rule FINANCIAL

The Sector Specialists

.net threading

Submitted by: Grzegorz Jachimko

on: November 2014

Version: 1.0

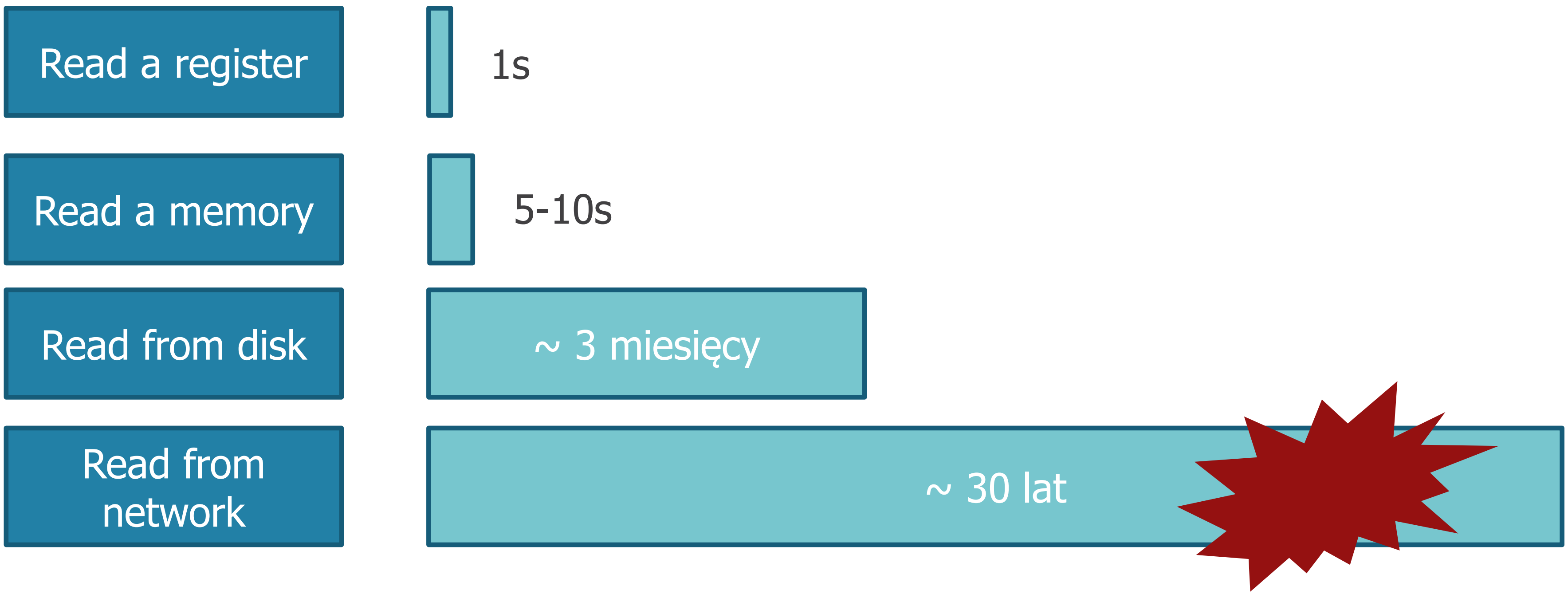
Confidential

Agenda

- Asynchronous processing in .net
- Async model in .net
- I/O operations
- Thread, ThreadPool, Task, async/await
- Synchronization

Some basics

- Processor → 1 cycle = 1 second



Asynchronous processing in .net

- Windows supports 1+ thread / process to enable multitasking
- But … thread is a cost
  - Kernel object (thread's properties, registers – a few KB)
  - Stack (1MB user mode, ~12/24 KB kernel mode)
  - Dll attach and detach
  - Time slots - ~20 ms per thread
- Switching between threads:
  - Enter kernel mode
  - Store register values
  - Select next runnable thread
  - Load register values
  - Exit kernel mode

