Multithreading Components in .Net

Composition of Separately Authored Components
in a Multi-core World

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Agenda

- Introduction
- Components and Multithreading
- Locking Rules and Best Practices
- Automatic Synchronization

Why Multithreading?

- **Improve responsiveness** → Use ”Async” programming

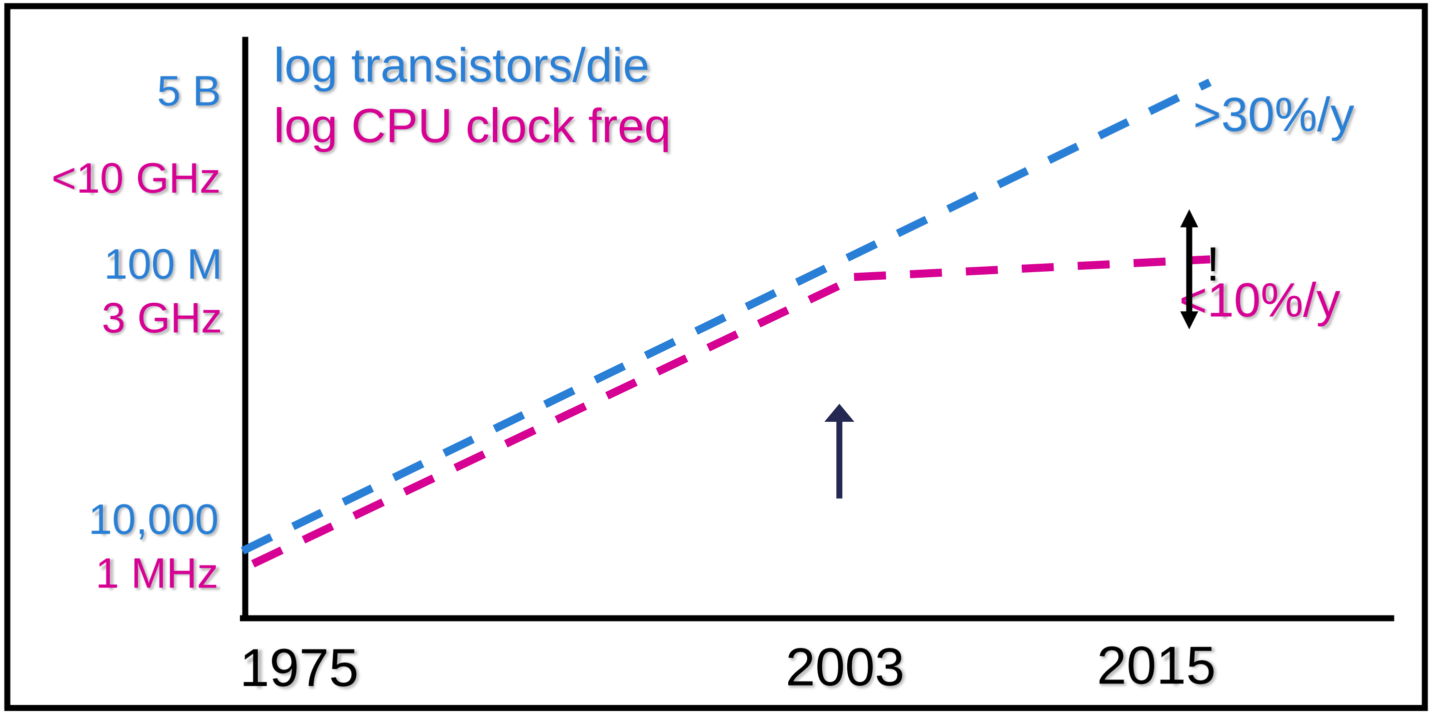
and / or

- **Improve performance** → Use ”Parrallel” programming

- Why not concurrency?
  - It brings non-determinism
  - Specific knowledge and discipline needed

Concurrency for Performance

- Welcome to the multi-core era!



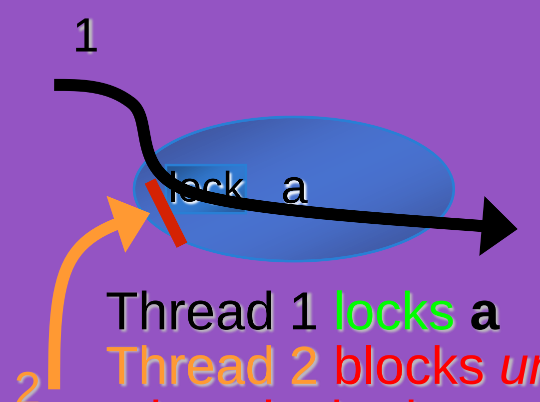
- Processors don’t get way faster
  - You just get more and more slow ones
- Multi-core hardware makes concurrency “required”

Threads, Shared Memory, and Locks: Concepts 1

- "Simultaneous" multiple threads of control
- Shared memory changes under you
  - Shared: global data, static fields, heap objects
  - Private: PC, registers, locals, stack, unshared heap objects, thread local storage
- **The three isolation techniques**
  - **Confinement: sharing less of your state**
  - **Immutability: sharing read-only state**
  - **Synchronization: using locks to serialize access to writeable shared state**

```
Thread 1:
...
lock(a) {
   ...
   ...
```

```
Thread 2:
...
...
lock(a) {
   ...
```



