Concurrent programming (part 3)

Agenda

- Atomic operations
- Concurrency utilities
- Testing concurrent applications

Atomic operations

Compare-and-Swap (CAS)

- Compare-and-swap is an atomic instruction that compares the location in memory with a given value an only if they are equals sets a new value

- CAS is used to implement atomic operations that achieve **optimistic locking** and are thread-safe

```
// represented by the following pseudocode
if (memoryLocation != value) {
        return false;
}
memoryLocation = newValue;
return true;
```

Compare-and-Swap (CAS)

- The JDK provides support for calling CAS instructions through native code as provided by the **jdk.internal.misc.Unsafe** class

- An alternative of Unsafe as of JDK 9 that provides support for CAS is provided by the **java.lang.invoke.VarHandle** class

```
Unsafe.compareAndSetInt(…)
```

```
VarHandle.compareAndSet (…)
```

Apart from performing CAS operations the Unsafe class also provides the possibility to access off-heap memory but since it is an internal class it Is encapsulated as of JDK 9

Atomic variables

- Maintaining a single variable that is updatable from many threads is a common scalability issue

- Atomic variables already present in the JDK serve as a means to implement updatable variables in a multithreaded environment

- Atomic variables are part of the **java.util.concurrent.atomic** package

Atomic variables

- They make use of the CAS support provided by the JDK to provide performant thread-safe operations over shared variables

- The other utilities that we already discussed that make use of CAS are concurrent collections like **ConcurrentHashMap**

Atomic variables

- Atomic variables are provided by the following classes:
  - AtomicBoolean
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicLong
  - AtomicLongArray
  - AtomicReference
  - AtomicReferenceArray
  - DoubleAccumulator
  - DoubleAdder
  - LongAccumulator
  - LongAdder

Atomic variables

```
AtomicInteger value = new AtomicInteger();
Thread thread = new Thread( () -> {
       value.getAndIncrement();
} );
thread.start();
value.getAndAdd(10);
thread.join();
System.out.println(value.get());
```

Atomic variables

```
DoubleAccumulator accumulator =
       new DoubleAccumulator((x, y) -> x + y , 0);
Thread thread = new Thread( () -> {
       accumulator.accumulate(0.9);
} );
thread.start();
accumulator.accumulate(10.1);
thread.join();
System.out.println(accumulator.get());
```

Atomic variables

```
AtomicReference<String> reference =
       new AtomicReference<String>("first");
Thread thread = new Thread( () -> {
             reference.getAndAccumulate(" someText",
             (x, y) -> {return x + y;});
       } );
thread.start();
reference.getAndAccumulate(" otherText",
       (x, y) -> {return x + y;});
thread.start();
thread.join();
System.out.println(reference.get());
```

Concurrency utilities

Fork/Join framework

- **ForkJoinPool** is an implementation of the **ExecutorService** interface introduced in JDK 7

- Provides the possibility to execute tasks that can be organized in a divide-and-conquer manner

- Tasks submitted to the **ForkJoinPool** are represented by a **ForkJoinTask** instance

Fork/Join framework

- Typical implementations of parallel tasks do not extend directly **ForkJoinTask**

- **RecursiveTask** instances can be used to execute parallel tasks that return a result

- **RecursiveAction** instances can be used to execute parallel tasks that do not return a result

Fork/Join framework

- A global **ForkJoinPool** instance is used for any ForkJoinTasks that are not submitted to a particular **ForkJoinPool**

- To get a reference to the global ForkJoinPool instance the static **ForkJoinPool.commonPool()** method can be used

- This is the case with **parallel streams**: they make use of the common ForkJoinPool for task execution

Fork/Join framework

- A global **ForkJoinPool** instance is used for any ForkJoinTasks that are not submitted to a particular **ForkJoinPool**

- To get a reference to the global ForkJoinPool instance the static **ForkJoinPool.commonPool()** method can be used

- This is the case with **parallel streams**: they make use of the common ForkJoinPool for task execution

