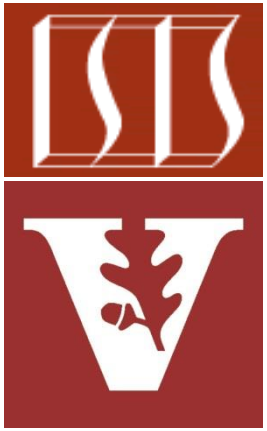
Java ReentrantReadWriteLock: Usage Considerations



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

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

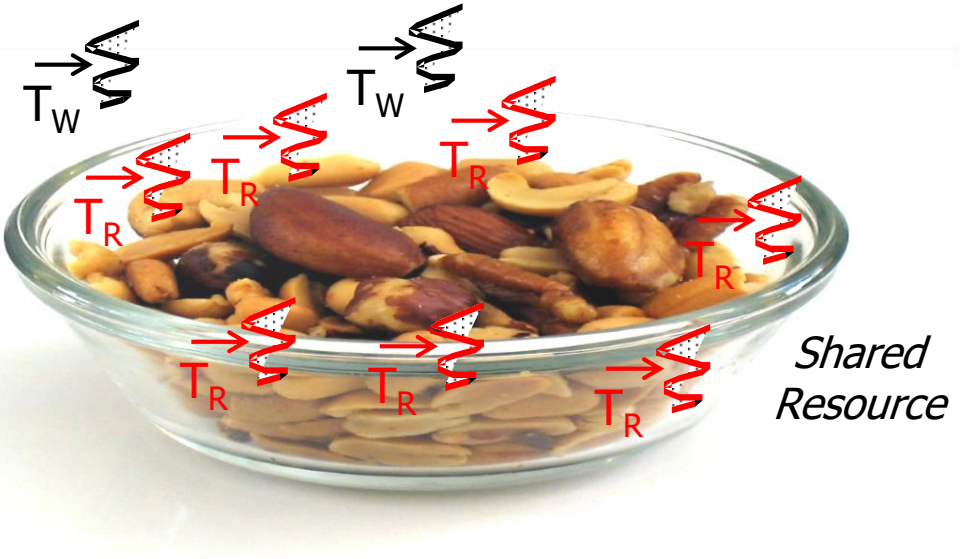
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of the Java ReentrantReadWriteLock class
- Know the key methods in Java ReentrantReadWriteLock
- Recognize how to apply Java ReentrantReadWriteLock in practice
- Appreciate Java ReentrantReadWrite Lock usage considerations

ReentrantReadWriteLock Usage Considerations

ReentrantReadWriteLock Usage Considerations

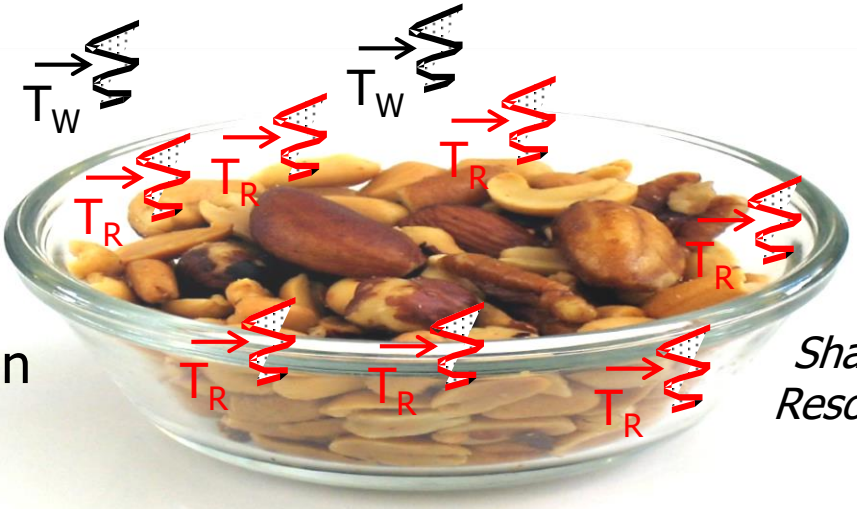
- ReentrantReadWriteLock enables higher levels of concurrency when accessing shared "read-only" data compared with a ReentrantLock

