C# Threads

By
Khaled Alanezi

Graduate Presentation
CSCI 5448 OOAD – Spring 2011
University of Colorado, Boulder

Agenda

- Introduction
- C# Specifics
- Basic Threads Operations
- Putting Basic Operations Together
- Locking Granularity
- C# Thread's Status
- C# Thread's Properties
- Thread Class Important Methods
- Mutex Class
- Semaphore Class
- ReaderWriterLock Class
- ThreadPool Class
- Conclusion
- Resources
- Where to look for additional information?

Introduction

- What is a Thread?
  - Multiple threads can be spawned from a single process
  - Threads share their process address space
  - At a single point in time, their will be multiple points of execution in a program
  - Threads can be actually executing concurrently or in round robin fashion depending on system design. However, without the user noticing that.

Introduction

- Life is simpler without concurrency. So, why use threads? 
  - Take advantage of multiprocessors
  - Driving slow devices (e.g. disks, networks, printers…etc). Instead of waiting for the job to finish, do something else.
  - Handling lengthy user tasks while still being able to interact with the user
  - Distributed systems (e.g. a server handling concurrent users requests)

C# Specifics

- System.Thread namespace provides classes and interfaces to support multithreaded programming
- Examples of classes:
  - Thread
  - Monitor
  - Mutex
  - ThreadPool
- We can serve basic threading requirements using the *Lock Statement*, the *Thread Class* and the *Monitor Class*
- Nevertheless, I'll discuss others to ensure good coverage

Basic Threads Operations

1. Creation of a thread:
   - In C# you create a thread by creating an object from the Thread Class & giving its constructor a “ThreadStart” delegate.
   - Delegate is an object created from an object and its method

- Code example:

```
Thread t = new Thread(new ThreadStart(foo.A));
t.Start();
foo.B();
t.Join();
```

foo.A & foo.B
execute concurrently

Basic Threads Operations

2. Mutual Exclusion:
   - Threads need to access same resources
   - It is the programmer's responsibility to avoid risks arising from such requirement
   - We will achieve mutual exclusion by using the Lock Statement which provides a means for creating critical sections

     Lock (statement) {embedded code}
   - The critical section is a region of code where only a single thread can execute at a time

Basic Threads Operations

- Continuing on Mutual Exclusion:
  - The general rule is to lock an object and update it's shared variables within the critical section of that lock statement.
  - However, C# doesn't impose any restrictions in this regard. You can lock an object and update another's instant variables!!
  - Not following the general rule will certainly lead to almost impossible to support code

Basic Threads Operations

- Continuing on Mutual Exclusion:
  - In OO language, shared variables can take two forms:
    - Instance variables of an object
    - Static variables of a class
  - Hence, we need to have a mechanism for locking each
  - For instance variables use object name or *this*
  - For static variables use *typeof(class name)* clause

Basic Threads Operations

- Continuing on Mutual Exclusion:

```
class KV {
string k, v;
public void SetKV(string nk, string nv) {
lock (this) { this.k = nk; this.v = nv; }
              }
…
}
```

Locks the corresponding object from KV

```
static KV head = null;
KV next = null;
public void AddToList() {
lock (typeof(KV)) {
System.Diagnostics.Debug.Assert(this.next == null);
this.next = head; head = this;
              }
}
```

Locks KV class

Basic Threads Operations

3. Waiting For a Condition:
   - Lock is so simple: one thread at a time
   - We need a more complex scheduling mechanism for the threads calling a locked object
   - The Monitor Class allows the synchronization between different threads calling an object
   - We will focus on three methods provided by the Monitor Class

Basic Threads Operations

- Continuing on waiting for a condition:
  - The three methods and their effect:
    - Monitor.Wait(Object obj):

      Unlocks the object and blocks the thread.
    - Monitor.Pulse(Object obj):

      Awaken a single thread waiting for the object
    - Monitor.PulseAll(Object obj):

      Awaken all the threads waiting for the object

Basic Threads Operations

4. Interrupting a thread:
   - The Interrupt Method in Thread Class is used to awaken a thread blocked in a long-term wait
   - If a thread "t" has performed a Monitor.Wait() on a certain object, another thread can call t.interrupt() to let "t" resume execution

```
public sealed class Thread {
        public void Interrupt() { … }
        …
}
```

Putting Basic Operations Together

- After we discussed the four basic operations, let us have an example:
  - A linked list calss methods of getFromList() & addToList() that can run on two separate threads
  - Both operations need to be mutually exclusive
  - We need getFromList() to execute only if the list is non empty
  - We need addToList() to notify a waiting getFromList() in case an item is added

