Formal Methods for Kernel Hackers

A practical introduction to TLA^+/PlusCal

Catalin Marinas <catalin.marinas@arm.com>

November 14, 2018

© 2018 Arm Limited

Agenda

- Introductory *TLA*$^+$ example
- LL/SC spinlock model in *PlusCal*
- Queued spinlock model
- Ideas for future work
- Resources

© 2018 Arm Limited
arm

Why use formal methods

Writing is nature's way of letting you know how sloppy your thinking is.

– Dick Guindon

Mathematics is nature's way of letting you know how sloppy your writing is.

Formal *mathematics is nature's way of letting you know how sloppy your mathematics is.*

– Leslie Lamport

- Formal verification allows checking/proving safety and liveness properties of a system
  - Formal proof: usually complex and (human) time consuming
  - Model checking: simpler but computing intensive. Requires finite number of states
- High level algorithm specification and verification
- Program refinement for a low level implementation of the high level algorithm

© 2018 Arm Limited
arm

Some Linux kernel formal models

- `arm64` Linux *ASID* allocator
  - Confirmed a bug previously found with *CnP* enabled
  - Uncovered a new bug in the *ASID* roll-over logic (requires very rare timing conditions)
- `arm64` *KPTI* vs *Software PAN*
  - Confirmed previously found bugs and verified the fix
- `arm64` Linux ticket spinlocks
  - Verified liveness properties with *LSE* atomics (not guaranteed with *exclusives*)
  - Uncovered bug in `spin_trylock()` on ticket roll-over (requires rare timing conditions)
- `arm64` KVM handling of *vGIC*
  - Modelling the *GIC* and the hypervisor interrupt handling, injection into guest, *vCPU* migration
  - Confirmed bug causing the loss of the source *vCPU* for an *SGI*
- Linux `context_switch()` handling of `mm_struct`
  - Verified safety properties of the `mm_users` and `mm_count` variables (chasing a use-after-free bug)
- `arm64` Linux *SVE/FPSIMD* register bank saving and restoring (work in progress)
  - Concurrency between kernel use of *FPSIMD*, context switching, user signal delivering, thread migration

© 2018 Arm Limited

TLA$^+$ and *PlusCal*

- *TLA*$^+$ (Temporal Logic of Actions) is a formal specification language developed by Leslie Lamport
  - Based on set theory and temporal logic, allows specification of invariant (safety) and liveness properties
  - Specification written in formal logic is amenable to finite model checking (using Yuan Yu's *TLC* model checker)
  - Can also be used for machine-checked proofs of correctness (using a theorem prover as back-end)
- *PlusCal* is a formal specification language which compiles to *TLA*$^+$
  - Pseudocode-like, better suited for specifying sequential algorithms
  - Simple way to describe concurrent threads/processes
- Notable real world uses
  - Specifying and model checking of the Alpha EV7 cache-coherency protocol
  - Amazon Web Services uncovering bugs in DynamoDB, S3, EBS
  - Microsoft Azure in designing Cosmos DB

© 2018 Arm Limited

Introductory TLA^+ example: specification 1

```
VARIABLES tick, count                         \* state

Init  == ∧ tick = 0                           \* state predicate
         ∧ count = 0                          \* (boolean state function)

Tick  == ∧ tick' = 1 - tick                   \* action (relation between
         ∧ UNCHANGED count                    \* successive states)

Count == ∧ count' = count + tick              \* action
         ∧ UNCHANGED tick


Next  == Tick ∨ Count                         \* action (disjunction)
Spec  == Init ∧ □[Next]_⟨tick, count⟩          \* temporal formula
                                              \* (specifies allowed behaviours)
```


© 2018 Arm Limited
arm

Introductory TLA^+ example: possible behaviours

Allowed system behaviour:

```
tick:   0   1   1   1   1   0   1   0   0   0   0   0   ...
count:  0   0   1   2   3   3   3   3   3   3   3   3   ...
        Tick    Count       Tick        Stuttering
```


