Explicit Locks

Alma Orucevic-Alagic

2013-11-28

Synchronized

- Java incorporates a cross-platform threading model & memory model into language specification.
- Thread class
- Synchronized & Volatile
- Atomicity & Visibility

```
synchronized (lockObject) {
    //update object state
}
```

- So why mess with a good thing?

Synchronized.. continued

- Can not Interrupt a Thread while Waiting to Acquire a Lock
- Might Need to Wait Forever to Acquire a Lock
- Must Release a Lock within Same Stack Frame where Acquired
- Lock Interface Provides More Extensive Locking Operations

Package java.util.concurrent.locks

- Framework offering greater flexibility for locking and conditions from built in synchronization and monitors.
- Interfaces: Condition, Lock, ReadWriteLock
- Classes:
  - AbstractOwnableSynchronizer
  - AbstractQueuedLongSynchronizer
  - AbstractQueuedSynchronizer
  - LockSupport
  - ReentrantLock
  - ReentrantReadWriteLock
  - ReentrantReadWriteLock.ReadLock
  - ReentrantReadWriteLock.WriteLock

Lock Interface

- Enables Access to Shared Resource by Multiple Threads
- Methods:
  - void lock(); - Acquires the lock
  - void lockInterruptibly(); - Acquires the lock unless the current thread is Interrupted.
  - Condition newCondition(); - Returns a new Condition that is bound by this Lock instance.
  - boolean tryLock(); - Acquires the lock only if it is free at the time of invocation.
  - boolean tryLock(long time, TimeUnit unit); - Returns true if the lock was acquired and false if waiting time expired before the lock was interrupted.
  - void unlock(); - Releases the Lock

Classes Implementing Lock Interface

- *(1)ReentrantLock, (2)ReentrantReadWriteLock.ReadLock, (3)ReentrantReadWriteLock.WriteLock*
- Reentrant Lock
  - Same behavior as the implicit monitor lock + some more
  - Lock owned by the thread with last successful locking and before unlocking

```
class X {
       private final ReentrantLock lock = new ReentrantLock();
       // ...

       public void m() {
         lock.lock();  // block until condition holds
         try {
           // ... method body
         } finally {
           lock.unlock();
         }
       }
     }
```

What happens if an exception is thrown?

Classes Implementing Lock Interface

ReentrantLock… continued

- Supports Fairness Policy – public ReentrantLock(boolean fair)
- Supports Interruptible Locks – void lockInterruptibly()
- Allows for Condition to be associated with this lock
- Provides Additional Methods for:
  - Queries:
    - Number of holds on this lock by the current thread
    - Whether current thread is waiting to acquire this lock
    - Whether any threads are waiting for the given condition associated with this lock
    - Whether lock is held by this thread
  - Returns a Collection of threads, the number of threads waiting for this lock (with or without the given Condition)

Classes Implementing Lock Interface

ReentrantLock… continued

- Factors out the Objects monitor methods (wait, notify, notifyAll) into distinct objects.
- BoundedBuffer


```
class BoundedBuffer {
      final Lock lock = new ReentrantLock();
      final Condition notFull  = lock.newCondition();
      final Condition notEmpty = lock.newCondition();

      final Object[] items = new Object[100];
      int putptr, takeptr, count;

      public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
          while (count == items.length)
            notFull.await();
          items[putptr] = x;
          if (++putptr == items.length) putptr = 0;
          ++count;
          notEmpty.signal();
        } finally {
          lock.unlock();
        }
      }
```

```
public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      Object x = items[takeptr];
      if (++takeptr == items.length) takeptr = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

Classes Implementing ReadWriteLock Interface

ReenterantReadWriteLock

- Supports Multiple Readers, but Only One Writer
- Implements ReadWriteLock Interface:
  - Lock readLock()
  - Lock writeLock()
- Encloses ReadWriteLock.ReadLock & ReadWriteLock.WriteLock classes that Implement Lock Interface
- Contains Similar Methods as ReentrantLock
- Condition Can Only Be Used with the Write Lock
- Writer can acquire a read lock, but not vice versa

Example 1: Avoid Lock Ordering Deadlock

- Transfer money from an account A to an account B
  - Using synchronized:

```
synchronized(fromAccount){
    synchronized(toAccount){
            //Money transfer logic
        }
}
```

Thread 1: Transfer from A to B
Thread 2: Transfer from B to A

  - Using locked:

```
while (!expired){
    if (fromAccount.lock.tryLock()){
        try {
            if(toAccount.lock.tryLock()){
                try {
                    //Money transfer logic...
                    return true;
                } finally {
                    toAccount.lock.unlock();
                }
            }
        }finally {
            fromAccount.lock.unlock();
        }

    if(expired)
        return false;
    //Sleep a little to reduce chance of live locks
    }
}
```

Expired: currentTime>=stopTime

Example 2: ReenterantReadWriteLock

- Try to Obtain Lock within Given Time Budget:

```
public boolean updateLDAPEntry(Account accountInfo, long time)
                                throws InterruptedException{

if (!lock.tryLock(time, TimeUnit.SECONDS))
    return false;
try{
    return updateLDAP(accountInfo);
}
  finally{
      lock.unlock();
  }
}
```

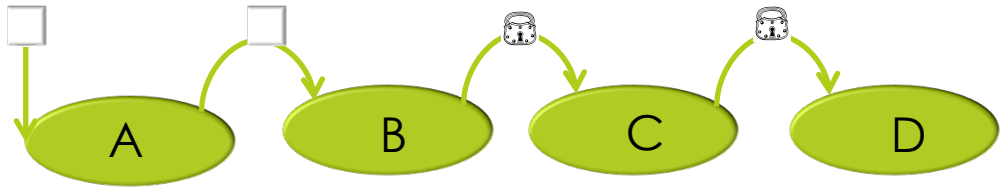
- Interruptible Locks

```
public boolean updateLDAPEntry(Account accountInfo)
          throws InterruptedException{
lock.lockInterruptibly();
try{
    return cancellableUpdateLDAP(accountInfo);
}
  finally{
      lock.unlock();
  }
}
```

```
private boolean cancellableUpdateLDAP(Account accountInfo)
                throws InterruptedException{
        //implement cancellable update
    return true; //or false
}
```

Hand-over-locking

- Intrinsic Locks Block Structured
- Reducing Lock Granularity can Enhance Scalability
- Lock interface Allows for Locks to be Acquired and Released in Different Scopes & Multiple Locks to be Acquired and Released



Performance Considerations

- Resources Expended on Lock Management & Scheduling
- Java 5.0 (Initial Locks framework released)

Figure 1. Throughput for synchronization and Lock, single CPU

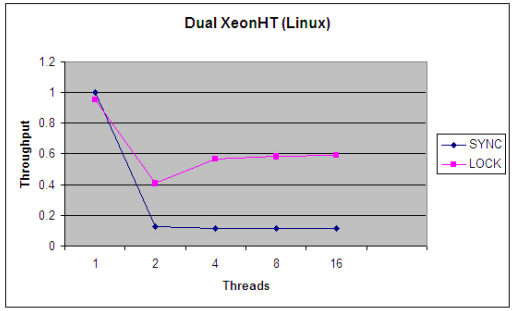
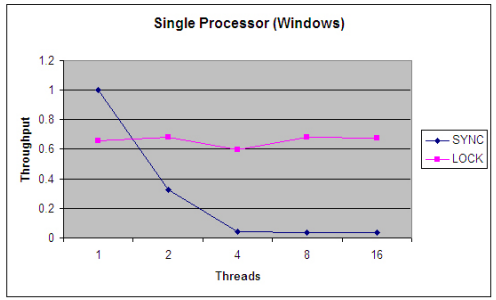
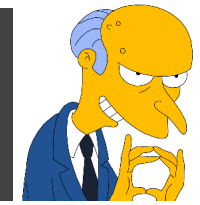


Figure 2. Throughput (normalized) for synchronization and Lock, four CPUs

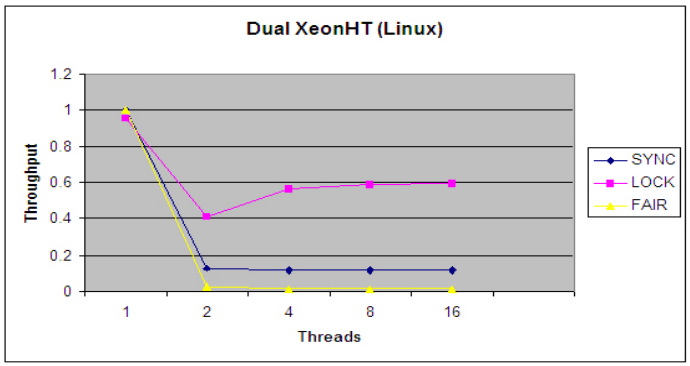


- Java 6 – Intrinsic and Explicit Scale Fairly Equally.
- Performance – Moving Target



Fairness

- Fair vs. NonFair Locks
- Performance cost!
- High Load Can Hinder Time of Thread Resuming Time vs. its Actual Run Time.
- Long Wait Times or Mean Time Between Lock Requests.
- Java 5:



Intrinsic (Synchronized) vs. Explicit Locks?

Feature	Intrinsic	Explicit
Timed Lock Wait	✗	✔
Interruptible Lock Wait	✗	✔
Fairness	✗	✔
Non-block structure locking	✗	✔
Familiar syntax, used extensively	✔	✗
Good idea to mix the two	N	O
More dangerous	✗	✔
Bright Future Awaiting ☺	✔	✗

According to Brian Goetz et al., Java Concurrency in Practice

Far, Far Away,
In the Galaxy of Java 10
the Anticipated Performance of
Intrinsic over Explicit Lock

Will Be: