Faculty of Computers, Informatics and Microelectronics

Technical University of Moldova

Network Programming

Laboratory work #2

Author:

Ganusceac Vlad

Supervisor:

Gavrilita Mihail

2019

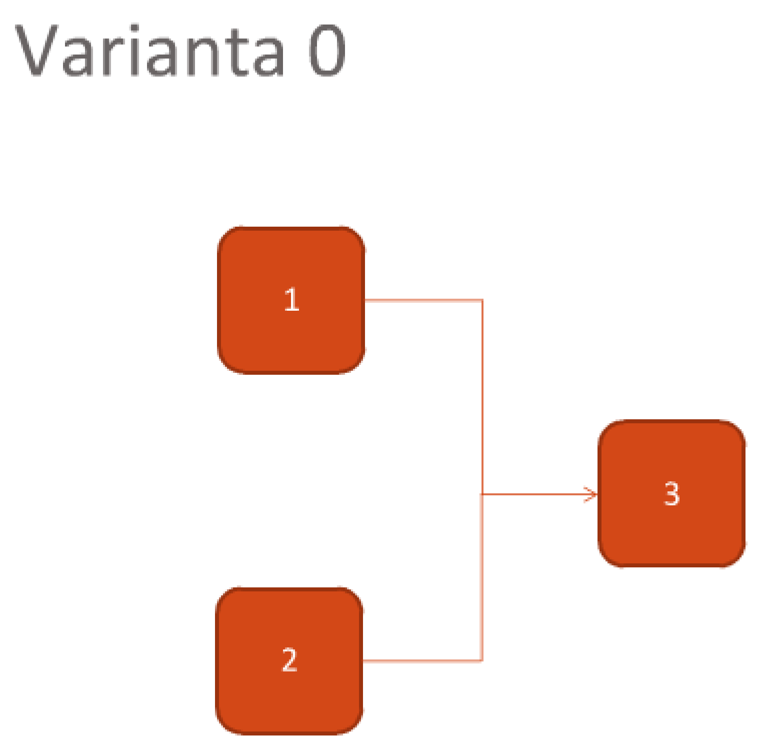
Chişinau

Parallel programming and concurrency

The concurrency is a very useful technique in parallel programming, because it increases program's speed of execution (especially if we are working with databases, GUI and some pre-calculation resources). There is a possibility to protect some special algorithms (isolating them into concrete threads). Working with threads is a good practice if we want to model our system as effective as it is possible.

But, unfortunatelly, there are some disadvantages of this technique. First of all, the complexity of the program is growing, because we should synchronize some of our threads, and also we should be able to transfer data properly between the threads (especially if more than one thread has direct access to specific data i.e. common variables). Besides, could appear the problem of communication between two or more threads (for exmple, if one of threads shouldn't be executed earlier then specific one).

At this laboratory work we deal with such a problem: in my variant exist three processes (I'll call them first, second and third process, respectively). First two depend on the third one (i.e. they can't be executed till the third is not finished). I'll say more: "The first two threads should wait till the third one will be launched and it will terminate its process".



The first method

> > There is always a good, fast, beautiful... And in the same time wrong solution!

There are different strategies how to implement this requirement. I've choosed the following one: all indicated threads above will have access to specific one field (of boolean type, for example). I'll call this variable ***OK***. If it has value *true* than it means that the third process has been launched, otherwise it wasn't launched. But, unfortunatelly, there is a problem: the fact that the process is launched doesn't gaurantee that it has been ended yet.

To solve this problem we should check if the thread is still alive. In c# class Thread has a propertie *IsAlive* which returns corresponding value if the thread wasn't terminated, otherwice - false.

It's everything what I need in order to implement the given laboratory work task.

