Show Submission Credentials

P3. Introduction to multithreaded programming in Java [Optional] .

16 days 2 hours left

Introduction to MultiThreaded Programming in Java

Introduction to Multi-Threaded Programming in Java

Information

Learning Objectives

This primer will encompass the following learning objectives:

1. Recall the basic Java classes and methods utilized in multithreaded programming
2. Adopt thread-safe classes in Java to avoid race conditions
3. Transform thread-unsafe classes into thread-safe classes using the Java synchronized keyword
4. Explore the implementation of read-write lock in Java and utilize it to ensure consistency in multi-threaded programs
5. Avoid common pitfalls in Java when building multithreaded applications

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 of the parallel operations 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 may cost you many hours of frustration in the coming weeks. By the end of this tutorial, you should be an expert in at least one mechanism on how to avoid race conditions.

Multi-Threads in 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, you should at least understand that you can create and run a thread through the following basic steps:

1. Implement the `Runnable` interface and define its `run()` method.
2. Implement your programming logic 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.

Beware the "final" Keyword

You may note that the input parameters contain the `final` keyword in some methods when you read some java programs. 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).

Synchronization

Coordinating multiple threads in a Java program to ensure thread-safety can be done using a **synchronized method** or **synchronized blocks** that is achieved through locks.

In Java, every object has a unique internal lock. When a non-static method is declared as `synchronized`, or a code snippet is enclosed by `synchronized(this)` block, the method or the code snippet will be protected by the internal lock of `this` instance. When any thread wants to enter the method or the code snippet, it must first try to acquire the internal lock. This ensures that only one instance of a method can be executed at any given point in time. Other methods can invoke the method or enter the code snippet, however they have to wait until the running thread releases the lock by exiting from the protected area or call wait() on the lock.

You can also use the lock of other objects to protect the synchronized code block. The lock object passed into a synchronized block will protect the code block.

For example, in the code below, when you synchronize on the `lock` object, you are using the internal lock of the object to protect the enclosed code snippet.

```
// Good example! It's thread-safe
private Object lock = new Object();
public void synchronization() {
    synchronized (lock) {
        // Do something
    }
}
```

A static synchronized statement behaves differently and it is out of the scope of this primer.

Aside on wait, notify and notifyAll

Synchronized communication is necessary to ensure ordered access of shared objects. The wait(), notify() and notifyAll() are simple methods of synchronized communication in Java that are used to handle concurrency. The wait() threads causes a thread to sleep, while notify() awakens a random thread and notifyAll() awakens all threads waiting on a shared object. To learn more about these methods and synchronized communication, a good tutorial can be found here (http://www.java-samples.com/showtutorial.php?tutorialid=306).

Information

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. It's important that you understand each method by reading the documentation. The table below is simply a reference table.

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?

Best Practice and Common Pitfalls in Java

Thread-safety

Before we proceed, let's give a formal definition of thread-safety. Thread-safe code must ensure that **when multiple threads are accessing a shared object, no matter how these threads are scheduled or in what order they are executing, the shared object will always behave correctly without any external synchronization, and every thread will be able to see any changes happened on this object immediately.**

To summarize, in order to ensure your code is thread-safe, your code must ensure *Atomicity* and *Visibility*. Let's discuss each item:

1. **Atomicity.** Any operations that change the state of the shared object must be atomic, which means that the operation should either be executed completely or not executed at all. No intermediate state should be exposed. For example, when you call notSafeUpdate() method, you add 1 to the internal counter and change its internal state.

```
// Bad example
public class NonAtomicCounter {
    int count;

    public void notSafeUpdate() {
        count++;
    }
}
```

Is this counter thread-safe? **No, the reason is that "++" is not atomic.** In fact, it is composed of three java byte code instructions: read [count] from memory, add 1 to it, and then write it back to memory. If thread B accesses this variable while thread A has added 1 to it but did not write it back, then inconsistency has occurred.

As shown below, two modifications can be made to the above example to ensure atomicity. The `safeUpdate()` method above can be synchronized using the internal lock of the AtomicCounter instance as shown below in `safeUpdate1()` and `safeUpdate2()`.

```
// Good example!
// safeUpdate1() and safeUpdate2() are equivalent.
public class AtomicCounter {
    int count;

    public synchronized void safeUpdate1() {
        count++;
    }

    public void safeUpdate2() {
        synchronized (this) {
            count++;
        }
    }
}
```

2. **Visibility.** The changes made by one thread should be immediately visible by any other thread. However, modern computers are equipped with multi-level caches, which is the root cause of invisibility in most situations. Incomplete or even corrupted data may be observed if invisibility occurs.

```
// Bad example
public class InvisibleCounter {
    int count;

    public int getCount() {
        return count;
    }

    public synchronized void safeUpdate() {
        count++;
    }
}
```

