Formal Modelling and Verification of Spinlocks at Instruction Level

Leping Zhang[1], Qianying Zhang[1,2,*], Guohui Wang[1,2], Zhiping Shi[1,3], Minhua Wu[1], *and* Yong Guan[1,4]
[1]*College of Information Engineering, Capital Normal University, Beijing, China*
[2]*Beijing Engineering Research Center of High Reliable Embedded System, Beijing, China*
[3]*Beijing Key Laboratory of Electronic System Reliability and Prognostics, Beijing, China*
[4]*International Science and Technology Cooperation Base of Electronic System Reliability and Mathematical Interdisciplinary, Beijing, China*
644496424@qq.com, {qyzhang, ghwang, shizp, wuminhua, guanyong}@cnu.edu.cn

***Abstract*—Spinlocks have been widely used as a solution for synchronous accesses to shared resources, and their correctness is critical to guarantee the consistency of concurrent processes. This paper presents formal models and machine-checked verification of the correctness of spinlocks at instruction level. We present the formal verification of two spinlocks, which are spinlocks implemented based on the ARM instructions and the x86 instructions, respectively. Our model formalizes the low-level instructions that are necessary to capture the execution of spinlocks, characterizes the processor hardware mechanisms related to each instruction, and considers the context switches on processors and two-level scheduling of processors and processes. We specify the correctness property of our models, that is, accesses of a critical section satisfy mutual exclusion, and verify that the models satisfy the property using the theorem prover Isabelle/HOL. With the verification experience, we give some suggestions on how to implement spinlock leveraging the ARM ISA.**

***Index Terms*—formal verification, spinlock, mutual exclusion, context switch, instruction level**

I. Introduction

Concurrent processing is pervasive in computing, while one challenge in designing concurrent programs is coordinating access to shared resources by multiple processes, which can lead to data inconsistency. Spinlocks are a synchronization mechanism for concurrent processes to enforce mutually exclusive access to shared resources, which has been widely used in concurrency control and adopted by mainstream operating systems. Generally, an implementation of a spinlock needs hardware-supported atomic read-modify-write operations to achieve synchronization. Fortunately, hardware in most contemporary processor architectures provides such atomic instructions, such as the **ldrex** and **strex** instructions provided by the ARM architecture, and the **xchg** instruction provided by the Intel x86 architecture. Spinlocks have been studied for years to improve performance, and a variety of implementations for different applications and processor architectures have been proposed.

One important property for a spinlock is its correctness. To be specific, the correctness of an implementation of a spinlock is critical to maintaining the consistency of processes operating concurrently, and thus ensure the consistency and correctness of the overall system. Therefore, it is necessary to verify the correctness property of spinlocks. One of the challenges for modelling and verification of spinlocks is to abstract at an appropriate level that allows capturing all relevant behavior of execution of spinlocks. There are several formal models of spinlocks have been presented for verification [1]–[4]. However, most of these works model high-level language source code of spinlocks, while verification of spinlocks at this level has the following disadvantages.

Firstly, verified properties of high-level language source code are not directly guaranteed for low-level instructions that are executed on a processor. This disadvantage comes from the presence of compilation, during which execution sequences of instructions on a processor can substantially differ from the source code. Although formally verified compilers [5]–[7] guarantee that compiled instructions preserve certain properties of the source code, there is still another disadvantage for source code-level verification. Secondly, hardware states which affect instruction execution and are crucial in the verification of programs, such as processor status, are not contained in the source code-level model. The successful execution of an instruction is determined by processor status, and will further change states of a processor. Therefore, it is necessary to consider hardware states when formally modelling a program, while low-level hardware states are usually not contained in source code-level model.

To solve the above problems, this paper formally models spinlocks at the instruction level, and mechanically verify the correctness of spinlocks which are implemented based on different Instruction Set Architectures (ISAs). We model the implementation of each spinlock as a state transition system, whose states contain process states and processor states, and transitions of states of the system are caused by instruction executions and context switches. The correctness property which describes mutual exclusion is defined on all possible states of the transitions of the system, and the verification is performed in a theorem prover Isabelle/HOL [8]. We verify two implementations of spinlocks, which leverage commonly used atomic instructions of two widespread processor ar-

*Corresponding author: Qianying Zhang, Capital Normal University, No.105, West 3rd Ring North Road, Haidian District, Beijing, China.

2640-0715/19/$31.00 ©2019 IEEE
DOI 10.1109/APSEC48747.2019.00055

chitectures: one implemented based on the **ldrex** and **strex** instructions provided by the ARM architecture [9], and the other implemented based on the **xchg** instruction provided by the Intel x86 architecture [10]. The verification results show that the two implementations possess correctness property, which demonstrates that concurrent processes mutually exclusive access to shared resources, i.e., there is at most one process access to shared resources at any time. During verification, we obtain experience on atomic instructions and hardware mechanisms of the ARM architecture related to the spinlock implementation and give some suggestions on how to implement spinlock leveraging the ARM ISA.

