntPtr.Zero, True)

   ' Public implementation of Dispose pattern callable by consumers.
   Public Sub Dispose() _
              Implements IDisposable.Dispose
      Dispose(True)
      GC.SuppressFinalize(Me)
   End Sub

   ' Protected implementation of Dispose pattern.
   Protected Overridable Sub Dispose(disposing As Boolean)
      If disposed Then Return

      If disposing Then
         handle.Dispose()
         ' Free any other managed objects here.
         '
      End If

      disposed = True
   End Sub
End Class
```

NOTE

The previous example uses a SafeFileHandle object to illustrate the pattern; any object derived from SafeHandle could be used instead. Note that the example does not properly instantiate its SafeFileHandle object.

Here's the general pattern for implementing the dispose pattern for a base class that overrides Object.Finalize.

```
using System;

class BaseClass : IDisposable
{
   // Flag: Has Dispose already been called?
   bool disposed = false;

   // Public implementation of Dispose pattern callable by consumers.
   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }

   // Protected implementation of Dispose pattern.
   protected virtual void Dispose(bool disposing)
   {
      if (disposed)
         return;

      if (disposing) {
         // Free any other managed objects here.
         //
      }

      // Free any unmanaged objects here.
      //
      disposed = true;
   }

   ~BaseClass()
   {
      Dispose(false);
   }
}
```

```
Class BaseClass : Implements IDisposable
   ' Flag: Has Dispose already been called?
   Dim disposed As Boolean = False

   ' Public implementation of Dispose pattern callable by consumers.
   Public Sub Dispose() _
              Implements IDisposable.Dispose
      Dispose(True)
      GC.SuppressFinalize(Me)
   End Sub

   ' Protected implementation of Dispose pattern.
   Protected Overridable Sub Dispose(disposing As Boolean)
      If disposed Then Return

      If disposing Then
         ' Free any other managed objects here.
         '
      End If

      ' Free any unmanaged objects here.
      '
      disposed = True
   End Sub

   Protected Overrides Sub Finalize()
      Dispose(False)
   End Sub
End Class
```

> **NOTE**
> In C#, you override Object.Finalize by defining a destructor.

Implementing the dispose pattern for a derived class

A class derived from a class that implements the IDisposable interface shouldn't implement IDisposable, because the base class implementation of IDisposable.Dispose is inherited by its derived classes. Instead, to release resources of a derived class, you provide the following:

- A `protected Dispose(Boolean)` method that overrides the base class method and performs the actual work of releasing the resources of the derived class. This method should also call the `Dispose(Boolean)` method of the base class and pass its disposing status for the argument.
- Either a class derived from SafeHandle that wraps your unmanaged resource (recommended), or an override to the Object.Finalize method. The SafeHandle class provides a finalizer that frees you from having to code one. If you do provide a finalizer, it should call the `Dispose(Boolean)` overload with a *disposing* argument of `false`.

Here's the general pattern for implementing the dispose pattern for a derived class that uses a safe handle:

```
using Microsoft.Win32.SafeHandles;
using System;
using System.Runtime.InteropServices;

class DerivedClass : BaseClass
{
   // Flag: Has Dispose already been called?
   bool disposed = false;
   // Instantiate a SafeHandle instance.
   SafeHandle handle = new SafeFileHandle(IntPtr.Zero, true);

   // Protected implementation of Dispose pattern.
   protected override void Dispose(bool disposing)
   {
      if (disposed)
         return;

      if (disposing) {
         handle.Dispose();
         // Free any other managed objects here.
         //
      }

      // Free any unmanaged objects here.
      //

      disposed = true;
      // Call base class implementation.
      base.Dispose(disposing);
   }
}
```

```
Imports Microsoft.Win32.SafeHandles
Imports System.Runtime.InteropServices

Class DerivedClass : Inherits BaseClass
   ' Flag: Has Dispose already been called?
   Dim disposed As Boolean = False
   ' Instantiate a SafeHandle instance.
   Dim handle As SafeHandle = New SafeFileHandle(IntPtr.Zero, True)

   ' Protected implementation of Dispose pattern.
   Protected Overrides Sub Dispose(disposing As Boolean)
      If disposed Then Return

      If disposing Then
         handle.Dispose()
         ' Free any other managed objects here.
         '
      End If

      ' Free any unmanaged objects here.
      '
      disposed = True

      ' Call base class implementation.
      MyBase.Dispose(disposing)
   End Sub
End Class
```

NOTE

The previous example uses a SafeFileHandle object to illustrate the pattern; any object derived from SafeHandle could be used instead. Note that the example does not properly instantiate its SafeFileHandle object.

Here's the general pattern for implementing the dispose pattern for a derived class that overrides Object.Finalize:

```
using System;

class DerivedClass : BaseClass
{
   // Flag: Has Dispose already been called?
   bool disposed = false;

   // Protected implementation of Dispose pattern.
   protected override void Dispose(bool disposing)
   {
      if (disposed)
         return;

      if (disposing) {
         // Free any other managed objects here.
         //
      }

      // Free any unmanaged objects here.
      //
      disposed = true;

      // Call the base class implementation.
      base.Dispose(disposing);
   }

   ~DerivedClass()
   {
      Dispose(false);
   }
}
```

```
Class DerivedClass : Inherits BaseClass
   ' Flag: Has Dispose already been called?
   Dim disposed As Boolean = False

   ' Protected implementation of Dispose pattern.
   Protected Overrides Sub Dispose(disposing As Boolean)
      If disposed Then Return

      If disposing Then
         ' Free any other managed objects here.
         '
      End If

      ' Free any unmanaged objects here.
      '
      disposed = True

      ' Call the base class implementation.
      MyBase.Dispose(disposing)
   End Sub

   Protected Overrides Sub Finalize()
      Dispose(False)
   End Sub
End Class
```

NOTE

In C#, you override Object.Finalize by defining a destructor.

Using safe handles

Writing code for an object's finalizer is a complex task that can cause problems if not done correctly. Therefore, we recommend that you construct System.Runtime.InteropServices.SafeHandle objects instead of implementing a finalizer.

Classes derived from the System.Runtime.InteropServices.SafeHandle class simplify object lifetime issues by assigning and releasing handles without interruption. They contain a critical finalizer that is guaranteed to run while an application domain is unloading. For more information about the advantages of using a safe handle, see System.Runtime.InteropServices.SafeHandle. The following derived classes in the Microsoft.Win32.SafeHandles namespace provide safe handles:

- The SafeFileHandle, SafeMemoryMappedFileHandle, and SafePipeHandle class, for files, memory mapped files, and pipes.
- The SafeMemoryMappedViewHandle class, for memory views.
- The SafeNCryptKeyHandle, SafeNCryptProviderHandle, and SafeNCryptSecretHandle classes, for cryptography constructs.
- The SafeRegistryHandle class, for registry keys.
- The SafeWaitHandle class, for wait handles.

Using a safe handle to implement the dispose pattern for a base class

The following example illustrates the dispose pattern for a base class, `DisposableStreamResource`, that uses a safe handle to encapsulate unmanaged resources. It defines a `DisposableResource` class that uses a SafeFileHandle to wrap a Stream object that represents an open file. The `DisposableResource` method also includes a single property, `Size`, that returns the total number of bytes in the file stream.

```
using Microsoft.Win32.SafeHandles;
using System;
using System.IO;
using System.Runtime.InteropServices;

public class DisposableStreamResource : IDisposable
{
   // Define constants.
   protected const uint GENERIC_READ = 0x80000000;
   protected const uint FILE_SHARE_READ = 0x00000001;
   protected const uint OPEN_EXISTING = 3;
   protected const uint FILE_ATTRIBUTE_NORMAL = 0x80;
   protected IntPtr INVALID_HANDLE_VALUE = new IntPtr(-1);
   private const int INVALID_FILE_SIZE = unchecked((int) 0xFFFFFFFF);

   // Define Windows APIs.
   [DllImport("kernel32.dll", EntryPoint = "CreateFileW", CharSet = CharSet.Unicode)]
   protected static extern IntPtr CreateFile (
                                 string lpFileName, uint dwDesiredAccess,
                                 uint dwShareMode, IntPtr lpSecurityAttributes,
                                 uint dwCreationDisposition, uint dwFlagsAndAttributes,
                                 IntPtr hTemplateFile);

   [DllImport("kernel32.dll")]
   private static extern int GetFileSize(SafeFileHandle hFile, out int lpFileSizeHigh);

   // Define locals.
   private bool disposed = false;
   private SafeFileHandle safeHandle;
   private long bufferSize;
   private int upperWord;
```

```
   public DisposableStreamResource(string filename)
   {
      if (filename == null)
         throw new ArgumentNullException("The filename cannot be null.");
      else if (filename == "")
         throw new ArgumentException("The filename cannot be an empty string.");

      IntPtr handle = CreateFile(filename, GENERIC_READ, FILE_SHARE_READ,
                                 IntPtr.Zero, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                                 IntPtr.Zero);
      if (handle != INVALID_HANDLE_VALUE)
         safeHandle = new SafeFileHandle(handle, true);
      else
         throw new FileNotFoundException(String.Format("Cannot open '{0}'", filename));

      // Get file size.
      bufferSize = GetFileSize(safeHandle, out upperWord);
      if (bufferSize == INVALID_FILE_SIZE)
         bufferSize = -1;
      else if (upperWord > 0)
         bufferSize = (((long)upperWord) << 32) + bufferSize;
   }

   public long Size
   { get { return bufferSize; } }

   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }

   protected virtual void Dispose(bool disposing)
   {
      if (disposed) return;

      // Dispose of managed resources here.
      if (disposing)
         safeHandle.Dispose();

      // Dispose of any unmanaged resources not wrapped in safe handles.

      disposed = true;
   }
}
```

```
Imports Microsoft.Win32.SafeHandles
Imports System.IO

Public Class DisposableStreamResource : Implements IDisposable
   ' Define constants.
   Protected Const GENERIC_READ As UInteger = &H80000000ui
   Protected Const FILE_SHARE_READ As UInteger = &H0000000i
   Protected Const OPEN_EXISTING As UInteger = 3
   Protected Const FILE_ATTRIBUTE_NORMAL As UInteger = &H80
   Protected INVALID_HANDLE_VALUE As New IntPtr(-1)
   Private Const INVALID_FILE_SIZE As Integer = &HFFFFFFFF

   ' Define Windows APIs.
   Protected Declare Function CreateFile Lib "kernel32" Alias "CreateFileA" (
                                       lpFileName As String, dwDesiredAccess As UInt32,
                                       dwShareMode As UInt32, lpSecurityAttributes As IntPtr,
                                       dwCreationDisposition As UInt32, dwFlagsAndAttributes As UInt32,
                                       hTemplateFile As IntPtr) As IntPtr
   Private Declare Function GetFileSize Lib "kernel32" (hFile As SafeFileHandle,
                                                        ByRef lpFileSizeHigh As Integer) As Integer
```

```
    ' Define locals.
    Private disposed As Boolean = False
    Private safeHandle As SafeFileHandle
    Private bufferSize As Long
    Private upperWord As Integer

    Public Sub New(filename As String)
       If filename Is Nothing Then
          Throw New ArgumentNullException("The filename cannot be null.")
       Else If filename = ""
          Throw New ArgumentException("The filename cannot be an empty string.")
       End If

       Dim handle As IntPtr = CreateFile(filename, GENERIC_READ, FILE_SHARE_READ,
                                         IntPtr.Zero, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                                         IntPtr.Zero)
       If handle <> INVALID_HANDLE_VALUE Then
          safeHandle = New SafeFileHandle(handle, True)
       Else
          Throw New FileNotFoundException(String.Format("Cannot open '{0}'", filename))
       End If

       ' Get file size.
       bufferSize = GetFileSize(safeHandle, upperWord)
       If bufferSize = INVALID_FILE_SIZE Then
          bufferSize = -1
       Else If upperWord > 0 Then
          bufferSize = (CLng(upperWord) << 32) + bufferSize
       End If
    End Sub

    Public ReadOnly Property Size As Long
       Get
          Return bufferSize
       End Get
    End Property

    Public Sub Dispose() _
               Implements IDisposable.Dispose
       Dispose(True)
       GC.SuppressFinalize(Me)
    End Sub

    Protected Overridable Sub Dispose(disposing As Boolean)
       If disposed Then Exit Sub

       ' Dispose of managed resources here.
       If disposing Then
          safeHandle.Dispose()
       End If

       ' Dispose of any unmanaged resources not wrapped in safe handles.

       disposed = True
    End Sub
End Class
```

Using a safe handle to implement the dispose pattern for a derived class

The following example illustrates the dispose pattern for a derived class, `DisposableStreamResource2`, that inherits from the `DisposableStreamResource` class presented in the previous example. The class adds an additional method, `WriteFileInfo`, and uses a SafeFileHandle object to wrap the handle of the writable file.

```
using Microsoft.Win32.SafeHandles;
using System;
using System.IO;
using System.Runtime.InteropServices;
using System.Threading;

public class DisposableStreamResource2 : DisposableStreamResource
{
   // Define additional constants.
   protected const uint GENERIC_WRITE = 0x40000000;
   protected const uint OPEN_ALWAYS = 4;

   // Define additional APIs.
   [DllImport("kernel32.dll")]
   protected static extern bool WriteFile(
                               SafeFileHandle safeHandle, string lpBuffer,
                               int nNumberOfBytesToWrite, out int lpNumberOfBytesWritten,
                               IntPtr lpOverlapped);

   // Define locals.
   private bool disposed = false;
   private string filename;
   private bool created = false;
   private SafeFileHandle safeHandle;

   public DisposableStreamResource2(string filename) : base(filename)
   {
      this.filename = filename;
   }

   public void WriteFileInfo()
   {
      if (! created) {
         IntPtr hFile = CreateFile(@".\FileInfo.txt", GENERIC_WRITE, 0,
                                   IntPtr.Zero, OPEN_ALWAYS,
                                   FILE_ATTRIBUTE_NORMAL, IntPtr.Zero);
         if (hFile != INVALID_HANDLE_VALUE)
            safeHandle = new SafeFileHandle(hFile, true);
         else
            throw new IOException("Unable to create output file.");

         created = true;
      }

      string output = String.Format("{0}: {1:N0} bytes\n", filename, Size);
      int bytesWritten;
      bool result = WriteFile(safeHandle, output, output.Length, out bytesWritten, IntPtr.Zero);
   }

   protected override void Dispose(bool disposing)
   {
      if (disposed) return;

      // Release any managed resources here.
      if (disposing)
         safeHandle.Dispose();

      disposed = true;

      // Release any unmanaged resources not wrapped by safe handles here.

      // Call the base class implementation.
      base.Dispose(disposing);
   }
}
```

```
Imports Microsoft.Win32.SafeHandles
Imports System.IO

Public Class DisposableStreamResource2 : Inherits DisposableStreamResource
   ' Define additional constants.
   Protected Const GENERIC_WRITE As Integer = &H40000000
   Protected Const OPEN_ALWAYS As Integer = 4

   ' Define additional APIs.
   Protected Declare Function WriteFile Lib "kernel32.dll" (
                              safeHandle As SafeFileHandle, lpBuffer As String,
                              nNumberOfBytesToWrite As Integer, ByRef lpNumberOfBytesWritten As Integer,
                              lpOverlapped As Object) As Boolean

   ' Define locals.
   Private disposed As Boolean = False
   Private filename As String
   Private created As Boolean = False
   Private safeHandle As SafeFileHandle

   Public Sub New(filename As String)
      MyBase.New(filename)
      Me.filename = filename
   End Sub

   Public Sub WriteFileInfo()
      If Not created Then
         Dim hFile As IntPtr = CreateFile(".\FileInfo.txt", GENERIC_WRITE, 0,
                                          IntPtr.Zero, OPEN_ALWAYS,
                                          FILE_ATTRIBUTE_NORMAL, IntPtr.Zero)
         If hFile <> INVALID_HANDLE_VALUE Then
            safeHandle = New SafeFileHandle(hFile, True)
         Else
            Throw New IOException("Unable to create output file.")
         End If
         created = True
      End If
      Dim output As String = String.Format("{0}: {1:N0} bytes {2}", filename, Size,
                                           vbCrLf)
      WriteFile(safeHandle, output, output.Length, 0&, Nothing)
   End Sub

   Protected Overloads Overridable Sub Dispose(disposing As Boolean)
      If disposed Then Exit Sub

      ' Release any managed resources here.
      If disposing Then
         safeHandle.Dispose()
      End If

      disposed = True
      ' Release any unmanaged resources not wrapped by safe handles here.

      ' Call the base class implementation.
      MyBase.Dispose(disposing)
   End Sub
End Class
```

See also

- SuppressFinalize
- IDisposable
- IDisposable.Dispose
- Microsoft.Win32.SafeHandles

- System.Runtime.InteropServices.SafeHandle
- Object.Finalize
- How to: Define and Consume Classes and Structs (C++/CLI)
- Dispose Pattern

minutes to read • Edit Online

The common language runtime's garbage collector reclaims the memory used by managed objects, but types that use unmanaged resources implement the IDisposable interface to allow the memory allocated to these unmanaged resources to be reclaimed. When you finish using an object that implements IDisposable, you should call the object's IDisposable.Dispose implementation. You can do this in one of two ways:

- With the C# `using` statement or the Visual Basic `Using` statement.
- By implementing a `try/finally` block.

The using statement

The `using` statement in C# and the `Using` statement in Visual Basic simplify the code that you must write to create and clean up an object. The `using` statement obtains one or more resources, executes the statements that you specify, and automatically disposes of the object. However, the `using` statement is useful only for objects that are used within the scope of the method in which they are constructed.

The following example uses the `using` statement to create and release a System.IO.StreamReader object.

```
using System;
using System.IO;

public class Example
{
   public static void Main()
   {
      Char[] buffer = new Char[50];
      using (StreamReader s = new StreamReader("File1.txt")) {
         int charsRead = 0;
         while (s.Peek() != -1) {
            charsRead = s.Read(buffer, 0, buffer.Length);
            //
            // Process characters read.
            //
         }
      }
   }
}
```

```
Imports System.IO

Module Example
   Public Sub Main()
      Dim buffer(49) As Char
      Using s As New StreamReader("File1.txt")
         Dim charsRead As Integer
         Do While s.Peek() <> -1
            charsRead = s.Read(buffer, 0, buffer.Length)
            '
            ' Process characters read.
            '
         Loop
      End Using
   End Sub
End Module
```

Note that although the StreamReader class implements the IDisposable interface, which indicates that it uses an unmanaged resource, the example doesn't explicitly call the StreamReader.Dispose method. When the C# or Visual Basic compiler encounters the `using` statement, it emits intermediate language (IL) that is equivalent to the following code that explicitly contains a `try/finally` block.

```
using System;
using System.IO;

public class Example
{
   public static void Main()
   {
      Char[] buffer = new Char[50];
      {
         StreamReader s = new StreamReader("File1.txt");
         try {
            int charsRead = 0;
            while (s.Peek() != -1) {
               charsRead = s.Read(buffer, 0, buffer.Length);
               //
               // Process characters read.
               //
            }
         }
         finally {
            if (s != null)
               ((IDisposable)s).Dispose();
         }
      }
   }
}
```

```
Imports System.IO

Module Example
   Public Sub Main()
      Dim buffer(49) As Char
''      Dim s As New StreamReader("File1.txt")
With s As New StreamReader("File1.txt")
      Try
         Dim charsRead As Integer
         Do While s.Peek() <> -1
            charsRead = s.Read(buffer, 0, buffer.Length)
            '
            ' Process characters read.
            '
         Loop
      Finally
         If s IsNot Nothing Then DirectCast(s, IDisposable).Dispose()
      End Try
End With
   End Sub
End Module
```

The C# `using` statement also allows you to acquire multiple resources in a single statement, which is internally equivalent to nested `using` statements. The following example instantiates two StreamReader objects to read the contents of two different files.

