Effective Software

Lecture 5: Data races, synchronization, atomic operations, non-blocking algorithms

David Šišlák
david.sislak@fel.cvut.cz
what can be the results for C and D?
Data Races – Multi-threaded Environments

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

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>
| public void method1() {
  int r2 = A;
  B = 1;
  D = r2;
} | public void method2() {
  int r1 = B;
  A = 2;
  C = r1;
} |

» what can be the results for C and D?

- C=0, D=0
- C=1, D=0
- C=0, D=2
- anything else?
instructions **reordered in C2 compiler:**

<table>
<thead>
<tr>
<th>RSI is this</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000010639924c: movl $0x1,0x10(%rsi) ;*putfield B</td>
</tr>
<tr>
<td>0x00000000106399253: mov 0xc(%rsi),%r11d</td>
</tr>
<tr>
<td>0x00000000106399257: mov %r11d,0x18(%rsi) ;*putfield D</td>
</tr>
</tbody>
</table>

» 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

IF: Instruction fetch
EX: Execution
MEM: Memory access
WB: Write back

Cycles

Instruction # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
---|---|---|---|---|---|---|---|---|---
Instruction i | IF | ID | EX | MEM | WB | | | | |
Instruction i+1 | IF | ID | EX | MEM | WB | | | | |
Instruction i+2 | IF | ID | EX | MEM | WB | | | | |
Instruction i+3 | IF | ID | EX | MEM | WB | | | | |
Instruction i+4 | IF | ID | EX | MEM | WB | | | | |
» CPU vs. core vs. thread

» 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)
  • prefetch instructions available
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:

```java
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 addl $0x0,(%rsp)
0x000000010710e09c: mov %r11d,0x18(%rsi)
```

- operations over volatile are not reordered in C2 compiler
- no need for read barriers – not reordered during execution in CPU
- **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 objects (references)
- 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
  - where variable is within complex synchronized operation
Counter Example - Volatile

```java
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

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

    public int get() {
        return cnt;
    }

    public void increment() {
        cnt++;
    }
}
```

Increment assembly code:
```
0x000000010911544c: mov    0xc(%rsi),%edi
0x000000001091154f: inc    %edi
0x0000000010911551: mov    %edi,0xc(%rsi)
0x0000000010911554: lock addl $0x0,(%rsp)
```

» 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

```java
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

```java
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?

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

```assembly
0x000000001117bbcc: mov 0xc(%rsi),%esi
0x000000001117bbcf: shl $0x3,%rsi ;*getfield array
                ; - datarace.VolatileIntArray::put@1 (line 15)

0x000000001117bbd3: movslq %edx,%rdi
0x000000001117bbd6: cmp 0xc(%rsi),%edx ; implicit exception: dispatches to 0x000000001117bbef
0x000000001117bbd9: jae 0x000000001117bbf9 ;ArrayOutOfBoundsException
0x000000001117bbdf: mov %ecx,0x10(%rsi,%rdi,4) ;*iastore
                ; - datarace.VolatileIntArray::put@6 (line 15)
```
**Volatile Arrays - Solution**

```java
private volatile int[] array;

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

» just array reference is volatile

» added unnecessary `array` reference update adds assembly code

```
0x0000000010db21a67: mov %rd,0xc(%rsi)
0x0000000010db21a80: lock addl $0x0,(%rsp)  ;*putfield array
               ;- datarace.VolatileIntArray::put@12 (line 16)
```

» **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

```java
public class SynchronizedCounter {
    private int cnt=0;

    public int get() {
        return cnt;
    }

    public synchronized void increment() {
        cnt++;
    }
}
```

```java
public class ReentrantCounter {
    private int cnt=0;
    private ReentrantLock lock = new ReentrantLock();

    public int get() {
        return cnt;
    }

    public void increment() {
        lock.lock();
        try {
            cnt++;
        } finally {
            lock.unlock();
        }
    }
}
```

» no issue with read-update-write operations
» synchronized
  • method vs. block
  • object instance vs. class instance (static methods)
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 locking** – using CAS instruction on lock/unlock to modify mark word
  • use spin-locking (10 cycles with volatile read + NOPs) before fat locking

» **fat locking** – monitor object on heap (created by inflating, deflating)
  • contended lock or call of *wait/*notify
  • monitor: original mark word, OS lock, conditions, set of threads; support parking
Mark word (64-bit JVM):

- **biasing locking** – fast locking/unlocking by single thread without any CAS
  - biasable – 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 – 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** or **fat lock** disable instance biasing locking
- assembly code optimized for biasing and thin locking
  check yourself
- biased locking startup options:
  -XX:-UseBiasedLocking
  -XX:BiasedLockingStartupDelay=0
Reentrant Locks

» extended operations in comparison to **synchronized**:
   • lock(), unlock()
   • lockInterruptibly() throws InterruptedException
   • boolean tryLock()
   • boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException

» **fairness**
   • **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

