Concurrency: Multi-core Programming & Data Processing

Lab 9

-- Read/Write Locks --

Use case

- Application in which read operations are dominant
- If a resource is not altered, synchronization between reading threads is useless and costly
- If write operation occurs, it must proceed alone
- **Read/Write lock**: specialized on these particular cases

Overview

- `java.util.concurrent` package, since Java 5
- `ReentrantReadWriteLock`
- Operations:
  - `readlock()` -> returns lock used for reading, can be acquired by multiple threads
  - `writelock()` -> return lock used for writing, exclusive
  - methods to determine if locks are held/contended
- Support for downgrading (from write lock to read lock)
  - doesn't work both ways
- `Condition` support (only for write lock)

ReentrantReadWriteLock Drawbacks

- Suffers from **starvation**
- Read lock cannot be upgraded
- No optimistic reads
  - in fact, all operations are **pessimistic**
  - if any thread holds the write lock then all reading threads get suspended
- **Overhead**
  - depends on the frequency the data is read vs. modified, duration of read and write, contention
  - short reads are bad

StampedLock

- Appears in Java 8
- 3 modes of controlling read/write access:
  - reading
  - writing
  - optimistic reading
- Conditional conversions across the modes
- E.g., useful when multiple fields need to be read at once optimistically

StampedLock Semantics

- When grabbing the lock: returns a long number (aka **the stamp**)
- To unlock: stamp is used
- Example:

```
StampedLock sl = new StampedLock();
long stamp = sl.writeLock();
try {
// do something that needs exclusive access
} finally {
  sl.unlockWrite(stamp); }
```

Exercises

- Modify the implementation of the read-write lock found in lab9.pdf so that it ensures that writers do not starve (i.e., readers cannot prevent writers from acquiring the lock infinitely). This implementation does not need to be FIFO nor reentrant.