



# C# Programming in Depth

Prof. Dr. Bertrand Meyer

March 2007 - May 2007

## Lecture 8: Threads

Lisa (Ling) Liu

# What is a thread?

---



- A thread is an independent execution path, able to run simultaneously with other threads
- A C# program starts in a single thread created automatically by the CLR and operating system (the "main" thread), and is made multithreaded by creating additional threads
- CLR assigns each thread its own memory stack so that local variables are kept separate
- Threads share data if they have a common reference to the same data

# How threading works

---



- Multithreading is managed internally by a **thread scheduler**, a function the CLR typically delegates to the operating system.
- On a single-processor computer, a thread scheduler performs **time-slicing** - rapidly switching execution between each of the active threads
- 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.

# Threads vs. Processes

---



- All threads within a single application are logically contained within a process - the operating system unit in which an application runs
- The key difference between threads and processes:
  - Processes are fully isolated from each other;
  - Threads share memory with other threads running in the same application

# Why use concurrency?

---



- Making use of multiprocessors
- Driving slow devices, such as disks, networks, terminals and printers
- Achieving timely response to the GUI's users
- Building a distributed system

# Thread facilities

---



- Thread creation
- Mutual exclusion
- Waiting for events
- Getting a thread out of an unwanted long-term wait

## C# facility:

- The System.Threading namespace
- C# lock statement

# Thread creation

---



1. Create a type method to be the entry point for the new thread
2. Create a new `ParameterizedThreadStart` (or `LegacyThreadStart`) delegate, passing the address of the method defined in step 1 to the constructor
3. Create a `Thread` object, passing the `ParameterizedThreadStart` / `ThreadStart` delegate as a constructor argument
4. Establish any initial thread characteristics (name, priority, etc.)
5. Call the `Thread.Start()` method. This starts the thread at the method referenced by the delegate created in step 2 as soon as possible



# Starting a thread

```
using System.Threading;  
Printer p = new Printer();  
int[] a = { 1, 2, 3, 4, 5, 6,7,8,9,10};
```

```
Thread myThread = new Thread (new ParameterizedThreadStart  
    (p.PrintNumbers));
```

```
myThread.Name = "Secondary";  
myThread.Start (a);
```

Function executed in the created thread

Establish the thread characteristics

```
public class Printer  
{  
    public void PrintNumbers(object a)  
    {  
        ...  
    }  
    ...  
}
```

Take object argument, no return value

# Foreground and background threads



```
using System.Threading;
```

```
...
```

```
myThread.Start (a);
```

← a foreground thread prevents the current application from terminating

```
using System.Threading;
```

```
...
```

```
myThread.IsBackground = true;
```

```
myThread.Start (a);
```

```
myThread.Join();
```

← a background thread can be terminated after all foreground threads terminates

← Blocking calling thread until it terminates

# Thread safety

---



The simplest way that threads interact is through access to shared memory. Thread safety means the shared memory is always in a correct state even when used concurrently by multiple threads

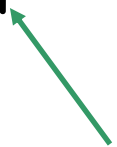
To achieve thread safety we need to synchronize the threads accessing the shared memory

In C#, this is achieved with the class "Monitor" and the language's "lock" statement

# Thread synchronization: lock

---



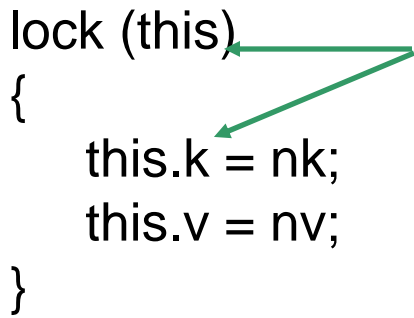
- Specify for a critical section that can only be executed by one thread at any time
- Syntax:  
`lock (expression) embedded-statement`  


Can be any object
- The "lock" statement locks the given object, then executes the contained statements, then unlock the object.

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

shared variables are instance fields,  
lock object

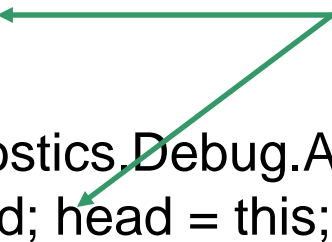
critical section



```
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;  
    }  
}
```

shared variables static fields, lock the  
type of the class



# Thread synchronization: Monitor type



- The *C#* `lock` statement is really just a shorthand notation for working with the `System.Threading.Monitor` type

```
public class KV
{
    string k;
    public void SetK (string nk)
    {
        lock (this)
        {
            this.k = nk;
        }
    }
}
```



```
public class KV
{
    string k;
    public void SetK (string nk)
    {
        Monitor.Enter(this);
        try
        {
            this.k = nk;
        }
        finally
        { Monitor.Exit (this);}
    }
}
```

