Lock-Free Programming: Pro Tips

Jean-Philippe BEMPEL
Performance Architect

@jpbempel
http://jpbempel.blogspot.com

Ullink
CONNECT. TRADE.

Agenda

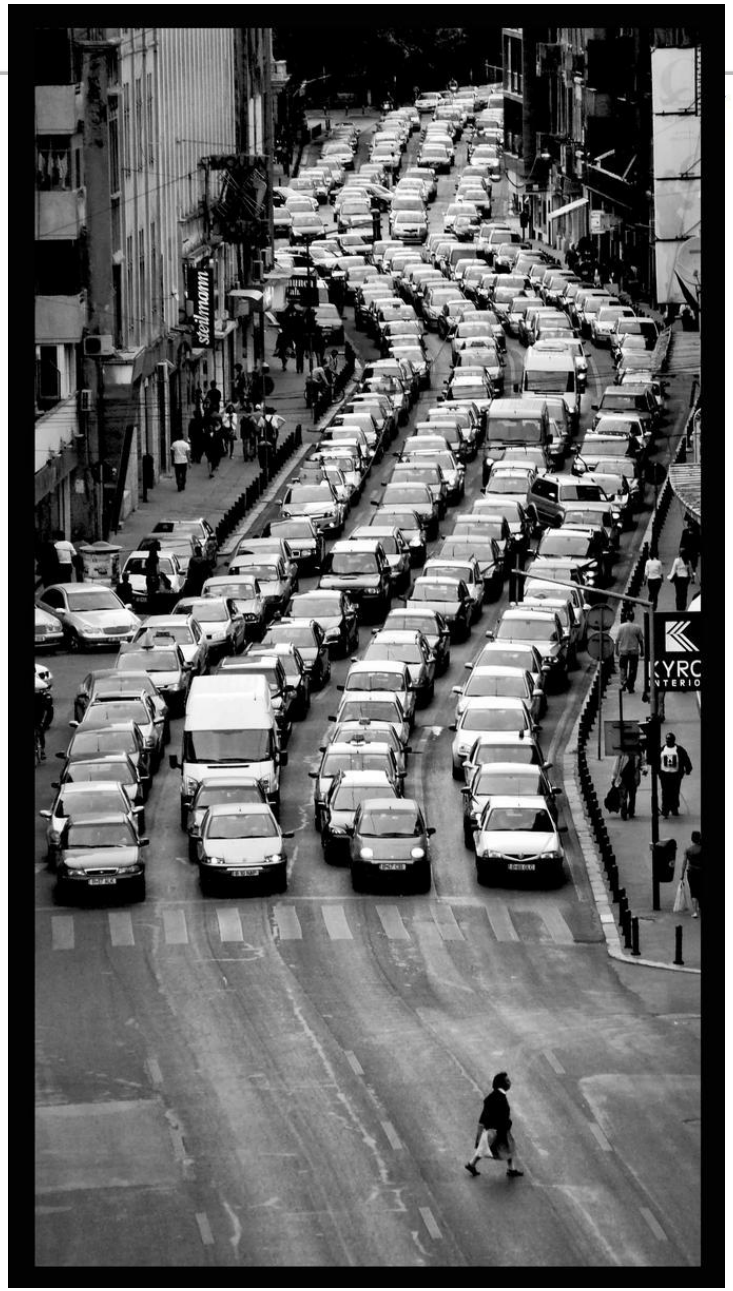
- Measuring Contention
- Lock Striping
- Compare-And-Swap
- Introduction to Java Memory Model
- Disruptor & RingBuffer
- Spinning
- Ticketing: OrderedScheduler

Immutability

Contention

Contention

- Two or more thread competing to acquire a lock
- Thread parked when waiting for a lock
- Number one reason we want to avoid lock



ullink

Measure, don’t guess!

Kirk Pepperdine & Jack Shirazi

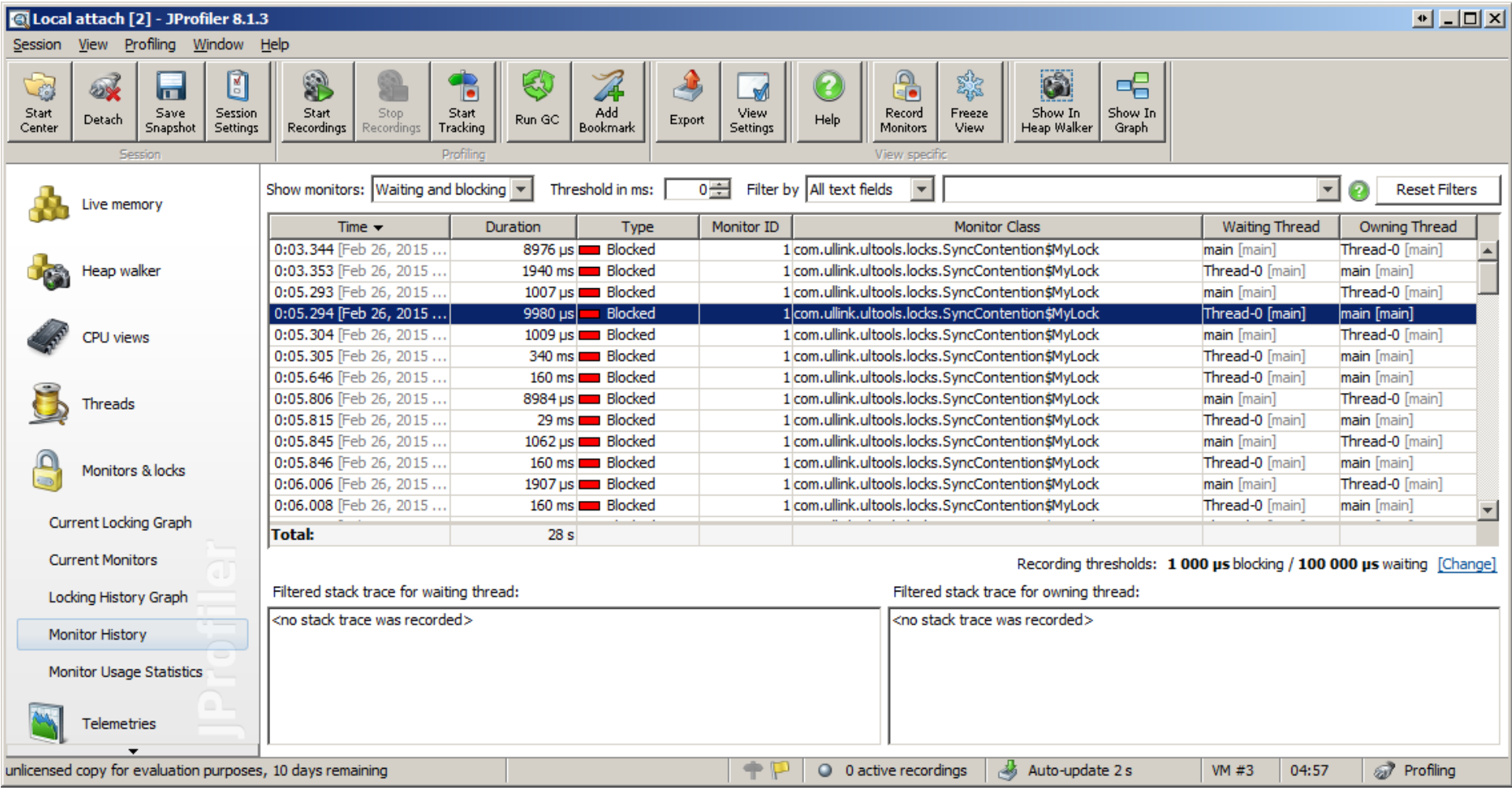
Measure, don’t premature!

Measuring Contention

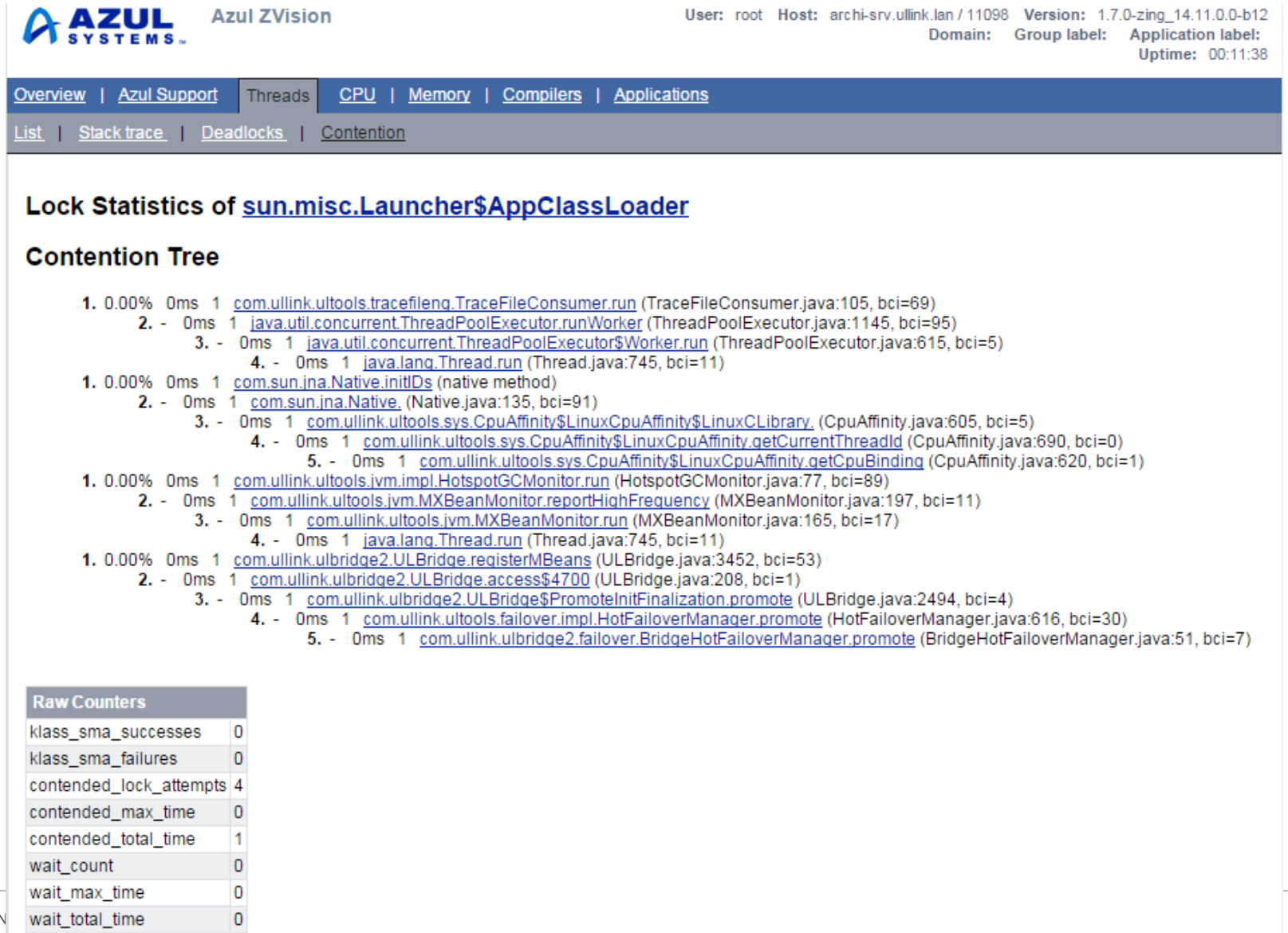
Synchronized blocks:

- Profilers (YourKit, JProfiler, ZVision)
- JVMTI native agent
- Results may be difficult to exploit

Measuring Contention: JProfiler



Measuring Contention: ZVision

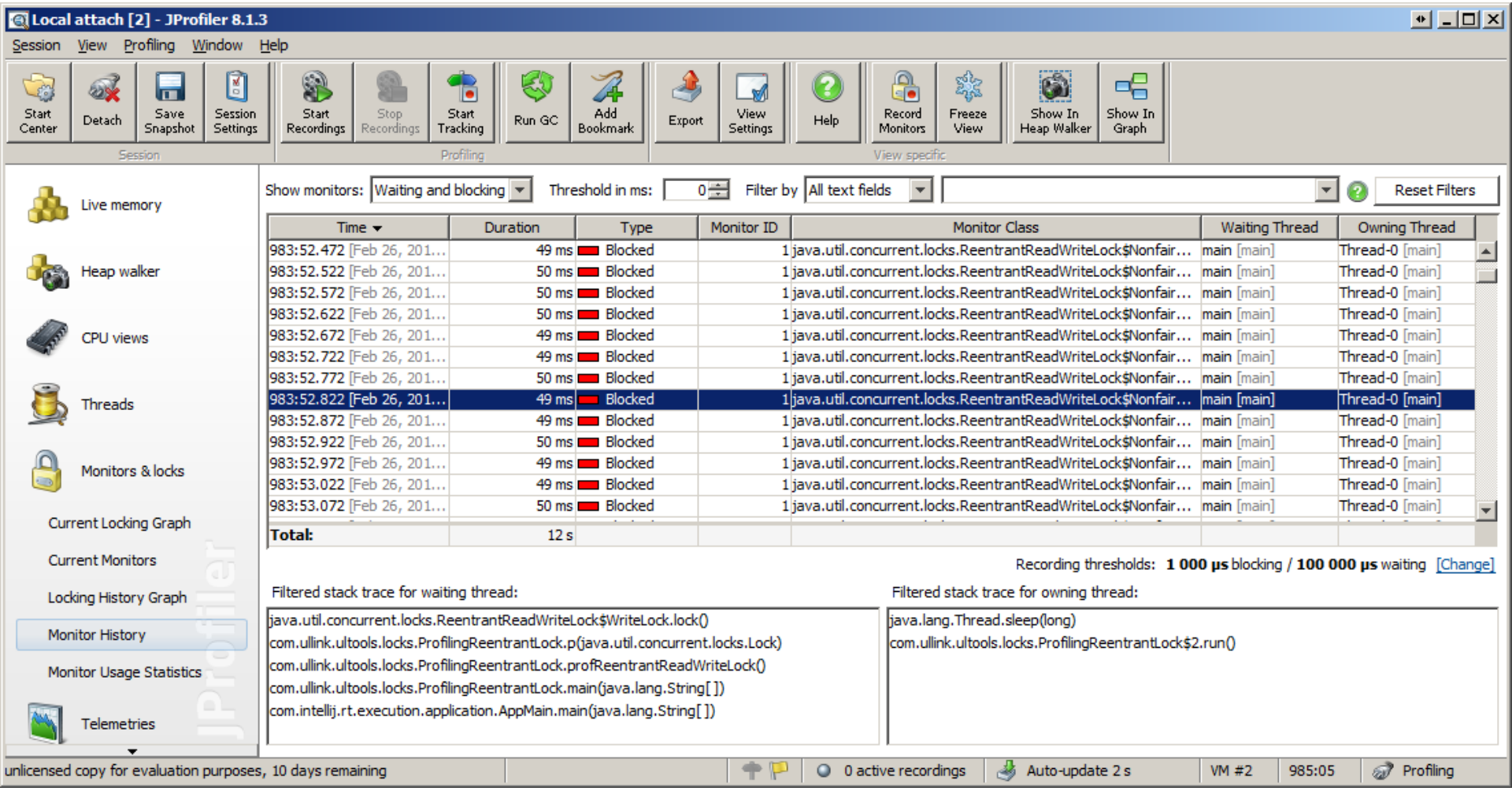


Measuring Contention

java.util.concurrent.Lock:

- JVM cannot helps us here
- JDK classes (lib), regular code
- JProfiler can measure them
- j.u.c classes modification + bootclasspath (jucprofiler)

Measuring Contention: JProfiler



Measuring Contention

- Insertion of contention counters
  - Identify place where lock fail to be acquired
  - increment counter
- Identify Locks
  - Call stacks at construction
  - Logging counter status
- How to measure existing locks in your code
  - Modify JDK classes
  - Reintroduce in bootclasspath

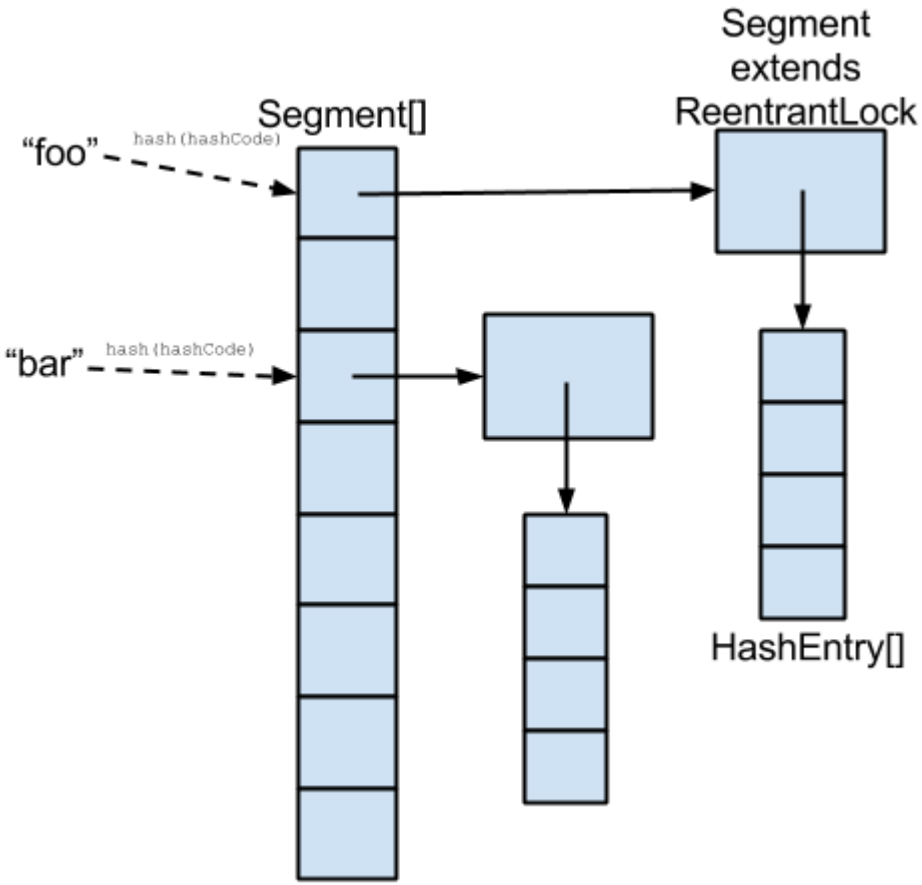
Lock striping

Lock striping

- Reduce contention by distributing it
- Not remove locks, instead adding more
- Good partitioning is key to be effective (like HashMap)

Lock striping

Best example in JDK: ConcurrentHashMap



Lock striping

- Relatively easy to implement
- Can be very effective as long as good partitioning
- Can be tuned (number of partition) regarding the contention/concurrency level

Compare-And-Swap

Compare-And-Swap

- Basic primitive for any lock-free algorithm
- Used to implement any locks or synchronization primitives
- Handled directly by the CPU (instructions)

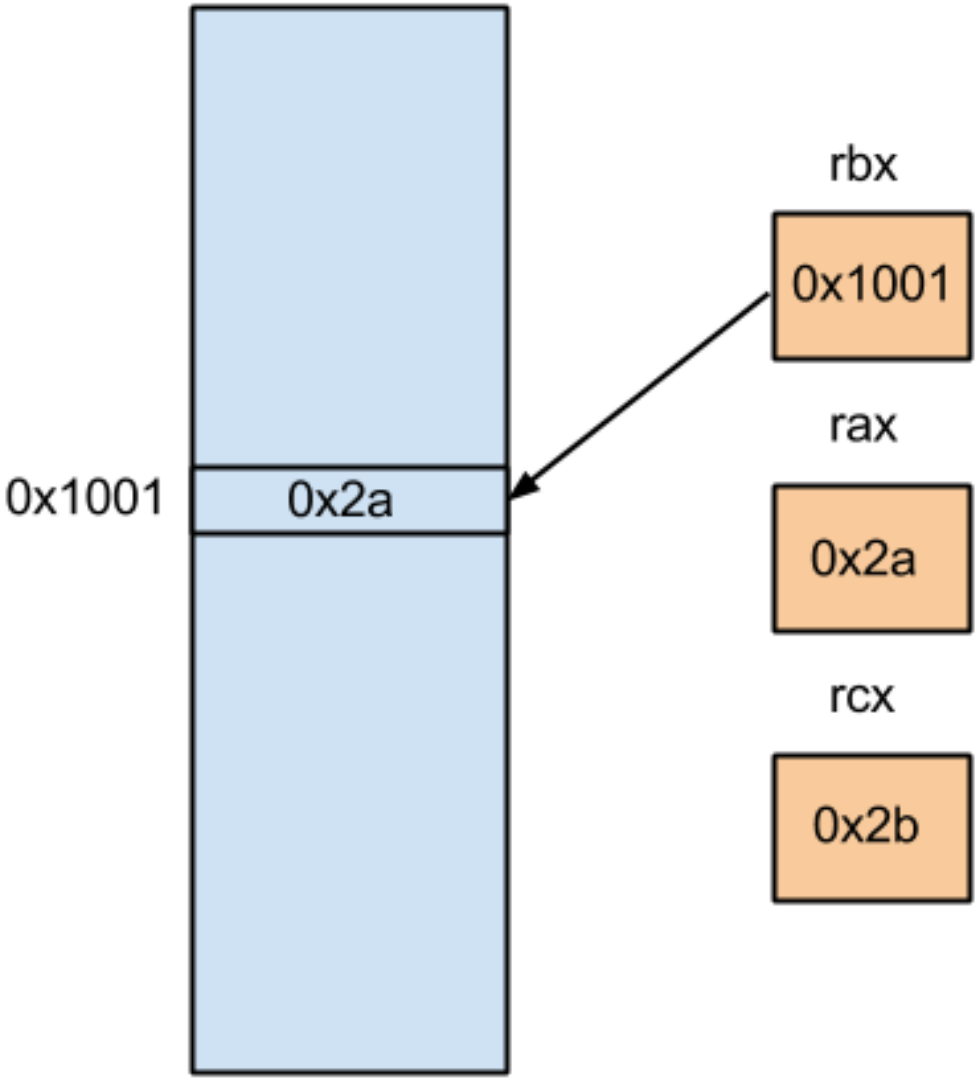
Compare-And-Swap

- Update atomically a memory location by another value if the previous value is the expected one

