Practical Rely/Guarantee Verification of an Efficient Lock for seL4 on Multicore Architectures

Robert J. Colvin[1,2], Ian J. Hayes[2], Scott Heiner[2], Peter Höfner[3], Larissa Meinicke[2], and Roger C. Su[3]

[1] Defence Science and Technology Group, Australia
[2] University of Queensland, Brisbane, Australia
{r.colvin, ian.hayes, s.heiner, l.meinicke}@uq.edu.au
[3] Australian National University, Canberra, Australia
{peter.hoefner, roger.su}@anu.edu.au

Abstract. Developers of low-level systems code providing core functionality for operating systems and kernels must address micro-architectural features of modern multicore processors, including and especially pipelined "out-of-order execution" of the code as written. The effects of out-of-order execution are typically summarised by a "weak memory model", a term which includes further complicating factors that may be introduced by compiler optimisations. In many cases, the nondeterminism inherent in weak memory models can be expressed as micro-parallelism, i.e., parallelism within threads and not just between them. Fortunately Jones' *rely/guarantee* reasoning provides a compositional method for verification of shared-variable concurrency, whether that be in terms of communication between top-level threads or micro-parallelism within threads. In this paper we provide an in-depth specification and verification of the lock algorithm used in the seL4 microkernel, using rely/guarantee to handle both inter-thread communication and micro-parallelism introduced by weak memory models. The proof is machine-checked in Isabelle/HOL, building on an existing theory for rely/guarantee reasoning, extended with rules specialised for data type invariants, spin loops, and nested parallelism.

Keywords: Rely/Guarantee · Lock algorithms · Weak memory models · Multicore architectures · Concurrency · Verification

1 Introduction

The CLH lock algorithm (named after co-inventors Craig and Landin & Hagerstein) [9,30,31] provides a space- and time-efficient method for ensuring mutual exclusion, and furthermore ensures that any thread attempting to acquire the lock will eventually succeed under reasonable fairness assumptions. CLH's efficiency arises partly from its utilisation of the hardware-level cache system to minimise contention in main memory. It is used in the seL4 microkernel [18,44,27] as a single system-wide lock.

In this paper, we formally verify the functional correctness of the CLH lock, and lay the foundation for proving progress properties. Additionally, we take into account the effects of *weak memory models* of potential architectures upon which seL4 may be deployed. We use Jones' *rely/guarantee reasoning* [22,23,24] for this task, and find it especially suited for hardware-level algorithms which involve weak memory effects. We use the proof assistant Isabelle/HOL [35,38], and build on existing library features (including a rely/guarantee encoding by Prensa Nieto [39]) to automate some parts of the proof.

The paper is structured as follows: Sect. 2 summarises Jones' rely/guarantee reasoning, with some specialisations for our context. Sect. 3 introduces the specification of a generic lock, and its specialisation to a queued lock. Sect. 4 proves functional correctness of the CLH lock currently used in seL4 with respect to the specification of a queued lock. Sect. 5 analyses weak memory effects on the CLH code, identifying the micro-parallelism that may be introduced during execution on multicore architectures, and whether or not that micro-parallelism may be retained or must be eliminated. Sect. 6 summarises related work.

2 Rely/Guarantee Reasoning

Developed by Jones in the 1980s [22,23,24], rely/guarantee reasoning provides a compositional framework for the verification of programs with shared-variable concurrency. The key idea is to extend pre/postcondition reasoning of classic Floyd-Hoare logic [13,20] by capturing the effect of the environment of a thread in a *rely* relation between program states; additionally, a thread summarises its own effects on its environment via a *guarantee* relation between program states.

In the context of a lock, a thread i *relies* on the fact that if i currently holds the lock, then some resource x cannot be modified by the environment. This is written as $\mathit{lock} = \mathsf{held}(\mathsf{i}) \Rightarrow x' = x$, a rely *relation* where lock and x are shared variables, with primed instances (x') referring to their values in the *post-state* (as opposed to unprimed instances (x) in the *pre-state*). The corresponding *guarantee* relation is $\mathit{lock} \neq \mathsf{held}(\mathsf{i}) \Rightarrow x' = x$; that is, thread i guarantees not to change x if it does not hold the lock. In this work, pre/postconditions are single-state predicates (where a state is a mapping from variables to values), and relies and guarantees are binary relations on states (with primed as well as unprimed variables). We allow the standard logical connectives to apply to either predicates or relations under their usual interpretations.

A rely/guarantee quintuple $\{p, r\}\, c\, \{g, q\}$ states that the program c achieves the postcondition q and guarantees each step satisfies g, provided that it starts in a state satisfying precondition p and each step of the environment satisfies the rely r. Inference using rely/guarantee takes place via the application of specialised syntax-driven rules. The rules we apply are adaptations of some of the rules provided by Coleman and Jones [2] and those encoded in Isabelle by Prensa Nieto [39].

$$\frac{p,\, q \text{ stable under } r \quad p_{\langle x \leftarrow e \rangle} \Rightarrow q \quad p \lhd (x' = e) \Rightarrow g}{\{p, r\}\, x := e\, \{g \vee \mathrm{id}, q\}} \quad (1) \qquad \frac{\{p, r\}\, c_1\, \{g, p_{mid}\} \quad \{p_{mid}, r\}\, c_2\, \{g, q\}}{\{p, r\}\, c_1 ;\ c_2\, \{g, q\}} \quad (2)$$

$$\frac{\{p_0, r_0\}\, c\, \{g_0, q_0\} \quad p \Rightarrow p_0 \quad r \Rightarrow r_0 \quad g_0 \Rightarrow g \quad q_0 \Rightarrow q}{\{p, r\}\, c\, \{g, q\}} \quad (3)$$

$$\frac{\{p, r \vee g_2\}\, c_1\, \{g_1, q_1\} \quad \{p, r \vee g_1\}\, c_2\, \{g_2, q_2\}}{\{p, r\}\, c_1 \parallel c_2\, \{g_1 \vee g_2, q_1 \wedge q_2\}} \quad (4)$$

$$\frac{\{p_1, r_1\}\, c_1\, \{g_1, q_1\} \quad \{p_2, r_2\}\, c_2\, \{g_2, q_2\} \quad g_1 \Rightarrow r_2 \wedge g_2 \Rightarrow r_1}{\{p_1 \wedge p_2, r_1 \wedge r_2\}\, c_1 \parallel c_2\, \{g_1 \vee g_2, q_1 \wedge q_2\}} \quad (5)$$

$$\frac{\forall\, i \,.\, \{p_i, r_i\}\, c_i\, \{g_i, q_i\} \quad \forall\, i \,.\, \forall\, j \neq i \,.\, g_i \Rightarrow r_j}{\{\bigwedge_i p_i, \bigwedge_i r_i\}\ \|_i .\ c_i\, \{\bigvee_i g_i, \bigwedge_i q_i\}} \quad (6)$$

$$\frac{p,\, q \text{ stable under } r \quad \mathrm{id} \Rightarrow g \quad p \wedge \neg b \Rightarrow q}{\{p, r\}\, \mathsf{while}\ b\ \mathsf{do}\ \mathsf{skip}\, \{g, q\}} \quad (7)$$

Fig. 1. Rely/guarantee inference rules adapted from [2,39].

2.1 Inference rules

The inference rules we apply in this paper are given in Fig. 1. Rule (1) states that an atomic assignment $x := e$ establishes q and guarantees $g \vee \mathrm{id}$ under precondition p and rely r, provided that neither p nor q is affected by the environment, and executing $x := e$ from a state satisfying p both establishes q in the post state and implies the guarantee g. As an example, we may immediately derive $\{\mathbf{true}, x' = x\}\, x := 5\, \{x \leftarrow 5 \vee \mathrm{id}, x = 5\}$, where '$x \leftarrow v$' is the relation formed from pairs of states $(\sigma, \sigma_{[x := v]})$, i.e., an update of x to v. The inclusion of the identity relation id to the guarantee is a technical requirement that accounts for the fact that the assignment statement inflicts no interference on the rest of the system prior to executing – between steps of c in '$c \parallel x := 1$' either the assignment occurs or nothing (id) occurs. We let $p_{\langle x \leftarrow e \rangle}$ be the predicate obtained by updating each state in p so that x takes the value of e in the pre-state; $p \lhd r$ be the restriction of the relation r to just the pre-states satisfying predicate p; and p stable under r states that p is maintained by r.

Rule (2) composes the properties of two commands in sequential composition provided they agree on a *mid-state*, written p_{mid} in the rule. The relies and guarantees are identical across both components.

Rule (3) is a general rule allowing the weakening of preconditions and relies, and strengthening of postconditions and guarantees. We tend to leave the application of this rule implicit where the implications are clear from the context.

