CSC352

Introduction to Thread safety and Synchronization

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Thinking about Thread Safety
2. Protecting multi-step operations

Thinking about Thread Safety

Tasks

1. What is Thread Safety?
2. Using thread confinement
3. Sharing immutable data
4. Working with atomic operations
5. Dealing with cache issues

What is thread safety?

- ✓ **Thread-safe code** is code which is *completely deterministic* regardless of how many threads are accessing the code simultaneously

- ✓ This means that no matter how many threads are run, and no matter what order they are run in, the behavior is always well-defined and always produces the same results

What makes code unsafe?

- Sharing <u>mutable</u> data across threads is dangerous – can lead to inconsistent data, or even corruption

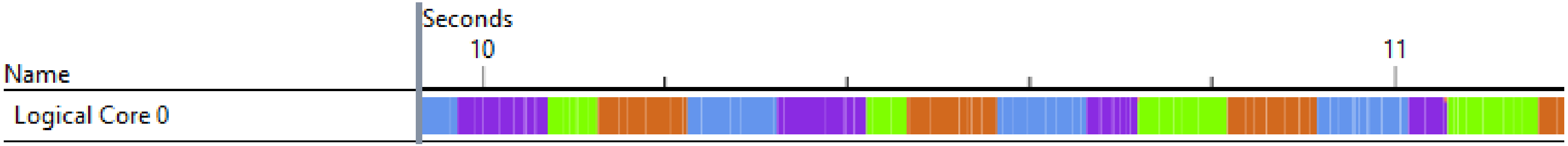
- **Must synchronize** our access to shared state, but not *over synchronize* because that reduces scalability

- Developers job is to select the proper synchronization style to balance this properly

Xamarin University

Recall: Threads vs. CPUs

- The CPU **switches** between threads on given intervals; this switch can occur **at any time** and **at any place** in your code



Thread ID	Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
1904	CLR Worker Thread	0	144	0.00 %
1204	CLR Worker Thread	0	131	0.00 %
4676	CLR Worker Thread	0	125	0.00 %
2908	CLR Worker Thread	0	113	0.00 %
5100	CLR Worker Thread	0	14	0.00 %

What makes code unsafe

- When a thread switch occurs, another thread could execute code which modifies shared data; this can lead to corruption and unexpected issues

```
event PropertyChangedEventHandler PropertyChanged;

void RaisePropertyChanged(string name)
{
    if (PropertyChanged != null)
    {
        PropertyChanged.Invoke(this,
            new PropertyChangedEventArgs(name));
    }
}
```

If multiple threads were interacting with this event, is there any chance for a null exception?

.NET thread safety

- Most .NET objects are <u>safe to access from multiple threads simultaneously;</u> it's only when those objects change at runtime and are used by multiple threads that we have to provide synchronization

MSDN documentation has **Thread Safety** section for every documented class – can refer to this documentation to see what guarantee the class itself makes

The System.Random class and thread safety

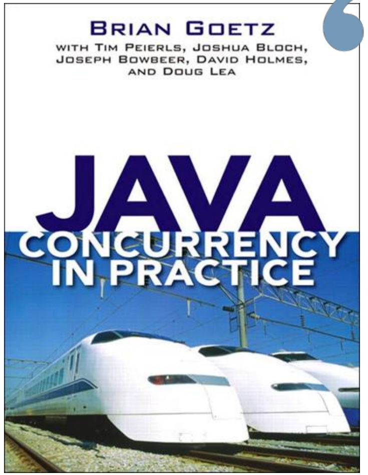
Instead of instantiating individual Random objects, we recommend that you create a single Random instance to generate all the random numbers needed by your app. However, Random objects are not thread safe. If your app calls Random methods from multiple threads, you must use a synchronization object to ensure that only one thread can access the random number generator at a time. If you don't ensure that the Random object is accessed in a thread-safe way, calls to methods that return random numbers return 0.

Thread safety techniques

- There are several ways to ensure our application code runs properly

There is no "one-size-fits-all" solution to thread safety; it's a hard problem to solve, and most applications will apply several, if not all, of these techniques

