Research in Computer Science Project Report

Verifying weak memory programs in the Viper ecosystem

Author:
Anouk Paradis
Master of Computer Science
ETH

Supervisors:
Gaurav Parthasarathy
PM Group
ETH

Pr. Alexander J.Summers
PM Group
ETH

September 16, 2019

Contents

Chapter 1

Introduction

When reasoning about concurrent programs, the most intuitive setting is sequential consistency [7], where we consider the program to behave as if each of the threads' executions were simply interleaved. However, due to the many optimizations performed by modern compilers and hardware, such a model is too simplistic to accurately represent the actual executions of the program. We can hence only assume a weak memory model, which allows many more possible executions of the accesses to shared memory locations. However, these new possibilities of execution make reasoning about programs more difficult, and at times counterintuitive. Writing correct programs is hence extremely challenging in such a setting.

FSL++ [5] is a program logic developed to reason about concurrent programs in C11, which are governed by the weak memory model defined in the C11 standard. This program logic allows for compositional proofs, through preconditions and postconditions for functions for instance. However, such proofs originally had to be encoded using the Coq interactive theorem prover, thus requiring a lot of ad hoc work for each new program considered.

To try to alleviate this proof burden, part of this program logic has been encoded into Viper [8] by Summers and Müller [10]. However it did not provide any encoding for the notion of custom *ghost state*, an important feature used to model information that cannot be deduced solely from the program state. This *ghost state* is defined quite generally in FSL++, and needs to be specialized for each example, which makes their encoding and their use quite challenging. Parthasarathy et al. [9] provided such a specialization, the EFC permission structure, and showed that it was sufficient to prove the correctness of a variety of example programs. Building on this work, Wiesmann et al. [11] was then able to define a specification syntax, token based reasoning, and its encoding in Viper, that allowed easier use of this specialization, without having to deal with lower level details. Wiesmann et al. [11] also implemented a C++ frontend for Viper using this specialization

to provide more automation for proofs.

This project explores this new infrastructure and try to define some of its limits, not only in terms of what it allows us to do, but also in how easily it allows us to do it. We were first able to identify a slight mismatch between the intuition of the EFC permission structure and token based reasoning. Furthermore, we found and studied three new real-world examples. The first two allowed us to identify strong limitations of FSL++ that forbid proving many kind of examples. We finally provide a proof of the last example, that uses two atomic locations, using the EFC permission structure.

Chapter 2

Background

2.1 Weak Memory

As explained in the introduction, we cannot expect a sequentially consistent behavior from any parallel program. We have to deal with all the behaviors introduced by weak memory. We will now explain in some more details how those work.

We consider two types of memory locations: atomics and non atomics. Non atomics are the default, the ones we use in most programs. However, they are not adapted for concurrent access: if two threads try to access a non atomic location at the same time, and at least one of them intends to write, there is a data race, and the behavior of the program is undefined. Hence we cannot use this kind of locations in a concurrent environment without making sure that no such racy access can happen.

The required synchronization is done using atomics. Operations on those locations are atomic: we can think of them as happening in an instant, so that nothing can happen at the same time (since they took so little time). Hence the race situation described above, with a thread reading from an atomic location at the same time as another thread is writing to it, cannot happen. As both operations are atomic, either the read happens first, or the write does. However, those atomic operations are more expensive than non atomic ones. Hence we only use them to create the necessary synchronization to avoid data races.

2.1.1 Synchronization on atomics

While atomics are unaffected by races, we still cannot trust them for sequential consistency. Due to potential reorderings by the compiler or the processor, or delayed synchronization of different caches, `a = 1; b = 1` may become `b = 1, a = 1`. We hence need some construct to enforce ordering between memory accesses, be them atomic or non atomic. We use fences for that. We here only mention three of them: sequentially consistent (SC)

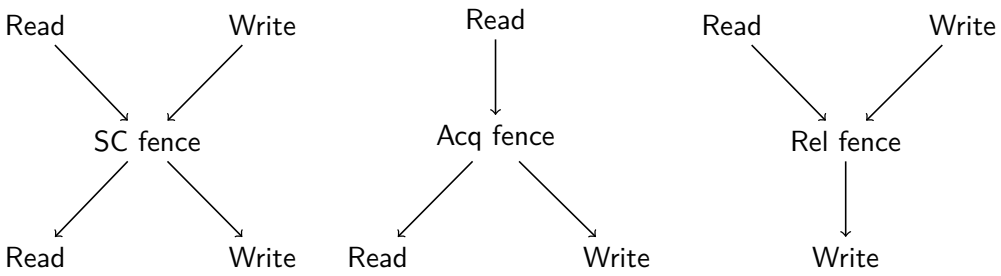

Figure 2.1: Fences synchronization

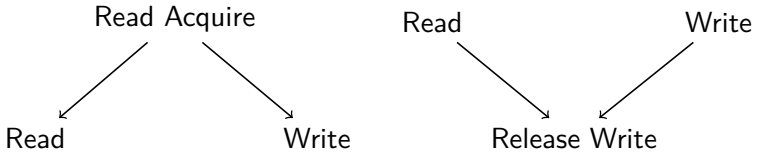

Figure 2.2: Release writes and acquire reads synchronization

ones, acquire (Acq) ones and release (Rel) ones. The synchronization they offer is summarized in Figure 2.1. The sequentially consistent fence is the stronger of them all: it forbids any reodering around it. However, it is quite expensive. Hence, most of the time we try to use weaker fences, that take less time to enforce. The release fence ensures that any read or write before it will not be re-ordered after the writes after it. Hence reading a value written by a write after this fence guarantees that all memory operations that happened before this fence are visible as well: they were *released* with the write. The acquire fence ensures that any read before it will not be re-ordered with any read or write after it.

The easiest way to think about the way those fences work, is to consider that the order in which different operations appear in the program have no impact on their actual execution order unless we use some special construct that enforces an order. This is kind of the way the *happen-before* relation defined in the C11 standard works [6]: we assume no order, and define using fences, operations reading on the same variable, ... a transitive relation that explicits the order that is preserved during the actual execution. Using this way of thinking, a release array creates arrows from all read and writes above it to itself, and from itself to all writes below it.

On top of the fences, we can enforce synchronization by doing a *release write* or an *acquire read*. Here the synchronization is slightly weaker than when using a fence, as shown on Figure 2.2. For instance, the release write only forbids reordering of previous read or writes with itself, and not with all writes after itself.

2.1.2 Data races

Using the *happen before* relation we gave an outline of above, it is easy to get a nicer intuition of data races: an program is racy if it contains two non atomic operations on a non atomic location, one of which is a write, that are not ordered with happen before.

Note that the bits of definition of *happen before* given here are in no way formal, and simply aims at giving a better intuition of the synchronization mechanisms we will study in this report.

2.2 Formalizing weak memory

We gave above some intuition of synchronization. However, this is far from enough to build any proof for a program, let alone build a verification tool. Quite a few formalization have been proposed as a formal definition of what can happen in this context. We will here use FSL++[5]. This separation logic formalizes a strengthening of what the C11 standard describes. The details of this strengthening are discussed in [5].

Let us first give an intuition of how this logic works. One further simplification imposed by this logic is that all locations are either atomic or non atomic (whereas in C++ a cast could change that).

2.2.1 Atomics as exchange points

As we explained in the previous section, we usually use atomics to protect non atomic locations from data races. Atomics are simply here to transfer some information between threads, to ensure proper synchronization, and let non atomics do the actual program work. This is for instance what we do if we build a parallel program using locks: the atomics within the locks enforce proper synchronization, and we use the rest of the variables to do all the computations we are interested in.

FSL++ builds on this idea of atomics as mere signal senders between threads. For each atomic variable, one can define a location invariant $\mathcal{Q} : v \mapsto \mathcal{Q}(v)$. This invariant maps each value that can be read or stored in the atomic variable to some predicate. Now when writing to an atomic location a value v, a thread has to give up $\mathcal{Q}(v)$, whereas when it reads v from this location, it gains $\mathcal{Q}(v)$. Some more rules are needed to ensure that when reading twice the same value from a location, we do not get twice the predicate. These rules are explained in more details in the parts of this report they are relevant to.

2.2.2 Ownership

We just saw that in FSL++ we use atomic locations to exchange predicates between threads, but what kind of predicates? We here introduce the notion

of *ownership*. When a new non atomic location l is allocated by a thread, the thread get full ownership of it, noted l^1. This full ownership allows the thread to read or write to the location. This ownership can then be transferred to another thread through an atomic location for example. It could also be split in two (or more) parts: $l^{0.5} * l^{0.5}$. Some of this fractional permission can then be sent to another thread, while the other part of it is kept. With this partial permission, a thread can only read from the non atomic location l, so as to forbid data races. As the total amount of ownership is 1, and ownership cannot be created except when a variable is allocated, there can always be at most one thread having full permission, and if this is the case no other thread has any amount of permission. Hence we make sure that at most one thread can write to a variable at any given point, and if this is the case, no other thread can read concurrently to the variable.

In [5], a notion of modality is also defined, to properly account for fences, however in this report, we never need to use those modalities, as synchronization always happen in such a way that no modality is present in the proof presented. We will hence not explain more about them here.

2.2.3 Ghost states

On top of ownership, the predicates stored in the location invariants of atomic locations can contains reference to a *ghost state*. A ghost state is simply a special variable, that we use as a helper in the proof, to model some information that cannot be translated simply using ownership. When defining this special variable, we also define the domain in which it takes its values. Carefully choosing this domain allows to model complex protocols, that could not be modeled using FSL++ otherwise. In this report, we will only use one kind of ghost state domain, the entity fractional-counting (EFC) permission structure. It was defined in [9], and allows to prove with few ghost locations the examples that were studied in [9].

The EFC permission structure is defined as

$$\left((\mathbb{N}_{>0} \times \mathbb{Q}_{>0} \times \{-,+\}) \cup \left\{(a,0)^+, (a,0)^- | a \in \mathbb{N}\right\}, \oplus, (0,0)^+, (0,0)^-\right)$$

with the partial commutative and associative operation $\oplus$ defined as

$$(c,d)^+ \oplus (a,b)^- := (a,b)^- \oplus (c,d)^+ := \begin{cases} (a-c, b-d)^- & \text{if } a-c \geq 0 \text{ and } b-d \geq 0 \text{ and } (a-c=0 \Rightarrow b-d=0) \\ \text{undefined} & \textit{otherwise} \end{cases}$$

$$(a,b)^+ \oplus (c,d)^+ := (a+c, b+d)$$

$$(a,b)^- \oplus (c,d)^- := \text{undefined}$$

The idea of this permission structure is to have a single source permission $(c, s)^-$, and some amount of tokens $(1, s)^+$. The operation $\oplus$ being undefined for two values with $-$ ensures that there can only be one source permission. The idea of this permission structure is to correspond to the intuition one can have of an atomic location l protecting a non atomic location d. We can then have in the location invariant of l the source permission, while threads that change the value of the atomic location gain or give up some tokens. The tokens are usually of the form $(1, s)^+$: they are hence unsplitable (because of the 1), and s usually specifies the amount of permission the thread owns to d. For instance a token $(1, 1)^+$ specifies write permission. As $\oplus$ is not defined for some values, a thread owning $(1, 1)^+$ ensures that the source must be $(c, 1)^-$, recording the fact that there is no permission left.

