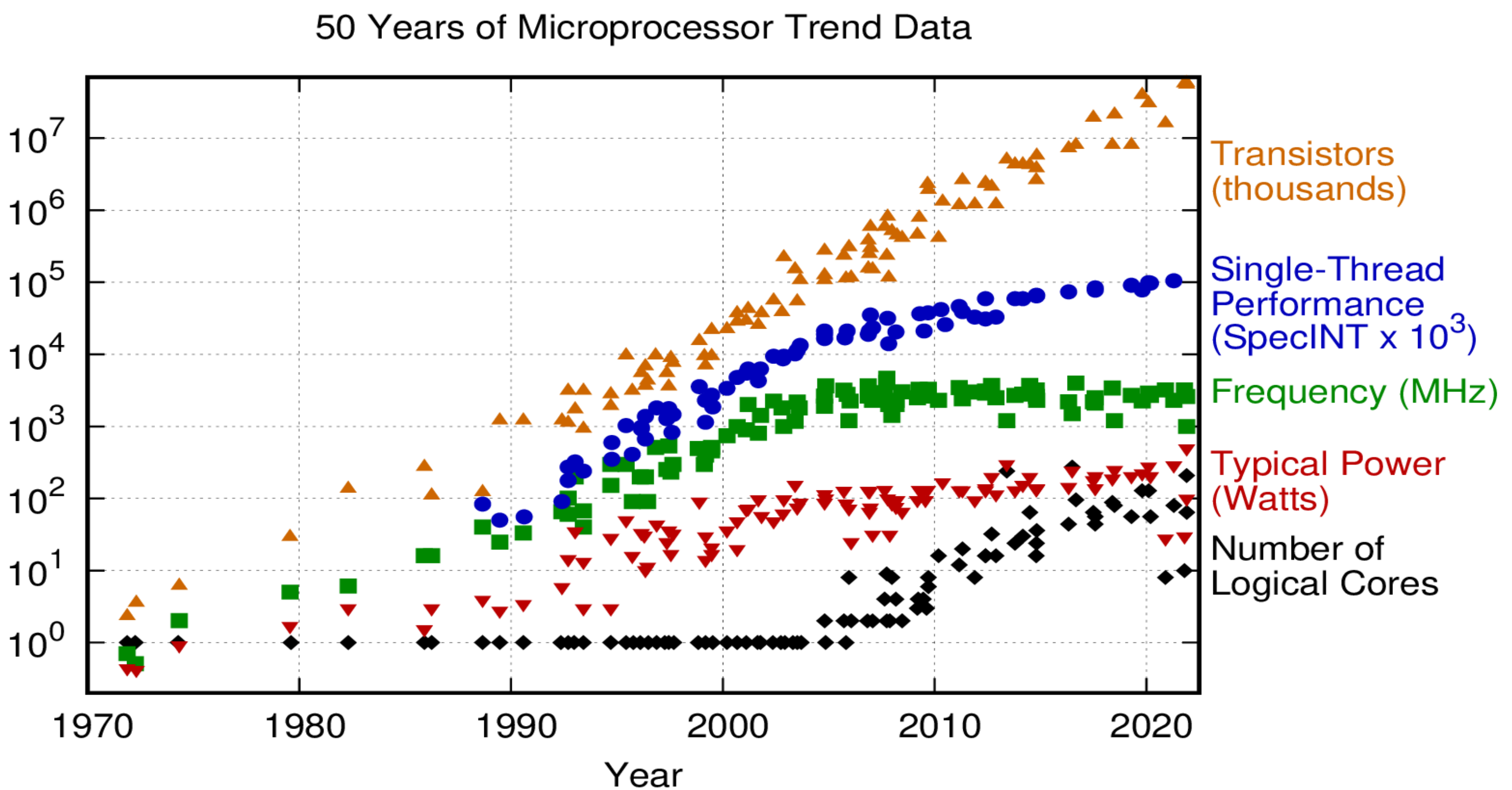
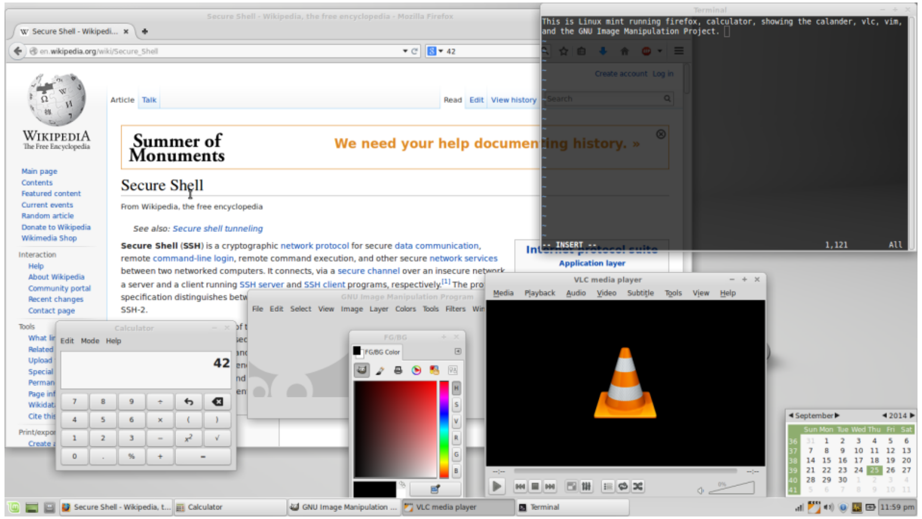
Managed Threading



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Multitasking

- Obviously, mechanisms are needed for the simultaneous execution of several tasks
- One approach is preemptive multitasking
- Allocate time slices to tasks



```
This is Linux mint running firefox, calculator, showing the calander, vlc, vim,
and the GNU Image Manipulation Project.
-- INSERT --                                                    1,121         All
```

Secure Shell

From Wikipedia, the free encyclopedia

See also: Secure shell tunneling

Secure Shell (**SSH**) is a cryptographic network protocol for secure data communication, remote command-line login, remote command execution, and other secure network services between two networked computers. It connects, via a secure channel over an insecure network a server and a client running SSH server and SSH client programs, respectively.[1] The pro specification distinguishes betw SSH-2.