Putting Basic Operation

If you find the LL empty, unlock the object and wait

- Continuing on the example:

```
public static KV GetFromList(){
        KV res;
        lock (typeof(KV)) {
          while (head == null)Monitor.Wait(typeof(KV));
           res = head; head = res.next;
        res.next = null; // for cleanliness
        }
return res;
}
```

```
public void AddToList() {
        lock (typeof(KV)) {
        /* We're assuming this.next == null */
        this.next = head; head = this;
        Monitor.Pulse(typeof(KV));
        }
}
```

Mutual Exclusion using locks

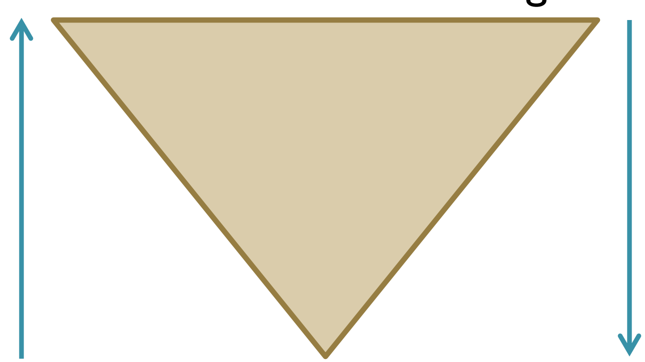
Once you add an item, notify a possibly waiting thread

Putting Basic Operations Together

- Continuing on the example:
  - Is it ok to let the thread blocks forever waiting for a new item to be added to the LL?
  - No, we can use interrupt.
  - An example for using interrupt is to handle a user clicking cancel on a thread blocked by Monitor.Wait()
- Our basic operations for threading can be done using Lock statement, Thread Class and Monitor Class