The significance of rely/guarantee lies in its handling of parallel composition. There are two compositional rules typically found in the literature that apply to two parallel processes, $c_1 \parallel c_2$. Rule (4) assumes some external environment "inflicting" r on the local system (in this case, the local system is $c_1 \parallel c_2$). The compositionality of the rule requires that the additional interference g_2 inflicted on c_1 by c_2 must be handled (by c_1), and vice-versa. Rule (5) instead insists that the environment must already respect the relies of both threads c_1 and c_2, and each thread must also respect each other's relies. Rule (4) is straightforwardly

derivable from Rule (5). Rule (4) is *decompositional*, in the sense that a single system is split into two parallel subprocesses, which does not affect its external interface (r); while Rule (5) is *compositional*, in the sense that two separate systems, each with their own pre-existing specification, are combined to give a single, new specification and interface ($r_1 \wedge r_2$). In our context, micro-parallelism within a single thread (as addressed in Sect. 5) is best handled by Rule (4), while at the system level, where a set of threads interact via a shared contract, Rule (5) is more suitable. This is given by the generalisation of Rule (5) to an arbitrary number of threads, written $\|_i\ .\ c_i$, as shown in Rule (6).

A common structure in lock implementations are *spin loops*, i.e., loops with an empty body, while b do skip (abbreviated to await $\neg b$). Rule (7) states that provided the p and q are stable, and that p and the negation of the guard imply q, one can assume that q holds if the loop terminates. A spin loop changes nothing, and hence guarantees the identity relation, id.

2.2 Invariants

We define the following shorthand for a (single-state) *data-type invariant* I (or any invariant), in the spirit of Morgan's refinement calculus [33].

$$\{p, r\}\ c /\!\!/ I\ \{g, q\} \mathrel{\widehat{=}} \{p \wedge I, r \wedge (I \Rightarrow I')\}\ c\ \{g \wedge (I \Rightarrow I'), q \wedge I\} \tag{8}$$

Definition (8) factors out the invariant, making specification more concise. It states that if I holds in the pre-state, and the environment can be relied upon to maintain the invariant, then c guarantees to maintain the invariant and establish it in the post-state (the relation $I \Rightarrow I'$ requires that I holds in the post-state provided it holds in the pre-state).

The use of an invariant helps to avoid the use of code-specific program-counter-style variables, particularly when used on auxiliary concepts (essentially forming the "specification" of a data type). Program-counters are preferably avoided, as they are brittle with respect to the structure of the code, and become cumbersome with complex programs, such as those with nested parallelisms.

3 Specification of Locks in Rely/Guarantee

In this section we discuss specifying locks in a rely/guarantee setting. The auxiliary variable *lock* $\in$ LockStatus represents the status of the lock, which is either free if no thread holds the lock, or held(i) if thread i $\in \mathcal{T}$ holds the lock.

$$\mathsf{LockStatus} ::= \mathsf{free} \mid \mathsf{held(i)}$$

3.1 Properties of a generic lock

A lock is used by multiple threads to ensure exclusive access to some shared resource x within a *critical section* of code, $\mathsf{crit_i}(x)$. The lock is *acquired* before

entering the critical section, and *released* afterwards. (Before acquiring the lock and after releasing it the thread may execute some non-critical code, which modifies neither x nor the lock, and for brevity we omit consideration of such code.) Hence, from the viewpoint of each thread $\mathsf{i} \in \mathcal{T}$, the lock is used in the pattern

$$\mathsf{acquire_i}() \,;\; \mathsf{crit_i}(x) \,;\; \mathsf{release_i}().$$

The intention is that while using or modifying x in the critical section, a thread may rely on the fact that no other thread concurrently modifies x. That is, in the calling code, $\mathit{lock} = \mathsf{held}(\mathsf{i}) \Rightarrow x' = x$.

The operation $\mathsf{acquire_i}()$ is called to attempt to hold the lock, and $\mathsf{release_i}()$ to release the lock after the thread has finished accessing the shared resource.[1] The key properties required are the following.

1. A call to $\mathsf{acquire_i}()$ establishes the postcondition $\mathit{lock} = \mathsf{held}(\mathsf{i})$; and
2. Thread i may *rely* on $\mathit{lock} = \mathsf{held}(\mathsf{i}) \Rightarrow \mathit{lock}' = \mathsf{held}(\mathsf{i})$.

The postcondition $\mathit{lock} = \mathsf{held}(\mathsf{i})$ is *stable* – maintained by the environment – if the environment satisfies the rely, that is, if i holds the lock, then it will never be taken away. These conditions necessitate the converse guarantee that thread i will respect the lock if some other thread j holds it.

In practice, these conditions allow simplified proofs of critical sections. A generic specification of a critical section is the following.

$$\{p(x), x' = x\}\ \mathsf{crit_i}(x)\ \{\mathrm{id}\,\overline{x}, q(x)\} \tag{9}$$

Here, the critical section establishes some property $q(x)$ on the shared data, under some (probably minimal) precondition $p(x)$ and provided it can assume that the environment does not modify x. Additionally, it guarantees to modify nothing other than x, written $\mathrm{id}(\overline{x})$. By composing $\mathsf{acquire_i}()$ and $\mathsf{crit_i}(x)$ the critical section may assume $\mathit{lock} = \mathsf{held}(\mathsf{i})$ in its initial state; and, given the overarching thread-level rely $\mathit{lock} = \mathsf{held}(\mathsf{i}) \Rightarrow (\mathit{lock}' = \mathsf{held}(\mathsf{i}) \wedge x' = x)$, the simpler rely $x' = x$ can be derived. The programmer may therefore implement $\mathsf{crit_i}(x)$ according to (9) as if operating in a single-process environment.[2]

3.2 Queued lock

When multiple threads call $\mathsf{acquire_i}()$ concurrently, a desirable property is that the lock is next obtained by the thread that has been waiting the longest. To capture this refinement of the lock concept, a generic lock can be modelled via an abstract queue of waiting threads, q, as given in Fig. 2. We let $\mathsf{i} \in q$ mean that i is an element of list q, and let $\mathsf{hd}(q)$ and $\mathsf{tl}(q)$ return q's first element and tail, respectively.

[1] Note that acquire and release have no explicit parameters, but in some implementations they may (usually some component of the lock data structure). They do have the thread identifier (i) as an implicit parameter. When these operations have no explicit parameters, this is also referred to as a standard interface [43].

[2] The structures above may be generalised to handle more resources than a single x, including local variables and other shared variables not controlled by *lock*.

$$\{\mathsf{i} \notin q\}\ \mathsf{acquire_i}();\quad \{\mathsf{i} = \mathsf{hd}(q)\}\ \mathsf{release_i}()\ \{\mathsf{i} \notin q\}$$

$$\begin{aligned}
\mathsf{invariant} &\mathrel{\widehat{=}} \mathsf{distinct}(q)\\
\mathsf{contract_i} &\mathrel{\widehat{=}} (\mathsf{i} \in q \Leftrightarrow \mathsf{i} \in q') \wedge (\mathsf{i} \in q \Rightarrow q'!\mathsf{i} \leq q!\mathsf{i})\\
\mathsf{rely_i} &\mathrel{\widehat{=}} \mathsf{invariant} \wedge \mathsf{contract_i}\\
\mathsf{guar_i} &\mathrel{\widehat{=}} \mathsf{invariant} \wedge (\forall \mathsf{j} \neq \mathsf{i} \,.\, \mathsf{contract_j})\\
\mathit{lock} &\mathrel{\widehat{=}} (\mathbf{if}\ q = [\,]\ \mathsf{then}\ \mathsf{free}\ \mathsf{else}\ \mathsf{held}(\mathsf{hd}(q)))
\end{aligned}$$

Fig. 2. Rely/guarantee specification of the queued lock

To emphasise the different parts of a specification, we separate out the guarantees in guar and the relies in rely, and the pre/postconditions are written in green braces. Hence, Fig. 2 implies a quintuple of the form in (8),

$$\{\mathsf{i} \notin q, \mathsf{rely_i}\}\ \mathsf{acquire_i}()\ ;\ \mathsf{release_i}()\ /\!\!/\ \mathsf{invariant}\ \{\mathsf{guar_i}, \mathsf{i} \notin q\}.$$

(At this level, we omit the critical section since it does not modify the lock.) The intermediate states added to the structure of the program, in this case, between $\mathsf{acquire_i}()$ and $\mathsf{release_i}()$, represent predicates that can be used to fulfil the antecedents of Rule (2) during a proof of the full quintuple. For instance, the postcondition for $\mathsf{acquire_i}()$ also forms the precondition for $\mathsf{release_i}()$ (similar to an Owicki-Gries-structured proof [36,37]). The data type invariant specifies that each element (thread identifier) in q is distinct, i.e., can appear at most once.

$$\mathsf{distinct}(t) \mathrel{\widehat{=}} \mathit{length}(t) = \mathit{size}(\mathsf{elems}(t))$$

The specification introduces a '$\mathsf{contract_i}$' for thread i, which is a relation describing thread i's assumption about how and when the state will be modified. Then, each thread guarantees to fulfil the contract for every other thread in the system. In this case, the contract states that i cannot be entered into or removed from the queue by any other thread, and additionally, that once in the queue it can only ever get closer to the head: $\mathsf{i} \in q \Rightarrow q'!\mathsf{i} \leq q!\mathsf{i}$. We use the notation $q!\mathsf{i}$ to denote the index of the element i in the list q. This expression is undefined if the list q does not contain i; we ensure throughout that such indexing is defined. From this invariant we can infer more specifically that once a thread i reaches the head of the queue it will remain there (until it removes itself).

$$\mathsf{contract_i}\ \Rightarrow\ \mathsf{hd}(q) = \mathsf{i} \Rightarrow \mathsf{hd}(q') = \mathsf{i}$$

The key component that establishes this meets the specification of a lock is the *derivation* of the *lock* state from the queue. If the queue is empty then the lock is free, and otherwise the first thread in the queue holds the lock. Under this definition, the predicates and relations in Fig. 2 imply the lock properties given in Sect. 3.1. We use a similar justification later for showing that CLH implements an (abstract) lock.