Secure Shell - Wikipedia, the free encyclopedia - Mozilla Firefox
en.wikipedia.org/wiki/Secure_Shell
Wikipedia The Free Encyclopedia
Summer of Monuments
We need your help documenting history. »
Calculator
42
GNU Image Manipulation Program
FG/BG Color
VLC media player
Media Playback Audio Video Subtitle Tools View Help
September 2014
Terminal
11:59 pm

Multitasking

- Multiprogramming

During batch processing, several different programs were loaded in the computer memory, and the first one began to run. When the first program reached an instruction waiting for a peripheral, the context of this program was stored away, and the second program in memory was given a chance to run. The process continued until all programs finished running.

- Cooperative multitasking

Early multitasking systems used applications that voluntarily ceded time to one another. This approach, which was eventually supported by many computer operating systems, is known today as cooperative multitasking.

As a cooperatively multitasked system relies on each process regularly giving up time to other processes on the system, one poorly designed program can consume all of the CPU time for itself, either by performing extensive calculations or by busy waiting; both would cause the whole system to hang. In a server environment, this is a hazard that makes the entire environment unacceptably fragile.

- Preemptive multitasking

I/O bound vs CPU bound

Computational tasks can be divided into two categories:

- I/O bound - requiring I/O
  - an operation that spends most of its time waiting for something to happen
  - an example is downloading a web page or calling Console.ReadLine
  - Thread.Sleep is also deemed I/O-bound.
- CPU bound (compute-bound) - loading the processor

I/O bound tasks can be blocked, allowing processes that need processing power to do their work

Process

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity

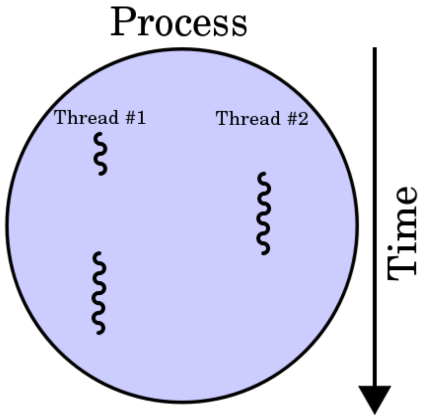
In general, a computer system process consists of (or is said to own) the following resources:

- An image of the executable machine code associated with a program.
- Memory, which includes the executable code, process-specific data, a call stack, and a heap
- Operating system descriptors of resources that are allocated to the process
- Security attributes, such as the process owner and the process' set of permissions
- Processor state (context), such as the content of registers and physical memory addressing

Thread in general

In computer science, a thread of execution is the smallest logical sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

Thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.



Process vs Thread

- processes are typically independent, while threads exist as subsets of a process
- processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
- processes have separate address spaces, whereas threads share their address space (heap)

Default threads in .NET

Console - the main thread, the GC thread & the finalizer thread (+ debugging process)

WPF - applications run with a minimum of the following two threads - for rendering: It runs in the background, so it is hidden. For managing UI interface (UI thread): Most of the WPF objects are tied with UI threads. It receives input, paints the screen, runs code and handles events. (+ see previous)

ASP.NET - each users request handles by threadpool thread (+ see previous console)

Advantages

- **Maintaining a responsive user interface**

By running time-consuming tasks on a parallel “worker” thread, the main UI thread is free to continue processing keyboard and mouse events.

- **Making efficient use of an otherwise blocked CPU**

Multithreading is useful when a thread is awaiting a response from another computer or piece of hardware. While one thread is blocked while performing the task, other threads can take advantage of the otherwise unburdened computer.

- **Parallel programming**

Code that performs intensive calculations can execute faster on multicore or multiprocessor computers if the workload is shared among multiple threads in a “divide-and-conquer” strategy.

- **Speculative execution**

On multicore machines, you can sometimes improve performance by predicting something that might need to be done, and then doing it ahead of time.

- **Allowing requests to be processed simultaneously**

On a server, client requests can arrive concurrently and so need to be handled in parallel (the .NET Framework creates threads for this automatically if you use ASP.NET, WCF, Web Services, or Remoting). This can also be useful on a client (e.g., handling peer-to-peer networking — or even multiple requests from the user).

Disadvantages

- Can increase complexity.
- Tricky interaction between threads (typically via shared data)
- Long development cycles and an ongoing susceptibility to intermittent and nonreproducible bugs. For this reason, it pays to keep interaction to a minimum, and to stick to simple and proven designs wherever possible
- Threading also incurs a resource and CPU cost in scheduling and switching threads (when there are more active threads than CPU cores) — and there’s also a creation/tear-down cost

Block 0

- What is os process
- What is thread
- Advantages of multithreading
- Disadvantages of multithreading

