Working with Threads

Basics

- The threads of a process all use the same address space and share all resources.
- Each thread has its own stack.
- Each thread can have specific properties such as a unique name, priority, etc.
- Threads can share objects, but must be sure to avoid the problem of race conditions (mutual exclusion) by using the synchronization and mutual exclusion features of the .NET FCL.

Basics - Contd.

- Not all of the .NET FCL is “thread safe.”
- This means that two or more processes can not invoke properties or methods of the same instance of such an object at the same time.
- Locking must be employed.
- The FCL documentation specifically addresses the thread safety of the class.

An Example of why Threads are Needed

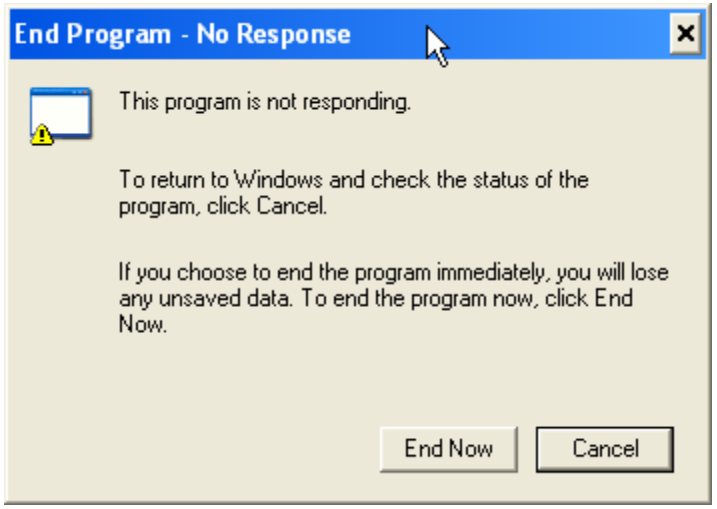
```
No Response - Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace NoResponse
{
    public partial class Form1 : Form
    {
```

Example – Contd.

```
        public Form1()
        {
            InitializeComponent();
        }
        private void Form1_MouseClick(object sender,
                MouseEventArgs e)
        {
            Cursor = Cursors.WaitCursor;
            //Loop for approximately 30 seconds (depends on
            //system speed)
            long i = 3000000000;
            while (--i>0) ;
            Cursor = Cursors.Default;
        }
    }
}
```

Ooops! The GUI Doesn’t Respond

You may see a message such as this:



The Thread Class

- In System.Threading namespace.
- Useful properties:
  1. CurrentThread – returns a reference to the callers thread
  2. Name – a string representing the programmer assigned name
  3. Priority – the thread priority (may be set to a non-default value)
  4. IsAlive – true if thread is running (not terminated)

Starting a Thread

```
Void myMethod()
{
   //do something
}

//-------------------------

ThreadStart startMethod = new ThreadStart(myMethod);
Thread myThread = new Thread(startMethod);
myThread.Start();
```

ThreadStart

- The method to be started by the thread can be an instance method or a static method.
- You don’t need to worry about how the thread is able to determine the object instance (*this)*.
- Use static fields, properties, and methods to handle data shared between threads (with locks as needed).
- .Remember, there are no global variables in C#!

Keep the GUI Alive

```
Good Response - First Try
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
namespace GoodResponse
{
    public partial class Form1 : Form
```

Keep the GUI Alive

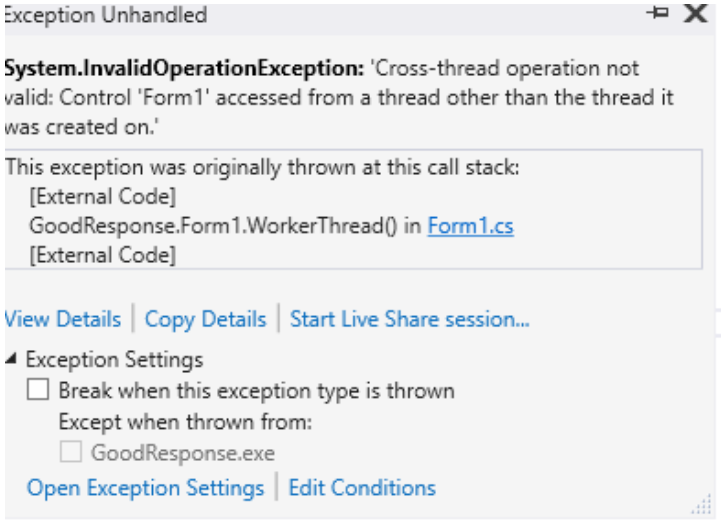
```
{
        private Thread workerThread;
        private ThreadStart workerStart;
        public Form1()
        {
            InitializeComponent();
        }
        private void Form1_MouseClick(object sender,
            MouseEventArgs e)
        {
            //Don't create more than one worker thread
            if (workerThread == null)
```

