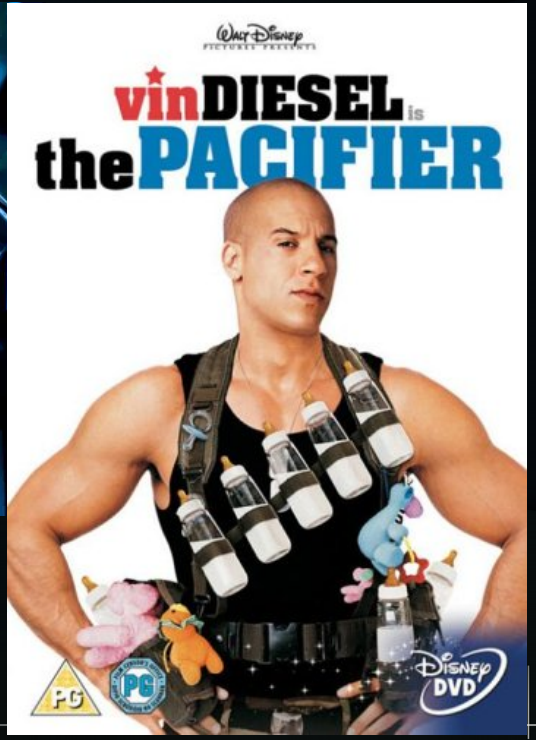
JCTools: Pit Of Performance

Nitsan Wakart/@nitsanw

You Look Familiar...



Why me?

- Main JCTools contributor
- Performance Eng.
- Find me on:
  - GitHub : nitsanw
  - Blog : psy-lob-saw.blogspot.com
  - Twitter : nitsanw
- Also: Cape Town JUG Organizer

JCTools?

- Concurrent Queues:
  - SPSC/MPSC/SPMC/MPMC
  - Linked/Array/LinkedArray/Compound
- And MOAR!!!

What is this about?

- Optimizations as found in the code
- Design pressures/choices
- Novel algorithm: MPSC linked queues

From the codebase...

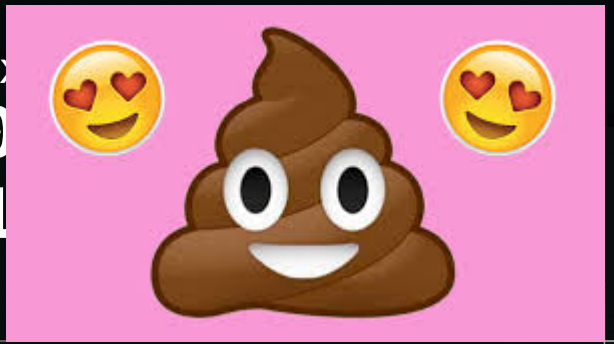
- Layout
- Unsafe
- Concurrency specialization
- Bits and bobs

Layout

```
abstract class ...L1Pad<E> extends ...ColdField<E> {
    long p00,p01,p02,p03,p04,p05,p06,p07,
         p08,p09,p10,p11,p12,p13,p14;
}

abstract class ...ProducerIndexFields<E> extends ...L1Pad<E> {
    private volatile long producerIndex;
    protected long producerLimit;
}

abstract class ...L2Pad<E> extends ...ProducerIndexFields<E> {
    long p00,p01,p02,p03,p04,p05,p06,p07,
         p08,p09,p10,p11,p12,p13,p14;
}
```

Layout

```
abstract class ...L1Pad<E> extends ...ColdField<E> {
    long p00,p01,p02,p03,p04,p05,p06,p07,
         p08,p09,p10,p11,p12,p13,p14;
}

abstract class ...ProducerIndexFields<E> extends ...L1Pad<E> {
    private volatile long producerIndex;
    protected long producerLimit;
}

abstract class ...L2Pad<E> extends ...ProducerIndex
    long p00,p01,p02,p03,p04,p0
         p08,p09,p10,p11,p12,p1
}
```



Measure with/with out:

```
With padding:  ~950 ns/op
Sans padding: ~5000 ns/op
```

*QueueBurst, 100 messages benchmark

Java Object Layout

- 8b aligned
- Object header (8/12/16b)
- **Sub-classes fields after parent fields**
- Field order within class can change

Let’s use JOL!!!