Fork/Join framework

```
public class NumberFormatAction extends RecursiveAction {
	…
	@Override
	protected void compute() {
		if (end - start <= 2) {
			System.out.println(start + " " + end);
		} else {
			int mid = start + (end - start) / 2;
			NumberFormatAction left =
			   new NumberFormatAction(start, mid);
			NumberFormatAction right =
			   new NumberFormatAction(mid + 1, end);
			invokeAll(left, right);
		}
	}
}
```

Fork/Join framework

```
public static void main(String[] args) {
        NumberFormatAction action =
                new NumberFormatAction(1, 50);
        ForkJoinPool.commonPool().invoke(action);
}
```

Parallel streams

- Parallel streams can be created as regular streams with the **parallelStream** method

```
list.parallelStream()
```

- Parallel streams should be used only for time-intensive tasks rather than IO

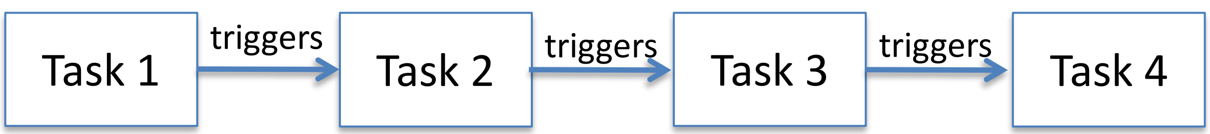
- In many cases regular streams outperform parallel ones so they need to be used with caution

When using parallel streams always measure performance with standard streams

CompletableFuture

- Provides a facility to create a chain of dependent non-blocking tasks

- An asynchronous task can be triggered as the result of a completion of another task



- A CompletableFuture may be completed/cancelled by a thread prematurely

CompletableFuture

- Provides a very flexible API that allows additionally to:
  - combine the result of multiple tasks in a CompletableFuture
  - provide synchronous/asynchronous callbacks upon completion of a task
  - provide a CompletableFuture that executes when first task in group completes
  - provide a CompletableFuture that executes when all tasks in a group complete

CompletableFuture

```
CompletableFuture<Integer> task1 = new
       CompletableFuture<Integer>();

// forcing completing of future by
specifying result
task1.complete(10);
```

CompletableFuture

```
CompletableFuture<Integer> task1 =
        CompletableFuture.supplyAsync(
        () -> { … return 10;});

// executed on completion of the future
task1.thenApply((x) -> {…});

// executed in case of exception or completion
// of the future
task1.handle((x, y) -> {…});

// can be completed prematurely with a result
// task1.complete(20);
System.err.println(task1.get());
```

CompletableFuture

```
CompletableFuture<Object> prev = null;
Supplier<Object> supplier = () -> { … };

for (int i = 0; i < count; i++) {
        CompletableFuture<Object> task;
        if (prev != null) {
                task = prev.thenCompose(
                    (x) -> { return CompletableFuture
                        .supplyAsync(supplier);});
        } else {
                task = CompletableFuture
                        .supplyAsync(supplier);
        }
        prev = task;
}
prev.get();
```

A few more things …

- There is a **Timer** utility that can be used for scheduling tasks but a scheduled executor thread pool is more preferable as a more robust alternative

- A **ThreadLocalRandom** utility is provided since JDK 7 that can be used to generate random numbers for the current thread

- A **Flow** class introduced in JDK 9 provides a reactive streams specification for the JDK that can be used to provide a publish-subscribe

Testing concurrent applications

Testing concurrent applications

- Testing concurrent applications for correctness is inherently difficult
- In many cases non-deterministic unit tests are written that try to simulate running a particular piece of code in a multithreaded manner
- Testing frameworks provide certain utilities that can be used to facilitate testing of concurrent applications

Testing concurrent applications

- A classic way of running multiple threads against code-under-test in a unit test is to use CountDownLatch:

```
@RepeatedTest(10)
public void testLinkedList()
        throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(100);
        for(int i = 0; i < 100; i++) {
                new Thread( () -> {
                        // unit under test …
                        latch.countDown();
                }).start();
        }
        latch.await();
        // perform asserts
}
```

Testing concurrent applications

- Frameworks like **ThreadWeaver** provide the possibility to interleave execution threads
- Interleaving happens using breakpoints (line by line) so that the unit-under-test is tested using different thread ordering
- For performance testing a framework like **JMH** can be used to performs proper application warmup before the test is executed

Questions ?