4yp

Thomas Aston

November 22, 2023

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. 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 some thread will eventually succeed in acquiring the lock, unless a thread holds the lock indefinitely. *Starvation freedom* is a liveness requirement that any thread that tries to gain the lock will eventually succeed; by contrast deadlock freedom allows one thread to never obtain the lock as long as others complete an infinite number of critical sections. Other requirements/useful properties of locks will be explored later.

1.1 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 hold 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:

- 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

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

Figure 1: A Scala interface for a simple lock

- end.t : The thread will make no further calls to the lock $\boxed{\text{Necessary?}}$

  $\boxed{\text{Worth talking about linearization?}}$

We can now specify ideal properties of locks using these channels:

- Mutual Exclusion: This specifies that at most one thread may be in its critical section at any one time; i.e. that once thread A obtains the lock, no other thread can obtain the lock until thread A unlocks. We can therefore deduce that a lock l with model X satisfies the trace refinement:

```
  Mutex = lockObtained.l?t → lockUnlocked.l.t → Mutex
  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 $\boxed{\text{Need to explain earlier}}$ if no thread is attempting to acquire the lock, but only if the following holds: $\boxed{\text{format this}}$

```
  ∀(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 if a thread attempts to acquire the lock then some thread obtains the lock

```
  AcquireLock(l, {}, TS) =
    end?t:TS → AcquireLock(l, {}, diff (TS, {t}))
    □ callLock.l?t:TS → AcquireLock(l, {t}, TS)
  AcquireLock(l, ts, TS) =
    end?t:(diff (TS, ts)) → AcquireLock(l, ts, diff (TS, {t}))
    □ callLock.l?t:(diff (TS, ts)) → AcquireLock(l, union(ts, {t}), TS)
    □ lockObtained.l?t:ts → AcquireLock(l, diff (ts, {t}), TS)
```

```
AcquireLock(l, {}, ThreadID) ⊑_F
    X \ (Σ − {|callLock.l.t, lockObtained.l, end|})
```

- Starvation Freedom: Every thread that attempts to acquire the lock eventually succeeds $\boxed{\text{Definition of starvation freedom}}$

1.2 A simple lock specification

$\boxed{\text{utility of this? Should cut down}}$

We define a specification for a lock

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 specifies deadlock-freedom since it can always communicate a callLock unless either ts == TS (in which causes some thread can communicate a lockObtained, followed later by a lockUnlocked) or TS = {} (where all threads have 'terminated' via exit and hence is deadlock-free since no threads will attempt to obtain the lock). 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 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, end} we can use the specification in more complex systems, reducing the size of the systems produced by FDR and hence allowing us to test larger cases than would otherwise be possible.

1.3 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 AtomicBoolean, has atomic get and set operations to read and write values respectively. In the TAS lock we also use the getAndSet operation which atomically sets the Boolean to a new value

```
LockSpec(l, ts, TS) =
  end?t:diff (TS, ts) → LockSpec(l, ts, diff (TS, t))
  □ 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) =
  end?t2:diff (TS, union(ts, t)) → LockSpecObtained(t, l, ts, diff (TS, t2))
  □ 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

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

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

  /** Acquire the Lock */
  def lock = while(state.getAndSet(true)){ }

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

```

Figure 3: Test-and-set lock from [2] Need to figure out figure placement

and returns the old value. 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.

1.3.1 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

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

```

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

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 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.

```
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; a getAndSet.t.False.True event in a trace can be linearized as the point at which the thread t obtains the lock.

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

We 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.

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

Finally, we construct the overall system as shown below. We first synchronise all the threads over the get, set and getAndSet channels with the State process. 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 to produce ActualSystemR. Finally, to obtain a system that only visibly communicates the four previously identified events, we hide the interal channels of the lock to produce.

```
-- All initially  do not hold the lock
AllThreads = ||| t : ThreadID • Thread(t)
-- Allow all threads to peform actions on the state variable
ActualSystem = (AllThreads [|InternalChannels|] State)
-- Rename lock acquisition and releasing and hide internal events
ActualSystemR = (ActualSystem
                  [[getAndSet.t.False.True \ lockObtained.L.0.t,
                  set.t.False \ lockUnlocked.L.0.t | t <- ThreadID]])
ActualSystemRExtDiv = ActualSystemR \ InternalChannels
```

1.3.2 Analysis

We will firstly examine whether this model fulfils the properties defined previously. The mutual exclusion and deadlock freedom tests from section 1.1 pass and the model does not diverge before it is first held; these were all expected results $\boxed{\text{Livelock?}}$. The TAS lock is also equivalent under traces with the simple lock specification earlier. However, once the lock is held, a thread attempting to obtain the lock can perform an infinite number of getAndSet operations; an example trace of this behaviour where T.0 obtains the lock follows

```
< callLock.L.0.T.0, callLock.L.0.T.1, getAndSet.T.0.False.True> ^ <
  getAndSet.T.1.True.True>^ω
```

This is problematic as 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 state from the caches, 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. $\boxed{\text{Level of detail regarding memory buses, performance, caching etc}}$

1.4 Test-and-Test-and-Set Lock

The Test-and-Test-and-Set (TTAS) lock makes use of this improvement, whilst otherwise remaining very similar to the TAS lock. The sole change is to the lock function, as can be seen in Figure 5. The TTAS lock can still produce traces with an unbound number of consecutive operations, but these are now get operations instead of getAndSet operations. This results in significant performance improvements as only the first get call by a thread can result in a cache miss; all of the further get operations are cache hits until the lock is released by some other thread.

