CSC 543 Multiprocessing & Concurrent Programming, Spring 2023

Dr. Dale E. Parson, Assignment 1, Thread-safe blocking queues in Producer -> Consumer Pipelines
This assignment is due by 11:59 PM on Friday February 16 via <u>make turnitin</u>.

The goals of this assignment are to: 1) Write your first multi-threaded Java program for the course, using my single-threaded code as a starting point; 2) Use non-blocking Queue-derived classes and BlockingQueue-derived classes from java.util.concurrent; 3) Compare the performance of the above classes in a pipeline.

Perform the following steps to set up for this semester's projects and to get my handout. Start out in your login directory on acad or mcgonagall. We are using Java version 8 on mcgonagall because the newer version on acad has some incompatibilities with Weka and jython tools that I hope to address in summer. Documentation is here:
https://docs.oracle.com/javase/8/docs/api/index.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html

```
cd  $HOME
mkdir  multip
cp  ~parson/multip/pipeline2023.problem.zip  multip/pipeline2023.problem.zip
```

Log into **mcgonagall** via the **ssh** command from acad.

```
ssh  mcgonagall
```

This is a Linux machine with 32 contexts (hardware threads) and the same architecture as **acad**.

After logging into **mcgonagall**, do the following.

```
cd  ./multip
unzip  pipeline2023.problem.zip
cd  ./pipeline2023
make  test
```

Running **make clean test** should work OK and create output that looks something like this:

```
$ make clean test
/bin/rm -f *.o *.class .jar core *.exe *.obj *.pyc
/bin/rm -f *.class *.out *.dif *.tmp sink.out *.csv junk1.txt junk2.txt
/bin/rm -f /tmp/CSC543_*_parson.* ~parson/tmp/parsonsink.out
/bin/rm -f testtime.out testtime.txt
/bin/bash -c "CLASSPATH=..:./jcip-annotations.jar /usr/bin/javac  PipeSinkFile.java"
/bin/bash -c "CLASSPATH=..:./jcip-annotations.jar /usr/bin/javac  PipeSourceRandom.java"
/bin/bash -c "CLASSPATH=..:./jcip-annotations.jar /usr/bin/javac  PipeStageMath.java"
/bin/bash -c "CLASSPATH=..:./jcip-annotations.jar /usr/bin/javac  BigDecimalPipelineBuilder.java"
Note: BigDecimalPipelineBuilder.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
chmod +x testscript
STUDENT=parson bash -x ./testscript
```

```
+ runatest false false java.util.LinkedList
+ CLASSPATH=..:jcip-annotations.jar
```
+ /bin/time '--format=%e ELAPSED %U USERCPU %S SYSTEMCPU' java pipeline2023.BigDecimalPipelineBuilder false false pipeline2023.PipeSourceRandom '12345 3000 3000' java.util.LinkedList pipeline2023.PipeStageMath + java.util.LinkedList pipeline2023.PipeSinkFile sink.out
```
There are 32 contexts on this machine.
RUNNING java
pipeline2023.BigDecimalPipelineBuilder|false|false|pipeline2023.PipeSourceRandom|12345 3000
3000|java.util.LinkedList|pipeline2023.PipeStageMath|+|java.util.LinkedList|pipeline2023.PipeSinkFile|si
nk.out
2.58 ELAPSED 24.48 USERCPU 8.86 SYSTEMCPU
+ exitStatus=0
+ '[' 0 -ne 0 ']'
+ diff ./sink.out sink.ref
+ exitStatus=0
+ '[' 0 -ne 0 ']'
+ /bin/rm -rf ./sink.out
```