The detailed contributions are as follows.

1) We propose formal models for verification of spinlocks at instruction level and model two spinlocks, which are implemented based on the ARM instructions and the Intel x86 instructions, respectively. Our model characterizes hardware states related to each instruction, formalizes low-level instructions to capture the atomicity of instruction executions and considers context switches on processors.
2) We mechanically verify the correctness of spinlocks using Isabelle/HOL. The correctness property is defined as that there is no state that two processes are executing in the critical section at the same time, which is a mutual exclusion property. We prove that the spinlocks being verified satisfy the correctness. In addition, we give an approach to finding invariants in the verification of spinlocks.
3) In the verification of spinlocks implemented based on the ARM ISA, we take two-level scheduling of processors and processes into account and obtain experience on hardware mechanisms of the ARM architecture related to implementing spinlock. Therefore, we give some suggestions on how to implement spinlock leveraging the ARM ISA and run spinlock on the ARM processor.

II. Preliminaries

A. Synchronization primitives provided by the ARM processor

This section describes the synchronization primitives provided by the ARM processor, including atomic instructions, exclusive monitors to assist these instructions, and a clear instruction to support synchronization.

Since ARMv6, ARM replaces the **swp** instruction with synchronous primitives to achieve synchronization. The synchronization primitives are implemented by a pair of instructions Load-Exclusive/Store-Exclusive which are designed as four pairs of instructions according to the machine word length. For example, **ldrex/strex** instruction pair is one machine word length. Without loss of generality, the following describes the **ldrex/strex** as the representative of the Load-Exclusive/Store-Exclusive instruction series to introduce the method of exclusive access to shared resources and the modelling and verification of spinlock of the ARM architecture. Both instructions split the operation of atomically updating memory into two separate steps. The **ldrex** instruction reads a word from shared resources, and the corresponding **strex** instruction writes a word into shared memory. Together they provide atomic updates in conjunction with exclusive monitors that track exclusive memory accesses.

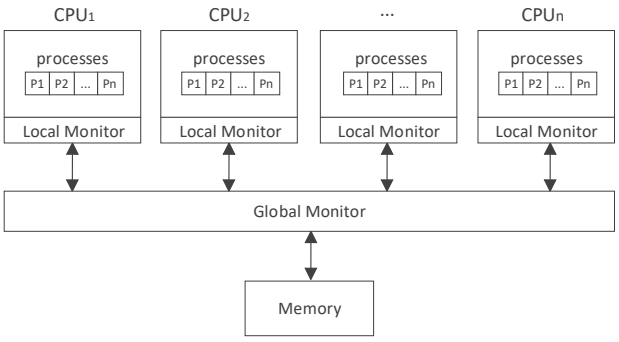

Fig. 1. Local and global monitors in an ARM multiprocessor architecture

The exclusive monitors are simple state machines and each monitor has two possible states, *open* and *exclusive*. To support synchronization between processors, a system must implement two sets of monitors, local and global. An **ldrex** instruction updates the monitors to the *exclusive* state and the corresponding **strex** operation accesses the monitors to determine whether it can complete successfully. As shown in Fig. 1, the architecture has multiple CPUs, each with a local monitor which is implemented to tag an address for its CPU. It also has one global monitor which can tag an address for each CPU. When a process performs the **ldrex** instruction on address x, the local monitor of the corresponding CPU tags x for exclusive use, and the global monitor tags x for exclusive use by that CPU. If both the local monitor of that CPU and the global monitor are in the *exclusive* state, the **strex** instruction will execute successfully.

The clear instruction, Clear-Exclusive(**clrex**), clears the local record of the executing processor that an address has had a request for exclusive access. After a context switch, the software executes the **clrex** instruction to ensure that the local monitor is in the *open* state.

B. Atomic instructions provided by the x86 processor

Similarly, the x86 processor also provides some atomic instructions for process synchronization, such as several bit instructions (like **bts**) and exchange instructions (like **xchg**). The **xchg** instruction is a common instruction used for implementing spinlocks. This paper takes the **xchg** instruction as an example and models and verifies a spinlock implemented based on this instruction.

The **xchg** instruction exchanges the contents of two operands. When a processor uses the **xchg** instruction to handle memory operands, its LOCK signal is automatically asserted. Therefore, this instruction is extremely useful for implementing a semaphore or mutex and is typically used

as an atomic operation to implement a spinlock for process synchronization.

C. Isabelle/HOL

Our verification of spinlocks is machine-checked, and performed in the interactive theorem prover Isabelle/HOL [8]. This section introduces Isabelle/HOL syntax used in this paper. In Isabelle/HOL, **locale** is used to define environment variables and their relationships, and its simplest form consists of keyword **fixes** and **assumes**. The keyword **datatype** defines a recursive type. For example, the natural number type is defined recursively as **datatype** *nat = 0 | Suc nat*, where *Suc* represents a type constructor. We introduce a state type that can be expressed as **datatype** *state = State (dom:"agent ⇒ domain")*, where *dom* is the name of the parameter of *State*. The *agent* represents a process and the *domain* represents a region where the process is located. For a given state, we can get the region of a process via its identity. There is a very useful type **option**. The *None* in it is a new element added to the existing type *'a*, and Some is added to distinguish between None and all elements of *'a*. For any type *'a*, the value of type *'a* **option** is either *None* or *Some a* for *a* in *'a*. For a typical application: accessing a non-existent value in the array should return *None*. Besides, the update function *fun_upd f(a := b)* is a function whose value at **a** is **b** and whose values at other inputs are the same as that for **f**.

III. Formal model

This section introduces the formal model of spinlocks. We model two spinlocks based on the ARM ISA and the x86 ISA implementations and represent them as spinlock_ARM and spinlock_x86, respectively. In order to demonstrate the correctness of these spinlocks, we need to show that all the circumstances in which the program is running are correct. Therefore, we first model each spinlock as a state transition system which generally consists of states and transitions between states. The initial state should be given before a system runs, and the transitions represent the execution of the system. In our model, the initial state(called *Init*) is statically defined. Then the transitions between states are expressed as a transition relation(called *Next*) that indicates whether any two states are reachable. Combined with the *Init* and the *Next*, we can thus determine all reachable states. A state s is reachable if there is a sequence of states $s_0, \ldots, s_n$, where $s_0 \in$ *Init*, each s_i, s_{i+1} satisfies *Next*, and $s_n = s$. To describe the correctness of spinlocks, it is necessary to define a property that shows the expected characteristics. Then, when all the reachable states of the system do not violate this property, we determine that the spinlocks are correct.

Based on the above overview of modelling, we describe the details of the model slightly. First, we model the software and hardware states related to the execution of spinlocks as the system states. For example, for the spinlock_ARM, we take a lock of software states and a register, local monitors and a global monitor of hardware states into account. In addition, we ignore some minor factors such as the correctness of context recovery. Second, we model each instruction as an event that causes the state transition of the system and guarantees the atomicity of each instruction. The detailed contents of an event are expressed by the extracted states according to the instruction description. Third, for the properties of the correctness, we define the mutual exclusion, that is, there is no state in which two processes simultaneously access the critical section during system execution. In particular, for the spinlock_ARM, we add some constraints as an execution environment, i.e., a *system configuration* to limit the states and their relationships. For example, the number of processes is different from that for processors, but these processes must be mapped to the processors without crossing the identity boundary. These variables and relationships should be constrained by the system configuration. Moreover, we consider two-level scheduling of inter-processor scheduling and processor internal process scheduling under a multiprocessor architecture.

```
1  enter_region:
2      ldrex   r0   lock
3      cmp     r0   0
4      beq     enter_region
5      strex   r0   1   lock
6      cmp     r0   0
7      beq     enter_region
8  leave_region:
9      str     0    lock
```

Fig. 2. The implementation of the spinlock_ARM

In this section, we describe the implementation and the formal model of each spinlock, respectively. In detail, the following introduces states and initial state, events and transitions, and property. Since the methods of modelling are the same due to the similarity of spinlock structure, we focus on the spinlock_ARM and point out the differences between spinlock_ARM and spinlock_x86.

A. Spinlock_ARM

1) The implementation of the spinlock_ARM: The low-level code of the spinlock based on the synchronization primitives of the ARM architecture is shown in Fig. 2. Before a process enters the critical section, it must execute this code. Line 2 indicates the lock is read into the register **r0**. Line 3 compares the value of **r0** and 0, where 0 is a flag indicating that no process occupies the critical section and is opposite to 1 in line 5. If both values are the same, the process will execute the **strex** instruction. Otherwise, it returns the entry enter_region to re-execute the **ldrex** instruction. The **strex** instruction tries to set the lock to 1, and if successful, **r0** is set to 0. Otherwise, **r0** is set to 1 indicating the request fails. Line 6 means the **cmp** instruction compares the value of **r0** with 0, if they are equal the process will enter the critical section. Ultimately, when the process leaves the critical section, the atomic operation **str** in line 9 directly stores 0 to the lock memory, indicating that the lock is released.