Although the write operation is correctly synchronized, this object is still thread-unsafe, since the read operation is not synchronized. Other threads can still access this variable from cache when one thread is modifying it. The change made by the writer thread is invisible to readers. *Hence, you need to always synchronize both read and write operations when you want to ensure thread-safety.*

The `getCount()` method can be synchronized using the internal lock of the AtomicCounter instance as shown below to ensure visibility and thread-safety.

```
// Good example: synchronize both read and write
public class VisibleCounter {
    int count;

    public synchronized int safeGetCount() {
        return count;
    }

    public synchronized void safeUpdate() {
        count ++;
    }
}
```

Read-write lock

With the prevalence of replication in a distributed datastore, read-write lock is widely used in data centers to ensure consistency as well as high throughput. A read-write lock allows multiple threads to read the data concurrently, but only one thread can write the data at a time. The read and write operations are atomic, which means reading and writing are mutually exclusive.

When designing a read-write lock, you need to decide which priority policy to use. Here we introduce two basic policies, read-preferring and write-preferring. In read-preferring policy, an incoming reader thread can acquire the lock in two scenarios:

1. When no writer has the lock and at least one reader is holding the lock
2. When no other threads (both reader and writer) have obtained the lock

Writes are only allowed when no other threads (both reader and writer) have obtained the lock. Read-preferring policy may lead to write-starvation, meaning that a write operation could be deferred indefinitely when there are lots of readers. On the other hand, write-preferring policy prevents new reader threads from holding the lock if there's any writer thread waiting in the queue. Readers are unable to acquire the lock until all writer threads have finished their operations and released the locks.

The below code is an example of a write-preferring read-write lock.

```
public class ReadWriteLock {

    private int readers = 0; // Numbers of readers holding the lock
    private int writers = 0; // Number of writers holding the lock
    private int writeRequests = 0; // Number of write requests in the queue

    // Use synchronized method so that these functions will not be executed concurrently.
    // This is the reason why we don't need to use AtomicInteger above
    public synchronized void lockRead() throws InterruptedException {
        while (writers > 0 || writeRequests > 0) {
            // While there is any writer holding the lock or pending write requests in th
e queue
            wait();
        }
        readers++;

    }

    public synchronized void unlockRead() {
        readers--;
        notifyAll(); // It will notify all waiting threads to check and see if they can r
un
    }

    public synchronized void lockWrite() throws InterruptedException {
        writeRequests++;

        while(readers > 0 || writers > 0) {
            wait();
        }

        writeRequests--;
        writers++;
    }

    public synchronized void unlockWrite() throws InterruptedException {
        writers--;
        notifyAll();
    }
}
```

In the above implementation, read requests can only be processed when there's no writer holding the lock or waiting in the queue (line 10), while writer threads can obtain the lock as soon as the lock is available. Note that these four functions shown in the snippet are synchronized methods, meaning that only one thread can call a synchronized method at the same time. The other threads calling the same method will have to wait until the former thread finishes.

Thread-safety in Java Libraries

When programming in Java, you may want to know if one class is thread-safe or not. If it is thread-safe, then you do not need to synchronize yourself. If it is not thread-safe, you may need to explicitly perform synchronization externally, e.g. Use a lock to protect every method accessing the object. There are five levels thread-safety in Java library:

1. **Immutable.** Immutable objects cannot be changed or modified, so we can safely share it among threads and do not need to do any synchronization.

   Example: String, Integer, Long (they are different from int or long) and BigInteger.

2. **Unconditionally thread-safe.** These objects are mutable, but have implemented sufficient "internal synchronization", which means that locking has already been designed and implemented by Java standard library developers, and we do not need to synchronize by ourselves manually. Thus, we can safely use them.

   Example: **Random, ConcurrentHashMap, ConcurrentHashSet, BlockingQueue PriorityBlockingQueue, AtomicInteger….** Vector and HashTable are thread-safe, but they have been deprecated in the latest JDK version.

   ```
   // Good example! It's thread-safe
   ConcurrentHashSet<String> set = new ConcurrentHashSet<>();
   public void add(String str) {
       set.add(str);
   }
   ```

3. **Conditionally thread-safe.** These objects are mutable, but have implemented sufficient internal synchronization on most of the methods. However, some methods still need to explicitly perform external synchronization. There are few examples in Java Library.

   Example: Sets wrapped by Collections.synchronized(). The iterators are not thread-safe.

4. **Thread-unsafe.** These objects are mutable and implemented with no internal synchronization. In order to use them safely, please explicitly synchronize ANY method call to these objects.

   Example: Most of the Java Collections, such as HashMap, HashSet, ArrayList, LinkedList and StringBuilder.

