Locks are tricky

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

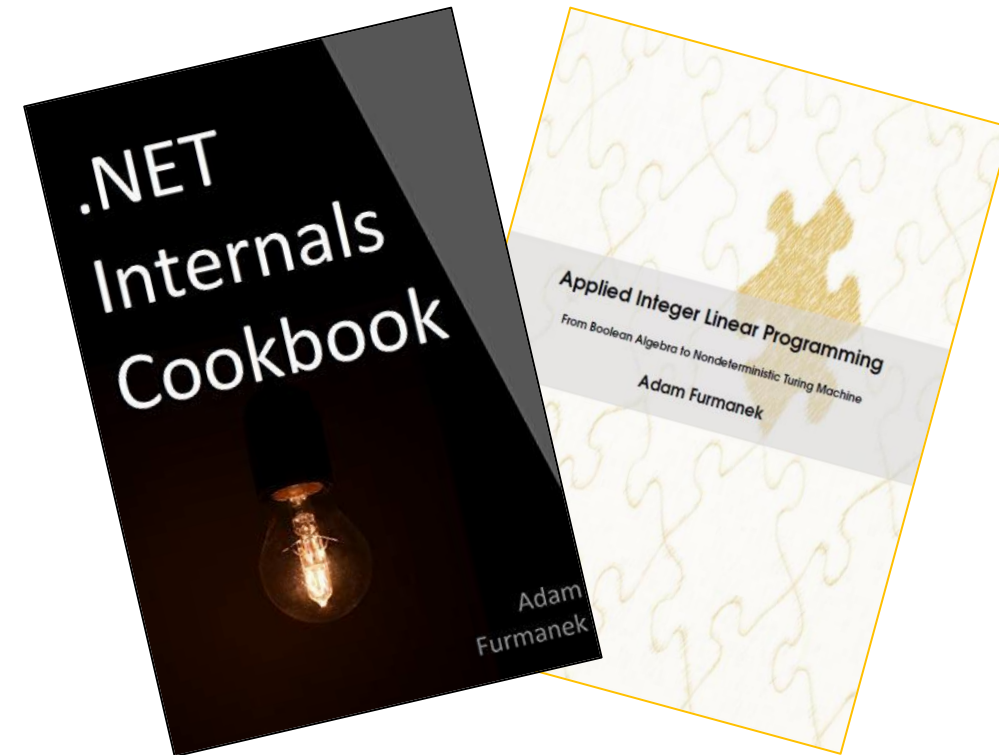# About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of *Applied Integer Linear Programming* and *.NET Internals Cookbook*.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

 furmanekadam

Agenda

General concepts.

Real life scenarios.

Algorithms.

Custom implementations.

Summary.

General concepts

Concurrency vs Parallelism

CONCURRENCY

Ability to decompose an application into multiple parts.

Each part can be executed out-of-order without affecting the final outcome.

Think of controling an access to a resource.

Concurrent execution is possible on a single-core machine.

PARALLELISM

Ability to execute multiple things at once.

Requires multiple execution units.

Examples

	Parallel	Not Parallel
Concurrent	True concurrent code with multiple cores	Single core with context switching
Not Concurrent	Bit level parallelism	Batch processing

Shared memory vs Message passing

SHARED MEMORY

When multiple execution units access the same physical memory.

Prone to word tearing, false sharing.

Requires critical sections or *test-and-set* operations.

Cannot be used between machines.

Can be faster.

MESSAGE PASSING

When multiple execution units send messages to each other.

Requires some way means of communication – like storage, queue etc.

Can be used between machines.

Typically slower.

Shared memory issues

Lost write
- ◦ variable++

Word tearing
- ◦ When we cannot access a single element and need to write multiple of them at once (like with bitset)

Atomic write
- ◦ When we cannot write a particular value at once (like with 64-bit variables on a 32-bit CPU)

How can we deal with this?

Critical section

Code which can be executed by one execution unit at a time.

Cannot be parallelized.

Should finish in a finite time.

Should be as small as possible.

What do we do if a process dies while executing a critical section?

LOCKS ARE TRICKY - ADAM FURMANEK

Mutex

Name comes from Mutual Exclusion.

Can be implemented with disabling interrupts in a single-processor system.

Requires some atomic operation a multi-processor system like test-and-set or compare-and-swap.

Can be implemented in the kernel or user mode.

Typically provded by the operating system.

Can be locked and unlocked only by the owner.

Typcially can be taken recursively.

Mutex

WinAPI:
◦ CreateMutex
◦ WaitForSingleObject
◦ ReleaseMutex

Called Mutant internally.

POSIX Threads (pthread):
◦ pthread_mutex_init
◦ pthread_mutex_lock
◦ pthread_mutex_unlock

Mostly implemented by Native Posix Thread Library (NPTL).

Other implementations exist like musl.

.NET: Mutex

Counting Semaphore

Counts execution units allowed to enter the section.

Two operations:
◦ V (increment) – comes from Dutch verhogen
◦ P (decrement) – comes from Dutch proberen/passeren/pakken

Can be unlocked by any thread, not only the „owning" ones.

What's the difference between a binary semaphore (counting up to one) and a mutex?

Mutex vs Binary semaphore

MUTEX

Can be unlocked by the owner only.

Can protect the owner from termination.

Supports priority inversion as we know who owns the mutex.

Allows for recursion.

Can notify when it is abandoned.

BINARY SEMAPHORE

Can be unlocked by anyone.

Doesn't protect from termination as the owner is not tracked.

Doesn't support priority inversion as the owner is not tracked.

Doesn't support recursion.

Doesn't notify when it is abandoned.

Counting Semaphore

WinAPI:
- CreateSemaphore
- WaitForSingleObject
- ReleaseSemaphore

POSIX Threads (pthread):
- sem_init
- sem_wait
- sem_post

.NET:
- Semaphore
- SemaphoreSlim

Sleep vs Spin wait

SLEEP

Entering the sleep mode is expensive for the operating system
- Typically requires switching to a kernel mode
- Requires some kernel structures

Entering the slep mode is expensive for the application
- Causes a context switch so consumes time

Doesn't block the thread.

Cannot be done in all circumstances.

SPIN WAIT

Can be done in all circumstances.

Blocks the thread so it's very CPU consuming.

Is often used initially when we expect the critical section to be released soon.

Very important in kernel programming.

Monitor with condition variable

```
for(;;) {
    lock (sync) {
        if (condition) {
            // Do work
            break;
        } else {
            // Loop again - but how?
        }
    }
}
```

Mutual exclusion is not enough. Often we need to wait until some condition holds true.

Question remains: how do we wait for the condition to hold?

If we don't sleep then we hog the CPU.

If we do sleep then it's hard to find the right delay.

Condition variable is a set of threads waiting for the condition to hold.

Monitor is a thread-safe construct using mutex and condition variables to protect its methods.