Keep the GUI Alive

```
 {
                workerStart = new ThreadStart(WorkerThread);
                workerThread = new Thread(workerStart);
                workerThread.Start();
            }
        }
        private void WorkerThread()
        {
            Cursor = Cursors.WaitCursor;
            //Loop for approximately 30 seconds (depends on
            //system speed)
            long i = 3000000000;
            while (--i > 0) ;
            Cursor = Cursors.Default;
            workerThread = null;
        }
    }
}
```

Cross Thread Problem

- If you run this program in the debugger VS is smart enough to provide the following warning shown below.
- We will fix this later.



Terminating a Thread

- A thread terminates when it returns from the method used for its start.
- A thread can be suspended by the Suspend method.
- The Resume method starts the thread running again.
- The Thread.Sleep method suspends a thread for a specific number of milliseconds. This is a static method.

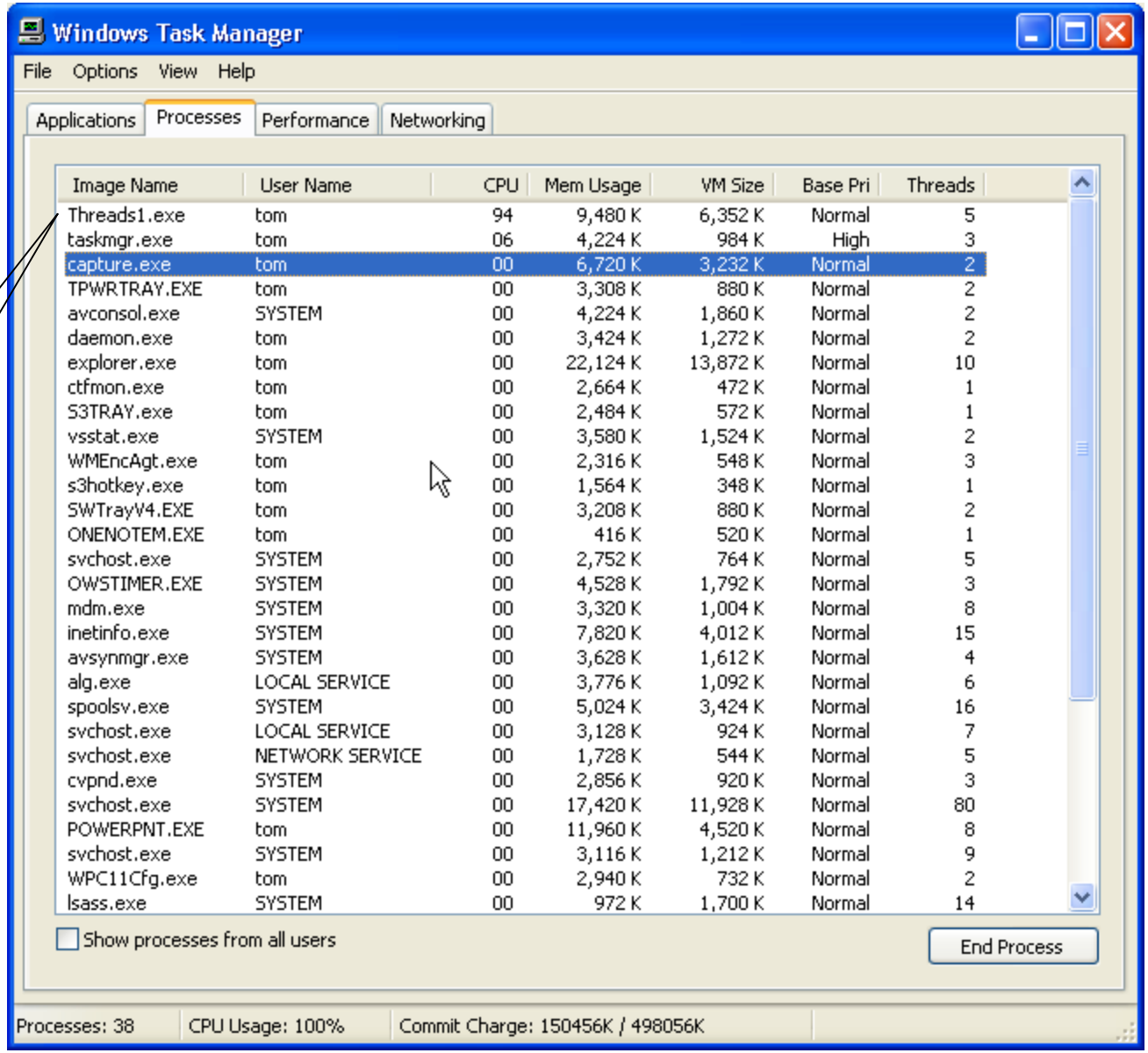
Foreground vs. Background Threads

- By default a thread is a foreground thread.
- An application doesn’t terminate until all foreground threads terminate.
- This is VERY insidious.
- Consider the following example.
- The thread is started and the main application is closed.
- It looks like the application is gone, but task manager shows it is still running and burning up CPU time.
- It doesn’t appear in the task bar and is completely invisible to the user.
- BEWARE!!!! I made this mistake. My system got slower and slower until I realized that there were many threads running.

Bad Thread

```
private void button1_Click(object sender, System.EventArgs e)
{
  if(t==null)
  {
    t = new Thread(new ThreadStart(ThreadProc));
    t.Start();
  }
  else
    MessageBox.Show("Thread running!");
}
private Thread t;
private void ThreadProc()
{
  while(true)
    ;
}
```

Thread still running



Solution

- Make the thread a background thread.
- Background threads are terminated when the foreground threads complete.

```
private void button1_Click(object sender, System.EventArgs e)
{
  if(t==null)
  {
    t = new Thread(new ThreadStart(ThreadProc));
