Formalizing Resource Ownership Semantics of Spinlocks with the Coq Proof Assistant

Sebastian Luis S. Ortiz[1], Justin Gabriel R. Enriquez[2], Henry N. Adorna[3], and Alfonso B. Labao[4]

Department of Computer Science
College of Engineering
University of the Philippines - Diliman
Quezon City, NCR, Philippines
[1]ssortiz@up.edu.ph
[2]jrenriquez4@up.edu.ph
[3]hnadorna@up.edu.ph
[4]ablabao@up.edu.ph

Abstract. As the transistor count in modern central processing units (CPUs) reaches the physical limits of Moore's Law, the parallelization of software-based compute has become the next step in maximizing the instruction-level and thread-level parallelism of modern multi-core architectures (e.g., advanced superscalar pipelined processors). With race conditions, memory corruption, and access violations afoot, systems-level programmers need fast, efficient, and correct concurrency primitives upon which to build concurrent data structures. One such primitive is the lock. This paper explores how the Coq proof assistant can be used to formalize and prove the correctness of these primitives. The case study focuses on the simplest implementation of a mutual exclusion lock: the spinlock. It focuses on definining the core functions that define the spinlock arrangement, albeit with some strong assumptions regarding the acquirement of locks. It is shown that a model based on resource ownership semantics is simpler to fathom and prove than tree-based alternatives that model non-determinism as branched and interleaved thread execution.

Keywords: Coq proof assistant, separation logic, resource ownership semantics, concurrency, multithreading, mutex, spinlock

1 Introduction

1.1 Motivation

As the transistor count in modern central processing units (CPUs) reaches the physical limits of Moore's Law, computer engineers now reach for the parallelization of computation to squeeze more performance out of computer chips. Multi-core architectures now dominate modern computing by necessity with the advent of advanced superscalar pipelined processors, where instruction-level parallelism and thread-level parallelism enable multitudes of work to be done within a single cycle.

J. Caro et al. (eds.), Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2024) , Atlantis Highlights in Computer Sciences 23,
https://doi.org/10.2991/978-94-6463-684-0_11

The onus of computation speed is thus no longer in the hands of the hardware engineers but on the software engineers who now have to shift from the simpler sequential single-core paradigm to the massively parallel multi-core paradigm. With race conditions, memory corruption, and access violations afoot, systems-level programmers need fast, efficient, and (most importantly!) *correct* concurrency primitives upon which to build concurrent data structures. One such core primitive is the **lock**.

The current paper however represents a work in progress and there is room for several improvements in model complexity. For instance, the paper makes the strong assumption that all threads eventually acquire the lock, and the number of locks is limited to a single one, which does not accurately reflect most real world implementations in which several locks are involved. In addition, the paper does not involve separation from task-side behavior or memory state.

1.2 The Mutex

A **lock** is a concurrency primitive that ensures a resource can only be accessed by a particular subset of CPU cores. The most basic example is the **mutual exclusion (mutex) lock**, which (as its name suggests) ensures that only a single core can access a resource at any time. Other variations of the mutex lock include **shared locks**, which allow shared *read-only* access to a resource and fall back to exclusive *read-and-write* access should the need arise. In other words, the shared lock optimizes read-many-but-write-once workloads. It is for this reason that these are also known as **read-write locks**.

1.3 The Coq Proof Assistant

This paper sets out to prove the correctness of the simplest implementation of a mutex lock, the **spinlock**, using the Coq proof assistant. Coq programs formally verify *user-defined* systems of mathematical definitions from which properties and theorems can be proven. A successful proof involves the use of built-in "tactics" that manipulate computer representations of hypotheses and propositions until these adhere to the form being asserted. Indeed, Guevers [4] notes that proof assistants facilitate "not so much the computing (numerical or symbolical) aspect of mathematics but the aspects of *proving* and *defining*."

Over the years, Coq has accumulated an extensively proven track record in many formal verification projects. Case studies and success stories include the verification of type system soundness [8], concurrent separation logic frameworks [2, 3, 6], machine-checked proofs in group theory (e.g., the Feit-Thompson Odd Order Theorem [5]), and compiler verification for mission-critical environments (e.g., the CompCert C compiler [7]).