4 Functional Correctness of CLH-seL4

The seL4 microkernel was the first operating system whose (C code) implementation was formally verified [28], and its development continues [18], including formal proofs of security criteria [34]. The original proof of correctness essentially assumed execution on a single core, but recent developments have seen deployment on multicore Arm, and potentially x86 and RISC-V, necessitating the need for managing concurrent processors. This is addressed via a system-wide lock to handle interrupts, and this lock is currently implemented via a version of CLH [44]. CLH is an interesting target for verification via rely/guarantee, because it is efficient, used in practice, and involves fine-grained concurrency that cannot be managed by programming language abstractions: as it is one of the fundamental operations of the microkernel, it cannot assume access to libraries, and is exposed to details of the microarchitecture.

We first present the acquire and release operations in a C-like language corresponding to the implementation currently in the release version of seL4. We apply rely/guarantee reasoning to this code, augmented with auxiliary variables capturing key concepts, and prove that it satisfies the conditions for a concurrent lock. We assume lines of code are executed in the order specified (*sequential consistency*) – weak memory effects are addressed in Sect. 5.

4.1 CLH-seL4 implementation

Fig. 3 contains code similar to that used in the seL4 microkernel (we have renamed some variables for consistency with other presentations of CLH). We omit interrupt-handling specifics, as that code is separate to the locking mechanism; we also omit details relating to cache line sizes, etc., as those are efficiency concerns not affecting correctness (for an explanation of CLH's computational advantages, see paragraph below under *Efficiency*).

Broadly speaking, the code maintains a "queue" of threads in a similar fashion to a linked list structure through the local `prev` variables. We give a brief description below, but defer a full explanation of the operation of the code until we augment it with the key auxiliary concepts.

```
type Status = Granted | Pending      type Node = struct{status : Status}
type proc = struct{node : *Node; tmp : *Node}
var procs : array N of proc := ...;      var tail : *Node := ...
tail→status := Granted;

acquire(i : T) =
  procs[i].node→status := Pending;
  var prev := xchg(&tail, procs[i].node);
  procs[i].tmp := prev;
  await prev→status = Granted;

release(i : T) =
  procs[i].node→status := Granted;
  procs[i].node := procs[i].tmp
```

Fig. 3. Excerpt of the seL4 CLH implementation (focussing on the logic) [44].

Types and initialisation. The code assumes a fixed number of threads, N. (This is a valid assumption for the seL4 use-case, as each thread corresponds to a single core on a multicore system, rather than dynamically created user-level processes.) Each thread i is initially allocated a pointer to its own `Node`, which can be accessed via `procs`[i].`node`. The node itself is simply a boolean which indicates whether the next waiting node must continue to wait (Pending) or may take the lock (Granted). There is one more `Node` than there are threads, with this extra node effectively marking the head of the abstract queue. The array `procs` is initialised so that each thread has a unique node, and `tail` is initialised to the unique unused node (we omit these straightforward details). Initially, the status of `tail` is set to Granted so that the first node to attempt to acquire the lock may do so.

Acquire. To acquire the lock a thread i initialises its node to have the status Pending, indicating to the subsequent thread j that joins the queue that j cannot (immediately) take the lock. An *atomic exchange* operation, `xchg`, is then called, which reads and returns the current value of the `tail` pointer, and updates `tail` to the node of the calling thread. Thus, the `prev` local variable records the previous node in the queue. This node is then stored in `procs`[i].`tmp` for access in the subsequent call to $\mathsf{release}_\mathsf{i}()$,[3] and eventual use by i the next time it calls acquire. In the last stage of acquire the thread spins on the status of the previous node, waiting for it to become Granted, at which point thread i takes the lock and enters its critical section. Recall that the await b command abbreviates a while loop with an empty body, i.e., while $\neg b$ do skip.

Release. To release the lock, the thread sets the status of its own node to Granted, signalling to the next waiting thread that it may proceed. It prepares for its next call to $\mathsf{acquire}_\mathsf{i}()$ by copying the node saved in `procs`[i].`tmp` into `procs`[i].`node`. Note that another thread may concurrently take the lock and proceed to its critical section before thread i executes this second line of $\mathsf{release}_\mathsf{i}()$.

Efficiency. In terms of space efficiency, one `Node` is allocated per-processor, plus one additional node. Threads swap nodes between themselves during execution, but the complete set of nodes never changes – there is no need for freeing/allocating memory during execution (as long as the total number of threads that will access the lock is known beforehand).

The second aspect of efficiency involves a specific interaction with the memory (cache) system. Although the status of a node can be recorded in a single bit, the `status` array is padded out so that each boolean takes up space equal to a single cache line – this is the physical unit that processors may read from main memory. Even if a single byte is requested from main memory, the entire cache line in which that byte resides is read into the processor's local cache. The size of a cache line is processor-specific; 64 bytes is a typical size used in contemporary architectures. As long as no process in the system changes any part of

[3] An alternative would be to return `prev` from acquire and pass that value as a parameter to the call to release, eliminating the need for the `tmp` field. However, this would expose implementation details in the interface to the lock and require a local variable to be kept across the critical section.

that 64 bytes, the local process can read from its copy of that chunk, minimising accesses of shared memory. CLH exploits this so that each processor's spin loop is effectively spinning on its own local cache, separately to every other process in the system. The only time that main memory is accessed is precisely when the previous processor in the queue flips the status from Pending to Granted, triggering a refresh of that cache line.

In contrast, a typical *ticket lock* implementation of a queued lock has a single shared counter, with each waiting process taking their own ticket number as they join the queue and then spin on the counter waiting for their turn. This requires only a single location in memory to represent the lock (the counter), but results in significant overhead on the memory system as each waiting thread continually polls main memory, and each local cache is updated every time the central counter changes.

4.2 Proof structure for CLH-seL4

Fig. 4 outlines the proof of the CLH-seL4, including the invariant, pre/postconditions, rely/guarantees, and intermediate states. The proof uses the following notation, where $a \# t$ represents a non-empty list with first (head) element a, and '$\frown$' is list concatenation.

$$\mathsf{injective}(\mathsf{f}) \mathrel{\widehat{=}} \forall\, i,\ j \,.\, i \neq j \Rightarrow f(i) \neq f(j)$$
$$f\langle\!\langle [\,] \rangle\!\rangle = [\,] \qquad f\langle\!\langle a \# t \rangle\!\rangle = f(a) \# (f\langle\!\langle t \rangle\!\rangle)$$

A function f is injective if it maps each member of its domain to a unique element. An application $f\langle\!\langle t \rangle\!\rangle$ is the list obtained by applying f to each of t's elements. We define $\mathsf{elems}(t)$ to be the set of elements in list t, and for brevity we use $a \in t$ as a shorthand for $a \in \mathsf{elems}(t)$.

Types, initialisation, and auxiliaries. We augment the code with several auxiliary variables to capture key ideas. The `structs` are carried over from Fig. 3. There is one local variable `prev` per thread, which we represent in the usual way as a mapping from thread identifiers to the type of the local – in this case, `*Node`. As such the `prev` for thread i is written $\texttt{prev}_\mathsf{i}$, although there is otherwise no distinction between a local variable and an array indexed by threads. Wherever nodes are dereferenced – to access their `status` – we implicitly access the *heap*, which is a mapping from `*Node` to `Node`.

The most complex part of the algorithm is how nodes are shared between threads. The auxiliary variable *reserved*, which by convention we write in red, keeps track of the unique node currently marked by a process for its own use. This auxiliary concept mostly corresponds to the concrete `procs`[i].`node`, except for a brief period during release, where a thread's reserved node may be found in `procs`[i].`tmp`. In a symbiotic way, the auxiliary η variable records the lone unreserved node in the system, which is always the previous (`prev`) node of the thread holding the lock, or `tail` if the queue is empty.

