Advanced Programming Methods

Lecture 13

This Lecture Overview

- Concurrency in C#

Thread pool

- when a thread starts, a few hundred microseconds are spent organizing such things as a fresh private local variable stack.
- each thread also consumes (by default) around 1 MB of memory.
- the thread pool cuts these overheads by sharing and recycling threads, allowing multithreading to be applied at a very granular level without a performance penalty.

Thread pool

Ways to enter the thread pool:

- Via the Task Parallel Library (from Framework 4.0)
- By calling ThreadPool.QueueUserWorkItem
- Via asynchronous delegates
- Via BackgroundWorker

Via Task Parallel Library

- **enter the thread pool using the Task class from System.Threading.Tasks**

```
static void Main()   {
  Task.Factory.StartNew (Go);
/*Task.Factory.StartNew returns a Task object, which can
   then be used to monitor the task — for instance, you
   can wait for it to complete by calling its Wait method */
}

static void Go(){
  Console.WriteLine ("Hello from the thread pool!");
}
```

```
static void Main(){
 //Task<TResult> class lets you get a return value back from
 //the task after it finishes executing
 Task<string> task = Task.Factory.StartNew<string>
   ( () => DownloadString ("http://www.aaa.com") );
// 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);}
```

ThreadPool.QueueUserWorkItem

- it is called 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);
}
```

Asynchronous delegates

```
static void Main() {
  Func<string, int> method = Work;
  IAsyncResult cookie = method.BeginInvoke ("test", null, null);
  //
  // ... here's where we can do other work in parallel...
  //
  int result = method.EndInvoke (cookie);
//EndInvoke waits for the asynchronous delegate to finish executing
//and it receives the return value
  Console.WriteLine ("String length is: " + result);
}

static int Work (string s) { return s.Length; }
```

Synchronization

- coordinating the actions of threads for a predictable outcome
- its constructs can be divided into four categories:

1. Simple blocking methods (e.g. Sleep, Join): wait for another thread to finish or for a period of time to elapse

Synchronization

2. Locking constructs (e.g. Lock): limit the number of threads that can perform some activity or execute a section of code at a time.

3. Signaling constructs: allow a thread to pause until receiving a notification from another, avoiding the need for inefficient polling.

4. Nonblocking synchronization constructs (e.g. Volatile): protect access to a common field by calling upon processor primitives

Blocking

- thread execution is paused for some reason
- thread consumes no processor time until blocking condition is satisfied

Unblocking happens in one of four ways:

- by the blocking condition being satisfied
- by the operation timing out (if a timeout is specified)
- by being interrupted via Thread.Interrupt
- by being aborted via Thread.Abort

Blocking Versus Spinning

A thread must pause until a certain condition is met:

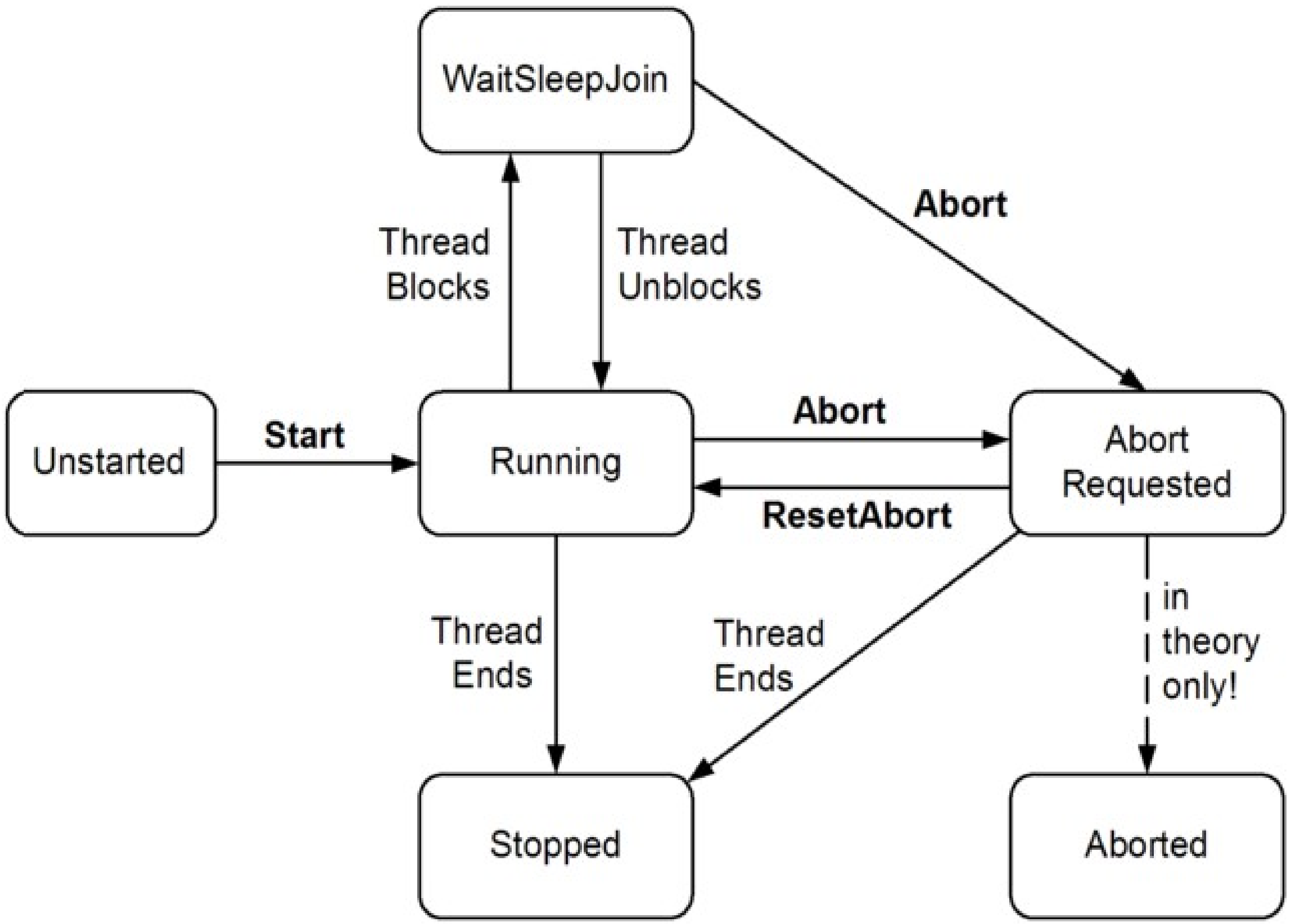
- efficiently: signaling and locking constructs achieve this by blocking until condition is satisfied
- simply but ineffiently: by spinning in a polling loop, e.g.:

```
while (!proceed);
```

or in a mixed way:

```
while (!proceed) Thread.Sleep (10);
```

ThreadState property



Locking

- **Exclusive locking is used to ensure that only one thread can enter particular sections of code at a time**

```
class ThreadSafe{
  static readonly object  _locker = new object();
  static int _val1, _val2;

  static void Go(){
    lock (_locker){
      if (_val2 != 0) Console.WriteLine (_val1 / _val2);
      _val2 = 0;
    }
}}
```

Locking

- lock statement is in fact a syntactic shortcut for a call to the methods Monitor.Enter and Monitor.Exit

```
Monitor.Enter (_locker);
try
{
  if (_val2 != 0) Console.WriteLine (_val1 / _val2);
  _val2 = 0;
}
finally { Monitor.Exit (_locker); }
```

Choosing the Synchronization Object

1. class ThreadSafe{

```
List <string> _list = new List <string>();
void Test(){
  lock (_list) {
    _list.Add ("Item 1");
    ...
```

2. lock the entire object: `lock (this) { ... }`

3. in case of static fields/methods: `lock (typeof (Widget)) { ... }`

When to Lock

- need to lock around accessing any writable shared field

```
class ThreadSafe{
  static readonly object  _locker = new object();
  static int _x;

  static void Increment() { lock (_locker)  _x++; }
  static void Assign()    { lock (_locker)  _x = 123; }
}
```

Locking and Atomicity

- if a group of variables are always read and written within the same lock, we can say the variables are read and written atomically

- for example x and y are accessed atomically:

lock (locker) { if (x != 0) y /= x; }

Nested Locking

```
static readonly object _locker = new object();

static void Main(){
  lock (_locker){
    AnotherMethod();
    // We still have the lock - because locks are reentrant.
  }
}
static void AnotherMethod(){
  lock (_locker) { Console.WriteLine ("Another method"); }
}
```

Deadlocks

- when two threads each waits for a resource held by the other, so neither can proceed