The goal of this paper then is to define and establish a formal system in which certain properties can be proven about the safety and correctness of the spinlock implementation. The rest of the paper is thus outlined as follows. Section 2 details the core mathematical data types used in modelling the spinlock in Coq. Section 3 defines the basic operations of the spinlock: `acquire`, `access`,

and `release`. With these foundations established, Section 4 finally attempts to prove the correctness of each operation in the simplified model by verifying mutual exclusion of memory access as the basis for race-free computation.

2 Data Types

Given Coq's preference for pure computation and deterministic functions, modelling concurrency in Coq (where impure computations and non-deterministic thread scheduling is inevitable) is a surprisingly onerous task. To prove the correctness of concurrent programs, one may resort to simulating all of the possible ways to interleave execution. Chappe et al. [1] model this exponentially branching behavior with *choice trees* or *CTrees* (inspired by *interaction trees* or *ITrees* by Xia et al. [9]), where "choices" in non-deterministic computing are encoded as branches of some execution history tree.

However, this paper follows a different approach proposed by Dietrich [2] and inspired by the "higher-order concurrent separation logic" of the Iris project [6] and Dockins et al. [3]. Instead of considering permutations of thread interleaving in the proof of correctness, the problem is reframed in terms of *resource ownership*. Without loss of generality, this paper considers ownership over memory locations. A thread *owns* a memory location if and only if it has exclusive read-and-write access to it. Proving the correctness of the spinlock thus boils down to verifying whether a thread can successfully *acquire* ownership, *access* memory locations it owns, and *release* exclusive access over resources it owns. This approach vastly simplifies the model and its corresponding proof in Section 4.

```
1 Variant Memory := Panic | Spin | Open | Exclusive (tid: nat).
```

Listing 1: basic model for representing the ownership states of a memory location

In Coq, the `Memory` variant type encodes the possible ownership states of a single memory location. This is a simplified model inspired by the *"Points-to" connective* in [2]. Instead of encoding $p \to q$ as the assertion that a thread *owns* (i.e., has exclusive access over) a memory location p pointing-to some value q, Listing 1 only concerns itself with the ownership status of that memory location p, paying no heed to its current value q.

The `Exclusive` variant encodes the state that the thread with ID `tid` *owns* a particular memory location `mem`. That is to say, thread `tid` has exclusive access over `mem`. By definition, no other thread can access `mem`. These invariants will be enforced later in Section 3.

The `Open` variant opposes the `Exclusive` variant. An `Open` memory location is owned by no threads. It is therefore legal for any one of them to *acquire* ownership over it at any time. However, if a thread fails to acquire ownership (e.g., another thread is already holding the lock), the `Spin` variant encodes the state where a thread must wait for the lock to be *released* before claiming ownership.

The `Spin` variant represents the state where a thread blocks itself (typically in an infinitely polling loop) until the lock released. To abstract the mechanics of an infinite loop, the `Spin` variant does not concern itself with the number of iterations. What is important is encoding the notion of spinning rather than the exact number of spins.

Finally, the `Panic` variant is a trap state that serves as a catch-all variant for supposedly impossible states that merit program termination. In practice, the `Panic` state abstracts the notion of thrown exceptions (as in Java and C++), thread panics (as in Rust), and process termination (as in C). The trap state is essential to the proof because it allows Coq to reject (i.e., `discriminate`) invalid execution paths.

3 Definitions

```
Require Import Bool.
Require Import PeanoNat.
```

Listing 2: a list of the imported modules from the Coq standard library used in the proof

Before defining the operations of the spinlock, it must be disclosed that some theorems and definitions are implied by the imported standard library modules in Listing 2. There are no other external third-party libraries used in this paper.

3.1 Acquiring Ownership

```
Definition acquire (tid: nat) (mem: Memory) :=
  match mem with
    | Spin | Open => Exclusive tid
    | Exclusive lock => if lock =? tid then Exclusive tid else Spin
    | _ => Panic
  end.
```

Listing 3: the acquire operation of the spinlock

To claim ownership over a memory location, a thread must first invoke the "acquire" operation. Listing 3 highlights the two arguments required for a thread to attempt acquiring a resource: a thread ID `tid` (represented as a natural number) and an instance `mem` of the `Memory` variant type from Listing 1 (representing the proof of current ownership). The `mem` can be thought of as a "token" or "brand" that asserts the current state of the memory location. For example,

the value `Exclusive 2` can be read as "thread 2 has exclusive access over this memory location".