- instruction with 3 arguments:
  - memory address (rbx)
  - expected value (rax)
  - new value (rcx)

```
movabs rax,0x2a
movabs rcx,0x2b
lock cmpxchg QWORD PTR [rbx],rcx
```

Compare-And-Swap



Compare-And-Swap

- In Java for AtomicXXX classes:

  ```
  boolean compareAndSet(long expect, long update)
  ```

- Memory address is the internal `value` field of the class

Compare-And-Swap: AtomicLong

- Atomic increment with CAS
  [JDK7] getAndIncrement():

```
while (true) {
    long current = get();
    long next = current + 1;
    if (compareAndSet(current, next))
        return current;
}
```

  [JDK8] getAndIncrement():

```
return unsafe.getAndAddLong(this, valueOffset, 1L);
```

  intrinsified to:

```
movabs rsi,0x1
lock xadd QWORD PTR [rdx+0x10],rsi
```

Compare-And-Swap: Lock implementation

ReentrantLock is implemented with a CAS:

```
volatile int state;

lock()
  compareAndSet(0, 1);
```

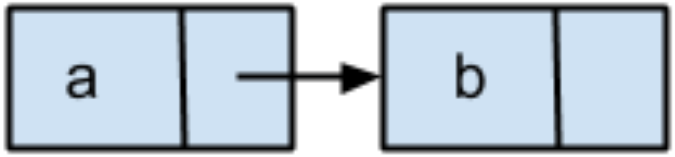
if CAS fails => lock already acquired

```
unlock()
  setState(0)
```

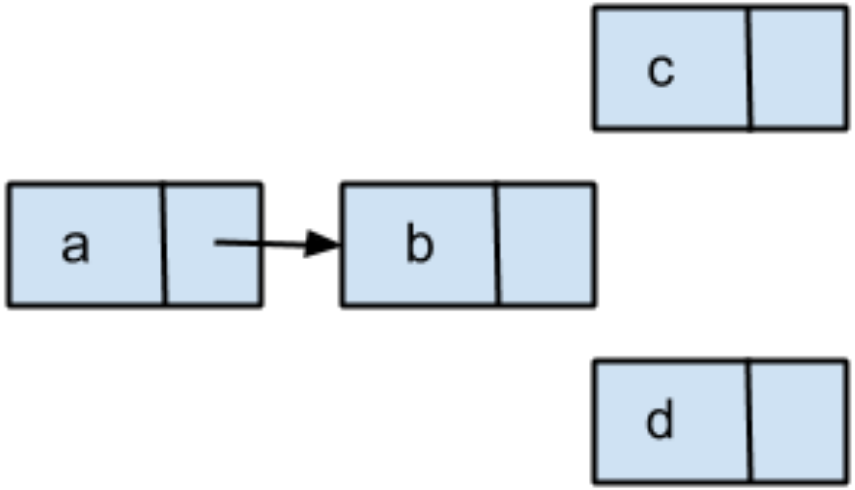
Compare-And-Swap: ConcurrentLinkedQueue

- Simplest lock-free algorithm
- Use CAS to update the next pointer into a linked list
- if CAS fails, means concurrent update happened
- Read new value, go to next item and retry CAS

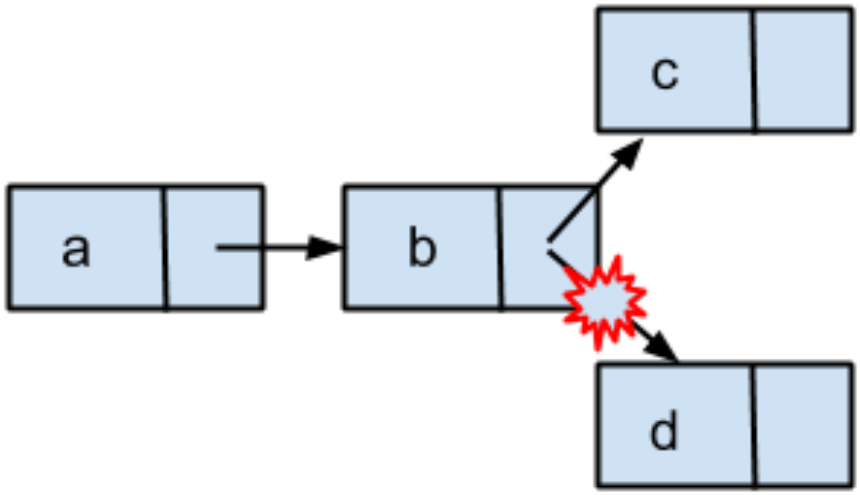
Compare-And-Swap: ConcurrentLinkedQueue



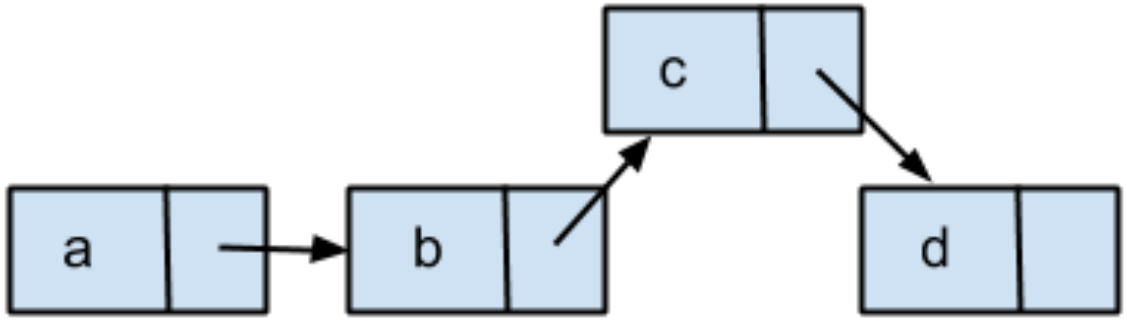
Compare-And-Swap: ConcurrentLinkedQueue



Compare-And-Swap: ConcurrentLinkedQueue



Compare-And-Swap: ConcurrentLinkedQueue



Java Memory Model (introduction)

Memory Model

- First language having a well defined memory model:
  Java JDK 5 (2004) with JSR 133
- C++ get a standard Memory Model in 2011 (C++11)
- Before that, some constructions may have undefined/different behavior on different platform (Double Check Locking)

Memory ordering

```
int a;
int b;
boolean enabled;
```

```
{
    a = 21;
    b = a * 2;
    enabled = true;
}
```

JIT Compiler

```
{
    enabled = true;
    a = 21;
    b = a * 2;
}
```

Memory ordering

```
int a;
int b;
boolean enabled;
```

Thread 1

```
{
    a = 21;
    b = a * 2;
    enabled = true;
}
```

Thread 2

```
{
    if (enabled)
    {
        int answer = b
        process(answer);
    }
}
```

Memory ordering

```
int a;
int b;
volatile boolean enabled;
```

Thread 1

```
{
    a = 21;
    b = a * 2;
    enabled = true;
}
```

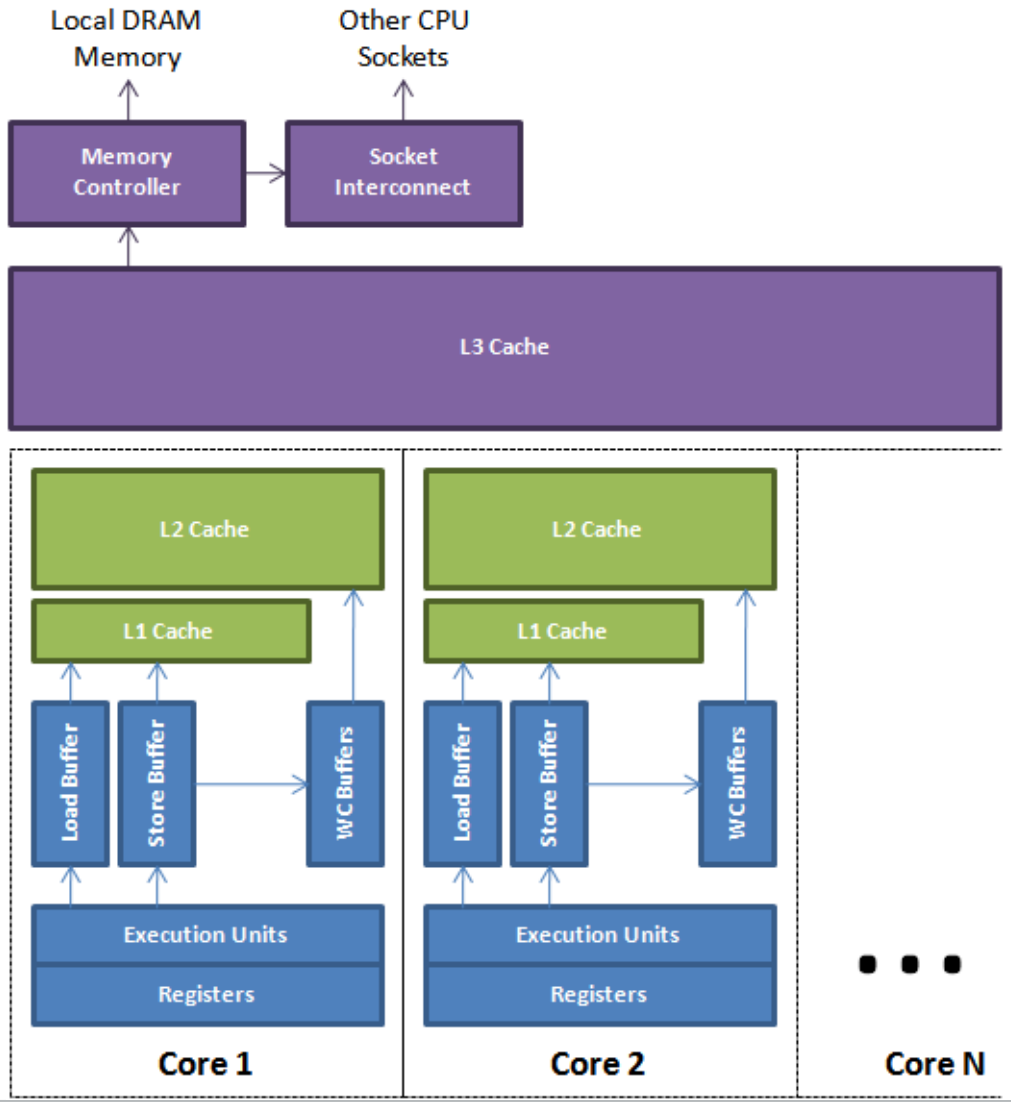
Thread 2

```
{
    if (enabled)
    {
        int answer = b
        process(answer);
    }
}
```

Memory barriers

- Can be at 2 levels: Compiler & Hardware
- Depending on CPU architecture, barrier is not required
- on x86: Strong model, limited reordering

Memory barriers



Memory barriers: volatile

- volatile field implies memory barrier
- Compiler barrier: prevent reordering
- Hardware barrier: Ensure drain of the memory buffers
- on X86, only store barrier emits an hardware one

```
lock add DWORD PTR [rsp],0x0
```

Memory barriers: CAS

- CAS is also a memory barrier
- Compiler: recognized by JIT to prevent reordering
- Hardware: all lock instructions is a memory barrier

Memory barriers: synchronized

- Synchronized blocks have implicit memory barriers
- Entering block: Load memory barrier
- Exiting block: store memory barrier

Memory barriers: synchronized

```
synchronized (this)
{
    enabled = true;
    b = 21;
    a = b * 2;
}
```

Memory barriers: lazySet

- method from AtomicXXX classes
- Compiler only memory barrier
- Does not emit hardware store barrier
- Still guarantee non reordering (most important) but not immediate effect for other thread

Disruptor & Ring Buffer

Disruptor

- LMAX library (incl. Martin Thompson)
- Not a new idea, circular buffers in Linux Kernel, Lamport
- Ported to Java

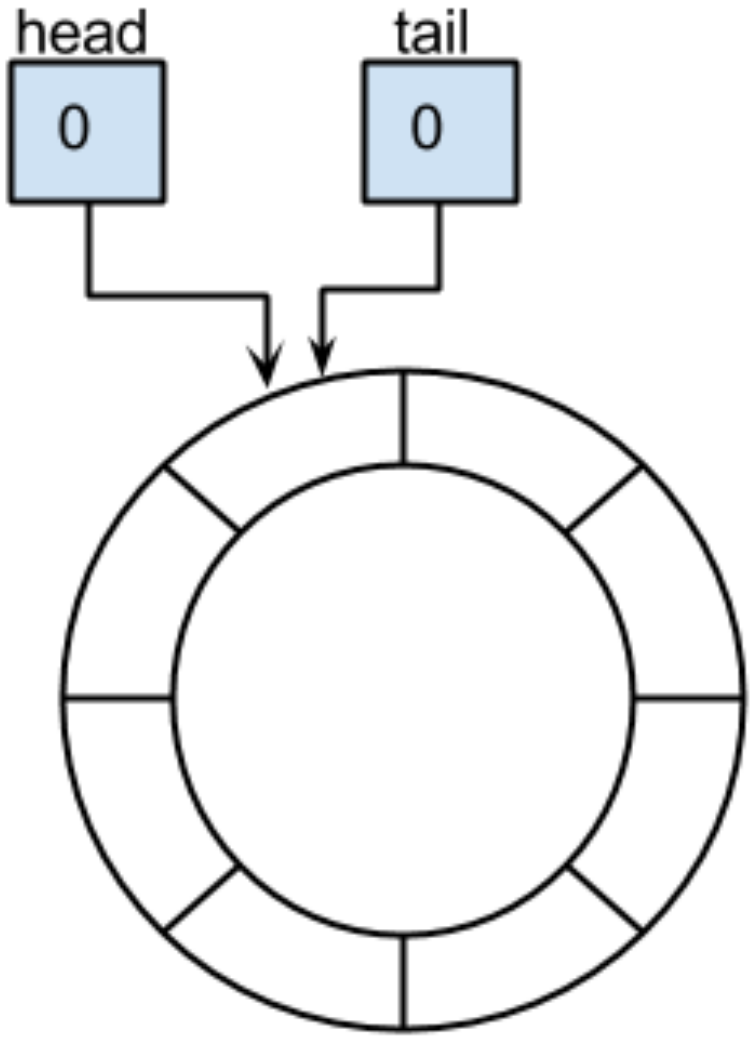
Disruptor

Why not used CLQ which is lock(wait)-free?

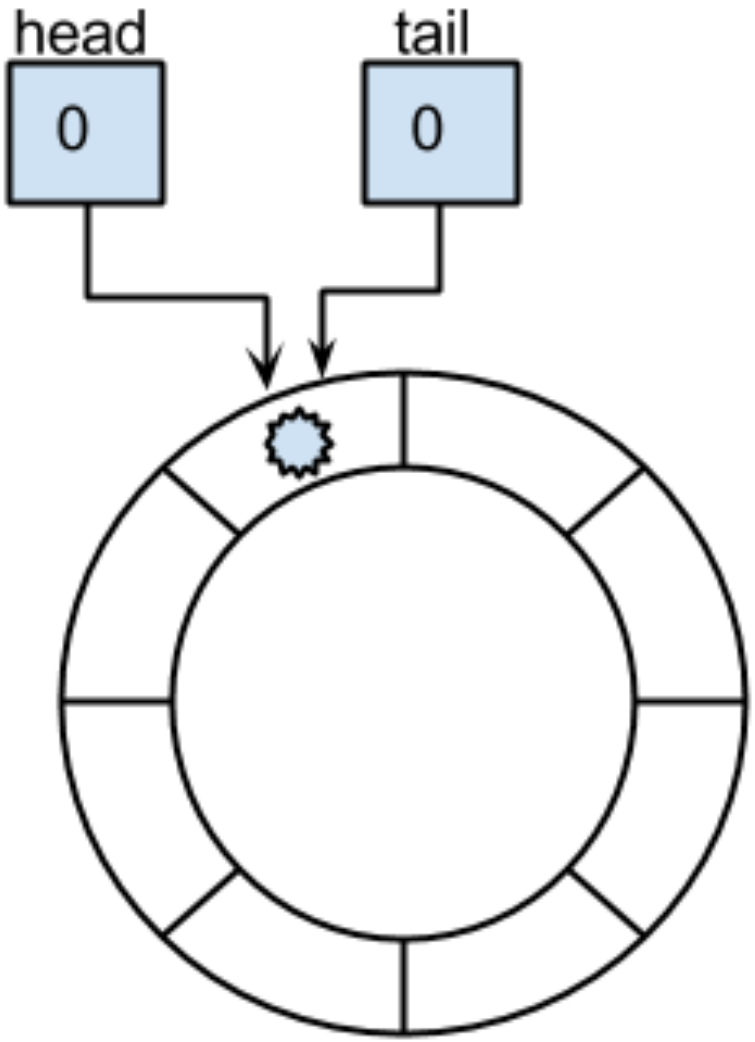
- Queue unbounded et non blocking
- Allocate a node at each insertion
- Not CPU cache friendly
- MultiProducer and MultiConsumer

Array/LinkedBlockingQueue: Not lock-free

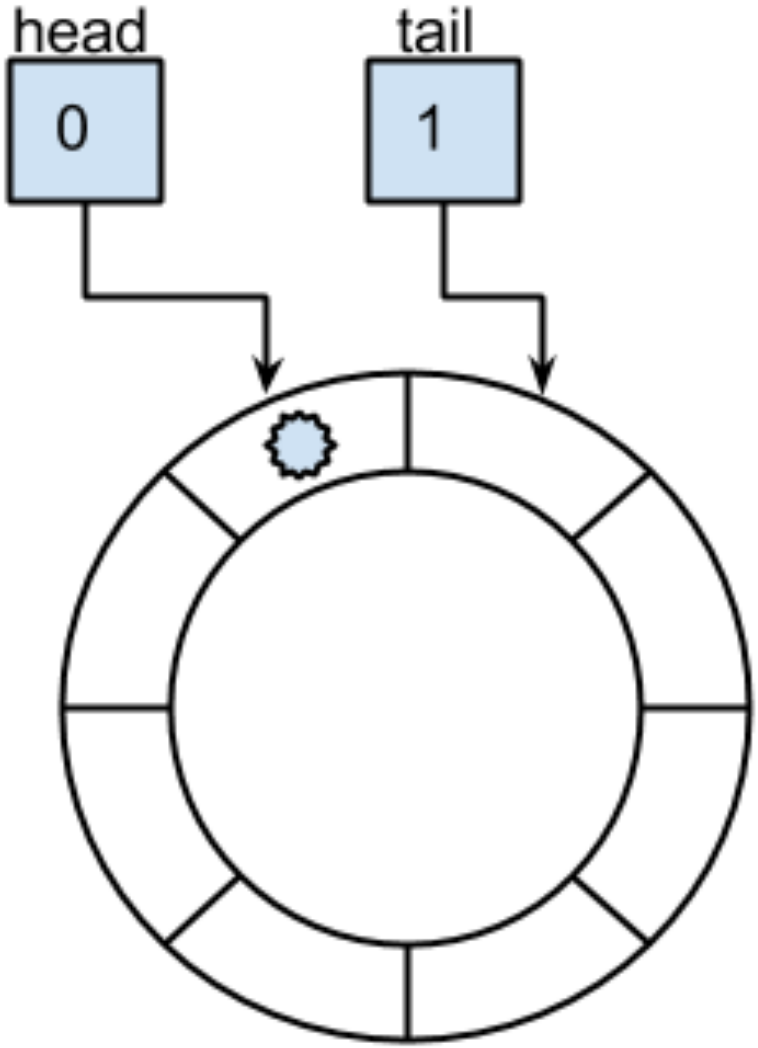
Ring Buffer: 1P 1C



Ring Buffer: 1P 1C



Ring Buffer: 1P 1C



Ring Buffer: 1P 1C

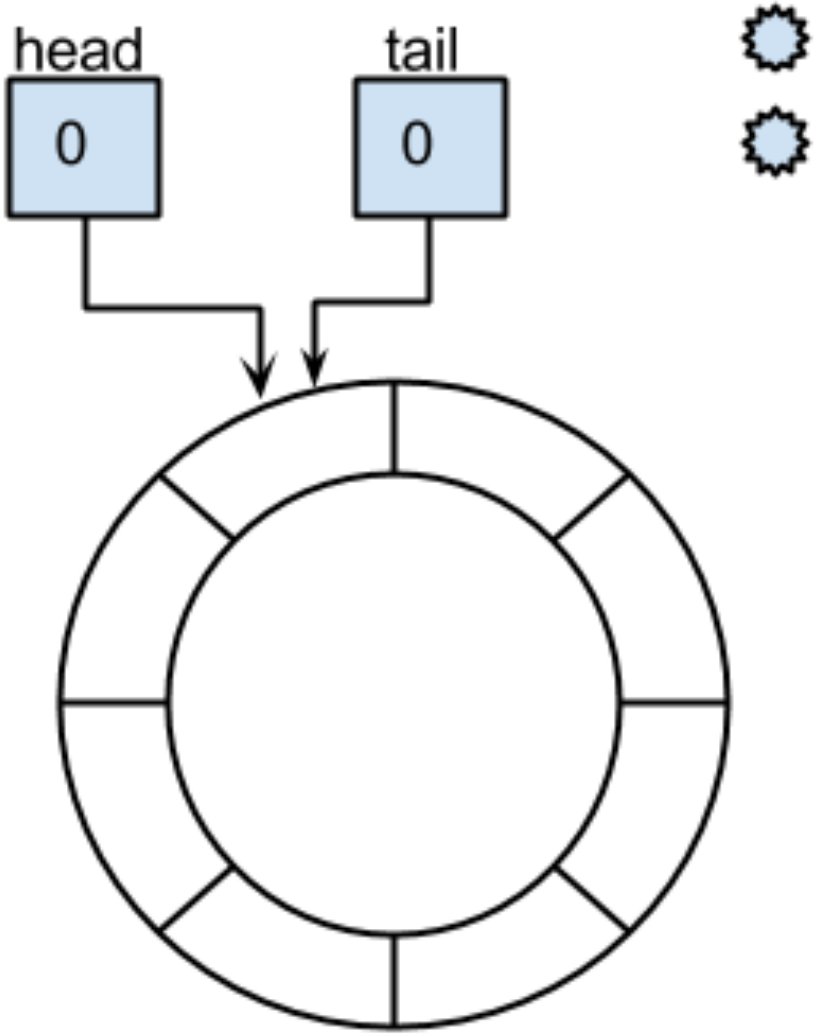
```
Object[] ringBuffer;
volatile int head;
volatile int tail;

public boolean offer(E e) {
    if (tail - head == ringBuffer.length)
        return false;
    ringBuffer[tail % ringBuffer.length] = e;
    tail++; // volatile write
    return true;
}
```