2) The formal model of the spinlock_ARM: According to the implementation of the spinlock_ARM, we introduce the corresponding formal model in this section, which includes system configuration, states, events, and property.

a) System Configuration: A system configuration is the specific definition of the elements that prescribe what a system is composed of. In our model, we define some variables related to the spinlock_ARM mechanism and their relationships as the system configuration. In fact, these variables here are chosen because there are fixed relationships between them, and they are not affected by the system state transitions. Hence, we define these variables(using the keyword **locale**) separately from the subsequent *states*, and the steps of defining the system configuration are as follows. First, considering that each processor has a local monitor, i.e., the identity of a processor is the same as that for its local monitor. We define the same type for processors and local monitors as *monitor* which is synonymous with natural number type *nat*, and the type represents the identities of processors or local monitors. Second, with the definition of the *monitor*, we express the number of processors as *cpu_m::monitor* and the number of processes as *process_n::agent*, where *agent* is also synonymous with *nat*. In addition, to show that all processes should be assigned to these processors, we define an assignment function as *assign::agent* $\Rightarrow$ *monitor*. Here, whatever strategy the spinlock mechanism adopts to assign these processes, the *assign* function will always cover all possible situations. Finally, we describe the relationship that all processes must be assigned to the processors, that is, for an arbitrary process *i*, the identity (starting from 0) of the processor which process *i* is assigned in, is less than the total number of processors. And the definition is shown as follows.

$$assign\_in\_range : \forall i < process\_n.\ assign\ i < cpu\_m$$

b) States: We extract necessary states related to the spinlock_ARM, and the definition contains the following five parts, all of which are the parameters of states.

1) **Lock.** Obviously, there are two states 0 or 1 for the lock. We define the lock as the type of *bool* and the *True* in *bool* means that there is already a process in the critical section.
2) **Domain.** Considering that the verified property involves process regions such as the critical section, we first need to add a region tag as follows.
$$domain = dom1 \mid dom2 \mid dom3 \mid dom4 \mid domCS \mid dom5$$
In the beginning, all processes will initially be in the $dom1$ region. If a process executes the **ldrex** instruction successfully, it will reach the $dom2$ region. In order to get the region where any process is located, we add a function type as one of the parameters, i.e., *dom:agent* $\Rightarrow$ *domain*. For a given state we can get the region of a process by its identity.
3) **Register.** The processes save the lock value obtained from memory in the register. Considering that a context switch saves the old process's register value in a stack, this value will be restored to the register when the process is swapped in. Here, we assume that the recovery of the register value of each process is correct when a context switch occurs because this is irrelevant to the correctness of the spinlock. So we regard that each process has its own register, which is defined as *reg:agent* $\Rightarrow$ *register*, where the type of *register* is the same as that for the lock.
4) **Monitor.** The local monitor of each processor may be *exclusive* or *open*, which can be defined as a function from *monitor* to *bool*. We define the *exclusive* state as *True* in *bool*, that is, when a local monitor is in an *exclusive* state, the return value of the function should be *True*. On the other hand, since the global monitor is sole, we use a global variable as a parameter and consider its type as follows. When some different processors expect to get the lock, they will be tagged in turn by the global monitor as exclusive access to the address of the lock. Such that, the most recent processors are useful. Therefore, we view the type of the global variable as *monitor* type that just records the latest processor identity instead of using an array or a list to preserve the global states of all processors. Besides, the global monitor will not tag the identity of any processor when it is in the initial state or its tag cleared by the **clrex** instruction. Finally, we define it as a global optional type, i.e., *global_monitor:monitor option*.
5) **Context Switch.** In order to determine if a context switch has occurred on a processor, we add two functions type into the states. Considering the two-level scheduling of processors and processes, we first need to provide a processor identity to indicate which processor has a context switch, then determine if there is a process scheduling on this processor. We define the first function as *current:monitor* $\Rightarrow$ *agent option*. The function describes the current process on a processor and indicates that we can get the most recent process number by the processor number. In spite of this, it is not clear which one is the previous process of the current process. This is because only two states(from previous to current) which reflect the current process has executed an instruction, cannot show whether there is a context switch. Hence, we define the second function, *last* function, whose type is the same as that for *current*. It records the previous process of the current process for a processor so that we can compare the values of the two function in an arbitrary state. And if they are not equal, there is a context switch. The *last* function is updated by *current* function during state transitions. For example, for a current state *S*, a process p on processor i executes a step to reach next state *S′*, which can be expressed as *current S′ i = p, last S′ i = current S i*.

```
definition strex :: agent ⇒ state ⇒ state ⇒ bool where
  strex self S S′ ⟷
    dom S self = dom3 ∧ dom S′ = (dom S) (self := dom4) ∧
    (if (current S (assign self)) = Some self ∧
      local_monitor S (assign self) ∧
      global_monitor S = Some (assign self)
    then lock S' = True ∧ reg S' = (reg S)(self := False) ∧
      local_monitor S′ = (local_monitor S) (assign self := False) ∧
      global_monitor S′ = None
    else (if current S (assign self) ∧ Some self
        then local_monitor S′ = (local_monitor S)(assign self := False)
        else local_monitor S′ = local_monitor S) ∧
      lock S′ = lock S ∧ reg S′ = (reg S)(self := True) ∧
      global_monitor S′ = global_monitor S) ∧
    last S′ = (last S)(assign self := current S (assign self)) ∧
    current S′ = (current S) (assign self := Some self)
```

Fig. 3. The definition of the **strex** instruction

Hence, the state of the transition system is defined as follows.