We explained above the idea of the tokens being related to an amount of permission. The tokens themselves do not give any permission to a thread, by defining carefully the invariant for l, we can make sure that a thread can only gain permission at the same time as gaining a token. This can then be used in the proof, by reasoning on the values of tokens and sources, to prove that the program respects the invariant.

2.2.4 Token-based reasoning in Viper

In [11], Wiesman et al. abstracted some of the complexity of FSL++ and the EFC permission structure, to be able to implement a tool that could be used in Viper, and automate as much as possible of the reasoning, for concurrent C++ programs. Token based reasoning unifies the permission to a location d^s, and the token $(1, s)^+$, into a new kind of token. It also removes the exact amount of the permission s, replacing it with either *write, read* or *none*. The possible tokes are hence

$$\texttt{Tok}\ (\text{loc}, n, \tau) \text{ where } n \in \mathbb{N}_{>0} \text{ and } \tau \in \{none, read, write\}$$

and the source is now

$$\texttt{Src}(loc, n, \tau) \text{ where } n \in \mathbb{N} \geq 0, \text{ and } \tau \in \{none, read, write\}$$

This allows for simpler automation rules, at the price of some approximation, as we will see in Section 3.

Chapter 3

Ease of use limitations of the EFC ghost location and tokens

In this chapter, we will use the Folly library reader writer lock [2] to explicit some limitations of the EFC monoid defined in [9] and the token-based reasoning developed in [11]. While, as we will see, those limitations can easily be overcome using some simple tricks, they make reasoning less intuitive, and those tools harder to use.

3.1 The Folly reader-writer spinlock

The Folly reader-writer lock is used to allow multiple threads to access concurrently a shared resource *res*, with either reader or writer privilege. This lock allows multiple reader threads to access *res* concurrently, but makes sure that if a thread has write access, all other threads are forbidden from reading or writing to it. The (simplified) implementation of this lock is shown in Figure 3.1. This implementation is taken from [9].

The idea of the implementation is quite simple. It uses an atomic location *bits*. The least significant bit of *bits* (which we will denote as *lsb(bits)*)indicates whether or not there is currently a thread with write access, while *bits / 4* indicates the number of threads currently having or attempting to have a read access. When a thread wanting a read access to *res* calls the function `try_lock_shared()`, it first increments *bits* by 4. Then it checks for the value *lsb(bits)* had when it was incremented by 4. If it is null, then there is currently no writer thread. The call to `try_lock_shared` succeeded, and the calling thread got read access. If it is not null, the thread decrements *bits* by 4 and the call failed: no access is gained. To release a reader lock, a thread simply decrements the *bits* variable by 4, using the `unlock_shared` function. To get a write-lock, a thread calls `try_lock`,

```
  bool try_lock_shared ()
{
  v0:=fetch_and_add_acq(bits,4)//RMW1
  if(lsb(v0) == 1) {
    v1:=fetch_and_add_rel(bits,4)//RMW2
    res := false
  }
  else {
    res := true
  }
  return res
}

void unlock_shared () {
  x:=fetch_and_add_rel(bits,4)
}

bool try_lock () {
  v:=CAS_rel_acq(bits,0,1)
  return (v == 0)
}

void unlock(bool getRead) {
  x:=fetch_and_and_rel(bits,1)
}

void unlock_and_lock_shared () {
  x:=fetch_and_add_acq(bits,4)//RMW1
  unlock(true)
}
```

Figure 3.1: Folly RW lock implementation (from [9])

which simply atomically checks if *bits* is 0 (that is to say there is no reader thread or thread currently attempting to get read access), and if so changes it to 1. Finally, to release a writer-lock, *lsb(bits)* is simply set to 0.

3.2 Proof using the EFC permission structure

As explained in more detail in Chapter 2, the EFC permission structure is based on the idea of giving entities, containing some amount of permission and a token, to each thread that asks for it. We then record the number of tokens, and the total amount of permission that were distributed, to be able to track if full permission is available or not.

Hence, when trying to prove this implementation to be correct using the EFC permission structure, the first idea that comes to mind is to use only one ghost state in the invariant, quantifying how many permissions, and which amount of permission were given away. This leads to the following invariant:

$$\begin{aligned}\mathcal{Q}^{EFC}(v) := &\text{let } n = \left\lfloor \frac{v}{4} \right\rfloor + lsb(v), w = \text{lsb}(v), \text{ in} \\ &\exists s \in \mathbb{Q} \cap [0,1] \cdot v \geq 0 \wedge (w = 1 \Leftrightarrow s = 1) \wedge \\ &resource(\text{bits})^{1-s} * \boxed{\lambda : (n,s)^-}\end{aligned}$$

while the read and write permission are represented by:

$$R_\lambda^{EFC}(\text{bits}) = \exists q \in (0,1].resource(\texttt{bits})^q * \boxed{\lambda : (1,q)^+}$$

$$W_\lambda^{EFC}(\text{bits}) = resource(\texttt{bits})^1 * \boxed{\lambda : (1,1)^+}$$

Here a read permission is tracked with a token and some non null amount of permission $\boxed{\lambda : (1,q)^+}$, while a write permission corresponds to the full permission 1: $\boxed{\lambda : (1,1)^+}$. The source $\boxed{\lambda : (n,s)^-}$ in the invariant tracks the number n of tokens that were given away, as well as the amount of permission s that was given along. Finally, in the `try_lock_shared` function, a thread may increment the location *bits* without actually gaining any permission, if there is already a writer thread along. In this case, the thread temporarily gets an empty token, with no permission $\boxed{\lambda : (1,0)^+}$. We show in Figure 3.2 the function specifications we would like to prove.

Note that here the tokens represented by the ghost location do not hold any permission by themselves, they are simply used to track the permission that is actually transmitted through $resource(\texttt{bits})^q$.

While this invariant and specifications seem quite straightforward, they are not sufficient to prove correctness of the `unlock_shared` function. Let us go through the proof.

$$\{\mathrm{U}(\text{ bits }, \mathcal{Q})\}\texttt{bool try\_lock\_shared()}\{y.(y?R_\lambda^{EFC}(\text{bits}) : \text{emp})\}$$

$$\left\{ \begin{array}{l} \{\mathrm{U}(\text{ bits }, \mathcal{Q})\}* \\ R_\lambda^{EFC}(\text{bits}) \end{array} \right\}\texttt{void unlock\_shared()}\{\text{emp}\}$$

$$\{\{\mathrm{U}(\text{ bits }, \mathcal{Q})\}\}\texttt{bool try\_lock()}\{y.(y?W_\lambda^{EFC}(\text{bits}) : \{\mathrm{U}(\text{ bits }, \mathcal{Q})\})\}$$

$$\left\{ \begin{array}{l} \{\mathrm{U}(\text{ bits }, \mathcal{Q})\}* \\ W_\lambda^{EFC}(\text{bits})* \\ (\texttt{getRead}? \\ \boxed{\lambda : (1,0)^+} : \\ \text{emp}) \end{array} \right\}\texttt{void unlock(bool getRead)}\left\{ \begin{array}{l} (\texttt{getRead}? \\ R_\lambda^{EFC}(\text{bits}) : \text{emp}) \end{array} \right\}$$

$$\left\{ \begin{array}{l} \{\mathrm{U}(\text{ bits }, \mathcal{Q})\}* \\ W_\lambda^{EFC}(\text{bits})* \end{array} \right\}\texttt{void unlock\_and\_lock()}\{R_\lambda^{EFC}(\text{bits})\}$$

Figure 3.2: Specifications of the lock functions

We use the fetch-and-add rule shown in Figure 7 of [5], using $\mathcal{P}_{\text{send}} = R_\lambda^{EFC}(\text{bits})$ and $\mathcal{P}_{\text{keep}} = \text{emp}$. It is hence sufficient to show that for all values $v \geq 0$, we have that

$$\begin{array}{l} \{\{\mathrm{U}(\text{ bits }, \mathcal{Q})\} * R_\lambda^{EFC}(\text{bits})\} \\ \texttt{CAS}_{\text{rel}}(\texttt{bits}, v, v-4) \\ \{y.(y = v \wedge \text{emp}) \vee (y \neq v \wedge \{\mathrm{U}(\text{ bits }, \mathcal{Q})\} * R_\lambda^{EFC}(\text{bits}))\} \end{array}$$

To do so, we use disjunction. If $v < 4$ and $\texttt{lsb}(v) = 0$, we have that $\mathcal{Q}(v) \Rightarrow \boxed{\lambda : (0,0)^-} * \text{true}$ (we here use that $(0, x)$ is in the domain of the EFC permission structure if and only if $x = 0$). We hence have that $\mathcal{Q}(v) * \boxed{\lambda : (1,0)^+} \Rightarrow \text{false}$, as $\boxed{\lambda : (1,0)^+} \oplus \boxed{\lambda : (0,0)^-}$ is undefined. The CAS-$\bot$ rule from [5] then allows us to conclude this case.

The second case is $v < 4$ and $\texttt{lsb}(v) = 1$. In this case we have that $\mathcal{Q}(v) \Rightarrow \boxed{\lambda : (1,1)^-}$. We hence have again that $\mathcal{Q}(v) * \boxed{\lambda : (1,0)^+} \Rightarrow \text{false}$, as $\boxed{\lambda : (1,0)^+} \oplus \boxed{\lambda : (1,1)^-}$ is undefined. We then use the CAS-$\bot$ rule again.

The proof then fails for the case $v \geq 4$ and $\texttt{lsb}(v) = 1$. In this case, we would like to use again the CAS-$\bot$ rule: in our implementation, this can never happen, as we cannot at the same time have given a write permission (as denoted by $\texttt{lsb}(v) = 1$) and a read permission (currently owned by the

$$\mathcal{Q}^{EFC}(v) := \text{let } n = \left\lfloor \frac{v}{4} \right\rfloor + lsb(v), w = \text{lsb}(v), \text{ in}$$
$$v \geq 0 \wedge (w = 1?\texttt{Src}(res, n, write) : \texttt{Src}(res, n, read))$$

$$R_\lambda^{EFC}(\text{bits}) = \texttt{Tok}(res, 1, read)$$

$$W_\lambda^{EFC}(\text{bits}) = \texttt{Tok}(res, 1, write)$$

Figure 3.3: Simple invariant and Read-Write permissions for the Folly Reader-Writer Spinlock

thread calling `unlock_shared`). However, the invariant fails to capture this idea. The only information we can get from it here is that the amount of permission that was given away in total is 1. For instance an invariant containing the ghost location $\boxed{\lambda : (3, 1)^-}$ could denote either that 3 readers each got permission 1/3 to the resource, or 1 writer got full permission, and two threads are trying to get reader access (and are currently running the `try_lock_shared` function). We could not express using only one ghost location that the first case cannot happen.

While this example shows that the EFC permission structure is a bit less expressive than we can think it to be at first sight, this proof can easily be made to work using one extra ghost location, as is done in [9]. This extra ghost location captures exactly the information we showed was lacking before: is there currently a thread owning write permission?

