CSC 543 Multiprocessing & Concurrent Programming, Spring 2023

Dr. Dale E. Parson, Assignment 2, Implicit & Explicit Locks & Condition Variables
This assignment is due by 11:59 PM on Sunday March 12 via make turnitin.

The goals of this assignment are to: 1) Use the implicit lock/condition variable capability built into java.lang.Object for synchronized communication between two "partner threads"; 2) Use the explicit lock/condition variable capability of ReentrantLock and Condition of java.util.concurrent.locks for the same purpose; 3) Make all classes in this project @ThreadSafe or @Immutable, and use the @GuardedBy annotation where appropriate.

The source code for my Processing demo programs using implicit and explicit locks that we went over in class are here.
https://faculty.kutztown.edu/parson/spring2023/CSC543DemoImageImplicitLock.txt
https://faculty.kutztown.edu/parson/spring2023/CSC543DemoImageExplicitLock.txt

Perform the following steps to set up this project and to get my handout. Start out in your login directory on csit (a.k.a. acad).

```
cd  $HOME
mkdir multip  # If already here from assignment 1 then ignore the error message.
cp  ~parson/multip/prisonerd2j2023.problem.zip  multip/prisonerd2j2023.problem.zip
```

Use machine **mcgonagall** for development and testing. Run **make turnitin** on acad or mcgonagall by the due date.

After logging into mcgonagall, do the following.

```
cd   ~/multip
unzip  prisonerd2j2023.problem.zip
cd  ./prisonerd2j2023
make  test
```

There are 3 Java files in this assignment, all of which contain **STUDENT** comments with instructions, and all of which you will change. **Please complete the header comments at the top of each source file**.

[:-) ~/multip/prisonerd2j2023] **$ ls -l *.java**
-rw-r--r--. 1 parson domain users 5821 Feb 20 10:11 PlayerImplicit.java
-rw-r--r--. 1 parson domain users 2758 Feb 20 10:12 PrisonerTest.java
-rw-r--r--. 1 parson domain users 3134 Feb 20 10:13 Tables.java

After you have all of the above files working according to requirements, you will **cp PlayerImplicit.java PlayerExplicit.java**, change the latter file to use explicit locks as explained in this document, remove a **STUDENT**-delimited comment in PrisonerTest.java in order to test explicit locks, verify that **make test** works on multiple platforms, and verify that all annotation and documentation requirements in this spec and in the code's **STUDENT** comments are met.

A summary of work follows. We will go over this in detail in class.

From Tables.java:

```
/**
  *  STUDENT: Make the data fields of this class as thread safe
        *    as possible. Add an appropriate annotation tag
        *    for the class. It must not be possible for code
        *    outside this class to change any of these data fields.
        *    The code inside the class does not change them after
        *    initialization. STUDENT must ensure that every data
        *    structure in this class is safe for multi-threaded
        *    access. See access methods at bottom of this file.
        *    Also annotate the class with one of the thread-safety
        *    annotations similar to Assignment 1.
**/
```

From PrisonerTest.java:

```
        /*
         * STUDENT: Make the data fields of this class as thread safe
         *    as possible. Note the @Immutable class tag.
         *    STUDENT Remove this comment and let the PlayerExplicit
         *    run after you code PlayerExplicit.
        } else {
           PlayerExplicit player = new PlayerExplicit(i);
           spairs[i] = new Thread(player);
        */
```

Most of your work will take place in file PlayerImplicit.java, in which you will use an implicit lock and associated condition variable to synchronize communication between two "partner threads", and in file PlayerExplicit.java that you will create by copying the completed PlayerImplicit.java, then changing it to use an explicit reentrant lock and associated condition variable from java.util.concurrent.locks. The logic of this application follows the logic of the current CSC 343 Operating Systems assignment here:

https://faculty.kutztown.edu/parson/spring2021/csc343fall2016assn1.pdf

There are 10 pairs of "partner threads" that play the Iterated Prisoner's Dilemma from game theory. The CSC 343 students coded an implementation using a Unified Modeling Language (UML) based state machine language built atop Python. We will be using Java threads with lock and condition variable-based synchronization in our assignment. Consult the above PDF file for the algorithm, which we will go over in class. I have implemented most of the Java algorithm. You must implement the inter-thread synchronization in PlayerImplicit.java and PlayerExplicit.java.

