4yp

Thomas Aston

November 14, 2023

Test

1 Modelling and analysing implementations of locks

In this section we will encapsulate the external behaviour of a lock before presenting and analysing a number of different lock implementations. We will first examine a Test-and-Test-and-Set (TTAS) lock implementation[1] before examining a Queue-lock implementation and a lock for n thread constructed from a tree of 2 thread locks.

2 Basic locks

The primary purpose of locks is to provide *mutual exclusion* between threads; that is to avoid two threads from operating concurrently on the same section of code, referred to as the *critical region*. A good lock should also fulfil some *liveness* requirements, essentially that something good will eventually happen. When devising liveness requirements we assume that no thread wil hold the lock indefinitely; otherwise most reasonable liveness requirements can be invalidated by a thread that gains the lock and never releases it. *Deadlock freedom* is a liveness requirement that if some thread is attempting to acquire the lock then, under the assumption that all threads eventually release the lock, some thread will eventually succeed in acquiring the lock. If a lock satisfies deadlock freedom and a thread tries to but never obtains the lock then other threads must complete an infinite number of critical sections. *Starvation freedom* is a liveness requirement that any thread that tries to gain the lock will eventually succeed. Other requirements/useful properties of locks will be explored later in this paper.

3 External interfaces

The most straightforward implementation of a lock can be seen in Figure 1. This provides a `lock` function for a thread to attempt to gain the lock, blocking if some other thread currently obtains the lock, and an `unlock` function for a thread to release the lock.

```
trait Lock{
  /** Acquire the Lock. */
  def lock : Unit
  /** Release the Lock. */
  def unlock : Unit
  ...
}
```

Figure 1: An interface for a simple lock

When a thread t uses a lock l which has just the lock and unlock operations available, there are three main events of importance to model:

- callLock.l.t : The thread calling the lock function
- lockObtained.l.t : The thread exiting the lock function, now holding the lock
- lockUnlocked.l.t : The thread has called the unlock function and the unlock function has been executed to the point where a thread can now reobtain the lock

We can now consider modelling ideal properties of locks using these three channels. We will let CS_A^k be interval where A is in it's critical section for the kth time; ie. the interval between $lockObtained.l.A^k$ and $lockUnlocked.l.A^k$

- Mutual Exclusion: This is specified by $CS_A^k \to CS_B^j$ or $CS_B^j \to CS_A^k$ where A and B are distinct threads. We can therefore deduce that a lock l with model X using these three channels satisfies the trace refinement:

```
Mutex = lockObtained.l?t → lockUnlocked.l.t → Mutex
X [|{|lockObtained.l.t, lockUnlocked.l.t|}|] Mutex ⊑T X
```

- Deadlock Freedom: This specifies that if some thread attempts to acquire the lock then some thread will succeed in acquiring the lock. This can be captured by the following trace refinement on lock l with threads = ThreadID

```
AcquireLock(ts) = callLock.l?t:{ThreadID \ ts} → AcquireLock(union(ts))
□ lockObtained.l?t:ts → AcquireLock(diff(ts, {t}))
Y = X [|{callLock.l.t, lockUnlocked.l.t | t ← ThreadID}|] AcquireLock({})
Y \ diff(AllChannels, {callLock.l.t, lockUnlocked.l.t | t ← ThreadID}) ⊑T
AcquireLock({})
```

- Starvation Freedom: Every thread that attempts to acquire the lock eventually succeeds

```
LockSpec :: (LockID, {ThreadID}, {ThreadID}) → Proc
LockSpec(l, ts, TS) = callLock.l?t:(diff (TS, ts)) → LockSpec(l, union(ts, {t}), TS)
                    □ lockObtained.l?t:ts → LockSpecObtained(t, l, ts, TS)
LockSpecObtained(t, l, ts, TS) = callLock.l?t2:(diff (TS, union(ts, {t}))) →
                                          LockSpecObtained(t, l, union(ts, {t2}), TS)
                    □ lockUnlocked.l.t → LockSpec(l, diff(ts, {t}), TS)
```

Figure 2: A non-starvation-free trace specification for a lock

3.1 A simple lock trace specification

Figure 2 shows a simple trace specification for a lock, where l is the identity of the lock, TS is the set of all threads and ts is the set of all threads that have communicated a callLock.l.t, but haven't yet followed that by a lockObtained.l.t. At any point, lock can be called by a thread that does not currently hold the lock and that hasn't called the lock since it last held the lock (ie. a thread cannot call the lock twice without holding it in between). If some thread X holds the lock then it can unlock whenever, with the; likewise if no thread hold the lock, then any thread that has called lock but not obtained the lock yet can obtain the lock.

