HW5 2150276 沈卓成

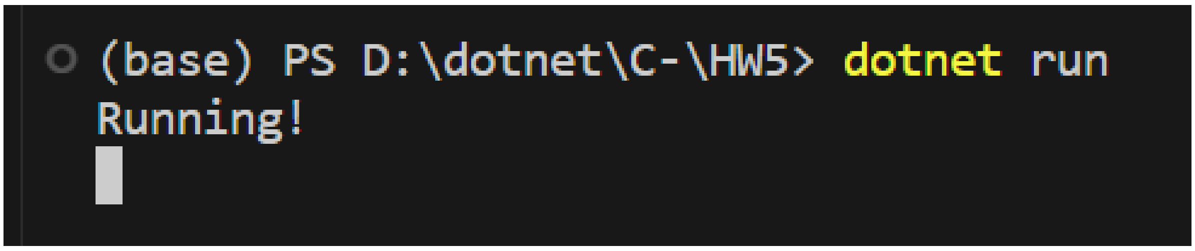
Part 1 课堂代码运行

1 Mutex

Code：

```
using System;
using System.Threading;
public class Program
{
    static void Main(string[] args)
    {
        const string MutexName = "CSharpThreadingMutex";
        var m = new Mutex(false, MutexName);
        try
        {
            if (!m.WaitOne(TimeSpan.FromSeconds(5)))
            {
                System.Console.WriteLine("Second instance is running!");
            }
            else
            {
                System.Console.WriteLine("Running!");
                System.Console.ReadLine();
                m.ReleaseMutex();
            }
        }
        finally
        {
            m.Dispose();
        }
    }
}
```

Ans:

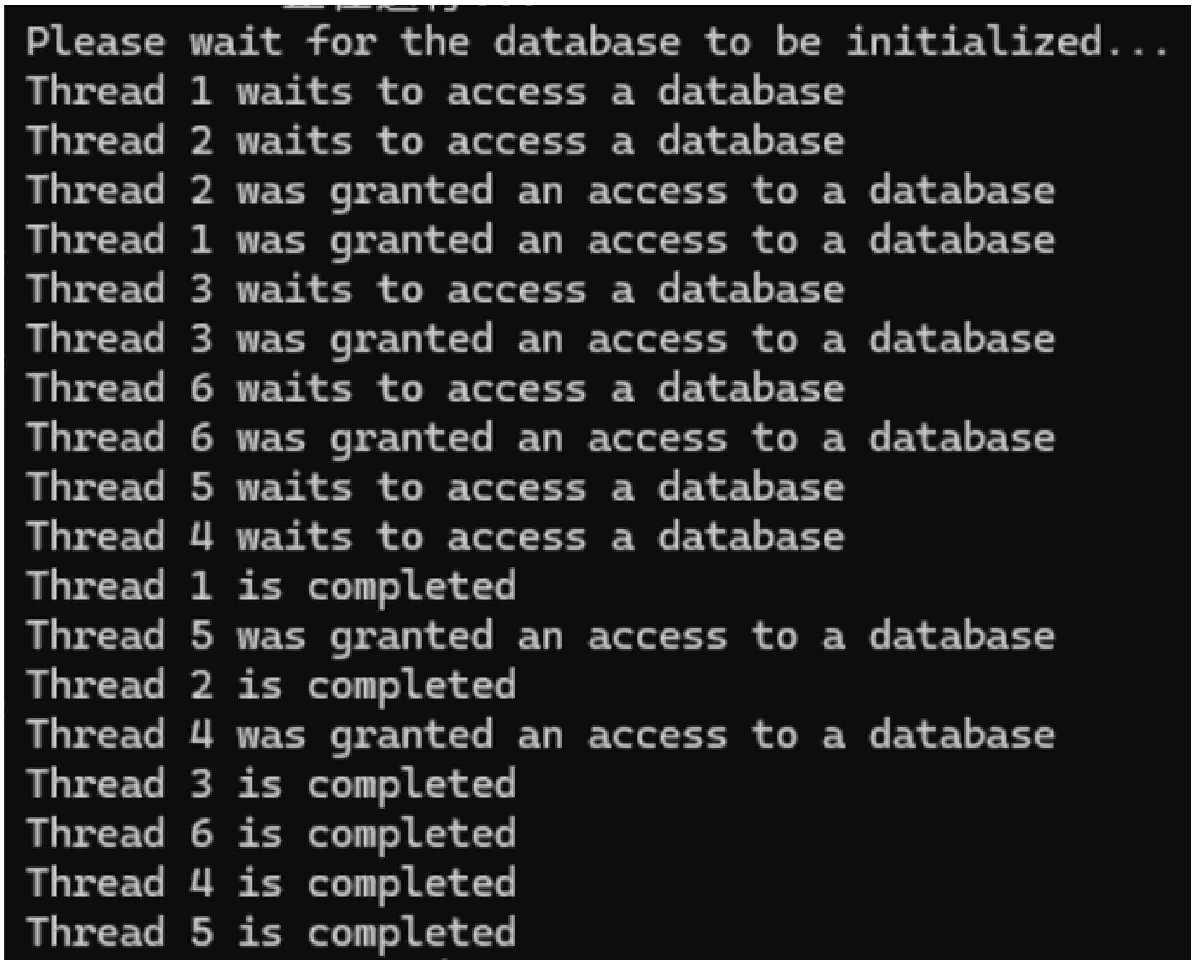


```
(base) PS D:\dotnet\C-\HW5> dotnet run
Running!
```

2 信号量

```
using System;
using System.Threading;
public class Program

{
    static SemaphoreSlim _semaphore = new SemaphoreSlim(4); // 初始化信号量为4
    static void AccessDatabase(string name, int seconds)
    {
        System.Console.WriteLine("{0} waits to access a database", name);
        _semaphore.Wait(); // 信号量减1, 等待访问数据库
        System.Console.WriteLine("{0} was granted an access to a database",
name);
        Thread.Sleep(TimeSpan.FromSeconds(seconds));
        System.Console.WriteLine("{0} is completed", name);
        _semaphore.Release(); // 信号量加1
    }
    public static void Main(string[] args)
    {
        WaitHandle[] arr = new WaitHandle[] { new AutoResetEvent(true), new
AutoResetEvent(false) };
        // 第一个自动重置事件已经触发，第二个自动重置事件未触发
        System.Console.WriteLine("Please wait for the database to be
initialized...");
        WaitHandle.WaitAll(arr); // 阻塞等待所有自动重置事件出发
        for (int i = 1; i <= 6; i++)
        {
            string threadName = "Thread " + i;
            int secondsToWait = 2 + 2 * i;
            var t = new Thread(() => AccessDatabase(threadName, secondsToWait));
            t.Start();
        }
    }
}
```

Ans:



```
Please wait for the database to be initialized...
Thread 1 waits to access a database
Thread 2 waits to access a database
Thread 2 was granted an access to a database
Thread 1 was granted an access to a database
Thread 3 waits to access a database
Thread 3 was granted an access to a database
Thread 6 waits to access a database
Thread 6 was granted an access to a database
Thread 5 waits to access a database
Thread 4 waits to access a database
Thread 1 is completed
Thread 5 was granted an access to a database
Thread 2 is completed
Thread 4 was granted an access to a database
Thread 3 is completed
Thread 6 is completed
Thread 4 is completed
Thread 5 is completed
```

1. `SemaphoreSlim`类被实例化为`_semaphore`，并初始化为4，表示最多允许4个线程同时访问一个资源（数据库）。
2. `AccessDatabase`方法模拟数据库访问。一个名为`name`的线程试图访问数据库，并在访问之前输出消息。通过`_semaphore.Wait()`调用，该线程请求进入信号量，如果成功（即信号量的计数未降至0），它将继续执行；否则，它将等待，直到信号量被先前的线程释放。一旦获得访问权限，它等待由变量`seconds`指定的一段时间（模拟访问数据库所花费的时间），然后输出完成的消息并通过`_semaphore.Release()`释放信号量。
3. 在`Main`方法中，首先创建了两个`AutoResetEvent`对象并把它们添加到`arr`数组中。这些对象通常用于线程同步。
    - `AutoResetEvent(true)`：已经触发状态，不会阻塞通过`WaitHandle.WaitAll(arr)`的调用。
    - `AutoResetEvent(false)`：非触发状态，将导致`WaitHandle.WaitAll(arr)`调用阻塞，直到它被设置为触发状态，但这段代码中未设置为触发，因此这会导致永久阻塞。
4. `WaitHandle.WaitAll(arr);`：这一行代码是用来等待所有的`WaitHandle`对象变为触发状态，但在这段代码中，第二个`AutoResetEvent`永不触发，会导致程序在这里永远等待。
5. 接下来，`Main`方法会启动6个线程，每个都执行`AccessDatabase`方法。每个线程都有一个唯一的名称和不同的等待秒数，通过`new Thread(...).Start();`启动。

