Thread Synchronization

introduction to thread synchronization in .NET

Overview

- **Introduction to thread synchronization in .NET**
  - Motivation
  - Techniques & primitives
    - atomic updates of machine word-sized values
    - data partitioning
    - wait-based synchronization primitives
  - Deadlock

Motivation

- **Most resources are not meant to be accessed concurrently**
  - Collections (arrays, linked-lists, etc.)
  - Files
  - …
  - Even integers

```
partial class Program
{
    static void AddOne()
    {
        sum++;
    }
}
```

...

```
partial class Program
{
    static int sum = 0;

    static void Main()
    {
        Thread[] threads = new Thread[10];

        for (int n = 0; n < threads.Length; n++)
        {
            threads[n] = new Thread(AddOne);
            threads[n].Start();
        }

        for (int n = 0; n < threads.Length; n++)
        {
            threads[n].Join();
        }

        Console.WriteLine(sum);
    }
}
```

What *should* this program display?
What *will* this program display?

Critical Sections

- A *critical section* is a region of code that accesses a shared resource

shared resource

critical section of code

```
partial class Program
{
    static void AddOne()
    {
        sum++;
    }
}
```

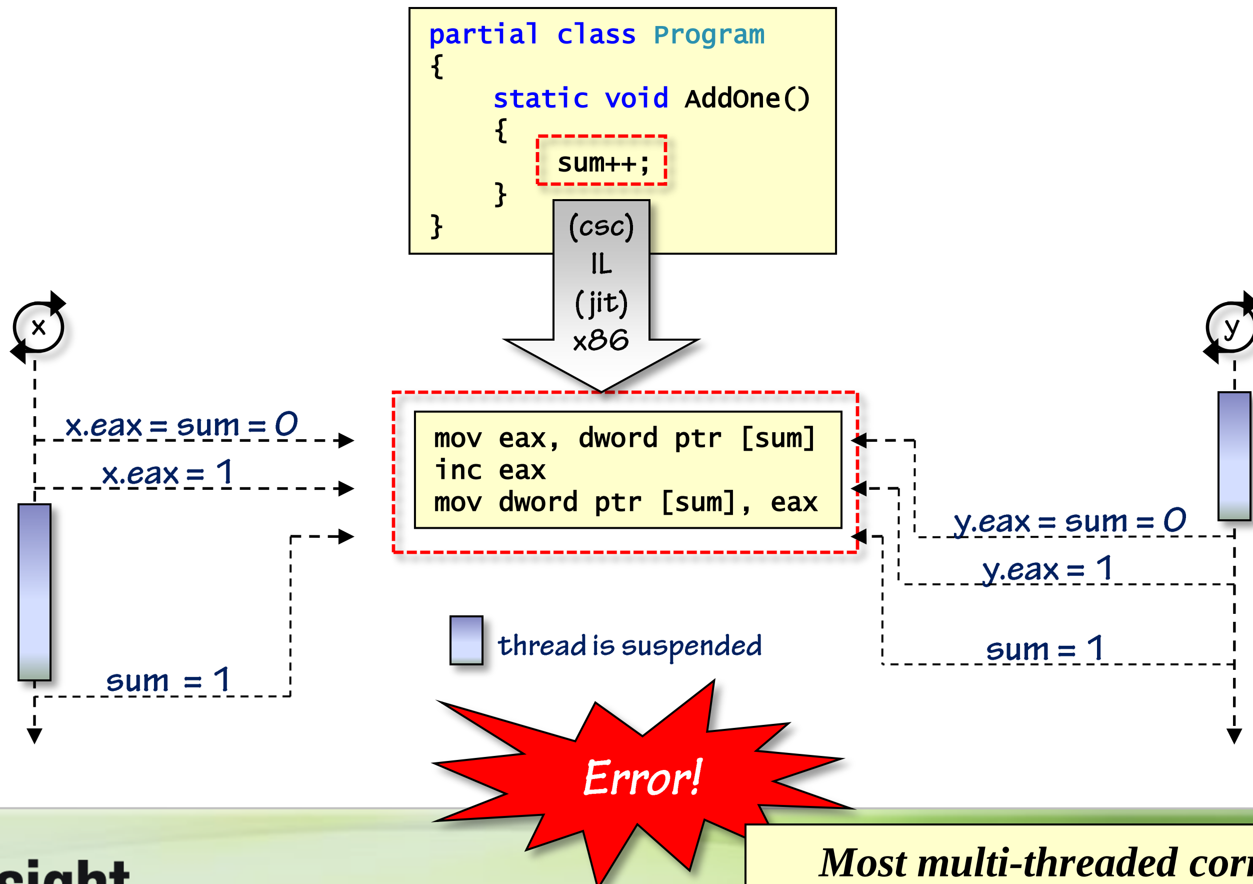
...

```
partial class Program
{
    static int sum = 0;

    static void Main()
    {
        Thread[] threads = new Thread[10];

        for (int n = 0; n < threads.Length; n++)
        {
            threads[n] = new Thread(AddOne);
            threads[n].Start();
        }

        for (int n = 0; n < threads.Length; n++)
        {
            threads[n].Join();
        }

        Console.WriteLine(sum);
    }
}
```

