**JAVA MTA**  
Multi-threading with Annotations

**What is Multi-threading**

A thread of execution is the smallest sequence of programmed instructions that can be managed independently.  
Multi-threading is the ability of a program or an operating system to execute concurrently multiple threads of execution.  
Multi-threading improves performances and concurrency, simplifies coding of Remote Procedure Calls and Messaging and makes programs and operating systems faster and more energy-efficient.  
Multi-threading is now allowed on any major CPU architecture, server, desktop or mobile.  
Multi-threading has also some (minor) drawbacks.  
Multithreadaded software is not easy to write nor to debug because of the different threads executed concurrently add a further (temporal) dimension to execution graphs.  
Access and modification to resources must be synchronized which is not easy and can cause subtle errors and serious performance issues.  
Porting singlethreaded code can be difficult and error prone and requires estensive testing.

**Multi-threading in Java**

Java natively supports multithreaded programming through the Thread class and the Runnable interface.  
Java provides resource locking and synchronization through the Semaphore class and synchronized methods and statements.  
 Java provides a rich set of Concurrent Collection classes to help avoid problems caused by concurrent access and modification to resources.  
Java also provides classes that support atomic operations on single variables.  
Many third party libraries and SDK also available.

**Java Annotations**

Java Annotations are a form of syntactic metadata.  
An annotation contains information about the source code itself.  
Java annotations can provide many different kind of informations to the compiler, the VM (runtime), some other process like Javadoc or the programmer too.  
 Java Annotations are present since Java 5.  
Custom annotations can be added easily.  
In this project some custom annotations are used to add to singlethreaded code a good degree or mutlthreaded capabilities.

**What is MTA**

MTA which stands for Multi-Threading Annotations is a technique that permits to execute a Singlethreaded Java program in a Multithreaded way with very little o no modification of program source code other than adding some Custom Annotations.  
At this time there are five annotations:

public boolean parallelize() default false;  
This annotation enables the unsynchronized multithreaded execution of the annotated method. At this time the return value is ignored when the execution is multithreaded.

public boolean synchronize() default false;  
This annotation enables the synchronized multithreaded execution of the annotated method. At this time the return value is ignored when the execution is multithreaded.

public String mutex() default "";  
This annotation creates and assignes a name to a mutex. Is usully used together with one of the two following annotations.

public boolean lock() default false;  
If the mutex isn’t locked yet, the execution of the annotated method continues normally.  
If the mutex is already blocked the execution of the annotated method stops until the mutex is released.

public boolean release() default false;  
If the mutex is locked, the mutex is released and the execution of the annotated method continues normally. If the mutex isn’t locked nothing happens and the execution continues normally.

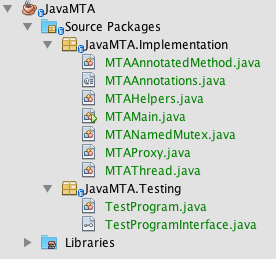
**How it works**

**Step 1:**  
Utilizing a custom class loader passed as an argument to JVM (-Djava.system.class.loader) the java program is patched accordingly.   
Should this way not working, the program classes can be loaded programmatically but this require some (simple) changes to the program source code.  
As the program is loaded, all classes are scanned and annotated methods are patched. At this time, that operation has some problems at VM level.

**Step 2:**  
During program execution the MTAProxy class intercepts the execution of class methods and through Java Reflection and accordingly to the Annotations which provide the necessary informations, those methods are executed concurrently from a pool of worker threads.  
Resources Locking and/or Synchronization are applied if requested.

**That’s it!**

**Sources freely available at** https://github.com/lucaseverini/CS159-Project



/\*

MTAAnnotatedMethod.java

CS159 - Class Project

April-1-2015

By Luca Severini (lucaseverini@mac.com)

\*/

package JavaMTA.Implementation;

import java.util.concurrent.Semaphore;

// Class MTAAnnotatedMethod

// ------------------------------------------------------------------

public class MTAAnnotatedMethod

{

private final String name;

private final MTAAnnotations annotation;

private Semaphore semaphore;

private MTANamedMutex mutex;

// MTAAnnotatedMethod

// ------------------------------------------------------------------

public MTAAnnotatedMethod(String name, MTAAnnotations annotation)

{

this.name = name;

this.annotation = annotation;

if(this.annotation.synchronize())

{

this.semaphore = new Semaphore(1);

}

if(this.annotation.lock() || this.annotation.release())

{

this.mutex = MTAHelpers.createNamedMutex(this.annotation.mutex());

}

}

// getName

// ------------------------------------------------------------------

public String getName()

{

return name;

}

// getAnnotation

// ------------------------------------------------------------------

public MTAAnnotations getAnnotation()

{

return annotation;

}

// acquireSemaphore

// ------------------------------------------------------------------

public void acquireSemaphore()

{

System.out.println(name + " : Acquiring Semaphore " + semaphore + " ...");

try

{

semaphore.acquire();

}

catch(InterruptedException ex)

{

ex.printStackTrace();

}

}

// releaseSemaphore

// ------------------------------------------------------------------

public void releaseSemaphore()

{

System.out.println(name + " : Releasing Semaphore " + semaphore + " ...");

semaphore.release();

}

// lockMutex

// ------------------------------------------------------------------

public void lockMutex()

{

System.out.println(name + " : Locking Mutex " + mutex.getName() + " ...");

try

{

mutex.acquire();

}

catch(InterruptedException ex)

{

ex.printStackTrace();

}

}

// releaseMutex

// ------------------------------------------------------------------

public void releaseMutex()

{

System.out.println(name + " : Releasing Mutex " + mutex.getName() + " ...");

mutex.release();

}

}

/\*

MTAAnnotations.java

CS159 - Class Project

April-1-2015

By Luca Severini (lucaseverini@mac.com)

\*/

package JavaMTA.Implementation;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

// Annotation Interface MTAAnnotations

// ------------------------------------------------------------------

public @interface MTAAnnotations

{

public boolean parallelize() default false;

public boolean synchronize() default false;

public String mutex() default "";

public boolean lock() default false;

public boolean release() default false;

}

/\*

MTAHelpers.java

CS159 - Class Project

April-1-2015

By Luca Severini (lucaseverini@mac.com)

\*/

package JavaMTA.Implementation;

import java.util.ArrayList;

// Class MTAHelpers

// ------------------------------------------------------------------

public class MTAHelpers

{

// sleep

// ------------------------------------------------------------------

public static void sleep(int microsecs)

{

try

{

Thread.sleep(microsecs);

}

catch(Exception ex)

{

ex.printStackTrace();

}

}

// createNamedMutex

// ------------------------------------------------------------------

public static MTANamedMutex createNamedMutex(String name)

{

ArrayList<MTANamedMutex> namedMutexes = MTAProxy.getNamedMutexes();

for(MTANamedMutex mutex : namedMutexes)

{

if(mutex.getName().equals(name))

{

return mutex;

}

}

MTANamedMutex mutex = new MTANamedMutex(name);

namedMutexes.add(mutex);

return mutex;

}

}

/\*

MTACore.java

CS159 - Class Project

April-1-2015

By Luca Severini (lucaseverini@mac.com)

\*/

package JavaMTA.Implementation;

import JavaMTA.Testing.TestProgram;

import JavaMTA.Testing.TestProgramInterface;

// Class MTAMain

// ------------------------------------------------------------------

public class MTAMain

{

public static void main(String[] args)

{

TestProgramInterface myTest = (TestProgramInterface)MTAProxy.newInstance(new TestProgram());

MTAProxy.setEnabled(true);

long start = System.nanoTime();

System.out.println("START");

TestProgram.main(myTest);

if(myTest != null)

{

MTAProxy.getProxy().waitForTermination();

System.out.println("END: " + myTest.getCounter());

}

long end = System.nanoTime();

System.out.println(String.format("Program executed in %d secs", (end - start) / 1000000000));

}

}

/\*

MTANamedMutex.java

CS159 - Class Project

April-1-2015

By Luca Severini (lucaseverini@mac.com)

\*/

package JavaMTA.Implementation;

import java.util.concurrent.Semaphore;

// Class NamedMutex

// ------------------------------------------------------------------

public class MTANamedMutex extends Semaphore

{

private static final long serialVersionUID = 1L;

private final String name;

// MTANamedMutex

// ------------------------------------------------------------------

public MTANamedMutex(String name)

{

super(1); // Creates a semaphore with 1 access only

this.name = name;

}

// getName

// ------------------------------------------------------------------

public String getName()

{

return name;

}

}

/\*

MTAProxy.java

CS159 - Class Project

April-1-2015

By Luca Severini (lucaseverini@mac.com)

\*/

// References

// http://www.javacodegeeks.com/2012/08/creating-java-dynamic-proxy.html

// http://www.mkyong.com/java/java-custom-annotations-example/

package JavaMTA.Implementation;

import JavaMTA.Testing.TestProgram;

import java.lang.annotation.Annotation;

import java.lang.reflect.Method;

import java.util.ArrayList;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

// Class MTAProxy

// ------------------------------------------------------------------

public class MTAProxy implements java.lang.reflect.InvocationHandler

{

private static boolean enabled = true;

private static MTAProxy proxy;

private final Object obj;

private final ExecutorService executor;

private static ArrayList<MTAAnnotatedMethod> annotatedMethods;

private static ArrayList<MTANamedMutex> namedMutexes;

// newInstance

// ------------------------------------------------------------------

public static Object newInstance(Object obj)

{

return java.lang.reflect.Proxy.newProxyInstance(obj.getClass().getClassLoader(), obj.getClass().getInterfaces(), new MTAProxy(obj));

}

// MTAProxy

// ------------------------------------------------------------------

private MTAProxy(Object obj)

{

this.obj = obj;

this.executor = Executors.newFixedThreadPool(15);

this.proxy = this;

annotatedMethods = new ArrayList<>();

namedMutexes = new ArrayList<>();

collectAnnotatedMethods();

}

// invoke

// ------------------------------------------------------------------

@Override

public Object invoke(Object proxy, Method m, Object[] args) throws Throwable

{

Object result = new Integer(0);

if(!enabled)

{

return m.invoke(obj, args);

}

try

{

System.out.println("Before method " + m.getName());

long start = System.nanoTime();

if(hasParallelizeAnnotation(m))

{

System.out.println("Invoking " + m.getName() + " Concurrently...");

Runnable worker = new MTAThread(m, obj, args);

executor.execute(worker);

}

else

{

if(hasSinchronizeAnnotation(m))

{

acquireSynchronization(m);

result = m.invoke(obj, args);

releaseSynchronization(m);

}

else if(hasMutexLockAnnotation(m))

{

lockMutex(m);

result = m.invoke(obj, args);

}

else if(hasMutexReleaseAnnotation(m))

{

releaseMutex(m);

result = m.invoke(obj, args);

}

else

{

result = m.invoke(obj, args);

}

}

long end = System.nanoTime();

System.out.println(String.format("%s took %d ns", m.getName(), end - start));

}

catch (Exception e)

{

throw new RuntimeException("Proxy exception: " + e.getMessage());

}

finally

{

System.out.println("After method " + m.getName());

}

return result;

}

// waitForTermination

// ------------------------------------------------------------------

public void waitForTermination()

{

System.out.println("Waiting for termination...");

MTAHelpers.sleep(1000);

executor.shutdown();

while (!executor.isTerminated())

{

Thread.yield();

}

}

// collectAnnotatedMethods

// ------------------------------------------------------------------

private void collectAnnotatedMethods()

{

Class<TestProgram> prog = TestProgram.class;

for (Method m : prog.getDeclaredMethods())

{

if (m.isAnnotationPresent(MTAAnnotations.class))

{

Annotation annotation = m.getAnnotation(MTAAnnotations.class);

MTAAnnotations parallelization = (MTAAnnotations) annotation;

// Show possible value for all annotations

System.out.println("Method " + m.getName() + " : parallelize = " + parallelization.parallelize());

System.out.println("Method " + m.getName() + " : synchronize = " + parallelization.synchronize());

System.out.println("Method " + m.getName() + " : mutex = " + parallelization.mutex());

System.out.println("Method " + m.getName() + " : lock = " + parallelization.lock());

System.out.println("Method " + m.getName() + " : release = " + parallelization.release());

annotatedMethods.add(new MTAAnnotatedMethod(m.getName(), parallelization));

}

}

}

// hasParallelizeAnnotation

// ------------------------------------------------------------------

public static boolean hasParallelizeAnnotation(Method m)

{

for(MTAAnnotatedMethod annotM : annotatedMethods)

{

if(annotM.getName().equals(m.getName()))

{

return annotM.getAnnotation().parallelize();

}

}

return false;

}

// hasSinchronizeAnnotation

// ------------------------------------------------------------------

public static boolean hasSinchronizeAnnotation(Method m)

{

for(MTAAnnotatedMethod annotM : annotatedMethods)

{

if(annotM.getName().equals(m.getName()))

{

return annotM.getAnnotation().synchronize();

}

}

return false;

}

// acquireSynchronization

// ------------------------------------------------------------------

public static void acquireSynchronization(Method m)

{

for(MTAAnnotatedMethod annotM : annotatedMethods)

{

if(annotM.getName().equals(m.getName()))

{

annotM.acquireSemaphore();

return;

}

}

}

// releaseSynchronization

// ------------------------------------------------------------------

public static void releaseSynchronization(Method m)

{

for(MTAAnnotatedMethod annotM : annotatedMethods)

{

if(annotM.getName().equals(m.getName()))

{

annotM.releaseSemaphore();

return;

}

}

}

// hasMutexLockAnnotation

// ------------------------------------------------------------------

public static boolean hasMutexLockAnnotation(Method m)

{

for(MTAAnnotatedMethod annotM : annotatedMethods)

{

if(annotM.getName().equals(m.getName()))

{

return annotM.getAnnotation().lock();

}

}

return false;

}

// hasMutexReleaseAnnotation

// ------------------------------------------------------------------

public static boolean hasMutexReleaseAnnotation(Method m)

{

for(MTAAnnotatedMethod annotM : annotatedMethods)

{

if(annotM.getName().equals(m.getName()))

{

return annotM.getAnnotation().release();

}

}

return false;

}

// lockMutex

// ------------------------------------------------------------------

public static void lockMutex(Method m)

{

for(MTAAnnotatedMethod annotM : annotatedMethods)

{

if(annotM.getName().equals(m.getName()))

{

annotM.lockMutex();

return;

}

}

}

// releaseMutex

// ------------------------------------------------------------------

public static void releaseMutex(Method m)

{

for(MTAAnnotatedMethod annotM : annotatedMethods)

{

if(annotM.getName().equals(m.getName()))

{

annotM.releaseMutex();

return;

}

}

}

// getNamedMutexes

// ------------------------------------------------------------------

public static ArrayList<MTANamedMutex> getNamedMutexes()

{

return namedMutexes;

}

// getProxy

// ------------------------------------------------------------------

public static MTAProxy getProxy()

{

return proxy;

}

// setEnabled

// ------------------------------------------------------------------

public static void setEnabled(boolean enable)

{

enabled = enable;

}

// getEnabled

// ------------------------------------------------------------------

public static boolean getEnabled()

{

return enabled;

}

}

/\*

MTAThread.java

CS159 - Class Project

April-1-2015

By Luca Severini (lucaseverini@mac.com)

\*/

package JavaMTA.Implementation;

import java.lang.reflect.Method;

// Class MTAThread

// ------------------------------------------------------------------

public class MTAThread implements Runnable

{

private final Method method;

private final Object object;

private final Object[] args;

private Object result;

// MTAThread

// ------------------------------------------------------------------

public MTAThread(Method m, Object o, Object[] a)

{

this.method = m;

this.object = o;

this.args = a;

}

// run

// ------------------------------------------------------------------

@Override

public void run()

{

// System.out.println(Thread.currentThread().getName() + " Start Worker for " + method);

try

{

System.out.println("MTA Thread " + Thread.currentThread().getId() + " for " + method.getName());

if(MTAProxy.hasSinchronizeAnnotation(method))

