Research internship report

Non confidential

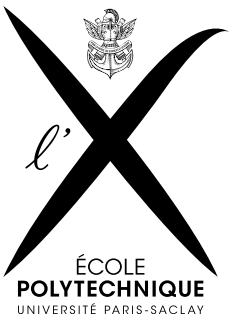
# Verifying weak memory programs in the Viper ecosystem

*Author:*
Anouk Paradis
Master of Computer Science
ETH

*Supervisors:*
Gaurav Parthasarathy
PM Group
ETH

Pr. Alexander J.Summers
PM Group
ETH

August 25, 2019



# Contents


**1 Introduction** **2**

**2 Background and related work** **4**

**3 Ease of use limitations of the EFC ghost location and tokens** **5**
3.1 The Folly reader-writer spinlock . . . . . 5
3.2 Proof using the EFC permission structure . . . . . 6
3.3 Proof using token-based reasoning . . . . . 8

**4 Limitations of FSL++** **10**
4.1 glibc Reader-Writer Lock . . . . . 10
4.1.1 Implementation . . . . . 10
4.1.2 Read-Modify-Write and Load . . . . . 12
4.1.3 Separation of tokens and permission . . . . . 14
4.2 Folly One Producer One Consumer Queue . . . . . 14
4.2.1 Re-using values for location invariants . . . . . 16
4.2.2 Infinite queue . . . . . 17

**5 New proven examples** **19**
5.1 Implementation . . . . . 20
5.1.1 Ensuring sequential execution . . . . . 21
5.1.2 Ensuring proper deletion when deleting references . . 22
5.2 Properties of the `SerialExecutor` . . . . . 23
5.3 Formalizing the specifications of the `SerialExecutor` . . . . 24

**6 Conclusion and Future work** **27**
6.1 . . . . . 27

**Bibliography** **31**

# Chapter 1

# Introduction

Concurrent programs opened the way for tremendous gains of performance, and are now everywhere in our environment, from mobile phones to critical plants control. In spite of their ominpresence, their subtle functionning is still poorly understood, and there is not yet any fully satisfactory model for any widely used programming language, such as Java or C++. The need for such a model is patent : weak memory behaviors are not only non deterministic, that is to say they do not occur at every execution of a program, but they also depend on the processor used, and the optimizations the compiler performed. Bugs hence are not often exposed, and become difficult to detect, and even more difficult to track down. Furthermore, relying solely on testing, even extensive, and programmer's folklore about what should or should not happen means that the introduction of a new processor or compiler optimisation could uncover new behaviors in a program.

However, when designing such a model, one is faced with two opposing challenges. To allow for reasoning and verification of programs, the model should provide strong garanties about the permitted behaviors. Alas, it should also allow for performance, and hence for the counterintuitive relaxed behaviors hardware and compiler optimisations may introduce.

On top of those two antagonistic intents, such a model should have the following characteristics to be fully usable. First, as most programs today are not built as one monolithic block of code, but rather as mutliple interconnected blocks, that might even be running on different processors, the model should aim for compositionality[1]. That is to say it should allow reasoning on the whole program to be the composition of reasoning on each block. Second, it should be usable by programmers. This can be done through different means. First the model should aim to be as simple and intuitive as possible. A simulator of the model on simple executions, as was provided by [2] for the C++11 standard is also of great help. Such a simulation is not always yet possible, for instance for [3], as it requires quantification over all future possible memory states. Another important

way to improve usability of the model is the so-called DRF-SC theorem. This theorem relies on the notion of sequential consistency [4] : the behavior of a program is sequentially consistent if it can be explained by a simple interleaving of the different threads actions. The DRF-SC theorem is then stated as : if a program is data race free for any sequentially consistent execution, then all its executions (even the relaxed ones) do not exhibit any behavior that is not exhibited by a sequentially consistent execution. It is extremely usefull, as it allows programmers to completely ignore weak memory behaviors when building a program with a strong lock discipline. Hence only programs with a strong need of performance need to deal with weak memory behaviors.

**Note :** The topic of my internship was initially "An RCU specification in fixed C++ concurrency". However it changed to focus on working on the memory model itself, as it was not yet usable for an application to the RCU algorithm.

