KTH Computer Science
and Communication

Concurrency on the JVM

An investigation of strategies for handling concurrency in Java, Clojure and Groovy.

PASCAL CHATTERJEE, JOAKIM CARSELIND

Bachelor's Thesis at NADA
Supervisor: Mads Dam
Examiner: Tjohej

TRITA xxx yyyy-nn

Abstract

This is a skeleton for KTH theses. More documentation regarding the KTH thesis class file can be found in the package documentation.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus. Fusce tempor. Nulla facilisi. Sed at turpis. Phasellus eu ipsum. Nam porttitor laoreet nulla. Phasellus massa massa, auctor rutrum, vehicula ut, porttitor a, massa. Pellentesque fringilla. Duis nibh risus, venenatis ac, tempor sed, vestibulum at, tellus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos.

Referat

Lorem ipsum dolor sit amet, sed diam nonummy nibh eui mod tincidunt ut laoreet dol

Denna fil ger ett avhandlingsskelett. Mer information om LaTeX-mallen finns i dokumentationen till paketet.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus. Fusce tempor. Nulla facilisi. Sed at turpis. Phasellus eu ipsum. Nam porttitor laoreet nulla. Phasellus massa massa, auctor rutrum, vehicula ut, porttitor a, massa. Pellentesque fringilla. Duis nibh risus, venenatis ac, tempor sed, vestibulum at, tellus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos.

Contents

Chapter 1

Introduction

Need for concurrent applications due to more available cores etc etc. Modern, dynamic languages like Ruby and Python feature a *Global Interpreter Lock* (GIL), so we need to use languages such as C/C++/Java to leverage multiple processors. We will focus on the JVM in this paper.

Part I

Introducing concurrency

1 Threads

2 Atomicity

One of the first things we should realise when writing concurrent programs is that most of the statements we use are not **atomic**. This means that although we tend to think of them as indivisible units of computation, they expand to multiple instructions when compiled to bytecode. It is these instructions that are atomic, not the statements we write in high-level JVM languages.

Let us take the simple example of incrementing an integer variable. In Java, we could write the function:

```
1 public static void add(int var, int num) {
2         var = var + num;
3 }
```

Intuitively, we might think that that if a context switch were to occur in our function, it would take place at line 1, 2 or 3. This would be the case if line 2 was atomic, but as it consists of addition *and* assignment, it is compiled to multiple bytecode instructions. We can see these instructions here:

```
1 public static void add(int, int);
2         iload_0
3         iload_1
4         iadd
5         istore_0
6         return
```

The second line from our Java `add` function generates the `iadd` and `istore0` instructions at lines 4 and 5 of the bytecode.

It is entirely possible for a thread switch to occur in between these instructions. Usually this is not problematic at all, and in fact this happens many times a second on all modern operating systems. However, if multiple threads attempt to change the value of the *same* variable at the same time, inconsistencies begin to arise.

3 Shared memory

In order to better illustrate the lack of atomicity in our add function, we can rewrite it to look like this:

```java
1 public static void add(int num)
2 throws InterruptedException {
3         int v = var;
4         Thread.sleep(1);
5         var = v + num;
6 }
```

Here `var` is an instance variable. The `Thread.sleep` at line 4 forces a context switch after `var` has been copied to the local variable `v`. If any other thread alters `var` during this time, those changes will be lost when the original thread resumes and writes `v + num` back to `var`. The following could well happen if two threads were to execute `add` simultaneously:

```java
   // var = 0
   // Thread 1
    int v = var; // v = 0
    Thread.sleep(1);
    // * Context Switch *
        // Thread 2
        int v = var // v = 0
        Thread.sleep(1)
        var = v + 1 // var = 1
        // * Context Switch *
    var = v + 1; // var = 1
```

As we can see, after this interleaving of statements, `var = 1` even though it was incremented *twice*. It is in this way that shared mutable memory, or state, can lead to inconsistent data even though the program logic is correct. We call this phenomenon - when the result of a program is dependent on the sequence or timing of other events - a **race condition**.

All of the concurrency strategies we will discuss in this paper aim to mitigate the effects of race conditions, and thereby ensure that programs behave in a deterministic way despite the activity of multiple threads. They do this by eliminating one of the factors from **uncontrolled access to shared, mutable state** that can lead to problems. The first approach we consider, threads and locks, uses locks to **control access** to shared mutable state.

Part II

Threads and Locks

1 Background

Background about threads (i.e. difference between processes and threads etc) goes here

We will use the example of bank accounts that allow withdrawals, deposits and reading the value of a balance to illustrate the concurrency strategies in this paper. Formally we can view this as an interface, that in Java can be written:

```java
public interface Account {
    public float getBalance();
    public boolean deposit(float amount);
    public boolean withdraw(float amount);
}
```

`getBalance` returns the current balance as a `float`; `deposit` and `withdraw` increment and decrement the current balance respectively, and return a boolean signifying whether they succeeded. `withdraw` can fail if more funds are requested than are present in the balance.

