Jthread, a deadlock-free mutex library

Johan Grande
Université Nice Sophia Antipolis
Les Algorithmes, Bât. Euclide B - BP 121
F-06903 Sophia Antipolis, Cedex
France
Johan.Grande@ens-cachan.org

Gérard Boudol
Inria Sophia Méditerranée
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France
Gerard.Boudol@inria.fr

Manuel Serrano
Inria Sophia Méditerranée
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France
Manuel.Serrano@inria.fr

Abstract

We design a mutex library for Hop – a dialect of Scheme which supports preemptive multithreading and shared memory – that mixes deadlock prevention and deadlock avoidance to provide an easy to use, expressive, and safe locking function. This requires an operation to acquire several mutexes simultaneously, for which we provide a starvation-free algorithm. Choosing a formal definition of starvation-freedom leads us to identify the concept of *asymptotic deadlock*. Preliminary observations seem to show that our library has negligible impact on the performance of real-life applications. Our work could be applied to other languages such as Java.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.4.1 [*Operating Systems*]: Process Management—Threads, Mutual exclusion, Deadlocks, Scheduling

Keywords shared memory concurrency, mutex, structured locking, nested lock, asymptotic deadlock, deadlock avoidance, starvation, modularity, formal semantics, language design, Scheme

1. Introduction

Shared memory concurrency, aka multithreading, is a popular concurrent programming paradigm that allows one to take advantage of multi-core architectures, now widely available in personal computers. A well-known issue with this programming style is the coordination of the threads which is required to avoid that concurrent accesses to a same memory location yield unpredictable results. Mutexes are commonly used to deal with this as a mutex allows a thread to temporarily claim a given resource and make the other threads that need to access it wait for their turn.

Using mutexes raises a serious problem, namely the possibility of *deadlocks*, which have been qualified, together with starvation, as "the plague of concurrency." The notion of a deadlock is well known: this is a situation where, during the execution of a concurrent program, two or more threads are waiting for each other to progress, while not being able to progress on their own, thus being indefinitely blocked. Deadlocks are difficult to debug because they may go unnoticed for some time if only a few of many threads of a program stop working. They are even more complex to debug in the context of pthread or Java libraries as they ensue from the non-deterministic scheduling of threads.

Three groups of methods to solve the problem of deadlocks have been identified long ago [4]:

- deadlock prevention;
- deadlock avoidance;
- deadlock detection and recovery.

The last technique is similar to optimistic concurrency control in database transactions implementation. The other two are closer to the programming languages area. One could regard deadlock avoidance as a dynamic form of deadlock prevention, where the run-time decision to enter a critical section, by locking a mutex, is deferred, based on the presumption that this could lead to a deadlock in the future. Deadlock avoidance is the purpose of Dijkstra's well-known Banker's algorithm [6], and this technique is the topic of some recent works [9, 14], but deadlock detection and deadlock prevention are far more popular approaches (see [3, 7, 10, 19] for the latter).

Deadlock prevention and avoidance techniques both *reduce the parallelism* in the program, by disallowing some paths that could otherwise be taken. Of course this reduction should be as minimal as possible. For deadlock avoidance, the prediction that a locking decision might lead to a deadlock should be as precise as possible and computing it as fast as possible.

We have designed and implemented a full-fledged mutex library that combines the approaches of deadlock prevention and deadlock avoidance. Programs using this library never run into a deadlock. Our library also offers condition variables, but with no such guarantee. The library can be used by static and dynamic languages as it does not assume type informations but in this paper we present its integration and implementation in the Hop language [15], a Lisp dialect supporting multithreading and shared memory. We shall use the syntax of this language throughout the paper.

The main idea is taken from the work [2] by one of the authors, which we recall briefly. This work relies on a construct[1]

```
(synchronize m body)
```

for *structured locking*, by which a thread attempts to lock the mutex `m` for the scope of a *critical section* `body` of code, and unlock it afterwards. A static analysis based on a type and effect system determines which mutexes are potentially locked while executing the critical section `body`. This information is used to translate the

[1] In the previously cited work, a different, ML-like syntax is used.

PPDP '15, July 14–16, 2015, Siena, Italy.

ACM 978-1-4503-3516-4/15/07…$15.00.
http://dx.doi.org/10.1145/2790449.2790523

locking construct into an intermediary form where the list p of such mutexes is recorded at the synchronization point `synchronize m`. The mutex `m` is not locked unless all the mutexes in the list `p` are simultaneously free.

Let us see an example which can be seen as a fragment of the well-known "dining philosophers" setting:

```
;; thread t1                    ;; thread t2
(synchronize m  ; 1       ||    (synchronize n  ; 2
   ...                             ...
   (synchronize n  ; 3             (synchronize m  ; 4
      ...                             ...
      ))                              ))
```

This is the typical example of a program whose execution can lead to a deadlock: if the numbered points 1, 2, 3, 4 in this program are executed in this order, then both t_1 and t_2 are blocked. Recording at point 1 that the mutex `n` is potentially locked before `m` is released, the *prudent* semantics of [2] requires both `m` *and* `n` to be free before locking `m` at point 1 (respectively `n` *and* `m` for thread t_2). Then if t_1 has just locked `m`, t_2 has to wait for t_1 exiting from its critical section protected by `m` before locking `n`. Notice that following the classical technique of deadlock prevention, which is to impose an order in which mutexes can be locked, we could not even statically accept such a nested locking scenario.

The static analysis used in [2] assumes static type information that is not available for a dynamically typed language such as Hop, and therefore we had to find an alternative way to implement the prudent semantics for such a language. Our solution is to promote the intermediate locking construct of [2] as a first class synchronization construct, namely

```
(synchronize m [:prelock p] body)
```

where `p` is an (optional) list that contains the mutexes that the `body` might lock. We call `p` an *authorization* list. Indeed, if the `body` of the critical section attempts to lock a mutex that is *not* in this list, our implementation raises an error – there are exceptions to this rule that are described below. Then, as usual, we replace typing failures of statically typed languages by run-time errors that can be caught by some exception handler if needed. These errors are the symptom of a potential deadlock.

The reduction of parallelism entailed by the prudent semantics is *sufficient* to avoid deadlocks, but it might be *unnecessary*. For instance, in the following program, if points 1, 2, 3 are executed in this order, the prudent semantics does not allow t_2 to pass point 2 before t_1 exits its critical section, even though the program is deadlock-free anyhow.

```
;; thread t1                 ;; thread t2
(synchronize m  ; 1   ||     (synchronize n :prelock m ; 2
   ...                          ...
   ; does not lock n           (synchronize m  ; 3
   ... )                          ... ))
```

We know that this program is deadlock-free as it falls under the standard argument of the deadlock prevention approach: if `m` can only be locked after `n` and not the other way around, then there cannot be a cycle causing a deadlock, and t_2 could proceed without waiting for t_1 to complete its critical section.

This programming pattern is supported by our library by the means of a dynamic form of deadlock prevention that complements what we described to ensure deadlock avoidance: each mutex can be created in a *region*:

```
(make-mutex [region])
```

where the optional argument `region` is just a name (which defaults to a fresh name when omitted). Regions are ordered at run-time. An exception is raised if an execution does not comply with this dynamically established order. (This is the classical technique of deadlock prevention, in a dynamic setting). In the previous example, if in the context of these two threads the mutex `m`, belonging to some region R_m, is never locked before `n`, one can safely gain some parallelism in the execution by removing the `prelock m` part, as no exception will be raised when locking `m` in thread t_2, even though `m` is not explicitly authorized when locking `n`.

