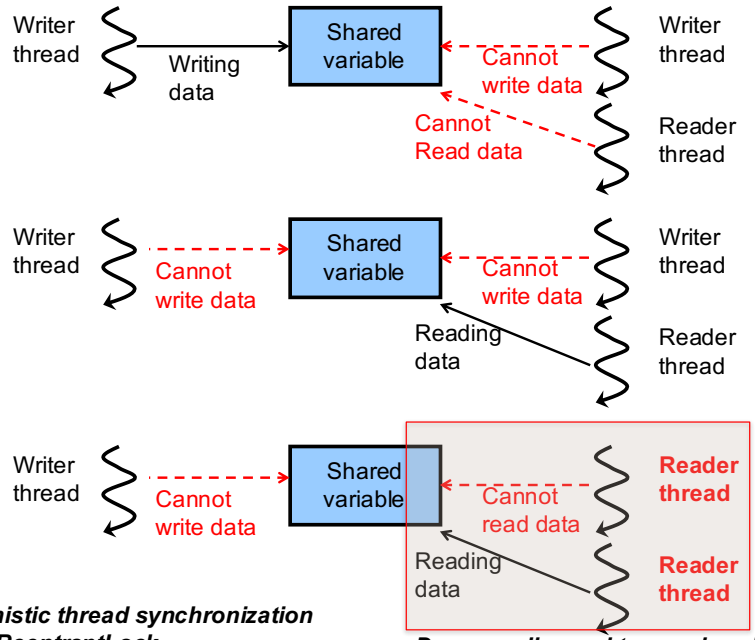
Read and Write Locks (Read-Write Locks)

Read-Write Locks

- Regular lock
  - A variable must be guarded/protected with a (regular) lock when multiple threads share the variable.
  - java.util.concurrent.locks.ReentrantLock
- Read-Write lock
  - A slight extension to a regular lock (ReentrantLock)
  - A bit more *optimistic* than a regular lock to seek potential performance improvement.
    - java.util.concurrent.locks.**ReentrantReadWriteLock**
      - java.util.concurrent.locks.ReentrantReadWriteLock.**ReadLock**
      - java.util.concurrent.locks.ReentrantReadWriteLock.**WriteLock**

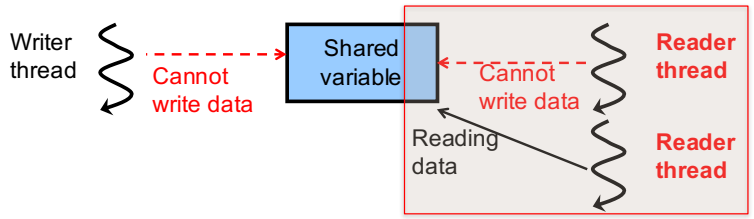
Room for Performance Improvement?

- Locking is often computationally expensive.
- Where to gain performance improvement?
  - If you have "reader" threads only, is it necessary to synchronize (i.e., mutually exclude) them?
    - No, if you have no "writer" threads.
    - Why not being optimistic about locking for "reader" threads?

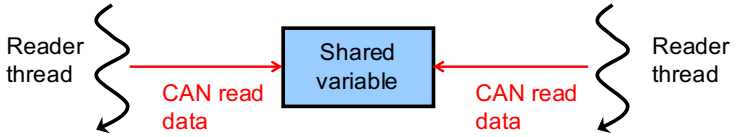


Pessimistic thread synchronization with a ReentrantLock

Do we really need to synchronize (i.e. mutually exclude) these readers?



Pessimistic thread synchronization with a ReentrantLock



Optimistic thread synchronization

ReadWriteLock

- ReentrantReadWriteLock
  - In java.util.concurrent.locks

```
public class ReentrantReadWriteLock implements ReadWriteLock{
 public class ReentrantReadWriteLock.ReadLock
     implements Lock{}
 public class ReentrantReadWriteLock.WriteLock
     implements Lock{}
 public ReentrantReadWriteLock.ReadLock readLock(){}
 public ReentrantReadWriteLock.WriteLock writeLock(){} }
```

  - Provides two locks; one for writers, and the other for readers.
    - ReadLock to read data from a shared variable.
    - WriteLock to write data to a shared variable.
    - Inner singleton classes
  - Provides factory methods for the two locks.

- A reader can acquire a read lock even if it is already held by another reader,
  - **AS FAR AS** no writers hold a write lock.
- Writers can acquire a write lock **ONLY IF** no other writers and readers hold read/write locks.