Program outcome is undefined.

Race Conditions

- A *race condition* **occurs when the outcome may be affected by timing**
  - e.g., uncontrolled access to a critical section



Most multi-threaded correctness issues revolve around race conditions…

Solution #1: Atomic Updates

- **Most processors support atomic updates of word-sized data**

```
mov eax, dword ptr [sum]
inc eax
mov dword ptr [sum], eax
```

not atomic

atomic

```
mov ecx, dword ptr [sum]
mov eax, 1
lock xadd dword ptr [ecx], eax
```

"lock" is an x86 instruction prefix that coordinates multi(core|processor) access to memory

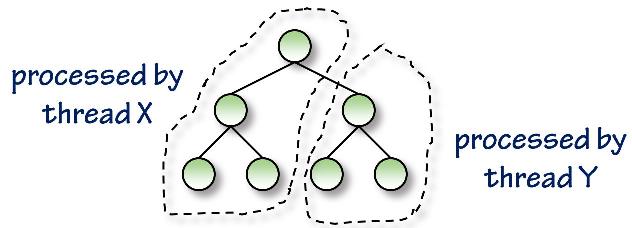
- **The FCL provides a processor-independent suite of atomic updates**
  - ***System.Threading.Interlocked.Xxx***

```
partial class Program
{
    static void AddOne()
    {
        Interlocked.Increment(ref sum);
    }
}
```

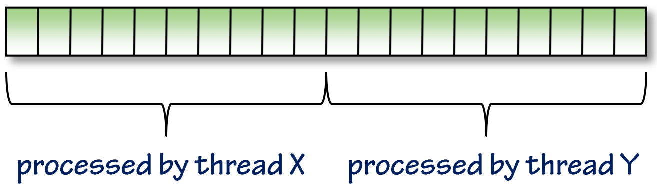
now a thread-safe read/modify/write

Solution #2: Data Partitioning

- **Sometimes can partition data to orchestrate multi-threaded access**
  - Depends on the data or resources being operated on
  - Requires problem-domain specific programming
  - "You work on this while I work on that" model
  - e.g., directory-based file operations



  - e.g., array manipulations



Solution #3: Wait-Based Synchronization

- **Sometimes threads need access to *exact same* resource — can't partition**
  - e.g., insert or delete node from a list while other thread(s) are navigating list
  - e.g., multiple threads trying to manipulate the same file
- **Sometimes *data dependencies* prevent a partitioned approach**
  - When the output of one thread is required as input to another
  - e.g., computing the Fibonacci sequence
    - sequence[0] = sequence[1] = 1
    - sequence[n] = sequence[n-1] + sequence[n-2]
- **Such situations require a *wait-based* approach to synchronization**
  - i.e. thread may have to *block* until access is allowed…