Condition variable

WinAPI:
- ◦ InitializeConditionVariable
- ◦ SleepConditionVariableCS
- ◦ WakeConditionVariable

POSIX Threads (pthread):
- ◦ pthread_cond_init
- ◦ pthread_cond_wait
- ◦ pthread_cond_signal

.NET:
- ◦ Monitor.Pulse + Monitor.Wait
- ◦ ReaderWriterLock

Spurious wake-up

```
lock (sync) {
    // This is WRONG - we need to use while
    if (!condition) {
        Monitor.Wait(sync);
    }
}
```

We may be woken up by *Monitor.PulseAll*.

We may be woken up by the the driver or APC.

We always need to use loop.

Local vs global

LOCAL LOCKS

Are held within one process only.

Are typically faster.

Can be implemented in user-mode.

GLOBAL LOCKS

Are held between processes.

Require a support from the operating system.

Need to be identified somehow – typically by name. Watch for collisions!

Are slower due to switching to the kernel mode.

Truly global locks

We need inter-machine locks.

They are typically implemented as leases with a timeout.

They must be discoverable – typically by name or some resource.

They require network communication so are much slower.

Issues with locks

What if a thread dies? Are we sure the data is safe?

What if it used a semaphore? How do we reset the counter? How do we get notification?

What if it was an inter-machine lock? What about timeout?

What about priority inversion?

How can we see who holds the lock?

How can we take the lock forcefully?

<u>Always acquire a lock with a timeout!</u>

Compare-and-swap

Atomically compares the contents of memory with a given value, and modifies the memory if allowed.

```
int CompareAndSwap(ref int memory, int baseValue, int newValue) {
    lock() {
        int oldValue = memory;
        if (oldValue == baseValue) {
            memory = newValue;
        }

        return oldValue;
    }
}
```

x86:
◦ CMPXCHG

ARM:
◦ CAS

C++:
◦ atomic_compare_exchange_weak

.NET:
◦ Interlocked.CompareExchange

JVM:
◦ AtomicInteger.compareAndSet

Real life scenarios

Finalizers

Finalizers may be executed when exiting
- This is implementation-dependent and unreliable

Finalizers may be executed in paralel. They may be locked or not.

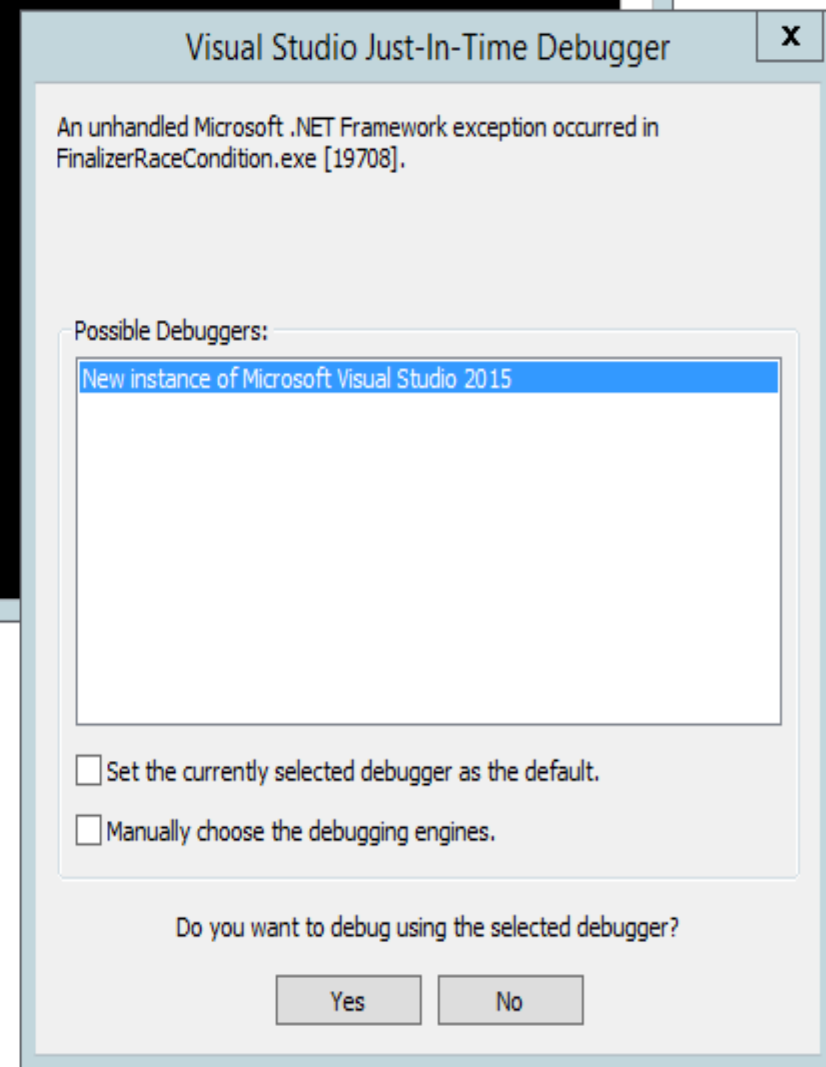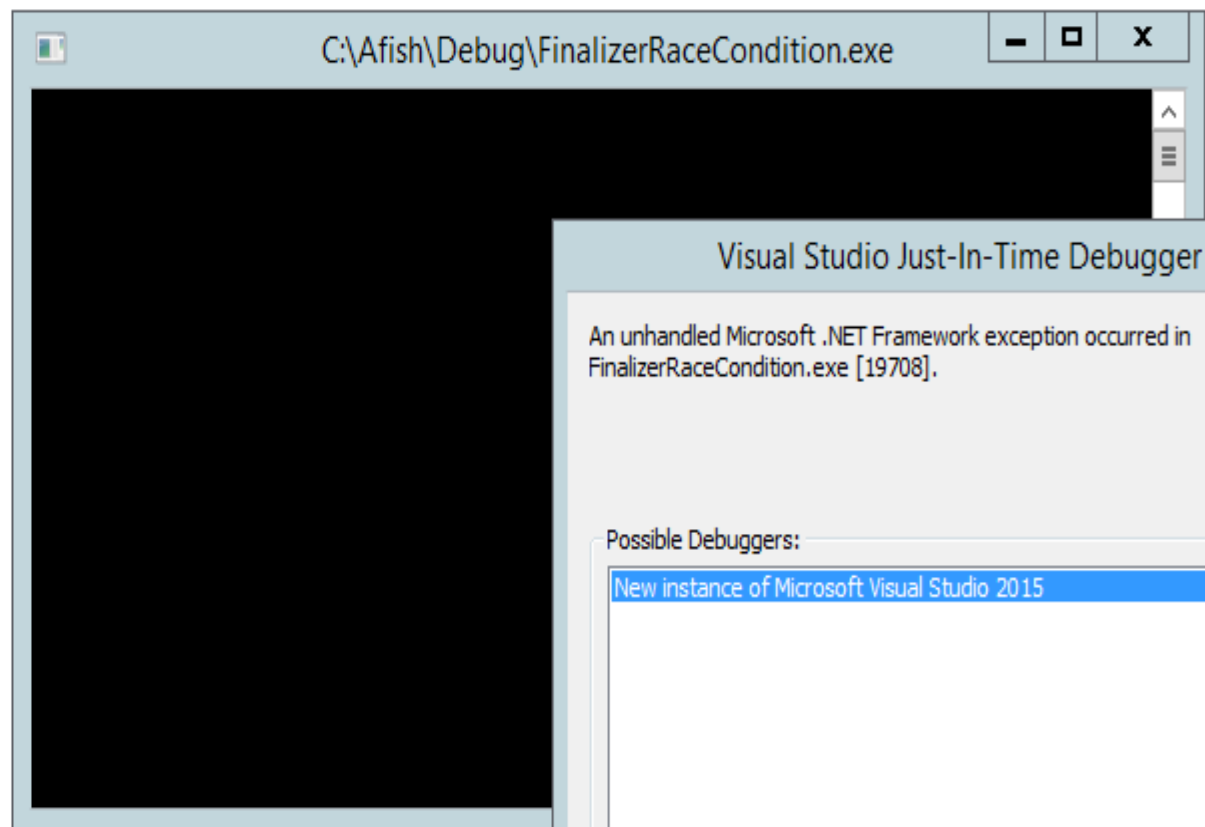Finalizing thread cannot run indefinitely. It will be terminated if it takes too long.

