Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C#: concurrency
Outline

- C# threads
  - thread implementation
  - sleep and join
  - threads that return values
- Thread synchronization
  - implicit locks and synchronized blocks
  - producer/consumer example
- More efficient concurrency
  - thread pools
  - atomic integers
- Other concurrency models
  - asynchronous programming
  - polyphonic C#
Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C# threads
C# threads

C#'s concurrency model is based on threads

Threads are created by instantiating class `Thread`

- The constructor takes a `ThreadStart delegate` that wraps the method which the thread will execute

Any method can be called with the delegate mechanism

- Unlike Java, any existing class can be used for multi-threaded execution without modifications

In all the examples, assume

```csharp
using System; using System.Threading;
```
A simple class (to be threaded)

```java
public class DumbClass {

    private String id;

    public DumbClass(String id) {
        this.id = id;
    }

    public void print_id() {
        // do something
        Console.WriteLine("This is " + id);
    }

}
```
Creating and starting a thread

Create the object with the method the thread will execute

```csharp
DumbClass db = new DumbClass("db");
```

Create a `Thread` object and pass method `print_id` to is using a `ThreadStart` delegate

```csharp
Thread mt = new Thread(
    new ThreadStart(db.print_id));
```

Start the thread

```csharp
mt.Start();
```

Optionally, wait for it to terminate

```csharp
mt.Join();  // wait until mt terminates
Console.WriteLine(
    "The thread has terminated");
```
Putting a thread to sleep

The `Sleep(int t) static` method suspends the thread in which it is invoked for \( t \) milliseconds

```java
Thread.Sleep(2000); // suspend for 2 seconds
```

- the timing may be more or less precise according to the real-time guarantees of the executing environment
Threads that return values

Threads can return values using additional delegates

- E.g., to have threads that return strings declare a delegate type:
  ```java
  public delegate void delForStrings(String s);
  ```
- A class stores a reference to the delegate and activates it when appropriate (to pass values to the caller)

```java
public class DullClass {
    private String id;
    // delegate used to return a value when terminating
    private delForStrings d;
    // the constructor must bind the actual method
    public DullClass(String id, delForStrings d) {
        this.id = id;  this.d = d;
    }
    public void give_id() {
        // call the delegate to return the value id
        if (d != null) { d(id); }
    }
}
```
Creating threads that return values

Define a method to process the information returned by the thread (its signature matches the delegate’s)

- for simplicity, we make it static

```csharp
public static void printValueSent(String s) {
    Console.WriteLine(“The thread sent: “ + s);
}
```

Create the object with the method the thread will execute and pass the delegate to it

```csharp
DullClass dl = new DullClass(“dl”,
    new delForString(printValueSent));
```
Creating threads that return values

Create a Thread object and pass method give_id to is using a ThreadStart delegate:

```csharp
Thread t = new Thread(
    new ThreadStart(dl.give_id));
```

Start the thread:

```csharp
t.Start();
```

After executing, it will invoke printValueSent through the delegate, which will print the given id.
Java and C# in depth
Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Thread synchronization
Synchronization with locks

The `lock` statement supports synchronization based on locks

- blocks of statements guarded by `lock(o)`
- the lock `o` itself can be any object (including `this`)
- locking/unlocking is implicit when entering/exiting the block
- useful to define critical regions and fine-grained synchronization
- monitors are implemented by locking the whole method body on `this`
Synchronization with locks

The `lock` statement supports synchronization based on locks

```java
// s must be accessed in mutual exclusion
private int s;

// dict is a read-only object, no concurrency problems
private List<String> dict;

public String decrement_and_lookup() {
    // critical region
    lock(this) {
        if (s > 0) { s = s - 1; }
    }
    // non-critical region
    return dict.Item(s);
}
```
Coordination with signals

Locked threads can communicate with signals, implemented as static methods of class `Monitor`:

- `Monitor.Wait(o)` : suspend and release the lock on `o` until some thread does a `Pulse(o)` or `PulseAll(o)`
- `Monitor.Pulse(o)` : resume one suspended thread (chosen nondeterministically) waiting on object `o`, which becomes ready for execution when possible
- `Monitor.PulseAll(o)` : resume all suspended threads waiting on object `o`, which become ready for execution when possible

- Analogues of Java’s `wait`, `notify`, `notifyAll`
Coordination with events

A more fine-grained (and possibly efficient) coordination uses services of the `WaitHandle` class to coordinate threads.