It is interesting to note that this problem is not encountered in the Rust Atomic Reference Counter proof using an EFC ghost state developed in [9]. This is simply because in this case, we only ever give away read accesses. It can hence be hardcoded in the invariant that the amount of permission given away is always strictly less than 1.

3.3 Proof using token-based reasoning

We saw in the previous section how the EFC permission structure is weaker in practice than what our intuition dictates. While this limitation can easily be overcome, it is at the price of slightly less intuitive proofs. We will see here that the token-based reasoning developed in [11] is itself weaker than the EFC permission structure it was based on.

If we were to straightforwardly translate the proof developed in the previous section using the tokens defined in [11], we would get the invariant and read and write permissions shown in Figure 3.3.

Now, in the `unlock_shared` function, we would like to prove that we cannot read $v < 4$ from the location `bits`. Here we cannot do it, as the

token we own is not incompatible with the location invariant. Reading $v < 4$ and $lsb(v) = 1$ from `bits` yields $\texttt{Src}(res, 1, write)$. Using the ANY-SRC-READ-TOK rule from [11], we could then combine the source with our read token to get $\texttt{Src}(res, 0, read)$, which using the ALL-TOKENS-A (from [11] as well) yields $\texttt{Src}(res, 0, write)$. We could not reach any contradiction here.

This is because the tokens allow for "downgrading" of a token, from write to read permission. Hence it could happen that a thread gets write permission, downgrades it to read permission, then gives back this permission. The source then has to be able to reconstruct from the number of tokens it has whether or not it now owns full permission or not. This could not happen using the EFC ghost state directly as such a downgrading of permission is not possible: the amount of permission q contained in a ghost location $\lambda : (1, q)$ cannot be made lower without further dividing the token.

We here see that the proof fails earlier than when using the EFC permission structure, showing a small difference between the two reasonings. This can again be easily fixed, by making the proof a little less straightforward. We can for instance use the idea developed in [11] for the Rust Atomic Reference Counter proof: defining a different amount of tokens for read an write permissions. This outcome can then be forbidden. By giving 3 tokens for read permission and 2 for write, we would reach a contradiction by trying to combine $\texttt{Src}(res, 2, write)$ and $\texttt{Tok}(res, 3, read)$, using rule TOO-MANY-TOKENS. This trick allows us to get as far in the proof as we did using the EFC ghost location, but we then get stuck in the next step: proving that $v \geq 4$ and $lsb(v) = 1$ is not possible. This is what we would expect: the tokens are not more powerful than the EFC ghost location.

Chapter 4

Limitations of FSL++

In the previous section, we showed some limitations of the EFC permission structure and token-based reasoning. However, those limitations could be overcome, though they made proofs less intuitive and hence harder to write. In this section, we will develop on some limitations of FSL++. Those limitations seem more fundamental, as we could not find ways to overcome them, and at least one would require an extension to the FSL++ logic. We will develop those limitations using two examples, from the glibc and Folly libraries.

4.1 glibc Reader-Writer Lock

We will first focus on the glibc Reader-Writer lock [4]. As we will see in Section 4.1.2, it cannot be proven in FSL++ for a deceptively simple reason. We will still describe it in details, as its synchronization mechanism is quite interesting, and shows how a slightly different implementation can make for different (or impossible in this case) proofs.

4.1.1 Implementation

The implementation of this lock is shown in Appendix A. This implementation was heavily simplified from the original one.

Just as the Folly reader-writer lock, this lock offers four main functions: `readLock` (equivalent of `lock_shared`), `readUnlock` (`unlock_shared`), `writeLock` (`lock`) and `writeUnlock` (`unlock`).

This implementation uses a single atomic location `__readers`, used similarly as the location `bits` in the Folly reader writer lock. This location is used as shown in Figure 4.1. The least significant bit is used to denote if the lock is in read or write phase. As can be expected, if it is in read phase, only reader access can be granted, while if it is in write phase, only writer access can be granted. The second least significant bit tells whether or not there is

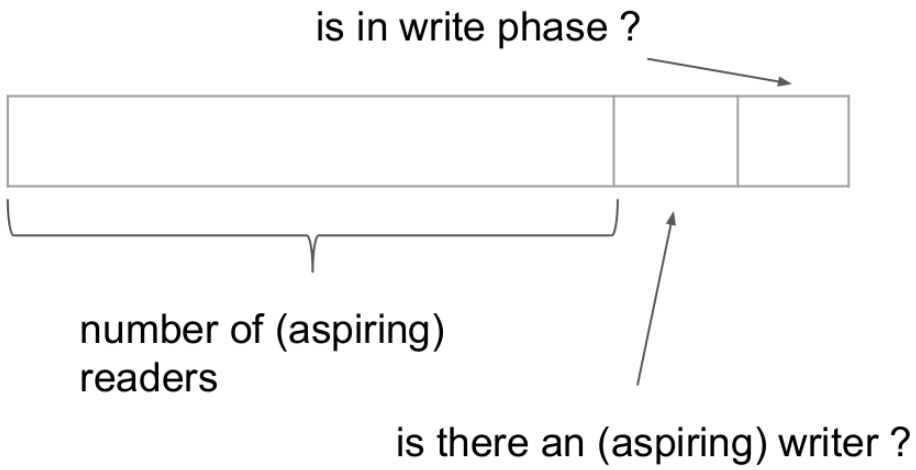

Figure 4.1: glibc reader-writer lock

currently a thread asking for or having write access. Finally, `__readers` /8 gives us the number of threads currently owning reader access or requesting it. For easier reading, we use the following denotation for the value of `readers__`: `readers___`/8|lsb(`readers___`/2)|lsb(`readers___`). For instance, 2|1|1 corresponds to `readers__` = 19, which means the lock is currently in write phase, and that one thread owns a write access, while two threads are requesting a read access. If it were to contain 2|0|1, it means it is still in write mode, but no thread owns write access to it. The implementation makes sure that in this case one of the threads requesting read access will change the least significant bit to 0, allowing for read access.

Each of the functions then uses appropriate synchronization mechanisms to ensure the value of `__readers` accurately reflects the permissions threads hold to the protected location, as well as proper synchronization when taking or releasing the lock. A more detailed explanation of the lock implementation can be found in the Appendix A. The synchronization mechanisms are quite complex, and not necessary to understand the following.

We will here only focus on a further simplification of the `readerLock` function which is sufficient to show the different mechanisms we are interested in. This simple code is shown in Figure 4.2.

In the rest of this section, we will only use two executions scenarios to support our reasoning. They are shown in Figure 4.3 and Figure 4.4. The first one is the most straightforward: there is only one thread trying to get the lock. It calls `readLock()`. The `fetchAndAdd` occurs, increasing the number of potential reader threads by 1. We then check if the least significant bit (corresponding to `WRITEPHASE`) is set to 0. As it is the case, the function returns: Thread 1 now has a read lock. In the second scenario,

```
Constants:
WRLOCKED = 2
WRPHASE = 1

int readLock(){
  int r = fetchAndAdd_Acq(__readers, 8) + 8
  if(r & WRPHASE == 0)
    return 0
  while(r & WRPHASE != 0 && r & WRLOCKED == 0){...}

  while(Load_Acq(__readers) & WRPHASE == 1){;}
}
```

Figure 4.2: `readLock function`

`__readers`	Thread 1
0 | 0 | 0	
	`fetchAndAdd_(__readers)`
1 | 0 | 0	
	`if(r & WRPHASE == 0)`
	`return 0`
	Read lock succeeds!

Figure 4.3: First `readLock()` execution scenario

Thread 1 holds a write lock when Thread 2 calls the `readLock` function. Hence after the `fetchAndAdd`, neither the `if` condition nor the following `while` condition succeed. Thread 1 then releases the lock. The condition of the final `while` is then satisfied. The function returns and Thread 2 now holds a read lock.

We see here that the calling thread can get read access to the resource in two distinct points [1]. This is quite different from the Folly reader-writer spinlock implementation, where the access was granted by the `fetchAndAdd` operation only. If the lock was already taken, the function would cancel this add (by subtracting the same value), and fail. We will see now how this proves to be a problem when formalizing this lock in FSL++.

4.1.2 Read-Modify-Write and Load

As we saw in the two executions scenarios described above, read permission can be given either by a fetch-and-add, or by a subsequent load. However, when looking into the proof rules of FSL++, none of them allow for gaining resources from a single atomic location both using a Load and a fetch-and-

[1] There is actually a third one in the elided code, but it is not relevant to our discussion here

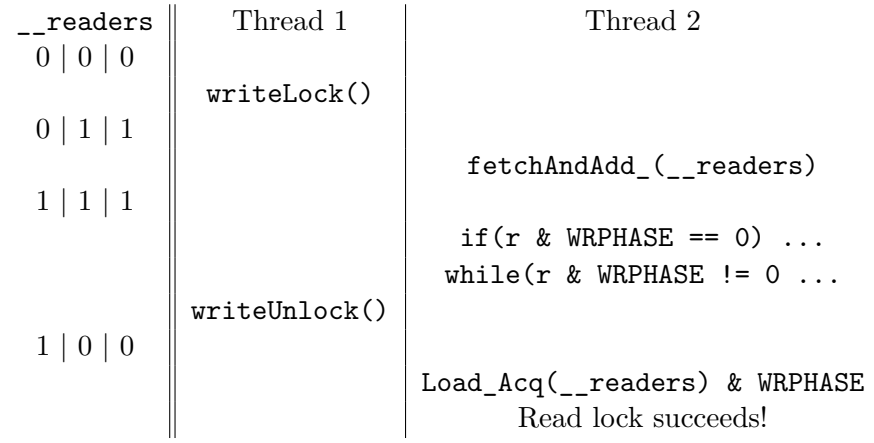

Figure 4.4: Second `readLock()` execution scenario

add. When looking further into it becomes quite clear why such a rule would be quite difficult to implement.

In FSL++, atomic locations are used to transfer ownership (here to the resources protected by the lock) between threads. It does so by using a location invariant $\mathcal{Q}$ which for each value v gives the ownership corresponding to this value. Then when writing value v to the atomic location, a thread has to give up the assertion $\mathcal{Q}(v)$, and when reading value v, a thread gains this assertion. However this simplistic idea is not sufficient: it could happen that Thread 1 writes v, giving up $\mathcal{Q}(v)$, then Thread 2, reads v, gaining $\mathcal{Q}(v)$, then Thread 3 reads as well v, gaining $\mathcal{Q}(v)$. Here, the assertion $\mathcal{Q}(v)$ has been duplicated, which is clearly unsound [2]. To forbid this outcome, FSL++ introduces the assertions $\mathsf{Acq}(\ell, \mathcal{Q})$ and $\mathsf{Rel}(\ell, \mathcal{Q})$. When a new variable is created, the corresponding permissions $\{\mathsf{Acq}(\ell, \mathcal{Q}) * \mathsf{Rel}(\ell, \mathcal{Q})\}$ are created. $\mathsf{Rel}(\ell, \mathcal{Q})$ can then be freely duplicated, whereas the $\mathsf{Acq}(\ell, \mathcal{Q})$ can only be transmitted, making sure only one thread is allowed to get ownership from the location (all threads can read from the location, but they will get no ownership from it).