Finalizer may be called when the variable is still being used.

```csharp
{
    var content = @"abc";

    if (args.Length > 0)
    {
        TestFile(content);
    }
    else
    {
        while (true)
        {
            Process.Start("FinalizerRaceCondition.exe", "slave").WaitForExit();
            if (new FileInfo("0").Length - 3 > 0)
            {
                Debugger.Launch();
                Debugger.Break();
                Thread.Sleep(100000000);
            }
            Thread.Sleep(1);
        }
    }
}

2 references
static bool TestFile(string content)
{
    var thread1 = new Thread(() =>
    {
        Write(content);
        Thread.Yield();
    });
    var thread2 = new Thread(() => Environment.Exit(1));
    thread1.Start();
    thread2.Start();

    return true;
}
```

C:\Afish\Debug\FinalizerRaceCondition.exe

Visual Studio Just-In-Time Debugger

An unhandled Microsoft .NET Framework exception occurred in FinalizerRaceCondition.exe [19708].

Possible Debuggers:

New instance of Microsoft Visual Studio 2015

☐ Set the currently selected debugger as the default.

☐ Manually choose the debugging engines.

Do you want to debug using the selected debugger?

Yes   No
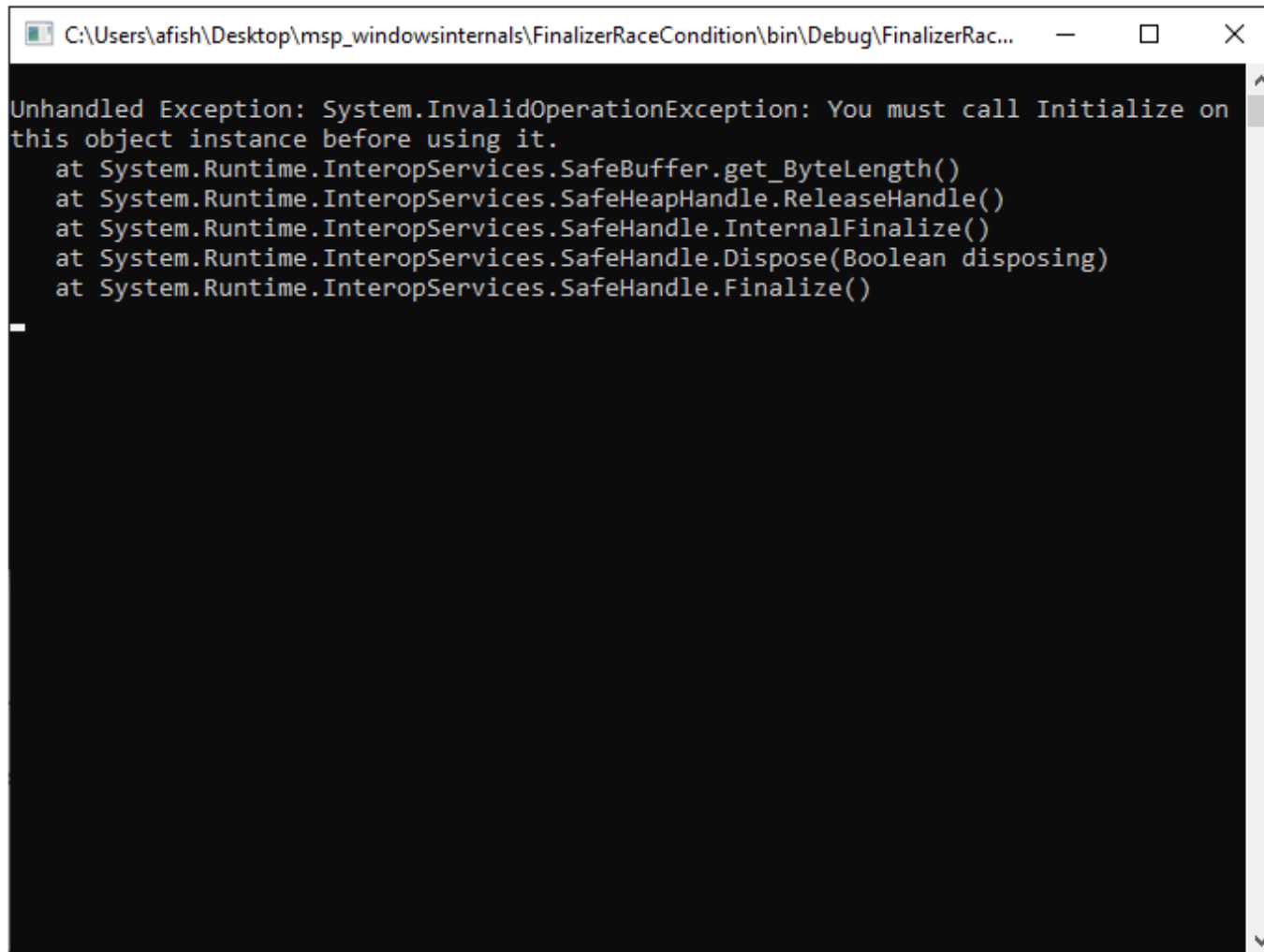
```
{
    var content = @"abc";

    if (args.Length > 0)
    {
        TestFile(content);
    }
    else
    {
        while (true)
        {
            Process.Start("FinalizerRaceCondition.exe", "slave").WaitForExit();
            if (new FileInfo("0").Length - 3 > 0)
            {
                Debugger.Launch();
                Debugger.Break();
                Thread.Sleep(100000000);
            }
            Thread.Sleep(1);
        }
    }
}

2 references
static bool TestFile(string content)
{
    var thread1 = new Thread(() =>
    {
        Write(content);
        Thread.Yield();
    });
    var thread2 = new Thread(() => Environment.Exit(1));
    thread1.Start();
    thread2.Start();

    return true;
}
```