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# WARNING | Compressed references base/shifts are guessed by the experiment!
# WARNING | Therefore, computed addresses are just guesses, and ARE NOT RELIABLE.
# WARNING | Make sure to attach Serviceability Agent to get the reliable addresses.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

Instantiated the sample instance via public org.jctools.queues.FFBuffer(int)
org.jctools.queues.SpscArrayQueue object internals:
 OFFSET  SIZE               TYPE DESCRIPTION                               VALUE
      0     4                    (object header)                           01 00 00 00 (00000001 00000000 00000000 00000000) (1)
      4     4                    (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
      8     4                    (object header)                           37 28 01 f8 (00110111 00101000 00000001 11111000) (-134141897)
     12     4                    (alignment/padding gap)
     16     8                long ConcurrentCircularArrayQueueL0Pad.p00     0
     24     8                long ConcurrentCircularArrayQueueL0Pad.p01     0
     32     8                long ConcurrentCircularArrayQueueL0Pad.p02     0
     40     8                long ConcurrentCircularArrayQueueL0Pad.p03     0
     48     8                long ConcurrentCircularArrayQueueL0Pad.p04     0
     56     8                long ConcurrentCircularArrayQueueL0Pad.p05     0
     64     8                long ConcurrentCircularArrayQueueL0Pad.p06     0
     72     8                long ConcurrentCircularArrayQueueL0Pad.p07     0
     80     8                long ConcurrentCircularArrayQueueL0Pad.p08     0
     88     8                long ConcurrentCircularArrayQueueL0Pad.p09     0
     96     8                long ConcurrentCircularArrayQueueL0Pad.p10     0
    104     8                long ConcurrentCircularArrayQueueL0Pad.p11     0
    112     8                long ConcurrentCircularArrayQueueL0Pad.p12     0
    120     8                long ConcurrentCircularArrayQueueL0Pad.p13     0
    128     8                long ConcurrentCircularArrayQueueL0Pad.p14     0
    136     8                long ConcurrentCircularArrayQueue.mask         3
    144     4  java.lang.Object[] ConcurrentCircularArrayQueue.buffer       [null, null, null, null]
    148     4                 int SpscArrayQueueColdField.lookAheadStep     1
    152     8                long SpscArrayQueueL1Pad.p00                   0
    160     8                long SpscArrayQueueL1Pad.p01                   0
    168     8                long SpscArrayQueueL1Pad.p02                   0
    176     8                long SpscArrayQueueL1Pad.p03                   0
    184     8                long SpscArrayQueueL1Pad.p04                   0
    192     8                long SpscArrayQueueL1Pad.p05                   0
    200     8                long SpscArrayQueueL1Pad.p06                   0
    208     8                long SpscArrayQueueL1Pad.p07                   0
    216     8                long SpscArrayQueueL1Pad.p08                   0
    224     8                long SpscArrayQueueL1Pad.p09                   0
    232     8                long SpscArrayQueueL1Pad.p10                   0
    240     8                long SpscArrayQueueL1Pad.p11                   0
    248     8                long SpscArrayQueueL1Pad.p12                   0
    256     8                long SpscArrayQueueL1Pad.p13                   0
    264     8                long SpscArrayQueueL1Pad.p14                   0
    272     8                long SpscArrayQueueProducerIndexFields.producerIndex   0
    280     8                long SpscArrayQueueProducerIndexFields.producerLimit   0
    288     8                long SpscArrayQueueL2Pad.p00                   0
    296     8                long SpscArrayQueueL2Pad.p01                   0
    304     8                long SpscArrayQueueL2Pad.p02                   0
    312     8                long SpscArrayQueueL2Pad.p03                   0
    320     8                long SpscArrayQueueL2Pad.p04                   0
    328     8                long SpscArrayQueueL2Pad.p05                   0
    336     8                long SpscArrayQueueL2Pad.p06                   0
    344     8                long SpscArrayQueueL2Pad.p07                   0
    352     8                long SpscArrayQueueL2Pad.p08                   0
    360     8                long SpscArrayQueueL2Pad.p09                   0
    368     8                long SpscArrayQueueL2Pad.p10                   0
    376     8                long SpscArrayQueueL2Pad.p11                   0
    384     8                long SpscArrayQueueL2Pad.p12                   0
    392     8                long SpscArrayQueueL2Pad.p13                   0
    400     8                long SpscArrayQueueL2Pad.p14                   0
    408     8                long SpscArrayQueueConsumerIndexField.consumerIndex   0
    416     8                long SpscArrayQueueL3Pad.p00                   0
    424     8                long SpscArrayQueueL3Pad.p01                   0
    432     8                long SpscArrayQueueL3Pad.p02                   0
    440     8                long SpscArrayQueueL3Pad.p03                   0
    448     8                long SpscArrayQueueL3Pad.p04                   0
    456     8                long SpscArrayQueueL3Pad.p05                   0
    464     8                long SpscArrayQueueL3Pad.p06                   0
    472     8                long SpscArrayQueueL3Pad.p07                   0
    480     8                long SpscArrayQueueL3Pad.p08                   0
    488     8                long SpscArrayQueueL3Pad.p09                   0
    496     8                long SpscArrayQueueL3Pad.p10                   0
    504     8                long SpscArrayQueueL3Pad.p11                   0
    512     8                long SpscArrayQueueL3Pad.p12                   0
    520     8                long SpscArrayQueueL3Pad.p13                   0
    528     8                long SpscArrayQueueL3Pad.p14                   0
Instance size: 536 bytes
```

Let’s use JOL!!!

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
```