© 2018 Arm Limited

arm

Introductory TLA^+ example: specification 2

```
VARIABLES tick, count, lasttick          \* state

Init  == ∧ tick = 0                      \* state predicate
         ∧ count = 0
         ∧ lasttick = 0
Tick  == ∧ tick' = 1 - tick              \* action
         ∧ tick = lasttick               \* (enabled if condition true)
         ∧ UNCHANGED ⟨count, lasttick⟩
Count == ∧ count' = count + tick         \* action
         ∧ tick ≠ lasttick               \* (enabled if condition true)
         ∧ lasttick' = tick
         ∧ UNCHANGED tick
Next  == Tick ∨ Count                    \* action (disjunction)
Spec  == Init ∧ □[Next]⟨tick, count, lasttick⟩   \* temporal formula
```


© 2018 Arm Limited
arm

LL/SC spinlock model in *PlusCal*

- *Load-link* reads the current `lock` value from memory
- *Store-conditional* writes the new `lock` value only if no updates have occurred since *LL*
- ARM hardware implementation using an *exclusive monitor*
- Classic LL/SC spinlock implementation using a single shared location for the `lock`
  - All CPUs polling the same memory location

© 2018 Arm Limited
arm

LL/SC spinlock model in *PlusCal*: variables

```
EXTENDS Naturals, Sequences, TLC

\* defined in the configuration file
CONSTANTS CPUS,           \* {p1, p2}
          ADDRS           \* {a1}

\* PlusCal algorithm placed inside a TLA+ comment
(* --algorithm spinlock {
variables
        memory    = [a ∈ ADDRS ↦ 0];          \* zero-initialised 'array'
        lock_addr = CHOOSE a ∈ ADDRS : TRUE; \* an address
        excl_mon  = [p ∈ CPUS ↦ "open"];     \* one monitor per CPU
        ...
} *)
```


© 2018 Arm Limited
arm

LL/SC spinlock model in *PlusCal*: exclusive monitor macros

```
\* PlusCal macros are modelled atomically
macro set_excl_mon(addr) {
        excl_mon[self] := addr;
}

\* reset the exclusive monitor to "open" if set to the given address
macro clear_excl_mon(addr) {
        excl_mon := [p ∈ CPUS ↦
                        IF excl_mon[p] = addr THEN "open" ELSE excl_mon[p]];
}
```


© 2018 Arm Limited
arm

LL/SC spinlock model in *PlusCal*: instruction macros

```
\* set the exclusive monitor to the load address
macro ldxr(reg, addr) {
        set_excl_mon(addr);
        reg := memory[addr];
}
\* update memory only if the exclusive monitor is set to the store address
macro stxr(stat, val, addr) {
        if (excl_mon[self] = addr) {
                clear_excl_mon(addr);
                memory[addr] := val;
                stat := 0;
        } else {
                stat := 1;
        }
}
```


© 2018 Arm Limited
arm

LL/SC spinlock model in *PlusCal*: instruction macros

```
\* classic load/store instructions
macro ldr(reg, addr) {
        reg := memory[addr];
}

\* clear the exclusive monitor if set to the store address
macro str(val, addr) {
        clear_excl_mon(addr);
        memory[addr] := val;
}
```

 © 2018 Arm Limited

arm

LL/SC spinlock model in *PlusCal*: locking procedures

```
procedure spin_lock(lock)
        variable lock_val, status;          \* local variables
{
l1:     ldxr(lock_val, lock);               \* each label represents a TLA+ step
l2:     if (lock_val ≠ 0)                    \* (labels can be automatically generated
                goto l1;                    \* but at a coarser grain)
l3:     stxr(status, 1, lock);
l4:     if (status ≠ 0)                      \* successful exclusive store?
                goto l1;
l5:     return;
}
procedure spin_unlock(lock)
{
u1:     str(0, lock);                       \* uncoditional lock release
u2:     return;
}
```