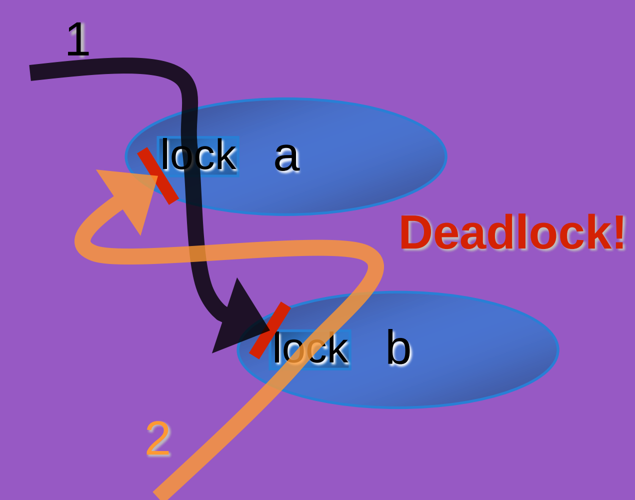
Thread 1 locks **a**
Thread 2 blocks *until*
a is unlocked

Concepts 2

- Invariants: logical correctness properties
- Safety vs. liveness vs. efficiency
  - Safety: program is “correct”: invariants hold
    - Race conditions → violated invariants → hard bugs
  - Liveness: program makes progress: no deadlocks
  - Efficiency: as parallel as possible
- Waiting (for another thread to do something)
  - Consumer awaits producer
  - Manager awaits workers

```
Thread 1:
lock(a) {
   ...
   ...
   lock(b) {
      ...
      ...
   }
   ...
   ...
}
```

```
Thread 2:
...
lock(b) {
   ...
   ...
   lock(a) {
      ...
      ...
   }
   ...
   ...
}
```



COMPONENTS AND MULTITHREADING

Components and Multithreading

- Component concurrency management is a core principle of component-oriented programming.
- A component vendor can't assume that multiple concurrent client threads will not access their components.
- As a result, unless the vendor states that its components aren't thread-safe, the vendor must provide thread-safe components!

Sequential vs. **Concurrent**

	Sequential	**Concurrent**
Behavior	Deterministic	Nondeterministic
Memory	Stable	In flux (unless private, read-only, or protected by a lock). If accessing multiple related data structures, locks for all structures must be held
Locks	Unnecessary	Essential
Invariants	Must hold only on method entry/exit or calls to external code	Anytime the protecting lock is not held
Deadlock	Impossible	Possible, but can be mitigated
Testing	Code coverage finds most bugs	Code coverage insufficient; races, timing, and environments probabilistically change
Debugging	Trace execution leading to failure; finding a fix is generally assured	Postulate a race and inspect code; root causes easily remain unindentified

Monitors

- The C# lock statement is really just a shorthand notation for working with the System.Threading.Monitor class type.
- A monitor only blocks other threads.
  It’s possible to reenter a locked block from the thread that holds the lock (e.g. recursive calls).
- Monitors can only lock reference types.
- Monitors can only be used within the same application domain.
- Because you virtually always want to release the lock on a thread at the end of a critical section, even when throwing an **Exception**, calls to **Monitor.Enter( )** and **Monitor.Exit( )** are usually wrapped in a try block:

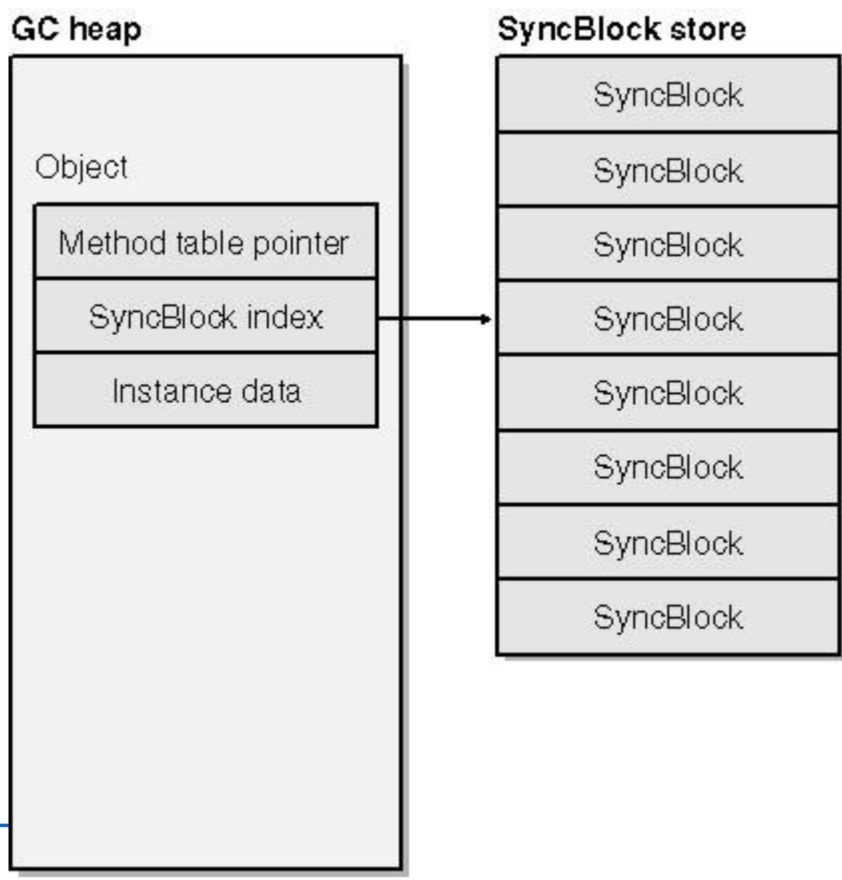
```
Monitor.Enter(this);
try{
  //critical section
  ...
}
finally{
  Monitor.Exit(this);
}
```

```
lock(this){
. . .
}
```

