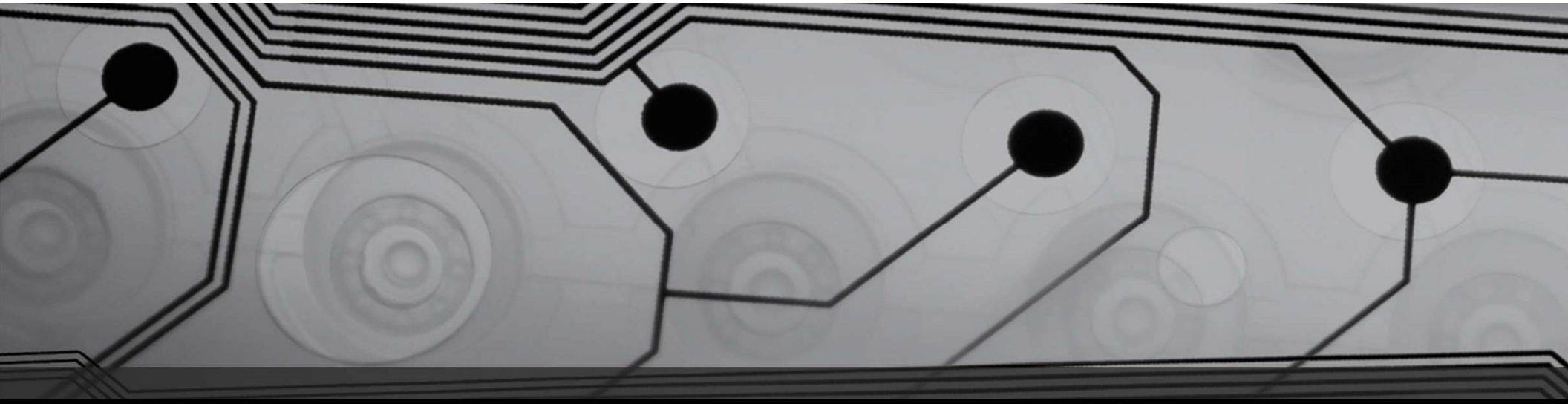
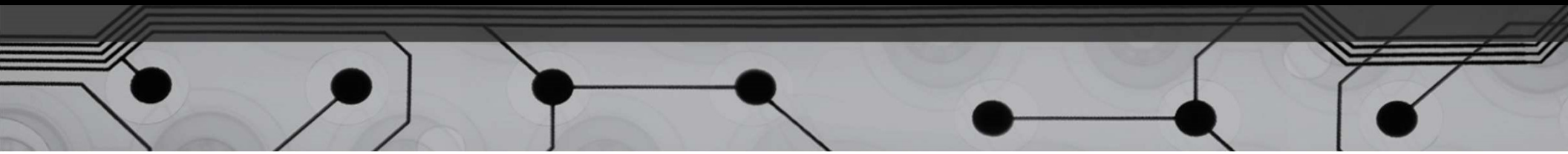


Low Latency Programming

Anthony Abate
anthony.abate@gmail.com



Latency

- Latency is the delay from input into a system to desired outcome

- Types:
  - Network / Transmission
  - Disk / Storage
  - Data Processing (This Presentation)

- 99 % of all code you write should not be written this way.
- This is for the 1% that needs to go ultra fast.
- Many concepts are language agnostic

Ground Rules

- C# Programming Domain Only
  - No Kernel Modules / Drivers
  - No Inline Assembly or C/C++
  - No Hardware (GPU, FPGA, ASIC)
  - Specific ‘tricks’ are done to influence the generated IL / Assembly code
- CPU Bound Problems Only
  - No Network or Disk I/O

Goal: learn techniques you can use without being ‘one with the hardware’

- Avoiding very ‘low level’ issues
  - CPU Cache Coherence problems
  - Memory barriers / fences
  - Instruction re-ordering (hyper threading, JIT optimizer)
  - Memory Paging / Swapping
- Develop on your target hardware or you will have bugs
- Intel i7 vs Xeon have different memory models
  - (weak vs strong)
- References / caveats where appropriate

Low-Level Hardware Considerations

- (Many topics very advanced, and each can be a presentation in themselves)
- Architectures have different CPU instructions
  - x86 vs x64 vs ARM
  - AMD vs Intel
- Different SIMD Optimized Instructions
  - MMX, SSE, AVX
- Different L1,L2,L3 Cache sizes
- Speed (Ram, Disk, PCI Bus, etc)

Better Hardware Helps ($$$)

- Faster / More Ghz = More CPU cycles per second
  - Overclocking / Liquid Cooled
- More Cores = More bandwidth
  - But, usually slower overall clock speed
- More CPU Cache
  - (We'll revisit this)

Time scale

Second	Millisecond	Microsecond	Nanosecond	Picosecond
1	10^{-3}	10^{-6}	10^{-9}	10^{-12}

- Milliseconds are Slow
  - Server Pings
  - Web Requests
- Low Latency is Fast = Micro to Nanoseconds
  - 1,000 – 1,000,000 faster

Overview

- Part 0 - Why Speed Matters
- Part I - Measuring Speed
- Part II - Coding (basic)
  - Optimizations / Best Practices
  - Signaling
  - Thread Creation
- Part III - Coding (Advanced)
  - Immutability
  - Advanced Optimization (Caching)
  - Lock-Free Design
  - Producer/Consumer

