

## Lab 2, TDDD04, muhis511

### Concurrency Testing 1: Deadlock detection

#### How would you have found the error in the program if you had not used JPF?

If I don't use JPF so it could be very difficult to find the errors. Because code seems working fine, and this error was happening on some specific path or conditions. I can try to run it on different conditions to find it. That take a lot of time to find.

#### How does JPF find the deadlock? Use information from the trace printouts from the program as well as the description of how JPF works to justify your answer.

JPF run the code again and again with different paths, and checking the code. By this it automatically detect the error that cause the deadlock. In trace printouts, it print the output again and again for each success path. And when error occur id generate Deadlock occurred exception and display the line of code that causing the error or problem.

### Concurrency Testing 1: Race condition detection

#### How would you have found the error in the program if you had not used JPF?

If I don't use JPF then I have to give multiple inputs to the program and observe the output. Like it is the output that I am expecting or not. If not that means some resources trying to overwrite it or changing it. Like in given code two threads trying to write output in file that creating problem in the output.

#### What happens if you make the `Updater.run` or `Pair.update` methods synchronized? Why?

If we change it to synchronized methods then it limit the thread access and only one thread can access the resources. And the conflict that was creating error is resolved.

### Symbolic Execution 1: Test case generation with Exception

#### What happens if you change the `Triangle.getType` method to not throw exceptions on invalid parameters but return `TriangleType.NaT` instead?

If we remove the exception for invalid parameters and use `Nat` in return then the output that generated will be result in `NAT` message that indicate triangle is not valid. And give result in no error generated. It also generate tests for triangle. To check it valid or not.

#### What test cases are generated if you replace the exception?

Not a triangle test cases are generated.

```
public class triangle_TriangleTest {

    private triangle.Triangle triangle_triangle;

    @Before
    public void setUp() throws Exception {
        triangle_triangle = new triangle.Triangle();
    }

    @Test
    public void test0() {
        triangle_triangle.getType(3,2,1);
    }

    @Test
    public void test1() {
        triangle_triangle.getType(-2147483646,-2147483647,-2147483648);
    }

    @Test
    public void test2() {
        triangle_triangle.getType(2147483647,2147483647,2147483647);
    }

    @Test
    public void test3() {
        triangle_triangle.getType(-2147483647,-2147483648,-2147483648);
    }

    @Test
    public void test4() {
        triangle_triangle.getType(-2147483647,-2147483648,-2147483647);
    }

    @Test
    public void test5() {
        triangle_triangle.getType(2,3,1);
    }

    @Test
    public void test6() {
        triangle_triangle.getType(-2147483647,-2147483646,-2147483648);
    }

    @Test
    public void test7() {
        triangle_triangle.getType(-2147483647,-2147483647,-2147483648);
    }

    @Test
    public void test8() {
        triangle_triangle.getType(-2147483648,-2147483647,-2147483648);
    }

    @Test
    public void test9() {
        triangle_triangle.getType(-2147483648,-2147483648,-2147483648);
    }
}

===== results
no errors detected
```

**What happens if you change `getType(sym#sym#sym)` to `getType(con#con#sym)`?**

If we replace it then generated test cases in previous question will change only #sym parameter and other two parameters are concrete so they will be constant. And triangle will be tested deeply (maximum coverage) with only one parameter variation.

**What does the notation `getType(con#con#sym)` mean and what is the effect on the generated test cases? How do you think it could be used when probing the program for execution paths?**

#con is use to mention concrete value and value will not be change for this parameter. #sym is use to indicate symbolic and its value change until it reach to the coverage of all code. Generated test cases will change only #sym value. Other value will remains constant.

If we change parameters to #sym and #con then it will be easy to test more code coverage for single path. Because it narrow down the path and mutation of single value will result in less generated test case with more coverage for that specific path.

## **Symbolic Execution 2: Test case generation with coverage analysis**

**How does the `SymbolicSequenceListener` seem to select test case input values?**

`symbolicSequenceListener` try to test all the code coverage. So it select values that generate max coverage.

**Do you get 100% statement coverage from the test cases? If not, why do you think that is so? Hint: consider the configuration parameters to the `jpf.symbc` package on how to select symbolic values for different datatypes.**

Not 100% it showing 80% coverage because I think the one (if) condition is not using symbolic parameter (usage) that's why it's not covering the full code coverage. That (if) condition might be not reaching the test coverage due to above reason.

**Does the coverage indication provided by the `CoverageAnalyzer` seem correct to you? Justify your answer.**

It seems not fully correct because if we try to test code with dry run, it showing full coverage but analyser is not reaching to that condition. Reason could be the value of integer. It ignoring the integer max value that causing to not reach last (if) condition.

## **Last Exercise: Experimentation**

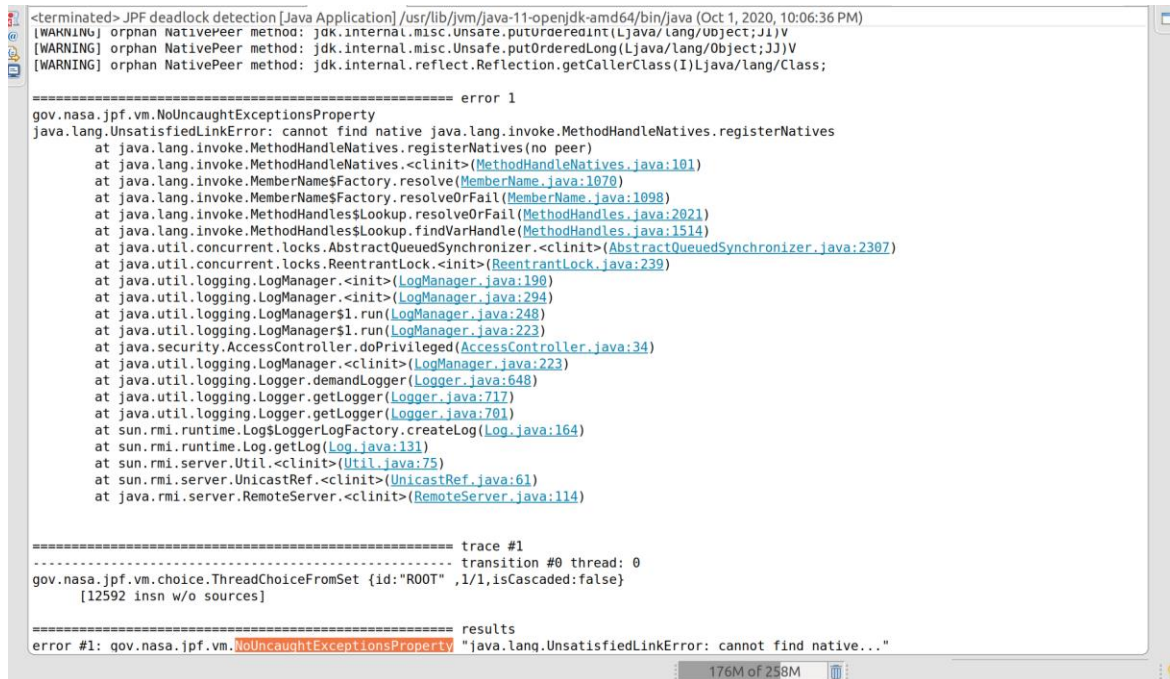
### **Alarmclock:**

In alarm clock file `MyLinkedList.java` have this error. (`PreciseRaceDetection`)

## log4j2:

### Errors:

When I run NumberCruncherServer I got unsatisfied link error. Also found a PreciseRaceDetection error.



```
<terminated> JPF deadlock detection [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Oct 1, 2020, 10:06:36 PM)
[WARNING] orphan NativePeer method: jdk.internal.misc.Unsafe.putOrderedInt(Ljava/lang/Object;J)V
[WARNING] orphan NativePeer method: jdk.internal.misc.Unsafe.putOrderedLong(Ljava/lang/Object;JJ)V
[WARNING] orphan NativePeer method: jdk.internal.reflect.Reflection.getCallerClass(I)Ljava/lang/Class;

===== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.UnsatisfiedLinkError: cannot find native java.lang.invoke.MethodHandleNatives.registerNatives
  at java.lang.invoke.MethodHandleNatives.registerNatives(no peer)
  at java.lang.invoke.MethodHandleNatives.<clinit>(MethodHandleNatives.java:101)
  at java.lang.invoke.MemberName$Factory.resolve(MemberName.java:1070)
  at java.lang.invoke.MemberName$Factory.resolveOrFail(MemberName.java:1098)
  at java.lang.invoke.MethodHandles$Lookup.resolveOrFail(MethodHandles.java:2021)
  at java.lang.invoke.MethodHandles$Lookup.findVarHandle(MethodHandles.java:1514)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.<clinit>(AbstractQueuedSynchronizer.java:2307)
  at java.util.concurrent.locks.ReentrantLock.<init>(ReentrantLock.java:239)
  at java.util.logging.LogManager.<init>(LogManager.java:190)
  at java.util.logging.LogManager.<init>(LogManager.java:294)
  at java.util.logging.LogManager$1.run(LogManager.java:248)
  at java.util.logging.LogManager$1.run(LogManager.java:223)
  at java.security.AccessController.doPrivileged(AccessController.java:34)
  at java.util.logging.LogManager.<clinit>(LogManager.java:223)
  at java.util.logging.Logger.demandLogger(Logger.java:648)
  at java.util.logging.Logger.getLogger(Logger.java:717)
  at java.util.logging.Logger.getLogger(Logger.java:701)
  at sun.rmi.runtime.Log$LoggerLogFactory.createLog(Log.java:164)
  at sun.rmi.runtime.Log.getLog(Log.java:131)
  at sun.rmi.server.Util.<clinit>(Util.java:75)
  at sun.rmi.server.UnicastRef.<clinit>(UnicastRef.java:61)
  at java.rmi.server.RemoteServer.<clinit>(RemoteServer.java:114)

===== trace #1
----- transition #0 thread: 0
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"R00T" ,l/1,isCascaded:false}
[12592 insns w/o sources]

===== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.UnsatisfiedLinkError: cannot find native..."

176M of 258M
```

### Differences in code:

I run files with diff command and it give some structure differences in comments, code format etc.

Like `/* */` replaced with `/* *** */`. And code like `public void` change into two lines.

UnitTestAppender.java have an additional print in testappend function.