 © 2018 Arm Limited

arm

LL/SC spinlock model in *PlusCal*: processes

```
\* one PlusCal process per CPU
process (cpu ∈ CPUS)
{
        \* infinite lock/unlock loop
start:  while (TRUE) {
lock:           call spin_lock(lock_addr);
cs:             skip;                   \* critical section (no-op)
unlock:         call spin_unlock(lock_addr);
        }
}
```


© 2018 Arm Limited

arm

LL/SC spinlock model in *PlusCal*: invariants (safety)

```
\* type invariant
TypeInv == ∧ memory ∈ [ADDRS → Nat]
           ∧ excl_mon ∈ [CPUS → ADDRS ∪ {"open"}]

\* no two CPUs can be in the critical section simultaneously
ExclInv == ∀ p1, p2 ∈ CPUS :
                p1 ≠ p2 => ¬((pc[p1] = "cs") ∧ (pc[p2] = "cs"))

THEOREM Spec => □TypeInv
THEOREM Spec => □ExclInv
```


© 2018 Arm Limited
arm

LL/SC spinlock model in *PlusCal*: configuration

```
SPECIFICATION Spec
CONSTANT defaultInitValue = defaultInitValue
\* Add statements after this line.

CONSTANTS       CPUS = {p1, p2}
                ADDRS = {a1}

INVARIANTS      TypeInv
                ExclInv
```


© 2018 Arm Limited
arm

LL/SC spinlock model in *PlusCal*: liveness properties

```
\* Weak fairness required to eliminate infinite stuttering steps:
\*     Spec == Init ∧ □[Next]_vars ∧ ∀ self ∈ CPUS : WF_vars(cpu(self))
fair process (cpu ∈ CPUS)
...
\* at least one CPU eventually enters the critical section
LivenessAny == ∃ p ∈ CPUS : pc[p] = "start" ~> pc[p] = "cs"
\* all CPUs eventually enter the critical section (implies LivenessAny)
LivenessAll == ∀ p ∈ CPUS : pc[p] = "start" ~> pc[p] = "cs"

THEOREM Spec => LivenessAny
THEOREM Spec => LivenessAll
-------------------------------------------------------------------------
\* .cfg file:
PROPERTIES      LivenessAny
                LivenessAll
```


© 2018 Arm Limited
arm

LL/SC spinlock model in *PlusCal*: checking with *TLC*

```
Error: Temporal properties were violated.
Error: The following behavior constitutes a counter-example:
...
State 11:
/\ pc = (p1 :> "l4" @@ p2 :> "l4")            \* stxr executed on both CPUs
/\ status = (p1 :> 0 @@ p2 :> 1)              \* p1 succeeded, p2 failed
/\ excl_mon = (p1 :> "open" @@ p2 :> "open")
/\ memory = (a1 :> 1)                         \* lock taken
...
State 25:
/\ pc = (p1 :> "l4" @@ p2 :> "l3")            \* p2 is about to execute stxr
/\ status = (p1 :> 0 @@ p2 :> 1)              \* p1 successfuly executed stxr
/\ excl_mon = (p1 :> "open" @@ p2 :> "open")
/\ memory = (a1 :> 1)                         \* lock taken
Back to state 11
```