$$\begin{aligned}&\textbf{datatype}\ state\ =\ State\ (lock : bool)\\&\quad(dom : agent \Rightarrow domain)\\&\quad(reg : agent \Rightarrow register)\\&\quad(local\_monitor : monitor \Rightarrow bool)\\&\quad(global\_monitor : monitor\ option)\\&\quad(current : monitor \Rightarrow agent\ option)\\&\quad(last : monitor \Rightarrow agent\ option)\end{aligned}$$

In addition, we give the definition *init_state* of the initial state of the system as follows.

$$\begin{aligned}&init\_state\ =\ State\ (False)(\lambda a.\ dom1)(\lambda a.\ False)\\&\quad(\lambda a.\ False)(None)(\lambda a.\ None)(\lambda a.\ None)\end{aligned}$$

c) Events: For the spinlock_ARM, we model all five instructions as five events, respectively. Each event carries three parameters, the process, the current state, the next state, and finally returns a value of *bool* type. For instance, we express an event e as *e::agent ⇒ state ⇒ state ⇒ bool*, indicating that for a process p, the current state S can reach the next state S' after p executes event e. The detailed contents are defined by the instruction implementations. Here, we take the **strex** instruction as an example and give its definition in Fig. 3.

According to these defined events, we can describe the transition relation, in other words, these events are executed to determine whether the transition from one state to the next can occur. In order to simplify the expression of state transition relation *Next*, an auxiliary definition *Next_P* is given to indicate whether there is an instruction for a specified process to transform the current state to the next state. The

```
1    enter_region:
2        move    reg    #1
3        xchg    reg    lock
4        cmp     reg    #0
5        jne     enter_region
6    leave_region:
7        move    lock   #0
```

Fig. 4. The implementation of the spinlock_x86

definition of *Next_P* is as follows.

$$\begin{aligned}&Next\_P\ self\ S\ S' \longleftrightarrow\\&\quad(\ ldrex\ self\ S\ S'\ \vee\ cmp1\ self\ S\ S'\ \vee\\&\quad\ strex\ self\ S\ S'\ \vee\ cmp2\ self\ S\ S'\ \vee\\&\quad\ enter\ self\ S\ S'\ \vee\ leave\ self\ S\ S'\ )\end{aligned}$$

That is, for the process *self*, as long as there is an instruction that can be executed successfully, the state S can be transformed to S'. With *Next_P*, we give the definition of the *Next*.

$$Next\ S\ S' \longleftrightarrow (\exists a.\ Next\_P\ a\ S\ S')$$

This indicates that whether the state S can reach S' is equivalent to whether there is a process to convert S into S'.

d) Property: Since each process needs to access the critical section with a mutual exclusion, we describe the mutual exclusion property to ensure the correctness of the spinlock. That is, in an arbitrary state S, two different processes $P1$ and $P2$ cannot be in the critical section at the same time, which can be expressed as follows.

$$\neg\ (dom\ S\ P1 = domCS\ \wedge\ dom\ S\ P2 = domCS)$$

B. Spinlock_x86

1) The implementation of the spinlock_x86: The implementation of the spinlock_x86 in [11] is shown in Fig.4, and the code shows that before entering the critical section, a process first sets the register **reg** to 1, then executes the **xchg** instruction that exchanges the lock whose initial value is 0 with the register values. Next, it compares the value of the register to 0 in line 3. If both are not equal, the process returns to program entry. Otherwise, it will enter the critical section. When a process completes access to the critical section, it sets lock to 0 by the **move** instruction.

2) The formal model of the spinlock_x86: Since the approach of modelling is similar to that for spinlock_ARM, this section only introduces the differences between spinlock_x86 and spinlock_ARM.

a) System Configuration: For the model of the spinlock_x86, we ignore the impact of processors, because the procedures of a process executing instructions in one processor are exactly the same as that for a process in another processor and there are no auxiliary hardware mechanisms to distinguish the processors for processes, which are different from the spinlock_ARM. So we take the process as an object without a processor, resulting in the absence of the allocation of

processes. This eliminates the need to define a part of variables and their relationships in advance. Therefore, we cancel the definition of the system configuration.

b) States: Secondly, according to the regions where a process is located after all instructions are executed, we change the region tag of the *states* defined as follows.

$$domain \ = \ dom1 \mid dom2 \mid dom3 \mid domCS \mid dom4$$

Moreover, we only need to consider the global lock, the process region, and the register value of each process, which simplifies the extraction of the auxiliary hardware mechanisms and the record of the processes before and after the context switch. It can be expressed as follows.

$$datatype\ state \ = \ State\ (lock : bool)$$
$$(dom : agent \Rightarrow domain)(reg : agent \Rightarrow register)$$

And the following is the definition of the initial state.

$$init\_state \ = \ State\ (False)(\lambda a.\ dom1)(\lambda a.\ False)$$

c) Events: Thirdly, we define five events according to the implementation of spinlock_x86. Then for the *Next*, the entire definition is the same as that for the spinlock_ARM, and we provide the auxiliary definition *Next_P* as follows.

$$\begin{aligned} Next\_P\ self\ S\ S' \longleftrightarrow\ & (\ move\ self\ S\ S' \vee \\ & xchg\ self\ S\ S'\ \vee cmp\ self\ S\ S' \vee \\ & enter\ self\ S\ S'\ \vee\ leave\ self\ S\ S'\ ) \end{aligned}$$

d) Property: Finally, both spinlocks satisfy the mutual exclusion property, we will not go into details here.