    t.IsBackground=true;
    t.Start();
  }
  else
    MessageBox.Show("Thread running!");
}
```

Fixed up Example

```
Good Response - Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
namespace GoodResponse
{
    public partial class Form1 : Form
    {
```

Fixed up Example

```
private Thread workerThread;
private ThreadStart workerStart;
public Form1()
{
    InitializeComponent();
}
private void Form1_MouseClick(object sender,
         MouseEventArgs e)
{
    //Don't creater more than one worker thread
    if (workerThread == null)
    {
```

Fixed up Example

```
                workerStart = new ThreadStart(WorkerThread);
                workerThread = new Thread(workerStart);
                workerThread.IsBackground = true;
                workerThread.Start();
            }
        }
        private void WorkerThread()
        {
            MessageBox.Show("Worker thread started.");
            //Loop for approximately 30 seconds (depends on
            //system speed)
            long i = 3000000000;
            while (--i > 0) ;
            MessageBox.Show("Worker thread terminated.");
            workerThread = null;
        }
    }
}
```

Communicating with a Thread

- Shared memory is the easiest.
- Unfortunately we need to worry about race conditions.
- C# provides some easy to use mechanisms to solve the *mutual exclusion* problem.

Cancelling a Thread Gracefully

- Rather than aborting the thread, which has its problems, we can use a *flag* to alert the worker thread that it should abort.
- This requires inserting a check inside whatever loop is used by the worker thread.
- We must assure that this *flag* is checked frequently for good application response.
- A simple example follows.

Cancel Example

```
Cancel Thread - Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
namespace CancelThread
{
    public partial class Form1 : Form
    {
```

Cancel Example

```
private Thread workerThread;
private ThreadStart workerStart;
private bool cancel = false;
public Form1()
{
    InitializeComponent();
}
private void Form1_Click(object sender,
   EventArgs e)
{
    //Don't create more than one worker thread
    if (workerThread == null)
    {
```