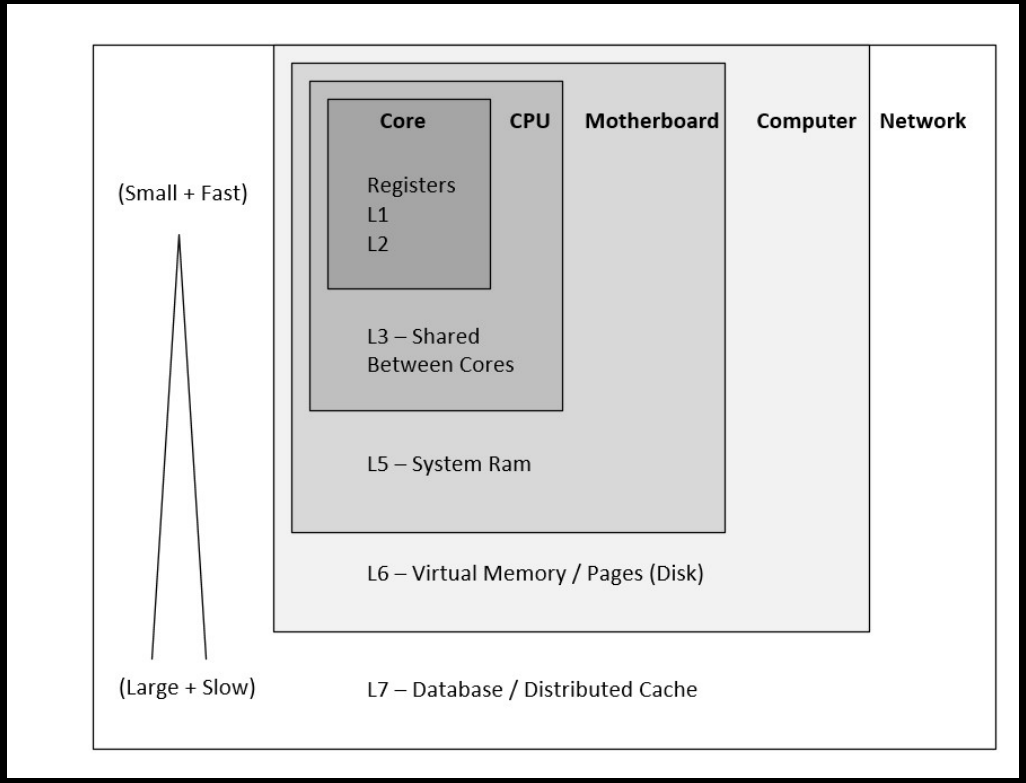
Part 0 – Why Optimize?

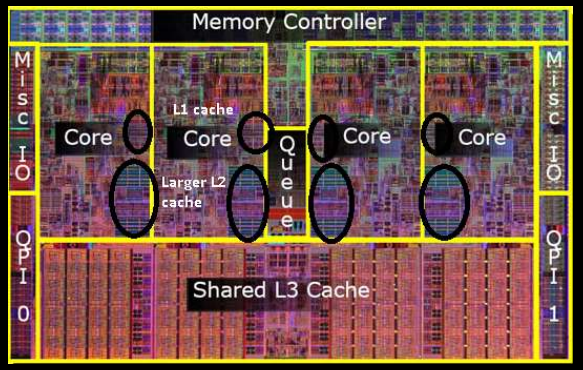
- Obviously, to be first! (fastest)
- Its not just writing fast code, write "Smart Code"
  - Don't Waste Memory
    - Data structures
  - Don't Waste Code
    - CPU Instructions take up memory too!
    - Function Calls add to Stack overhead
  - Don't Waste Time
    - Algorithms

CPU Cache – The Fastest Memory

- The fastest memory in your computer is the L1 Cache. Its also the smallest.
- CPU's will fetch memory from RAM containing code and data (Stack / Heap)
- The OS Kernel and User code are constantly moving around the caches / ram and process/threads operate on data and instructions.
- CPU's with Large caches are very expensive!
  - Less data moving around the motherboard
- L3 Cache shared among all cores
  - If needed data is in L3 (faster), no need to fetch it from RAM (slow)
- L1-L2 Cache Coherence / Synchronization Problem
  - Multiple copies of same data in different cores. If one changes, the others also need to change!)
  - JITer can reoder code based on the Memory Model unless you specify: Memory Barriers / Fences / Volatility

CPU Cache Levels





Don’t Waste Memory

- Your server has 1 TB of RAM its cache is only 50Mb (if your lucky)
- Using a lot of memory (or different parts of memory) will require Juggling.
  - Traversing Pointers vs Array (more later)
- If you really start wasting memory may have to fetch it DISK (Paging / Thrashing)
- Virtual Memory / Paging (Large Topic in itself)
  - Depending on OS / Configuration windows may store parts of memory on disk
  - Have to deal with Win32 API to optimize
  - Page Faults are bad (slow)
  - Turn Page File off?
  - x86 processes only have 3.5 gigs of addressable space

Part I – Measuring Speed

- Why do we care About Measuring Speed?
  - Bottlenecks are not always obvious
  - 1. Test
  - 2. Validate
  - 3. Improve
    - Previous time speeds serve as a baseline to evaluate changes

Real-Time Logging Dilemma

- Logging is good for diagnosing Production code
- Real-time / real-world timing data needed
- Logging is usually very slow (string + IO operations)

How (Not) To Measure (DateTime)

- Very low precision
- Time Scale in milliseconds
- A lot can happen in a Millisecond!

How To Measure (Stopwatch)

- Higher Precision
- Hardware based (Uses High Resolution Timer)
- Timescale is in ticks (not the same as DateTime ticks)
- Scale based on CPU Frequency
  - 1 Mhz = 1 tick
  - 1 Ghz = 1000 ticks

```
double ticks = sw.ElapsedTicks;
double seconds = ticks / Stopwatch.Frequency
double milliseconds = (ticks / Stopwatch.Frequency) * 1000
double nanoseconds = (ticks / Stopwatch.Frequency) * 1000000000
```

