CMPU4021 Distributed Systems - Labs

Concurrency and Multithreading in Java

Learning Outcomes:

1. Be able to create Java threads using both the Thread class and the Runnable interface.
2. Be able to describe the various states that a Thread can be in, and how to transition between these states.
3. Be able to develop Java code that uses built in locks and monitors.

Tasks

1. Take a look, compile and execute T1\`ThreadShowName.java`. Notice how the two threads overlap in execution.

1.1. Take a look, compile and execute `T1_1\SleepMessages.java`
The `SleepMessages` example uses sleep to print messages at four-second intervals. Notice that main declares that it throws InterruptedException. This is an exception that sleep throws when another thread interrupts the current thread while sleep is active. Since this application has not defined another thread to cause the interrupt, it doesn't bother to catch InterruptedException.

2. Take a look, compile and execute T2`\ThreadHelloCount.java.` Notice how the two threads belong to different classes.

3. Take a look, compile and execute `T3\RunnableShowName.java` and `T3\RunnableHelloCount.java`. Observe how the `Runnable` interface is used instead of the Thread class. Observe also, how the instantiated object is still a Thread.

4. Take a look, compile and execute the client server application (T4\`MultiEchoServer.java` and T4\`MultiEchoClient.java`). This application is the same as last week's client server application, except every client that connects to the server has its own thread on the server.

4.1. Take a look, compile and execute `T4_1\ SimpleThreads.java`
`SimpleThreads` consists of two threads. The first is the main thread that every Java application has. The main thread creates a new thread from the `Runnable` object, `MessageLoop`, and waits for it to finish. If the `MessageLoop` thread takes too long to finish, the main thread interrupts it.

The `MessageLoop` thread prints out a series of messages. If interrupted before it has printed all its messages, the `MessageLoop` thread prints a message and exits.

5. Download T5\`ResourceServer.java`, T5\Resource.java, T5\`ClientThread.java` and T5\`Producer.java`. Take a look, compile and run the server. Observe the operation of the server and the synchronized blocks.

Note: This is a producer-consumer problem, in which a producer is generating instances of some resource and a consumer is removing instances of the resource. The resource is modelled by the *Resource* class, the producer by the *Producer* class. The *Producer* class is a thread class, extending class *Thread.* The server program, *ResourceServer* creates *Resource* object and then a *Producer* thread, passing the constructor for this thread a reference to the *Resource* object. The server then starts the thread running and begin accepting connections from client. As each client makes connection, the server creates an instance of *ClientThread* (another Thread class), which is responsible for handling all subsequent dialogue with the client.

6. Take a look, compile and execute `T6\SemApp.java.`
In some enterprise systems, it's not uncommon for developers to need to throttle the number of open requests (threads/actions) against a particular resource. While it is possible to try to write the throttling code by hand, it is easier to use the semaphore class, which takes care of throttling for you, as shown in SemApp.java:

Even though the 10 threads in this sample are running, only three are active. The other seven are held at bay until one of the semaphore counts is released. The `Semaphore` class supports acquiring and releasing more than one *permit* at a time, but it would not be appropriate in this scenario.

Note: The concept of “throttling concurrency,” refers to the number of threads that are allowed to do work at a particular time. It is a policy to decide how many threads can be run simultaneously without hurting performance. It is a way to limit the amount of possible active threads at one time.

T7. **Write** a client for the ResourceServer. Call it ConsumerClient.

Note: Take a copy of MultiEchoClient, renaming it ConsumerClient. Using this file as a template, modify the code that it acts as a client to ResourceServer. Ensure that the user can pass only 0 or 1 to the server (e.g. show an error if user enters 2). Test the operation of the server with two clients.

T8. Write a server that maintains a list of **(at least)** two clients. Clients take turns sending messages to the server. When the server receives a message from either client it sends the received message to the other client. Use multi-threading where appropriate.

Further Reading

1. Java Tutorial section on Concurrency
   https://docs.oracle.com/javase/tutorial/essential/concurrency/

   https://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html

2. Chapter 3 of Introduction to Network Programming in Java by Graba.

3. 5 things you didn't know about ... java.util.concurrent, Part 2
(http://www.ibm.com/developerworks/library/j-5things5/)

4. The additional code that we spoke about in class is available here the `Addition` subdirectory.:

- Main.java
- Producer.java
- Consumer.java
- MyData.java (Version 1)
- Rename MyData_v1.java to MyData.java. Take a look, compile all the code and run main. Notice how the threads are not properly synchronised.

- MyData.java (Version 2) - uses busy loops.
- MyData.java (Version 3) - uses Java object monitors and busy loops.
- MyData.java (Version 4) - uses Java object monitors on code segments rather than whole methods. This helps avoid any deadlock situations.
- MyData.java (Version 5) - uses Java object monitors with `wait`() and `notify`() methods to avoid the requirement for busy loops.