More synchronization mechanisms

- DEADLOCK
- READERS/WRITER LOCKS
- CONDITION VARIABLES

Reading vs. writing

- Recall data race:
  - Multiple concurrent reads of same memory: *not* a problem
  - Multiple concurrent writes of same memory: problem
  - Multiple concurrent read & write of same memory: problem
- So far:
  - If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time
- But this is unnecessarily conservative:
  - Could still allow multiple simultaneous readers!

Example: hashtable

- Consider a hashtable with one coarse-grained lock
  - So only one thread can perform operations at a time

- But suppose:
  - There are many simultaneous `lookup` operations
  - `insert` operations are very rare

- Note: important that `lookup` does not actually mutate shared memory (like a move-to-front list operation or splay trees would)

Readers/writer locks

A new synchronization ADT: the readers/writer lock

$0 \leq writers \leq 1$
$0 \leq readers$
$writers*readers=0$

- A lock's states fall into three categories:
  - "not held"
  - "held for writing" by one thread
  - "held for reading" by *one or more* threads

- `new:` make a new lock, initially "not held"
- `acquire_write:` block if currently "held for reading" or "held for writing", else make "held for writing"
- `release_write:` make "not held"
- `acquire_read:` block if currently "held for writing", else make/ keep "held for reading" and increment *readers count*
- `release_read:` decrement readers count, if 0, make "not held"

Pseudocode example (not Java)

```
class Hashtable<K,V> {
  …
  // coarse-grained, one lock per table
  RWLock lk = new RWLock();
  V lookup(K key) {
       int bucket = hasher(key);
       lk.acquire_read();
       … read array[bucket] …
       lk.release_read();
  }
  void insert(K key, V val) {
       int bucket = hasher(key);
       lk.acquire_write();
       … write array[bucket] …
       lk.release_write();
  }
}
```

Semantic details

- A readers/writer lock implementation ("not our problem") usually gives *priority* to writers:
  - Once a writer blocks, no readers *arriving later* will get the lock before the writer
  - Otherwise an `insert` could ***starve*** (e.g., if readers are very common)
- Some libraries support *upgrading* from reader to writer
- Re-entrant?
  - Mostly an orthogonal issue
- Why not use readers/writer locks with more fine-grained locking, like on each bucket?
  - Not wrong, but likely not worth it due to low contention

In Java

- Java's `synchronized` statement does not support readers/writer
- Instead, library `java.util.concurrent.locks.ReentrantReadWriteLock`
    - Different interface:
        - methods `readLock` and `writeLock` return objects that themselves have `lock` and `unlock` methods;
        - need to release the lock explicitly (e.g., exceptions)
    - Does ***not*** have writer priority or reader-to-writer upgrading
        - Always read the documentation!