We also maintain an abstract list of the waiting threads in q. While this value could be *derived* from the concrete state, making it explicit in the code also acts

as a type of abstract "program counter", indicating which threads are currently attempting to acquire the lock.

$$
\begin{array}{ll}
\mathsf{var}\ \mathtt{prev} : \mathcal{T} \to *\mathtt{Node}; & (\textit{uninitialised local}) \\
\mathsf{var}\ \textit{reserved} : \mathsf{array}\ \mathcal{T}\ \mathsf{of}\ *\mathtt{Node}; & \mathsf{initially}\ \ \lambda\, \mathsf{i}\,.\, \mathtt{procs}[\mathsf{i}].\mathtt{node} \\
\mathsf{var}\ \eta : *\mathtt{Node}; & \mathsf{initially}\ \ \mathtt{tail} \\
\mathsf{var}\ q : \mathcal{T}\ \textit{list}; & \mathsf{initially}\ \ [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathsf{acquire}_\mathsf{i}() \mathrel{\widehat{=}} \\
\quad \{\mathsf{i} \notin q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \textit{reserved}[\mathsf{i}]\} \\
\ \mathtt{procs}[\mathsf{i}].\mathtt{node} \to \mathtt{status} := \mathsf{Pending}; \\
\quad \{\mathsf{i} \notin q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \textit{reserved}[\mathsf{i}] \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} \to \mathtt{status} = \mathsf{Pending}\} \\
\ \langle \mathtt{prev}_\mathsf{i} := \mathtt{tail};\ \mathtt{tail} := \mathtt{procs}[\mathsf{i}].\mathtt{node};\ q := q \frown [\mathsf{i}]\rangle; \\
\quad \{\mathsf{i} \in q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \textit{reserved}[\mathsf{i}]\} \\
\ \mathtt{procs}[\mathsf{i}].\mathtt{tmp} := \mathtt{prev}_\mathsf{i}; \\
\quad \{\mathsf{i} \in q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \textit{reserved}[\mathsf{i}] \wedge \mathtt{procs}[\mathsf{i}].\mathtt{tmp} = \mathtt{prev}_\mathsf{i}\} \\
\ \mathsf{await}\ \mathtt{prev} \to \mathtt{status} = \mathsf{Granted}\,; \\
\quad \{\mathsf{i} \in q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \textit{reserved}[\mathsf{i}] \wedge \mathtt{procs}[\mathsf{i}].\mathtt{tmp} = \eta \wedge \mathsf{i} = \mathsf{hd}(q)\} \\
\mathsf{release}_\mathsf{i}() \mathrel{\widehat{=}} \\
\quad \{\mathsf{i} \in q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \textit{reserved}[\mathsf{i}] \wedge \mathtt{procs}[\mathsf{i}].\mathtt{tmp} = \eta \wedge \mathsf{i} = \mathsf{hd}(q)\} \\
\ \langle \mathtt{procs}[\mathsf{i}].\mathtt{node} \to \mathtt{status} := \mathsf{Granted}\,;\ \eta \rightleftharpoons \textit{reserved}[\mathsf{i}]\,;\ q := \mathsf{tl}(q)\rangle\,; \\
\quad \{\mathsf{i} \notin q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{tmp} = \textit{reserved}[\mathsf{i}]\} \\
\ \mathtt{procs}[\mathsf{i}].\mathtt{node} := \mathtt{procs}[\mathsf{i}].\mathtt{tmp} \\
\quad \{\mathsf{i} \notin q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \textit{reserved}[\mathsf{i}]\}
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{invariant} \mathrel{\widehat{=}} & \mathsf{distinct}(q) \wedge \mathsf{injective}(\mathsf{reserved}) \wedge \eta \notin \mathsf{ran}\ \textit{reserved} \wedge \\
& \eta \to \mathtt{status} = \mathsf{Granted} \wedge (\forall\, \mathsf{i} \in q\,.\, \textit{reserved}[\mathsf{i}] \to \mathtt{status} = \mathsf{Pending}) \wedge \\
& \eta \,\#\, \textit{reserved}\langle\!\langle q \rangle\!\rangle = \mathtt{prev}\langle\!\langle q \rangle\!\rangle \frown [\mathtt{tail}] \qquad (10) \\
\mathsf{contract}_\mathsf{i} \mathrel{\widehat{=}} & \mathsf{id}(\mathtt{procs}[\mathsf{i}], \textit{reserved}[\mathsf{i}], \textit{reserved}[\mathsf{i}] \to \mathtt{status}) \wedge \\
& (\mathsf{i} \in q \Leftrightarrow \mathsf{i} \in q') \wedge (\mathsf{i} \in q \Rightarrow q'!\mathsf{i} \le q!\mathsf{i}) \qquad (11) \\
\mathsf{rely}_i \mathrel{\widehat{=}} & \mathsf{invariant} \wedge \mathsf{contract}_i \\
\mathsf{guar}_i \mathrel{\widehat{=}} & \mathsf{invariant} \wedge (\forall\, j \neq i\,.\, \mathsf{contract}_j) \\
\mathit{lock} \mathrel{\widehat{=}} & (\mathbf{if}\ q = [\,]\ \mathsf{then}\ \mathsf{free}\ \mathsf{else}\ \mathsf{hd}(q)) \qquad (12)
\end{array}
$$

Fig. 4. seL4 CLH proof structure

Acquire. The atomic exchange instruction is represented as an atomic sequence of instructions enclosed in angled brackets, $\langle \ldots \rangle$.

$$\mathtt{prev} := \mathtt{xchg}(\&\mathtt{tail}, \mathtt{procs}[\mathsf{i}].\mathtt{node}) \rightsquigarrow \langle \mathtt{prev} := \mathtt{tail};\ \mathtt{tail} := \mathtt{procs}[\mathsf{i}].\mathtt{node} \rangle$$

The `prev` local is updated to the current value of `tail`, which is then updated to thread i's node. Additionally, in Fig. 4, the abstract queue q is updated with

i appended to the end, indicating the concept of i joining the queue of waiting threads. The rest of the code corresponds to that of Fig. 3.

Release. The relatively benign-seeming line of code to set the status of the node of the releasing thread to Granted in fact holds much of the subtlety of the data structure. This is reflected in the abstract swap of the *reserved* node for thread i with what is currently the abstract η variable, $\eta \rightleftharpoons \mathit{reserved}[\mathsf{i}]$ (this is a simultaneous assignment, $\eta, \mathit{reserved}[\mathsf{i}] := \mathit{reserved}[\mathsf{i}], \eta$). Thus, thread i takes the now-unused former head of the node queue for itself, to be used in its next attempt to acquire the lock; and it simultaneously leaves behind its own node to become the current head of the node queue. Abstractly, this is reflected in the queue of threads by removing the first thread – which is i – from q. The *reserved* and η auxiliary variables are only modified at this key line of release, which straightforwardly ensures that *reserved* remains injective, and that η is not reserved.

Data type invariant. The invariant describes the relationships of the variables in the system; in particular, it outlines the auxiliaries η and *reserved*, properties of q, and the `status` of nodes. The key aspect in this regard is that *reserved* nodes of threads currently in the queue are always Pending, while the η node is always Granted. Additionally, the *reserved* mapping is always injective, and η is never reserved by a thread (recall there is exactly one more node than there are threads in the system). Finally, as shown in (10), the relationship between the abstract queue q and the concrete state is that the list of nodes obtained by reading from the auxiliary head through the reserved nodes in the queue, $\eta \# \mathit{reserved}\langle\langle q \rangle\rangle$, is equal to taking the `prev` node of each thread in the queue, with `tail` appended to the end, $\texttt{prev}\langle\langle q \rangle\rangle \frown [\texttt{tail}]$. This relationship gives a direct, list-based approach to describing CLH's pseudo-linked list, and could form the "coupling invariant" in a data-refinement style proof [17,32,41]. From this we may derive the concrete representation of an empty queue: if `tail` is not reserved by any thread then the queue is empty.

$$\mathsf{invariant} \;\Rightarrow\; \texttt{tail} = \eta \Leftrightarrow q = [\,]$$

Contract between threads. The contract amounts to not changing the local variables and array indexes assigned to a particular thread i (we use $\mathrm{id}(x, y, \ldots)$ for the relation that does not alter x, y, etc.); and additionally, that the `status` of the *reserved* node is never changed (establishing this key fact without an explicit *reserved* variable proved cumbersome). The contract also contains those parts already described in the specification of a queued lock from Fig. 2: a thread cannot be entered into or removed from the queue by another thread, and a waiting thread only ever gets closer to the front of the queue.

