COSI 131a: Fall 2016
Programming Assignment 2

Overview

Your project is to implement a simulation of customers issuing requests to use shared resources. There will be rules that limit when different customers may make use of a resource. The simulation will represent customers using Java Threads and implement the rules using Java synchronization.

The project is split into two tasks. In task 1, your simulation program will need to use Java default synchronized methods and busy waiting to enforce the rules limiting the resource use. Busy waiting can work well if the waiting is very short but can be inefficient if the waiting is long.
In task 2, you will modify the simulation program to avoid busy waiting and to provide FIFO order for consumer requests. You will have a centralized controller that coordinates consumer access to multiple instances of the shared resources. Your controller will build on the synchronized code you developed in task 1, and will add Java condition variable synchronization to avoid busy waiting.

Task 1 has an earlier due date than Task 2. See LATTE for due dates.

Customer types and rules limiting consumer access.

The following rules limit, subject to customer type, when, where and with whom client threads representing customers can get access to resources managed by servers. Your code must enforce these rules in both Task 1 and Task 2.

- Client (General)
    - Has the following customer attributes: type, name, industry, request level, speed.
    A client must have a toString() method that returns “INDUSTRY TYPE NAME” i.e. “TECH BASIC AA1”.
    - **Type** can be *shared* or *basic*.
    - Concurrent customers of the same **industry** can not share the same resource so concurrent clients getting access to the same server must be of a different **industry**.
    - **Request level** determines the number of requests that a client still needs to process before the client ceases operation. A client with request level 0 will not attempt to connect to a server. Each successful connection decrements request level. A client starts with level 3.
    - **Speed** determines how many milliseconds a client will hold the resource (to simulate request processing) before disconnecting from the server after successfully gaining

access to a server. Higher speed value implies shorter processing. The exact formula for holding time is *((10 - - speed) * 100) ms*. I.e. a client with speed 8 will gain access to a server, wait for 200ms then disconnect from the server. A client can have a speed between 0 and 9.

- Servers (General)
  - Clients make a request to utilize a server by calling Server.connect()
  - Server.connect() will return true (client has successfully connected) if the client's joining would not violate any of the constraints of the server (summarized below under Basic Server). Otherwise Server.connect() returns false.
  - Clients make a request to release the server by calling Server.disconnect()

- Basic Server (Task 1)
  - An instance of a basic server will not service:
    - more than two shared clients at the same time.
    - more than one basic client at a time.
    - a shared and basic client at the same time.
    - two shared clients of the same industry.

- Master Server (Task 2)
  - Distributes clients to their respective basic servers.
  - Keeps list of references to basic servers (in a HashMap).
  - Keeps list of references to the queues of each basic server.

As you know from learning about concurrency, without proper synchronization, a client thread trying to enforce these constraints is prone to race conditions. E.g. a thread could check to see it can use a server, then be suspended before it can actually use the server, at which point another thread may also check to see if it can use the server and do so. When the first thread resumes again and proceeds to use the server, one of the invariants may be violated (e.g., maybe a basic client and shared client will be using the same server at the same time).
Part of your job in this assignment is to use synchronization to prevent such race conditions from occurring.

To implement your solutions you will need to use the code base provided by us, described below. This code base includes a test harness that allows us (and you) to test the correctness of your solutions. You will be therefore instructed to follow certain conventions in your code and will be not allowed to modify certain classes.
For example, your code will not be starting or joining client threads. Instead, the threads will be started by the test harness. Please follow the instructions carefully.

You will need to understand how the entire provided code works to complete your assignment.

Provided Code

The code for this assignment is divided into the following packages

edu.brandeis.cs131.Common.Abstract

The Abstract package contains the abstract classes and interfaces that your program must implement. You are not allowed to modify any of the files within this package.

edu.brandeis.cs131.Common.Abstract.Log

The Abstract.Log package contains classes used to record the actions of clients. These classes are used by the test packages to evaluate that constraints have not been violated. The log is not a means for passing messages. The only reference to Log classes in your code should be in the ConcreteFactory.createNewMasterServer method. You are not allowed to modify any of the files within this package.

edu.brandeis.cs131.Common.YourName

The *YourName* package contains a few mostly empty classes that implement the classes found in the Abstract package. You should initially rename this package to your first then last name with no spaces i.e. JaneSmith. All of the code pertaining to your solution of this assignment must go in this package. You are free to add, edit, rename, and delete files in this package in the course of implementing your solutions.

Test Files - edu.brandeis.cs131.Common.Test

You must modify TestUtilities to include your version of ConcreteFactory (Modify the import statement to include your ConcreteFactory), in order for the test suite to run. You may not edit any of the other test classes, but we highly encourage you to study them for your understanding and to aid in discovering why a test is failing.

API Documentation

You will want to inspect the java files yourself, but to get you started we include the API docs generated with javadoc from the files we provide.

Task 1: The Basic Server

You must make class(es) that extend the abstract class Client and display the behavior described previously.
You must create a class that extends the abstract class Server and carries out the following tasks:

- Enforce the Server Restrictions described previously.