2 No locks

We begin by implementing the `Account` interface in the simplest possible way.

```
public class NaiveAccount implements Account {
    private float balance = 0;

    public float getBalance() {
                    Thread.sleep(1);
        return balance;
    }

    public void deposit(float amount)
    throws InterruptedException {
        float b = balance;
        Thread.sleep(1);
        balance = b + amount;
    }

    public void withdraw(float amount)
    throws InterruptedException {
        float b = balance;
        Thread.sleep(1);
        balance = b - amount;
    }
}
```

As we explained in section 1.3, we insert a `Thread.sleep` in the middle of `deposit` and `withdraw` in order to highlight the danger of context switches.

2.1 Testing correctness

We can (informally) test the correctness of this implementation by initially depositing a certain amount in an account, carrying out a certain number of deposits and withdrawals, and then making sure that the resulting balance is as we expected.

In these tests the initial balance is `10`, and we carry out a sequence of 10 deposits and 10 withdrawals, each for the amount of `1` unit. As these cancel out, our finishing balance should also be `10` as that is what we started with. These operations are themselves carried out 10 times to see what happens when they are repeated.

Single Threaded

The collected final balances of the single threaded tests are shown below:

```
[10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0]
```

As we can see, these final balances are exactly what we would expect given an initial balance of 10, followed by 10 deposits and 10 withdrawals of 1 unit. The results were also unchanged over the course of 10 trials.

Multiple Threaded

Now let's see what happens when we run the withdrawals and the deposits in separate threads.

```
[8.0, 20.0, 20.0, 20.0, 6.0, 0.0, 16.0, 18.0, 8.0, 0.0]
```

Here the final balance ranged between 0 and 20, which is an error of $-10 \leq error \leq 10$. This shows that in some runs all our withdrawals disappeared; in others all our deposits disappeared; and sometimes we saw a mixture of these two extremes. Such disappearances of actions from our results happened when a certain interleaving of statements from the two threads occurred as described in section 1.3.

This makes it very obvious that the `deposit` and `withdraw` methods are **critical sections** - a section of code that should only be executed by one thread at any time. These sections need to be **mutually exclusive** so that we can reason about their effects as if they were atomic actions. Our problems arise only when a thread context switches while leaving the shared, mutable balance variable in an inconsistent state.

3 Locking with `synchronized`

Our first solution to this problem will be to use Java's `synchronized` concept to ensure that even if a context switch occurs within a critical section, other threads are blocked from entering until the currently executing thread completes its actions.

The changes to the code to facilitate this are minimal: we simply insert the keyword `synchronized` into the signature of any method that references the shared variable `balance`. For us, this is all three methods (even `getBalance` which should not be allowed access to `balance` during a `deposit` or `withdraw` as it is by definition inconsistent at that time).

3.1 Testing correctness

Running the multi-threaded test, with simultaneous deposits and withdrawals, yields the results:

```
[10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0]
```

Our problems seem to be solved! Unfortunately, this form of overzealous locking suffers from some performance issues which we will discuss next.

3.2 Performance

Threads attempting to enter `synchronized` methods have to acquire an object's intrinsic lock, or **monitor**, before they can execute any code[1]. This ensures that all `synchronized` methods are mutually exclusive, which is good for our `deposit` and `withdraw` operations, but can be wastfeul for `getBalance`. The difference, of course, is that `deposit` and `withdraw` are mutators whereas `getBalance` is simply an accessor, and while mutators should mutually exclude all other operations, there is no reason why accessors should exclude other accessors as they do not change the state of an object.

We can see the performance implications of this by carrying out a test in which 9 threads execute `getBalance` and 1 thread executes `deposit` in parallel. If this test takes around the same time to complete as the inverse, where 9 threads execute `deposit` and 1 thread executes `getBalance`, then we can conclude that accessors and mutators are all mutually exclusive.

```
Synchronized Read Frenzy: 121.0 ms
Synchronized Write Frenzy: 124.0 ms
```

As there is no perceptible difference between the read frenzy, with more reads than writes, and the write frenzy, with the inverse, we can conclude that both reads and writes have been serialised by the object's monitor which we invoked using `synchronized`.

[1] http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

4 Explicit locks

Luckily for us, Java has included support for more finely-grained locks than an object's monitor since version 1.5. Of these, the `ReentrantReadWriteLock` is most suitable for our purposes. This object actually consists of two locks, a read-lock and a write-lock. The write-lock, like the object's monitor, mutually excludes everything. The read-lock however, allows multiple threads to acquire it at the same time, but still excludes other threads from acquiring the write-lock.

This has the effect of allowing read operations to execute in parallel while serialising writes. The reentrant part of the lock's name signifies that either lock may be acquired multiple times - such as read methods calling other read methods, with no ill effects.

