Theory: Thread synchronization

1243 users solved this topic. Latest completion was about 5 hours ago.

40 minutes 0 / 5 problems solved

Skip this topic Start practicing

Working concurrently with shared data from multiple threads may cause unexpected or erroneous behavior. Fortunately, Java provides a mechanism to control the access of multiple threads to a shared resource of any type. The mechanism is known as **thread synchronization**.

§1. Important terms and concepts

Before we start using synchronization in our code, let's introduce terms and concepts we're going to use.

1) A **critical section** is a region of code that accesses shared resources and should not be executed by more than one thread at the same time. A shared resource may be a variable, file, input/output port, database or something else.

Let's consider an example. A class has a static field named `counter`:

```
1 public static long counter = 0;
```

Two threads increment the field (increase by 1) 10 000 000 times concurrently. The final value should be 20 000 000. But, as we've discussed in previous topics, the result often might turn out wrong, for example, 9 999 843.

This happens because sometimes a thread does not see changes of shared data performed by another thread, and sometimes a thread may see an intermediate value of the non-atomic operation. Those are visibility and atomicity problems we deal with while working with shared data.

This is why increasing value by multiple threads is a **critical section**. Of course, this example is very simple, a critical section may be way more complicated.

2) The **monitor** is a special mechanism to control concurrent access to an object. In Java, each object and class has an associated implicit monitor. A thread can acquire a monitor, then other threads cannot acquire this monitor at the same time. They will wait until the owner (the thread that acquired the monitor) releases it.

Thus, a thread can be locked by the **monitor** of an object and wait for its release. This mechanism allows programmers to protect **critical sections** from being accessed by multiple threads concurrently.

§2. The synchronized keyword

The "classic" and simplest way to protect code from being accessed by multiple threads concurrently is using the keyword **synchronized**.

It is used in two different forms:

- synchronized method (a static or an instance method)
- synchronized blocks or statements (inside a static or an instance method)

A synchronized method or block needs an object for locking threads. The monitor associated with this object controls concurrent access to the specified critical section. Only one thread can execute code in a synchronized block or method at the same time. Other threads are blocked until the thread inside the synchronized block or method exits it.

§3. Static synchronized methods

Current topic:

Thread synchronization

Topic depends on:

× Working with shared data and problems

Topic is required for:

Collections and thread-safety

Table of contents:

When we synchronize static methods using the **synchronized** keyword the monitor is the class itself. Only one thread can execute the body of a synchronized static method at the same time. This can be summarized as *"one thread per class"*.

Here is an example of a class with a single synchronized static method named `doSomething`.

```
class SomeClass {

    public static synchronized void doSomething() {

        String threadName = Thread.currentThread().getName();

System.out.println(String.format("%s entered the method", threadName));

System.out.println(String.format("%s leaves the method", threadName));
    }
}
```

The method `doSomething` is declared as synchronized. It can be invoked only from one thread at the same time. The method is synchronized on the class the static method belongs to (the monitor is `SomeClass`).

Let's call the method from two threads concurrently. The result will always similar to:

```
Thread-0 entered the method
Thread-0 leaves the method
Thread-1 entered the method
Thread-1 leaves the method
```

It's impossible for more than one thread to execute code inside the method.

§4. Instance synchronized methods

Instance methods are synchronized on the instance (object). The monitor is the current object (`this`) that owns the method. If we have two instances of a class, each instance has a monitor for synchronizing.

Only one thread can execute code in a synchronized instance method of a particular instance. But different threads can execute methods of different objects at the same time. This can be summarized as *"one thread per instance"*.

Here is an example of a class with a single synchronized instance method named `doSomething`. The class also has a constructor for distinguishing instances.

```
class SomeClass {

    private String name;

    public SomeClass(String name) {
        this.name = name;
    }

    public synchronized void doSomething() {



        String threadName = Thread.currentThread().getName();


    System.out.println(String.format("%s entered the method of %s", threadName
, name));


    System.out.println(String.format("%s leaves the method of %s", threadName,
 name));

    }

}
```

Let's create two instances of the class and three threads invoking `doSomething`. The first and second threads take the same instance of the class, and the third thread takes another one.

```
SomeClass instance1 = new SomeClass("instance-1");
SomeClass instance2 = new SomeClass("instance-2");

MyThread first = new MyThread(instance1);
MyThread second = new MyThread(instance1);
MyThread third = new MyThread(instance2);

first.start();
second.start();

third.start();
```

The result will look like this:

```
Thread-0 entered the method of instance-1
Thread-2 entered the method of instance-2
Thread-0 leaves the method of instance-1
Thread-1 entered the method of instance-1
Thread-2 leaves the method of instance-2
Thread-1 leaves the method of instance-1
```

As you can see, there are no threads executing the code in `doSomething` of the `instance-1` at the same time. Try running it many times.

§5. Synchronized blocks (statements)

Sometimes you need to synchronize only a part of a method. This is possible by using synchronized blocks (statements). They must specify an object for locking threads.

Here is a class with a static and an instance method. Both methods are unsynchronized but have synchronized parts inside.

```
class SomeClass {

    public static void staticMethod() {

        // unsynchronized code

        synchronized (SomeClass.class) { // synchronization on the class
            // synchronized code
        }

    }



    public void instanceMethod() {



        // unsynchronized code



        synchronized (this) { // synchronization on this instance

            // synchronized code

        }

    }

}
```

The block inside `staticMethod` is synchronized on the class that means only one thread can execute code in this block.