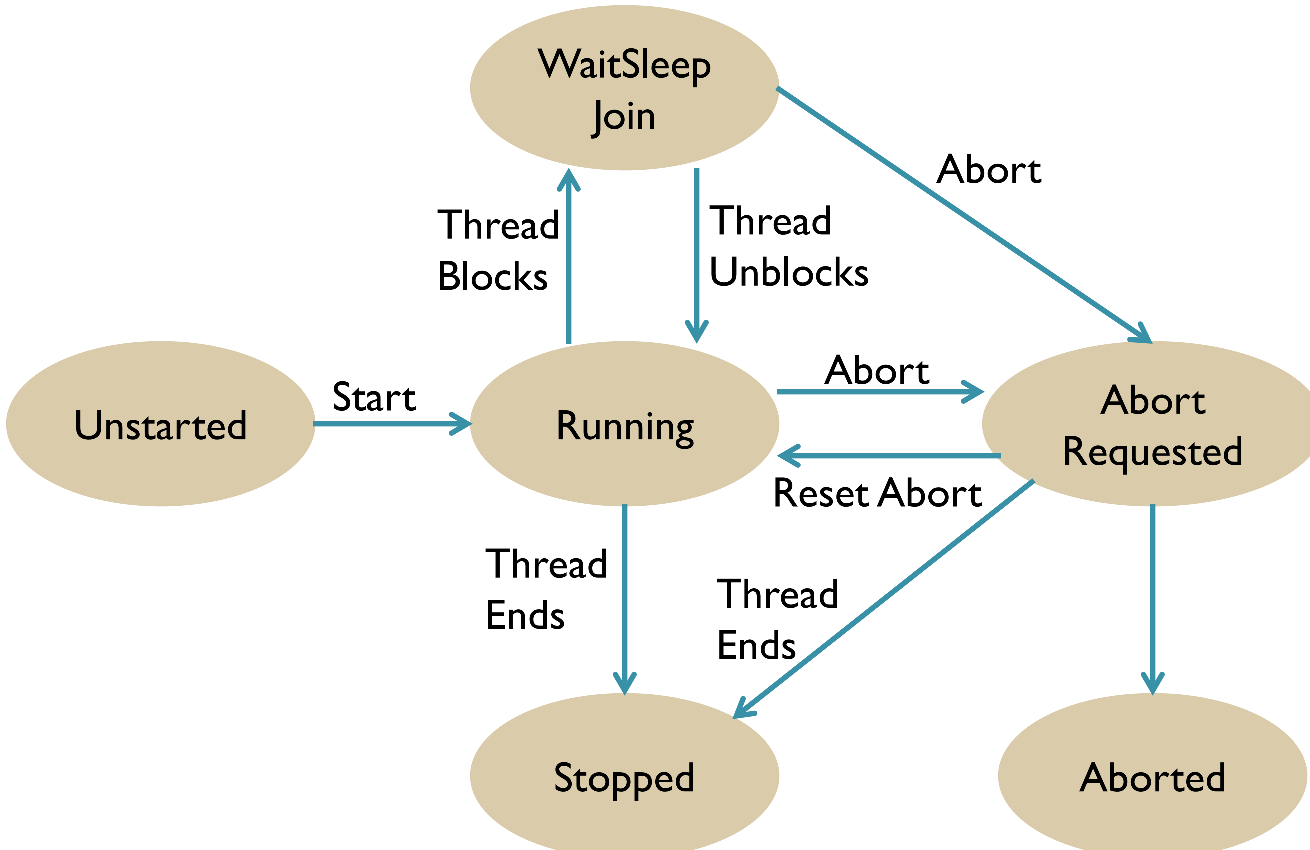
Locking Granularity

- As we discussed earlier, we use locks for "all instance object fields" or "all class fields"
  - Why not locking at field level rather than object or class level?



  - C# try to balance by locking at object level.

C# Thread's Status:



Taken From:
Threading in C# (see references)

C# Thread's Properties

- Is.Background:
  - When selected the thread terminates automatically when the process terminates (i.e. all foreground threads terminates)
- Thread's Priority:
  - Can take the following values:
    - Lowest
    - BelowNormal
    - Normal (default)
    - AboveNormal
    - Highest
  - Used by the OS to schedule threads

Thread Class Important Methods

- Before going deeper in System.Thread namespace, let's specify some other methods that are important in the thread class:

Method Name	Method description
Start()	Run a thread that is created and waiting in the un-started state
Join()	Block the main thread, until the specified thread finishes execution
Sleep()	Suspend the thread for a specified period of time
Suspend()	Change the status of a running thread to suspended
Resume()	Change the status of a suspended thread to running

Mutex Class

- Let's now go more deep in C# System.Thread classes. We will first look at the Mutex Class
- Mutex is used to synchronize access to shared resources
- In case a thread already acquired the mutex, another thread requiring the mutex will be blocked until the mutex is released
- But, how is Mutex different than using Monitor??

Mutex Class

- Unlike monitor, mutex can be used for inter-process synchronization.
- Mutex can be of two types:
  - Local Mutex (also called unnamed mutex): this type is used to synchronize threads within a process
  - Gloabal Mutex (also called named mutex): this type is used to synchronize inter-process threads (at OS level)

Mutex Class

- Differentiating the two occurs during creation:
  - If you create a mutex without giving it a name it is a local mutex. Example:

```
private static Mutex mut = new Mutex();
```

  - If you create a mutex and specify a name for it, the OS links it with an OS mutex with that name.

Mutex name

```
private static Mutex mut = new Mutex(Boolean, String);
```

Mutex Class

- Microsoft recommends using monitor for inter-thread communication and mutex for inter-process communication
- The reason is that mutex implementation is heavy
- Abandoned Mutex:
  - A mutex must be released using MutexRelease() method before the thread ends. Otherwise, it is said to be an abandoned Mutex and will throw an exception
  - An abandoned mutex questions the integrity of the data being protected by the mutex

Semaphore Class

- Almost similar to the Monitor Class, the only difference is that a semaphore defines the maximum number of threads to access the resource rather than restricting it to one thread
- Below an example of defining a semaphore:

```
private static Semaphore _pool = new Semaphore(0, 3);
```

You can reserve some entries during creation which is 0 here

Number of allowed concurrent entries here is 3

Semaphore Class

- A semaphore is acquired by calling Wait() method and released by calling Release()
- Whenever a semaphore is acquired the remaining number of slots to access the resource is decremented. When the number is 0, the calling thread will be suspended.
- No ordering (e.g. FIFO or LIFO) for suspended threads