Properties of an abstract lock. The status of the system lock may be derived from the auxiliary queue as in Fig. 2: if the queue is empty no thread holds the lock, otherwise it is held by the first thread (12).

Relies and guarantees. Each thread i relies on its contract being fulfilled, and that the invariant is maintained; and conversely, guarantees to fulfil the contract for every other thread in the system and to maintain the invariant.

Establishing that the guarantee is maintained by each line of code is the most complex part of the proof; establishing the postconditions is relatively straightforward by careful choice of intermediate states.

Pre/postcondition and intermediate states for acquire. The precondition for $\mathsf{acquire}_\mathsf{i}()$ is that i is not already in the queue, and that the concrete `node` entry accurately holds i's unique reserved node. The intermediate states in Fig. 4 are relatively straightforward updates of this initial precondition based on the relevant assignment. At each point we track the *reserved* node for i, which is kept in $\mathtt{procs}[\mathsf{i}].\mathtt{node}$. Implicitly, each intermediate state preserves the invariant. The non-trivial part is establishing that the current thread is indeed the head of the abstract queue when the loop terminates; this is derivable from the invariant, specifically (10) and the status of waiting nodes.

$$\mathsf{invariant} \;\Rightarrow\; \mathsf{i} \in q \Rightarrow (\mathtt{prev}_\mathsf{i} \rightarrow \mathtt{status} = \mathsf{Granted} \Leftrightarrow \mathsf{i} = \mathsf{hd}(q)) \tag{13}$$

Pre/postcondition and intermediate states for release. The $\mathsf{release}_\mathsf{i}()$ operation assumes that i is at the head of the abstract queue, that $\mathtt{procs}[\mathsf{i}].\mathtt{node}$ holds the reserved node, and that $\mathtt{procs}[\mathsf{i}].\mathtt{tmp}$ points to the auxiliary head of the queue. After the first line of $\mathsf{release}_\mathsf{i}()$, thread i is no longer in the queue, and has taken the former head of the queue as its reserved node; and this node may be found in $\mathtt{procs}[\mathsf{i}].\mathtt{tmp}$ (one cannot access $\mathtt{prev}_\mathsf{i}$ directly since it is local to $\mathsf{acquire}_\mathsf{i}()$). The thread therefore updates $\mathtt{procs}[\mathsf{i}].\mathtt{node}$ to $\mathtt{procs}[\mathsf{i}].\mathtt{tmp}$, ready for the next call to $\mathsf{acquire}_\mathsf{i}()$. As with $\mathsf{acquire}_\mathsf{i}()$ the invariant is implicitly maintained.

4.3 Thread and system proofs

We now state the top-level theorems showing that the CLH-seL4 implementation is correct. All proofs have been encoded and machine-checked using Isabelle/HOL. We include an invariant in the quintuple statements (8), which adds the invariant to pre/postconditions and rely/guarantee relations.

Theorem 1 (Acquire).

$$\begin{array}{l}\{\mathsf{i} \notin q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \mathit{reserved}[\mathsf{i}], \mathsf{contract}_\mathsf{i}\} \\ \mathsf{acquire}_\mathsf{i}() /\!\!/ \mathsf{invariant} \\ \{\forall \mathsf{j} \neq \mathsf{i} \,.\, \mathsf{contract}_\mathsf{j}, \\ \quad \mathsf{i} \in q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \mathit{reserved}[\mathsf{i}] \wedge \mathtt{procs}[\mathsf{i}].\mathtt{tmp} = \eta \wedge \mathsf{i} = \mathsf{hd}(q)\}\end{array}$$

Theorem 2 (Release).

$$\begin{array}{l}\{\mathsf{i} \in q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \mathit{reserved}[\mathsf{i}] \wedge \mathtt{procs}[\mathsf{i}].\mathtt{tmp} = \eta \wedge \mathsf{i} = \mathsf{hd}(q), \\ \quad \mathsf{contract}_\mathsf{i}\} \\ \mathsf{release}_\mathsf{i}() /\!\!/ \mathsf{invariant} \\ \{\forall \mathsf{j} \neq \mathsf{i} \,.\, \mathsf{contract}_\mathsf{j}, \mathsf{i} \notin q \wedge \mathtt{procs}[\mathsf{i}].\mathtt{node} = \mathit{reserved}[\mathsf{i}]\}\end{array}$$

Proof. Individual lines of code are shown to satisfy their pre/post states outlined in Fig. 4 using the inference rules for assignments (Rule (1)), and spin

loops (Rule (7)), and are joined using Rule (2) using the relevant mid-states. The relies and guarantees are kept constant throughout. □

The top-level property for a system of N threads calling acquire and release repeatedly is given below. The initial state is derived from that given at the top of Fig. 4. The overall postcondition (we leave unconstrained the condition under which a thread may finish execution) shows that the queue is empty, though the rest of the state could have arbitrary values. Note that the guarantee $\forall\, \mathsf{i} \,.\, \mathsf{contract_i}$ is a simplification of $\forall\, \mathsf{i} \,.\, \forall\, \mathsf{j} \neq \mathsf{i} \,.\, \mathsf{contract_j}$. These simplifications of pre/postconditions and rely/guarantees follow from Rule (3).

Theorem 3 (Locking system).

$$\begin{array}{l} \{q = [\,] \wedge \mathit{reserved} = \ldots, \forall\, \mathsf{i} \,.\, \mathsf{contract_i}\} \\ \|_{\mathsf{i}} \,.\, \mathsf{while}\ \_\ \mathsf{do}\ (\mathsf{acquire_i}()\,;\ \mathsf{release_i}())\ /\!\!/\ \mathsf{invariant} \\ \{\forall\, \mathsf{i} \,.\, \mathsf{contract_i}, q = [\,] \wedge \ldots\} \end{array}$$

Proof. We apply Rule (6): each individual thread satisfies its own part from Theorems 1 and 2 (and that the postcondition of $\mathsf{acquire_i}()$ is the precondition of $\mathsf{release_i}()$, using Rule (2)); careful choice of rely_i and guar_i from Fig. 4 ensure that each thread's guarantee fulfils the rely of every other thread. □

For simplicity, Theorem 3 explicitly uses the contract between individual threads; however, from the perspective of a system that incorporates the lock, the rely of the lock system is simply that none of its variables are modified, and conversely the lock system guarantees to change nothing other than its own variables. These are both implied by $\forall\, \mathsf{i} \,.\, \mathsf{contract_i}$.

That the lock itself is used in such a way as to guarantee mutual exclusion to some system variable x is something that must be shown for each client system, i.e., that x is only ever modified between calls to $\mathsf{acquire_i}()$ and $\mathsf{release_i}()$.

5 CLH-seL4 with Weak Memory

The CLH lock may be deployed at the lowest level of the software stack, where there is little abstraction from hardware-level concerns. Chief among these are *weak memory effects*, which are primarily caused by *out-of-order execution*, a feature of processors since the 1960s [45,46].[4] Given consecutive instructions that are "independent", it may be more efficient to execute them in parallel, rather than to wait for the first to complete before the second begins. Such parallelism can be achieved, for instance, by utilising multiple on-chip arithmetic units for instructions that access different sets of registers.

The key principle of out-of-order execution is that accesses to the same variable must be kept in order, while independent instructions (e.g., accessing different parts of memory) may be reordered. On a single processor, this principle maintains the correctness of sequential code while improving performance by

[4] Another cause is *forwarding* [1,3] (also called *bypassing*), where a value calculated in an earlier instruction can be used to partially evaluate a later instruction. Forwarding also needs to be addressed in general, but it does not impact on CLH.

reducing unused processor cycles. However, such parallelism is observable on modern multicore architectures, with potentially serious effects on the correctness of concurrent programs, which typically require a precise order of execution for inter-thread communication.

5.1 Identifying Weak Memory Effects

We identify the weak memory effects in CLH using the analysis technique of [3], which searches for sources of out-of-order execution via a pointwise comparison of instructions. Where micro-parallelism is found, either i) the code is transformed to make this parallelism explicit, where it does not affect correctness, or ii) fences or other ordering constraints are inserted to enforce the required ordering.

For a given memory model M (which can be instantiated to memory models such as Arm, x86, RISC-V, C, etc.), we write $\alpha \overset{M}{\Leftarrow} \beta$ if instruction β may be reordered with instruction α under M. Conversely, we write $\alpha \overset{M}{\not\Leftarrow} \beta$ if they may not be reordered, because, for instance, they access the same variables, or involve artificial ordering constraints such as fences.