Let’s use JOL!!!

```
org.jctools.queues.SpscArrayQueue object internals:
 OFFSET  SIZE                TYPE DESCRIPTION
      0     12                     (object header)
     12     4                      (alignment/padding gap)
     16     120                long ConcurrentCircularArrayQueueL0Pad.p00 [... p14]
    136     8                  long ConcurrentCircularArrayQueue.mask
    144     4   java.lang.Object[] ConcurrentCircularArrayQueue.buffer
    148     4                   int SpscArrayQueueColdField.lookAheadStep
    152     120                long SpscArrayQueueL1Pad.p00 [... p14]
    272     8                  long SpscArrayQueueProducerIndexFields.producerIndex
    280     8                  long SpscArrayQueueProducerIndexFields.producerLimit
    288     120                long SpscArrayQueueL2Pad.p00 [... p14]
    408     8                  long SpscArrayQueueConsumerIndexField.consumerIndex
    416     120                long SpscArrayQueueL3Pad.p00 [... p14]

Instance size: 536 bytes
```

Let's use JOL!!!

```
org.jctools.queues.SpscArrayQueue object internals:
 OFFSET  SIZE               TYPE DESCRIPTION
      0    12                    (object header)
     12     4                    (alignment/padding gap)
     16   120                    long ConcurrentCircularArrayQueueL0Pad.p00 [... p14]
    136     8                    long ConcurrentCircularArrayQueue.mask
    144     4   java.lang.Object[] ConcurrentCircularArrayQueue.buffer
    148     4                     int SpscArrayQueueColdField.lookAheadStep
    152   120                    long SpscArrayQueueL1Pad.p00 [... p14]
    272     8                    long SpscArrayQueueProducerIndexFields.producerIndex
    280     8                    long SpscArrayQueueProducerIndexFields.producerLimit
    288   120                    long SpscArrayQueueL2Pad.p00 [... p14]
    408     8                    long SpscArrayQueueConsumerIndexField.consumerIndex
    416   120                    long SpscArrayQueueL3Pad.p00 [... p14]

Instance size: 536 bytes
```

Let’s use JOL!!!

```
org.jctools.queues.SpscArrayQueue object internals:
 OFFSET  SIZE                    TYPE DESCRIPTION
      0     12                         (object header)
     12     4                          (alignment/padding gap)
     16    120  long ConcurrentQueueL0Pad.p00 [... p14]
    136     8                    long ConcurrentCircularArrayQueue.mask
    144     4   java.lang.Object[] ConcurrentCircularArrayQueue.buffer
    148     4                     int SpscArrayQueueColdField.lookAheadStep
    152    120    long SpscArrayQueueL1Pad.p00 [... p14]
    272     8                    long SpscArrayQueueProducerIndexFields.producerIndex
    280     8                    long SpscArrayQueueProducerIndexFields.producerLimit
    288    120    long SpscArrayQueueL2Pad.p00 [... p14]
    408     8                    long SpscArrayQueueConsumerIndexField.consumerIndex
    416    120    long SpscArrayQueueL3Pad.p00 [... p14]

Instance size: 536 bytes
```

