Architecture of software systems

Course 6: Threads, synchronization, atomic operations, non-blocking algorithms

David Šišlák
david.sislak@fel.cvut.cz

Threads

- processes vs. threads
  - both support concurrent execution
  - one process has one or multiple threads
  - threads share the same address space (data and code)
  - context switching between threads is less expensive
  - thread intercommunication is relatively efficient
- a thread executes sequence of code with own stack with frames

  t.getStackTrace()
  - own local variables
  - own method parameters
- thread creation by
  - subclass of **java.lang.Thread**
  - implementation of **java.lang.Runnable**

Threads

- Thread.currentThread()

- each thread has
  - id – unique long id, only get
  - name - get/set
  - priority – get/set
    - Thread.MIN_PRIORITY (1), NORM_PRIORITY (5), MAX_PRIORITY (10)
  - thread group – get/set
  - uncaught exception handler – get/set + get/setDefaultExceptionHandler
    - UncaughtExceptionHandler
  - daemon flag – is/set
  - context class loader – get/set, used to load classes and resources inside
  - interrupted – interrupt(), isInterrupted(), interrupted()
    - InterruptedException
  - status – see next slide

Threads

- thread states - t.getState()
  - **new**
    - after creation
  - **runnable**
    - start()
  - **blocked**
    - waiting for a lock, (re)enter synchronized method/block
  - **waiting** (can be interrupted)
    - o.wait(), t.join(), LockSupport.park() – not return interrupted
  - **timed waiting** (can be interrupted)
    - Thread.sleep(x), o.wait(x), t.join(x)
    - LockSupport.parkNanos(x), LockSupport.parkUntil(time)
  - **terminated**
    - finished t.run() method, Runtime.exit(), t.~~stop~~()
- Thread.yield() – allow other threads to execute

Threads – usage of Runnable

```
class RunnableThread implements Runnable {

    Thread runner;
    ...

    public RunnableThread(String threadName) {
        runner = new Thread(this, threadName);
        runner.start();
    }

    public void run() {
        System.out.println(Thread.currentThread());
        ...
    }

}
```

Threads – subclass of Thread

```
class XThread extends Thread {
     ...

     XThread(String threadName) {
          super(threadName);
          start();
     }

     public void run() {
          System.out.println(Thread.currentThread().getName());
          ...
     }

}
```

ExecutorService

- concept of thread pooling since 1.5
- suitable for execution of large number of asynchronous tasks
  - e.g. HTTP requests in server
- reduce overhead with Thread creation for each task, context switching
- interface - java.util.concurrent.ExecutorService
  - shutdown(), shutdownNow(), awaitTermination
  - execute(Runnable r)
  - Future<?> submit(Runnable r), Future<T> submit(Callable<T> c)
- java.util.concurrent.Future<T>
  - boolean cancel(boolean mayInterruptIfRunning)
  - isCancelled(), isDone()
  - V get(), V get(long timeout, TimeUnit unit)
- java.util.concurrent.Executors (optionally with ThreadFactory)
  - newSingleThreadExecutor()
  - newFixedThreadPool(nThreads)
  - newCachedThreadPool() – 60 seconds keepalive

Data races

Thread 1	Thread 2
1: r2 = A;	3: r1 = B;
2: B = 1;	4: A = 2;

» r1 and r2 are local variables

» A and B are shared variables (heap located) initially set to 0

» what can be the results for r1 and r2?

Thread 1	Thread 2
1: r2 = A;	3: r1 = B;
2: B = 1;	4: A = 2;

» r1 and r2 are local variables

» A and B are shared variables (heap located) initially set to 0

» what can be the results for r1 and r2?

- r1=0, r2=0
- r1=1, r2=0
- r1=0, r2=2
- anything else?

Synchronized

» each object is associated with *monitor*

» **synchronized** is implemented using *monitors*

```
public class Test {
    private static int var1 = 0;
    // ...

    public static synchronized void method1() {
        // ...
        var1++;
    }

    public synchronized void method2() {
        // ...
        var1++;
    }

    // ...
}
```

» Is this correct?