The behaviour of a program c is viewed as the set of possible traces over the instructions in c. To determine the behaviour of c under some memory model M, we interpret each sequential composition (;) in c as a *parallelised sequential composition* that is parameterised by M ($\overset{M}{;}$). Then, given a pair of instructions composed in this way, $\alpha \overset{M}{;} \beta$, if α and β cannot be reordered according to M we replace '$\overset{M}{;}$' with ';', since they will be executed in order (14). If they can be reordered, we replace '$\overset{M}{;}$' with '$\|$', since they can be executed in either order (15).

$$\alpha \overset{M}{\not\Leftarrow} \beta \quad \Rightarrow \quad \alpha \overset{M}{;} \beta = \alpha \,;\, \beta \tag{14}$$

$$\alpha \overset{M}{\Leftarrow} \beta \quad \Rightarrow \quad \alpha \overset{M}{;} \beta = \alpha \parallel \beta \tag{15}$$

Rule (15) introduces micro-parallelism within a process.

Most hardware platforms include some sort of fence instruction (also called *barriers*), which forbids reordering. For instance, the following rule applies specifically to full fences, but the principle applies to other ordering constraints, such as C's `memory_order` type.

$$c_1 \overset{M}{;} \mathsf{fence} \overset{M}{;} c_2 \quad = \quad c_1 \,;\, c_2 \tag{16}$$

Further rules can be derived for more complex program structures, including conditionals and loops [3,4]. As an example, consider five instructions running under some memory model M, $i_1 \overset{M}{;} i_2 \overset{M}{;} i_3 \overset{M}{;} i_4 \overset{M}{;} i_5$, and assume that an analysis of the potential reorderings reveals the following.

$$i_1 \overset{M}{\not\Leftarrow} (i_2 \overset{M}{\Leftarrow} i_3) \overset{M}{\not\Leftarrow} (i_4 \overset{M}{\Leftarrow} i_5) \tag{17}$$

That is, both i_2 and i_3 must be executed later than i_1, but i_2 and i_3 may be parallelised; furthermore, both i_4 and i_5 must take place later than i_2 and i_3,

but i_4 and i_5 themselves may be executed in either order. Applying rules (14) and (15) with reference to (17) we obtain the following.

$$i_1 \overset{\text{M}}{;} i_2 \overset{\text{M}}{;} i_3 \overset{\text{M}}{;} i_4 \overset{\text{M}}{;} i_5 \quad = \quad i_1 \,;\, (i_2 \parallel i_3) \,;\, (i_4 \parallel i_5) \tag{18}$$

We have replaced references to the memory model with explicit sequential or parallel execution, and can now apply standard reasoning techniques to the transformed code. This example corresponds approximately to the structure of acquire in Sect. 5.2, where M is instantiated to ARM.

Introducing extra parallelism can potentially invalidate previous correctness arguments. This can be determined via rely/guarantee reasoning; that is, if $\{p, r\}\ i_1 \,;\ i_2\ \{g, q\}$ had earlier been proved, then $\{p, r\}\ i_1 \parallel i_2\ \{g, q\}$ must now be shown. This may be complicated by the case where the midstate for the original sequential proof was essential to discharging the proof obligation for i_2, or where i_1 or i_2 modify variables the other is reading.

5.2 Weak Memory Effects in Acquire

We now apply the reasoning method of Sect. 5.1 to address the weak memory effects of the CLH lock. As the reasoning method is based on traces of atomic instructions, the atomicity of the CLH code must be precisely considered. Therefore we split the command $\texttt{procs[i].node} \rightarrow \texttt{status} := \mathsf{Pending}$ into its two atomic accesses of main memory: $\texttt{r} := \texttt{procs[i].node}\,;\ \texttt{r} \rightarrow \texttt{status} := \mathsf{Pending}$, where r is a local variable. Taking this into account the five individual lines of acquire from Fig. 3 are restated below. They are grouped according to relevant potential reorderings, calculated based on the memory model for Arm that was experimentally validated in [3] and adapted for pointers and arrays as in [5].

$$\begin{array}{l} \texttt{r} := \texttt{procs[i].node} \overset{\text{ARM}}{\not\Longleftarrow} \\ \qquad (\texttt{r} \rightarrow \texttt{status} := \mathsf{Pending} \overset{\text{ARM}}{\Longleftarrow} \texttt{prev} := \texttt{xchg}(\texttt{\&tail}, \texttt{r})) \\ \texttt{prev} := \texttt{xchg}(\texttt{\&tail}, \texttt{r}) \overset{\text{ARM}}{\not\Longleftarrow} \\ \qquad (\texttt{procs[i].tmp} := \texttt{prev} \overset{\text{ARM}}{\Longleftarrow} \mathsf{await}\ \texttt{prev} \rightarrow \texttt{status} = \mathsf{Granted}) \end{array} \tag{19}$$

We abbreviate the above to $i_1 \overset{\text{ARM}}{\not\Longleftarrow} (i_2 \overset{\text{ARM}}{\Longleftarrow} i_3)$ and $i_3 \overset{\text{ARM}}{\not\Longleftarrow} (i_4 \overset{\text{ARM}}{\Longleftarrow} i_5)$. The parentheses state that neither i_2 nor i_3 can be reordered with i_1 according to ARM, but i_3 can be reordered with i_2. Similarly, neither i_4 nor i_5 can be reordered with i_3, but i_4 may be reordered with i_5. As will be explained later, the relationships between i_1 & i_2 and i_4 & i_5 are not relevant. We now examine the reasons behind the reordering relationships and their consequences.

Set-up and joining the queue. In (19), the instructions i_1 and i_2 may not be reordered, as i_1 modifies r, and i_2 reads r (the expression $\texttt{r} \rightarrow \texttt{status}$ in i_2 implicitly reads the address stored in r). Instruction i_3 reads r and modifies prev and tail, so i_3 cannot reorder with i_1, but may reorder with i_2. Hence, $i_1 \overset{\text{ARM}}{;} i_2 \overset{\text{ARM}}{;} i_3$ is trace-equivalent to $i_1 \,;\, (i_2 \parallel i_3)$.

The potential for parallelism between i_2 and i_3 affects the program's correctness. If the `xchg` instruction i_3 occurred first, this would violate the part of the invariant that requires all queuing nodes to have status Pending, and thus invalidate the key property of (13). Therefore, for this code to execute correctly under ARM, we must explicitly enforce ordering between the setting of `status` in i_2 and the `xchg` of i_3. On Arm platforms, ordering can be enforced by inserting a fence such as `dsb`, or by placing a *release* ordering constraint [14] (REL) on the second store operation.

Recording `prev` and spinning. Instruction i_4 (`procs[i].tmp := prev`) reads `prev`, and the value of `prev` is set by i_3 (the `xchg`), so $i_3 \overset{\text{ARM}}{\not\Longleftarrow} i_4$.

Command i_5 is an await, consisting of a sequence of loads of `prev → status`. All of these loads access `prev`, which is set by i_3, and hence must occur later than i_3, i.e., $i_3 \overset{\text{ARM}}{\not\Longleftarrow} i_5$. However, they are all independent of i_4, which reads but does not modify `prev`; hence $i_4 \overset{\text{ARM}}{\Longleftarrow} i_5$. We may therefore show that $i_3 \overset{\text{ARM}}{;} i_4 \overset{\text{ARM}}{;} i_5$ is trace-equivalent to $i_3 ; (i_4 \parallel i_5)$.

In contrast to the potential reordering between i_2 and i_3 which would break the relevant correctness argument, the parallelism between i_4 and i_5 makes no difference, and hence there is no need to enforce the original ordering. Formally, this is because the parallel composition $i_4 \parallel i_5$ satisfies the same rely/guarantee quintuple as $i_4 ; i_5$.

Theorem 4 (Safe reordering of the spin loop in Acquire).

$$\begin{array}{l}\{\mathsf{i} \in q \wedge \texttt{procs}[\mathsf{i}].\texttt{node} = \mathit{reserved}[\mathsf{i}], \mathsf{contract}_\mathsf{i}\} \\ \texttt{procs}[\mathsf{i}].\texttt{tmp} := \texttt{prev} \parallel \textsf{await}\ \texttt{prev} \to \texttt{status} = \mathsf{Granted} \mathbin{/\!/} \textsf{invariant} \\ \quad \{\forall \mathsf{j} \neq \mathsf{i} \,.\, \mathsf{contract}_\mathsf{j}, \\ \quad\ \mathsf{i} \in q \wedge \texttt{procs}[\mathsf{i}].\texttt{node} = \mathit{reserved}[\mathsf{i}] \wedge \texttt{procs}[\mathsf{i}].\texttt{tmp} = \eta \wedge \mathsf{i} = \mathsf{hd}(q)\}\end{array}$$

Proof. For such proofs involving nested parallelism, we apply Rule (4) (as opposed to Rule (5)). In this case, the left-hand side of the parallel composition (`procs[i].tmp := prev`) causes more interference on the loop than we had taken into account in Fig. 4. However, Rule (7) still applies – the extra interference by the potential change to `procs[i].tmp` does not compromise the stability of the pre/postconditions. Furthermore, being a spin loop, the right-hand side of the parallel composition does not inflict any more interference on the left-hand side assignment than had previously been accounted for. □