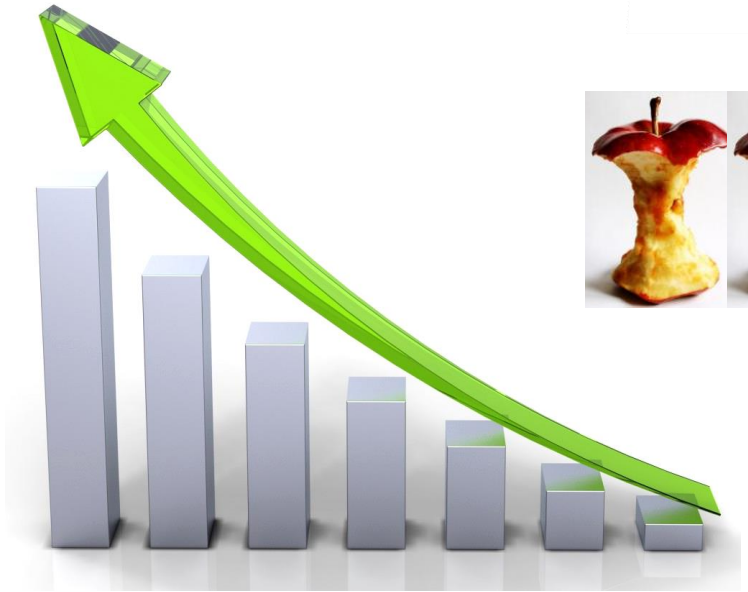


See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html

ReentrantReadWriteLock Usage Considerations

- ReentrantReadWriteLock enables higher levels of concurrency when accessing shared "read-only" data compared with a ReentrantLock
  - *May* improve performance if data are read from much more often than written to on multi-core systems





ReentrantReadWriteLock Usage Considerations

- However, ReentrantReadWriteLock has several limitations

See javaspecialists.eu/talks/jfokus13/PhaserAndStampedLock.pdf

ReentrantReadWriteLock Usage Considerations

- However, ReentrantReadWriteLock has several limitations
  - Both read & write locks are "pessimistic" & thus assume contention will always occur

ReentrantReadWriteLock Usage Considerations

- However, ReentrantReadWriteLock has several limitations
  - Both read & write locks are "pessimistic" & thus assume contention will always occur
    - In contrast, StampedLock has an "optimistic" read mode & generally performs better

See upcoming lesson on "*Java StampedLock*"

ReentrantReadWriteLock Usage Considerations

- However, ReentrantReadWriteLock has several limitations
  - Both read & write locks are "pessimistic"
  - Can starve readers or writers, depending on their priority

See en.wikipedia.org/wiki/Readers-writer_lock

ReentrantReadWriteLock Usage Considerations

- However, ReentrantReadWriteLock has several limitations
  - Both read & write locks are "pessimistic"
  - Can starve readers or writers, depending on their priority
    - Java 5 (readers priority) & 6+ (writers priority) semantics differ

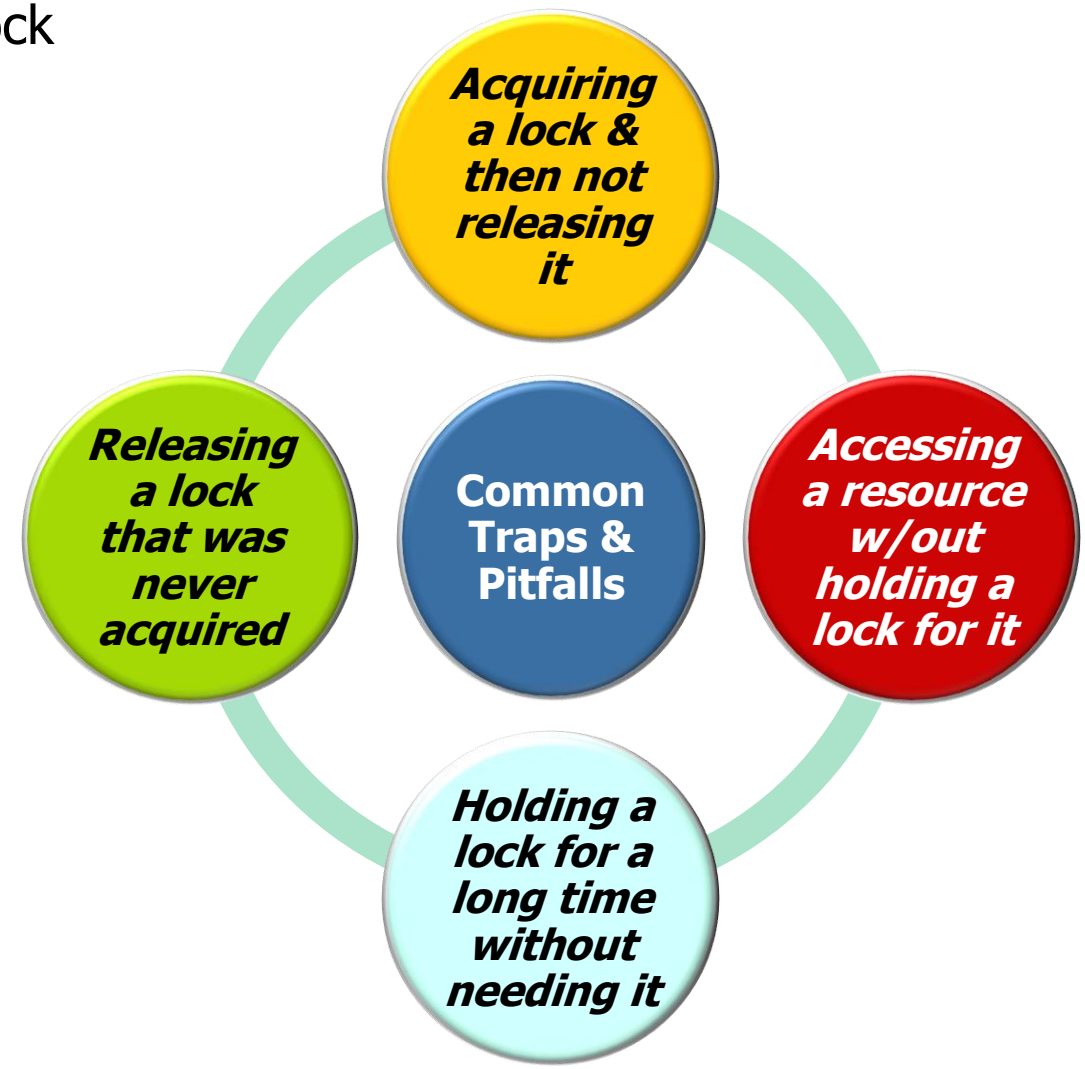
See www.javaspecialists.eu/archive/Issue165.html