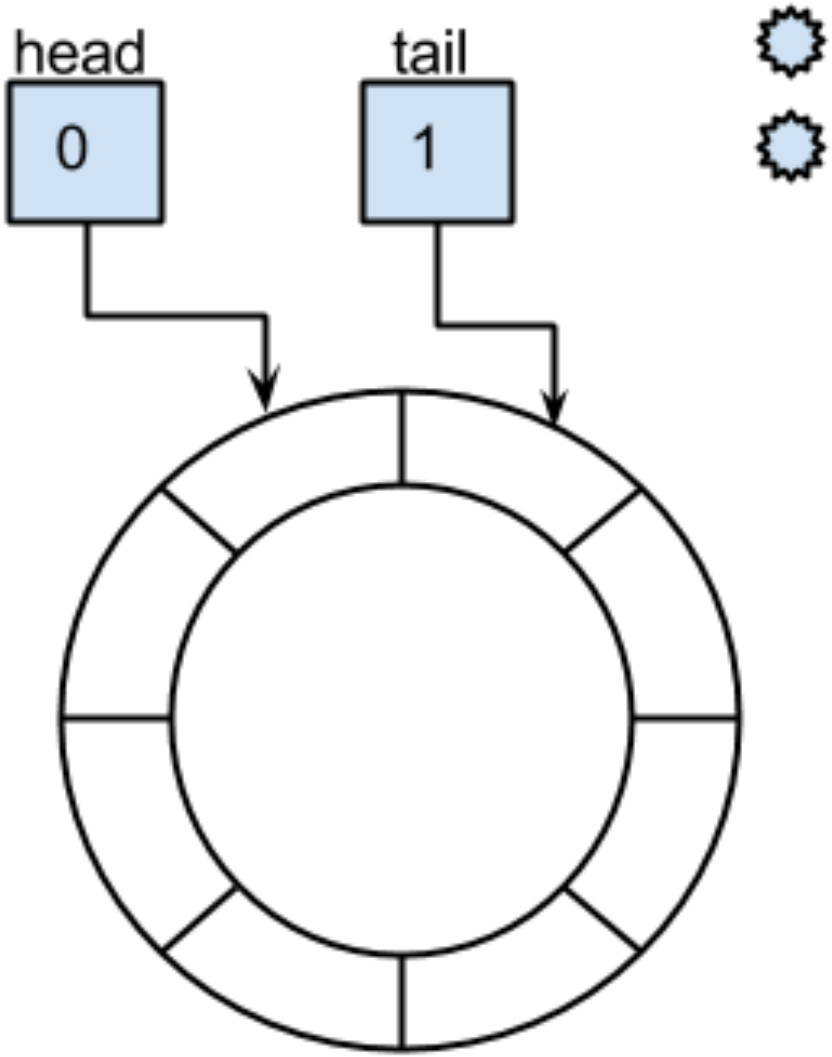
Ring Buffer: 1P 1C

```
public E poll() {
    if (tail == head)
        return null;
    int idx = head % ringBuffer.length
    E element = ringBuffer[idx];
    ringBuffer[idx] = null;
    head++; // volatile write
    return element;
}
```

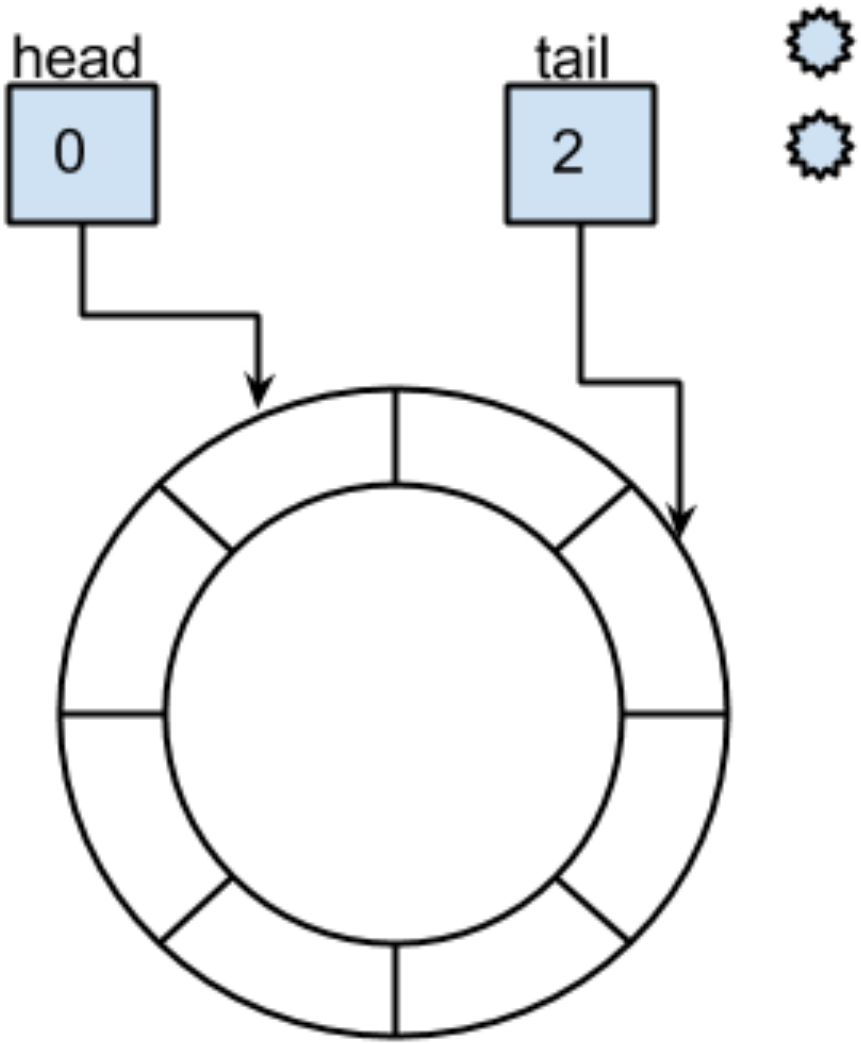
Ring Buffer: nP 1C



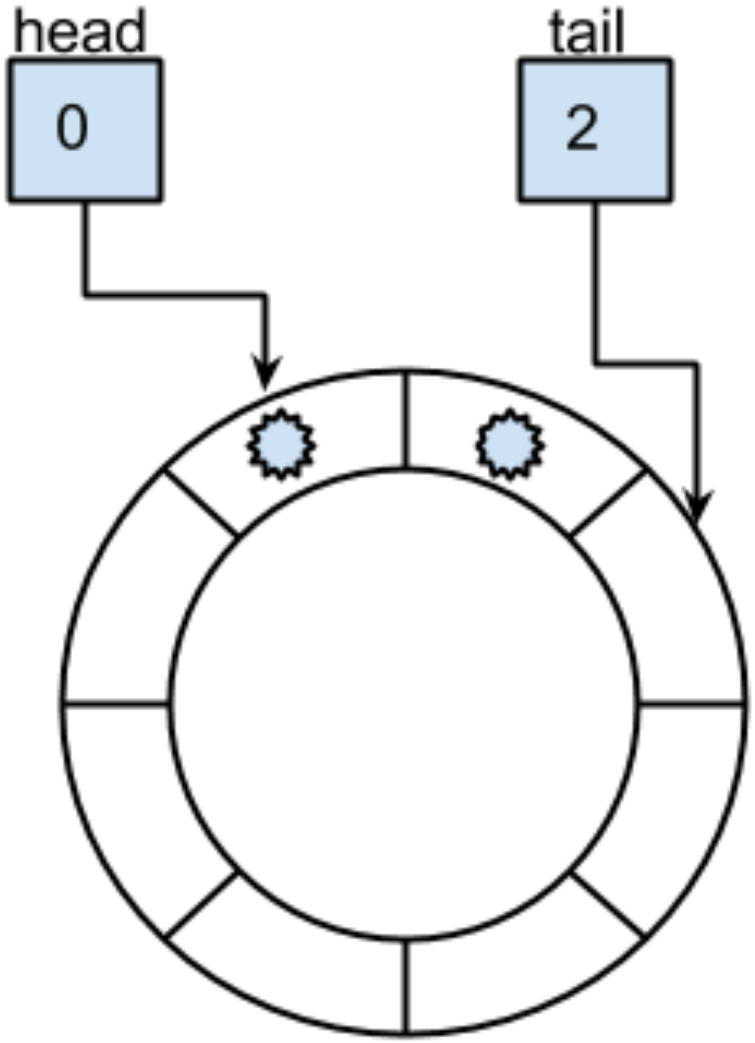
Ring Buffer: nP 1C



Ring Buffer: nP 1C



Ring Buffer: nP 1C



Ring Buffer: nP 1C

```
AtomicReferenceArray ringBuffer;
volatile long head;
AtomicLong tail;
```

Ring Buffer: nP 1C

```
public boolean offer(E e) {
    long curTail;
    do {
        curTail = tail.get();
        if (curTail - head == ringBuffer.length())
            return false;
    } while (!tail.compareAndSet(curTail, curTail+1));
    int idx = curTail % ringBuffer.length();
    ringBuffer.set(idx, e); // volatile write
    return true;
}
```

Ring Buffer: nP 1C

```
public E poll() {
    int index = head % ringBuffer.length();
    E element = ringBuffer.get(index);
    if (element == null)
        return null;
    ringBuffer.set(index, null);
    head++; // volatile write
    return element;
}
```

Disruptor

- Very flexible for different usages (strategies)
- Very good performance
- Data transfer from one thread to another (Queue)

Spinning

spinning

- Active wait
- very good for consumer reactivity
- Burns a cpu permanently

spinning

- Some locks are implemented with spinning (spinLock)
- Synchronized blocks spin a little bit on contention
- use of the pause instruction (x86)

spinning

How to avoid burning a core ?

Backoff strategies:

- Thread.yield()
- LockSupport.parkNanos(1)
- Object.wait()/Condition.await()/LockSupport.park()

Ticketing: OrderedScheduler

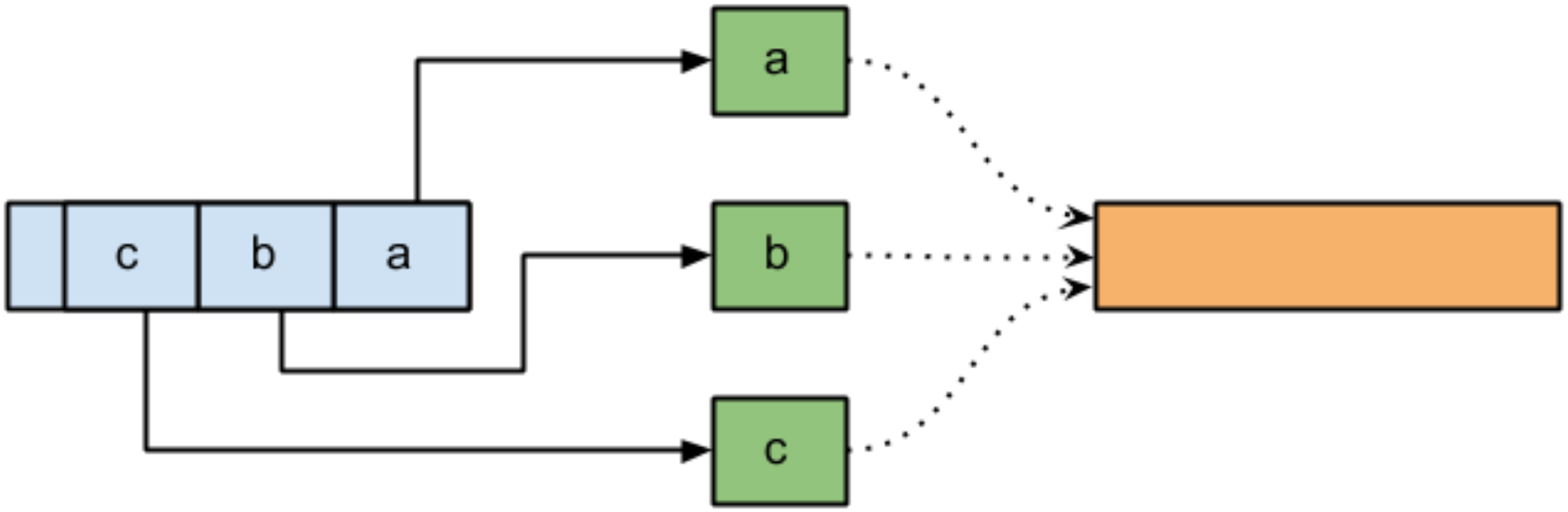
Ticketing

How to parallelize tasks while keeping ordering ?