Now for read-modify-write operations, that is to say operations that read and write atomically to a variable, the situation is different. As the value of the variable is read and changed in one atomic operation, the duplicating problem does not occur anymore. If thread 1 does a fetch-and-add and reads value v, it gets $\mathcal{Q}(v)$, and gives $\mathcal{Q}(v+1)$. Hence if another thread then does another fetch-and-add there is no risk of inadvertently getting twice the same permission. Hence we can be more permissive here, allowing multiple threads to perform read-modify-writes to the same location, and all gaining ownership. To allow for this, when a new variable is created, FSL++ offers a

[2] For instance if $\mathcal{Q}(v) = \mathit{resource}(\texttt{bits})^1$, this would lead to total permission access to a location striclty greater than 1.

choice between $\mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \mathcal{Q})$ or $\mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{RMWAcq}(\ell, \mathcal{Q})$. The latter is then freely duplicable and can be transmitted to all threads requiring it. There is however no rule that would allow to have both $\mathsf{Acq}(\ell, \mathcal{Q}_1)$and $\mathsf{RMWAcq}(\ell, \mathcal{Q}_2)$ for the same location, with $\mathcal{Q}_1$ not empty.

A new logic rule that would allow for this would be difficult to design, as it would have to take into account the duplicability of $\mathsf{RMWAcq}(\ell, \mathcal{Q})$ and non duplicability of $\mathsf{Acq}(\ell, \mathcal{Q})$.

4.1.3 Separation of tokens and permission

We saw in the previous section how FSL++ rules do not allow for a proof of the glibc reader-writer lock. Another interesting point of this lock when thinking in terms of the EFC permission structure, is that it separates tokens from permission. In the second scenario Figure 4.4, thread 2 increments the number of readers with the fetch-and-add, but only gets permission with the load in the second while. This would hence make counting the number of permissions given away (as recorded by `__readers / 8`) difficult to link with the amount of permission given (done in the load here).

4.2 Folly One Producer One Consumer Queue

We saw in the previous section that we could not use FSL++ to prove strong results about the glibc reader-writer lock, because it lacked support for both RMW and Loads to the same variable. We will now focus on a different example, using only Store/Load, to avoid encountering this shortcoming again.

This new example is the Folly ProducerConsumerQueue [1], a concurrent fixed size queue. It allows for exactly one producer thread, and one consumer thread. The queue is implemented using a circular buffer, as well as release stores and acquire loads for synchronization. The full code, including some extra functions that we will not discuss can be found in Appendix B. We will here use the simplification of the code shown in Figure 4.5.

An object `Queue` contains four fields: its size, an array `records` of size `size_`, and two atomic indices `readIndex, writeIndex`. When creating the object, both indices are set to 0. The queue offers two main functions: `write` which takes an argument and adds it to the queue if it is not full, and `read` which returns the last element if the queue is not empty.

Note here that in spite of its name, the read function requires full ownership to the array slot it is reading, as it deletes it when returning it. This means that the producer and consumer threads need to transfer full ownership between one another during those function calls.

This code is quite simple. Let us focus on the `read` function in more detail. This function first reads the current read index. Note that this

```
bool write(toWrite){
  int curWrite = Load_Rlx(writeIndex)
  int nextWrite = (curWrite + 1) % size_
  int curRead = Load_Acq(readIndex)
  if(nextWrite != curRead){
    records[curWrite] = toWrite
    Store_Rel(writeIndex, nextWrite)
    return true
  }
  return false
}
bool read(&valRead){
  int curRead = Load_Rlx(readIndex)
  int nextRead = (curRead + 1) % size_
  int curWrite = Load_Acq(writeIndex)
  if(curRead == curWrite)
    return false
  valRead = records[curRead]
  delete records[curRead]
  Store_Rel(readIndex, nextRead)
  return true
}
```

Figure 4.5: Simplified code for the Folly one producer one consumer queue

read is relaxed. It is used here only to have a nicer interface for the function: as there is only one consumer thread, this thread could remember the `readIndex` value from one call to another, and pass it as an argument to the function, without fundamentally changing the function. We then compute the position of the next slot in the array (which we use as a circular buffer hence the need for %). The first synchronization then comes in: the value of `writeIndex` is read with an `acquire` synchronization. This synchronization ensures that all reads or writes that come after will not be re-ordered before. We then check whether the queue is empty. If not, we store the current record in the reference that was passed as argument to the function, then delete the record. Finally, we use a release store to update the value of `readIndex`.

To better explain the synchronization ensuring that there is no data race on `records`, we focus on a simple scenario: we start from an empty queue, the producer thread then stores a value in `records`, and the consumer thread then reads it. Part of this scenario is shown in Figure 4.6. The release store to `writeIndex` ensures that any memory operation happening before it will be visible to other threads when they see the value released. In particular, any thread reading the new value of `writeIndex` will have the proper value stored in `records[0]`. This is materialized by the red arrow. Then the acquire load ensures that any memory operation after it will not be re-ordered. In particular, reading in `records[0]` will happen after this value

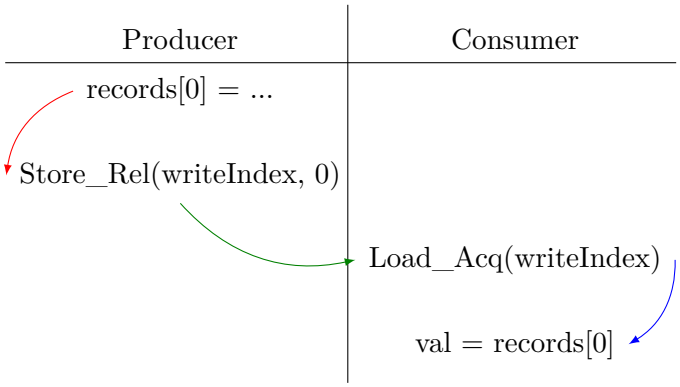

Figure 4.6: Synchronization in the Folly one producer one consumer queue

was read, as shown by the blue arrow. Finally, as in this scenario the load acquire read the value from the previous release store, we have an ordering between the two shown by the green arrow. There is hence a proper ordering between the write access to `records[0]` by the producer thread and the read access to it by the reader thread: data races are avoided.

4.2.1 Re-using values for location invariants

We saw in the previous section about the glibc reader-writer lock the need for $\mathsf{Acq}(\ell, \mathcal{Q})$ predicates, to ensure ownership could not be duplicated, for instance by two threads reading to the same value. The restrictions on this predicate are actually even stronger than what we developed there. The rule for a read[3] in FSL++ is

$$\{\mathsf{Acq}(\ell, \mathcal{Q}) * \mathsf{Init}(\ell)\}\texttt{Load\_Acq}(\ell)\{v.\mathsf{Acq}(\ell, \mathcal{Q}[v := \mathsf{emp}]) * \mathcal{Q}(v)\}$$

Hence reading value v from a location ℓ uses our permission to do so. When the thread next reads the same value from this variable, it will not be able to gain any ownership from it. The only way we could gain ownership twice or more from reading the same value would be by splitting the $\mathsf{Acq}(\ell, \mathcal{Q})$ beforehand, using the following rule

$$\mathsf{Acq}(\ell, \mathcal{Q}_1) * \mathsf{Acq}(\ell, \mathcal{Q}_2) \equiv \mathsf{Acq}(\ell, \lambda v. \mathcal{Q}_1(v) * \mathcal{Q}_2(v))$$

However, as noted when presenting the implementation of this queue, both the `read` and `write` function require full ownership. As full ownership cannot be split into two (or more) full ownership, we cannot apply this rule here. Beside, FSL++ does not contain any rule that would allow for making stronger an $\mathsf{Acq}(\ell, \mathcal{Q})$ permission. This means that when the location

[3]This rule is slightly simplified, some conditions are required on $\mathcal{Q}$.

```
bool write(toWrite){
  int curWrite = Load_Rlx(writeIndex)
  int nextWrite = (curWrite + 1)
  records[curWrite] = toWrite
  Store_Rel(writeIndex, nextWrite)
  return true
}
bool read(&valRead){
  int curRead = Load_Rlx(readIndex)
  int nextRead = (curRead + 1)
  int curWrite = Load_Acq(writeIndex)
  if(curRead == curWrite)
    return false
  valRead = records[curRead]
  delete records[curRead]
  Store_Rel(readIndex, nextRead)
  return true
}
```

Figure 4.7: Simplified code for the Folly queue with infinite buffer

invariant in an acquire permission is set to `emp` for some value v, it can never be made non empty again.

So, if we were to prove the `read` code correct, we would need some invariant $\mathcal{Q}$, for the location `writeIndex` (as the synchronization uses this location). This invariant would need to contain full permission for the slot in the `records` array that the consumer thread can read. Now after some calls to the `read` and `write` functions, we may read the same value again from `writeIndex`, as the slots in the circular buffer are used again and again. At this point, since $\mathcal{Q}$ cannot be split while keeping full permission, it has been kept full, and $\mathsf{Acq}(\ell, \mathcal{Q})$ now has $\mathcal{Q}(v) = \mathsf{emp}$. We cannot gain any new ownership. This is the case no matter which location invariant $\mathcal{Q}$ we chose, as long as it contains the full ownership required by the `read` function. Hence we cannot prove this code to be correct using FSL++.

4.2.2 Infinite queue

As we have seen above, as the queue implementation uses many times the same slots in the `records` array, used as a circular buffer, we cannot prove it using FSL++. A simplification that immediately comes to mind would be to use an infinite array for `records`. The queue would not be limited anymore, and most importantly for us, slots in `records` are not re-used anymore.

Sadly, here again we were not able to use FSL++ to prove this simplified code correct. The code of the simplification is shown in Figure 4.7.

Again, the problem lies in how we store ownership in location invariants. We denote $\mathcal{Q}$ the location invariant of `writeIndex`. When the producer

thread adds a new value to the queue, it updates the value of `writeIndex`, and hence needs to give up the corresponding assertion $\mathcal{Q}(v)$. Now if the `read` function is then called, it reads `writeIndex`, and gets the corresponding $\mathcal{Q}(v)$. However, if this is not the case and `write` is called again, and gives up $\mathcal{Q}(v+1)$. The consumer thread calling `read` will hence have access to $\mathcal{Q}(v+1)$, but has no way to access $\mathcal{Q}(v)$ again. All the ownership contained in it is lost: neither the producer nor the consumer can gain access to it again.

Here the correctness of this implementation relies on the fact that only the consumer thread can gain ownership through `writeIndex`, and this variable is always increased by 1. Hence if the consumer thread reads 1 then 4 from it, it is as if it had read 2 and 3 as well. We were not able to model this idea using FSL++. This is not a problem when using read-modify-write, as in this case all intermediate states of the variable are observed by one of the threads: each write to this variable has to be preceded by a read to it.

Note that it would be tempting here to use the fact that the producer thread begins the `write` function by reading the value `writeIndex`. One could imagine that through this read the producer thread can get back the location invariant it previously gave up. However, remember that $\mathsf{Acq}(\ell, \mathcal{Q})$ containing full ownership cannot be split between two threads, as explained for the circular queue. This idea hence cannot be used here.

