Concurrent programming (part 2)

Agenda

- Thread communication
- Thread-safe collections
- Synchronizers

Thread communication

Thread communication

- So far we saw how we can we can use implicit and explicit locks to provide synchronization between threads
- However locking might not be very flexible in coordinating threads
- A supplemental mechanism whereby threads can notify ("wake up") each other and wait to be notified is provided by the JVM

Provides a publish-subscribe mechanism at the thread level in the JVM

Wait and notify

- This is achieved by the **wait**, **notify** and **notifyAll** methods provided by the **java.lang.Object** class

- These methods work over a monitor lock that must be held from threads (i.e. used within a **synchronized** method or block)

- **notify** wakes up only one thread waiting on the monitor lock while **notifyAll** wakes up all threads

Be careful when to use notify and notifyAll: in many cases it is more proper to use notifyAll.

Exiting waits

- Waking up from the **wait** method can happen in the following situations:
  - when **notify**/**notifyAll** is called from another thread
  - If timeout expires (in case the overloaded **wait** methods are used)
  - the waiting thread is interrupted by calling the **interrupt** method
  - on rare occasions the OS or JVM may wake up the thread (also called **spurious wakeup**)

Spurious wake-ups

- To guard against spurious wake ups **wait** must always be called in a loop !

NO !

```
if(condition) {
   wait();
}
```

CALL WAIT IN A LOOP

```
while(condition) {
   wait();
}
```

Wait/notify example

```
public synchronized void subscribe() {
        while(message == null) {
                try {
                        wait();
                } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                }
        }
        System.out.println("Message received: " + message);
}
```

```
public synchronized void publish(String message) {
        this.message = message;
        System.out.println("Notifying all threads ...");
        notifyAll();
}
```

Conditions

- JDK 5 introduced a more flexible (and preferable way) to specify wait conditions using the **java.util.concurrent.locks.Condition** interface

- Additional capabilities of the Condition interface include:

  – awaitUntil(Date date) method that waits until a specified date

  – **awaitUniterruptibly()** method that awaits until the thread is signalled

Condition example

```
public void subscribe() {
       try {
              lock.lock();
              while(message == null) {
                     try {
                            condition.await();
                     } catch (InterruptedException e) {
                            Thread.currentThread().interrupt();
                     }
              }
              System.out.println("Message received: " +
                     message);
       } finally {
              lock.unlock();
       }
}
```

Condition example (cont.)

```
public void publish(String message) {
        try {
                lock.lock();
                this.message = message;
                System.out.println("Notifying all threads ...");
                condition.signalAll();
        } finally {
                lock.unlock();
        }
}
```

Thread-safe collections

Concurrent collections

- The standard JDK collections (such as LinkedList and ArrayList) are NOT thread-safe (except for legacy **Vector** and **Hashtable**)

- The JDK provides several types of thread-safe collections:

  - synchronized collections (such as the ones that can be created with the **Collections.synchronizedXXX** methods)

  - lock-free thread-safe collections (such as ConcurrentHashMap)

Before JDK 5 ConcurrentHashMap was NOT lock-free.

Synchronized collections

- The following methods from the Collections class can be used to create synchronized collections:

  - synchronizedCollection
  - synchronizedSet
  - synchronizedSortedSet
  - synchronizedList
  - synchronizedMap
  - synchronizedSortedMap
  - synchronizedNavigableMap

BlockingQueue

- Other thread-safe lock-based collections are also provided by the **java.util.concurrent** package such as the implementations of the **BlockingQueue** interface:
  - ArrayBlockingQueue
  - LinkedBlockingQueue
  - LinkedBlockingDeque
  - DelayQueue
  - PriorityBlockingQueue
  - SynchronousQueue

BlockingQueue

- BlockingQueue implementations provide the possibility to for threads to wait on operations for adding or removing of elements
- If the blocking queue is full threads block until space becomes available for adding an element
- If the blocking queue is empty threads block until an element is inserted in the queue so it can be removed

BlockingQueues are used to hold tasks submitted to an Executor thread pool. Fixed thread pool, for example, uses by default a LinkedBlockingQueue

BlockingQueue

```java
// blockingQueue is i.e. an ArrayBlockingQueue
public void add() {
	try {
		blockingQueue.put("first");
		blockingQueue.put("second");
		blockingQueue.put("third");
	} catch (InterruptedException e) {
		Thread.currentThread().interrupt();
	}
}
```

```java
public void remove() {
	try {
		System.out.println(blockingQueue.take());
		System.out.println(blockingQueue.take());
		System.out.println(blockingQueue.take());
	} catch (InterruptedException e) {
		Thread.currentThread().interrupt();
	}
}
```

Lock-free collections

- Lock-free collections provided by the JDK come in two flavors:
  - copy-on-write collections
  - concurrent collections based on atomic (compare-and-swap) operations