The behavior of "acquire" mainly depends on the current state of `mem`. If the memory location is already `Open`, then there should be no issues in claiming ownership. In this case, the `acquire` operation simply returns the `Exclusive` variant with the `tid` as the marker for ownership. That is to say, thread `tid` now has `Exclusive` access over the memory location `mem`.

Interestingly, the behavior for the `Spin` variant is the same as that of the `Open` variant. This is a simplifying implementation detail under the assumption that all spinning threads must *eventually* acquire the lock, hence the same behavior with the `Open` variant. As mentioned in Section 2, the number of spins is irrelevant to the proof.

However, if a thread `tid` already has `Exclusive` access over the memory location `mem`, the `acquire` operation checks whether the provided `tid` is equal to the one branded in `mem`. If so, the thread that currently owns the resource is attempting to acquire that resource again. This should be a harmless operation, hence the no-op treatment. Otherwise, if the thread `tid` attempts to acquire the `mem` owned by another thread `lock`, then the thread `tid` must `Spin` indefinitely.

Finally, if `mem` happens to be any other `Memory` variant aside from those mentioned above, the `acquire` operation must emit the `Panic` variant. This should terminate the thread as noted in Section 2.

In summary, a successful (i.e., non-panicking) `acquire` operation only has two possible outcomes: (1) the thread claims ownership over the memory location or (2) the thread indefinitely spins until the lock is released. This fact will be useful later in the correctness proof for the `acquire` operation in Section 4.

3.2 Accessing the Resource

```
1  Definition access (tid: nat) (mem: Memory) :=
2    match mem with
3      | Exclusive lock => if tid =? lock then Exclusive tid else Panic
4      | _ => Panic
5    end.
```

Listing 4: the access operation of the spinlock

Assuming exclusive access over a memory location, the `access` operation is an abstraction over any read or write operations on the underlying resource. This is why there is only one successful (i.e., non-panicking) execution path for the `access` operation.

If the `mem` is already exclusively owned by the same thread `tid`, the `access` operation proceeds successfully. For any other variants, the `access` must emit a `Panic`. Similar to the `acquire` function in Section 3.1, Listing 4 encodes these

rules in Coq code as the `access` function, which accepts two arguments: the thread with ID `tid` and the current state of the memory location `mem`.

3.3 Releasing Ownership

```
Definition release (tid: nat) (mem: Memory) :=
  match mem with
    | Exclusive lock => if lock =? tid then Open else Panic
    | _ => Panic
  end.
```

Listing 5: the release operation of the spinlock

Similar to the `access` operation in Section 3.2, the `release` operation already assumes exclusive access over a memory location. Any other state is considered to be invalid.

Listing 5 defines the `release` operation in Coq, which enforces that a successful (i.e., non-panicking) `release` operation only occurs when the invoking thread is indeed the owner of the memory location in question. When this is the case, the `release` function emits the `Open` state, which signals a successful release of the lock. For any other `mem` state, the operation must `Panic`.

4 Lemmas and Theorems

To prove the correctness of the spinlock as a race-free concurrency primitive, this paper sets out to prove five important biconditional properties about the implementation in Section 3.

1. As exhaustively proven in Listing 6, the `acquire` operation enables mutually exclusive access.
2. As exhaustively proven in Listing 7, the `access` operation requires mutually exclusive access.
3. As exhaustively proven in Listing 8, the `release` operation requires mutually exclusive access.
4. As exhaustively proven in Listing 9, all threads must go through the `acquire-access-release` cycle (in that order only). That is to say, an `Open` memory location may be acquired so that it is now `Exclusive`; then it must be able to be accessed by the owner thread; and finally, it must be able to be released as `Open` by that same thread.
5. As exhaustively proven in Listing 10, given a locked resource, all threads that attempt to `acquire` that resource must spin indefinitely (thereby blocking further execution).