{

MTAProxy.acquireSynchronization(method);

result = method.invoke(object, args);

MTAProxy.releaseSynchronization(method);

}

else

{

result = method.invoke(object, args);

}

System.out.println("MTA Thread " + Thread.currentThread().getId() + " for " + method.getName() + " : " + result);

}

catch (Exception e)

{

throw new RuntimeException("MTA Thread Exception: " + e.getMessage());

}

// System.out.println(Thread.currentThread().getName() + " End Worker for " + method);

}

// toString

// ------------------------------------------------------------------

@Override

public String toString()

{

return "MTA Thread for " + method;

}

}

/\*

TestProgram.java

CS159 - Class Project

April-1-2015

By Luca Severini (lucaseverini@mac.com)

\*/

package JavaMTA.Testing;

import JavaMTA.Implementation.MTAAnnotations;

import JavaMTA.Implementation.MTAHelpers;

// Class TestProgram

// ------------------------------------------------------------------

public class TestProgram implements TestProgramInterface

{

static private TestProgramInterface progInterface;

private int counter;

// TestProgram

// ------------------------------------------------------------------

public TestProgram()

{

counter = 0;

}

// main

// ------------------------------------------------------------------

public static int main(TestProgramInterface prog)

{

if(prog != null)

{

progInterface = prog;

}

else

{

progInterface = new TestProgram();

}

progInterface.test();

return 0;

}

// test

// ------------------------------------------------------------------

@MTAAnnotations(parallelize=false)

@Override

public void test()

{

for(int idx = 1; idx <= 100; idx++)

{

int result = progInterface.func(idx);

}

}

// func

// ------------------------------------------------------------------

@MTAAnnotations(parallelize=true)

@Override

public int func(int param)

{

progInterface.section1Enter();

// This piece of code needs a critical section

int result = progInterface.getCounter() + 100;

MTAHelpers.sleep(300);

progInterface.setCounter(result);

progInterface.section1Exit();

// This code doesn't need a critical section

// int result = t.addToCounter(100);

printLog("Thread: " + Thread.currentThread().getId());

MTAHelpers.sleep(1000);

return result;

}

// addToCounter

// ------------------------------------------------------------------

@MTAAnnotations(synchronize=true) // Makes this method Synchronized (true) or not (false)

@Override

public int addToCounter(int val)

{

counter += val;

return counter;

}

// setCounter

// ------------------------------------------------------------------

@MTAAnnotations(synchronize=false) // Makes this method Synchronized (true) or not (false)

@Override

public void setCounter(int val)

{

counter = val;

}

// getCounter

// ------------------------------------------------------------------

@MTAAnnotations(synchronize=false) // Makes this method Synchronized (true) or not (false)

@Override

public int getCounter()

{

return counter;

}

// section1Enter

// ------------------------------------------------------------------

@MTAAnnotations(mutex="section1", lock=true) // Makes this method a Critical Section entry point (true) or not (false)

@Override

public void section1Enter()

{

System.out.println("Entered section1...");

}

// section1Exit

// ------------------------------------------------------------------

@MTAAnnotations(mutex="section1", release=true) // Makes this method a Critical Section exit point (true) or not (false)

@Override

public void section1Exit()

{

System.out.println("Exited section1...");

}

// printLog

// ------------------------------------------------------------------

public void printLog(String text)

{

progInterface.section1Enter(); // Enter Critical Section

System.out.println("Just print something: \*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Just print something: \*\*\*\*\*\*\*\*\*\*\*");

System.out.println(text);

System.out.println("Just print something: \*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Just print something: \*\*\*\*\*\*\*\*\*\*\*");

progInterface.section1Exit(); // Exit Critical Section

}

}

/\*

TestProgramInterface.java

CS159 - Class Project

April-1-2015

By Luca Severini (lucaseverini@mac.com)

\*/

package JavaMTA.Testing;

// Interface TestProgramInterface

// ------------------------------------------------------------------

public interface TestProgramInterface

{

public int func(int param);

public void test();

public void setCounter(int val);

public int getCounter();

public int addToCounter(int val);

public void section1Enter();

public void section1Exit();

}