```
using System;
using System.IO;

public class Example
{
   public static void Main()
   {
      Char[] buffer1 = new Char[50], buffer2 = new Char[50];

      using (StreamReader version1 = new StreamReader("file1.txt"),
                          version2 = new StreamReader("file2.txt")) {
         int charsRead1, charsRead2 = 0;
         while (version1.Peek() != -1 && version2.Peek() != -1) {
            charsRead1 = version1.Read(buffer1, 0, buffer1.Length);
            charsRead2 = version2.Read(buffer2, 0, buffer2.Length);
            //
            // Process characters read.
            //
         }
      }
   }
}
```

Try/finally block

Instead of wrapping a `try/finally` block in a `using` statement, you may choose to implement the `try/finally` block directly. This may be your personal coding style, or you might want to do this for one of the following reasons:

- To include a `catch` block to handle any exceptions thrown in the `try` block. Otherwise, any exceptions thrown by the `using` statement are unhandled, as are any exceptions thrown within the `using` block if a `try/catch` block isn't present.
- To instantiate an object that implements IDisposable whose scope is not local to the block within which it is declared.

The following example is similar to the previous example, except that it uses a `try/catch/finally` block to instantiate, use, and dispose of a StreamReader object, and to handle any exceptions thrown by the StreamReader constructor and its ReadToEnd method. Note that the code in the `finally` block checks that the object that implements IDisposable isn't `null` before it calls the Dispose method. Failure to do this can result in a NullReferenceException exception at run time.

```
using System;
using System.Globalization;
using System.IO;

public class Example
{
   public static void Main()
   {
      StreamReader sr = null;
      try {
         sr = new StreamReader("file1.txt");
         String contents = sr.ReadToEnd();
         Console.WriteLine("The file has {0} text elements.",
                           new StringInfo(contents).LengthInTextElements);
      }
      catch (FileNotFoundException) {
         Console.WriteLine("The file cannot be found.");
      }
      catch (IOException) {
         Console.WriteLine("An I/O error has occurred.");
      }
      catch (OutOfMemoryException) {
         Console.WriteLine("There is insufficient memory to read the file.");
      }
      finally {
         if (sr != null) sr.Dispose();
      }
   }
}
```

```
Imports System.Globalization
Imports System.IO

Module Example
   Public Sub Main()
      Dim sr As StreamReader = Nothing
      Try
         sr = New StreamReader("file1.txt")
         Dim contents As String = sr.ReadToEnd()
         Console.WriteLine("The file has {0} text elements.",
                           New StringInfo(contents).LengthInTextElements)
      Catch e As FileNotFoundException
         Console.WriteLine("The file cannot be found.")
      Catch e As IOException
         Console.WriteLine("An I/O error has occurred.")
      Catch e As OutOfMemoryException
         Console.WriteLine("There is insufficient memory to read the file.")
      Finally
         If sr IsNot Nothing Then sr.Dispose()
      End Try
   End Sub
End Module
```

You can follow this basic pattern if you choose to implement or must implement a `try/finally` block, because your programming language doesn't support a `using` statement but does allow direct calls to the Dispose method.

See also

- Cleaning Up Unmanaged Resources
- using Statement (C# Reference)
- Using Statement (Visual Basic)

minutes to read • Edit Online

.NET's garbage collector manages the allocation and release of memory for your application. Each time you create a new object, the common language runtime allocates memory for the object from the managed heap. As long as address space is available in the managed heap, the runtime continues to allocate space for new objects. However, memory is not infinite. Eventually the garbage collector must perform a collection in order to free some memory. The garbage collector's optimizing engine determines the best time to perform a collection, based upon the allocations being made. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory.

Related Topics

TITLE	DESCRIPTION
Fundamentals of Garbage Collection	Describes how garbage collection works, how objects are allocated on the managed heap, and other core concepts.
Garbage Collection and Performance	Describes the performance checks you can use to diagnose garbage collection and performance issues.
Induced Collections	Describes how to make a garbage collection occur.
Latency Modes	Describes the modes that determine the intrusiveness of garbage collection.
Optimization for Shared Web Hosting	Describes how to optimize garbage collection on servers shared by several small Web sites.
Garbage Collection Notifications	Describes how to determine when a full garbage collection is approaching and when it has completed.
Application Domain Resource Monitoring	Describes how to monitor CPU and memory usage by an application domain.
Weak References	Describes features that permit the garbage collector to collect an object while still allowing the application to access that object.

Reference

System.GC

System.GCCollectionMode

System.GCNotificationStatus

System.Runtime.GCLatencyMode

System.Runtime.GCSettings

GCSettings.LargeObjectHeapCompactionMode

Object.Finalize

System.IDisposable

See also

- Cleaning Up Unmanaged Resources

Fundamentals of garbage collection

15 minutes to read • Edit Online

In the common language runtime (CLR), the garbage collector (GC) serves as an automatic memory manager. It provides the following benefits:

- Enables you to develop your application without having to manually free memory.
- Allocates objects on the managed heap efficiently.
- Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors do not have to initialize every data field.
- Provides memory safety by making sure that an object cannot use the content of another object.

This article describes the core concepts of garbage collection.

Fundamentals of memory

The following list summarizes important CLR memory concepts.

- Each process has its own, separate virtual address space. All processes on the same computer share the same physical memory and the page file, if there is one.
- By default, on 32-bit computers, each process has a 2-GB user-mode virtual address space.
- As an application developer, you work only with virtual address space and never manipulate physical memory directly. The garbage collector allocates and frees virtual memory for you on the managed heap.

  If you are writing native code, you use Windows functions to work with the virtual address space. These functions allocate and free virtual memory for you on native heaps.
- Virtual memory can be in three states:
  - Free. The block of memory has no references to it and is available for allocation.
  - Reserved. The block of memory is available for your use and cannot be used for any other allocation request. However, you cannot store data to this memory block until it is committed.
  - Committed. The block of memory is assigned to physical storage.
- Virtual address space can get fragmented. This means that there are free blocks, also known as holes, in the address space. When a virtual memory allocation is requested, the virtual memory manager has to find a single free block that is large enough to satisfy that allocation request. Even if you have 2 GB of free space, the allocation that requires 2 GB will be unsuccessful unless all of that free space is in a single address block.
- You can run out of memory if there isn't enough virtual address space to reserve or physical space to commit.

  The page file is used even if physical memory pressure (that is, demand for physical memory) is low. The first time physical memory pressure is high, the operating system must make room in physical memory to store data, and it backs up some of the data that is in physical memory to the page file. That data is not paged until it's needed, so it's possible to encounter paging in situations where the physical memory pressure is low.

Conditions for a garbage collection

Garbage collection occurs when one of the following conditions is true:

- The system has low physical memory. This is detected by either the low memory notification from the OS or low memory as indicated by the host.
- The memory that is used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
- The GC.Collect method is called. In almost all cases, you do not have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

The managed heap

After the garbage collector is initialized by the CLR, it allocates a segment of memory to store and manage objects. This memory is called the managed heap, as opposed to a native heap in the operating system.

There is a managed heap for each managed process. All threads in the process allocate memory for objects on the same heap.

To reserve memory, the garbage collector calls the Windows VirtualAlloc function and reserves one segment of memory at a time for managed applications. The garbage collector also reserves segments, as needed, and releases segments back to the operating system (after clearing them of any objects) by calling the Windows VirtualFree function.

> **IMPORTANT**
>
> The size of segments allocated by the garbage collector is implementation-specific and is subject to change at any time, including in periodic updates. Your app should never make assumptions about or depend on a particular segment size, nor should it attempt to configure the amount of memory available for segment allocations.

The fewer objects allocated on the heap, the less work the garbage collector has to do. When you allocate objects, do not use rounded-up values that exceed your needs, such as allocating an array of 32 bytes when you need only 15 bytes.

When a garbage collection is triggered, the garbage collector reclaims the memory that is occupied by dead objects. The reclaiming process compacts live objects so that they are moved together, and the dead space is removed, thereby making the heap smaller. This ensures that objects that are allocated together stay together on the managed heap, to preserve their locality.

The intrusiveness (frequency and duration) of garbage collections is the result of the volume of allocations and the amount of survived memory on the managed heap.

The heap can be considered as the accumulation of two heaps: the large object heap and the small object heap.

The large object heap contains very large objects that are 85,000 bytes and larger. The objects on the large object heap are usually arrays. It is rare for an instance object to be extremely large.

> **TIP**
>
> You can configure the threshold size for objects to go on the large object heap.

Generations

The heap is organized into generations so it can handle long-lived and short-lived objects. Garbage collection

primarily occurs with the reclamation of short-lived objects that typically occupy only a small part of the heap. There are three generations of objects on the heap:

- **Generation 0**. This is the youngest generation and contains short-lived objects. An example of a short-lived object is a temporary variable. Garbage collection occurs most frequently in this generation.

  Newly allocated objects form a new generation of objects and are implicitly generation 0 collections. However, if they are large objects, they go on the large object heap in a generation 2 collection.

  Most objects are reclaimed for garbage collection in generation 0 and do not survive to the next generation.

- **Generation 1**. This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.

- **Generation 2**. This generation contains long-lived objects. An example of a long-lived object is an object in a server application that contains static data that's live for the duration of the process.

Garbage collections occur on specific generations as conditions warrant. Collecting a generation means collecting objects in that generation and all its younger generations. A generation 2 garbage collection is also known as a full garbage collection, because it reclaims all objects in all generations (that is, all objects in the managed heap).

Survival and promotions

Objects that are not reclaimed in a garbage collection are known as survivors and are promoted to the next generation. Objects that survive a generation 0 garbage collection are promoted to generation 1; objects that survive a generation 1 garbage collection are promoted to generation 2; and objects that survive a generation 2 garbage collection remain in generation 2.

When the garbage collector detects that the survival rate is high in a generation, it increases the threshold of allocations for that generation. The next collection gets a substantial size of reclaimed memory. The CLR continually balances two priorities: not letting an application's working set get too large by delaying garbage collection and not letting the garbage collection run too frequently.

Ephemeral generations and segments

Because objects in generations 0 and 1 are short-lived, these generations are known as the ephemeral generations.

Ephemeral generations must be allocated in the memory segment that is known as the ephemeral segment. Each new segment acquired by the garbage collector becomes the new ephemeral segment and contains the objects that survived a generation 0 garbage collection. The old ephemeral segment becomes the new generation 2 segment.

The size of the ephemeral segment varies depending on whether a system is 32-bit or 64-bit, and on the type of garbage collector it is running. Default values are shown in the following table.

	32-BIT	64-BIT
Workstation GC	16 MB	256 MB
Server GC	64 MB	4 GB
Server GC with > 4 logical CPUs	32 MB	2 GB
Server GC with > 8 logical CPUs	16 MB	1 GB

The ephemeral segment can include generation 2 objects. Generation 2 objects can use multiple segments (as many as your process requires and memory allows for).

The amount of freed memory from an ephemeral garbage collection is limited to the size of the ephemeral

segment. The amount of memory that is freed is proportional to the space that was occupied by the dead objects.

What happens during a garbage collection

A garbage collection has the following phases:

- A marking phase that finds and creates a list of all live objects.
- A relocating phase that updates the references to the objects that will be compacted.
- A compacting phase that reclaims the space occupied by the dead objects and compacts the surviving objects. The compacting phase moves objects that have survived a garbage collection toward the older end of the segment.

  Because generation 2 collections can occupy multiple segments, objects that are promoted into generation 2 can be moved into an older segment. Both generation 1 and generation 2 survivors can be moved to a different segment, because they are promoted to generation 2.

  Ordinarily, the large object heap (LOH) is not compacted, because copying large objects imposes a performance penalty. However, in .NET Core and in .NET Framework 4.5.1 and later, you can use the GCSettings.LargeObjectHeapCompactionMode property to compact the large object heap on demand. In addition, the LOH is automatically compacted when a hard limit is set by specifying either:

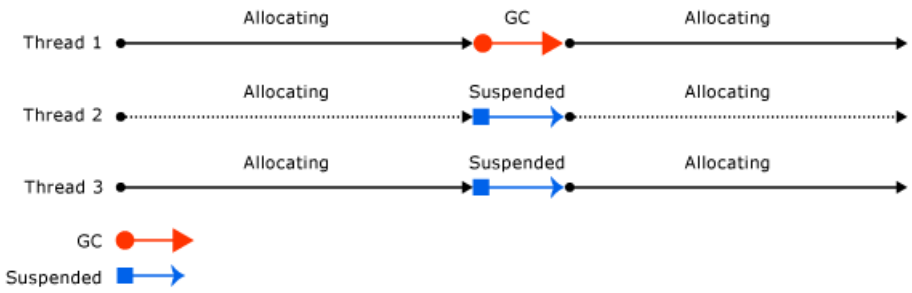
  - a memory limit on a container, or
  - the GCHeapHardLimit or GCHeapHardLimitPercent run-time configuration options

The garbage collector uses the following information to determine whether objects are live:

- **Stack roots**. Stack variables provided by the just-in-time (JIT) compiler and stack walker. JIT optimizations can lengthen or shorten regions of code within which stack variables are reported to the garbage collector.
- **Garbage collection handles**. Handles that point to managed objects and that can be allocated by user code or by the common language runtime.
- **Static data**. Static objects in application domains that could be referencing other objects. Each application domain keeps track of its static objects.

Before a garbage collection starts, all managed threads are suspended except for the thread that triggered the garbage collection.

The following illustration shows a thread that triggers a garbage collection and causes the other threads to be suspended.



Manipulate unmanaged resources

If managed objects reference unmanaged objects by using their native file handles, you have to explicitly free the unmanaged objects, because the garbage collector only tracks memory on the managed heap.

Users of the managed object may not dispose the native resources used by the object. To perform the cleanup, you can make the managed object finalizable. Finalization consists of cleanup actions that execute when the object is no

longer in use. When the managed object dies, it performs cleanup actions that are specified in its finalizer method.

When a finalizable object is discovered to be dead, its finalizer is put in a queue so that its cleanup actions are executed, but the object itself is promoted to the next generation. Therefore, you have to wait until the next garbage collection that occurs on that generation (which is not necessarily the next garbage collection) to determine whether the object has been reclaimed.

For more information about finalization, see Object.Finalize().

Workstation and server garbage collection

The garbage collector is self-tuning and can work in a wide variety of scenarios. You can use a configuration file setting to set the type of garbage collection based on the characteristics of the workload. The CLR provides the following types of garbage collection:

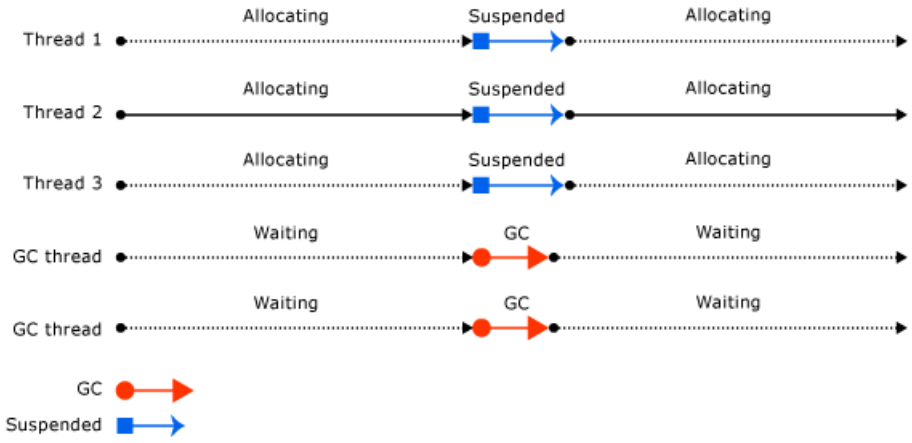
- Workstation garbage collection (GC) is designed for client apps. It is the default GC flavor for standalone apps. For hosted apps, for example, those hosted by ASP.NET, the host determines the default GC flavor.

  Workstation garbage collection can be concurrent or non-concurrent. Concurrent garbage collection enables managed threads to continue operations during a garbage collection. Background garbage collection replaces concurrent garbage collection in .NET Framework 4 and later versions.

- Server garbage collection, which is intended for server applications that need high throughput and scalability.

  - In .NET Core, server garbage collection can be non-concurrent or background.
  - In .NET Framework 4.5 and later versions, server garbage collection can be non-concurrent or background (background garbage collection replaces concurrent garbage collection). In .NET Framework 4 and previous versions, server garbage collection is non-concurrent.

The following illustration shows the dedicated threads that perform the garbage collection on a server:



Compare workstation and server garbage collection

The following are threading and performance considerations for workstation garbage collection:

- The collection occurs on the user thread that triggered the garbage collection and remains at the same priority. Because user threads typically run at normal priority, the garbage collector (which runs on a normal priority thread) must compete with other threads for CPU time. (Threads that run native code are not suspended on either server or workstation garbage collection.)

- Workstation garbage collection is always used on a computer that has only one processor, regardless of the configuration setting.

The following are threading and performance considerations for server garbage collection:

- The collection occurs on multiple dedicated threads that are running at `THREAD_PRIORITY_HIGHEST` priority level.
- A heap and a dedicated thread to perform garbage collection are provided for each CPU, and the heaps are collected at the same time. Each heap contains a small object heap and a large object heap, and all heaps can be accessed by user code. Objects on different heaps can refer to each other.
- Because multiple garbage collection threads work together, server garbage collection is faster than workstation garbage collection on the same size heap.
- Server garbage collection often has larger size segments. However, this is only a generalization: segment size is implementation-specific and is subject to change. Don't make assumptions about the size of segments allocated by the garbage collector when tuning your app.
- Server garbage collection can be resource-intensive. For example, imagine that there are 12 processes that use server GC running on a computer that has 4 processors. If all the processes happen to collect garbage at the same time, they would interfere with each other, as there would be 12 threads scheduled on the same processor. If the processes are active, it's not a good idea to have them all use server GC.

If you're running hundreds of instances of an application, consider using workstation garbage collection with concurrent garbage collection disabled. This will result in less context switching, which can improve performance.

Background workstation garbage collection

In background workstation garbage collection, ephemeral generations (0 and 1) are collected as needed while the collection of generation 2 is in progress. Background workstation garbage collection is performed on a dedicated thread and applies only to generation 2 collections.

Background garbage collection is enabled by default and can be enabled or disabled with the gcConcurrent configuration setting in .NET Framework apps or the System.GC.Concurrent setting in .NET Core apps.

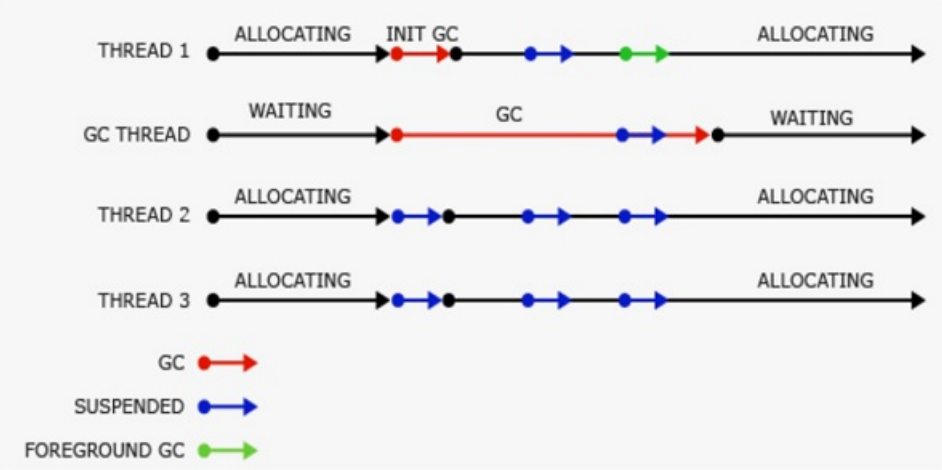
> **NOTE**
> Background garbage collection replaces concurrent garbage collection and is available in .NET Framework 4 and later versions. In .NET Framework 4, it's supported only for workstation garbage collection. Starting with .NET Framework 4.5, background garbage collection is available for both workstation and server garbage collection.

A collection on ephemeral generations during background garbage collection is known as foreground garbage collection. When foreground garbage collections occur, all managed threads are suspended.

When background garbage collection is in progress and you've allocated enough objects in generation 0, the CLR performs a generation 0 or generation 1 foreground garbage collection. The dedicated background garbage collection thread checks at frequent safe points to determine whether there is a request for foreground garbage collection. If there is, the background collection suspends itself so that foreground garbage collection can occur. After the foreground garbage collection is completed, the dedicated background garbage collection thread and user threads resume.

Background garbage collection removes allocation restrictions imposed by concurrent garbage collection, because ephemeral garbage collections can occur during background garbage collection. Background garbage collection can remove dead objects in ephemeral generations. It can also expand the heap if needed during a generation 1 garbage collection.

The following illustration shows background garbage collection performed on a separate dedicated thread on a workstation:



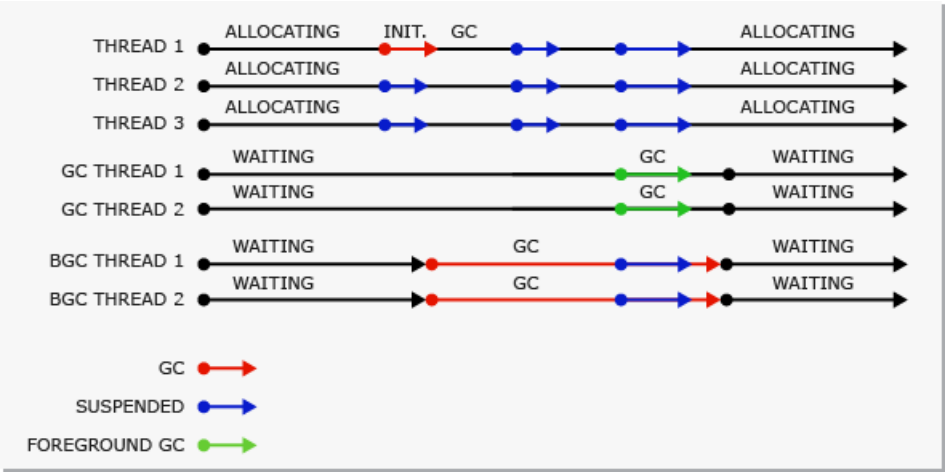
Background server garbage collection

Starting with .NET Framework 4.5, background server garbage collection is the default mode for server garbage collection.

Background server garbage collection functions similarly to background workstation garbage collection, described in the previous section, but there are a few differences:

- Background workstation garbage collection uses one dedicated background garbage collection thread, whereas background server garbage collection uses multiple threads. Typically, there's a dedicated thread for each logical processor.
- Unlike the workstation background garbage collection thread, these threads do not time out.

The following illustration shows background garbage collection performed on a separate dedicated thread on a server:



Concurrent garbage collection

> **TIP**
>
> This section applies to:
>
> - .NET Framework 3.5 and earlier for workstation garbage collection
> - .NET Framework 4 and earlier for server garbage collection
>
> Concurrent garbage is replaced by background garbage collection in later versions.

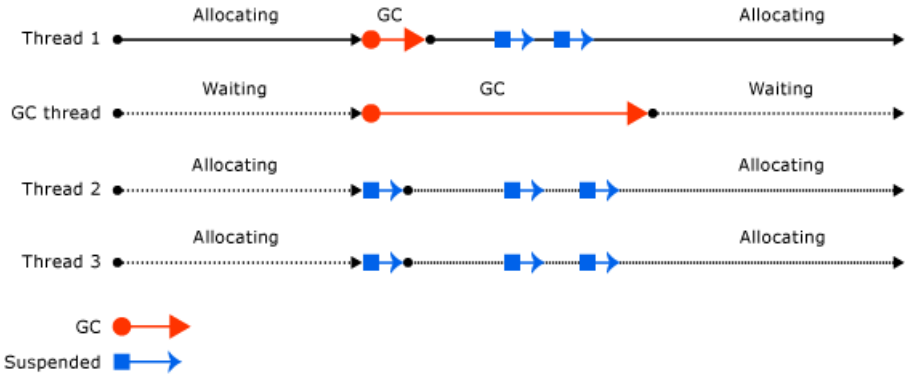
In workstation or server garbage collection, you can enable concurrent garbage collection, which enables threads to run concurrently with a dedicated thread that performs the garbage collection for most of the duration of the

collection. This option affects only garbage collections in generation 2; generations 0 and 1 are always non-concurrent because they finish very fast.

Concurrent garbage collection enables interactive applications to be more responsive by minimizing pauses for a collection. Managed threads can continue to run most of the time while the concurrent garbage collection thread is running. This results in shorter pauses while a garbage collection is occurring.

Concurrent garbage collection is performed on a dedicated thread. By default, the CLR runs workstation garbage collection with concurrent garbage collection enabled. This is true for single-processor and multi-processor computers.

The following illustration shows concurrent garbage collection performed on a separate dedicated thread.



See also

- Configuration options for GC
- Garbage collection

The large object heap on Windows systems

15 minutes to read • Edit Online

The .NET Garbage Collector (GC) divides objects up into small and large objects. When an object is large, some of its attributes become more significant than if the object is small. For instance, compacting it -- that is, copying it in memory elsewhere on the heap -- can be expensive. Because of this, the .NET Garbage Collector places large objects on the large object heap (LOH). In this topic, we'll look at the large object heap in depth. We'll discuss what qualifies an object as a large object, how these large objects are collected, and what kind of performance implications large objects impose.

> **IMPORTANT**
>
> This topic discusses the large object heap in the .NET Framework and .NET Core running on Windows systems only. It does not cover the LOH running on .NET implementations on other platforms.

How an object ends up on the large object heap and how GC handles them

If an object is greater than or equal to 85,000 bytes in size, it's considered a large object. This number was determined by performance tuning. When an object allocation request is for 85,000 or more bytes, the runtime allocates it on the large object heap.

To understand what this means, it's useful to examine some fundamentals about the .NET GC.

The .NET Garbage Collector is a generational collector. It has three generations: generation 0, generation 1, and generation 2. The reason for having 3 generations is that, in a well-tuned app, most objects die in gen0. For example, in a server app, the allocations associated with each request should die after the request is finished. The in-flight allocation requests will make it into gen1 and die there. Essentially, gen1 acts as a buffer between young object areas and long-lived object areas.

Small objects are always allocated in generation 0 and, depending on their lifetime, may be promoted to generation 1 or generation2. Large objects are always allocated in generation 2.

Large objects belong to generation 2 because they are collected only during a generation 2 collection. When a generation is collected, all its younger generation(s) are also collected. For example, when a generation 1 GC happens, both generation 1 and 0 are collected. And when a generation 2 GC happens, the whole heap is collected. For this reason, a generation 2 GC is also called a *full GC*. This article refers to generation 2 GC instead of full GC, but the terms are interchangeable.

Generations provide a logical view of the GC heap. Physically, objects live in managed heap segments. A *managed heap segment* is a chunk of memory that the GC reserves from the OS by calling the VirtualAlloc function on behalf of managed code. When the CLR is loaded, the GC allocates two initial heap segments: one for small objects (the small object heap, or SOH), and one for large objects (the large object heap).

The allocation requests are then satisfied by putting managed objects on these managed heap segments. If the object is less than 85,000 bytes, it is put on the segment for the SOH; otherwise, it is put on an LOH segment. Segments are committed (in smaller chunks) as more and more objects are allocated onto them. For the SOH, objects that survive a GC are promoted to the next generation. Objects that survive a generation 0 collection are now considered generation 1 objects, and so on. However, objects that survive the oldest generation are still considered to be in the oldest generation. In other words, survivors from generation 2 are generation 2 objects; and survivors from the LOH are LOH objects (which are collected with gen2).

User code can only allocate in generation 0 (small objects) or the LOH (large objects). Only the GC can "allocate" objects in generation 1 (by promoting survivors from generation 0) and generation 2 (by promoting survivors from generations 1 and 2).

When a garbage collection is triggered, the GC traces through the live objects and compacts them. But because compaction is expensive, the GC *sweeps* the LOH; it makes a free list out of dead objects that can be reused later to satisfy large object allocation requests. Adjacent dead objects are made into one free object.

.NET Core and .NET Framework (starting with .NET Framework 4.5.1) include the GCSettings.LargeObjectHeapCompactionMode property that allows users to specify that the LOH should be compacted during the next full blocking GC. And in the future, .NET may decide to compact the LOH automatically. This means that, if you allocate large objects and want to make sure that they don't move, you should still pin them.

Figure 1 illustrates a scenario where the GC forms generation 1 after the first generation 0 GC where `Obj1` and `Obj3` are dead, and it forms generation 2 after the first generation 1 GC where `Obj2` and `Obj5` are dead. Note that this and the following figures are only for illustration purposes; they contain very few objects to better show what happens on the heap. In reality, many more objects are typically involved in a GC.

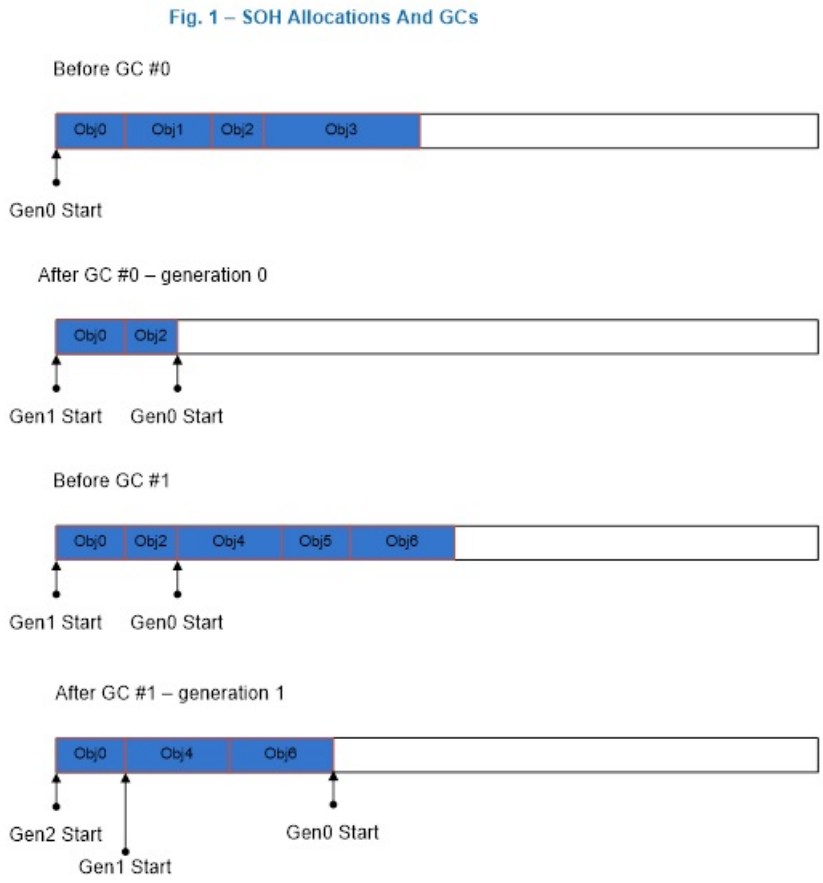


Figure 1: A generation 0 and a generation 1 GC.

Figure 2 shows that after a generation 2 GC which saw that `Obj1` and `Obj2` are dead, the GC forms contiguous free space out of memory that used to be occupied by `Obj1` and `Obj2`, which then was used to satisfy an allocation request for `Obj4`. The space after the last object, `Obj3`, to end of the segment can also be used to satisfy allocation requests.

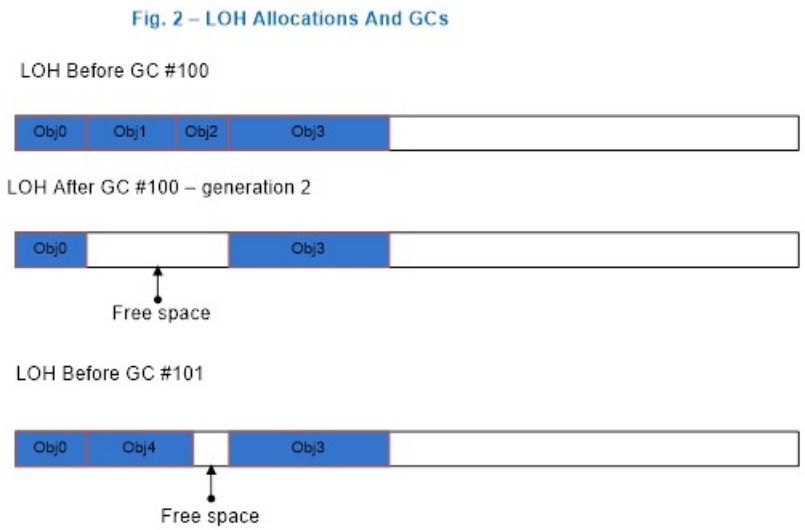


Figure 2: After a generation 2 GC

If there isn't enough free space to accommodate the large object allocation requests, the GC first attempts to acquire more segments from the OS. If that fails, it triggers a generation 2 GC in the hope of freeing up some space.

During a generation 1 or generation 2 GC, the garbage collector releases segments that have no live objects on them back to the OS by calling the VirtualFree function. Space after the last live object to the end of the segment is decommitted (except on the ephemeral segment where gen0/gen1 live, where the garbage collector does keep some committed because your application will be allocating in it right away). And the free spaces remain committed though they are reset, meaning that the OS doesn't need to write data in them back to disk.

Since the LOH is only collected during generation 2 GCs, the LOH segment can only be freed during such a GC. Figure 3 illustrates a scenario where the garbage collector releases one segment (segment 2) back to the OS and decommits more space on the remaining segments. If it needs to use the decommitted space at the end of the segment to satisfy large object allocation requests, it commits the memory again. (For an explanation of commit/decommit, see the documentation for VirtualAlloc.

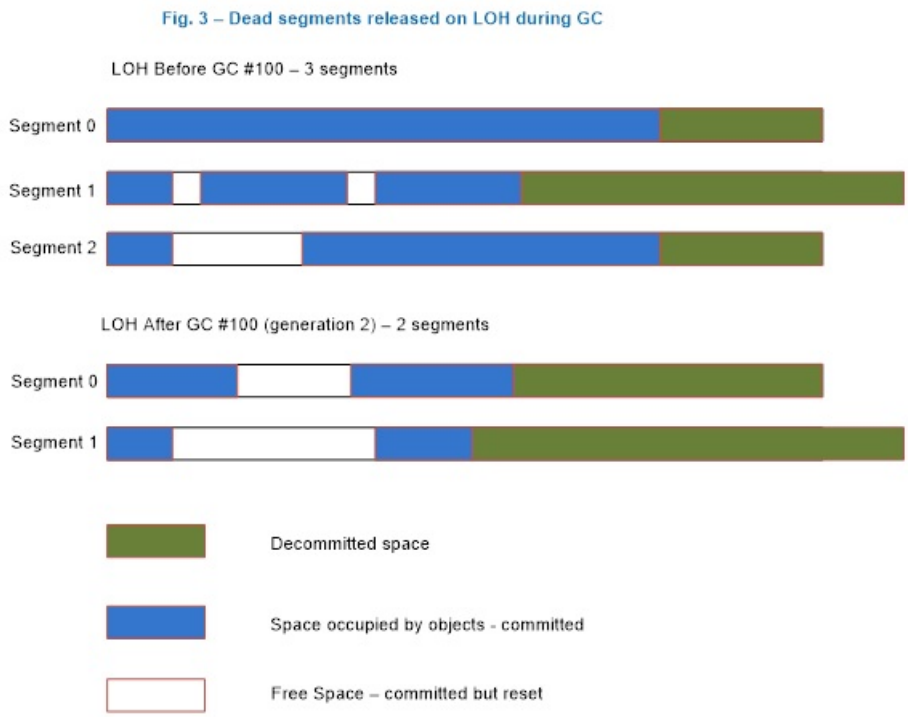


Figure 3: The LOH after a generation 2 GC

When is a large object collected?

In general, a GC occurs when one of the following 3 conditions happens:

- Allocation exceeds the generation 0 or large object threshold.

  The threshold is a property of a generation. A threshold for a generation is set when the garbage collector allocates objects into it. When the threshold is exceeded, a GC is triggered on that generation. When you allocate small or large objects, you consume generation 0 and the LOH's thresholds, respectively. When the garbage collector allocates into generation 1 and 2, it consumes their thresholds. These thresholds are dynamically tuned as the program runs.

  This is the typical case; most GCs happen because of allocations on the managed heap.

- The GC.Collect method is called.

  If the parameterless GC.Collect() method is called or another overload is passed GC.MaxGeneration as an argument, the LOH is collected along with the rest of the managed heap.

- The system is in low memory situation.

  This occurs when the garbage collector receives a high memory notification from the OS. If the garbage collector thinks that doing a generation 2 GC will be productive, it triggers one.

LOH Performance Implications

Allocations on the large object heap impact performance in the following ways.

- Allocation cost.

  The CLR makes the guarantee that the memory for every new object it gives out is cleared. This means the allocation cost of a large object is completely dominated by memory clearing (unless it triggers a GC). If it takes 2 cycles to clear one byte, it takes 170,000 cycles to clear the smallest large object. Clearing the memory of a 16MB object on a 2GHz machine takes approximately 16ms. That's a rather large cost.

- Collection cost.

  Because the LOH and generation 2 are collected together, if either one's threshold is exceeded, a generation 2 collection is triggered. If a generation 2 collection is triggered because of the LOH, generation 2 won't necessarily be much smaller after the GC. If there's not much data on generation 2, this has minimal impact. But if generation 2 is large, it can cause performance problems if many generation 2 GCs are triggered. If many large objects are allocated on a very temporary basis and you have a large SOH, you could be spending too much time doing GCs. In addition, the allocation cost can really add up if you keep allocating and letting go of really large objects.

- Array elements with reference types.

  Very large objects on the LOH are usually arrays (it's very rare to have an instance object that's really large). If the elements of an array are reference-rich, it incurs a cost that is not present if the elements are not reference-rich. If the element doesn't contain any references, the garbage collector doesn't need to go through the array at all. For example, if you use an array to store nodes in a binary tree, one way to implement it is to refer to a node's right and left node by the actual nodes:

```
class Node
{
   Data d;
   Node left;
   Node right;
};

Node[] binary_tr = new Node [num_nodes];
```

If `num_nodes` is large, the garbage collector needs to go through at least two references per element. An alternative approach is to store the index of the right and the left nodes:

```
class Node
{
   Data d;
   uint left_index;
   uint right_index;
} ;
```

Instead of referring the left node's data as `left.d`, you refer to it as `binary_tr[left_index].d`. And the garbage collector doesn't need to look at any references for the left and right node.

Out of the three factors, the first two are usually more significant than the third. Because of this, we recommend that you allocate a pool of large objects that you reuse instead of allocating temporary ones.

Collecting performance data for the LOH

Before you collect performance data for a specific area, you should already have done the following:

1. Found evidence that you should be looking at this area.
2. Exhausted other areas that you know of without finding anything that could explain the performance problem you saw.

See the blog Understand the problem before you try to find a solution for more information on the fundamentals of memory and the CPU.

You can use the following tools to collect data on LOH performance:

- .NET CLR memory performance counters
- ETW events
- A debugger

.NET CLR Memory Performance counters

These performance counters are usually a good first step in investigating performance issues (although we recommend that you use ETW events). You configure Performance Monitor by adding the counters that you want, as Figure 4 shows. The ones that are relevant for the LOH are:

- **Gen 2 Collections**

  Displays the number of times generation 2 GCs have occurred since the process started. The counter is incremented at the end of a generation 2 collection (also called a full garbage collection). This counter displays the last observed value.

- **Large Object Heap size**

  Displays the current size, in bytes, including free space, of the LOH. This counter is updated at the end of a garbage collection, not at each allocation.

A common way to look at performance counters is with Performance Monitor (perfmon.exe). Use "Add Counters" to add the interesting counter for processes that you care about. You can save the performance counter data to a log file, as Figure 4 shows:

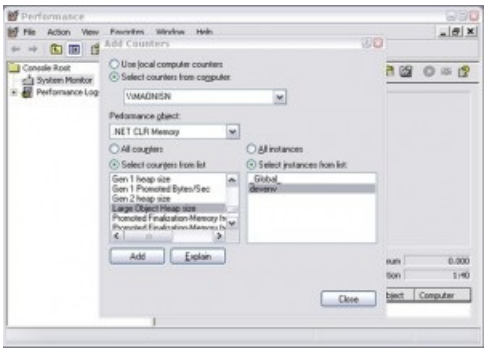


Figure 4: The LOH after a generation 2 GC

Performance counters can also be queried programmatically. Many people collect them this way as part of their routine testing process. When they spot counters with values that are out of the ordinary, they use other means to get more detailed data to help with the investigation.

> **NOTE**
>
> We recommend that you to use ETW events instead of performance counters, because ETW provides much richer information.

ETW events

The garbage collector provides a rich set of ETW events to help you understand what the heap is doing and why. The following blog posts show how to collect and understand GC events with ETW:

- GC ETW Events - 1
- GC ETW Events - 2
- GC ETW Events - 3
- GC ETW Events - 4

To identify excessive generation 2 GCs caused by temporary LOH allocations, look at the Trigger Reason column for GCs. For a simple test that only allocates temporary large objects, you can collect information on ETW events with the following PerfView command line:

```
perfview /GCCollectOnly /AcceptEULA /nogui collect
```

The result is something like this:

Gen 2 for Process 10720: testlarge

GC Events by Time

All times are in msec. Hover over columns for help.

GC Index	Pause Start	Trigger Reason	Gen	Suspend MSec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB	Gen1 Survival Rate %	Gen1 Frag %	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %	LOH MB	LOH Survival Rate %	LOH Frag %	Finalizable Surv MB	Pinned Obj
1	2,248.409	AllocLarge	2N	0.007	0.422	0.0	NaN	0.000	0.00	3.244	0.043	76.19	0.041	0.000	85	0.00	0.007	0	15.25	0.000	0	0.00	0.035	1	0.36	0.00	5
2	2,248.902	AllocLarge	2N	0.002	0.056	43.3	NaN	0.000	0.00	3.244	0.043	76.14	0.041	0.000	0	0.00	0.000	100	0.00	0.007	0	15.53	0.035	1	0.36	0.00	5
3	2,249.003	AllocLarge	2N	0.001	0.052	52.9	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
4	2,249.101	AllocLarge	2N	0.001	0.050	51.8	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
5	2,249.197	AllocLarge	2N	0.001	0.051	51.4	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
6	2,249.293	AllocLarge	2N	0.001	0.051	52.5	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
7	2,249.389	AllocLarge	2N	0.001	0.052	53.1	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
8	2,249.485	AllocLarge	2N	0.001	0.051	52.7	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
9	2,249.572	AllocLarge	2N	0.001	0.043	53.4	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
10	2,249.651	AllocLarge	2N	0.001	0.043	53.2	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
11	2,249.748	AllocLarge	2N	0.001	0.048	46.3	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
12	2,249.834	AllocLarge	2N	0.001	0.068	63.7	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
13	2,249.937	AllocLarge	2N	0.001	0.068	64.9	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
14	2,250.060	AllocLarge	2N	0.001	0.037	39.6	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
15	2,250.143	AllocLarge	2N	0.001	0.051	52.2	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
16	2,250.232	AllocLarge	2N	0.001	0.042	51.2	NaN	0.005	121.73	3.249	0.047	68.47	0.046	0.000	99	0.00	0.005	0	0.49	0.007	100	15.53	0.035	1	0.36	0.00	5
17	2,250.309	AllocLarge	2N	0.001	0.040	52.0	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	99	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
18	2,250.384	AllocLarge	2N	0.001	0.039	51.7	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
19	2,250.459	AllocLarge	2N	0.001	0.038	50.6	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
20	2,250.534	AllocLarge	2N	0.001	0.039	50.9	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
21	2,250.609	AllocLarge	2N	0.001	0.042	53.1	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
22	2,250.680	AllocLarge	2N	0.001	0.042	52.7	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
23	2,250.763	AllocLarge	2N	0.001	0.044	54.6	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
24	2,250.842	AllocLarge	2N	0.001	0.039	51.3	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5

Figure 5: ETW events shown using PerfView

As you can see, all GCs are generation 2 GCs, and they are all triggered by AllocLarge, which means that allocating a large object triggered this GC. We know that these allocations are temporary because the **LOH Survival Rate %**

column says 1%.

You can collect additional ETW events that tell you who allocated these large objects. The following command line:

```
perfview /GCOnly /AcceptEULA /nogui collect
```

collects an AllocationTick event which is fired approximately every 100k worth of allocations. In other words, an event is fired each time a large object is allocated. You can then look at one of the GC Heap Alloc views which show you the callstacks that allocated large objects:

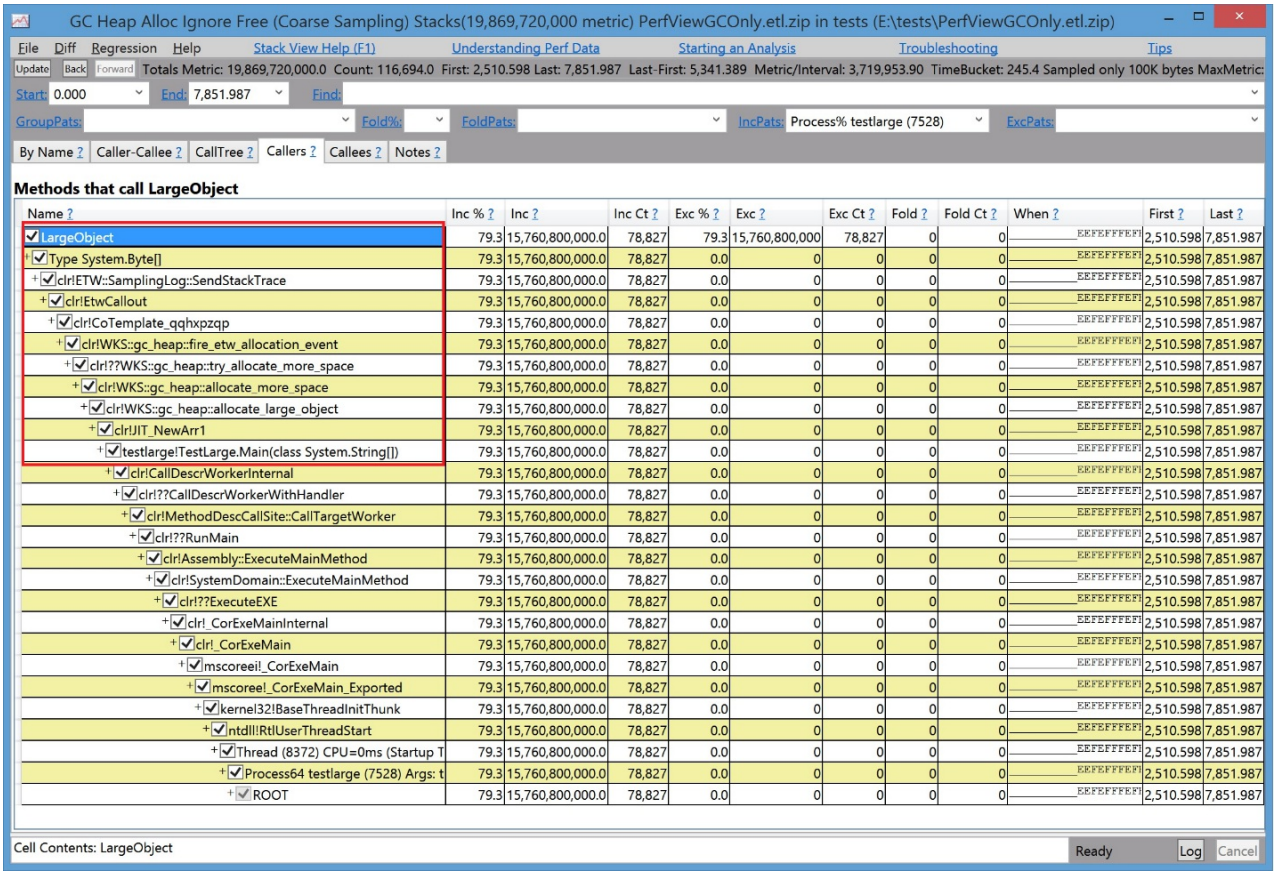


Figure 6: A GC Heap Alloc view

As you can see, this is a very simple test that just allocates large objects from its `Main` method.

A debugger

If all you have is a memory dump and you need to look at what objects are actually on the LOH, you can use the SoS debugger extension provided by .NET.

> **NOTE**
>
> The debugging commands mentioned in this section are applicable to the Windows Debuggers.

The following shows sample output from analyzing the LOH:

```
0:003> .loadby sos mscorwks
0:003> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x013e35ec
sdgeneration 1 starts at 0x013e1b6c
generation 2 starts at 0x013e1000
ephemeral segment allocation context: none
segment   begin allocated     size
0018f2d0 790d5588 790f4b38 0x0001f5b0(128432)
013e0000 013e1000 013e35f8 0x000025f8(9720)
Large object heap starts at 0x023e1000
segment   begin allocated     size
023e0000 023e1000 033db630 0x00ffa630(16754224)
033e0000 033e1000 043cdf98 0x00fecf98(16699288)
043e0000 043e1000 05368b58 0x00f87b58(16284504)
Total Size 0x2f90cc8(49876168)
------------------------------
GC Heap Size 0x2f90cc8(49876168)
0:003> !dumpheap -stat 023e1000 033db630
total 133 objects
Statistics:
MT    Count   TotalSize Class Name
001521d0       66     2081792     Free
7912273c       63     6663696 System.Byte[]
7912254c       4     8008736 System.Object[]
Total 133 objects
```

The LOH heap size is (16,754,224 + 16,699,288 + 16,284,504) = 49,738,016 bytes. Between addresses 023e1000 and 033db630, 8,008,736 bytes are occupied by an array of System.Object objects, 6,663,696 bytes are occupied by an array of System.Byte objects, and 2,081,792 bytes are occupied by free space.

Sometimes, the debugger shows that the total size of the LOH is less than 85,000 bytes. This happens because the runtime itself uses the LOH to allocate some objects that are smaller than a large object.

Because the LOH is not compacted, sometimes the LOH is thought to be the source of fragmentation. Fragmentation means:

- Fragmentation of the managed heap, which is indicated by the amount of free space between managed objects. In SoS, the `!dumpheap -type Free` command displays the amount of free space between managed objects.
- Fragmentation of the virtual memory (VM) address space, which is the memory marked as `MEM_FREE`. You can get it by using various debugger commands in windbg.

  The following example shows fragmentation in the VM space:

```
0:000> !address
00000000 : 00000000 - 00010000
Type     00000000
Protect 00000001 PAGE_NOACCESS
State   00010000 MEM_FREE
Usage   RegionUsageFree
00010000 : 00010000 - 00002000
Type     00020000 MEM_PRIVATE
Protect 00000004 PAGE_READWRITE
State   00001000 MEM_COMMIT
Usage   RegionUsageEnvironmentBlock
00012000 : 00012000 - 0000e000
Type     00000000
Protect 00000001 PAGE_NOACCESS
State   00010000 MEM_FREE
Usage   RegionUsageFree
… [omitted]
-------------------- Usage SUMMARY --------------------------
TotSize (     KB)   Pct(Tots) Pct(Busy)   Usage
701000 (   7172) : 00.34%   20.69%   : RegionUsageIsVAD
7de15000 ( 2062420) : 98.35%   00.00%   : RegionUsageFree
1452000 (   20808) : 00.99%   60.02%   : RegionUsageImage
300000 (   3072) : 00.15%   08.86%   : RegionUsageStack
3000 (     12) : 00.00%   00.03%   : RegionUsageTeb
381000 (   3588) : 00.17%   10.35%   : RegionUsageHeap
0 (       0) : 00.00%   00.00%   : RegionUsagePageHeap
1000 (       4) : 00.00%   00.01%   : RegionUsagePeb
1000 (       4) : 00.00%   00.01%   : RegionUsageProcessParametrs
2000 (       8) : 00.00%   00.02%   : RegionUsageEnvironmentBlock
Tot: 7fff0000 (2097088 KB) Busy: 021db000 (34668 KB)

-------------------- Type SUMMARY --------------------------
TotSize (     KB)   Pct(Tots) Usage
7de15000 ( 2062420) : 98.35%   : <free>
1452000 (   20808) : 00.99%   : MEM_IMAGE
69f000 (   6780) : 00.32%   : MEM_MAPPED
6ea000 (   7080) : 00.34%   : MEM_PRIVATE

-------------------- State SUMMARY --------------------------
TotSize (     KB)   Pct(Tots) Usage
1a58000 (   26976) : 01.29%   : MEM_COMMIT
7de15000 ( 2062420) : 98.35%   : MEM_FREE
783000 (   7692) : 00.37%   : MEM_RESERVE

Largest free region: Base 01432000 - Size 707ee000 (1843128 KB)
```

It's more common to see VM fragmentation caused by temporary large objects that require the garbage collector to frequently acquire new managed heap segments from the OS and to release empty ones back to the OS.

To verify whether the LOH is causing VM fragmentation, you can set a breakpoint on VirtualAlloc and VirtualFree to see who call them. For example, to see who tried to allocate virtual memory chunks larger than 8MBB from the OS, you can set a breakpoint like this:

```
bp kernel32!virtualalloc "j (dwo(@esp+8)>800000) 'kb';'g'"
```

This command breaks into the debugger and shows the call stack only if VirtualAlloc is called with an allocation size greater than 8MB (0x800000).

CLR 2.0 added a feature called *VM Hoarding* that can be useful for scenarios where segments (including on the large and small object heaps) are frequently acquired and released. To specify VM Hoarding, you specify a startup flag called `STARTUP_HOARD_GC_VM` via the hosting API. Instead of releasing empty segments back to the OS, the CLR decommits the memory on these segments and puts them on a standby list. (Note that the CLR doesn't do this for

segments that are too large.) The CLR later uses those segments to satisfy new segment requests. The next time that your app needs a new segment, the CLR uses one from this standby list if it can find one that's big enough.

VM hoarding is also useful for applications that want to hold onto the segments that they already acquired, such as some server apps that are the dominant apps running on the system, to avoid out of memory exceptions.

We strongly recommend that you carefully test your application when you use this feature to ensure your application has fairly stable memory usage.

Garbage Collection and Performance

25 minutes to read • Edit Online

This topic describes issues related to garbage collection and memory usage. It addresses issues that pertain to the managed heap and explains how to minimize the effect of garbage collection on your applications. Each issue has links to procedures that you can use to investigate problems.

Performance Analysis Tools

The following sections describe the tools that are available for investigating memory usage and garbage collection issues. The procedures provided later in this topic refer to these tools.

Memory Performance Counters

You can use performance counters to gather performance data. For instructions, see Runtime Profiling. The .NET CLR Memory category of performance counters, as described in Performance Counters in the .NET Framework, provides information about the garbage collector.

Debugging with SOS

You can use the Windows Debugger (WinDbg) to inspect objects on the managed heap.

To install WinDbg, install Debugging Tools for Windows from the Download Debugging Tools for Windows page.

Garbage Collection ETW Events

Event tracing for Windows (ETW) is a tracing system that supplements the profiling and debugging support provided by the .NET Framework. Starting with the .NET Framework 4, garbage collection ETW events capture useful information for analyzing the managed heap from a statistical point of view. For example, the `GCStart_V1` event, which is raised when a garbage collection is about to occur, provides the following information:

- Which generation of objects is being collected.
- What triggered the garbage collection.
- Type of garbage collection (concurrent or not concurrent).

ETW event logging is efficient and will not mask any performance problems associated with garbage collection. A process can provide its own events in conjunction with ETW events. When logged, both the application's events and the garbage collection events can be correlated to determine how and when heap problems occur. For example, a server application could provide events at the start and end of a client request.

The Profiling API

The common language runtime (CLR) profiling interfaces provide detailed information about the objects that were affected during garbage collection. A profiler can be notified when a garbage collection starts and ends. It can provide reports about the objects on the managed heap, including an identification of objects in each generation. For more information, see Profiling Overview.

Profilers can provide comprehensive information. However, complex profilers can potentially modify an application's behavior.

Application Domain Resource Monitoring

Starting with the .NET Framework 4, Application domain resource monitoring (ARM) enables hosts to monitor CPU and memory usage by application domain. For more information, see Application Domain Resource Monitoring.

Troubleshooting Performance Issues

The first step is to determine whether the issue is actually garbage collection. If you determine that it is, select from the following list to troubleshoot the problem.

- An out-of-memory exception is thrown
- The process uses too much memory
- The garbage collector does not reclaim objects fast enough
- The managed heap is too fragmented
- Garbage collection pauses are too long
- Generation 0 is too big
- CPU usage during a garbage collection is too high

Issue: An Out-of-Memory Exception Is Thrown

There are two legitimate cases for a managed OutOfMemoryException to be thrown:

- Running out of virtual memory.

  The garbage collector allocates memory from the system in segments of a pre-determined size. If an allocation requires an additional segment, but there is no contiguous free block left in the process's virtual memory space, the allocation for the managed heap will fail.

- Not having enough physical memory to allocate.

PERFORMANCE CHECKS
Determine whether the out-of-memory exception is managed.
Determine how much virtual memory can be reserved.
Determine whether there is enough physical memory.

If you determine that the exception is not legitimate, contact Microsoft Customer Service and Support with the following information:

- The stack with the managed out-of-memory exception.
- Full memory dump.
- Data that proves that it is not a legitimate out-of-memory exception, including data that shows that virtual or physical memory is not an issue.

Issue: The Process Uses Too Much Memory

A common assumption is that the memory usage display on the **Performance** tab of Windows Task Manager can indicate when too much memory is being used. However, that display pertains to the working set; it does not provide information about virtual memory usage.

If you determine that the issue is caused by the managed heap, you must measure the managed heap over time to determine any patterns.

If you determine that the problem is not caused by the managed heap, you must use native debugging.

PERFORMANCE CHECKS
Determine how much virtual memory can be reserved.
Determine how much memory the managed heap is committing.
Determine how much memory the managed heap reserves.
Determine large objects in generation 2.
Determine references to objects.

Issue: The Garbage Collector Does Not Reclaim Objects Fast Enough

When it appears as if objects are not being reclaimed as expected for garbage collection, you must determine if there are any strong references to those objects.

You may also encounter this issue if there has been no garbage collection for the generation that contains a dead object, which indicates that the finalizer for the dead object has not been run. For example, this is possible when you are running a single-threaded apartment (STA) application and the thread that services the finalizer queue cannot call into it.

PERFORMANCE CHECKS
Check references to objects.
Determine whether a finalizer has been run.
Determine whether there are objects waiting to be finalized.

Issue: The Managed Heap Is Too fragmented

The fragmentation level is calculated as the ratio of free space over the total allocated memory for the generation. For generation 2, an acceptable level of fragmentation is no more than 20%. Because generation 2 can get very big, the ratio of fragmentation is more important than the absolute value.

Having lots of free space in generation 0 is not a problem because this is the generation where new objects are allocated.

Fragmentation always occurs in the large object heap because it is not compacted. Free objects that are adjacent are naturally collapsed into a single space to satisfy large object allocation requests.

Fragmentation can become a problem in generation 1 and generation 2. If these generations have a large amount of free space after a garbage collection, an application's object usage may need modification, and you should consider re-evaluating the lifetime of long-term objects.

Excessive pinning of objects can increase fragmentation. If fragmentation is high, too many objects could have been pinned.

If fragmentation of virtual memory is preventing the garbage collector from adding segments, the causes could be one of the following:

- Frequent loading and unloading of many small assemblies.
- Holding too many references to COM objects when interoperating with unmanaged code.
- Creation of large transient objects, which causes the large object heap to allocate and free heap segments frequently.

  When hosting the CLR, an application can request that the garbage collector retain its segments. This

reduces the frequency of segment allocations. This is accomplished by using the STARTUP_HOARD_GC_VM flag in the STARTUP_FLAGS Enumeration.

PERFORMANCE CHECKS
Determine the amount of free space in the managed heap.
Determine the number of pinned objects.

If you think that there is no legitimate cause for the fragmentation, contact Microsoft Customer Service and Support.

Issue: Garbage Collection Pauses Are Too Long

Garbage collection operates in soft real time, so an application must be able to tolerate some pauses. A criterion for soft real time is that 95% of the operations must finish on time.

In concurrent garbage collection, managed threads are allowed to run during a collection, which means that pauses are very minimal.

Ephemeral garbage collections (generations 0 and 1) last only a few milliseconds, so decreasing pauses is usually not feasible. However, you can decrease the pauses in generation 2 collections by changing the pattern of allocation requests by an application.

Another, more accurate, method is to use garbage collection ETW events. You can find the timings for collections by adding the time stamp differences for a sequence of events. The whole collection sequence includes suspension of the execution engine, the garbage collection itself, and the resumption of the execution engine.

You can use Garbage Collection Notifications to determine whether a server is about to have a generation 2 collection, and whether rerouting requests to another server could ease any problems with pauses.

PERFORMANCE CHECKS
Determine the length of time in a garbage collection.
Determine what caused a garbage collection.

Issue: Generation 0 Is Too Big

Generation 0 is likely to have a larger number of objects on a 64-bit system, especially when you use server garbage collection instead of workstation garbage collection. This is because the threshold to trigger a generation 0 garbage collection is higher in these environments, and generation 0 collections can get much bigger. Performance is improved when an application allocates more memory before a garbage collection is triggered.

Issue: CPU Usage During a Garbage Collection Is Too High

CPU usage will be high during a garbage collection. If a significant amount of process time is spent in a garbage collection, the number of collections is too frequent or the collection is lasting too long. An increased allocation rate of objects on the managed heap causes garbage collection to occur more frequently. Decreasing the allocation rate reduces the frequency of garbage collections.

You can monitor allocation rates by using the `Allocated Bytes/second` performance counter. For more information, see Performance Counters in the .NET Framework.

The duration of a collection is primarily a factor of the number of objects that survive after allocation. The garbage collector must go through a large amount of memory if many objects remain to be collected. The work to compact the survivors is time-consuming. To determine how many objects were handled during a collection, set a breakpoint in the debugger at the end of a garbage collection for a specified generation.

PERFORMANCE CHECKS
Determine if high CPU usage is caused by garbage collection.
Set a breakpoint at the end of garbage collection.

Troubleshooting Guidelines

This section describes guidelines that you should consider as you begin your investigations.

Workstation or Server Garbage Collection

Determine if you are using the correct type of garbage collection. If your application uses multiple threads and object instances, use server garbage collection instead of workstation garbage collection. Server garbage collection operates on multiple threads, whereas workstation garbage collection requires multiple instances of an application to run their own garbage collection threads and compete for CPU time.

An application that has a low load and that performs tasks infrequently in the background, such as a service, could use workstation garbage collection with concurrent garbage collection disabled.

When to Measure the Managed Heap Size

Unless you are using a profiler, you will have to establish a consistent measuring pattern to effectively diagnose performance issues. Consider the following points to establish a schedule:

- If you measure after a generation 2 garbage collection, the entire managed heap will be free of garbage (dead objects).
- If you measure immediately after a generation 0 garbage collection, the objects in generations 1 and 2 will not be collected yet.
- If you measure immediately before a garbage collection, you will measure as much allocation as possible before the garbage collection starts.
- Measuring during a garbage collection is problematic, because the garbage collector data structures are not in a valid state for traversal and may not be able to give you the complete results. This is by design.
- When you are using workstation garbage collection with concurrent garbage collection, the reclaimed objects are not compacted, so the heap size can be the same or larger (fragmentation can make it appear to be larger).
- Concurrent garbage collection on generation 2 is delayed when the physical memory load is too high.

The following procedure describes how to set a breakpoint so that you can measure the managed heap.

To set a breakpoint at the end of garbage collection

- In WinDbg with the SOS debugger extension loaded, type the following command:

  bp mscorwks!WKS::GCHeap::RestartEE "j (dwo(mscorwks!WKS::GCHeap::GcCondemnedGeneration)==2) 'kb';'g'"

  where **GcCondemnedGeneration** is set to the desired generation. This command requires private symbols.

  This command forces a break if **RestartEE** is executed after generation 2 objects have been reclaimed for garbage collection.

  In server garbage collection, only one thread calls **RestartEE**, so the breakpoint will occur only once during a generation 2 garbage collection.

Performance Check Procedures

This section describes the following procedures to isolate the cause of your performance issue:

- Determine whether the problem is caused by garbage collection.
- Determine whether the out-of-memory exception is managed.
- Determine how much virtual memory can be reserved.
- Determine whether there is enough physical memory.
- Determine how much memory the managed heap is committing.
- Determine how much memory the managed heap reserves.
- Determine large objects in generation 2.
- Determine references to objects.
- Determine whether a finalizer has been run.
- Determine whether there are objects waiting to be finalized.
- Determine the amount of free space in the managed heap.
- Determine the number of pinned objects.
- Determine the length of time in a garbage collection.
- Determine what triggered a garbage collection.
- Determine whether high CPU usage is caused by garbage collection.

To determine whether the problem is caused by garbage collection

- Examine the following two memory performance counters:
  - **% Time in GC**. Displays the percentage of elapsed time that was spent performing a garbage collection after the last garbage collection cycle. Use this counter to determine whether the garbage collector is spending too much time to make managed heap space available. If the time spent in garbage collection is relatively low, that could indicate a resource problem outside the managed heap. This counter may not be accurate when concurrent or background garbage collection is involved.
  - **# Total committed Bytes**. Displays the amount of virtual memory currently committed by the garbage collector. Use this counter to determine whether the memory consumed by the garbage collector is an excessive portion of the memory that your application uses.

  Most of the memory performance counters are updated at the end of each garbage collection. Therefore, they may not reflect the current conditions that you want information about.

To determine whether the out-of-memory exception is managed

1. In the WinDbg or Visual Studio debugger with the SOS debugger extension loaded, type the print exception (**pe**) command:

   !pe

   If the exception is managed, OutOfMemoryException is displayed as the exception type, as shown in the following example.