- Use the Java default synchronized keyword to achieve mutual exclusion in the critical sections and prevent race conditions.

You must setup your ConcreteFactory to create and return your newly created classes.

When this task is complete your code should pass all tests included in BehaviorTest, and SimulationTest.Basic_Server_Test.

Task 2: The Master Server

You have now successfully programmed BasicServer that enforces entry criteria and prevents race conditions. However, there are deficiencies in the design of BasicServer:

- When the server is not available to a client, that client busy--waits (loops over the list of known servers repeatedly inside of its run() method).
- The server does not provide any guarantees on the ordering of the requests.

In this task, you will use Java locks and condition variable synchronization to improve your simulation. First, you program will need to avoid busy waiting. Second, each client will have a designated instance of a basic server (handled by the hashmap in the master server) based on its speed and the access to a particular server will be granted in the FIFO order of requests.

When this task is complete your code should pass all tests included with this assignment.

Implementation

You must create a new class (MasterServer) that extends Server and contains a map of BasicServers, each of which have their own FIFO wait-queue The client in the front of a BasicServer's wait-queue will attempt to connect to that server, and if unsuccessful wait for a client currently connected to that server to disconnect. BasicServer should be the same class you implemented in the first part of this project.

MasterServer will carry out the following tasks:

- Keep references to BasicServers and their queues as private member variables inside MasterServer.
- When a call is made to connect(client) on a MasterServer instance, the following general steps should be executed:
    - The client should use the provided getKey() method to determine what server it has been assigned access to. If the queue for this server is non-empty, the client adds itself to the tail of the queue and waits for its turn.
    - The head of the queue is allowed to proceed. That client will try to make a request to that server. If it cannot make a request due to a server condition being

violated, the client thread must wait. Otherwise, the client successfully connects to the server.
    - After a client successfully connects to a server it should remove itself from the queue and inform the new head of the queue so it may attempt to connect. Recall, sometimes Shared Clients can enter a server at once.

- When a Client exits the MasterServer by calling disconnect on a MasterServer instance, the MasterServer must call disconnect() on the BasicServer the client is currently connected to, possibly allowing the head of the server queue to gain access. Remember, you may not modify Server.java or any of the other classes provided to you, and should not have to change your implementation of BasicServer, so you must solve this problem within MasterServer.

.

- A client only removes itself from the queue, if it successfully requests from a BasicServer, otherwise it waits.
- Only the head of the queue tries to enter a BasicServer. All other clients wait their turn (even if they could enter simultaneously).
- Use condition variables to avoid busy -waiting when the server assigned to a client is busy. Make sure the use of the condition variables is safe!
- When using condition variables, avoid informing clients waiting to gain access to one server about events pertaining to another server.

No ‘main’ method

Note that your only point of entry in the code we provide is through the JUnit tests, which will set up the environment in which your client threads will run. Your tasks for this assignment do not include writing a main method. Rather, you must rely on your understanding of busy-waiting/monitors, and the JUnit tests to know if you have completed the assignment.

Important Note about Disallowed Java Tools

In PA1, you were instructed to consider using a synchronized class provided by Java for inter--thread communication (LinkedBlockingQueue) to solve the problem. For this project, that is not allowed; you may not use any synchronized data structure included in the Java API. You must write your own, using the "synchronized" keyword, Java locks and condition variables. Of course, you can and should use non--synchronized data structures in the Java standard library. You can consult the API docs to see if a data structure is synchronized.

Submission Guidelines

Create a zip or tar.gz file containing your completed code. You should name this file: last name underscore first name. For example if your name is Jane Smith you should name your file: Smith_Jane.zip or Smith_Jane.tar.gz. This file should contain the src folder within the archive -not as the top level (when the archive is opened the src folder should be visible). Upload this archive to the submission folder on Latte. Please also include a README file that contains a description of your implementation and any issues your code might have. This is an individual project and all submitted work must be your own. To this effect, to receive any credit, you will have to include in your README a statement, ***attesting the submitted work is yours***. So while you are allowed to discuss your solution with other students, make sure you ***do not share your code***. If you have extensive discussions with another student, please indicate this in your README file.

Helpful Hints

‘instanceof’ operator

You might find the instanceof operator helpful. instanceof can tell you if an object can be downcast from a parent type into a child type.

Here is an example:

```
Public class TestInstanceof{
     private static class  Parent     {}
     private static class  Child1 extends Parent {}
     private static class  Child2 extends Parent {}

     public static void main(String[] args) {
            Parent p1 = new     Child1();
            Parent p2 = new     Child2();

            if (p1 instanceof Child1) {
                System.out.println("p1 is instance of Child1");
            }
            if (p1 instanceof Child2) {
                System.out.println("p1 is instance of Child2");
            }
            if (p2 instanceof Child1) {
                System.out.println("p2 is instance of Child1");
            }
            if (p2 instanceof Child2) {
                System.out.println("p2 is instance of Child2");
             }
     }
}
```

This outputs:

```
p1 is instance of Child1
p2 is instance of Child2
```