**Homework 3**

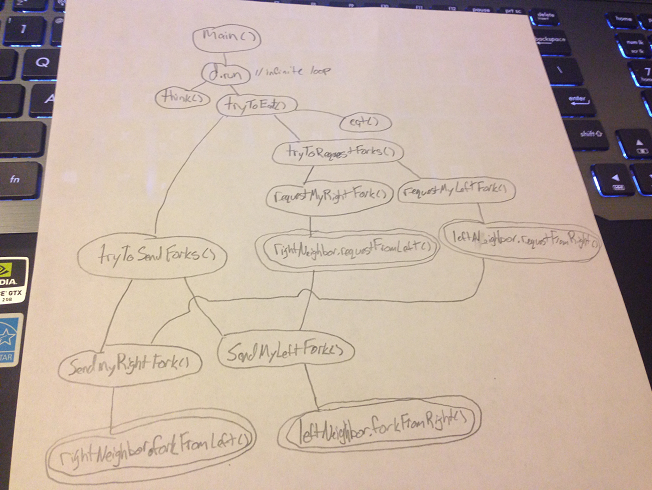
**Distributed Dining Philosophers Algorithm**

For this homework you will implement Chandy & Misra’s distributed algorithm for the Dining Philosophers resource sharing problem. You will start with an existing implementation: our first distributed algorithm (V2) in the version that uses RMI to communicate between Diner objects. This version is V2.3\_RMI\_DiningPhilosophers; it’s in the lab folder, or it can be downloaded from this assignment in d2l.

First you will investigate the existing implementation and answer some questions. This is designated below as Part 1. Then you will modify the implementation to use Chandy & Misra’s algorithm. This is Part 2.

**Part1: Investigate implementation V2.3 of the Distributed algorithm.**

1. Draw a call graph of Diner’s methods, starting with its method main(). Do not show invocations of RandomSleep, System.out, or SharedFork methods. Do not show Diner.showState(). Put a double-circle around the methods in IDinerNeighbor.



2. Why does requestFromRight() need to call sendMyRightFork(), and requestFromLeft() need to call sendMyLeftFork()? What could we have done instead?

requestFromRight() needs to call sendMyRightFork() because it is telling the diner that their neighbor is requesting a fork (from the right), so it needs to send its right shared fork to the diner with higher priority. The same holds true for the left, requestFromLeft() is telling the diner that their neighbor is requesting a fork (from the left) and so it needs to send its left shared fork to the diner with higher priority. Alternatively, we could have made it so requestFromRight() and requestFromLeft() both call tryToSendForks(), but that adds another invocation to a method with comparisons that may affect run-time which isn’t exactly needed here because we already know which fork to send.

3. There are four “server” methods (the ones in IDinerNeighbor) , and there are four “client” methods. The client methods are the ones that *invoke* a method in IDinerNeighbor. What are the four client methods?

The client methods that invoke the methods in IDinerNeighbor are requestMyRightFork(), requestMyLeftFork(), sendMyRightFork(), and sendMyLeftFork().

4. The current implementation prints out information about the thread that is executing. It does this in the four server methods and the four client methods. Is a Diner multi-threaded? Figure out what thread(s) a Diner is using, and explain it.

If you are executing Diners5 from the command line, a simple way to be able to read the messages is to kill a Diner while it is eating. This will stop its neighbors, which will in turn stop their neighbors. Then you can read the output of the remaining diners.

Yes, the diner is multi-threaded. By using the method described above to read the output, I closed the JVM that contained diner5. After looking at the information that is printed out by Thread.currentThread().getName(), we see that diner1 uses threads that are named main, RMI TCP Connection(1), and RMI TCP Connection(2). Similarly, diner2 uses main, RMI TCP Connection(2), and RMI TCP Connection(3). Diner3 uses main, RMI TCP Connection(2), and RMI TCP Connection(3) also. Diner4 uses main, RMI TCP Connection(1), and RMI TCP Connection(2). This shows that the diners are indeed multithreaded. If they were not, all diners would be either all using main or all using the same TCP Connections.

5. Make the four client methods synchronized. What is the behavior? Explain it.

With the four client methods synchronized, the program executes normally, but now only 1 thread can access the client methods at a time. With synchronization, once a thread calls a synchronized method a lock is put on it. This forces all of the other threads that try to access the method to wait until the current thread is done running. Here, the synchronized methods are requestMyRightFork(), requestMyLeftFork(), sendMyRightFork(), and sendMyLeftFork(). The diners are taking turns requesting and sending their forks, thus the program runs without any deadlock or livelock.

6. In addition to synchronizing the client methods, make the four server methods synchronized. What is the behavior? Explain it.