```
Exception object: 39594518
Exception type: System.OutOfMemoryException
Message: <none>
InnerException: <none>
StackTrace (generated):
```

2. If the output does not specify an exception, you have to determine which thread the out-of-memory exception is from. Type the following command in the debugger to show all the threads with their call stacks:

   ~*kb

   The thread with the stack that has exception calls is indicated by the `RaiseTheException` argument. This is the managed exception object.

```
28adfb44 7923918f 5b61f2b4 00000000 5b61f2b4 mscorwks!RaiseTheException+0xa0
```

3. You can use the following command to dump nested exceptions.

   !pe -nested

   If you do not find any exceptions, the out-of-memory exception originated from unmanaged code.

To determine how much virtual memory can be reserved

- In WinDbg with the SOS debugger extension loaded, type the following command to get the largest free region:

  !address -summary

  The largest free region is displayed as shown in the following output.

```
Largest free region: Base 54000000 - Size 0003A980
```

  In this example, the size of the largest free region is approximately 24000 KB (3A980 in hexadecimal). This region is much smaller than what the garbage collector needs for a segment.

  -or-

- Use the **vmstat** command:

  !vmstat

  The largest free region is the largest value in the MAXIMUM column, as shown in the following output.

```
TYPE        MINIMUM   MAXIMUM     AVERAGE   BLK COUNT   TOTAL
~~~~        ~~~~~~~   ~~~~~~~     ~~~~~~~   ~~~~~~~~~~  ~~~~
Free:
Small       8K        64K         46K       36          1,671K
Medium      80K       864K        349K      3           1,047K
Large       1,384K    1,278,848K  151,834K  12          1,822,015K
Summary     8K        1,278,848K  35,779K   51          1,824,735K
```

To determine whether there is enough physical memory

1. Start Windows Task Manager.
2. On the **Performance** tab, look at the committed value. (In Windows 7, look at **Commit (KB)** in the **System group**.)

If the **Total** is close to the **Limit**, you are running low on physical memory.

To determine how much memory the managed heap is committing

- Use the `# Total committed bytes` memory performance counter to get the number of bytes that the managed heap is committing. The garbage collector commits chunks on a segment as needed, not all at the same time.

  > **NOTE**
  > Do not use the `# Bytes in all Heaps` performance counter, because it does not represent actual memory usage by the managed heap. The size of a generation is included in this value and is actually its threshold size, that is, the size that induces a garbage collection if the generation is filled with objects. Therefore, this value is usually zero.

To determine how much memory the managed heap reserves

- Use the `# Total reserved bytes` memory performance counter.

  The garbage collector reserves memory in segments, and you can determine where a segment starts by using the **eeheap** command.

  > **IMPORTANT**
  > Although you can determine the amount of memory the garbage collector allocates for each segment, segment size is implementation-specific and is subject to change at any time, including in periodic updates. Your app should never make assumptions about or depend on a particular segment size, nor should it attempt to configure the amount of memory available for segment allocations.

- In the WinDbg or Visual Studio debugger with the SOS debugger extension loaded, type the following command:

  !eeheap -gc

  The result is as follows.

```
Number of GC Heaps: 2
------------------------------
Heap 0 (002db550)
generation 0 starts at 0x02abe29c
generation 1 starts at 0x02abdd08
generation 2 starts at 0x02ab0038
ephemeral segment allocation context: none
  segment    begin allocated     size
02ab0000 02ab0038  02aceff4 0x0001efbc(126908)
Large object heap starts at 0x0aab0038
  segment    begin allocated     size
0aab0000 0aab0038  0aab2278 0x00002240(8768)
Heap Size   0x211fc(135676)
------------------------------
Heap 1 (002dc958)
generation 0 starts at 0x06ab1bd8
generation 1 starts at 0x06ab1bcc
generation 2 starts at 0x06ab0038
ephemeral segment allocation context: none
  segment    begin allocated     size
06ab0000 06ab0038  06ab3be4 0x00003bac(15276)
Large object heap starts at 0x0cab0038
  segment    begin allocated     size
0cab0000 0cab0038  0cab0048 0x00000010(16)
Heap Size    0x3bbc(15292)
------------------------------
GC Heap Size   0x24db8(150968)
```

The addresses indicated by "segment" are the starting addresses of the segments.

To determine large objects in generation 2

- In the WinDbg or Visual Studio debugger with the SOS debugger extension loaded, type the following command:

  !dumpheap –stat

  If the managed heap is big, **dumpheap** may take a while to finish.

  You can start analyzing from the last few lines of the output, because they list the objects that use the most space. For example:

```
2c6108d4   173712     14591808 DevExpress.XtraGrid.Views.Grid.ViewInfo.GridCellInfo
00155f80      533     15216804      Free
7a747c78   791070     15821400 System.Collections.Specialized.ListDictionary+DictionaryNode
7a747bac   700930     19626040 System.Collections.Specialized.ListDictionary
2c64e36c    78644     20762016 DevExpress.XtraEditors.ViewInfo.TextEditViewInfo
79124228   121143     29064120 System.Object[]
035f0ee4    81626     35588936 Toolkit.TlkOrder
00fcae40     6193     44911636 WaveBasedStrategy.Tick_Snap[]
791242ec    40182     90664128 System.Collections.Hashtable+bucket[]
790fa3e0  3154024    137881448 System.String
Total 8454945 objects
```

  The last object listed is a string and occupies the most space. You can examine your application to see how your string objects can be optimized. To see strings that are between 150 and 200 bytes, type the following:

  !dumpheap -type System.String -min 150 -max 200

  An example of the results is as follows.

```
Address  MT           Size  Gen
1875d2c0 790fa3e0      152    2 System.String HighlightNullStyle_Blotter_PendingOrder-
11_Blotter_PendingOrder-11
…
```

Using an integer instead of a string for an ID can be more efficient. If the same string is being repeated thousands of times, consider string interning. For more information about string interning, see the reference topic for the String.Intern method.

To determine references to objects

- In WinDbg with the SOS debugger extension loaded, type the following command to list references to objects:

  !gcroot

  `-or-`

- To determine the references for a specific object, include the address:

  !gcroot 1c37b2ac

  Roots found on stacks may be false positives. For more information, use the command `!help gcroot`.

```
ebx:Root:19011c5c(System.Windows.Forms.Application+ThreadContext)->
19010b78(DemoApp.FormDemoApp)->
19011158(System.Windows.Forms.PropertyStore)->
… [omitted]
1c3745ec(System.Data.DataTable)->
1c3747a8(System.Data.DataColumnCollection)->
1c3747f8(System.Collections.Hashtable)->
1c376590(System.Collections.Hashtable+bucket[])->
1c376c98(System.Data.DataColumn)->
1c37b270(System.Data.Common.DoubleStorage)->
1c37b2ac(System.Double[])
Scan Thread 0 OSTHread 99c
Scan Thread 6 OSTHread 484
```

The **gcroot** command can take a long time to finish. Any object that is not reclaimed by garbage collection is a live object. This means that some root is directly or indirectly holding onto the object, so **gcroot** should return path information to the object. You should examine the graphs returned and see why these objects are still referenced.

To determine whether a finalizer has been run

- Run a test program that contains the following code:

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

If the test resolves the problem, this means that the garbage collector was not reclaiming objects, because the finalizers for those objects had been suspended. The GC.WaitForPendingFinalizers method enables the finalizers to complete their tasks, and fixes the problem.

To determine whether there are objects waiting to be finalized

1. In the WinDbg or Visual Studio debugger with the SOS debugger extension loaded, type the following command:

   !finalizequeue

Look at the number of objects that are ready for finalization. If the number is high, you must examine why these finalizers cannot progress at all or cannot progress fast enough.

2. To get an output of threads, type the following command:

   threads -special

   This command provides output such as the following.

```
   OSID     Special thread type
2    cd0    DbgHelper
3    c18    Finalizer
4    df0    GC SuspendEE
```

   The finalizer thread indicates which finalizer, if any, is currently being run. When a finalizer thread is not running any finalizers, it is waiting for an event to tell it to do its work. Most of the time you will see the finalizer thread in this state because it runs at THREAD_HIGHEST_PRIORITY and is supposed to finish running finalizers, if any, very quickly.

To determine the amount of free space in the managed heap

- In the WinDbg or Visual Studio debugger with the SOS debugger extension loaded, type the following command:

  !dumpheap -type Free -stat

  This command displays the total size of all the free objects on the managed heap, as shown in the following example.

```
total 230 objects
Statistics:
      MT    Count    TotalSize Class Name
00152b18      230     40958584      Free
Total 230 objects
```

- To determine the free space in generation 0, type the following command for memory consumption information by generation:

  !eeheap -gc

  This command displays output similar to the following. The last line shows the ephemeral segment.

```
Heap 0 (0015ad08)
generation 0 starts at 0x49521f8c
generation 1 starts at 0x494d7f64
generation 2 starts at 0x007f0038
ephemeral segment allocation context: none
segment  begin     allocated  size
00178250 7a80d84c  7a82f1cc   0x00021980(137600)
00161918 78c50e40  78c7056c   0x0001f72c(128812)
007f0000 007f0038  047eed28   0x03ffecf0(67103984)
3a120000 3a120038  3a3e84f8   0x002c84c0(2917568)
46120000 46120038  49e05d04   0x03ce5ccc(63855820)
```

- Calculate the space used by generation 0:

  ? 49e05d04-0x49521f8c

  The result is as follows. Generation 0 is approximately 9 MB.

```
Evaluate expression: 9321848 = 008e3d78
```

- The following command dumps the free space within the generation 0 range:

  !dumpheap -type Free -stat 0x49521f8c 49e05d04

  The result is as follows.

```
------------------------------
Heap 0
total 409 objects
------------------------------
Heap 1
total 0 objects
------------------------------
Heap 2
total 0 objects
------------------------------
Heap 3
total 0 objects
------------------------------
total 409 objects
Statistics:
      MT    Count TotalSize Class Name
0015a498      409   7296540      Free
Total 409 objects
```

  This output shows that the generation 0 portion of the heap is using 9 MB of space for objects and has 7 MB free. This analysis shows the extent to which generation 0 contributes to fragmentation. This amount of heap usage should be discounted from the total amount as the cause of fragmentation by long-term objects.

To determine the number of pinned objects

- In the WinDbg or Visual Studio debugger with the SOS debugger extension loaded, type the following command:

  !gchandles

  The statistics displayed includes the number of pinned handles, as the following example shows.

```
GC Handle Statistics:
Strong Handles:      29
Pinned Handles:      10
```

To determine the length of time in a garbage collection

- Examine the `% Time in GC` memory performance counter.

  The value is calculated by using a sample interval time. Because the counters are updated at the end of each garbage collection, the current sample will have the same value as the previous sample if no collections occurred during the interval.

  Collection time is obtained by multiplying the sample interval time with the percentage value.

  The following data shows four sampling intervals of two seconds, for an 8-second study. The `Gen0`, `Gen1`, and `Gen2` columns show the number of garbage collections that occurred during that interval for that generation.

```
Interval    Gen0    Gen1    Gen2    % Time in GC
        1       9       3       1              10
        2      10       3       1               1
        3      11       3       1               3
        4      11       3       1               3
```

This information does not show when the garbage collection occurred, but you can determine the number of garbage collections that occurred in an interval of time. Assuming the worst case, the tenth generation 0 garbage collection finished at the start of the second interval, and the eleventh generation 0 garbage collection finished at the end of the fifth interval. The time between the end of the tenth and the end of the eleventh garbage collection is about 2 seconds, and the performance counter shows 3%, so the duration of the eleventh generation 0 garbage collection was (2 seconds * 3% = 60ms).

In this example, there are 5 periods.

```
Interval    Gen0    Gen1    Gen2     % Time in GC
        1       9       3       1               3
        2      10       3       1               1
        3      11       4       2               1
        4      11       4       2               1
        5      11       4       2              20
```

The second generation 2 garbage collection started during the third interval and finished at the fifth interval. Assuming the worst case, the last garbage collection was for a generation 0 collection that finished at the start of the second interval, and the generation 2 garbage collection finished at the end of the fifth interval. Therefore, the time between the end of the generation 0 garbage collection and the end of the generation 2 garbage collection is 4 seconds. Because the `% Time in GC` counter is 20%, the maximum amount of time the generation 2 garbage collection could have taken is (4 seconds * 20% = 800ms).

- Alternatively, you can determine the length of a garbage collection by using garbage collection ETW events, and analyze the information to determine the duration of garbage collection.

  For example, the following data shows an event sequence that occurred during a non-concurrent garbage collection.

```
Timestamp    Event name
513052        GCSuspendEEBegin_V1
513078        GCSuspendEEEnd
513090        GCStart_V1
517890        GCEnd_V1
517894        GCHeapStats
517897        GCRestartEEBegin
517918        GCRestartEEEnd
```

  Suspending the managed thread took 26us (`GCSuspendEEEnd` – `GCSuspendEEBegin_V1`).

  The actual garbage collection took 4.8ms (`GCEnd_V1` – `GCStart_V1`).

  Resuming the managed threads took 21us (`GCRestartEEEnd` – `GCRestartEEBegin`).

  The following output provides an example for background garbage collection, and includes the process, thread, and event fields. (Not all data is shown.)

```
timestamp(us)    event name            process    thread    event field
42504385        GCSuspendEEBegin_V1    Test.exe    4372             1
42504648        GCSuspendEEEnd         Test.exe    4372
42504816        GCStart_V1             Test.exe    4372        102019
42504907        GCStart_V1             Test.exe    4372        102020
42514170        GCEnd_V1               Test.exe    4372
42514204        GCHeapStats            Test.exe    4372        102020
42832052        GCRestartEEBegin       Test.exe    4372
42832136        GCRestartEEEnd         Test.exe    4372
63685394        GCSuspendEEBegin_V1    Test.exe    4744             6
63686347        GCSuspendEEEnd         Test.exe    4744
63784294        GCRestartEEBegin       Test.exe    4744
63784407        GCRestartEEEnd         Test.exe    4744
89931423        GCEnd_V1               Test.exe    4372        102019
89931464        GCHeapStats            Test.exe    4372
```

The `GCStart_V1` event at 42504816 indicates that this is a background garbage collection, because the last field is `1`. This becomes garbage collection No. 102019.

The `GCStart` event occurs because there is a need for an ephemeral garbage collection before you start a background garbage collection. This becomes garbage collection No. 102020.

At 42514170, garbage collection No. 102020 finishes. The managed threads are restarted at this point. This is completed on thread 4372, which triggered this background garbage collection.

On thread 4744, a suspension occurs. This is the only time at which the background garbage collection has to suspend managed threads. This duration is approximately 99ms ((63784407-63685394)/1000).

The `GCEnd` event for the background garbage collection is at 89931423. This means that the background garbage collection lasted for about 47seconds ((89931423-42504816)/1000).

While the managed threads are running, you can see any number of ephemeral garbage collections occurring.

To determine what triggered a garbage collection

- In the WinDbg or Visual Studio debugger with the SOS debugger extension loaded, type the following command to show all the threads with their call stacks:

  ~*kb

  This command displays output similar to the following.

```
0012f3b0 79ff0bf8 mscorwks!WKS::GCHeap::GarbageCollect
0012f454 30002894 mscorwks!GCInterface::CollectGeneration+0xa4
0012f490 79fa22bd fragment_ni!request.Main(System.String[])+0x48
```

  If the garbage collection was caused by a low memory notification from the operating system, the call stack is similar, except that the thread is the finalizer thread. The finalizer thread gets an asynchronous low memory notification and induces the garbage collection.

  If the garbage collection was caused by memory allocation, the stack appears as follows:

```
0012f230 7a07c551 mscorwks!WKS::GCHeap::GarbageCollectGeneration
0012f2b8 7a07cba8 mscorwks!WKS::gc_heap::try_allocate_more_space+0x1a1
0012f2d4 7a07cefb mscorwks!WKS::gc_heap::allocate_more_space+0x18
0012f2f4 7a02a51b mscorwks!WKS::GCHeap::Alloc+0x4b
0012f310 7a02ae4c mscorwks!Alloc+0x60
0012f364 7a030e46 mscorwks!FastAllocatePrimitiveArray+0xbd
0012f424 300027f4 mscorwks!JIT_NewArr1+0x148
000af70f 3000299f fragment_ni!request..ctor(Int32, Single)+0x20c
0000002a 79fa22bd fragment_ni!request.Main(System.String[])+0x153
```

A just-in-time helper (`JIT_New*`) eventually calls `GCHeap::GarbageCollectGeneration` . If you determine that generation 2 garbage collections are caused by allocations, you must determine which objects are collected by a generation 2 garbage collection and how to avoid them. That is, you want to determine the difference between the start and the end of a generation 2 garbage collection, and the objects that caused the generation 2 collection.

