Java ReentrantLock
Reentrant Mutex Semantics

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand the concept of mutual exclusion in concurrent programs
- Note a human-known use of mutual exclusion
- Recognize the structure & functionality of Java ReentrantLock
- Be aware of reentrant mutex semantics

![Diagram showing mutual exclusion states: unlocked (holdCount = 0), locked (holdCount = 1), and critical section.]
Learning Objectives in this Part of the Lesson

• Understand the concept of mutual exclusion in concurrent programs
• Note a human-known use of mutual exclusion
• Recognize the structure & functionality of Java ReentrantLock
• Be aware of reentrant mutex semantics
  • In the context of Java ReentrantLock & the Android CountDownTimer framework

Overview of Reentrant Mutex Semantics
Overview of Reentrant Mutex Semantics

- A ReentrantLock supports “reentrant mutex” semantics

See en.wikipedia.org/wiki/Reentrant_mutex
Overview of Reentrant Mutex Semantics

- A ReentrantLock supports “reentrant mutex” semantics
- The thread holding the mutex can reacquire it without self-deadlock
Overview of Reentrant Mutex Semantics

- A ReentrantLock supports “reentrant mutex” semantics
- The thread holding the mutex can reacquire it without self-deadlock
Overview of Reentrant Mutex Semantics

- Reentrant mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization.

```java
boolean nonfairTryAcquire(int acquires) {
    Thread t = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(t);
            return true;
        }
    } else if (t == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        ...
        setState(nextc);
        return true;
    }
    return false;
}
```

See [src/share/classes/java/util/concurrent/locks/ReentrantLock.java](src/share/classes/java/util/concurrent/locks/ReentrantLock.java)
Overview of Reentrant Mutex Semantics

- Reentrant mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization.

```java
boolean nonfairTryAcquire
    (int acquires) {
    Thread t =
        Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0,
                                acquires)) {
            setExclusiveOwnerThread(t);
            return true;
        }
    } else if (t ==
                getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        ... setState(nextc);
        return true;
    }
    return false;
}
```
• Reentrant mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization

```java
boolean nonfairTryAcquire(int acquires) {
    Thread t =
        Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0,
                        acquires)) {
            setExclusiveOwnerThread(t);
            return true;
        }
    } else if (t ==
        getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        ...  
        setState(nextc);
        return true;
    }
    return false;
}
```
Overview of Reentrant Mutex Semantics

- Reentrant mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization.

```java
boolean nonfairTryAcquire
    (int acquires) {
        Thread t =
            Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (compareAndSetState(0,
                    acquires)) {
                setExclusiveOwnerThread(t);
                return true;
            }
        } else if (t ==
            getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            ...
            setState(nextc);
            return true;
        }
        return false;
    }
```

Atomically acquire the lock if it's available
Overview of Reentrant Mutex Semantics

- Reentrant mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization.

```java
boolean nonfairTryAcquire(int acquires) {
    Thread t = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(t);
            return true;
        }
    } else if (t == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        ...setState(nextc);
        return true;
    }
    return false;
}
```

*Simply increment lock count if the current thread is lock owner*
Overview of Reentrant Mutex Semantics

- Reentrant mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization.

```java
boolean nonfairTryAcquire(int acquires) {
    Thread t = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(t);
            return true;
        }
    } else if (t == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        ...
        setState(nextc);
        return true;
    }
    return false;
}
```

Return false if the calling thread doesn’t own the lock.
Overview of Reentrant Mutex Semantics

- Reentrant mutex semantics are useful for frameworks that hold locks during callbacks to user code.

```java
mLock.lock();
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}
```

```java
if (...) {
    cancel();
}
```

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex24](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex24)
Overview of Reentrant Mutex Semantics

- Reentrant mutex semantics are useful for frameworks that hold locks during callbacks to user code.

```java
mLock.lock();
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}
```

Schedule a countdown until a time in the future, with regular notifications on intervals along the way via the `onTick()` hook method.

```java
if (...) {
    cancel();
}
```

See developer.android.com/reference/android/os/CountDownTimer
Overview of Reentrant Mutex Semantics

- Reentrant mutex semantics are useful for frameworks that hold locks during callbacks to user code.

```java
mLock.lock();
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}
```

Framework calls `onTick()` hook method in a background thread with the `mLock` held.

```java
mLock.lock();
try {
    ...
    onTick(millisLeft);
    ...
} finally {
    mLock.unlock();
}
```

```java
if (...) {
    cancel();
}
```
Overview of Reentrant Mutex Semantics

Reentrant mutex semantics are useful for frameworks that hold locks during callbacks to user code.

```java
mLock.lock();
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}
```

The app can override the `onTick()` hook method to conditionally call `cancel()`.

```java
if (...) {
    cancel();
}
```

The diagram illustrates the `cDTimer: CountDownTimer` class with methods such as `start()`, `onFinish()`, `cancel()`, `onTick()`, and `run()`. The `mLock` variable is used to acquire and release locks as needed.
Overview of Reentrant Mutex Semantics

- Reentrant mutex semantics are useful for frameworks that hold locks during callbacks to user code.

```java
mLock.lock();
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}

if (...) {
    cancel();
}
```

cancel() also acquires mLock, which must be recursive or self-deadlock will occur.
Overview of Reentrant Mutex Semantics

• Reentrant mutex semantics are useful for frameworks that hold locks during callbacks to user code

```
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}
```

```
if (...) {
    cancel();
}
```

```
mLock.lock();
try {
    ...
    onTick(millisLeft);
    ...
} finally {
    mLock.unlock();
}
```

*unlock() will be called multiple times to unwind the reentrant lock*
Overview of Reentrant Mutex Semantics

This example shows the difference between a reentrant lock (e.g., Java ReentrantLock) and a non-reentrant lock (e.g., Java StampedLock) when applied in a framework that allows callbacks where the framework holds a lock protecting internal framework state.

As you’ll see when you run this program, the reentrant lock supports this use-case nicely, whereas the non-reentrant lock incurs "self-deadlock."

```java
public class ex24 {
    /**
     * Used to wait for the test to finish running.
     */
    private static CountDownLatch sCd1;

    /**
     * Main entry point into the test program.
     */
    public static void main (String[] argv)
        throws IOException, InterruptedException {
```
End of Java ReentrantLock
Reentrant Mutex Semantics