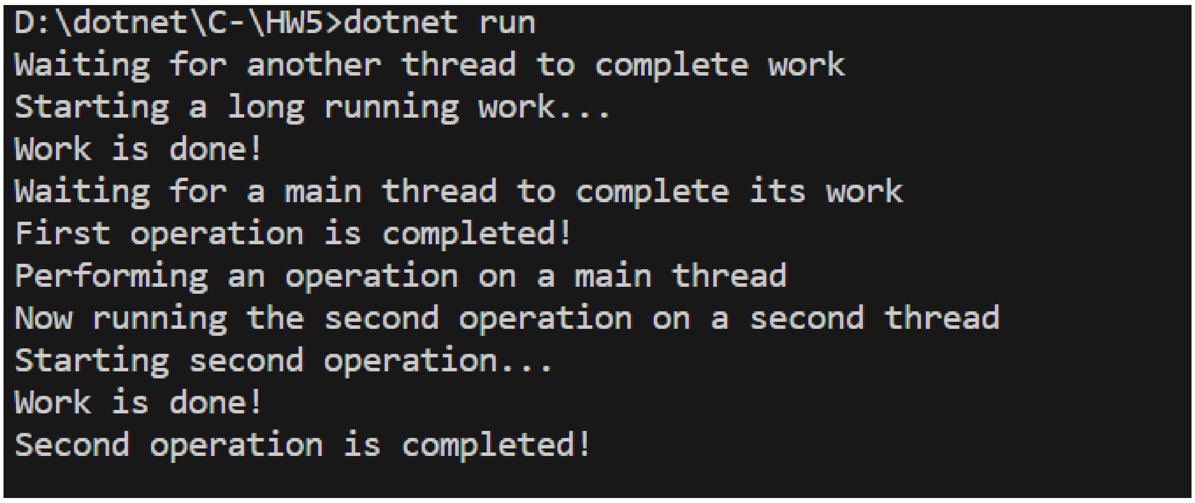
3 AutoResetEvent

Code:

```
using static System.Console;
using static System.Threading.Thread;
class Program
{
    private static AutoResetEvent _workerEvent = new AutoResetEvent(false);
    private static AutoResetEvent _mainEvent   = new AutoResetEvent(false);

    static void Process(int seconds)
    {
        WriteLine("Starting a long running work...");
        Sleep(TimeSpan.FromSeconds(seconds));
        WriteLine("Work is done!");
        _workerEvent.Set();
        WriteLine("Waiting for a main thread to complete its work");
        _mainEvent.WaitOne();
        WriteLine("Starting second operation...");
        Sleep(TimeSpan.FromSeconds(seconds));
        WriteLine("Work is done!");
        _workerEvent.Set();
    }

    static void Main(string[] args)
    {
        var t = new Thread(() => Process(10));
        t.Start();

        WriteLine("Waiting for another thread to complete work");
        _workerEvent.WaitOne();
        WriteLine("First operation is completed!");
        WriteLine("Performing an operation on a main thread");
        Sleep(TimeSpan.FromSeconds(5));
```

```
            _mainEvent.Set();
            WriteLine("Now running the second operation on a second thread");
            _workerEvent.WaitOne();
            WriteLine("Second operation is completed!");
        }
    }
```

运行结果:



```
D:\dotnet\C-\HW5>dotnet run
Waiting for another thread to complete work
Starting a long running work...
Work is done!
Waiting for a main thread to complete its work
First operation is completed!
Performing an operation on a main thread
Now running the second operation on a second thread
Starting second operation...
Work is done!
Second operation is completed!
```

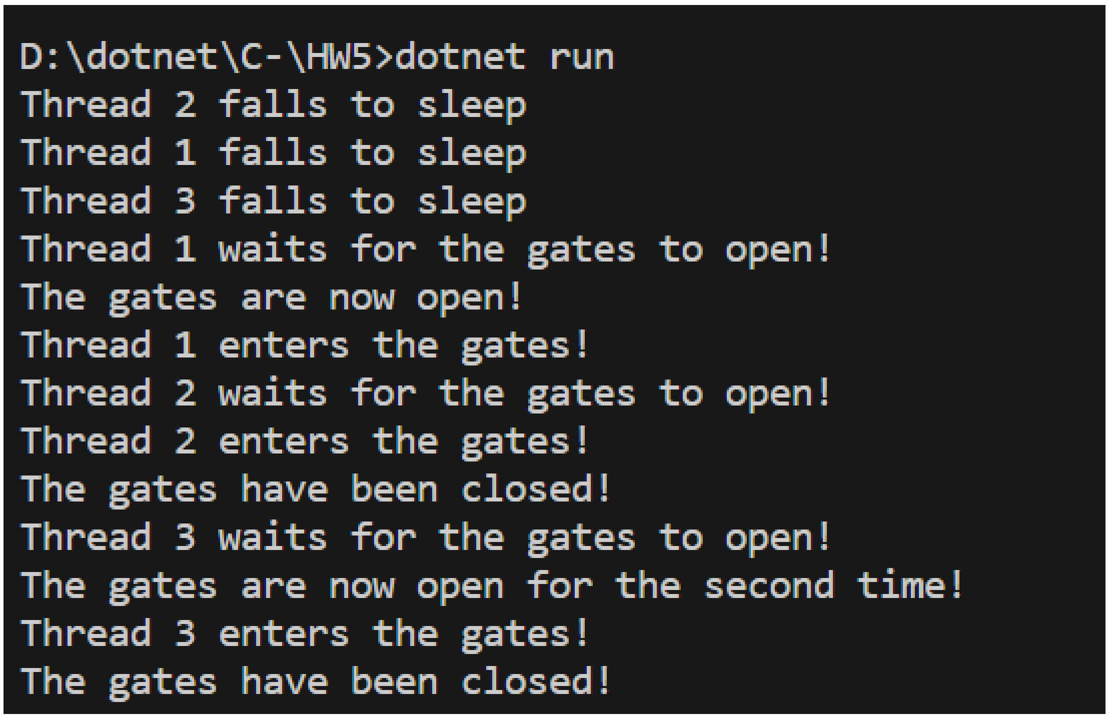
1. 引入`System.Console`和`System.Threading.Thread`命名空间，这样可以直接使用`WriteLine`和`Sleep`方法，而不需要完整的命名空间前缀。
2. 定义了两个`AutoResetEvent`对象：`_workerEvent`和`_mainEvent`。`AutoResetEvent`是一个同步原语，可以在信号状态和非信号状态之间自动切换。
3. `Process`方法模拟了一个需要较长时间执行的操作。该方法执行以下步骤：
    - 输出开始工作的消息。
    - 休眠指定秒数（模拟长时间运行的工作）。
    - 输出工作完成的消息，并通过`_workerEvent.Set()`方法发送信号，告知主线程工作完成。
    - 然后，输出等待主线程完成它的工作的消息，并通过`_mainEvent.WaitOne()`方法等待主线程发出的信号。
    - 主线程信号到来后，输出开始第二项操作的消息，并再次休眠相同的秒数来模拟第二项操作的运行。
    - 第二项操作完成后，再次通过`_workerEvent.Set()`方法告知主线程操作已完成。
4. `Main`方法启动工作线程，该工作线程执行`Process`方法，传递了10秒作为参数。
5. 主线程会执行以下步骤：
    - 输出等待工作线程完成工作的消息，并通过调用`_workerEvent.WaitOne()`等待工作线程的第一次工作完成。
    - 工作线程完成第一项工作后，输出表示第一项操作完成的消息。
    - 接下来，主线程模拟进行自己的操作，休眠5秒。
    - 完成后，通过`_mainEvent.Set()`方法向工作线程发出信号，表明它可以继续执行第二项操作。
    - 主线程再次等待工作线程完成第二次操作，通过调用`_workerEvent.WaitOne()`。
    - 当工作线程完成第二项操作并通过调用`_workerEvent.Set()`告知主线程时，输出第二项操作完成的消息。

4 ManualResetEventSlim

Code:

```
using static System.Console;
using static System.Threading.Thread;

class Program
{
    static ManualResetEventSlim _mainEvent = new ManualResetEventSlim(false);
    static void TravelThroughGates(string threadName, int seconds)
    {
        WriteLine($"{threadName} falls to sleep");
        Sleep(TimeSpan.FromSeconds(seconds));
        WriteLine($"{threadName} waits for the gates to open!");
        _mainEvent.Wait();
        WriteLine($"{threadName} enters the gates!");
    }
    static void Main(string[] args)
    {
        var t1 = new Thread(() => TravelThroughGates("Thread 1", 5));
        var t2 = new Thread(() => TravelThroughGates("Thread 2", 6));
        var t3 = new Thread(() => TravelThroughGates("Thread 3", 12));
        //var t3 = new Thread(() => TravelThroughGates("Thread 3", 6));
        t1.Start();
        t2.Start();
        t3.Start();
        Sleep(TimeSpan.FromSeconds(6));
        WriteLine("The gates are now open!");
        _mainEvent.Set();
        Sleep(TimeSpan.FromSeconds(2));
        _mainEvent.Reset();
        WriteLine("The gates have been closed!");
        Sleep(TimeSpan.FromSeconds(10));
        WriteLine("The gates are now open for the second time!");
        _mainEvent.Set();
        Sleep(TimeSpan.FromSeconds(2));
        WriteLine("The gates have been closed!");
        _mainEvent.Reset();
    }
}
```

运行结果:



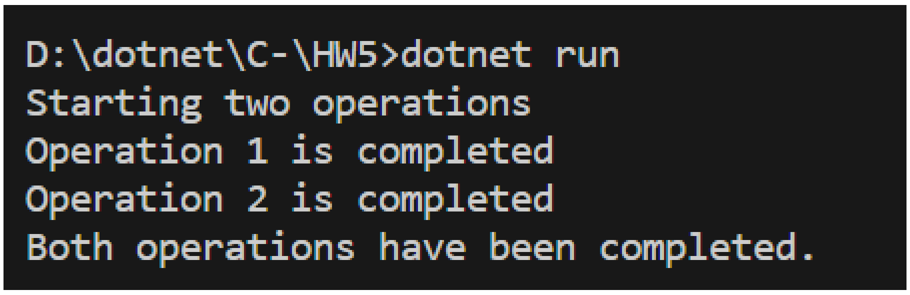
```
D:\dotnet\C-\HW5>dotnet run
Thread 2 falls to sleep
Thread 1 falls to sleep
Thread 3 falls to sleep
Thread 1 waits for the gates to open!
The gates are now open!
Thread 1 enters the gates!
Thread 2 waits for the gates to open!
Thread 2 enters the gates!
The gates have been closed!
Thread 3 waits for the gates to open!
The gates are now open for the second time!
Thread 3 enters the gates!
The gates have been closed!
```

5 CountdownEvent

Code:

```
using static System.Console;
using static System.Threading.Thread;

class Program
{
    static CountdownEvent _countdown = new CountdownEvent(2);

    static void PerformOperation(string message, int seconds)
    {
        Sleep(TimeSpan.FromSeconds(seconds));
        WriteLine(message);
        _countdown.Signal();
    }

    static void Main(string[] args)
    {
        WriteLine("Starting two operations");
        var t1 = new Thread(() => PerformOperation("Operation 1 is completed",
4));
        var t2 = new Thread(() => PerformOperation("Operation 2 is completed",
8));
        t1.Start();
        t2.Start();
        _countdown.Wait();
        WriteLine("Both operations have been completed.");
        _countdown.Dispose();
    }
}
```