We are hence unable to prove this simplification of the queue to be correct.

Chapter 5

Proving intertwined synchronization mechanisms

In this chapter we will focus on a new example from the Folly library: the Serial Executor [3]. This example makes use of two independent synchronization mechanisms, using two atomic locations. We will show how FSL++ allows us to prove those two mechanisms almost independently.We have not proven it in the proof framework developed in [11], but we believe that this would be possible.

To the best of our knowledge, this is the first proof of this executor in FSL++, as well as the only example making use of more than one synchronization mechanism.

We now describe in more details this new example. An *executor* is a tool that allows a thread to delegate the execution of a task. The thread simply adds this task to the executor, and the executor guarantees that the task will be executed at some point. Depending on the type of executor, the executor may provide some extra guarantees on the execution order of the task submitted to it. The executor we all know of is the CPU scheduler: it has many tasks to execute (everything currently running on the computer mostly) and schedules them as it sees fit.

The Folly Serial Executor is an executor that guarantes that all the tasks submitted to it will be executed serially: no two of them will be executed concurrently. An example use case of such an executor would be in a server for a port. We want to make sure that no two threads try to access the same port at the same time. We can hence create a serial executor `s` dedicated to that port. Now, any thread requiring action on this port will add the task to this executor. The executor will then execute those tasks one after the other, making sure no two tasks try to write to the port at the same time.

Note that it is possible that the executor uses multiple threads to execute the tasks assigned to the serial executor. The serial executor simply guarantees that this will never be done concurrently: we will not have two

threads executing some tasks from the serial executor at the same time.

Besides, a serial executor guarantees that even if it is deleted, the tasks that were added to it will be executed sequentially, as if it had not been deleted.[1]

5.1 Implementation

A simplified implementation of the executor is shown in Figure 5.1. This code is quite different from the original, as it was originally divided among two classes, and used pointers, as well as customs destructors. A code more faithful to the original can be found in the Appendix C.

An executor is made of four fields. The field `queue_` is used to record the tasks that are submitted to the executor. The field `parent_` refers to another executor: the one that is the serial executor's parent. Indeed, the serial executor is merely a middleman here. It records the tasks that are submitted to it (which we will call OT) in the queue of the `queue_` field, and using this queue creates new tasks (NT) that it then add to its parent, another executor, not necessarily serial. Those NT tasks use the `scheduled_` field to synchronize between one another, making sure that only one of them executes a OT tasks at any point in time. The NT tasks may execute concurrently, but they make sure not to concurrently run the OT tasks. Finally the field `keepAliveCounter_` is used to track the number of references to this executor, and deleting it when there are no references to it anymore. The `scheduled_` and `keepAliveCounter_` fields are atomic locations.

5.1.1 Ensuring sequential execution

Let us now describe the functions of the serial executor interface. The `add` one is quite straightforward. It adds a new task, here denoted by `func`, to the executor `s`. This is done by simply adding the task to the queue, and sending a new task to the parent executor: running the serial executor `run` function. Because the parent is an executor, it guarantees that this `run_` function will be executed at some later point in time. Besides, this `run_` function is private and only called by the `add_` function in the serial executor class. We hence have that there will be exactly one execution of `run_` for each call to `add_`.

The `run` function is the one ensuring serial execution of the tasks, using the `scheduled_` location. To explain how it works, it helps to think about what could have been an alternative implementation of this function. We could have simply used `scheduled_` as a flag recording whether or not there

[1] Unless the serial executor parent, that is the executor it delegates the tasks to is deleted.

```
Executor newSerialExecutor(Executor parent){
  s = alloc();
  s.parent_ = parent;
  s.queue_ = new Queue();
  s.scheduled_ = 0;
  s.keepAliveCounter_ = 1;
  return s;
}

void add(Func func, Executor s){
  s.queue_.enqueue(func);
  s.parent_.add({this->run()});
}

void run(Executor s){ //private function
  if(s.scheduled_.fetch_add(1, acquire) > 0)
    return;
  do {
    Func func = s.queue_.dequeue();
    func();
  } while(s.scheduled_.fetch_sub(1, release) > 1);
}

void drop(Executor s){
  int c = s.keepAliveCounter_.fetch_sub(1, release_acquire);
  if(c == 1)
    free(s);
}

Executor copy(Executor s){
  s.keepAliveCounter_.fetch_add(1, relaxed);
  return s;
}
```

Figure 5.1: Simplified code for the Folly SerialExecutor

is currently a task executing. The simplified code would have been the following

```
void run(Executor s){
  if(CAS_Acq(s.scheduled_, 0, 1)){
    while(!s.queue_.empty()){
      func = s.queue.pop();
      func()
    }
    CAS_Rel(s.scheduled_, 1, 0);
  }
}
```

We first try to set atomically `scheduled_` to 1. If this succeeds, this is now the only `run()` task from the serial executor `s` that can be executed. Then as long as the queue is not empty, we keep executing one after the other all the tasks that were added to it. When none are left, the `run()` task releases the `scheduled_` flag by setting it to 0. However, this function requires two calls to `queue_` per loop execution. One checking if it is empty or not and the second popping an element[2]. The key remark here is that we can use scheduled to count the number of elements in the queue reliably. This is what is done in the `run` function shown in Figure 5.1.

Every time this function is called, it first increments `scheduled_` by one. As `run` is called every time an element is added to the `queue_` by the `add` function, `scheduled_` records the number of elements in the queue. Now if `scheduled_` was zero before the task incremented it, the task starts executing the tasks contained in `queue_`. This is similar to the case where the `CAS_Acq` on `scheduled_` succeeded in the alternative implementation discussed above. Every time a task from `queue_` is executed, `scheduled_` is decreased by one. As `scheduled_` tracks the number of elements in the queue, this ensures that we never try to pop from an empty queue[3]. When `scheduled_` reaches 0, that is to say when the value returned by `fetch_sub` is 1, the task terminates. As in the alternative implementation shown above, `scheduled_` has been set back to 0, allowing another `run()` task to start executing later.

We explained above the idea of the `run` function. Let us now develop on the precise synchronization mechanisms used. As we have seen, it is `scheduled_` that allows execution to "pass" from one thread to another. For instance, imagine Thread 1 calls `run`. It does the first fetch and add, and sees that the former value was 0. It can hence start executing tasks. At some point, the fetch and sub brings `scheduled_` back to 0. Thread 1

[2]Note that here, there is no risk of having a non empty queue when the condition check is done, then its elements being removed by another thread before the `pop` is done. Indeed, as `scheduled_` is set to one, only the current task can remove elements from the queue.

[3]This is not trivial actually, it relies on the fact that no other thread can pop from the queue, and that due to the careful of ordering of actions on `scheduled_` and `queue_`, after the `fetch_sub`, `scheduled_` is smaller or equal to the number of elements in the queue.

stops executing tasks. Then at some later point, Thread 2 calls `run`, and starts executing tasks. Here when Thread 2 takes over execution, we have to make sure that for any other observer thread, all the memory events done by Thread 1 as part of executing the tasks stay before all the memory actions Thread 2 will do. As Thread 1 used a fetch and sub release, we know that all events executed in Thread 1 before this fetch and sub will stay before the action that set `scheduled_` to 0. Besides, as Thread 2 uses a fetch and add acquire, all events that are executed after this fetch and add cannot be re-ordered with it. We hence have the required synchronization:

$$\{\text{Tasks 1}\} \xrightarrow{\text{release}} \{\texttt{scheduled\_} = 0\} \xrightarrow{\text{acquire}} \{\text{Tasks 2}\}$$

where we use Tasks 1 (resp. 2) to refer to the tasks executed by Thread 1 (resp. 2).

5.1.2 Ensuring proper deletion when deleting references

The functions `drop` and `copy` ensure that when all references to a serial executor `s` are gone, it is deleted, but not before that. They do so in a fashion extremely similar to that of the Rust Atomic Reference Counter, studied in [5]. `keepAliveCounter` tracks the number of references to the serial executor. Those references can be obtained through the `copy` function, and deleted using the `drop` function. The latter then deletes the serial executor if the reference dropped is the last one.

The only difference between this code and the one of the Rust ARC is in the `drop` function. In the Rust ARC, the code is the following (taken from [5])

```
int c = s.keepAliveCounter_.fetch_sub(1. release);
if(c == 1){
  fence_acq;
  free(s);
}
```

For the serial executor, instead of using a fence acquire only for the last decrement, every fetch and sub contains an acquire synchronization. For all the decrements that do not yield 0, this extra synchronization does not affect the behavior we want to achieve. Finally, for the last decrement, having acquire included in the fetch and sub instead of a standalone fence does not change the synchronization provided by the function[4].

For a more in depth explanation of the synchronization mechanisms at play here, refer to [5].

[4]In practice, there is a slight difference: a fence acquire prevents re-ordering of any read before it with any read or write after it, whereas a load acquire only forbids re-ordering between the load itself and any subsequent read or write.

5.2 Properties of the `SerialExecutor`

We explained above the implementation of the serial executor. We will here detail the few properties we set out to prove about this code

The first property we want to show about this code, is that the `SerialExecutor` is indeed serial, that is to say all functions passed to it through `add` are executed sequentially, and never concurrently. As we are only interested in the synchronization mechanism here, we model this using the following simplification. We consider one non-atomic location `protected`, and consider that all functions passed to `add` need full ownership of this location for their execution, and nothing else. We hence have to show that when they are run, they are in a thread that does own this permission. The code with this simplification is shown in Figure 5.2. Let us now explain why proving this simplification correct ensures non concurrent execution of the tasks.

If we manage to prove this simplification correct, we will have proven that there is no data race on the non atomic location `protected_`. This means, by definition of a data race as given in Section 2, that all memory operations on `protected_` are ordered by the happen-before relation, that is to say they are all executed serially, and not concurrently. This would then allow us to conclude that the synchronization in the `run` function is enough to ensure ordering of the all operations executed within the repeat loop: setting `protected_` to some value `a`. If we were to replace this operation by any function execution, as is the case in the original Serial Executor code, we keep this property on ordering: the functions submitted to the Serial Executor are all executed serially.

Note: One could think that proving the simplified code correct only shows that the Serial Executor provides serial execution of the write memory operations submitted to it, and not necessarily read operations. Indeed, we mentioned in Section 2 that the release and acquire fences provide different ordering constraints to read and write operations. However, in the `run` function, the only synchronization operations are a release fetch and sub and an acquire fetch and add. Both of those operations do not discriminate between read and writes in the synchronization they provide, as shown in Figure 2.2. Proving ordering of the write operations is hence enough to conclude on ordering of all memory operations, read and write alike.

