Java Concurrency

Anatomy of a Synchronizer

Jakob Jenkov
Last update: 2014-10-01

Even if many synchronizers (locks, semaphores, blocking queue etc.) are different in function, they are often not that different in their internal design. In other words, they consist of the same (or similar) basic parts internally. Knowing these basic parts can be a great help when designing synchronizers. It is these parts this text looks closer at.

Note: The content of this text is a part result of a M.Sc. student project at the IT University of Copenhagen in the spring 2004 by Jakob Jenkov, Toke Johansen and Lars Bjørn. During this project we asked Doug Lea if he knew of similar work. Interestingly he had come up with similar conclusions independently of this project during the development of the Java 5 concurrency utilities. Doug Lea's work, I believe, is described in the book **"Java Concurrency in Practice"**. This book also contains a chapter with the title "Anatomy of a Synchronizer" with content similar to this text, though not exactly the same.

The purpose of most (if not all) synchronizers is to guard some area of the code (critical section) from concurrent access by threads. To do this the following parts are often needed in a synchronizer:

1. **State**
2. **Access Condition**

Not all synchronizers have all of these parts, and those that have may not have them exactly as they are described here. Usually you can find one or more of these parts, though.

State

The state of a synchronizer is used by the access condition to determine if a thread can be granted access. In a **Lock** the state is kept in a `boolean` saying whether the `Lock` is locked or not. In a **Bounded Semaphore** the internal state is kept in a counter (int) and an upper bound (int) which state the current number of "takes" and the maximum number of "takes". In a **Blocking Queue** the state is kept in the `List` of elements in the queue and the maximum queue size (int) member (if any).

Here are two code snippets from both `Lock` and a `BoundedSemaphore`. The state code is marked in bold.

```
public class Lock{

  //state is kept here
  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  ...
}
```

```
public class BoundedSemaphore {

  //state is kept here
      private int signals = 0;
      private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    this.signal++;
    this.notify();
  }
  ...
}
```

Access Condition

The access conditions is what determines if a thread calling a test-and-set-state method can

In a **Lock** the access condition simply checks the value of the `isLocked` member variable. In a **Bounded Semaphore** there are actually two access conditions depending on whether you are trying to "take" or "release" the semaphore. If a thread tries to take the semaphore the `signals` variable is checked against the upper bound. If a thread tries to release the semaphore the `signals` variable is checked against 0.

Here are two code snippets of a `Lock` and a `BoundedSemaphore` with the access condition marked in bold. Notice how the conditions is always checked inside a while loop.

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    //access condition
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  ...
}
```

```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    //access condition
    while(this.signals == bound) wait();
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    //access condition
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
  }
}
```

State Changes

Once a thread gains access to the critical section it has to change the state of the synchronizer to (possibly) block other threads from entering it. In other words, the state needs to reflect the fact that a thread is now executing inside the critical section. This should affect the access conditions of other threads attempting to gain access.

Here are two code snippets with the state change code marked in bold:

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    //state change
    isLocked = true;
  }

  public synchronized void unlock(){
    //state change
    isLocked = false;
    notify();
  }
}
```

```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    //state change
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    //state change
    this.signals--;
    this.notify();
  }
}
```

Notification Strategy

Once a thread has changed the state of a synchronizer it may sometimes need to notify other waiting threads about the state change. Perhaps this state change might turn the access condition true for other threads.

Notification Strategies typically fall into three categories.

1. Notify all waiting threads.
2. Notify 1 random of N waiting threads

Once a thread want to notify the waiting threads it calls `notifyAll()` on the object the waiting threads called `wait()` on.

Notifying a single random waiting thread is also pretty easy. Just have the notifying thread call `notify()` on the object the waiting threads have called `wait()` on. Calling `notify` makes no guarantee about which of the waiting threads will be notified. Hence the term "random waiting thread".