..And again

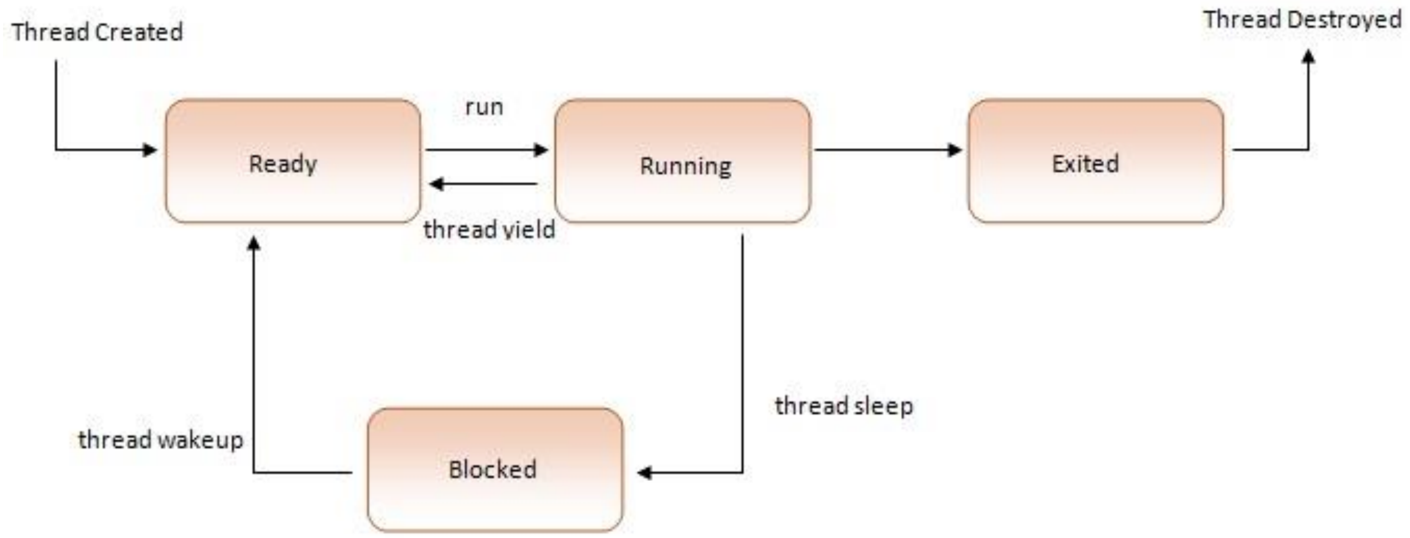
Multithreading is managed internally by a thread scheduler, a function the CLR typically delegates to the operating system. A thread scheduler ensures all active threads are allocated appropriate execution time, and that threads that are waiting or blocked (for instance, on an exclusive lock or on user input) do not consume CPU time.

On a single-processor computer, a thread scheduler performs *time-slicing* — rapidly switching execution between each of the active threads. Under Windows, a time-slice is typically in the tens-of-milliseconds region — much larger than the CPU overhead in actually switching context between one thread and another (which is typically in the few-microseconds region).

On a multi-processor computer, multithreading is implemented with a mixture of time-slicing and genuine *concurrency*, where different threads run code simultaneously on different CPUs. It's almost certain there will still be some time-slicing, because of the operating system's need to service its own threads — as well as those of other applications.

A thread is said to be *preempted* when its execution is interrupted due to an external factor such as time-slicing. In most situations, a thread has no control over when and where it's preempted.

Thread lifecycle



CLR`s Thread

Represented by System.Threading.Thread class

Constructors:

- Thread(ThreadStart) Initializes a new instance of the Thread class.
- Thread(ParameterizedThreadStart) - specifying a delegate that allows an object to be passed to the thread when the thread is started.
- Thread(ParameterizedThreadStart, Int32) - specifying a delegate that allows an object to be passed to the thread when the thread is started and specifying the maximum stack size for the thread.
- Thread(ThreadStart, Int32) - Initializes a new instance of the Thread class, specifying the maximum stack size for the thread.

CLR`s Thread

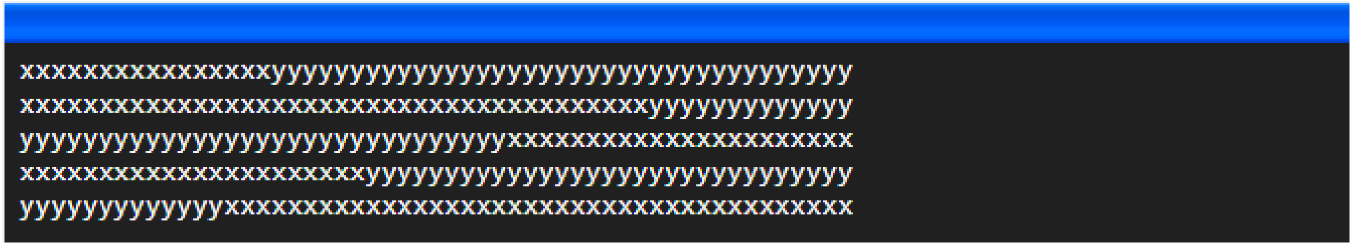
Delegate definitions:

- `public delegate void ThreadStart();`
- `public delegate void ParameterizedThreadStart(object obj);`

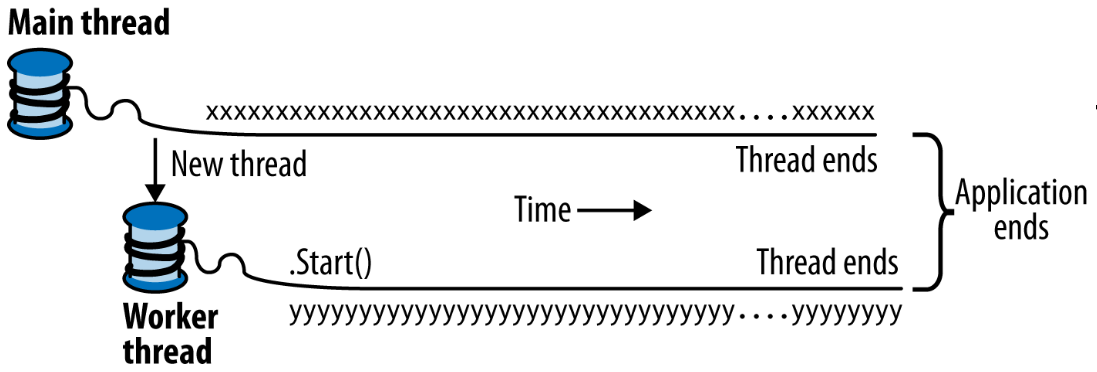
CLR`s Thread: Example

```
using System;
using System.Threading;

static void Main()
{
    Thread t = new Thread(WriteY);  // Kick off a new thread
    t.Start();                      // running WriteY()

    // Simultaneously, do something on the main thread.
    for (int i = 0; i < 1000; i++) Console.Write("x");
}

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

CLR`s Thread: Example