Here is what happens when you run **make test** on the handout code:

```
Unexpected exception: null
Exception in thread "Thread-3" Unexpected exception: null
Unexpected exception: null
Exception in thread "Thread-7" Unexpected exception: null
Exception in thread "Thread-5" Unexpected exception: null
Exception in thread "Thread-9" Exception in thread "Thread-1" Unexpected exception: null
java.lang.RuntimeException: Unexpected exception
```

```
        at prisonerd2j2023.PlayerImplicit.run(PlayerImplicit.java:117)
        at java.lang.Thread.run(Thread.java:748)
Caused by: java.lang.NullPointerException
        at prisonerd2j2023.Tables.getPenalty(Tables.java:65)
        at prisonerd2j2023.PlayerImplicit.run(PlayerImplicit.java:100)
        ... 1 more
Unexpected exception: null
MANY MORE SIMILAR FAILURES IN OTHER THREADS
```

Those failures occur because of synchronization problems.

From PlayerImplicit.java:

```
  /** The run() method runs this object's Thread.
   *  STEPS: (The ones without STUDENT are already done.)
   *    -1. STUDENT: Make the data fields of this class as thread safe
   *    as possible.
   *    RUN() code:
   *    0. STUDENT: Start Thread for playernum 1 from playernum 0.
   *    1. Initialize variables for this state machine. DONE.
   *    2. Run this state machine that **interacts with its partner Thread**.
   *      STUDENT ADD USE OF IMPLICIT LOCK & CONDITION VARIABLE WHERE NEEDED.
   *      IN ADDITION TO ACTUALLY MANIPULATING AN IMPLICIT LOCK AND
   *      CONDVAR, STUDENT MAY HAVE TO ADD SOME ADDITIONAL STATEMENTS
   *      FOR READING OR WRITING THE DATA PROTECTED BY THE LOCK.
   *      YOU NEED TO MAKE THIS INTERACTION WORK WITHOUT RACE CONDITIONS
   *      OR OTHER BUGS. The state machine logic itself is correct.
   *      I removed the inter-thread synchronization logic.
   *      Everything else is OK with respect to application logic.
   *    3. Accept state "terminated" returns from this function.
   *    4. STUDENT: Add any other inter-thread synchronization needed.
   *    5. STUDENT: Annotate this class with one of @NotThreadSafe,
   *       @ThreadSafe, or @Immutable, and use the @GuardedBy annotation
   *       correctly where appropriate.
   *       Annotations are not necessarily in run().
  **/
```

Here is what a successful test run looks like:

```
$ make test
/bin/rm -f *.o *.class .jar core *.exe *.obj *.pyc
/bin/rm -f *.class *.out *.dif *.tmp sink.ref
/bin/rm -f /tmp/CSC543_*_parson.* ~parson/tmp/parsonsink.out
/bin/bash -c "CLASSPATH=..:./jcip-annotations.jar /usr/bin/javac  PlayerExplicit.java"
/bin/bash -c "CLASSPATH=..:./jcip-annotations.jar /usr/bin/javac  PrisonerTest.java"
bash -c "CLASSPATH=..:./jcip-annotations.jar /usr/bin/java prisonerd2j2023.PrisonerTest implicit |sort >
implicit.out"
bash ./diffcheck implicit.out implicit.ref /usr/local/bin/python3.7
bash -c "CLASSPATH=..:./jcip-annotations.jar /usr/bin/java prisonerd2j2023.PrisonerTest explicit |sort >
explicit.out"
bash ./diffcheck explicit.out explicit.ref /usr/local/bin/python3.7
[:-) ~/.../solutions/prisonerd2j2023]
```

The **diffcheck** script compares the amount of time spent in the *timeInJail* state in my reference file to the *timeInJail* in your output file, flagging an error if they differ by more than 20%. Originally I did not seed the Random number generator in PlayerImplicit.java, and occasionally I would get statistical diffs for the *halfsy* strategy. The code now seeds a Random seed value of 12345. I expect that the output files will now match the reference files identically, although an exact match is not a requirement. A statistical match is sufficient.

Some non-thread-safe code may run to completion without blowing up because race condition bugs are dependent on timing. A successful **make test** run does not guarantee thread safety. Therefore, please re-read the STUDENT comments and this handout after successful testing to ensure that you do not miss any requirements.

Run **make turnitin** on one of our Linux machines by the due date. The late penalty is 10% per day, and I will not accept solutions after I go over an assignment. Plan to attend class if possible and ask questions.