ReentrantReadWriteLock Usage Considerations

- However, ReentrantReadWriteLock has several limitations
  - Both read & write locks are "pessimistic"
  - Can starve readers or writers, depending on their priority
  - Can be tedious & error-prone to program

ReentrantReadWriteLock Usage Considerations

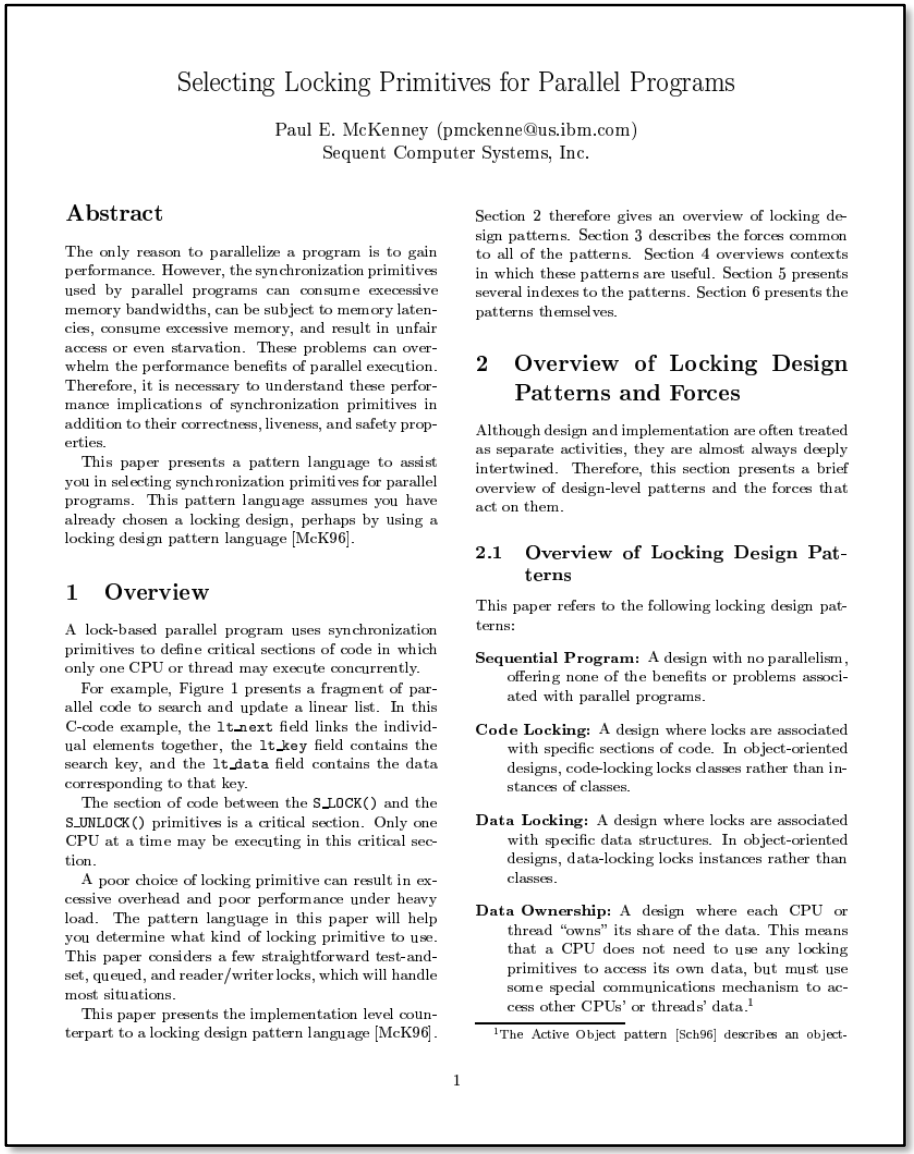
- However, ReentrantReadWriteLock has several limitations
  - Both read & write locks are "pessimistic"
  - Can starve readers or writers, depending on their priority
  - Can be tedious & error-prone to program