The second property we are interested in is a proper use of the queue: we need to make sure that at any point there is at most one consumer, while there can be many producers. To model this, we introduce two abstract predicates $\mathsf{Consumer}(\texttt{s.queue})$ and $\mathsf{Producer}(\texttt{s.queue})$. They are governed by the following properties:

$$\frac{}{\{\mathsf{emp}\}\texttt{q = new Queue()}\{\mathsf{Consumer}(\texttt{q}) * \mathsf{Producer}(\texttt{q})\}} \quad (\textsc{new queue})$$

$$\mathsf{Producer}(\mathtt{q}) \iff \mathsf{Producer}(\mathtt{q}) * \mathsf{Producer}(\mathtt{q}) \quad \text{(PRODUCER DUPLICATION)}$$

$$\frac{}{\{\mathsf{Consumer}(\mathtt{s.queue})\}\mathtt{q.dequeue()}\{y.\mathsf{Consumer}(\mathtt{s.queue})\}} \quad \text{(DEQUEUE)}$$

$$\frac{}{\{\mathsf{Producer}(\mathtt{s.queue})\}\mathtt{q.enqueue(a)}\{\mathsf{Producer}(\mathtt{s.queue})\}} \quad \text{(ENQUEUE)}$$

The (NEW QUEUE) allows us to create a new queue, and creates permission for a consumer and a producer. The (PRODUCER DUPLICATION) equivalence allows us to duplicate the producer permission, allowing mutliple threads to add to the queue. (DEQUEUE) and (ENQUEUE) simply encode that a thread con only enqueue (resp. dequeue) an element from the queue if it owns the $\mathsf{Producer}(\mathtt{s.queue})$ (resp. $\mathsf{Consumer}(\mathtt{s.queue})$) predicate. Those two predicates should be thought of as some placeholders, that could be replaced with more precise predicates defining the behavior of a single consumer multiple producers queue.

Finally we want to prove that the `keepAliveCounter` works as expected, that is to say like the Rust Atomic Reference Counter. The specification is quite simple: the deallocation of the executor `s` is not done until all threads are done with using the fields of `s` that are deallocated. However, note that the specification of the `SerialExecutor` requires that even if it is deleted, all the tasks submitted to it will still be executed with the same guarantees. We can rephrase this property about the `keepAliveCounter` more precisely as: no read to the `parent_` field of the executor should race with the deallocation of this executor.

Some other properties could be proven on this code, for instance the fact that we never try to get an element from the empty queue. However, this would require first defining and proving the properties of a single consumer, multiple producer queue.

5.3 Formalizing the specifications

To produce a formal proof of the properties described above, we need to rewrite again the code of the serial executor, using only language constructs allowed in FSL++. The main change is the use of a `protected` non atomic variable to model the resources the serial executor protects. Instead of pushing tasks `func` to the queue, we now push some integer value, and executing a task is simply assigning this value to the `protected` field. We also replace the `do ... while` loop with a `repeat`, that repeats its body until it returns a non-zero value. Finally, the `drop` function now returns

the value it read from `keepAliveCounter_`, so that we can have a more precise specification of this function. The resulting simplified code is shown in Figure 5.2.

Verifying our first property, that is to say that the executor guarantees serial execution now amounts to proving that when the write to `s.protected_` happens in the `add` function, we currently have full ownership of the `s.protected_` location.

Verifying the second property simply requires showing that when calling `s.queue_.dequeue()`, a thread holds the assertion $\mathsf{Consumer}(\texttt{s.queue})$, and when calling `s.queue_.enqueue()` it holds $\mathsf{Producer}(\texttt{s.queue})$.

Finally, to show that `keepAliveCounter_` works as expected, we need to show that there exists a predicate $\mathsf{SE}_\lambda(\mathsf{s})$ such that the function specifications shown in Figure 5.3 hold. This predicate is parameterized by a ghost location we will define in further detail later. For the sake of simplicity, we remove the dependency on the `protected` location from this predicate.

Besides, this predicate should allow us to satisfy the two first properties.

5.4 Proof of the specifications

We will now prove each of the formalized specifications defined in the previous section. To do so, we define a location invariant for `s.scheduled_` and `s.keepAliveCounter_`, and define the $\mathsf{SE}_\gamma(s)$ predicate.

Definition 1 (`scheduled_` invariant)**.** *For a location x, we define the following map, from values to assertions*

$$\mathcal{Q}_{x,s}(c) = c \geq 0 \wedge \textit{ if } c = 0 \textit{ then } \texttt{s.protected\_}^1 * \mathsf{Consumer}(\texttt{s.queue}) \textit{ else } \mathsf{emp}$$

This invariant simply contains ownership of the `protected_` resource, as well as predicate encoding consuming elements from the queue.

Definition 2 (`keepAliveCounter_` invariant)**.** *For a location x and a ghost location γ, we define the following map, from values to assertions*

$$\mathcal{R}_{\gamma,x}(c) = c \geq 0 \wedge \textit{ if } c = 0 \textit{ then } \boxed{\gamma : (0,0)^-} * \texttt{s.parent\_}^1$$
$$\textit{else } \exists f \in \mathbb{Q} \cap [0,1].\texttt{s.parent\_}^f * \boxed{\gamma : (c, 1-f)^-}$$

This invariant uses the EFC permission structure for the ghost state γ. It is almost identical to the invariant defined in [9] for the proof of the Rust ARC.

Finally, we define the predicate $\mathsf{SE}_\gamma(s)$.

Definition 3 (Serial executor resource)**.**

$$\mathsf{SE}_\gamma(s) = \mathsf{U}(\texttt{s.scheduled\_}, \mathcal{Q}) * \mathsf{U}(\texttt{s.keepAliveCounter\_}, \mathcal{P})$$
$$* \mathsf{Producer}(\texttt{s.queue}) * \exists q \in \mathbb{Q} \cap (0,1].\texttt{s.parent\_}^q$$
$$* \boxed{\gamma : (1,q)^+}$$

```
Executor newSerialExecutor(Executor parent, Ress protected){
  s = alloc();
  s.protected_ = protected;
  s.parent_ = parent;
  s.queue_ = new Queue();
  Store_Rel(s.scheduled_, 0);
  Store_Rlx(s.keepAliveCounter_, 1);
  return s;
}

void add(int f, Executor s){
  s.queue_.enqueue(f);
  s.parent_.add({this->run()});
}

void run(Executor s){
  c = s.scheduled_.fetch_add(1, acquire);
  if(c > 0) {}
  else{
    repeat (
      a = s.queue_.dequeue();
      s.protected_.set(a);
      c = s.scheduled_.fetch_sub(1, release) > 1;
      (c = 1)//return value of the expression within repeat
    )
  }
}

int drop(Executor s){
  int c = s.keepAliveCounter_.fetch_sub(1, release_acquire);
  if(c == 1)
    free(s);
  return c;
}

Executor copy(Executor s){
  s.keepAliveCounter_.fetch_add(1, relaxed);
  return s;
}
```

Figure 5.2: FSL++ compatible Serial Executor code

$$\{\texttt{protected}^1\}\texttt{newSerialExecutor}(\texttt{parent}, \texttt{protected})\{s.\exists\gamma.\mathsf{RE}_\gamma(s)\}$$

$$\{\mathsf{SE}_\gamma(s)\}\texttt{add}(\texttt{f}, \texttt{s})\{\mathsf{SE}_\gamma(s)\}$$

$$\{\mathsf{U}(\texttt{s.scheduled\_}, \mathcal{Q})\}\texttt{run}(\texttt{s})\{\mathsf{U}(\texttt{s.scheduled\_}, \mathcal{Q})\}$$

$$\{\mathsf{SE}_\gamma(s)\}\texttt{copy}(\texttt{s})\{\mathsf{SE}_\gamma(s) * \mathsf{SE}_\gamma(s)\}$$

$$\{\mathsf{SE}_\gamma(s)\}\texttt{drop}(\texttt{s})\{y.(y > 1 \wedge \mathsf{emp}) \vee (y = 1 \wedge \texttt{s.parent\_}^1)\}$$

Figure 5.3: Serial executor fonctions specifications

In this definition, we use $\mathsf{U}(\ell, \mathcal{R})$ as a shortcut for

$$\mathsf{Rel}(\ell, \mathcal{R}) * \mathsf{RMWAcq}(\ell, \mathcal{R}) * \mathsf{Init}(\ell)$$

as is done in [5].

We can now set out to prove that all of the formal properties outlined in the previous sections hold.

5.4.1 Function `newSerialExecutor`

The proof outline of this function is given in Figure 5.4. We use here that a ghost state can be introduced anytime, to introduce $\boxed{\gamma : (0.0)^-}$.

5.4.2 Function `add`

Verifying this function is trivial, as the Serial Executor predicate $\mathsf{SE}_\gamma(s)$ contains the predicate $\mathsf{Producer}(\texttt{s.queue})$ allowing to push elements to the queue.

5.4.3 Function `run`

The proof outline for this function is shown in Figure 5.5. Remember that this function is not run by the thread using the executor, but by the executor parent. As there is no way to model this delegation of execution in FSL++, we here simply assume the weakest precondition needed to prove this function correct: $\mathsf{U}(\texttt{s.scheduled\_}, \mathcal{Q})$. This precondition is freely duplicable, and available in any thread calling the `add` function. Hence, the only thing we do not formally model in FSL++ is how this predicate is passed to the parent executor in the line `s.parent_.add(..)`.

$\texttt{newSerialExecutor}(\texttt{parent}, \texttt{protected})$
$\{\texttt{protected}^1\}$
$\texttt{s} = \texttt{alloc}()$
$$\left\{ \begin{array}{l} \mathsf{RMWAcq}(\texttt{s.scheduled\_}, \mathcal{Q}) * \mathsf{Rel}(\texttt{s.scheduled\_}, \mathcal{Q}) * \\ \mathsf{RMWAcq}(\texttt{s.keepAliveCounter\_}, \mathcal{R}) * \mathsf{Rel}(\texttt{s.keepAliveACounter\_}, \mathcal{R}) * \\ \texttt{s.protected\_}^1 * \texttt{s.parent\_}^1 * \texttt{protected} \end{array} \right\}$$
$\texttt{s.protected\_} = \texttt{protected}$
$\texttt{s.parent\_} = \texttt{parent}$
$$\left\{ \begin{array}{l} \mathsf{RMWAcq}(\texttt{s.scheduled\_}, \mathcal{Q}) * \mathsf{Rel}(\texttt{s.scheduled\_}, \mathcal{Q}) * \\ \mathsf{RMWAcq}(\texttt{s.keepAliveCounter\_}, \mathcal{R}) * \mathsf{Rel}(\texttt{s.keepAliveACounter\_}, \mathcal{R}) * \\ \texttt{s.parent\_}^1 * \texttt{s.protected}^1 \end{array} \right\}$$
$\texttt{s.queue\_} = \texttt{newQueue}()$
$$\left\{ \begin{array}{l} \mathsf{RMWAcq}(\texttt{s.scheduled\_}, \mathcal{Q}) * \mathsf{Rel}(\texttt{s.scheduled\_}, \mathcal{Q}) * \\ \mathsf{RMWAcq}(\texttt{s.keepAliveCounter\_}, \mathcal{R}) * \mathsf{Rel}(\texttt{s.keepAliveACounter\_}, \mathcal{R}) * \\ \texttt{s.parent\_}^1 * \texttt{s.protected}^1 * \mathsf{Consumer}(\texttt{s.queue}) * \mathsf{Producer}(\texttt{s.queue}) \end{array} \right\}$$
$\texttt{Store\_Rel}(\texttt{s.scheduled\_}, 0)$
$$\left\{ \begin{array}{l} \mathsf{U}(\texttt{s.scheduled\_}, \mathcal{Q}) * \mathsf{RMWAcq}(\texttt{s.keepAliveCounter\_}, \mathcal{R}) * \\ \mathsf{Rel}(\texttt{s.keepAliveACounter\_}, \mathcal{R}) * \boxed{\gamma : (0,0)^-} \\ \texttt{s.parent\_}^1 * \mathsf{Producer}(\texttt{s.queue}) \end{array} \right\}$$
$\texttt{Store\_Rlx}(\texttt{s.keepAliveCounter\_}, 1)$
$$\left\{ \begin{array}{l} \mathsf{U}(\texttt{s.scheduled\_}, \mathcal{Q}) * \mathsf{U}(\texttt{s.keepAliveCounter\_}, \mathcal{R}) * \\ \texttt{s.parent\_}^1 * \mathsf{Producer}(\texttt{s.queue}) * \boxed{\gamma : (1,1)^+} \end{array} \right\}$$

