Using Locks and Semaphores for the Producer / Consumer Pattern

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Agenda

About synchronization

Synchronized & volatile: intrinsic locking

Explicit locking with Lock

Wait / notify with Lock and Condition

Semaphores

Intrinsic / Explicit Locking

```java
public class Person {
   private final Object key = new Object();

   public String init() {
      synchronized(key) {
         // do some stuff
      }
   }
}
```

This code prevents more than one thread to execute the synchronized block at the same time

```java
public class Person {
   private final Object key = new Object();

   public String init() {
      synchronized(key) {
         // do some stuff
      }
   }
}
```

What happens if several threads are trying to execute the init() block?

One of them will be allowed in the block, the others will have to wait for their turn

```java
public class Person {
   private final Object key = new Object();

   public String init() {
      synchronized(key) {
         // do some stuff
      }
   }
}
```

What happens if a thread is “blocked” inside the block?

All the other threads are also blocked

There is no way to release them…

The Lock Pattern Brings a Richer API to Handle This Case

```java
Object key = new Object();

synchronized(key) {
   // do some stuff
}
```

Instead of writing this code

```java
Lock lock = new ReentrantLock();
try {
   lock.lock();
   // do some stuff
} finally {
   lock.unlock();
}
```

We write this code

Lock is an interface, implemented by ReentrantLock

It offers the same guarantees (exclusion, read & write ordering)

… and more functionalities!

Let Us Analyze This Pattern

Instead of passing an instance of Object

To guard several blocks of code

We create a lock object

With methods on it

Interruptible lock acquisition

Timed lock acquisition

Fair lock acquisition

```java
Lock lock = new ReentrantLock();
try {
    lock.lock();
    // do some stuff
} finally {
    lock.unlock();
}
```

This is the basic pattern

```java
Lock lock = new ReentrantLock();
try {
   lock.lockInterruptibly();
   // do some stuff
} finally {
   lock.unlock();
}
```

This is the interruptible pattern

The thread will wait until it can enter the guarded block of code

But another thread can interrupt it by calling its interrupt() method

This can be costly, or hard to achieve though…

Interruptible lock acquisition

Timed lock acquisition

Fair lock acquisition

```java
Lock lock = new ReentrantLock();
if (lock.tryLock()) {
  try {
    // guarded block of code
  } finally {
    lock.unlock();
  }
} else { ... }
```

This is a timed lock acquisition

If a thread is already executing the guarded block of code

Then tryLock() returns false, immediately

```java
Lock lock = new ReentrantLock();
if (lock.tryLock(1, TimeUnit.SECONDS)) {
    try {
        // guarded block of code
    } finally {
        lock.unlock();
    }
} else { ... }
```

This is a timed lock acquisition

If a thread is already executing the guarded block of code

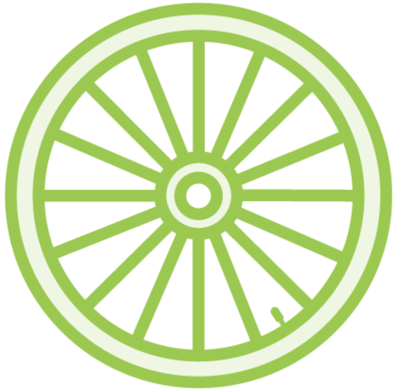
Then tryLock() returns false, immediately

One can also pass a time out as a parameter

Interruptible lock acquisition

Timed lock acquisition

Fair lock acquisition



When several threads are waiting for a lock

Whether it is an intrinsic or explicit lock…

The first one to enter the guarded block of code is chosen randomly

Fairness means that the first to enter the wait line is the first to enter the block of code

```java
Lock lock = new ReentrantLock();
try {
   lock.lock();
   // guarded block of code
} finally {
   lock.unlock();
}
```

A ReentrantLock built in the normal way is non-fair

```java
Lock lock = new ReentrantLock(true); // fair
try {
   lock.lock();
   // guarded block of code
} finally {
   lock.unlock();
}
```