SCHOOL OF ENGINEERING

Monitor Internals

- Every object on the GC heap has two overhead members associated with it:
  - A method table pointer containing the address of the object's method table
  - A SyncBlock index referencing a SyncBlock created by the .NET Framework.
- The relationship between objects allocated on the GC heap and SyncBlocks:



AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Lock encapsulation

- .NET and Java automatically provide a monitor-per-object

```
class Foo {
…
  lock(this) {…}
}
```

  - But many developers prefer to encapsulate synchronization

```
class Foo {
    private object m_lock; /*…*/
    /*…*/ lock (m_lock) … /*…*/
    /*…*/ Monitor.PulseAll(m_lock); /*…*/
}
```

- If a caller wants to compose such components…
  - Break the encapsulation and expose internal locks/events?

```
public object GetLock() { return m_lock; }
```

  - Or they must devise an orthogonal locking scheme
    - this is difficult-to-impossible if coordination is needed

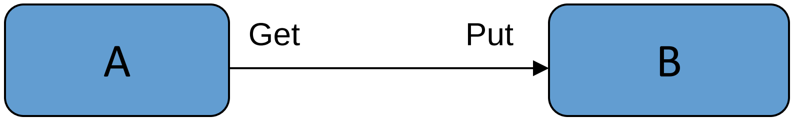
Example: bounded buffer

```
class BoundedBuffer<T> {
    private int capacity;
    private Queue<T> data = new Queue<T>();
    private object sync = new object();

    public BoundedBuffer(int c) { capacity = c; }

    public void Put(T item) {
        lock (sync) {
             while (data.Count == capacity)
                 Monitor.Wait(sync);
             data.Enqueue(item);
        }
    }

    public T Get() {
        T item;
        lock (sync) {
            if (data.Count == 0)
                throw new BufferEmptyException();
            item = data.Dequeue();
            Monitor.PulseAll(sync);
        }
        return item;
    }
}
```

Example: transfer between two buffers

- We'd like to transfer a single item between two bounded-buffers? Remember…
  - Each buffer does synchronization internally
  - Locking to ensure adds/removes are safe
  - Waits/pulses for when the buffer is full/empty

- What's the proper design/implementation?



Example: transfer between two buffers (cont.)

- Simplest possible approach would be ~

```
BoundedBuffer<T> a = …;
BoundedBuffer<T> b = …;
…
b.Put(a.Get());
```

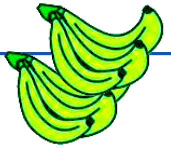
- But of course, simple won’t work here…
  - What if a failure happens between Get and Put? Missing item!
  - And even if failure doesn’t occur, there’s still a window of time in which the item exists neither in A nor B (broken serializability)

Example: transfer between two buffers (cont.)

- OK, so how ‘bout we lock on both objects for the duration of the move?

```
BoundedBuffer<T> a = …;
BoundedBuffer<T> b = …;
…
lock (a) {
    lock (b) {
        b.Put(a.Get());
    }
}
```

- What happens if someone moves from A$\rightarrow$B while another thread moves from B$\rightarrow$A? Deadlock!
- And recall that B might do a Wait in Put if the buffer is full
  - Releases the lock on B, but not A!

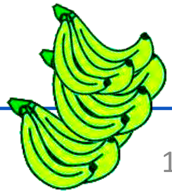


Example: transfer between two buffers (cont.)

- Maybe we should just use a single lock

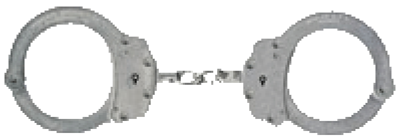
```
BoundedBuffer<T> a = …;
BoundedBuffer<T> b = …;
object transferLock = new object();
…
lock (transferLock) {
    b.Put(a.Get());
}
```

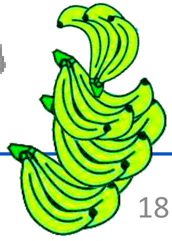
- But wait! This won't scale very well at all…
  - What if there are 50 buffers?
  - And if B blocks (because it's full), we still retain the global lock… possibly deadlocking the entire program!



No single approach will always work

- Unfortunately, there is no *easy* solution
  - What's wrong here? Isn't programming all about building larger abstractions out of smaller ones? This seems *busted*
  - Messy dependencies and broken layering can be used internal to a library/app… nasty… but it can work
- In a composable, reusable library it's harder
  - .NET Framework tried in 1.0…
    - SyncRoot object on collections
    - "Exposed" implementation details
  - Ran far, far away in 2.0
- Pluggable synchronizable objects?
  - Give us your monitor and we'll use it for locking and condition variables?
  - Breaks abstractions, has public contract implications; we dislike this

