MCTE 4327

Software Engineering

Week 08 Asynchronous Software Engineering

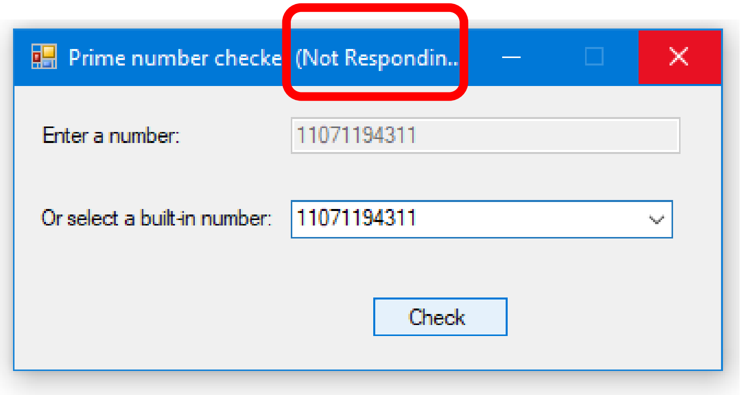
Outline

- Threads
- Tasks
- Asynchronous function calls

Motivation for asynchronous processing

- Building a responsive user interface

  Doing heavy operations in UI events make the UI non-responsive.

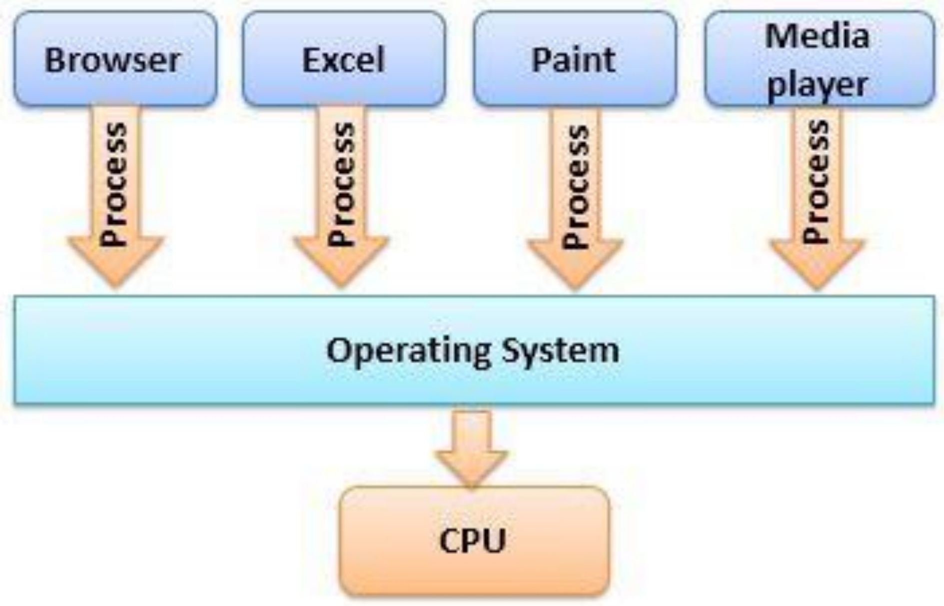


- Allowing requests to process simultaneously

  Example: web servers need to serve many requests simultaneously

Multi-tasking

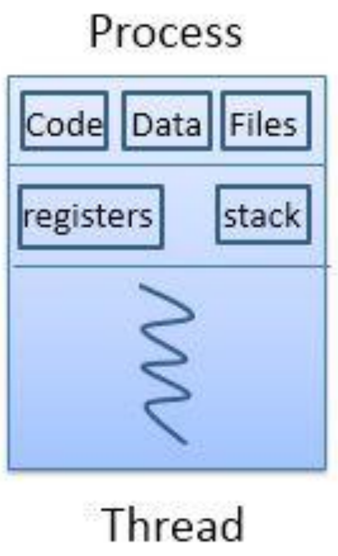
- An OS allows multiple programs or processes to be run concurrently.
- On a single-core computer, the OS must allocate “slices” of time for each process (e.g. 20ms in Windows) to simulated concurrency.
- On a multi-core or multiple-processor machine, multiple processes can genuinely be executed in parallel.



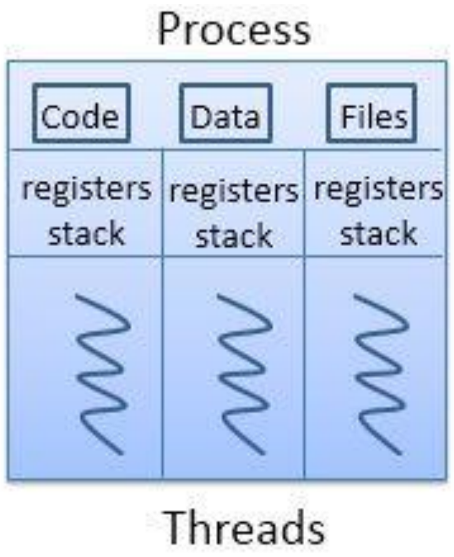
Multitasking

Multi-threading

- A thread is a basic execution unit which has its own program counter, set of the register, stack but it shares the code, data, and file of the process to which it belongs.
- Within each program/process, there can be multiple concurrent threads.
- The CPU switches among these threads so frequently making an impression on the user that all threads are running simultaneously and this is called multithreading.