Again a failure to comply with the ordering of regions appears as an explicit run-time error, which is another symptom of a potential deadlock. In the event of such an error, for instance with nested locking scenarios exemplified by the simple "dining philosophers" above, the programmer has to resort to the prudent semantics, modifying the program by making the appropriate authorizations explicit in the code. (In this case the mutex to be locked and the authorized ones belong to the same region.) We believe that rather than letting the program silently run into a deadlock, it is much better, for debugging and program development purposes, to let the potential deadlocks appear as explicit run-time errors, thus providing information about how a deadlock could happen. In a sense, although the purpose of the jthread library is to completely avoid deadlocks, our approach also offers a form of (potential) deadlock detection and a limited form of recovery, via error handling.

Deadlock avoidance using only prelocks is incompatible with separate compilation as the program must mention, and then must *know*, all the mutexes potentially locked on a synchronization point. Region-based deadlock prevention removes this limitation. Regions achieve modularity by assuming that the mutexes in a module N are in regions that are smaller, in the region ordering, than those of a module M depending on N.

The paper describes the Hop implementation of our library. Our work could be implemented for another host language, provided that this language relies on structured programming (no `goto`s), and only uses `synchronize` blocks instead of separate `lock`/`unlock` functions. Java, for instance, complies with these two requirements. In Hop we use the existing *pthread* (POSIX threads) library to provide ourselves with an underlying mutual exclusion mechanism in order to implement our library which we call *jthread*. The basic idea to build the implementation of the prudent semantics is very simple and natural: executing a `(synchronize m :prelock p ...)` consists in

- acquiring the desired mutex `m` and the mutexes from the list `p` in a row;
- unlocking the mutexes in `p`.

(we say that the mutexes in `p` are only "prelocked", since they are immediately released). However, such a naive implementation does not guarantee the absence of starvation, and therefore we have added a scheduling mechanism to achieve such a property, in addition to deadlock avoidance. We are inspired here by Lamport's Bakery algorithm [11] and its generalization to several resources [12]. However, unlike the Bakery algorithm, our solution does rely on an underlying mutual exclusion mechanism, but it allows nested locks.

In order to define starvation-freedom in a context with nonterminating programs and arbitrary numbers of threads and mutexes we introduce the concept of *asymptotic deadlock*. We use this term to describe a situation in which, while there is no dependency cycle between threads, a thread t depends on a number of threads that tends toward $+\infty$ and is thus indefinitely blocked. We could not find any usage of this concept in previous works. We then prove that programs using our library are starvation-free under the hypothesis that no asymptotic deadlocks arise.

The reader can refer to the formal semantics given in Appendix A which summarizes the description of our library given in this paper.

Regarding the performance of our implementation, we did not hope to compete with pthread on which there is a huge worldwide effort, and indeed the raw locking/unlocking performance of jthread is much slower than that of pthread. However, this did not lead to a significant difference in performance between using pthread and jthread in the two realistic applications that we benchmarked. Moreover, in some cases jthread allows to express an efficient algorithm that would be prone to deadlocks if pthread was used; in such a case the program using jthread can be faster than its closest deadlock-free equivalent (with less parallelism) using pthread.

2. The deadlock-free mutex library

In this section we describe our jthread library offering mutexes and deadlock-free structured locking. We recall that a *mutex* (a shorthand for *mutual exclusion object*) is an object whose state can be either *locked* by a unique thread (the *owner* of the mutex) or *unlocked* (or *free*). A thread can become the owner of a mutex by *acquiring* it, *i.e.*, using a blocking operation that *waits* until the mutex is free and returns after *actually taking* it. Mutexes can be either first-class resources, or implicitly associated with other objects, as in Java.

A thread waiting for one or more mutexes is *blocked*. A thread is *making progress* if it is not blocked. We consider that a thread waiting on anything other than our mutexes (e.g. I/O), or for instance executing an endless empty loop, is making progress.

Languages featuring mutexes can offer independent functions to `lock` and `unlock` mutexes (*explicit locking*), or a single construct to lock a mutex for the scope of a *critical section* (or *synchronized block*) of code and unlock it afterwards (*structured locking*). For instance, the standard POSIX threads (*pthread*) library offers explicit locking, whereas Java relies on structured locking. When using explicit locking the programmer has to ensure that at runtime each call to `lock` is balanced with a call to `unlock`, whereas in structured locking this is enforced by the syntax. Structured locking versus explicit locking can be compared to structured programming versus programming with `goto`s. On the one hand, explicit locking lets programmers deploy very clever and tricky implementations. On the other hand, they are difficult to use correctly and to maintain because locks and unlocks can be deeply intricated.

A *recursive* mutex `m` can be acquired by a thread that already owns `m`. In various *non-recursive* mutex implementations, whenever a thread t attempts to acquire a mutex that t already owns, either t is indefinitely blocked or an error is raised. We developed recursive mutexes as in [2] but for the sake of simplicity we assume non-recursive mutexes in this paper; a thread t attempting to acquire a mutex that t already owns results in an error.

The jthread library supports structured locking and prudent nested locking with the form:

```
(synchronize m [:prelock p] body)
```

where `m` is the mutex to lock, and the optional parameter `p` is the list of mutexes to prelock (to be explained hereafter) which defaults to an empty list. The `body` consists in one or more expressions. As suggested in the introduction, the implementation of the structured locking primitive relies on an explicit locking primitive that locks a whole list of mutexes at once. More precisely, the `synchronize` form implementation relies on a `lock-n` function that takes as arguments a mutex and a list of mutexes (the `m` and `p` above); it locks all these mutexes, and immediately unlocks the mutexes from its second argument. The implementation of `lock-n` is the central element of the implementation of the *jthread* library. It is described in section 4 along with some optimizations. The `lock-n` operation is implemented on top of the standard, simpler mutual exclusion mechanism offered by the *pthread* library.

From now on, in case of ambiguity, identifiers pertaining to the pthread and jthread libraries will be prefixed by respectively `p` and `j`.

More precisely, our implementation creates one global pthread mutex named `big-lock`, and all the algorithms described hereafter are executed while owning `big-lock` (except during waiting phases). Here is a sketch of the implementation of the `synchronize` form:

```
(define (synchronize m p thunk)
   ;; m is the mutex to lock
   ;; p is the list of mutexes to prelock
   (p-synchronize big-lock
      [enforce deadlock avoidance rules]
      (lock-n m p)
      (try-finally
         (thunk)
         (unlock m) )))
```

The basic principle used to prevent deadlocks is the following: at a given point in the execution of a thread t, there may be some mutexes that t is not authorized to lock. By "authorized to lock a mutex" we mean: authorized to call the blocking function `lock-n` with this mutex as a parameter. Attempting to lock unauthorized mutexes with jthread's high-level functions raises an error at runtime. If authorizations are respected in a program, the execution is deadlock-free, and free from such errors. The jthread functions combine two complementary mechanisms for authorizations: an implicit one – *ordered regions* – and an explicit one – *prelocks* – that we now explain.

2.1 Regions

Jthread mutexes are grouped into *regions*, denoted R, which are used to control the safe nesting of locking. (Another part of this control is taken care of by the deadlock avoidance mechanism.) Each mutex belongs to a region, the same one for its whole lifespan. The programmer can create regions and may choose the region of a mutex at the time of its creation, with the following functions:

```
(make-region)

(make-mutex [region])
```

We chose to make the default region for a new mutex `m` be a new region which initially only contains `m` as it seems that this choice corresponds to the programmer's will in many cases.