运行结果:



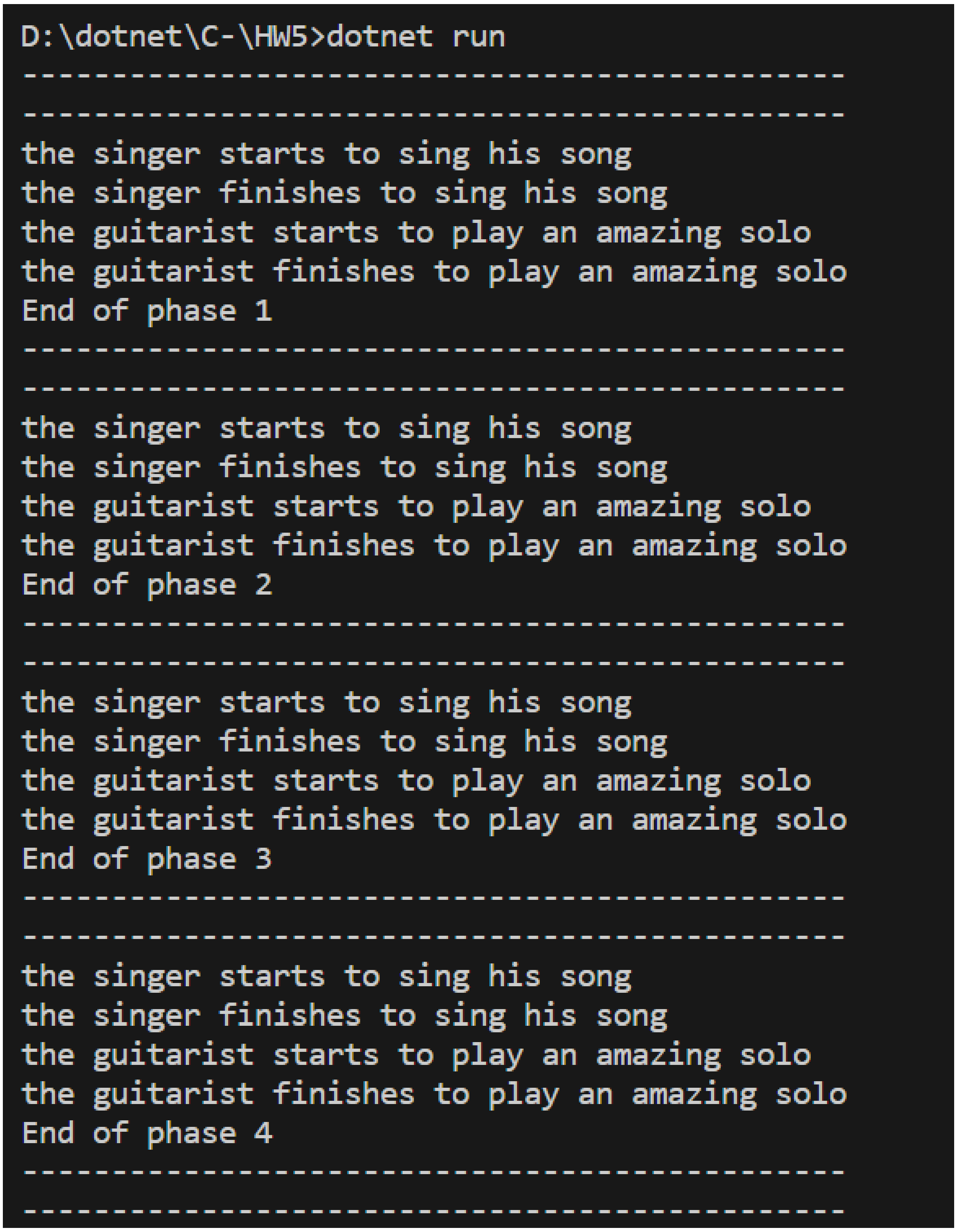
```
D:\dotnet\C-\HW5>dotnet run
Starting two operations
Operation 1 is completed
Operation 2 is completed
Both operations have been completed.
```

6 Barrier

代码:

```
using static System.Console;
using static System.Threading.Thread;
class Program
{
    static void Main(string[] args)
    {
        var t1 = new Thread(() => PlayMusic("the guitarist", "play an amazing
solo", 5));
        var t2 = new Thread(() => PlayMusic("the singer", "sing his song", 2));

        t1.Start();
        t2.Start();
    }

    //指定同步两个线程,
    static Barrier _barrier = new Barrier(2,
        b => WriteLine($"End of phase {b.CurrentPhaseNumber + 1}"));

    static void PlayMusic(string name, string message, int seconds)
    {
        for (int i = 1; i < 30; i++)
        {
            WriteLine("----------------------------------------------");
            Sleep(TimeSpan.FromSeconds(seconds));
            WriteLine($"{name} starts to {message}");
            Sleep(TimeSpan.FromSeconds(seconds));
            WriteLine($"{name} finishes to {message}");

            //在SignalAndWait处停下来,直到两个人都到了，才开始下一个回合
            _barrier.SignalAndWait();
        }
    }
}
```

Ans:

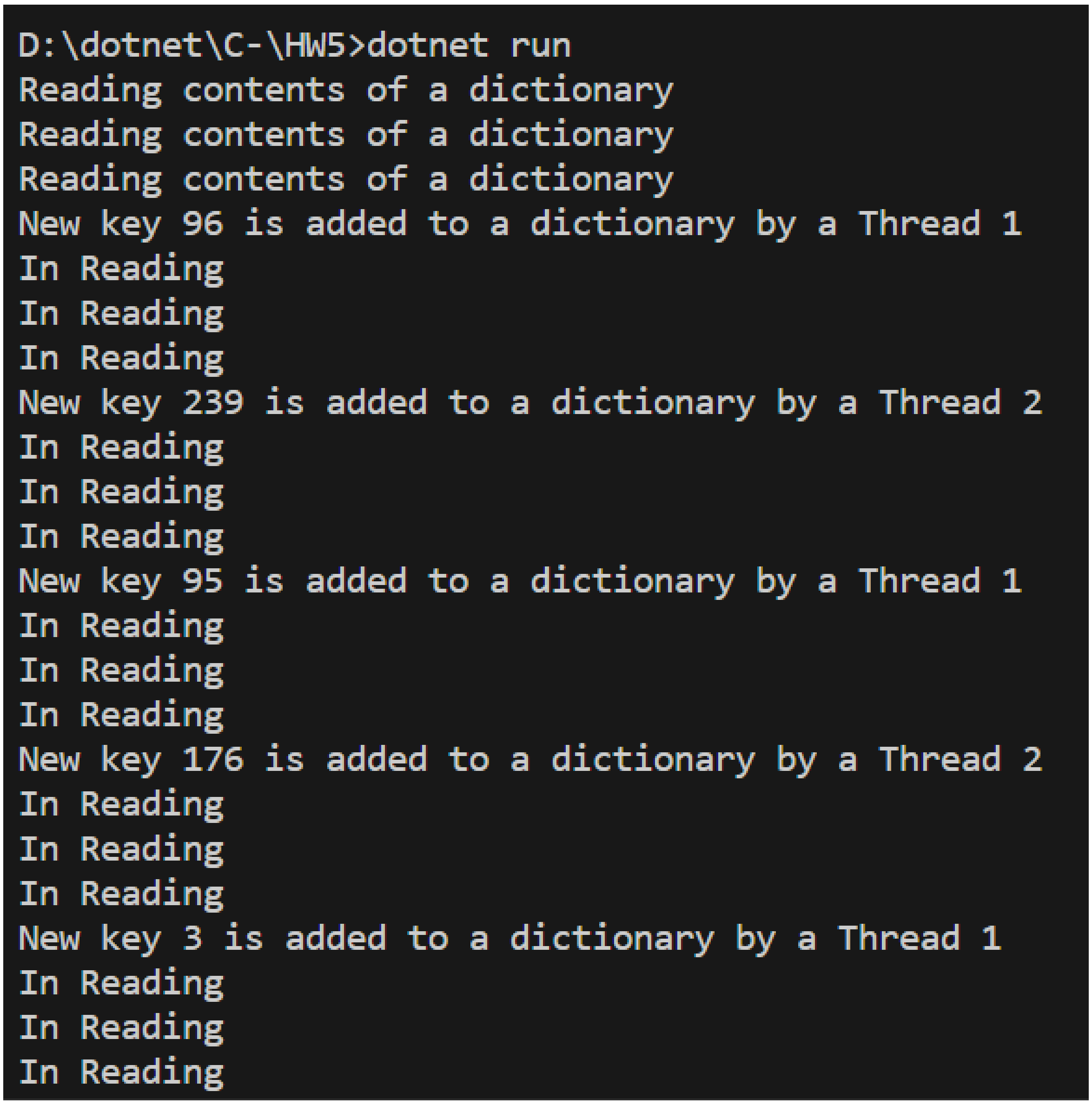


```
D:\dotnet\C-\HW5>dotnet run
----------------------------------------------
----------------------------------------------
the singer starts to sing his song
the singer finishes to sing his song
the guitarist starts to play an amazing solo
the guitarist finishes to play an amazing solo
End of phase 1
----------------------------------------------
----------------------------------------------
the singer starts to sing his song
the singer finishes to sing his song
the guitarist starts to play an amazing solo
the guitarist finishes to play an amazing solo
End of phase 2
----------------------------------------------
----------------------------------------------
the singer starts to sing his song
the singer finishes to sing his song
the guitarist starts to play an amazing solo
the guitarist finishes to play an amazing solo
End of phase 3
----------------------------------------------
----------------------------------------------
the singer starts to sing his song
the singer finishes to sing his song
the guitarist starts to play an amazing solo
the guitarist finishes to play an amazing solo
End of phase 4
----------------------------------------------
----------------------------------------------
```