Acquire in full. The acquire code running under Arm, $\mathsf{acquire}_\mathsf{i}^{\text{SEL4}}$, is obtained by replacing sequential compositions with '$\overset{\text{ARM}}{;}$', and injecting a REL constraint on i_3. By the reasoning above, this is trace equivalent to the the program $\mathsf{acquire}_\mathsf{i}^{\parallel}$. We summarise this below, where await `prev → status = Granted` is abbreviated by wait.

$$\begin{array}{rcl}\mathsf{acquire}_\mathsf{i}^{\text{SEL4}} & \widehat{=} & \mathtt{r} := \texttt{procs}[\mathsf{i}].\texttt{node} \overset{\text{ARM}}{;} \mathtt{r} \to \texttt{status} := \mathsf{Pending} \overset{\text{ARM}}{;} \\ & & \texttt{prev} := \texttt{xchg}(\texttt{\&tail}, \mathtt{r})^{\text{REL}} \overset{\text{ARM}}{;} \texttt{procs}[\mathsf{i}].\texttt{tmp} := \texttt{prev} \overset{\text{ARM}}{;} \textsf{wait} \\ \mathsf{acquire}_\mathsf{i}^{\parallel} & \widehat{=} & \mathtt{r} := \texttt{procs}[\mathsf{i}].\texttt{node} ; \mathtt{r} \to \texttt{status} := \mathsf{Pending} ; \\ & & \texttt{prev} := \texttt{xchg}(\texttt{\&tail}, \mathtt{r}) ; (\texttt{procs}[\mathsf{i}].\texttt{tmp} := \texttt{prev} \parallel \textsf{wait})\end{array}$$

Theorem 5 (Acquire under ARM). *By trace-equivalence modulo fences and* REL *ordering constraints,* $\mathsf{acquire}_\mathsf{i}^{\text{SEL4}} = \mathsf{acquire}_\mathsf{i}^{\|}$.

Proof. The proof is structurally similar to the refinement in (18), but with $i_2 \overset{\text{ARM}}{\not\Longleftarrow} i_3$ (due to the injected REL constraint), which results in a sequential composition. □

5.3 Weak Memory Effects in Release

The two lines of code in release involve the loading of `procs`[i].`node` followed by the modification of the same; hence the key parts of the two instructions are kept strictly ordered by any memory model (breaking each of the two lines of code in release into their atomic assembler instructions introduces some parallelism as in acquire, but not for the key store and load to `procs`[i].`node`).

$$\texttt{procs}[\mathsf{i}].\texttt{node} \rightarrow \texttt{status} := \mathsf{Granted} \overset{\text{ARM}}{\not\Longleftarrow} \texttt{procs}[\mathsf{i}].\texttt{node} := \texttt{procs}[\mathsf{i}].\texttt{tmp} \qquad (20)$$

As such, the sequential composition in release may be treated as strict.

Theorem 6 (Release under ARM). $\mathsf{release}_\mathsf{i}^{\text{SEL4}} = \mathsf{release}_\mathsf{i}()$. □

Proof. Immediate from (20) and (14). □

5.4 Reordering with the Critical Section

The reasoning above is sufficient for considering the code of each operation in isolation; however, further reorderings with code between the calls to acquire and release are possible and need to be considered.

Before acquire has fully completed or while release is still being executed, in principle there should not be any access to the shared variables that are protected by the lock. As outlined in Sect. 3.1, only after the final instruction of acquire does a thread detect that it has the lock, while the first instruction of release signals that a thread no longer holds the lock.

Nevertheless, in a calling context $\mathsf{acquire}_\mathsf{i}() \overset{\text{ARM}}{;} \mathsf{crit}_\mathsf{i}(x) \overset{\text{ARM}}{;} \mathsf{release}_\mathsf{i}()$, memory accesses in $\mathsf{crit}_\mathsf{i}(x)$ may reorder with instructions in $\mathsf{acquire}_\mathsf{i}()$ and $\mathsf{release}_\mathsf{i}()$. To mitigate this behaviour, the simplest approach is to append a full fence to acquire (in Arm, the `dsb` instruction would suffice), and similarly to prepend a full fence to release. This leads to the following.

$$\mathsf{acquire}_\mathsf{i}() \overset{\text{ARM}}{;} \mathsf{crit}_\mathsf{i}(x) \overset{\text{ARM}}{;} \mathsf{release}_\mathsf{i}() \quad = \quad \mathsf{acquire}_\mathsf{i}() \,;\, \mathsf{crit}_\mathsf{i}(x) \,;\, \mathsf{release}_\mathsf{i}()$$

The exact types of the artificial constraints that are needed after acquire and before release depend on the memory model. For instance, Arm already prevents later stores from executing before a branch point (contained in the loop guard), so one can use a constraint weaker than the full fence.

In contrast to the critical section, code prior to a call to acquire need not be fenced, under the reasonable assumption that no code other than that of acquire

and release modifies the lock-specific variables in Fig. 3, as discussed in Sect. 3. Similar reasoning holds for non-critical code after a call to release.

We must also consider the case where thread i attempts to reacquire the lock after releasing (across a non-critical section of code). In this case, the last instruction of release modifies `procs`[i].`node`, while the first instruction of acquire reads `procs`[i].`node`. Hence, reordering is not possible, i.e., $\mathsf{release_i}() \overset{\text{ARM}}{;} \mathsf{acquire_i}() = \mathsf{release_i}() ; \mathsf{acquire_i}()$.

5.5 Top-level proof

Theorem 7 (Correctness of CLH under ARM). *The implementation of CLH in Fig. 3, updated with a* REL *constraint as in Theorem 1, and augmented with* `dsb` *barriers (or equivalent) at the end of* acquire *and at the beginning of* release*, works as a lock under* ARM*.*

Proof. By Theorem 1, Theorem 4, and Theorem 5, the implementation of acquire satisfies the pre and postconditions of Fig. 4; and similarly for the implementation of release by Theorem 2 and Theorem 6. These may be composed identically to Theorem 3. The definition and validity of the interpretation of the Arm memory model is taken from [3]. □

Taking into account weak memory models (in this case, Arm, though the reasoning applies to essentially any real model, such as RISC-V and x86-TSO [3]), we find that artificial orderings must be inserted to enforce the order of the initialisation of the status of the local node to Pending and the atomic exchange; however, the standard constraints of standard memory models either enforce ordering of statements, or, in the case of the loop, the program order of the statements is not strictly necessary for correctness.

5.6 Model Checking termination and liveness under ARM

To model check termination and system-wide progress of CLH we encoded the operations of Fig. 3 in SPIN [21]. We confirmed that, without weak memory effects, small, finite-size systems terminate and enforce mutual exclusion.

Re-using Theorems 5 and 6, we also model-checked the transformed version of CLH under the ARM memory model, and again confirmed that finite-size systems terminate and enforce mutual exclusion. In SPIN, the encoding of the nested parallelism involving a loop was inelegant but tractable for code of this size. To handle nested parallelism more naturally, one could employ a refinement-based model checker such as FDR [15].

6 Related work

The verification of concurrent algorithms, including locks [19,26,42], has a long history comprising a variety of techniques and tools. Approaches that have been

applied specifically to CLH include several works which have focussed on termination/liveness properties using separation logics [12,29,40]. The application of the Starling framework to CLH [47] focusses on functional correctness, with an underlying technique based on Owicki-Gries reasoning [37]. It is unclear how these works can be adapted to handle the extra complexity associated with weak memory concerns, which must be taken into account when considering deployment of CLH on modern devices. As we show in Sect. 5, some but not all of the micro-parallelism that multicore architectures introduce can be retained. A verification approach that insists on restoring full sequential ordering would therefore introduce unnecessary inefficiencies associated with the overuse of fences. By basing our approach on Jones' rely/guarantee reasoning we get compositionality – helpful for managing micro-parallelism of weak memory effects – and an existing machine-checked theory base in Isabelle/HOL [39].

If we were instead to attempt to verify CLH directly using a weak-memory-specific technique such as [10,11,25] we would typically need to rework the verification conditions (pre/postconditions) into a technique-specific assertion language, designed to capture weak memory effects, and apply technique-specific inference rules. Our approach to verification is based on [3,4,5], where weak memory effects are elucidated separately to the main verification process, and we inherit the benefits of separating the two analyses. For instance, we reused Theorems 5 and 6 to apply model checking for experimental evidence that the micro-parallelism introduced by Arm processors does not affect termination (see Sect. 5.6). The approach of [7] is based on a similar semantic model to ours, but is, again, specific to a particular verification technique; however, it has been adapted to handle the Power architecture, which includes other micro-architectural features that complicate correctness [8]. The separation-of-concerns approach has a further advantage, particularly important in our case, in that the inclusion of auxiliary variables does not complicate the reasoning about weak memory effects.