Our implementation dynamically creates an ordering $R \geq R'$ of regions, which is used to forbid any circularity in locking mutexes that belong to different regions. More precisely, our implementation enforces the following rule, as regards the authorization to lock some mutex:

Rule 1: *A thread may lock a mutex* `n` *in region* R_n *while owning a mutex* `m` *in region* R_m *only if* $R_m \geq R_n$.
That is, executing the following code

```
(synchronize m
   ...
   (synchronize n
      ... ))
```

raises an error if the region R_m is not greater than, or equal to R_n. If $R_m \neq R_n$ this corresponds to the standard way of preventing deadlocks – acquiring nested locks in some fixed order –, although this is usually guaranteed by means of a static analysis (see [3, 7, 10, 19] for instance). From the rule above we can conclude that when a thread does not own any mutex, it is allowed to lock any mutex. Only nested locking is subject to restrictions.

The partial order on regions is constructed from one `synchronize` to the other as the minimal partial order that satisfies Rule 1. For instance:

```
(define m (make-mutex))  ;; new region Rm
(define n (make-mutex))  ;; new region Rn
(synchronize m
   (synchronize n  ;; no relation yet between Rm and Rn
                   ;; set Rm > Rn
      ... ))
(synchronize n
   (synchronize m  ;; relation exists: Rm > Rn
      ;; -> error
      ... ))
```

The algorithm for ordering regions is as follows: each thread has a stack of regions. The topmost region is called the *current region*. When the thread is started, its current region is initialized to Top, a region greater than all other regions. Then, when a thread is about to enter a critical section, executing

```
(synchronize m [:prelock p] body)
```

first checks whether `m` and the mutexes in the list `p` are in the same region, raising an error if not (the reason for this will become clear by the end of this section). If this test succeeds, and R_m is the region of the mutex `m`, it is checked whether the current region R associated with the thread is such that $R_m > R$. If $R_m > R$ an error is raised, because the thread might be attempting to lock mutexes in a dangerous order. Otherwise, if $R_m \neq R$, R_m is pushed onto the stack of regions associated with the thread. It then becomes the current region of the thread and the global ordering of regions is updated by adding the relation $R > R_m$ (again, if $R \neq R_m$). Otherwise ($R_m = R$), the stack of regions and region ordering are not modified. The current region is popped off the stack when the thread leaves the critical section.

Given this way of computing the region ordering, we can reformulate our Rule 1 as follows:

Rule 1 bis: *A thread in (current) region R may lock a mutex* `m` *in region R_m only if $R \geq R_m$.*

We shall not have any further requirement in the case where $R > R_m$. That is, the condition $R > R_m$ is actually sufficient to conclude that the thread is authorized to lock `m`. If, instead, $R_m = R$, any mutex from region R_m could be locked by the thread, according to this rule. For instance, if the programmer assigns the same region to all the mutexes, then all these mutexes would be authorized to be locked by any thread if we were to stop controlling the authorizations at this point, which is unacceptable. Nevertheless, we shall not definitively ban nested locking inside a given region since we argued in favor of allowing the programmer to use nested locking schemes that do not comply with a fixed locking order. For that, a second authorization mechanism called *prelocking* is needed.

2.2 Prelocks

When the program relies on nested locking in a way that would not be permitted by the region ordering discipline only, the mutexes must be grouped in a single same region, and the *prudent* semantics must be used. For this, our implementation enforces the following rule:

Rule 2: *If mutexes* `m` *and* `n` *belong to the same region, then a thread t may lock* `n` *while owning* `m` *only if, when locking* `m`*, t prelocked* `n`.

For instance:

```
(synchronize m :prelock n
   ;; mutexes m, n, and p belong to the same region
   ...
   (synchronize n  ;; authorized
      ... )
   ...
   (synchronize p  ;; unauthorized
      ... )
   ... )
```

When several `synchronize` forms are nested, the mutexes of the inner critical sections must be prelocked in all the superior levels [2]:

```
(synchronize m :prelock (list n p)
   (synchronize n :prelock p
      (synchronize p
         ... ))
```

In other words, mutexes that are not authorized to be locked are not authorized to be prelocked either:

```
(synchronize m :prelock n
   (synchronize n :prelock p  ; error: p not authorized
      ... ))
```

To enforce proper nesting and prelocking each thread has a variable set of explicitely authorized mutexes. This set is modified and restored respectively at the beginning and at the end of each `synchronize` block. Each time a thread enters a new region, the set of authorized mutexes is updated to be the prelock set of the current innermost `synchronize` block. Then we can reformulate Rule 2 into:

Rule 2 bis: *A thread t may lock and prelock mutexes that belong to its current region only if the mutexes are members of the prelock set of the innermost* `synchronize` *block t is in.*

In [2] it has been proved that this discipline, which is enforced in the prudent semantics, guarantees the absence of deadlocks when we consider mutexes that belong to the same region.

2.2.1 Newly created mutexes

If a thread creates a new mutex in the same region as its current region, Rule 2 does not grant the thread the authorization to lock it, whereas in [2] any newly created mutex is automatically authorized inside its lexical scope in the thread that creates it, and this is shown to be safe. Then we state a third rule, enforced by our implementation, as an exception to Rule 2:

Rule 3: *Every newly created mutex in region R is authorized in all currently executing* `synchronize` *blocks in all threads that are in the same region R.*

For instance

```
(define R (make-region))
(define m (make-mutex R))
(define n (make-mutex R))
(define x #f)
(synchronize m :prelock n
   (synchronize n
      (set! x (make-mutex R))
      ;; x is authorized
      ... )
   ;; x is authorized
   (synchronize n
      ;; x is not authorized
      ... )
   ;; x is authorized
   ... )
```

Authorizing newly created mutexes in this way does not yield any deadlock. Indeed, with respect to authorizations, a newly created mutex is equivalent to a mutex that would have been created at the beginning of the current thread and correctly prelocked to the current point, and never used in any other thread, which is a safe situation.

Rule 3 is enforced by giving mutexes increasing numerical identifiers and by making each thread entering a critical section memorize the (global) last given identifier. The threads can then determine whether a given mutex was created before or after this point. For example, if `last-id` is the global variable holding the last given mutex identifier:

```
; last-id = 0
(define R (make-region))
(define m (make-mutex R))      ; gets id = 1
; last-id = 1
(synchronize m                 ; memorizes last-id = 1
   (let ((n (make-mutex R)))   ; gets id = 2
      (synchronize n
         ... )))
```

At the second `synchronize`, `n` is in the same region as `m` and has not been explicitely authorized but `n`'s id is greater than the `last-id` at the time of `synchronize m`, therefore `n` is authorized.

2.3 Unsafe mode

In the previous subsections we described rules that the programmer has to follow and algorithms to check at runtime that they are respected, throwing errors otherwise. One can consider that those checks are a tool to debug a program and that there is no point in performing them once the program has been tested and debugged. (The checks in question should not be confused with the *use* of prelock sets which *are* needed at runtime to avoid deadlocks.) Then we offer an "unsafe" mode in which those checks are deactivated, in order to improve the performance of programs that are considered to be tested. However when we benchmarked our implementation the cost of our functions was dominated by the cost of the underlying functions described in Section 4; we could not measure a significant difference between the "safe" and "unsafe" modes of our library.

2.4 Condition variables

Condition variables, or *condvars*, are a common synchronization mechanism. They allow threads to *wait* on a condvar until it is *signaled* by another thread. A condvar should be protected by a mutex, *i.e.*, threads signaling or waiting on a particular condvar must own the associated mutex, and the waiting function takes this mutex as an argument to atomically unlock it before waiting, and reacquire it after receiving a signal.

The pthread library offers condvars, and it turns out that pthread condvars can be used directly with jthread mutexes, with the `pcondvar/jmutex` waiting function defined as a simple wrapper around the original `pcondvar/pmutex` waiting function.

Indeed the new waiting function, given a pthread condvar `cv` and a jthread mutex `m`, only needs to free `m` and wait on `cv` with a pthread mutex (these two operation performed atomically), and then reacquire `m`.

The needed atomicity as well as the unlocking and reacquisition of `m` require the protection of `big-lock`. Therefore `big-lock` is also used whenever our waiting function calls the pthread waiting function.

If we neglect implementation details the code is as follows. It uses the lower-level `lock-n` function that will be described in Section 4.

```
(define (j-condition-variable-wait! cv::pcondvar m::jmutex)
   (p-synchronize big-lock
      (unlock m)
      (p-condition-variable-wait! cv big-lock)
      (lock-n m) ))
```

Note that while our library allows the use of condvars, it doesn't prevent condvar-based deadlocks: any thread can start waiting on a condvar that will never be signaled.

3. Starvation and asymptotic deadlock

Starvation is a non-consensual notion, especially for a language such as Hop which features purposely non-terminating programs and dynamic creation of an arbitrary number of threads and mutexes.

To explain what it means for our library to be starvation-free we must first define the concept of asymptotic deadlock.

3.1 Dependency between threads

Hereafter we use the following formalism: in a program using mutexes, at a given instant in the execution, we define a partial order over threads as follows: $t_1 \prec t_2$ if and only if $\exists\, m.\ t_1$ owns $m \ \wedge\ t_2$ is waiting to lock m.

Let $\prec^+$ and $\prec^*$ be respectively the transitive closure and the symmetric transitive closure of $\prec$. We say that t_2 *depends on* t_1 whenever $t_1 \prec^+ t_2$. We say that t_1 is an *ascendancy* of t_2 and that t_2 is a *descendancy* of t_1.

We say that a program is in a *deadlock* situation if $\exists\, t.\ t \prec^+ t$, *i.e.*, if relation $\prec^*$ is not antisymmetric. Our library, by using the mechanisms described in Section 2, guarantees the absence of deadlock according to this definition.

3.2 Asymptotic deadlock

During a non-terminating execution of a program, a thread t is in an *asymptotic deadlock* situation if the number of t's ascendancies tends toward $+\infty$.

3.2.1 Example

For example, here is a program that yields an asymptotic deadlock:

```
(define (m0 (make-mutex)))
(define (P m)
   (thread
      (synchronize m
         (let ((n (make-mutex)))
            (P n)
            ...  ; takes some time
            (synchronize n
               #f )))))
(P m0)
```

This program creates an infinite number of threads and equally many mutexes. Each thread creates a mutex `n`, starts a thread, and gives `n` to it. Each thread t locks the mutex it got from its parent for the whole duration of its execution, and t tries to lock the mutex that it created itself.

Let us assume that each thread takes such time between starting its child thread and executing its `synchronize n` that the `synchronize` is executed only after the child thread could lock `n`. The parent can thus never lock it. Therefore no mutex is ever unlocked and/because no thread ever terminates.

This situation is not a deadlock as there is no dependency cycle. This program is *authorized* in our library, whether the mutexes are in different regions (no cycle between regions) or in the same one (Rule 3).

This is not a starvation situation in the classical sense either as each thread is waiting for a mutex that is owned by another thread, while starvation usually refers to a situation in which a resource is free (or periodically freed) but never given to a particular thread that wants it.

Therefore we categorize this situation under a separate concept which we call *asymptotic deadlock*.

3.2.2 Detection

It seems to us that one rudimentary means to help the programmer solving asymptotic deadlocks would be to instrument the locking functions used so as to maintain the number of ascendancies of each waiting thread, and to output a warning whenever this number exceeds a given threshold. This is left for future work.

3.3 Starvation-freedom

In this paper we use the following "negative" definition of starvation: We say that a program is *starvation-free* if and only if during any execution, **if** each thread making progress eventually releases all the mutexes it owns **and if** there are no asymptotic deadlocks **then** each thread eventually makes progress.

The multiple locking algorithm described in the next section is starvation-free, *i.e.*, any program using it is starvation-free.

4. Multiple locking

In this section we describe the design of mutexes and of an operation locking n mutexes simultaneously while guaranteeing starvation-freedom.

In this section our mutexes support explicit locking, *i.e.*, separate (un)locking functions that we call `lock-n` and `unlock` (we only need to unlock one mutex at a time). These are the basic blocks used to implement the higher-level structured locking introduced in the previous section. We then implement prelocks. Prelocks are first locked and immediately released. This pattern is handled efficiently by our implementation.

Our algorithms rely on an underlying mutual exclusion mechanism which in our implementation is the pthread library.

Reminder: the semantics presented in Appendix A summarizes what we is explained here.

4.1 Problem

The semantics of [2] avoids deadlocks but not starvation as it imposes no constraints on the `lock-n` operation. A valid implementation could rely on a simple waiting mechanism such as: if all n mutexes are free, then acquire them, else wait for a time when they all be free, which may well never happen if other threads repeatedly lock the n mutexes separately and in a non coordinated way. The thread that wants to lock all n mutexes is starving.

Our solution relies on a *fair* scheduling of threads that are waiting to lock mutexes. This scheduling is presented in this section.

Note: We are talking about high-level scheduling with no preemption, unrelated to the low-level preemptive scheduling of threads by the operating system.

4.2 Basics of our solution

At any given instant in the execution of a program, we called $\mathscr{A}$ the set of *active* threads, *i.e.*, the threads that own mutexes and/or are waiting to lock some. The main idea of our solution is to define on $\mathscr{A}$ a total order $<$ with the following invariant:

$$\text{(I)} \qquad \forall\, t_1, t_2.\ t_1 \prec^* t_2 \Rightarrow t_1 < t_2$$

This is equivalent to considering that active threads are in a global (waiting) queue. We will use the two points of view indifferently and we shall note the queue $Q = (\mathscr{A}, <)$. Note that the global queue is used in the formal presentation of the algorithms but the actual implementation never constructs it explicitly.

The order relationship is defined algorithmically: at the beginning of the execution Q is empty; during the execution, the order is modified each time Q gains or loses a thread or the situation evolves relevantly:

- when a thread t becomes active, *i.e.*, when it does not own any mutex and starts waiting to lock one, t is inserted at the end of Q, *i.e.*, set to be greater than all previously active threads;
- when a thread t becomes inactive, *i.e.*, when it unlocks its last mutex, it is removed from Q.

The order relationship is used as follows: a thread waiting to lock one or more mutexes may do so if and only if no smaller thread is owning or waiting to lock any of those mutexes.

Notation We shall write the concatenation of two queues Q_1 and Q_2 as follows: $Q_1 \cdot Q_2$

We shall define *subqueues* using the following list comprehension: for $Q = (\mathscr{A}, <)$ and a given predicate p on threads:

$[t \text{ in } Q \mid p(t)] \triangleq (\{t \in \mathscr{A} \mid p(t)\}, <)$

4.3 Nested locks

A lock is *nested* if it is executed by a thread that already owns some mutexes. This happens during the execution of nested jthread `synchronize` blocks, but we do not take their lexical scope into account in this section.

We have to define how Q evolves when an already active thread t_X starts waiting for new mutexes m_i. We call *strategies* the possible evolutions of Q. To preserve invariant (I), a strategy must take into account the new $\prec$ relations that appear between t_X and the owners of mutexes m_i if they exist.

One natural strategy could be to systematically move t_X to the end of Q as if it was a new active thread. However, some threads might be waiting for some mutexes owned by t_X; these threads should then also be moved to be kept behind t_X in Q. Q would evolve as follows:

$Q \to [t \text{ in } Q \mid t_X \not\prec^* t] \cdot [t \text{ in } Q \mid t_X \prec^* t]$

4.3.1 Possibility of starvation

In practice we have never observed starvation with this simple strategy, although it can happen as with the following example:

```
(define R (make-region))
(define m1 (make-mutex R))
(define m2 (make-mutex R))
(define m3 (make-mutex R))

(thread  ; tx
   ...
   (synchronize m1 :prelock (list m2 m3)
      #f ))

(while #t
   (thread  ; ta(i)
      (synchronize m1 :prelock m2
         ...
         (synchronize m2
            #f )))
   (thread  ; tb(i)
      (synchronize m3 :prelock m2
         ...
         (synchronize m2
            #f )))
   )
```

This program creates 3 mutexes, a thread t_x that tries to lock all 3 mutexes at the same time, and infinitely many pairs of threads $(t_{a,i}, t_{b,i})$ locking respectively $\{\texttt{m1}, \texttt{m2}\}$ and $\{\texttt{m3}, \texttt{m2}\}$.

During the execution, t_x may be blocked indefinitly. Here is why. We assume t_x is not the first one to execute its `synchronize` and several $t_{a,i}$ and $t_{b,i}$ are already in Q.

Whenever a $t_{a,n}$ executes its `synchronize m2` it is moved to the end of Q. At this point $t_{a,n}$ owns only `m1`; therefore its descendancies include t_x and the other $t_{a,i}$ but no $t_{b,i}$. t_x goes behind all $t_{b,i}$ but keeps its relative position to all $t_{a,i}$. Conversely whenever a $t_{b,n}$ executes its `synchronize m2` t_x is moved behind all $t_{a,i}$ but keeps its relative position to all $t_{b,i}$.

Therefore, as long as $t_{a,i}$ and $t_{b,i}$ keep being run, t_x is perpetually moving backwards in Q and never enters its critical section.

4.4 Starvation-free nested locks

One way to prevent starvation is to always favor the thread which has been waiting for the longest time over others. We call this thread the doyen and we denote it t_d.

The doyen may depend on other threads; therefore, we stipulate that this set of threads (t_d included) must always be before the others in Q. We now have 2 invariants:

$$\text{(I)} \qquad \forall\, t_1, t_2.\, t_1 \prec^+ t_2 \;\Rightarrow\; t_1 < t_2$$

$$\text{(J)} \qquad \forall\, t_1, t_2.\, t_1 \prec^* t_d \;\wedge\; t_2 \not\prec^* t_d \;\Rightarrow\; t_1 < t_2$$

At any time threads may enter or leave the set of t_d's ascendancies and thus go from one side of t_d in Q to the other.

A proof that this mechanism is starvation-free is given in Appendix B.

4.4.1 Strategy

Invariants (I) and (J) are sufficient to make our library starvation-free regardless of the strategy used. In our implementation, here is what we chose to do when a thread t_X already in Q starts waiting for new mutexes:

- If $t_d < t_X$ then t_X and its descendencies are moved to the back of Q, *i.e.*,
  $Q \to [t \text{ in } Q \mid t_X \not\prec^* t] \cdot [t \text{ in } Q \mid t_X \prec^* t]$
- If $t_X < t_d$ then t_X keeps its position in Q and among its new ascendancies, those that must be moved because they are behind t_X are moved to the front of Q along with their own ascendancies, *i.e.*,
  $Q \to \text{let } S = \{t \in Q \mid t \prec^+ t_X \wedge t_X <_Q t\} \text{ in}$
  $[t \text{ in } Q \mid \exists\, t' \in S.\, t \prec^* t'] \cdot [t \text{ in } Q \mid \nexists\, t' \in S.\, t \prec^* t']$

We only experimented with one strategy as differences between strategies mainly show in the case of complex dependencies between threads, and we had a hard time finding programs with complex nesting of critical sections (see Section 5). We chose the present strategy for its relative simplicity given the data structures we chose for our implementation (see below): mutexes are always moved to one end of Q; our choice of which mutexes to move seems to us like a relatively natural way to satisfy (I).

4.5 Implementation

In this section we sketch the implementation of the jthread library.

4.5.1 Data structures

The global order on threads Q is represented by the following data structures:

- Each mutex `m` has a doubly linked queue (list) containing `m`'s owner and the possible threads waiting for `m` (if they exist) with respect to the global order. These local queues allow to browse thread dependencies without having to traverse a global, arbitrarily long queue.
- Each thread has an integer *rank* (that can be read by the other threads) such that the order on ranks matches the order on threads. Ranks are used for instance to efficiently insert an already active thread into the queue of a mutex.

We also maintain a global queue `Qw` of waiting threads. Threads are put in and out of `Qw` respectively when the start waiting for mutexes and when they enter their critical section. This queue is never reorganized. Its sole purpose is to determine t_d, which is the first element of `Qw` unless `Qw` is empty.

4.5.2 Basic optimization of prelocks

Our locking operation is used to acquire mutex and also to prelock them, *i.e.*. In the case of prelocking, the mutexes are released immediately after having been locked. This is optimized by using *lightweight* locking.

Our lock function actually takes as arguments a list of mutexes to really lock and a list of mutexes to prelock. Our function doesn't actually lock and unlock the latter; it starts waiting as for locking but at the last moment it simply does nothing.

Furthermore, in order for a thread t to enter its critical section, t doesn't need to be the first in the queue of each mutex to prelock, but only to be the first in the queue of the mutex to lock and that all the mutexes to prelock are free. In our implementation, threads take such opportunities to make progress whenever they arise.

However, our current implementation does not prioritize threads differently according to whether they must lock or prelock the mutexes they are acquiring, to take into account the lower resource occupation of prelocking. Such an optimization is left for future work.

5. Performance

In this section we present measures of the performance of our library compared to pthread in several micro-benchmarks and two "real" programs. The results are as follows:

- as for the raw locking/unlocking performance with and without contention, jthread mutexes are much slower than pthread mutexes;
- in another micro-benchmark, the program using jthread is slightly faster because deadlock avoidance allows to use a more efficient algorithm that leads to deadlocks when implemented with pthread;
- we were able to use jthread instead of pthread in the Hop Web server and the Hop MP3 decoder with no change to the programs themselves and it had no impact on their performance.

The two libraries (pthread and jthread) benefit from several optimizations of mutex locking that we developed relatively independently of our present work [16]. One of these optimizations is *synchronization lifting* which makes structured locking almost as fast as explicit locking despite the fact that structured locking constructs have to handle exceptions.

All our benchmarks used the safe mode of our library. As we said in Section 2.3, in other measurements we did not observe a significant performance gap between the safe and unsafe modes.

All the benchmarks except the Hop Web server were run on a Linux 3.12 kernel hosted by an Intel Xeon E5-1660.

5.1 Micro-benchmarks

5.1.1 Raw locking/unlocking performance

We wrote two micro-benchmarks in which respectively one or several threads repeatedly lock one unique mutex to perform an elementary floating-point operation.

Reported times are obtained by summing cpu and system times.

Without contention The value reported here is the minimal value observed out of two consecutive runs.

	pthread	jthread
sync	1.06s	13.23s

With contention See Figure 1.

We can see that as for the raw locking/unlocking performance, our library is approximately 12 times slower than pthread without contention, and between 3 and 50 slower than pthread with contention.