```
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
}
```

Mutex

- is like a lock, but it can work across multiple processes
- can be computer-wide as well as application-wide
- Mutex class:
    - WaitOne method to lock
    - ReleaseMutex to unlock
- a Mutex can be released only from the same thread that obtained it.

Example: A common use for a cross-process Mutex is to ensure that only one instance of a program can run at a time

```
class OneAtATimePlease{
 static void Main() {
  // Naming a Mutex makes it available computer-wide.
  using (var mutex = new Mutex (false, "UniqueName")){
    // Wait a few seconds, in case another instance
    // of the program is still in the process of shutting down.
    if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false)) {
      Console.WriteLine ("Another app instance is running. Bye!");
      return; }
    RunProgram();
  } }

 static void RunProgram(){
   Console.WriteLine ("Running. Press Enter to exit");
   Console.ReadLine();
 }}
```

Semaphore

- preventing too many threads from executing a particular piece of code at once.

- is like a room, it has a certain capacity. Once it's full, no more people can enter, and a queue builds up outside. Then, for each person that leaves, one person enters from the head of the queue.

- it has no owner, any thread can call release on a semaphore

```
class TheRoom {
  static SemaphoreSlim _sem = new SemaphoreSlim (3);    // Capacity of 3

  static void Main(){
    for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);
  }

  static void Enter (object id){
    Console.WriteLine (id + " wants to enter");
    _sem.Wait();
    Console.WriteLine (id + " is in!");           // Only three threads
    Thread.Sleep (1000 * (int) id);               // can be here at
    Console.WriteLine (id + " is leaving");       // a time.
    _sem.Release();
  }
}
```

Signaling with Event Wait Handles

Signaling:

- when one thread waits until it receives notification from another

Event wait handles:

- are the simplest of the signaling constructs
- are unrelated to C# events
- come in three flavors:
  - AutoResetEvent,
  - ManualResetEvent,
  - CountdownEvent

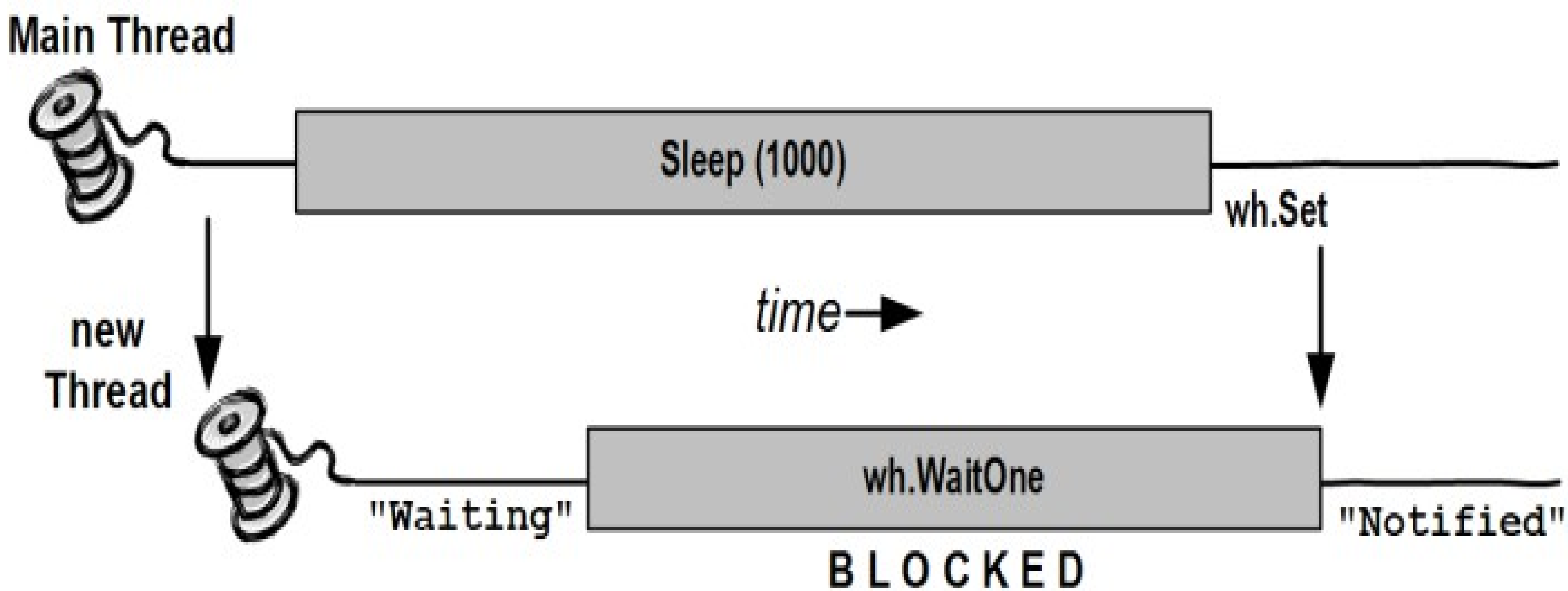
AutoResetEvent

- is like a ticket turnstile: inserting a ticket lets exactly one person through
- “auto” in the class’s name refers to the fact that an open turnstile automatically closes or “resets” after someone steps through
- a thread waits, or blocks, at the turnstile by calling WaitOne
- a ticket is inserted by calling the Set method

AutoResetEvent

- if a number of threads call WaitOne, a queue builds up behind the turnstile
- any thread with access to the AutoResetEvent object can call Set on it to release one blocked thread
- If Set is called when no thread is waiting, the handle stays open for as long as it takes until some thread calls WaitOne
- calling Set repeatedly on a turnstile at which no one is waiting: only the next single person is let through and the extra tickets are “wasted.”

Example: a thread is started whose job is simply to wait until signaled by another thread:



```
class BasicWaitHandle {
  static EventWaitHandle _waitHandle = new AutoResetEvent (false);

  static void Main(){
    new Thread (Waiter).Start();
    Thread.Sleep (1000);                 // Pause for a second...
    _waitHandle.Set();                   // Wake up the Waiter.
  }

  static void Waiter(){
    Console.WriteLine ("Waiting...");
    _waitHandle.WaitOne();               // Wait for notification
    Console.WriteLine ("Notified");
  }
}
```

Two-way Signaling

Example:

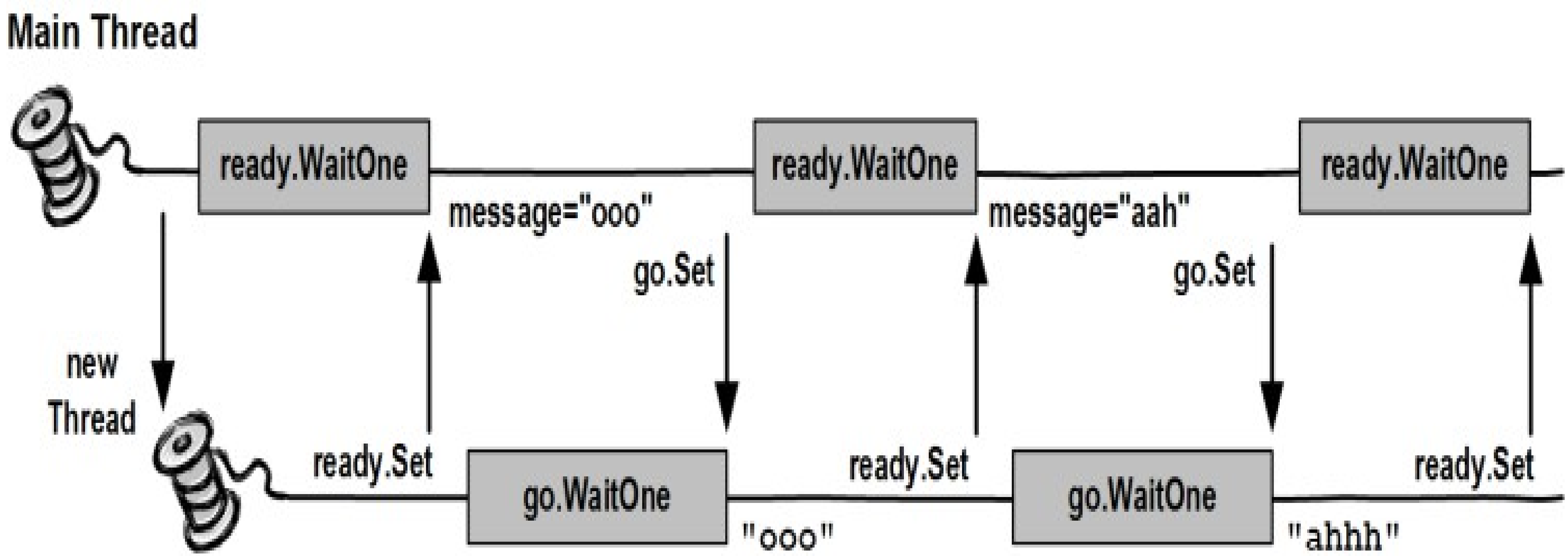
the main thread must signal a worker thread three times in a row

Bad solution:

- If the main thread simply calls Set on a wait handle several times in rapid succession, the second or third signal may get lost, since the worker may take time to process each signal.

Correct solution:

- the main thread must wait until the worker is ready before signaling it (using 2 AutoResetEvent)



