Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

Synchronization

Chapter Overview

- What happens when two entities want to use the same thing at the same time?

Synchronization is an issue that arises when multiple animacies share state. In Java, this means that there are multiple Threads directly or indirectly accessing some field of an object. These Threads may either be explicitly created or automatically generated as, e.g., the user interface Thread in `java.awt`.

When an object accesses state, it does so either to obtain or to set a value. If the access does not change the value, we call it a ***read***. An access that changes the value of some state is called a ***write***. If more than one thread can access a state, we call it ***shared state***. Shared state can lead to problems if there are multiple accesses of the state at the same time and at least one of those accesses is a write. To avoid these problems, we can prevent sharing, we can prevent writing, or we can use specialized mechanisms or protocols to minimize conflict.

An Example of Conflict

When I was in high school, we took a class trip to Washington D.C. While we were there, we had a class photograph taken on the Capitol steps. Since there were a lot of us, they used a panning camera. The photographer started off pointing to the left, then scanned across the class until he got to the rightmost edge. The entire process took a minute or two.

The interesting part came when the photograph was developed. One of my classmates appeared in the upper left-hand corner of the picture. He *also* appeared in the upper right-hand corner! Here's how he did it: He started in the left edge of the group. As soon as the camera had moved past him -- during the minute or so that the photographer was scanning the group -- this classmate ran from one end of the group to the other. By the time that the photographer got to the right edge of the group, he had reached that side and was standing among the students there.

This is a synchronization failure. The problem is that the scanning of the group took time. Between the time that the scan started and the time that the scan completed, the student was able to change his position, so that the camera recorded him in both places. This is called a read-write conflict: the camera "read" a value that was incorrect. (My classmate's new position had already been recorded. A similar problem would arise if he'd run the other way -- then neither position would be recorded.)

A second example arises when two writes conflict. Say that our bank account contains $1000. We go to deposit $100. The ATM (automated teller machine) reads our current balance -- $1000 -- and goes off to calculate the new balance. At the same time, the bank's computer goes to give us our periodic 1% interest. It, too, reads our current balance ($1000) and sets out to compute our new balance from this.

In the meantime, the ATM finishes computing our new balance and stores it in the central accounting ledgers -- $1100. Finally, the banks' computer calculates our balance after interest -- 101% of $1000 is $1010 -- and writes that value to the central ledger. Unfortunately, the result is that after deposit and interest, we have a balance of $1010, not $1110.

These failures occur because there are two things going on at once -- a student running and a camera photographing, or two processes computing new balances -- that interact in inappropriate ways.

Synchronization

Synchronization is required when ***two or more threads of control*** (animacies) ***access the same*** (piece of) ***state and that state changes***. Synchronization prevents one animacy from reading the state while the other might be changing it. In Java, it also ensures that each read is of an up-to-date version of the state.

Synchronization is only necessary when there can be a write to shared state.

`Java` synchronized

The primary means of ensuring mutual exclusion in Java is through `synchronized` methods.

methods

In Java, a method may be declared `synchronized`. In each object at most one `synchronized` method can run at any one time. We say that a `synchronized` method ***obtains a lock*** on its containing object before it can execute. Since there is only one lock for each object, this prevents any other `synchronized` method from running until this method completes: no other method can obtain a lock on the object until this method releases its lock. This one-animacy-running-at-a-time property is called ***mutual exclusion***.

Locking an object only prevents access to other methods or code blocks that also require a lock on the same object. Locking an object does not prevent other (non-synchronized) methods of the object from running, nor does it prevent other use of the object.

(blocks)

Java has a second form of synchronized execution. A special `synchronized` statement type can be used to provide mutual exclusion on its body. Unlike `synchronized` methods, the `synchronized` statement (sometimes called a `synchronized` block) must explicitly specify the object it locks. The syntax of this statement is:

```
synchronized ( objectReference ) {
    statements
}
```

Here, *`objectReference`* is some expression whose value is of an object type; the locked object is the expression's value. The *`objectReference`* expression should be one whose value does not change; otherwise, careless coding can easily lead to a failure of mutual exclusion.

What synchronization buys you

Consider the class photograph described above. If the photographer had had synchronization, he would have been able to tell us not to move -- and would have been able to enforce it -- until after the photograph was done. My classmate never could have appeared in the single picture twice. (Well, at least not without digital enhancement.)

The bank example is similar. Real ATMs lock the account during the transaction, so that the interest figuring process couldn't read the balance until the ATM was done. In this case, the two computations would not overlap and the correct final balance would be reached.

Safety Rules

Sometimes, a set of data is interdependent. For example, we might have two fields corresponding to a street address and a zip code. Changing an address might involve changing both of these fields. If the zip code is changed without a corresponding change to the street address, the data may be inconsistent or incoherent. Such a set of operations, which must be done as a unit -- i.e., either all of the operations are executed or none are -- in order to ensure consistency of the data, is called a ***transaction***. The property of "doing all or none" is called ***atomicity***. A system in which all transactions are atomic is ***transaction-safe***.

The following rule suffices to ensure that your system is transaction-safe:

> ***All (potentially changeable) shared data is accessed only through the synchronized methods of a single object; no interdependent piece can be accessed independently.***

Note that this means that shared data cannot be returned by these methods for access by other methods. If shared data is to be returned, a (non-shared) copy must be made. Further, if interdependent values are to be returned (i.e., a portion of the shared data is to be used by other methods), all of the relevant values must be returned in a single transaction.