**Contributions** When I began my internship, a simple language and its semantics had already been defined, as well as a model that could deal with dependancies. A more complicated model had also been defined to deal with fences and ordering. However, it did not work properly for many widely accepted small programs. At the end of my four months internship, we were able to develop a different model, using the idea of dependency developped in the first model, that could also deal with fences and atomics, as they are defined in the C++ language. It behaves properly on the litmus tests demonstrating many weak behaviors introduced by Power Multiprocessors, as presented by [5]. We also defined a similar model for non atomics and locks and were able to prove that the DRF-SC theorem holds for this simplified model. Finally, these models are built compositionaly, that is to say we have a representation even of incomplete programs, and can compose them together. This could allow for reasoning on a subpart of a program, and using those results when reasoning on a whole program.

# Chapter 2

# Background and related work

djksljflkdsjfldks

fetch and add + cas - false rules (fig 3 et 4 de gaurav p/13)

# 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 [**?**] to explicit some limitations of the EFC monoid defined in [**?**] and the token-based reasoning developped in [**?**].

## 3.1 The Folly reader-writer spinlock

The Folly reader-writer lock is used to allow multiple threads to access concurrently a shared ressource *res*, with either reader or writer priviliege. 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 **??**. This implementation is taken from [**?**].

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 *lsbbits*)indicates wether 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 *lsbbits* 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 by 4 *bits* 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`, 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 **??**, the EFC permission structure is based on the idea of giving entities, containing some amount of permission, 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 ot 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:

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

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

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 3.1 the function specifications we would like to prove. We can easily prove the specifications for the `try_lock_shared` functions, as shown in Appendix **??**.

Note that here the tokens represented by the ghost location do not hold any permission by themsleves, 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 3.1: Specifications of the lock functions

We use the fetch-and-add rule shown in Figure **??**, 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 value $v$, 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 shown in Figure **??** 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

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

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

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

/captionSimple invariant and Read-Write permissions for the Folly Reader-Writer Spinlock /labelfig:invToksRWFolly

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 $\lambda : (3, 1)^-$ could denote either that 3 readers each got permission 1/3 to the ressource, or 1 writer got full permission, and to threads are trying to get reader access (and are currently running the `try_lock_shared` function). We could not express using only one ghost locationt that the first case cannot happen.

While this example shows that the EFC permission structure is a bit less expressive can 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 [**?**]. 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 developped in [**?**]. 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 easilly be overcome, it is at the price of slightly less intuitive proofs. We will see here that the token-based reasoning devolopped in [**?**] is itslef weaker than the EFC permission structure it was based on.

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

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, we could then combine the source with our read token to get $\texttt{Src}(res, 0, read)$, which using the ALL-TOKENS-A 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 is could happen that a threads 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 wether or not it now owns full permission or not. This could not happen using the EFC ghost state directly as such a downgrading of tokens 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 easilly fixed, by making the proof a little straightforward. We can for instance use the idea developped in [**?**] 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 the 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. 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 synchronisation 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 **??**. This implementation was heavilly simplified from the original one. In particular, futexes (special variables used to wake-up a thread when an event occurs) which where used to prevent extra spining where removed. While they are essential to make the code more efficient, they are not part of the synchronization mechanisms used to ensure correctness of this lock.

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 implmentation uses a single atomic location `__readers`. This location is as showed in Figure **??**. The least significant bit is used to denote

