Effective Software

Lecture 6: Data races, synchronization, atomic operations, non-blocking algorithms

David Šišlák <u>david.sislak@fel.cvut.cz</u>

[1] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier, 2008.

[2] Fog, A.: The microarchitecture of Intel, AMD and VIA CPU, 2016.

[3] Russell, K., Detlefs, D.: Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing in OOPSLA'06. ACM, USA 2006.

[4] Oaks, S.: Java Performance: The Definitive Guide. O'Reilly, USA 2014.

public int A = 0; public int B = 0; public int C = 0; public int D = 0;

Thread 1	Thread 2
<pre>public void method1() { int r2 = A; B = 1; D = r2; }</pre>	<pre>public void method2() { int r1 = B; A = 2; C = r1; }</pre>

» what can be the results for C and D?

public int A = 0; public int B = 0; public int C = 0; public int D = 0;

Thread 1	Thread 2
<pre>public void method1() { int r2 = A; B = 1; D = r2; }</pre>	<pre>public void method2() { int r1 = B; A = 2; C = r1; }</pre>

- » what can be the results for C and D?
 - C=0, D=0
 - C=1, D=0
 - C=0, D=2
 - anything else?

Data Races – Disassembled Method and Assembly Code

<pre>public void method1() { int r2 = A:</pre>	0: aload_0 1: getfield	#2 // Field A:I	Heap object structure:
B = 1; D = r2:	4: istore_1 5: aload 0		8B - mark word
}	6: iconst_1		4B / 8B – Klass ref.
	7: putfield 10: aload_0 11: iload_1	#3 // Field B:1	object data
	12: putfield 15: return	#5 // Field D:I	Klass – internal JVM representation of class Metadata
instructions reordered	in C2 compiler:		4B – 32bit, or 64bit <32GB heap 8B – 64bit no compressedOOP
0x000000010639924c: mo∨l	RSI is this \$0x1,0x10(%rsi)	;*putfield B ; - datarace.Data	Race::method1@7 (line 11)
0x0000000106399253: mov	0xc(%rsi).%r11d		

%r11d,0x18(%rsi) ;*putfield D 0x0000000106399257: mov ; - datarace.DataRace::method1@12 (line 12)

note: all machine code examples are from JVM 8 64-bit <32GB, Intel Haswell CPU in AT&T syntax

- » the same reordering happens in method2 resulting into fourth output
 - C=1, D=2

Data Races – CPU Execution Pipelining

» simplified non-parallel instruction pipelining in each core



» each step is parallelized as well, e.g. Haswell does 4 instructions in single cycle (execution depends on type and independency of instructions)

Data Races – CPU Memory Model

» CPU vs. core vs. thread

					L1 Data Cache				
Core	Core	Core	Core	Core	Core	Size	Line Size	Latency	Associativty
	core		core	core		32 KB	64 bytes	4 ns	8-way
			L1 Instruction Cache						
L1 L1	L1 L1	L1 L1	L1 L1	L1 L1	L1 L1	Size	Line Size	Latency	Associativty
inst dat	inst dat	inst dat	inst dat	inst dat	at inst dat	32 KB	64 bytes	4 ns	4-way
					L2 Cache				
L2	L2	L2	L2	L2	L2	Size	Line Size	Latency	Associativty
						256 KB	64 bytes	10 ns	8-way
						L3 Cache			
L3						Size	Line Size	Latency	Associativty
						12 MB	64 bytes	50 ns	16-way
					Main Memory				
Main Memory						Size	Line Size	Latency	Associativty
							64 bytes	75 ns	

- » all writes to main memory are done in write-back cache mode
 - standard writes requires data to be cached (expensive cache miss)
 - non-temporal writes (especially useful for large block writes)
 - content directly queued to memory without caching at all
 - prefetch instructions available

Data Races – CPU Execution Pipelining – Superscalar Execution

- » modern CPUs have multiple execution units in each core (8 in Intel Haswell)
 - units have various capabilities (4x integer ALU, 2x FPU mul, 2x mem read, ...)
 - multiple µops with various latency executed in parallel during each per cycle
- » independent instructions can be executed out-of-order or in parallel
 - not using the same register or address
- » memory reads are never reordered
 - parallel independent reads
- » later (independent) reads can be reordered and executed before writes
 - serialized writes only



Volatile Variable – Memory Barrier

making A and B volatile:

```
public volatile int A = 0;
public volatile int B = 0;
public int C = 0;
public int D = 0;
public void method1() {
    int r2 = A;
    B = 1;
    D = r2;
}
```

results into assembly code:

0x000000010710e08c:	mov	0xc(%rsi),%r11d
0x000000010710e090:	movl	\$0x1,0x10(%rsi)
0x000000010710e097:	lock ad	ldl \$0x0,(%rsp)
0x000000010710e09c:	mov	%r11d,0x18(%rsi)