Datetime vs Stopwatch Resolution

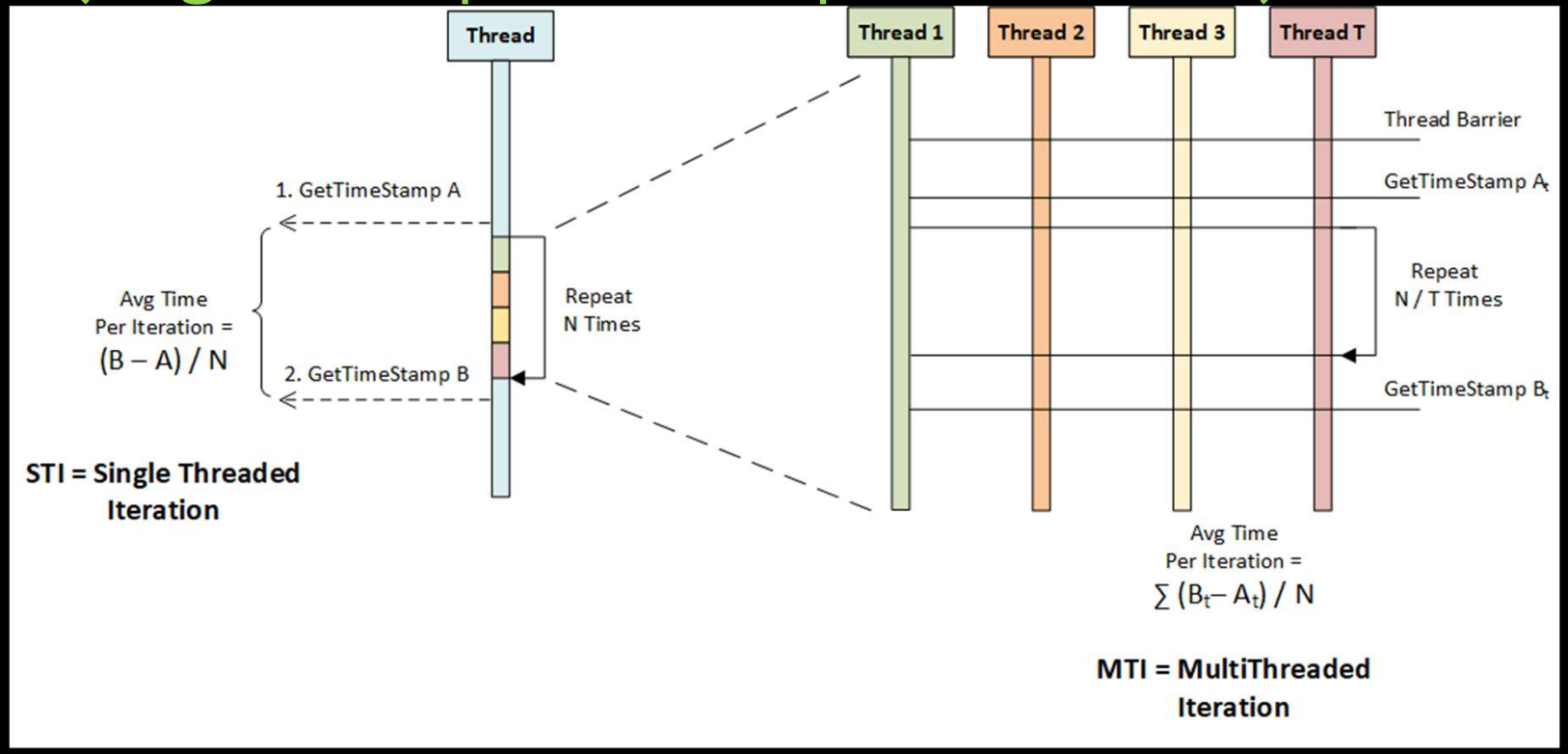
- Code Demo

Technique 1.A (Tight Loop – Single Thread)

- Averages time
- Can infer sub-tick level timings
- Does not measure directly
- Does not measure “Latency”
- Not “realistic”
- Between Start / Stop is a Mystery

```
var sw = new Stopwatch();

sw.Start();

for (int i = 0; i < 10000000; ++i)
{
    // Do Something
}

sw.Stop();
```

Technique 1.B
(Tight Loop – Multiple Threads)



- STI vs MTI = Overhead due to threading (Atomic Operations or Locking)

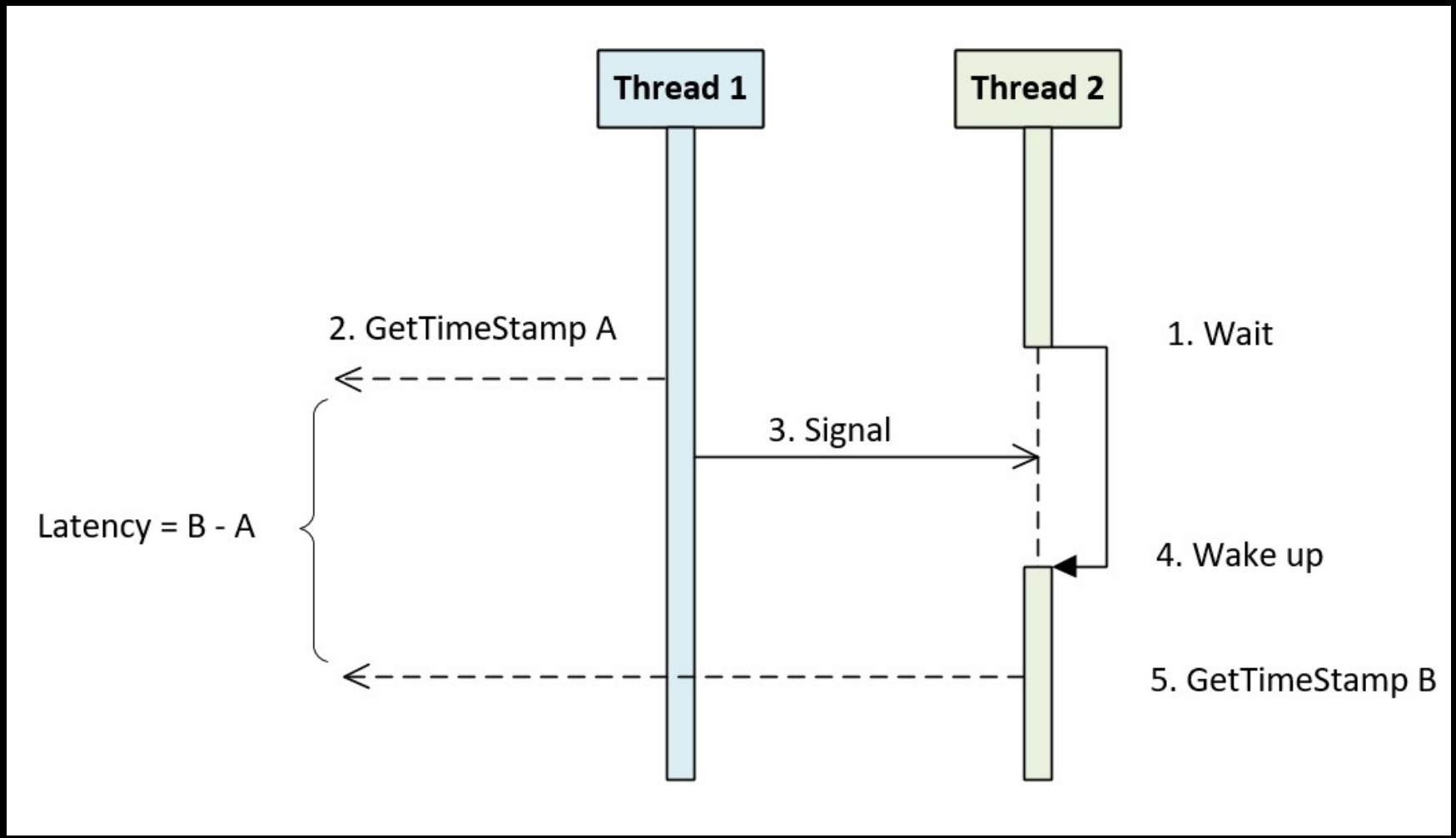
Notes About Testing

- First Execution of code may be very slow
  - Loading Assemblies
  - JIT
  - Static Initialization
- Debug vs Release
- IL Optimization
  - May re-order your code!
  - Memory Barriers / Fences to prevent (Advanced)