Single-threading



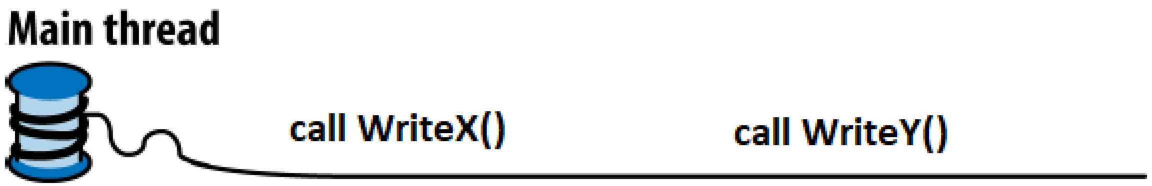
Multi-threading

Creating a thread

- By default, a program starts in a single thread that is created automatically by the OS. It is called the “main” thread.
- Here the application lives out its life as a single-threaded application, unless you do otherwise by creating more threads (directly or indirectly).
- A new thread can be created by instantiating a Thread object (from System.Threading namespace) and calling its ‘Start’ method.

Single-threaded application

```
static void Main()
{
    WriteX();
    WriteY();

    Console.Read();
}

static void WriteX()
{
    for (int i=0; i<1000; i++)
    {
        Console.Write("x");
    }
}
static void WriteY()
{
    for (int i = 0; i < 1000; i++)
    {
        Console.Write("y");
    }
}
```



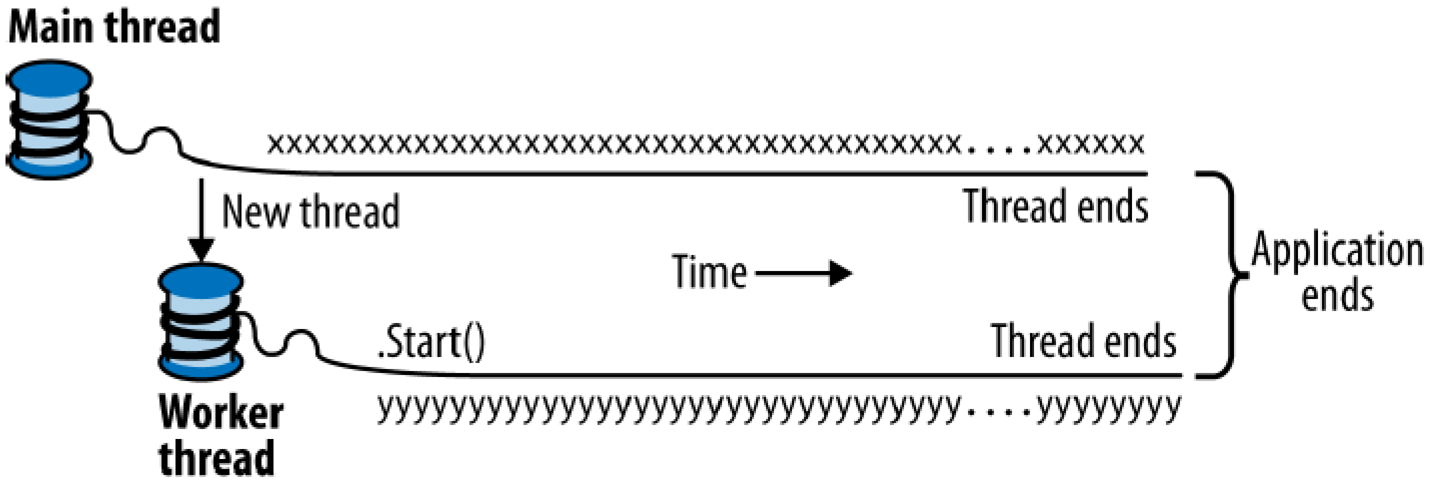
Output:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyy
```

```csharp
static void Main()
{
    Thread thread = new Thread(WriteY);
    thread.Start();

    WriteX();

    Console.Read();
}
```

Or

```csharp
static void Main()
{
    new Thread(WriteY).Start();
    WriteX();

    Console.Read();
}
```




```
// Typical Output:
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Lambda expression can also be used to create threads.

```
Thread t = new Thread(() => { for (int i = 0; i < 1000; i++) Console.Write("x");});
t.Start();
```

Or

```
new Thread(() => { for (int i = 0; i < 1000; i++) Console.Write("x");}).Start();
```

- On a single-core computer, the OS must allocate "slices" of time for each process (e.g. 20ms in Windows) to simulated concurrency.
- On a multi-core or multiple-processor machine, multiple threads can genuinely be executed in parallel.
- The output is non-deterministic because it depends on the scheduling algorithm of the OS.

Join

You can wait for another thread to end by calling its ‘Join’ method.

In the example below, the output is the same as using the single-threaded approach.

```
static void Main()
{
    Thread thread = new Thread(WriteY);
    thread.Start();
    thread.Join(); //The main thread will be blocked until the child thread ends
    WriteX();

    Console.Read();
}
```

Sleep

- Thread.Sleep function pauses the current thread of a specified number of milliseconds

```
Thread.Sleep(500);                    // Sleep for 500 milliseconds
Thread.Sleep(TimeSpan.FromHours(1)); // Sleep for 1 hour
```

- While waiting on a Sleep or Join, a thread is ‘**blocked**’. It relinquishes the thread’s current time slice immediately, voluntarily handing over the CPU to other threads.

- When a thread blocks or unblocks, the OS performs a context switch. This incurs a small overhead, typically one or two microseconds.

Sharing variables among threads

- Each thread has its own memory stack so that local variables are kept separate.
- The code below creates two threads that use Go(). Each thread has separate local variable i. The output is, preditablely, 10 question marks.

```
static void Main()
{
    new Thread(Go).Start();
    new Thread(Go).Start();

    Console.Read();
}

static void Go()
{
    for (int i=0; i<5; i++)
    {
        Console.Write("?");
    }
}
```

Output:

```
??????????
```

Sharing variables among threads

```
class Program
{
    static bool done;

    static void Main()
    {
        new Thread(Go).Start();
        new Thread(Go).Start();

        Console.Read();
    }

    static void Go()
    {
        if (!done)
        {
            done = true;
            Console.WriteLine("Done");
        }
    }
}
```

- Here two threads share a common variable called “done”
- This results in “Done” being printed once instead of twice (or is it?)
- Output is actually indeterminate: it is possible that (though unlikely) that “Done” could be printed twice if the two threads are perfectly parallel.

