

3.6 Semaphores and Locks in Java

The Java language does not provide a semaphore construct, but Java's built-in synchronization constructs can be used to simulate counting and binary semaphores.

J2SE 5.0 (Java 2 Platform, Standard Edition 5.0) introduces package *java.util.concurrent*, which contains a collection of synchronization classes, including classes *Semaphore* and *ReentrantLock*.

Below is an abstract class named *semaphore*:

```
public abstract class semaphore {
    protected abstract void P();
    protected abstract void V();
    protected semaphore(int initialPermits) {permits = initialPermits;}
    protected int permits;
}
```

3.6.1 Class *countingSemaphore*

The implementations of methods *P()* and *V()* in class *countingSemaphore* are based on Implementation 2 in Listing 3.3, which is reproduced below:

```
// Implementation 2 uses a queue of blocked threads. The value of permits may be // negative.
P(): permits = permits - 1;
    if (permits < 0) wait on a queue of blocked threads
V(): permits = permits + 1;
    if (permits <= 0)    notify one waiting thread;
```

To complete this implementation:

- provide mutual exclusion for accessing shared variable *permits*
- provide implementations for the wait and notify operations.

Each Java object is associated with a built-in lock.

If a thread calls a method on an object, and the method is declared with the synchronized modifier, then the calling thread must wait until it acquires the object's lock.

```
public synchronized void P() {...};  
public synchronized void V() {...};
```

Only one thread at a time can execute in the synchronized methods of an object.

If an object's data members are only accessed in synchronized methods, then the thread that owns the object's lock has exclusive access to the object's data members.

We can use Java's wait operation to complete the implementation of *P()*.

A thread must hold an object's lock before it can execute a wait operation. (The use of synchronized ensures this restriction is satisfied.)

- When a thread executes wait, it releases the object's lock and waits in a "wait set" that is associated with the object.
- The thread waits until it is notified or interrupted. (A thread *T* is interrupted when another thread calls *T.interrupt()*.)

Here is the implementation of method *P()*:

```
synchronized public void P() {  
    permit--;  
    if (permit < 0)  
        try { wait(); }           // same as this.wait();  
        catch (InterruptedException e) { }  
}
```

If a thread is interrupted before it is notified, the thread returns from wait by throwing *InterruptedException*. This is why wait usually appears in a try-catch block.

Since none of our programs use interrupts, we have elected to catch *InterruptedException* with an empty catch block instead of adding the clause “*throws InterruptedException*” to the header of method *P()*.

A notify operation notifies one of the waiting threads, but not necessarily the one that has been waiting the longest or the one with the highest priority.

If no threads are waiting, a notify operation does nothing. Here is method *V()* using Java’s notify operation:

```
synchronized public void V() {  
    ++permits;  
    if (permits <= 0)  
        notify();    // same as this.notify()  
}
```

A notified thread must reacquire the lock before it can begin executing in the method.

- notified threads that are trying to reacquire an object’s lock compete with any threads that have called a method of the object and are trying to acquire the lock for the first time.
- the order in which these notified and calling threads obtain the lock is unpredictable.

The complete implementation of class *countingSemaphore* is shown in Listing 3.15. Notice that this implementation does not allow threads that call *P()* to barge ahead of waiting threads and “steal” permits:

- When a waiting thread is notified, it may compete with other threads for the object’s lock,
- but when it eventually reacquires the lock, it will not execute wait again.
- Instead, it will start executing in *P()* after the wait operation, which means it will definitely be allowed to complete its *P()* operation

```
public final class countingSemaphore extends semaphore {
    public countingSemaphore(int initialPermits) {super(initialPermits);}
    synchronized public void P() {
        permits--;
        if (permits<0)
            try { wait(); } catch (InterruptedException e) {}
    }
    synchronized public void V() {
        ++permits;
        if (permits <=0)
            notify();
    }
}
```

Listing 3.15 Java class *countingSemaphore*.

