

Concurrency: Best Practices

11, 12 Dec. 2009

Pune, India

IndicThreads.com Conference On Java Technology

Pramod B Nagaraja

IBM India Pvt LTD

Agenda

- Why Concurrency?
- Concurrency Panoramas
- Synchronization
 - When and What to synchronize
- Guidelines for designing Concurrent Applications
- Excessive Synchronization
- Synchronization Optimization
- Latches & Barriers



1.0 Why Concurrency

Story so far....

- Most of the applications ran on system with few processors
- Relied on faster hardware, rather than *Software Concurrency/Parallelism*, for better performance

1.0 Why Concurrency

...Current scenario

- Moore's Law predicts that the number of transistors on a chip doubles every two years
- Faster Dual core machines demand Concurrency
- Multithreading is the pulse of Java

2.0 Concurrency Panoramas

- Atomicity
 - Certain pieces of an application must all be executed as one unit

2.0 Concurrency Panoramas

- Atomicity

```
synchronized int getBalance() {  
    return balance;  
}
```

```
synchronized void setBalance(int x) {  
    balance = x;}  
}
```

```
void deposit(int x) {  
    int b = getBalance();  
    setBalance(b + x);}  
}
```

```
void withdraw(int x) {  
    int b = getBalance();  
    setBalance(b - x);}  
}
```



2.0 Concurrency Panoramas

- Visibility
 - Changes that you make to a value to be visible precisely when you intend them to be

2.0 Concurrency Panoramas

- Visibility

```
class LoopMayNeverEnd {
```

```
    boolean done = false;
```

```
    void work() {
```

```
        while (!done) {
```

```
            // do work
```

```
        }
```

```
    }
```

```
    void stopWork() {  
        done = true;  
    }
```

```
}
```

3.0 Synchronization

*Things which matter **MOST** should never be at mercy of things which matter **LEAST***

3.1 When to Synchronize

Pitfalls when synchronizing:

- Forgetting to synchronize a resource that should be synchronized
- Synchronizing the wrong resource
- Synchronizing too often



3.2 What to Synchronize

- Not just the resource, but also the Transaction involving the resource
- Pay attention to Data integrity at multiple levels of granularity



3.2 What to Synchronize

```
private int foo;
    public synchronized int getFoo() {
        return foo;
    }
    public synchronized void setFoo(int f) {
        foo = f;
    }
    setFoo(getFoo() + 1); //not thread safe
```

4. Guidelines for designing Concurrent Applications

- 4 common concurrency mechanisms generally used in combination with each other to safely access data from multiple threads
 - Dynamic Exclusion Through Implicit Locks
 - Structural Exclusion Through Confinement
 - Immutability
 - Cooperation

4.1 Dynamic Exclusion Through Implicit Locks

Lock associated with an object is acquired when a method is declared synchronized.

```
public synchronized void setName(String name);
```

- *Limitation*

- When synchronization is used inappropriately or excessively it causes poor performance and deadlock.



4.1 Dynamic Exclusion Through Implicit Locks

- Best Practice Tip

- ✓ Do not perform CPU intensive and I/O operations inside synchronized method.
- ✓ When possible synchronize on block of code instead of synchronizing entire method.

Ex: `synchronize(lock) { //critical section guarded by lock }`

4.2 Structural Exclusion Through Confinement

- **Thread confinement** : Access object from a single thread using thread-per-session approach
- **Instance confinement** : Use encapsulation techniques to restrict access to object state from multiple threads
- **Method confinement** : Do not allow an object to escape from method by referencing it through local variables

4.2 Structural Exclusion Through Confinement

- **Limitations :**
 - Confinement cannot be relied on unless a design is leak proof
- **Best Practice Tip :**
 - ✓ Combine confinement with appropriate locking discipline
 - ✓ Use ThreadLocal variables to maintain Thread Confinement

4.3 Immutability

- Immutable objects do not need additional synchronization

```
Public final class ImmutableClass {  
    private final int field1 = 100;  
    private final String field2="FIN";  
    .....  
}
```

