Lock and ReentrantLock

Lock interface:

- Lock acquisition:
  unconditional
  polled
  timed
  interruptible
- Explicit lock and unlock operations
- Provide the same memory-visibility semantics as intrinsic locks
- Can differ in locking semantics, scheduling algorithms, ordering guarantees, performance characteristics

Lock interface Methods:

- unconditional acquisition
  void lock()
  void lockInterruptibly() throws InterruptedException
- conditional acquisition
  boolean tryLock()
  boolean tryLock(long timeout, TineUnit unit) throws InterruptedException
- **unlock()**
- **Condition newCondition()**

Why new locking mechanism?

- Impossible **to interrupt** thread waiting to acquire intrinsic lock
- Impossible **to avoid waiting** for intrinsick lock acquisition forever
- Intrinsic locks **must be released in the same block** of code in which they are acquired

Using **Lock**:

- **lock** must be released in a finally block —- must be guaranteed that it is released if an exception is thrown
- also consider what happens if an exception is thrown inside **try** block, objects used must be left in consistent state

Example, **ReentrantLock**:

- same memory semantics as **synchronized** block
  acquiring **ReentrantLock** entering *synchronized* block ** releasing *ReentrantLock* exiting **synchronized** block
- reentrant semantics equivalent to **synchronized** block

Polled and timed acquisition

Polled and timed acquisition (**tryLock**) methods allow to avoid deadlocks without lock ordering.

- trying to acquire locks one by one, backing off and releasing all of them if failed to acquire one of them
  can retry until succeeded
  can implement time budget support

Interruptible lock acquisition

lockInterruptibly (and **tryLock(long, TimeUnit)**) can be used to implement cancellable tasks.

Non-block-structured locking

Reducing lock granularity can enhance scalability. **Lock** implementations:

- more complex due to manual lock release
- more flexible

Performance considerations

For synchonization primitives, contended performance is the key to scalability. If more resources are expended on lock management and scheduling, fewer are available for the applicaiton. Better lock implementation makes fewer system calls, forces fewer context switches, initiates less memory-synchronization traffic on the shared memory bus.

Since Java 6 intrinsic locking uses an improved algorithm for managing intrinsic locking, similar to that used by **ReentrantLock**, which closes the gap between the two considerably.

Fairness

- Threads acquire a **fair** lock in the order in which they requested it.
  Thread is queued if the lock is held or other threads are waiting for it.
- Nonfair* lock permits barging, threads requesting lock can jump ahead of the queue of waiting threads if the lock is available when it it requested.
  Thread is queued only if the lock is currently held.
- **tryLock** always barges, even for fair locks.

When threads **lock frequently** and **for a short period of time fair** locking will be less performant
Fairness has a significant cost of suspending and resuming threads. While next in the queue thread waiting thread resumes, another (running) thread could have already acquired and released the lock.
Statistical fairness guarantee —- promising that a blocked thread will eventually acquire the lock —- is good enough, most algorithms don’t require fair locking for their correctness.

Intrinsic locking is unfair.

Choosing between synchronized and ReentrantLock

- **Intrinsic locking** advantages
  simpler and less dangerous
  information about lock acquisition of **ReentrantLock** cannot be tied to specific stack frames due to non-block-structured nature of **ReentrantLock**
  current performance optimizations (locks elision for thread-confined lock objects, lock coarsening)
  future performance improvements are likely to favor **synchronized** over **ReentrantLock**
- **ReentrantLock** advantage is more available flexibility (described earlier)

ReadWriteLock

3 kinds of overlap:

- writer/writer
- reader/writer
- reader/reader

In many cases data structures are “read-mostly”. **ReadWriteLock** allows multiple readers to read data at once: many readers, but only a single writer at one moment. Can improve performance for frequently accessed read-moslty data.

The interaction between **read** and **write** locks allows for a number of possible implementations:

- **Release preference** —- when a writer releases the lock, who should be given priority, readers, writers, or whoever asked first?
- **Reader barging** —- if the lock is held by readers, some writers are queued, should newly arrived readers be granted immediate access, or should they wait behind writers? (if given —- higher scalability by the cost of starving writers)
- **Reentrancy** —- are the read and write locks reentrant?
- **Downgrading** —- can write lock be exchanged for a read lock? other writers would still be unable to modify data
- **Upgrading** —- can read lock be exchanged for a write lock? if two readers try to upgrade lock, one of them must is to fail and must release the reader lock

ReentrantReadWriteLock characteristics:

- reentrant locking semantics for both kinds of locks
- supports both **fair** and **non-fair** modes
  fair mode consequence: if readers acquired the lock and writer is queued, no more readers can acquire the lock until the writer lock is obtained and released
  non-fair mode, ordering is not specified
- downgrading is permitted, upgrading is not (possibility of a deadlock)