Sometimes you may need to notify a specific rather than a random waiting thread. For instance if you need to guarantee that waiting threads are notified in a specific order, be it the order they called the synchronizer in, or some prioritized order. To achive this each waiting thread must call `wait()` on its own, separate object. When the notifying thread wants to notify a specific waiting thread it will call `notify()` on the object this specific thread has called `wait()` on. An example of this can be found in the text **Starvation and Fairness**.

Below is a code snippet with the notification strategy (notify 1 random waiting thread) marked in bold:

```
public class Lock{

  private boolean isLocked = false;

  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      //wait strategy - related to notification strategy
      wait();
    }
    isLocked = true;
  }

  public synchronized void unlock(){
    isLocked = false;
    notify(); //notification strategy
  }
}
```

Test and Set Method

Synchronizer most often have two types of methods of which `test-and-set` is the first type (`set` is the other). Test-and-set means that the thread calling this method **tests** the internal state of the synchronizer against the access condition. If the condition is met the thread **sets** the internal state of the synchronizer to reflect that the thread has gained access.

The state transition usually results in the access condition turning false for other threads trying to gain access, but may not always do so. For instance, in a **Read - Write Lock** a thread gaining read access will update the state of the read-write lock to reflect this, but other threads requesting read access will also be granted access as long as no threads has requested write access.

It is imperative that the test-and-set operations are executed atomically meaning no other threads are allowed to execute in the test-and-set method in between the test and the setting of the state.

1. Set state before test if necessary
2. Test state against access condition
3. If access condition is not met, wait
4. If access condition is met, set state, and notify waiting threads if necessary

The `lockWrite()` method of a **ReadWriteLock** class shown below is an example of a test-and-set method. Threads calling `lockWrite()` first sets the state before the test (`writeRequests++`). Then it tests the internal state against the access condition in the `canGrantWriteAccess()` method. If the test succeeds the internal state is set again before the method is exited. Notice that this method does not notify waiting threads.

```
public class ReadWriteLock{
    private Map<Thread, Integer> readingThreads =
        new HashMap<Thread, Integer>();

    private int writeAccesses    = 0;
    private int writeRequests    = 0;
    private Thread writingThread = null;

    ...


        public synchronized void lockWrite() throws InterruptedException{
        writeRequests++;
        Thread callingThread = Thread.currentThread();
        while(! canGrantWriteAccess(callingThread)){
        wait();
        }
        writeRequests--;
        writeAccesses++;
        writingThread = callingThread;
        }


    ...
}
```

The `BoundedSemaphore` class shown below has two test-and-set methods: `take()` and `release()`. Both methods test and sets the internal state.

```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }


      public synchronized void take() throws InterruptedException{
      while(this.signals == bound) wait();
      this.signals++;
      this.notify();
      }

      public synchronized void release() throws InterruptedException{
      while(this.signals == 0) wait();
```

```
}
```

Set Method

The set method is the second type of method that synchronizers often contain. The set method just sets the internal state of the synchronizer without testing it first. A typical example of a set method is the `unlock()` method of a `Lock` class. A thread holding the lock can always unlock it without having to test if the `Lock` is unlocked.

The program flow of a set method is usually along the lines of:

1. Set internal state
2. Notify waiting threads

Here is an example `unlock()` method:

```
public class Lock{

  private boolean isLocked = false;

      public synchronized void unlock(){
      isLocked = false;
      notify();
      }

}
```

Next: Non-blocking Algorithms


Project Polymorph
Introduction


7 + 2 Things I Love
About Java


False Sharing
in Java
Jakob Jenkov
JENKOV.COM


Java Concurrency +
Cache Coherence
Jakob Jenkov
JENKOV.COM


Compare and Swap
in Java
Jakob Jenkov
JENKOV.COM


Producer Consumer Pattern
in Java
Jakob Jenkov
JENKOV.COM


Java Concurrency
and Multithreading
Jakob Jenkov