Listing 11 combines all of these lemmas into one composite master theorem that asserts the correctness of the spinlock as a valid race-free implementation of the mutex.

It is worth noting that all of the lemmas and theorems below are proven as biconditional assertions. This is done to further strengthen that only certain states and transitions (that enable race-free concurrency) are allowed by the implementation. The other invalid states are simply discarded as impossible at best and non-sense at worst (i.e., `discriminate`). The implementation thus aims to be correct by construction.

```
Lemma acquire_enables_exclusive:
  forall (tid: nat) (mem: Memory),
    acquire tid mem = Exclusive tid
    <-> mem = Open \/ mem = Spin \/ mem = Exclusive tid.
Proof.
  split.
  - intros. unfold acquire in H. destruct mem.
    * discriminate.
    * right. left. trivial.
    * left. trivial.
    * destruct (tid0 =? tid) eqn:H'.
      + apply Nat.eqb_eq in H'. right. right. f_equal. apply H'.
      + discriminate.
  - intros. destruct H.
    * rewrite -> H. reflexivity.
    * destruct H.
      + rewrite -> H. reflexivity.
      + rewrite -> H. simpl.
        destruct (tid =? tid) eqn:H'.
        -- trivial.
        -- apply Nat.eqb_neq in H'. contradiction.
Qed.
```

Listing 6: The `acquire` operation enables exclusive access.

```
Lemma access_requires_exclusive:
  forall (tid: nat) (mem: Memory),
    access tid mem = Exclusive tid <-> mem = Exclusive tid.
Proof.
  split.
  - intros. unfold access in H. destruct mem.
      + discriminate.
      + discriminate.
      + discriminate.
      + f_equal. symmetry. destruct (tid =? tid0) eqn:H'.
        * apply Nat.eqb_eq in H'. apply H'.
        * discriminate.
  - intros. rewrite -> H. simpl. destruct (tid =? tid) eqn:H'.
    + trivial.
    + apply Nat.eqb_neq in H'. contradiction.
Qed.
```

Listing 7: The `access` operation requires exclusive access.

```
Lemma release_requires_exclusive:
  forall (tid: nat) (mem: Memory),
    release tid mem = Open <-> mem = Exclusive tid.
Proof.
  split.
  - intros. unfold release in H. destruct mem.
    * discriminate.
    * discriminate.
    * discriminate.
    * f_equal. destruct (tid0 =? tid) eqn:H'.
      + apply Nat.eqb_eq. apply H'.
      + discriminate.
  - intros. rewrite -> H. simpl. destruct (tid =? tid) eqn:H'.
    * trivial.
    * apply Nat.eqb_neq in H'. contradiction.
Qed.
```

Listing 8: The `release` operation requires exclusive access.

```
Lemma acq_acc_rel_cycle:
  forall (tid: nat),
    release tid (access tid (acquire tid Open)) = Open.
Proof.
  intros. unfold release.
  destruct (acquire tid Open) eqn:H'.
  - unfold acquire in H'. discriminate.
  - unfold acquire in H'. discriminate.
  - unfold acquire in H'. discriminate.
  - simpl in H'. inversion H'.
    destruct access eqn:A'.
    * simpl in A'. destruct (tid0 =? tid0) eqn:T'.
      + discriminate.
      + apply Nat.eqb_neq in T'. contradiction.
    * simpl in A'. destruct (tid0 =? tid0) eqn:T'.
      + discriminate.
      + discriminate.
    * simpl in A'. destruct (tid0 =? tid0) eqn:T'.
      + discriminate.
      + discriminate.
    * simpl in A'. destruct (tid0 =? tid0) eqn:T'.
      + inversion A'. destruct (tid1 =? tid1) eqn:U'.
        -- trivial.
        -- apply Nat.eqb_neq in U'. contradiction.
      + discriminate.
Qed.
```

Listing 9: All threads must go through the `acquire-access-release` cycle (in that order).

```
Lemma other_thread_must_spin:
  forall (self other : nat),
    self <> other <-> acquire other (acquire self Open) = Spin.
Proof.
  split.
  - intros. simpl. destruct (self =? other) eqn:X.
    * apply Nat.eqb_eq in X. contradiction.
    * trivial.
  - intros. simpl in H. destruct (self =? other) eqn:X.
    * discriminate.
    * apply Nat.eqb_neq in X. apply X.
Qed.
```