Wait-Based Thread Synchronization

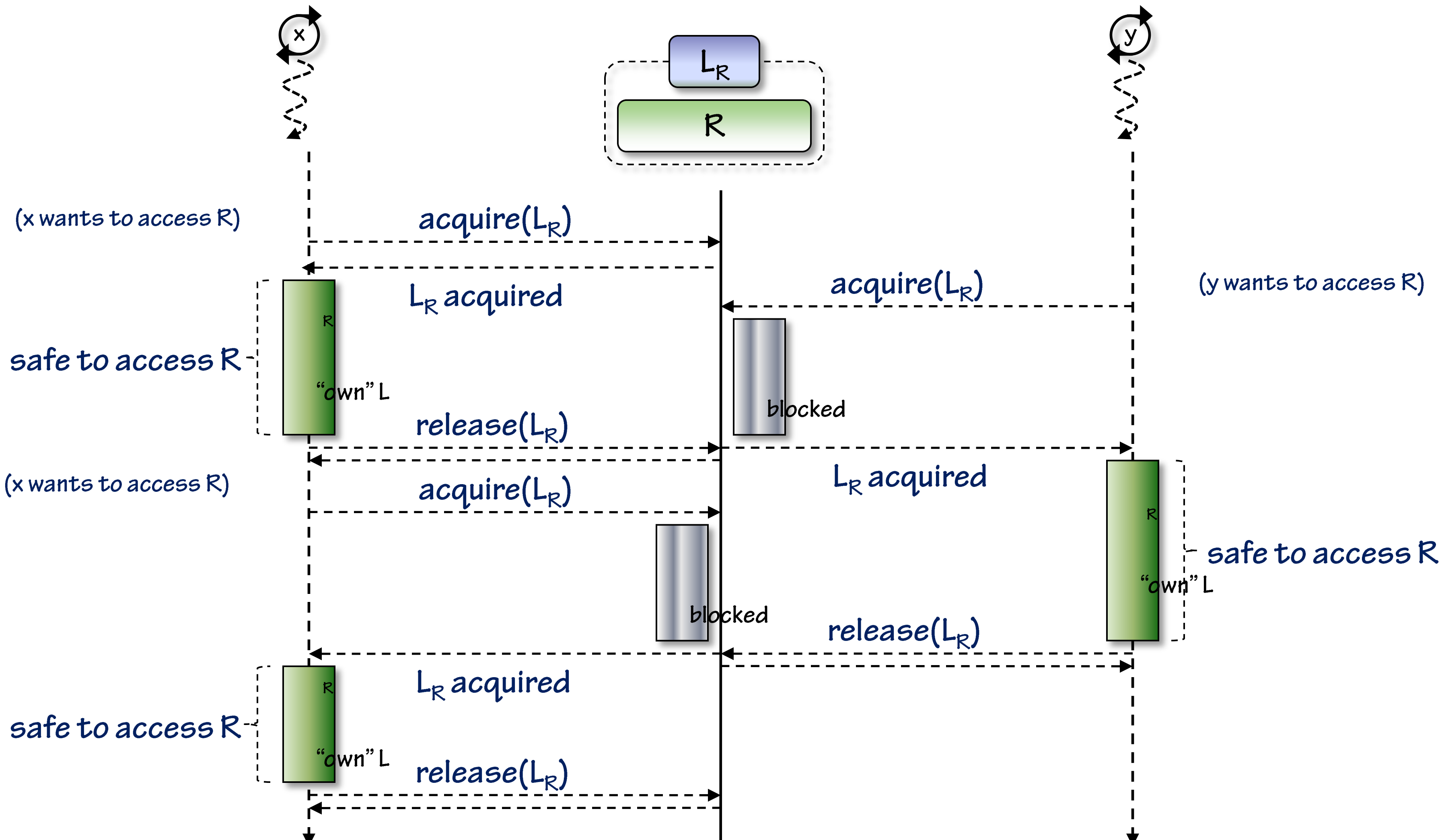
- **Wait-based synchronization is based on a voluntary protocol**
  - A “gentleman’s agreement” or “handshake” model
  - Elements of the protocol:
    - a shared resource is identified (“that array over there”)
    - a synchronization primitive/tool is agreed on (Monitor, mutex, …)
    - an agreed upon *instance* of that primitive is identified
      - lock X guards file X, lock Y guards file Y, etc..
    - any thread wishing to accessing the resource agrees to:
      1. acquire ownership of the agreed upon synchronization primitive
      2. access the shared resource only after ownership acquired
      3. release ownership of the synchronization primitive once access is complete