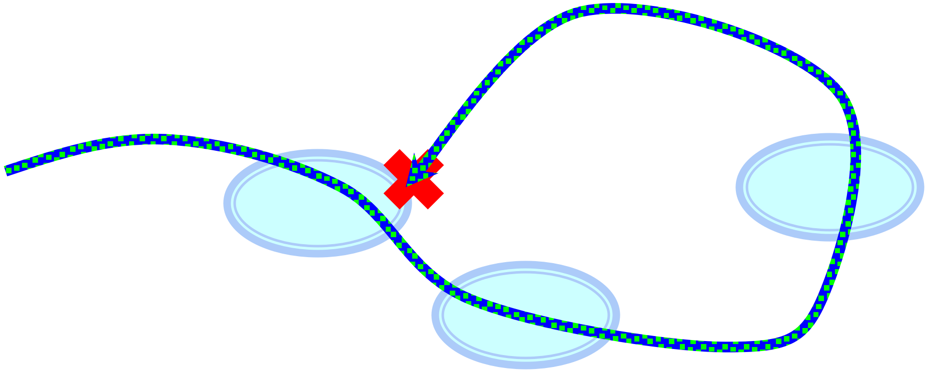




AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Recursion: deadlocks or corruption

- Designs with recursion are brittle
  - Recursion is to *reacquire* a lock when it's already held
  - Typically "lock acquire" means "invariant safe point"
  - Recursion invalidates this assumption…
- Pick your poison
  - Recursive locks == broken invariants
  - Non-recursive locks == deadlocks



Recursion abstinence

- To avoid recursion poisoning…
  - Don't rely on it in your design
  - Never make a virtual call with a lock held
  - Similarly, only call your own (well known and tested) code
  - Don't block with a lock held
- In other words: you can't compose *anything* together!
- Unfortunately, hard to follow
  - Recursive algorithms, internal helper methods
  - Split into "no lock held yet" entry points and "assert that a lock is held" (internal) entry points
- When push comes to shove, deadlock is better than broken invariants (and reliability holes)

Deadlocks

- Automated deadlock detection and resolution is not a reality today

```
lock  (a) {
    // s0
    lock (b) { // Deadlock! Undo s0? Automatically?
        // …
    }
}
```

- Deadlock avoidance is do-able
  - Lock leveling (aka ordered locks)
  - Timeouts and "manual" back-off
  - Still can't undo *s0* automatically—but you can (at least) deterministically test for this and rid the system of bugs
- Remember: Locks don't compose
  - So by inference, deadlock avoidance also typically doesn't compose
  - Both work for closed systems, but separately authored components still can't be stitched together reliably with either approach

Deadlock avoidance example: Leveling

- All locks assigned a “level”
  - Often numeric, but sometimes relative
  - And then only locks at monotonically decreasing levels can be taken
- For example, given:

```
internal static LeveledLock s_lock0 = new LeveledLock(0);
internal static LeveledLock s_lock1 = new LeveledLock(1);
```

  - This code succeeds:

```
lock (s_lock1) { lock (s_lock0) { … } }
```

  - But this code fails:

```
lock (s_lock0) { lock (s_lock1) { … } }
```

  - Thus, ensuring deadlock freedom

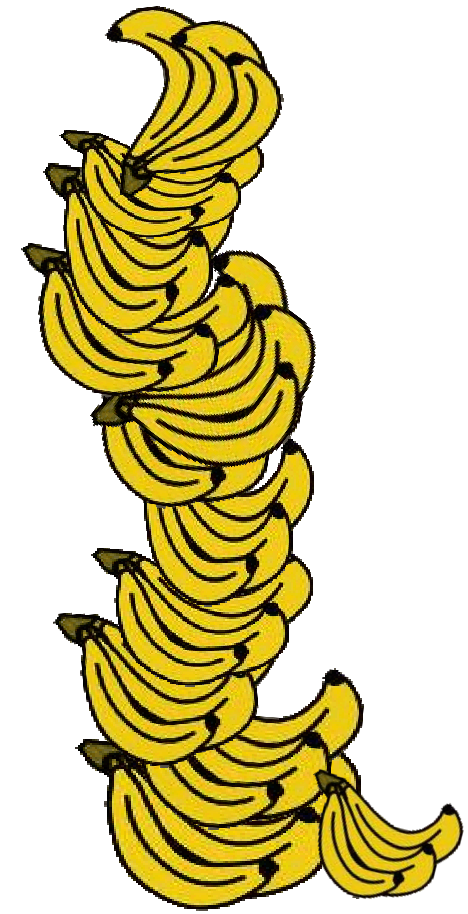
Locking model

- Keep it simple
- But let your users know what to expect
  e.g.:
  - State shared within the library is always accessed thread-safely
  - Anything else is not (you share it, you lock it)
  - Simple, predictable, understandable:
    - for library builders AND library users

In the words of Simon Peyton Jones…

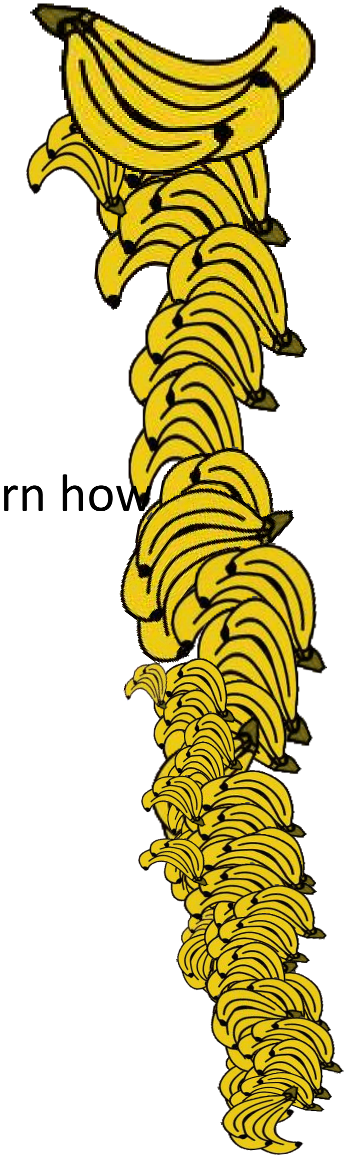
- Building programs with threads and locks is like trying to build a skyscraper out of bananas

```
lock (this) {
    while (pa && !pb) {
        Monitor.Wait(this);
    }
    foo();

    lock (that) {
        bar(); baz();
    }
}
```

