Java Readers-Writer Locks:
Introduction

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

- Recognize the intent of readers-writer locks
Learning Objectives in this Part of the Lesson

• Recognize the intent of readers-writer locks

• Note a human known use of readers-writer locks
Overview of Readers-Writer Locks
Overview of Readers-Writer Locks

• Java synchronizers like ReentrantLock & synchronized methods/statements treat all waiting threads the same

See earlier lessons on “Java ReentrantLock” & “Java Monitor Objects”
Overview of Readers-Writer Locks

- Java synchronizers like ReentrantLock & synchronized methods/statements treat all waiting threads the same.
- This behavior is fine if an object’s methods perform a mix of reading & writing to its state.
Overview of Readers-Writer Locks

• In contrast, readers-writer locks are used to solve “readers-writer problems”

See en.wikipedia.org/wiki/Readers-writer_lock
Conversely, readers-writer locks are used to solve “readers-writer problems”, e.g. 

- Multiple threads at a time need read access
Overview of Readers-Writer Locks

- Conversely, readers-writer locks are used to solve “readers-writer problems”, e.g.
  - Multiple threads at a time need read access
  - Only one thread at a time can have exclusive write access
Overview of Readers-Writer Locks

• Readers-writer locks are most useful on multi-core systems where many threads need to access a shared resource at the same time.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html)
Overview of Readers-Writer Locks

- Readers-writer locks are most useful on multi-core systems where many threads need to access a shared resource at the same time, i.e.
- All threads that require read-only access can run concurrently
Overview of Readers-Writer Locks

- Readers-writer locks are most useful on multi-core systems where many threads need to access a shared resource at the same time.
  - All threads that require read-only access can run concurrently.
  - However, only one thread at a time can have write access.

---

• Readers-writer locks are most useful on multi-core systems where many threads need to access a shared resource at the same time.
  • All threads that require read-only access can run concurrently.
  • However, only one thread at a time can have write access.
Overview of Readers-Writer Locks

- Java supports two types of readers-writer locks
Overview of Readers-Writer Locks

Java supports two types of readers-writer locks

- ReentrantReadWriteLock

  This is the “classic” way of supporting readers-writer locks in Java

---

**Class ReentrantReadWriteLock**

```java
java.lang.Object
    java.util.concurrent.locks.ReentrantReadWriteLock
```

All Implemented Interfaces:
Serializable, ReadWriteLock

---

```java
public class ReentrantReadWriteLock
extends Object
implements ReadWriteLock, Serializable

An implementation of ReadWriteLock supporting similar semantics to ReentrantLock.

This class has the following properties:

- **Acquisition order**

  This class does not impose a reader or writer preference ordering for lock access. However, it does support an optional fairness policy.

  **Non-fair mode (default)**

  When constructed as non-fair (the default), the order of entry to the read and write lock is unspecified, subject to reentrancy constraints. A nonfair lock that is continuously contended may indefinitely postpone one or more reader or writer threads, but will normally have higher throughput than a fair lock.

  **Fair mode**

  When constructed as fair, threads contend for entry using an approximately arrival-order policy. When the currently held lock is released, either the longest-waiting single writer thread will be assigned the write lock, or if there is a group of reader threads waiting longer than all waiting writer threads, that group will be assigned the read lock.
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html)
Overview of Readers-Writer Locks

- Java supports two types of readers-writer locks
  - ReentrantReadWriteLock
    - This is the “classic” way of supporting readers-writer locks in Java

ReentrantReadWriteLock is “pessimistic”, i.e., it assumes contention may occur
Overview of Readers-Writer Locks

- Java supports two types of readers-writer locks
  - ReentrantReadWriteLock
  - StampedLock
  - This is the newer & better way of supporting readers-writer locks in Java 8+

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html
Overview of Readers-Writer Locks

- Java supports two types of readers-writer locks
  - ReentrantReadWriteLock
  - StampedLock
    - This is the newer & better way of supporting readers-writer locks in Java 8+

Class StampedLock

```java
import java.lang.Object;
import java.util.concurrent.locks.StampedLock;

interface Serializable;

public class StampedLock extends Object implements Serializable
```

A capability-based lock with three modes for controlling read/write access. The state of a StampedLock consists of a version and mode. Lock acquisition methods return a stamp that represents and controls access with respect to a lock state; "try" versions of these methods may instead return the special value zero to represent failure to acquire access. Lock release and conversion methods require stamps as arguments, and fail if they do not match the state of the lock. The three modes are:

- **Writing.** Method `writeLock()` possibly blocks waiting for exclusive access, returning a stamp that can be used in method `unlockWrite(long)` to release the lock. Untimed and timed versions of `tryWriteLock` are also provided. When the lock is held in write mode, no read locks may be obtained, and all optimistic read validations will fail.

- **Reading.** Method `readLock()` possibly blocks waiting for non-exclusive access, returning a stamp that can be used in method `unlockRead(long)` to release the lock. Untimed and timed versions of `tryReadLock` are also provided.

- **Optimistic Reading.** Method `tryOptimisticRead()` returns a non-zero stamp only if the lock is not currently held in write mode. Method `validate(long)` returns true if the lock has not been acquired in write mode since obtaining a given stamp. This mode can be thought of as an extremely weak version of a read-lock, that can be broken by a writer at any time. The use of optimistic mode for short read-only code segments often reduces contention and improves throughput. However, its use is inherently fragile. Optimistic read sections should only read fields and hold them in local variables for later use after validation. Fields read while in optimistic mode may be wildly inconsistent, so

**StampedLock is more “optimistic”, i.e., it can assume contention won’t occur**
Human Known Use of Readers-Writer Locks
Human Known Use of Readers-Writer Locks

A human known use of a readers-writer locking protocol is the process for periodic updates of material on websites that change infrequently.
Human Known Use of Readers-Writer Locks

• A human known use of a readers-writer locking protocol is the process for periodic updates of material on websites that change infrequently, e.g.

• The Vanderbilt undergraduate course catalog is accessed continuously, but only changes once a year.
A human known use of a readers-writer locking protocol is the process for periodic updates of material on websites that change infrequently, e.g.

- The Vanderbilt undergraduate course catalog is accessed continuously, but only changes once a year.
- Therefore, multiple readers can access the catalog concurrently.
A human known use of a readers-writer locking protocol is the process for periodic updates of material on websites that change infrequently, e.g.

- The Vanderbilt undergraduate course catalog is accessed continuously, but only changes once a year.
- Therefore, multiple readers can access the catalog concurrently.
- But only one writer can access the catalog when it’s updated annually.
End of Java Readers-Writer
Locks: Introduction