Coordination events are in one of two states: **signaled** and **unsignaled**

- Method `Set` puts an event in the signaled state
  - that is, it issues the signal
- Method `Reset` puts an event in the unsignedaled state
  - that is, it cancels the signal
Coordination with events

A more fine-grained (and possibly efficient) coordination uses services of the *WaitHandle* class to coordinate threads

Two main classes implement coordination events

- **AutoResetEvent**
  - automatically resets to unsignaled after being received by one of the waiting threads

- **ManualResetEvent**
  - does not automatically reset, hence it can be received by more than one waiting thread
  - can be reset with method `Reset()`
Coordination with events

Use services of the `WaitHandle` class to coordinate threads.

A thread can block waiting for an event using some methods of the class:

- `WaitHandle.WaitOne()` waits for the event to be signaled (and blocks until then).
- `static WaitHandle.WaitAny(WaitHandle[] e)` waits for any of the events in array `e`.
  - The method returns when an event is received.
  - It returns an integer `i`, which is an index within array `e`.
  - `e[i]` is the event that has been received.
- `static WaitHandle.WaitAll(WaitHandle[] e)` waits for all the events in array `e` to be signaled.

Unlike `Monitor.Wait`, if these wait primitives occur in a lock block they do not release the lock while waiting.
The producer-consumer problem

Two threads, the **Producer** and the **Consumer**, work concurrently on a shared **Buffer** of bounded size.

**The Producer puts new messages in the buffer**
- if the buffer is full, the Producer must wait until the Consumer takes some messages
- the Producer also signals the last message

**The Consumer takes messages from the buffer**
- if the buffer is empty, the Consumer must wait until the Producer puts some new messages
- the Consumer terminates after the last message

Consistent access to the **Buffer** requires locks and synchronization.

One way is to define critical regions when accessing the buffer data structure (with `lock`) and signal events.
The main class

```java
public class ProducerConsumer {

    public static void main(String[] args) {
        // create a synchronizer object
        Synchronizer s = new Synchronizer();
        // create a buffer of size 3
        Buffer b = new Buffer(3, s);
        // create producer and consumer
        Producer p = new Producer(b, s);
        Consumer c = new Consumer(b, s);
        // instantiate threads
        Thread pT = new Thread(p.produce);
        Thread cT = new Thread(c.consume);
        // start them
        pT.Start();  cT.Start();
    }
}
```
using System; using System.Threading;  
using System.Collections;  
using System.Collections.Generic; 

public class Synchronizer {  
    private EventWaitHandle takeEvent;  
    public EventWaitHandle TakeEvent  
    { get { return takeEvent; } } 

    private EventWaitHandle giveEvent;  
    public EventWaitHandle GiveEvent  
    { get { return giveEvent; } } 

    private EventWaitHandle endEvent;  
    public EventWaitHandle EndEvent  
    { get { return endEvent; } } 
}
public Synchronizer() {
    // events initialized to unsignaled state
    takeEvent = new AutoResetEvent(false);
    giveEvent = new AutoResetEvent(false);
    endEvent = new ManualResetEvent(false);
}

- **takeEvent** is an **AutoResetEvent** so it is received by exactly one waiting thread among all those waiting for a take to happen.
- **giveEvent** is an **AutoResetEvent** so it is received by exactly one waiting thread among all those waiting for a give to happen.
- **endEvent** is a **ManualResetEvent** so it is received by all waiting threads: they will all be notified that they can terminate.
public class Buffer {

    public Buffer(int max_size, Synchronizer s) {
        this.max_size = max_size;
        this.messages = new Queue<String>();
        this.s = s;
    }

    // buffer of messages, managed as a queue
    private Queue<String> messages;

    // maximum number of elements in the buffer
    private int max_size;

    // reference to events for synchronization
    private Synchronizer s;
}
public String take() {
    if (messages.Count == 0) {
        // only one thread receives the event
        WaitHandle.WaitAny(
            // wait until a give occurs
            new WaitHandle[] { s.GiveEvent });
    }
    // now the buffer is not empty
    lock(this) {
        m = messages.Dequeue();
    }
    // signal that a take has occurred
    s.TakeEvent.Set();
    return m;
}
The shared Buffer (3/3)

```csharp
public void give(String msg) {
    if (messages.Count == max_size) {
        // only one thread receives the event
        WaitHandle.WaitAny(
            // wait until a take occurs
            new WaitHandle[] {s.TakeEvent});
    }
    // now the buffer has at least an available slot
    lock(this) {
        messages.Enqueue(msg);
    }
    // signal that a give has occurred
    s.GiveEvent.Set();
}
```
public class Producer {

    // a reference to the shared buffer
    private Buffer b;

    // events to synchronize on
    private Synchronizer s;

