ReadWrite Lock

In this lesson, we discuss a common interview question involving the synchronization of multiple reader threads and a single writer thread.

Read Write Lock

Imagine you have an application where you have multiple readers and a single writer. You are asked to design a lock that lets multiple readers read at the same time, but only one writer writes at a time.

Solution

First of all, let us define the APIs our class will expose. We'll need two for writer and two for reader. These are:

- acquireReadLock
- releaseReadLock
- acquireWriteLock
- releaseWriteLock

This problem becomes simple if you think about each case:

1. Before we allow a reader to enter the critical section, we need to make sure that there's no **writer** in progress. It is ok to have other readers in the critical section since they aren't making any modifications.

2. Before we allow a writer to enter the critical section, we need to make sure that there's **no reader or writer** in the critical section.

Let's start with the reader use case. We can have multiple readers acquire the read lock and to keep track of all of them; we'll need a count. We increment this count whenever a reader acquires a read lock and decrement it whenever a reader releases it.

Releasing the read lock is easy, but before we acquire the read lock, we need to be sure that no other writer is currently writing. Again, we'll need some variable to keep track of whether a writer is writing. Since only a single writer can write at a given point in time, we can just keep a boolean variable to denote if the write lock is acquired or not.

Additionally, we'll also need a condition variable for the readers and writers to wait while the other party is in progress. We'll use the associated mutex lock with the condition variable to guard the sections of the code where we manipulate any shared variables. Let's translate what we have discussed so far into code.

```
class ReadersWriteLock

  def initialize()
    @condVar = ConditionVariable.new
    @mutex = Mutex.new
    @writeInProgress = false
    @readers = 0
  end

  def acquireReadLock()
  end

  def releaseReadLock()
  end

  def acquireWriteLock()
  end

  def releaseWriteLock()
  end
```

```
end
```

Next we'll take a stab at the `acquireReadLock()` method.

```
def acquireReadLock()
  @mutex.lock()

  while @writeInProgress == true
    @condVar.wait(@mutex)
  end

  @readers += 1
  @mutex.unlock()
end
```

Note that multiple reader threads can invoke the above method and not block if no write is in progress thus satisfying the condition that multiple readers can read at the same time. Next let's examine the `releaseReadLock()` method.

```
def releaseReadLock()
  @mutex.lock()
  @readers -= 1
  @condVar.broadcast()
  @mutex.unlock()
end
```

When releasing the read lock, each reader decrements the `readers` count. Since multiple reader threads can modify the shared readers' count, we guard it with the mutex lock. Also, we notify any writer threads waiting on the condition variable. Note that a reader notifies the condition variable even if the readers' count is not zero. We can improve the code as follows, but the above implementation will also work.

```
def releaseReadLock()
  @mutex.lock()

  @readers -= 1

  if @readers == 0
    @condVar.broadcast()
  end
```

```
    @mutex.unlock()
  end
```

Now let's turn to the writer thread case. A writer should only proceed if no reader or writer is currently in progress, implying we need to check for two conditions. The implementation is as follows:

```
  def acquireWriteLock()
    @mutex.lock()

    while @readers != 0 or @writeInProgress == true
      @condVar.wait(@mutex)
    end
    @writeInProgress = true

    @mutex.unlock()
  end
```

The last piece is the `releaseWriteLock()` which sets the `writeInProgress` to false and notifies any reader threads waiting on the condition variable.

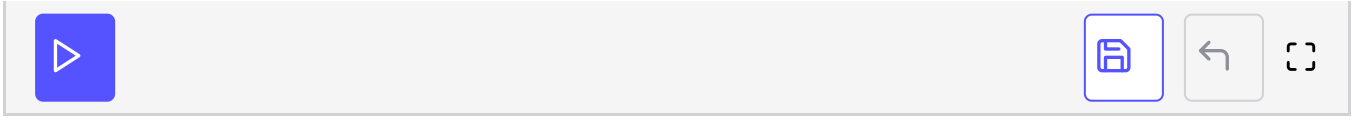
```
  def releaseWriteLock()
    @mutex.lock()

    @writeInProgress = false
    @condVar.broadcast()

    @mutex.unlock()
  end
```

The complete code appears below in the code widget.

```
require 'date'
class ReadersWriteLock

  def initialize()
    @condVar = ConditionVariable.new
    @mutex = Mutex.new
    @writeInProgress = false
    @readers = 0
  end

  def acquireReadLock()
    @mutex.lock()

    while @writeInProgress == true
```

```
      @condVar.wait(@mutex)
    end


    @readers += 1
    @mutex.unlock()
  end

  def releaseReadLock()
    @mutex.lock()

    @readers -= 1

    if @readers == 0
      @condVar.broadcast()
    end
    @mutex.unlock()
  end

  def acquireWriteLock()
    @mutex.lock()

    while @readers != 0 or @writeInProgress == true
      @condVar.wait(@mutex)
    end
    @writeInProgress = true

    @mutex.unlock()
  end

  def releaseWriteLock()
    @mutex.lock()

    @writeInProgress = false
    @condVar.broadcast()

    @mutex.unlock()
  end

  attr :readers
  attr :writeInProgress
end

def writerThread(lock)
  while true
    lock.acquireWriteLock()
    p "#{Thread.current.__id__} writing at #{DateTime.now} and current readers = #{lock.reade
    write_for = rand()
    sleep(write_for)
    lock.releaseWriteLock()
    sleep(1)
  end
end


def readerThread(lock)
  while true do
    lock.acquireReadLock()
    p "Thread #{Thread.current.__id__} reading at #{DateTime.now} and write in progress = #{l
    read_for = rand()
    sleep(read_for)
    lock.releaseReadLock()
    sleep(1)
```

```
  end
end

lock = ReadersWriteLock.new

Thread.new do
  writerThread(lock)
end

Thread.new do
  writerThread(lock)
end

3.times do |i|
  Thread.new do
    readerThread(lock)
  end
end

sleep(15)
```



The above simulation runs for 15 seconds and from the output, you can verify that a writer thread gets to write exclusively. The acquire and release statements for a writer thread must appear consecutively to prove the correct working.