Output:

```
Done
```

Sharing variables among threads

```
class Program
{
    static bool done;

    static void Main()
    {
        new Thread(Go).Start();
        new Thread(Go).Start();

        Console.Read();
    }

    static void Go()
    {
        if (!done)
        {
            Console.WriteLine("Done");
            done = true;
        }
    }
}
```

- When we swap the order of statements in the Go method, the odds of “Done” being printed twice go up dramatically.
- The problem is that one thread can be evaluating the if statement right at the moment the other thread is executing the WriteLine statement – before it had a chance to set done to true.

Output:

```
Done
Done
```

Thread safety

- We can fix the previous example by obtaining an exclusive lock while reading and writing to the shared field.
- When two threads simultaneously contend a lock (which can be upon any reference-type object, in this case, locker of object type), one thread waits, or blocks, until the lock becomes available.
- A method or a class is called ‘**thread-safe**’ if it can be accessed from multiple threads without any potential problem.

```
class Program
{
    static bool done;
    static readonly object locker = new object();

    static void Main()
    {
        new Thread(Go).Start();
        new Thread(Go).Start();
        Console.Read();
    }

    static void Go()
    {
        lock (locker)
        {
            if (!done)
            {
                Console.WriteLine("Done");
                done = true;
            }
        }
    }
}
```

Example: a class that is not thread-safe

- This class on the left is not thread-safe
- If Go was called by two threads simultaneously, it would be possible to get a division-by-zero error, because b could be set to zero in one thread right as the other thread was in between executing the if statement and Console.WriteLine.
- It can be rectified to make it 'thread-safe'.

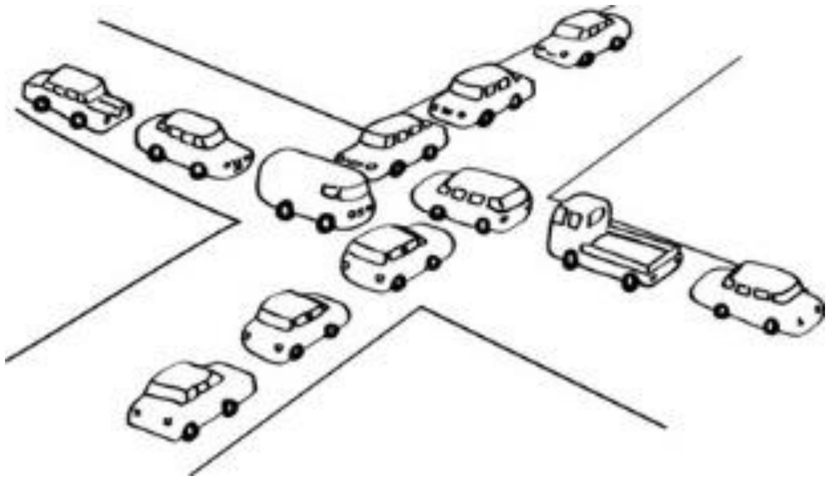
```
class ThreadUnsafe
{
    static int a = 1, b = 1;
    static void Go()
    {
        if (b != 0)
        {
            Console.WriteLine(a / b);
        }
        b = 0;
    }
}
```

```
class ThreadSafe
{
    static int a = 1, b = 1;
    static readonly object locker = new object();
    static void Go()
    {
        lock (locker)
        {
            if (b != 0)
            {
                Console.WriteLine(a / b);
            }
            b = 0;
        }
    }
}
```

Deadlock

A deadlock happens when two threads each wait for a resource held by the other, so neither can proceed. The easiest way to illustrate this is with two locks:

```
object locker1 = new object();
object locker2 = new object();

new Thread(() =>
{
    lock (locker1)
    {
        Thread.Sleep(1000);
        lock (locker2) ; // Deadlock
    }
}).Start();

lock (locker2)
{
    Thread.Sleep(1000);
    lock (locker1) ; // Deadlock
}
```



Sending signals among threads

A thread can wait for a signal from another thread (using AutoResetEvent)

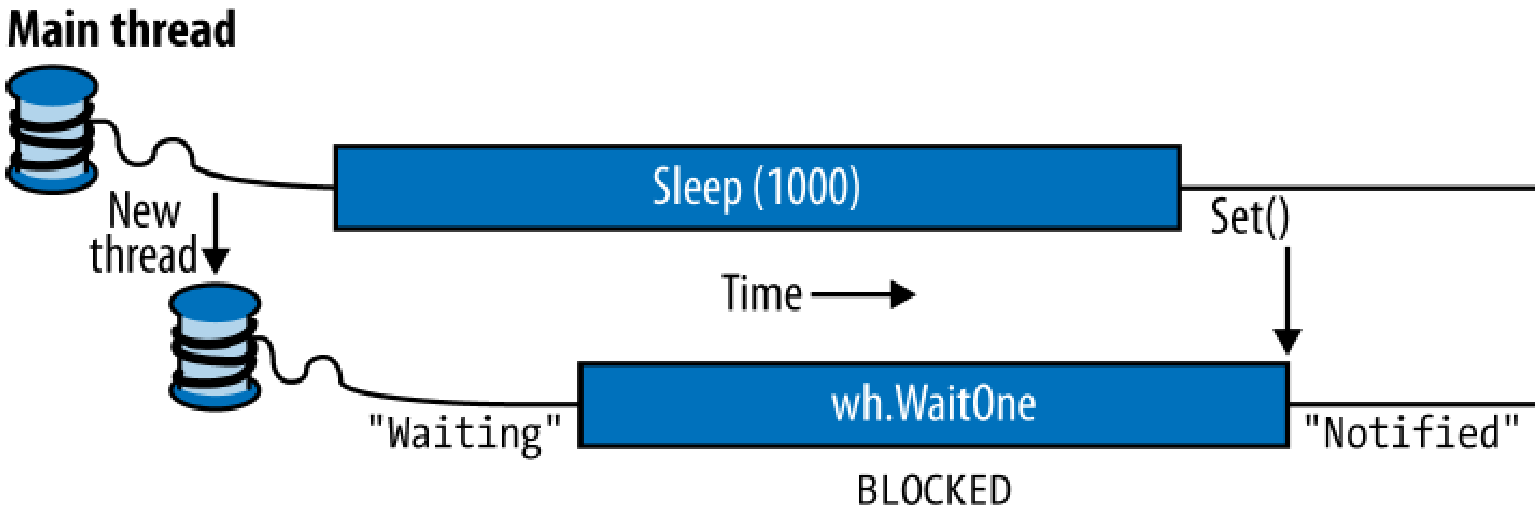
```
static AutoResetEvent signal = new AutoResetEvent(false);

static void Main()
{
    new Thread(Waiter).Start();
    Thread.Sleep(1000); // Pause for a second...
    signal.Set(); // Wake up the Waiter.
}
static void Waiter()
{
    Console.WriteLine("Waiting...");
    signal.WaitOne(); // Wait for notification
    Console.WriteLine("Notified");

    Console.Read();
}
```