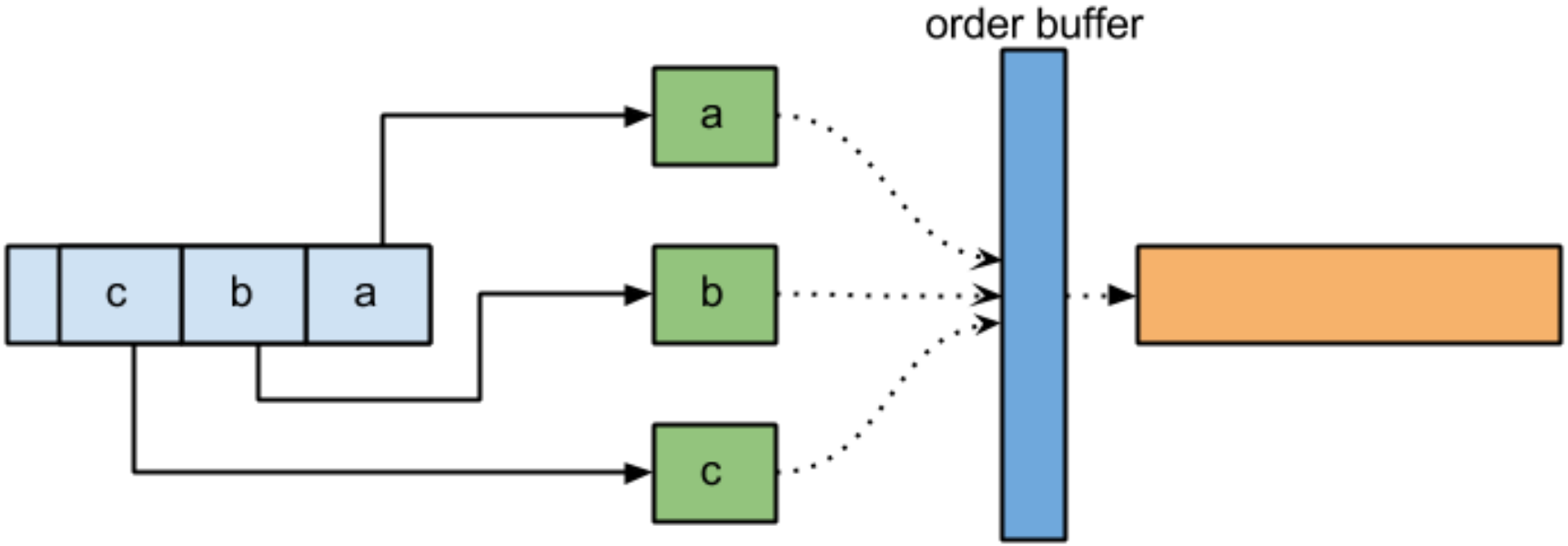
Example: Video stream processing

- read frame from the stream
- processing of the frame (parallelisable)
- writing into the output (in order)

Ticketing



Ticketing



Ticketing

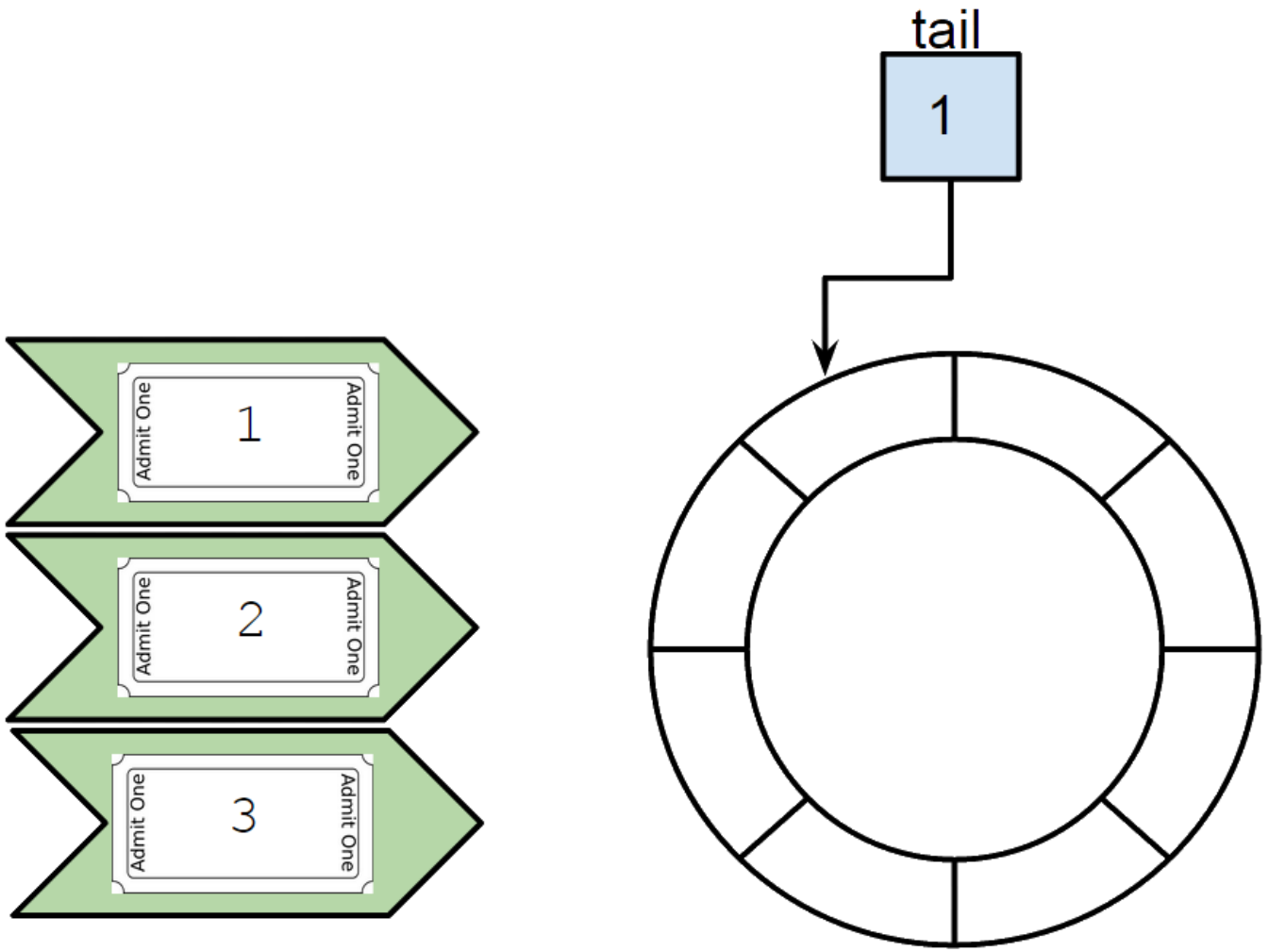
Can do this with Disruptor, but with a consumer thread

OrderedScheduler can do the same but:

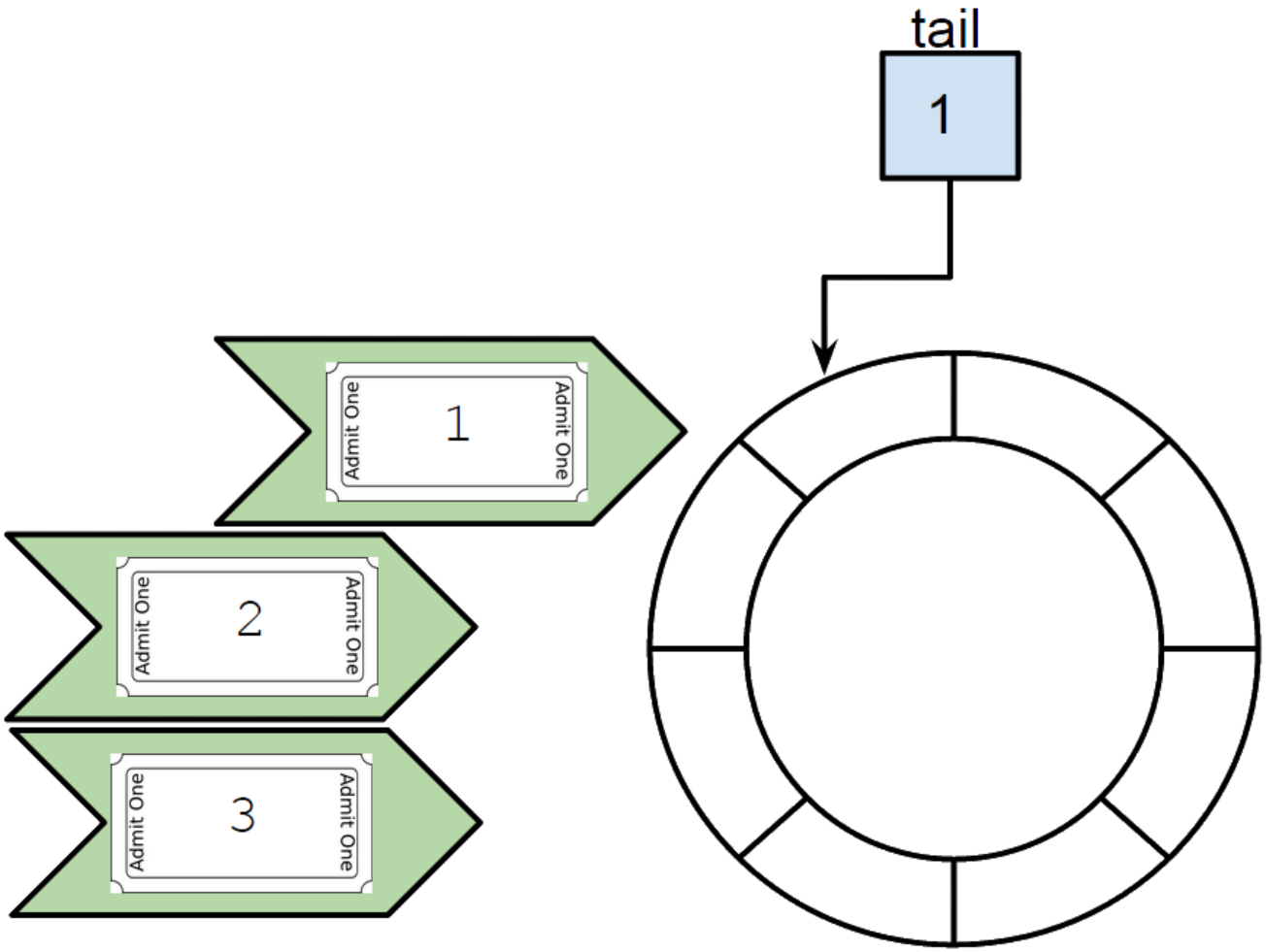
- no inter-thread communication overhead
- no additional thread
- no wait strategy

Take a ticket...