Let’s use JOL!!!

```
org.jctools.queues.SpscArrayQueue object internals:
 OFFSET  SIZE                TYPE DESCRIPTION
      0     12                     (object header)
     12     4                      (alignment/padding gap)
     16     120                long ConcurrentCircularArrayQueueL0Pad.p00 [... p14]
    136     8                  long ConcurrentCircularArrayQueue.mask
    144     4   java.lang.Object[] ConcurrentCircularArrayQueue.buffer
    148     4                   int SpscArrayQueueColdField.lookAheadStep
    152     120                long SpscArrayQueueL1Pad.p00 [... p14]
    272     8                  long SpscArrayQueueProducerIndexFields.producerIndex
    280     8                  long SpscArrayQueueProducerIndexFields.producerLimit
    288     120                long SpscArrayQueueL2Pad.p00 [... p14]
    408     8                  long SpscArrayQueueConsumerIndexField.consumerIndex
    416     120                long SpscArrayQueueL3Pad.p00 [... p14]
```

Instance size: 536 bytes

Why bother with layout?

- False sharing
- Minimize load misses
- Offheap?
  - Atomicity
  - Alignment requirements (line/page/sector)
  - Different method (see Aeron)

Sharing?

header	header	pIndex	cIndex	mask	buffer		

False Sharing

header	header	**pIndex**	**cIndex**	Mask	buffer		

False Sharing

header	header	**pIndex**	**cIndex**	Mask	buffer		

Pad to resolve...

???	header	header	pad12	pad13	pad14	pad15	pad16
pad17	pad20	pad21	pad22	pad23	pad24	pad25	pad26
pIndex	pMask	pBuffer	pad10	pad11	pad12	pad13	pad14
pad15	pad16	pad17	pad20	pad21	pad22	pad23	pad24
pad25	pad26	pad27	cIndex	cMask	cBuffer	pad10	pad11

Alternatives To Method?

- ~~Rely on field order~~
- @Contended (field/class level, -XX:-Restrict)
- Go Offheap

What would be nice?

```
@ForceLayout(align=64) // state alignment preference
Class X {
   long field1;   // still type aligned
   byte[120] pad; // allocate presized array inline
   long field2;
}
```

Unsafe

Looks safe enough...

```
abstract class ...ProducerIndexFields<E> extends SpscArrayQueueL1Pad<E>
{
    final static long P_INDEX_OFFSET =
       fieldOffset(...Fields.class, "producerIndex");
    private volatile long producerIndex;
    protected long producerLimit;

    long lvProducerIndex() {
        return producerIndex;
    }

    long lpProducerIndex() {
        return UNSAFE.getLong(this, P_INDEX_OFFSET);
    }

    void soProducerIndex(long newValue) {
        UNSAFE.putOrderedLong(this, P_INDEX_OFFSET, newValue);
    }
}
```

Why do we need Unsafe?

- Better than a poke in the eye with a sharp stick?
- Performance?
- Compatibility????!??!?!?!?

Unsafe for Compatibility?

- Just Works!™ (JDK6,7,8,11)
- Shouldn’t we use VarHandle?

Unsafe vs. A*FU (pre-8u101)

```
With Unsafe :  ~950 ns/op
With A*FU@7 : ~1350 ns/op
```

```
* Tested with JDK 7u80 as an example
```

Unsafe vs. A*FU (post 8u101/11)

```
With Unsafe  :  ~950 ns/op
With A*FU@8  : ~1050 ns/op
With A*FU@11 : ~1150 ns/op
```

See: https://shipilev.net/blog/2015/faster-atomic-fu/

Unsafe vs VarHandles (>=11)

```
With Unsafe       :  ~950 ns/op
With VarHandle    : ~1150 ns/op
```

Unsafe for performance?

```
With Unsafe@7,8,11 :  ~950 ns/op
With      A*FU@7   : ~1300 ns/op
With      A*FU@8   : ~1050 ns/op
With      A*FU@11  : ~1150 ns/op
With VarHandle@11  : ~1150 ns/op
```

Unsafe vs Alternatives

- Pre-JDK11?
  - Volatile
  - AtomicBoolean/Integer/Long/*Array
  - Atomic*FieldUpdater
- JDK11+?
  - Should move to VarHandle?
  - Multi-version jars
- Offheap? GO UNSAFE!!!