Initial state is “off”

The new thread is waiting for a signal from the main thread.

While waiting, the new thread is “blocked”. The CPU gives away “time slices” to other processes and threads.

The main thread sends the signal to the new thread after 1 second, the new thread resumes.



Exception handling

- Any try/catch/finally blocks in effect when a thread is created are of no relevance to the thread when it starts executing. Consider the following program:

```
public static void Main()
{
    try
    {
        new Thread(Go).Start();
    }
    catch
    {
        // We'll never get here!
        Console.WriteLine(“Error");
    }
}

static void Go() { throw null; } //Intentionally throws an exception
```

- The try/catch statement in this example is ineffective, and the newly created thread will be encumbered with an unhandled NullReferenceException. This behavior makes sense when you consider that each thread has an independent execution path.

Exception

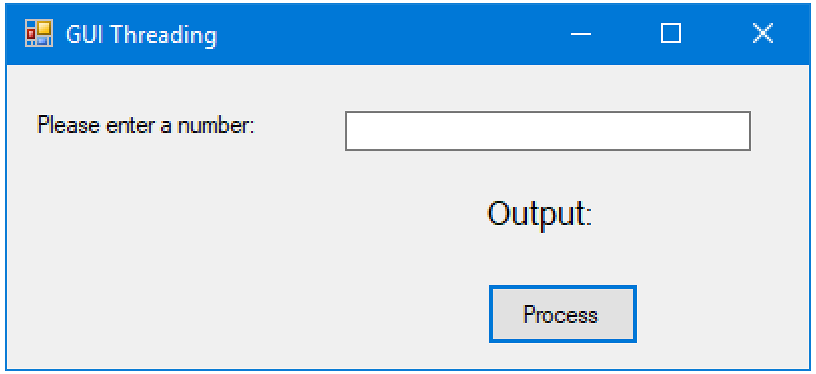
- The remedy is to move the exception handler into the Go method:

```
public static void Main()
{
    new Thread(Go).Start();
}
static void Go()
{
    try
    {
        throw null; //Intentionally throws an exception
    }
    catch
    {
        //The exception will be caught here
    }
}
```

Threading in GUI-based applications

- Consider an application that accepts a number as input, performs a heavy operation and produces an output.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += Button1_Click;
    }

    private void Button1_Click(object sender, EventArgs e)
    {
        label2.Text = "Processing...";

        Thread.Sleep(5000); //Simulate heavy processing

        label2.Text = "Answer = 53"; //Dummy answer
    }
}
```



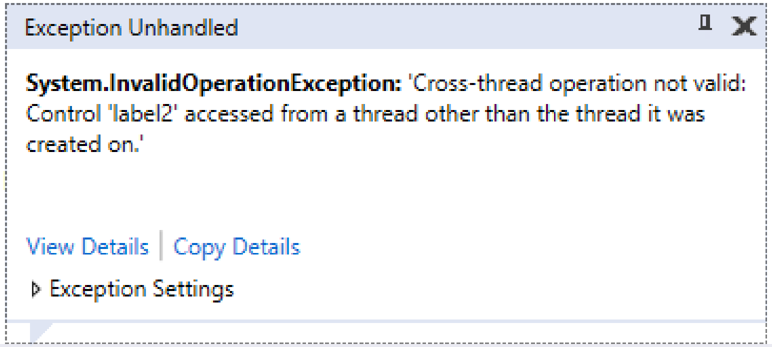
Observations

- The GUI freezes during the operation.
- The text “Processing…” is not seen.

Simple threading solution

But the problem is that properties of UI controls/components can only be modified from the main thread.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

        button1.Click += Button1_Click;
    }

    1 reference
    private void Button1_Click(object sende
    {
        new Thread(() =>
        {
            label2.Text = "Processing...";
            Thread.Sleep(5000); //Simulate heavy processing
            label2.Text = "Answer = 53"; //Dummy answer
        }).Start();
    }
}
```



- UI elements and controls can be accessed only from the thread that created them (typically the main UI thread).
- Hence when you want to update the UI from another thread, you must forward the request to the UI thread (the technical term is *marshal*).
- UI elements can be ‘marshalled’ through Invoke or BeginInvoke methods of the UI elements. (BeginInvoke is asynchronous. It does not wait for the update to finish).
- Invoke/BeginInvoke work by enqueuing the delegate to the UI thread’s message queue (the same queue that handles keyboard, mouse, and timer events).