This Java implementation of *countingSemaphore* does not guarantee a FCFS notification policy among waiting threads. That is, the order in which threads are notified is not necessarily the same order in which the threads waited.

Java implementations are permitted to perform “spurious wakeups”, i.e., wakeup threads without there having been any explicit Java instructions to do so. If spurious wakeups occur, class *countingSemaphore* may fail (since the awakened thread will not execute *wait()* again).

3.6.2 Class *mutexLock*

Listing 3.16 shows Java class *mutexlock*. Class *mutexlock* is more flexible than the built-in Java locks obtained through the synchronized keyword.

- A *mutexlock* object can be locked in one method and unlocked in another, and one method can contain multiple calls to *lock()* and *unlock()*.
- On the other hand, calls to *lock()* and *unlock()* are not automatically made on method entry and exit as they are in synchronized methods.

Member variable *owner* holds a reference to the current owner of the lock.

- This reference is obtained by calling *Thread.currentThread()*, which returns the thread that is currently executing method *lock()*.
- The value of member variable *free* is *true* if the lock is not currently owned.
- A thread that calls *lock()* becomes the owner if the lock is free; otherwise, the calling thread increments *waiting* and blocks itself by executing *wait()*.
- When a thread calls *unlock()*, if there are any waiting threads, *waiting* is decremented and one of the waiting threads is notified.
- The notified thread is guaranteed to become the new owner, since any threads that barge ahead of the notified thread will find that the lock is not free, and will block themselves on *wait()*.
- The owning thread must call *unlock()* the same number of times that it called *lock()* before another thread can become the owner of the lock.

```

public final class mutexLock {
    private Thread owner = null; // owner of this lock
    private int waiting = 0;     // number of threads waiting for this lock
    private int count = 0; // number of times mutexLock has been locked
    private boolean free = true; // true if lock is free; false otherwise
    public synchronized void lock() {
        if (free) {count = 1; free = false; owner = Thread.currentThread(); }
        else if (owner == Thread.currentThread()) {++count;}
        else {
            ++waiting;
            while (count > 0)
                try {wait();} catch (InterruptedException ex) {}
            count = 1; owner = Thread.currentThread();
        }
    }
    public synchronized void unlock() {
        if (owner != null) {
            if (owner == Thread.currentThread()) {
                --count;
                if (count == 0) {
                    owner = null;
                    if (waiting > 0) {
                        --waiting;
                        notify(); // free remains false
                    }
                    else {free = true;}
                    return;
                }
            }
            else {
                return;
            }
        }
    }
    throw new ownerException();
}
}

```

Listing 3.16 Java class *mutexLock*.

3.6.3 Class *Semaphore*

Package *java.util.concurrent* contains class *Semaphore*.

Class *Semaphore* is a counting semaphore with operations *acquire()* and *release()* instead of *P()* and *V()*.

The constructor for class *Semaphore* optionally accepts a fairness parameter:

- When this fairness parameter is false, class *Semaphore* makes no guarantees about the order in which threads acquire permits, as barging is permitted.
- When the fairness parameter is set to true, the semaphore guarantees that threads invoking *acquire()* obtain permits in FCFS order.

```
Semaphore mutex = new Semaphore(1,true);// FCFS semaphore initialized to 1
Thread1          Thread2
mutex.acquire(); mutex.acquire();
/* critical section */ /* critical section */
mutex.release();  mutex.release();
```

Class *Semaphore* also provides method *tryAcquire()*, which acquires a permit only if one is available at the time of invocation. Method *tryAcquire()* returns true if a permit was acquired and false otherwise.

```
if (mutex.tryAcquire()) { ... }
// does not honor the fairness setting - immediately acquire a permit if one is
// available, whether or not other threads are currently waiting.
```

A limit can be placed on the amount of time a thread will wait for a permit:

```
if ( mutex.tryAcquire(50L, TimeUnit.MILLISECONDS) ) { ... }
```

Acquires a permit if one becomes available within 50 milliseconds and the current thread has not been interrupted.

