Shared Objects:
Locks, Condition Variables, and Best Practices

Too Much Milk: Lessons

- Last solution works, but it is really unsatisfactory:
  - Complicated; proving correctness is tricky even for the simple example
  - Inefficient: while thread is waiting, it is consuming CPU time
  - Asymmetric: hard to scale to many threads
  - Incorrect(?) : instruction reordering can produce surprising results

A better way

- How can we do better?
  - Define higher-level programming abstractions (shared objects, synchronization variables) to simplify concurrent programming
    - lock.acquire() – wait until lock is free, then grab it • atomic
    - lock.release() – unlock, waking up a waiter, if any • atomic

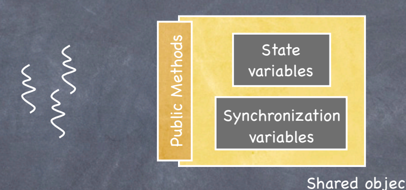
Jack/Jill/even Dame Dob!

```
Kitchen::buyIfNeeded() {
    lock.acquire():
    if (milk == 0) {
          milk++;
      }
    lock.release();
}
```

  - Use hardware to support atomic operations beyond load and store

A better way

- Extend the modularity of OO programming to multithreaded programming



- Details of synchronization are hidden behind a clean interface
- Synchronization variables regulate access to shared variables
- Hardware support for more powerful atomic operations

Concurrent Program

Shared Objects
(bounded buffer, barber chair...)

Synchronization Objects
(lock, condition variable,...)

Atomic Read-Modify-Write
(test&set, disable interrupts...)

Critical Sections

- A critical section is a segment of code involved in reading and writing a shared data area
  - It appears to execute atomically
- Critical sections are used profusely in an OS to protect data structures (e.g., queues, shared variables, lists, ...)
- Key assumptions:
  - Finite Progress Axiom: Processes execute at a finite, but otherwise unknown, speed.
  - Processes can halt only outside of the critical section (by failing, or just terminating)

The Critical Section Problem

- Mutual Exclusion: At most one thread is executing in the CS (Safety)

The Critical Section Problem

- Mutual Exclusion: At most one thread is executing in the CS (Safety)
- Access Opportunity: If no threads are executing in the CS and some threads attempts to enter the CS, then eventually a thread succeeds (Liveness)

The Critical Section Problem

- Mutual Exclusion: At most one thread is executing in the CS (Safety)
- Access Opportunity: If no threads are executing in the CS and some threads attempts to enter the CS, then eventually a thread succeeds (Liveness)
- Bounded waiting: If thread T attempts to enter the CS, then there exists a bound on the number of times other threads succeed in entering the CS before T does. (Safety? Liveness?)
  - If the bound is left unspecified, it is a liveness property, because I could always extend the execution to show that a bound exists
  - As soon as a specific bound is offered, though, it becomes a safety property, since it must hold in every prefix of the execution

Locks: API

- Two states
  - Busy
  - Free
- Two methods
  - Lock::acquire()
    - waits until lock is Free and then atomically makes lock Busy
  - Lock::release()
    - makes lock Free. If there are pending acquire(), causes one to proceed

Locks and critical section

- Mutual Exclusion: At most one thread holds a lock (Safety)
- Access Opportunity: If no threads holds the lock and some threads attempt to acquire it, then eventually a thread succeeds in acquiring it (Liveness)
- Bounded waiting: If thread T attempts to acquire the lock, then there exists a bound on the number of times other threads successfully acquire the lock before T does. (Safety? Liveness?)
  - If the bound is left unspecified, it is a liveness property, because I could always extend the execution to show that a bound exists
  - As soon as a specific bound is offered, though, it becomes a safety property, since it must hold in every prefix of the execution

Locks and critical section

- Mutual Exclusion: At most one thread holds a lock (Safety)
- Access Opportunity: If no threads holds the lock (has returned from acquire() more often than release()) and some threads attempt to acquire it, then eventually a thread succeeds in acquiring it (Liveness)
- Bounded waiting: If thread T attempts to acquire (not yet returned from a call to acquire()) the lock, then there exists a bound on the number of times other threads successfully acquire the lock before T does. (Safety? Liveness?)
  - If the bound is left unspecified, it is a liveness property, because I could always extend the execution to show that a bound exists
  - As soon as a specific bound is offered, though, it becomes a safety property, since it must hold in every prefix of the execution

A thread-safe queue

```
const int MAX = 10

class TSQueue {
private:
   Lock lock;

   int items[MAX];
   int nFull;
   int firstFull;
   int nextEmpty;

public:
   TSQueue();
   ~TSQueue(){ };
   bool tryInsert(int item);
   bool tryRemove(int *item);
```

```
bool
TSQueue::tryInsert(int item)
{
   bool ret = false;
   lock.Acquire();
   if (nFull < MAX){
      items[nextEmpty] = item;
      nFull++;
      nextEmpty = (nextEmpty + 1) % MAX
      ret = true;
   }
   lock.Release();
   return ret;
}
```

```
bool
TSQueue::tryRemove(int *item)
{
   bool ret = false;
   lock.Acquire();
   if (nFull > 0){
      *item = items[firstFull] ;
      nFull--;
      firstFull = (firstFull + 1) % MAX
      ret = true;
   }
   lock.Release();
   return ret;
}
```

Using the queue

```
int main (int argc, char **argv)
{
   TSQueue * queue[3];
   sthread_t workers[3];;
   int ii, jj, ret;
   bool success;

   for (ii = 0; ii < 3; ii++) {
      queues[ii] = new TSQueue();
      sthread_create_p(&workers[ii], putSome,
                       queues[ii]);

   for (ii = 0; ii < 3; ii++) {
      printf ("Queue %d: \n", ii);
      for (jj = 0; jj < 20; jj++) {
         success = queues[ii]->tryRemove(&ret);
         if (success) {
            printf("Got %d\n", ret);
         }
         else {
            printf("Nothing there\n");
         }
      }
   }
}
```

```
void *putSome(void *tsqueuePtr) {
   int ii;
   TSQueue * queue = (TSQueue *) tsqueuePtr;

   for (ii = 0; ii < 100; ii++) {
      queue->tryInsert(ii);
   }
   return NULL;
}
```

Implementing locks

- Generally requires some degree of hw support
- Two common approaches
  - Disable interrupts
    - uniprocess architectures only
  - Atomic read-modify-writes instructions
    - uni and multi-processor architectures

Disabling Interrupts

- Key observations:
  - On a uni-processor, an operation is atomic if no context-switch in the middle of the operation
    - Mutual exclusion by preventing context switch
  - Context switch occurs because of:
    - Internal events: system calls and exceptions
    - External events: interrupts
- Preventing context switches
  - Eliminate internal events: easy (under program control)
  - Eliminate external events: disable interrupts!

A simple solution

```
Lock::Acquire() { disable interrupts(); }
Lock::Release() { enable interrupts(); }
```

A ~~simple~~ flawed solution

```
Lock::Acquire() { disable interrupts(); }
Lock::Release() { enable interrupts(); }
```

- Once interrupts are disabled, thread can't be stopped
- Critical section can be very long
  - can't wait too long to respond to interrupts

A better solution
(queueing locks on a uniprocessor)

- Disable interrupts just to protect the lock's data structure
- Reenable interrupts as soon as lock is acquired

```
class Lock {
  private:
    int value = FREE;
    Queue waiting;

  public:
    void Lock::Acquire() {
      disableInterrupts();
      if (value = BUSY) {
        waiting.add (current thread's TCB);
        suspend();
      }
      else {
        value = BUSY;
      }
      enableInterrupts();
    }

    void Lock::Release() {
      disableInterrupts();
      if (waiting.notEmpty() {
        move one TCB from waiting to ready
      }
      else {
        value = FREE;
      }
      enableInterrupts();
    }
}
```

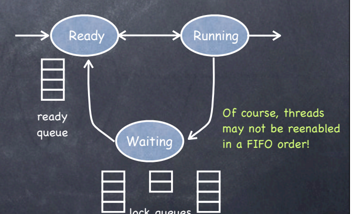
A better solution
(queueing locks on a uniprocessor)