Figure 5.4: Proof outline for the `newSerialExecutor` function

$\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}), \mathcal{Q})\}$
$\mathtt{run}(\mathtt{s})$
$\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q})\}$
$\mathtt{c} = \mathtt{s.scheduled\_.fetch\_add}(1, \mathtt{acquire})$
$\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * (c = 0?\mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue}) : \mathsf{emp})\}$
$\mathtt{if}(\mathtt{c} > 0)\{\}$
$\mathtt{else}$
$\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue})\}$
$\mathtt{y} = \mathtt{repeat}($
$\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue})\}$
$\mathtt{a} = \mathtt{s.queue\_.dequeue}()$
$\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue})\}$
$\mathtt{s.protected\_.set}(\mathtt{a})$
$\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue})\}$
$\mathtt{c} = \mathtt{s.scheduled\_.fetch\_sub}(1, \mathtt{release})$
$\{c \leq 1 \wedge \mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * (c = 1?\mathsf{emp} : \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue})\}$
$(\mathtt{c} = 1)$
$)\mathtt{end}$
$\{y.\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) \wedge (y = 0?\mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue}) : \mathsf{emp})\}$

Figure 5.5: Proof outline for the `run` function

We first need to prove the fetch and add, that is to say show that:

$\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q})\}$
$\mathtt{c} = \mathtt{s.scheduled\_.fetch\_add}(1, \mathtt{acquire})$
$\left\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * (c = 1 ? \mathsf{emp} : \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue}))\right\}$

For this we use the fetch and add rule outlined in [5]. We first need the fact that $\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q})$ is duplicable. We now choose $\mathcal{P}_{send} = \mathsf{emp}$ and $\mathcal{P}_{keep} = \mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q})$. We then need to show that for any value $t >= 0$, we have

$\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathsf{emp}\}$
$\mathsf{CAS}_{acquire}(\mathtt{s.scheduled\_}, t, t+1)$
$\{\ y.(y = t \wedge \mathcal{Q}(t)) \vee (y \neq t \wedge \mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathsf{emp})\ \}$

This compare and swap is now trivial to prove, as the only two cases are $t = 0$ and $t \neq 0$.

We now move on to the `repeat` instruction. To prove it correct, we use the following rule:

$$\frac{\{P\}\, E\, \{y.Q\} \qquad Q\,[0/y] \Rightarrow P}{\{P\}\, \mathtt{repeat} E \mathtt{end}\, \{y.Q \wedge y \neq 0\}}$$

where we chose

$$P = \mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue})$$

$Q = \mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * y = 0 ? \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue}) : \mathsf{emp}$

We hence need to prove that

$\left\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue})\right\}$
$\mathtt{a} = \mathtt{s.queue\_.dequeue}(); \mathtt{s.protected\_.set}(\mathtt{a});$
$\left\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue})\right\}$
$\mathtt{c} = \mathtt{s.scheduled\_.fetch\_sub}(1, \mathtt{release})$
$\left\{c \leq 1 \wedge \mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * (c = 1 ? \mathsf{emp} : \mathtt{s.protected\_}^1 * \mathsf{Consumer}(\mathtt{s.queue})\right\}$

We use again the fetch and add rule outlined in [5], and the fact that $\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q})$ is duplicable. We chose for all $t \neq 1$ $\mathcal{P}_{send}(t) = \mathsf{emp}$ and $\mathcal{P}_{keep}(t) = \mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathtt{s.protected}^1 * \mathsf{Consumer}(\mathtt{s.queue})$, and $\mathcal{P}_{send}(1) = \mathtt{s.protected}^1 * \mathsf{Consumer}(\mathtt{s.queue})$ and $\mathcal{P}_{keep}(1) = \mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q})$. We then need to show that for all $t \leq 0$, we have that

$\left\{\mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathtt{s.protected}^1 * \mathsf{Consumer}(\mathtt{s.queue})\right\}$
$\mathsf{CAS}_{release}(\mathtt{s.scheduled\_}, t, t-1)$
$\{\ y.(y = t \wedge \mathcal{Q}(t)) \vee (y \neq t \wedge \mathsf{U}(\mathtt{s.scheduled\_}, \mathcal{Q}) * \mathcal{P}_{send})\ \}$

The only non trivial part is making sure that the fetch and sub cannot read 0 from `s.scheduled_`. This simply requires the $CAS - \bot$, and the fact that $\ell^1 * \ell^1 \equiv \bot$: we cannot have more than 1 amount of permission for a single location (`s.scheduled_` here). This concludes the proof.

Note that here, we have that when a `dequeue()` operation is done, the predicate Consumer(`s.queue`) holds, as well as the fact that when writing to `s.protected_`, we have `s.protected_`^{1}.

5.4.4 Functions `drop` and `copy`

The proof of those functions is exactly the same as the one of the `copy` and `drop` functions provided for the Rust ARC in [9].

5.4.5 Some notes on constructors and destructors

Here the `drop` function only gives back permission to `s.parent_`. This is because as explained in the beginning of this chapter, the serial executor is not fully deleted when all references to it are removed. The queue and protected location are freed only when all tasks submitted to the executor have been executed, on top of the deletion of the executor. It would be interesting to further delve into the original code of the executor, to pinpoint the mechanism ensuring this deletion, and try to model it in FSL++.

Besides, as we model here the constructor of the serial executor as a `new` function, we have to use a release synchronization for the initialization of `s.scheduled_`, This synchronization is not present in the code, and is only needed here because we could not model the construction of an object properly: the fields of this object can only be accessed once its creation (and initialization happening within) have been finished.

Chapter 6

Conclusion and future work

In this project I investigated potential limitations of the token-based reasoning tool devised in [11], based on the EFC permission structure defined in [9], using FSL++ logic. At the beginning of this project, we expected to find limitations due to using only the EFC permission structure for ghost states, or to the simplifications further introduced by token-based reasoning. However, those new limitations were merely in ease of use of the system: those new tools are slightly diverging from our intuition of them, but we did not find any case where they actually restricted what could be proven. Surprisingly, the hard limits we encountered were those of FSL++. We showed that neither glibc reader writer lock [4] nor the Folly one producer one consumer queue [1] could be proven using this logic. Hence, while the infrastructure developed in [9] and [11] has proven efficient so far in harnessing the potential of FSL++ for automated reasoning, it seems that FSL++ is limited in some fundamental ways, especially for synchronization mechanisms using loads and stores, and not only read-modify-writes.

Finally, I provided the first, to the best of my knowledge, proof of the Folly Serial Executor. This proof demonstrates the effectiveness of FSL++ to prove correct intertwined independent synchronization mechanisms.

6.1 Future work

As developed above, the main limits we encountered were those of FSL++. This open at least two directions of work: trying to extend FSL++ to overcome those limits, or finding a new, more expressive, logic to build an infrastructure on. The first direction seems difficult, as the extensions would require some heavy changes to the logic, endangering its soundness proof. The second direction would require developing new abstractions and tools, and a more expressive logic would probably prove difficult to automate, at least partially. Finally, a third way would be to keep FSL++, and the infrastructure built in [9] and [11], for what they seem extremely efficient at:

proving correct synchronization mechanisms using only read-modify-writes, and exploring new examples that fit this category,

Bibliography

[1] Facebook folly one producer one consumer queue. `https://github.com/facebook/folly/blob/master/folly/ProducerConsumerQueue.h`, 2019 (accessed August 27, 2019).

[2] Facebook folly reader writer lock. `https://github.com/facebook/folly/blob/master/folly/RWSpinLock.h`, 2019 (accessed August 27, 2019).

[3] Facebook folly serial executor. `https://github.com/facebook/folly/blob/master/folly/executors/SerialExecutor.cpp`, 2019 (accessed August 27, 2019).

[4] Glibc pthread reader-writer lock. `https://sourceware.org/git/?p=glibc.git;a=blob;f=nptl/pthread_rwlock_common.c;h=8db861fdcb49f6d5a3fba6df151af8d38512d131;hb=HEAD`, 2019 (accessed August 27, 2019).

[5] Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In *European Symposium on Programming (ESOP)*, pages 448–475. Springer, 2017.

[6] ISO. *C11 Standard*, 2011. ISO/IEC 9899:2011.

[7] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, July 1997.

[8] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[9] Gaurav Parthasarathy. Applying and extending the weak-memory logic FSL++. Research in Computer Science Project, ETH Zürich, 2017.

[10] A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 190–209. Springer-Verlag, 2018.

[11] Pascal Wiesmann. Deductive verification of real-world C++ weak-memory programs. Master's thesis, ETH Zürich, 2019.

Appendix A

Glibc reader writer lock implementation

We show below a simplification of the code of the glibc `pthread_rwlock`[4]. In the following, we use the following abbreviations: Acq for acquire, Rel for release and Rlx for relaxed. While a lot of care was taken when transcribing this code, it may still be possible that some errors can be found, as a lot of simplifications were needed from the original code to this one.

The original code offers the possibility to prefer writers over readers. We removed the code allowing this. We also removed extra variables that were used for futex. These variables allowed threads to be added to a wait queue, until a particular change to the variable awoke them. This avoided the need for constant spinning on the `readers__` location. The correctness of the lock is not affected by those variables.

The constant `WRLOCKED` allows us to directly access the second least significant bit, denoting if there is a thread requesting or having write access, while `WRPHASE` corresponds to the least significant bit, telling if the lock is in read or write phase.

The number of threads currently having or requesting write access to the lock is `__readers / 8`, as the third least significant bit is used in the actual implementation. We do not mention it here, as it is only used when using the lock in a particular mode we are not interested in here.