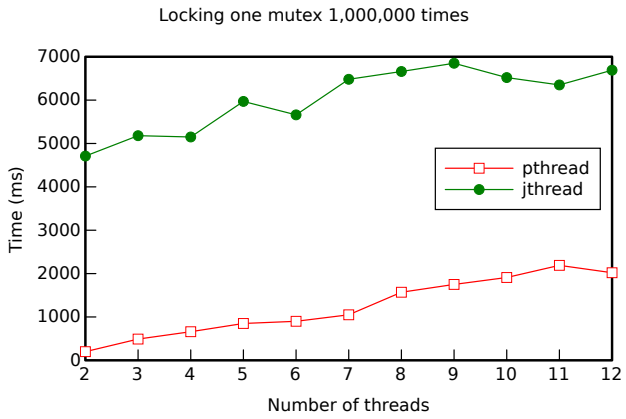

Figure 1. Raw locking performance with contention. This benchmark starts n thread, each of which to lock and unlock one same mutex $10^6/n$ times.

5.1.2 Deadlock avoidance

We describe here a small benchmark with about 100 lines of code. In this benchmark `n` threads access `p` mutexes. Each thread repeatedly executes a task. This task is to randomly choose 2 mutexes `m1` and `m2`, and then to do a "job" in 2 phases. The thread must own `m2` for the second phase and `m1` during all of the job. Here is a prototype implementation:

```
(synchronize m1
   (work)
   (synchronize m2
      (work)
      ))
```

In our implementation to "work" is to do some number of arithmetic operations.

With the pthread library the above program compiles but leads to deadlocks, for instance when one thread choses mutexes `mi` and `mj` and another thread choses `mj` and `mi` (the same ones, swapped).

To make the program deadlock-free we shall assume an ordering on mutexes and never nest `synchronize`s in the reverse order:

```
(if (< m1 m2)
    ; then
    (synchronize m1
       (work)
       (synchronize m2
          (work)
          ))
    ; else
    (synchronize m2
       (synchronize m1
          (work)  ; requires m1
          (work)  ; requires m1 and m2
          )))
```

With the jthread library the first program creates region cycles at runtime and thus raises errors. To prevent errors, one must assign all `p` mutexes to the same region and then use the appropriate `prelock`s:

```
(synchronize m1 :prelock m2
   (work)
   (synchronize m2
      (work)
      ))
```

Results We ran the program with 4 threads and 4 mutex in order to observe a high rate of contention. The value reported here is the minimal real running time observed out of three consecutive runs.

	pthread	jthread
avoid	9.35s	8.34s

This micro-benchmark is an example of a case in which our library allows for more parallelism, by locking `m2` later in the cases where `m2` $<$ `m1`. In this example the gain in parallelism compensates for the intrinsic cost of our functions, making the program using our library (slightly) faster.

5.2 Benchmarking Realistic Applications

The first two micro-benchmarks presented in Section 5.1 measure the intrinsic speeds of acquiring and releasing locks of the `jthread` library and compare them to those of standard posix-like libraries. This is an important comparison elements which shows that deadlock avoidance and deadlock detection, as currently implemented in the `jthread` library, come with a price.

To go one step beyond, we conducted an experiment for estimating the impact of the `jthread` library on realistic applications: we used two existing multi-threaded Hop applications that we linked against the `jthread` library and compared the performances of those of the initial applications.

Both applications were already working and apparently deadlock-free, which was confirmed when we used our library. In both cases we applied no modification to the existing code. Thus, by default, each mutex was in its own region. In both cases, the application was executed in safe mode and completed without any region cycle error (Rule 1), *i.e.*, no prelock was required. As there was only one mutex in each region, Rule 2 could not have been violated.

These two experiments contrasts with the result of the micro-benchmarks as they reveal a minor impact of the `jthread` library. These two experiments are described in the two following sections.

5.2.1 Hop

The Hop runtime environment consists of a full-fledged Web server equipped with dynamic compilers that compile Hop client side programs dynamically into HTML and JavaScript. The Web server is optimized for serving dynamic responses as fast as possible. Hop is bootstrapped. Its source code counts 75 KLOC. We used this application, *i.e.*, Hop itself, to measure the impact of the `jthread` library on a typical server-like application.

In order to handle several requests in parallel, Hop is multi-threaded. The threads are executed in a shared memory and they are preempted by the operating system. Hop relies on mutexes to implement the critical sections such as the ones needed by the various caches that Hop uses for avoiding recompiling client-side codes.

For comparing the performances of the two Hop versions we followed the traditional methodology for measuring the performances of Web servers. It consists in emitting a variable amount of requests to a server and measuring the number of responses the server can sustain. The workloads are generated by httperf [5], a dedicated tool. We executed the tests on a Linux 3.9 kernel hosted by an Intel Xeon W3570.

We compared the performances of the two Hop servers on two kinds of requests: static requests and dynamic requests. The results are presented in Figures 2 and 3[2].

[2] This benchmark was performed using a previous version of our multiple locking algorithm from which we expect only minor differences in performance on this benchmark.

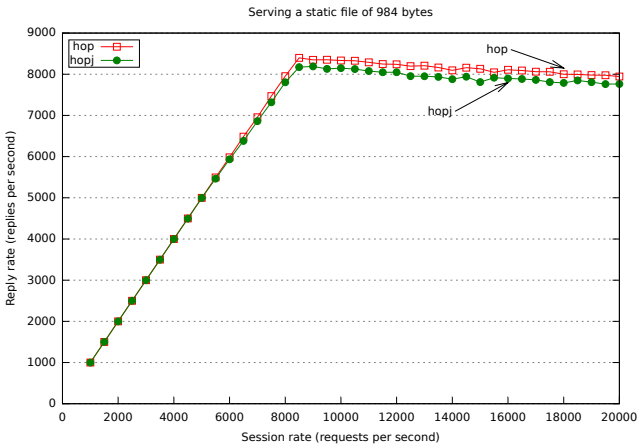

Figure 2. the throughput a web server can sustain when delivering 984 bytes long static files. Each session consists in 5 consecutive requests that are sent using a single persistent connection.

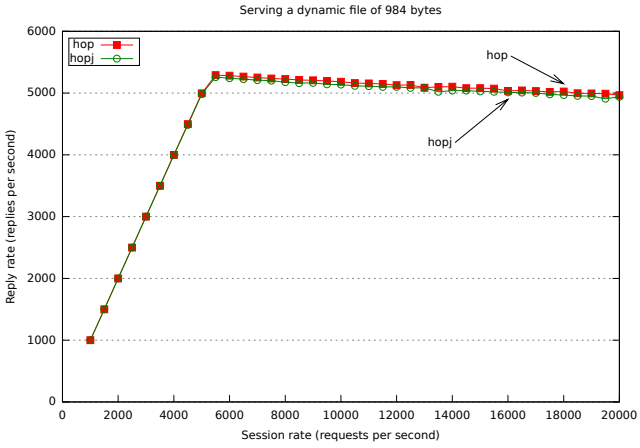

Figure 3. the throughput a web server can sustain when delivering dynamic contents. Each session consists in 5 consecutive requests that are sent using a single persistent connection.

The experiment shows no significant impact on the overall Hop performance due to the `jthread` library. The two versions of the server behave almost identically.

5.2.2 Performance Evaluation of a Multimedia Application

Our second experiment estimates the impact of the `jthread` library on the performance of a MP3 music player implemented in Hop. This benchmark is a Hop application of about 1,300 lines of code. It has been extracted from a larger multimedia application which has been conceived for low-end ARM machines. Since it is critical for the MP3 player to be fast enough, the source code has been highly optimized for speed. In particular, the implementation of the parallelism has been carefully crafted and lock acquisition minimized as much as possible. In this experiment, the time spend in the actual hardware audio system being irrelevant to our experiment, we modified the application to discard the decoded bytes instead of playing them.