See earlier lessons on "*Java ReentrantLock*" & "*Java Semaphore*"

ReentrantReadWriteLock Usage Considerations

- Profiling is essential to see if a ReentrantReadWriteLock is suited for a particular use-case



Selecting Locking Primitives for Parallel Programs

Paul E. McKenney (pmckenne@us.ibm.com)
Sequent Computer Systems, Inc.

Abstract

The only reason to parallelize a program is to gain performance. However, the synchronization primitives used by parallel programs can consume execessive memory bandwidths, can be subject to memory latencies, consume excessive memory, and result in unfair access or even starvation. These problems can overwhelm the performance benefits of parallel execution. Therefore, it is necessary to understand these performance implications of synchronization primitives in addition to their correctness, liveness, and safety properties.

This paper presents a pattern language to assist you in selecting synchronization primitives for parallel programs. This pattern language assumes you have already chosen a locking design, perhaps by using a locking design pattern language [McK96].

1 Overview

A lock-based parallel program uses synchronization primitives to define critical sections of code in which only one CPU or thread may execute concurrently.

For example, Figure 1 presents a fragment of parallel code to search and update a linear list. In this C-code example, the `lt_next` field links the individual elements together, the `lt_key` field contains the search key, and the `lt_data` field contains the data corresponding to that key.

The section of code between the `S_LOCK()` and the `S_UNLOCK()` primitives is a critical section. Only one CPU at a time may be executing in this critical section.

A poor choice of locking primitive can result in excessive overhead and poor performance under heavy load. The pattern language in this paper will help you determine what kind of locking primitive to use. This paper considers a few straightforward test-and-set, queued, and reader/writer locks, which will handle most situations.

This paper presents the implementation level counterpart to a locking design pattern language [McK96]. Section 2 therefore gives an overview of locking design patterns. Section 3 describes the forces common to all of the patterns. Section 4 overviews contexts in which these patterns are useful. Section 5 presents several indexes to the patterns. Section 6 presents the patterns themselves.

2 Overview of Locking Design Patterns and Forces

Although design and implementation are often treated as separate activities, they are almost always deeply intertwined. Therefore, this section presents a brief overview of design-level patterns and the forces that act on them.

2.1 Overview of Locking Design Patterns

This paper refers to the following locking design patterns:

Sequential Program: A design with no parallelism, offering none of the benefits or problems associated with parallel programs.

Code Locking: A design where locks are associated with specific sections of code. In object-oriented designs, code-locking locks classes rather than instances of classes.

Data Locking: A design where locks are associated with specific data structures. In object-oriented designs, data-locking locks instances rather than classes.

Data Ownership: A design where each CPU or thread "owns" its share of the data. This means that a CPU does not need to use any locking primitives to access its own data, but must use some special communications mechanism to access other CPUs' or threads' data.[1]

[1]The Active Object pattern [Sch96] describes an object-

1

See www2.rdrop.com/~paulmck/scalability/paper/mutexprimpat.pdf

ReentrantReadWriteLock Usage Considerations

- Profiling is essential to see if a ReentrantReadWriteLock is suited for a particular use-case
  - ReentrantReadWriteLock's overhead is nearly always greater than any benefits it provides..

See www.takipiblog.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized

End of Java Reentrant ReadWriteLock: Usage Considerations