- » memory operations over store in volatile are **not reordered** in C1/C2 compiler
- » no need for read barriers not reordered during execution in CPU
- » instruction lock prefix forbids all reordering around and synchronize previous writes to be visible by all others CPUs
- » *lock addl \$0x0,(%rsp)* is fastest memory barrier no operation inside CPU

Volatile Variable

- » never cached thread-locally all access directly to main memory
- » guarantees **atomic read and write** operations (defines memory barrier)
- » can be used for both primitives and references to objects
- » don't block thread execution
- » BUT:
 - volatile writes are much slower due to cache flush (~100x)
 - volatile reads (if there are writes) are slower (~25x, #CPU/cores)
 - due to invalidated cache
 - still faster than synchronization/locks
- » not necessary for:
 - immutable objects
 - variable accessed by only one thread (context switch properly flushes cache already)
 - where variable is within complex synchronized operation

Counter Example - Volatile

```
public class VolatileCounter {
    private volatile int cnt=0;
    public int get() {
        return cnt;
    }
    public void increment() {
        cnt++;
    }
}
```

» will it work as expected in multi-threaded environment?

Counter Example - Volatile

```
public class VolatileCounter {
    private volatile int cnt=0;
```

```
public int get() {
    return cnt;
}
public void increment() {
    cnt++;
}
```



- » will it work as expected in multi-threaded environment?
 NO
- » volatile
 - not suitable for read-update-write operations
 - useful for one-thread write (e.g. termination flag)
 - must be used if flag is set by different thread otherwise C2 compiler could create infinite loop without testing

Volatile Arrays

```
public class VolatileIntArray {
    private volatile int[] array;

public VolatileIntArray(int capacity) {
    array = new int[capacity];
  }

public int get(int index) {
    return array[index];
  }

public void put(int index, int value) {
    array[index] = value;
  }
}
```

» Is put operation to array member volatile?

Volatile Arrays

```
public class VolatileIntArray {
    private volatile int[] array;
```

```
public VolatileIntArray(int capacity) {
    array = new int[capacity];
}
```

```
public int get(int index) {
    return array[index];
}
```

}

```
public void put(int index, int value) {
    array[index] = value;
}
```

```
8B - mark word
```

4B / 8B – Klass ref.

... object data

8B - mark word

4B / 8B - Klass ref.

4B – array length

sequence of values

» Is put operation to array member volatile?

NO – see assembly code, there is no cache synchronization with lock

```
# this:
           rsi:rsi
                     = 'datarace/VolatileIntArray'
# parm0:
           rdx
                     = int
# parm1:
                     = int
           rcx
0x00000011170bbcc: mov
                         0xc(%rsi),%esi
0x000000011170bbcf: shl
                          $0x3,%rsi
                                            ;*getfield array
                                            ; - datarace.VolatileIntArray::put@1 (line 15)
0x00000011170bbd3: movslq %edx,%rdi
0x00000011170bbd6: cmp
                          0xc(%rsi),%edx
                                            ; implicit exception: dispatches to 0x00000011170bbef
                          0x00000011170bbf9 — ArrayOutOfBoundsException
0x000000011170bbd9: jae
0x00000011170bbdf: mov
                         %ecx,0x10(%rsi,%rdi,4) ;*iastore
                                            ; - datarace.VolatileIntArray::put@6 (line 15)
```

Volatile Arrays - Solution

private volatile int[] array; public void put(int index, int value) { array[index] = value; array = array; }

8B - mark word 4B / 8B – Klass ref. ... object data

- » just array reference is volatile
- » added unnecessary array reference update adds assembly code

0x000000010db21a67: mov %r8d,0xc(%rsi)
0x00000010db21a80: lock addl \$0x0,(%rsp) ;*putfield array
; - datarace.VolatileIntArray::put@12 (line 16)

- » instruction **lock prefix** forbids all reordering around and synchronize previous writes to be visible by all others CPUs
- » not suitable for read-update-write operations

Counter Example – Synchronized and ReentrantLock



- » no issue with read-update-write operations
- » synchronized
 - method vs. block
 - object instance vs. class instance (static methods)

JVM - Synchronize Implementation

Mark word (64-bit JVM):



- prototype mark word in Klass **>>**
- **lock records** in stack (on pre-compiled locations for compiled code) **>>**
 - 8B displacement of original object mark word recursive lock has 0
 - 4B / 8B compressedOOP/OPP to locked object
- thin lock is using CAS instruction on lock/unlock to modify mark word **》**
 - use spin-locking (10 cycles with volatile read + NOPs) before fat locking
- **fat lock** is using monitor object on heap (inflating creates, deflating destroys) **>>**
 - contended lock or call of wait/notify
 - monitor: original mark word, OS lock, conditions, set of threads; support parking