# Waiting for a condition

---



- Resource scheduling policy embodied by lock() :
  - Only one thread can access the shared resources at a time
  
- Requirement for more complicated resource scheduling policy
  - Allow a thread to block until some condition is true
    - In C# and Java, there is no separate type for this mechanism
    - Every object inherently implements one condition variable
    - The "Monitor" class provides static "Wait", "Pulse" and "PulseAll" methods to manipulate an object's condition variables

# Manipulate condition variables


---



- A thread that calls "Wait" must already hold the object's lock
- The "Wait" operation automatically unlocks the object and blocks the thread (a thread is blocked in this way is said to be "waiting on the object")
- The "Pulse" method awakes at least one thread that is waiting on the object (possibly more than one)
- The "PulseAll" method awakes all threads that are awaiting on the object
- When a thread is awoken it relocks the object

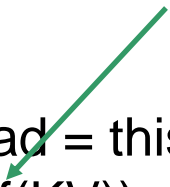
```
public static KV GetFromList()
{
    KV res;
    lock (typeof(KV))
    {
        while (head = null) Monitor.Wait (typeof(KV))
        res = head; head = res.next;
        res.next = null
    }
    return res;
}
```

unlock typeof(KV) and blocks



```
public void AddToList()
{
    lock (typeof(KV))
    {
        this.next = head; head = this;
        Monitor.Pulse(typeof(KV));
    }
}
```

wake up a thread that was waiting for the locked variable





# Interrupting a thread

---

- Interrupt a thread to bring it out of a long-term wait
- Calling `Interrupt` on a blocked thread forcibly release it, throwing a `ThreadInterruptedException`.

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

# Using locks: accessing shared data

---



**Basic rule:** in a multi-threaded program all shared mutable data must be protected by associating it with some object's lock, and you must access the data only from a thread that is holding that lock (i.e., from a thread executing a "lock" statement that locked the object).

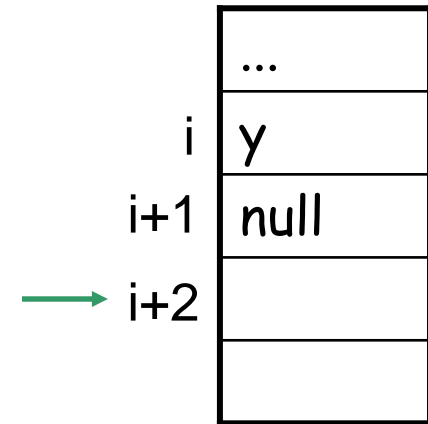
# Unprotected data



The simplest bug related to locks occurs when you fail to protect some mutable data and then you access it without the benefits of synchronization.

```
public class Table
{
    object[] table = new object [1000]
    int i = 0;

    public void Insert (object obj)
    {
        if (obj != null)
        {
            table[i] = obj;
            i++;
        }
    }
}
```



thread A: Insert (x) ← thread B: Insert (y)  
← thread A: Insert (x) ← thread B: Insert (y)

```
public class Table
{
    object[] table = new object [1000]
    int i = 0;

    public void Insert (object obj)
    {
        if (obj != null)
        {
            lock (this)
            {
                table[i] = obj;
                i++;
            }
        }
    }
}
```

# Locking granularity

---



Simple and coarse rule:

Use object instance's lock to protect all the instance fields of a class

Use `typeof(theClass)` to protect the static fields

# Deadlock



- A deadlock is a bug when two threads are trying to access resources, which are locked by each other.

```
public class DeadLock
{
    static object a = new object();
    static object b = new object();

    public static void Get()
    {
        lock (a)
        {
            lock (b)
            {
                ...
            }
        }
    }
}
```

...

```
public static void Give()
{
    lock (b)
    {
        lock (a)
        {
            ...
        }
    }
}
...
}
```

**To avoid this deadlock...**



- Have a partial order for the acquisition of locks in your program
  - Can avoid deadlocks involving only locks
- Partition the locked data into smaller pieces protected by separate locks

But:

The smaller of your lock granularity, the more complicated your locking becomes, and the more likely you are to become confused about which lock is protecting which data, and end up with some unsynchronized access to shared data.

