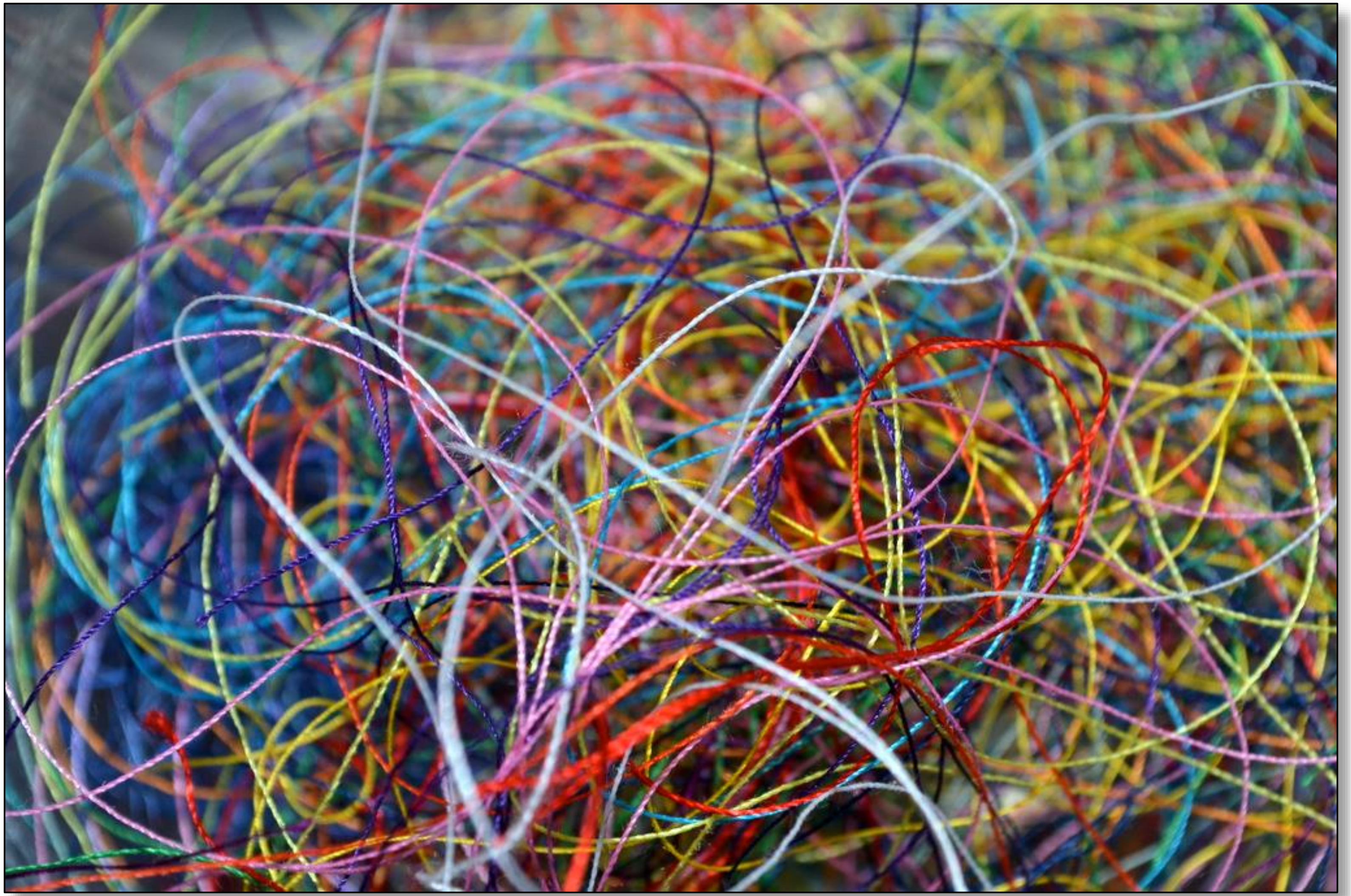
Module 02:

”Blocking Synchronization”

TEKNOLOGISK
INSTITUT

Agenda

- Introducing Synchronization
- Synchronization by Locking
- Best Practices for Locking
- More Blocking Thread Synchronization
- Cross-Process Synchronization



"I Love Close Up Shots." by Jayne ~ Twiggy & Opal is licensed under CC BY-NC-SA 2.0

The Need for Synchronization

- Processor and operating system schedule threads in and out repeatedly
  - Thread context switch can occur at any time
    - Even in the middle of assignments and increments etc.
- Hence computations need to be computationally safe
  - Some operations must be performed indivisibly!
  - Race conditions should be avoided
- Basically three solutions
  - **Blocking synchronization** of access to critical regions of code
  - Signaling between threads
  - Nonblocking synchronization

Agenda

- Introducing Synchronization
- **Synchronization by Locking**
- Best Practices for Locking
- More Blocking Thread Synchronization
- Cross-Process Synchronization

Mutual Exclusion using Monitors

- The **`Monitor`** class is a light-weight mutual exclusion mechanism for use within a single process
  - `Monitor.Enter()`
  - `Monitor.Exit()`

```
object syncObject = new object();
...

Monitor.Enter( syncObject);
_counter++;
Monitor.Exit( syncObject);
```

- What if we forget to exit?
- What about exceptions…?

The C# `lock` Keyword

- The `lock` keyword in C# is based on `Monitor` and `try-finally`

```
object syncObject = new object();
...

lock( syncObject )
{
   _counter++;
}
```

- Note: `lock` can <u>only</u> lock reference types…! Why?

Which Synchronization Object?

- Always choose a reference type instance
- Best practice is to choose independent, private object
- Might even give it a descriptive name

```
object _counterAccessSyncObject = new object();
...

lock( counterAccessSyncObject )
{
   _counter++;
}
```

- Is the **`this`** reference a good choice? Not really!

Access to Static Members

- For exclusive access to static members of some type, convention is to use its Type object

```
lock( typeof(Resource) )
{
   Counter++;
}
```

- Alternatively, create a static synchronization object
  - `typeof(Resource)` suffers similar caveats as **`this`**

Variations: Monitor.TryEnter()

- Enter with a timeout
  - `Monitor.TryEnter()`

```
bool wasAcquired = Monitor.TryEnter( _counterAccessSyncObject );
if( wasAcquired )
{
    _counter++;
    Monitor.Exit( _counterAccessSyncObject );
}
```

Variations: Lock Taken Overloads

- Extreme subtleties lead to more overloads in C# 4.0

```
bool wasLockTaken = false;
try
{
    Monitor.Enter(_counterAccessSyncObject, ref wasLockTaken);
    _counter++;
}
finally
{
    if (wasLockTaken)
    {
        Monitor.Exit(_counterAccessSyncObject);
    }
}
```

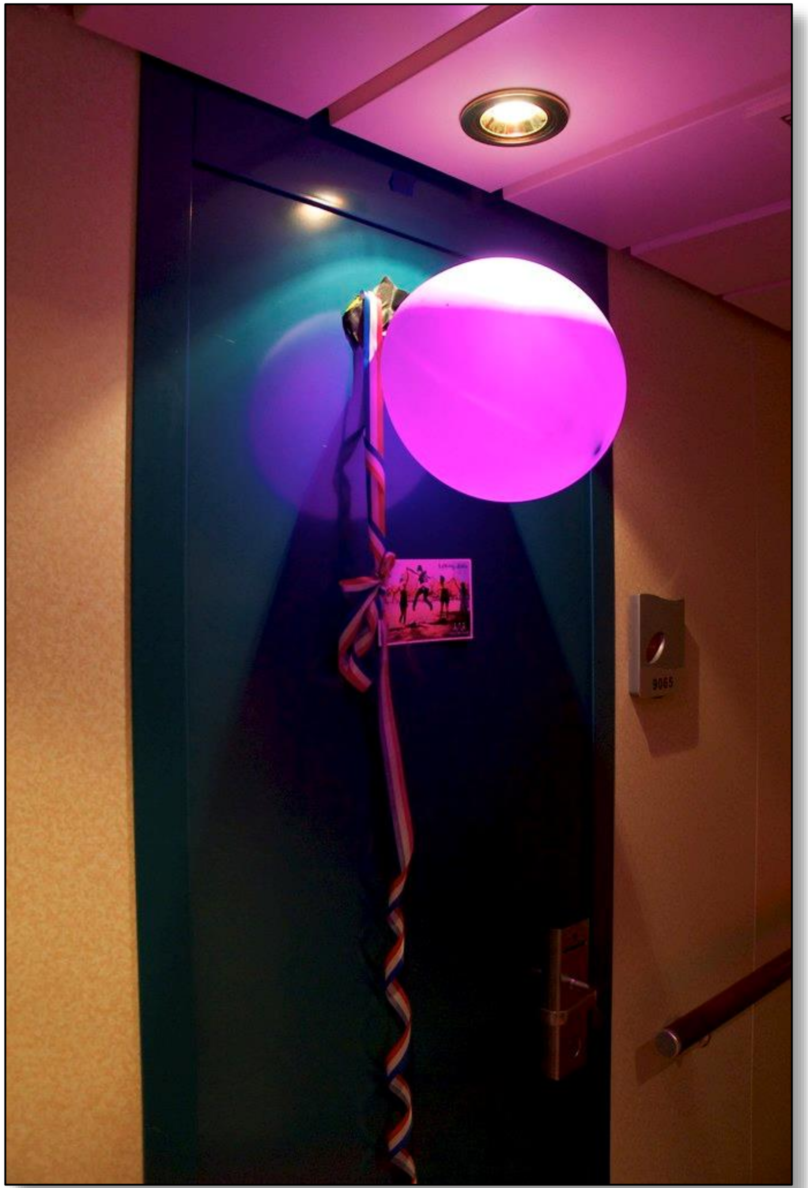
- This is how lock is implemented internally in C# 4

Agenda

- Introducing Synchronization
- Synchronization by Locking
- **Best Practices for Locking**
- More Blocking Thread Synchronization
- Cross-Process Synchronization

Locking is… well… Subtle!

