# Module 04: "Nonblocking Synchronization"

# Agenda

▸ **Introducing Nonblocking Synchronization**
▸ Interlocking
▸ Memory Barriers a.k.a. Fences
▸ Volatile

# Thread Safety

*"A program or method is said to be thread-safe if it has no indeterminacy in the face of any multithreading scenario"*

# What does Thread Safety mean?

▸ It's quite hard to give a preci

▸ Produces predictable and consistent answers regardless of scheduling

▸ Free of deadlocks
▸ Free of livelocks
▸ Free of resource starvation
▸ Free of race conditions

▸ …

▸ Best practice is to use locking (and perhaps) signaling to obtain thread safety

# What is Nonblocking Synchronization?

▸ For extreme performance scenarios there are some very advanced options for avoiding blocking

▸ Often require very intricate knowledge of
- CLR
- CPU
- Compiler optimizations

# Agenda

▸ Introducing Nonblocking Synchronization

▸ **Interlocking**

▸ Memory Barriers a.k.a. Fences

▸ Volatile

# Atomicity

▸ A single read or write of 32 bits (or less) is atomic on 32-bit, but not atomic on 64-bit

```
int _x, _y;
long _z;
...

long l;
_x = 3;        ⟵ ——————— Atomic
_z = 3;        ⟵ ——————— Not atomic
l = _z;        ⟵ ——————— Not atomic
_y += _x;      ⟵ ——————— Not atomic
_x++;          ⟵ ——————— Not atomic
```

▸ Locking essentially simulates atomicity for blocks of code

# Interlocked

▸ Helper class to make these assignments atomic
▸ **Interlocked.**
  - Read() + Add()
  - Increment() + Decrement()
  - Exchange() + CompareExchange()

```
long b;

Interlocked.Increment(ref _a);
Interlocked.Add(ref _a, 42);
Console.WriteLine(Interlocked.Read(ref _a));

Interlocked.Exchange(ref _a, 87);
Interlocked.CompareExchange(ref _a, 176, 87);
```

# Agenda

▸ Introducing Nonblocking Synchronization
▸ Interlocking
▸ **Memory Barriers a.k.a. Fences**
▸ Volatile

# A Simple Puzzle (or...?)

```csharp
private int _answer = 42;
private bool _complete;

public void Access1()
{
    _answer = 87;
    _complete = true;
}

public void Access2()
{
    if (_complete)
    {
        Console.WriteLine(_answer); // <-- What is printed?
    }
}
```

# Safe Optimizations

▸ Safe optimizations for single-threaded programs:

▸ The compiler, CLR, or CPU may reorder your program's instructions to improve efficiency.

▸ The compiler, CLR, or CPU may cache variables such that assignments to variables won't be visible to other threads right away.

▸ Are these safe for multi-threaded programs also?

# MemoryBarrier

▸ Manually suppress unsafe optimizations

```
public void Access1()
{
    _answer = 87;
    Thread.MemoryBarrier();
    _complete = true;
    Thread.MemoryBarrier();
}

public void Access2()
{
    Thread.MemoryBarrier();
    if (_complete)
    {
        Thread.MemoryBarrier();
        Console.WriteLine(_answer); // <-- 87 is printed!
    }
}
```

# Full Memory Barriers

▸ At least all of these generate a full memory barrier:
- `Thread.MemoryBarrier()`
- Interlocked methods
- `Thread.Sleep()` + `Thread.Join()` + `Thread.SpinWait()`

- C# **lock** statement
- `Monitor.Enter()` + `Monitor.Exit()`

- Any event signaling and waiting

- `Thread.VolatileRead()` + `Thread.VolatileWrite()`

- `Task.Start()` + `Task.Wait()` + Task continuations

*Module 02*

*Module 03*

*Up shortly*

*Later…*

# Agenda

▸ Introducing Nonblocking Synchronization
▸ Interlocking
▸ Memory Barriers a.k.a. Fences
▸ **Volatile**

# Volatile

- The **volatile** keyword
  - Generates *Acquire-fence* on reads
    - prevents other reads/writes from being moved *before* the fence
  - Generates *Release-fence* on writes
    - a release-fence prevents other reads/writes from being moved *after* the fence

- Confused? That's quite understandable! ☺

```
volatile bool _complete;

public void Access1()
{
    _answer = 87;
    _complete = true;
}
public void Access2()
{
    if (_complete)
    {
        WriteLine(_answer);
    }
}
```

# What Volatile Means

| Operation | Hardware | | | CLR 2.0+ | |
| --- | --- | --- | --- | --- | --- |
| | Intel x86 & Intel64 | IA 64 | AMD64 | Without Volatile | With Volatile |
| Read, Read | No | Yes | No | Yes | No |
| Read, Write | No | Yes | No | Yes | No |
| Write, Write | No | Yes | No | No | No |
| Write, Read | Yes, only if load and store are to different locations. | Yes | Yes, only if load and store are to different locations. | **Yes** | **Yes** |

▸ No, volatile does NOT incur "latest is always read"!

