Homework Assignment 8

This assignment will give you practice with Java's concurrency mechanisms. The last question will have you implement the JTalk lab from CS360, so it will also give you an opportunity to compare and contrast the C and Java implementations (at least for those of you who have taken CS360 or are currently taking it).

1. Write a Java program called that simulates a bicycle race among a number of threads. Your program must take one command line argument representing the number of contestants. Each contestant in the bike race will be represented by a thread. The bike race is divided into 4 segments. For each segment, a thread will generate a random number between 1000 and 8000 milliseconds, which represents the time it takes the contestant to finish that segment of the race. The thread will then sleep for this number of milliseconds. When it awakens, it will report to a central time keeper that it has completed this segment of the race and it will report its time for this segment of the race. When the thread has finished all four segments, it will ask the central time keeper to add it to a finish queue.

   The central time keeper will be a class that keeps track of the following information:

   a. the cumulative time of each contestant, the number of segments completed by each contestant, whether a new segment time has been reported since the last score report (more on this shortly), and the time required by the contestant to complete the last segment.
   b. the finish order of the contestants, which will be represented by the order in which the contestants are added to the finish queue. There is a slight possibility that one contestant could finish all four segments before another contestant, but the race is not over until the contestant is in the finish queue!

   Every two seconds your program should print an updated scoreboard showing each thread's contestant number (which will be a number from 1 to n where n is the number of contestants), the number of segments completed by the contestant, the contestant's cumulative time to complete these segments, and if the thread reported a new result in the last two seconds, then the time of the last segment completed by the contestant (this last piece of information will help both you and the TA determine what is happening in your program). Once all the contestants have finished, your program should print both the final scoreboard and the order in which the contestants actually finished, by printing the finish queue. You will need to implement a separate thread for this reporting task, although you may place the methods for printing the scoreboard and the finish queue in the time keeping class (the reporter thread will call these methods).

   Additional requirements:

   a. You must use 5 classes for this problem:
      i. Contestant.java: The thread class for a contestant
      ii. Reporter.java: The thread class for the reporter. I understand that you could do the reporting in main, but I want you to have experience creating separate thread classes.
      iii. TimeKeeper.java: The timekeeping class
      iv. SynchronizedRandom.java: A class that your contestants use to obtain a random number. It should have a constructor that creates an instance of Java's Random() class. This instance automatically gets seeded, so you do not have to seed it yourself. SynchronizedRandom should also have a synchronized method called `nextInt()` that returns a random number. An instance of this class should use an instance of Java's Random() class to generate the random numbers. All contestants should use the same instance of SynchronizedRandom to obtain their random numbers (you will need to pass this instance to the constructor for each Contestant).
      v. Race.java: The "glue" class that contains main and which creates instances of the TimeKeeper, Contestant, and Reporter classes.
   b. You must place your classes in a package named `race`.
   c. Your central time keeper class should use synchronized methods only when it is necessary.
   d. Do not use wait or join. Only use sleep for this problem.

   Sample Executable

   `/home/bvanderz/cs365/hw/hw8/race.jar` contains a sample executable for this problem. A sample invocation for 3 contestants would be:

   ```
   java -jar race.jar 3
   ```

2. In the central time keeping class for question 1, there were two methods you needed to write (these two methods are mandated by the paragraph that describes the Contestant thread):

   a. a time reporting method that allows each thread to report its time when it completes a segment, and
   b. a finish method that allows a thread to report that it has finished the race.

   For each method, indicate whether or not you made the method synchronized, and justify your decision for doing or not doing so.

3. Professor Jim Plank's JTalk assignment implemented in Java (**you do not need to look at his description of the assignment--I have completely specified the problem**). I am modifying it somewhat so that it works for Java. You will be creating a chat server using Java threads and Java sockets, rather than pthreads and telnet/jtelnet. The syntax of your server is (you will have to use a classpath flag with all these examples):

   ```
   UNIX> java jtalk.ChatServer port Chat-Room-Names ...
   ```

   For example, if you'd like to serve chat rooms for football, bridge and politics on port 8000, you would type:

   ```
   UNIX> java jtalk.ChatServer 8000 Football Bridge Politics
   ```

   You will need to be on the server machine when you launch the chat server.