With the client and server methods all synchronized, the program reaches deadlock almost instantly. All diners start out thinking, diner1 has both forks, diners 2-4 have right forks and diner 5 doesn’t have any. In addition to the methods being synchronized, in the sendMyLeftFork() and sendMyRightFork() methods, the left and right fork objects are also synchronized respectively, meaning that the method locks the left and right fork to update their states and wont unlock until its done. When I ran my program I saw that diner5, diner4, and diner2 become hungry. Diner5 requests right fork from diner1, diner4 requests left fork from diner 3, diner2 requests left fork from diner1. Diner1 is still thinking and won’t give up his forks.

7. Now remove the “synchronized” from the client methods, but leave it on the server methods. What is the behavior? Explain it.

The behavior here is livelock. The program looks like it is executing nicely, but the diners need to take turns requesting and sending/recieving their forks. By watching each diners JVM’s run together, I saw that sometimes the text would stop for a short amount of time until one of them updated and then the rest of them would continue. To me this looks like what I explained above, the diners had to wait to request or send the fork, then the program would continue to execute.

8. Now add “synchronized” to requestMyLeftFork(). (The other three client methods are not synchronized, but all server methods are synchronized.) What is the behavior? Explain it.

Here we have reached another deadlock. Diner1 starts with both forks, diners 2-5 became hungry. Diner2 requests the left fork in main, diner3 requests the left fork in main, diner 4 requests the left fork in main, diner5 requests the right fork in main. Diner1 tries to send his right fork to diner2 in RMI TCP (7), but the program halts, diner2 never receives the left fork because sendMyLeftFork() puts a lock on the left fork object.

9. What is the purpose of the attribute “initialized” in Diner?

The purpose of initialized is that the program cannot print the initial state or start handling requests until all the diners are sitting at the table. The initialized Boolean is set to true once every diner has a left and right neighbor. Once all the neighbors are set, then the program can run with no errors.

10. Wasn’t this fun? Ok, maybe not exactly fun. But did it help you understand the current implementation?

I thought it was fun at first, but tracing and retracing the algorithm started making my head spin. I do feel much more comfortable with understanding the concepts of deadlock and livelock though!

**Part 2: Modify the implementation to use Chandy & Misra’s algorithm.**

Below you will see pseudo-code for the current implementation, followed by pseudo-code for the Chandy-Misra algorithm.

* Make sure you update method showState() to show whether a fork is clean or dirty.
* You can run your solution using Diners.bat and Diners5.bat.
* Make sure your initial state is not symmetric—one of the diners should hold two forks, one should hold none, and the other diners should have exactly one fork. All forks are initially dirty, and all diners are initially THINKING.
* Keep all methods in SharedFork synchronized.

It is hard to debug the algorithm, because you see each diner’s state separately. If you’re having a lot of trouble debugging your algorithm, temporarily turn the program back into a multithreaded version running in one JVM. Then you will be able to see how the diners’ executions are interleaved.

**Existing algorithm:**

THINKING = true; HUNGRY = false; EATING = false;

Loop forever

{

/\* can I request a fork or eat? \*/

if HUNGRY and not have\_right\_fork

request\_right\_fork();

if HUNGRY and have\_right\_fork and not have\_left\_fork

request\_left\_fork();

if HUNGRY and have\_right\_fork and have\_left\_fork {

HUNGRY = false;

EATING = true;

sleep();

}

/\* do I have to send a fork? \*/

if THINKING and have\_right\_fork and right\_neighbor\_wants\_fork

send\_right\_fork();

if THINKING and have\_left\_fork and left\_neighbor\_wants\_fork

send\_left\_fork ();

if HUNGRY and have\_left\_fork and left\_neighbor\_wants\_fork

and not have\_right\_fork

send\_left\_fork ();

if EATING and done\_eating {

EATING = false;

THINKING = true;

sleep();

}

if THINKING and getting\_hungry\_again {

THINKING = false;

HUNGRY = true;

}

}

**Chandy-Misra algorithm:**

THINKING = true; HUNGRY = false; EATING = false;

Loop forever

{

/\* can I request a fork or eat? \*/

if HUNGRY and not have\_right\_fork

request\_right\_fork();

if HUNGRY and and not have\_left\_fork

request\_left\_fork();

if HUNGRY and have\_right\_fork and have\_left\_fork {

HUNGRY = false;

EATING = true;

sleep();

}

**/\* do I have to send a fork? \*/**

**if not EATING and left\_neighbor\_wants\_fork and left\_fork\_is\_dirty {**

**set\_left\_fork\_clean();**

**send\_left\_fork();**

**}**

**if not EATING and right\_neighbor\_wants\_fork and right\_fork\_is\_dirty {**

**set\_right\_fork\_clean();**

**send\_right\_fork();**

**}**

if EATING and done\_eating {

EATING = false;

THINKING = true;

set\_left\_fork\_dirty;

set\_right\_fork\_dirty;

sleep();

}

if THINKING and getting\_hungry\_again {

THINKING = false;

HUNGRY = true;

}

}