7 ReaderWriterLockSlim

```
using System;
using System.Collections.Generic;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;

class Program
{
    static ReaderWriterLockSlim _rw    = new ReaderWriterLockSlim();
    static Dictionary<int, int> _items = new Dictionary<int, int>();

    static void Main(string[] args)
    {
        new Thread(Read){ IsBackground = true }.Start();
        new Thread(Read){ IsBackground = true }.Start();
        new Thread(Read){ IsBackground = true }.Start();
        new Thread(() => Write("Thread 1")){ IsBackground = true }.Start();
        new Thread(() => Write("Thread 2")){ IsBackground = true }.Start();

        Sleep(TimeSpan.FromSeconds(30));
    }

    static void Read()
    {
        WriteLine("Reading contents of a dictionary");
        while (true)
        {
            try
            {
                _rw.EnterReadLock();
                WriteLine("In Reading");
                foreach (var key in _items.Keys)
                {
                    Sleep(TimeSpan.FromSeconds(0.1));
                }
            }
            finally
            {
                _rw.ExitReadLock();
            }
        }
    }

    static void Write(string threadName)
    {
        while (true)
        {
            try
            {
                int newKey = new Random().Next(250);
                _rw.EnterUpgradeableReadLock();
                if (!_items.ContainsKey(newKey))
                {
                    try
                    {
                        _rw.EnterWriteLock();
                        _items[newKey] = 1;
                        WriteLine($"New key {newKey} is added to a dictionary by
a {threadName}");
                    }
                    finally
                    {
                        _rw.ExitWriteLock();
                    }
                }
                Sleep(TimeSpan.FromSeconds(0.1));
            }
            finally
            {
                _rw.ExitUpgradeableReadLock();
            }
        }
    }
}
```

Ans:



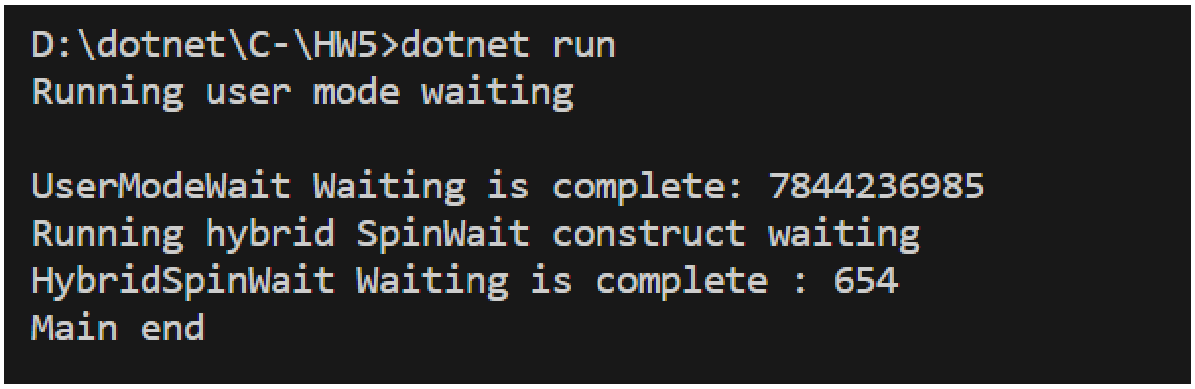
```
D:\dotnet\C-\HW5>dotnet run
Reading contents of a dictionary
Reading contents of a dictionary
Reading contents of a dictionary
New key 96 is added to a dictionary by a Thread 1
In Reading
In Reading
In Reading
New key 239 is added to a dictionary by a Thread 2
In Reading
In Reading
In Reading
New key 95 is added to a dictionary by a Thread 1
In Reading
In Reading
In Reading
New key 176 is added to a dictionary by a Thread 2
In Reading
In Reading
In Reading
New key 3 is added to a dictionary by a Thread 1
In Reading
In Reading
In Reading
```

8 SpinWait

Code:

```
using static System.Console;
using static System.Threading.Thread;
class Program
{
    static volatile bool _isCompleted = false;
    static void UserModeWait()
    {
        long count = 0;
        while (!_isCompleted)
        {
            count++;
        }
        WriteLine();
        WriteLine($"UserModeWait Waiting is complete: {count}");
    }
    static void HybridSpinWait()
    {
        long count = 0;
        var w = new SpinWait();
        while (!_isCompleted)
        {
            w.SpinOnce();
            count++;
        }
        WriteLine($"HybridSpinWait Waiting is complete : {count}");
    }
    static void Main(string[] args)
    {
        var t1 = new Thread(UserModeWait);
        var t2 = new Thread(HybridSpinWait);
        WriteLine("Running user mode waiting");
        t1.Start();
        Sleep(10000);
        _isCompleted = true;
        Sleep(TimeSpan.FromSeconds(1));
        _isCompleted = false;
        WriteLine("Running hybrid SpinWait construct waiting");
        t2.Start();
        Sleep(10000);
        _isCompleted = true;
        Sleep(1000);
        WriteLine("Main end");
    }
}
```

Ans:



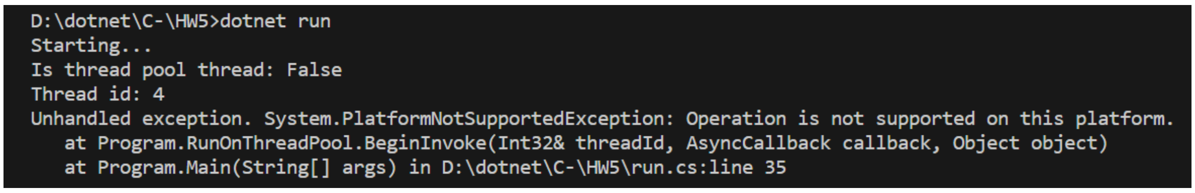
```
D:\dotnet\C-\HW5>dotnet run
Running user mode waiting

UserModeWait Waiting is complete: 7844236985
Running hybrid SpinWait construct waiting
HybridSpinWait Waiting is complete : 654
Main end
```

9 APM 异步编程模型

代码：

```
using static System.Console;
using static System.Threading.Thread;
class Program
{
    private delegate string RunOnThreadPool(out int threadId);

    private static void Callback(IAsyncResult ar)
    {
        WriteLine("Starting a callback...");
        WriteLine($"State passed to a callbak: {ar.AsyncState}");
        WriteLine($"Is thread pool thread: {CurrentThread.IsThreadPoolThread}");
        WriteLine($"Thread pool worker thread id in Callback:
{CurrentThread.ManagedThreadId}");
    }
    private static string Test(out int threadId)
    {
        WriteLine("Starting...");
        WriteLine($"Is thread pool thread: {CurrentThread.IsThreadPoolThread}");
        Sleep(TimeSpan.FromSeconds(2));
        threadId = CurrentThread.ManagedThreadId;
        return $"Thread pool worker thread id was: {threadId}";
    }

    static void Main(string[] args)
    {
        int threadId = 0;

        RunOnThreadPool poolDelegate = Test;

        var t = new Thread(() => Test(out threadId));
        t.Start();
        t.Join();

        WriteLine($"Thread id: {threadId}");

        IAsyncResult r = poolDelegate.BeginInvoke(out threadId, Callback, "a
delegate asynchronous call");
        r.AsyncWaitHandle.WaitOne();

        string result = poolDelegate.EndInvoke(out threadId, r);

        Sleep(TimeSpan.FromSeconds(2));
        WriteLine($"Thread pool worker thread id: {threadId}");
        WriteLine($"EndInvoke result: {result}");
        Sleep(TimeSpan.FromSeconds(2));
    }
}
```

Ans:



```
D:\dotnet\C-\HW5>dotnet run
Starting...
Is thread pool thread: False
Thread id: 4
Unhandled exception. System.PlatformNotSupportedException: Operation is not supported on this platform.
   at Program.RunOnThreadPool.BeginInvoke(Int32& threadId, AsyncCallback callback, Object object)
   at Program.Main(String[] args) in D:\dotnet\C-\HW5\run.cs:line 35
```

可以发现，该操作在windows 10操作系统上并不支持。

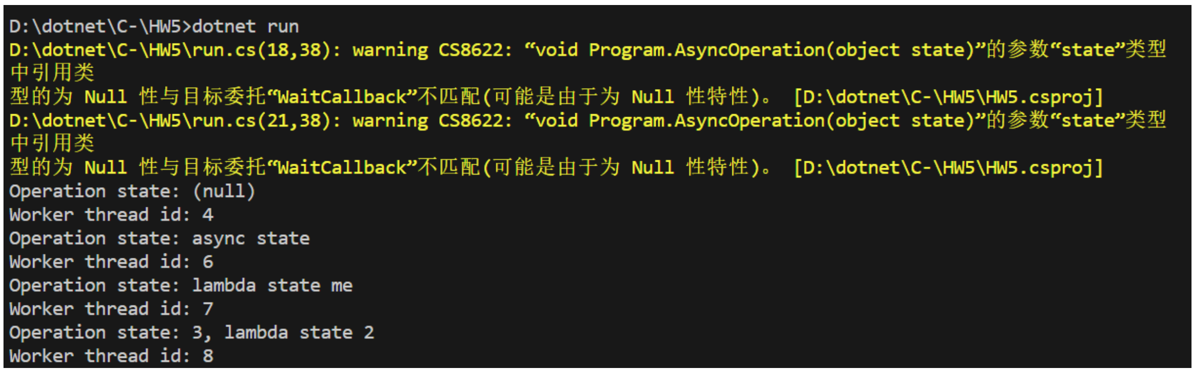
10 QueueUserWorkItem

代码：

```
using static System.Console;
using static System.Threading.Thread;

class Program
```

```
{
    private static void AsyncOperation(object state)
    {
        WriteLine($"Operation state: {state ?? "(null)"}");
        WriteLine($"Worker thread id: {CurrentThread.ManagedThreadId}");
        Sleep(TimeSpan.FromSeconds(2));
    }
    static void Main(string[] args)
    {
        const int x = 1;
        const int y = 2;
        const string lambdaState = "lambda state 2";

        ThreadPool.QueueUserWorkItem(AsyncOperation);
        Sleep(TimeSpan.FromSeconds(1));

        ThreadPool.QueueUserWorkItem(AsyncOperation, "async state");
        Sleep(TimeSpan.FromSeconds(1));

        ThreadPool.QueueUserWorkItem( state =>
        {
            WriteLine($"Operation state: {state}");
            WriteLine($"Worker thread id: {CurrentThread.ManagedThreadId}");
            Sleep(TimeSpan.FromSeconds(2));
        }, "lambda state me");

        ThreadPool.QueueUserWorkItem( _ =>
        {
            WriteLine($"Operation state: {x + y}, {lambdaState}");
            WriteLine($"Worker thread id: {CurrentThread.ManagedThreadId}");
            Sleep(TimeSpan.FromSeconds(2));
        }, "lambda state other");

        Sleep(TimeSpan.FromSeconds(2));
    }
}
```