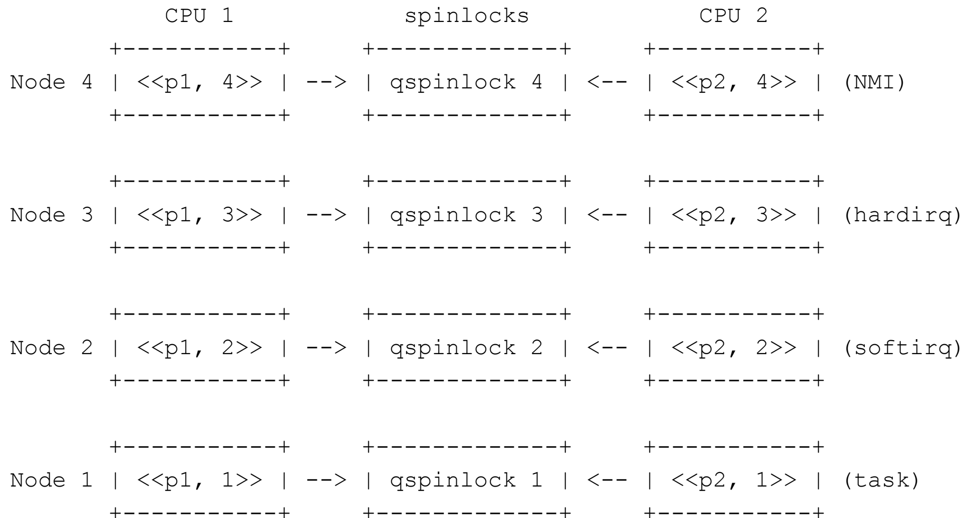

© 2018 Arm Limited
arm

Queued spinlock model

- Aims to guarantee liveness for all CPUs
- Scalable with the number of CPUs
- Contending CPUs adding themselves to a queue and spinning on own data structure
  - Needs to handle multiple nesting contexts per CPU (task, softirq, hardirq, NMI – modelled as *nodes*)
  - One *MCS lock* per CPU per *node* (nesting context)
- The formal model specifies $CPUs \times Nodes$ threads and *Nodes* locks
  - Threads represented as $\langle p, n \rangle$ tuples

© 2018 Arm Limited
arm

Queued spinlock model



© 2018 Arm Limited
arm

Queued spinlock model: constants

```
CONSTANTS CPUS,                    \* {p1, p2}
          MAX_NODES,               \* 2
          PENDING_LOOPS            \* 1

\* assumptions on the configuration
ASSUME MAX_NODES ∈ Nat \ {0}

\* abstract value not matching any CPU
NoCPU     == CHOOSE cpu : cpu ∉ CPUS
NODE_ZERO == ⟨NoCPU, 0⟩

\* MAX_NODES threads per CPU: e.g. ⟨p1, 1⟩, ⟨p1, 2⟩, ⟨p2, 1⟩, ⟨p2, 2⟩
THREADS == CPUS × (1..MAX_NODES)
```


© 2018 Arm Limited
arm

Queued spinlock model: data types

```
QLockType == [locked:   BOOLEAN,                 \* union {
              pending:  BOOLEAN,                 \*         atomic_t val;
              tail_idx: Nat,                     \*         struct {
              tail_cpu: CPUS ∪ {NoCPU}]          \*                 u8  locked;
                                                 \*                 u8  pending;
                                                 \*         struct {
                                                 \*                 u16 locked_pending;
                                                 \*                 u16 tail;
                                                 \*         };
                                                 \* };

McsLockType == [next:   THREADS ∪ {NODE_ZERO},   \* struct mcs_spinlock *next;
                locked: BOOLEAN,                 \* int locked;
                count:  Nat]                     \* int count;
```


© 2018 Arm Limited
arm

Queued spinlock model: helper operators

```
\* QLockType constructor
LockVal(l, p, i, c) == [locked   ↦ l,
                        pending  ↦ p,
                        tail_idx ↦ i,
                        tail_cpu ↦ c]
\* pre-defined values
ZERO_VAL    == LockVal(FALSE, FALSE, 0, NoCPU)
LOCKED_VAL  == LockVal(TRUE, FALSE, 0, NoCPU)
PENDING_VAL == LockVal(FALSE, TRUE, 0, NoCPU)

\* (val & ~_Q_LOCKED_MASK) in Linux
NEG_LOCKED_MASK(val) == val.pending ∨ val.tail_idx ≠ 0 ∨ val.tail_cpu ≠ NoCPU
\* (val & _Q_TAIL_MASK) in Linux
TAIL_MASK(val)       == val.tail_idx ≠ 0 ∨ val.tail_cpu ≠ NoCPU
```