Technique 2 (Latency)

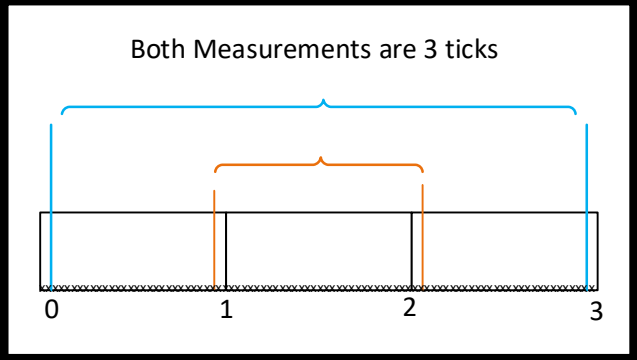
- Multithreaded
- Close to “Actual time”
- 1 Tick level Resolution
- Directly measures code blocks
- measures “Latency”
- “Real world” Scenarios

```
var sw = new Stopwatch();

//Thread 1
sw.Start();

//--- (Signal) -------

//Thread 2
sw.Stop();
```


Thread 1
Thread 2
2. GetTimeStamp A
1. Wait
3. Signal
Latency = B - A
4. Wake up
5. GetTimeStamp B

1 Tick = Latency Resolution

- 1 Tick = ~300 nanoseconds (my computer)
- A lot happens in a tick
- .. But.. Its “good enough”
- If you have latency, it will be MUCH HIGHER than 300ns
- Bottle necks become obvious



“Observer Effect”

- “changes that the act of observation will make on a phenomenon being observed”

- Profiling tools instrument and alter code
- Excessive logging

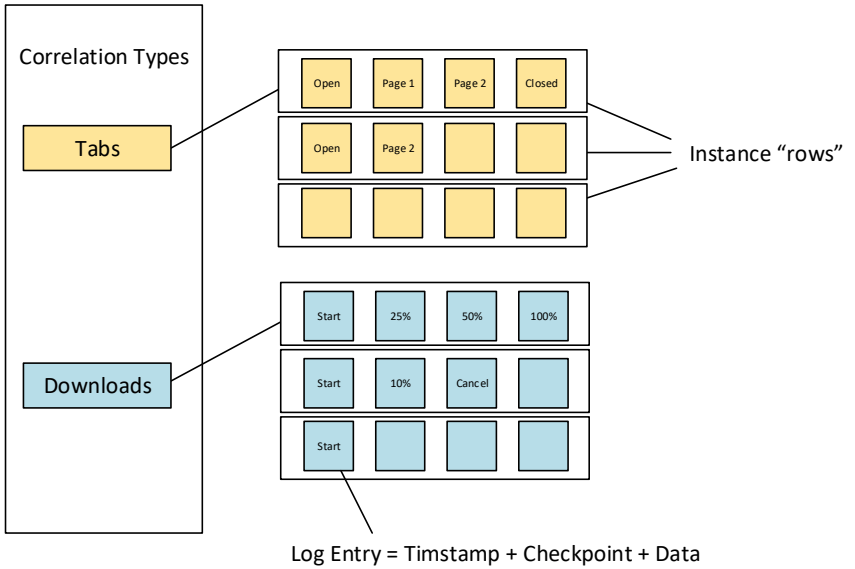
Log4net Overhead

- Code Demo

- I use log4net minimally
  - Error logging
  - Interesting ‘events’

Nano Logger (low Latency)

- Can build a simple logger using all the previous techniques combined
- Record
  - Timestamp (ticks)
  - Code “Checkpoint” Location
- 4 level Tree
  - Correlation Type
  - Instance Id
  - Checkpoint
  - Checkpoint Detail


Correlation Types
Tabs
Downloads
Open
Page 1
Page 2
Closed
Open
Page 2
Instance “rows”
Start
25%
50%
100%
Start
10%
Cancel
Start
Log Entry = Timstamp + Checkpoint + Data

Nano Logger

- Pre-Init / Pre-Allocate
  - Arrays of Arrays
- Integers only
- No allocations
- Non-Blocking, No Locks
- Shutdown for analysis

Nano Logger performance

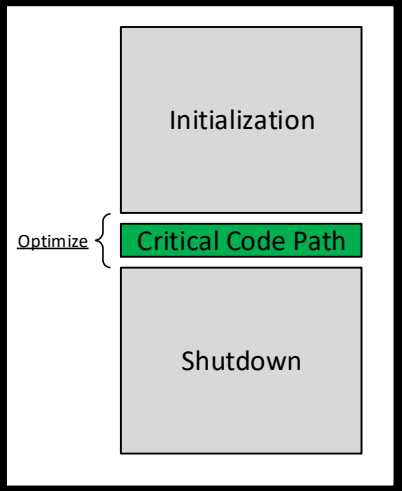
- NanoLogger vs Log4net
- 16-20ns vs 3-5ms (300,000x faster)
- 300 ns vs 3ms (10,000x more accurate)
- **5-6 Orders of Magnitude Difference!**

Part II - Coding (basic)

- "Common Sense"

- Program Code can be split into 3 Categories:
  - Startup / Initialization
  - Critical Path
  - Shutdown / Post Initialization

- Only optimize the "critical path"
- Correct Data structure / Algorithm choice for problem
- Constantly Measure and Test "critical path"



Pre-Allocate

- Bad to use ‘new’ in the critical code path
  - Takes ‘time’
  - can trigger possible Garbage Collection (blocks all threads)
- Requires Heap
  - Shared among all threads => Requires synchronization
    - Simultaneous heap allocations will Block

Pre-Initialize (Eager Loading)

- Some objects have long startups
- Pre Start threads
- Pre Open connections
- Factory / Builders Patterns
- Pre-Compute (more later)

Use Integer Types

- Fits into normal CPU registers
  - Most CPUs are 64 bit now
- Interlocked / Atomic operations
- Assembly opcode optimized
  - Arithmetic
  - Bit Operations

- Bool (1 bit)
- Byte / Sbyte (8bit)
- Ushort/ Short (16bit)
- Uint / Int (32bit)
- Ulong Long (64bit)

Enums are integers Too!