Whereas the TAS lock can have an unbounded number of getAndSet operations for each time the lock is obtained, the TTAS lock has performs at most one getAndSet

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

```

Figure 5: Test-and-test-and-set lock from [2]

operation per thread when the lock becomes available. This is the case when all threads trying to obtain the lock read get.t.False before the first getAndSet is performed; all threads know the lock was not held so try to obtain it via a getAndSet, with only one of the threads succeeding. We now have a linear bound on the number of unsuccessful getAndSet operations, resulting in much more efficient usage of caching and shared memory. $\boxed{\text{Show bound using a CSP regulator function?}}$

1.5 Tree lock

$\boxed{\text{Might be worth introducing Peterson lock for starvation freedom both earlier and here?}}$

Suppose we have an implementation of a lock that works for upto n threads and now wanted to extend this to work with more threads trying to obtain a single lock. One approach to solving this problem is to arrange a number of the n thread locks into a tree structure. The threads are assigned a leaf node and, once they have obtained that lock, they progress up the tree obtaining the next lock and so on. Once the thread reaches the root of the tree and has obtained the 'root lock' and hence holds the lock; to unlock, the thread unlocks the root lock and progressively unlocks all the locks it held until it is back at the leaf lock. To consider a simple case where n = 2 and NThreads = 8 we have the following structure: $\boxed{\text{Draw properly}}$

Here, for T.5.1 to obtain the root lock L.0, it must first obtain L.5 then L.2 then it can attempt to lock L.0. If it holds L.0 then all of the other 7 threads can't enter their critical section. Once T.5.1 wants to release the lock it unlocks L.0, then unlocks L.2 then finally unlocks L.5. T.5.1 can now either terminate or try to reobtain the root lock.

1.5.1 Modelling

We model the tree structure so that the structure of the tree is independent from the implementation of the individual locks; this will allow us to examine how different the different properties of the locks affect the overall structure.

```
                 L.0
               /     \
              /       \
             /         \
            /           \
          L.1           L.2
         /  \           /  \
        /    \         /    \
       /      \       /      \
      L.3     L.4    L.5     L.6
     /  |     / |    |  \    |  \
    /   |    /  |    |   \   |   \
T.3.1   | T.4.1 |    | T.5.2 |   T.6.2
     T.3.2   T.4.2    T.5.1   T.6.1
```

Figure 6: An example of the tree lock with 2 threads per lock and 8 threads

For convenience, we shall call the threads T.{y}.{z} where y is the leaf lock that the thread will first try and obtain and z is just an index over the threads starting at that leaf lock. Since the thread should be agnostic to the implementation of the lock, the thread initially either exits or calls LockTree to try and obtain the lock. Once the thread has obtained the lock, it then releases it by a call to UnlockTree. Both LockTree and UnlockTree are recursively defined, traversing their way up and down the tree respectively before terminating once they have locked the root lock or unlocked a leaf lock respectively. nextUnlock(x, y) is used as a helper function to generate the next lock to release

```
nextUnlock(x, y) = if (y / 2 > x) then nextUnlock(x, y/2) else L.y

LockTree(t, L.0) = SKIP
LockTree(T.y.z, L.x) = Lock(T.y.z, L.x); LockTree(T.x.y, L.((x−1)/2))

UnlockTree(T.y.z, L.x) = if (x = y) then Unlock(T.y.z, L.x); SKIP
                         else Unlock(T.y.z, L.y);
                              UnlockTree(T.y.z, nextUnlock(x, y))

Thread(T.y.z) = callLock.L.y.T.y.z → LockTree(T.y.z, L.y);
                                     UnlockTree(T.y.z, L.0);
                                     Thread(T.y.z)
                □ end.T.y.z → SKIP
```

We can use the lockObtained and lockUnlocked events of the root L.0 lock as the points where a given thread obtains and releases the lock and the callLock events of the leaf nodes to represent when a thread tries to access the lock. To check that this lock fulfils the required properties we will hide all internal locking events; we do not care how the lock functions internally as long as it follows the required specifications. We will also rename the the three above events to appear to occur on the lock L.0. The system is therefore constructed as follows:

```
-- Initialise threads to not hold their locks
AllThreads = ||| (T.y.z) : ThreadID • Thread(T.y.z)
-- Initialise all the locks
AllLocks = ||| (L.x) : LockID • LockSystem(L.x)
-- Sychronise the threads with all the locks
TreeInternal  = (AllThreads [|LockEvents|] AllLocks)
-- Hide unwanted lock events from internal locks
TreeHidden = TreeInternal  \  Union(diff({|lockObtained|}, {|lockObtained.L.0|}),
                                    diff ({|unlockedLock|}, {|unlockedLock.L.0|}),
                                    diff ({|callLock|},
                                          {callLock.L.y.T.y.z | T.y.z <- ThreadID}))
-- Rename linearization points so all refer to L.0
TreeInternalRenamed = TreeHidden [[callLock.L.x.t \ callLock.L.0.t]]
```

Add analysis of beheviours with different node locks etc

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. *Concurrent Algorithms and Data Structures Lecture Notes*. 2023. URL: https://www.cs.ox.ac.uk/teaching/courses/2023-2024/cads/.