Listing 10: Given a locked resource, all threads that attempt to `acquire` that resource must spin indefinitely (thereby blocking further execution).

```
Theorem spinlock_is_correct:
  forall (self other : nat) (mem: Memory),
    (acquire self mem = Exclusive self
      <-> mem = Open \/ mem = Spin \/ mem = Exclusive self)
    /\ (access self mem = Exclusive self <-> mem = Exclusive self)
    /\ (release self mem = Open <-> mem = Exclusive self)
    /\ (release self (access self (acquire self Open)) = Open)
    /\ (self <> other <-> acquire other (acquire self Open) = Spin).
Proof.
  intros. split.
  - apply acquire_enables_exclusive.
  - split.
    * apply access_requires_exclusive.
    * split.
        + apply release_requires_exclusive.
        + split.
            -- apply acq_acc_rel_cycle.
            -- apply other_thread_must_spin.
Qed.
```

Listing 11: the master theorem of correctness that invokes all previous lemmas

5 Conclusion

It is shown in this paper that an ownership-centric model to correctness proofs offers an effective framework for proving properties about the most basic implementation of a mutual exclusion lock: the spinlock. An alternate formal model may assert certain properties about all of the possible ways to interleave instruction execution. This requires a more powerful model that considers nondeterministic, recursive, and impure programs, namely a model such as the choice trees presented by Chappe et al. [1] or the interaction trees presented by Xia et al. [9].

For future work, as indicated in the Introduction, the paper's strong assumption that all threads eventually acquire the lock can be relaxed, relative to multiple number of locks. In addition, there has to be separation from task-side behavior or memory state. Exploration as well can be done in ownership semantics in the context of other concurrency primitives such as (but not limited to) semaphores, barriers, primitives, and queues. Here, the notion of "resource ownership" generalizes beyond the access exclusion of memory locations. That is to say, a "resource" can be any aspect of a system that requires some exclusivity in order to prove correctness.

For instance, the right to execute code can be considered a "resource" to which ownership is conferred. In the case of semaphores, correctness proofs can reframe internal counters as "permits" or "tickets" (i.e., resources!) over which individual threads claim ownership. When formalized and modelled correctly, the Coq proof assistant can be an invaluable tool for highly concurrent systems.

References

1. Chappe, N., He, P., Henrio, L., Zakowski, Y., Zdancewic, S.: Choice trees: Representing nondeterministic, recursive, and impure programs in coq. Proceedings of the ACM on Programming Languages **7**(POPL), 1770–1800 (2023). DOI 10.1145/3571254. URL http://dx.doi.org/10.1145/3571254
2. Dietrich, E.: A beginner guide to iris, coq and separation logic (2021). URL https://arxiv.org/abs/2105.12077
3. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: Z. Hu (ed.) Programming Languages and Systems, pp. 161–177. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
4. Geuvers, H.: Proof assistants: History, ideas and future. Sadhana **34**(1), 3–25 (2009). DOI 10.1007/s12046-009-0001-5. URL https://doi.org/10.1007/s12046-009-0001-5
5. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: A Machine-Checked Proof of the Odd Order Theorem. In: S. Blazy, C. Paulin, D. Pichardie (eds.) ITP 2013, 4th Conference on Interactive Theorem Proving, *LNCS*, vol. 7998, pp. 163–179. Springer, Rennes, France (2013). DOI 10.1007/978-3-642-39634-2_14. URL https://inria.hal.science/hal-00816699
6. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming **28**, e20 (2018). DOI 10.1017/S0956796818000151

7. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). DOI 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814
8. Timany, A., Krebbers, R., Dreyer, D., Birkedal, L.: A logical approach to type soundness. J. ACM (2024). DOI 10.1145/3676954. URL https://doi.org/10.1145/3676954. Just Accepted
9. Xia, L.y., Zakowski, Y., He, P., Hur, C.K., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees: representing recursive and impure programs in coq. Proc. ACM Program. Lang. **4**(POPL) (2019). DOI 10.1145/3371119. URL https://doi.org/10.1145/3371119