Asynchronous processing in .net

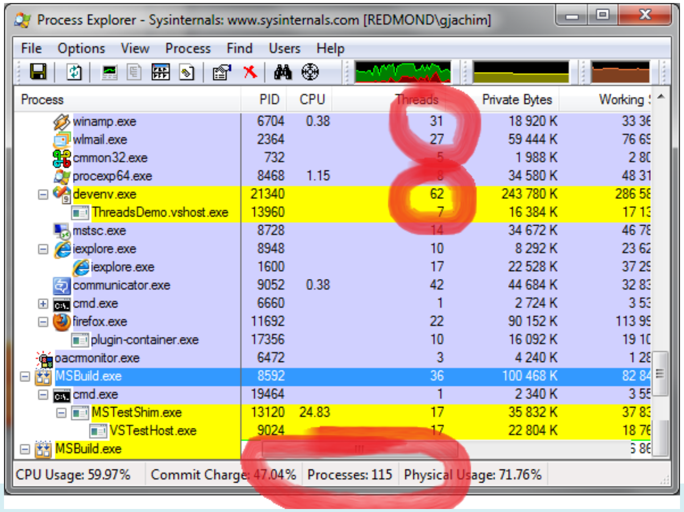
DEMO

How many threads can you run in 32 bit process?

Asynchronous processing in .net

Threads – cons vs pros

- Context switching is a waste of time
  - … but required to keep the system roboust and responsive ☹
- So?
  - Avoid threads whenever possible,
  - Use threads to ensure scalability and responsivness
- Ideally – 1 process (1 thread) per processor
- Reality:



Process Explorer - Sysinternals: www.sysinternals.com [REDMOND\gjachim]

File Options View Process Find Users Help

Process	PID	CPU	Threads	Private Bytes	Working S
winamp.exe	6704	0.38	31	18 920 K	33 36
wlmail.exe	2364		27	59 444 K	76 69
cmmon32.exe	732		5	1 988 K	2 80
procexp64.exe	8468	1.15	8	34 580 K	48 31
devenv.exe	21340		62	243 780 K	286 58
ThreadsDemo.vshost.exe	13960		7	16 384 K	17 13
mstsc.exe	8728		14	34 672 K	46 78
iexplore.exe	8948		10	8 292 K	23 62
iexplore.exe	1600		17	22 528 K	37 29
communicator.exe	9052	0.38	42	44 684 K	32 83
cmd.exe	6660		1	2 724 K	3 53
firefox.exe	11692		22	90 152 K	113 99
plugin-container.exe	17356		10	16 092 K	19 10
oacmonitor.exe	6472		3	4 240 K	1 28
MSBuild.exe	8592		36	100 468 K	82 84
cmd.exe	19464		1	2 340 K	3 55
MSTestShim.exe	13120	24.83	17	35 832 K	37 83
VSTestHost.exe	9024		17	22 804 K	18 76
MSBuild.exe					5 86

CPU Usage: 59.97% Commit Charge: 47.04% Processes: 115 Physical Usage: 71.76%

Async model in .net

Definition

- Asynchronous programming model is a **KEY** to create **scallable** and **non-blocking** applications
  - Threads are not blocked while waiting for the operation to complete
    - Blocked thread still consumes resources,
    - Blocked thread causes more threads to be created,
    - More threads = more hassle when running GC

Async model in .net

Types of asynchronous operations

- **CPU bound operations** – a set of instructions to be executed by the processor only

- **I/O bound** – hardware (disk, network card, dvd player) has to fetch some data for us – thread has nothing to do!

