4yp

Thomas Aston

November 20, 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.

1.1 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.

1.2 External interfaces

The most straightforward interfaces 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.

When a thread `t` uses a lock `l` with there are four main events of importance to model in CSP:

```
1    trait Lock{
2      /** Acquire the Lock. */
3      def lock : Unit
4      /** Release the Lock. */
5      def unlock : Unit
6      ...
7    }
```

Figure 1: A Scala interface for a simple lock

- callLock.l.t : The thread calls the lock function;

- lockObtained.l.t : The thread exits the lock function, now holding the lock;

- lockUnlocked.l.t : The thread has calls the unlock function and the unlock function has been executed to the point where a thread can now reobtain the lock

- end.t : The thread will make no further calls to the lock Necessary? Worth talking about linearizat

We can now specify 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; i.e. the interval between $lockObtained.l.A^k$ and $lockUnlocked.l.A^k$

- Mutual Exclusion: This is specified by $CS_A^k \rightarrow CS_B^j$ or $CS_B^j \rightarrow CS_A^k$ 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:

```
1      Mutex = lockObtained.l?t → lockUnlocked.l.t → Mutex
2      Mutex ⊑_T  X \ (Σ − [|lockObtained.l, lockUnlocked.l|])
```

- Deadlock Freedom: This specifies that if some thread attempts to acquire the lock then some thread will succeed in acquiring the lock[1]. This does allow a CSP deadlock if no thread is attempting to acquire the lock, but only if the following holds: format this

```
1      ∀(s, ref )∈ failures (P) . ref  = Σ ⇒ #(s↑{|callLock|}) = #(s↑{|lockObtained|})
```

This can be captured by the following failures refinement on lock l with the set of all threads called ThreadID. This process can non-deterministically deadlock when no threads are attempting to obtain the lock and otherwise ensures that threads are either attempting to obtain the lock or obtaining the lock

```
1      AcquireLock({}) = callLock.l?t → AcquireLock({t}) ⊓ STOP
2      AcquireLock(ts)  = callLock.l ?t:{ThreadID \ ts} → AcquireLock(union(ts, {t}))
```

```
3              □ lockObtained.l?t:ts → AcquireLock(diff(ts, {t}))
4
5      AcquireLock({}) ⊑_F  X \ (Σ − [|callLock.l.t, lockObtained.l|])
6
7
```

- Starvation Freedom: Every thread that attempts to acquire the lock eventually succeeds Definition of starvation freedom

1.3   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:

```
1    < callLock.l.A> ⌢ < callLock.l.B, lockObtained.l.B, lockUnlocked.l.B>^ω
```

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.

1.4   Test-and-Set Lock

The Test-and-Set (TAS) lock implementation is based on using an AtomicBoolean called state to capture whether the lock is currently held; true meaning that some thread holds the lock and false meaning that the lock is currently free. The full Scala code

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

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

```scala
1    import java.util.concurrent.atomic.AtomicBoolean
2
3    /** A lock based upon the test−and−set operation
4      * Based on Herlihy & Shavit, Chapter 7. */
5    class TASLock extends Lock{
6      /** The state of the lock: true represents locked */
7      private val state = new AtomicBoolean(false)
8
9      /** Acquire the Lock */
10     def lock = while(state.getAndSet(true)){ }
11
12     /** Release the Lock */
13     def unlock = state.set(false)
14   }
15
```

Figure 3: Test-and-set lock from [2]

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.

1.4.1 Modelling with CSP

Firstly, in order to model the TAS 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 represent the state variable from the Scala implementation by the following
CSP State = Var(false, get, set, getAndSet), with a communication of any of the channels
equivalent to a thread calling that operation in Scala. We use false to indicate that no

```
1    Var(value, get, set, getAndSet) =
2      get ?__!value → Var(value, get, set, getAndSet)
3      □ set ?__?value' → Var(value', get, set, getAndSet)
4      □ getAndSet ?__!value ?value' → Var(value', get, set, getAndSet)
5
```

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

thread holds the lock initially.

We can then model the operations of the lock itself. The Unlock procedure is quite trivial, simply setting the state to false and then terminating.