- Disable interrupts just to protect the lock's data structure
- Reenable interrupts as soon as lock is acquired

```
class Lock {
  private:
    int value = FREE;
    Queue waiting;

  public:
    void Lock::Acquire() {
      disableInterrupts();
      if (value = BUSY) {
        waiting.add (current thread's TCB);
        suspend();
      }
      else {
        value = BUSY;
      }
      enableInterrupts();
    }

    void Lock::Release() {
      disableInterrupts();
      if (waiting.notEmpty() {
        move one TCB from waiting to ready
      }
      else {
        value = FREE;
      }
      enableInterrupts();
    }
}
```



A better solution
(queueing locks on a uniprocessor)

- Disable interrupts just to protect the lock's data structure
- Reenable interrupts as soon as lock is acquired

```
class Lock {
  private:
    int value = FREE;
    Queue waiting;

  public:
    void Lock::Acquire() {
      disableInterrupts();
      if (value = BUSY) {
        waiting.add (current thread's TCB);
        suspend();
      }
      else {
        value = BUSY;
      }
      enableInterrupts();
    }

    void Lock::Release() {
      disableInterrupts();
      if (waiting.notEmpty() {
        move one TCB from waiting to ready
      }
      else {
        value = FREE;
      }
      enableInterrupts();
    }
}
```

Thread calls suspend() with interrupts disabled: who reenables them?
- The next thread to run!

What about multiprocessors?

- Disabling interrupts is not enough!
- Atomic Read-Modify write instructions
  - Test&Set
    - atomically
      - reads a value from a memory location
      - writes "1" to that location
  - Compare&Swap (CAS)
    - atomically
      - compares content of a memory location to a given value
      - if identical, sets memory location to a given new value
  - Load linked/Store conditional (LL/SC)
    - LL returns the value of a memory location
    - A subsequent SC to that memory location succeeds only if that location has not been updated since LL

Multiprocessor spinlocks

```
class SpinLock {
  private:
    int value = 0; // 0 = FREE; 1 = BUSY

  public:
    void SpinLock::Acquire() {
      while (test_and_set (&value)) // while BUSY
        ;  // spin
    }

    void SpinLock::Release() {
      value = 0;
    }
}
```

- A thread waiting for a BUSY lock "spins"
  - not too bad as long as critical section is much shorter than time between context switches

Multiprocessor queueing locks

```
class Lock {
  private:
    SpinLock spinlock;
    int value = FREE;
    Queue waiting;

  public:
    void Lock::Release() {
      spinlock.Acquire();;
      if (waiting.notEmpty() {
        otherTCB = waiting.removeOne();
        readyList->add(otherTCB);
      }
      else {
        value = FREE;
      }
      spinLock.Release();
    }

    void Lock::Acquire() {
      spinlock.Acquire();
      if (value = BUSY) {
        disableInterrupts();
        readyList->removeSelf(myTCB);
        waiting.add (myTCB);
        spinlock.Release();
        suspend();
        enableInterrupts();
      }
      else {
        value = BUSY;
        spinlock.Release();
      }
    }
}
```

Multiprocessor queueing locks

```
class Lock {
  private:
    SpinLock spinlock;
    int value = FREE;
    Queue waiting;

  public:
    void Lock::Release() {
      spinlock.Acquire();
      if (waiting.notEmpty() {
        otherTCB = waiting.removeOne();
        readyList->add(otherTCB);
      }
      else {
        value = FREE;
      }
      spinLock.Release();
    }

    void Lock::Acquire() {
      spinlock.Acquire();
      if (value = BUSY) {
        disableInterrupts();
        readyList->removeSelf(myTCB);
        waiting.add (myTCB);
        spinlock.Release();
        suspend();
        enableInterrupts();
      }
      else {
        value = BUSY;
        spinlock.Release();
      }
    }
}
```

Disable interrupts to avoid "context switch of death"