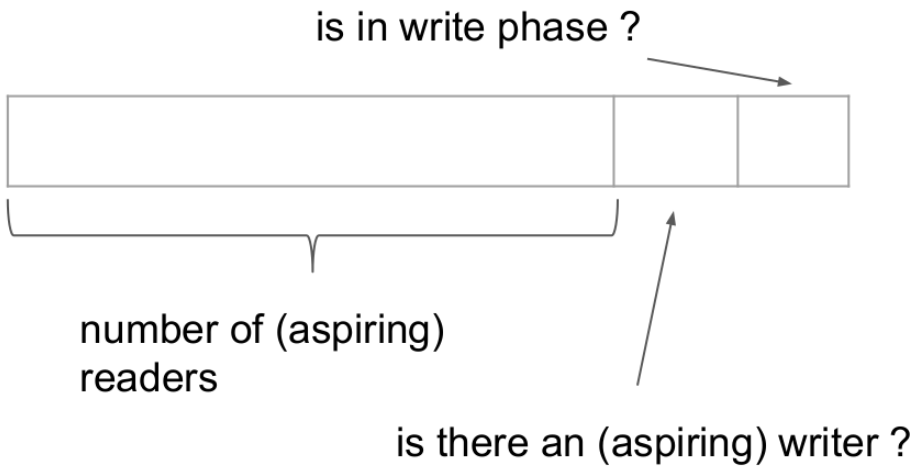



Figure 4.1: glibc reader-writer lock

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 wether or not there is 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. Hence if this atomic location contains 2|1|1, that 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 significatn bit to 0, allowing for read access.

Each of the function 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.

The full code for the lock is quite long, and 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 scenarii to support our reasoning. They are shown in Figure **??** and Figure **??**. 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 to 1. We then check if the least significant bit (corresponding to `WRITEPHASE`) is set to 0. As it is the case, the function returns: the thread now has a read lock. In the second scenario, Thread 1

```
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

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 substracting 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 scenarii 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 ressources from a single atomic location both using a Load and a fetch-and-add. When looking further into it becomes quite clear while such a rule

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

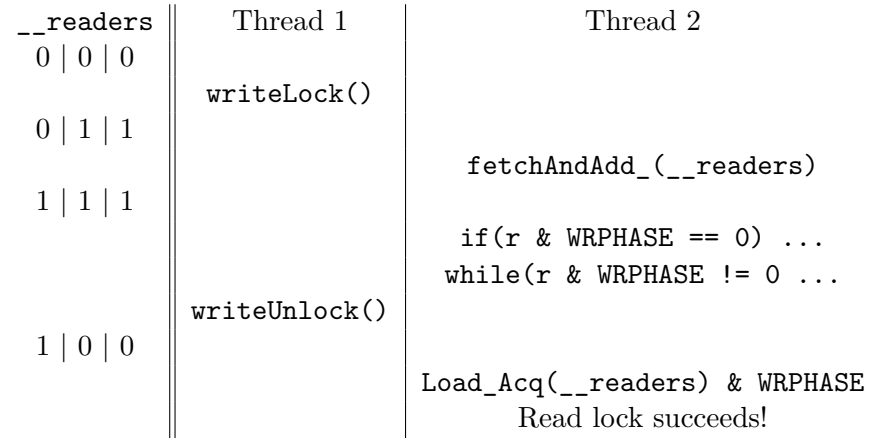



Figure 4.4: Second `readLock()` execution scenario

would be quite difficult to implement.

In FSL++, atomic locations are used to transfer ownership (here to the ressoucres protected by the lock) between threads. It soed so by using a location invariant $\mathcal{Q}$ which for each value $v$ gives the ownership corresponding to this value. Then whne 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 heere, 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 choice between $\mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \mathcal{Q})$ or $\mathsf{RMWAcq}(\ell, \mathcal{Q})$. The

[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.

latter is then freely duplicable and can be transmitted to all threads requiring it. There is howver no rule that would allow to have both $\mathsf{Acq}(\ell, \mathcal{Q}_\infty)$and $\mathsf{RMWAcq}(\ell, \mathcal{Q}_\in)$ 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 **??**, 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 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 function that we will not discuss here is included in Appendix **??**. 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 indexes `readIndex, writeIndex`. When creating the object, both index 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. Hence in spite of its name, it is not enough for this function to have only read permission to the location. This means thatthe 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 read

```
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

is relaxed. It is used here only tom have a nicer interface for the function: as there is only one reader 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` symchronization. This synchronization ensures that all reads or writes that come after will not be re-ordered before. We then check wether 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: the producer thread stores a value in `records`, and the consumer thread then reads it. Part of this scenarion is shown in Figure **??**. The release store to `writeIndex` ensures that any memory operation happening before it will be visible to other threads when they see the value release. In particular, any thread reading the new value of `writeIndex` will have the proper value stored in `records[curWrite]`. This is materialized by the red arrow. Then the acquire load ensures that any memory operation after it will not bere-ordered. In particular, reading in