C:\Users\afish\Desktop\msp_windowsinternals\FinalizerRaceCondition\bin\Debug\FinalizerRac...   —   □   ✕

```
Unhandled Exception: System.InvalidOperationException: You must call Initialize on
this object instance before using it.
    at System.Runtime.InteropServices.SafeBuffer.get_ByteLength()
    at System.Runtime.InteropServices.SafeHeapHandle.ReleaseHandle()
    at System.Runtime.InteropServices.SafeHandle.InternalFinalize()
    at System.Runtime.InteropServices.SafeHandle.Dispose(Boolean disposing)
    at System.Runtime.InteropServices.SafeHandle.Finalize()
```

```csharp
if (args.Length > 0)
{
    HijackMethod(
        typeof(FileStream).GetMethod("WriteFileNative", BindingFlags.NonPublic | BindingFlags.Instance),
        typeof(Program).GetMethod(nameof(Hijacked), BindingFlags.Instance | BindingFlags.Public)
    );
    TestFile(content);
}
else
{
    while (true)
    {
        var process = Process.Start(new ProcessStartInfo
        {
            Arguments = "slave",
            RedirectStandardOutput = true,
            FileName = "FinalizerRaceCondition.exe",
            UseShellExecute = false,
            CreateNoWindow = false,
            WindowStyle = ProcessWindowStyle.Normal
        });
        process.WaitForExit();
        var output = process.StandardOutput.ReadToEnd();

        if (new FileInfo("0").Length - 3 > 0 || !string.IsNullOrEmpty(output))
        {
            Console.WriteLine(output);
            Debugger.Launch();
            Debugger.Break();
            Thread.Sleep(100000000);
        }
        Thread.Sleep(1);
    }
}
```

```
1 reference
public unsafe int Hijacked(SafeFileHandle handle, byte[] bytes, int offset, int co
{
    if (stacktrace != null)
    {
        Console.WriteLine("Current stacktrace");
        Console.WriteLine(Environment.StackTrace);
        Console.WriteLine("Other stacktrace");
        Console.WriteLine(stacktrace);
    }

    stacktrace = Environment.StackTrace;
    Thread.Yield();

    hr = 0;
    return 3;
}
```

C:\Windows\system32\cmd.exe

```
Current stacktrace
   at System.Environment.GetStackTrace(Exception e, Boolean needFileInfo)
   at System.Environment.get_StackTrace()
   at FinalizerRaceCondition.Program.Hijacked(SafeFileHandle handle, Byte[] bytes, Int32 offset, Int32 count, NativeOverlapped* ove
rlapped, Int32& hr)
   at System.IO.FileStream.WriteCore(Byte[] buffer, Int32 offset, Int32 count)
   at System.IO.FileStream.FlushWrite(Boolean calledFromFinalizer)
   at System.IO.FileStream.Dispose(Boolean disposing)
   at System.IO.FileStream.Finalize()
Other stacktrace
   at System.Environment.GetStackTrace(Exception e, Boolean needFileInfo)
   at System.Environment.get_StackTrace()
   at FinalizerRaceCondition.Program.Hijacked(SafeFileHandle handle, Byte[] bytes, Int32 offset, Int32 count, NativeOverlapped* ove
rlapped, Int32& hr) in C:\Users\afish\Desktop\msp_windowsinternals\FinalizerRaceCondition\Program.cs:line 122
   at System.IO.FileStream.WriteCore(Byte[] buffer, Int32 offset, Int32 count)
   at System.IO.FileStream.FlushInternalBuffer()
   at System.IO.FileStream.Flush(Boolean flushToDisk)
   at System.IO.FileStream.Flush()
   at System.IO.StreamWriter.Flush(Boolean flushStream, Boolean flushEncoder)
   at System.IO.StreamWriter.Dispose(Boolean disposing)
   at System.IO.TextWriter.Dispose()
   at FinalizerRaceCondition.Program.Write(String content) in C:\Users\afish\Desktop\msp_windowsinternals\FinalizerRaceCondition\Pr
ogram.cs:line 68
   at FinalizerRaceCondition.Program.<>c__DisplayClass3_0.<TestFile>b__0() in C:\Users\afish\Desktop\msp_windowsinternals\Finalizer
RaceCondition\Program.cs:line 50
   at System.Threading.ThreadHelper.ThreadStart_Context(Object state)
   at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object state, Bool
ean preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback, Object state, Boolean pres
erveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
```

Common wrong pattern: Acquire outside of *try*

First, take the lock.

Then, enter the *try* block.

Release the lock in *finally*.

But what if there was something wrong between acquiring the lock and entering the *try* block?

```
AcquireLock();
try
{
    // ...
}
finally
{
    ReleaseLock();
}
```

Common wrong pattern: double-checked-lock

Invalid, because *helper* may be already set, but *new Helper()* may be still running (due to memory reordering).

Use:
- *VarHandle* (JVM)
- *Lazy* (.NET)
- *call_once* (C++)

```java
class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            synchronized (this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
}
```

Heap lock

Multiple *malloc* implementations lock the heap when allocating memory.

So there can be a lock taken when calling *new object()*.

If a thread dies for any reason – the lock is abandoned, and the heap may be broken.

Don't kill threads! Don't suspend threads!

DLLMain lock

When a new module is loaded into the process, a lock is taken.

DLLMain function is effectively executed in a critical section.

This often leads to deadlocks. Especially when doing a dll-injection.

Lock on executables

Windows locks executables (.exe and .dll). You can't erase them from the drive while they're running.

Windows cannot force unloading a module that is still used.

How can you install updates without rebooting the system?

Why do OS functions start with *move edi, edi* (which is basically a no-op)?

fork without *exec*

When you fork a proces, only the calling thread runs in the child.

If some other thread held a lock, the lock remains locked forever.

fork without *exec* is popular for multiprocessing libraries.

Always use *fork + exec* in your code.

Never call anything between *fork* and *exec.* Even *malloc* can cause a deadlock.

APC

Asynchronous Procedure Call is a mechanism to deliver a message to a thread.

When taking a lock (like mutex), we need to disable APC. Otherwise, we risk a deadlock.

However, different locking primitives disable different APC types.

Asynchronous I/O may use APC for delivering a result. But it may also use I/O completion port. Make sure you wait on a correct primitve.

