Shared Objects, Interference, and Mutual Exclusion

SFWRENG 3BB4:

Software Design III — Concurrent System Design

Shared Objects, Interference, and Mutual Exclusion

Definition Interference occurs when different threads operating on the same data interleave in an unwanted (or unexpected) fashion.

Example The code is designed so that in each loop step `counter` increments its current value by `1`.

```
public void run()
{
    int current = 0;

    for ( int i = 0; i < n; i++ )
    {
        System.out.print( "c " );
        this.pause( 1 );

        current = counter.value( );
        counter.set( ++current );

        System.out.print( "d " );
        this.pause( 1 );

        current = counter.value( );
        counter.set( ++current );
    }
}
```

Problem:

Since counter is referenced from multiple threads, interference between threads may prevent this increment from happening as expected.

Important: Even simple statements can translate to multiple steps by the Java virtual machine.

Shared Objects, Interference, and Mutual Exclusion

```
class Main
{
    public static void main(final String[] args)
    {
        int i = 1;
        int j = 2;

        j = i++;
    }
}
```

```
class Main {
  Main();
    Code:
       0: aload_0
       1: invokespecial #1   // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_1
       1: istore_1
       2: iconst_2
       3: istore_2
       4: iload_1
       5: iinc          1, 1
       8: istore_2
       9: return
}
```

Shared Objects, Interference, and Mutual Exclusion

1

If the operation `i++` would be atomic you wouldn't have the chance to read the value from it. This is exactly what you want to do using `i++` (instead of using `++i`).

For example look at the following code:

```
public static void main(final String[] args) {
    int i = 0;
    System.out.println(i++);
}
```

In this case we expect the output to be: `0` (because we post increment, e.g. first read, then update)

This is one of the reasons the operation can't be atomic, because you need to read the value (and do something with it) and **then** update the value.

The other important reason is that doing something atomically *usually* takes more time because of locking. It would be silly to have all the operations on primitives take a little bit longer for the rare cases when people want to have atomic operations. That is why they've added `AtomicInteger` and other atomic classes to the language.

share improve this answer

answered Aug 7 '14 at 9:04

Roy van Rijn

622 • 5 • 14

http://stackoverflow.com/questions/25168062/why-is-i-not-atomic

Shared Objects, Interference, and Mutual Exclusion

Definition Interference occurs when methods running in different threads but operating on the same data interleave in an unwanted (or unexpected) fashion.

Example The code is designed so that in each loop step `counter` increments its current value by `1`.

```
public void run()
{
    int current = 0;

    for ( int i = 0; i < n; i++ )
    {
        System.out.print( "c " );
        this.pause( 1 );

        current = counter.value( );
        counter.set( ++current );

        System.out.print( "d " );
        this.pause( 1 );

        current = counter.value( );
        counter.set( ++current );
    }
}
```

The problem?

Since counter is referenced from multiple threads, interference between threads may prevent this increment from happening as expected.

Important: Even simple statements can translate to multiple steps by the Java virtual machine.

Interference bugs are extremely difficult to locate. The general solution is to give methods mutually exclusive access to shared objects.

Shared Objects, Interference, and Mutual Exclusion

Mutual Exclusion in Java

Java provides two basic mechanisms for guaranteeing mutual exclusion:

- Synchronized methods
- Synchronized statements.

Example

```
{public|private}? synchronized returnType methodName( parameters )
{
    // Code in this area is executed in
    // a mutually exclusive manner
}
```

```
synchronized ( object )
{
    // Code in this area is executed in
    // a mutually exclusive manner
}
```

- Java associates a lock with every object.
- The Java compiler inserts code to acquire the lock before executing the body of the synchronized method and code to release the lock before the method returns.
- Threads are blocked until the lock is released.

Shared Objects, Interference, and Mutual Exclusion

Reading Material

Chapter 4 of
Magee and Kramer Concurrency: State, Models, and Java Programming.