IV. Formal Verification

The overview of our verification is shown in Fig. 5. First, we represent each spinlock implementation as a state transition system and describe the expected mutual exclusion property, which are introduced in detail in Section III. In order to prove the system satisfy this property, we add a mediation, system invariants. Since the system satisfies the invariants, we can determine whether the system holds the property by judging whether the system invariants imply the property, and thus assert whether the implementation is correct. To ensure the correctness of the solved invariants, we provide several lemmas which enforce the initial state *Init* and the transition relation *Next* satisfy the invariants. In the end, we need to describe the property as a theorem and prove it in Isabelle/HOL.

In this section, we give the definitions including lemmas, invariants, and theorem. For the invariants, we solve each of them by hand and get rich experience, therefore, we also provide an analysis method for finding the system invariants due to their complexity.

A. Lemmas

For the correctness of the invariants, the state transition system needs to satisfy the following lemmas.

Lemma 1 (inv_init): The Init satisfies the system invariants.

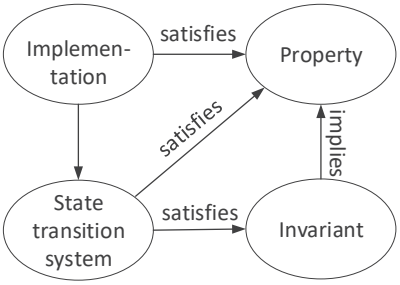

Fig. 5. The overview of our verification

$$\text{“}inv\ init\_state\text{”}$$

The lemma **inv_init** represents the initial state satisfies the invariants. And we can prove it by unfolding the definition of the invariants and the initial state.

Lemma 2 (inv_next): The Next satisfies the system invariants.

assumes “$inv\ S$” “$Next\ S\ S'$” **shows** “$inv\ S'$”

The lemma **inv_next** represents the state transitions satisfy the invariants, which needs to prove that any process satisfies the invariants after executing each single-step instruction. For this single-step execution, we need to define some detailed lemmas. For example, the transition of executing the **strex** instruction holds:

Lemma 3 (inv1_next): The transition of executing an strex event satisfies the system invariants.

assumes “$inv\ S$” “$strex\ self\ S\ S'$” **shows** “$inv\ S'$”

The lemma **inv1_next** indicates that for an arbitrary process *self* executing **strex** instruction, the new state S' also satisfies the system invariants when the current state S satisfies the invariants. In the next section, we just judge whether the invariant is correct through these lemmas. In fact, we can judge whether the implementations of these spinlocks exist some errors as well.

B. Invariants

For these models, we manually solve the corresponding invariants. To simplify the representation of invariants, we give an auxiliary definition based on a process as well, i.e., *inv_i::agent* ⇒ *state* ⇒ *bool* indicating that a process satisfies the invariants in any state. With the *inv_i*, the invariants hold:

$$\text{“}inv\ S\ \longleftrightarrow\ (\forall a.\ inv\_i\ a\ S)\text{”}$$

The state S satisfies system invariants only and if only any process satisfies the invariants in this state, so the problem is transformed into solving the contents of *inv_i*. The following is the analysis method for solving invariants.

Analysis Method. We can reverse analysis from bottom to top according to the mutual exclusion property. First, for a spinlock, we assume its implementation is correct, hence the mutual exclusion should be satisfied. So for the first step,

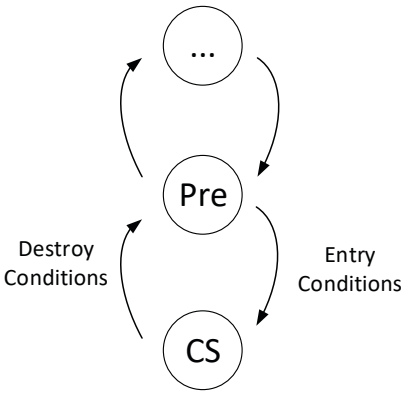

Fig. 6. The analysis method

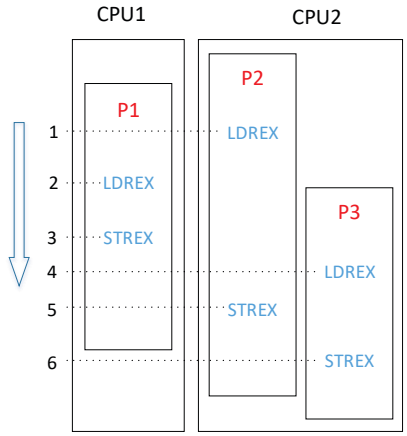

Fig. 7. A sequence of instructions in an ARM multiprocessor architecture

the logic formula of mutual exclusion that other processes are not in the critical section when a process has already been in the critical section should be expressed into *inv_i*. Then we add some additional formulae to ensure that each lemma in Section IV-A is workable. As shown in Fig. 6, considering all processes enter the critical section(CS) with mutual exclusion, we ensure that other processes can not enter it from the previous zone (Pre) when there is already a process in CS, so we destroy the entry conditions from Pre to CS, i.e., we must find the conditions that don't make it works then add it to the *inv_i*. Next, if a process in Pre has the conditions to enter CS, it will enter in some time. So the other processes cannot enter CS, yet. Actually, the invariants of CS and Pre are always similar at this point. Finally, to ensure all formulae in Pre can satisfy the lemmas, we observe the characteristics of the previous region of Pre upwards, repeatedly, and add them into the invariants until all lemmas are correct.

C. Theorem

The mutual exclusion property is expressed as a final theorem in Isabelle/HOL, which is shown as follows.

Theorem 1 (Mutual exclusion): In an arbitrary state S that satisfies the system invariants, two different processes cannot be in the critical section at the same time.

assumes "$inv\ S$" "$P1 \neq P2$"

shows "$\neg\ (dom\ S\ P1 = domCS \wedge dom\ S\ P2 = domCS)$"

Since the system invariants contain the final formula and the above lemmas are proved, we just prove it with some automated tactics.

V. Verification Results and Discussions

Verification Results. We propose a formal model and machine-checked verification of the correctness of spinlocks in Isabelle/HOL. The models and proofs are described by the structured language *Isar* whose style is easier for human understanding in Isabelle. For each spinlock we describe over 200 lines of scripts, including 11 definitions, 8 lemmas, and a final theorem. Finally, we prove that the implementations of both spinlocks are correct according to their models satisfying the mutual exclusion property. With the verification experience, we give some suggestions on how to implement spinlock leveraging the ARM ISAs and discuss as follows.

Discussions. During verification, we found the spinlock_ARM should be used carefully. For a uniprocessor architecture, switching context will not affect the final correctness even if the system does not enforce **clrex** instruction to clear the local monitor's exclusive access tag. However, in a multiprocessors architecture, the tag must be cleared to the *open* state after a context switch, otherwise, errors will occur. In Fig. 7, we show that the sequence can cause an error if not clear local monitors state in a multiprocessor architecture. There are two processors, CPU1 and CPU2, and we assume that the lock value is 0 in the beginning. Firstly, the process P2 executes the **ldrex** instruction, then the local monitor of CPU2 is set to the *exclusive* state, and the global monitor points to the CPU2. Next,the process P1 in CPU1 completes the **ldrex** and **strex** operations, and obviously, P1 successfully acquires the lock and enters the critical section. At this time, the lock value is 1, and the global monitor points to CPU1. Immediately, the process P3 executes the **ldrex** instruction, hence, the global monitor is changed back to CPU2, so the subsequent **strex** instruction of P2 is executed successfully and finally, both P1 and P2 are in the critical section. Therefore, for the use of the spinlock_ARM, the **clrex** instruction must be used to reset the local monitors to *open* state in multiprocessor architectures when a context switch occurs.

VI. Related Work

Verification of locks. In the area of verification of locks, related researchers have verified a variety of locks. Feng et al. [4] used the theorem prover Coq to verify a couple of synchronization primitives and a spinlock implemented based on them by extended separation logic, however, the spinlock is used for a uniprocessor x86 machine. In [3], Desnoyers et al. modeled and verified the spinlock implementation in the

PowerPC architecture, and the code is close to the C language. Some other locks like the reader-writer lock were verified in [12]–[14]. For example, Flanagan et al. [12] demonstrated the mutual exclusion property for a reader-writer lock implementation. In addition, some algorithms related to locks, which are generally implemented by high-level language like C were also verified [15]–[19], and the classical Peterson's algorithm has been verified by different tools and methods [20]–[22]. Cousineau et al. verified the Peterson's algorithm using TLA$^+$ in [20]. Paulson's inductive method [21] was used to prove that Peterson's algorithm satisfies the mutual exclusion property. This paper makes up for the lack of the above work, and formalizes and verifies the spinlocks at instruction level, and considers two-level scheduling of processors and processes, and models associated hardware details.