Ans:



```
D:\dotnet\C-\HW5>dotnet run
D:\dotnet\C-\HW5\run.cs(18,38): warning CS8622: “void Program.AsyncOperation(object state)”的参数“state”类型
中引用类
型的为 Null 性与目标委托“WaitCallback”不匹配(可能是由于为 Null 性特性)。 [D:\dotnet\C-\HW5\HW5.csproj]
D:\dotnet\C-\HW5\run.cs(21,38): warning CS8622: “void Program.AsyncOperation(object state)”的参数“state”类型
中引用类
型的为 Null 性与目标委托“WaitCallback”不匹配(可能是由于为 Null 性特性)。 [D:\dotnet\C-\HW5\HW5.csproj]
Operation state: (null)
Worker thread id: 4
Operation state: async state
Worker thread id: 6
Operation state: lambda state me
Worker thread id: 7
Operation state: 3, lambda state 2
Worker thread id: 8
```

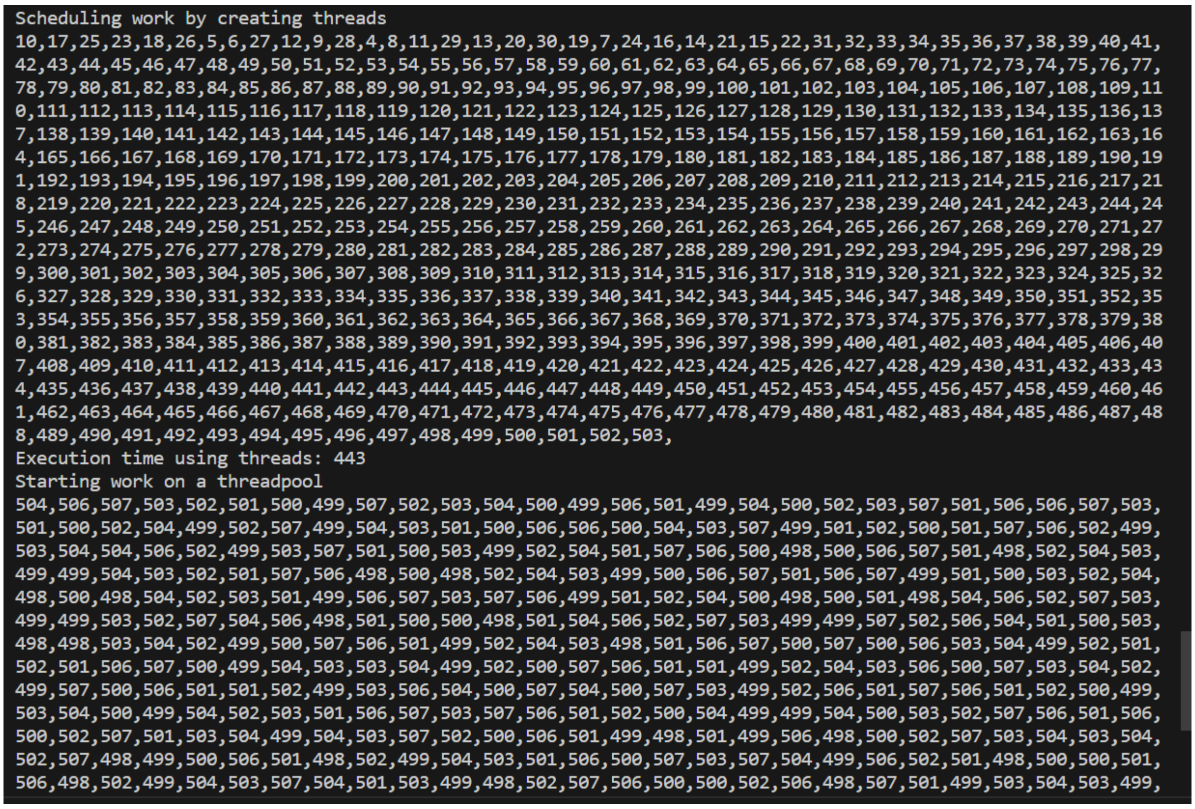
11 线程池并行度

Code:

```
using System.Diagnostics;
using static System.Console;
using static System.Threading.Thread;
```

```
class Program
{
    static void UseThreads(int numberOfOperations)
    {         using (var countdown = new CountdownEvent(numberOfOperations))
        {            WriteLine("Scheduling work by creating threads");
            for (int i = 0; i < numberOfOperations; i++)
            {                var thread = new Thread(() =>
                {
                    Write($"{CurrentThread.ManagedThreadId},");
                    Sleep(TimeSpan.FromSeconds(0.1));
                    countdown.Signal();
                });               thread.Start();
            }          countdown.Wait();
            WriteLine();
        }    }
    static void UseThreadPool(int numberOfOperations)
    {         using (var countdown = new CountdownEvent(numberOfOperations))
        {            WriteLine("Starting work on a threadpool");
            for (int i = 0; i < numberOfOperations; i++)
            {                ThreadPool.QueueUserWorkItem( _ =>
                {
                    Write($"{CurrentThread.ManagedThreadId},");
                    Sleep(TimeSpan.FromSeconds(0.1));
                    countdown.Signal();
                });           }            countdown.Wait();
            WriteLine();
        }    }
    static void Main(string[] args)
    {         const int numberOfOperations = 500;
        var sw = new Stopwatch();
        sw.Start();
        UseThreads(numberOfOperations);
        sw.Stop();
        WriteLine($"Execution time using threads: {sw.ElapsedMilliseconds}");

        sw.Reset();
        sw.Start();
        UseThreadPool(numberOfOperations);
        sw.Stop();
        WriteLine($"Execution time using the thread pool:
{sw.ElapsedMilliseconds}");
    }}
```

Ans:



```
Scheduling work by creating threads
10,17,25,23,18,26,5,6,27,12,9,28,4,8,11,29,13,20,30,19,7,24,16,14,21,15,22,31,32,33,34,35,36,37,38,39,40,41,
42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,
78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,109,11
0,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,13
7,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,16
4,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,19
1,192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,21
8,219,220,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,24
5,246,247,248,249,250,251,252,253,254,255,256,257,258,259,260,261,262,263,264,265,266,267,268,269,270,271,27
2,273,274,275,276,277,278,279,280,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,296,297,298,29
9,300,301,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,319,320,321,322,323,324,325,32
6,327,328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,344,345,346,347,348,349,350,351,352,35
3,354,355,356,357,358,359,360,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,38
0,381,382,383,384,385,386,387,388,389,390,391,392,393,394,395,396,397,398,399,400,401,402,403,404,405,406,40
7,408,409,410,411,412,413,414,415,416,417,418,419,420,421,422,423,424,425,426,427,428,429,430,431,432,433,43
4,435,436,437,438,439,440,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460,46
1,462,463,464,465,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480,481,482,483,484,485,486,487,48
8,489,490,491,492,493,494,495,496,497,498,499,500,501,502,503,
Execution time using threads: 443
Starting work on a threadpool
504,506,507,503,502,501,500,499,507,502,503,504,500,499,506,501,499,504,500,502,503,507,501,506,506,507,503,
501,500,502,504,499,502,507,499,504,503,501,500,506,506,500,504,503,507,499,501,502,500,501,507,506,502,499,
503,504,504,506,502,499,503,507,501,500,503,499,502,504,501,507,506,500,498,500,506,507,501,498,502,504,503,
499,499,504,503,502,501,507,506,498,500,498,502,504,503,499,500,506,507,501,506,507,499,501,500,503,502,504,
498,500,498,504,502,503,501,499,506,507,503,507,506,499,501,502,504,500,498,500,501,498,504,506,502,507,503,
499,499,503,502,507,504,506,498,501,500,500,498,501,504,506,502,507,503,499,499,507,502,506,504,501,500,503,
498,498,503,504,502,499,500,507,506,501,499,502,504,503,498,501,506,507,500,507,500,506,503,504,499,502,501,
502,501,506,507,500,499,504,503,503,504,499,502,500,507,506,501,501,499,502,504,503,506,500,507,503,504,502,
499,507,500,506,501,501,502,499,503,506,504,500,507,504,500,507,503,499,502,506,501,507,506,501,502,500,499,
503,504,500,499,504,502,503,501,506,507,503,507,506,501,502,500,504,499,499,504,500,503,502,507,506,501,506,
500,502,507,501,503,504,499,504,503,507,502,500,506,501,499,498,501,499,506,498,500,502,507,503,504,503,504,
502,507,498,499,500,506,501,498,502,499,504,503,501,506,500,507,503,507,504,499,506,502,501,498,500,500,501,
506,498,502,499,504,503,507,504,501,503,499,498,502,507,506,500,500,502,506,498,507,501,499,503,504,503,499,
```