Copy-on-write collections

- Copy-on-write collections create a new collection every time an element is added or removed

- In that regard they are immutable and can be safely accessed from multiple threads

- Copy-on-write collections provided by the JDK include:

  – CopyOnWriteArrayList

  – CopyOnWriteArraySet

CopyOnWriteCollections require extra performance due to the copying of the collection and should be avoided in scenarios where performance is critical

Concurrent collections

- Concurrent collections do not use locks but atomic compare-and-swap (CAS) operations

- The JDK provides support for compare-and-swap instructions provided by modern CPUs

- Avoiding locks (thus context switching) makes concurrent collections more performant in many scenarios than synchronized, blocking and copy-on-write collections

Concurrent collections

- Concurrent collections provided by the JDK include:
  - ConcurrentHashMap
  - ConcurrentLinkedQueue
  - ConcurrentLinkedDeque
  - ConcurrentSkipListMap
  - ConcurrentSkipListSet

ConcurrentHashMap

- ConcurrentHashMap provides additional thread-safe atomic operations over a traditional HashMap such as:
  - getOrDefault(key, value)
  - putIfAbsent(key, value)
  - remove(key, value)
  - replace(key, oldValue, newValue)
  - replace(key, newValue)
  - replaceAll(function)
  - computeIfAbsent(key, function)
  - computeIfPresent(key, function)
  - compute (key, function)
  - merge(key, value, function)

ConcurrentHashMap

```
ConcurrentHashMap<Integer, Integer> map =
       new ConcurrentHashMap<>();
map.put(1, 0);
for(int i = 0; i < 100; i++) {
       new Thread(() ->  {
              map.compute(1, (k, v) -> { return v + 1;});
       } ).start();
}
Thread.sleep(1000);
System.out.println(map.getOrDefault(1, -1));
```

Synchronizers

Synchronizers

- Synchronizers provide more specific mechanisms for synchronization between a number of threads

- The JDK provides several synchronizers that can be used by applications:

  - CountDownLatch
  - CyclicBarier
  - Semaphore
  - Exchanger
  - Phaser

Synchronizer: CountDownLatch

- Used when a predefined number of releases should happen before a thread is awakened

```
CountDownLatch latch = new CountDownLatch(5);
new Thread(() -> {
      try {
            latch.await();
            System.out.println("Workers have finished !");
      } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
      }}).start();
for (int i = 0; i < 5; i++) {
      new Thread(() -> {
            System.out.println("Starting worker: " +
                  Thread.currentThread().getName());
            latch.countDown();
      }).start();
}
```

Synchronizer: CyclicBarrier

- Allows a number of threads to await for a predefined number of waits after which they are released

```
CyclicBarrier barrier = new CyclicBarrier(5, () -> {
       System.out.println("Workers have finished !");});
for (int i = 0; i < 5; i++) {
       new Thread(() -> {
              System.out.println("Starting worker: " +
                     Thread.currentThread().getName());
              try {
                     barrier.await();
              } catch (InterruptedException e) {
                     Thread.currentThread().interrupt();
              } catch (BrokenBarrierException e) {}
              System.out.println("Ending worker: " +
                     Thread.currentThread().getName());
       }).start();
} // barrier can be reset with barrier.reset()
```

Synchronizer: Semaphore

- Each thread is blocked until a permit is available, semaphore is initialized with a number of permits

```
Semaphore semaphore = new Semaphore(3);
for (int i = 0; i < 5; i++) {
      new Thread(() -> {
            try {
                  System.out.println("Starting worker: " +
                        Thread.currentThread().getName());
                  semaphore.acquire();
                  System.out.println("Ending worker: " +
                        Thread.currentThread().getName());
                  } catch (InterruptedException e) {…}
      }).start();
}
Thread.sleep(2000);
semaphore.release();
semaphore.release();
```

Synchronizer: Exchanger

- Provides the possibility to two threads to exchange (swap) objects

```
Exchanger<Integer> exchanger = new Exchanger<>();
for (int i = 0; i < 2; i++) {
       new Thread(() -> {
              try {
                     Random random = new Random();
                     Integer value = random.nextInt();
                     Integer exchanged =
                            exchanger.exchange(value);
                     System.out.println("Exchanged " + value
                            + " for " + exchanged);
              } catch (InterruptedException e) {
                     Thread.currentThread().interrupt();
              }
       }).start();
}
```

Synchronizer: Phaser

- Similar to CountDownLatch and CyclicBarrier but provides the ability to change the number of awaiting threads before they can proceed

```
Phaser phaser = new Phaser(1);
phaser.bulkRegister(2);
for (int i = 0; i < 3; i++) {
        new Thread(() -> {
                System.out.println("Thread arriving at phaser");
                phaser.arriveAndAwaitAdvance();
                System.out.println("Thread leaving phaser");
        }).start();
}
```

Questions ?