a blocking operation

  - ***Key point: the protocol is voluntary***

R = a shared resource
L_R = the designated "lock" "protecting" R

Wait-Based Synchronization



Wait-Based Synchronization Primitives

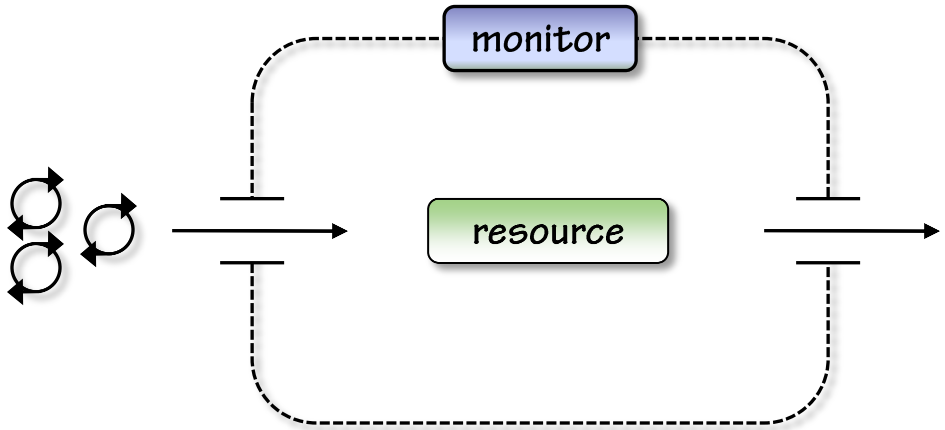
- **Several wait-based synchronization primitives available in the CLR**
  - Monitor
  - Mutex
  - ReaderWriterLockSlim
  - ManualResetEvent, AutoResetEvent
  - Semaphore

classes in System.Threading namespace

- **The first 3 share the same basic usage model**
  - Make a function call to acquire ownership of the "lock"
  - Use the shared resource the designated "lock" is meant to protect
  - Make a function call to release "lock" ownership once no longer needed

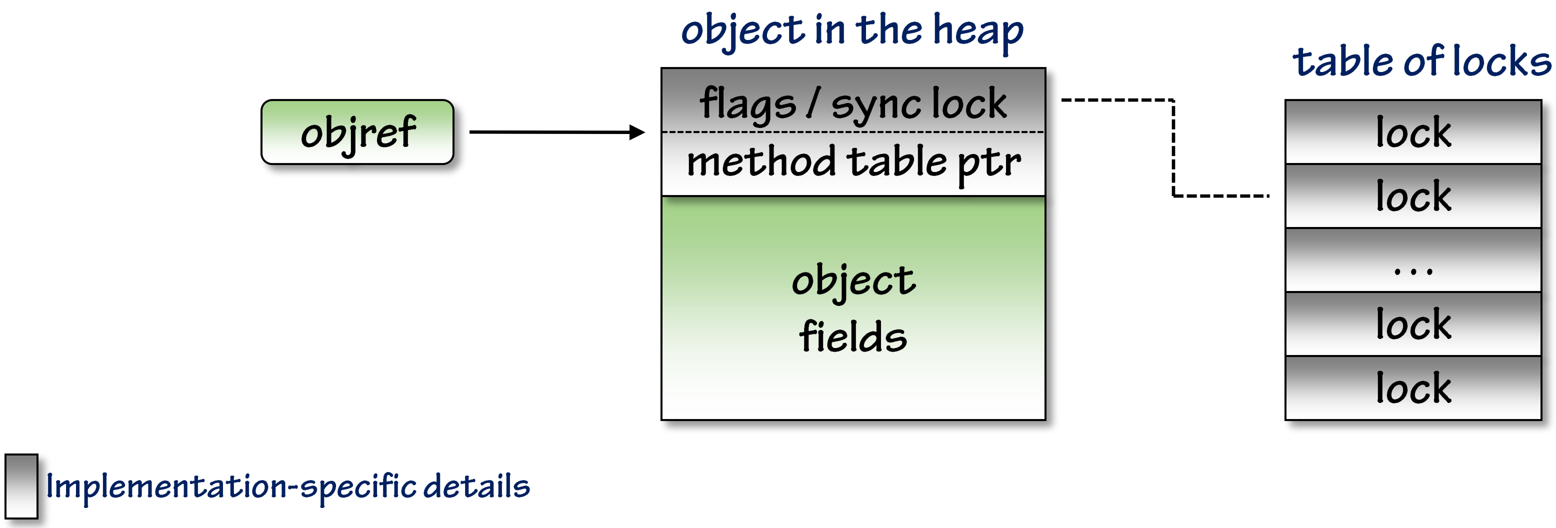
System.Threading.Monitor

- **Monitors models gated access to a resource**
  - Threads agree to “enter” the monitor before accessing shared resource
    - CLR allows only one thread at a time to enter monitor
    - Other threads attempting to enter monitor while in use are blocked
    - May be recursively entered by same thread
  - Threads agree to “exit” the monitor once access to resource is complete
    - Next thread waiting for entry to monitor (if any) is allowed in
    - Recursive entrance operations by same thread require balanced exit operations