Synchronized

- » each object is associated with *monitor*
- » **synchronized** is implemented using *monitors*

```java
public class Test {
    private static int var1 = 0;
    // ...

    public static synchronized void method1() {
        // ...
        var1++;
    }

    public void method2() {
        synchronized (Test.class) {
            // ...
            var1++;
        }
    }

    // ...
}
```

Reentrant locks

- » java.util.concurrent.locks.ReentrantLock since 1.5
- » 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 !
  - fair locks are slower !

Synchronized VS. reentrant lock

```
public class Test {
    // ...

    public void method1() {
        // ...
        synchronized (this) {
            // ...
        }
        // ...
    }

    // ...
}
```

```
public class Test {
    private ReentrantLock lock = new ReentrantLock();
    // ...

    public void method1() {
        // ...
        lock.lock();
        // ...
        lock.unlock();
        // ...
    }

    // ...
}
```

» Is this correct?

Synchronized VS. reentrant lock

```java
public class Test {
    // ...

    public void method1() {
        // ...
        synchronized (this) {
            // ...
        }
        // ...
    }

    // ...
}
```

```java
public class Test {
    private ReentrantLock lock = new ReentrantLock();
    // ...

    public void method1() {
        // ...
        lock.lock();
        try {
            // ...
        } finally {
            lock.unlock();
        }
        // ...
    }

    // ...
}
```

- Is this correct?
  - NO – need catch exceptions

Reentrant lock performance

- see HighContentionSimulator implementation
- Sun's JDK 1.6.0_14 on 2x Intel Xeon E5420 2.5GHz (8 cores in total)

- results for 8 threads:

LOCK operations/second 3 499 925
SYNC operations/second 1 104 862
LOCK operations/second 3 478 742
SYNC operations/second 1 149 406
LOCK operations/second 3 500 417
SYNC operations/second 1 121 584

- but ReentrantLock is standard object on heap

Deadlock

```
public class Test {
    private static class Resource {
        private Resource() {
            // ...
        }

        private synchronized void directAccess() {
            // ...
        }

        private synchronized void accessWithSubResource(Resource subResource) {
            // ...
            subResource.directAccess();
        }
    }

    public static void main(String[] args) {
        final Resource resource1 = new Resource();
        final Resource resource2 = new Resource();

        new Thread(new Runnable() {
            @Override
            public void run() { resource1.accessWithSubResource(resource2); }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() { resource2.accessWithSubResource(resource1); }
        }).start();
    }
}
```

Thread run control

```java
public class Test {
    // ...

    public void method1() {
        // ...
        synchronized (this) {
            // ...
            try {
                wait();
            } catch (InterruptedException e) {
                // ...
            }
            // ...
        }
    }

    public void method2() {
        // ...
        synchronized (this) {
            // ...
            notify();
            // ...
        }
    }

    // ...
}
```

```java
public class Test {
    private ReentrantLock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();
    // ...

    public void method1() {
        // ...
        lock.lock();
        try {
            // ...
            try {
                condition.await();
            } catch (InterruptedException e) {
                // ...
            }
            // ...
        } finally {
            lock.unlock();
        }
    }

    public void method2() {
        // ...
        lock.lock();
        try {
            // ...
            condition.signal();
            // ...
        } finally {
            lock.unlock();
        }
    }

    // ...
}
```

Thread run control

```
public class StoppableTask extends Thread {
  private boolean pleaseStop;

  public void run() {
    while (!pleaseStop) {
      // ...
    }
  }

  public void tellMeToStop() {
    pleaseStop = true;
  }
}
```

Thread run control

```
public class StoppableTask extends Thread {
  private volatile boolean pleaseStop;

  public void run() {
    while (!pleaseStop) {
      // ...
    }
  }

  public void tellMeToStop() {
    pleaseStop = true;
  }
}
```