A ReentrantLock built in the normal way is non-fair

And one can also pass a boolean to it

True: this lock is fair, false: this lock is non-fair

A fair lock is costly…

Using Lock Gives Wiggle Room

A lock can be interrupted: possible but hard, costly

It can be blocked for a set amount of time

It can be fair, letting in the threads on a first come first served basis

Implementing the Producer / Consumer Pattern

Intrinsic locks: the producer / consumer pattern is implemented with wait / notify

Obviously, wait / notify cannot work with explicit locking

So we need another pattern!

```
Object lock = new Object();
```

```
class Producer {
  public void produce() {
    synchronized(lock) {
      while (isFull(buffer))
        lock.wait();
      buffer[count++] = 1;
      lock.notifyAll();
    }
  }
}
```

```
Object lock = new Object();
```

```
class Producer {
  public void produce() {
    synchronized(lock) {
      while (isFull(buffer))
        lock.wait();
      buffer[count++] = 1;
      lock.notifyAll();
    }
  }
}
```

```
class Consumer {
  public void consume() {
    synchronized(lock) {
      while (isEmpty(buffer))
        lock.wait();
      buffer[--count] = 0;
      lock.notifyAll();
    }
  }
}
```

```java
Object lock = new Object();
```

```java
class Producer {
  public void produce() {
    synchronized(lock) {
      while (isFull(buffer))
        lock.wait();
      buffer[count++] = 1;
      lock.notifyAll();
    }
  }
}
```

```java
class Consumer {
  public void consume() {
    synchronized(lock) {
      while (isEmpty(buffer))
        lock.wait();
      buffer[--count] = 0;
      lock.notifyAll();
    }
  }
}
```

```
Object lock = new Object();
```

```
class Producer {
  public void produce() {
    synchronized(lock) {
      while (isFull(buffer))
        lock.wait();
      buffer[count++] = 1;
      lock.notifyAll();
    }
  }
}
```

```
class Consumer {
  public void consume() {
    synchronized(lock) {
      while (isEmpty(buffer))
        lock.wait();
      buffer[--count] = 0;
      lock.notifyAll();
    }
  }
}
```

```java
Lock lock = new ReentrantLock();
```

```java
class Producer {
  public void produce() {
    try {
      lock.lock();
      while (isFull(buffer))
        // wait
      buffer[count++] = 1;
      // notify
    } finally {
      lock.unlock();
    }
  }
}
```

```java
Lock lock = new ReentrantLock();
```

```java
class Producer {
  public void produce() {
    try {
      lock.lock();
      while (isFull(buffer))
        // wait
      buffer[count++] = 1;
      // notify
    } finally {
      lock.unlock();
    }
  }
}
```

```java
class Consumer {
  public void consume() {
    try {
      lock.lock();
      while (isEmpty(buffer))
        // wait
      buffer[--count] = 0;
      // notify
    } finally {
      lock.unlock();
    }
  }
}
```

```java
Lock lock = new ReentrantLock();
```

```java
class Producer {
  public void produce() {
    try {
      lock.lock();
      while (isFull(buffer))
        // wait
      buffer[count++] = 1;
      // notify
    } finally {
      lock.unlock();
    }
  }
}
```

```java
class Consumer {
  public void consume() {
    try {
      lock.lock();
      while (isEmpty(buffer))
        // wait
      buffer[--count] = 0;
      // notify
    } finally {
      lock.unlock();
    }
  }
}
```

```java
Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
```

```java
class Producer {
  public void produce() {
    try {
      lock.lock();
      while (isFull(buffer))
        notFull.await();
      buffer[count++] = 1;
      // notify
    } finally {
      lock.unlock();
    }
  }
}
```

```java
class Consumer {
  public void consume() {
    try {
      lock.lock();
      while (isEmpty(buffer))
        // wait
      buffer[--count] = 0;
      notFull.signal();
    } finally {
      lock.unlock();
    }
  }
}
```

```java
Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();
```

```java
class Producer {
  public void produce() {
    try {
      lock.lock();
      while (isFull(buffer))
        notFull.await();
      buffer[count++] = 1;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
}
```

```java
class Consumer {
  public void consume() {
    try {
      lock.lock();
      while (isEmpty(buffer))
        notEmpty.await();
      buffer[--count] = 0;
      notFull.signal();
    } finally {
      lock.unlock();
    }
  }
}
```

```java
Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();
```

```java
class Producer {
  public void produce() {
    try {
      lock.lock();
      while (isFull(buffer))
        notFull.await();
      buffer[count++] = 1;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
}
```

