Java 8 Concurrency Tutorial: Synchronization and Locks

April 30, 2015

Welcome to the second part of my Java 8 Concurrency Tutorial out of a series of guides teaching multi-threaded programming in Java 8 with easily understood code examples. In the next 15 min you learn how to synchronize access to mutable shared variables via the synchronized keyword, locks and semaphores.

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

The majority of concepts shown in this article also work in older versions of Java. However the code samples focus on Java 8 and make heavy use of lambda expressions and new concurrency features. If you're not yet familiar with lambdas I recommend reading my Java 8 Tutorial first.

For simplicity the code samples of this tutorial make use of the two helper methods `sleep(seconds)` and `stop(executor)` as defined here.

Synchronized

In the previous tutorial we've learned how to execute code in parallel via executor services. When writing such multi-threaded code you have to pay particular attention when accessing shared mutable variables concurrently from multiple threads. Let's just say we want to increment an integer which is accessible simultaneously from multiple threads.

We define a field `count` with a method `increment()` to increase count by one:

```
int count = 0;

void increment() {
    count = count + 1;
}
```

When calling this method concurrently from multiple threads we're in serious trouble:

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000)
    .forEach(i -> executor.submit(this::increment));

stop(executor);

System.out.println(count);  // 9965
```

Instead of seeing a constant result count of 10000 the actual result varies with every execution of the above code. The reason is that we share a mutable variable upon different threads without synchronizing the access to this variable which results in a race condition.

Three steps have to be performed in order to increment the number: (i) read the current value, (ii) increase this value by one and (iii) write the new value to the variable. If two threads perform these steps in parallel it's possible that both threads perform step 1 simultaneously thus reading the same current value. This results in lost writes so the actual result is lower. In the above sample 35 increments got lost due to concurrent unsynchronized access to count but you may see different results when executing the code by yourself.

Luckily Java supports thread-synchronization since the early days via the `synchronized` keyword. We can utilize `synchronized` to fix the above race conditions when incrementing the count:

```
synchronized void incrementSync() {
    count = count + 1;
}
```

When using `incrementSync()` concurrently we get the desired result count of 10000. No race conditions occur any longer and the result is stable with every execution of the code:

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000)
    .forEach(i -> executor.submit(this::incrementSync));
```

```
stop(executor);

System.out.println(count);  // 10000
```

The `synchronized` keyword is also available as a block statement.

```
void incrementSync() {
    synchronized (this) {
        count = count + 1;
    }
}
```

Internally Java uses a so called *monitor* also known as monitor lock or intrinsic lock in order to manage synchronization. This monitor is bound to an object, e.g. when using synchronized methods each method share the same monitor of the corresponding object.

All implicit monitors implement the *reentrant* characteristics. Reentrant means that locks are bound to the current thread. A thread can safely acquire the same lock multiple times without running into deadlocks (e.g. a synchronized method calls another synchronized method on the same object).

Locks

Instead of using implicit locking via the `synchronized` keyword the Concurrency API supports various explicit locks specified by the `Lock` interface. Locks support various methods for finer grained lock control thus are more expressive than implicit monitors.

Multiple lock implementations are available in the standard JDK which will be demonstrated in the following sections.

ReentrantLock

The class `ReentrantLock` is a mutual exclusion lock with the same basic behavior as the implicit monitors accessed via the `synchronized` keyword but with extended capabilities. As the name suggests this lock implements reentrant characteristics just as implicit monitors.

Let's see how the above sample looks like using `ReentrantLock`:

```
ReentrantLock lock = new ReentrantLock();
int count = 0;

void increment() {
```

```
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
```

A lock is acquired via `lock()` and released via `unlock()`. It's important to wrap your code into a `try/finally` block to ensure unlocking in case of exceptions. This method is thread-safe just like the synchronized counterpart. If another thread has already acquired the lock subsequent calls to `lock()` pause the current thread until the lock has been unlocked. Only one thread can hold the lock at any given time.

Locks support various methods for fine grained control as seen in the next sample:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
ReentrantLock lock = new ReentrantLock();

executor.submit(() -> {
    lock.lock();
    try {
        sleep(1);
    } finally {
        lock.unlock();
    }
});

executor.submit(() -> {
    System.out.println("Locked: " + lock.isLocked());
    System.out.println("Held by me: " + lock.isHeldByCurrentThread());
    boolean locked = lock.tryLock();
    System.out.println("Lock acquired: " + locked);
});

stop(executor);
```

While the first task holds the lock for one second the second task obtains different information about the current state of the lock:

```
Locked: true
Held by me: false
Lock acquired: false
```

The method `tryLock()` as an alternative to `lock()` tries to acquire the lock without pausing the current thread. The boolean result must be used to check if the lock has actually been acquired before accessing any shared mutable variables.

ReadWriteLock

The interface `ReadWriteLock` specifies another type of lock maintaining a pair of locks for read and write access. The idea behind read-write locks is that it's usually safe to read mutable variables concurrently as long as nobody is writing to this variable. So the read-lock can be held simultaneously by multiple threads as long as no threads hold the write-lock. This can improve performance and throughput in case that reads are more frequent than writes.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
ReadWriteLock lock = new ReentrantReadWriteLock();

executor.submit(() -> {
    lock.writeLock().lock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.writeLock().unlock();
    }
});
```

The above example first acquires a write-lock in order to put a new value to the map after sleeping for one second. Before this task has finished two other tasks are being submitted trying to read the entry from the map and sleep for one second:

```
Runnable readTask = () -> {
    lock.readLock().lock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.readLock().unlock();
    }
};

executor.submit(readTask);
executor.submit(readTask);

stop(executor);
```

When you execute this code sample you'll notice that both read tasks have to wait the whole second until the write task has finished. After the write lock has been released both read tasks are executed in parallel and print the result simultaneously to the console. They don't have to wait for each other to finish because read-locks can safely be acquired concurrently as long as no write-lock is held by another thread.