```java
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("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 {
    var5 = this.getIntVolatile(var1, var2);
  } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));
  return var5;
}

public final int getAndIncrement() {
  return unsafe.getAndAddInt(this, valueOffset, i: 1);
}
```

non-blocking pattern
Counter Example – AtomicInteger – Assembly Code

C2 compiler assembly code for AtomicCounter::getAndIncrement:

```assembly
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000010b108d4c</td>
<td>mov 0xc(%rsi),%r11d     ;*getfield cnt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>; - datarace.AtomicCounter::increment@1 (line 13)</td>
<td></td>
</tr>
<tr>
<td>0x0000000010b108d50</td>
<td>test %r11d,%r11d</td>
<td>null pointer check with exception</td>
</tr>
<tr>
<td>0x0000000010b108d53</td>
<td>je 0x0000000010b108d68</td>
<td></td>
</tr>
<tr>
<td>0x0000000010b108d55</td>
<td>lock addl $0x1,0xc(%r12,%r11,8) ;*invokevirtual getAndAddInt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>; - java.util.concurrent.atomic.AtomicInteger::incrementAndGet@8 (line 186)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>; - datarace.AtomicCounter::increment@4 (line 13)</td>
<td></td>
</tr>
</tbody>
</table>
```

- while cycle optimized and replaced with **single instruction**
- lock prefix forbids all reordering around and synchronize previous writes to be visible by all others CPUs
- **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 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 version in `java.util.concurrent.atomic`
  • AtomicBoolean
  • AtomicInteger
  • AtomicLong
  • AtomicReference
  • AtomicIntegerArray
  • AtomicLongArray
  • AtomicReferenceArray
Atomic Field Updaters

» suitable with large number of object of the given type – it saves memory
  • don’t require single instance to have an extra object embedded
  » refer variable “normally” without getter and setters

```java
public class ObjectWithAtomic {
  private final AtomicInteger value =
    new AtomicInteger(0);
  // ...

  public void method1() {
    // ...
    if (value.compareAndSet(1, 2)) {
      // ...
    }
  }
}
```

```java
public class ObjectWithAtomic {
  private static AtomicIntegerFieldUpdater<ObjectWithAtomic>
    valueUpdater = AtomicIntegerFieldUpdater.newUpdater(ObjectWithAtomic.class, "value");
  private volatile int value = 0;
  // ...

  public void method1() {
    // ...
    if (valueUpdater.compareAndSet(this, 1, 2)) {
      // ...
    }
  }
}
```
Atomic Field Updaters

» but beware of **less efficient** operations over atomic field updaters

» AtomicIntegerFieldUpdater:

```java
private void fullCheck(T obj) {
    if (!tclass.getInstance(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 updater exists
Atomic Complex Types

» AtomicMarkableReference
   • object reference along with a mark bit

» AtomicStampedReference
   • object reference along with an integer “stamp”

» notes:
   • 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-wideCAS (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
  • object reference along with an integer “stamp”

```java
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);
        }
    }

    private volatile Pair<V> pair;

    public boolean compareAndSet(V expectedReference,
                                   V newReference,
                                   int expectedStamp,
                                   int newStamp) {
        Pair<V> current = pair;
        return expectedReference == current.reference &&
               expectedStamp == current.stamp &&
               casPair(current, Pair.of(newReference, newStamp));
    }
}
```
Non-blocking Algorithms

» **lock-free**, block-less but not usually **wait-free** (note while loops)
  • based on CMPXCHG and LOCKed instructions

» shared resources secured by locks:
  • high-priority thread can be blocked (e.g. interrupt handler)
  • 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:** **wait-free is mandatory** for real-time systems
Non-blocking stack (LIFO)

» Treiber’s algorithm (1986)

```java
static class Node<E> {
    final E item;
    Node<E> next;

    public Node(E item) { this.item = item; }
}

AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

public void push(E item) {
    Node<E> newNode = new Node<E>(item);
    Node<E> oldHead;
    do {
        oldHead = head.get();
        newNode.next = oldHead;
    } while (!head.compareAndSet(oldHead, newNode));
}

disassemble node

public E pop() {
    Node<E> oldHead;
    Node<E> newNode;
    do {
        oldHead = head.get();
        if (oldHead == null)
            return null;
        newNode = oldHead.next;
    } while (!head.compareAndSet(oldHead, newNode));
    return oldHead.item;
}

sequence of removal-addition if address is reused cause ABA
```
Thread-safe collections and maps

» blocking variants:
  • 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 variants:
  • ConcurrentHashMap (interface Map):
    – putIfAbsent(K key, V value), remove(Object key, Object value)
    – replace(K key, V oldValue, V newValue)
  • ConcurrentHashMap (interface SortedMap), ConcurrentSkipListSet (interface SortedSet)
ConcurrentHashMap

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

» bin contention
  • lists while <8
  • balanced tree to reduce search times – maintains next for iteration
ConcurrentHashMap

» table resizing (occupancy exceed load factor)
  • 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