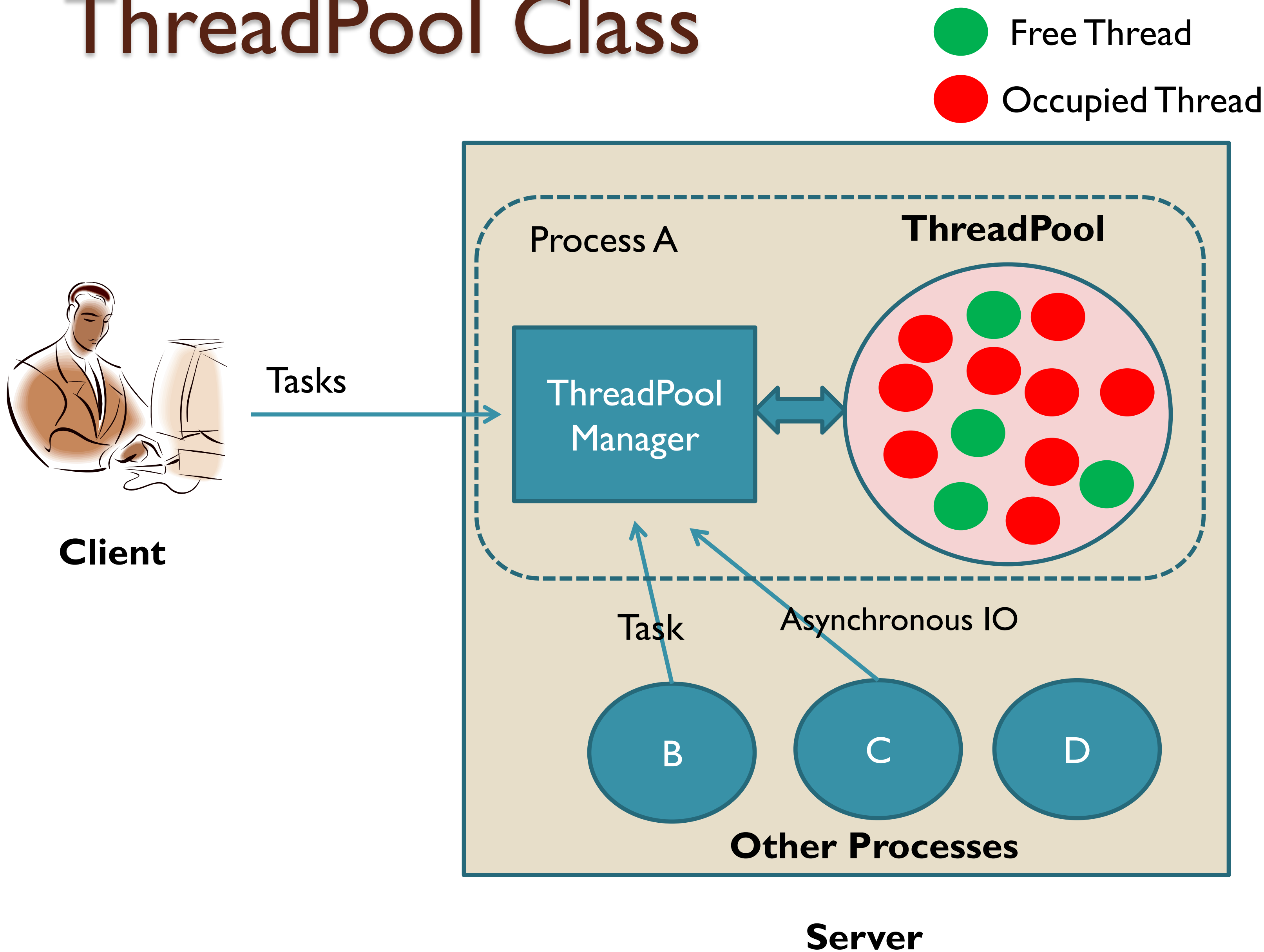
ReaderWriterLock Class

- In terms of synchronization, having multiple reads doesn't impose any security risks. However, a single write can create a problem.
- This class provides a ready made locking mechanism where at a single point in time we have either:
  - One thread writing
  - Multiple threads reading
- This provides a better throughput when compared the use of monitor class

ThreadPool Class

- Provides an automated threads management framework inside a process
- Consider the example of a server handling client requests
- A received request (i.e. task) will be assigned a thread from the threadpool without interrupting the main thread
- The pool manager can recycle threads
- Threads in the threadpool are background threads

ThreadPool Class



ThreadPool Class

- Important methods in ThreadPool Class:

Method Name	Method description
QueueUserWorkItem()	Queue the delegate item for the next available thread in the ThreadPool
SetMaxThreads()	Defines the maximum number of threads that can be handled concurrently by the pool
GetAvailableThreads()	Returns the number of free threads in the ThreadPool

Conclusion

- Conceptually, we divided threads into basic operations and more advanced ones
- If only small threading requirements are required the basic operations will be enough
- To build complex systems (e.g. client\server with concurrent users) look for the more complex ones
- System.Thream namespace contains large number of classes with different methods. But, their unique goal is to support multithreading

Resources

- Birrell A. An introduction to programming with C# threads. *Technical Report MSR-TR-2005-68*, Microsoft Research, Redmond May 2005.
- ***msdn****.microsoft.com*
- *Threading in C#, Joseph Albahari*

Where to look for additional info?

- ***msdn****.microsoft.com*

Microsoft provides a complete guide through their above microsoft developers network website about the System.Threading namespace.