25th March 2019

JVM - Synchronize Implementation



- » **biased lock** is fast locking/unlocking by single thread without any CAS instruction
 - biasable state enabled 4 seconds after JVM start (startup-up, learning)
 - different thread and valid epoch -> instance re-biasing OR thin/fat locking
 - global safe pointing needed for setting biasable, re-biasing, bias revocation
 - bulk operations amortizing cost for safe pointing (all instance types)
 >20 re-biasing -> bulk re-biasing (increment epoch in prototype, scan locks)
 >40 re-biasing -> bulk revocation (change in prototype)
 - mark word **normalization** during GC preserve *hashed*, *locked*, *un-biasable*
 - identity hash (Object.toString) or fat lock disable instance biased locking

JVM - Synchronize Implementation



Reentrant Locks

- » bias-based locking with extended operations in comparison to synchronized
 - lock(), unlock()
 - lockInterruptibly() throws InterruptedException
 - boolean tryLock()
 - boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException
- » fairness
 - blocked threads are **ordered** for fair locking
 - **new** ReentrantLock(boolean fair), by default unfair
 - **synchronized** is unfair
 - unfair ReentrantLocks are slightly faster than synchronized
 - but another instance in HEAP
 - fair locks are slower (~100x)

Counter Example – AtomicInteger

```
public class AtomicCounter {
     private AtomicInteger cnt = new AtomicInteger( initialValue: 0);
     public int get() {
         return cnt.get();
     public void increment() {
         cnt.incrementAndGet();
     }
 }
AtomicInteger implementation
private static final long valueOffset;
static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField( name: "value"));
    } catch (Exception ex) { throw new Error(ex); }
private volatile int value;
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
                                                                                  non-blocking
        var5 = this.getIntVolatile(var1, var2);
                                                                                  pattern
    } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));
    return var5;
public final int getAndIncrement() {
                                                                                                20
    return unsafe.getAndAddInt( o: this, valueOffset, i: 1);
```

Counter Example – AtomicInteger – Assembly Code

<u>C2 compiler assembly code for AtomicCounter::increment</u>

RSI is **this**



- » while cycle optimized and replaced with single instruction
- » instruction **lock prefix** forbids all reordering around and synchronize previous writes to be visible by all others CPUs
- » instruction **lock prefix** ensures that core has exclusive ownership of the appropriate cache line for the duration of the operation
 - cache coherency using MESIF (Haswell) with fall-back to mem bus lock
- » AtomicInteger-based counter is fastest of all for multi-threaded

Atomic Operations

- » 32-bit CPUs support 64-bit CAS operations
 - **cmpxchg** src_operand, dst_operand implicit instruction lock prefix
- » 64-bit CPUs support 128-bit CAS operations
 - **cmpxchg16b** works with RDX:RAX and RCX:RBX register pairs
- » JAVA uses only 64-bit CAS operations in java.util.concurrent.atomic
 - AtomicBoolean
 - AtomicInteger
 - AtomicLong
 - AtomicReference
 - AtomicIntegerArray
 - AtomicLongArray
 - AtomicReferenceArray

Atomic Field Updaters

- » suitable with large number of objects of the given type it saves memory
 - don't require single instance to have an extra object embedded
- » refer volatile variable directly without getter and setters

```
public class ObjectWithAtomic {
          private final AtomicInteger value =
              new AtomicInteger(0);
          // ...
          public void method1() {
              // ...
              if (value.compareAndSet(1, 2)) {
                   // ...
              3
          3
      }
      public class ObjectWithAtomic {
          private static AtomicIntegerFieldUpdater<ObjectWithAtomic>
               valueUpdater = AtomicIntegerFieldUpdater.nevUpdater(ObjectWithAtomic.class, "value");
          private volatile int value = 0;
          // ...
          public void method1() {
              // ...
               if (valueUpdater.compareAndSet(this, 1, 2)) {
                   // ...
               3
           }
25<sup>th</sup> March 2019
                                                ESW – Lecture 6
```

Atomic Field Updaters

- » but beware of less efficient operations for atomic field updaters
- » AtomicIntegerFieldUpdater implementation

```
private void fullCheck(T obj) {
    if (!tclass.isInstance(obj))
        throw new ClassCastException();
    if (cclass != null)
        ensureProtectedAccess(obj);
}
public boolean compareAndSet(T obj, int expect, int update) {
    if (obj == null || obj.getClass() != tclass || cclass != null) fullCheck(obj);
    return unsafe.compareAndSwapInt(obj, offset, expect, update);
}
```