For example, the address and zip code of the previous example should not be returned by two separate method calls if they are to be assumed consistent.

```
public class AddressData {

     private String streetAddress;
     private String zipCode;
     public AddressData( String streetAddress, String zipCode) {
         this.setAddress( streetAddress, zipCode );
         ....
     }
     public synchronized void setAddress( String streetAddress,
                                          String zipCode) {
         // validity checks
         ....
         // set fields
         ....
     }
     public synchronized String getStreetAddress() { // problematic!
         return this.streetAddress;
     }
     public synchronized String getZipCode() { // problematic!
         return this.zipCode;
     }

}
```

If this class definition were used, e.g. for

```
     printMailingLabel( address.getStreetAddress(),
                        address.getZipCode() );
```

it would in principle be possible to get an inconsistent address. For example, between the calls to `address.getStreetAddress()` and `address.getZipCode()`, it is possible that a call to `address.setAddress` could occur. In this case, `getStreetAddress` would return the old street address, while `getZipCode()` would return the new zip code.

Instead, `getStreetAddress()` and `getZipCode()` should be replaced by a single `synchronized` method which returns a copy of the fields of the `AddressData` object:

```
     public synchronized SimpleAddressData getAddress() {
```

```
            return new SimpleAddressData( this.streetAddress,
                                          this.zipCode );
        }
```

The `SimpleAddressData` class can contain just `public` `streetAddress` and `zipCode` fields, without accessors. It is being used solely to return the two objects at the same time.

Deadlock

If you are not careful, it is not too difficult to get into a situation where multiple active objects each prevent the other from running.

Consider two objects which each need to control both the chalk and the eraser in order to write on the blackboard. The first uses the following algorithm:

1. Wait until the chalk is available, then pick it up.
2. Wait until the eraser is available, then pick it up.
3. Write (and erase).
4. Release the eraser.
5. Release the chalk.

The second uses the following algorithm:

1. Wait until the eraser is available, then pick it up.
2. Wait until the chalk is available, then pick it up.
3. Write (and erase).
4. Release the chalk.
5. Release the eraser.

If the two processes time things just right, it could be the case that they each complete their first steps before reaching their second. Now, the first process will be stuck waiting for the eraser (which the second process has), while the second will be stuck waiting for the chalk (which the first has). This situation -- in which neither process can do anything, and both are stuck waiting -- is called **deadlock**. (The processes in this case are effectively **dead**.)

There is an analogous situation that arises when both processes put down the objects they have and pick up the other object (repeatedly). In this situation, although both processes are still alive, neither is making any progress. This is called **livelock**.

The desirable property of a system that doesn't reach deadlock is **liveness**. In general, there is a tradeoff between safety and liveness, and a significant part of programming concurrent applications is designing to simultaneously maximize both.

Obscure details

This section is not for the faint of heart. While it is true, it is not pretty. Feel free to skip it.

Synchronization and local copies of state

In Java, each `Thread` may keep its own copy of shared state. This means that one copy may be inconsistent with another. Using `synchronized` forces a `Thread` to refresh all of its shared state, ensuring that it does not have a stale copy. Thus, even if timing constraints guarantee that only one `Thread` can access the state at a time, it may still be necessary to use `synchronized`. However, in this case the identity of the locked object is irrelevant; any `synchronized` method or block will do. (An alternate solution to this problem, though not to synchronization in general, is the `volatile` keyword

on fields.)

Synchronized blocks and lock object references

It is the value returned by the expression (at the time that the lock is obtained), and not the expression itself, that is locked. For example, given the following class definition

```
class SynchronizationFailure {

     Object foo = new Object();
     void failToSynchronize() {
         synchronized (foo) {
             foo = new Object();
             other statements
         }
     }

}
```

the `synchronized` block does not provide proper mutual exclusion. Consider a particular `SynchronizationFailure` instance, `popularObject`. If `Jack` and `Jill` both call `popularObject.failToSynchronize()` with appropriate timing, here is what could happen:

1. `Jack`'s call to `failToSynchronize` obtains a lock on `popularObject.foo`'s current value, say object 1.
2. When the line `foo = new Object();` is executed, `popularObject.foo` is assigned a new value, object 2.
3. `Jack`'s call continues to execute *`other statements`*.
4. In the meantime, `Jill` calls `popularObject.failToSynchronize()`. When `Jill`'s call reaches the `synchronized` block, it attempts to obtain a lock on `popularObject.foo`'s *current* value, object *2*. Although `Jack's` call is still inside the `synchronized` block, `Jill`'s call is able to enter because it attempts to lock a *different* object from `Jack`'s call.

Note that this failure can arise any time the value of the *`objectReference`* expression can change, even when it does not change inside the `synchronized` block. To avoid such failures, the synchronization expression (i.e., the *`objectReference`* on which the lock is obtained) should generally be an expression whose value does not change.

Chapter Summary

- Conflict can arise when multiple animacies access mutable state. For example, an entity may read an impossible value.
- This kind of conflict can be prevented by limiting state access to single animacy or by making all shared state immutable.
- When shared mutable state is desired, access can be controlled through Java's synchronization mechanisms.
    - Each object has its own "lock".
    - At most one animacy can hold this lock at any time.
    - A method may be declared `synchronized`. An animacy cannot execute a synchronized method until it holds the lock of the object to which the method belongs.
    - A block may be declared `synchronized` *on a particular object*. An animacy cannot execute a synchronized method until it holds the lock of the specified object.
- A transaction is a group of operations which must either be completely executed or not executed at all. Partial execution is not legal. A system is transaction-safe if all of its transactions are

executed atomically, i.e., partial execution is not possible.
- In general, increasing (transaction-)safety means decreasing liveness, a program's ability to run towards completion.
    - Transactions that interfere with one another so that all execution stops are called deadlocked.
    - Sometimes transactions interfere so that execution continues, but no progress can be made towards completion. This is called livelock.

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>