    // set the reference to the buffer and synchronizer
    public Producer(Buffer b, Synchronizer s) {
        this.b = b;
        this.s = s;
    }
}
public void produce() {

    // work for 20 turns
    for (int i = 0; i < 20; i++) {
        // put a message in the buffer
        b.give(i.ToString());
    }

    // signal that production has ended
    s.EndEvent.Set();
}
The Consumer (1/2)

```java
public class Consumer {

    // a reference to the shared buffer
    private Buffer b;

    // events to synchronize on
    private Synchronizer s;

    // set the reference to the buffer and synchronizer
    public Producer(Buffer b, Synchronizer s) {
        this.b = b;
        this.s = s;
    }
}
```
public void consume() {
    // loop as new events arrive, until:
    // EndEvent is signaled AND b is empty
    while ( WaitHandle.WaitAny( new WaitHandle[]
        {s.GiveEvent, s.EndEvent}) != 1
        || !b.Empty ) {
        string m = b.take();
        Console.WriteLine(
            "Consumer has consumed message " + m );
    }
}
Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

More efficient concurrency
Concurrency and performance

Thread creation is time-consuming

- massive thread creation can annihilate responsiveness
- C#’s solution: thread pools

Lower-level primitives are available

- **Mutex** class for mutexes
  - less efficient than monitors and **lock** (unlike Java)
- **Interlocked static** class
  - atomic operations on integers

**Tip**: don’t forget the efficiency/abstraction trade-off
Thread pools

Thread pools are an efficient way of running multi-threaded applications

- maintain a pool of worker threads
- when a client requests a new task to run, preempt one of the available worker threads and assign it to the task
- no creation overhead upon task invocation

C#'s static class `System.Threading.ThreadPool`

- `QueueUserWorkItem(WaitCallback w, Object o)`: schedule delegate `w` for execution by a worker thread, when possible; `o` is passed as argument to `w`.
Create a wrapper delegate for each method to be threaded

- In the Producer/Consumer example:

```csharp
public static void Main(string[] args) {
    Producer p = new Producer(b, s);
    Consumer c = new Consumer(b, s);
    ThreadPool.QueueUserWorkItem(new
        WaitCallback(consuming), c);
    ThreadPool.QueueUserWorkItem(new
        WaitCallback(producing), p);
}

public static void consuming(object o)
{ ((Consumer) o).consume();  }

public static void producing(object o)
{ ((Producer) o).produce();  }
```

There’s an undesirable side-effect with this code as is. What is it?
Create a wrapper delegate for each method to be threaded

- In the Producer/Consumer example:

```
public static void Main(string[] args) {
    Producer p = new Producer(b, s);
    Consumer c = new Consumer(b, s);
    ThreadPool.QueueUserWorkItem(new
        WaitCallback(consuming), c);
    ThreadPool.QueueUserWorkItem(new
        WaitCallback(producing), p);
}
```

There’s an undesirable side-effect with this code as is. What is it?

- **Main** terminates after invoking **QueueUserWorkItem**; hence the **ThreadPool** object is deallocated and the worker threads forcefully terminated!
Interlocked class

C#'s implementation of atomic operations on integers

// shared variable
int s;

...

// this is equivalent to an atomic s++
Interlocked.Increment(ref s);

...

// this is equivalent to an atomic s--
Interlocked.Decrement(ref s);
Java and C# in depth
Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Other concurrency models:
Asynchronous programming
Concurrency and correctness

Programming thread-safe data structures is error-prone
  - Thread-safe collections are available since C# 4.0
  - Current collections provide a `SyncRoot` object for synchronization

Threads and monitors are too general for straightforward parallel computation
  - C#’s solution: asynchronous methods

Tip: don’t forget the efficiency/abstraction trade-off
Asynchronous programming

C# 5.0 introduced simple mechanisms to have methods execute asynchronously and wait for one another.

The model is based on asynchronous methods:

```csharp
async Task<T> DoAsync()
```

- `DoAsync` may execute asynchronously from its clients
- In turn, its clients can wait for `DoAsync`’s to complete (and only then access its result).