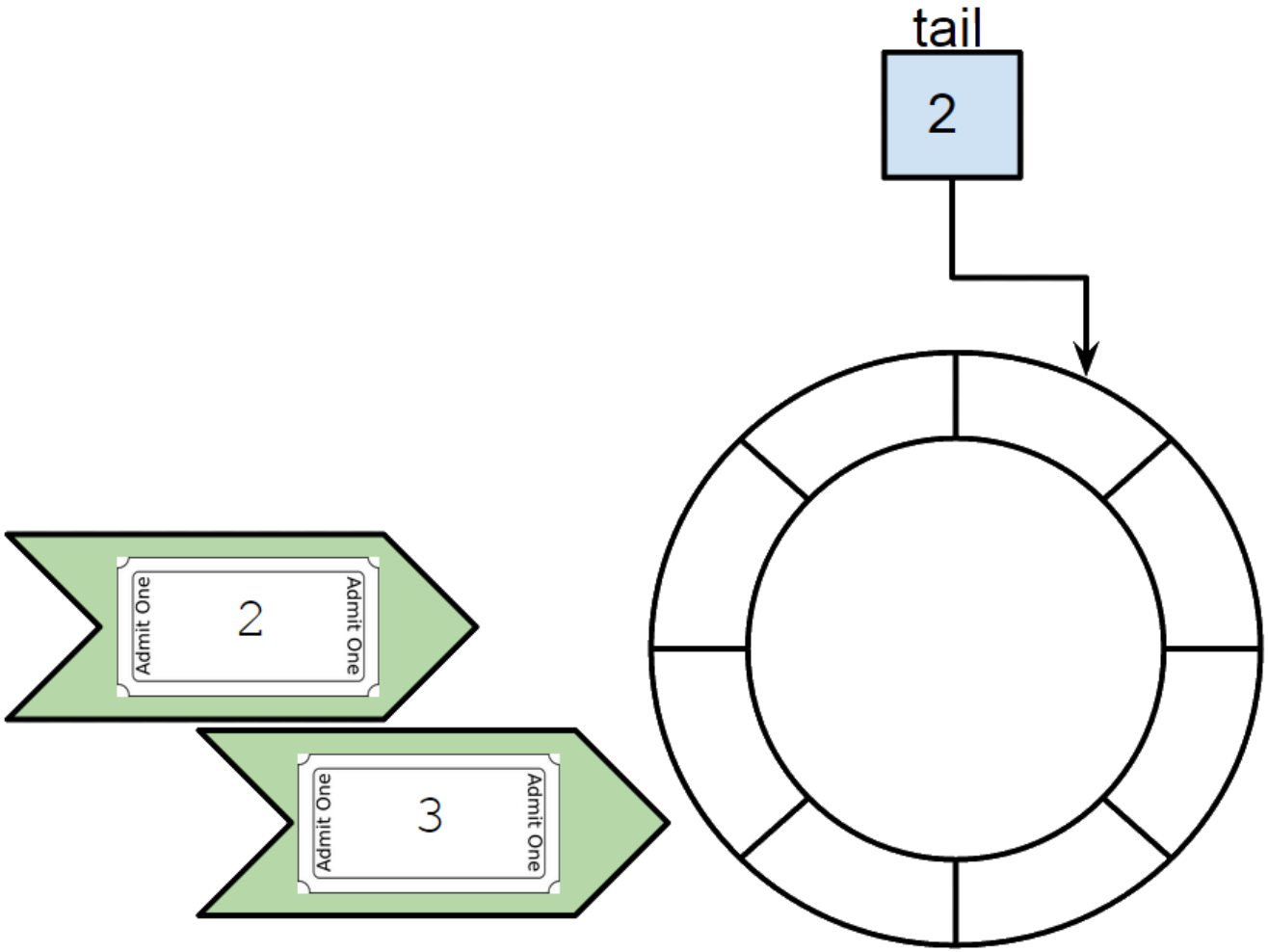
OrderedScheduler



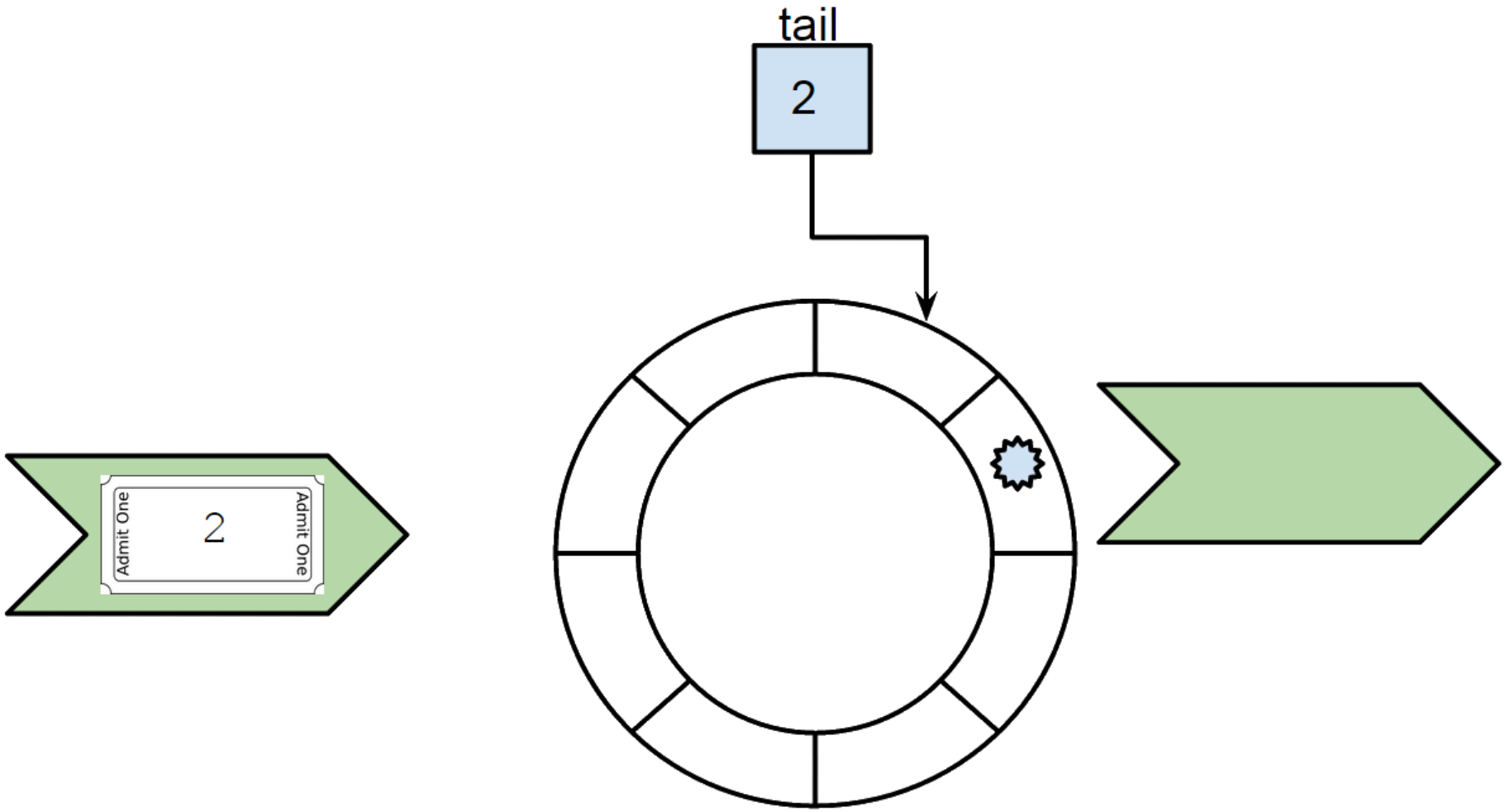
OrderedScheduler



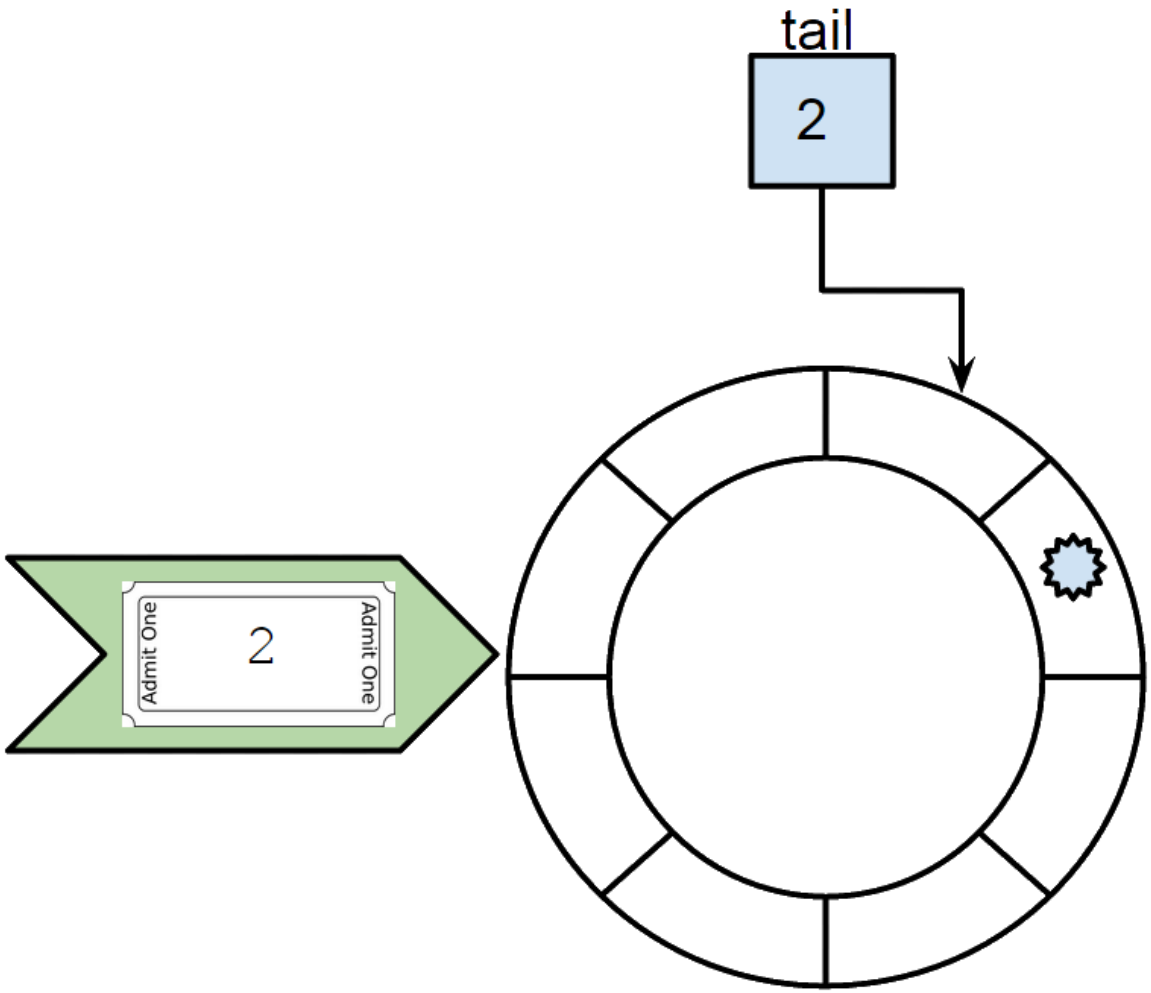
OrderedScheduler



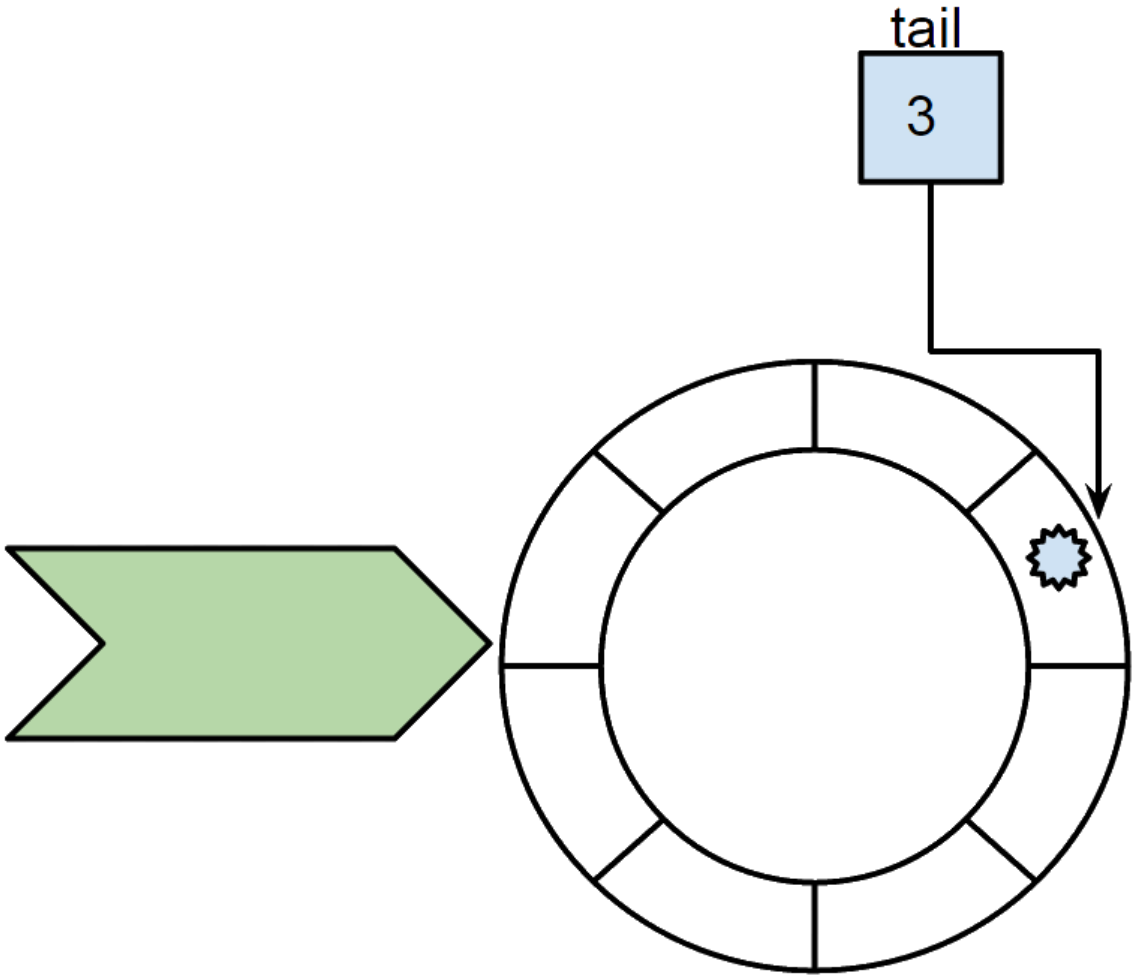
OrderedScheduler



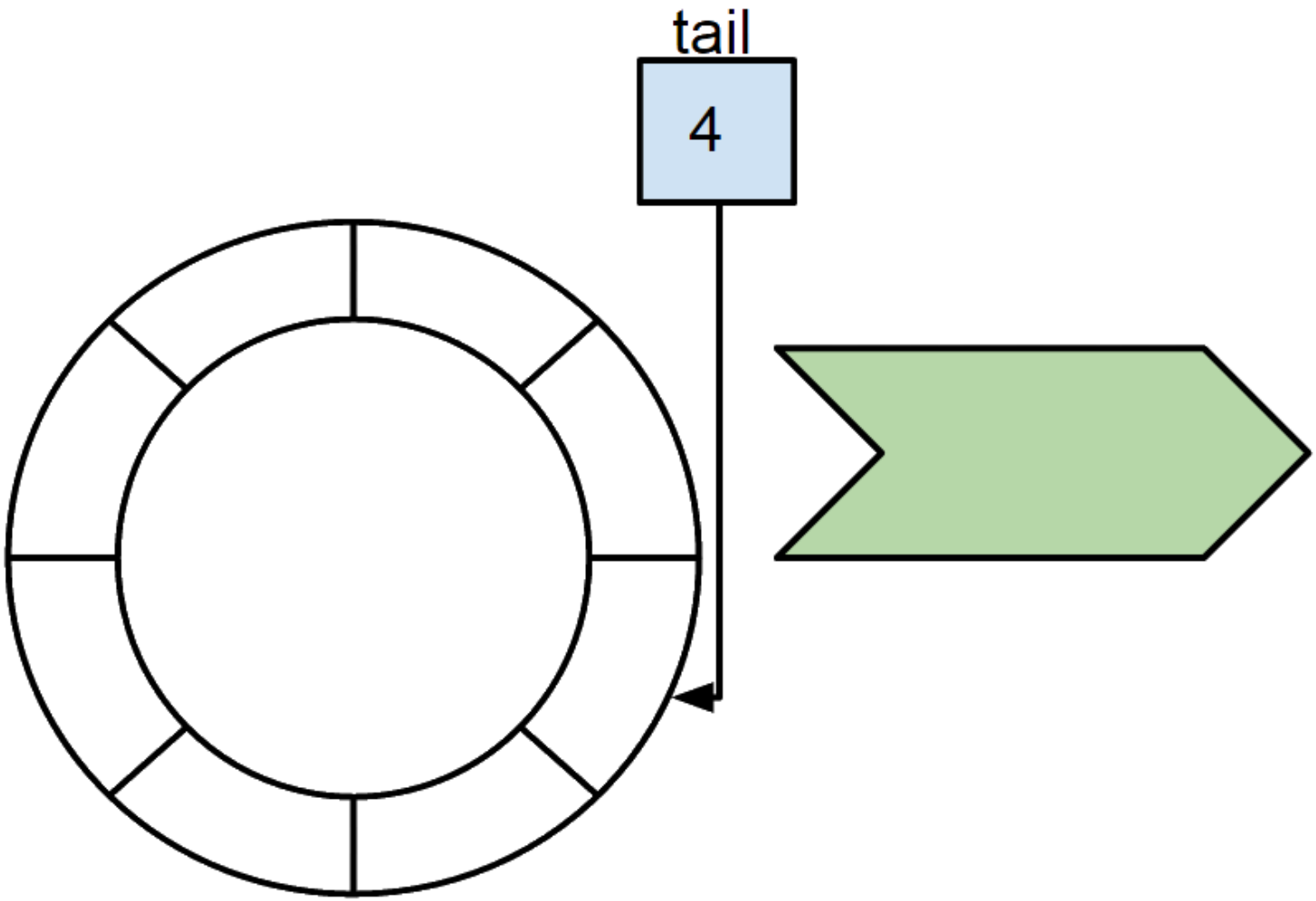
OrderedScheduler



OrderedScheduler



OrderedScheduler



OrderedScheduler

```
public void execute() {
    synchronized (this) {
        FooInput input = read();
        BarOutput output = process(input);
        write(output);
    }
}
```

OrderedScheduler

```
OrderedScheduler scheduler = new OrderedScheduler();

public void execute() {
    FooInput input;
    long ticket;
    synchronized (this) {
        input = read();
        ticket = scheduler.getNextTicket();
    }
[...]
```

OrderedScheduler

```
public void execute(){
[...]
    BarOutput output;
    try {
        output = process(intput);
    }
    catch (Exception ex) {
        scheduler.trash(ticket);
        throw new RuntimeException(ex);
    }
    scheduler.run(ticket, { () => write(output); });
}
```

Ticketing

- Open sourced on GitHub
- Opened to PR & discussion on the design
- Used internally

Takeaways

Takeaways

- Measure
- Distribute
- Atomically update
- Order
- RingBuffer: easy lock-free
- Ordered without consumer thread

References

- jucProfiler: http://www.infoq.com/articles/jucprofiler
- Java Memory Model Pragmatics: http://shipilev.net/blog/2014/jmm-pragmatics/
- Memory Barriers and JVM Concurrency: http://www.infoq.com/articles/memory_barriers_jvm_concurrency
- JSR 133 (FAQ): http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html
- CPU cache flushing fallacy: http://mechanical-sympathy.blogspot.fr/2013/02/cpu-cache-flushing-fallacy.html