- Enums are value types (stack allocated)
- Can be used to assign meaning to numbers
- Flags
  - Can be used for bitmask
  - BIT Operations

Integer Type safety

- Type safety will help you find bugs via the compiler
- Use unsigned variants
  - For loop counters
  - Array indexers
  - Age fields
- Enums
  - Only allow the defined values
- Bool
  - True/false (1 bit)

Take Care with Floating Point

- Requires special CPU registers
- Possibly Larger than native CPU Register
  - Decimals (and Double on x86)
- Operations require more CPU Cycles

- Double / Float
  - Modern CPUs slightly slower than long data type

- Float (32bit)
- Double (64bit)
- Decimal (128bit) (slow)

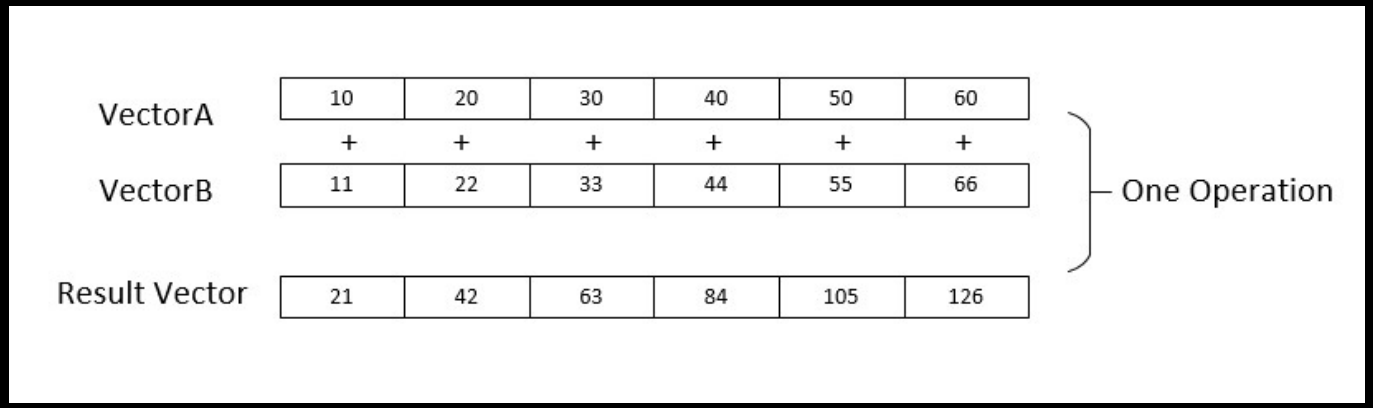
Int / Float Performance Test

- Test Following:
- ADD
- MULTIPLY
- DIVIDE

SIMD
(Single Instruction Multiple Data)

- Prepare arrays of data for computation in parallel
- System.Numerics Namespace
- Types are JIT'd using specialized CPU Instructions / OpCodes
- Behavior based on CPU (# of registers available for data type)
  - Vector<Float>.Length = 8 (8 – 32bit registers)
  - Vector<Long>.Length = 4 (4 – 64bit registers)
- Ideal for Linear Algebra Operations
  - Linear Regression - Model / Prediction
  - Matrix/Vector functions
  - Statistics

Float vs SIMD



BIT Operations

- Often forgotten in Modern Programming
- Shift Operations
  - Fast Multiple / Divide By Powers of 2
  - 100 << 2 = 400 `001100100 => 110010000`
  - 100 >> 2 = 25 `001100100 => 000011001`
- Boolean Operations / Bit Masks
  - Manipulate Each Bit Independently
  - Encode Data
  - Often used with Masks

```
110011100 (Data)
000001000 (Mask)
110010100 (XOR Result)
```

Bits vs Array vs List vs Dictionary

Don’t use strings!

- String operations are very slow
- Arbitrary size means unknown performance
- Heap allocated
- CPU operations not optimized
- Immutable
  - intermediate strings may be created with every operation
- Use the ‘initialize’ phase of your code to convert strings to integers
- Use the ‘shutdown’ phase to build strings / messages

Use The Stack…

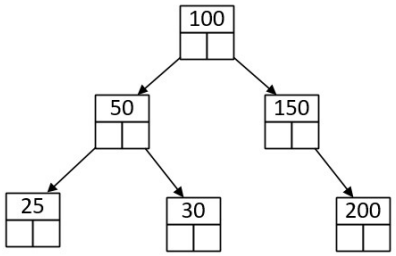
- Stack variables avoid the heap
- Value Types (Structs, Enums)
- Stackalloc (similar to ALLOCA() in C++)
- Thread Local Storage (TLS)

Use Arrays

- Indexed using integers
- Can be stack allocated
- The basis of many non-blocking / lock free data structures
- Can be ‘thread-safe’ without locking

Tree Data Structures

Tree Nodes



Fragmented Memory - Each Tree Node Class and Reference points to a different part of the heap. Traversals will be slow

Multiple Arrays

Data	100	50	25	150	30	200
Left Index	1	2	-	-	-	-
Right Index	3	4	-	5	-	-

Contiguous Memory – Fits better in CPU Caches. Traversals will be much faster

More Things to Avoid

- Boxing / Unboxing
- Serialization
- Exceptions
- Heavy Function Calls
  - System Calls
  - P-Invoke
  - Thunking (C++/CLI)
- Fancy Runtime features
  - Reflection
  - Dynamic object
  - Dynamic proxy
  - MarshalByRefObject

Shutdown / Time after Critical path

- While your program is ending, or coming out of time critical path use that time to extract more information
- Convert timestamps to DateTimes
- Convert Integers to strings for logging
- Do string operations if needed

.Net Performance (Demos)