Monitors in the CLR

- **Monitor methods operate on object references**
  - Every object in the heap can *potentially* be associated with a lock
  - Lock is demand-initialized by the CLR only if/when needed
  - Index/reference to lock is stored within the CLR-managed object header
  - Therefore, any object may effectively be used as a monitor



Monitor Usage

```
public class Widget2D
{
    int _x;
    int _y;

    public Widget2D(int x, int y)
    {
        _x = x;
        _y = y;
    }

    public void MoveBy(int deltaX, int deltaY)
    {
        _x += deltaX;
        _y += deltaY;
    }

    public void GetPos(out int x, out int y)
    {
        x = _x;
        y = _y;
    }
}
```

critical sections

```
public class Widget2D
{
    int _x;
    int _y;
    object _lock = new object();

    public Widget2D(int x, int y)
    {
        _x = x;
        _y = y;
    }

    public void MoveBy(int deltaX, int deltaY)
    {
        Monitor.Enter(_lock);
        _x += deltaX;
        _y += deltaY;
        Monitor.Exit(_lock);
    }

    public void GetPos(out int x, out int y)
    {
        Monitor.Enter(_lock);
        x = _x;
        y = _y;
        Monitor.Exit(_lock);
    }
}
```

used to gate access to widget fields

access to fields is synchronized

access to fields is synchronized

Exception-Safe Monitor Usage

```
public class Widget2D
{
    int _x;
    int _y;
    object _lock = new object();

    public Widget2D(int x, int y)
    {
        _x = x;
        _y = y;
    }

    public void MoveBy(int deltaX, int deltaY)
    {
        Monitor.Enter(_lock);
        _x += deltaX;
        _y += deltaY;
        Monitor.Exit(_lock);
    }

    public void GetPos(out int x, out int y)
    {
        Monitor.Enter(_lock);
        x = _x;
        y = _y;
        Monitor.Exit(_lock);
    }
}
```

exception-prone

exception-safe

```
public class Widget2D
{
    int _x;
    int _y;
    object _lock = new object();

    public void MoveBy(int deltaX, int deltaY)
    {
        Monitor.Enter(_lock);

        try {
            _x += deltaX;
            _y += deltaY;
        }
        finally {
            Monitor.Exit(_lock);
        }
    }

    public void GetPos(out int x, out int y)
    {
        Monitor.Enter(_lock);

        try {
            x = _x;
            y = _y;
        }
        finally {
            Monitor.Exit(_lock);
        }
    }
}
```

C# Monitor Usage

```
public class Widget2D
{
    int _x;
    int _y;
    object _lock = new object();

    public void MoveBy(int deltaX, int deltaY)
    {
        Monitor.Enter(_lock);

        try {
            _x += deltaX;
            _y += deltaY;
        }
        finally {
            Monitor.Exit(_lock);
        }
    }

    public void GetPos(out int x, out int y)
    {
        Monitor.Enter(_lock);

        try {
            x = _x;
            y = _y;
        }
        finally {
            Monitor.Exit(_lock);
        }
    }
}
```

C# equivalent

```
public class Widget2D
{
    int _x;
    int _y;
    object _lock = new object();

    public void MoveBy(int deltaX, int deltaY)
    {
        lock(_lock)
        {
            _x += deltaX;
            _y += deltaY;
        }
    }

    public void GetPos(out int x, out int y)
    {
        lock(_lock)
        {
            x = _x;
            y = _y;
        }
    }
}
```

Hold & Wait

- **Sometimes a thread needs to wait for something *while* holding a lock**
  - e.g.: a resource to be provided or replenished
  - e.g.: another lock
- **Resource replenishment or other condition**
  - producer/consumer model for resource handling
  - blocking semantics for consumers when resource(s) not available
- **Multiple lock acquisition**
  - atomic updates of multiple resources, each protected by own lock