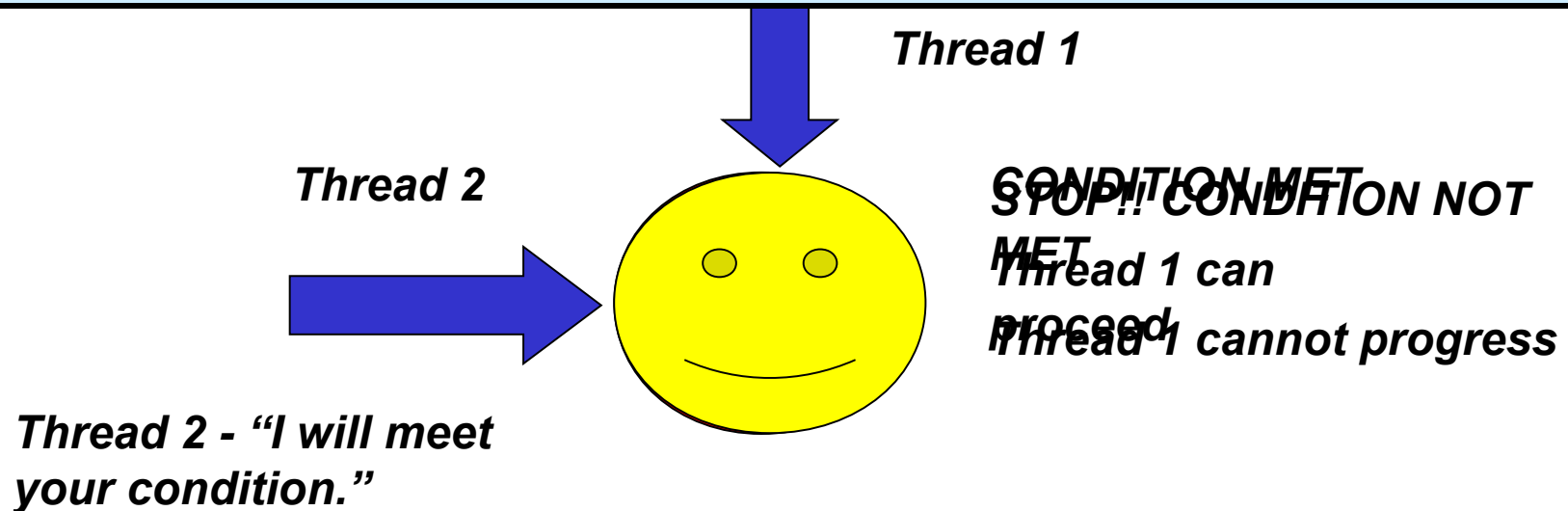
Inherently Thread-safe

4.3 Immutability

- **Limitation**
 - In a software application only some classes can be immutable.
- **Best Practice Tip**
 - Even when class is not fully immutable, declare its non changeable fields as final to limit its mutability.

4.4 Cooperation

What is purpose of wait-and-notify mechanism?
When one thread needs a certain condition to exist and another thread can create that condition then we use wait and notify methods.



4.4 Cooperation

Limitation

- When there are multiple waiting threads, you cannot be certain which thread is woken up.
- Due to race conditions there could be missed or early notifications.

Best Practice Tip

- ✓ Safe to wake up all the threads using `notifyAll()` method instead of waking an arbitrary thread using `notify()`
- ✓ Sometimes condition that caused thread to wait is not achieved when `wait()` returns, therefore put `wait()` call in a while loop.
- ✓ `Wait()` and `notify()` must always be called inside synchronized code.

5. Excessive Synchronization

- Poor performance
- Deadlock
 - Can occur when multiple threads each acquire multiple locks in different orders

5.1 Deadlock

```
public static Object cacheLock = new Object();
```

```
public static Object tableLock = new Object();
```

```
...
```

```
public void oneMethod() {  
    synchronized (cacheLock) {  
        synchronized (tableLock) {  
            doSomething(); } } }
```

```
public void anotherMethod() {  
    synchronized (tableLock) {  
        synchronized (cacheLock) {  
            doSomethingElse();  
        } } }
```

Need not be this obvious !!!!