In this code, each CAS instruction not only reads and updates the value of `readers__`, it also updates the value of `r` with the value it read in `readers__`. This is done by passing a reference to `r` to the custom compare and swap in the original code. We removed the explicit reference, but need to keep in mind this update.

```
Constants:
WRLOCKED = 2
WRPHASE = 1

int readLock(){
```

```
  int r = fetchAndAdd_Acq(__readers, 8) + 8
  if(r & WRPHASE == 0)
    return 0
  while(r & WRPHASE != 0 && r & WRLOCKED == 0){
    if(CAS_Acq(__readers, &r, r ^ WRPHASE))
      return 0;
  }
  //there is a writer (maybe in waiting),
  //it will get us to read mode at some point
  while(Load_Acq(__readers) & WRPHASE == 0){;}
}

int readUnlock(){
  int r = Load_Rlx(__readers)
  while(1){
    int rNew = r - 8
    if(rNew == 0){
      if(rNew & WRLOCKED)
        rNew |= WRPHASE
    }
    if(CAS_Rel(__readers, r, rNew))
      break
  }
}

int writeLock(){
  int r = FetchAndOr_Acq(__readers, WRLOCKED)
  if(r & WRLOCKED){
    while(1){
      if(r & WRLOCKED == 0){
        if(CAS_Acq(__readers, r, r | WRLOCKED))
          break
        continue
      }
      r = Load_Rlx(__readers)
    }
    r |= WRLOCKED
  }
  if(r & WRPHASE)
    return 0
  while(r & WRPHASE == 0 && r / 8 == 0){
    if(CAS_Acq(__readers, r, r | WRPHASE)){
      return 0
    }
  }
  while(Load_Rlx(__readers) & WRPHASE == 0) {;}
  return 0;
}

int writeUnlock(1){
  r = Load_Rlx(__readers)
  while(1){
    int rNew = r ^ WRLOCKED
    if(r / 8 > 0)
```

```
      rNew ^= WRPHASE
    if(CAS_Rel(__readers, r, rNew))
      break;
  }
}
```

A.1 Implementation explanation

Let us now explain in a bit more details how this code works. Note that this is not necessary to understand why we cannot use FSL++ to prove this code. It is however interesting, as it provides a completely different approach to the reader writer lock than the one offered by the Folly library [2].

To approach this code, we will first outline the protocol readers and writer abide by, and then explain how each function uses and enforces that protocol.

Definition 4 (Unlock Protocol)**.** *Both the `readUnlock()` and `writeUnlock()` functions abide by the following:*

1. *They are genial: when a writer (resp. reader) unlocks, if there are other readers (resp. writer) waiting, they will change the phase from write to read (resp. read to write). Note that a `readUnlock()` can only do so if there are no other readers (`readers__ / 8 = 0`);*

2. *They prefer their kind: when a writer (resp. reader) unlocks, if there are no readers (resp. writer) waiting, they will not change the phase, leaving it as write (resp. read).*

A.1.1 Required synchronization

We want to make sure this lock enforces proper synchronization, that is to say there cannot be any data races on the location it protects from multiples threads using the lock. To enforce this it is sufficient to make sure that every unlock contains a release write to `readers__`, while every lock contains an acquire read to this location. The release write forbids any memory actions from the first thread to be re-ordered past the write, which synchronizes with the acquire read reading from it, and the latter forbids any memory actions by the second thread to be re-ordered before the read. It is not sufficient to have an acquire load in the lock function for this to hold: this acquire has to be the one that signals the lock begin acquired, that is to say the last one before the function returns. Those conditions are satisfied by the lock and unlock functions of this thread. We will hence not mention these synchronization points in the remainder of this section.

A.1.2 `readLock()` function

Let us first focus on the read lock. This function first increases `readers__` by 8, that is to say signals itself as a new aspiring reader. It then checks what used to be the value r of `readers__` before it incremented it. If this value indicates a reader phase (that is to say `r && WRPHASE` $= 0$), the read lock was acquired and the function returns. If `r` indicates a write phase, there are two cases. If there is no writer thread currently (that is to say `r && WRLOCKED` $= 0$), the function tries to set the phase to read. This is done in a loop, as the attempt may fail if sime other reader thread incremented `readers__` in the meanwhile. This loop is repeated until it succeeds, or some other thread sets the phase to read, or some writer thread gets the lock (by setting the least second bit to 1). If the loop did not succeed, we are in one of the two following cases: either some other reader thread has set the phase to read, or some write thread got the lock. In both cases, it is enough to wait until we read that the phase is now read. In the latter case, we rely on the first part of the unlock etiquette given in Definition 4.

A.1.3 `writeLock()` function

The `writeLock()` function first uses a fetch and or to set the `WRLOCKED` bit to 1. It then checks what used to be the value `r` of `readers__`. If `r && WRLOCKED` $= 0$, it succeeded in signaling itself and can move on. If it is not the case, the fetch and or had no effect: the current thread was not registered as an aspiring writer. It hence needs to enter a loop, trying to set this bit from 0 to 1, using a compare and swap. This loop uses a relaxed load to lessen the synchronization: this load checks on `readers__`, and if it notices that the compare and swap could succeed (ie the second least significant bit is 0), the `if` statement containing the compare and swap is attempted[1].

Once the second least significant bit of `readers__` has been successfully set to 1, we check if we are in read phase. If this is the case, we simply return (note that the latest read to `readers__` is then the compare and swap acquire, or the fetch and or acquire, fulfilling the synchronization conditions outlined in section A.1.1). If this is not the case, either there are no readers, in which case we try to set the phase to write using another compare and swap. We keep trying this until we either succeed, or there are readers. If we did not succeed, we then check on `readers__` until it is in write phase, relying on Definition 4.

A.1.4 `readUnlock()` function

This functions simply repeatedly attemps a compare and swap until it succeeds. The new value is chosen so that we respect Definition 4: if there are

[1]When the compare and swap is attempted, it updates the value of `r`. If it fails, we can hence skip reading (using the relaxed load) `readers__` again, using the `continue`.

no more readers, and an aspiring writer, we set the phase to write.

A.1.5 `writeUnlock()` function

Similarly, this function uses a compare and swap, and sets the phase to read if there are any aspiring readers.

Appendix B

Folly ProducerConsumerQueue implementation

We show below a simplification of the code of the Folly ProducerConsumerQueue [1].

```
class ProducerConsumerQueue{
  int size_;
  T* records_;

  //Atomic locations
  int readIndex_;
  int writeIndex_;


  ProducerConsumerQueue(int size){
    size_ = size;
    records = malloc(sizeof(T) * size);
    readIndex_ = 0;
    writeIndex_ = 0;
  }

  bool write(Arg recordArg){
    int currentWrite = Load_Rlx(writeIndex_);
    int nextRecord = currentWrite + 1;
    if(nextRecord == size_){
      nextRecord = 0;
    }
    if(nextRecord != Load_Acq(readIndex_)){
      records[currentWrite] = new T(recordArg);
      Store_Rel(writeIndex_, nextRecors);
      return true;
    }
    return false;
  }
```

```
  bool read(T& record){
    int currentRead = Load_Rlx(readIndex_);
    if(currentRead == Load_Acq(writeIndex_))
      return false; //queue is empty
    int nextRecord = currentRead + 1;
    if(nextRecord == size_){
      nextRecord = 0;
    }
    record = records_[currentRead];
    records_[currentRead].~T();
    Store_Rel(readIndex_, nextRecord);
    return true;
  }

  T* frontPtr(){
    int currentRead = Load_Rlx(readIndex_);
    if(currentRead == Load_Acq(writeIndex_))
      return nullptr;//queue is empty
    return &records_[currentRead];
  }

  bool isEmpty() const {
    return Load_Acq(readIndex_) == Load_Acq(writeIndex_);
  }

  bool isFull() const {
    int nextRecord = Load_Acq(writeIndex_) + 1;
    if(nextRecord == size_)
      nextRecord = 0;
    if(nextRecord != Load_Acq(readIndex_));
      return false;
    return true;
  }

  //if called by consumer, true size may be more
  //if called by producer true size may be less
  size_t sizeGuess() const {
    int ret = Load_Acq(writeIndex_) - Load_Acq(readIndex_);
    if(ret < 0)
      ret += size_;
    return ret;
  }
}
```

Appendix C

Serial Executor implementation

We present below a simplified code for the Folly SerialExecutor. What this executor does is explained in more details in Chapter 5. The code ruling the behavior of this executor can be found in the following files:

- `https://github.com/facebook/folly/blob/master/folly/executors/SerialExecutor.cpp`
- `https://github.com/facebook/folly/blob/master/folly/executors/SerialExecutor.h`
- `https://github.com/facebook/folly/blob/master/folly/Executor.h`

.

As the code is originally in C++, it makes heavy use of pointers. Remember that in C++, objects are manipulated as values, whereas they are manipulated as references in Java. Hence when passing an object around in C++, if we do not make use of pointers, we make multiple copies, and lose access to the original object. The simplified code presented in Chapter 5 is redacted in a more Java like simplified language, allowing us to avoid using pointers.

If one were to dive back in the original code, it is worth knowing that the `Executor` pointer encapsulated by the `KeepAlive` class is used to store a flag in its least significant bit, on top of storing the actual value of the pointer. This does not cause any problem, as this pointer is aligned in memory to multiples of at least 4 (as it contains some integers). Hence, this pointer is a multiple of 4, and setting its least significant bit to 0 allows us to get back to the original pointer at anytime. We have here removed this flag, as it would always be set to true in the case of the `SerialExecutor`.

Finally, in the simplified version presented and proved correct in Chapter 5, we removed the `KeepAlive` class to make the code simpler, and avoided constructors and destructors to replace them with functions `newSerialExecutor`, `drop` and `copy`. It could be interesting to see if it is possible and not too cumbersome to prove a code making use of the `KeepAlive` class, as well as constructors and destructors.

As a side note, we kept the function names `keepAliveAcquire` and `keepAliveRelease`, even though they are unrelated to the actual synchronization used inside of the functions.

```
class SerialExecutor{
  Executor parent_;
  size_t scheduled_; //atomic
  UnboundedQueue queue_; //multiple producers single consumer
  int keepAliveCounter_; //atomic

  SerialExecutor(parent){
    parent_ = parent;
    queue = newQueue();
    scheduled_ = 0;
    keepAliveCounter_ = 1;
  }

  void add(Func func){
    queue_.enqueue(func);
    parent_->add({this->run()});
  }

  void run(){
    if(scheduled_.fetch_add(1, acquire) > 0)
      return;
    do {
      Func func = queue_.dequeue();
      func();
    } while(scheduled_.fetch_sub(1, release) > 1);
  }

  void keepAliveAcquire(){
    int c = keepAliveCounter_.fetch_add(1, relaxed);
  }

  void keepAliveRelease(){
    int c = keepAliveCounter_.fetch_sub(1, release_acquire);
    if(c == 1)
      delete this;
  }

  KeepAlive getKeepAliveCounter(Executor* executor){
    executor->keepAliveAcquire();
    return KeepAlive(executor, false);
  }
```

```
  KeepAlive create(Executor parent){
    return KeepAlive(SerialExecutor(parent));
  }
}

class KeepAlive{
  Executor* executor_;

  KeepAlive(KeepAlive other){
    *this = getKeepAliveToken(other.get());
  }

  KeepAlive(Executor* executor, bool v) : executor_(executor){}

  ~KeepAlive() {
    executor->keepAliveRelease();
  }

  Executor* get(){
    return executor;
  }

  KeepAlive copy(){
    executor.keepAliveAcquire();
    return KeepAlive(executor);
  }
}
```