Message pump is not an alertable state, and doesn't proces APCs.

https://www.osr.com/nt-insider/2015-issue3/the-state-of-synchronization/
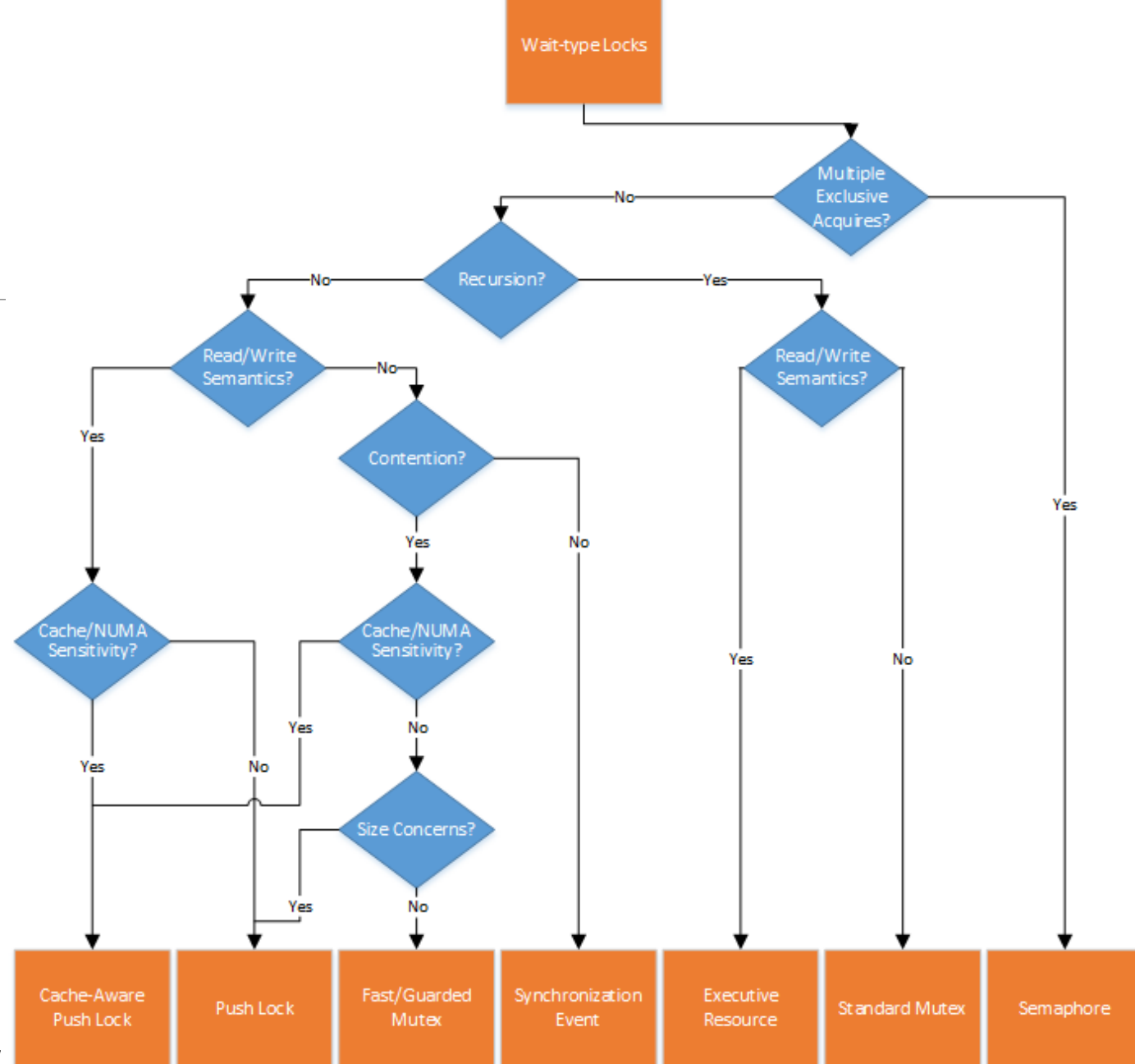
Drivers

Windows cannot risk breaking a physical device.

If there is a buggy driver, then Windows won't try killing it forcefully.

If a process waits for a buggy driver then the process cannot be killed.

You need to restart your computer to solve the issue.

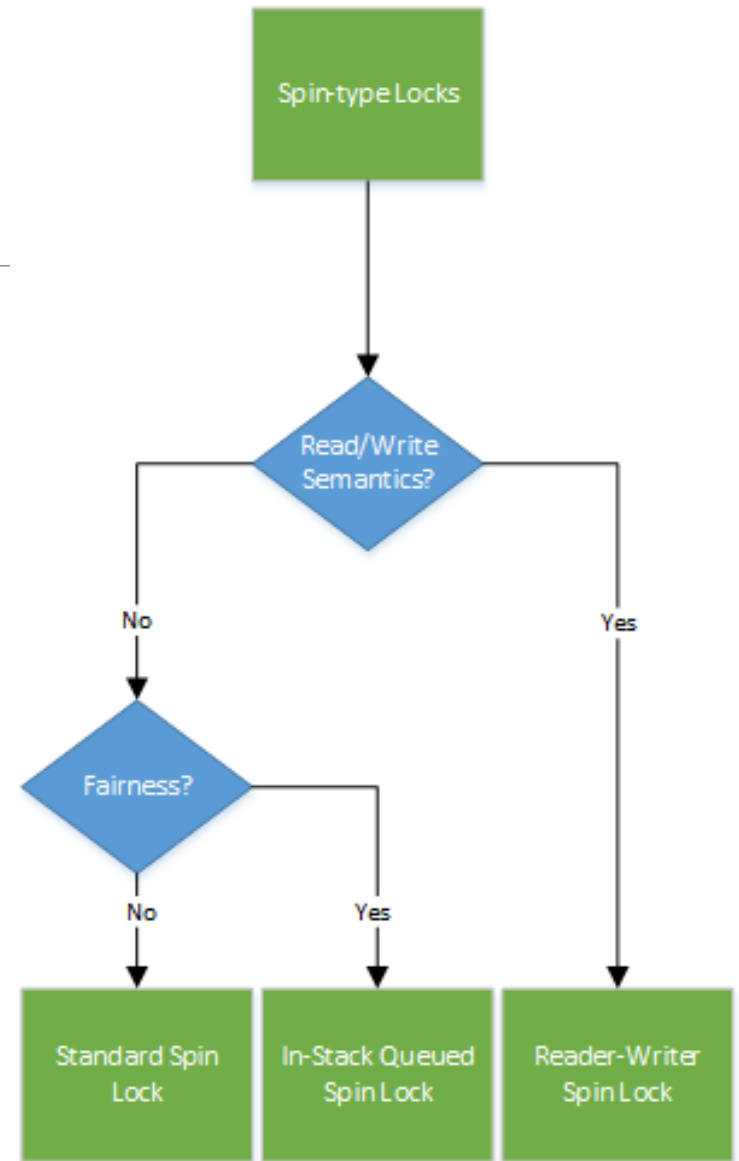Or use ProcessHacker, but it can crash the machine.

Kernel-mode locks

We may be forced to use spin-based locks depending on the IRQL level.

Spin-based locks do not support recursion.

What about reader/writer semantics?

Fairness? Contention?

https://www.osr.com/nt-insider/2015-issue3/the-state-of-synchronization/

Interrupt Request Level (IRQL)

When there is an interrupt, the operating system maps it to the correct priority level.

Higher levels preempt lower ones.

Interesting levels:

◦ 0: PASSIVE_LEVEL – „regular" one, all Windows components are available. All synchronization primitives work

◦ 2: DISPATCH_LEVEL – for DPC and some drivers, not all components are available. Thread scheduler runs here. Only spin locks can be used

◦ 12: SYNCH_LEVEL – for synchronizing data between CPUs

:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

25% complete

For more information about this issue and possible fixes, visit

http://windows.com/stopcode

If you call a support person, give them this info:
Stop code IRQL_NOT_LESS_OR_EQUAL

MS SQL

Low-level locks used to maintain the memory consistency.
Protect b-trees, heaps, pages.
Buffer latches, non-buffer latches, IO latches.
Cannot be controlled directly by the user.