5.1 Deadlock

```
public void transferMoney(Account fromAccn,  
    Account toAccn, Amount amnt) {  
    synchronized (fromAccn) {  
        synchronized (toAccn) {  
            if (fromAccn.hasSufficientBalance(amnt) {  
                fromAccn.debit(amnt);  
                toAccn.credit(amnt);}}}
```

Thread A : transferMoney(accountOne, accountTwo, amount);

Thread B : transferMoney(accountTwo, accountOne, amount);



5.1 Deadlock

How to avoid Deadlock

- Avoid acquiring more than one lock at a time
- Allow a thread to voluntarily give up its resources if a second level or successive lock acquisition fails, this is called *Two Phase Locking*
- Never call a synchronized method of another class from a synchronized method
- Follow a fixed order while acquiring and releasing locks
- Induce Lock ordering, if required
 - `Object.identityHashCode()` method



5.1 Deadlock

Banking example revisited

```
public void transferMoney(Account fromAccn,  
Account toAccn, Amount amnt) {  
    Account firstLck, secondLck;  
    if (fromAccn.accountNumber() <  
        toAccn.accountNumber()) {  
        firstLck = fromAccn;  
        secondLck = toAccn;  
    } else {  
        firstLck = toAccn;  
        secondLck = fromAccn; }  
}
```

5.1 Deadlock

```
synchronized (firstLck) {  
    synchronized (secondLck) {  
        if (fromAccn.hasSufficientBalance (amnt) {  
            fromAccn.debit (amnt);  
            toAccn.credit (amnt);    }  
        }  
    }  
}
```

6. Synchronization Optimization

JVM optimizes synchronization [under the hood]:

- Lock Elision
- Biased Locking
- Lock Coarsening
- Thread Spinning vs Thread Suspending

6.1 Synchronization Optimization

- Lock Elision :: Locks on local scope references can be elided

```
public String concatBuffer(String s1,String s2,String s3) {  
    StringBuffer sb = new StringBuffer();  
    sb.append(s1);  
    sb.append(s2);  
    sb.append(s3);  
    return sb.toString(); }
```

6.2 Synchronization Optimization

- Biased Locking :: Most locks are never accessed by more than one thread during their life time
 - Release the lease only if another thread attempts to acquire the same lock
 - Default in Java 6 Hotspot/JIT

6.3 Synchronization Optimization

- Lock coarsening :: Adjacent synchronized blocks may be merged into one synchronized block

```
public static String concatToBuffer(StringBuffer sb, String s1,  
    String s2, String s3) {  
    sb.append(s1);  
    sb.append(s2);  
    sb.append(s3);  
    return sb.toString();}
```

6.4 Synchronization Optimization

- Thread Suspending vs Thread Spinning
- In 1.4.2, spin for (default value of) 10 iterations before being suspended
- Adaptive Spinning
 - Introduced in 1.6
 - Spin duration based on the previous spin attempts and state of the lock owner

7.1 Latches

- Latches allows one or more threads to wait until a set of operations being performed in other threads completes.
- When N concurrent sub-tasks occur, the coordinating thread will wait until each sub-task is executed

7.1 Latches

```
class TestLatch {  
    final CountdownLatch latch = new CountdownLatch(3);  
    class myThread extends Thread {  
        public void run() { task.run();  
            latch.countDown();}  
    }  
    public static void main (String[] args){  
        TestLatch tst = new TestLatch();  
        tst.new myThread().start();  
        tst.new myThread().start();  
        tst.new myThread().start();  
        latch.await();//Thread execution completed}}  
}
```

7.2 Barriers

- Barrier allows a set of threads to wait for each other at a common barrier point.
- When Barrier point occurs all the threads waiting on it are released and run() method of Barrier object is called.

```
class TestBarrier {  
    final static int no_of_engines = 4;  
    Runnable task = new Runnable() {  
        public void run() { flyplane() }}  
}
```

7.2 Barriers

```
final CyclicBarrier barrier = new CyclicBarrier(no_of_engines,task);  
  
class Worker implements Runnable {  
    public void run() {  
        startEngine();  
        barrier.await();} }  
  
public static void main(String[] args) {  
    TestBarrier flight = new TestBarrier ();  
    for (int i = 0; i < no_of_engines; ++i)  
        new Thread(flight.new Worker(i)).start();    }}  

```

References

- <https://www6.software.ibm.com/developerworks/education/j-concur/index.html>
- <http://www.infoq.com/articles/java-threading-optimizations-p1>
- <http://www.infoq.com/presentations/goetz-concurrency-past-present>

Q&A

Thank You !!!!