# Joe Duffy's Bizarre Example

▸ May end up as a == 0 and b == 0

```
volatile int x, y;

public void Access1()
{
    x = 1;              // Volatile write (release-fence)
    int a = y;          // Volatile read (acquire-fence)
    Console.WriteLine(a);
}

public void Access2()
{
    y = 1;              // Volatile write (release-fence)
    int b = x;          // Volatile read (acquire-fence)
    Console.WriteLine(b);
}
```
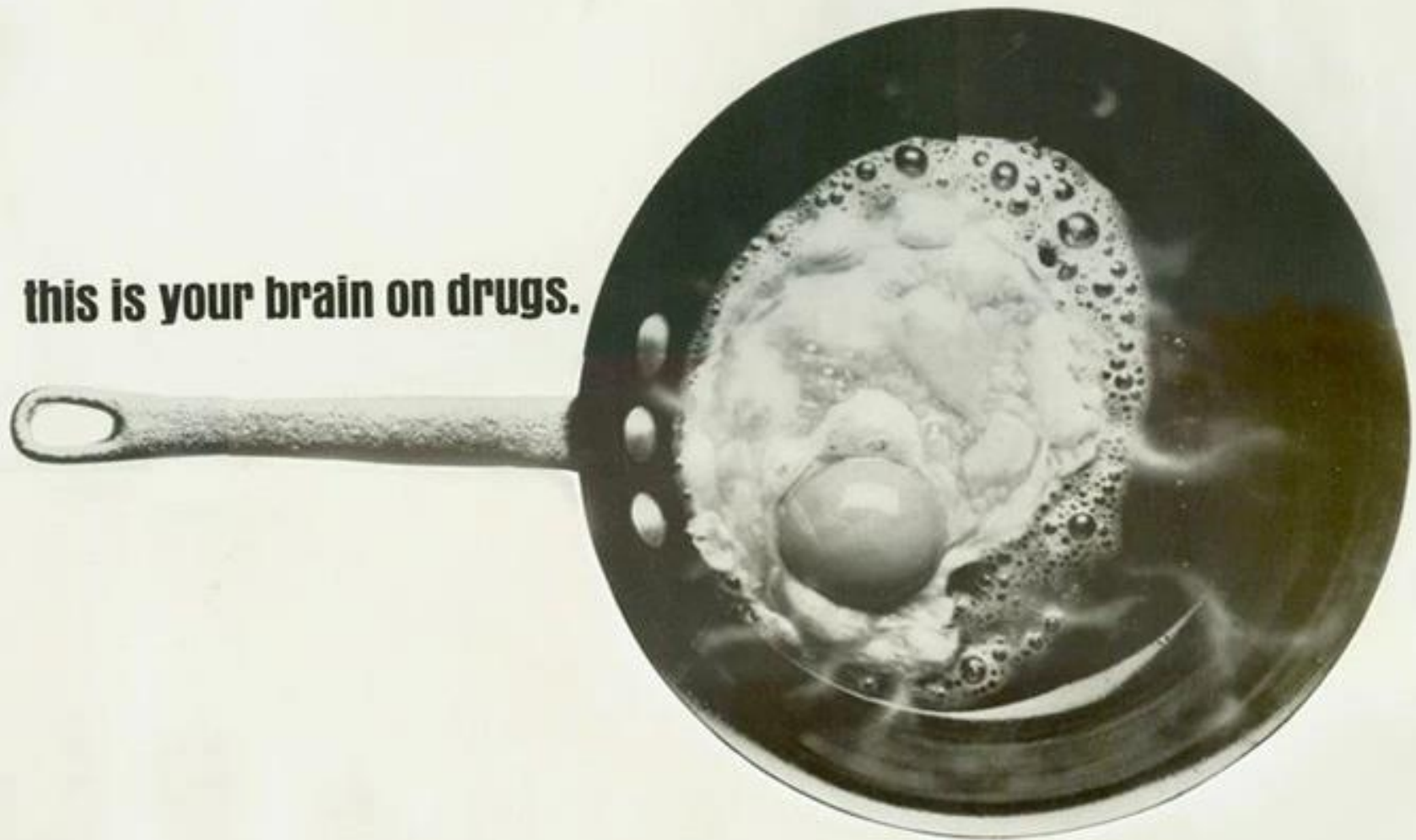
# Volatile Reads and Writes

▸ volatile ~ Thread.VolatileRead + Thread.VolatileRead

```
class Thread
{
    ...
    public static void VolatileWrite( ref int address, int value)
    {
        MemoryBarrier(); address = value;
    }

    public static int VolatileRead (ref int address)
    {
        int num = address; MemoryBarrier(); return num;
    }
}
```

this is your brain on drugs.

# Best Practices for Synchronization

▶ Do
- Use lock
- Use Monitors
- Use Events and Handles
- Use Interlocked

▶ Don't
- Use `volatile`
- Use `Thread.VolatileRead()` or `Thread.VolatileWrite()`
- Use explicit memory barriers

# Summary

▸ Introducing Nonblocking Synchronization
▸ Interlocking
▸ Memory Barriers a.k.a. Fences
▸ Volatile

# 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