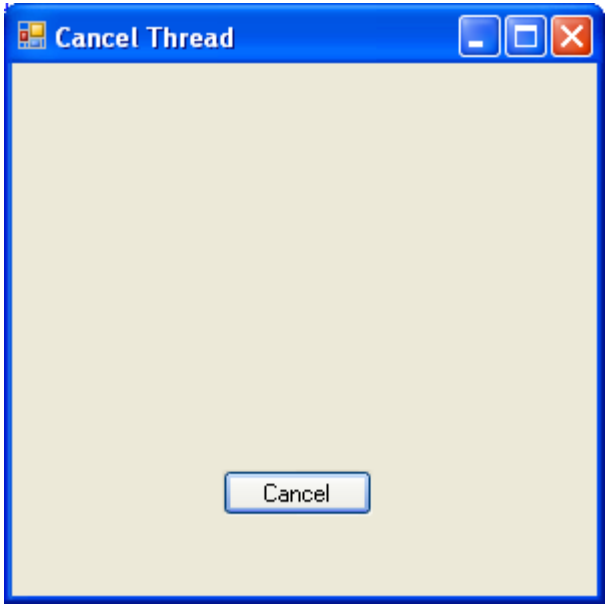
Cancel Example

```
            workerStart = new ThreadStart(WorkerThread);
              workerThread = new Thread(workerStart);
              workerThread.IsBackground = true;
              cancel = false;
              workerThread.Start();
          }
      }
      private void WorkerThread()
      {
          MessageBox.Show("Worker thread started.");
          //Loop for approximately 30 seconds
          //(depends on system speed)
          long i = 3000000000;
```

Cancel Example

```
            while (--i > 0) if(cancel) break;
            MessageBox.Show("Worker thread terminated.");
            workerThread = null;
        }
        private void cancelButton_Click(object sender,
           EventArgs e)
        {
            cancel = true;
        }
    }
}
```

The Form



Using Events with Threads

- It will probably cross your mind that the use of C# events might be a simple way to invoke a method in one thread from another.
- Unfortunately this is illusory. Events work perfectly, but the handler for the event is executed on the thread that triggered the event.
- There is a subtle way for us to accomplish asynchronous method invocation across thread boundaries.

Control.Invoke()

- Any class derived from control has a method named Invoke.
- If it is called with a reference to a delegate, the delegate is invoked on the thread that owns the controls underlying window.
- This is perfect for invoking a method in a UI thread from a worker thread.

Invoke Example

```
Invoke Example - Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
namespace InvokeExample
{
    public partial class Form1 : Form
    {
```

Invoke Example

```
private Thread workerThread;
private ThreadStart workerStart;
private bool cancel = false;
private delegate void Done();
private Done done;
public Form1()
{
    InitializeComponent();
    done = new Done(ThreadDone);
}
private void Form1_Click(object sender,
        EventArgs e)
```

Invoke Example

```
        {
            //Don't create more than one worker thread
            if (workerThread == null)
            {
                workerStart =
                     new ThreadStart(WorkerThread);
                workerThread = new Thread(workerStart);
                workerThread.IsBackground = true;
                cancel = false;
                Cursor = Cursors.WaitCursor;
                workerThread.Start();
            }
        }
```

Invoke Example

```
private void WorkerThread()
{
    //Loop for approximately 30 seconds (depends
    //on system speed)
    long i = 3000000000;
    while (--i > 0) if(cancel) break;
    Invoke(done);
    //done();
    workerThread = null;
}
```

Invoke Example

```
        private void cancelButton_Click(object sender,
                EventArgs e)
        {
            cancel = true;
        }
        private void ThreadDone()
        {
            Cursor = Cursors.Arrow;
        }
    }
}
```

Thread Safety

- If we execute a method in more than one thread at a time and there is any shared data we need to ensure that the method is *thread safe*.
- Race conditions are the root cause of this problem.
- You should be familiar with race conditions from operating systems theory.
- The .NET documentation usually mentions if a method is *thread safe*.
- It is usually a good idea to err on the side of caution.
- Don't execute methods in two or more threads that operate on behalf of the same object instance.

Mutual Exclusion

- The *Monitor* class is used to guarantee mutual exclusion. (Not the same as the classic monitor from operating systems.)
- We learned in operating systems how important mutual exclusion is when in a *critical section*.
- Monitor.Enter(Object obj) and Monitor.Exit(Object obj) bracket a critical section.
- Careful – don’t use a value type as argument to Enter and Exit. A value type will be boxed and therefore become a new object. Mutual exclusion will not be guaranteed since each thread that boxes the value type gets a new reference type instead.