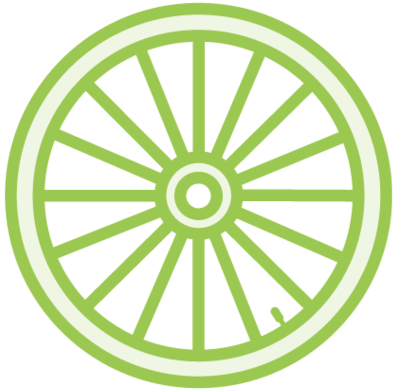
```java
class Consumer {
  public void consume() {
    try {
      lock.lock();
      while (isEmpty(buffer))
        notEmpty.await();
      buffer[--count] = 0;
      notFull.signal();
    } finally {
      lock.unlock();
    }
  }
}
```

Condition

A Condition object is used to park and awake threads

A Lock object can have any number of Conditions

A Condition object extends Object, so it has wait() and notify() methods… not to be taken for await() and signal()!



The await() call is blocking, can be interrupted

There are five versions for await:
- await()
- await(time, timeUnit)
- awaitNanos(nanosTimeout)
- awaitUntil(date)
- awaitUninterruptibly()

These are ways to prevent the blocking of waiting threads with the Condition API

A fair Lock generates fair Condition

Lock & Condition

Another implementation of the wait / notify pattern

It gives wiggle room to build better concurrent systems

Controlled interruptability, time out, fairness

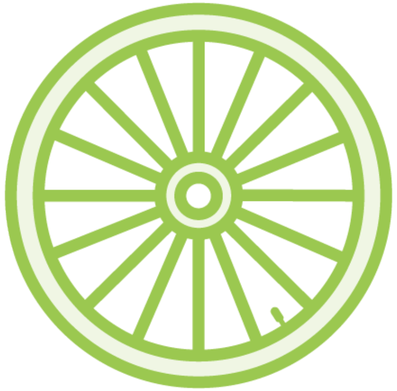
Read / Write Locks

In some cases what we need is exclusive writes

And allow parallel reads

This is not how regular locks work

This is what read / write lock does

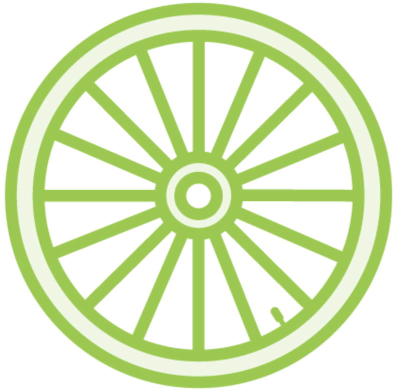


ReadWriteLock is an interface with only two methods

readLock() to get a read lock

writeLock() to get a write lock

Both are instances of Lock



Only one thread can hold the write lock

When the write lock is held, no one can hold the read lock

As many threads as needed can hold the read lock

```java
ReadWriteLock readWritelock = new ReentrantReadWriteLock();

Lock readLock = readWritelock.readLock();
Lock writeLock = readWritelock.writeLock();
```