Lets see some concrete aspects of my code written in .Net:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace checkThreads
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread first = new Thread(() => Print(1));
            Thread second = new Thread(() => Print(2));
            Thread third = new Thread(() => Print(3));
            bool OK = false;
            do
            {
                if (!OK)
                {
                    third.Start();
                    OK = true;
                }
                else
                {
                    if (!third.IsAlive)
                    {
                        first.Start();
                        second.Start();
                        OK = false;
                    }
                }
            } while (OK);
            Console.ReadKey();
        }

        static void Print(int ord)
        {
            for (int i = 0; i < 200; i++)
            {
                System.Console.Write(ord);
            }
        }
    }
}
```

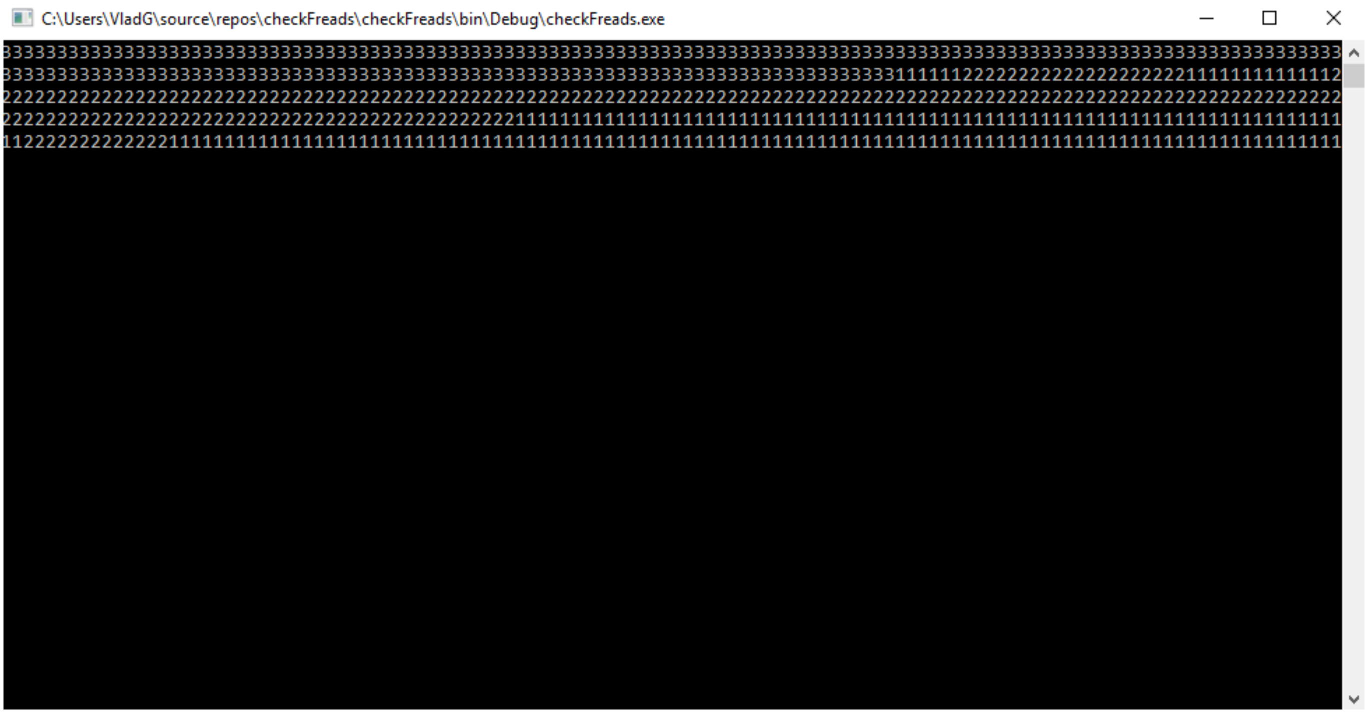
Thirst of all, in the code above I have initialied three threads, but didn't put them on the execution. There are some reasons why I did so.

In the next step I have set false value to the boolean variable ***OK***. It means that the thread what I need has not been launched yet.

In the ***do-while*** loop I'm trying to check if the third thread wasn't launched. If it's so, I'm calling the Start() method of the corresponding thread and the common variable ***OK*** has true value right now.

The execution part of this loop is terminated, but the condition of it is true. That means that it will be executed another one time. Now the variable ***OK*** has true value (This means that I have to check if the third thread has been terminated: if it's so the first two threads will be launched, otherwise the loop goes to next iteration).

The output:



C:\Users\VladG\source\repos\checkFreads\checkFreads\bin\Debug\checkFreads.exe

```
333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333111112222222222222222222111111111111112
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222111111111111111111111111111111111111111111111111111111111111111111111111111
112222222222222111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

After that the third thread has been executed, the first two were executed simultanesly (parallel). In this context I mean that the kernel was switching from first to second process and viceversa (giving each of them the concrete period to work). The kernel did so while at both processes were alive. In a moment was remaining only one process and it was working till its end.