Concurrency Specialization

Generic vs. Specialized

- JDK?
  - MPMC
  - Lock based (ArrayBlockingQ/LinkedBlockingQ)
  - Lock-less (ConcurrentLinkedQ)
  - Allocation?
- JCTools?
  - MPMC/SPMC/MPSC/SPSC
  - Array backed/Linked/both!
  - Lock-less/lock-free/wait-free
  - Zero allocation options

Generic vs. Specialized

- Load: plain, opaque, volatile
- Store: plain, ordered, volatile
- compareAndSwap
- getAndAdd

Single Writer

- Load: plain/volatile
- Store: ordered

Multi-Writer

- Load: plain/volatile
- Store: getAndAdd/compareAndSwap

Generic vs. Specialized

```
SPSC –   950 ns/op
MPSC –  6500 ns/op
MPMC –  5500 ns/op
CLQ  – 13800 ns/op
ABQ  – 36000 ns/op
```

Lock less/Lock free/Wait free

```
SPSC –    950 ns/op        – WF
MPSC –  5000/6500 ns/op – LF/LL
MPMC –  5000/5500 ns/op – LF/LL
CLQ  – 13800 ns/op         – LL
ABQ  – 36000 ns/op         – Locks...
```

Allocations

CLQ – 2400b per op(100 messages)

LBQ – ~3000b

ABQ – ?

Allocations

CLQ – 2400b per op(100 messages)

LBQ – ~3000b

ABQ – ~1500b!!!

JCTools array backed - 0b

Helping the Compiler

Const vs Final

```
private final long size = <some power of 2>;

long offset = REF_ARRAY_BASE +
  ((consumerIndex % size) *
    REF_ELEMENT_SCALE);


long offset = REF_ARRAY_BASE +
  ((consumerIndex & (size-1)) *
  REF_ELEMENT_SCALE);
```

Const vs Final

```
long offset = REF_ARRAY_BASE +
  ((consumerIndex & (size-1)) *
     REF_ELEMENT_SCALE);
```

```
long offset = REF_ARRAY_BASE +
  ((consumerIndex & mask) <<
     REF_ELEMENT_SHIFT);
```

Const vs Final

```
long offset = REF_ARRAY_BASE +
   ((consumerIndex & mask) <<
     REF_ELEMENT_SHIFT);
```

```
long offset = calcElementOffset(consumerIndex, mask);
```

Loads Across an Volatile Load

```
public E poll()
{
    long consumerIndex = lpConsumerIndex();
    E[] buffer = this.buffer;
    long offset = calcElementOffset(consumerIndex, mask);

    E e = (E) UNSAFE.getObjectVolatile(buffer, offset);
    if (null == e)
        return null; // EMPTY

    UNSAFE.putOrderedObject(buffer, offset, null);

    soConsumerIndex(consumerIndex + 1);
    return e;
}
```

A Novel Algorithm

Typical Offerings

- Array backed
  - Pre-allocated, fixed sized
- Linked
  - Allocate cell per element
  - No pre-allocation
  - Bounded/Unbounded

Actor Use Case

- Lots of queues
- Some full & busy
- Most empty
- MPSC

Linked Array Queues?

- Middle Ground?
- Allocate a multi-element ‘cell/chunk’
- Stay in ‘cell/chunk’? → no allocation
- Resizing on the fly

Challenges?

- Lock less algo?
- Linking new array
- Sizing

Hanging on a Bit

- Steal producerIndex parity bit
  - Fixup code to match
- CAS parity to block ALL other producers
  - Let them spin

JUMP indicator

- Notify consumer needs to jump
- Use extra cell in array as next pointer

Performance?

- Slightly slower than array backed MPSC
- Very similar throughput
- Allocates as needed:
  - Stay in chunk → no allocation
  - Growable → allocate until capacity
  - Chunked → bounded, fixed chunks
  - Unbounded → blow yer heap, in fixed increments!

In Closing...

Thanks!