(The class `Task` can also be used independent of asynchronous methods, mostly to introduce forms of data-bound parallelism.)
Asynchronous methods

```csharp
async Task<T> DoAsync()
```

Asynchronous methods:
- Are declared as such with the keyword `async`
- Can have only specific return types:
  - `Task<T>` for methods returning values of type `T`
  - `Task` for methods returning no values
  - `void` for methods returning no values used as event handlers
- Cannot have `ref` or `out` arguments (there’s no way to “wait” for those)
- By convention, have name ending in “Async”
- Can wait for other asynchronous methods to complete using the `await` instruction in their bodies.
async Task<T> DoAsync()

When an asynchronous method **DoAsync** executes an **await**:

- Control **may** return to the caller (the compiler/runtime decides if a context switch is worth the cost)
- The caller will be able to retrieve the result later when available, after **awaiting**
- No new thread is created: the asynchronous computation uses the thread executing **DoAsync**

The result obtained when **awaiting** for an asynchronous method with return type **Task<T>** has type **T**.
Asynchronous programming: example

Write a method `AvgAgesAsync` that computes the average age of the population of several cities.

The data for each city is accessible remotely using a library method:

```csharp
async Task<List<int>> GetAgesAsync(String city)
```
A call return a list of ages, one for each person of the city.

Calls to `AvgAgesAsync` may take time, but can be executed asynchronously:

1. First, the client start the asynchronous computation:
   ```csharp
   Task<double> t = AvgAgesAsync(listOfCities);
   ```
2. Now, the client can do other stuff while `AvgAgesAsync` executes in parallel.
3. Eventually, the client will get the final results with a call:
   ```csharp
double avg = await t;
```
async Task<double> AvgAgesAsync (List<string> cities) {
    int i = 0, pop = 0; double avg = 0;
    foreach (string c in cities) {
        // wait for results from GetAgesAsync
        // (but AvgAgesAsync’s caller needn’t block)
        List<int> v = await GetAgesAsync(c);
        avg = // new average, from old one
              ((avg * pop) + v.Sum()) / (pop + v.Count);
        pop += v.Count; // new total population
        i++; // one more city done
        Console.WriteLine(
            “Done {0}% of cities. Current average: {1}”,
            (i / cities.Count * 100), avg);
    } return avg;
}
Java and C# in depth
Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Other concurrency models: Polyphonic C#
Introducing Polyphonic C#

- Polyphonic C# is an extension of C# with a few high-level primitives for concurrency
  - not part of .NET framework
  - based on join calculus (Fournet & Gonthier, 1996)
  - taken up by Microsoft’s Cω project
- JoinJava is a similar extension for Java
- Based on two basic notions
  - Asynchronous methods
  - Chords

(M. Mussorgsky, Pictures at an exhibition)
Polyphonic asynchronous methods

Calls to asynchronous methods return immediately without returning any result

- The callee is scheduled for execution in a different thread
- similar to sending a message or raising an event
- declared using `async` keyword instead of `void`

```csharp
public async startComputation () {
    // computation
}
```

- asynchronous methods do not return any value and cannot have `ref` or `out` arguments
Chords: syntax

A chord is an extension of the notion of method definition

- The signature of a chord is a collection of (traditional) method declarations joined by &
- The body of a chord is all similar to the body of a traditional method

```java
class Example {
    public int get() & public async put(int i) {
        return i;
    }
}
```

- Within a chord:
  - at most one method can be non-asynchronous
- Within a class:
  - the same method can appear in more than one chord
- (We do not discuss additional rules for inheritance and overloading)
A chord is only executed once all the methods in its signature have been called

- Calls are buffered until there is a matching chord
  - the implicit buffer supports complex synchronization patterns with little code (see Producer/Consumer later)

- If multiple matches are possible, nondeterminism applies

- Execution returns a value to the only non-asynchronous method in the chord (if any)
public class Buffer() {
    public int get() & public async put(int i) {
        return i;
    }
}

... 
Buffer b = new Buffer();
b.put("okey")
Console.WriteLine(b.get()); // prints "okey"
b.put("okey"); b.put("dokey");
    // prints "okeydokey" or "dokeyokey"
Console.WriteLine(b.get() + b.get());
b.get(); // blocks until some other thread calls put
public class Buffer {

    public void give(String s) & async available(int a) {
        if (a == 1) {
            // just one slot available and giving: become full
            full();
        } else {
            // more than one slot available and giving:
            // enable more giving
            available(a - 1);
        }
        // buffer message for takes
        inBuffer(s);
    }
}
public String take() & async inBuffer(String s) & async full() {
    // full and taking: one slot becomes available
    available(1);
    // return message in queue
    return s;
}

public String take() & async inBuffer(String s) & async available(int a) {
    // not full: one more slot becomes available
    available(a + 1);
    // return message in queue
    return s;
}
// constructor
public Buffer(int capacity) {
    available(capacity);
}

Note: unlike in the examples we developed with locks, here there is no guarantee of ordered retrieval because any message in the implicit buffer can be retrieved at any time