The MP3 player implements a traditional producer/consumer algorithm. The producer reads the bytes of a MP3 music file from the local file system or from the network. The consumer decodes them into raw music bytes. The two threads are synchronized using mutexes and condition variables.

We ran the audio player on a large MP3 file of 60 minutes of music. This whole execution locks approximately 18,700 mutexes.

The time figures reported here are the minimum cpu+sys time of 3 consecutive executions.

	mp3 decoder
hop	3.72s
hopj	3.94s

The time figures shows that on this realistic concurrent application, the `jthread` library does not impact on the overall performance. This contrasts the results obtained on micro-benchmarks and confirms the results already observed with the Hop benchmark.

5.3 Complex benchmarks

We have not found yet a program commonly used for benchmarking mutexes and with complex nesting of critical sections, which would allow to meaningfully compare different strategies. In particular, among SPECjvm2008 [17], the Java Grande Forum Multithreaded Benchmarks [18], and the performance tests of the JSR 166 project [13] several programs use structured locking but no nested locks. In the Debian Shootout [8] two programs use mutexes: in `chameneos-redux` there are no nested locks, while the use of mutexes in `thread-ring` does not come down to structured locking.

We see two possible interpretations of this rarity – at least among benchmarks – of programs using mutexes in a complex way: either such programs are too difficult to develop and maintain, or there is little need for them. These two interpretations lead to two opposite assessments of the usefulness of the present work.

The fact that mutexes are one of the most widely used synchronization mechanism in shared memory concurrency reassures us in our project to make safe use of mutexes easier.

6. Conclusion and future work

We designed and implemented a mutex library that guarantees the absence of deadlocks, partly by raising outright errors which are easier to debug than actual deadlocks, as in dynamic typing, and partly by actively avoiding deadlocks, thus allowing to express efficient algorithms that would otherwise be prone to deadlocks. Our functions require more information from the programmer in some cases, but our library is easy to program with and mixes well with modularity, and we were able to use it in two existing applications with no modification. We showed that while our library has bad raw locking performance compared to pthread, using jthread instead of pthread had little impact on the performance of those two real-size applications.

The starvation-free algorithm for multiple locking that we provided could be incrementally improved upon, or replaced altogether as the high-level deadlock-avoiding part of our library should integrate well with other multiple locking algorithms which could, for instance, trade the guarantee of starvation-freedom for better raw locking performance.

Given that our deadlock avoidance mechanism relies on the ability to acquire several mutexes simultaneously, we could offer this feature to the programmer with a construct such as

```
(synchronize* l :prelock p)
```

with `l` and `p` being the lists of mutexes to respectively lock and prelock. We experimented with this but did not yet settle on a semantics for this construct.

Finally, our work could be ported to another language based on structured programming and structured locking and, as in [2], it could also be coupled with a static analysis that would infer region and/or prelock annotations.

As for asymptotic deadlocks, which we defined in this paper, it would be interesting to know if existing or novel locking mechanisms would allow to prevent them.

References

[1] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[2] G. Boudol. A Deadlock-Free Semantics for Shared Memory Concurrency. In *Theoretical Aspects of Computing - ICTAC 2009*, volume 5684 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2009.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. *ACM SIGPLAN Notices*, 37(11):211–230, 2002.

[4] E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[5] M. D. and T. Jin. httperf: A Tool for Measuring Web Server Performance. In *First Workshop on Internet Server Performance*, pages 59–67. ACM, 1998.

[6] E. W. Dijkstra. The Mathematics Behind the Banker's Algorithm. 1977.

[7] C. Flanagan and M. Abadi. Types for Safe Locking. In *Programming Languages and Systems*, pages 91–108. Springer, 1999.

[8] B. Fulgham. The Computer Language Benchmarks Game aka Debian Shootout: http://benchmarksgame.alioth.debian.org/.

[9] P. Gerakios, N. Papaspyrou, and K. Sagonas. A Type and Effect System for Deadlock Avoidance in Low-Level Languages. In *Proceedings of the 7th ACM SIGPLAN workshop on Types in language design and implementation*, pages 15–28. ACM, 2011.

[10] N. Kobayashi. A New Type System for Deadlock-Free Processes. In *CONCUR 2006–Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2006.

[11] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8):453–455, Aug. 1974.

[12] L. Lamport. The Synchronization of Independent Processes. *Acta Informatica*, 7:15–34, 1976.

[13] D. Lea. JSR-166: http://gee.cs.oswego.edu/dl/concurrency-interest/.

[14] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization Inference for Atomic Sections. In *ACM SIGPLAN Notices*, volume 41, pages 346–358. ACM, 2006.

[15] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a Language for Programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, Portland, Oregon, USA, Oct. 2006.

[16] M. Serrano and J. Grande. Locking Fast. In *Proceedings of the ACM Symposium on Applied Computing (SAC'14)*, Gyeongju, Korea, Mar. 2014.

[17] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 Performance Characterization. In *Computer Performance Evaluation and Benchmarking*, pages 17–35. Springer, 2009.

[18] L. A. Smith and J. M. Bull. A Multithreaded Java Grande Benchmark Suite. In *Proceedings of the third workshop on Java for high performance computing*, 2001.

[19] K. Suenaga. Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References. In *Programming Languages and Systems*, pages 155–170. Springer, 2008.

A. Formal semantics

In this appendix we present a formal semantics that combines the mechanisms that we described in sections 2 and 4. The semantics is parametrized by the strategy used in the multiple locking algorithm. The strategy appears in the semantic rules as a function $\mathscr{S}$ which takes as arguments the global queue Q and most of the state of the system and returns a new queue Q'. We described in Section 4.4.1 the strategy that we chose for our implementation.

We give the formal semantics of a minimalistic language only featuring the constructs of our library. For the sake of clarity we exclude condition variables from this semantics. For each aspect of the semantics the reader is referred to the explanations given in previous sections.

A.1 Language

In our language, the only values are mutexes, sets of mutexes, and *unit*, and the only operations are structured locking and the launching of a thread. This language only makes sense as part of a larger language – such as Hop – featuring general-purpose constructs such as functions.

In the present language, "mutexes" are simply integers distinct from one another and attributed in increasing order: $m_1 < m_2$ if and only if m_1 was instantiated before m_2.

r			$regions$
x			$variables$
v	::=	$() \mid x \mid \{v*\}$	$values$
e	::=	v	$expressions$
		$\mid \{e*\}$	
		$\mid (\mathsf{thread}\ e)$	
		$\mid (\mathsf{let}\ (x\ (\mathsf{mutex}\ r))\ e)$	
		$\mid (\mathsf{sync}\ e\ e\ e)$	
		$\mid (\mathsf{wait}\ v\ v\ e)$	$(runtime\ expression)$
		$\mid (\mathsf{insync}\ e)$	$(runtime\ expression)$

Informal typing We only consider expressions in which:

- sets $\{e*\}$ only contain expressions that reduce to mutexes;
- in $(\mathsf{sync}\ e_0\ e_1\ e_2)$, e_0 always reduces to a mutex and e_1 always reduces to a set.

The programmer can only create sets. However, our semantics internally uses the usual mathematical operations on those sets.