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += Button1_Click;
    }

    private void Button1_Click(object sender, EventArgs e)
    {
        button1.Enabled = false; //Stops user from pressing multiple times
        label2.Text = "Processing...";

        new Thread(() =>
        {
            Thread.Sleep(5000); //Simulate heavy processing
            label2.BeginInvoke(new Action(() => label2.Text = "Answer = 53"));
            button1.BeginInvoke(new Action(() => button1.Enabled = true));

        }).Start();
    }
}
```

Observations

It works but if the user closes the form while processing, the application does not end.

Foreground vs Background Threads

- By default, threads you create explicitly are *foreground threads*.
- Foreground threads. keep the application alive for as long as any one of them is running, whereas *background threads* do not.
- Once all foreground threads finish, the application ends, and any background threads still running abruptly terminate.

```csharp
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += Button1_Click;
    }
    private void Button1_Click(object sender, EventArgs e)
    {
        button1.Enabled = false; //Stops user from pressing multiple times
        label2.Text = "Processing...";
        Thread t = new Thread(() =>
        {
            Thread.Sleep(5000); //Simulate heavy processing
            label2.BeginInvoke(new Action(() => label2.Text = "Answer = 53"));
            button1.BeginInvoke(new Action(() => button1.Enabled = true));

        });
        t.IsBackground = true; //Sets as a background thread
        t.Start();
    }
}
```

Aborting threads

- A running thread can be aborted by calling its Abort() method.

```
Thread t = new Thread(MyFunction);
t.Start();


t.Abort(); //Abort it
```

Thread Pool

- Whenever you start a thread, a few hundred microseconds are spent organizing such things as a fresh local variable stack.
- The *thread pool* cuts this overhead by having a pool of pre-created recyclable threads.
- Thread pooling is essential for efficient parallel programming and fine-grained concurrency; it allows short operations to run without being overwhelmed with the overhead of thread startup.

Entering a thread pool

- The easiest way to explicitly run something on a pooled thread is to use Task.Run.
- The Task class is stored in the *System.Threading.Tasks* namespace.

```
Task.Run(() => Console.WriteLine("Hello from the thread pool"));
```

Wait

Calling Wait on a task blocks until it completes and is the equivalent of calling Join on a thread.

```
Task task = Task.Run(() =>
{
    Thread.Sleep(2000);
    Console.WriteLine("Foo");
});
Console.WriteLine(task.IsCompleted); // False
task.Wait(); // Blocks until task is complete
```

Returning values

- A thread cannot return values. But a task can return values.

```
public static void Main()
{
    Task<int> task = Task.Run(() =>
    {
        Console.WriteLine("Foo");
        return 3;
    });

    int result = task.Result; // Blocks if not already finished
    Console.WriteLine(result); // 3
}
```

OnCompleted

- The OnCompleted method accepts a method to be executed then the task is completed.

```
private void Button1_Click(object sender, EventArgs e)
{

    button1.Enabled = false; //Stops user from pressing multiple times
    label2.Text = "Processing...";

    Task<int> task = Task.Run(() =>
    {
        Thread.Sleep(5000); //Simulate heavy processing
        return 5; //Dummy value;

    });

    task.GetAwaiter().OnCompleted(() =>
    {
        label2.Text = task.Result.ToString();
        button1.Enabled = true;
    });

}
```

Task.Delay

The Task.Delay method is equivalent to Thread.Sleep.

```
Task.Delay(5000); //Delay for 5 seconds
Console.WriteLine("Hi");
```

Or

```
Task.Delay(5000).GetAwaiter().OnCompleted(() => Console.WriteLine("Hi"));
```

Synchronous vs Asynchronous operations

- A *synchronous operation* does its work *before* returning to the caller.
- An *asynchronous operation* does (most or all of) its work *after* returning to the caller.

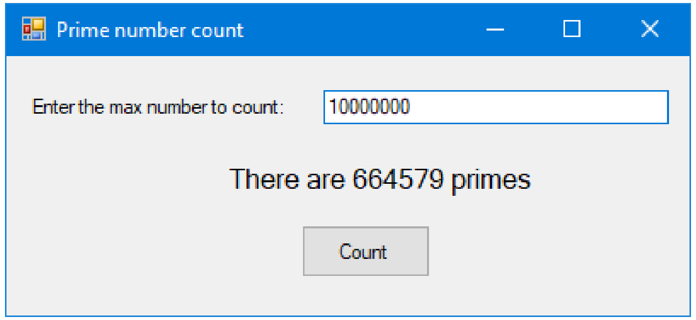
- The principle of asynchronous programming is that you write long-running (or potentially long-running) functions asynchronously.

Example

- The following functions count how many prime numbers exist from 0 to the number specified in the input.
- The first one is synchronous while the second one is asynchronous.
- An asynchronous function returns a Task.

```
int GetPrimesCount(int max) //Synchronous function
{
    return ParallelEnumerable.Range(2, max).Count(n =>
        Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0));
}


Task<int> GetPrimesCountAsync(int max) //Asynchronous function
{
    return Task.Run(() => ParallelEnumerable.Range(2, max).Count(n =>
        Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)));
}
```

```
private void Button1_Click(object sender, EventArgs e)
{

    button1.Enabled = false; //Stops user from pressing multiple times
    label2.Text = "Processing...";
    int number = int.Parse(textBox1.Text); //For simplicity, ignore input validation

    var awaiter = GetPrimesCountAsync(number).GetAwaiter();

    awaiter.OnCompleted(()=>
    {
        label2.Text = "There are " + awaiter.GetResult().ToString() + " primes";
        button1.Enabled = true;
    });

}
```



async and await keywords

- C# 5.0 introduced the async and await keywords.
- These keywords let you write asynchronous code that has the same structure and simplicity as synchronous code and eliminates the “plumbing” of asynchronous programming.