This specification has the required property of mutual exclusion - once a thread has performed a lockObtained.l.t, no other threads can perform a lockObtained.l.t' until after the original thread releases the lock via lockObtained.l.t. It also satisfies deadlock-freeness since it can always communicate a callLock unless ts == TS in which causes some thread can communicate a lockObtained, followed later by a lockUnlocked. Livelock-freeness is also satisfied as all actions performed make progress towards obtaining the lock or releasing the lock once it is held. This specification does not satisfy the property of starvation-freeness as the following trace is possible:

```
callLock.l.A → (callLock.l.B → lockObtained.l.B → lockUnlocked.l.B)*
```

with thread A never gaining the lock; this is intended as locks are not required to be starvation-free, as seen later regarding the Test-and-test-and-set lock (TTAS).

This specification process for locks will be very useful later as if we can show trace-equivalence between this specification and some implementation of a lock over {callLock, lockObtained, lockUnlocked} we can use the specification in more complex systems, reducing the size of the systmes produced by FDR and hence allowing us to test larger cases than would otherwise be possible.

4 Test-and-Test-and-Set Lock

The TTAS implementation of a lock revolves around using an AtomicBoolean called state to capture whether the lock is currently held; true meaning that some thread holds the

```scala
import java.util.concurrent.atomic.AtomicBoolean

/** A lock based upon the test-and-set operation
  * Based on Herlihy & Shavit, Chapter 7. */
class TTASLock extends Lock{
  /** The state of the lock: true represents locked */
  private val state = new AtomicBoolean(false)

  /** Acquire the lock */
  def lock =
    do{
      while(state.get()){ } // spin until state = false
    } while(state.getAndSet(true)) // if state = true, retry

  /** Release the lock */
  def unlock = state.set(false)

  /** Make one attempt to acquire the lock
    * @return a Boolean indicating whether the attempt was successful */
  def tryLock : Boolean = !state.get && !state.getAndSet(true)
}
```

Figure 3: Test-and-test-and-set lock from [1]

lock and false meaning that the lock is currently free. The full Scala code can be seen in Figure 3. When a thread attempts to obtain the lock, it performs a state.getAndSet(true); a getAndSet(true) that returns false can be treated as having gained the lock, whereas a true indicates that some other thread already holds the lock. To release the lock a set(false) is done to mark the lock as available to other threads

This is a very similar implementation to the Test-and-set lock ***TODO: create figure for TASLock; Maybe also implement in CSP and compare no. getAndSets etc. (No establishable bound though so of dubious benefit)*** found in [1], however this performs get operations before attempting any getAndSet operations. Any getAndSet operation causes a broadcast on the shared memory bus between the processors, delaying all processors whilst also forcing each thread to invalidate the value of the lock from its cache, regardless of whether the value is actucally changed. As a result, it is preferrable to use less costly get operations in order to limit the usage of getAndSet operations to situations where they are likely to change the value of the underlying lock. ***Unsure as to how much detail to go into regarding performance of getAndSet, memory buses and caching etc; if relevant elsewhere then may be worth making into its own subsection?***

```
Var(value, get, set, gAS) =
  get?_!value → Var(value, get, set, gAS)
  □ set?_?value' → Var(value', get, set, gAS)
  □ gAS?_!value?value' → Var(value', get, set, gAS)

```

Figure 4: A process encapsulating an Atomic variable with get, set and getAndSet operations

4.1 Modelling with CSP

Firstly, in order to model the TTAS lock, we need a process that acts as an AtomicBoolean to model the state variable. Figure 4 shows a process Var than takes an initial value, and channels get, set : ThreadID −> Bool and getAndSet : ThreadID −> Bool −> Bool. By initialising this with a value of false, it can be used to represent the state variable from the Scala implementation as it offers the same operations as are used in the Scala implementation.

We can then implement the operations of the lock itself. The Unlock procedure is quite trivial, simply setting the state to false and then perparing to try to obtain the lock again

```
Unlock(t) = setState.t! False → NotHolding(t) −− def unlock = state.set(false)
```

The lock operation gets the value of State repeatedly until it returns false, with this being effectively equivalent to while(state.get()){}. Once it has returned false, a getAndSet is tried on state; if the previous value was false then the thread now holds the lock, otherwise some other thread has just obtained the lock and we return to repeating the getState checks. We can treat an occurance of gASState.t?False!True as thread t obtaining the lock.

```
Lock(t) = getState.t?s → if s == True then Lock(t) −− while(state.get()){ }
          else gASState.t?v!True → if v == False then Holding(t)
               −− do ... while(state.getAndSet(true))
          else Lock(t)
```