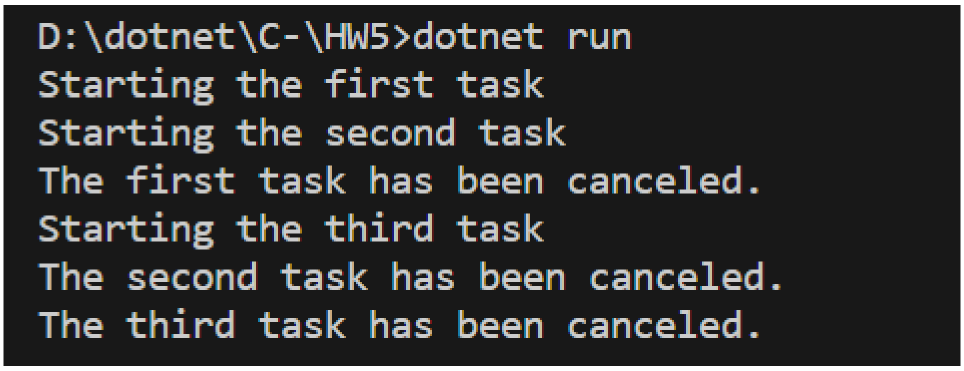
12 取消线程池中的操作

Code:

```
using static System.Console;
using static System.Threading.Thread;

class Program
{
    //轮询的方式检查IsCancellationRequested属性
    static void AsyncOperation1(CancellationToken token)
    {
        WriteLine("Starting the first task");
        for (int i = 0; i < 5; i++)
        {
            if (token.IsCancellationRequested)
            {
                WriteLine("The first task has been canceled.");
                return;
            }
            Sleep(TimeSpan.FromSeconds(1));
        }
        WriteLine("The first task has completed succesfully");
    }

    //如果取消抛出异常OperationCanceledException
    static void AsyncOperation2(CancellationToken token)
    {
        try
        {
            WriteLine("Starting the second task");
            for (int i = 0; i < 5; i++)
            {
                token.ThrowIfCancellationRequested();
                Sleep(TimeSpan.FromSeconds(1));
            }
            WriteLine("The second task has completed succesfully");
        }
        catch (OperationCanceledException)
        {
            WriteLine("The second task has been canceled.");
        }
    }

    //注册一个回调函数
    static void AsyncOperation3(CancellationToken token)
    {
        bool cancellationFlag = false;
        token.Register(() => cancellationFlag = true);
        WriteLine("Starting the third task");
        for (int i = 0; i < 5; i++)
        {
            if (cancellationFlag)
            {
                WriteLine("The third task has been canceled.");
                return;
            }
            Sleep(TimeSpan.FromSeconds(1));
        }
        WriteLine("The third task has completed succesfully");
    }

    static void Main(string[] args)
    {
        int nWaitingSeconds = 2;
        using (var cts = new CancellationTokenSource())
        {
            CancellationToken token = cts.Token;
            ThreadPool.QueueUserWorkItem(_ => AsyncOperation1(token));
            Sleep(TimeSpan.FromSeconds(nWaitingSeconds));
            cts.Cancel();
        }

        using (var cts = new CancellationTokenSource())
        {
            CancellationToken token = cts.Token;
            ThreadPool.QueueUserWorkItem(_ => AsyncOperation2(token));
            Sleep(TimeSpan.FromSeconds(nWaitingSeconds));
            cts.Cancel();
        }

        using (var cts = new CancellationTokenSource())
        {
            CancellationToken token = cts.Token;
            ThreadPool.QueueUserWorkItem(_ => AsyncOperation3(token));
            Sleep(TimeSpan.FromSeconds(nWaitingSeconds));
            cts.Cancel();
        }
        Sleep(TimeSpan.FromSeconds(2));
    }
```

```
}
```

Ans:



```
D:\dotnet\C-\HW5>dotnet run
Starting the first task
Starting the second task
The first task has been canceled.
Starting the third task
The second task has been canceled.
The third task has been canceled.
```

Part 2 作业

HW 1

Q1: Mutex中如果拥有互斥体的线程没有释放互斥而结束，其他等待线程是否会死等?

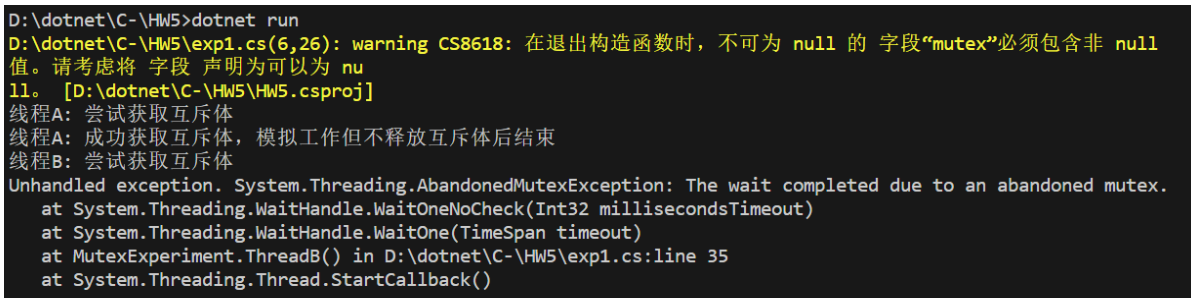
实验思路:

1. 创建一个互斥体（Mutex），确保它在不同线程间共享。
2. 启动一个线程（线程A），让它获取互斥体的所有权。
3. 在线程A中，不释放互斥体，直接结束线程A的执行。
4. 启动另一个线程（线程B），尝试获取同一个互斥体的所有权。
5. 观察线程B的行为，判断其是否会无限等待或者能够获取到互斥体的所有权。

实验代码:

```
using System;
using System.Threading;

class MutexExperiment
{
    private static Mutex mutex;

    static void Main(string[] args)
    {
        mutex = new Mutex(false, "TestMutex");

        // 启动线程A
        Thread threadA = new Thread(ThreadA);
        threadA.Start();
        threadA.Join(); // 等待线程A结束

        // 启动线程B
        Thread threadB = new Thread(ThreadB);
        threadB.Start();
        threadB.Join(); // 等待线程B结束
    }

    static void ThreadA()
    {
        Console.WriteLine("线程A: 尝试获取互斥体");
        mutex.WaitOne(); // 获取互斥体
        Console.WriteLine("线程A: 成功获取互斥体，模拟工作但不释放互斥体后结束");
        // 注意：这里不调用 mutex.ReleaseMutex();
        // 线程A结束，没有释放互斥体
    }

    static void ThreadB()
    {
        Console.WriteLine("线程B: 尝试获取互斥体");
        bool gotMutex = mutex.WaitOne(TimeSpan.FromSeconds(5)); // 尝试获取互斥体，等待最多5秒

        if (gotMutex)
        {
            Console.WriteLine("线程B: 成功获取互斥体");
            mutex.ReleaseMutex(); // 释放互斥体
        }
        else
        {
            Console.WriteLine("线程B: 获取互斥体超时");
        }
    }
}
```

实验分析：



```
D:\dotnet\C-\HW5>dotnet run
D:\dotnet\C-\HW5\exp1.cs(6,26): warning CS8618: 在退出构造函数时，不可为 null 的 字段“mutex”必须包含非 null
值。请考虑将 字段 声明为可以为 nu
ll。 [D:\dotnet\C-\HW5\HW5.csproj]
线程A: 尝试获取互斥体
线程A: 成功获取互斥体，模拟工作但不释放互斥体后结束
线程B: 尝试获取互斥体
Unhandled exception. System.Threading.AbandonedMutexException: The wait completed due to an abandoned mutex.
   at System.Threading.WaitHandle.WaitOneNoCheck(Int32 millisecondsTimeout)
   at System.Threading.WaitHandle.WaitOne(TimeSpan timeout)
   at MutexExperiment.ThreadB() in D:\dotnet\C-\HW5\exp1.cs:line 35
   at System.Threading.Thread.StartCallback()
```

运行结果表明，在实验中发生了一个`System.Threading.AbandonedMutexException`异常，这是因为互斥体被线程A“遗弃”了。这种情况发生在当一个线程（线程A）在没有释放互斥体的情况下结束时，而另一个线程（线程B）正在等待获取这个互斥体。

异常解释：

- **AbandonedMutexException**: 这个异常表示一个线程尝试获取一个已被另一个线程获取但未正确释放的互斥体。在.NET中，如果一个线程拥有一个互斥体的所有权并且在没有调用`ReleaseMutex`方法释放互斥体的情况下终止，互斥体就被视为“遗弃”的。
- 当另一个线程（如线程B）在等待获取这个“遗弃”的互斥体时，.NET运行时允许这个等待的线程获取互斥体的所有权，但同时会抛出`AbandonedMutexException`异常来指示互斥体是在未正常释放的情况下被获得的。

如何处理这个异常？

- 在实践中，当遇到`AbandonedMutexException`时，你应该仔细检查并修改代码，确保所有的互斥体在使用后都被正确释放，即在`finally`块中调用`ReleaseMutex`方法。
- 如果`AbandonedMutexException`是在预期内的，可以在获取互斥体时捕获这个异常，并根据应用程序的需要适当地处理它。

实验结论：

- 当一个线程获取了互斥体的所有权但在结束前没有释放互斥体时，其他线程在尝试获取这个互斥体的所有权将会根据等待策略而定。如果设置了超时时间，线程会在超时后继续执行；如果没有设置超时时间，线程将会无限等待。
- 在.NET环境中，当拥有互斥体的线程结束时，互斥体不会自动释放。这意味着，如果其他线程试图获取这个互斥体，而没有设置超时，那么这些线程将会无限等待。因此，始终在使用完互斥体后调用`ReleaseMutex`方法来释放互斥体是一个好的编程实践。
- 这个异常指出了一个重要的资源管理原则：确保你获取的所有资源（如互斥体）在不再需要时都被正确释放。这不仅是良好的编程实践，也是防止资源泄露和其他潜在问题的重要措施。