   ```
   // Good example! It's thread-safe
   HashSet<String> set  = new HashSet<>();
   public synchronized void add(String str) {
       set.add(str);
   }
   ```

5. **Thread-hostile.** These objects are not thread-safe even if you have already used external synchronization on ANY method call. Fortunately, the Java library has only a few examples. Example: System.runFinalizerOnExit(), deprecated since Java 1.6

   Example: System.runFinalizerOnExit(), deprecated since Java 1.6

Common Mistakes

The following are the most common mistakes that programmers make when building multithreaded applications in Java:

1. **No external lock on method call series.** Although containsKey() and put() method calls are respectively thread-safe, it's not thread-safe to call them in series without further synchronization.

   For ease of understanding, consider a scenario when thread A finds that key is not contained, it will enter the condition block and may be stopped by another thread B. Thread B can put a value inside the map, and when A is switched back, it will unconditionally override the value put by B. Boom! This causes inconsistency to occur.

   ```
   // Bad example
   ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
   public void add(String key, Integer value) {
       if (!map.containsKey(key)) {
           map.put(key, value);
       }
   }
   ```

Two modifications will be provided here. Please read the document for the usage of putIfAbsent() (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html#putIfAbsent(K,%20V))

```
// Good example! It's thread-safe
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

public void add1(String key, Integer value) {
    synchronized (this) {
        if (!map.containsKey(key)) {
            map.put(key, value);
        }
    }
}

// This is the idiomatic approach
public void add2(String key, Integer value) {
    map.putIfAbsent(key, value);
}
```

2. **Only synchronizing on write operations but not read operations** (an example is provided above in the `Visibility` section above)
3. **Be careful when using notify()**

   Unless you know exactly what you are doing, We recommend that you use notifyAll() instead of notify(). notify() makes no guarantee on which thread it will wake. Most of the time, incorrect usage of notify() will lead to a deadlock, in which every thread is blocked and waiting to be notified, but no one can notify others. **Additionally, you should always call notify() or notifyall() inside a synchronized block or a synchronized method.**
4. **Wrong pattern when using wait().**

   A thread can wake up without being notified, interrupted, or without timing. This is also known as a spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and force the thread to continue to wait if the condition is not satisfied. **Never call wait() outside a loop.**

```
private Object lock = new Object();
public void conditionalRun() {
    synchronized (lock) {
        while (<condition is not true>) {
            lock.wait();
        }
    }

    // Do something
}
```

   See here for more examples on usage of wait() (https://www.tutorialspoint.com/java/lang/object_wait.htm) and notifyAll() (https://www.tutorialspoint.com/java/lang/object_notifyall.htm).
5. **Synchronize on a wrong lock.**

```
// Bad example
public void uselessSynchronization() {
    synchronized (new Object()) {
        // Do something
    }
}
```

The above synchronization is totally useless. In order to realize mutual exclusive access to any shared object, **the read and write operations should be protected under the same lock**. In the above example, every method invoker will synchronize on different objects, which, unfortunately, has no synchronization effect at all.

```
// Good example! It's thread-safe
private Object lock = new Object();
public void synchronization() {
    synchronized (lock) {
        // Do something
    }
}
```

A more common mistake is synchronizing on a threads' own lock. Remember our previous discussion on synchronization, when you passed "this" inside the synchronized block, the block will be protected by the internal lock of that thread instance. Since each instance has its own internal lock, the code block is not protected by a global lock and every thread can access it at any time. Thus, this kind of synchronization will fail to work.

```
// Bad example
public void uselessSync() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (this) {
                // Do something
            }
        }
    }).start();
}
```

Instead, you should always synchronize on the same lock before accessing a shared resource. In the below example, the lock object is a global variable defined outside the Thread class and thus can be used as a shared lock for all the instances of the Thread class.

```
// Good example! It's thread-safe
private Object lock = new Object();
public void sync() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (lock) {
                // Do something
            }
        }
    }).start();
}
```

6. **Over-synchronization.** Over-synchronization is considered as a bad practice. There are two reasons why you should not over synchronize:
    1. You will pay a performance penalty.
    2. Sometimes over-synchronization can lead to a deadlock.

   Typical over-synchronization is when you synchronize on a stack variable. Stack variables are enclosed inside the current thread, so they cannot be shared among different threads. Thus, we need not synchronize on the stack variables.

   In example below, the `map` object is declared inside the function and thus it is only accessible from the thread executing the function. There is no need to synchronize the `put` operation on this object.

```
// Bad example
private Object lock = new Object();
public void sync() {
    HashMap<String, Integer> map = new HashMap<>();
    synchronized (lock) {
        map.put("Over-Sync", 0);
    }
}
```

7. **If you don't know the thread-safety of any object or method call, please check the Java documentation. Don't make an assumption, it will lead to bad results.**

Summary

That's all! If you want to know more about multithreading, we recommend two great books: *Effective Java* and *Java Concurrency in Practice*.

May the force be with you.