We finally have two seperate processes Holding(t) and NotHolding(t) which are used to represent threads that either have the lock or don't respectively and are about to either unlock or try to lock.

```
Holding(t) = Unlock(t)
NotHolding(t) = callLock.L.0.t → Lock(t)
```

All the threads can now be interleaced and synchronized over internal channels with the state variable in order to produce the lock system. Since gASState.t.False.True corresponds to when a thread obtains the lock and setState.t.False corresponds to a thread releasing the lock, we can hence rename these communications to lockObtained.L.0.t and lockUnlocked.L.0.t respectively and hide all other internal communications in order to produce ActualSystemR

```
AllThreads = ||| t : ThreadID • NotHolding(t)
InternalChannels = {|getState, setState , gASState|}
LockEvents = {|lockObtained, lockUnlocked, callLock|}
AllChannels = Union({InternalChannels, LockEvents})
ActualSystem = (AllThreads [|InternalChannels|] State)
ActualSystemR = (ActualSystem [[gASState.t.False.True \ lockObtained.L.0.t,
  setState.t.False \ lockUnlocked.L.0.t | t ← ThreadID]]) \ InternalChannels
```

4.2 Model Refinements

There are four main checks we perform on the system to check that it behaves as we'd expect

1. Firstly we check whether the ActualSystem models are both divergence and deadlock free, which it is. We then check that ActualSystemR does diverge; this is expected as after thread A obtains the lock, thread B can perform an infinite number of getState.t?Trues so hiding all getState events leads to an infinite number of hidden events in the case and therefore diverges.

```
assert ActualSystem :[divergence free]
assert ActualSystemR :[divergence free]
assert ActualSystem :[deadlock free]
```

2. We next check mutual exclusion. We construct a process CheckMutualExclusion(L.0) which allows any thread to communicate lockObtained.L.0?t, but then forces the next communication over the lockObtained and lockUnlocked channels to be lockUnlocked.L.0.t. This test also passes as expected.

```
assert CheckMutualExclusion(L.0) ⊑T ActualSystemR \ diff(LockEvents,
  OnlyRootObtain(L.0, ThreadID))
```

3. The next check we perform is that the lock does not diverge before it is first obtained; as shown above the lock can diverge after it is held due to infinitely repeating get.t.Trues. The lock does however not diverge before it is first obtained, hence this test returns true

```
CheckNoDiv = gASState?t!False.True → STOP
             □ getState?t._ → CheckNoDiv
```

```
assert (ActualSystem [|{|gASState, getState|}|] CheckNoDiv) \ {|getState|}
    :[divergence free]
```

4. The last test we complete is that ActualSystemR is trace equivalent to LockSpec(L.0, {}, ThreadID). This again returns true, hence all assertions pass.

```
assert LockSpec(L.0, {}, ThreadID) ⊑T ActualSystemR
assert ActualSystemR ⊑T LockSpec(L.0, {}, ThreadID)
```

5 Queue Lock

6 Tree lock

End of Test

References

[1] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123973375.

```scala
import ox.cads.util.ThreadID
import java.util.concurrent.atomic.{AtomicInteger,AtomicIntegerArray}

/** A lock using an array to store waiting threads in a queue.
  * Based on Herlihy & Shavit, Section 7.5.1.
  * @param capacity the number of workers who can use the lock. */
class ArrayQueueLock(capacity: Int) extends Lock{
  // ThreadLocal variable to record the slot for this thread
  private val mySlotIndex =
    new ThreadLocal[Int]{ override def initialValue = 0 }
  //println("AQL")

  private val padding = 16 // # words per cache line
  private val size = padding*capacity // # entries in the flag array

  // flag(i) is set to Go to indicate that the thread waiting on it can
  // proceed. We only use slots that are a multiple of padding, to avoid
  // false sharing.
  private val flags = new AtomicIntegerArray(size)
  private val Go = 1; private val Wait = 0
  flags.set(0, Go) // other flags initially equal Wait

  private val tail = new AtomicInteger(0) // the next free slot / padding

  // Array, indexed by ThreadIDs, where threads store the index of their flag
  // in flags.
  //private val slotIndices = new Array[Int](capacity)

  def lock = {
    val slot = (tail.getAndIncrement * padding) % size
    // slotIndices(ThreadID.get) = slot
    mySlotIndex.set(slot)
    while(flags.get(slot) == Wait){ } // spin on flag(slot)
  }

  def unlock = {
    val slot = mySlotIndex.get // slotIndices(ThreadID.get)
    flags.set(slot, Wait) // anyone waiting here must wait
    flags.set((slot+padding)%size, Go) // next thread can progress
  }

  // I don't think tryLock can be implemented.
  def tryLock : Boolean = ???
}
```

Figure 5: A lock implementation storing a queue of waiting threads from [1]