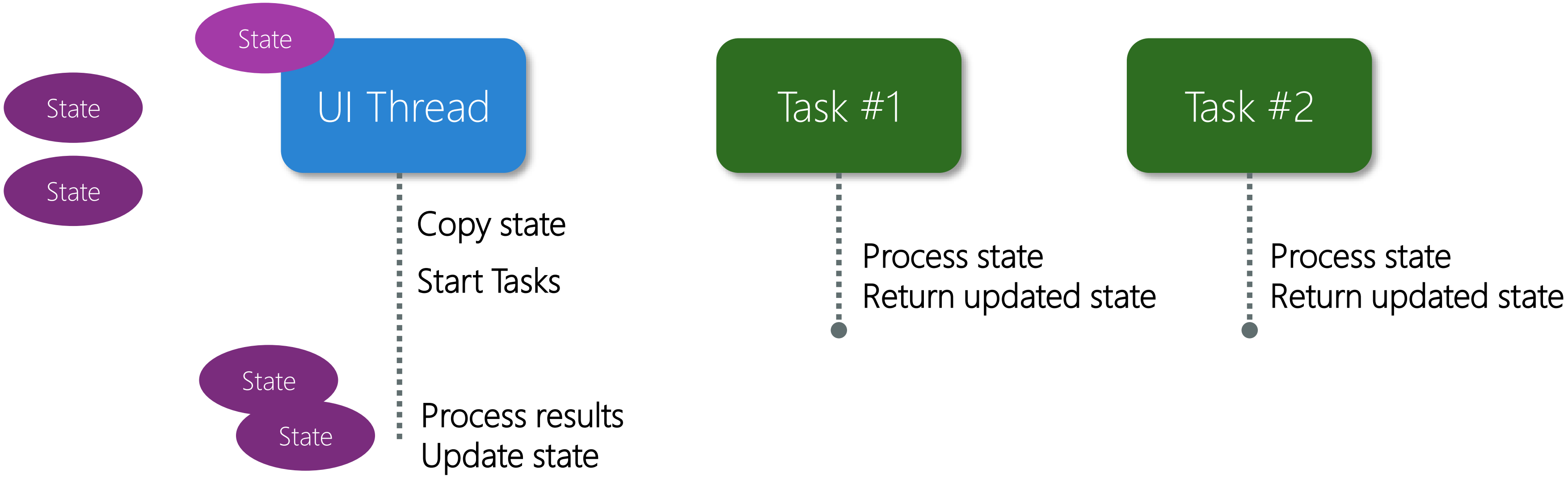
Don't share data across threads



“Accessing shared, mutable data requires using **synchronization**; one way to avoid this requirement is to **not share**. If data is only accessed from a single thread, no synchronization is needed. This technique, **thread confinement**, is one of the simplest ways to achieve thread safety. When an object is confined to a thread, such usage is automatically thread-safe even if the confined object itself is not.”

Using thread confinement

- Can make a copy of any required data and pass into the task to work with – the task then makes changes and returns the modified copy



Only share immutable objects

- **Immutable objects** are objects which, once created, **never change state**; this means different threads can access the same object simultaneously without side effects

Methods intended to *modify* the object always return a *new* copy of the object

```
class ImmutableType
{
    public double Value { get; }
    public ImmutableType(double x) {
        Value = x;
    }

    public ImmutableType Square() {
        return new ImmutableType(
              Value*Value);
    }
}
```


Esperi el seu torn
Wait your turn
Espere su turno

Xamarin University

What is an atomic operation?

- An operation that is guaranteed to complete without interruption once started is referred to as *atomic*; these sorts of operations are always thread safe

```
class Program
{
    static Int64 value;
    static void SetValue()
    {
        value = 0x200000001;
    }
}
```

If another thread *reads* **value**, will it always get a valid value – even if **SetValue** is executing?

What is an atomic operation?

- To be atomic, the operation cannot be interrupted and must appear *instantaneous* with regards to the rest of the application

```
class Program
{
    static Int64 value;
    static void SetValue()
    {
        value = 0x200000001;
    }
}
```

X64 can do this in one machine instruction – which makes it atomic since we *cannot be interrupted in the middle of an instruction*

```
mov qword ptr [value], 200000001h
```

What is an atomic operation?

❖ To be atomic, the operation cannot be interrupted and must appear *instantaneous* with regards to the rest of the application

```
class Program
{
    static Int64 value;
    static void SetValue()
    {
        value = 0x200000001;
    }
}
```

But x86 cannot deal with values this large in a single register and therefore this requires *multiple instructions* and is ***not*** atomic!

```
mov dword ptr ds:[value], 1
mov dword ptr ds:[value+4], 2
```

What about simple increments?

? If two threads call `Increment` 500 times, what will the value of counter be?

```
public class Counter
{
    int counter;
    public int Value => counter;

    public virtual void Increment() {
        counter++;
    }
}
```

What is an increment?