Follow-up Question

Since only a single writer thread can be active at any given point in time, we can also design the writer APIs as follows:

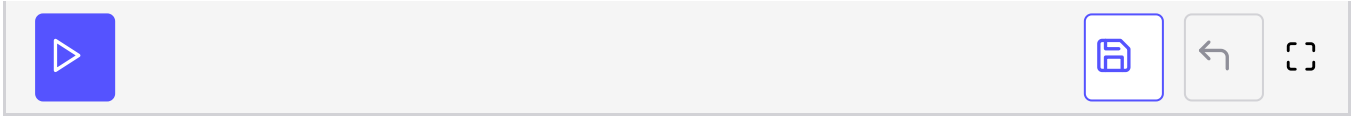
```
  def acquireWriteLock()
    @mutex.lock()

    while @readers != 0 or @writeInProgress == true
      @condVar.wait(@mutex)
    end
    @writeInProgress = true

  end

  def releaseWriteLock()

    @writeInProgress = false
    @condVar.broadcast()
```

```
    @mutex.unlock()
  end
```

The acquisition and release of the mutex variable is split across two methods. The code works just as fine, as shown in the code widget below. However, can you think of why this might be a bad idea? Read the explanation which appears after the code widget.

```
require 'date'
class ReadersWriteLock

  def initialize()
    @condVar = ConditionVariable.new
    @mutex = Mutex.new
    @writeInProgress = false
    @readers = 0
  end

  def acquireReadLock()
    @mutex.lock()

    while @writeInProgress == true
      @condVar.wait(@mutex)
    end

    @readers += 1
    @mutex.unlock()
  end

  def releaseReadLock()
    @mutex.lock()

    @readers -= 1

    if @readers == 0
      @condVar.broadcast()
    end
    @mutex.unlock()
  end

  def acquireWriteLock()
    @mutex.lock()

    while @readers != 0
      @condVar.wait(@mutex)
    end
    @writeInProgress = true

  end

  def releaseWriteLock()

    @writeInProgress = false
    @condVar.broadcast()
```

```
    @mutex.unlock()
  end

  attr :readers
end

def writerThread(lock)
  while true
    lock.acquireWriteLock()
    p "#{Thread.current.__id__} writing at #{DateTime.now} and current readers = #{lock.reade
    write_for = rand()
    sleep(write_for)
    lock.releaseWriteLock()
    sleep(1)
  end
end


def readerThread(lock)
  while true do
    lock.acquireReadLock()
    p "Thread #{Thread.current.__id__} reading at #{DateTime.now}"
    read_for = rand()
    sleep(read_for)
    lock.releaseReadLock()
    sleep(1)
  end
end

lock = ReadersWriteLock.new

Thread.new do
  writerThread(lock)
end

Thread.new do
  writerThread(lock)
end

3.times do |i|
  Thread.new do
    readerThread(lock)
  end
end

sleep(15)
```



> The above approach might seem more efficient since a writer thread only acquires and releases the condition variable once during operation. Also, we can eliminate the `writeInProgress == true` condition in the while loop of the `acquireWriteLock()` method. However, the Achilles' heel of the above solution is that if the writer thread dies between the two method

calls, the entire system would enter a deadlock!

In the code widget below, we kill the writer thread after it acquires the write lock and the program enters a deadlock. The program doesn't make any progress after the writer thread is killed as observed from the output of running the code widget below. Note that Ruby unlocks a mutex when a thread that locked it exits without unlocking. However, the deadlock, in this case, occurs because other threads are waiting on the condition variable and there's no thread to signal them.

```
require 'date'
class ReadersWriteLock

  def initialize()
    @condVar = ConditionVariable.new
    @mutex = Mutex.new
    @writeInProgress = false
    @readers = 0
  end

  def acquireReadLock()
    @mutex.lock()

    while @writeInProgress == true
      @condVar.wait(@mutex)
    end

    @readers += 1
    @mutex.unlock()
  end

  def releaseReadLock()
    @mutex.lock()

    @readers -= 1

    if @readers == 0
      @condVar.broadcast()
    end
    @mutex.unlock()
  end

  def acquireWriteLock()
    @mutex.lock()

    while @readers != 0
      @condVar.wait(@mutex)
    end
    @writeInProgress = true

  end

  def releaseWriteLock()
```

```
    @writeInProgress = false
    @condVar.broadcast()

    @mutex.unlock()
  end

  attr :readers
end

def writerThread(lock)
  while true
    lock.acquireWriteLock()
    p "#{Thread.current.__id__} writing at #{DateTime.now} and current readers = #{lock.reade
    write_for = rand()
    sleep(write_for)

    p "Killing writer thread #{Thread.current.__id__}"
    Thread.kill(Thread.current)
  end
end


def readerThread(lock)
  while true do
    lock.acquireReadLock()
    p "Thread #{Thread.current.__id__} reading at #{DateTime.now}"
    read_for = rand()
    sleep(read_for)
    lock.releaseReadLock()
    sleep(1)
  end
end

lock = ReadersWriteLock.new

Thread.new do
  writerThread(lock)
end

Thread.new do
  writerThread(lock)
end

3.times do |i|
  Thread.new do
    readerThread(lock)
  end
end

sleep(15)
```