Used to maintain the transaction consistency.
Can be controlled directly by the user with query hints.
Affect results of queries – dirty reads, phantoms, snapshots.
What is your default isolation level?

Cloud leases

Azure Blob Lease.

Azure Queue Lease.

Azure Service Bus Lock.

Zookeeper locks.

Similar idea: network timeouts.

What happens if you don't refresh the lease?

Logical transaction locks

Ever tried buying a ticket just to see that you need to confirm the transaction in 15 minutes?

How do we scale them?

How do we make sure that we don't overbook?

Is overbooking a problem?

What to do if one ticket can be sold by multiple vendors?

Algorithms

Dekker's Algorithm

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer


wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0    // or 1
```

First known algorithm.

Works for 2 processes only.

Uses busy-wait.

Requires memory barriers or code reordering disabled.

```
p0:
    wants_to_enter[0] ← true
    while wants_to_enter[1] {
        if turn ≠ 0 {
            wants_to_enter[0] ← false
            while turn ≠ 0 {
                // busy wait
            }
            wants_to_enter[0] ← true
        }
    }

    // critical section
    ...
    turn ← 1
    wants_to_enter[0] ← false
    // remainder section
```

```
p1:
    wants_to_enter[1] ← true
    while wants_to_enter[0] {
        if turn ≠ 1 {
            wants_to_enter[1] ← false
            while turn ≠ 1 {
                // busy wait
            }
            wants_to_enter[1] ← true
        }
    }

    // critical section
    ...
    turn ← 0
    wants_to_enter[1] ← false
    // remainder section
```

Peterson's algorithm

Works for N processes.

Requires memory barriers or code reordering disabled.

There are other algorithms: Szymański's, Lamport's bakery, Eisenberg & McGuire etc. They have favorable properties like fairness, linear wait etc.

```
level : array of N integers
last_to_enter : array of N - 1 integers
```

```
i ← ProcessNo
for ℓ from 0 to N - 1 exclusive
    level[i] ← ℓ
    last_to_enter[ℓ] ← i
    while last_to_enter[ℓ] = i and there exists k ≠ i, such that level[k] ≥ ℓ
        wait
level[i] ←- 1
```

How to reason about locks?

Multiple models:

- Petri nets
- Calculus of communicating systems (CCS)
- Communicating sequential processes (CSP)
- Π-calculus
- Tuple spaces
- Parallel random-access-machine (PRAM)
- Actor model

Temporal logic of actions (TLA+):

- Formal specification language used to design concurrent and distributed systems
- IDE and PlusCal language

https://github.com/exactly-once/model-checking/blob/15b007f5ce403f1e4f8bd0a52bf73abee9085570/exactly_once_none_atomic.tla

```
variables
    inputQueue = [id : MessageIds, dupId : DupIds],
    store = [history |-> <<>>, ver |-> 0, tx |-> NULL],
    outbox = [r \in MessageIds |-> NULL],
    outboxStagging = [t \in TxIdx |-> NULL],
    output = { },
    processed = { }

define
    InputMessages == [id : MessageIds, dupId : DupIds]
    OutputMessages == [msgId : MessageIds, ver : VersionIds]
    TypeInvariant ==
        /\ inputQueue \in SUBSET InputMessages
        /\ output \in SUBSET OutputMessages
        /\ processed \in SUBSET InputMessages
        /\ store.ver \in VersionIds
        /\ store.tx \in (TxIdx \union {NULL})
        /\ Range(store.history) \in SUBSET [ver : VersionIds, msgId : MessageIds ]
        /\ Range(outbox) \in SUBSET (OutputMessages \union {NULL})
        /\ Range(outboxStagging) \in SUBSET (OutputMessages \union {NULL})

    AtMostOneStateChange ==
        \A id \in MessageIds : Cardinality(WithId(Range(store.history),id)) <= 1

    AtMostOneOutputMsg ==
        \A id \in MessageIds : Cardinality(WithId(output, id)) <= 1

    ConsistentStateAndOutput ==
        LET InState(id)  == CHOOSE x \in WithId(Range(store.history), id) : TRUE
            InOutput(id) == CHOOSE x \in WithId(output, id) : TRUE
        IN \A m \in processed: InState(m.id).ver = InOutput(m.id).ver

    Safety == AtMostOneStateChange /\ AtMostOneOutputMsg /\ ConsistentStateAndOutput

    Termination == <>(/\ \A self \in Processes: pc[self] = "LockInMsg"
                      /\ IsEmpty(inputQueue))
end define;
```

Caches and memory models

Starting with IBM 801 in 1976 CPUs have caches.

They can be of multiple levels (L1, L2, L3) or purpose (data, instructions).

Each read from or write to a main memory goes to cache and only then accesses the RAM.

Typically, each core has its own L1 cache.

We need to take care when writing to a variable used by multiple cores concurrently – volatile, memory barriers, cache flush.

CPU is allowed to reorder instructions – this is called a **memory model**.

Compiler is allowed to do the same.

We can't rely on a memory being read/written in the same order as we see in the source code.

We need to take care when using variables outside of any synchronization primitives, barriers, locks, etc.

MESI (Illinois) protocol

Protocol for maintaing cache coherence.

4 states:
- ◦ Modified (M)
- ◦ Exclusive (E)
- ◦ Shared (S)
- ◦ Invalid (I)

When CPU1 read data which is modified in CPU2's cache line then we need to synchronize this situation. Data is flushed to the main memory (that's a great simplification but gives an idea how it works).

volatile — this makes sure that the variable is not cached, but read directly from the memory instead.

False sharing

Cache consists of memory lines, for instance 64-byte long.

When we try accessing two elements on the same cache line then we share the memory.

```csharp
using System;
using System.Diagnostics;
using System.Threading;

public class Program
{
    public static void Main()
    {
        var stopwatch = Stopwatch.StartNew();

        int iterations = 100000000;
        var array = new int[17];

        Thread t1 = new Thread(() => Run(array, 0, iterations));
        // Change to 16 instead of 1
        Thread t2 = new Thread(() => Run(array, 1, iterations));

        t1.Start();
        t2.Start();
        t1.Join();
        t2.Join();

        stopwatch.Stop();
        Console.WriteLine(stopwatch.ElapsedMilliseconds);
    }

    static void Run(int[] array, int which, int iterations){
        for(int i=0;i<iterations;++i){
            array[which]++;
        }
    }
}
```
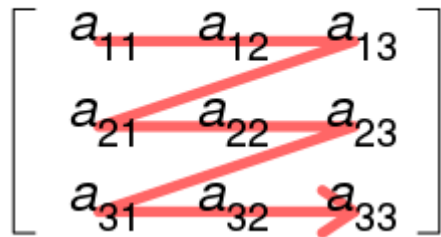
Loop interchange

```
for i from 0 to 10
  for j from 0 to 20
    a[i,j] = i + j
```

Row-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Both code snippets do the same.

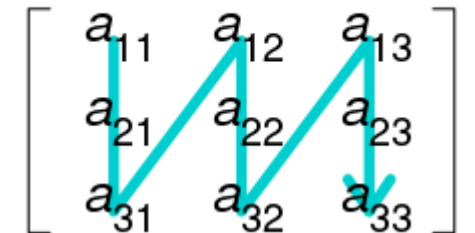They access elements in different order.

Which one is better – depends on your language!

Row-major is better in C. Column-major is better in FORTRAN.

It's all about caching.