❖ Incrementing values is **not** an atomic operation because it involves **multiple machine instructions**

```
counter++;
```

```
int temp = counter;
temp = temp + 1;
counter = temp;
```

```
MOV R0,[counter]
ADD R0,1
MOV [counter],R0
```

Multi-threaded increments

- When two threads are involved, there is always the possibility of getting stale values with non-atomic operations

Counter: 6

We've lost one increment because the two threads retrieved the same non-atomic value

Thread #1	Thread #2
MOV R0,[counter]	MOV R0,[counter]
ADD R0,1	ADD R0,1
MOV [counter],R0	MOV [counter],R0
R0: 6	R0: 6

Protecting simple arithmetic values

- We can fix the problem by using the `Interlocked` class which makes the increment atomic

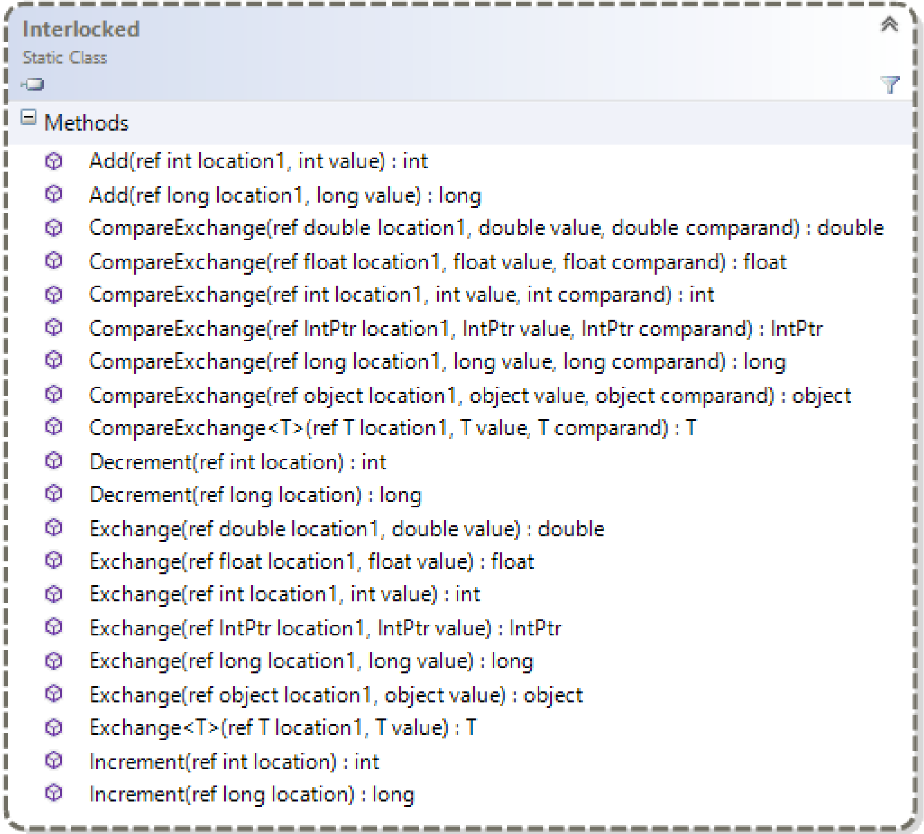
```
public class SafeCounter
{
    int counter;
    public int Value { get { return counter; } }

    public virtual void Increment() {
        Interlocked.Increment(ref counter);
    }
}
```

This increments the counter and returns the updated value

Interlocked class

- `Interlocked` is a static class with methods to perform basic arithmetic on integer (and pointer-sized) values atomically

- Guarantees that no other thread will access the value while it is being modified and that the CPU will not reorder it with other operations



What does Interlocked do?

- `Interlocked` forces the CPU to **lock the cache line** associated with our value which prevents other CPUs from accessing the memory *and* ensures no context switch occurs until the increment is complete

```
Interlocked.Increment(
      ref counter)
```