HW 2

Q2: 设计一个实验说明EnterUpgradeableReadLock的优势

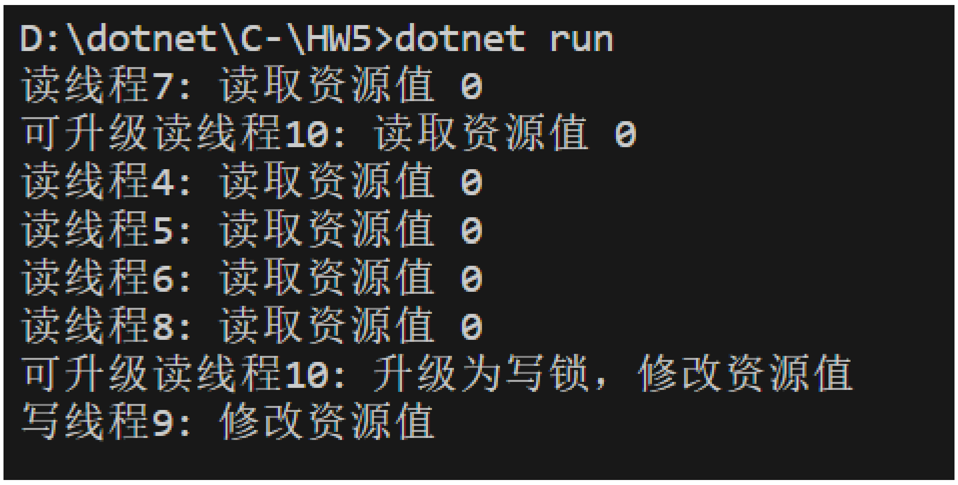
实验思路：

1. 使用`ReaderWriterLockSlim`类来设计实验，这个类允许多个线程同时读取资源但只允许一个线程写入，以提高并发性。
2. 使用`EnterUpgradeableReadLock`方法获取可升级的读锁，它允许一个线程在读取资源时预留写入的能力。
3. 创建多个读线程和写线程，让它们尝试读取和修改共享资源。
4. 比较在同样的情况下，使用`EnterReadLock`和`EnterUpgradeableReadLock`时的性能和行为差异。

实验代码：

```
using System;
using System.Threading;

class ReaderWriterLockSlimExperiment
{
    private static ReaderWriterLockSlim rwLock = new ReaderWriterLockSlim();
    private static int sharedResource = 0;

    static void Main(string[] args)
    {
        // 启动读线程
        Thread[] readers = new Thread[5];
        for (int i = 0; i < readers.Length; i++)
        {
            readers[i] = new Thread(ReadResource);
            readers[i].Start();
        }

        // 启动写线程
        Thread writer = new Thread(WriteResource);
        writer.Start();

        // 启动可升级读线程
        Thread upgradableReader = new Thread(UpgradeableReadResource);
        upgradableReader.Start();

        foreach (var reader in readers)
        {
            reader.Join();
        }
        writer.Join();
        upgradableReader.Join();
    }

    static void ReadResource()
    {
        rwLock.EnterReadLock();
        try
        {
            Console.WriteLine($"读线程{Thread.CurrentThread.ManagedThreadId}: 读取
资源值 {sharedResource}");
            Thread.Sleep(100); // 模拟读取操作
        }
        finally
        {
            rwLock.ExitReadLock();
        }
    }

    static void WriteResource()
    {
        rwLock.EnterWriteLock();
        try
        {
            Console.WriteLine($"写线程{Thread.CurrentThread.ManagedThreadId}: 修改
资源值");
            sharedResource = new Random().Next(100); // 模拟写操作
            Thread.Sleep(100); // 模拟写操作
        }
        finally
        {
            rwLock.ExitWriteLock();
        }
    }

    static void UpgradeableReadResource()
    {
        rwLock.EnterUpgradeableReadLock();
        try
        {
            Console.WriteLine($"可升级读线程{Thread.CurrentThread.ManagedThreadId}:
读取资源值 {sharedResource}");
            if (sharedResource < 50) // 条件判断，决定是否需要写操作
            {
                rwLock.EnterWriteLock();
                try
                {
                    Console.WriteLine($"可升级读线程
{Thread.CurrentThread.ManagedThreadId}: 升级为写锁，修改资源值");
                    sharedResource = new Random().Next(100); // 模拟写操作
                    Thread.Sleep(100); // 模拟写操作
                }
                finally
                {
                    rwLock.ExitWriteLock();
                }
            }
            Thread.Sleep(100); // 模拟读取操作
        }
        finally
        {
            rwLock.ExitUpgradeableReadLock();
        }
    }
}
```

实验分析：



```
D:\dotnet\C-\HW5>dotnet run
读线程7: 读取资源值 0
可升级读线程10: 读取资源值 0
读线程4: 读取资源值 0
读线程5: 读取资源值 0
读线程6: 读取资源值 0
读线程8: 读取资源值 0
可升级读线程10: 升级为写锁, 修改资源值
写线程9: 修改资源值
```

- `ReaderWriterLockSlim`允许多个线程同时执行读操作，这样可以提高读取数据的并发性。
- 可升级读锁 `EnterUpgradeableReadLock` 允许一个线程在读取数据的同时保留升级为写锁的能力，这对于需要读取后可能需要写入的场景非常有用。
- 写操作是排他的。当一个线程获取写锁时，其他线程无论是读线程还是写线程都必须等待，直到写锁被释放。

实验结论：

- `EnterUpgradeableReadLock`的优势在于它提供了一种机制，使得一个线程可以在保持读取操作的同时，根据需要安全地升级为写操作，而无需释放锁并重新竞争写锁。这减少了线程切换和锁竞争，提高了效率。
- 在需要读取后可能会写入的场景下，使用`EnterUpgradeableReadLock`可以优化性能，因为它避免了从读锁转换到写锁的额外竞争。
- 使用`EnterUpgradeableReadLock`时，需要注意，只能有一个线程持有可升级的读锁。如果其他线程也需要写入，它们必须等待可升级的读锁被释放或升级。

HW 3

Q3: 使用单个线程与多个线程(6个)测试对比ConcurrentDictionary数据结构的性能, 涉及操作均包括读取、写入、删除等操作。

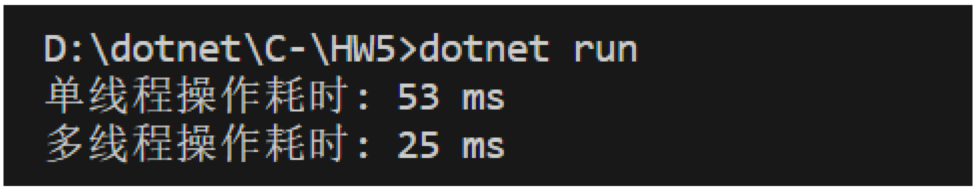
实验思路：

1. 创建一个 `ConcurrentDictionary` 实例以供单线程和多线程访问。
2. 设计实验以测试以下操作：
   - 写入：向`ConcurrentDictionary`中添加元素。
   - 读取：从`ConcurrentDictionary`中读取元素。
   - 删除：从`ConcurrentDictionary`中删除元素。
3. 首先使用单个线程执行一定数量的读取、写入、删除操作，并记录所需时间。
4. 然后创建六个线程，每个线程执行类似的操作，并记录总时间。
5. 比较单线程和多线程操作`ConcurrentDictionary`的性能差异。

实验代码：

```
using System;
using System.Collections.Concurrent;
using System.Diagnostics;
using System.Threading;

class ConcurrentDictionaryPerformanceTest
{
    private static ConcurrentDictionary<int, int> concurrentDictionary = new
ConcurrentDictionary<int, int>();
    private static int itemCount = 100000; // 操作的项目数量

    static void Main(string[] args)
    {
        Stopwatch stopwatch = new Stopwatch();

        // 单线程测试
        stopwatch.Start();
        PerformDictionaryOperations();
        stopwatch.Stop();
        Console.WriteLine($"单线程操作耗时: {stopwatch.ElapsedMilliseconds} ms");

        concurrentDictionary.Clear(); // 清空字典准备多线程测试

        // 多线程测试
        stopwatch.Restart();
        Thread[] threads = new Thread[6];
        for (int i = 0; i < threads.Length; i++)
        {
            threads[i] = new Thread(PerformDictionaryOperations);
            threads[i].Start();
        }

        foreach (var thread in threads)
        {
            thread.Join(); // 等待所有线程完成
        }
        stopwatch.Stop();
        Console.WriteLine($"多线程操作耗时: {stopwatch.ElapsedMilliseconds} ms");
    }

    static void PerformDictionaryOperations()
    {
        for (int i = 0; i < itemCount; i++)
        {
            concurrentDictionary.TryAdd(i, i);
        }
        for (int i = 0; i < itemCount; i++)
        {
            concurrentDictionary.TryGetValue(i, out int value);
        }
        for (int i = 0; i < itemCount; i++)
        {
            concurrentDictionary.TryRemove(i, out int value);
        }
    }
}
```

实验分析：



```
D:\dotnet\C-\HW5>dotnet run
单线程操作耗时: 53 ms
多线程操作耗时: 25 ms
```

- 在单线程环境下，`ConcurrentDictionary`的性能可能不如普通的`Dictionary`，因为`ConcurrentDictionary`为了线程安全付出了额外的性能代价。
- 在多线程环境下，`ConcurrentDictionary`的性能优势应该更为明显，因为它允许多线程并发无锁地对字典进行操作，这通常会比使用锁或其他同步机制要快。

实验结论：

- 如果实验结果显示多线程操作比单线程快很多，那么可以得出结论`ConcurrentDictionary`在多线程环境中比单线程环境更能体现其性能优势。
- 如果多线程操作并没有显著提速，可能是因为线程间的竞争导致了性能瓶颈，或者操作数量不足以体现并发操作的优势。
- 总的来说，`ConcurrentDictionary`是为了多线程环境设计的，并且在这种环境下运行得更好。在单线程应用中使用`ConcurrentDictionary`可能不会得到最佳性能。

HW 4

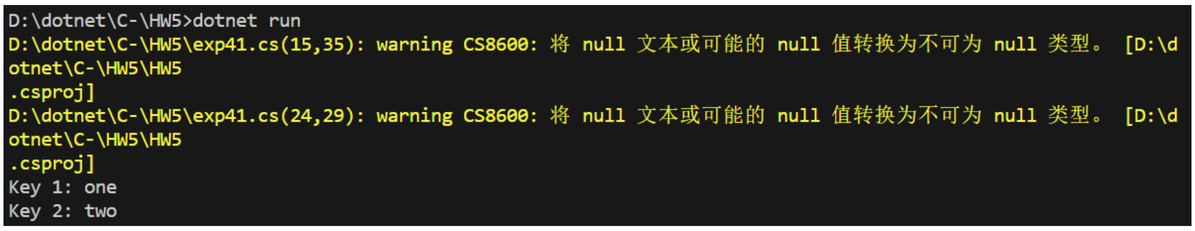
Q4：查找资料，熟悉以下数据结构的用法
(1) ConcurrentDictionary
(2) ConcurrentQueue
(3) ConcurrentStack
(4) ConcurrentBag
(5) BlockingCollection