```
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```


Main thread
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx....xxxxxx
New thread
Thread ends
Time
Application ends
.Start()
Thread ends
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy....yyyyyyyy
Worker thread

Transfer of control is non-deterministic

CLR`s Thread: Example

```
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread(new ThreadStart(Go));

    t.Start();   // Run Go() on the new thread.
    Go();        // Simultaneously run Go() in the main thread.
  }

  static void Go()
  {
    Console.WriteLine("hello!");
  }
}
```

CLR`s Thread: Sharing data example

```
class ThreadTest
{
  static void Main()
  {
    bool _done = false;
    new Thread(Go).Start();
    Go();
  }
  static void Go()
  {
      if (!_done)
      {
          _done = true;
          Console.WriteLine("Done"); // If we swap the order of statements in the Go method, the
odds of “Done” being printed twice go up dramatically
      }
  }
}
```

CLR`s Thread: Passing data example

Passing data:

```
static void Main()
{
    Thread t = new Thread(Print);
    t.Start("Hello from t!");
}

static void Print(object messageObj)
{
    string message = (string)messageObj;   // We need to cast here
    Console.WriteLine(message);
}
```

CLR`s Thread: Another example

Passing data and lambda:

```
static void Main()
{
    Thread t = new Thread(() => Print("Hello from t!"));
    t.Start();
}

static void Print(string message)
{
    Console.WriteLine(message);
}
```

CLR`s Thread: Returning value

```
Retrieving data:

void Main()
{
    object value = null; // Used to store the return value
    var thread = new Thread(
              () =>
               {
                  value = "Hello World"; // Publish the return value
               });
    thread.Start();
    thread.Join();
    Console.WriteLine(value); // Use the return value here
}
```

Block 1

- Thread execution on single and multi core systems
- How to pass data into a thread?
- How to return data from a thread?

Thread.Start Method

- Start() - causes the operating system to change the state of the current instance to Running.
- Start(Object) - causes the operating system to change the state of the current instance to Running, and optionally supplies an object containing data to be used by the method the thread executes.

Once a thread is in the ThreadState.Running state, the operating system can schedule it for execution. The thread begins executing at the first line of the method represented by the ThreadStart or ParameterizedThreadStart delegate supplied to the thread constructor. Note that the call to Start does not block the calling thread.

Once the thread terminates, it cannot be restarted with another call to Start.

Thread.Join Method

- Join() - blocks the calling thread until the thread represented by this instance terminates
- Join(Int32) - blocks the calling thread until the thread represented by this instance terminates or the specified time elapses,
- Join(TimeSpan) - blocks the calling thread until the thread represented by this instance terminates or the specified time elapses

If the thread has already terminated when Join is called, the method returns immediately.

Thread.Join Method

This method changes the state of the calling thread to include ThreadState.WaitSleepJoin. You cannot invoke Join on a thread that is in the ThreadState.Unstarted state.

```
static void Main()
{
  Thread t = new Thread(Go);
  t.Start();
  t.Join();
  Console.WriteLine("Thread t has ended!");
}

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

```
public class Example
{
   static Thread thread1, thread2;

   public static void Main()
   {
      thread1 = new Thread(ThreadProc);
      thread1.Name = "Thread1";
      thread1.Start();

      thread2 = new Thread(ThreadProc);
      thread2.Name = "Thread2";
      thread2.Start();
   }

   private static void ThreadProc()
   {
      Console.WriteLine("\nCurrent thread: {0}", Thread.CurrentThread.Name);
      if (Thread.CurrentThread.Name == "Thread1" &&
          thread2.ThreadState != ThreadState.Unstarted)
         if (thread2.Join(2000))
            Console.WriteLine("Thread2 has termminated.");
         else
            Console.WriteLine("The timeout has elapsed and Thread1 will resume.");

      Thread.Sleep(4000);
      Console.WriteLine("\nCurrent thread: {0}", Thread.CurrentThread.Name);
      Console.WriteLine("Thread1: {0}", thread1.ThreadState);
      Console.WriteLine("Thread2: {0}\n", thread2.ThreadState);
   }
}
```

```
// The example displays the following
output:
//       Current thread: Thread1
//
//       Current thread: Thread2
//       The timeout has elapsed and Thread1
will resume.
//
//       Current thread: Thread2
//       Thread1: WaitSleepJoin
//       Thread2: Running
//
//
//       Current thread: Thread1
//       Thread1: Running
//       Thread2: Stopped
```

Thread.Sleep Method

- Sleep(Int32) - suspends the current thread for the specified number of milliseconds.
- Sleep(TimeSpan) - suspends the current thread for the specified amount of time.

The thread will not be scheduled for execution by the operating system for the amount of time specified. This method changes the state of the thread to include WaitSleepJoin.

Blocking

- A thread is deemed blocked when its execution is paused for some reason, such as when Sleeping or waiting for another to end via Join.
- A blocked thread immediately yields its processor time slice, and from then on it consumes no processor time until its blocking condition is satisfied
- When a thread blocks or unblocks, the OS performs a context switch. This incurs a small overhead, typically one or two microseconds

Blocking versus spinning

- I/O-bound operations that wait synchronously spend most of their time blocking a thread. They can also “spin” in a loop periodically:

```
while (DateTime.Now < nextStartTime)
    Thread.Sleep(100);
// or
while (DateTime.Now < nextStartTime)
```

- In general, this is very wasteful on processor time: as far as the CLR and OS are concerned, the thread is performing an important calculation and thus is allocated resources accordingly. In effect, we’ve turned what should be an I/O-bound operation into a compute-bound operation.

Thread.Interrupt

Interrupts a thread that is in the `WaitSleepJoin` thread state.

If this thread is not currently blocked in a wait, sleep, or join state, it will be interrupted when it next begins to block.
ThreadInterruptedException is thrown in the interrupted thread, but not until the thread blocks. If the thread never blocks, the exception is never thrown, and thus the thread might complete without ever being interrupted.