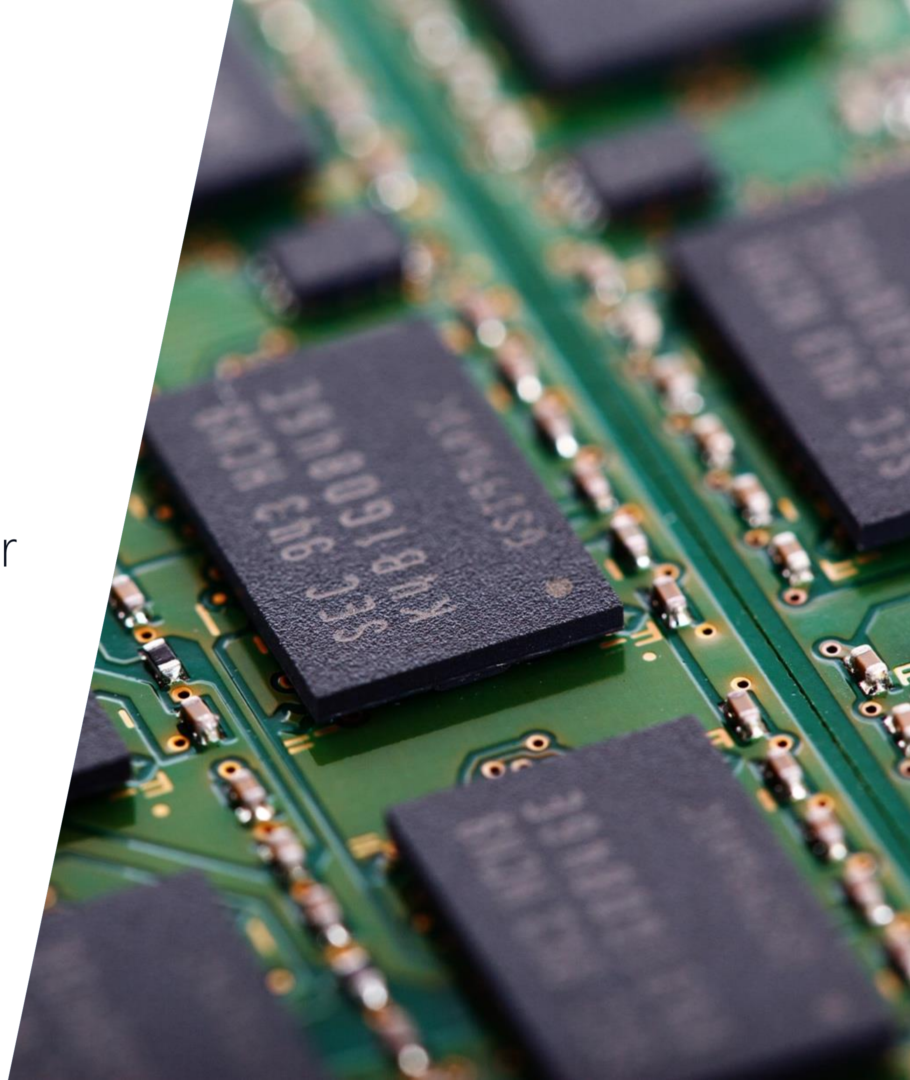
```
LEA R0,counter
LOCK INC [R0]
```

What is a cache line?

- CPUs transfer cache (L1/L2) to memory in ranges (typically 64-128 bytes) called *cache lines*

- Locking a cache line ensures no other core can access that block; instead, they *stall* until the cache line is unlocked

- This produces unnecessary overhead if the value is not used by multiple threads



Compiler optimizations

- Compilers and CPUs aggressively cache and re-order your code

Assume one thread is executing this code

.. and another thread does this

```
static bool terminate = false;

static void BadLoop() {
   int x = 0;
   while (!terminate)
      x++;
}

...
terminate = true;
```

What if the compiler "optimizes" access to the **terminated** value in **BadLoop**?

These types of synchronization issues are very difficult to spot in your code because they don't always happen on every platform or build type

Indicating volatility

- We can avoid these problems by telling the compiler and runtime that the shared flag can be altered outside the visible execution path and that it should get the current value from memory

```
static bool terminate = false;
static void BadLoop() {
   int x;
   while (!Thread.VolatileRead(ref terminate))
      x++;
}
...
Thread.VolatileWrite(ref terminate, true);
```

Indicating volatility

- A simpler, but less efficient approach is to use the C# **volatile** keyword – this ensures that *every* access to the field uses the most current value

```
static volatile bool terminate = false;
static void BadLoop() {
   int x;
   while (!terminate)
      x++;
}
...
terminate = true;
```

Flash Quiz

Xamarin University

Flash Quiz

① What are some features of thread-safe code?

a) It is deterministic – even when executed by multiple threads
b) It only allows one thread at a time to execute the code
c) It ensures that threads run in a specific order
d) It uses the `volatile` keyword

Flash Quiz

① What are some features of thread-safe code?

a) It is deterministic – even when executed by multiple threads
b) It only allows one thread at a time to execute the code
c) It ensures that threads run in a specific order
d) It uses the `volatile` keyword

Flash Quiz

② Which line of code will properly *subtract 5* from an integer "x" in a thread-safe fashion?

a) `Interlocked.Decrement(ref x)` [execute 5 times]

b) `Interlocked.Add(ref x, -5)`

c) `Interlocked.Subtract(ref x, 5)`

Flash Quiz

② Which line of code will properly *subtract 5* from an integer "x" in a thread-safe fashion?

a) `Interlocked.Decrement(ref x)` [execute 5 times]

b) **`Interlocked.Add(ref x, -5)`**

c) `Interlocked.Subtract(ref x, 5)`

Summary

1. What is Thread Safety?
2. Using thread confinement
3. Sharing immutable data
4. Working with atomic operations
5. Dealing with cache issues

Protecting multi-step operations

Tasks

1. Making multiple statements Atomic
2. What is a Monitor?
3. Choosing your Monitor
4. Releasing a Monitor
5. Dealing with Exceptions
6. Coordinating Locks

A more complex example

- What problems might we encounter if multiple threads use this code?

```
decimal SavingsBalance, CheckingBalance;

public void Transfer(decimal amount) {
    SavingsBalance  += amount;
    CheckingBalance -= amount;
}

public decimal TotalAmount {
    get { return SavingsBalance + CheckingBalance; }
}
```

What if one thread were here

... and another one here

A more complex example

- What problems might we encounter if multiple threads use this code?

```
decimal SavingsBalance, CheckingBalance;

public void Transfer(decimal amount) {
   SavingsBalance  += amount;
   CheckingBalance -= amount;
}

public decimal TotalAmount {
   get { return SavingsBalance + CheckingBalance; }
}
```

Could we solve this problem using `Interlocked`?