What are disadvantages?

- First of all, the thing that the program always gives us the expected result doesn't say that the logic inside the code has been implemented correctly. Why do we need manually to fix the order of the execution of all threads inside the main one? For example: what would we do if will be 100 or 1000 threads running in our program?
- The complexity of the code written in such a manner rises exponentialy and depends directly on number of threads. There is a huge probability that the person who had written this code will not be able all the cases.

The second method (Blocking + Spinning)

This one method will be based on previous, but inside it will be implemented threads which will communicate between each other through common *static* variable. And what is really important - this time I'll not fix the program inside the main thread.

In order to prevent different ambiguities, I'll use the locker, because by definition to any class object can't reffer more than one thread simultanesly (i.e. if more than one thread contends the lock, they are queued on a “ready queue” and granted the lock on a first-come, first-served basis). So, I have used **lock**. Another one way was to use **Mutex**, but the lock construct is faster and more convenient.

Also in the example below I have added lambda expressions in order to have more flexibility in the following:

- If I want to enter more than one methods inside thread's body, I should not to write another method which encapsulates corresponding threads;
- The parameter to my thread should be viewed not as a method, but as a set of instructions;
- Otherwise I should use the delegate method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace _checkThreads2
{
    class Program
    {
        static bool OK = false;
        static readonly Object _lock = new object();

        static void Print(int val)
        {
            for (int i = 0; i < 200; i++)
            {
                Console.Write(val);
            }
        }

        static void Main(string[] args)
        {
            new Thread(() => {
                while (!OK)
                {
                    Thread.Sleep(100);
                }
                Print(1);
            }).Start();
            new Thread(() =>
            {
                while (!OK)
                {
                    Thread.Sleep(100);
                }
                Print(2);
            }).Start();

            new Thread(() =>
            {
                Print(3);
                lock (_lock)
                {
                    OK = true;
                }

            }).Start();
            Console.ReadKey();
        }
    }
}
```

This code example gives the same output as the precedent one, but it is also written correctly from logic perspective.

As I mentioned above it is possible to use *delegate()* keyword instead of lambda expression - the result will be the same. For example, the first thread would look like this:

```

        new Thread(delegate()
            {
                while (!OK)
                {
                    Thread.Sleep(100);
                }
                Print(1);
            }).Start();
```

What are disadvantages?

- First of all, this is very wasteful on processor time: as far as the CLR and operating system are concerned, the thread is performing an important calculation, and so gets allocated resources accordingly! ***But I should admit that it may be very effective iff we know precisely that the condition which breaks the spinnig will appear very soon***.
- Unfortunately, some problems may arise and their nature is based on concurrency issues linked with the proceed flag;

- .Net Framework provides the following mechanisms in order to prevent bad spinning: *SpinLock* and *SpinWait*.

The *lock* block could be written in the following manner:

```
new Thread(() =>
    {
        Print(3);

        object __lockObj = _lock;
        bool __lockWasTaken = false;
        try
        {
            OK = true;
        }
        finally
        {
            if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
        }
    }).Start();
```

The __lockWasTaken is a boolean variable which checks if the object still exists. It could be explained that in a moment the object which is participating in the lock statement was aborted/disposed outside of the thread. Thus, not only current thread can have access to the lock's body content. It can lead to some unexpected behaviour (depends what kind of parallel processes will have simultan access to common data).

But there are good news: the code above is explicitely implemented in .Net when I'm writing **lock(_lock){ ... }**.

Another one question during this laboratory work may appeare is why did I not used Join() methods (at least after that the main thread has finished). The answer is that by default all the thread processes are *foreground* (it means that the program (here: the main thread) will wait untill at least one process is alive/running).

The second method modified (SpinSleep + [Synchronization] & ContextBoundObject)

This time I'll not use the locker explicitly, but will create a class which will be synchronized and its object will be bounded by the context.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
//
using System.Runtime.Remoting.Contexts;

namespace _checkThreads6
{
    class Program
    {
        static bool OK = false;

        static void Show(Printer printer)
        {
            printer.Print();
        }

        [Synchronization]
        class Printer : ContextBoundObject
        {
            private readonly int val;
            private readonly int print_N_times;

            public Printer(int val, int print_N_times)
            {
                this.val = val;
                this.print_N_times = print_N_times;
            }

            public Printer(Printer printer)
            {
                this.val = printer.val;
                this.print_N_times = printer.print_N_times;
            }

            public Printer Clone()
            {
                return new Printer(this);
            }

            public void Print()
            {
                for (int i = 0; i < print_N_times; i++)
                {
                    Console.Write(val);
                }
            }
        }

        static void Main(string[] args)
        {
            new Thread(_ => { while (!OK) Thread.SpinWait(30); new Printer(1, 
200).Print(); }).Start();

            new Thread(_ => { while (!OK) Thread.SpinWait(30);  new Printer(2, 
200).Print(); }).Start();

            new Thread(_ => { new Printer(3, 200).Print(); OK = true; }).Start();

            Console.ReadKey();
        }
    }
}
```