Mutual Exclusion – Contd.

- Any object can be used to obtain the lock, but usually we use the object containing the critical data.
- Normally we use the Monitor with a try finally block to guarantee that Exit will be called.

```
Monitor.Enter(obj);
try
{ … }
finally {Monitor.Exit(obj);}
```

C# *lock* Keyword

- C# includes the lock keyword that is functionally equivalent to the previous try finally construct using the Monitor.
- It is simpler.

```
lock(obj)
{
    //critical section code
}
```

- That's it!

Timer

- The *Thread.Sleep* method can be used to suspend a thread for some number of milliseconds.
- The resolution of this timer is somewhat worse than a millisecond and shouldn’t be used for exact timing purposes.
- We often insert brief delays in applications when we know, for example, that an external event needs to be allowed time to respond.
- We gain better efficiency by not burning up CPU cycles.
- Of course if we can make a blocking call that is always a better choice rather than some type of polling.

Progress Bar Example

```
SleepEx - Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
namespace SleepEx
{
    public partial class Form1 : Form
    {
```

Progress Bar Example

```
private Thread workerThread;
private ThreadStart workerStart;
private delegate void Step();
private Step step;
public Form1()
{
    InitializeComponent();
    step = new Step(StepBar);
}
private void startButton_Click(object sender,
        EventArgs e)
{
    //Don't create more than one worker thread
```

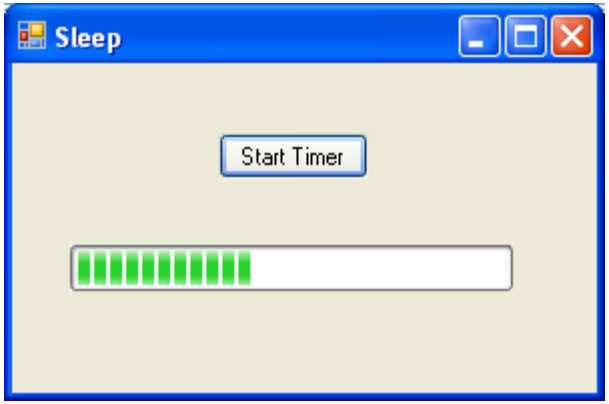
Progress Bar Example

```
        if (workerThread == null)
        {
            pBar.Value = 0; //reset progress bar
            workerStart =
                new ThreadStart(WorkerThread);
            workerThread = new Thread(workerStart);
            workerThread.IsBackground = true;
            workerThread.Start();
        }
    }
    private void WorkerThread()
    {
```

Progress Bar Example

```
            for (int i = 0; i < 30; ++i)
            {
                Thread.Sleep(1000);
                Invoke(step);
            }
            workerThread = null;
        }
        private void StepBar()
        {
            pBar.PerformStep();
        }
    }
}
```

Output



Thread Synchronization

- How can one thread wait for another thread without constantly examining a memory location to see if it has changed?
- *Semaphores* are a classic solution.
- .NET includes an easy alternative that I will demonstrate.

AutoResetEvent

- AutoResetEvent objects solve *synchronization* problems.
- If we want one thread to wait for an operation to be completed in another thread we use the AutoResetEvent class.
- The WaitOne and Set methods are similar to *Wait* and *Signal* as used in operating system texts.

Join

- We can wait for a thread to terminate by using the Join method.

```
Thread myThread;
//create and start the thread
myThread.Join(); //waits for thread to terminate
```

FlipFLop Example without Synchronization

```
FlipFlop without synchronization
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.IO;
namespace FlipFlop
{
    class FlipFlop
    {
        static void Main(string[] args)
        {
            Thread flip = new Thread(new
                ThreadStart(flipT));
```