- Increment
- Tasks / Threads
- Signal / Notify threads
- Functions
  - Static - global
  - Class Member - Should be slower: needs to load memory offset for member variables
  - Virtual Function – Should be slower: needs to look for function in VTable

.Net Performance Conclusion

- Increment:
  - Fastest is no lock (hard to design… but we'll revisit this soon)
  - Interlocked fastest 'thread safe' when no competition
  - SpinLock => slightly faster than Lock for multiple threads
- Task / Create
  - Best to have dedicated threads already running / pre-initialized
- Signaling
  - Busy/while loop is fastest, but dedicates a thread to a core
- Functions
  - No perceptible difference between static, class, virtual

Part III - Coding (ADVANCED CONCEPTS)

Garbage Collector

- If the Garbage Collector is running, you have bigger problems, but its nice to have some deterministic control with the following:
- Put the GC in low latency mode to minimize collections

```
System.Runtime.GCSettings.LatencyMode = System.Runtime.GCLatencyMode.LowLatency;
```

- Prevent GC from running in a block of code as long as the request memory exists

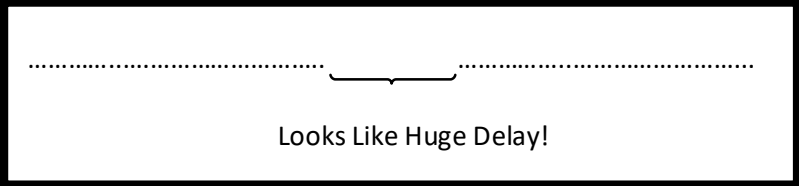
```
var heap = 10000000;
var loHeap = 10000000;
var preventBlocking = true

if (GC.TryStartNoGCRegion(heap,loHeap,preventBlocking))
{
    // DO WORK   (Possible OutOfMemoryExceptions)

    GC.EndNoGCRegion();
}
```

Cores / Threads / Context switches

- Manually create and manage threads
- Dedicate cores to specific work threads
- Set Thread priority where appropriate
- Be mindful that context switches can occur if system is overloaded

Context Switch

- Hard to avoid them without manually balancing thread/core workloads
- Default Time Slice on windows is ~30 ms (can change with Win32 Api)
- Can't predict them
- Can't measure them… directly
  - Keep All Cores Busy
  - Continuously call Stopwatch.TimeStamp()
  - There should be a "Large" gap in time
- Code Demo



Unroll functions / Loops (maybe)

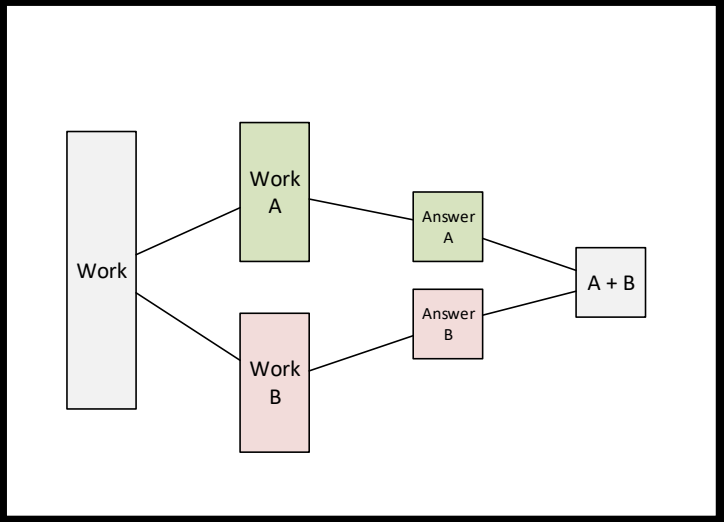
- C# doesn't have "inline" like C++
- JITer will inline functions
- "Aggressive Inline"
- Unroll for loop / counter
- Static vs Non-Static function calls

Immutable Designs

- Single assignment
- Value can never change
- Avoids blocking / concurrency issues
- Erlang (RabbitMQ)

Partition & Merge (Divide & Conquer)

- Split work into “immutable” partitions
- Partitions should be isolated and Non-overlapping
- All work to complete should be present
- Differ merge till after critical code path
  - (threads can log locally, on shutdown merge with parent)
- Merge surface area should be as small as possible
  - Record Version #



DB Transaction (optimistic)

- Databases use optimistic concurrency
- Unlikely 2 clients update same data at once
- No need for 'heavy lock'
- Rows have a 'version number'
  - SQL "Timestamp" type
  - very small merge 'surface area'

- 1. Client caches data with version
- 2. Client does work on data
- 3. Client submits data back with same version
- 4. if DB version == Client Version => Commit
  - else Concurrency Error!
- 5. Client must start over (Store Wins)

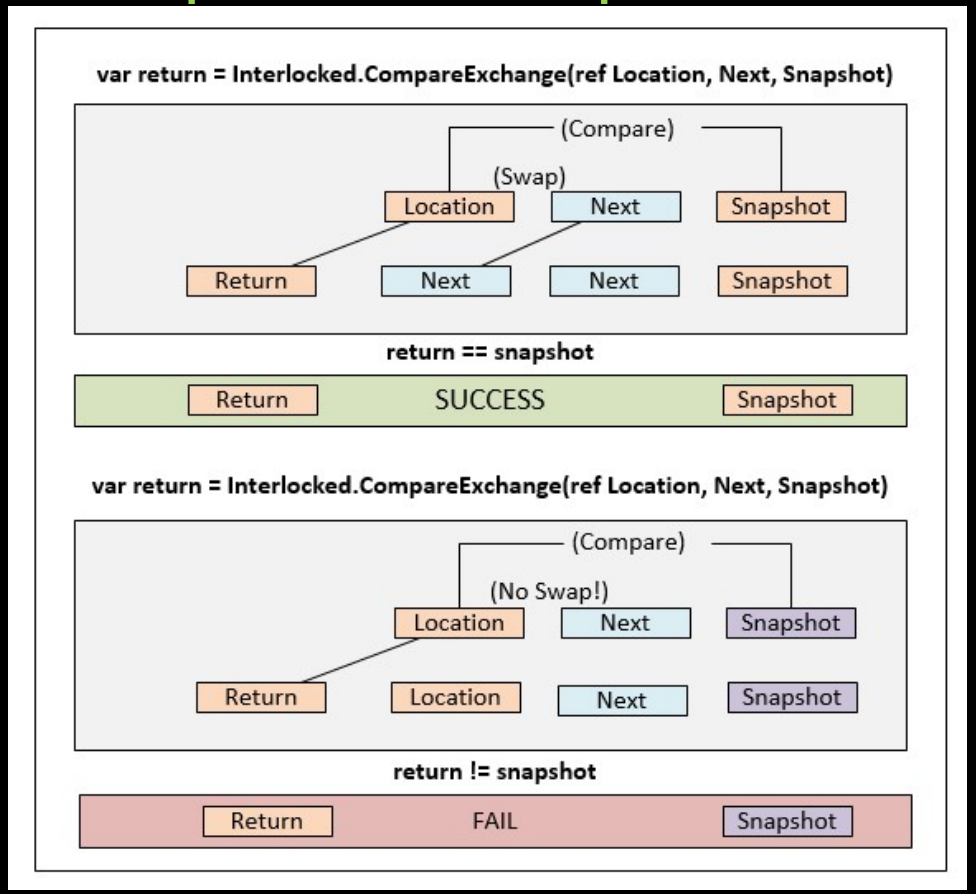
```
while(true)
{
    var versionSnapshot = this.storage.Snapshot();

    var next = this.OnComputeNext(versionSnapshot, data);

    // commit checks version numbers
    var committed = this.storage.Commit(next, versionSnapshot);

    if (committed)
    {
        return true;
    }
}
```