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

The Lock procedure is also trivial, with the thread just communicating over getAndSet. The procedure terminates once the getAndSet communicates that the original value of the statevariable was false, indicating that thread t now holds the lock.

```
1    −− while(state.getAndSet(true)){ }
2    Lock(t) = getAndSet.t?v!True → if v = False then SKIP
3                                         else  Lock(t)
```

We finally model the threads that are attempting to obtain the lock by a process Thread(x), where x is the identity of the thead. Each thread can either choose to either terminate or obtain the lock, release the lock and repeat its choice; we use external choice here so that we can regulate the behaviour of the threads when analysing the lock's properties.

```
1    Thread(t) = callLock.L.0.t → Lock(t); Unlock(t); Thread(t)
2                 □ end.t → SKIP
```

1.4.2 Analysis

We will firstly examine whether this model fulfils the properties defined previously

1.5 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 lock and false meaning that the lock is currently free. The full Scala code can be seen in Figure 5. 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

```
1    import java.util.concurrent.atomic.AtomicBoolean
2
3    /** A lock based upon the test−and−set operation
4       * Based on Herlihy & Shavit, Chapter 7. */
5    class TTASLock extends Lock{
6      /** The state of the lock: true represents locked */
7      private val state = new AtomicBoolean(false)
8
9      /** Acquire the lock */
10     def lock =
11       do{
12         while(state.get()){ } // spin until state = false
13       } while(state.getAndSet(true)) // if state = true, retry
14
15     /** Release the lock */
16     def unlock = state.set(false)
17
18     /** Make one attempt to acquire the lock
19        * @return a Boolean indicating whether the attempt was successful */
20     def tryLock : Boolean = !state.get && !state.getAndSet(true)
21   }
22
23
```

Figure 5: Test-and-test-and-set lock from [1]

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?**

1.6   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

```
1   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.

```
1   Lock(t) =  getState.t?s → if s =  True then Lock(t)  −− while(state.get()){ }
2           else  gASState.t?v!True → if v =  False  then Holding(t)
3               −− do ... while(state.getAndSet(true))
4           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.

```
1   Holding(t) = Unlock(t)
2   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

```
1   AllThreads = ||| t :  ThreadID • NotHolding(t)
2   InternalChannels  = {|getState, setState ,  gASState|}
3   LockEvents = {|lockObtained, lockUnlocked, callLock|}
4   AllChannels = Union({InternalChannels,  LockEvents})
5   ActualSystem = (AllThreads [|InternalChannels|] State)
```

```
6    ActualSystemR = (ActualSystem [[gASState.t.False.True \ lockObtained.L.0.t,
       setState.t.False \ lockUnlocked.L.0.t | t ← ThreadID]]) \ InternalChannels
```

1.7   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 dead-
   lock 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.

   ```
   1    assert  ActualSystem :[divergence  free ]
   2    assert  ActualSystemR :[divergence  free ]
   3    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.

   ```
   1    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

   ```
   1     CheckNoDiv = gASState?t!False.True → STOP
   2              □ getState?t._ → CheckNoDiv
   3   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.

   ```
   1    assert  LockSpec(L.0, {}, ThreadID)  ⊑_T  ActualSystemR
   2    assert  ActualSystemR  ⊑_T  LockSpec(L.0, {}, ThreadID)
   ```

8

1.8 Queue Lock

1.9 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.

[2] Hanno Nickau and Gavin Lowe. *Spin Locks and Contention.* 2023. URL: https://www.cs.ox.ac.uk/teaching/courses/2023-2024/cads/.

```scala
1    import ox.cads.util.ThreadID
2    import java.util.concurrent.atomic.{AtomicInteger,AtomicIntegerArray}
3
4    /** A lock using an array to store waiting threads in a queue.
5      * Based on Herlihy & Shavit, Section 7.5.1.
6      * @param capacity the number of workers who can use the lock. */
7    class ArrayQueueLock(capacity: Int) extends Lock{
8      // ThreadLocal variable to record the slot for this thread
9      private val mySlotIndex =
10        new ThreadLocal[Int]{ override def initialValue = 0 }
11     //println("AQL")
12
13     private val padding = 16 // # words per cache line
14     private val size  = padding*capacity // # entries in the flag array
15
16     // flag(i) is set to Go to indicate that the thread waiting on it can
17     // proceed.  We only use slots that are a multiple of padding, to avoid
18     // false sharing.
19     private val flags  = new AtomicIntegerArray(size)
20     private val Go = 1; private val Wait = 0
21     flags.set(0, Go) // other flags initially equal Wait
22
23     private val tail  = new AtomicInteger(0) // the next free slot / padding
24
25     // Array, indexed by ThreadIDs, where threads store the index of their flag
26     // in flags.
27     //private val slotIndices = new Array[Int](capacity)
28
29     def lock = {
30       val slot  = (tail.getAndIncrement * padding) % size
31       // slotIndices(ThreadID.get) = slot
32       mySlotIndex.set(slot)
33       while(flags.get(slot) == Wait){ } // spin on flag(slot)
34     }
35
36     def unlock = {
37       val slot  = mySlotIndex.get // slotIndices(ThreadID.get)
38       flags.set(slot, Wait) // anyone waiting here must wait
39       flags.set((slot+padding)%size, Go) // next thread can progress
40     }
41
42     // I don't think tryLock can be implemented.
43     def tryLock : Boolean = ???
44   }
45
```

Figure 6: A lock implementation storing a queue of waiting threads from [1]