The block inside `instanceMethod` is synchronized on `this` instance that means only one thread can execute the block of the instance. But some other thread is able to execute the block of different instances at the same time.

Synchronized blocks may remind synchronized methods but allow programmers to synchronize only necessary parts of methods.

§6. Synchronization and visibility of changes

The *Java Language Specification* guarantees that changes performed by a thread are visible to other threads if they are synchronized on the same monitor. More precisely, if a thread has changed shared data (for example, a variable) inside a synchronized block or a method and released the monitor, other threads can see all changes after acquiring the same monitor.

§7. Example: a synchronized counter

Here is an example. It's a synchronized counter with two synchronized instance methods: `increment` and `getValue`.

```
class SynchronizedCounter {

    private int count = 0;

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

    public synchronized int getValue() {

        return count;

    }

}
```

When multiple threads invoke `increment` on the same instance, no problem arises because the **synchronized** keyword protects the shared field. Only one thread can change the field. Other threads will wait until the thread releases the monitor. All changes of the variable `count` are visible.

The method `getValue` doesn't modify the field. It only reads the current value. The method is synchronized so that the reading thread always reads the actual value; otherwise, there is no guarantee that the reading thread will see the `count` as it is after it's changed.

Here is a class called `Worker` that extends `Thread`. The class takes an instance of `SynchronizedCounter` and calls the method `increment` 10 000 000 times.

```
class Worker extends Thread {

    private final SynchronizedCounter counter;

    public Worker(SynchronizedCounter counter) {
        this.counter = counter;
    }

    @Override

    public void run() {

        for (int i = 0; i < 10_000_000; i++) {

            counter.increment();

        }

    }

}
```

The following code creates an instance of `SynchronizedCounter`, starts threads and prints the result.

```
SynchronizedCounter counter = new SynchronizedCounter();

Worker worker1 = new Worker(counter);
Worker worker2 = new Worker(counter);

worker1.start();
worker2.start();

worker1.join();

worker2.join();



System.out.println(counter.getValue()); // the result is 20_000_000
```

Sometimes, however, there's no need to synchronize methods that only read shared data (including getters):

- If we have a guarantee that the reading thread successfully returns from `join` on all writing threads when it reads a field. That's true about the code above and we can remove the synchronized keyword from the declaration of `getValue`.
- If a shared field is declared with the **volatile** keyword. In that case, we will always see the actual value of this field.

Be extra careful when you decide not to synchronize read methods.

§8. One monitor and multiple synchronized methods and blocks

Important: an object or a class that has only one monitor and only one thread can execute synchronized code on the same monitor.

It means that if a class has several synchronized instance methods and a thread invokes one of them, other threads cannot execute either of these methods on the same instance until the first thread releases the monitor of the instance.

Here is an example: a class with three instance methods. Two methods are synchronized and the third one has an internal synchronized block. Both methods and the block are synchronized on the monitor of `this` instance.

```
class SomeClass {

    public synchronized void method1() {
        // do something useful
    }

    public synchronized void method2() {
        // do something useful
    }



    public void method3() {

        synchronized (this) {

            // do something useful

        }

    }

}
```

If a thread invokes `method1` and executes statements inside the method, no other thread can execute statements inside `method2` or in the synchronized block in `method3` because `this` monitor is already acquired. The threads will

wait for the release of the monitor.

The same behavior is correct when a class monitor is used.

§9. Reentrant synchronization

A thread cannot acquire a lock owned by another thread. But a thread can acquire a lock that it already owns. This behavior is called **reentrant synchronization**.

Take a look at the following example:

```
class SomeClass {

    public static synchronized void method1() {

method2(); // legal invocation because a thread has acquired monitor of SomeClass
    }

    public static synchronized void method2() {
        // do something useful
    }

}
```

The code above is correct. When a thread is inside `method1` it can invoke `method2` because both methods are synchronized on the same object (`SomeClass`).

§10. Fine-grained synchronization

Sometimes a class has several fields that are never used together. It's possible to protect these fields by using the same monitor, but in this case, only one thread will be able to access one of these fields, despite their independence. To improve the concurrency rate it's possible to use an idiom with additional objects as monitors.

Here is an example: a class with two methods. The class stores the number of calls to each method in a special field.

```
class SomeClass {

    private int numberOfCallingMethod1 = 0;
    private int numberOfCallingMethod2 = 0;

    final Object lock1 = new Object(); // an object for locking
    final Object lock2 = new Object(); // another object for locking

    public void method1() {

        System.out.println("method1...");



        synchronized (lock1) {

            numberOfCallingMethod1++;

        }

    }



    public void method2() {

        System.out.println("method2...");



        synchronized (lock2) {

            numberOfCallingMethod2++;

        }

    }

}
```

As you can see, the class has two additional fields that are the locks for separating monitors for each critical section.

If we have an instance of the class, one thread may work inside the synchronized block of the first method and, at the same time, another thread may work inside the synchronized block of the second method.

§11. Synchronization and performance of programs

Remember, the code protected by the synchronization mechanism can be executed only by one thread at the same time. It reduces the parallelism and responsiveness of the program.

Do not synchronize all your code. Try to use synchronization only when it really is necessary. Determine small parts of the code to be synchronized. Sometimes it's better to use a synchronization block instead of synchronizing a whole method (if the method is complex).

Report a typo

171 users liked this theory. **3** didn't like it. **What about you?**

Start practicing

Comments (19) Hints (0) Useful links (0) Show discussion