```
var result = await expression;
statement(s);
```

Is equivalent to

```
var awaiter = expression.GetAwaiter();
awaiter.OnCompleted (() =>
{
  var result = awaiter.GetResult();
  statement(s);
});
```

Application: Raspberry PI

- Unlike Arduino, Raspberry Pi needs multiple assembly instructions to turn on/off GPIOs.
- Need to deal with registers.
- There are many libraries to access GPIOs on Raspberry PI.

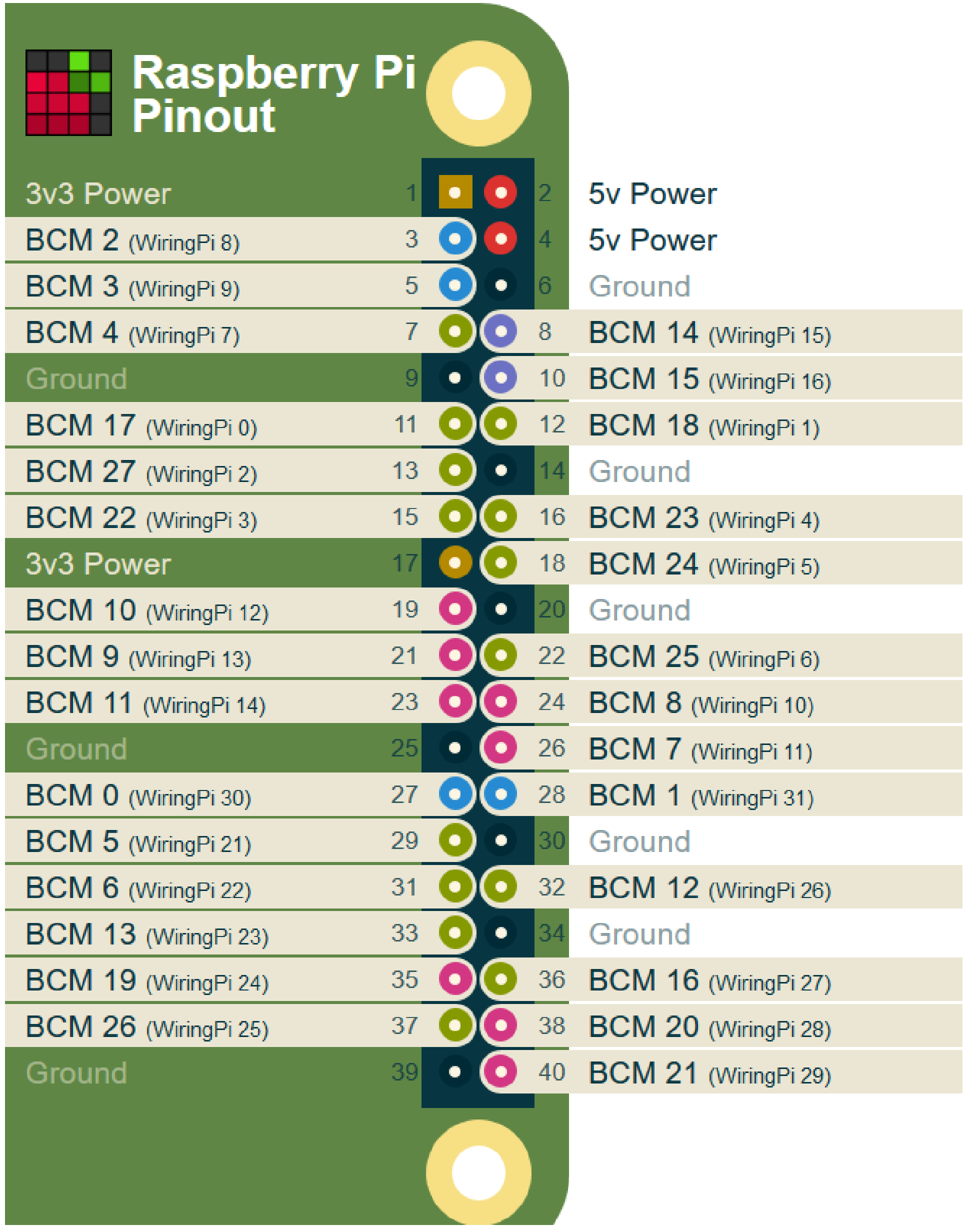
Language	Library	Square wave
Shell	/proc/mem access	2.8 kHz
Python	RPi.GPIO	70 kHz
C	Native library	22 MHz
C	BCM 2835	5.4 MHz
C	wiringPi	4.1 – 4.6 MHz

http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/

Wiring PI library

- WiringPi library is written in C. It uses functions similar to that of the famous Arduino library. For example: digitalRead, digitalWrite, pinMode, delay, millis

