CSC 543 Multiprocessing & Concurrent Programming, Fall 2018

Dr. Dale E. Parson, Assignment 3, latches, barriers, atomic arrays, thread pools, etc. This assignment is due by 11:59 PM on Friday November 16 via make turnitin.

This assignment is a re-do of parts of both assignments 1 and 2 to use new mechanisms out of the Java concurrency library that we have not used before. The source file comments labeled with STUDENT give the details.

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
```

Use machine mcgonagall for development and testing. Machine harry will not work for the class StudentBlockingQueue that you are going to complete because that class is based on a later version of Java. Run make turnitin on acad or mcgonagall by the due date.

Always run make clean test on one of these machines before running make turnitin on a Linux machine to ensure that last-minute edits or file transfers have not created problems.

After logging into mcgonagall, do the following.

```
    cd ./multip
    unzip PipesPrisoners2018.problem.zip
    cd ./PipesPrisoners2018
    make clean test
```

Running make clean test on the handout code passes because it consists of my solutions to Assignments 1 and 2, which constitute your starting point. There are two portions to the assignment, those tested by running make clean testprison for testing your updated Assignment 2 (this is the easier part), and running make clean testpipe for testing the updated Assignment 1 pipeline (harder). You can do them in either order. Running make clean test tests both. The following source file comments explain the work you have to do. I have some more documentation added with these comments below.

Source code changes to the updated make clean testprison:

PrisonerTest.java

33     // STUDENT F18 ASSN3 #1: For the IMPLICIT players, construct
34     // a CountDownLatch object, initialized to the appropriate count for
35     // this main thread + ALL of the player threads, and have each
36     // of them enter that CountDownLatch object in the appropriate
37     // place in the code, instead of of the join() loop in PrisonerTest.
38     // For the EXPLICIT players, use a CyclicBarrier in the same way.

49     // STUDENT F18 ASSN3 #1: Get rid of this join() for-loop.
50     // See #1 instructions above.
PlayerImplicit.java

// STUDENT F18 ASSN3 #1: See PrisonerTest.java IMPLICIT section
// of F18 ASSN3 #1 STUDENT comments. Remove the conditional
// .join() call below, since both player 0 and player 1 must enter
// the CountDownLatch instead of joining.
// STUDENT F18 ASSN3 #2: Replace all uses of the intrinsic lock
// and its condition variable on messageBuffer access below,
// and replace the String [] messageBuffer itself, with a 2-element
// java.util.concurrent.atomic.AtomicReferenceArray<String> object
// field called messageBuffer that is used in an equivalent way.
// There will be no actual locking-waiting. Instead, a player must
// spin in a loop until the messageBuffer into which it is writing is null,
// and it must spin in a loop until the messageBuffer from which it is
// reading is non-null. The basic logic is the same, but it uses
// Make sure to update the @GuardedBy annotation if necessary.

PlayerExplicit.java

// STUDENT F18 ASSN3 #1: See PrisonerTest.java EXPLICIT section
// of F18 ASSN3 #1 STUDENT comments. Remove the conditional
// .join() call below, since both player 0 and player 1 must enter
// the CyclicBarrier instead of joining.
// STUDENT F18 ASSN3 #3: Replace all uses of the extrinsic lock
// and its condition variable on messageBuffer access below,
// with two two-element arrays of java.util.concurrent.Semaphore
// objects called isNull (initialized to a count of 1 and fair = true),
// and isReady (initialized to a count of 0 and fair = true).
// State "sendMyAction" must acquire the correct isNull element,
// then write a message, then release the correct isReady element.
// State "awaitOtherAction" must acquire the correct isReady element,
// then read and null a message slot, then release the correct isNull
// element. In this case the String [] messageBuffer is @GuardedBy
// both Semaphores; one guards writing, and the other guards
// reading/nulling. Semaphore-based approaches predate condition
// variables, and don't quite fit the @GuardedBy semantics.

Source code changes to the updated make clean testpipe:

PipelineStage.java

// STUDENT F18 ASSN3 #4: Change interface Runnable to be interface
// Callable<LongPair>, where the LongPair returned by call()
// is an @Immutable object whose class you must complete in this
// package PipesPrisoners2018 that constructs and stores the
// pipeline stage's thread's cpuTimeCounter and userTimeCounter values
// as defined by Assignment 1.
// Change the "public void run()" declaration below to
// "public LongPair call ()".
LongPair.java

// STUDENT F18 ASSN3 #5: Make this class immutable (and therefore