Making multiple statements atomic

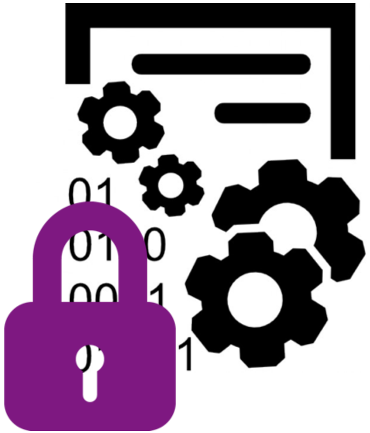
- The problem with this code is that we need *multiple statements* to execute atomically, **`Interlocked`** can only protect one statement

```
public void Transfer(decimal amount) {
    SavingsBalance  += amount;
    CheckingBalance -= amount;
}
```

While inside this method, the object is in an *invalid* state – need a way to stop other threads from accessing these two values until the method is complete

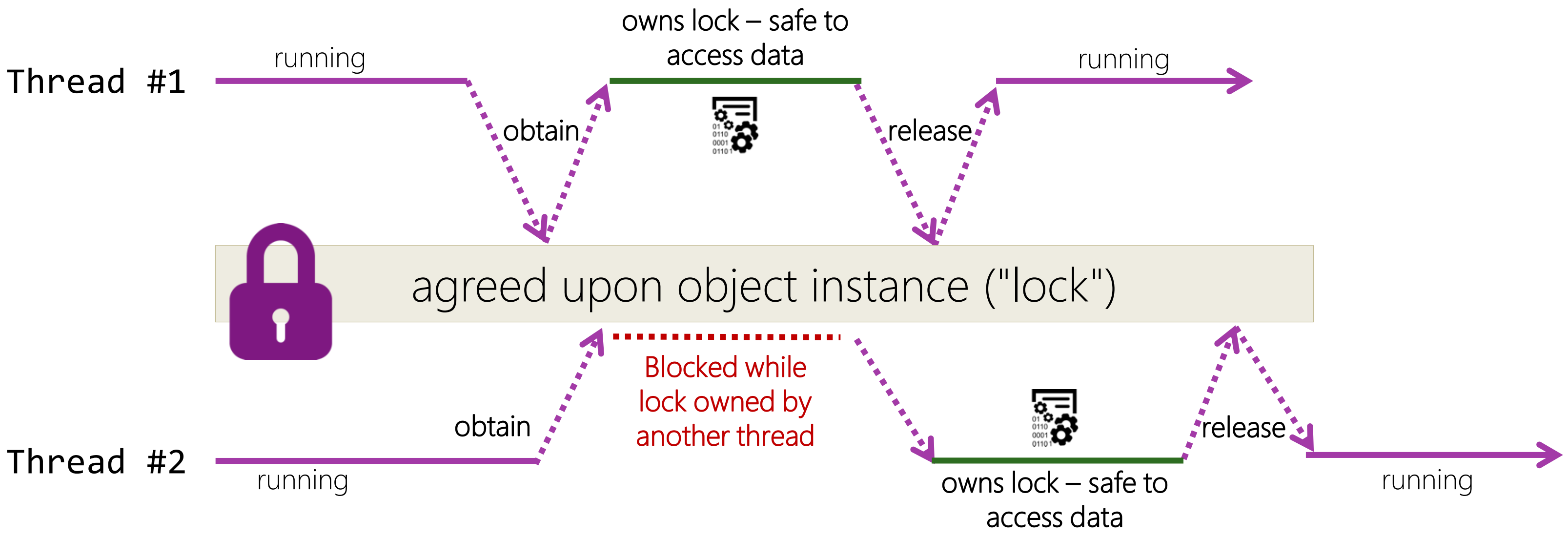
Monitor-based synchronization

- .NET provides a synchronization mechanism built around reference types called a *monitor* which allows you to ensure two (or more) blocks of code are synchronized by using an agreed upon object as a "lock"



Monitor-based synchronization

- Threads use agreed upon object to coordinate access to some piece of shared data, should only access the data when the lock is owned



Creating a Monitor lock object

- Any reference type can be used as a lock object; by convention, we typically create a unique `object` instance to be used specifically as a lock to protect a specific resource

```
private object guard = new object();
```

Using a Monitor to protect data

- A static **`Monitor`** class is used to acquire and release the lock object

```
private object guard = new object();

public void Transfer(decimal amount) {
   Monitor.Enter(guard);
   SavingsBalance += amount;
   CheckingBalance -= amount;
   Monitor.Exit(guard);
}
```