Multiprocessor queueing locks

```
class Lock {
  private:
    SpinLock spinlock;
    int value = FREE;
    Queue waiting;

public:
  void Lock::Release() {
    spinlock.Acquire();
    if (waiting.notEmpty() {
      otherTCB = waiting.removeOne();
      readyList->add(otherTCB);
    }
    else {
      value = FREE;
    }
    spinLock.Release();
  }
}
```

```
void Lock::Acquire() {
  spinlock.Acquire();
  if (value = BUSY) {
    disableInterrupts();
    readyList->removeSelf(myTCB);
    waiting.add (myTCB);
    spinlock.Release();
    suspend();
    enableInterrupts();
  }
  else {
    value = BUSY;
    spinlock.Release();
  }
}
```

Disable interrupts to avoid "context switch of death"

readyList is protected by its own (spin) lock!

Beyond mutual exclusion

- Locks provide mutual exclusion
  - protect critical sections
  - implementation may involve a critical section
    - Atomic RMW-operations to break cycle
- "There are more things in heaven and earth..."
  - wait for another thread to take action
    - wait to remove item until bounded queue is not empty

Polling

- Check repeatedly the state of interest

```
int  TSQueue::remove()
{
  int ret;
  bool empty;
  do {
    empty = tryRemove(&ret);
  } until (!empty)
  return ret;
}
```

- Wasteful
  - may actually delay running the thread that will change the state and restore progress!
- Adding a delay after each check is no fix
  - suspending and scheduling is not free
  - higher latency

Condition Variables

- Enable threads to wait efficiently for changes to shared state protected by a lock
- Has no state... just a waiting queue
  - not much of a variable!
- Three methods
  - CV::wait(Lock *lock)
    - releases lock and atomically suspends calling thread by moving its TCB on the waiting queue
  - CV:signal()
    - moves one thread from waiting queue to ready list; no-op if none
  - CV::broadcast()
    - moves all threads from waiting queue to ready list; no-op if none

How do we use condition variables?

SharedObject::someMethodThatWaits()

```
{
   lock.Acquire();
   // read or write shared state here
      while(!testOnSharedState()) {
         cv.wait(&lock);
      }
   assert(testOnSharedState());
   // read or write shared state here
   lock.Release()
}
```

SharedObject::someMethodThatSignals()

```
{
   lock.Acquire();
   // read or write shared state here

   // If the state has changed in a way
   that allows another thread to make
   progress, signal (or broadcast) on the
   appropriate cv
   cv.signal();
   lock.Release()
}
```

- IMPORTANT
  - no atomicity between signal() and return from wait()
    - when formerly waiting thread finally runs, test on shared state may not pass!
  - wait must always be called within a loop

Blocking Bounded Queue

BBQ.h

```
#include "Cond.h"
const int MAX = 10;
class BBQ {
  private
     // Syncrhonization variables
     Lock lock;
     Cond itemAdded;
     Cond itemRemoved;
     // State variables
     int items[MAX];
     int nFull;
     int firstFull;
     int nextEmpty;

  public:
     BBQ();
     ~BBQ(){ };
     bool insert(int item);
     bool tryRemove(int *item);

  private:
     inline bool isFull() {
        return (nFull == MAX ? true : false);
     }
     inline bool isEmpty() {
        return (nFull == 0 ? true : false);
     }
}
```

BBQ.cc

```
BBQ:: BBQ()
{
   nFull = 0;
   firstFull = 0;
   nextEmpty = 0;
}
```

```
void BBQ:: insert(int item)
{
   lock.Acquire();
   while(isFull()) {
         itemRemoved.Wait(&lock);
   }
   assert(! isFull());
   items[nextEmpty] = item;
   nFull++;
   nextEmpty = (nextEmpty + 1) % MAX

   itemAdded.Signal()
   lock.Release()
   return;
}
```

```
int BBQ:: remove(void)
{
   int ret;
   lock.Acquire();
   while(isEmpty()) {
         itemAdded.Wait(&lock);
   }
   assert(! isEmpty());
   ret = items[firstFull];
   nFull--;
   firstFull = (firstFull + 1) % MAX

   itemRemoved.Signal()
   lock.Release()
   return ret;
}
```

CV semantics: Hansen vs. Hoare

- The condition variables we have defined obey Hansen (or Mesa) semantics
  - signaled thread is moved to ready list, but mot guaranteed to run right away
- Hoare proposes an alternative semantics
  - signaling thread is suspended and, atomically, ownership of the lock is passed to one of the waiting threads, whose execution is immediately resumed

What are the implications?