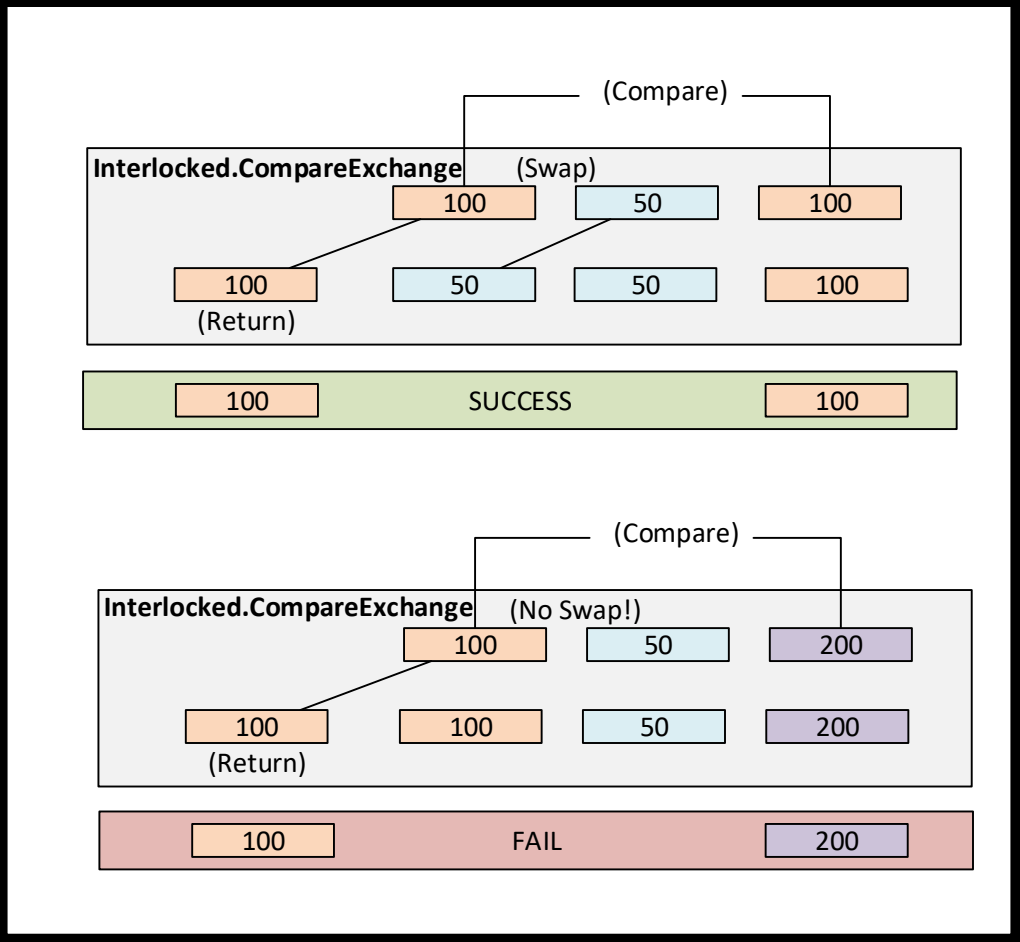
CompareExchange / CompareAndSwap

- Very confusing…
- but behaves just like
- DB Transaction

```
public static int CompareExchange(
        ref int location1,
        int value,
        int comparand
```

- Check result to determine success




(Compare)
Interlocked.CompareExchange
(Swap)
100
50
100
100
50
50
100
(Return)
100
SUCCESS
100
(Compare)
Interlocked.CompareExchange
(No Swap!)
100
50
200
100
100
50
200
(Return)
100
FAIL
200

Caching / Memoization

- Store values that require computation for future use
- Requires a hash table / dictionary
  - Prefer integer based arrays
- caveats:
  - may require locking
  - If no locking, there might be duplicate initialization
  - Expiration?
- Memoization – function cache: F(x,y,z) => value

Pre-Caching / Memoization

- Memory is cheap, computers have lots of it
- Have lots of time during ‘start up’
- Precompute all possible outputs based on all ‘expected’ inputs
  - Multi dimensional arrays / jagged arrays
  - cache[var1][var2][var3] => value
  - F(var1,var2,var3) => value (just like memoization)
- Once initialized “read only”
- Thread safe

Lock-free / Non-Blocking Structures

- Avoid waiting for lock (ie blocking) and yield its time slice (major delay)
- Combine many of the basic and advance techniques
- Very Hard to write correctly
  - Volatile Modifier
  - Memory Fences
    - Volatile.Read
    - Thread.VolatileRead
    - Interlocked.Read
    - Thread.MemoryBarrier

Newer .NET Collections

Concurrent (Thread safe)

(caveat : callbacks can be called more than once!)