Verification methods and frameworks of concurrent programs. For the verification of concurrent programs, many methods and reasoning frameworks were proposed. The earliest is the OG method [23] based on Hoare logic in 1976, which extends the proof method of sequence programs with interference freedom to verify concurrent programs based on shared variables. In 2002, Prensa Nieto et al. formalized the OG and RG methods in Isabelle/HOL as Hoare-Parallel framework [24] for reasoning concurrent programs. Recently, Sidney Amani et al. proposed the COMPLX concurrency reasoning framework [25] that combined Hoare-Parallel and SIMPL, a generic imperative language. However, it is difficult to use these frameworks to verify the system that has many states. On the one hand, it is complicated to establish the model. On the other hand, it is difficult to find system invariants. In [20], Cousineau et al. described the TLA$^+$ proof for the Peterson's algorithm. It is quite intuitive and easily expressed in the modeling of the statement, which can bring us convenience. Therefore, we combine some of these modeling ideas with the analysis method we presented for solving invariants, then solve the above problems and achieve good results.

VII. Conclusions

In this paper, we present formal models and mechanical verification of correctness for spinlocks, which are implemented leveraging the ARM instructions and the Intel x86 instruction, respectively. Our models abstract spinlocks at instruction level and contain descriptions of hardware mechanisms, such as context switches. We verify the correctness of these spinlocks in Isabelle/HOL, and the verification results show that they satisfy the property, which proves that there is never more than one process in the critical section. In addition, we give some suggestions on how to implement spinlock leveraging the ARM instructions.

Acknowledgement

This work was supported by the National Key R & D Plan (2017YFB1301100), National Natural Science Foundation of China (61602325, 61802375, 61876111, 61877040), and the Project of Beijing Municipal Education Commission (KM20190028005).

References

[1] C. Baier, M. Daum, B. Engel, H. Härtig, J. Klein, S. Klüppelholz, S. Märcker, H. Tews, and M. Völp, "Waiting for locks: How long does it usually take?" in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2012, pp. 47–62.

[2] Y. Cui, Y. Wang, Y. Chen, and Y. Shi, "Locksim: An event-driven simulator for modeling spin lock contention," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 185–195, 2014.

[3] M. Desnoyers, P. E. McKenney, and M. R. Dagenais, "Multi-core systems modeling for formal verification of parallel algorithms." *Operating Systems Review*, vol. 47, no. 2, pp. 51–65, 2013.

[4] X. Feng, Z. Shao, Y. Dong, and Y. Guo, "Certifying low-level programs with hardware interrupts and preemptive threads," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 170–182.

[5] A. Chlipala, "A certified type-preserving compiler from lambda calculus to assembly language," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 54–65.

[6] G. Klein and T. Nipkow, "A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 4, pp. 619–695, 2006.

[7] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[8] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[9] A. Holdings, "ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition," 2014.

[10] P. Guide, "Intel® 64 and IA-32 Architectures Software Developers Manual," 2010.

[11] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Pearson, 2015.

[12] C. Flanagan, S. N. Freund, and S. Qadeer, "Thread-modular verification for shared-memory programs," in *European Symposium on Programming*. Springer, 2002, pp. 262–277.

[13] M. A. Hillebrand and D. C. Leinenbach, "Formal verification of a reader-writer lock implementation in C," *Electronic Notes in Theoretical Computer Science*, vol. 254, pp. 123–141, 2009.

[14] J. Shirako, N. Vrvilo, E. G. Mercer, and V. Sarkar, "Design, verification and applications of a new read-write lock algorithm," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012, pp. 48–57.

[15] N. Bogunovic and E. Pek, "Verification of mutual exclusion algorithms with SMV system," in *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, vol. 2. IEEE, 2003, pp. 21–25.

[16] W. H. Hesselink, "Correctness and concurrent complexity of the Black-White Bakery Algorithm," *Formal Aspects of Computing*, vol. 28, no. 2, pp. 325–341, 2016.

[17] T. Mizutani and K. Tomita, "Formalization of mutual exclusion algorithms in N-labeled calculus," in *2014 IEEE 7th Joint International Information Technology and Artificial Intelligence Conference*. IEEE, 2014, pp. 83–88.

[18] S. Qadeer and N. Shankar, "Verifying a self-stabilizing mutual exclusion algorithm," in *Programming Concepts and Methods PROCOMET98*. Springer, 1998, pp. 424–443.

[19] I. Sergey, A. Nanevski, and A. Banerjee, "Mechanized verification of fine-grained concurrent programs," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 77–87.

[20] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto, "TLA$^+$ proofs," in *International Symposium on Formal Methods*. Springer, 2012, pp. 147–154.

[21] X. Ji and L. Song, "Mutual Exclusion Verification of Peterson's Solution in Isabelle/HOL," in *Trustworthy Systems and their Applications (TSA), 2016 Third International Conference on*. IEEE, 2016, pp. 81–86.

[22] T. Ridge, "Petersons algorithm in Isabelle/HOL," 2006.

[23] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta informatica*, vol. 6, no. 4, pp. 319–340, 1976.

[24] P. Nieto *et al.*, "Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL," Ph.D. dissertation, Technische Universität München, 2002.

[25] S. Amani, J. Andronick, M. Bortin, C. Lewis, C. Rizkallah, and J. Tuong, "Complx: a verification framework for concurrent imperative programs," in *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM, 2017, pp. 138–150.