- » existing field updaters
 - AtomicIntegerFieldUpdater
 - AtomicLongFieldUpdater
 - AtomicReferenceFieldUpdater
- » no array field updaters

Atomic Complex Types

- » AtomicMarkableReference
 - object reference along with a mark bit
- » AtomicStampedReference
 - **object reference** along with an **integer "stamp"**
- » <u>notes</u>:
 - useful for ABA problem
 - A -> B and B -> A, how can I know that A has been changed since the last observation?
 - doesn't use double-wide CAS (CAS2, CASX) -> much slower than simple atomic types due to **object allocation**

Atomic Complex Types – Larger Than 64-bits

- » AtomicMarkableReference
 - object reference along with a mark bit
- » AtomicStampedReference

25th March 2019

• object reference along with an integer "stamp"

```
public class AtomicStampedReference<V> {
    private static class Pair<T> {
        final T reference;
        final int stamp;
        private Pair(T reference, int stamp) {
            this.reference = reference;
            this.stamp = stamp;
        static <T> Pair<T> of(T reference, int stamp) {
            return new Pair<T>(reference, stamp);
        3
    }
    private volatile Pair<V> pair;
                                    expectedReference,
   public boolean compareAndSet(V
                                V newReference,
                                 int expectedStamp,
                                 int newStamp) {
       Pair<V> current = pair;
       return
           expectedReference == current.reference &&
           expectedStamp == current.stamp &&
           ((newReference == current.reference &&
             newStamp == current.stamp) ||
             casPair(current, Pair.of(newReference, newStamp)));
   }
```

Non-blocking Algorithms

- » lock-free, block-less but not usually wait-free (because of unbounded loops)
 - based on CMPXCHG and LOCK prefixed instructions •
- » shared resources secured by locks have drawbacks
 - high-priority thread can be blocked (e.g. interrupt handler) ٠
 - parallelism reduced by coarse-grained locking (unfair locks) •
 - fine-grained locking and fair locks increases overhead •
 - can lead to **deadlocks**, priority inversion (low-priority thread holds a • shared resource which is required by high-priority thread)
- non-blocking algorithms properties: **>>**
 - outperform blocking algorithms because most of CMPXCHG succeeds • on the first try
 - removes cost for synchronization, thread suspension, context switching •
- note: **real-time systems require wait-free algorithms** (finite number of **>>** 25th NSteps19 ESW – Lecture 6

Non-blocking stack (LIFO)

Treiber's algorithm (1986) **>>**

```
static class Node<E> {
    final E item:
    Node<E> next;
                                                                        н
                                                                                 А
   public Node(E item) { this.item = item; }
}
AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();
                                                                                 С
public void push(E item) {
    Node<E> newHead = new Node<E>(item);
                                                                                 С
    Node<E> oldHead;
    do {
        oldHead = head.get();
                                                                        н
                                                                                 С
        newHead.next = oldHead;
    } while (!head.compareAndSet(oldHead, newHead));
}
public E pop() {
   Node<E> oldHead;
    Node<E> newHead;
                                                                        н
    do {
        oldHead = head.get();
        if (oldHead == null)
            return null;
        newHead = oldHead.next;
    } while (!head.compareAndSet(oldHead,newHead));
    return oldHead.item;
}
```





н	 в	_	→

push after pop can cause ABA problem if address is reused !

Thread-safe collections and maps

» blocking collections and maps

- static<T> Collection<T> Collections.synchronizedCollection(Collection<T> c)
- static<T> List<T> Collections.synchronizedList(List<T> list)
- static<K,V> Map<K,V> Collections.synchronizedMap(Map<K,V> m)
- static<T> Set<T> Collections.synchronizedSet(Set<T> s)
- also for SortedSet and SortedMap

» non-blocking collections and maps

- ConcurrentLinkedQueue (interface Collection, Queue):
 - E peek(), E poll(), add(E)
- ConcurrentHashMap (interface Map):
 - putIfAbsent(K key, V value), remove(Object key, Object value)
 - replace(K key, V oldValue, V newValue)
- ConcurrentSkipListMap (interface SortedMap), ConcurrentSkipListSet (interface SortedSet)
- » non-blocking collections and maps are slower for single-threaded access
 - due to usage of CASE instructions in comparison to biased locking

ConcurrentHashMap

- » concurrent reads get, iterator
- » minimize update contention
 - initial concurrency level 16 (can be changed) # updating threads
 - initial insertion into empty segment uses CMPXCHG operation
 - later modifications are based on segment-based locks



ConcurrentHashMap

- » table resizing (occupancy exceed load factor 0.75)
 - power of two expansions
 - same index or power of two index
 - reusing internal Node if next is not changed majority of cases
 - any thread can help resizing instead of block
 - Forward nodes to notify users about moved