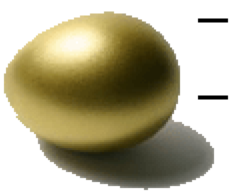


Those skilled in the art can get by with it
But it's not a very solid foundation on top of which to write robust software

☹ What to do?



- Sadly…
  - Yesterday we had bananas (30+ years ago);
  - Today we have bananas;
  - And tomorrow we will still have bananas
- Applications need concurrency (today!)
  - Therefore, (some) library builders probably need to learn how to build libraries out of bananas…
  - Microsoft is trying to learn this—others are too
- Some day we'll have STM – perhaps?
  - But until then, it's bananas all the way down!
  - STM = State Transactional Memory



Not likely

LOCKING RULES AND BEST PRACTICES

Locking Rules and Best Practices

- Encapsulate lock
  - To guard against undisciplined access, encapsulate the lock inside every public method and property

```
class MyList<T> {
  T[] items;
  int n;
  void Add(T item) {
    items[n] = item;
    n++;
  }
}
class client {
  MyList<int> list = new MyLi…
  void Foo () {
    Monitor.Enter(list);
    list.Add(27);
    Monitor.Exit(list);
    …
```

```
class MyList<T> {
  T[] items;
  int n;
  void Add(T item) {
    lock (this) {
      items[n] = item;
      n++;
    }
  }
  …
}

class client {
  MyList<int> list = new MyLi…
  void Foo () {
    list.Add(27);
    …
```

Locking Rules and Best Practices

- Encapsulate lock
- Lock over all writeable shared state
  - And always use the same lock for the given state

```
class MyList<T> {
  T[] items;
  int n;

  void Add(T item) {
    items[n] = item;
    n++;
  }
  …
}
```

```
class MyList<T> {
  T[] items;
  int n;

  void Add(T item) {
    lock (this) {
      items[n] = item;
      n++;
    }
  }
  …
}
```

Locking Rules and Best Practices

- Encapsulate lock
- Lock over all writeable shared state
  - And always use the same lock for the given state
- Lock over entire invariant

```
class MyList<T> {
  T[] items;
  int n;

  void Add(T t) {
    lock(this) items[n] = t;
    lock(this) n++;
  }
  …
}
```

```
class MyList<T> {
  T[] items;
  int n;
  // invariant: n is count
  // of valid items in list
  // and items[n] == null

  void Add(T t) {
    lock(this) {
      items[n] = t;
      n++;
    }
  }
  …
}
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Locking Rules and Best Practices

- Encapsulate lock
- Lock over all writeable shared state
  - And always use the same lock for the given state
- Lock over entire invariant
- Don’t use private lock objects
  - And don’t lock on strings
  - And only lock on Types for static members

```
class MyList<T> {
  T[] items;
  int n;
  // invariant: n is count
  …
  object lk = new object();

  void Add(T t) {
    lock(lk) {
      items[n] = t;
      n++;
    }
  …
```

```
class MyList<T> {
  T[] items;
  int n;
  // invariant: n is count
  …

  void Add(T t) {
    lock(this) {
      items[n] = t;
      n++;
    }
  …
```

Locking Rules and Best Practices

- Encapsulate lock
- Lock over all writeable shared state
  - And always use the same lock for the given state
- Lock over entire invariant
- Use private lock objects
  - And don't lock on Types or strin
- Don't call others' code while you hold locks

```
class MyList<T> {
  T[] items;
  int n;
  object lk = new object();

  void Add(T t) {
    lock(lk) {
      items[n] = t;
      n++;
      Listener.Notify(this);
    }
  …
```

```
class MyList<T> {
  T[] items;
  int n;
object lk = new object();

  void Add(T t) {
    lock(lk) {
      items[n] = t;
      n++;
    }
    Listener.Notify(this);
  }
  …
```

Locking Rules and Best Practices

- Encapsulate lock
- Lock over all writeable shared state
  - And always use the same lock for the given state
- Lock over entire invariant
- Use private lock objects
  - And don't lock on Types or strin
- Don't call others' code while you hold locks
- Use appropriate lock granularitie

```
class MyService {
  static object lk_all = …;

  static void StaticDo() {
    lock(lk_all) { … }
  }
  void Do1() {
    lock(lk_all) { … }
  }
  void Do2() {
    lock(lk_all) { … }
  }
}
```

```
class MyService {
  static object lk_all = …;
  object lk_inst = …;


  static void StaticDo() {
    lock(lk_all) { … }
  }
  void Do1() {
    lock(lk_inst) { … }
  }
  void Do2() {
    lock(lk_inst) { … }
  }
}
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Locking Rules and Best Practices