7 Conclusions

We have shown an end-to-end machine-checked verification of a complex, practical, in-use concurrent algorithm for multicore architectures such as Arm. The approach is centred on Jones' rely/guarantee concept, which is suitable for describing top-level specifications (Sect. 3), detailed proofs of individual threads (Sect. 4), and compositional reasoning about inter- and intra-thread parallelism (Sect. 5). We utilise and specialise mature tool support for theorem proving using rely/guarantee concepts [39,35,38] (theory files available on request).

In constructing the proof in Sect. 4 we also proved several simpler algorithms, including: abstract versions of locks, one with only a single *lock* variable, and one with an explicit *q* variable as in Sect. 3; two versions of an array-based queue lock (which suffers from space considerations when multiple locks are used within a system); and an alternative version of CLH given by Scott [43]. In this last proof we were able to reuse many of the results from Sect. 4, and to show that both

implementations are *compatible*, in the sense that different threads could use the different implementations without clashing. This reuse is owed in large part to abstracting the details in auxiliary variables and the inherent compositionality of rely/guarantee reasoning, as well as separating the weak memory effect reasoning (again, much of which was reused). A more typical approach is to use program counters or the application of Owicki-Gries, both of which would make proof reuse across different implementations much harder.

The proofs of Theorems 1-6 have been machine-checked in Isabelle/HOL (a full formalisation of the statement of Theorem 7 is ongoing work). We are currently working towards the automation of the program-level reasoning in Sect. 5 and tool support for *derivational* rely/guarantee proofs [16,6].

Acknowledgements. We thank Cliff Jones for many conversations, theoretical and practical, about the foundations and applications of rely/guarantee over many years. It is a testament to the fundamental nature of Cliff's rely/guarantee reasoning framework that it is highly relevant and applicable to modern multicore architectures, even though these did not become commonplace until decades after his original contribution.

This work was conducted in partnership with the Defence Science & Technology Group through the Next Generation Technologies Fund. We thank David Gwynne for help with understanding the efficiency implications of CLH.

References

1. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (Jul 2014). https://doi.org/10.1145/2627752
2. Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. J. Log. Comput. **17**(4), 807–841 (2007). https://doi.org/10.1093/logcom/exm030
3. Colvin, R.J.: Parallelized sequential composition and hardware weak memory models. In: SEFM. pp. 201–221 (2021). https://doi.org/10.1007/978-3-030-92124-8_12
4. Colvin, R.J.: Separation of concerning things: A simpler basis for defining and programming with the C/C++ memory model. In: ICFEM. pp. 71–89 (2022). https://doi.org/10.1007/978-3-031-17244-1_5
5. Colvin, R.J.: A fine-grained semantics for arrays and pointers under weak memory models. In: FM. pp. 301–320 (2023). https://doi.org/10.1007/978-3-031-27481-7_18
6. Colvin, R.J., Hayes, I.J., Meinicke, L.A.: Designing a semantic model for a wide-spectrum language with concurrency. Form. Asp. Comput. **29**(5), 853–875 (Sep 2017). https://doi.org/10.1007/s00165-017-0416-4
7. Coughlin, N., Smith, G.: Compositional noninterference on hardware weak memory models. Sci. Comput. Program. **217**, 102779 (2022). https://doi.org/10.1016/j.scico.2022.102779
8. Coughlin, N., Winter, K., Smith, G.: Compositional reasoning for non-multicopy atomic architectures. Form. Asp. Comput. **35**(2) (2023). https://doi.org/10.1145/3574137

9. Craig, T.S.: Building FIFO and priority-queuing spin locks from atomic swap. Tech. Rep. 93-02-02, University of Washington (1993)
10. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: Compositional reasoning for concurrent programs. In: POPL. p. 287–300 (2013). https://doi.org/10.1145/2429069.2429104
11. Doherty, S., Dalvandi, S., Dongol, B., Wehrheim, H.: Unifying operational weak memory verification: An axiomatic approach. ACM Trans. Comput. Log. **23**(4) (2022). https://doi.org/10.1145/3545117
12. D'Osualdo, E., Sutherland, J., Farzan, A., Gardner, P.: TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs. ACM Trans. Program. Lang. Syst. **43**(4) (2021). https://doi.org/10.1145/3477082
13. Floyd, R.W.: Assigning meanings to programs. Math. Asp. Comput. Sci. **19**, 19–32 (1967)
14. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors. In: ISCA. pp. 15–26 (1990). https://doi.org/10.1145/325164.325102
15. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 — a modern refinement checker for CSP. In: TACAS. pp. 187–201 (2014). https://doi.org/10.1007/978-3-642-54862-8_13
16. Hayes, I.J., Colvin, R.J., Meinicke, L.A., Winter, K., Velykis, A.: An algebra of synchronous atomic steps. In: FM. pp. 352–369 (2016). https://doi.org/10.1007/978-3-319-48989-6_22
17. He, J., Hoare, C., Sanders, J.: Data refinement refined (resume). In: ESOP. pp. 187–196 (1986). https://doi.org/10.1007/3-540-16442-1_14
18. Heiser, G., Klein, G., Andronick, J.: seL4 in Australia: From research to real-world trustworthy systems. Commun. ACM **63**(4), 72–75 (2020). https://doi.org/10.1145/3378426
19. Hesselink, W.H.: Mechanical verification of Lamport's Bakery algorithm. Sci. Comput. Program. **78**(9), 1622–1638 (2013). https://doi.org/10.1016/j.scico.2013.03.003
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
21. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
22. Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. Ph.D. thesis, Oxford University (1981)
23. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress. pp. 321–332 (1983)
24. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983). https://doi.org/10.1145/69575.69577
25. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL. pp. 175–189 (2017). https://doi.org/10.1145/3009837.3009850
26. Kim, J., Sjöberg, V., Gu, R., Shao, Z.: Safety and liveness of MCS lock—layer by layer. In: APLAS. pp. 273–297 (2017). https://doi.org/10.1007/978-3-319-71237-6_14
27. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an os microkernel. ACM Trans. Comput. Syst. **32**(1) (2014). https://doi.org/10.1145/2560537

28. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: SOSP. pp. 207–220 (2009). https://doi.org/10.1145/1629575.1629596
29. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. In: POPL. pp. 385–399 (2016). https://doi.org/10.1145/2837614.2837635
30. Magnusson, P., Landin, A., Hagersten, E.: Efficient software synchronization on large cache coherent multiprocessors. Tech. Rep. T94-07, Swedish Inst. C. S. (1994)
31. Magnusson, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: IPPS. pp. 165–171 (1994). https://doi.org/10.1109/IPPS.1994.288305
32. Morgan, C., Gardiner, P.: Data refinement by calculation. Acta Inform. **27**, 481–503 (1990). https://doi.org/10.1007/BF00277386
33. Morgan, C.: Programming from Specifications. Prentice Hall, second edn. (1994)
34. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: seL4: From general purpose to a proof of information flow enforcement. In: SP. pp. 415–429 (2013). https://doi.org/10.1109/SP.2013.35
35. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
36. Nipkow, T., Prensa Nieto, L.: Owicki/Gries in Isabelle/HOL. In: FASE. pp. 188–203 (1999). https://doi.org/10.1007/978-3-540-49020-3_13
37. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inform. **6**(4), 319–340 (1976). https://doi.org/10.1007/BF00268134
38. Paulson, L.C.: Isabelle: a generic theorem prover. Springer (1994)
39. Prensa Nieto, L.: The rely-guarantee method in Isabelle/HOL. In: ESOP. pp. 348–362 (2003). https://doi.org/10.1007/3-540-36575-3_24
40. Reinhard, T., Jacobs, B.: Ghost signals: Verifying termination with busy waiting. In: CAV. pp. 27–50 (2021). https://doi.org/10.1007/978-3-030-81688-9_2
41. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press (2001)
42. Schellhorn, G., Travkin, O., Wehrheim, H.: Towards a thread-local proof technique for starvation freedom. In: IFM. pp. 193–209 (2016). https://doi.org/10.1007/978-3-319-33693-0_13
43. Scott, M.L.: Shared-Memory Synchronization. Synthesis Lectures on Computer Architecture, Morgan and Claypool (2013). https://doi.org/10.1007/978-3-031-01740-7
44. The seL4 Foundation: CLH lock in seL4. https://github.com/seL4/seL4/blob/master/include/smp/lock.h (2020)
45. Thornton, J.E.: Parallel operation in the Control Data 6600. In: Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems. p. 33–40. AFIPS '64 (1964). https://doi.org/10.1145/1464039.1464045
46. Tomasulo, R.M.: An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development **11**(1), 25–33 (1967)
47. Windsor, M., Dodds, M., Simner, B., Parkinson, M.J.: Starling: Lightweight concurrency verification with views. In: CAV. pp. 544–569 (2017). https://doi.org/10.1007/978-3-319-63387-9_27