"dead locker" by _boris is licensed under CC BY-NC-SA 2.0



"DSC00128" by askingdave is licensed under CC BY-SA 2.0

TEKNOLOGISK INSTITUT

Locking is just Convention!

- Remember that locks are based on convention
  - They work only in everybody plays along nicely…!
- Locking
  - Requires discipline from everybody
  - Is easy to forget
  - Is hard to detect that it is forgotten
  - Is just hard to get 100% right!
  - Is… Subtle! ☺

Locking Allows Reentrancy

- Same thread can acquire the same lock **multiple** times
  - Only blocks on the initial attempt to acquire lock

```
lock (_counterAccessSyncObject)
{
    ...
    lock (_counterAccessSyncObject)
    {
        _counter++;
    }
}
```

- Try to avoid doing this too extensively

Accessing Multiple Resources

- Use one or more locks for multiple resources

```
lock (_counterFromAccessSyncObject)
{
    lock (_counterToAccessSyncObject)
    {
        _counterFrom--;
        _counterTo++;
    }
}
```

- Beware of granularity!
  - Fewer locks a.k.a. *"Coarse-grained"* => Performance Hit?
  - More locks a.k.a. *"Fine-grained"* => Deadlock Risk?

Deadlocks

- When
  - Thread 1 has acquired Resource A and waits for Resource B
  - Thread 2 has acquired Resource B and waits for Resource A

  then a deadlock has occurred..!

- Deadlocks
  - might not occur deterministically
  - cannot be detected automatically by humans or compiler
  - are hard to find and debug!

- "Livelocks" also exist and are equally painful (but rare)

Lock-Levelling

- Use strict locking discipline called "Lock-Levelling"
  - Assign some fictitious number to each resource
  - Ensure that any thread always **only locks** a lock with a **higher** number than **any** lock it already holds

Thread1 → Resource A ← Thread2
Thread1 → Resource B
Thread1 → Resource C ← Thread2
Resource D ← Thread2

- Alternative is nonblocking synchronization later

Best Practices for Locking

- Lock access to *any* (writeable,) shared fields!
- Carefully consider your granularity of locking
- Use lock-levelling and document the levels in code
- Never lock when invoking blocking methods
  - E.g. calling external WCF services

Agenda

- Introducing Synchronization
- Synchronization by Locking
- Best Practices for Locking
- **More Blocking Thread Synchronization**
- Cross-Process Synchronization

Reader/Writer Locks

- If there are many readers and only occasional writers, the `ReaderWriterLockSlim` might be more performant

```
try
{
    _rwLock.EnterReadLock();
    Console.WriteLine(_number);
}
finally
{
    _rwLock.ExitReadLock();
}
```

```
try
{
    _rwLock.EnterWriteLock();
    _number = 87;
}
finally
{
    _rwLock.ExitWriteLock();
}
```

- A write lock is universally exclusive.
- A read lock is compatible with other read locks.

- For advanced scenarios read locks may be upgraded to a write lock and vice versa downgraded

TEKNOLOGISK
INSTITUT

"Ok, Folks" by Valerie Reneé is licensed under CC BY-NC-ND 2.0

Semaphores

- Semaphores limit the count of concurrency to a resource
  - Semaphore(1) ~ Monitor
  - Note: Semaphores have no record of owner threads!

```
_semaphore = new SemaphoreSlim(3);
...

_semaphore.Wait();
...
_semaphore.Release();
```

- `SemaphoreSlim`
  - Lightweight .NET 4.0 version of **`Semaphore`**
  - Has asynchronous features
  - Local-only!

Agenda

- Introducing Synchronization
- Synchronization by Locking
- Best Practices for Locking
- More Blocking Thread Synchronization
- **Cross-Process Synchronization**

Mutexes

- A mutex is a cross-process version of Monitor
  - Uses Windows Kernel object in OS

```
_mutex = new Mutex(false, "MyResourceMutex");
...

_mutex.WaitOne();
...
_mutex.ReleaseMutex();
```

- Can be both local and cross-process, e.g.
  - Ensure mutually exclusive access to machine-wide ressource
  - Ensure at most one instance of application is running

Semaphore

- `Semaphore`
  - Uses Windows Kernel object in OS
  - Can be both local and cross-process
- `SemaphoreSlim`
  - Faster and better than `Semaphore` when local
  - Local-only!

Summary

- Introducing Synchronization
- Synchronization by Locking
- Best Practices for Locking
- More Blocking Thread Synchronization
- Cross-Process Synchronization

TEKNOLOGISK
INSTITUT

WINCUBATE

Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net

Ringgårdsvej 4A
8270 Højbjerg
Denmark