```
class ThreadInterrupt
{
    static void Main()
    {
        var stayAwake = new StayAwake();
        var newThread =
            new Thread(stayAwake.ThreadMethod);
        newThread.Start();

        // The following line causes an exception to be
thrown
        // in ThreadMethod if newThread is currently blocked
        // or becomes blocked in the future.
        newThread.Interrupt();
        Console.WriteLine("Main thread calls Interrupt on
newThread.");

        // Tell newThread to go to sleep.
        stayAwake.SleepSwitch = true;

        // Wait for newThread to end.
        newThread.Join();
    }
}
```

```
//Main thread calls Interrupt on newThread.
// newThread is executing ThreadMethod.
// newThread going to sleep.
// newThread cannot go to sleep -
interrupted by main thread.
```

```
class StayAwake
{
    bool sleepSwitch = false;
    public bool SleepSwitch
    {
        set{ sleepSwitch = value; }
    }

    public void ThreadMethod()
    {
        Console.WriteLine("newThread is executing
ThreadMethod.");
        while(!sleepSwitch)
        {
            // Use SpinWait instead of Sleep to demonstrate the
            // effect of calling Interrupt on a running thread.
            Thread.SpinWait(10000000);
        }
        try
        {
            Console.WriteLine("newThread going to sleep.");

            // When newThread goes to sleep, it is immediately
            // woken up by a ThreadInterruptedException.
            Thread.Sleep(Timeout.Infinite);
        }
        catch(ThreadInterruptedException e)
        {
            Console.WriteLine("newThread cannot go to sleep - " +
                "interrupted by main thread.");
        }
    }
}
```

Thread.Abort Method

- Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
- If `Abort` is called on a thread that has not been started, the thread will abort when Start is called. If `Abort` is called on a thread that is blocked or is sleeping, the thread is interrupted and then aborted.
- After `Abort` is invoked on a thread, the state of the thread includes AbortRequested. After the thread has terminated as a result of a successful call to `Abort`, the state of the thread is changed to Stopped. With sufficient permissions, a thread that is the target of an `Abort` can cancel the abort using the `ResetAbort` method. For an example that demonstrates calling the `ResetAbort` method, see the `ThreadAbortException` class.
- When this method is invoked on a thread, the system throws a ThreadAbortException in the thread to abort it. `ThreadAbortException` is a special exception that can be caught by application code, but is re-thrown at the end of the `catch` block unless ResetAbort is called. `ResetAbort` cancels the request to abort, and prevents the `ThreadAbortException` from terminating the thread. Unexecuted `finally` blocks are executed before the thread is aborted.

Thread.Abort() is obsolete as of .NET 5. This information applies only to older versions of the .NET

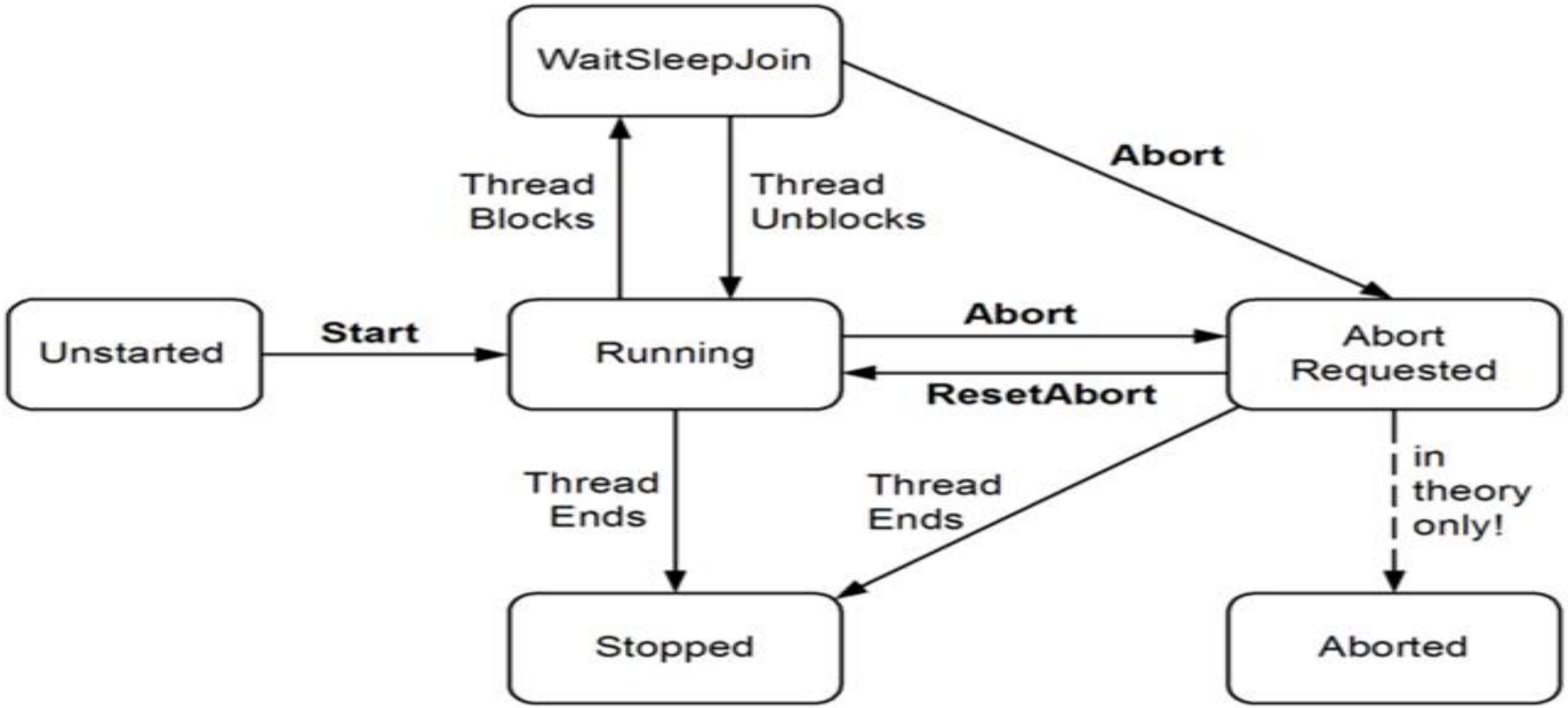
```
public static void Main()
    {
        Thread newThread  = new Thread(new ThreadStart(TestMethod));
        newThread.Start();
        Thread.Sleep(1000);