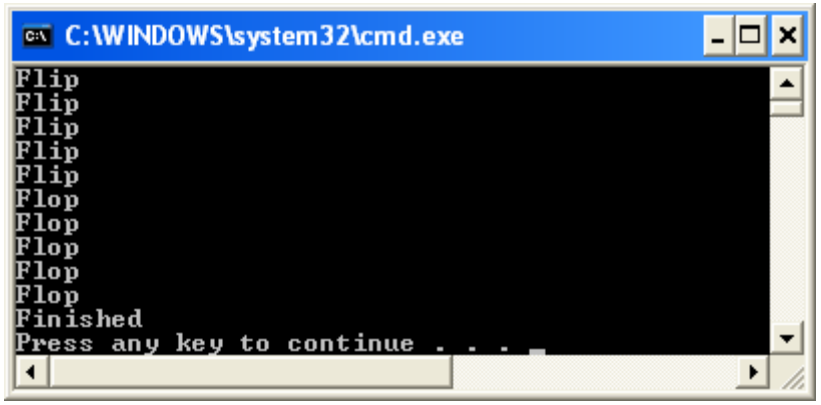
FlipFLop Example without Synchronization

```
        Thread flop = new Thread(new
              ThreadStart(flopT));
        flip.Start();
        flop.Start();
        flip.Join();
        flop.Join();
        Console.WriteLine("Finished");
    }
    static void flipT()
    {
        for(int i=0;i<5;++i)
        {
            Console.WriteLine("Flip");
```

FlipFLop Example without Synchronization

```
            }
        }
        static void flopT()
        {
            for (int i = 0; i < 5; ++i)
            {
                Console.WriteLine("Flop");
            }
        }
    }
}
```

Output



FliFlop Example with Synchronization

```
FlipFlop - FlipFlop.cs
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.IO;
namespace FlipFlop
{
    class FlipFlop
    {
        static AutoResetEvent flipE = new
               AutoResetEvent(true);
```

FliFlop Example with Synchronization

```
static AutoResetEvent flopE = new
     AutoResetEvent(false);
static void Main(string[] args)
{
    Thread flip = new Thread(new
       ThreadStart(flipT));
    Thread flop = new Thread(new
       ThreadStart(flopT));
    flip.Start();
    flop.Start();
    flip.Join();
    flop.Join();
    Console.WriteLine("Finished");
}
```

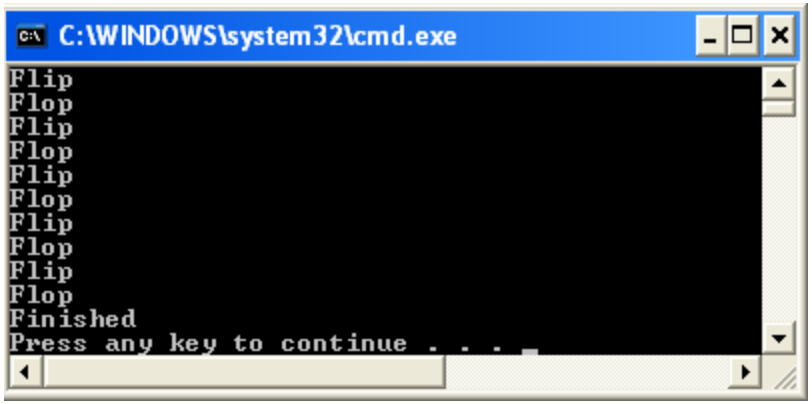
FliFlop Example with Synchronization

```
static void flipT()
{
    for(int i=0;i<5;++i)
    {
        flipE.WaitOne();
        Console.WriteLine("Flip");
        flopE.Set();
    }
}
```

FliFlop Example with Synchronization

```
        static void flopT()
        {
            for (int i = 0; i < 5; ++i)
            {
                flopE.WaitOne();
                Console.WriteLine("Flop");
                flipE.Set();
            }
        }
    }
}
```

Output



Thread Priorities

- A threads scheduling priority can be set.
- The Priority property can be set to:
  - Highest
  - AboveNormal
  - Normal (default)
  - BelowNormal
  - Lowest

Terminating a Thread (Aborting)

- A thread can be forcefully terminated by using the Thread.Abort method.
- Aborting a thread is not always a clean approach.
- Things like database connections may not be closed.
- Use *finally* blocks to ensure releasing these resources.
- Beware a bug discussed in the Prosise book (page 670).

Reader/Writer Locks

- The FCL includes a special lock that allows concurrent reading and mutually exclusive writing.
- AcquireReaderLock and AcquireWriterLock are the essential methods.
- Otherwise this class works similarly to a Monitor.

Mutexes

- Mutexes are similar to monitors except that they can span processes.
- To do this we give the mutex a name that can be used by more than one application.
- I am not going to cover this.
- I don’t feel it has use except for very advanced applications. Use a Monitor (lock) instead.