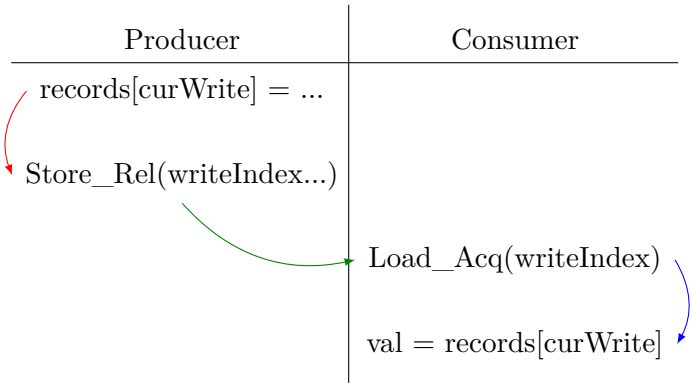



Figure 4.6: Synchronization in the Folly one producer one consumer queue

`records[curWrite]`[3] will happen after this value 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 betwenn the write access to `records[curWrite]` 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 developped there. The rule for a read[4] 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 $\ell$ 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

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

However, as noted when presenting the implemantation of this queue, both the `read` and `write` function require full ownership. As full ownership

[3] In the more general case, the consumer thread reads in `records[curRead]`, for simplification reason, we assume here that `curRead = curWrite`.

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

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.

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 cirular buffer is used again and agin. 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, as it is used as a circular buffer, we cannot prove it using FSL++. A simplification that immediatly comes to mind would be to use an infinite array fo r`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 can be found in Appendix **??**.

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 $\mathcal{Q}(\sqsubseteq)$. However, if this is not the case, `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 are 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 intermidiate states of the variable are observed by one of the threads, as 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. Ths idea hence cannot be used here.

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

# Chapter 5

# New proven examples

In this section we will focus on a new example we were able to prove using the EFC permission structure. While we did not have time to complete this proof using token-based reasoning, it is likely to be possible.

This new example is the Folly Serial Executor. An executor allows a thread to delegate the call of a task, by simply adding this task to the executor. Depending on its characteristics, the executor will then execute this task depending on its priority, or sequentially as is the serial executor case, or any other ordering. This is quite similar to what a thread scheduler does: it is given some tasks (the threads), and then choses how to run them.

In the case of the serial executor, the executor ensures that all the tasks that are added to it are executed sequentially. 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 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 schedule those tasks one after the other, making sure not two tasks try to wrtie to the port at the same time for instance.

Note that it is possible that multiple threads execute the tasks assigned to the serial executor over time. The serial executor simply garantees that this will not be done concurrently: we will not have two threads executing some tasks from the serial executor at the same time.

Besides, a serial executor garanties 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]

[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.scheduled_ = 0;
  s.keepAliveCounter_ = 1;
  s.queue_ = new Queue();
  return s;
}

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

void run(Executor s){
  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

## 5.1 Implementation

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

An executor is made of four fields. The field `parent_` refers to its parent executor, the one it will delegate the tasks to. Indeed, this serial executor is merely a middleman here. It records all the tasks that are submitted to it, then transforms into one continuous task that it submits to its parent, another executor.[2] Note that we will hence be talking about two different

[2]There is no infinite delegation here, at some point the executor's parent will be the CPU scheduler directly.

kinds of tasks in the following dicsussion: the ones that were submitted to the serial executor, and the new ones that the serial executor created and submitted to its parent executor. The field `queue_` is used to record the task to be executed. The queue that is used here is a concurrent single consumer multiple producer queue. The field `scheduled_` is used to ensure sequential execution. Finally the field `keepAliveCounter_` is used to track the number of references to this executor, and deleting it when no one holds references anymore. The last two fileds 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 this executor `run` function.

The `run` function is the one ensuring serial execution of the tasks. To do so it uses the `scheduled_` location. TO explain how it works, it help to think about what could have been an alternative implementation of this function. We could have simply used `scheduled` as a flag recording wether or not there is currently a task executing. The simplified code would then be the follwing

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

We first try to set atomically the `scheduled_` to 1. If this succeeds, this is now the only 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 task (not added to the serial executor, but created by it) releases the `scheduled_` flag by setting it to 0. However, this function requires to calls to `queue_` per loop execution. One checking if it is empty or not and the second popping an element[3]. 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 in Figure **??**.