# Poor performance through lock conflicts

---



- Whenever a thread is holding a lock, it is potentially stopping another thread from making progress
- When the thread that is holding a lock ceases to make progress, the total throughput of your program is degraded
- Protect different fields of an object with different locks, in order to get better efficiency by accessing them simultaneously from different threads

```

public class F
{
    static F head = null; //protected by typeof(F)
    string myName;        //immutable
    F next = null;       //protected by typeof(F)
    D data;               //protected by "this"

    public static F Open (string name)
    {
        lock (typeof(F)) ← use one lock for operations
                           on global file list
        {
            for (F f = head; f != null; f = f.next)
            {
                if (name.Equals(f.myName)) return f;
            }

            //Else get a new F, enqueue it on head and return it
            return ...;
        }
    }

    public void Write (F f, string msg) ← lock per file for operations
                                         only affecting that file
    {
        lock (this) { ...} // access "f.data"
    }
}

```

# Using wait and pulse: scheduling shared resources

---



- Using Wait  
`while (!expression) Monitor.Wait(obj);`
- Using PulseAll
  - PulseAll is useful in the schedule policy known as shared/exclusive locking (or readers/writers locking)

# readers/writers locking policy example



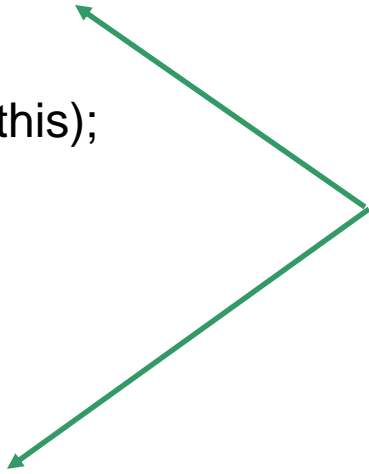
```
public class RW {
    int i = 0; //protected by this
    public void AcquireExclusive () {
        lock (this) {
            while (i != 0) Monitor.Wait (this);
            i = -1;
        }
    }
    public void ReleaseExclusive() {
        lock (this) {
            i = 0;
            Monitor.PulseAll (this);
        }
    }
    ...
}
```

Any thread wanting to modify the data calls “AcquireExclusive”, then modifies the data, then calls “ReleaseExclusive”

```
public void AcquireShared () {  
    lock (this) {  
        while (i < 0) Monitor.Wait (this);  
        i ++;  
    }  
}
```

```
public void ReleaseShared() {  
    lock (this) {  
        i --;  
        if (i == 0) Monitor.Pulse(this);  
    }  
}
```

```
}// class RW
```



Any thread wanting to read the data calls "AcquireShared", then modifies the data, then calls "ReleaseShared"

When the variable "i" is greater than zero, it counts the number of active readers. When it is negative there is an active writer. When it is zero, no thread is using the data

# Starvation



- The resource scheduling decisions should be fair so that all threads are equal or some more to access the resources
- Starvation: some thread will *never* make progress. Consider following thread schedule in readers/writers locking example :

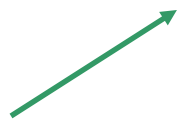
Thread A calls "AcquireShared"; i = 1;

Thread B calls "AcquireShared"; i = 2;

Thread A calls "ReleaseShared"; i = 1;

Thread C calls "AcquireShared"; i = 2

Thread B calls "Release Shared"; i = 1;



a blocked potential writer thread is starving

...

```
int writeWaiters = 0;
public void AcquireExclusive () {
    lock (this) {
        writeWaiters++;
        while (i != 0) Monitor.Wait (this);
        writeWaiters--;
        i = -1;
    }
}
```

```
public void AcquireShared () {
    lock (this) {
        if (writeWaiters > 0) Monitor.Wait (this);
        while (i < 0) Monitor.Wait (this);
        i ++;
    }
}
```

# Deadlock



- You can introduce deadlocks by waiting on objects, even though you have been careful to have a partial order on acquiring locks

```
public class DeadLock
{
    static object a = new object();
    static object b = new object();
    static bool ready = false
```

```
public static void Get() {
```

```
    lock (a) {
```

```
        lock (b) {
```

```
            while (!ready) Monitor.Wait (b);
```

```
        }
```

```
    }
```

```
}
```

```
...
```

unlock b, but a is still  
locked

```
public static void Give()  
{  
    lock (b)  
    {  
        lock (a)  
        {  
            ready = true;  
            Monitor.Pulse(b);  
        }  
    }  
}  
...  
}
```

deadlock happens when  
Thread A call Get;  
Thread B call Give;

# Pipelining

---



- Build a chain of producer-consumer relationships, known as **pipeline**, for example (three-stage pipeline):
  - Thread A initiates an action, all it does is enqueue a request in a buffer
  - Thread B takes the action from the buffer, performs part of the work, then enqueues it in a second buffer
  - Thread C takes it from there and does the rest of the work

## Using interrupt: diverting the flow of control

---



- The purpose of the "Interrupt" method of a thread is to tell the thread that it should abandon what is doing, and let control return to a higher level abstraction, presumably the one that make the call of "interrupt".

# Reference

---



- Andrew D. Birrell, *An introduction to programming with C# threads*