Using a Monitor to protect data

- A static **`Monitor`** class is used to acquire and release the lock object

```
private object guard = new object();

public void Transfer(decimal
   Monitor.Enter(guard);
   SavingsBalance += amount;
   CheckingBalance -= amount
   Monitor.Exit(guard);
}
```

Call Monitor.Enter to acquire the lock – this blocks until the current thread can *own* the specific lock passed

Using a Monitor to protect data

- A static **`Monitor`** class is used to acquire and release the lock object

```
private object guard = new object();

public void Transfer(decimal amount) {
    Monitor.Enter(guard);
    SavingsBalance += amount;
    CheckingBalance -= amount;
    Monitor.Exit(guard);
}
```

Can then read or write the data protected by the lock

Using a Monitor to protect data

- A static **`Monitor`** class is used to acquire and release the lock object

```
private object guard = new object();

public void Transfer(decimal amount) {
    Monitor.Enter(guard);
    SavingsBalance += amount;
    CheckingBalance -= amount;
    Monitor.Exit(guard);
}
```

Must call Monitor.Exit to release the lock – if you fail to do this, other threads will remain blocked forever!

Where should the lock be acquired

- Must make sure to acquire the **same** monitor object *every time* you want to access the protected data
- Never touch the protected data unless you own the lock, even to *read* it since it could be altered at any point

```
readonly object guard = new object();
decimal SavingsBalance, CheckingBalance;

public void Transfer(decimal amount) {
   Monitor.Enter(guard);
   SavingsBalance += amount;
   CheckingBalance -= amount;
   Monitor.Exit(guard);
}

public decimal TotalAmount {
   get {
      Monitor.Enter(guard);
      try { return SavingsBalance
                 + CheckingBalance; }
      finally {Monitor.Exit(guard); }
   }
}
```

Choosing your Monitor

1. Any reference type can be used as a monitor; `private` objects are best

2. Do not use `this` as a monitor object

3. The object used as a lock has no explicit relationship to the data it is used to protect except what the developer provides

Flash Quiz

Flash Quiz

① Which statement is a **good** example of a monitor?

a) `Monitor.Enter(this);`

b) `Monitor.Enter(typeof(object));`

c) `Monitor.Enter("Hello");`

d) `Monitor.Enter(10);`

e) None of the above

Flash Quiz

① Which statement is a **good** example of a monitor?

a) Monitor.Enter(this);

b) Monitor.Enter(typeof(object));

c) Monitor.Enter("Hello");

d) Monitor.Enter(10);

e) <u>None of the above</u>

Individual Exercise

Use a monitor lock to control access to a resource

Releasing Monitors

- Monitors are *reentrant* – a single thread can obtain a monitor lock multiple times; each call to **`Enter`** must be matched by an **`Exit`** call

```
private object guard = new object();

public void Transfer(decimal amount) {
   Monitor.Enter(guard);
   if (amount < 0) return;
   SavingsBalance += amount;
   CheckingBalance -= amount;
   Monitor.Exit(guard);
}
```

Releasing Monitors

- Monitors are *reentrant* – a single thread can obtain a monitor lock multiple times; each call to **`Enter`** must be matched by an **`Exit`** call

```
private object guard = new object();

public void Transfer(decimal amount) {
    if (amount < 0) return;
    Monitor.Enter(guard);
    SavingsBalance += amount;
    CheckingBalance -= amount;
    Monitor.Exit(guard);
}
```

What if we have too much money and overflow one of our balances?

Dealing with exceptions

- Should always catch exceptions and release locks

```
private object guard = new object();

public void Transfer(decimal amount) {
    Monitor.Enter(guard);
    try {
        SavingsBalance += amount;
        CheckingBalance -= amount;
    }
    finally {
        Monitor.Exit(guard);
    }
}
```

Shortcut: lock statement

- C# includes the **lock** statement which provides a shortcut to using a monitor with **try** / **finally**

```
public void Transfer(decimal amount) {
    lock (guard) {
        SavingsBalance += amount;
        CheckingBalance -= amount;
    }
}
```

Compiler rewrites this code to be this:

```
public void Transfer(decimal amount) {
    Monitor.Enter(guard);
    try {
        SavingsBalance += amount;
        CheckingBalance -= amount;
    }
    finally {
        Monitor.Exit(guard);
    }
}
```

Adding timeout conditions

- `Monitor.Enter` and `lock` both wait **indefinitely** for the lock to be acquired – use `Monitor.TryEnter` method to support timeouts

```
if (!Monitor.TryEnter(guard, 5000))
   throw new Exception("Failed to obtain lock!");
try {
   ... // Access data
}
finally {
   Monitor.Exit(guard);
}
```

Unfortunately, there is no `lock` statement variant that takes a timeout, but there are several `IDisposable` implementations out there – see http://bit.ly/timedlock