```
for j from 0 to 20
  for i from 0 to 10
    a[i,j] = i + j
```

Column-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Matrix multiplication: ijk vs ikj

The order matters when we multiply matrixes.

It's typically assumed that *ijk* order (snippet on the left) is slower.

ikj order is faster because the compiler can optimize the code.

It's all about caching.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        for (int j = 0; j < n; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        int temp = A[i][k];
        for (int j = 0; j < n; j++) {
            C[i][j] += temp * B[k][j];
        }
    }
}
```

Producer-consumer

BlockingCollection in .NET.

Task.Run in .NET*.*

ExecutorService in Java.

Scatter + Gather in MPI.

Barrier in MPI.

Generally – avoid building those primitives by hand.

Other solutions

OpenMP:
- A library for multi-platform shared memory programming

```
int main(void)
{
    #pragma omp parallel      ⬅
    printf("Hello, world.\n");
    return 0;
}
```

MPI:
- Standard for passing messages
- MPI_Send, MPI_Recv, MPI_Scatter, MPI_Barrier
- MPI.NET

Actors:
- Actor has an interface and messaging capabilities
- Akka.NET

Agents:
- Can observe the environment
- Agents.NET

COM:
- Binary interface for software components
- Supports multiple apartments for threading (STA, MTA, NA)

Lock free code

Avoid locks and use *test-and-set, compare-and-swap*, or *interlocked* operations.

Multiple problems like ABA, memory reordering, starvation etc.

Typical approach is based on moving a single step – instead of fixing the whole data structure, we adjust it by just a bit at a time.

Requires memory barriers.

https://preshing.com/20120612/an-introduction-to-lock-free-programming/
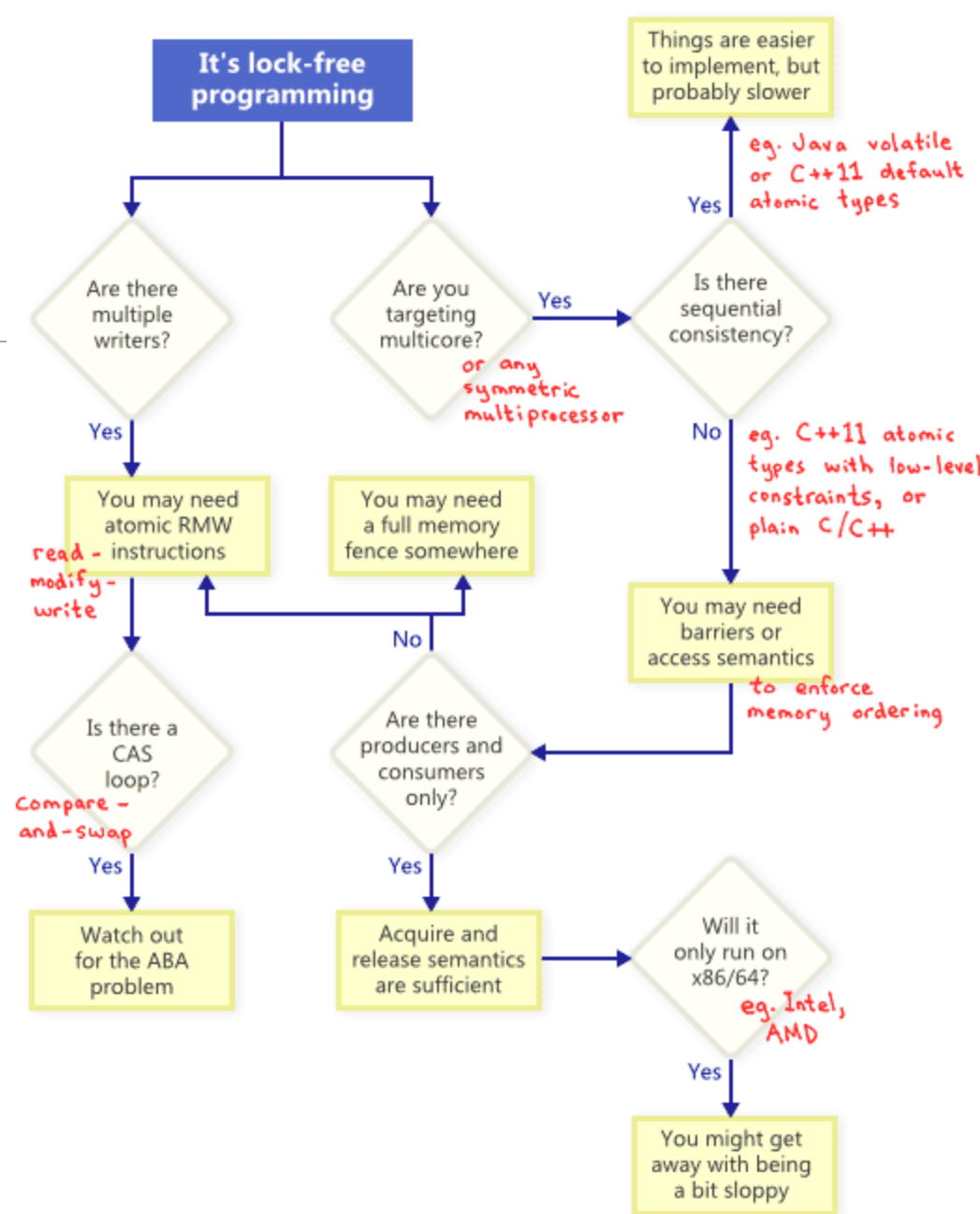
What about async?

It's not about concurrency nor parallelism. It's about optimizing the resource usage under the hood.

It may spawn additional threads, but can also use one thread for all code blocks.

Code may migrate between threads, so there is no proper locks semantic.

Typically implemented as a coroutine transformation (C#, JS, Python), but can be implemented as green threads as well (JVM).