   The syntax for attaching a client to the server is:

   ```
   UNIX> java jtalk.TalkClient hostname portname chatname
   ```

   Suppose, I have launched the chat server on hydra4.eecs.utk.edu using port 8000 and a person wishes to attach to the server with the chatname "Dr-Plank". The person would type:

   ```
   UNIX> java jtalk.TalkClient hydra4.eecs.utk.edu 8000 Dr-Plank
   ```

   Here is a sample chat between a number of users (dark red input is what the user types in the console window):

```
UNIX> java jtalk.TalkClient hydra4.eecs.utk.edu 8000 Dr-Plank
Bridge:
Football:
Politics:

Enter chat room:
Bridge
Dr-Plank has joined
There's no one here...
Dr-Plank: There's no one here...















Goofus has joined
Hi Goofus -- do you like bridge?
Dr-Plank: Hi Goofus -- do you like bridge?

Goofus: Bridge? You mean that card game my gramma plays?
Indeed
Dr-Plank: Indeed

Goofus: Loser.  Bye.


Goofus has left
Can't say I liked him.
Dr-Plank: Can't say I liked him.















Gallant has joined
```

On cetus4:

```
UNIX> java jtalk.TalkClient hydra4.eecs.utk.edu 8000 Goofus
Chat Rooms:

Bridge: Dr-Plank
Football:
Politics:

Enter chat room:
Bridge
Goofus has joined

Dr-Plank: Hi Goofus -- do you like bridge?
Bridge? You mean that card game my gramma plays?
Goofus: Bridge? You mean that card game my gramma plays?

Dr-Plank: Indeed
Loser.  Bye.
Goofus: Loser.  Bye.
<CNTL-C>
UNIX>

UNIX> java jtalk.TalkClient hydra4.eecs.utk.edu 8000 Gallant
Chat Rooms:

Bridge: Dr-Plank
Football:
Politics:

Enter chat room:
Bridge
Gallant has joined
Hi Dr. P
Gallant: Hi Dr. P

Dr-Plank: Greetings, Gallant
```

```
Gallant: Hi Dr. P
Greetings, Gallant
Dr-Plank: Greetings, Gallant

Gallant: After memorizing your lecture notes,

Gallant: I like to read books on bridge.
I will recommend you for many jobs & scholarships.
Dr-Plank: I will recommend you for many jobs & scholarships.
<CNTL-C>
UNIX>
```

```
After memorizing your lecture notes,
Gallant: After memorizing your lecture notes,
I like to read books on bridge.
Gallant: I like to read books on bridge.

Dr-Plank: I will recommend you for many jobs & scholarships.



Dr-Plank has left
I didn't get a chance to be more sycophantic!
Gallant: I didn't get a chance to be more sycophantic!
<CNTL-C>
UNIX>
```

To be descriptive, when a client joins, the server sends it information about the current chat rooms. The chat room names will be listed lexicographically, and the name of each person chatting should be listed with each chat room, separated by a space. The order of that listing should be the order in which the chatters joined.

The client then prompts the user for a chat room. It should error check for a bad chat room but do *not* worry about a premature EOF. Once the person joins the chat room, a message is sent to all others in the chat room that the person has joined. Lines of input entered by the clients are sent to all the clients in the chat room.

The server should support *any* number of clients, and should work seamlessly when clients leave, as Goofus, and later Dr-Plank did above. The server must not print any output. Feel free to add output to your server while testing it, but please remove it in your final submission. We will only test the behavior of the clients.

Structure

This lab is involved, and will use Java threads, synchronized methods (in place of the C mutexes and condition variables you used in the CS360 C implementation), and Java sockets. If you are having trouble getting started with the lab, look at the Java KnockKnock server socket tutorial, as its structure is what you should start with. At the bottom of the tutorial is an implementation for a multi-client KnockKnock server, which is what I used to start writing the client and server classes for this assignment.

Server Side Implementation

Suppose there are *r* chat rooms and *c* clients. Then your **ChatServer** will have *c+1* threads and *r* instances of a chat room class named `ChatData` that will handle the message traffic for the chat rooms. Note that unlike the CS360 JTalk lab, you will not use threads to implement the chat rooms.