The class inside itself keeps the value which should be printed and the number of iterations which will do the printed statement. In the *Main()* function this time I'm not calling the static method, because it is inside the class, it is thread-safe, and depending on the parametrized constructor parameters will display the respective output.

Only one thread can execute code at a time in an object of this class.

This is very flexible instrument, because the `[Synchronization]` flag may embibe different parameters:

Constant	***Meaning***
NOT_SUPPORTED	Equivalent to not using the Synchronized attribute
SUPPORTED	Joins the existing synchronization context if instantiated from another synchronized object, otherwise remains unsynchronized
REQUIRED (default)	Joins the existing synchronization context if instantiated from another synchronized object, otherwise creates a new context
REQUIRES_NEW	Always creates a new synchronization context

Example:

```
[Synchronization (SynchronizationAttribute.REQUIRES_NEW)]
public class SynchronizedB : ContextBoundObject { ...
```

The third method (semaphore)

This method requires at least two variables:

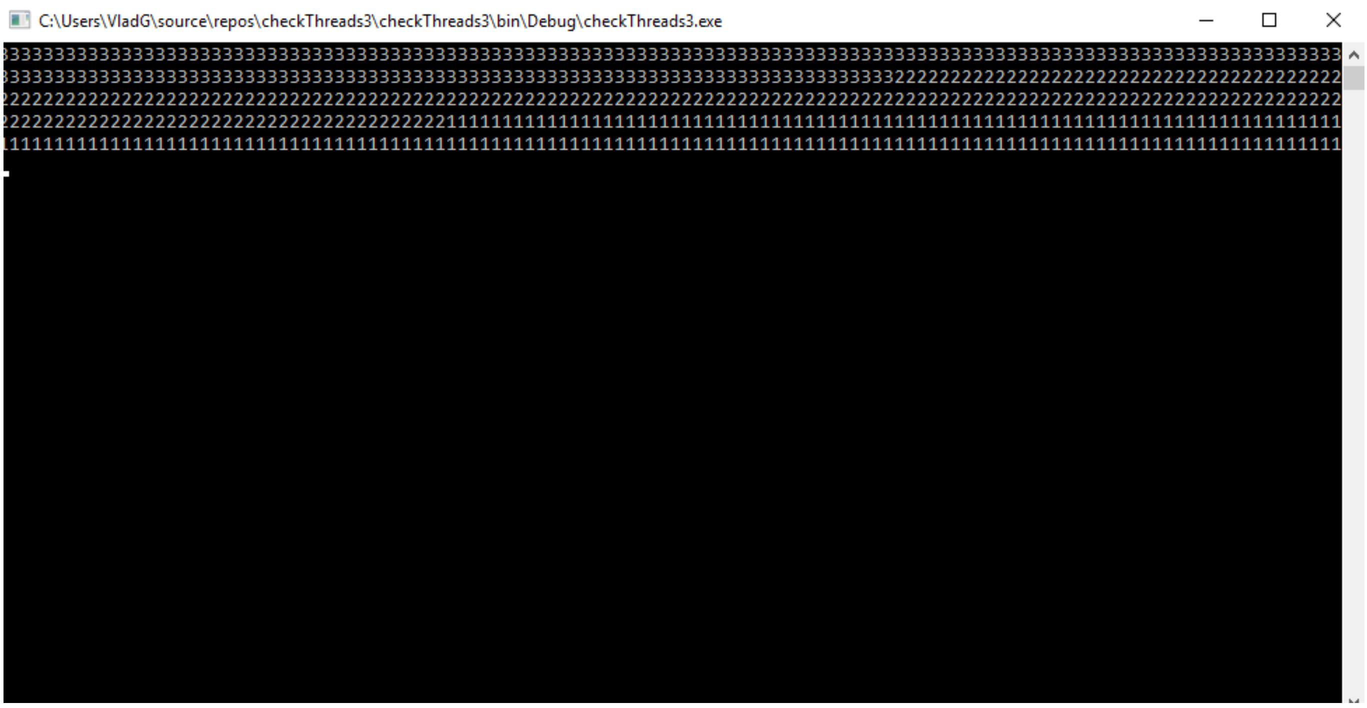
- The maximum namber of simultanesly running processes;
- The total number of threads.