StampedLock

Java 8 ships with a new kind of lock called `StampedLock` which also support read and write locks just like in the example above. In contrast to `ReadWriteLock` the locking methods of a `StampedLock` return a stamp represented by a `long` value. You can use these stamps to either release a lock or to check if the lock is still valid. Additionally stamped locks support another lock mode called *optimistic locking*.

Let's rewrite the last example code to use `StampedLock` instead of `ReadWriteLock`:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.unlockWrite(stamp);
    }
});

Runnable readTask = () -> {
    long stamp = lock.readLock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.unlockRead(stamp);
    }
};

executor.submit(readTask);
executor.submit(readTask);

stop(executor);
```

Obtaining a read or write lock via `readLock()` or `writeLock()` returns a stamp which is later used for unlocking within the finally block. Keep in mind that stamped locks don't implement reentrant characteristics. Each call to lock returns a new stamp and blocks if no lock is available even if the same thread already holds a lock. So you have to pay particular attention not to run into deadlocks.

Just like in the previous `ReadWriteLock` example both read tasks have to wait until the write lock has been released. Then both read tasks print to the console simultaneously because multiple reads doesn't block each other as long as no write-lock is held.

The next example demonstrates *optimistic locking*:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.tryOptimisticRead();
    try {
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        sleep(1);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        sleep(2);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
    } finally {
        lock.unlock(stamp);
    }
});

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        System.out.println("Write Lock acquired");
        sleep(2);
    } finally {
        lock.unlock(stamp);
        System.out.println("Write done");
    }
});

stop(executor);
```

An optimistic read lock is acquired by calling `tryOptimisticRead()` which always returns a stamp without blocking the current thread, no matter if the lock is actually available. If there's already a write lock active the returned stamp equals zero. You can always check if a stamp is valid by calling `lock.validate(stamp)`.

Executing the above code results in the following output:

```
Optimistic Lock Valid: true
Write Lock acquired
Optimistic Lock Valid: false
Write done
Optimistic Lock Valid: false
```

The optimistic lock is valid right after acquiring the lock. In contrast to normal read locks an optimistic lock doesn't prevent other threads to obtain a write lock instantaneously. After sending the first thread to sleep for one second the second thread obtains a write lock without

waiting for the optimistic read lock to be released. From this point the optimistic read lock is no longer valid. Even when the write lock is released the optimistic read locks stays invalid.

So when working with optimistic locks you have to validate the lock every time *after* accessing any shared mutable variable to make sure the read was still valid.

Sometimes it's useful to convert a read lock into a write lock without unlocking and locking again. `StampedLock` provides the method `tryConvertToWriteLock()` for that purpose as seen in the next sample:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.readLock();
    try {
        if (count == 0) {
            stamp = lock.tryConvertToWriteLock(stamp);
            if (stamp == 0L) {
                System.out.println("Could not convert to write lock");
                stamp = lock.writeLock();
            }
            count = 23;
        }
        System.out.println(count);
    } finally {
        lock.unlock(stamp);
    }
});

stop(executor);
```

The task first obtains a read lock and prints the current value of field `count` to the console. But if the current value is zero we want to assign a new value of `23`. We first have to convert the read lock into a write lock to not break potential concurrent access by other threads. Calling `tryConvertToWriteLock()` doesn't block but may return a zero stamp indicating that no write lock is currently available. In that case we call `writeLock()` to block the current thread until a write lock is available.

Semaphores

In addition to locks the Concurrency API also supports counting semaphores. Whereas locks usually grant exclusive access to variables or resources, a semaphore is capable of maintaining whole sets of permits. This is useful in different scenarios where you have to limit the amount concurrent access to certain parts of your application.

Here's an example how to limit access to a long running task simulated by `sleep(5)`:

```
ExecutorService executor = Executors.newFixedThreadPool(10);

Semaphore semaphore = new Semaphore(5);

Runnable longRunningTask = () -> {
    boolean permit = false;
    try {
        permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
        if (permit) {
            System.out.println("Semaphore acquired");
            sleep(5);
        } else {
            System.out.println("Could not acquire semaphore");
        }
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    } finally {
        if (permit) {
            semaphore.release();
        }
    }
}

IntStream.range(0, 10)
    .forEach(i -> executor.submit(longRunningTask));

stop(executor);
```

The executor can potentially run 10 tasks concurrently but we use a semaphore of size 5, thus limiting concurrent access to 5. It's important to use a `try/finally` block to properly release the semaphore even in case of exceptions.

Executing the above code results in the following output:

```
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
```

The semaphores permits access to the actual long running operation simulated by `sleep(5)` up to a maximum of 5. Every subsequent call to `tryAcquire()` elapses the maximum wait