- atomic<> Weapons: http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2

References

- circular buffers: http://lwn.net/Articles/378262/
- Mr T queues: https://github.com/mjpt777/examples/tree/master/src/java/uk/co/real_logic/queues
- Lock-free algorithms by Mr T: http://www.infoq.com/presentations/Lock-free-Algorithms
- Futex are tricky U. Drepper: http://www.akkadia.org/drepper/futex.pdf
- JCTools: https://github.com/JCTools/JCTools
- Nitsan Wakart blog: http://psy-lob-saw.blogspot.com

References

- Exploiting data parallelism in ordered data streams: https://software.intel.com/en-us/articles/exploiting-data-parallelism-in-ordered-data-streams
- OrderedScheduler: https://github.com/Ullink/ordered-scheduler
- Concurrency Freaks: http://concurrencyfreaks.blogspot.com
- Preshing on programming: http://preshing.com/
- Is Parallel Programming Hard, And If So, What Can You Do About It? https://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2015.01.31a.pdf

ABOUT ULLINK

ULLINK's electronic financial trading software solutions and services address the challenges of new regulations and increased fragmentation. Emerging markets require fast, flexible and compliant solutions for buy-side and sell-side market participants; providing a competitive advantage for both low touch and high touch trading.

www.ullink.com

FIND OUT MORE

Contact our Sales Team to learn more about the services listed herein as well as our full array of offerings:

+81 3 3664 4160 (Japan)

+852 2521 5400 (Asia Pacific)

+1 212 991 0816 (North America)

+55 11 3171 2409 (Latin America)

+44 20 7488 1655 (UK and EMEA)

connect@ullink.com