实验思路：

1. 对于每个并发集合，首先查找官方文档或可靠资料，以理解它们的特性和典型用法。
2. 设计简单的代码示例，演示每种数据结构的基本操作，如添加、移除和遍历。
3. 分析每种数据结构在特定场景下的适用性和性能特点。
4. 得出结论，总结每种数据结构的最佳使用场景。

(1) ConcurrentDictionary

```
using System;
using System.Collections.Concurrent;

public class ConcurrentDictionaryExample
{
    public static void Main()
    {
        var cd = new ConcurrentDictionary<int, string>();

        // 添加元素
        cd.TryAdd(1, "one");
        cd.TryAdd(2, "two");

        // 读取元素
        if (cd.TryGetValue(1, out string value1))
        {
            Console.WriteLine($"Key 1: {value1}");
        }

        // 更新元素
        cd.TryUpdate(1, "ONE", "one");

        // 删除元素
        cd.TryRemove(1, out string removedValue);

        // 遍历字典
        foreach (var kvp in cd)
        {
            Console.WriteLine($"Key {kvp.Key}: {kvp.Value}");
        }
    }
}
```

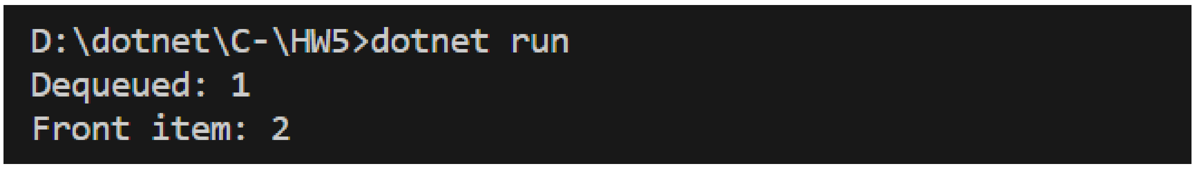


```
D:\dotnet\C-\HW5>dotnet run
D:\dotnet\C-\HW5\exp41.cs(15,35): warning CS8600: 将 null 文本或可能的 null 值转换为不可为 null 类型。 [D:\d
otnet\C-\HW5\HW5
.csproj]
D:\dotnet\C-\HW5\exp41.cs(24,29): warning CS8600: 将 null 文本或可能的 null 值转换为不可为 null 类型。 [D:\d
otnet\C-\HW5\HW5
.csproj]
Key 1: one
Key 2: two
```

实验分析与结论：
`ConcurrentDictionary` 是一个线程安全的字典，适合在多线程环境中使用，能有效地处理并发的添加、获取、更新和删除操作。它提供了原子操作，避免了使用锁的开销。适合于需要频繁读写的情况，特别是键值对数量动态变化的场景。

(2) ConcurrentQueue

```
using System;
using System.Collections.Concurrent;

public class ConcurrentQueueExample
{
    public static void Main()
    {
        var cq = new ConcurrentQueue<int>();

        // 入队
        cq.Enqueue(1);
        cq.Enqueue(2);

        // 出队
        if (cq.TryDequeue(out int result))
        {
            Console.WriteLine($"Dequeued: {result}");
        }

        // 查看队首元素
        if (cq.TryPeek(out result))
        {
            Console.WriteLine($"Front item: {result}");
        }
    }
}
```

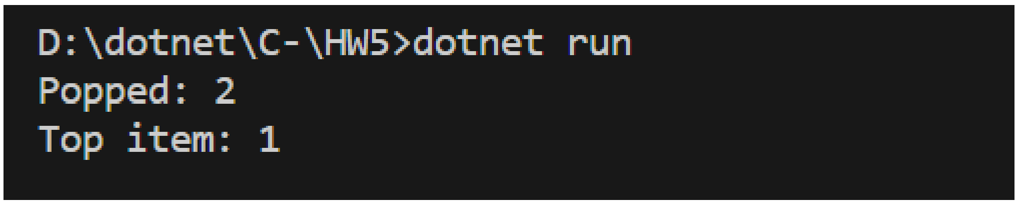


```
D:\dotnet\C-\HW5>dotnet run
Dequeued: 1
Front item: 2
```

实验分析与结论:
`ConcurrentQueue`是一个线程安全的先进先出（FIFO）集合。它适用于生产者-消费者模式，其中生产者线程入队元素，消费者线程出队元素。由于它提供无锁的并发操作，性能通常比传统同步队列要好。

(3) ConcurrentStack

```
using System;
using System.Collections.Concurrent;

public class ConcurrentStackExample
{
    public static void Main()
    {
        var cs = new ConcurrentStack<int>();

        // 入栈
        cs.Push(1);
        cs.Push(2);

        // 出栈
        if (cs.TryPop(out int result))
        {
            Console.WriteLine($"Popped: {result}");
        }

        // 查看栈顶元素
        if (cs.TryPeek(out result))
        {
            Console.WriteLine($"Top item: {result}");
        }
    }
}
```

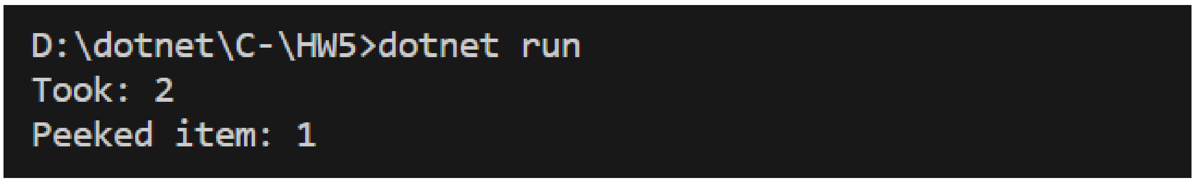


```
D:\dotnet\C-\HW5>dotnet run
Popped: 2
Top item: 1
```

实验分析与结论:
`ConcurrentStack`是一个线程安全的后进先出（LIFO）集合。它适合于实现并发的栈操作，特别是在需要后进元素先处理的场景。与`ConcurrentQueue`类似，它也提供无锁并发操作。

(4) ConcurrentBag

```
using System;
using System.Collections.Concurrent;

public class ConcurrentBagExample
{
    public static void Main()
    {
        var cb = new ConcurrentBag<int>();

        // 添加元素
        cb.Add(1);
        cb.Add(2);

        // 尝试取出
        if (cb.TryTake(out int result))
        {
            Console.WriteLine($"Took: {result}");
        }

        // 查看一个元素
        if (cb.TryPeek(out result))
        {
            Console.WriteLine($"Peeked item: {result}");
        }
    }
}
```

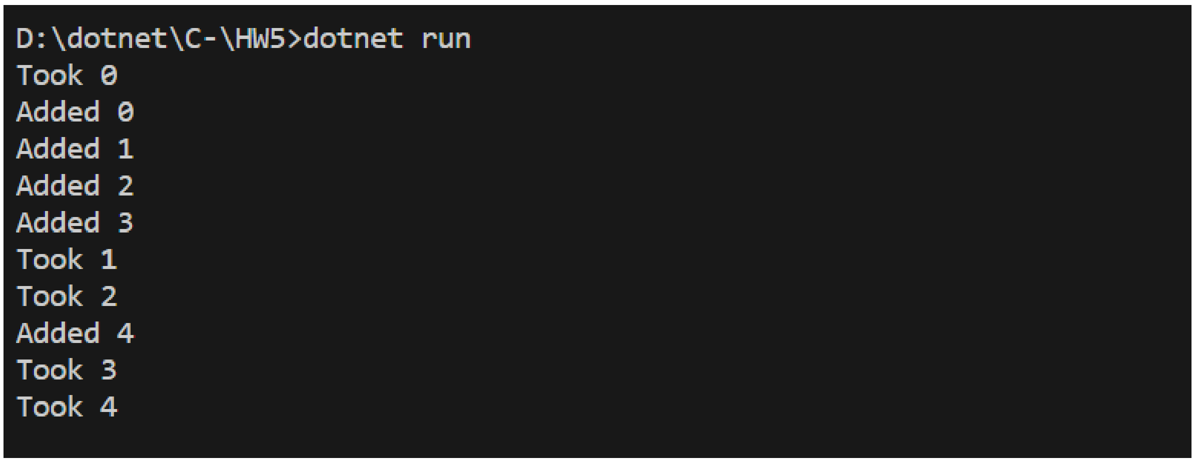


```
D:\dotnet\C-\HW5>dotnet run
Took: 2
Peeked item: 1
```

实验分析与结论：
`ConcurrentBag`是一个线程安全的无序集合，适合在多线程环境中使用，特别是在不关心元素顺序的情况下。它提供了无锁的并发添加和删除操作，适合于工作窃取和任务调度场景。

(5) BlockingCollection

```
using System;
using System.Collections.Concurrent;
using System.Threading.Tasks;

public class BlockingCollectionExample
{
    public static void Main()
    {
        var bc = new BlockingCollection<int>(boundedCapacity: 2);

        // 生产者任务
        Task producer = Task.Run(() =>
        {
            for (int i = 0; i < 5; i++)
            {
                bc.Add(i);
                Console.WriteLine($"Added {i}");
            }
            bc.CompleteAdding();
        });

        // 消费者任务
        Task consumer = Task.Run(() =>
        {
            while (!bc.IsCompleted)
            {
                if (bc.TryTake(out int item, TimeSpan.FromSeconds(1)))
                {
                    Console.WriteLine($"Took {item}");
                }
            }
        });

        Task.WaitAll(producer, consumer);
    }
}
```



```
D:\dotnet\C-\HW5>dotnet run
Took 0
Added 0
Added 1
Added 2
Added 3
Took 1
Took 2
Added 4
Took 3
Took 4
```

实验分析与结论:
`BlockingCollection`是一个线程安全的集合，可以设置最大容量，并提供了阻塞和限界功能，适用于生产者-消费者模式。生产者可以添加元素直到容量达到限制，而消费者可以取出元素。如果集合为空，消费者会等待直到有元素可用；如果集合已满，生产者会等待直到有空间添加新元素。这个特性使得它适合于任务调度和流量控制场景。