- Encapsulate lock
- Lock over all writeable shared state
  - And always use the same lock for the given state
- Lock over entire invariant
- Use private lock objects
  - And don't lock on Types or strings
- Don't call others' code while you hold locks
- Use appropriate lock granularities
- Order locks to avoid deadlock

```
class MyService {
  A a;
  B b;
  void DoAB() {
    lock(a) lock(b) {
      a.Do(); b.Do();
    } }
  void DoBA() {
    lock(b) lock(a) {
      b.Do(); a.Do();
    } }
}
```

```
class MyService {
  A a;    // lock: lkA
  B b;    // lock: lkB
  // order! lock(a) < lock(b)
  …
  void DoAB() {
    lock(a) lock(b) {
      a.Do(); b.Do();
    } }
  void DoBA() {
    lock(a) lock(b) {
      b.Do(); a.Do();
    } }
}
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Waiting Rules and Best Practices

- Wait example
- Use only the one true wait pattern
  - Test and retest condition while holding lock

```
while (!full)
  lock (mon) {
    Monitor.Wait(mon);
  …

lock (mon) {
  if (!full)
    Monitor.Wait(mon);
  …

lock (mon) {
  while (!full)
    Monitor.Wait(mon);
```

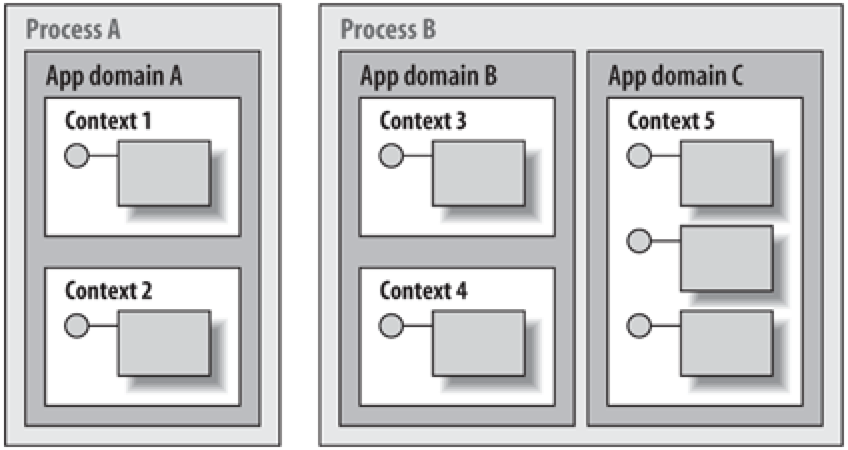
```
public class Channel<T> {
  T t;
  bool full;
  object mon = new object();

  public T Get() {
    T t = default(T);
    lock (mon) {
      while (!full)
        Monitor.Wait(mon);
      t = this.t; full = false;
      Monitor.PulseAll(mon);
    }
    return t;
  }

  public void Put(T t) {
    lock (mon) {
      while (full)
        Monitor.Wait(mon);
      this.t = t; full = true;
      Monitor.PulseAll(mon);
    }
  }
}
```

AUTOMATIC SYNCHRONIZITION

Don't
use!

NET app domains and contexts

- The app domain isn't the innermost execution scope of a .NET component.
- .NET provides a level of indirection between components and app domains, in the form of *contexts*.
- Contexts enable .NET to provide *component services* such as thread synchronization to a component.



ContextBoundObject

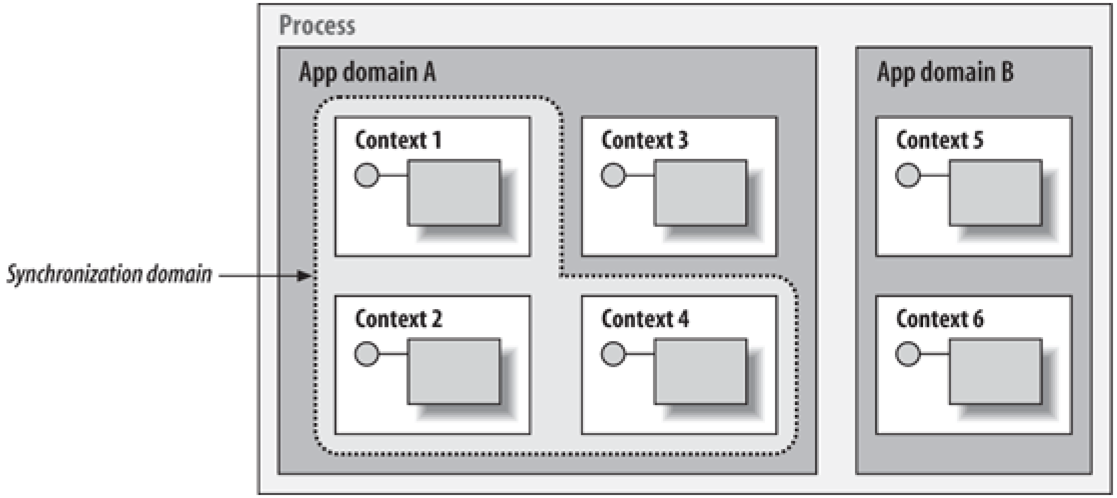
- By default, .NET components aren't aware that contexts exist.
- When a client in the app domain creates an object, .NET gives back to the client a direct reference to the new object.
  - Such objects always execute in the context of the calling client.
- In order to take advantage of .NET component services, components must be context-bound,
  - meaning they must always execute in the same context.
  - Such components must derive directly or indirectly from the class ContextBoundObject:

```
public class MyClass : ContextBoundObject {...}
```

ContextBoundObject

- Clients never have a direct reference to a context-bound object.
- Instead, they have a reference to a proxy.
- .NET provides its component services by intercepting the calls clients make into the context via the proxy and performing some pre- and post-call processing

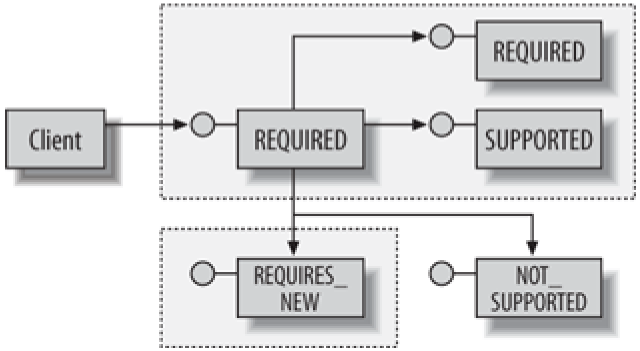
Synchronization Domains and Contexts



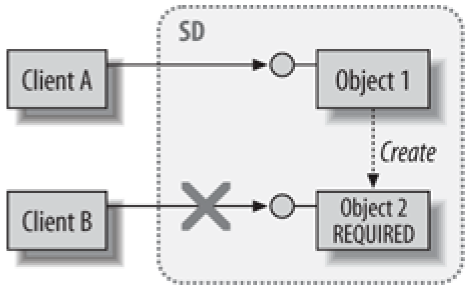

```
using System.Runtime.Remoting.Contexts;