This method really will prevent an exchaterating number of running at the same time processes (but depends on how much did we permited).

In the context of the variant of my laboratory work it is not a good solution to use this method. Although:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace checkThreads4
{
    class Demo
    {
        static Thread[] threads = new Thread[3];

        static Semaphore semaphore = new Semaphore(1, 1);

        static void Print(int element)
        {
            semaphore.WaitOne();
            for (int i = 0; i < 200; i++)
            {
                Console.Write(element);
            }
            semaphore.Release();
        }

        static void Main(string[] args)
        {
            for (int j = 3; j-- > 0; )
            {
                int tmp = j + 1;
                threads[j] = new Thread(() => { Print(tmp); });
            }
            while(!threads[2].IsAlive)
            {
                threads[2].Start();
            }
            for(int i = 0; i < 2; i++)
            {
                threads[i].Start();
            }
            Console.Read();
        }
    }
}
```

To write `threads[2].Start()` doesn't say us that this thread was immediatly launched. That's why I have put an while-loop where the current state of the `thread[2]` was checked.


C:\Users\VladG\source\repos\checkThreads3\checkThreads3\bin\Debug\checkThreads3.exe

```
333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333333333333222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222211111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

The good thing in all this is that the third thread has been executed, and after that the last two were executed too. The second and after that the first - the order of execution (while I've setted the first and after second). It prouves that the semaphore works right.

In the context of this task, probably, it isn't the best solution, because the first and the second threads weren't running simultanesly.

The fourth method (propper ThreadManager)

It's quite clear from the name of the method that I have to write a ThreadManager. The idea is that I will need an object of this class and should call the method Start().

I'll achieve the expected result because inside *ThreadManager* class will be a Dependency matrix which will ilustrate the current state of each thread. Thus, for example, if the threads ***i*** has been terminated, in this matrix for each thread will be indicated that it they doesn't depend more on the ***i***-*th* one.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace _ThreadManager_
{
    class ThreadManager
    {
        private int nThreads;
        private bool[ , ] DependencyMatrix;
        Thread[] threads;

        static void Print(int element)
        {
            for (int i = 0; i < 200; i++)
            {
                Console.Write(element);
            }
        }

        public ThreadManager(int nThreads = 3)
        {
            this.nThreads = nThreads;
            DependencyMatrix = new bool[ , ]  { { false, false, true},
                                                { false, false, true},
                                                { false, false, false} };

            threads = new Thread[nThreads];
            for(int i = 0; i < nThreads; i++)
            {
                int tmp = i + 1;
                threads[i] = new Thread(() =>
                {
                    Print(tmp);
                    for (int ind = 0; ind < nThreads; ind++)
                        DependencyMatrix[ind, tmp - 1] = false;
                });
            }
        }
        private bool CanGo(int index)
        {
            for(int i = 0; i < nThreads; i++)
            {
                if (DependencyMatrix[index, i]) return false;
            }
            return true;
        }