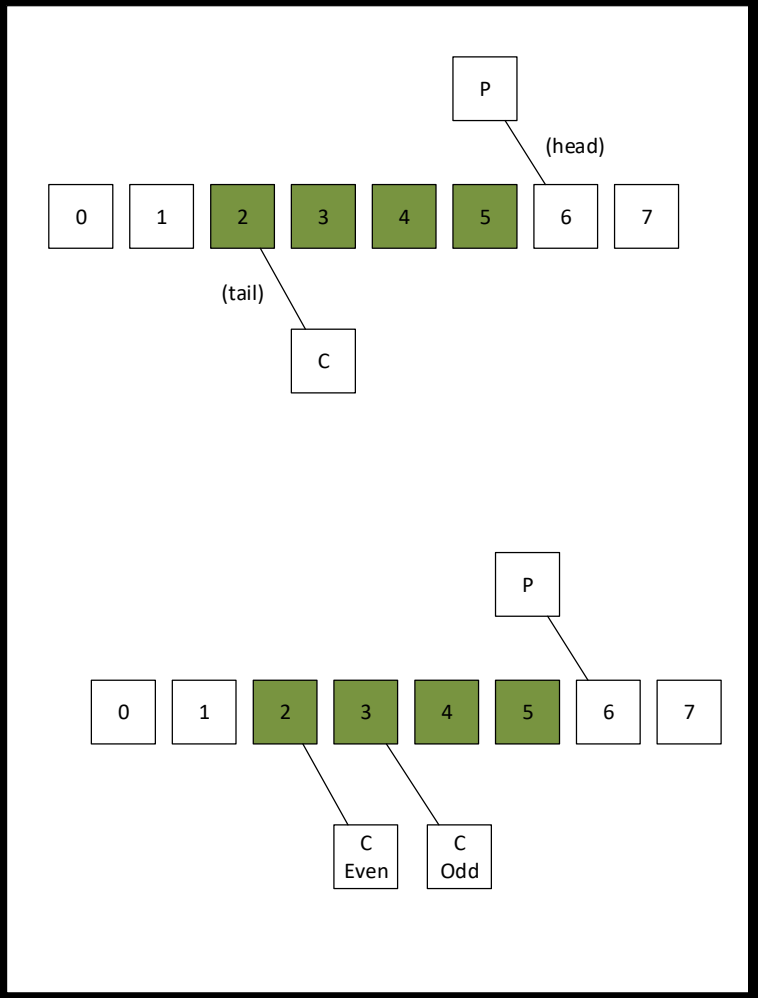
- ConcurrentBag (unordered List)
- ConcurrentDictionary (Hashset also)
- ConcurrentQueue
- ConcurrentStack

Immutable

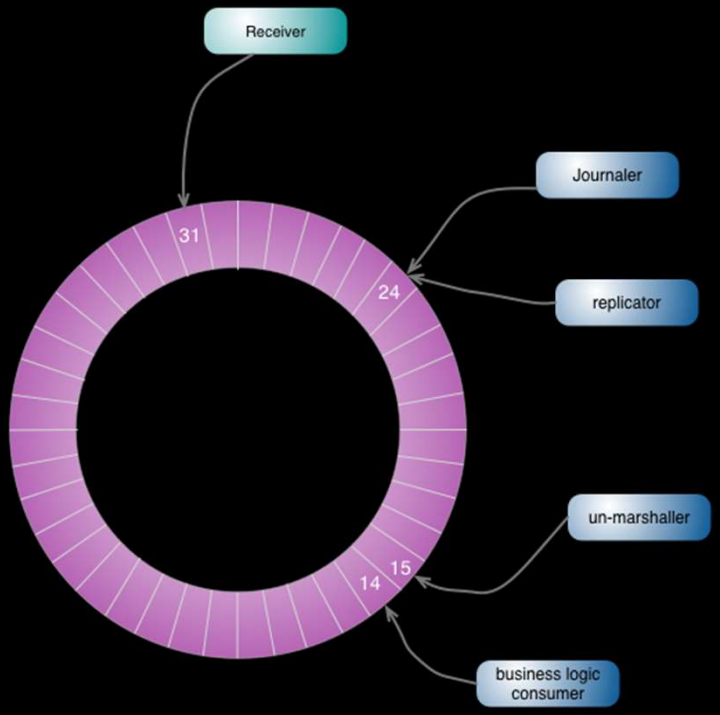
- ImmutableList
- ImmutableDictionary
- ImmutableQueue
- ImmutableStack
- ImmutableArray
- ImmutableHashSet

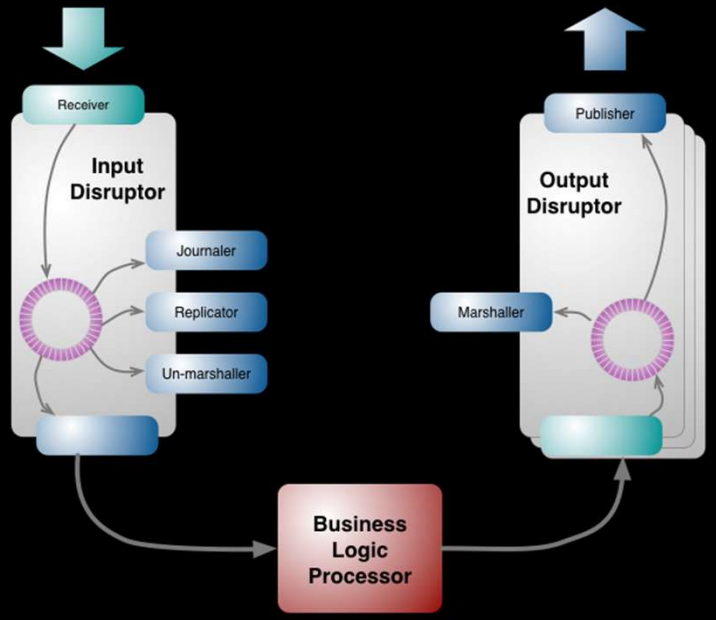
Circular Queues

- Queues inherently “thread safe”
  - Read / Write at opposite ends
- 1 produce / 1 consumer (Ring buffer)
- N Producers / M consumers



Disrupter Pattern





Advanced Functions / Sync objects

- Spinlock (struct)
- Spinwait (struct)
- yielding

Future Slides

- Binary Search vs Linear Search
- Processor Differences (instruction re-ordering)
  - 8 Cores vs 12 Cores (Interlocked slower with more cores ?? Causes more cache sync)