3.6.4 Class *ReentrantLock*

Package *java.util.concurrent.locks* provides a mutex lock class called *ReentrantLock* with methods *lock()* and *unlock()*.

Class *ReentrantLock* has the same behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements.

Like class *Semaphore*, the constructor for class *ReentrantLock* accepts an optional fairness parameter. When set to true, priority is given to the longest-waiting thread.

Class *ReentrantLock* also provides untimed and timed *tryLock()* methods that behave the same as the untimed and timed *tryAcquire()* methods of class *Semaphore*.

It is recommended to immediately follow a call to *lock()* with a try block:

```
class illustrateTryBlock {
    private final ReentrantLock mutex = new ReentrantLock();
    public void foo() {
        mutex.lock();
        try {
            /* body of method foo() */
        } finally { // ensures that unlock() is executed
            mutex.unlock()
        }
    }
}
```

3.6.5 Example: Java Bounded Buffer

Listing 3.17 is a Java solution to the bounded buffer problem for two producers and two consumers.

```
public final class boundedBuffer {
    public static void main (String args[]) {
        final int size = 3;    Buffer b = new Buffer(size);
        mutexLock mutexD = new mutexLock();           // exclusion for Producers
        mutexLock mutexW = new mutexLock();           // exclusion for Consumers
        countingSemaphore emptySlots = new countingSemaphore(size);
        countingSemaphore fullSlots = new countingSemaphore(0);
        Producer p1 = new Producer (b,emptySlots,fullSlots,mutexD,1);
        Producer p2 = new Producer (b,emptySlots,fullSlots,mutexD,2);
        Consumer c1 = new Consumer (b,emptySlots,fullSlots,mutexW,1);
        Consumer c2 = new Consumer (b,emptySlots,fullSlots,mutexW,2);
        p1.start(); c1.start(); p2.start(); c2.start();
    }
}

final class Producer extends Thread {
    private Buffer b = null;    private int num;
    private countingSemaphore emptySlots; private countingSemaphore fullSlots;
    private mutexLock mutexD;
    Producer (Buffer b, countingSemaphore emptySlots,
        countingSemaphore fullSlots, mutexLock mutexD, int num) {
        this.b = b;    this.num = num;    this.emptySlots = emptySlots;
        this.fullSlots = fullSlots; this.mutexD = mutexD;
    }
    public void run () {
        System.out.println ("Producer Running");
        for (int i = 0; i < 3; i++) {
            emptySlots.P();
            mutexD.lock();
            b.deposit(i); System.out.println("Producer # "+num+" deposited "+i);
            System.out.flush();
            mutexD.unlock();
            fullSlots.V();
        }
    }
}
```

```

final class Consumer extends Thread {
    private Buffer b;      private int num;
    private countingSemaphore emptySlots; private countingSemaphore fullSlots;
    private mutexLock mutexW;
    Consumer (Buffer b, countingSemaphore emptySlots,
             countingSemaphore fullSlots, mutexLock mutexW, int num) {
        this.b = b; this.num = num; this.emptySlots = emptySlots;
        this.fullSlots = fullSlots; this.mutexW = mutexW;
    }
    public void run () {
        System.out.println ("Consumer running");
        int value = 0;
        for (int i = 0; i < 3; i++) {
            fullSlots.P();
            mutexW.lock();
            value = b.withdraw();
            System.out.println ("Consumer # " + num + " withdrew " + value);
            System.out.flush();
            mutexW.unlock();
            emptySlots.V();
        }
    }
}

final class Buffer {
    private int[] buffer = null; private int in = 0, out = 0; private int capacity;
    public Buffer(int capacity) {
        this.capacity = capacity; buffer = new int[capacity];
    }
    public int withdraw () {
        int value = buffer[out];
        out = (out + 1) % capacity; // out is shared by consumers
        return value;
    }
    public void deposit (int value) {
        buffer[in] = value;
        in = (in + 1) % capacity; // in is shared by producers
    }
}

```

Listing 3.17 Java bounded buffer using semaphores and locks.