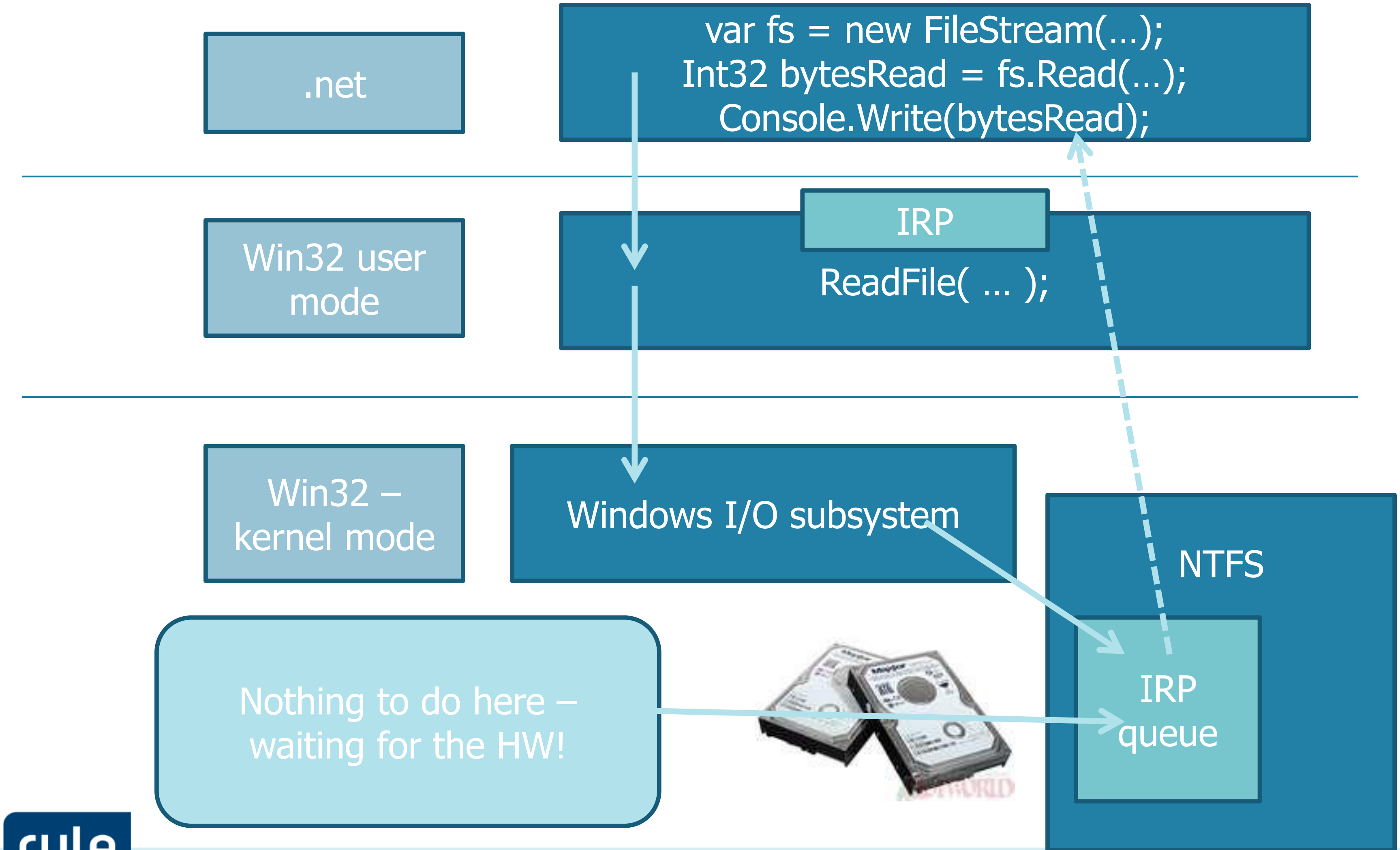
Async model in .net

DEMO

- Present the difference between IO and CPU bound operations

I/O operations

How does windows handle synchronous I/O?