```
#include <wiringPi.h>
int main(void)
{
    wiringPiSetup();
    pinMode(0, OUTPUT);
    for (; ; )
    {
        digitalWrite(0, HIGH);
        delay(500);
        digitalWrite(0, LOW);
        delay(500);
    }
    return 0;
}
```

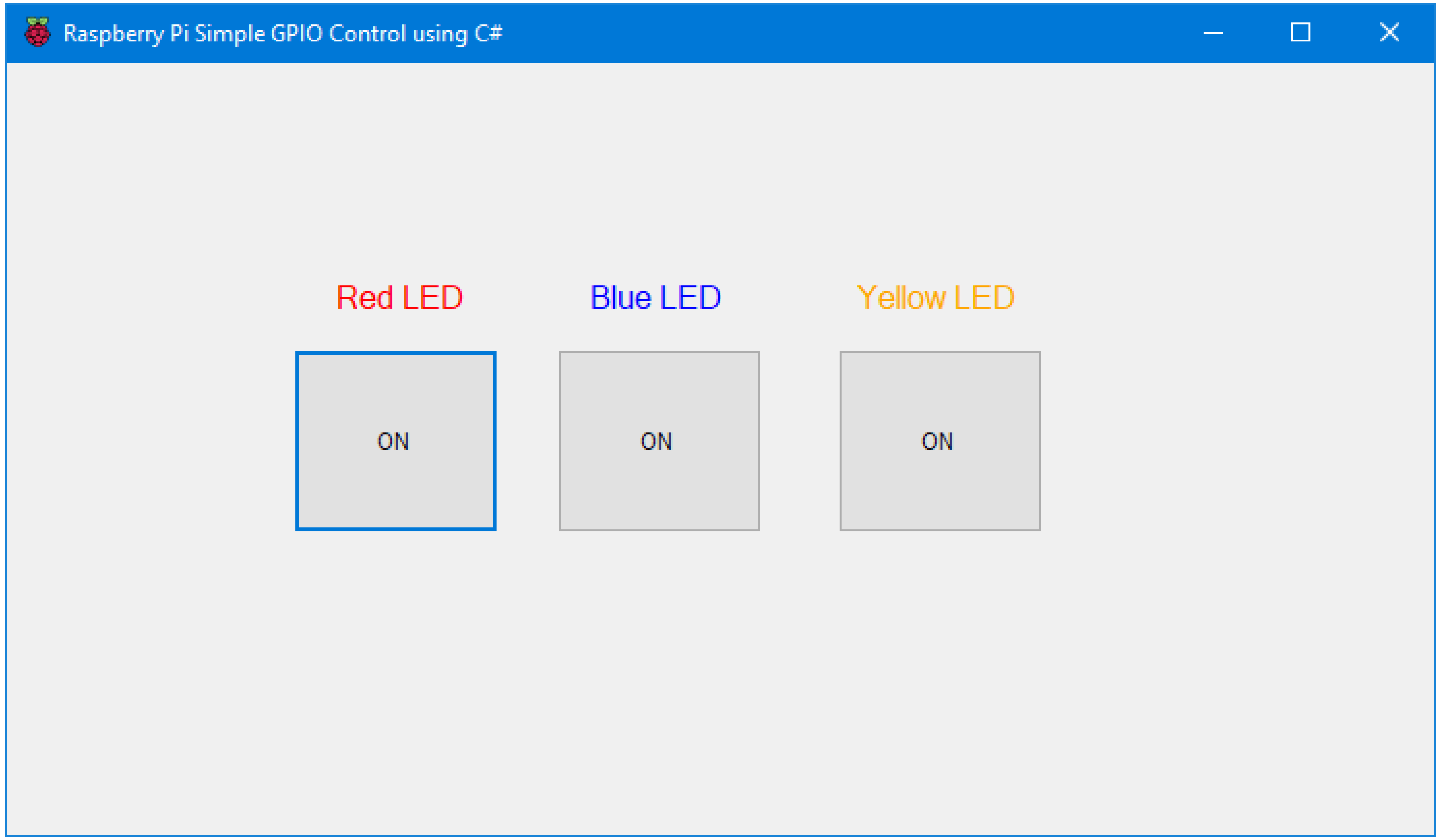


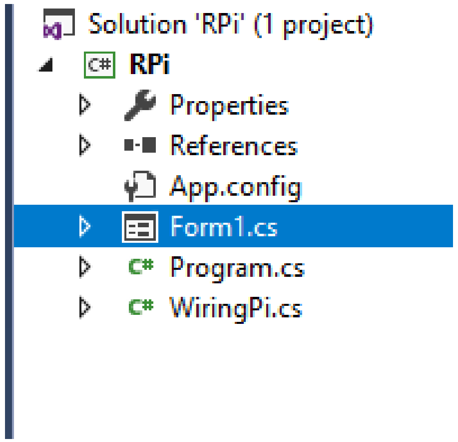
WiringPI pin numbers are different from BCM pin numbers

Calling C/C++ functions from C#

- WiringPI can be compiled as a shared object named “libwiringPi.so”
- We can defined external functions on C# that point to the shared object.

```
[DllImport("libwiringPi.so", EntryPoint = "pinMode")]
public static extern void pinMode(int pin, int mode);

[DllImport("libwiringPi.so", EntryPoint = "digitalWrite")]
public static extern void digitalWrite(int pin, int value);

[DllImport("libwiringPi.so", EntryPoint = "digitalRead")]
public static extern int digitalRead(int pin);
```


Raspberry Pi Simple GPIO Control using C#
Red LED
Blue LED
Yellow LED
ON
ON
ON


Solution 'RPi' (1 project)
RPi
Properties
References
App.config
Form1.cs
Program.cs
WiringPi.cs

WiringPI.cs

```
using System.Runtime.InteropServices;
namespace WiringPi
{

    public class GPIO
    {
        [DllImport("libwiringPi.so", EntryPoint = "wiringPiSetup")]
        public static extern int WiringPiSetup();

        [DllImport("libwiringPi.so", EntryPoint = "pinMode")]
        public static extern void pinMode(int pin, int mode);

        [DllImport("libwiringPi.so", EntryPoint = "digitalWrite")]
        public static extern void digitalWrite(int pin, int value);

        [DllImport("libwiringPi.so", EntryPoint = "softPwmCreate")]
        public static extern int softPwmCreate(int pin, int initialValue, int pwmRange);

        [DllImport("libwiringPi.so", EntryPoint = "softPwmWrite")]
        public static extern void softPWMWrite(int pin, int value);
    }