Hansen/Mesa semantics

- signal() and broadcast() are hints
  - adding them affects performance, never safety
- Shared state must be checked in a loop (could have changed)
  - robust to spurious wakeups
- Simple implementation
  - no special code for thread scheduling or acquiring lock
- Used in most systems
- Sponsored by a Turing Award
  - Butler Lampson

Hoare semantics

- signaling is atomic with the resumption of waiting thread
  - shared state cannot change before waiting thread is resumed
- Shared state can be checked using an if statement
- Makes it easier to prove liveness
- Tricky to implement
- Used in most books
- Sponsored by a Turing Award
  - Tony Hoare

Implementing Condition Variables

```
class Cond
{
   private:
      Spinlock spinlock;
      Queue = waiting;

   public:
```

```
void Cond::Wait(Lock *lock) {
   spinlock.Acquire();
   disableInterrupts();
   readyList->removeSelf(myTCB);
   waiting.add(myTCB);
   lock->Release();
   spinlock.Release();
   suspend;

   enableInterrupts();
   lock.Acquire();
}
```

```
void Cond::Signal() {
   spinlock.Acquire();
   if (waiting.notEmpty()) {
      otherTCB = waiting removeOne();
      readyList->add(otherTCB)
   }
   spinlock.Release();
}
```

```
void Cond::Broadcast() {
   spinlock.Acquire();
   if (waiting.notEmpty()) {
      move all TCBs from
      waiting to ready;
   }
   spinlock.Release();
}
```

```
}
```

Semaphores

- Introduced by Dijkstra in the THE operating system
- Stateful
  - a semaphore has a non negative VALUE associated to it
- Two operations

Semaphore::P()

- wait until VALUE is positive
- when so, atomically decrement VALUE by 1

Semaphore::V()

- increment VALUE by 1
- resume (if any) a thread is waiting on P(); that thread will decrement VALUE and return

Semaphores in mutex and condition synchronization

Semaphore new mutex(1)

Jack/Jill/even Dame Dob!

```
Kitchen::buyIfNeeded() {
  mutex.P():
  if (milk == 0) {
      milk++;
    }
  mutex.(V)();
}
```

- General synchronization
  - initialize VALUE to 0
  - Semaphore::P() similar to Cond::Wait(&lock)
  - Semaphore::(V) similar to Cond::Signal()
  - BIG DIFFERENCE
    - if no one is waiting, signal() is a no-op
    - V() always increments VALUE
    - useful when hw device and OS share a data structure

Designing multithreaded programs

- Building a shared object class involves familiar steps
  - decompose the problem into objects
  - for each object
    - define a clear interface
    - identify right internal state an invariants
    - implement methods that manipulate state appropriately
- The new steps are straightforward
  - add a lock
  - add code to acquire and release the lock
  - identify and add condition variables
  - add loops to wait using condition variable(s)
  - add signal() and broadcast() calls

Managing locks

- Add a lock as a member variable for each object in the class, to enforce mutual exclusion on the object's shared state
- Acquire a lock at the start of each public method
- Release the lock at the end of each public method
  - You will be tempted to acquire/release lock midway through a method
  - RESIST!

Identifying condition variables

- Ask yourself: when can this method wait?
- Map each opportunity for waiting to a condition variable
  - itemRemoved vs itemAdded in BBQ example
- But you can also live with a single CV
  - in BBQ, just use somethingChanged

Identifying condition variables

- Ask yourself: when can this method wait?
- Map each opportunity for waiting to a condition variable
  - itemRemoved vs itemAdded in BBQ example
- But you can also live with a single CV
  - in BBQ, just use somethingChanged
  - ...but now insert() and remove() need to call broadcast(), not signal()

Waiting using condition variables

- Every call to Condition::Wait() should be enclosed in a loop
- Loop tests the appropriate predicate on the state
- Hint: encapsulate details of state testing in a private method function
  - get the structure of the public method right before worrying about the details

Signal vs Broadcast

- It is always safe to use broadcast() instead of signal()
  - all that is affected is performance
- signal() is preferable when
  - at most one waiting thread can make progress
  - any thread waiting on the condition variable can make progress
- broadcast() is preferable when
  - multiple waiting threads may be able to make progress
  - the same condition variable is used for multiple predicates
    - some waiting threads can make progress; others can't

The Six Commandments

1. **Thou shalt always do things the same way**
   - habit allows you to focus on core problem
   - easier to review, maintain and debug your code
2. **Thou shalt always synchronize with locks and condition variables**
   - either CV & locks or semaphores
   - CV and locks make code clearer
3. **Thou shalt always acquire the lock at the beginning of a method and release at the end**
   - make a chunk of code that requires a lock its own procedure

The Six Commandments

4. **Always hold a lock when operating on a condition variable**
   - condition variables are useless without shared state
   - shared state should only be accessed using a lock
5. **Always wait in a while() loop**
   - while works every time if does
   - makes signals hints
   - protects against spurious wakeups
6. **(Almost) never sleep()**
   - use sleep() only if an action should occur at a specific real time
   - never wait on sleep()

Readers/Writers

- Two types of users
  - Readers: never modify data
  - Writers: read and modify data
- The problem: shared database access
  - Multiple threads can safely read a record
  - If a thread is writing a record, no other thread should be reading or writing that record
- Using a lock for mutual exclusion is inefficient
  - implement new RWLock shared object

To read

```
rwLock->startRead();
// Read database entry
rwLock->doneRead();
```

To write

```
rwLock->startWrite();
// Read/Write database entry
rwLock->doneWrite();
```

Readers/Writers

- Two types of users
  - Readers: never modify data
  - Writers: read and modify data
- The problem: shared database access
  - Multiple threads can safely read a record
  - If a thread is writing a record, no other thread should be reading or writing that record
- Using a lock for mutual exclusion is inefficient
  - implement new RWLock shared object

To read

```
rwLock->startRead();
// Read database entry
rwLock->doneRead();
```

To write

```
rwLock->startWrite();
// Read/Write database entry
rwLock->doneWrite();
```

Interface and member variables

```
class RWLock{

private:
  // Synchronization

  // State variables

public:
  RWLock();
  ~RWLock() {};

  // Public methods

private:
  // Functions testing state

}
```

Interface and member variables

```
class RWLock{

private:
  // Synchronization variables
  Lock lock;
  Cond readGo;
  Cond writeGo;

  // State variables
  int activeReaders;  } whether
  int activeWriters;  } to wait
  int waiting Readers; } whom to
  int waitingWriters;  } signal

public:
  RWLock();
  ~RWLock() {};

  // Public methods
  void startRead();
  void doneRead();
  void startWrite();
  void doneWrite();

private:
  // Functions testing state
  bool readShouldWait();;
  bool writeShouldWait();

}
```

Reading methods

To read

```
rwLock->startRead();
// Read database entry
rwLock->doneRead();
```

```
void RWLock::startRead()
{

}

void RWLock::doneRead()
{

}
```

Reading methods

To read

```
rwLock->startRead();
// Read database entry
rwLock->doneRead();
```

```
void RWLock::startRead()
{
  lock.Acquire();

  lock.Release();
}
```

```
void RWLock::doneRead()
{
  lock.Acquire();

  lock.Release();
}
```

Reading methods

To read

```
rwLock->startRead();
// Read database entry
rwLock->doneRead();
```

```
void RWLock::startRead()
{
  lock.Acquire();
  while(readShouldWait() {

    goRead.wait(&lock);

  }

  lock.Release();
}
```

```
void RWLock::doneRead()
{
  lock.Acquire();

  lock.Release();
}
```

