Introduction to multithreaded programming in Java .

Show Submission Password

✔ Introduction to MultiThreaded Programming in Java

✔ A Simple MultiThreaded Banking Application

✔ Away with Race Conditions

Introduction to MultiThreaded Programming in Java

Introduction to MultiThreaded Programming in Java

As you may have discovered upon the completion of Project 1.1 and 1.2, there are distinct advantages in performance when using parallel programming as opposed to sequential programming. This is because modern computers are often optimized to have enough resources to handle many different tasks at the same time, such as having multiple CPUs. Java is a multithreaded programming language that takes advantage of the fact that operations within a task can often be performed in parallel. For more technical information about multithreaded programming and Java, feel free to explore tutorials like this one (http://www.tutorialspoint.com/java/java_multithreading.htm). For the context of Cloud Computing, you should at least understand that Java threads can run in parallel with one another, within the context of a single application.

Not All is Easy and Golden

You may be wondering why the whole programming world hasn't been parallelized yet, since the advantages are so obvious- more efficient use of resources and quicker overall program completion time. There are many reasons as to why this is, ranging anywhere from simple reality to high complexity. When confronting real world problems, we often face the common issue of whether or not a set of operations can even be parallelized at all. Some problems are inherently sequential, occurring in many different stages whose results depend upon previous stages. Thus, even if we were to parallelize these stages, it would still need to wait for the other stages' results to continue. Many of the problems presented to you here in Cloud Computing are designed specifically to be highly parallelizable, but this is not guaranteed in the real world.

And then, we have the issue of actually trying to program in parallel. Make no mistake, programming in parallel is complex and difficult, and prone to many subtle and unpredictable bugs. If you had any previous experience in distributed or parallel design, then you should know about the most famous concurrency bug- the race condition. In very simple terms, a race condition results when a program attempts to do some parallel operations at the same time, but requires that the operations are done in a specific order that is not guaranteed. They are difficult to detect because oftentimes, the ordering may be correct, making the system appear functional. Then, every once in awhile, one operation may complete out of order, causing the whole system to fail. Famous examples have caused disasters ranging from massive power failure (NorthEastern Blackout of 2003 (https://en.wikipedia.org/wiki/Northeast_blackout_of_2003)) to human fatalities (Therac-25 (https://en.wikipedia.org/wiki/Therac-25)).

Of course, it is unlikely that you will kill anyone with race conditions in this course, but failing to understand the key concepts behind race conditions and what causes them will cost you many hours of frustration in the coming weeks. By the end of this tutorial, you should be an expert on at least one mechanism on how to avoid race conditions.

A Simple MultiThreaded Banking Application

A Simple MultiThreaded Banking Application

In this tutorial, you will be completing a multithreaded banking application, which handles both deposit and withdrawal applications concurrently using threads.

The application has the following two functions available to the user:

Method Name	Purpose of Method
`deposit(final int amt)`	Adds the amount specified to the user's balance
`withdraw(final int amt)`	Removes the amount specified from the user's balance

Beware the "final" Keyword

Note that both input parameters contain the `final` keyword. This is because the inputs need to be immutable to ensure that its values do not change suddenly during the thread's lifespan. This idea falls under the category of thread safety, which you can read more about here (http://www.javamex.com/tutorials/synchronization_final.shtml).

The application also has an authentication method hidden to the user, called `authenticate()`, which you should not change. Due to poor optimizations, this function always takes 500 milliseconds to run. What's even worse, because this operation is so slow, the original designer has decided that the user only needs to be authenticated during withdraw requests, and not during deposit requests, since it doesn't matter if somebody wants to add money, only when they want to take money out. Thus, the authenticate function will only be called inside the withdraw method. Additionally, the withdraw method will read what the current balance is BEFORE authenticating, and uses this balance to carry out the withdraw operations after authenticating.

Because of this poor design, the application designer has been fired, and his work is incomplete. He has completed the methodology for authenticate, as well as withdraw. However, he did get to finish implementing the deposit function, except to add a wrapper print statement. Your task is to implement the actual method for deposit.

Warning

Retrieving the Incomplete Application Code

To proceed, download a copy of his unfinished work here (https://s3.amazonaws.com/15319-javamtp-primer/BankUserMultiThreaded.java).

To understand how to implement your own thread in Java, you should first read up on the Java documentation on Threads (https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html) and Runnables (https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html). For the context of this primer (as well as Project 3), you should at least understand that you can create and run a thread through the following basic steps:

1. Create a `Runnable`, and define its `run()` method.
2. Fill out all operations inside the `run()` method.
3. Create a `Thread` by passing your `Runnable` in as a constructor input.
4. Start the `Thread` you just created by calling `start()` on it.

To see all of this in action, look at the implementation of the withdraw method in the java file. Then, once you are familiar with how it works, try to fill out the code for deposit operations in the deposit method. Remember to move the print message to the spot you think appropriate, or will output strange things.

Remember, the deposits does NOT need authenticating, so make sure you don't call `authenticate()` like withdraw does!

For demonstration purposes, the image below outlines the basic steps described above in the actual code for the withdraw function

```
public static void withdraw(final int amt) {
    Long timestamp = get_time();
    Thread t = new Thread(new Runnable() {
        public void run() {
            int holdings = balance;
            if (!authenticate()) {
                return;
            }
            if (holdings < amt) {
                System.out.println("Overdraft Error: You have insufficient funds for this with
                return;
            }
            balance = holdings - amt;
            System.out.println("Withdrew " + Integer.toString(amt) + " from funds. Now at " +
        }
    });
    t.start();
}
```

1. Define new Thread with new Runnable
2. Fill in all operations inside run()
3. Start the thread when ready by calling start()

Luckily for you, the original designer actually created test cases in advance to test his work. You can make use of his test cases by uncommenting the calls to each test in the `main()` function. The first test case will test to ensure that you have implemented the deposit correctly. Once you have filled out the code for deposit, go ahead and run your Java application. To do this, simply compile and run the application (if on Eclipse), or use the following two commands on Linux:

```
javac BankUserMultiThreaded.java
java BankUserMultiThreaded
```

You should see a logging message for each deposit operation, and a final message telling whether or not your tests passed. If you've failed the tests, double check your implementation of deposit, and also make sure that you are not calling `authenticate()`.

Up until now, you haven't done much to ensure that the threads are playing nicely with one another, and that they are not accessing the internal balance in strange orders. The initial test #0 has been designed to avoid concurrency issues, but the remaining tests are not as forgiving.

Uncomment each of the tests 1 at a time (and only 1 at a time, or you may experience non deterministic results). Now, look at each of the test cases, predict what you think should be printed or happen, and then run your current application to see what actually happens.

Away with Race Conditions

Away with Race Conditions

As you may have noticed from running tests 1 to 4, the current implementation of the application is quite broken. The main problem resides from the fact that there is no insurance on which operations can take place, and when. This is a rather common issue when it comes to concurrent programming, and can usually be resolved by restricting access to a shared resource (in this case, the user's balance), and also ensuring an explicit ordering of the operations.

The application has the following two functions available to the user:

1. All operations must be executed in the same order they are received.
2. Only one operation can access the balance resource at the same time. During the period of access, no other operations can read or change its value.

These are generally the two most common principals enforced in concurrent programming- some sort of explicit ordering, often times in a queue-like fashion of first in first out, and a restriction of a shared resource when accessed by one thread.

Now that we have defined an ordering for operations, let's actually enforce it in our application. For ease of understanding, you are provided with the timestamp of the operation through the `get_time()` helper function, which gives you the current time as a `Long` type, with nanosecond precision. You can assume that all operations have a unique timestamp, and that if one timestamp is less than another, then the smaller timestamp occurred earlier.

Information

Notice the Timing

Pay close attention to the fact that the timestamp is created before and outside of the `Thread` 's run function in `withdraw` . Can you think of why that's important? Should this be done for deposit as well? The answer may not be as obvious as you think! Hint: Getting a thread to start running takes some overhead, so it may not start the instant you call `start()` on it.