Individual Exercise

Using the lock statement in place of Monitor.Enter

Xamarin University

Xamarin University

Lock granularity

- Granularity is an important consideration when defining the lock > data relationship; too coarse and you synchronize too often, too fine and you complicate your code and increase the time it takes to do the work

Use a unique monitor object for each piece of unique protected data

should only share locks when the data is related

Dealing with lock contention

- Should do as little as necessary while you own a lock and release it as soon as possible to reduce lock contention and keep your apps responsive

```
private List<string> data = ...;
...
lock (data) {
    string[] updates = GetUpdatesFromServer();
    data.AddRange(updates);
}
```

Don't ever execute blocking logic while you own a lock!

Dealing with lock contention

- Should do as little as necessary while you own a lock and release it as soon as possible to reduce lock contention and keep your apps responsive

```
private List<string> data = ...;
...
string[] updates = GetUpdatesFromServer();
lock (data) {
    data.AddRange(updates);
}
```

Individual Exercise

Refine the lock usage to reduce contention

Multiple lock coordination

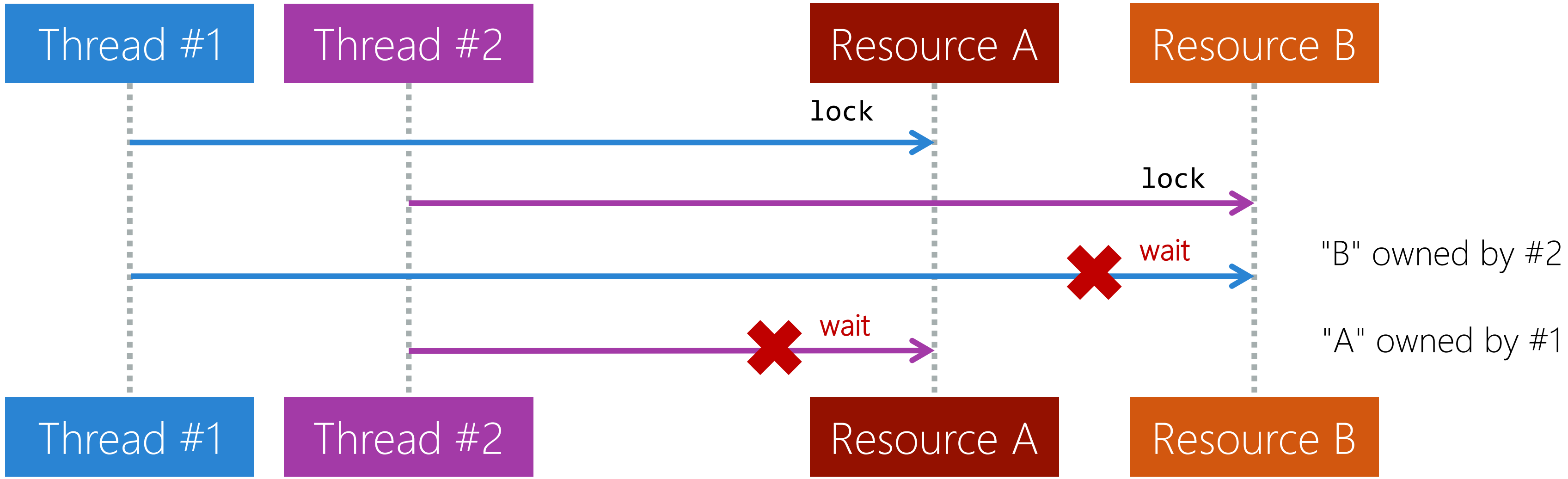
- Be careful about waiting on multiple monitors – this will almost always produce race conditions and deadlock scenarios in your code

```
public class Account
{
    object accountLock = new object();

    public static void Transfer(Account account1, Account account2, decimal amount)
    {
        lock (account1.accountLock)
        {
            lock (account2.accountLock)
            {
                ... // Transfer money
            }
        }
    }
}
```

What if some other thread owns this lock and is waiting on the first one?

Multiple lock coordination

- Be careful about waiting on multiple monitors – this will almost always produce race conditions and deadlock scenarios in your code