[Synchronization]
public class MyClass : ContextBoundObject
{
   public MyClass( ) {}
   public void DoSomething( ) {}
   //Other methods and data members
}
```

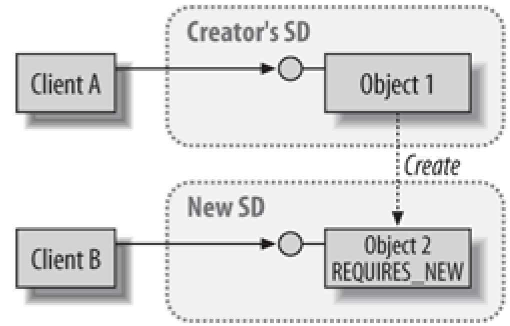
Synchronization domain flow

- An object's synchronization domain is determined at the time of its creation.



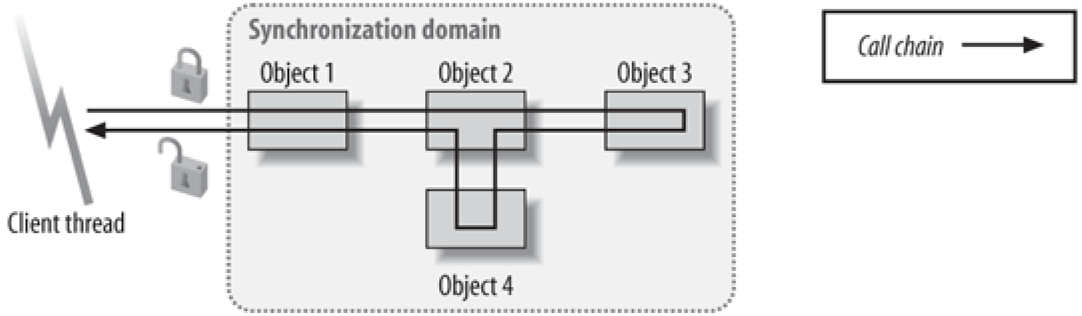
Having separate synchronization domains enables clients to be served more efficiently



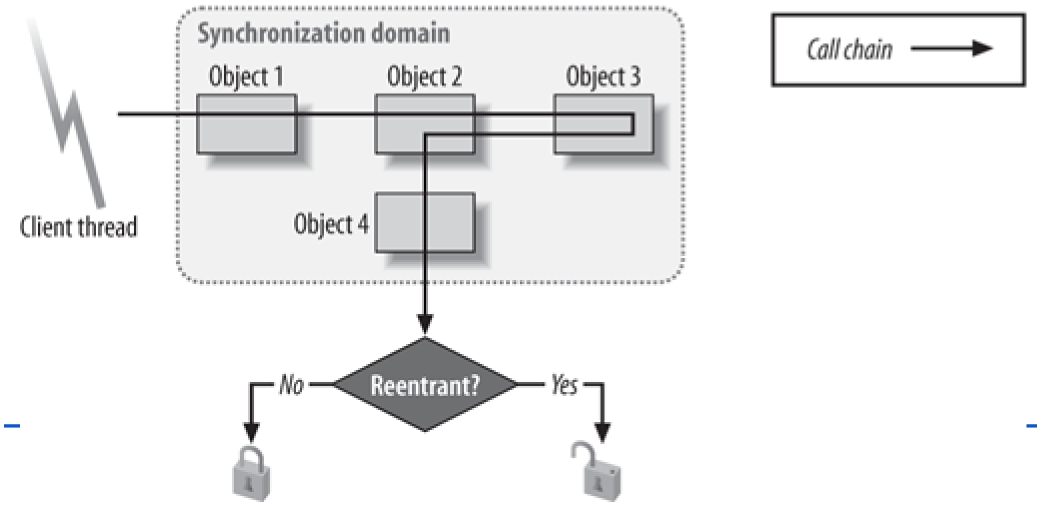


Synchronization-Domain Reentrancy

- Releasing a lock when the call chain exits on the original entry object



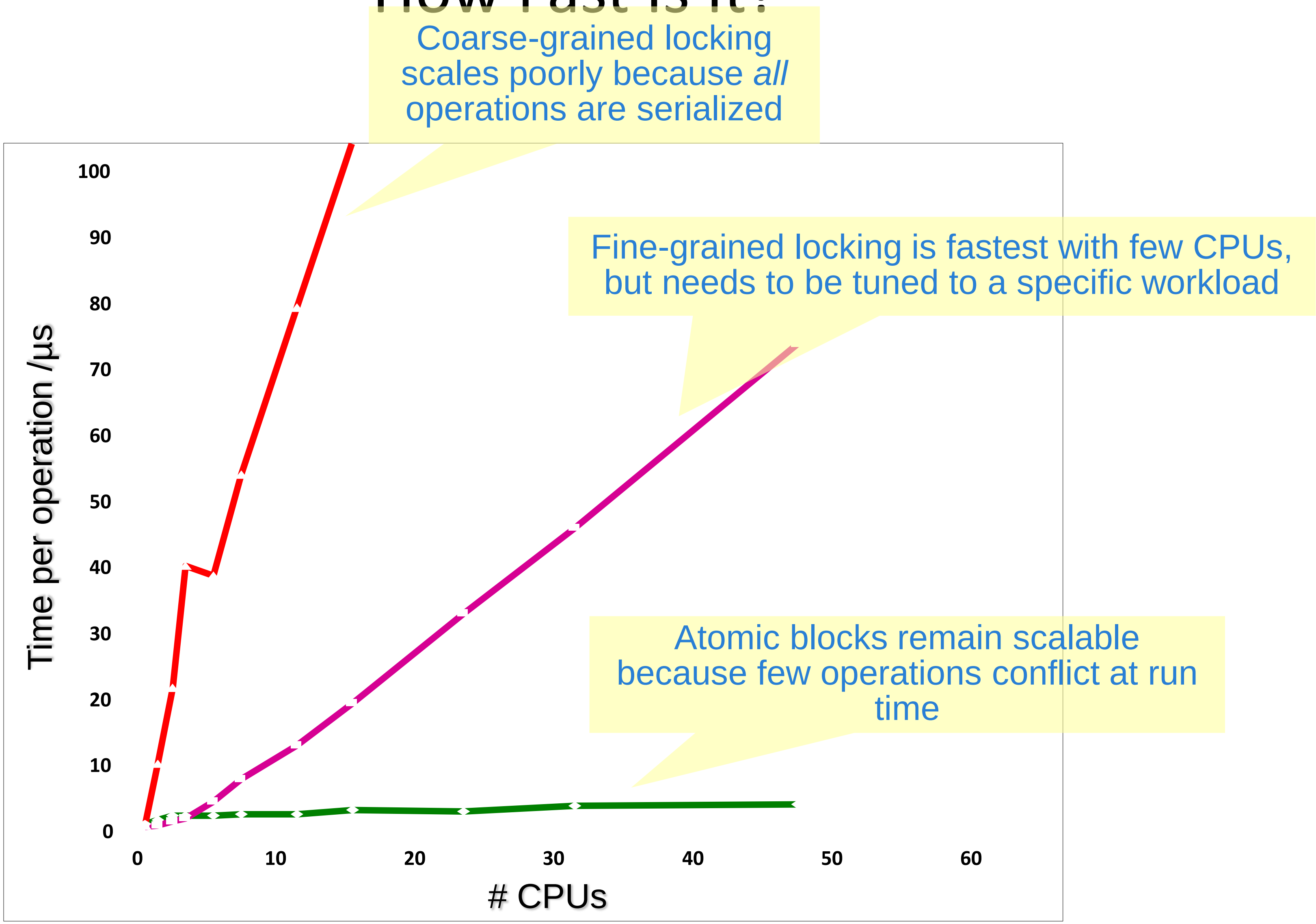
- If reentrancy is set to true, NET releases the synchronization domain lock while the outgoing call is in progress



Synchronization Domain Pros and Cons

- Automatic synchronization for context-bound objects via synchronization domain is by far the easiest synchronization mechanism available to .NET developers
- BUT:
  - You can use them only with context-bound objects. For all other .NET types, you must still use manual synchronization objects.
  - There is a penalty for accessing context-bound objects via proxies and interceptors. In some intense calling patterns, this can pose a problem.
  - A synchronization domain doesn't protect static class members and static methods. For those, you must use manual synchronization objects.
  - A synchronization domain isn't a throughput-oriented mechanism. The incoming thread locks a whole set of objects, even if it interacts with only one of them. That lock precludes other threads from accessing these objects, which can degrade your application throughput

How Fast Is It?



- Threads using a shared hashtable
- 16% updates, keys 1..4096

Resources

- Joe Duffy, ***Concurrency and the impact on reusable libraries***
  http://www.bluebytesoftware.com/blog/PermaLink,guid,f8404ab3-e3e6-4933-a5bc-b69348deedba.aspx
- Joseph Albahari, **Threading in C#** ☆ ☆ ☆
  http://www.albahari.com/threading/
  (Click Download pdf)
- **Parallel Programming with .NET**
  http://blogs.msdn.com/b/pfxteam/
- **PATTERNS OF PARALLEL PROGRAMMING**
  UNDERSTANDING AND APPLYING PARALLEL PATTERNS
  WITH THE .NET FRAMEWORK 4 AND VISUAL C#
  Stephen Toub
  http://www.microsoft.com/en-us/download/details.aspx?id=19222