Volatile variable

- never cached thread-locally – all access directly to main memory
- guarantees atomic read and write operations (using memory barrier)
- can be used also for primitives or null objects
- cannot block thread execution

```
public class Counter {
    private volatile int i = 0;

    public int get() {
        return i;
    }

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

Volatile variable

- never cached thread-locally – all access directly to main memory
- guarantees atomic read and write operations (using memory barrier)
- can be used also for primitives or null objects
- cannot block thread execution

```
public class Counter {
    private volatile int i = 0;

    public int get() {
        return i;
    }

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

→

```
public void increment() {
    int temp;
    synchronized (iAccessLock) {
        temp = i;
    }
    temp = temp + 1;
    synchronized (iAccessLock) {
        i = temp;
    }
}
```

Volatile variable

- never cached thread-locally – all access directly to main memory
- guarantees atomic read and write operations (using memory barrier)
- can be used also for primitives or null objects
- cannot block thread execution
- useful for one-thread write
- not suitable for read-update-write operations

```
public class Counter {
    private volatile int i = 0;

    public int get() {
        return i;
    }

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

→

```
public void increment() {
    int temp;
    synchronized (iAccessLock) {
        temp = i;
    }
    temp = temp + 1;
    synchronized (iAccessLock) {
        i = temp;
    }
}
```

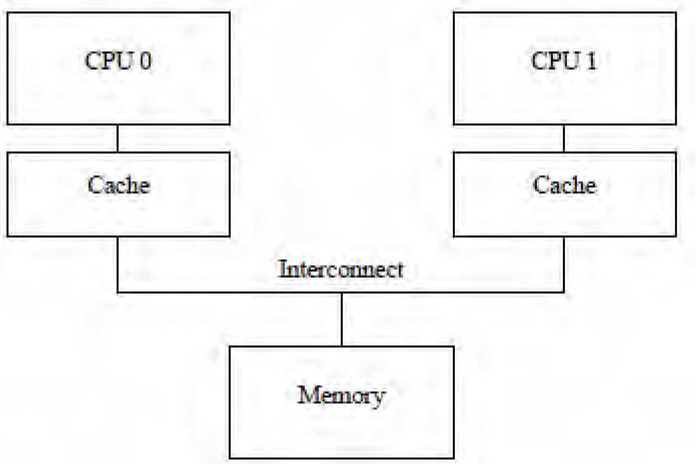
- not necessary for:
  - immutable objects
  - variable accessed by only one thread
  - where variable is within complex synchronized operation

Volatile example – multi-CPU

```
volatile int f = 0;
volatile int x = 0;

public int reader() {
    while (f == 0);
    return x;
}

public void writer(int value) {
    x = 42;
    f = 1;
}
```

Volatile example – multi-CPU

» memory changes made by one CPU can be propagated back to main memory out-of-order



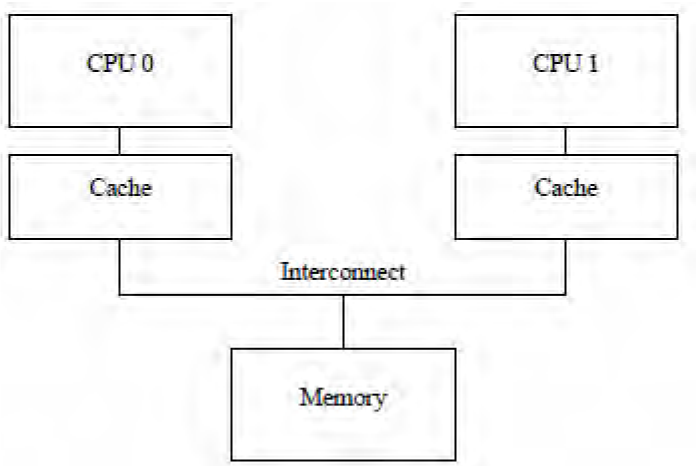