Reading methods

To read

```
rwLock->startRead();
// Read database entry
rwLock->doneRead();
```

```
void RWLock::startRead()
{
  lock.Acquire();
  waitingReaders++;
  while(readShouldWait() {
    goRead.wait(&lock);
  }
  waitingReaders--;
  activeReaders++;
  lock.Release();
}
```

```
void RWLock::doneRead()
{
  lock.Acquire();

  lock.Release();
}
```

Reading methods

To read

```
rwLock->startRead();
// Read database entry
rwLock->doneRead();
```

```
void RWLock::startRead()
{
  lock.Acquire();
  waitingReaders++;
  while(readShouldWait() {
    goRead.wait(&lock);
  }
  waitingReaders--;
  activeReaders++;
  lock.Release();
}
```

```
void RWLock::doneRead()
{
  lock.Acquire();

  if (waitingWriters > 0 and activeReaders == 0) {
    goWrite.signal();
  }
  lock.Release();
}
```

Reading methods

To read

```
rwLock->startRead();
// Read database entry
rwLock->doneRead();
```

```
void RWLock::startRead()
{
  lock.Acquire();
  waitingReaders++;
  while(readShouldWait() {
    goRead.wait(&lock);
  }
  waitingReaders--;
  activeReaders++;
  lock.Release();
}
```

```
void RWLock::doneRead()
{
  lock.Acquire();
  activeReaders--;
  if (waitingWriters > 0 and activeReaders == 0) {
    goWrite.signal();
  }
  lock.Release();
}
```

Writing methods

To write

```
rwLock->startWrite();
// Read database entry
rwLock->doneWrite();
```

```
void RWLock::startWrite()
{

}
```

```
void RWLock::doneWrite()
{

}
```

Writing methods

To write

```
rwLock->startWrite();
// Read database entry
rwLock->doneWrite();
```

```
void RWLock::startWrite()
{
  lock.Acquire();

  lock.Release();
}
```

```
void RWLock::doneWrite()
{
  lock.Acquire();

  lock.Release();
}
```

Writing methods

To write

```
rwLock->startWrite();
// Read database entry
rwLock->doneWrite();
```

```
void RWLock::startWrite()
{
  lock.Acquire();

  while(writeShouldWait() {
    goWrite.wait(&lock);
  }

  lock.Release();
}
```

```
void RWLock::doneWrite()
{
  lock.Acquire();

  lock.Release();
}
```

Writing methods

To write

```
rwLock->startWrite();
// Read database entry
rwLock->doneWrite();
```

```
void RWLock::startWrite()
{
   lock.Acquire();
   waitingWriters++;
   while(writeShouldWait() {
      goWrite.wait(&lock);
   }
   waitingWriters--;
   activeWriters++;
   lock.Release();
}
```

```
void RWLock::doneWrite()
{
   lock.Acquire();
   activeWriters--;
   if (waitingWriters > 0} {
      goWrite.signal();
   }
   else {
      goRead.broadcast():
   }
   lock.Release();
}
```

State testing functions

```
bool RWLock::readShouldWait()
{
}
```

```
bool RWLock::writeShouldWait()
{
}
```

State testing functions

```
bool RWLock::readShouldWait()
{
}
```

```
bool RWLock::writeShouldWait()
{
   if (activeWriters > 0 ||
         activeReader > 0) {
      return true;
   }
   return false;
}
```

State testing functions

```
bool RWLock::readShouldWait()
{
   if (activeWriters > 0
                         {
      return true;
   }
   return false;
}
```

```
bool RWLock::writeShouldWait()
{
   if (activeWriters > 0 ||
         activeReader > 0) {
      return true;
   }
   return false;
}
```

State testing functions

```
bool RWLock::readShouldWait()
{
  if (activeWriters > 0  ||
        waitingWriters > 0) {
        return true;
  }
  return false;
}
```

```
bool RWLock::writeShouldWait()
{
  if (activeWriters > 0 ||
        activeReader > 0) {
     return true;
  }
  return false;
}
```