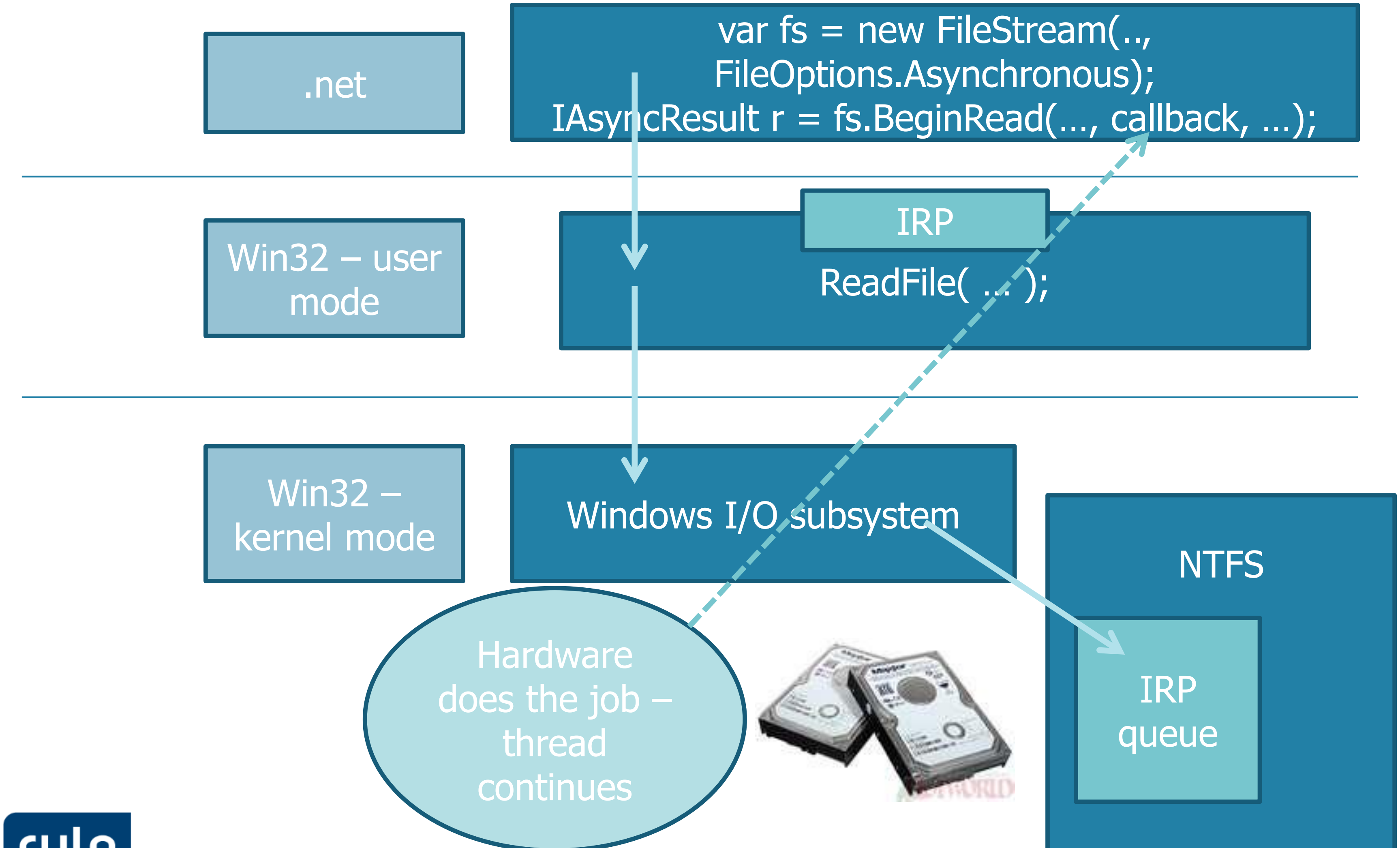
I/O operations

Why this is important?

- 2 main goals:
  - Do not block a thread while performing I/O operation
  - Keep the processor busy with least threads possible
- Windows I/O completion port
- I/O operations done in async:
  - Not blocking calling thread
  - Completed IRP get to I/O CP (CLR)
  - Each completed operation needs a thread assigned

I/O operations

How does windows handle asynchronous I/O?



I/O operations

DEMO

- How to read a file using APM ?

Thread, ThreadPool, Task, async/await

What we have here?