```
volatile int f = 0;
volatile int x = 0;

public int reader() {
    while (f == 0);
    return x;
}

public void writer(int value) {
    x = 42;
    f = 1;
}
```

» **correct behavior** since JAVA 1.5 where memory barrier is used for volatile

- usage of specific CPU instruction to guarantee it

Volatile example – multi-CPU

» memory changes made by one CPU can be propagated back to main memory out-of-order




```
volatile int f = 0;
volatile int x = 0;

public int reader() {
    while (f == 0);
    return x;
}

public void writer(int value) {
    x = 42;
    f = 1;
}
```

» **correct behavior** since JAVA 1.5 where memory barrier is used for volatile

- usage of specific CPU instruction to guarantee it

» but what for multi-thread write and read-update-write operations ?

- synchronization
- atomic classes

Atomic operations

- specific CPU instruction – **CMPXCHG** compare-and-exchange or **CAS** (compare-and-swap)
- modern processors support 128-bit CAS operations
- JAVA 5.0 utilizes 64-bit version in java.util.concurrent.atomic:
  - AtomicBoolean
  - AtomicInteger
  - AtomicLong
  - AtomicReference
- basic operations in AtomicInteger:
  - int get(), set(int value), boolean compareAndSet(int expect, int update)
  - int addAndGet(int delta)
  - int incrementAndGet(), int decrementAndGet()

Atomic counter

```
public class Counter {
    private final AtomicInteger i = new AtomicInteger(0);

    public int get() {
        return i.get();
    }

    public void increment() {
        i.incrementAndGet();
    }
}
```

» how is the atomic incrementAndGet implemented using CAS instruction?

AtomicInteger – implementation

```java
private volatile int value;

public final int get() {
    return value;
}


public final void set(int newValue) {
    value = newValue;
}


public final boolean compareAndSet(int expect, int update) {
return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}
```

```java
public final int addAndGet(int delta) {
    for (;;) {
        int current = get();
        int next = current + delta;
        if (compareAndSet(current, next))
            return next;
    }
}
```

Volatile arrays

```
volatile int arr[] = new int[SIZE];

public void method1() {
    int x = arr[0];
    arr[0] = 1;
}
```

Volatile arrays – solution 1

```
volatile int arr[] = new int[SIZE];

public void method1() {
    int x = arr[0];
    arr[0] = 1;
}
```

- atomic array versions:
  - AtomicIntegerArray
  - AtomicLongArray
  - AtomicReferenceArray
- basic operations for AtomicIntegerArray:
  - int get(int i), set(int i, int newValue)
  - boolean compareAndSet(int i, int expectedValue, int newValue)
  - int incrementAndGet(int i), int decrementAndGet(int i)

Volatile arrays – solution 2

```
volatile int arr[] = new int[SIZE];

public void method1() {
    int x = arr[0];
    arr[0] = 1;
}
```

```
volatile int arr[] = new int[SIZE];

public void method1() {
    arr[0] = 1;
    arr = arr;
}
```

- » do not require wrapper object
- » but slightly inefficient due to another read-write operation
- » do not support CAS operations

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

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

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

```
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:

```
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 updater exists

Atomic complex types

» AtomicMarkableReference:
  - **object reference** along with a **mark bit**

» AtomicStampedReference:
  - **object reference** along with an **integer “stamp”**

» see the implementation:
  - 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 previous version

Non-blocking algorithms

» lock-free, wait-free, based on CAS 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 CAS succeeds on the first try
  - removes penalty for synchronization, thread suspension, context switching

» note: required for real-time systems

Non-blocking stack (LIFO)

» based on Treiber's algorithm (1986)

```
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> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
        oldHead = head.get();
        newHead.next = oldHead;
    } while (!head.compareAndSet(oldHead, newHead));
}

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

Thread-safe collection and map

» 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:

- ConcurrentLinkedQueue (interface Collection, Queue):
  - E peek(), E poll()
- 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)