© 2018 Arm Limited
arm

Queued spinlock model: variables

```
\* One qspinlock per node (e.g. task, softirq, hardirq, NMI)
qspinlock = [n ∈ 1..MAX_NODES ↦ LockVal(FALSE, FALSE, 0, NoCPU)];

\* One mcs_lock per thread (per CPU per node)
mcs_lock = [t ∈ THREADS ↦ [next   ↦ NODE_ZERO,
                            locked ↦ FALSE,
                            count  ↦ 0]];

\* 'self' represents the current thread, defined as a ⟨cpu, node⟩ tuple
CPU(self)     == self[1]
Lock(self)    == self[2]
McsNode(p, i) == ⟨p, i⟩
```


© 2018 Arm Limited
arm

Queued spinlock model: invariants (safety)

```
TypeInv == ∧ mcs_lock ∈ [THREADS → McsLockType]
           ∧ qspinlock ∈ [1..MAX_NODES → QLockType]

\* no two threads contending on the same lock can be in the critical
\* section simultaneously
ExclInv == ∀ t1, t2 ∈ THREADS : CPU(t1) ≠ CPU(t2) ∧ Lock(t1) = Lock(t2) =>
                ¬((pc[t1] = "cs") ∧ (pc[t2] = "cs"))

THEOREM Spec => □TypeInv
THEOREM Spec => □ExclInv
```


© 2018 Arm Limited
arm

Queued spinlock model: liveness

```
\* at least one thread eventually enters the critical section
LivenessAny == ∃ t ∈ THREADS : pc[t] = "start" ~> pc[t] = "cs"

\* all CPUs eventually enter the critical section in at least one context
LivenessAll == ∀ p ∈ CPUS : ∃ n ∈ 1..MAX_NODES :
                    pc[⟨p, n⟩] = "start" ~> pc[⟨p, n⟩] = "cs"

THEOREM Spec => LivenessAny
THEOREM Spec => LivenessAll
```


© 2018 Arm Limited

Queued spinlock model: findings

- *LivenessAll* properties violated prior to Linux 4.18
  - Two-CPU scenario fixed by commit `59fb586b4a07 ("locking/qspinlock: Remove unbounded cmpxchg() loop from locking slowpath")`
  - Three-CPU scenario fixed by commit `6512276d97b1 ("locking/qspinlock: Bound spinning on pending->locked transition in slowpath")`
  - The above commits are sufficient for `arm64` with *LSE* atomics extensions (ARMv8.1)
  - Avoiding `fetch_or()` (which uses a `cmpxchg()` loop on `x86`), commit `7aa54be29765 ("locking/qspinlock, x86: Provide liveness guarantee")`
- Does not implement memory ordering models (sequential consistency only)
- Exponential state space growth
  - Liveness checking: under 1 min for two threads, hours for three threads, days for four threads
  - Invariant checking significantly faster with *symmetry* optimisations
  - `-simulate` mode for checking random behaviours

© 2018 Arm Limited
arm

Ideas for future models

- CPU hotplug state machine
  - Deadlock freedom, liveness properties
- Page cache page properties
  - Safety: not seeing other process's data (e.g. *Dirty CoW*)
  - Liveness: page eventually reaches the block device
- *RCU* - anything left to model?
- Other tools
  - SPIN/Promela: model checker using the Promela specification language
  - CBMC: bounded model checker for ANSI-C
  - Alloy: declarative specification language and model checker
  - …

© 2018 Arm Limited
arm

Resources

- Main *TLA*$^+$ page
  `https://lamport.azurewebsites.net/tla/tla.html`
- *PlusCal* manual
  `https://lamport.azurewebsites.net/tla/c-manual.pdf`
- "Specifying Systems"
  `https://lamport.azurewebsites.net/tla/book.html`
- *TLA*$^+$ *Tools*
  `https://lamport.azurewebsites.net/tla/tools.html` (pre-built)
  `https://github.com/tlaplus/tlaplus/tree/master/tlatools` (source)
- Linux kernel specs
  `https://git.kernel.org/pub/scm/linux/kernel/git/cmarinas/kernel-tla.git`

© 2018 Arm Limited

Thanks

The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks

© 2018 Arm Limited