For example, type the following command in the debugger to show the beginning of a generation 2 collection:

!dumpheap –stat

Example output (abridged to show the objects that use the most space):

```
79124228    31857      9862328 System.Object[]
035f0384    25668     11601936 Toolkit.TlkPosition
00155f80    21248     12256296      Free
79103b6c   297003     13068132 System.Threading.ReaderWriterLock
7a747ad4   708732     14174640 System.Collections.Specialized.HybridDictionary
7a747c78   786498     15729960 System.Collections.Specialized.ListDictionary+DictionaryNode
7a747bac   700298     19608344 System.Collections.Specialized.ListDictionary
035f0ee4    89192     38887712 Toolkit.TlkOrder
00fcae40     6193     44911636 WaveBasedStrategy.Tick_Snap[]
7912c444    91616     71887080 System.Double[]
791242ec    32451     82462728 System.Collections.Hashtable+bucket[]
790fa3e0  2459154    112128436 System.String
Total 6471774 objects
```

Repeat the command at the end of generation 2:

!dumpheap –stat

Example output (abridged to show the objects that use the most space):

```
79124228    26648      9314256 System.Object[]
035f0384    25668     11601936 Toolkit.TlkPosition
79103b6c   296770     13057880 System.Threading.ReaderWriterLock
7a747ad4   708730     14174600 System.Collections.Specialized.HybridDictionary
7a747c78   786497     15729940 System.Collections.Specialized.ListDictionary+DictionaryNode
7a747bac   700298     19608344 System.Collections.Specialized.ListDictionary
00155f80    13806     34007212      Free
035f0ee4    89187     38885532 Toolkit.TlkOrder
00fcae40     6193     44911636 WaveBasedStrategy.Tick_Snap[]
791242ec    32370     82359768 System.Collections.Hashtable+bucket[]
790fa3e0  2440020    111341808 System.String
Total 6417525 objects
```

The `double[]` objects disappeared from the end of the output, which means that they were collected. These objects account for approximately 70 MB. The remaining objects did not change much. Therefore, these `double[]` objects were the reason why this generation 2 garbage collection occurred. Your next step is to determine why the `double[]` objects are there and why they died. You can ask the code developer where

these objects came from, or you can use the **gcroot** command.

To determine whether high CPU usage is caused by garbage collection

- Correlate the `% Time in GC` memory performance counter value with the process time.

  If the `% Time in GC` value spikes at the same time as process time, garbage collection is causing a high CPU usage. Otherwise, profile the application to find where the high usage is occurring.

See also

- Garbage Collection

minutes to read • Edit Online

In most cases, the garbage collector can determine the best time to perform a collection, and you should let it run independently. There are rare situations when a forced collection might improve your application's performance. In these cases, you can induce garbage collection by using the GC.Collect method to force a garbage collection.

Use the GC.Collect method when there is a significant reduction in the amount of memory being used at a specific point in your application's code. For example, if your application uses a complex dialog box that has several controls, calling Collect when the dialog box is closed could improve performance by immediately reclaiming the memory used by the dialog box. Be sure that your application is not inducing garbage collection too frequently, because that can decrease performance if the garbage collector is trying to reclaim objects at non-optimal times. You can supply a GCCollectionMode.Optimized enumeration value to the Collect method to collect only when collection would be productive, as discussed in the next section.

GC collection mode

You can use one of the GC.Collect method overloads that includes a GCCollectionMode value to specify the behavior for a forced collection as follows.

`GCCOLLECTIONMODE` VALUE	DESCRIPTION
Default	Uses the default garbage collection setting for the running version of .NET.
Forced	Forces garbage collection to occur immediately. This is equivalent to calling the GC.Collect() overload. It results in a full blocking collection of all generations. You can also compact the large object heap by setting the GCSettings.LargeObjectHeapCompactionMode property to GCLargeObjectHeapCompactionMode.CompactOnce before forcing an immediate full blocking garbage collection.
Optimized	Enables the garbage collector to determine whether the current time is optimal to reclaim objects. The garbage collector could determine that a collection would not be productive enough to be justified, in which case it will return without reclaiming objects.

Background or blocking collections

You can call the GC.Collect(Int32, GCCollectionMode, Boolean) method overload to specify whether an induced collection is blocking or not. The type of collection performed depends on a combination of the method's `mode` and `blocking` parameters. `mode` is a member of the GCCollectionMode enumeration, and `blocking` is a Boolean value. The following table summarizes the interaction of the `mode` and `blocking` arguments.

`MODE`	`BLOCKING` = `TRUE`	`BLOCKING` = `FALSE`

`MODE`	`BLOCKING` = `TRUE`	`BLOCKING` = `FALSE`
Forced or Default	A blocking collection is performed as soon as possible. If a background collection is in progress and generation is 0 or 1, the Collect(Int32, GCCollectionMode, Boolean) method immediately triggers a blocking collection and returns when the collection is finished. If a background collection is in progress and the `generation` parameter is 2, the method waits until the background collection is finished, triggers a blocking generation 2 collection, and then returns.	A collection is performed as soon as possible. The Collect(Int32, GCCollectionMode, Boolean) method requests a background collection, but this is not guaranteed; depending on the circumstances, a blocking collection may still be performed. If a background collection is already in progress, the method returns immediately.
Optimized	A blocking collection may be performed, depending on the state of the garbage collector and the `generation` parameter. The garbage collector tries to provide optimal performance.	A collection may be performed, depending on the state of the garbage collector. The Collect(Int32, GCCollectionMode, Boolean) method requests a background collection, but this is not guaranteed; depending on the circumstances, a blocking collection may still be performed. The garbage collector tries to provide optimal performance. If a background collection is already in progress, the method returns immediately.

See also

- Latency Modes
- Garbage Collection

Latency modes

2 minutes to read • Edit Online

To reclaim objects, the garbage collector (GC) must stop all the executing threads in an application. The period of time during which the garbage collector is active is referred to as its *latency*.

In some situations, such as when an application retrieves data or displays content, a full garbage collection can occur at a critical time and impede performance. You can adjust the intrusiveness of the garbage collector by setting the GCSettings.LatencyMode property to one of the System.Runtime.GCLatencyMode values.

Low latency settings

Using a "low" latency setting means the garbage collector intrudes less in your application. Garbage collection is more conservative about reclaiming memory.

The System.Runtime.GCLatencyMode enumeration provides two low latency settings:

- GCLatencyMode.LowLatency suppresses generation 2 collections and performs only generation 0 and 1 collections. It can be used only for short periods of time. Over longer periods, if the system is under memory pressure, the garbage collector will trigger a collection, which can briefly pause the application and disrupt a time-critical operation. This setting is available only for workstation garbage collection.
- GCLatencyMode.SustainedLowLatency suppresses foreground generation 2 collections and performs only generation 0, 1, and background generation 2 collections. It can be used for longer periods of time, and is available for both workstation and server garbage collection. This setting cannot be used if background garbage collection is disabled.

During low latency periods, generation 2 collections are suppressed unless the following occurs:

- The system receives a low memory notification from the operating system.
- Application code induces a collection by calling the GC.Collect method and specifying 2 for the `generation` parameter.

Scenarios

The following table lists the application scenarios for using the GCLatencyMode values:

LATENCY MODE	APPLICATION SCENARIOS
Batch	For applications that have no user interface (UI) or server-side operations. When background garbage collection is disabled, this is the default mode for workstation and server garbage collection. Batch mode also overrides the gcConcurrent setting, that is, it prevents background or concurrent collections.
Interactive	For most applications that have a UI. This is the default mode for workstation and server garbage collection. However, if an app is hosted, the garbage collector settings of the hosting process take precedence.

LATENCY MODE	APPLICATION SCENARIOS
LowLatency	For applications that have short-term, time-sensitive operations during which interruptions from the garbage collector could be disruptive. For example, applications that render animations or data acquisition functions.
SustainedLowLatency	For applications that have time-sensitive operations for a contained but potentially longer duration of time during which interruptions from the garbage collector could be disruptive. For example, applications that need quick response times as market data changes during trading hours. This mode results in a larger managed heap size than other modes. Because it does not compact the managed heap, higher fragmentation is possible. Ensure that sufficient memory is available.

Guidelines for using low latency

When you use GCLatencyMode.LowLatency mode, consider the following guidelines:

- Keep the period of time in low latency as short as possible.
- Avoid allocating high amounts of memory during low latency periods. Low memory notifications can occur because garbage collection reclaims fewer objects.
- While in the low latency mode, minimize the number of new allocations, in particular allocations onto the large object heap and pinned objects.
- Be aware of threads that could be allocating. Because the LatencyMode property setting is process-wide, OutOfMemoryException exceptions can be generated on any thread that is allocating.
- Wrap the low latency code in constrained execution regions. For more information, see Constrained execution regions.
- You can force generation 2 collections during a low latency period by calling the GC.Collect(Int32, GCCollectionMode) method.

See also

- System.GC
- Induced Collections
- Garbage Collection

minutes to read • Edit Online

If you are the administrator for a server that is shared by hosting several small Web sites, you can optimize performance and increase site capacity by adding the following `gcTrimCommitOnLowMemory` setting to the `runtime` node in the Aspnet.config file in the .NET directory:

```
<gcTrimCommitOnLowMemory enabled="true|false"/>
```

> **NOTE**
> This setting is recommended only for shared Web hosting scenarios.

Because the garbage collector retains memory for future allocations, its committed space can be more than what is strictly needed. You can reduce this space to accommodate times when there is a heavy load on system memory. Reducing this committed space improves performance and expands the capacity to host more sites.

When the `gcTrimCommitOnLowMemory` setting is enabled, the garbage collector evaluates the system memory load and enters a trimming mode when the load reaches 90%. It maintains the trimming mode until the load drops under 85%.

When conditions permit, the garbage collector can decide that the `gcTrimCommitOnLowMemory` setting will not help the current application and ignore it.

Example

The following XML fragment shows how to enable the `gcTrimCommitOnLowMemory` setting. Ellipses indicate other settings that would be in the `runtime` node.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <runtime>
    . . .
    <gcTrimCommitOnLowMemory enabled="true"/>
    </runtime>
    . . .
</configuration>
```

See also

- Garbage Collection

Garbage Collection Notifications

28 minutes to read • Edit Online

There are situations in which a full garbage collection (that is, a generation 2 collection) by the common language runtime may adversely affect performance. This can be an issue particularly with servers that process large volumes of requests; in this case, a long garbage collection can cause a request time-out. To prevent a full collection from occurring during a critical period, you can be notified that a full garbage collection is approaching and then take action to redirect the workload to another server instance. You can also induce a collection yourself, provided that the current server instance does not need to process requests.

The RegisterForFullGCNotification method registers for a notification to be raised when the runtime senses that a full garbage collection is approaching. There are two parts to this notification: when the full garbage collection is approaching and when the full garbage collection has completed.

> **WARNING**
> Only blocking garbage collections raise notifications. When the <gcConcurrent> configuration element is enabled, background garbage collections will not raise notifications.

To determine when a notification has been raised, use the WaitForFullGCApproach and WaitForFullGCComplete methods. Typically, you use these methods in a `while` loop to continually obtain a GCNotificationStatus enumeration that shows the status of the notification. If that value is Succeeded, you can do the following:

- In response to a notification obtained with the WaitForFullGCApproach method, you can redirect the workload and possibly induce a collection yourself.
- In response to a notification obtained with the WaitForFullGCComplete method, you can make the current server instance available to process requests again. You can also gather information. For example, you can use the CollectionCount method to record the number of collections.

The WaitForFullGCApproach and the WaitForFullGCComplete methods are designed to work together. Using one without the other can produce indeterminate results.

Full Garbage Collection

The runtime causes a full garbage collection when any of the following scenarios are true:

- Enough memory has been promoted into generation 2 to cause the next generation 2 collection.
- Enough memory has been promoted into the large object heap to cause the next generation 2 collection.
- A collection of generation 1 is escalated to a collection of generation 2 due to other factors.

The thresholds you specify in the RegisterForFullGCNotification method apply to the first two scenarios. However, in the first scenario you will not always receive the notification at the time proportional to the threshold values you specify for two reasons:

- The runtime does not check each small object allocation (for performance reasons).
- Only generation 1 collections promote memory into generation 2.

The third scenario also contributes to the uncertainty of when you will receive the notification. Although this is not a guarantee, it does prove to be a useful way to mitigate the effects of an inopportune full garbage collection by

redirecting the requests during this time or inducing the collection yourself when it can be better accommodated.

Notification Threshold Parameters

The RegisterForFullGCNotification method has two parameters to specify the threshold values of the generation 2 objects and the large object heap. When those values are met, a garbage collection notification should be raised. The following table describes these parameters.

PARAMETER	DESCRIPTION
`maxGenerationThreshold`	A number between 1 and 99 that specifies when the notification should be raised based on the objects promoted in generation 2.
`largeObjectHeapThreshold`	A number between 1 and 99 that specifies when the notification should be raised based on the objects that are allocated in the large object heap.

If you specify a value that is too high, there is a high probability that you will receive a notification, but it could be too long a period to wait before the runtime causes a collection. If you induce a collection yourself, you may reclaim more objects than would be reclaimed if the runtime causes the collection.

If you specify a value that is too low, the runtime may cause the collection before you have had sufficient time to be notified.

Example

Description

In the following example, a group of servers service incoming Web requests. To simulate the workload of processing requests, byte arrays are added to a List<T> collection. Each server registers for a garbage collection notification and then starts a thread on the `WaitForFullGCProc` user method to continuously monitor the GCNotificationStatus enumeration that is returned by the WaitForFullGCApproach and the WaitForFullGCComplete methods.

The WaitForFullGCApproach and the WaitForFullGCComplete methods call their respective event-handling user methods when a notification is raised:

- `OnFullGCApproachNotify`

  This method calls the `RedirectRequests` user method, which instructs the request queuing server to suspend sending requests to the server. This is simulated by setting the class-level variable `bAllocate` to `false` so that no more objects are allocated.

  Next, the `FinishExistingRequests` user method is called to finish processing the pending server requests. This is simulated by clearing the List<T> collection.

  Finally, a garbage collection is induced because the workload is light.

- `OnFullGCCompleteNotify`

  This method calls the user method `AcceptRequests` to resume accepting requests because the server is no longer susceptible to a full garbage collection. This action is simulated by setting the `bAllocate` variable to `true` so that objects can resume being added to the List<T> collection.

The following code contains the `Main` method of the example.

```
using namespace System;
```

```
using namespace System::Collections::Generic;
using namespace System::Threading;

namespace GCNotify
{
    ref class Program
    {
    private:
        // Variable for continual checking in the
        // While loop in the WaitForFullGCProc method.
        static bool checkForNotify = false;

        // Variable for suspending work
        // (such servicing allocated server requests)
        // after a notification is received and then
        // resuming allocation after inducing a garbage collection.
        static bool bAllocate = false;

        // Variable for ending the example.
        static bool finalExit = false;

        // Collection for objects that
        // simulate the server request workload.
        static List<array<Byte>^>^ load = gcnew List<array<Byte>^>();


    public:
        static void Main()
        {
            try
            {
                // Register for a notification.
                GC::RegisterForFullGCNotification(10, 10);
                Console::WriteLine("Registered for GC notification.");

                checkForNotify = true;
                bAllocate = true;

                // Start a thread using WaitForFullGCProc.
                Thread^ thWaitForFullGC = gcnew Thread(gcnew ThreadStart(&WaitForFullGCProc));
                thWaitForFullGC->Start();

                // While the thread is checking for notifications in
                // WaitForFullGCProc, create objects to simulate a server workload.
                try
                {
                    int lastCollCount = 0;
                    int newCollCount = 0;


                    while (true)
                    {
                        if (bAllocate)
                        {
                            load->Add(gcnew array<Byte>(1000));
                            newCollCount = GC::CollectionCount(2);
                            if (newCollCount != lastCollCount)
                            {
                                // Show collection count when it increases:
                                Console::WriteLine("Gen 2 collection count: {0}",
GC::CollectionCount(2).ToString());
                                lastCollCount = newCollCount;
                            }

                            // For ending the example (arbitrary).
                            if (newCollCount == 500)
                            {
                                finalExit = true;
                                checkForNotify = false;
```

```
                            break;
                        }
                    }
                }

            }
            catch (OutOfMemoryException^)
            {
                Console::WriteLine("Out of memory.");
            }


            finalExit = true;
            checkForNotify = false;
            GC::CancelFullGCNotification();

        }
        catch (InvalidOperationException^ invalidOp)
        {

            Console::WriteLine("GC Notifications are not supported while concurrent GC is enabled.\n"
                + invalidOp->Message);
        }
    }

public:
    static void OnFullGCApproachNotify()
    {
        Console::WriteLine("Redirecting requests.");

        // Method that tells the request queuing
        // server to not direct requests to this server.
        RedirectRequests();

        // Method that provides time to
        // finish processing pending requests.
        FinishExistingRequests();

        // This is a good time to induce a GC collection
        // because the runtime will induce a full GC soon.
        // To be very careful, you can check precede with a
        // check of the GC.GCCollectionCount to make sure
        // a full GC did not already occur since last notified.
        GC::Collect();
        Console::WriteLine("Induced a collection.");

    }


public:
    static void OnFullGCCompleteEndNotify()
    {
        // Method that informs the request queuing server
        // that this server is ready to accept requests again.
        AcceptRequests();
        Console::WriteLine("Accepting requests again.");
    }

public:
    static void WaitForFullGCProc()
    {
        while (true)
        {
            // CheckForNotify is set to true and false in Main.
            while (checkForNotify)
            {
                // Check for a notification of an approaching collection.
                GCNotificationStatus s = GC::WaitForFullGCApproach();
                if (s == GCNotificationStatus::Succeeded)
```

```
                if (s == GCNotificationStatus::Succeeded)
                {
                    Console::WriteLine("GC Notifiction raised.");
                    OnFullGCApproachNotify();
                }
                else if (s == GCNotificationStatus::Canceled)
                {
                    Console::WriteLine("GC Notification cancelled.");
                    break;
                }
                else
                {
                    // This can occur if a timeout period
                    // is specified for WaitForFullGCApproach(Timeout)
                    // or WaitForFullGCComplete(Timeout)
                    // and the time out period has elapsed.
                    Console::WriteLine("GC Notification not applicable.");
                    break;
                }

                // Check for a notification of a completed collection.
                s = GC::WaitForFullGCComplete();
                if (s == GCNotificationStatus::Succeeded)
                {
                    Console::WriteLine("GC Notification raised.");
                    OnFullGCCompleteEndNotify();
                }
                else if (s == GCNotificationStatus::Canceled)
                {
                    Console::WriteLine("GC Notification cancelled.");
                    break;
                }
                else
                {
                    // Could be a time out.
                    Console::WriteLine("GC Notification not applicable.");
                    break;
                }
            }


            Thread::Sleep(500);
            // FinalExit is set to true right before
            // the main thread cancelled notification.
            if (finalExit)
            {
                break;
            }
        }
    }

private:
    static void RedirectRequests()
    {
        // Code that sends requests
        // to other servers.

        // Suspend work.
        bAllocate = false;

    }

    static void FinishExistingRequests()
    {
        // Code that waits a period of time
        // for pending requests to finish.

        // Clear the simulated workload.
        load->Clear();
```

```
        }

        static void AcceptRequests()
        {
            // Code that resumes processing
            // requests on this server.