- Threads
  - Full control over creation and configuration,
  - Expensive (only when you call Start)
  - Use them – if your operation is long running (>500 ms) or does blocking I/O
  - new Thread(ThreadStart threadFunction)

Thread, ThreadPool, Task, async/await

What we have here?

- ThreadPool
  - CLR controls lifetime
  - Do not change thread’s properties (priority, name, etc.)
  - Suggested for short, CPU bound operations (<500 ms)
  - ThreadPool.QueueUserWorkItem(WaitCallback callback, object state)

Thread, ThreadPool, Task, async/await

What we have here?

- TaskPool
  - CLR controls lifetime
  - Abstraction on top of ThreadPool, same rules apply
  - Optimizations for multi-core environments
  - Supports reading operation’s value!
  - Task.Factory.StartNew<TResult>(Func<TResult> func)

Thread, ThreadPool, Task, async/await

What we have here?

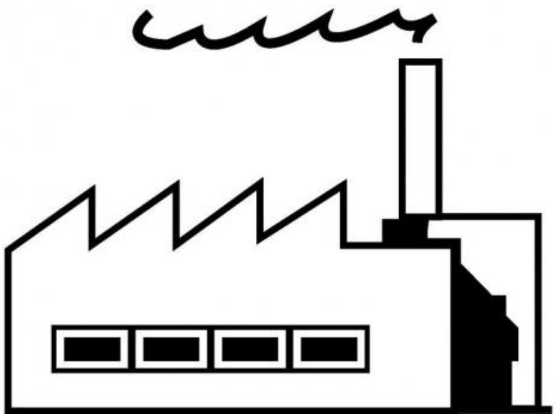
- Async/await
  - Syntactic sugar on top of Tasks
  - Provides out of the box synchronization
  - private Task<int> Foo() { … }

    private async Task<int> UseFoo() {

        int result = await Foo();

        return result;

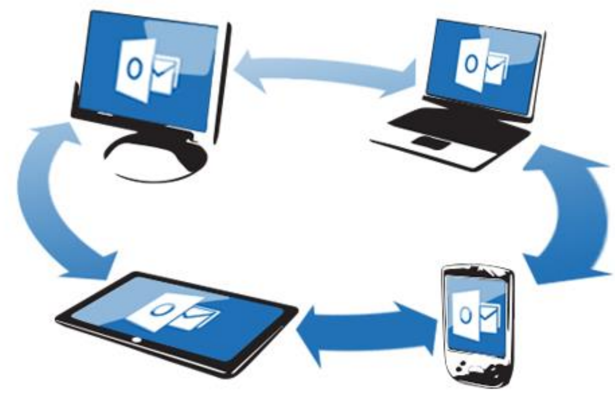
    }



Synchronization

Why?

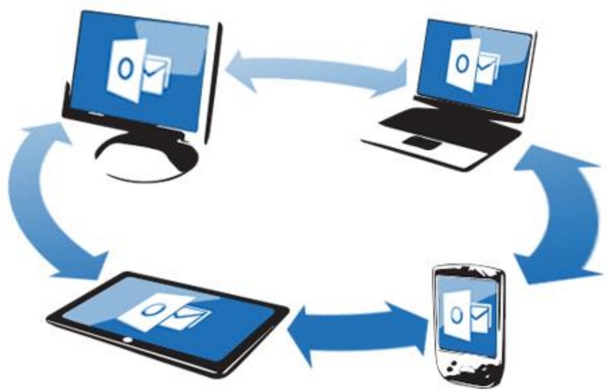
- Avoid deadlocks
  - Occurs when 2 threads try to lock their own resources
- Avoid out-of-orderexecution
  - Occurs when code gets optimized and is invoked in multithreaded environment
- Avoid data corruption
  - Occurs when one thread is writing data, and other is trying to read it



Synchronization

What options do we have

- Interlocked methods
  - Add, Exchange, CompareExchange, … - no blocking, ensure operation is atomic
- Critical section
  - Great for in-process communication, acquiring is lightning fast* (lock (_obj) { … })
- WaitHandles (Manual|Auto ResetEvent, Semaphore, Mutex)
  - Huge performance impact, but great control capabilities
- UI synchronization
  - UI applications have STA model, all access operations need to be synchronized - SynchronizationContext