        // Abort newThread.
        Console.WriteLine("Main aborting new thread.");
        newThread.Abort("Information from Main.");

        // Wait for the thread to terminate.
        newThread.Join();
        Console.WriteLine("New thread terminated - Main exiting.");
    }

    static void TestMethod()
    {
        try
        {
            while(true)
            {
                Console.WriteLine("New thread running.");
                Thread.Sleep(1000);
            }
        }
        catch(ThreadAbortException abortException)
        {
            Console.WriteLine((string)abortException.ExceptionState);
        }
    }
```

Interrupt vs Abort

First, Thread.Abort says "I don't care what you're doing, just stop doing it, and leave everything as it is right now". It's basically the programming way of saying "Hey, beat it".

Secondly, Thread.Interrupt is a rather strange beast. It basically says "I don't care what you're waiting for, stop waiting for it". The strange thing here is that if the thread isn't currently waiting for anything, it's instead "I don't care what you're going to wait for next, but when you do, stop waiting for it immediately".

Thread`s state diagram



Thread`s properties

```
public ThreadState ThreadState { get; } // You can test for a thread being blocked via its 
ThreadState property
public string Name { get; set; } // For debugging. You can set a thread’s name just once; attempts 
to change it later will throw an exception
public int ManagedThreadId { get; } // An integer that represents a unique identifier for this 
managed thread.
public bool IsAlive { get; } // true if this thread has been started and has not terminated 
normally or aborted; otherwise, false
public bool IsBackground { get; set; }
public ThreadPriority Priority { get; set; }
public bool IsThreadPoolThread { get; }

and many many more...

// test for a thread being blocked
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```

Thread.IsBackground Property

Gets or sets a value indicating whether or not a thread is a background thread.

A thread is either a background thread or a foreground thread. Background threads are identical to foreground threads, except that background threads do not prevent a process from terminating. Once all foreground threads belonging to a process have terminated, the common language runtime ends the process. Any remaining background threads are stopped and do not complete.
By default, the following threads execute in the foreground (that is, their IsBackground property returns `false`):

- The primary thread (or main application thread).
- All threads created by calling a Thread class constructor.

By default, the following threads execute in the background (that is, their IsBackground property returns `true`):

- Thread pool threads, which are a pool of worker threads maintained by the runtime. You can configure the thread pool and schedule work on thread pool threads by using the ThreadPool class.

```
static void Main()
{
     BackgroundTest shortTest = new BackgroundTest(10);
     Thread foregroundThread =
         new Thread(new ThreadStart(shortTest.RunLoop));
       BackgroundTest longTest = new BackgroundTest(50);
        Thread backgroundThread =
            new Thread(new ThreadStart(longTest.RunLoop));
        backgroundThread.IsBackground = true;

        foregroundThread.Start();
        backgroundThread.Start();
}

class BackgroundTest
{
    int maxIterations;

    public BackgroundTest(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public void RunLoop()
    {
        for (int i = 0; i < maxIterations; i++) {
            Console.WriteLine("{0} count: {1}",
                Thread.CurrentThread.IsBackground ?
                   "Background Thread" : "Foreground Thread", i);
            Thread.Sleep(250);
        }
        Console.WriteLine("{0} finished counting.",
                          Thread.CurrentThread.IsBackground ?
                          "Background Thread" : "Foreground Thread");
    }
}
```

```
// The example displays output like the following:
//    Foreground Thread count: 0
//    Background Thread count: 0
//    Background Thread count: 1
//    Foreground Thread count: 1
//    Foreground Thread count: 2
//    Background Thread count: 2
//    Foreground Thread count: 3
//    Background Thread count: 3
//    Background Thread count: 4
//    Foreground Thread count: 4
//    Foreground Thread count: 5
//    Background Thread count: 5
//    Foreground Thread count: 6
//    Background Thread count: 6
//    Background Thread count: 7
//    Foreground Thread count: 7
//    Background Thread count: 8
//    Foreground Thread count: 8
//    Foreground Thread count: 9
//    Background Thread count: 9
//    Background Thread count: 10
//    Foreground Thread count: 10
//    Background Thread count: 11
//    Foreground Thread finished counting.
```

Thread`s priority

A thread can be assigned any one of the following priority ThreadPriority values:

- `Highest`
- `AboveNormal`
- `Normal`
- `BelowNormal`
- `Lowest`

Operating systems are not required to honor the priority of a thread.

The thread with the highest priority (of those threads that can be executed) is always scheduled to run first. If multiple threads with the same priority are all available, the scheduler cycles through the threads at that priority, giving each thread a fixed time slice in which to execute. As long as a thread with a higher priority is available to run, lower priority threads do not get to execute.

Exception handling

```
try
{
    new Thread(Go).Start();
}
catch (Exception ex)
{
    // We'll never get here!
    Console.WriteLine ("Exception!");
}

void Go() { throw null; } // Throws a NullReferenceException
```

Exception handling

```
new Thread(Go).Start();
void Go()
{
   try
   {
       ...
       throw null; // The NullReferenceException will get caught below
       ...
   }
   catch(Exception ex)
   {
       // Typically log the exception, and/or signal another thread
       // that we've come unstuck
       ...
   }
}
```

Exceptions

An unhandled exception causes the whole application to shut down

Why? When threads are allowed to fail silently, without terminating the application, serious programming problems can go undetected. This is a particular problem for services and other applications which run for extended periods. As threads fail, program state gradually becomes corrupted. Application performance may degrade, or the application might hang.