```
class TwoWaySignaling {
  static EventWaitHandle _ready = new AutoResetEvent (false);
  static EventWaitHandle _go = new AutoResetEvent (false);
  static readonly object _locker = new object();
  static string _message;

  static void Main() {
    new Thread (Work).Start();
    _ready.WaitOne();                  // First wait until worker is ready
    lock (_locker) _message = "ooo";
    _go.Set();                         // Tell worker to go

    _ready.WaitOne();
    lock (_locker) _message = "ahhh";  // Give the worker another message
    _go.Set();
```

```
_ready.WaitOne();
  lock (_locker) _message = null;    // Signal the worker to exit
  _go.Set();
 }

 static void Work(){
   while (true){
     _ready.Set();                          // Indicate that we're ready
     _go.WaitOne();                          // Wait to be kicked off...
     lock (_locker){
       if (_message == null) return;        // Gracefully exit
       Console.WriteLine (_message);
     }
   }
}}
```

Producer/consumer queue

A producer/consumer queue works as follows:

- a queue is set up to describe work items — or data upon which work is performed.
- when a task needs executing, it's enqueued, allowing the caller to get on with other things.
- one or more worker threads plug away in the background, picking off and executing queued items.

```
using System;
using System.Threading;
using System.Collections.Generic;
 class ProducerConsumerQueue : IDisposable {
  EventWaitHandle _wh = new AutoResetEvent (false);
  Thread _worker;
  readonly object _locker = new object();
  Queue<string> _tasks = new Queue<string>();

  public ProducerConsumerQueue() {
    _worker = new Thread (Work);
    _worker.Start();}

  public void EnqueueTask (string task){
    lock (_locker) _tasks.Enqueue (task);
    _wh.Set(); }
```

```
void Work() {
  while (true){
    string task = null;
    lock (_locker)
      if (_tasks.Count > 0) {
        task = _tasks.Dequeue();
        if (task == null) return;
      }
    if (task != null){
      Console.WriteLine ("Performing task: " + task);
      Thread.Sleep (1000);  // simulate work...
    } else
      _wh.WaitOne();         // No more tasks - wait for a signal
  }
}
```

```
public void Dispose(){
    EnqueueTask (null);     // Signal the consumer to exit.
    _worker.Join();         // Wait for the consumer's thread to finish.
    _wh.Close();            // Release any OS resources.
  }
static void Main(){
  using (ProducerConsumerQueue q = new ProducerConsumerQueue()){
    q.EnqueueTask ("Hello");
    for (int i = 0; i < 10; i++) q.EnqueueTask ("Say " + i);
    q.EnqueueTask ("Goodbye!");
  }
  // Exiting the using statement calls q's Dispose method, which
  // enqueues a null task and waits until the consumer finishes.
}
}
```

ManualResetEvent

- is useful in allowing one thread to unblock many other threads

- functions like an ordinary gate

- calling Set opens the gate, allowing any number of threads calling WaitOne to be let through

- calling Reset closes the gate

-threads that call WaitOne on a closed gate will block; when the gate is next opened, they will be released all at once.

CountdownEvent

- allows waiting on more than one thread

- the class is instantiated with the number of threads or “counts” that have to be waited on
- calling Signal decrements the “count”
- calling Wait blocks until the count goes down to zero

```
static CountdownEvent _countdown = new CountdownEvent (3);

static void Main(){
  new Thread (SaySomething).Start ("I am thread 1");
  new Thread (SaySomething).Start ("I am thread 2");
  new Thread (SaySomething).Start ("I am thread 3");
  _countdown.Wait();   // Blocks until Signal has been called 3 times
  Console.WriteLine ("All threads have finished speaking!");
}

static void SaySomething (object thing){
  Thread.Sleep (1000);
  Console.WriteLine (thing);
  _countdown.Signal();
}
```