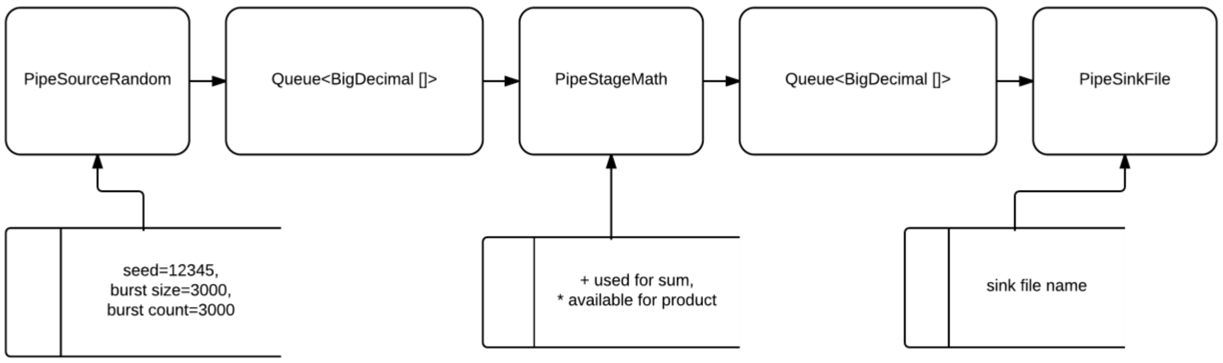
The test driver in class pipeline2023.BigDecimalPipelineBuilder builds a pipeline of concrete, active objects derived from interface PipelineStage, alternating with connectors derived from interface Queue<E>, with <E> bound to <java.math.BigDecimal []>.

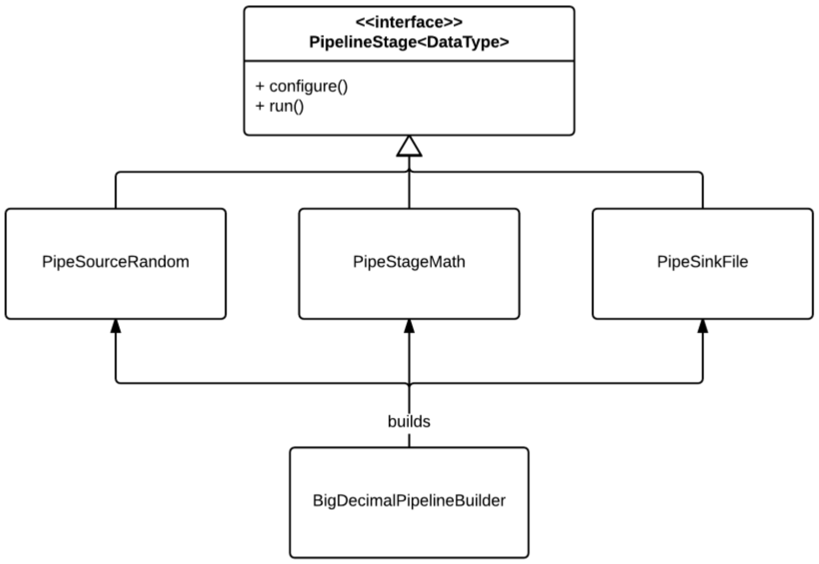
A dataflow diagram for the pipeline constructed by BigDecimalPipelineBuilder, and a class diagram for the classes in the project, appear on the next page. The pipeline generates a series of bursts of pseudo-random java.math.BigDecimal values in the first stage, sums a given burst in the second, and writes the summations to a file in the third. The second stage also supports multiplication, but the contribution of multithreading is less helpful because the BigDecimal terms grow so big that memory consumption-based paging comes to dominate execution time. My solution shows similar performance enhancements on all test platforms for summation.

There are **STUDENT** comments in each of these files that outline your work.

```
$ grep 'STUDENT.*%' *.java testscript
BigDecimalPipelineBuilder.java: *  STUDENT 4%: ANNOTATE CLASS AND CHECK FIELD TYPE
DECLARATIONS PER ASSN1 SPEC.
BigDecimalPipelineBuilder.java:                 // STUDENT 10%: CODE GOES HERE:
PipeSinkFile.java: *  STUDENT 10%: ANNOTATE CLASS AND CHECK FIELD TYPE DECLARATIONS
PER ASSN1 SPEC.
PipeSinkFile.java:                        inbatch = inbq.poll(); // STUDENT: 2% CHANGE
TO BLOCKING CALL
PipeSourceRandom.java: *  STUDENT 10%: ANNOTATE CLASS AND CHECK FIELD TYPE
DECLARATIONS PER ASSN1 SPEC.
PipeSourceRandom.java:                    oubq.add(result); // STUDENT 2%: CHANGE TO
BLOCKING CALL
PipeSourceRandom.java:                oubq.add(eof); // STUDENT 2%: CHANGE TO BLOCKING
CALL
PipeSourceRandom.java:            oubq.add(eof); // STUDENT 2%: CHANGE TO BLOCKING
CALL
PipeStageMath.java: *  STUDENT 10%: ANNOTATE CLASS AND CHECK FIELD TYPE DECLARATIONS
PER ASSN1 SPEC.
PipeStageMath.java:                        inbatch = inbq.poll(); // STUDENT 2%:
CHANGE TO BLOCKING CALL
PipeStageMath.java:                        oubq.add(eof);  // STUDENT 2%: CHANGE TO
BLOCKING CALL
```

```
PipeStageMath.java:                    oubq.add(outbatch);  // STUDENT 2%: CHANGE TO
BLOCKING CALL
PipeStageMath.java:                oubq.add(eof); // STUDENT 2%: CHANGE TO BLOCKING
CALL
testscript:# STUDENT 40% of assignment CODE TO GO AT THE BOTTOM:
```





For each Java class (not the interface) you must precede the class declarations with one of the annotations from the textbook as found in jcip-annotations.jar **(@NotThreadSafe, @ThreadSafe,** and **@Immutable**), and for the latter two categories, you must ensure that the classes satisfy that tag. The annotation goes immediately before the class declaration in the code like this:

@Annotation (one of **(@NotThreadSafe, @ThreadSafe,** and **@Immutable** in place of @Annotation)
public class BigDecimalPipelineBuilder {

1. Make any data field **private** that does not have to be more openly exposed.
2. Make any data field that does not need to change after construction **final**. These would be all fields for **@Immutable**. **@Immutable** applies if and only if all such **final** fields are a) primitive types like integers, or b) immutable objects like java.lang.String, or c) mutable objects for which your containing class never make mutations or provides methods for mutation on these objects after construction. Category c is known as being *effectively immutable*.

3. Make any data field accessed by the Pipe* constructor call (which occurs in the main thread) and the active object thread (which runs in the run() method) as **final** if possible, else **volatile**, since we need to guarantee that changes made in the constructor are visible to the object's run() thread. We could get the same cache-consistency effect by locking the object, but we are not programming with locks yet.
4. At the top of the run() method **copy any volatile field into a local variable**, and **use that local variable in the run() method**. These volatile fields are not used outside the run() thread once it begins; copying them for use into locals eliminates unnecessary cache flushes and main memory fetches; local variables are thread-confined and not subject to inter-thread cache consistency problems. (Note: If there are no volatile fields, you can ignore this requirement.)

I will be grading on all of these requirements. See rubric percentages on the previous page of this doc.

For the above Pipe* files, you need to change a number of non-blocking calls to blocking calls for BlockingQueue objects. See **STUDENT** comments. Those blocking methods declare that they throw java.lang.InterruptedException. In places where my handout code does not catch this exception, you must catch and ignore it. There can be no InterruptedExceptions in this program because we do not use Java interrupts for communication between threads. In later assignments we will use them. For now an empty catch clause where mandated by the compiler is fine.

In BigDecimalPipelineBuilder you will find this comment:

```
// STUDENT write this block of code.
// Write code to run each pipeline stage in transformers
// in its own Thread (start running from transformers.get(0) in
// order to pump up the queues),
// See main thread in
// ~parson/multip/demo1spring2023/GarbledOutput.java
// for loops to construct, start, and join threads.
// STUDENT 10%: CODE GOES HERE:
```

Your solution in this block of code must **start a Thread** of execution in left-to-right sequence for each pipeline stage, then **join()** each Thread **after they have all started** via Thread.join() (a separate join loop).

Finally, you must add tests to script **testscript**, whose listing appears below. I have highlighted requirements below. NOTES ADDED AFTER INITIAL DESIGN SPEC FOR ASSIGNMENT:

1. The classes that you add in the testscript **runatest** lines should all come from package java.util.concurrent. Do not add Queue classes from java.util or elsewhere.
2. Try all classes from java.util.concurrent that implement the Queue<E> interface. Some of those also implement the BlockingQueue<E> interface, for which you can also add the runatest **multithreading, blocking** parameters of "true, true" in addition to "false, false" and "true, false". Do not consult base classes such as java.util.AbstractQueue<E> in determining which classes to try. Not all valid classes in java.util.concurrent subclass such abstract base classes.
3. Some of these java.util.concurrent queue classes will not work **for reasons that should be obvious from their documentation**. If not, ask in class or consult with Dr. Parson.

testscript

```
#!/bin/bash
# testscript runs a series of tests for csc543 assignment 1 Spring 2023 D. Parson

# HERE IS DR. PARSON'S HANDOUT TEST FUNCTION. KEEP IT UNCHANGED & CALL IT.
function runatest() {
    # Parameters:
    # $1 is USETHREADS
    # $2 is USEBLOCK
    # $3 is the name of the Queue class; use its full package path in the call.
    # $4 is 1 to exit on run-time error, 0 to not exit
    CLASSPATH=..:jcip-annotations.jar time java pipeline2023.BigDecimalPipelineBuilder $1 $2
pipeline2023.PipeSourceRandom "12345 3000 3000" $3 pipeline2023.PipeStageMath '+' $3
pipeline2023.PipeSinkFile sink.out
    exitStatus=$?
    if [ $exitStatus -ne 0 ]
    then
        echo "TEST ERROR" 1>&2
        exit $exitStatus
    fi
    diff ./sink.out sink.ref > sink.dif
    exitStatus=$?
    if [ $exitStatus -ne 0 ]
    then
        echo "DIFF ERROR" 1>&2
        exit $exitStatus
    fi
    /bin/rm -rf ./sink.out
}

# 1. HERE IS MY TEST. STUDENT TESTS GO AT THE BOTTOM.
runatest false false java.util.LinkedList   # This is the first test.
# The first "false" above is USETHREADS, and the second is USEBLOCK.

# IF POSSIBLE:
# You should run all of your tests with USETHREADS=false, and then, for the
# Queue objects that are thread-safe, run again with USETHREADS=true.
#
# For the Queue objects that are BlockingQueue objects, make a third test run
# with USETHREADS=true and USEBLOCK=true.

# We are testing the multithreaded
# BlockingQueues using both their non-blocking and their blocking interfaces.
# You may not be able to run some of the Queue types in all possible
# testing configurations. For example, trying to write to a size-bounded
# queue may throw an exception and kill the program. Also, some of the
# queue type(s) in java.util.concurrent require constructor parameter(s), which
# are not supported by this test setup. For the ones that you cannot run,
# ADD A COMMENT line below explaining why you cannot run that set of
# USETHREADS and USEBLOCK parameters.
```

# STUDENT 40% of assignment CODE TO GO AT THE BOTTOM:

```
# Write your tests, substituting for java.util.LinkedList a
# thread-safe Queue or BlockingQueue class path,
# and varying USETHREADS and USEBLOCK command line arguments as indicated in
# the above paragraphs. Use blank line(s) to separate tests so I can see them.

# NEXT ONE NOT THREAD SAFE, IT BLOWS UP IN PREP SOMETIMES.
# runatest true false java.util.LinkedList
```

To earn all 40% you must test **ALL** Queue and BlockingQueue classes in package java.util.concurrent with the correct arguments in testscript. Use all **VALID** combinations of USETHREADS and USEBLOCK for each queue class you test.

Get all of your Queue and BlockingQueue classes for testscript from java.util.concurrent. As noted above, some combinations of USETHREADS and USEBLOCK may not work for some of those classes, and in fact some of the classes may not work with our test setup. Use ALL of the ones in java.util.concurrent that complete without inherent errors (fix your bugs), and WRITE A COMMENT FOR EACH Queue/BlockingQueue test line that does not work, and explain why.

NOTE THAT YOU WILL ONLY BE ADDING TEST LINES THAT LOOK LIKE THIS

```
runatest false false java.util.LinkedList   # Replacing false with true and other Queue classes, but do not change
# function runatest. It is already done.
```

Run **make testtime** to create a time-sorted execution profile of tests on each machine after you have the program working. Profiling output appears in file **testtime.txt**. You must **make testtime** on **mcgonagall**.

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.