Lock in async

```csharp
1   using System;
2   using System.Threading.Tasks;
3   using System.Threading;
4
5   public class Program
6   {
7       public static void Main()
8       {
9           Test().Wait();
10      }
11
12      public static async Task Test(){
13          var o = new object();
14          lock(o){
15              Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
16              await Task.Delay(100);
17              Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
18          }
19      }
20  }
21
```

```
Compilation error (line 16, col 4): The 'await' operator cannot be used in the body of a lock statement
```

```csharp
1  using System;
2  using System.Threading.Tasks;
3  using System.Threading;
4
5  public class Program
6  {
7      public static void Main()
8      {
9          Test().Wait();
10     }
11
12     public static async Task Test(){
13         var o = new object();
14         bool taken = false;
15         try{
16             Monitor.Enter(o, ref taken);
17             Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
18             await Task.Delay(100);
19             Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
20         }finally{
21             if(taken){
22                 Monitor.Exit(o);
23             }
24         }
```

```
1
4
Unhandled exception. System.AggregateException: One or more errors occurred. (Object synchronization method was called from an unsynchronized block of code.)
 ---> System.Threading.SynchronizationLockException: Object synchronization method was called from an unsynchronized block of code.
   at System.Threading.Monitor.Exit(Object obj)
   at Program.Test()
   --- End of inner exception stack trace ---
   at System.Threading.Tasks.Task.Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken)
   at System.Threading.Tasks.Task.Wait()
   at Program.Main()
Command terminated by signal 6
```

Reentrant recursive async lock

Reentrant recursive async lock is possible but the reasoning about it is much harder.

```
await Operation().ConfigureAwait(false);
```

Rule of thumb: replace all your awaits with threads and see if it deadlocks.

```csharp
class AsyncLock : IAsyncDisposable {
    private object locker = new object();
    private string path = "";
    private AsyncLocal<string> contextPath;
    private Random random = new Random();
    private List<TaskCompletionSource> waiters = new List<TaskCompletionSource>();

    public AsyncLock(){
        contextPath = new AsyncLocal<string>();
        contextPath.Value = "";
    }

    public Task<IAsyncDisposable> LockAsync(){
        bool taken = false;
        TaskCompletionSource waiter = null;

        while(!taken){
            lock(locker){
                char nextCharacter = (char)((int)'a' + random.Next('z' - 'a'));
                if(contextPath.Value == path){
                    path = path + nextCharacter;
                    contextPath.Value = path;
                    taken = true;
                }else{
                    waiter = new TaskCompletionSource(TaskCreationOptions.RunContinuationsAsynchronously);
                    waiters.Add(waiter);
                }
            }

            if(!taken && waiter != null){
                return waiter.Task.ContinueWith(t => LockAsync()).Result;
            }
        }

        return Task.FromResult((IAsyncDisposable)this);
    }

    public ValueTask DisposeAsync (){
        lock(locker){
            if(path.Length == 1){
                path = "";
            }else{
                path = path.Substring(0, path.Length - 1);
            }

            contextPath.Value = path;

            foreach(var waiter in waiters){
                waiter.SetResult();
            }
            waiters.Clear();
            Monitor.PulseAll(locker);
        }

        return ValueTask.CompletedTask;
    }
}
```

Custom implementations

LOCKS ARE TRICKY - ADAM FURMANEK

Semaphore with Mutexes

```csharp
var semaphore = new Semaphore(2, 2, "semaphoreName");

semaphore.WaitOne();
Console.WriteLine("Acquired!");

Console.ReadLine();
semaphore.Release();
Console.WriteLine("Done");
```

```csharp
var mutexes = Enumerable.Range(0, 2).Select(i => new Mutex(false, "mutexName" + i)).ToArray();

int id = -1;
try
{
    id = WaitHandle.WaitAny(mutexes);
}
catch(AbandonedMutexException e)
{
    id = e.MutexIndex;
}

Console.WriteLine("Acquired");

Console.ReadLine();

mutexes[id].ReleaseMutex();

Console.WriteLine("Released");
```

File locking

We can use files to implement a lock.

However, there is no fairness here and we may need to spin-wait for the access (depends on the runtime under the hood).

May not work across file systems.

```
new FileStream(name, FileMode.Open, FileAccess.Read, FileShare.None);
```

```
File file = new File("filename");
FileChannel channel = new RandomAccessFile(file, "rw").getChannel();
```

Mutex with ownership tracking

Windows Mutex doesn't tell you who owns it (neither does pthread).

We want to wait for some finite amount of time and then kill the owner because we assume it deadlocked or died.

We want to take a memory dump of the proces before killing it.

We want to be safe when it comes to access violation exceptions which we cannot handle.

How do we know who to kill? How to implement such a lock?

Mutex with ownership tracking

We create a memory-mapped file which anyone can view.

We then perform compare-and-swap to store 64 bits: 32 bits for the process id, 32 bits for the thread id.

When CAS fails, we look for the thread and see if it's still alive. If it's not then we just forcefully clear the lock.

If the thread is alive and it held the lock for too long then we take a memory dump of it and kill it.

We then forcefully clean the lock.

Challenges

No fairness.

Taking a memory dump is risky – it may block the process.

This is a spin-wait effectively so it hogs the CPU.

Ideally we should store 128 bits of data.

It doesn't support reentrancy.

However, it's successfully working in production for a couple of years now but it wasn't scientifically verified. Consider it a proof of concept.

Summary

Summary

Avoid locks when possible but keep in mind they are there.

Keep your critical sections as short as possible.

Use built-in data structures.

Always use a timeout when taking a lock.

Keep in mind semaphores are nasty.

Don't use locks in async.

Q&A

References

Joe Duffy - „Concurrent Programming on Windows"

Andrew S. Tanenbaum - „Structured Computer Organization"

Richard L. Sites - „Understanding Software Dynamics"

Maurice Herlihy, Nir Shavit - „The Art. Of Multiprocessor Programming"

Joseph Albahari - „Threading in C#"

Sasha Goldshtein, Dima Zurbalev, Ido Flatow - „Pro .NET Performance"

Stephen Toub - „Patterns of Parallel Programming"

References

https://devblogs.microsoft.com/oldnewthing/20140905-00/?p=63 – File lock

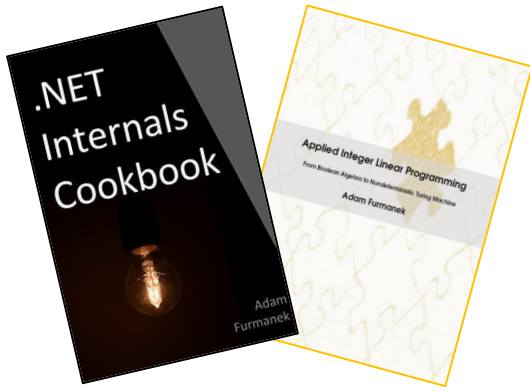https://itnext.io/reentrant-recursive-async-lock-is-impossible-in-c-e9593f4aa38a - Async lock

https://github.com/dotnet/wcf/blob/ce7428c2962e4ea4fce9c9c3e5999758a52bc4b9/src/System.Private.ServiceModel/src/Internals/System/Runtime/AsyncLock.cs - Async lock again

https://preshing.com/20120612/an-introduction-to-lock-free-programming/ - Lock free programming

https://www.osr.com/nt-insider/2015-issue3/the-state-of-synchronization/ - Kernel locks

https://wiki.c2.com/?ActorVsAgent – Actors and agents

https://blog.adamfurmanek.pl/2018/04/28/concurrency-part-1/ - my series about concurrency

Random IT Utensils

IT, operating systems, maths, and more.

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

🐦 FURMANEKADAM