Your *c+1* server threads will be allocated as follows:

- There will be one server "listening" thread that is spinning on a **while()** loop, waiting for clients to attach to the socket. When it detects a client, it will create a client thread. The server's "listening" thread will be your main method in jtalk.ChatServer. In my implementation, this thread also used a sorted map to keep track of the names of the chat rooms and their associated ChatData objects. It passed this map to each client thread that it forked, so that the client thread could send that information to the client. You should not use this thread to determine which chat room the client intends to join, since you would tie up the server and prevent other clients from joining. You want to be in a tight loop, so let the client threads determine which chat room the client wishes to join.

- There will be one client thread for each client. When that thread starts running it should send the names of the chat rooms and the chat room users to the client and ask the client to send back a chat room selection. The thread should then add the client to the appropriate ChatData object and then enter a while loop that reads from the socket. That thread will typically be blocked while reading from the socket. When it receives a line of text from the socket, it will construct the proper string from it and notify the appropriate ChatData object. When the client exits, the thread will drop out of the loop and should ask the ChatData object to remove the client. Note that unlike the CS360 JTalk lab, there will not be any subtle issues in dealing with client exit--you will get a null value when you try to read from the client's input stream and you can simply fall out of your read loop.

Your synchronized methods and data will be restricted to the ChatData objects. Each ChatData object needs to keep track of which clients have been allocated to it, so that it can distribute messages to these clients. My ChatData object had 4 primary methods:

a. distributeMessage: broadcasts a message to all clients in this chat room.
b. addClient: adds a client socket to the data structures for this ChatData object and creates a message indicating that the user has joined the chat room. This message is broadcast by distributeMessage.
c. deleteClient: removes a client socket from the data structures for this ChatData object and creates a message indicating that the user has left the chat room. This message is broadcast by distributeMessage.
d. broadcastMembers: writes a list of the chat rooms users to a new client so that the client can display the list of users in this chat room.

You are free to design your own implementation of ChatData and do not have to use the above structure.

Client Side Implementation

The TalkClient class will use main to establish a connection to the server and select a chat room. You will then face the problem of having to listen to two input streams: 1) stdin, where the user can enter new messages that you should send to the server, and 2) the socket's input stream, which will send you the list of messages to print in the chat window. You can't listen to two streams at once, since you will have to block while listening. Hence you will need to fork a thread that listens to one of the two streams, while main listens to the other stream. You will be managing three streams: 1) stdin, 2) the socket's input stream, and 3) the socket's output stream. You will pass lines from stdin to the socket's output stream. The output stream will in turn send them to the server. Finally, the server will send you lines to print in the console window via the socket's input stream.

You may want to draw yourself some pictures to help visualize the interactions between the server client threads, the chat data objects, and the threads on the client side.

Example Server and Clients

I have included two jar files named ChatServer.jar and TalkClient.jar with this assignment. They are located on the hydra machines at `/home/bvanderz/cs365/hw/hw8/`. You can run them if you have questions about how your program should work. The output from them is a bit different then the output in the examples. That is fine. Add blank lines if you like to make the output easier to read.

4. **Extra Credit (30 points)**: Do the online TNVoice evaluation for me and any of the TAs you wish to, and then upload a screenshot of the confirmation page to the Canvas TNVoice link. The link will become available once I receive word from UTK that TNVoice is ready to go. The link will be available through Wednesday, April 28.

What To Submit

1. Question 1: An executable jar file named `race.jar` that contains your 5 source files.
2. Question 2: An ascii text file or pdf file labeled hw8.
3. Question 3: Two executable jar files named `ChatServer.jar` and `TalkClient.jar`. ChatServer.jar should contain your source files, which at a minimum must include:
   a. ChatServer.java: Your main program for the server
   b. TalkClient.java: Your main program for the client
   c. ChatData.java: The class that manages the message traffic for each chat room.

   You will also need separate .java files on the server side for the server thread class and on the client side for the client thread class (i.e., the class that listens to either the stdin input stream or the socket's input stream). **Finally you must put all your files in a package named jtalk**.