Exceptions:

- A ThreadAbortException is thrown in a thread because Abort was called.
- An AppDomainUnloadedException is thrown in a thread because the application domain in which the thread is executing is being unloaded.

Block 2

- How exceptions work in thread
- How to “resume” thread
- How to cancel thread
- How to resume aborted thread

Thread.Yield

`public static bool Yield()` - *method causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.*

Using of Thread-Local Storage

Sometimes, you want to keep data isolated, ensuring that each thread has a separate copy

- **[ThreadStatic]**
- **ThreadLocal<T>**
- **GetData and SetData**

ThreadStatic

- Only for static fields
- Each thread sees a separate copy
- Field initializers — they execute **only *once*** on the thread that's running when the static constructor executes

```
[ThreadStatic]
static int Number;

static Thread t1;
static Thread t2;
public static void Main()
{
    t1 = new Thread(ParentMethod);
    t2 = new Thread(ChildMethod);
    t1.Start();
    t2.Start();
    t1.Join();
    Console.WriteLine("main - " + Number);
}

static void ParentMethod()
{
    Number = Thread.CurrentThread.ManagedThreadId;
    t2.Join();
    Console.WriteLine("t1 - " + Number);
}
static void ChildMethod()
{
    Number = Thread.CurrentThread.ManagedThreadId;
    Console.WriteLine("t2 - " + Number);
}
```

```
// t2 - 4
// t1 - 3
// main - 0
```

```
[ThreadStatic]
static int I = 100;
public static void Main()
{
    var t = new Thread(() =>
    {
        Console.WriteLine("thread " + I);
    });
    t.Start();
    Console.WriteLine("main" + I);
}
```

```
// main100

// thread 0
```

ThreadLocal<T>

- It provides thread-local storage for both static and instance fields
- Allows you to specify default values
- Lazily evaluated: the factory function evaluates on the first call (for each thread)

Constructors

- ThreadLocal<T>()
- ThreadLocal<T>(Boolean) Specifies whether all values are accessible from any thread.
- ThreadLocal<T>(Func<T>) Instance with the specified `valueFactory` function.
- ThreadLocal<T>(Func<T>, Boolean) Specified `valueFactory` function and a flag that indicates whether all values are accessible from any thread.

```
    var localRandom = new ThreadLocal<int>(() =>
Guid.NewGuid().GetHashCode(), true);

    Action action = () =>
        {
            bool repeat = localRandom.IsValueCreated;

            int id = Thread.CurrentThread.ManagedThreadId;

            Console.WriteLine("Name = {0} {1} {2}",
           id, repeat ? "(repeat)" : "new", localRandom.Value);
        };

   System.Threading.Tasks.Parallel.Invoke(action,
     action, action, action, action, action, action, action);

   foreach(var v in localRandom.Values)
   {
       Console.WriteLine(v);
   }
```

```
// Name = 3 new 608799565
// Name = 5 new -1857818647
// Name = 6 new -794854433
// Name = 6 (repeat) -794854433
// Name = 1 new 1266633520
// Name = 4 new -97509854
// Name = 3 (repeat) 608799565
// Name = 5 (repeat) -1857818647

// -97509854
// -794854433
// 1266633520
// -1857818647
// 608799565
```

Thread class methods: GetData and SetData

- public sealed class LocalDataStoreSlot
  - Encapsulates a memory slot to store local data
  - The common language runtime allocates a multi-slot data store array to each process when it is created

- public static void SetData(LocalDataStoreSlot slot, object data)
- public static object GetData(LocalDataStoreSlot slot)

```
public static void Main()
        {
            Thread.AllocateNamedDataSlot("ErrNo");
            Thread.AllocateNamedDataSlot("ErrSource");
            Thread th2 = new Thread(SetError);
            th2.Name = "t2";
            th2.Start();
            Thread th3 = new Thread(SetError);
            th3.Name = "t3";
            th3.Start();

            Thread.FreeNamedDataSlot("ErrNo");
            Thread.FreeNamedDataSlot("ErrSource");
            Console.Read();
        }
```

```
public static void WriteError()
        {
           Console.WriteLine("Error number = " +
            Thread.GetData(Thread.GetNamedDataSlot("ErrNo")));
           Console.WriteLine("Error source = " +
            Thread.GetData(Thread.GetNamedDataSlot("ErrSource")));
        }
```

```
        public static void SetError()
        {
            Thread.SetData(Thread.GetNamedDataSlot("ErrNo"),
Thread.CurrentThread.ManagedThreadId);
            Thread.SetData(Thread.GetNamedDataSlot("ErrSource"),
Thread.CurrentThread.Name);
            WriteError();
        }
```

```
//  Error number = 3

// Error source = t2

// Error number = 4

// Error source = t3
```