}
```

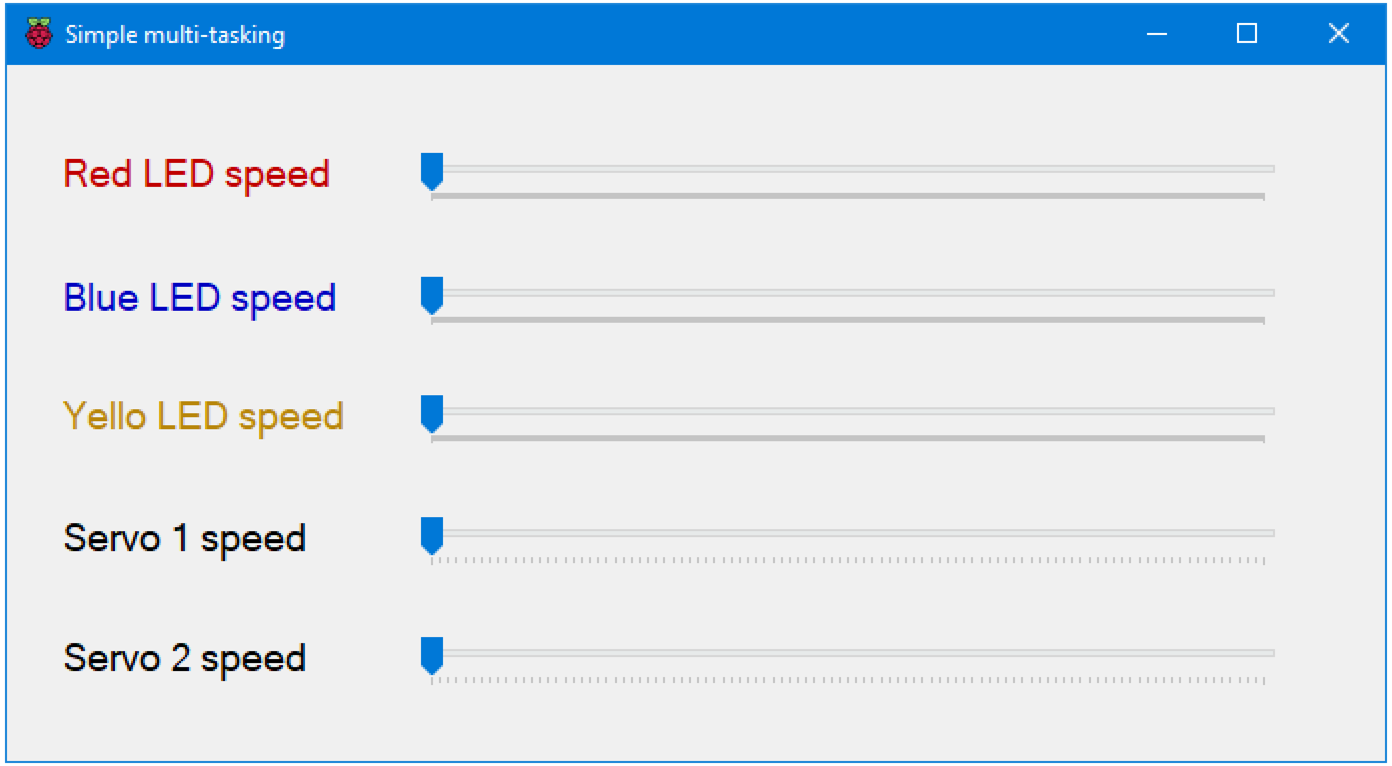
Form1.cs

```
namespace RPi
{
    public partial class Form1 : Form
    {
        private const int OUTPUT = 1;
        private const int INPUT = 0;
        private const int HIGH = 1;
        private const int LOW = 0;

        private readonly object locker = new object();

        private bool Setup()
        {
            try
            {
                lock(locker) WiringPi.GPIO.WiringPiSetup(); //Make it thread-safe
                return true;
            }
            catch
            {
                return false;
            }
        }
        private bool pinMode(int pin, int value)
        {
            try
            {
                lock (locker) WiringPi.GPIO.pinMode(pin, value); //Make it thread-safe
                return true;
            }
            catch
            {
                return false;
            }
        }
        private bool digitalWrite(int pin, int value)
        {
            try
            {
                lock (locker) WiringPi.GPIO.digitalWrite(pin, value); //Make it thread-safe
                return true;
            }
            catch
            {
                return false;
            }
        }
```

```
public Form1()
{
    InitializeComponent();
    Setup();

    pinMode(8, OUTPUT);
    pinMode(9, OUTPUT);
    pinMode(7, OUTPUT);

    digitalWrite(8, LOW);
    digitalWrite(9, LOW);
    digitalWrite(7, LOW);
}

private void btnRed_Click(object sender, EventArgs e)
{
    if (btnRed.Text == "ON")
    {
        btnRed.Text = "OFF";
        btnRed.BackColor = lblRed.ForeColor;
        digitalWrite(8, HIGH);
    }
    else
    {
        btnRed.Text = "ON";
        btnRed.BackColor = BackColor;
        digitalWrite(8, LOW);
    }
}
```

```
private void btnBlue_Click(object sender, EventArgs e)
{
    if (btnBlue.Text == "ON")
    {
        btnBlue.Text = "OFF";
        btnBlue.BackColor = lblBlue.ForeColor;
        digitalWrite(9, HIGH);
    }
    else
    {
        btnBlue.Text = "ON";
        btnBlue.BackColor = BackColor;
        digitalWrite(9, LOW);
    }
}

private void btnYellow_Click(object sender, EventArgs e)
{
    if (btnYellow.Text == "ON")
    {
        btnYellow.Text = "OFF";
        btnYellow.BackColor = lblYellow.ForeColor;
        digitalWrite(7, HIGH);
    }
    else
    {
        btnYellow.Text = "ON";
        btnYellow.BackColor = BackColor;
        digitalWrite(7, LOW);
    }
}
```

Example 2


Simple multi-tasking
Red LED speed
Blue LED speed
Yello LED speed
Servo 1 speed
Servo 2 speed