            // Resume work.
            bAllocate = true;
        }
    };
}

int main()
{
    GCNotify::Program::Main();
}
```

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace GCNotify
{
    class Program
    {
        // Variable for continual checking in the
        // While loop in the WaitForFullGCProc method.
        static bool checkForNotify = false;

        // Variable for suspending work
        // (such servicing allocated server requests)
        // after a notification is received and then
        // resuming allocation after inducing a garbage collection.
        static bool bAllocate = false;

        // Variable for ending the example.
        static bool finalExit = false;

        // Collection for objects that
        // simulate the server request workload.
        static List<byte[]> load = new List<byte[]>();

        public static void Main(string[] args)
        {
            try
            {
                // Register for a notification.
                GC.RegisterForFullGCNotification(10, 10);
                Console.WriteLine("Registered for GC notification.");

                checkForNotify = true;
                bAllocate = true;

                // Start a thread using WaitForFullGCProc.
                Thread thWaitForFullGC = new Thread(new ThreadStart(WaitForFullGCProc));
                thWaitForFullGC.Start();

                // While the thread is checking for notifications in
                // WaitForFullGCProc, create objects to simulate a server workload.
                try
                {

                    int lastCollCount = 0;
                    int newCollCount = 0;

                    while (true)
```

```
                    while (true)
                    {
                        if (bAllocate)
                        {
                            load.Add(new byte[1000]);
                            newCollCount = GC.CollectionCount(2);
                            if (newCollCount != lastCollCount)
                            {
                                // Show collection count when it increases:
                                Console.WriteLine("Gen 2 collection count: {0}",
GC.CollectionCount(2).ToString());
                                lastCollCount = newCollCount;
                            }

                            // For ending the example (arbitrary).
                            if (newCollCount == 500)
                            {
                                finalExit = true;
                                checkForNotify = false;
                                break;
                            }
                        }
                    }
                }
                catch (OutOfMemoryException)
                {
                    Console.WriteLine("Out of memory.");
                }

                finalExit = true;
                checkForNotify = false;
                GC.CancelFullGCNotification();
            }
            catch (InvalidOperationException invalidOp)
            {

                Console.WriteLine("GC Notifications are not supported while concurrent GC is enabled.\n"
                    + invalidOp.Message);
            }
        }

        public static void OnFullGCApproachNotify()
        {

            Console.WriteLine("Redirecting requests.");

            // Method that tells the request queuing
            // server to not direct requests to this server.
            RedirectRequests();

            // Method that provides time to
            // finish processing pending requests.
            FinishExistingRequests();

            // This is a good time to induce a GC collection
            // because the runtime will induce a full GC soon.
            // To be very careful, you can check precede with a
            // check of the GC.GCCollectionCount to make sure
            // a full GC did not already occur since last notified.
            GC.Collect();
            Console.WriteLine("Induced a collection.");
        }

        public static void OnFullGCCompleteEndNotify()
        {
            // Method that informs the request queuing server
            // that this server is ready to accept requests again.
            AcceptRequests();
            Console.WriteLine("Accepting requests again.");
        }
```

```
}

public static void WaitForFullGCProc()
{
    while (true)
    {
        // CheckForNotify is set to true and false in Main.
        while (checkForNotify)
        {
            // Check for a notification of an approaching collection.
            GCNotificationStatus s = GC.WaitForFullGCApproach();
            if (s == GCNotificationStatus.Succeeded)
            {
                Console.WriteLine("GC Notification raised.");
                OnFullGCApproachNotify();
            }
            else if (s == GCNotificationStatus.Canceled)
            {
                Console.WriteLine("GC Notification cancelled.");
                break;
            }
            else
            {
                // This can occur if a timeout period
                // is specified for WaitForFullGCApproach(Timeout)
                // or WaitForFullGCComplete(Timeout)
                // and the time out period has elapsed.
                Console.WriteLine("GC Notification not applicable.");
                break;
            }

            // Check for a notification of a completed collection.
            GCNotificationStatus status = GC.WaitForFullGCComplete();
            if (status == GCNotificationStatus.Succeeded)
            {
                Console.WriteLine("GC Notification raised.");
                OnFullGCCompleteEndNotify();
            }
            else if (status == GCNotificationStatus.Canceled)
            {
                Console.WriteLine("GC Notification cancelled.");
                break;
            }
            else
            {
                // Could be a time out.
                Console.WriteLine("GC Notification not applicable.");
                break;
            }
        }

        Thread.Sleep(500);
        // FinalExit is set to true right before
        // the main thread cancelled notification.
        if (finalExit)
        {
            break;
        }
    }
}

private static void RedirectRequests()
{
    // Code that sends requests
    // to other servers.

    // Suspend work.
    bAllocate = false;
}
```

```
        private static void FinishExistingRequests()
        {
            // Code that waits a period of time
            // for pending requests to finish.

            // Clear the simulated workload.
            load.Clear();
        }

        private static void AcceptRequests()
        {
            // Code that resumes processing
            // requests on this server.

            // Resume work.
            bAllocate = true;
        }
    }
}
```

```
Imports System.Collections.Generic
Imports System.Threading

Class Program
    ' Variables for continual checking in the
    ' While loop in the WaitForFullGcProc method.
    Private Shared checkForNotify As Boolean = False

    ' Variable for suspending work
    ' (such as servicing allocated server requests)
    ' after a notification is received and then
    ' resuming allocation after inducing a garbage collection.
    Private Shared bAllocate As Boolean = False

    ' Variable for ending the example.
    Private Shared finalExit As Boolean = False

    ' Collection for objects that
    ' simulate the server request workload.
    Private Shared load As New List(Of Byte())


    Public Shared Sub Main(ByVal args() As String)
        Try
            ' Register for a notification.
            GC.RegisterForFullGCNotification(10, 10)
            Console.WriteLine("Registered for GC notification.")

            bAllocate = True
            checkForNotify = True

            ' Start a thread using WaitForFullGCProc.
            Dim thWaitForFullGC As Thread = _
                New Thread(New ThreadStart(AddressOf WaitForFullGCProc))
            thWaitForFullGC.Start()

            ' While the thread is checking for notifications in
            ' WaitForFullGCProc, create objects to simulate a server workload.
            Try
                Dim lastCollCount As Integer = 0
                Dim newCollCount As Integer = 0


                While (True)
                    If bAllocate = True Then
```

```
                    load.Add(New Byte(1000) {})
                    newCollCount = GC.CollectionCount(2)
                    If (newCollCount <> lastCollCount) Then
                        ' Show collection count when it increases:
                        Console.WriteLine("Gen 2 collection count: {0}", _
                                          GC.CollectionCount(2).ToString)
                        lastCollCount = newCollCount
                    End If

                    ' For ending the example (arbitrary).
                    If newCollCount = 500 Then
                        finalExit = True
                        checkForNotify = False
                        bAllocate = False
                        Exit While
                    End If

                End If
            End While

        Catch outofMem As OutOfMemoryException
            Console.WriteLine("Out of memory.")
        End Try

        finalExit = True
        checkForNotify = False
        GC.CancelFullGCNotification()

    Catch invalidOp As InvalidOperationException
        Console.WriteLine("GC Notifications are not supported while concurrent GC is enabled." _
                          & vbLf & invalidOp.Message)
    End Try
End Sub

Public Shared Sub OnFullGCApproachNotify()
    Console.WriteLine("Redirecting requests.")

    ' Method that tells the request queuing
    ' server to not direct requests to this server.
    RedirectRequests()

    ' Method that provides time to
    ' finish processing pending requests.
    FinishExistingRequests()

    ' This is a good time to induce a GC collection
    ' because the runtime will induce a ful GC soon.
    ' To be very careful, you can check precede with a
    ' check of the GC.GCCollectionCount to make sure
    ' a full GC did not already occur since last notified.
    GC.Collect()
    Console.WriteLine("Induced a collection.")
End Sub

Public Shared Sub OnFullGCCompleteEndNotify()
    ' Method that informs the request queuing server
    ' that this server is ready to accept requests again.
    AcceptRequests()
    Console.WriteLine("Accepting requests again.")
End Sub

Public Shared Sub WaitForFullGCProc()

    While True
        ' CheckForNotify is set to true and false in Main.

        While checkForNotify
            ' Check for a notification of an approaching collection.
            Dim s As GCNotificationStatus = GC.WaitForFullGCApproach
```

```
                If (s = GCNotificationStatus.Succeeded) Then
                    Console.WriteLine("GC Notification raised.")
                    OnFullGCApproachNotify()
                ElseIf (s = GCNotificationStatus.Canceled) Then
                    Console.WriteLine("GC Notification cancelled.")
                    Exit While
                Else
                    ' This can occur if a timeout period
                    ' is specified for WaitForFullGCApproach(Timeout)
                    ' or WaitForFullGCComplete(Timeout)
                    ' and the time out period has elapsed.
                    Console.WriteLine("GC Notification not applicable.")
                    Exit While
                End If

                ' Check for a notification of a completed collection.
                s = GC.WaitForFullGCComplete
                If (s = GCNotificationStatus.Succeeded) Then
                    Console.WriteLine("GC Notifiction raised.")
                    OnFullGCCompleteEndNotify()
                ElseIf (s = GCNotificationStatus.Canceled) Then
                    Console.WriteLine("GC Notification cancelled.")
                    Exit While
                Else
                    ' Could be a time out.
                    Console.WriteLine("GC Notification not applicable.")
                    Exit While
                End If

            End While
            Thread.Sleep(500)
            ' FinalExit is set to true right before
            ' the main thread cancelled notification.
            If finalExit Then
                Exit While
            End If

        End While
    End Sub

    Private Shared Sub RedirectRequests()
        ' Code that sends requests
        ' to other servers.

        ' Suspend work.
        bAllocate = False
    End Sub

    Private Shared Sub FinishExistingRequests()
        ' Code that waits a period of time
        ' for pending requests to finish.

        ' Clear the simulated workload.
        load.Clear()

    End Sub

    Private Shared Sub AcceptRequests()
        ' Code that resumes processing
        ' requests on this server.

        ' Resume work.
        bAllocate = True
    End Sub
End Class
```

The following code contains the `WaitForFullGCProc` user method, that contains a continuous while loop to check

for garbage collection notifications.

```
public:
    static void WaitForFullGCProc()
    {
        while (true)
        {
            // CheckForNotify is set to true and false in Main.
            while (checkForNotify)
            {
                // Check for a notification of an approaching collection.
                GCNotificationStatus s = GC::WaitForFullGCApproach();
                if (s == GCNotificationStatus::Succeeded)
                {
                    Console::WriteLine("GC Notifiction raised.");
                    OnFullGCApproachNotify();
                }
                else if (s == GCNotificationStatus::Canceled)
                {
                    Console::WriteLine("GC Notification cancelled.");
                    break;
                }
                else
                {
                    // This can occur if a timeout period
                    // is specified for WaitForFullGCApproach(Timeout)
                    // or WaitForFullGCComplete(Timeout)
                    // and the time out period has elapsed.
                    Console::WriteLine("GC Notification not applicable.");
                    break;
                }

                // Check for a notification of a completed collection.
                s = GC::WaitForFullGCComplete();
                if (s == GCNotificationStatus::Succeeded)
                {
                    Console::WriteLine("GC Notification raised.");
                    OnFullGCCompleteEndNotify();
                }
                else if (s == GCNotificationStatus::Canceled)
                {
                    Console::WriteLine("GC Notification cancelled.");
                    break;
                }
                else
                {
                    // Could be a time out.
                    Console::WriteLine("GC Notification not applicable.");
                    break;
                }
            }


            Thread::Sleep(500);
            // FinalExit is set to true right before
            // the main thread cancelled notification.
            if (finalExit)
            {
                break;
            }
        }
    }
```

```
public static void WaitForFullGCProc()
{
    while (true)
    {
        // CheckForNotify is set to true and false in Main.
        while (checkForNotify)
        {
            // Check for a notification of an approaching collection.
            GCNotificationStatus s = GC.WaitForFullGCApproach();
            if (s == GCNotificationStatus.Succeeded)
            {
                Console.WriteLine("GC Notification raised.");
                OnFullGCApproachNotify();
            }
            else if (s == GCNotificationStatus.Canceled)
            {
                Console.WriteLine("GC Notification cancelled.");
                break;
            }
            else
            {
                // This can occur if a timeout period
                // is specified for WaitForFullGCApproach(Timeout)
                // or WaitForFullGCComplete(Timeout)
                // and the time out period has elapsed.
                Console.WriteLine("GC Notification not applicable.");
                break;
            }

            // Check for a notification of a completed collection.
            GCNotificationStatus status = GC.WaitForFullGCComplete();
            if (status == GCNotificationStatus.Succeeded)
            {
                Console.WriteLine("GC Notification raised.");
                OnFullGCCompleteEndNotify();
            }
            else if (status == GCNotificationStatus.Canceled)
            {
                Console.WriteLine("GC Notification cancelled.");
                break;
            }
            else
            {
                // Could be a time out.
                Console.WriteLine("GC Notification not applicable.");
                break;
            }
        }

        Thread.Sleep(500);
        // FinalExit is set to true right before
        // the main thread cancelled notification.
        if (finalExit)
        {
            break;
        }
    }
}
```

```
Public Shared Sub WaitForFullGCProc()

    While True
        ' CheckForNotify is set to true and false in Main.

        While checkForNotify
            ' Check for a notification of an approaching collection.
            Dim s As GCNotificationStatus = GC.WaitForFullGCApproach
            If (s = GCNotificationStatus.Succeeded) Then
                Console.WriteLine("GC Notification raised.")
                OnFullGCApproachNotify()
            ElseIf (s = GCNotificationStatus.Canceled) Then
                Console.WriteLine("GC Notification cancelled.")
                Exit While
            Else
                ' This can occur if a timeout period
                ' is specified for WaitForFullGCApproach(Timeout)
                ' or WaitForFullGCComplete(Timeout)
                ' and the time out period has elapsed.
                Console.WriteLine("GC Notification not applicable.")
                Exit While
            End If

            ' Check for a notification of a completed collection.
            s = GC.WaitForFullGCComplete
            If (s = GCNotificationStatus.Succeeded) Then
                Console.WriteLine("GC Notifiction raised.")
                OnFullGCCompleteEndNotify()
            ElseIf (s = GCNotificationStatus.Canceled) Then
                Console.WriteLine("GC Notification cancelled.")
                Exit While
            Else
                ' Could be a time out.
                Console.WriteLine("GC Notification not applicable.")
                Exit While
            End If

        End While
        Thread.Sleep(500)
        ' FinalExit is set to true right before
        ' the main thread cancelled notification.
        If finalExit Then
            Exit While
        End If

    End While
End Sub
```

The following code contains the `OnFullGCApproachNotify` method as called from the `WaitForFullGCProc` method.

```
public:
    static void OnFullGCApproachNotify()
    {
        Console::WriteLine("Redirecting requests.");

        // Method that tells the request queuing
        // server to not direct requests to this server.
        RedirectRequests();

        // Method that provides time to
        // finish processing pending requests.
        FinishExistingRequests();

        // This is a good time to induce a GC collection
        // because the runtime will induce a full GC soon.
        // To be very careful, you can check precede with a
        // check of the GC.GCCollectionCount to make sure
        // a full GC did not already occur since last notified.
        GC::Collect();
        Console::WriteLine("Induced a collection.");

    }
```

```
public static void OnFullGCApproachNotify()
{

    Console.WriteLine("Redirecting requests.");

    // Method that tells the request queuing
    // server to not direct requests to this server.
    RedirectRequests();

    // Method that provides time to
    // finish processing pending requests.
    FinishExistingRequests();

    // This is a good time to induce a GC collection
    // because the runtime will induce a full GC soon.
    // To be very careful, you can check precede with a
    // check of the GC.GCCollectionCount to make sure
    // a full GC did not already occur since last notified.
    GC.Collect();
    Console.WriteLine("Induced a collection.");
}
```

```
Public Shared Sub OnFullGCApproachNotify()
    Console.WriteLine("Redirecting requests.")

    ' Method that tells the request queuing
    ' server to not direct requests to this server.
    RedirectRequests()

    ' Method that provides time to
    ' finish processing pending requests.
    FinishExistingRequests()

    ' This is a good time to induce a GC collection
    ' because the runtime will induce a ful GC soon.
    ' To be very careful, you can check precede with a
    ' check of the GC.GCCollectionCount to make sure
    ' a full GC did not already occur since last notified.
    GC.Collect()
    Console.WriteLine("Induced a collection.")
End Sub
```

The following code contains the `OnFullGCApproachComplete` method as called from the `WaitForFullGCProc` method.

```
public:
    static void OnFullGCCompleteEndNotify()
    {
        // Method that informs the request queuing server
        // that this server is ready to accept requests again.
        AcceptRequests();
        Console::WriteLine("Accepting requests again.");
    }
```

```
public static void OnFullGCCompleteEndNotify()
{
    // Method that informs the request queuing server
    // that this server is ready to accept requests again.
    AcceptRequests();
    Console.WriteLine("Accepting requests again.");
}
```

```
Public Shared Sub OnFullGCCompleteEndNotify()
    ' Method that informs the request queuing server
    ' that this server is ready to accept requests again.
    AcceptRequests()
    Console.WriteLine("Accepting requests again.")
End Sub
```

The following code contains the user methods that are called from the `OnFullGCApproachNotify` and `OnFullGCCompleteNotify` methods. The user methods redirect requests, finish existing requests, and then resume requests after a full garbage collection has occurred.

```
private:
    static void RedirectRequests()
    {
        // Code that sends requests
        // to other servers.

        // Suspend work.
        bAllocate = false;

    }

    static void FinishExistingRequests()
    {
        // Code that waits a period of time
        // for pending requests to finish.

        // Clear the simulated workload.
        load->Clear();

    }

    static void AcceptRequests()
    {
        // Code that resumes processing
        // requests on this server.

        // Resume work.
        bAllocate = true;
    }
```

```
private static void RedirectRequests()
{
    // Code that sends requests
    // to other servers.

    // Suspend work.
    bAllocate = false;
}

private static void FinishExistingRequests()
{
    // Code that waits a period of time
    // for pending requests to finish.

    // Clear the simulated workload.
    load.Clear();
}

private static void AcceptRequests()
{
    // Code that resumes processing
    // requests on this server.

    // Resume work.
    bAllocate = true;
}
```

```
Private Shared Sub RedirectRequests()
    ' Code that sends requests
    ' to other servers.

    ' Suspend work.
    bAllocate = False
End Sub

Private Shared Sub FinishExistingRequests()
    ' Code that waits a period of time
    ' for pending requests to finish.

    ' Clear the simulated workload.
    load.Clear()

End Sub

Private Shared Sub AcceptRequests()
    ' Code that resumes processing
    ' requests on this server.

    ' Resume work.
    bAllocate = True
End Sub
```

The entire code sample is as follows:

```
using namespace System;
using namespace System::Collections::Generic;
using namespace System::Threading;

namespace GCNotify
{
    ref class Program
    {
    private:
        // Variable for continual checking in the
        // While loop in the WaitForFullGCProc method.
        static bool checkForNotify = false;

        // Variable for suspending work
        // (such servicing allocated server requests)
        // after a notification is received and then
        // resuming allocation after inducing a garbage collection.
        static bool bAllocate = false;

        // Variable for ending the example.
        static bool finalExit = false;

        // Collection for objects that
        // simulate the server request workload.
        static List<array<Byte>^>^ load = gcnew List<array<Byte>^>();


    public:
        static void Main()
        {
            try
            {
                // Register for a notification.
                GC::RegisterForFullGCNotification(10, 10);
                Console::WriteLine("Registered for GC notification.");

                checkForNotify = true;
                bAllocate = true;
```

```
                // Start a thread using WaitForFullGCProc.
                Thread^ thWaitForFullGC = gcnew Thread(gcnew ThreadStart(&WaitForFullGCProc));
                thWaitForFullGC->Start();

                // While the thread is checking for notifications in
                // WaitForFullGCProc, create objects to simulate a server workload.
                try
                {
                    int lastCollCount = 0;
                    int newCollCount = 0;


                    while (true)
                    {
                        if (bAllocate)
                        {
                            load->Add(gcnew array<Byte>(1000));
                            newCollCount = GC::CollectionCount(2);
                            if (newCollCount != lastCollCount)
                            {
                                // Show collection count when it increases:
                                Console::WriteLine("Gen 2 collection count: {0}",
GC::CollectionCount(2).ToString());
                                lastCollCount = newCollCount;
                            }

                            // For ending the example (arbitrary).
                            if (newCollCount == 500)
                            {
                                finalExit = true;
                                checkForNotify = false;
                                break;
                            }
                        }
                    }

                }
                catch (OutOfMemoryException^)
                {
                    Console::WriteLine("Out of memory.");
                }


                finalExit = true;
                checkForNotify = false;
                GC::CancelFullGCNotification();

            }
            catch (InvalidOperationException^ invalidOp)
            {

                Console::WriteLine("GC Notifications are not supported while concurrent GC is enabled.\n"
                    + invalidOp->Message);
            }
        }

    public:
        static void OnFullGCApproachNotify()
        {
            Console::WriteLine("Redirecting requests.");

            // Method that tells the request queuing
            // server to not direct requests to this server.
            RedirectRequests();

            // Method that provides time to
            // finish processing pending requests.
            FinishExistingRequests();
```

```
            // This is a good time to induce a GC collection
            // because the runtime will induce a full GC soon.
            // To be very careful, you can check precede with a
            // check of the GC.GCCollectionCount to make sure
            // a full GC did not already occur since last notified.
            GC::Collect();
            Console::WriteLine("Induced a collection.");

        }


    public:
        static void OnFullGCCompleteEndNotify()
        {
            // Method that informs the request queuing server
            // that this server is ready to accept requests again.
            AcceptRequests();
            Console::WriteLine("Accepting requests again.");
        }

    public:
        static void WaitForFullGCProc()
        {
            while (true)
            {
                // CheckForNotify is set to true and false in Main.
                while (checkForNotify)
                {
                    // Check for a notification of an approaching collection.
                    GCNotificationStatus s = GC::WaitForFullGCApproach();
                    if (s == GCNotificationStatus::Succeeded)
                    {
                        Console::WriteLine("GC Notifiction raised.");
                        OnFullGCApproachNotify();
                    }
                    else if (s == GCNotificationStatus::Canceled)
                    {
                        Console::WriteLine("GC Notification cancelled.");
                        break;
                    }
                    else
                    {
                        // This can occur if a timeout period
                        // is specified for WaitForFullGCApproach(Timeout)
                        // or WaitForFullGCComplete(Timeout)
                        // and the time out period has elapsed.
                        Console::WriteLine("GC Notification not applicable.");
                        break;
                    }

                    // Check for a notification of a completed collection.
                    s = GC::WaitForFullGCComplete();
                    if (s == GCNotificationStatus::Succeeded)
                    {
                        Console::WriteLine("GC Notification raised.");
                        OnFullGCCompleteEndNotify();
                    }
                    else if (s == GCNotificationStatus::Canceled)
                    {
                        Console::WriteLine("GC Notification cancelled.");
                        break;
                    }
                    else
                    {
                        // Could be a time out.
                        Console::WriteLine("GC Notification not applicable.");
                        break;
                    }
                }
```

```
                }


                Thread::Sleep(500);
                // FinalExit is set to true right before
                // the main thread cancelled notification.
                if (finalExit)
                {
                    break;
                }
            }
        }

    private:
        static void RedirectRequests()
        {
            // Code that sends requests
            // to other servers.

            // Suspend work.
            bAllocate = false;

        }

        static void FinishExistingRequests()
        {
            // Code that waits a period of time
            // for pending requests to finish.

            // Clear the simulated workload.
            load->Clear();

        }

        static void AcceptRequests()
        {
            // Code that resumes processing
            // requests on this server.

            // Resume work.
            bAllocate = true;
        }
    };
}

int main()
{
    GCNotify::Program::Main();
}
```