        public void Start()
        {
            bool Execute = true;
            while (Execute)
            {
                Execute = false;
                for (int i = 0; i < nThreads; i++)
                {
                    if (CanGo(i))
                    {
                        if (threads[i].ThreadState == ThreadState.Stopped ||
threads[i].ThreadState == ThreadState.Running) continue;
                        threads[i].Start();
                    }
                    else Execute = true;
                }
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ThreadManager manager = new ThreadManager();
            manager.Start();
            Console.ReadKey();
        }
    }
}
```

The logic is quite simple and in order to not run already executing or finished thread, inside the loop which iterates state of each thread I have put checking this situation using *ThreadState* class.

What are disadvantages?

- First of all, in this example can be low performance during processing the threads (in comparison with another methods which I have implemented yet). It could be explained that in my example the graph (graph of dependencies is too rare/sparse).

- The solution is written in a short form, but it is complex in understanding what exactly is happenning right now in the graph. I have said this, because could be a situation when in the graph the thread depends on itself (it's clear that this thing is impossible in real life, but the programmer may introduce this error puttind in DependencyMatrix[i][i] value **true**).

Anyway the performance of this method may be increased - all we need is to use the array of adjacency lists where we will keep only vertices on which the current one depends. Another one solution is to use Fibonacci heap and other different techniques.

The fifth method (AutoResetEvent)

The main idea of this method (in the context of my example) is that I should initialize three threads: inside first two I should call method *WaitOne()* of one of the objects of **AutoResetEvent** class and in the third one I'll call *Set()* method. The AutoResetEvent.Set() submits the signal, but AutoResetEvent.WaitOne() iwaiting for corresponding signal. Thus it's easy to control precisely all the processes (their order of execution).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace _checkThreads4
{
    class Program
    {
        static EventWaitHandle autoFirstWaitsThird = new AutoResetEvent(false);
        static EventWaitHandle autoSecondWaitsThird = new AutoResetEvent(false);

        static void Print(int val)
        {
            for (int i = 0; i < 200; i++)
            {
                Console.Write(val);
            }
        }

        static void Main(string[] args)
        {
            new Thread(_ => {
                autoFirstWaitsThird.WaitOne();
                Print(1);
            }).Start();

            new Thread(_ => {
                autoSecondWaitsThird.WaitOne();
                Print(2);
            }).Start();

            new Thread(_ => {
                Print(3);
                autoFirstWaitsThird.Set();
                autoSecondWaitsThird.Set();
            }).Start();

            Console.ReadKey();
        }
    }
}
```

This is probably the easiest example which can be implemented. Everything is clear not only on intuitive level of understanding.

The *autoFirstWaitsThird* object could be initialized also in the following manner: `var autoFirstWaitsThird = new EventWaitHandle (false, EventResetMode.AutoReset);`.

The sixth method (ManualResetEvent)

The NanualResetEvent can also be constructed in two ways. The fact is that this method works faster than the previous one. The main difference is that calling the *Set()* method opens the gate, allowing any number of threads calling *WaitOne()* to be let through. But calling *Reset()* closes the gate, thus all the threads which previously have called *WaitOne()* method will be released.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace _checkThreads5
{
    class Program
    {
        static EventWaitHandle manualWaitThird = new ManualResetEvent(false);

        static void Print(int val)
        {
            for (int i = 0; i < 200; i++)
            {
                Console.Write(val);
            }
        }

        static void Main(string[] args)
        {
            new Thread(_ => {
                manualWaitThird.WaitOne();
                Print(1);
            }).Start();

            new Thread(_ => {
                manualWaitThird.WaitOne();
                Print(2);
            }).Start();

            new Thread(_ => {
                Print(3);
                manualWaitThird.Set();
            }).Start();

            Console.ReadKey();
        }
    }
}
```

As it was seen a ***ManualResetEvent*** is useful in allowing one thread to unblock many other threads.

A ***ManualResetEvent*** is useful in allowing one thread to unblock many other threads. The reverse scenario is covered by **CountdownEvent**.

The fact: ManualResetEventSlim and CountdownEvent can be up to 50 times faster in short-wait scenarios, because of their nonreliance on the operating system and judicious use of spinning constructs.

Conclusion

I have described a lot of different techniques with its various aspects. Which of them to use? It depends on the concrete task and on programmer's vision of the problem. The main thing to remember is that during implementation of these concurrency parallelism may appear the following problems: deadlocking, reentrancy, and emasculated concurrency.

Sometimes they are very hard to fix. That's why we should have concrete plan where are described all threads (maybe graphically). More precisely to say: Their dependences on each other.