Block 3

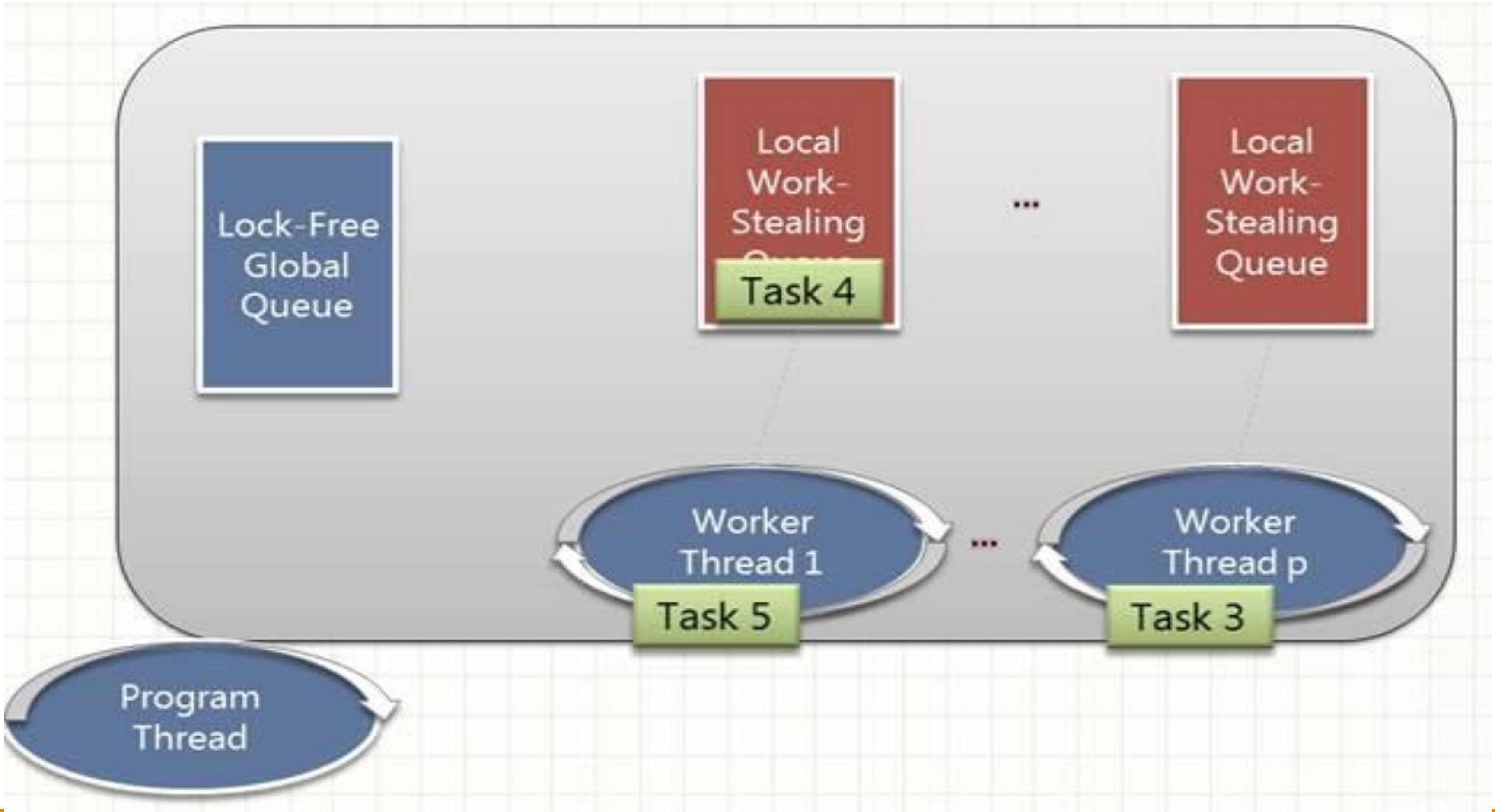
- What does Thread.Yield method do ?
- What does [ThreadStaticAttribute] do ?
- How to store thread specific data ?

Thread overhead in Windows

- Thread kernel object (processor registers)
- Thread Environment Block (exceptions tracking stack, local storage)
- User-mode stack (call-stack, local stack)
- Kernel-mode stack (interaction with os kernel functions)
- Virtual address space switching (if it necessary to switch process)
- Cpu context reloading
- Cpu cache reloading
- Garbage collecting (more threads - more local stacks)

ThreadPool

- One per process
- Sharing and recycling threads
- The thread pool also keeps a lid on the total number of worker threads it will run simultaneously
- Each thread is in Normal priority
- Each thread is Background thread


Lock-Free Global Queue
Local Work-Stealing Queue
Task 4
...
Local Work-Stealing Queue
Worker Thread 1
Task 5
...
Worker Thread p
Task 3
Program Thread

…And where it's better to use dedicated thread

- You require a foreground thread.
- You require a thread to have a particular priority.
- You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.
- You need to have a stable identity associated with the thread, or to dedicate a thread to a task.

Block 4

- What is thread pool advantages ?
- Where is it better to use dedicated thread ?

Worker threads vs I/O completion threads

- Both are normal clr threads
- Two sub-pools for each type
- I/O completions threads needed to handle native I/O callbacks
- To avoid a situation where high demand on worker threads exhausts all the threads available to dispatch native I/O callbacks
- Some kind of optimization

Additional optimization

```
public static bool SetMaxThreads (int workerThreads, int completionPortThreads);
```

the defaults are:

- 1023 in Framework 4.0 in a 32-bit environment
- 32768 in Framework 4.0 in a 64-bit environment
- 250 per core in Framework 3.5
- 25 per core in Framework 2.0

```
public static bool SetMinThreads (int workerThreads, int completionPortThreads);
```

`defaults are the number of processor cores available for process`

Thread pool is used implicitly:

- Via the Task Parallel Library (from Framework 4.0)
- PLINQ
- Via BackgroundWorker
- WCF, ASP.NET, and ASMX Web Services application servers
- System.Timers.Timer and System.Threading.Timer

ThreadPool using

To use **QueueUserWorkItem**, simply call this method with a delegate that you want to run on a pooled thread:

```
static void Main()
{
  ThreadPool.QueueUserWorkItem(Go);
  ThreadPool.QueueUserWorkItem(Go, 123);
  Console.ReadLine();
}

static void Go (object data)   // data will be null with the first call.
{
  Console.WriteLine ("Hello from the thread pool! " + data);
}
```

public delegate void WaitCallback (object state);

ThreadPool using

```
static void Main()
{
  // Start the task executing:
  Task<string> task = Task.Factory.StartNew<string>
        ( () => DownloadString ("http://www.linqpad.net") );

  // We can do other work here and it will execute in parallel:
  RunSomeOtherMethod();

  // When we need the task's return value, we query its Result property:
  // If it's still executing, the current thread will now block (wait)
  // until the task finishes:
  string result = task.Result;
}

static string DownloadString (string uri)
{
  using (var wc = new System.Net.WebClient())
        return wc.DownloadString (uri);
}
```

Block 5

- Where is thread pool used implicitly ?
- Difference between worker threads and I/O completions threads