Critical section

Incrementation problem

```
class Program {
    private static int index;
    static void Main(string[] args) {
        index = 0;
        ThreadPool.QueueUserWorkItem(IncrementLoop);
        ThreadPool.QueueUserWorkItem(IncrementLoop);
        ThreadPool.QueueUserWorkItem(IncrementLoop);
        Console.ReadLine();
    }
    private static void IncrementLoop(Object state) {
        for (int i = 0; i < 20; i++)
            Console.WriteLine(index++);
        }
    }
```

Critical section

- Piece of code that accesses a shared resource
- Cannot be concurrently accessed by more than one thread
- Requests for access from threads are queued
- Requires synchronization mechanism to ensure exclusive access – e.g. semaphore

Synchronization mechanisms

Monitor vs Semaphore

- About `Semaphore`:
  - Limits the number of threads that can access a resource or pool of resources concurrently by using `WaitOne` and `Release` methods.
  - There is no guaranteed order, such as FIFO or LIFO, in which blocked threads enter the semaphore.
  - A thread can enter semaphore multiple times by calling `WaitOne`.
  - Does not enforce thread identity on calls to `WaitOne` or `Release` – it is programmer's responsibility to ensure that threads do not release semaphore too many times.
  - There are two types of semaphores:
    - Local semaphore – exists within process
    - Named system semaphore – visible through the OS, can be used to synchronize the activity of processes

- About `Monitor`:
  - Simple synchronization mechanism.
  - Marks a statement block as a critical section.
  - Ensures that one thread does not enter a critical section while another thread is in critical section.
  - Queues threads that try to enter locked code, they will wait until critical section is released.

Synchronization mechanisms

Monitor vs Semaphore

Semaphore

```
private static Semaphore pool;
private static int sum = 0;

private static void Main (int value) {
    pool = new Semaphore(0, 3);
    for (int i = 0; i < 25; i++)
        ThreadPool.QueueUserWorkItem(Sum, i);
}
private void Sum(object state) {
    int value = (int)state;

    pool.WaitOne();
    sum += value ;
    pool.Release();
}
```

Monitor

```
private static object syncLock;
private static int sum = 0;

private static void Main (int value) {

    for (int i = 0; i < 25; i++)
        ThreadPool.QueueUserWorkItem(Sum, i);
}
private void Sum(object state) {
    int value = (int)state;

    Monitor.Enter(syncLock);
    sum += value ;
    Monitor.Exit(syncLock);
}
```

Synchronization mechanisms
Lock & Interlocked

- About `lock`:
  - Is a nice looking wrapper for Monitor, calling Enter at the start and Exit a the end of block of code.
  - Lock can be acquired on a variable of any reference type.

- About `Interlocked`:
  - Provides atomic operations for variables that are shared by multiple threads.
  - Protects against errors that can occur when scheduler switches context while a thread is updating a schared variable, or when multiple threads are executing concurrently on separate processors.
  - Methods of this class do not throw exceptions.

Synchronization mechanisms

Lock & Interlocked

Lock

```
class A {
    private static readonly object syncLock =
        new object();
    private int a = 0;

    public int Add (int value) {
        lock(syncLock) {
            a += value;

            return a;
        }
    }
}
```

Interlocked

```
class A {
    private int a = 0;

    public int Add (int value) {
        return Interlocked.Add(ref a, value);
    }
}
```

Synchronization mechanisms
Mutex

- Synchronization primitive.
- Can be used for interprocess synchronization.
- Grants exclusive access to shared resource to only one thread.
- One thread can request ownership multiple times using `WaitOne` without blocking its execution. However must call `ReleaseMutex` the same number of times to release ownership.
- Enforce thread identity, so a mutex can be released only by the thread that acquired it, as oopposed to Semaphore.
- Like semaphore, there are two types of mutexes:
  - Local (unnamed) mutex – exists only within a process
  - Named system mutex – visible through the OS and can be used to synchronize the activities of processes