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace GCNotify
{
    class Program
    {
        // Variable for continual checking in the
        // While loop in the WaitForFullGCProc method.
        static bool checkForNotify = false;

        // Variable for suspending work
        // (such servicing allocated server requests)
        // after a notification is received and then
        // resuming allocation after inducing a garbage collection.
        static bool bAllocate = false;
```

```
        // Variable for ending the example.
        static bool finalExit = false;

        // Collection for objects that
        // simulate the server request workload.
        static List<byte[]> load = new List<byte[]>();

        public static void Main(string[] args)
        {
            try
            {
                // Register for a notification.
                GC.RegisterForFullGCNotification(10, 10);
                Console.WriteLine("Registered for GC notification.");

                checkForNotify = true;
                bAllocate = true;

                // Start a thread using WaitForFullGCProc.
                Thread thWaitForFullGC = new Thread(new ThreadStart(WaitForFullGCProc));
                thWaitForFullGC.Start();

                // While the thread is checking for notifications in
                // WaitForFullGCProc, create objects to simulate a server workload.
                try
                {

                    int lastCollCount = 0;
                    int newCollCount = 0;

                    while (true)
                    {
                        if (bAllocate)
                        {
                            load.Add(new byte[1000]);
                            newCollCount = GC.CollectionCount(2);
                            if (newCollCount != lastCollCount)
                            {
                                // Show collection count when it increases:
                                Console.WriteLine("Gen 2 collection count: {0}",
GC.CollectionCount(2).ToString());
                                lastCollCount = newCollCount;
                            }

                            // For ending the example (arbitrary).
                            if (newCollCount == 500)
                            {
                                finalExit = true;
                                checkForNotify = false;
                                break;
                            }
                        }
                    }
                }
                catch (OutOfMemoryException)
                {
                    Console.WriteLine("Out of memory.");
                }

                finalExit = true;
                checkForNotify = false;
                GC.CancelFullGCNotification();
            }
            catch (InvalidOperationException invalidOp)
            {

                Console.WriteLine("GC Notifications are not supported while concurrent GC is enabled.\n"
                    + invalidOp.Message);
            }
```

```
    }

    public static void OnFullGCApproachNotify()
    {

        Console.WriteLine("Redirecting requests.");

        // Method that tells the request queuing
        // server to not direct requests to this server.
        RedirectRequests();

        // Method that provides time to
        // finish processing pending requests.
        FinishExistingRequests();

        // This is a good time to induce a GC collection
        // because the runtime will induce a full GC soon.
        // To be very careful, you can check precede with a
        // check of the GC.GCCollectionCount to make sure
        // a full GC did not already occur since last notified.
        GC.Collect();
        Console.WriteLine("Induced a collection.");
    }

    public static void OnFullGCCompleteEndNotify()
    {
        // Method that informs the request queuing server
        // that this server is ready to accept requests again.
        AcceptRequests();
        Console.WriteLine("Accepting requests again.");
    }

    public static void WaitForFullGCProc()
    {
        while (true)
        {
            // CheckForNotify is set to true and false in Main.
            while (checkForNotify)
            {
                // Check for a notification of an approaching collection.
                GCNotificationStatus s = GC.WaitForFullGCApproach();
                if (s == GCNotificationStatus.Succeeded)
                {
                    Console.WriteLine("GC Notification raised.");
                    OnFullGCApproachNotify();
                }
                else if (s == GCNotificationStatus.Canceled)
                {
                    Console.WriteLine("GC Notification cancelled.");
                    break;
                }
                else
                {
                    // This can occur if a timeout period
                    // is specified for WaitForFullGCApproach(Timeout)
                    // or WaitForFullGCComplete(Timeout)
                    // and the time out period has elapsed.
                    Console.WriteLine("GC Notification not applicable.");
                    break;
                }

                // Check for a notification of a completed collection.
                GCNotificationStatus status = GC.WaitForFullGCComplete();
                if (status == GCNotificationStatus.Succeeded)
                {
                    Console.WriteLine("GC Notification raised.");
                    OnFullGCCompleteEndNotify();
                }
                else if (status == GCNotificationStatus.Canceled)
```

```
                    else if (status == GCNotificationStatus.Canceled)
                    {
                        Console.WriteLine("GC Notification cancelled.");
                        break;
                    }
                    else
                    {
                        // Could be a time out.
                        Console.WriteLine("GC Notification not applicable.");
                        break;
                    }
                }

                Thread.Sleep(500);
                // FinalExit is set to true right before
                // the main thread cancelled notification.
                if (finalExit)
                {
                    break;
                }
            }
        }

        private static void RedirectRequests()
        {
            // Code that sends requests
            // to other servers.

            // Suspend work.
            bAllocate = false;
        }

        private static void FinishExistingRequests()
        {
            // Code that waits a period of time
            // for pending requests to finish.

            // Clear the simulated workload.
            load.Clear();
        }

        private static void AcceptRequests()
        {
            // Code that resumes processing
            // requests on this server.

            // Resume work.
            bAllocate = true;
        }
    }
}
```

```
Imports System.Collections.Generic
Imports System.Threading

Class Program
    ' Variables for continual checking in the
    ' While loop in the WaitForFullGcProc method.
    Private Shared checkForNotify As Boolean = False

    ' Variable for suspending work
    ' (such as servicing allocated server requests)
    ' after a notification is received and then
    ' resuming allocation after inducing a garbage collection.
    Private Shared bAllocate As Boolean = False

    ' Variable for ending the example.
    Private Shared finalExit As Boolean = False
```

```
Private Shared finalExit As Boolean = False

' Collection for objects that
' simulate the server request workload.
Private Shared load As New List(Of Byte())


Public Shared Sub Main(ByVal args() As String)
    Try
        ' Register for a notification.
        GC.RegisterForFullGCNotification(10, 10)
        Console.WriteLine("Registered for GC notification.")

        bAllocate = True
        checkForNotify = True

        ' Start a thread using WaitForFullGCProc.
        Dim thWaitForFullGC As Thread = _
            New Thread(New ThreadStart(AddressOf WaitForFullGCProc))
        thWaitForFullGC.Start()

        ' While the thread is checking for notifications in
        ' WaitForFullGCProc, create objects to simulate a server workload.
        Try
            Dim lastCollCount As Integer = 0
            Dim newCollCount As Integer = 0


            While (True)
                If bAllocate = True Then

                    load.Add(New Byte(1000) {})
                    newCollCount = GC.CollectionCount(2)
                    If (newCollCount <> lastCollCount) Then
                        ' Show collection count when it increases:
                        Console.WriteLine("Gen 2 collection count: {0}", _
                                          GC.CollectionCount(2).ToString)
                        lastCollCount = newCollCount
                    End If

                    ' For ending the example (arbitrary).
                    If newCollCount = 500 Then
                        finalExit = True
                        checkForNotify = False
                        bAllocate = False
                        Exit While
                    End If

                End If
            End While

        Catch outofMem As OutOfMemoryException
            Console.WriteLine("Out of memory.")
        End Try

        finalExit = True
        checkForNotify = False
        GC.CancelFullGCNotification()

    Catch invalidOp As InvalidOperationException
        Console.WriteLine("GC Notifications are not supported while concurrent GC is enabled." _
                          & vbLf & invalidOp.Message)
    End Try
End Sub

Public Shared Sub OnFullGCApproachNotify()
    Console.WriteLine("Redirecting requests.")

    ' Method that tells the request queuing
    ' server to not direct requests to this server
```

```
        ' server to not direct requests to this server.
        RedirectRequests()

        ' Method that provides time to
        ' finish processing pending requests.
        FinishExistingRequests()

        ' This is a good time to induce a GC collection
        ' because the runtime will induce a ful GC soon.
        ' To be very careful, you can check precede with a
        ' check of the GC.GCCollectionCount to make sure
        ' a full GC did not already occur since last notified.
        GC.Collect()
        Console.WriteLine("Induced a collection.")
    End Sub

    Public Shared Sub OnFullGCCompleteEndNotify()
        ' Method that informs the request queuing server
        ' that this server is ready to accept requests again.
        AcceptRequests()
        Console.WriteLine("Accepting requests again.")
    End Sub

    Public Shared Sub WaitForFullGCProc()

        While True
            ' CheckForNotify is set to true and false in Main.

            While checkForNotify
                ' Check for a notification of an approaching collection.
                Dim s As GCNotificationStatus = GC.WaitForFullGCApproach
                If (s = GCNotificationStatus.Succeeded) Then
                    Console.WriteLine("GC Notification raised.")
                    OnFullGCApproachNotify()
                ElseIf (s = GCNotificationStatus.Canceled) Then
                    Console.WriteLine("GC Notification cancelled.")
                    Exit While
                Else
                    ' This can occur if a timeout period
                    ' is specified for WaitForFullGCApproach(Timeout)
                    ' or WaitForFullGCComplete(Timeout)
                    ' and the time out period has elapsed.
                    Console.WriteLine("GC Notification not applicable.")
                    Exit While
                End If

                ' Check for a notification of a completed collection.
                s = GC.WaitForFullGCComplete
                If (s = GCNotificationStatus.Succeeded) Then
                    Console.WriteLine("GC Notifiction raised.")
                    OnFullGCCompleteEndNotify()
                ElseIf (s = GCNotificationStatus.Canceled) Then
                    Console.WriteLine("GC Notification cancelled.")
                    Exit While
                Else
                    ' Could be a time out.
                    Console.WriteLine("GC Notification not applicable.")
                    Exit While
                End If

            End While
            Thread.Sleep(500)
            ' FinalExit is set to true right before
            ' the main thread cancelled notification.
            If finalExit Then
                Exit While
            End If

        End While
```

```
    End Sub

    Private Shared Sub RedirectRequests()
        ' Code that sends requests
        ' to other servers.

        ' Suspend work.
        bAllocate = False
    End Sub

    Private Shared Sub FinishExistingRequests()
        ' Code that waits a period of time
        ' for pending requests to finish.

        ' Clear the simulated workload.
        load.Clear()

    End Sub

    Private Shared Sub AcceptRequests()
        ' Code that resumes processing
        ' requests on this server.

        ' Resume work.
        bAllocate = True
    End Sub
End Class
```

See also

- Garbage Collection

Application Domain Resource Monitoring

4 minutes to read • Edit Online

Application domain resource monitoring (ARM) enables hosts to monitor CPU and memory usage by application domain. This is useful for hosts such as ASP.NET that use many application domains in a long-running process. The host can unload the application domain of an application that is adversely affecting the performance of the entire process, but only if it can identify the problematic application. ARM provides information that can be used to assist in making such decisions.

For example, a hosting service might have many applications running on an ASP.NET server. If one application in the process begins consuming too much memory or too much processor time, the hosting service can use ARM to identify the application domain that is causing the problem.

ARM is sufficiently lightweight to use in live applications. You can access the information by using event tracing for Windows (ETW) or directly through managed or native APIs.

Enabling Resource Monitoring

ARM can be enabled in four ways: by supplying a configuration file when the common language runtime (CLR) is started, by using an unmanaged hosting API, by using managed code, or by listening to ARM ETW events.

As soon as ARM is enabled, it begins collecting data on all application domains in the process.If an application domain was created before ARM is enabled, cumulative data starts when ARM is enabled, not when the application domain was created.Once it is enabled, ARM cannot be disabled.

- You can enable ARM at CLR startup by adding the <appDomainResourceMonitoring> element to the configuration file, and setting the `enabled` attribute to `true`. A value of `false` (the default) means only that ARM is not enabled at startup; you can activate it later by using one of the other activation mechanisms.
- The host can enable ARM by requesting the ICLRAppDomainResourceMonitor hosting interface. Once this interface is successfully obtained, ARM is enabled.
- Managed code can enable ARM by setting the static (`Shared` in Visual Basic) AppDomain.MonitoringIsEnabled property to `true`. As soon as the property is set, ARM is enabled.
- You can enable ARM after startup by listening to ETW events. ARM is enabled and begins raising events for all application domains when you enable the public provider `Microsoft-Windows-DotNETRuntime` by using the `AppDomainResourceManagementKeyword` keyword. To correlate data with application domains and threads, you must also enable the `Microsoft-Windows-DotNETRuntimeRundown` provider with the `ThreadingKeyword` keyword.

Using ARM

ARM provides the total processor time that is used by an application domain and three kinds of information about memory use.

- **Total processor time for an application domain, in seconds**: This is calculated by adding up the thread times reported by the operating system for all threads that spent time executing in the application domain during its lifetime. Blocked or sleeping threads do not use processor time. When a thread calls into native code, the time that the thread spends in native code is included in the count for the application domain where the call was made.

- Managed API: AppDomain.MonitoringTotalProcessorTime property.
- Hosting API: ICLRAppDomainResourceMonitor::GetCurrentCpuTime method.
- ETW events: `ThreadCreated`, `ThreadAppDomainEnter`, and `ThreadTerminated` events. For information about providers and keywords, see "AppDomain Resource Monitoring Events" in CLR ETW Events.

- **Total managed allocations made by an application domain during its lifetime, in bytes**: Total allocations do not always reflect memory use by an application domain, because the allocated objects might be short-lived. However, if an application allocates and frees huge numbers of objects, the cost of the allocations could be significant.
  - Managed API: AppDomain.MonitoringTotalAllocatedMemorySize property.
  - Hosting API: ICLRAppDomainResourceMonitor::GetCurrentAllocated method.
  - ETW events: `AppDomainMemAllocated` event, `Allocated` field.
- **Managed memory, in bytes, that is referenced by an application domain and that survived the most recent full, blocking collection**: This number is accurate only after a full, blocking collection. (This is in contrast to concurrent collections, which occur in the background and do not block the application.) For example, the GC.Collect() method overload causes a full, blocking collection.
  - Managed API: AppDomain.MonitoringSurvivedMemorySize property.
  - Hosting API: ICLRAppDomainResourceMonitor::GetCurrentSurvived method, `pAppDomainBytesSurvived` parameter.
  - ETW events: `AppDomainMemSurvived` event, `Survived` field.
- **Total managed memory, in bytes, that is referenced by the process and that survived the most recent full, blocking collection**: The survived memory for individual application domains can be compared to this number.
  - Managed API: AppDomain.MonitoringSurvivedProcessMemorySize property.
  - Hosting API: ICLRAppDomainResourceMonitor::GetCurrentSurvived method, `pTotalBytesSurvived` parameter.
  - ETW events: `AppDomainMemSurvived` event, `ProcessSurvived` field.

Determining When a Full, Blocking Collection Occurs

To determine when counts of survived memory are accurate, you need to know when a full, blocking collection has just occurred. The method for doing this depends on the API you use to examine ARM statistics.

Managed API

If you use the properties of the AppDomain class, you can use the GC.RegisterForFullGCNotification method to register for notification of full collections. The threshold you use is not important, because you are waiting for the completion of a collection rather than the approach of a collection. You can then call the GC.WaitForFullGCComplete method, which blocks until a full collection has completed. You can create a thread that calls the method in a loop and does any necessary analysis whenever the method returns.

Alternatively, you can call the GC.CollectionCount method periodically to see if the count of generation 2 collections has increased. Depending on the polling frequency, this technique might not provide as accurate an indication of the occurrence of a full collection.

Hosting API

If you use the unmanaged hosting API, your host must pass the CLR an implementation of the IHostGCManager interface. The CLR calls the IHostGCManager::SuspensionEnding method when it resumes execution of threads that have been suspended while a collection occurs. The CLR passes the generation of the completed collection as

a parameter of the method, so the host can determine whether the collection was full or partial. Your implementation of the IHostGCManager::SuspensionEnding method can query for survived memory, to ensure that the counts are retrieved as soon as they are updated.

See also

- AppDomain.MonitoringIsEnabled
- ICLRAppDomainResourceMonitor Interface
- <appDomainResourceMonitoring>
- CLR ETW Events

minutes to read • Edit Online

The garbage collector cannot collect an object in use by an application while the application's code can reach that object. The application is said to have a strong reference to the object.

A weak reference permits the garbage collector to collect the object while still allowing the application to access the object. A weak reference is valid only during the indeterminate amount of time until the object is collected when no strong references exist. When you use a weak reference, the application can still obtain a strong reference to the object, which prevents it from being collected. However, there is always the risk that the garbage collector will get to the object first before a strong reference is reestablished.

Weak references are useful for objects that use a lot of memory, but can be recreated easily if they are reclaimed by garbage collection.

Suppose a tree view in a Windows Forms application displays a complex hierarchical choice of options to the user. If the underlying data is large, keeping the tree in memory is inefficient when the user is involved with something else in the application.

When the user switches away to another part of the application, you can use the WeakReference class to create a weak reference to the tree and destroy all strong references. When the user switches back to the tree, the application attempts to obtain a strong reference to the tree and, if successful, avoids reconstructing the tree.

To establish a weak reference with an object, you create a WeakReference using the instance of the object to be tracked. You then set the Target property to that object and set the original reference to the object to `null`. For a code example, see WeakReference in the class library.

Short and Long Weak References

You can create a short weak reference or a long weak reference:

- Short

  The target of a short weak reference becomes `null` when the object is reclaimed by garbage collection. The weak reference is itself a managed object, and is subject to garbage collection just like any other managed object. A short weak reference is the parameterless constructor for WeakReference.

- Long

  A long weak reference is retained after the object's Finalize method has been called. This allows the object to be recreated, but the state of the object remains unpredictable. To use a long reference, specify `true` in the WeakReference constructor.

  If the object's type does not have a Finalize method, the short weak reference functionality applies and the weak reference is valid only until the target is collected, which can occur anytime after the finalizer is run.

To establish a strong reference and use the object again, cast the Target property of a WeakReference to the type of the object. If the Target property returns `null`, the object was collected; otherwise, you can continue to use the object because the application has regained a strong reference to it.

Guidelines for Using Weak References

Use long weak references only when necessary as the state of the object is unpredictable after finalization.

Avoid using weak references to small objects because the pointer itself may be as large or larger.

Avoid using weak references as an automatic solution to memory management problems. Instead, develop an effective caching policy for handling your application's objects.

See also

- Garbage Collection