An implementation of a `ReadWriteLockAccount` is as follows:

```java
public class ReadWriteLockAccount implements Account {
    private float balance = 0;
    private final ReentrantReadWriteLock rwl =
                    new ReentrantReadWriteLock();
    private final Lock readLock = rwl.readLock();
    private final Lock writeLock = rwl.writeLock();

    public float getBalance()
    throws InterruptedException {
        readLock.lock();
        try {
            Thread.sleep(1);
            return balance;
        }
        finally { readLock.unlock(); }
    }

    public boolean deposit(float amount)
    throws InterruptedException {
        writeLock.lock();
        try {
            float b = balance;
            Thread.sleep(1);
            balance = b + amount;
            return true;
        } finally { writeLock.unlock(); }
    }

    public boolean withdraw(float amount)
    throws InterruptedException {
        writeLock.lock();
        try {
            if (balance - amount >= 0) {
                float b = balance;
                Thread.sleep(1);
                balance = b - amount;
                return true;
            } else { return false; }
        } finally { writeLock.unlock(); }
    }
}
```

4.1 Performance

Let's see how this Account performs during read and write frenzies.

```
Read-Write-Lock Read Frenzy: 26.0 ms
Read-Write-Lock Write Frenzy: 123.0 ms
```

Now that read operations such as `getBalance` can execute in parallel, the read frenzy test is significantly faster than the write frenzy, in which no parallelisation is possible. Also worth noting is that the write frenzy here took around the same time as our `synchronized` version. This shows that using a `ReadWriteLock` should usually yield *at least as good* performance as the `synchronized` keyword, with performance during heavy read activity receiving the most benefits and heavy write activity staying the same.

4.2 Boilerplate code

One area in which the `synchronized` Account beats the `ReadWriteLock` version is in the amount of boilerplate code that is required to maintain correctness. The `synchronized` Account required only three extra words compared to our original Account, whereas our latest version requires explicit locking and unlocking of specific locks to surround the body of each critical method.

In our example this is not so bad, especially considering the increased performance these locks have given us, but in larger projects the amount of code-overhead introduced by explicit locking can be significant. In fact, code which includes overhead like this is harder to parse (as a programmer), maintain, and is also more fragile, as forgetting to unlock in just one place can introduce severe bugs into a system.

More flexible languages than Java combat this problem by using macros and higher-order functions to abstract away such boilerplate code, as we will see in later sections.

5 Transfers

Now that we have a correct and performant Account implementation, our job seems to be done. As before, things are not quite so simple. Our latest Account implementation appears to work in isolation, but things can get trickier when we bring multiple Accounts into the mix.

Let us imagine that we want to transfer funds between Accounts. A `transfer` method could be defined in some sort of `AccountTransferService` class, which would take 2 Accounts and an amount as input, withdraw `amount` from the first Account, and then deposit `amount` in the second Account. These `transfer`s are also critical sections, as Accounts should not be altered or read mid-transfer as they are in an inconsistent state.

To ensure the integrity of this critical section we have to acquire locks on both Accounts, carry out the transfer, and then release the lock. The object monitor version is shown below (an explicit lock version would acquire the objects' write-locks instead):

```
1  public static void transfer(Account from, Account to,
2  float amount) throws InterruptedException {
3          synchronized(from) {
4              Thread.sleep(1);
5              synchronized(to) {
6                  from.withdraw(amount);
7                  to.deposit(amount);
8              }
9          }
10     }
```

This code will work as expected the vast majority of the time, but there is a case in which not only will the program be incorrect, it will actually hang forever. As you can imagine, this is because of the inconveniently placed `Thread.sleep` on line 4.

Like before, this line forces a context switch that could occur in the normal execution of the code. This is usually not a problem, except for the case in which a transfer **from** Account A **to** Account B, and a transfer **from** Account B **to** Account A occur simultaneously. This will result in the following sequence of events:

1. Transfer 1: Acquires Account A's lock
2. Context switch
3. Transfer 2: Acquires Account B's lock
4. Transfer 1 waits to acquire Account B's lock
5. Transfer 2 waits to acquire Account A's lock

As both transfers are waiting for each other, their threads will block indefinitely in a situation known as **deadlock**.

This dangerous juggling of locks is possibly the greatest problem that arises when using Threads and Locks to manage concurrency. Other concurrency strategies like the ones we will discuss later abstract away the handling of locks, meaning that it is much harder to make mistakes involving them.

6 Composability / Reuse

Removing context-specific locking code also enhances the composability of a system and encourages code reuse etc etc.

Part III

Actors

1 Background

2 A naive version

account sim.groovy -> deadlock.

3 Introducing brokers

4 Active Objects

4.1 Inheritance

4.2 Code reuse

4.3 Problems

transfer is a "fake" transaction.

Part IV

Software Transactional Memory

1 Background

2 Concurrency in Clojure

2.1 Immutability

2.2 Transactions

Here throw an OutOfMoneyException from withdraw instead of returning false <– more semantic.

3 Agents

4 Agents and STM

Part V

Conclusion

Appendix A

RDF

And here is a figure

Figure A.1. Several statements describing the same resource.

that we refer to here: A.1