Synchronization mechanisms

Mutex

```
private static Mutex mutex;
private static int sum = 0;

private static void Main (int value) {
    pool = new Mutex();

    for (int i = 0; i < 25; i++)
        ThreadPool.QueueUserWorkItem(Sum, i);
}
private void Sum(object state) {
    int value = (int)state;

    mutex.WaitOne();
    sum += value ;
    mutex.ReleaseMutex();
}
```

Synchronization mechanisms
AutoResetEvent & Task

- About `AutoResetEvent`:
  - Notifies a waiting thread that an event has occured, which allows threads to communicate witch each other by signaling.
  - Thread waits for signal by calling non-blocking loop – `WaitOne`.
  - Calling `Set` signals waiting thread.
  - Can be used with static `WaitAll` and `WaitAny` methods.

- Synchronization using `Task`:
  - `WaitAll` is similar to `AutoResetEvent.WaitAll` – it is non-blocking loop waiting for all tasks to complete execution.
  - `WaitAny` is similar to `AutoResetEvent.WaitAny` – it is non-blocking loop waiting for any of the tasks to complete execution.
  - `WhenAll` creates a new task that will complete when all of the provided tasks have completed.
  - `WhenAny` creates a new task that will complete when any of the provided tasks have completed.

Multithreading issues

Deadlock

Coffman conditions

- Mutual exclusion – two or more resources must not be shareable, and only one process can use the resource at the time.

- Hold and wait or resource holding – process is currently holding at least one resource and requesting addidtional resources which are being held by other processes.

- No preemption – resources must be released by holding process voluntarily, they must not be deallocated by operating system.

- Circular wait – process must be waiting for a resource which is being held by another process, which in turn is waiting for first process to release the resource.

Multithreading issues
Deadlock

```
private void Transfer(Account accountA, Account accountB, double amount)
{
    lock (accountA)
    {
        lock (accountB)
        {
            if (accountA.Balance < amount)
                throw new Exception("Insufficient funds.");

            accountA.Balance -= amount;
            accountA.Balance += amount;
        }
    }
}
```

Multithreading issues

Deadlock - solution

```
private void Transfer(Account accountA, Account accountB, double amount)
{
    if (accountA.GetHashCode() > accountB.GetHashCode()
        lock (accountA)
            lock (accountB)
            {
                // logic
            }
    else
        lock (accountB)
            lock (accountA)
            {
                // logic
            }
}
```

Multithreading issues

Race condition

```
class A {
    int result = 0;
    void Work1() { result = 1; }
    void Work2() { result = 2; }
    void Work3() { result = 3; }

    static void Main(string[] args) {
        A a = new A();
        Thread worker1 = new Thread(a.Work1);
        Thread worker2 = new Thread(a.Work2);
        Thread worker3 = new Thread(a.Work3);
        worker1.Start();
        worker2.Start();
        worker3.Start();
        Console.WriteLine(a.result);
        Console.Read();
    }
}
```

Result of the code might be 1, 2, 3 or even 0.

Good practices

- Prefer using methods of Interlocked class for simple state changes, instead of using lock keyword.
- Use producer/consumer model.
- Do not use types as lock objects.
- Use caution when locking on instances, e.g. lock(this).
- Do ensure that a thread that has entered a monitor always leaves that monitor even if an exception occurs while thread is in the monitor, lock statement provides this behavior automatically, employing a finally block to ensure that Monitor.Exit() is called.

References

- http://msdn.microsoft.com/en-us/magazine/cc164040.aspx
- http://stackoverflow.com/questions/154551/volatile-vs-interlocked-vs-lock
- http://www.pzielinski.com/?p=1692
- http://www.c-sharpcorner.com/UploadFile/1d42da/race-conditions-in-threading-C-Sharp/
- https://www.wintellect.com/blogs/jeffreyr
- http://www.softwareinteractions.com/blog/2009/12/8/intraprocess-message-queue.html

Multithreading - synchronization

Q & A