The right pattern is to create a read / write lock

Then get a read and a write lock as a pair from this read / write lock

```java
Map<Long, User> cache = new HashMap<>();

try {
   readLock.lock();
   return cache.get(key);
} finally {
   readLock.unlock();
}
```

It can be used to create a thread safe cache

Read-locking the get operation allows for concurrent reads

```java
Map<Long, User> cache = new HashMap<>();

try {
   writeLock.lock();
   cache.put(key, value);
} finally {
   writeLock.unlock();
}
```

It can be used to create a thread safe cache

Write-locking the put operation protects the non-concurrent map against internal corruption

It can also be achieved with a ConcurrentHashMap

Read / Write Locks

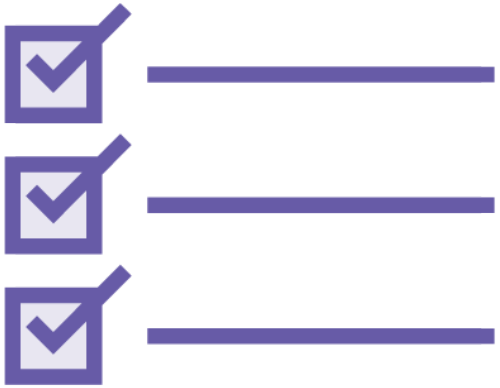
Works with a single read / write lock object used to get a write lock and a read lock

Write operations are exclusive of other writes and reads

Read operations can be made in parallel

It thus allows for superior throughput

Semaphores



A Semaphore is a well-known concept in concurrent programming

It looks like a lock, but allows several threads in the same block of code

```java
Semaphore semaphore = new Semaphore(5); // permits
try {
   semaphore.acquire();
   // guarded block of code
} finally {
   semaphore.release();
}
```

A Semaphore built in the normal way is non-fair

The acquire() call is blocking until a permit is available

At most 5 threads can execute the guarded code at the same time

```java
Semaphore semaphore = new Semaphore(5, true); // fair
try {
    semaphore.acquire(2);
    // guarded block of code
} finally {
    semaphore.release(2);
}
```

A Semaphore can be fair

The acquire() can ask for more than one permit

Then the release() call must release them all

```java
Semaphore semaphore = new Semaphore(5);
try {
   semaphore.acquireUninterruptibly();
   // guarded block of code
} finally {
   semaphore.release();
}
```

By default if a waiting thread is interrupted it will throw an InterruptedException

Uninterruptibility means that the thread cannot be interrupted

It can be only be freed by calling its release() method

```java
Semaphore semaphore = new Semaphore(5);
try {
   if (semaphore.tryAcquire())
      // guarded block of code
   else
      // I could not enter the guarded code
} finally {
   semaphore.release();
}
```

Acquisition can be made immediate…

… and can fail if there is already a thread in the guarded block of code

```java
Semaphore semaphore = new Semaphore(5);
try {
  if (semaphore.tryAcquire(1, TimeUnit.SECONDS))
    // guarded block of code
  else
    // I could not enter the guarded code
} finally {
  semaphore.release();
}
```

One can also set a timeout before failing to acquire a permit

This pattern can also request more than one permit

Handling Permits and Waiting Threads

One can reduce the number of permits (cannot increase it)

One can check the waiting threads:
- are there any waiting threads?
- how many threads are waiting?
- get the collection of the waiting threads

Semaphore

It has a number of permits, which can be acquired in different ways, and released

One can query a semaphore for the number of waiting threads

Demo

Let us see some code!

Let us see this producer / consumer pattern in action!

And how read / write lock can be used to create a concurrent cache

Demo Wrapup

What did we see?

How to properly lock a Producer / Consumer with Lock

How to deal with exceptions and timeout

How HashMap can fail in a concurrent application

How to properly synchronize it using a pair of read / write locks

Module Wrapup

What did we learn?

The difference between intrinsic and explicit locking

Give wiggle room to create efficient concurrent applications

Locks, read / write locks

Semaphores

Interruptability, time outs, fairness