Before proceeding, can you think of what data structures may be useful in determining the ordering of events based on their timestamps? If you're thinking that there are tons of options available, then you are correct! It's not difficult to think of a data structure that keeps ordering amongst its elements, priority queues, min heaps and sorted lists are all good options. In the application, a private `PriorityQueue` for `Long` s named `operations` has already been initialized for you, you may use this if you wish, or replace it with any other data structures you find fit. The java documentation for `PriorityQueue` s can be found here (https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html).

It will be up to you to figure out how to use this data structure to maintain ordering amongst the requests. If you are using `PriorityQueues` (or any other Queue-like structure), the `peek()` , `poll()` and `remove()` functions will prove essential.

Once you've determined how and which data structures you will use to satisfy requirement 1, it's time to move on to working out a way to satisfy requirement 2.

To properly understand how to tackle this task in Java, you should first read up on `wait()` , `notify()` , and `notifyAll()` . A good tutorial on these methodscan be found here (http://www.java-samples.com/showtutorial.php?tutorialid=306).

Information

Aside on wait, notify and notifyAll

Note that these methods are available under `java.lang.Object` , meaning that they are available for use for all Java Objects like `PriorityQueue` , but not primitives such as `int` . If you wish to use these methods on a primitive type like `int` , try using the corresponding `Object` instead, like `Integer` .

Below are the basic functionalities of each method for the purpose of our application. It's important that you understand each method by reading the documentation. The table below is simply a reference table for simpler usage.

Method Name	Purpose of Method
`wait()`	Causes the thread to sleep until another thread calls either `notify()` or `notifyAll()` on that same object.
`notify()`	Awakens an arbitrary thread that called `wait()` on that object.
`notifyAll()`	Awakens all the threads that called `wait()` on that object.

Information

Thinking about Objects

It is important to realize that a thread sleeping on `wait` will not be awoken if the object it's waiting on is changed. To wake a thread sleeping in `wait` , you must explicitly call `notify` or `notifyAll` . As a short exercise, what do you think would happen to a thread if it calls `wait()` on an object, and nothing ever calls `notify()` or `notifyAll()` on that object? Proceeding forward, for our application, can you think about the object that we need to call wait and notify on?

The last question was actually a trick question- it actually doesn't matter from a functional perspective, for our application at least. The `wait` and `notify` are actually simple methods of synchronized communication that is used to handle concurrency. Thus, we can use any `Object` to call `wait` and `notify` on, so long as it does not interfere with other resources using the same object to communicate between threads. Since we aren't using `wait` and `notify` anywhere else in our application, we can actually use any `Object` we want to call `wait` and `notify` on, so long as it's the same object that is accessible by all threads. However, it is generally considered proper protocol to use an object that's related to what we are doing to communicate- it both helps with readability, and also helps maintain the code in case you grow your application and need more concurrency management. In our case, we already have a data structure (`PriorityQueue` if you're following the skeleton code) for managing the ordering of operations, so why not use that?

Warning

Completing the Application

Now that you have a firm grasp of how `wait`, `notify` and `notifyAll` works, can you think of a way to incorporate that with the data structure you selected for managing ordering to fulfill both listed requirements? Try to write down the steps from each operation's lifespan from initiation to conclusion before actually implementing it. Then, if you're confident that you are ready, then go ahead and complete the application.

2 helper functions have already been created for you named `acquire_lock` and `release_lock`. If you wish, you can fill these in, and then simply call them in appropriate positions in the `withdraw` and `deposit` operations to complete the app. Notice that the `acquire_lock` operation takes in the timestamp, whereas `release_lock` does not. Think about why this may be if you are confused. Remember, you need to update both `deposit` and `withdraw` requests to be thread safe! And be warned, infinite loops are extremely common when dealing with concurrent programming, so don't get frustrated if your application hangs the first few tries. If you are stuck, log the status of your threads, what's hanging where, how your data structure is being managed and updated, and such.

Finally, once you are fully confident that you've implemented everything correctly, go ahead and run the tests again. Are the results what you expected now? Remember, even if things seem correct the first time around, be sure to run it multiple times to make sure you aren't missing any hidden race conditions! If your application has different behavior between tests, then something is wrong!