[3]Note that here, there is no risk of having a non empty queue qhen 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.

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 taks contained in `queue_`. This is similar to the case where the `CAS` 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[4]. 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 shwon above, `scheduled_` has been set back to 0, allowing another task created by the serial executor to start executing later.

We explained above the idea of th e`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 fecth and sub brings `scheduled_` back to 0. Thread 1 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, all the memory events done by Thread 1 as part of executing the tasks stay before all the memory acctions 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 used 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:

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

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` have gone, it is deleted, but not before that. They do so in a fashion extremely similar to that of the Rust Atomic Reference Counter. `keepAliveCounter` tracks the number or references to the serial executor. Those references can be obtained through the `copy` function, and deleted

[4]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_subb`, `scheduled_` is smaller or equal to the number of elements in the queue.

using the `drop` function. The latter then deletes the serial executor if the reference dropped is the last one.

The code is almost identical to the Rust ARC. The only difference is in the `drop` function. In the Rust ARC, the code is the following (taken from [**?**])

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

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 behavior as modeled in FSL++[5].

For a more indepth explanation of the synchronization mechanisms at play here, refer to [**?**].

## 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. 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 fonctions 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 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 producer. 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:

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

$$\mathsf{Producer}(\texttt{q}) \iff \mathsf{Producer}(\texttt{q}) * \mathsf{Producer}(\texttt{q}) \quad \text{(producer duplication)}$$

[5] 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, while a load acquire only forbid re-ordering between the load itself and any subsequent read or write.

The (new queue) allows us to create a new queue, and creates permission for a consumer and a producer. The (**??**) equivalence allows us to duplicate the producer permission, allowing mutliple threads to add to the queue. These predicates should be thought of as some placeholders, that could be replaced with more precise predicates defining the behavior of a single consumer multiple producer queue in more details.

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[6] of the `SerialExecutor` requires that even if it is deleted, all the tasks submitted to it are will still be xecuted with the same garantees. Besides, in the original code the field `queue` of the `SerialExecutor` is not deleted upon deallocation of the executor. Thus 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 emty queue. However, this would require first defining and proving the properties of a single consumer, multiple producer queue.

repeat ( s.protected = a;
)

## 5.3 Formalizing the specifications of the `SerialExecutor`

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 ressources 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 garantees sequential 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, which can be rewritten as `protected_`[1].

[6] Read comments in the code

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

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

void run(Executor s){
  if(s.scheduled_.fetch_add(1, acquire) > 0) {}
  else{
    repeat (
      a = s.queue_.dequeue();
      s.protected_.set(a);
      s.scheduled_.fetch_sub(1, release) > 1
    )
  }
}

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

$$\{\exists f \in \mathbb{Q} \cap (0,1) * \texttt{protected}^1\} \texttt{newSerialExecutor}(\texttt{parent}, \texttt{protected}) \{s. \exists \lambda. \mathsf{RE}_\lambda\}$$

Figure 5.3: Serial executor fonctions specifications

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

Finally, for the final property, we need to show that there exists a predicate $\mathsf{SE}_\lambda$ such that the function specifications shown in Figure 5.3 hold.

# Chapter 6

# Conclusion and Future work

## 6.1

We show below a simplification of the code of the glibc `pthread_rwlock`. In the following, we use the following abreviations: 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 still be found, as a lot of simplifications were needed from the original code to this one.

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.

```
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;
  }
}
```

We present below a simplified code for the Folly SerialExecutor. What this executor does is explained in more details in Chapter **??**. The code rulling the behavior of this executor is seperated between `folly/executors/SerialExecutor.cpp`, `folly/executors/SerialExecutor.h` and `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++,ăif we do

not make use of pointers, we make multiple copies, and lose access to the original object. The simplified code presented in Chapter **??** 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 remove this flag, as it would always be true in the case of the `SerialExecutor`.

Finally, in the simplified version presented and proved correct in Chapter **??**, 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_;

  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);
  }
}
```

i

# Chapter 7

# Bibliography


[1] Mark Batty. Compositional relaxed concurrency. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 375 2104, 2017.

[2] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[3] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL*, 2017.

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

[5] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.