Lock ordering

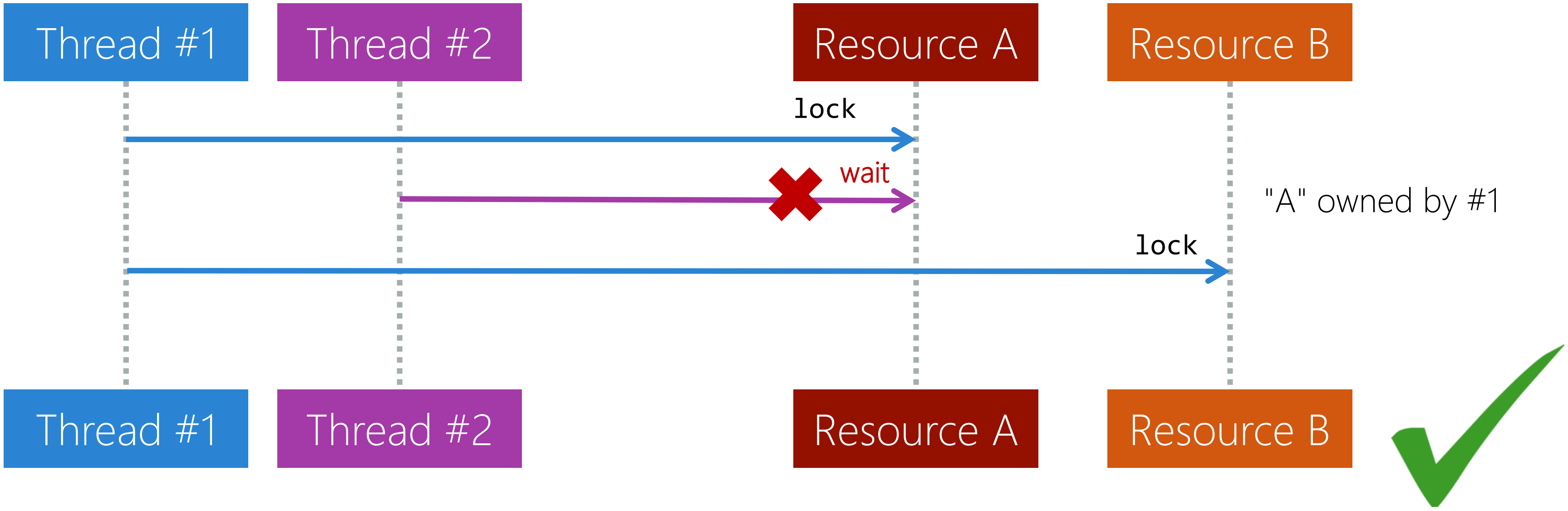
- Can ensure to take locks in a specific, enforced order to avoid deadlock

```
public class Account
{
   object accountLock = new object();
   public static void Transfer(Account account1, Account account2, decimal amount)
   {
      object first = (account1.Id < account2.Id) ? account1.accountLock : account2.accountLock;
      object second = (account1.Id < account2.Id) ? account2.accountLock : account1.accountLock;
      lock (first)
      {
         lock (second)
         {
            ... // Transfer
         }
      }
   }
}
```

Use some ordering technique to ensure you always take locks in a specific order

Multiple lock coordination

- Can ensure to take locks in a specific, enforced order to avoid deadlock



Individual Exercise

Using the lock ordering technique to remove potential deadlocks

Xamarin University

Summary

1. Making multiple statements atomic
2. What is a Monitor?
3. Choosing your Monitor
4. Releasing a Monitor
5. Dealing with Exceptions
6. Coordinating locks

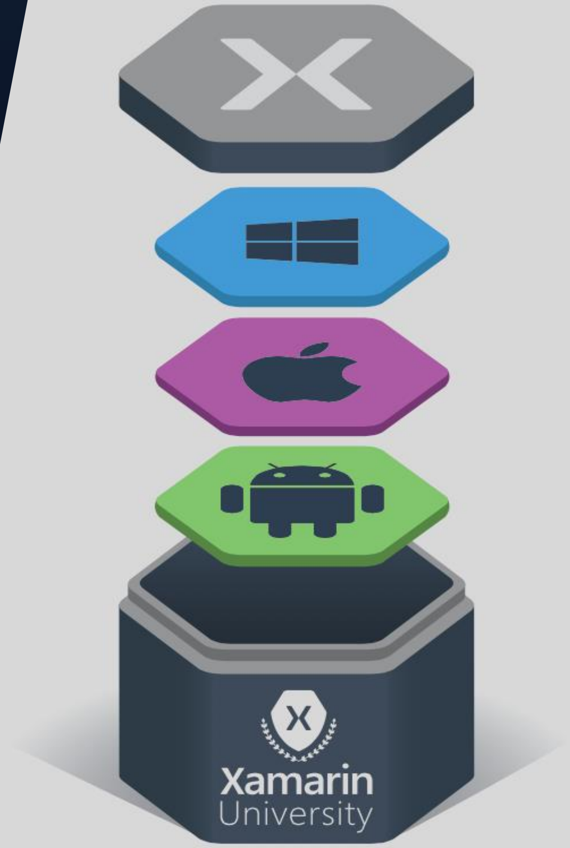
Where are we going from here?

- Thread safety is the hard problem of using multiple threads, but .NET provides several solutions we can use

- In the next class **CSC353** we will look at some more advanced synchronization techniques you can use in your code, including some thread-safe collections!

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile



Microsoft