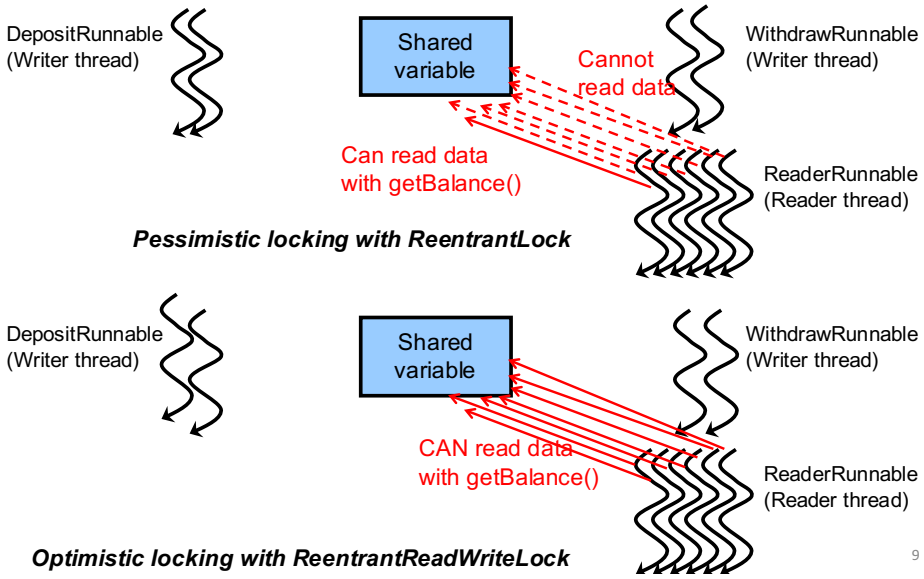
Can a thread acquire…? \ When another thread holds …?	ReadLock	WriteLock
ReadLock	Y	N
WriteLock	N	N

ReadLock and WriteLock

- Work similarly to ReentrantLock.
  - Support nested locking and thread reentrancy.
  - A condition object is returned when calling newCondition() on a write lock.
  - Calling newCondition() on a read lock generates an UnsupportedOperationException.
    - Readers never need condition objects.
    - Readers never call signal() and signalAll().

Sample Code

- ThradSafeBankAccount3 and ThradSafeBankAccount4



Sample Code

- ThradSafeBankAccount3
  - 43 msec
- ThradSafeBankAccount4
  - 33 msec
    - 23% (10/43) faster
      - thanks to optimistic locking

When to Use a Read-Write Lock?

- When many reader threads run.
- When reader threads run more often than writer threads.
- When a read operation takes a long time.

HW 16

- Recall a previous HW to implement a concurrent access counter, assuming the development of a web server
- AccessCounter
  - Maintains a HashMap that pairs a relative file path and access count.
  - increment() accepts a file path and increments the file's access count.
  - getCount() accepts a file path and returns the file's access count.
- Use a ReentrantReadWriteLock rather than ReentrantLock.
  - Writer threads
    - Threads to call increment()
  - Reader threads
    - Threads to call getCount()

HW 17

- Recall a previous HW to implement a concurrent caching mechanism, assuming the development of a web server
- Replace a ReentrantReadWriteLock with a ReentrantLock.
  - FileCache (abstract class)
  - FileCacheLFU (w/ ReentrantLock)
  - FileCacheLRU (w/ ReentrantLock)
  - FileCacheLFURW (w/ ReentrantReadWriteLock)
  - FileCacheLRURW (w/ ReentrantReadWriteLock)

Note

- Using a ReentrantLock

```
public String fetch( String targetFile ){
  acquire a lock;
  if ( targetFile is cached ){
      return targetFile's content;  }
  return cacheFile( targetFile );
  release a lock; }

private String cacheFile( … ){
   open and cache targetFile;
   return its content; }
```

- Using a RW lock

```
public String fetch( String targetFile ){
  acquire a readlock;
  if ( targetFile is cached ){
     return targetFile's content;  }
  release a readlock;
  acquire a writelock;
  return cacheFile( targetFile );
  release a writelock; }

private String cacheFile( … ){
   open and cache targetFile;
   return its content; }
```

- Is this thread safe?

Note

- Using a ReentrantLock

```
public String fetch( String targetFile ){
  acquire a lock;
  if ( targetFile is cached ){
      return targetFile's content;  }
  return cacheFile( targetFile );
  release a lock; }

private String cacheFile( … ){
   open and cache targetFile;
   return its content; }
```

- Using a RW lock

```
public String fetch( String targetFile ){
  acquire a readlock;
  if ( targetFile is cached ){
     return targetFile's content;  }
  release a readlock;
  // Ctx switch can occur here.
  acquire a writelock;
  return cacheFile( targetFile );
  release a writelock; }

private String cacheFile( … ){
   open and cache targetFile;
   return its content; }
```

- Is this thread safe? NO. What if a context switch occurs b/w releasing a read lock and acquiring a write lock?

Note

- Using a RW lock

```
public String fetch( String
targetFile ){
  acquire a readlock;
  if ( targetFile is cached ){
     return targetFile's content;  }
  release a readlock;
  // Ctx switch can occur here.
  acquire a writelock;
  return cacheFile( targetFile );
  release a writelock; }

private String cacheFile( … ){
   open and cache targetFile;
   return its content; }
```

- Using a RW lock

```
public String fetch( String targetFile ){
  acquire a writelock;
  if ( targetFile is NOT cached ){
     return cacheFile( targetFile ); }
  acquire a readlock;
  release a writelock;
  return targetFile's content;
  release a readlock; }

private String cacheFile( … ){
   open and cache targetFile;
   return its content; }
```

- A thread can acquire the write lock and then the read lock before releasing the write lock.
  - Lock downgrading
  - Lock upgrading is not possible.