Hold & Wait with Monitors

- **The Monitor class supports safe hold-and-wait operations**
  - Via the Pulse, PulseAll, Wait methods

consumers

producer

```
public class Bakery
{
    Queue<Donut> _donutTray = new Queue<Donut>();

    public Donut GetDonut()
    {
        lock (_donutTray)
        {
            if (_donutTray.Count == 0) {
                // ??? wait for producer to refill
            }

            return _donutTray.Dequeue();
        }
    }

    public void RefillTray(Donut[] freshDonuts)
    {
        lock (_donutTray)
        {
            foreach (Donut d in freshDonuts) {
                _donutTray.Enqueue(d);
            }

            // ??? unblock waiting consumers (if any)
        }
    }
}
```

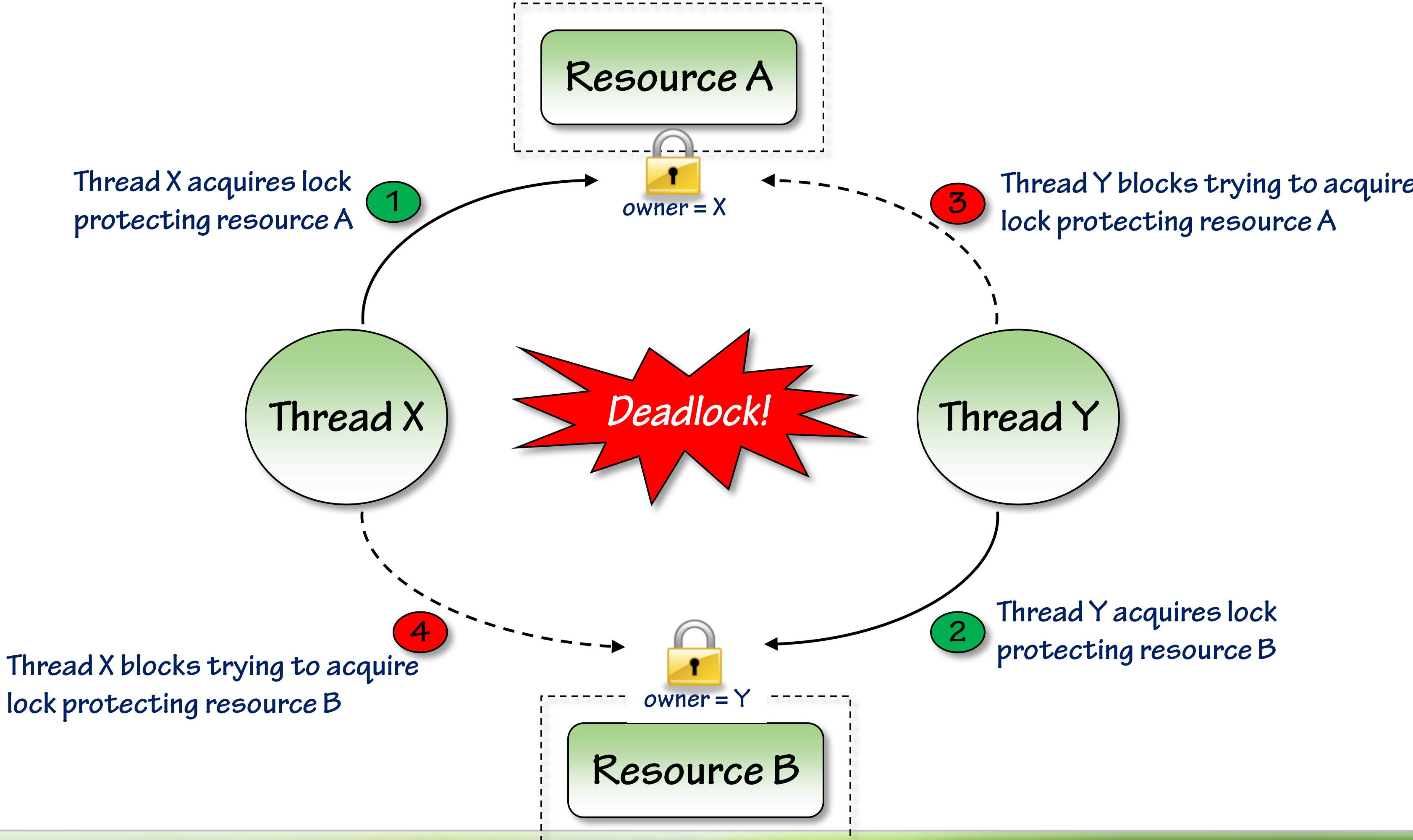
```
public class Bakery
{
    Queue<Donut> _donutTray = new Queue<Donut>();

    public Donut GetDonut()
    {
        lock (_donutTray)
        {
            while (_donutTray.Count == 0) {
                Monitor.Wait(_donutTray);
            }

            return _donutTray.Dequeue();
        }
    }

    public void RefillTray(Donut[] freshDonuts)
    {
        lock (_donutTray)
        {
            foreach (Donut d in freshDonuts) {
                _donutTray.Enqueue(d);
            }

            Monitor.PulseAll(_donutTray);
        }
    }
}
```

Deadlock

- **Deadlock can occur whenever a *hold-and-wait* situation is possible**
  - While holding one lock, a thread attempts to acquire another lock

- **Note that the above is rife with caveats:**
  - Deadlock can occur (but won't necessarily occur)
    - requires 2 or more threads competing for the same set of locks
  - Deadlock is possible (but not necessarily probable)
    - probability increases as # of threads/processors/cores increases
  - Deadlock may only be temporary
    - if timeouts are used in all lock acquisition calls

Deadlock Illustrated



pluralsight
see what you can learn

Mutexes

- **A *mutex* is a Win32 kernel object**
  - ***System.Threading.Mutex*** provides an FCL wrapper for managed code
- **Benefits**
  - Supports timeout-limited lock acquisition
  - Nameable; enabling cross-process (same-machine) thread synchronization
  - Enables deadlock-free multiple lock acquisition via ***WaitHandle.WaitAll***
- **Tradeoffs**
  - Acquisition & release calls always incur roundtrip to/from kernel mode
  - Underlying kernel object must be closed when no longer needed
    - handled automatically (but on a delayed basis) by GC/finalization mechanics

Deadlock-Prone Multiple Lock Acquisition

```
public class BankAccount
{
    double _balance;
    object _lock = new object();

    public void Credit(double amt)
    {
        lock (_lock)
        {
            _balance += amt;
        }
    }

    public void Debit(double amt)
    {
        lock (_lock)
        {
            _balance -= amt;
        }
    }

    public void TransferFrom(BankAccount otherAcct, double amt)
    {
        lock (this._lock)
        {
            lock (otherAcct._lock)
            {
                otherAcct.Debit(amt);
                this.Credit(amt);
            }
        }
    }
}
```

thread-safe

Vulnerable to deadlock!

Hold-and-wait potential exists based on timing

pluralsight
see what you can learn

Deadlock-Free Multiple Lock Acquisition

```
public partial class BankAccount
{
    double _balance;
    Mutex _lock = new Mutex();

    public void Credit(double amt)
    {
        if (_lock.WaitOne())
        {
            try
            {
                _balance += amt;
            }
            finally
            {
                _lock.ReleaseMutex();
            }
        }
    }

    public void Debit(double amt)
    {
        if (_lock.WaitOne())
        {
            try
            {
                _balance -= amt;
            }
            finally
            {
                _lock.ReleaseMutex();
            }
        }
    }
}
```

now using a Mutex

Mutex based lock acquisition

```
public partial class BankAccount
{
    public void TransferFrom(BankAccount otherAcct, double amt)
    {
        Mutex[] locks = { this._lock, otherAcct._lock };

        if (WaitHandle.WaitAll(locks))
        {
            try
            {
                otherAcct.Debit(amt);
                this.Credit(amt);
            }
            finally
            {
                foreach (Mutex m in locks)
                {
                    m.ReleaseMutex();
                }
            }
        }
    }
}
```

all locks acquired without risk of deadlock

Summary

- **Most resources are <u>not</u> meant to be accessed concurrently**
  - Improper access yields race conditions and incorrect results
    - very difficult to detect, debug, and fix
- **Common solutions**
  - Atomic updates of machine-word-sized values
  - Data partitioning
  - Wait-based synchronization (Monitor, Mutex, others)
- **Beware**
  - Critical sections
  - Race conditions
  - Deadlock