// thread-safe) and tag it as such with an annotation.

BigDecimalPipelineBuilder.java

// STUDENT F18 ASSN3 #6: Replace AtomicLong variables
// cpuTimeCounter and userTimeCounter with simple
// long variables of the same names, initialized
// to 0.

// STUDENT F18 ASSN3 #7: Remove the final two elements
// from the argsarray and typesarray arrays below --
// cpuTimeCounter, userTimeCounter for argsarray,
// and AtomicLong.class, AtomicLong.class for
// typesarray -- because those constructors no longer
// take AtomicLong arguments. Those pipeline stages
// now return LongPair objects from call().

// STUDENT F18 ASSN3 #8: Get rid of the threads array,
// the .start() loop, and the .join() loop below.
// Construct local variables "ExecutorService threadpool"
// and "ExecutorCompletionService<LongPair> completion"
// similar to those constructs in file
// ~parson/multip/PixelVisitor/PixelArrayMapReducer.java.
// Submit() each Callable<LongPair> task in transformers to
// "completion" in what was previously the .start() loop,
// and then take() each of the transformers.size()
// completed tasks from completion, .get() its resulting
// LongPair, and sum the LongPair's fields into local
// long variables cpuTimeCounter and userTimeCounter.
// Replace cpuTimeCounter.get() and userTimeCounter.get()
// below with cpuTimeCounter and userTimeCounter, since
// these are no longer atomics.
// STUDENT: Get rid of threads[].

// STUDENT: Replace this loop with a loop that submits
// Callable tasks from transformers to "completion".

// STUDENT: Replace this loop with a loop that retrieves
// Futures from "completion" and then performs
// get() on these to retrieve LongPair return values,
// summing these values into cpuTimeCounter & userTimeCounter.
// Run shutdown() on the threadpool after the loop is done.

PipeSourceRandom.java

// STUDENT F18 ASSN3 #9: Change this class to conform with the
// documentation in PipelineStage.java, changing it from Runnable
// to Callable<LongPair> with a call method that returns cputime, usertime
// in a LongPair object. Remove the constructor's cpuTimeCounter and
// userTimeCounter parameters, and convert the cputime and usertime
// fields in the object into long local variables within call().

PipeStageMath.java

// STUDENT F18 ASSN3 #10: Change this class to conform with the
// documentation in PipelineStage.java, changing it from Runnable
// to Callable<LongPair> with a call method that returns cputime, usertime
// in a LongPair object. Remove the constructor's cpuTimeCounter and
// userTimeCounter parameters, and convert the cputime and usertime
// fields in the object into long local variables within call().

PipeSinkFile.java

// STUDENT F18 ASSN3 #11: Change this class to conform with the
// documentation in PipelineStage.java, changing it from Runnable
// to Callable<LongPair> with a call method that returns cputime, usertime
// in a LongPair object. Remove the constructor's cpuTimeCounter and
// userTimeCounter parameters, and convert the cputime and usertime
// fields in the object into long local variables within call().

StudentBlockingQueue.java (see additional comments below)

* STUDENT F18 ASSN3 #12: Create your own class StudentBlockingQueue<E>
* inside this package PipesPrisoners2018, that implements interface
* java.util.concurrent.BlockingQueue<E>, and that is implemented using
* a java.util.LinkedList<E> within StudentBlockingQueue<E> in a thread-safe
* manner. Simply lock access to the contained LinkedList for every
* Queue and BlockingQueue method. Since LinkedList already implements
* Queue, you can delegate those Queue operations to the LinkedList object
* after locking access to it; make sure to unlock before returning.
* For the methods added to Queue by BlockingQueue, lock using the same
* lock, and use a condition variable associated with that lock to
* implement blocking waits on BlockingQueue. Make ABSOLUTELY SURE
* that all methods of your class use the same lock, either intrinsic
* or extrinsic (your choice).
*
* When implementing the BlockingQueue blocking functions, STUDENTs
* may assume that the offer() and put() calls will always succeed,
* since LinkedList does not have a fixed bound on size. Also,
* where TimeUnit appears, just assume that it is in milliseconds.
*
* After the above is done, uncomment the following three lines in testscript:
* runatest false false PipesPrisoners2018.StudentBlockingQueue
* runatest true false PipesPrisoners2018.StudentBlockingQueue
* runatest true true PipesPrisoners2018.StudentBlockingQueue

ADDED COMMENTS:

I have written a Jython script called GenQueueMethods.py that reads the class definition of
BlockingQueue via reflection and generates unsafe functions like this one that delegate their mechanics to the underlying LinkedList called container. Running make genstudent generates starting code into StudentBlockingQueue.start.txt:

```java
public boolean add(
    E arg0
) {
    return container.add(arg0);
}
```

If you email me a definition of that function that makes it thread-safe by limiting the number of threads that can access container to one thread at a time, in a way that satisfies Java thread safety requirements and memory model constraints and is applicable to all of the functions in StudentBlockingQueue.java that do not throw InterruptedException, I will rerun make genstudent with your thread safety enhancement and email you the StudentBlockingQueue.start.txt file in return. DO NOT CHANGE container to be something other than LinkedList. The goal is for you to create a thread-safe BlockingQueue class that uses LinkedList for element storage.

Once that is completed, what remains are the functions specific to BlockingQueue that throw InterruptedException. Consult the BlockingQueue Javadoc documentation for details. Here is an example:

```java
public E take(
) throws java.lang.InterruptedException {
    /* STUDENT BlockingQueue problem */
    return null ;
}
```

Note that I have annotated such functions with the comment “STUDENT BlockingQueue problem”. For any BlockingQueue operations that must block when READING an empty queue, use the condition variable associated with your locking mechanism to wait until contents appear. For any BlockingQueue operations that must block when WRITING a queue, assume that LinkedList can complete that write operation without failing. Also, do not catch InterruptedException within StudentBlockingQueue methods; just let it propagate out to the callers to your class.

You can assumed the TimeUnit associated with time-limited blocking calls is milliseconds. We are not using any of them in our test framework, and I don’t want to complicate things any further.

testscript

# STUDENT F18 ASSN3 #12: Create your own class StudentBlockingQueue<E>
# inside this package PipesPrisoners2018, that implements interface
# java.util.concurrent.BlockingQueue<E>, and that is implemented using
# a java.util.LinkedList<E> within StudentBlockingQueue<E> in a thread-safe
# manner. Simply lock access to the contained LinkedList for every
# Queue and BlockingQueue method. Since LinkedList already implements
# Queue, you can delegate those Queue operations to the LinkedList object
# after locking access to it; make sure to unlock before returning.
# For the methods added to Queue by BlockingQueue, lock using the same
# lock, and use a condition variable associated with that lock to
# implement blocking waits on BlockingQueue. Make ABSOLUTELY SURE
# that all methods of your class use the same lock, either intrinsic
# or extrinsic (your choice).

# STUDENT, After the above is done, uncomment the following three lines:
# runatest false false PipesPrisoners2018.StudentBlockingQueue
# runatest true false PipesPrisoners2018.StudentBlockingQueue
# runatest true true PipesPrisoners2018.StudentBlockingQueue

You can see the performance of your queue class by running `make testcsv` after make test works correctly, and inspecting the generated .csv file.

**Visually check your classes to make sure their field declarations and scopes of their locks are thread safe. Testing is not enough by itself. Update thread safety annotations where needed.**

After a final visual check and a successful run of `make clean test`, run `make turnitin` by the deadline.