A.2 Redexes and evaluation contexts

ρ	::=	$(\mathsf{thread}\ e)$	$redexes$
		$\mid (\mathsf{let}\ (x\ (\mathsf{mutex}\ r))\ e)$	
		$\mid (\mathsf{sync}\ v\ v\ e)$	
		$\mid (\mathsf{wait}\ v\ v\ e) \mid (\mathsf{insync}\ v)$	
$\mathbf{E}$	::=	$[] \mid \mathbf{E}[\mathbf{F}]$	$evaluation\ contexts$
$\mathbf{F}$	::=	$[] \mid \{..., [], ...\}$	$frames$
		$\mid (\mathsf{sync}\ []\ e\ e) \mid (\mathsf{sync}\ e\ []\ e)$	
		$\mid (\mathsf{insync}\ [])$	

A.3 Configurations

Threads are of the form $\mathbf{E}[\rho]_{id}$ where

- id is an identifier;
- $\mathbf{E}$ is an evaluation context;
- ρ is a redex.

The only thing we expect from thread identifiers is that they be distinct from each other.

Association sets of stacks

Each thread has some values associated to it, some of which change (only) when the thread enters a critical section and are restored to their previous value when the thread exits that critical section.

$$\frac{j \text{ fresh}}{R^T \mid\mid A \mid\mid N \mid\mid O \mid\mid W \mid\mid \mathbf{E}[(\mathsf{thread}\ e)]_i \ \rightarrow\ \top_j * R^T \mid\mid \emptyset_j * A \mid\mid 0_j * N \mid\mid \emptyset_j * O \mid\mid \emptyset_j * W \mid\mid \mathbf{E}[()]_i \mid\mid e_j} \ (\textsc{Thread})$$

$$\frac{}{R^M \mid\mid R^R \mid\mid n \mid\mid \mathbf{E}[(\mathsf{let}\ (x\ (\mathsf{mutex}\ r))\ e)]_i \ \rightarrow\ r_{n+1} * R^M \mid\mid (r, \top) * R^R \mid\mid n+1 \mid\mid \mathbf{E}[(\{x \mapsto n+1\}e)]_i} \ (\textsc{Let Mutex})$$

$$\frac{\begin{array}{c} L = \{l\} \cup P \qquad L \cap o_i = \emptyset \qquad \forall\, m \in L.\ R^M(m) = r \\ \forall\, m \in L.\ m \in A(i) \vee m \geq N(i) \qquad Q' = \mathscr{S}(Q, Q^W, o_i : O, W, i, l, P) \end{array}}{\begin{array}{rl} & Q \mid\mid \quad Q^W \mid\mid \quad (r : rs)_i * R^T \mid\mid R^M \mid\mid A \mid\mid o_i : O \mid\mid \quad W \mid\mid \mathbf{E}[(\mathsf{sync}\ l\ P\ e)]_i \\ \rightarrow & Q' \mid\mid Q^W \cdot i \mid\mid (r : r : rs)_i * R^T \mid\mid R^M \mid\mid A \mid\mid o_i : O \mid\mid L_i * W \mid\mid \mathbf{E}[(\mathsf{wait}\ l\ P\ e)]_i \end{array}} \ (\begin{array}{c}\textsc{Wait}\\ \textsc{Rules 2\&3}\end{array})$$

$$\frac{\begin{array}{c} L = \{l\} \cup P \qquad L \cap o_i = \emptyset \qquad \forall\, m \in L.\ R^M(m) = r' \\ r \not<^* r' \qquad Q' = \mathscr{S}(Q, Q^W, o_i : O, W, i, L, P) \end{array}}{\begin{array}{rl} & Q \mid\mid \quad Q^W \mid\mid \quad (r : rs)_i * R^T \mid\mid R^M \mid\mid \quad R^R \mid\mid o_i : O \mid\mid \quad W \mid\mid \mathbf{E}[(\mathsf{sync}\ l\ P\ e)]_i \\ \rightarrow & Q' \mid\mid Q^W \cdot i \mid\mid (r' : r : rs)_i * R^T \mid\mid R^M \mid\mid (r', r) * R^R \mid\mid o_i : O \mid\mid L_i * W \mid\mid \mathbf{E}[(\mathsf{wait}\ l\ P\ e)]_i \end{array}} \ (\begin{array}{c}\textsc{Wait}\\ \textsc{Rule 1}\end{array})$$

$$\frac{(\{l\} \cup P) \cap O = \emptyset \qquad \forall\, j.\ i \leq_Q j \vee l \notin W(j) \qquad Q_l^W \neq [] \vee Q_r^W = []}{\begin{array}{rl} & Q \mid\mid Q_l^W \cdot i \cdot Q_r^W \mid\mid \quad a_i * A \mid\mid \quad ns_i * N \mid\mid \quad o_i * O \mid\mid w_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{wait}\ l\ P\ e)]_i \\ \rightarrow & Q \mid\mid \quad Q_l^W \cdot Q_r^W \mid\mid (P : a)_i * A \mid\mid (n : ns)_i * N \mid\mid (l : o)_i * O \mid\mid \emptyset_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{insync}\ e)]_i \end{array}} \ (\begin{array}{c}\textsc{Enter}\\ \textsc{No New Doyen}\end{array})$$

$$\frac{(\{l\} \cup P) \cap O = \emptyset \qquad \forall\, j.\ i \leq_Q j \vee l \notin W(j) \qquad Q' = [j \text{ in } Q \mid j \prec^* d] \cdot [j \text{ in } Q \mid j \not\prec^* d]}{\begin{array}{rl} & Q \mid\mid i \cdot d \cdot Q^W \mid\mid \quad a_i * A \mid\mid \quad ns_i * N \mid\mid \quad o_i * O \mid\mid w_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{wait}\ l\ P\ e)]_i \\ \rightarrow & Q' \mid\mid \quad d \cdot Q^W \mid\mid (P : a)_i * A \mid\mid (n : ns)_i * N \mid\mid (l : o)_i * O \mid\mid \emptyset_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{insync}\ e)]_i \end{array}} \ (\begin{array}{c}\textsc{Enter}\\ \textsc{New Doyen}\end{array})$$

$$\frac{o \neq \emptyset}{\begin{array}{rl} & (r_0 : r)_i * R^T \mid\mid (a_0 : a)_i * A \mid\mid (n_0 : ns)_i * N \mid\mid (o_0 : o)_i * O \mid\mid \mathbf{E}[(\mathsf{insync}\ v)]_i \\ \rightarrow & \quad r_i * R^T \mid\mid \quad a_i * A \mid\mid \quad ns_i * N \mid\mid \quad o_i * O \mid\mid \mathbf{E}[v]_i \end{array}} \ (\begin{array}{c}\textsc{Unlock}\\ \textsc{Nested}\end{array})$$

$$\frac{}{\begin{array}{rl} & Q_l \cdot i \cdot Q_r \mid\mid (r_0 : \top)_i * R^T \mid\mid (a_0 : \emptyset)_i * A \mid\mid (n_0 : 0)_i * N \mid\mid (o_0 : \emptyset)_i * O \mid\mid \mathbf{E}[(\mathsf{insync}\ v)]_i \\ \rightarrow & \quad Q_l \cdot Q_r \mid\mid \quad \top_i * R^T \mid\mid \quad \emptyset_i * A \mid\mid \quad 0_i * N \mid\mid \quad \emptyset_i * O \mid\mid \mathbf{E}[v]_i \end{array}} \ (\begin{array}{c}\textsc{Unlock}\\ \textsc{Top-Level}\end{array})$$

Figure 4. Semantic rules

In configurations, these values are represented as sets of pairs associating to each thread identifier a stack of values, with the top value of the stack being the current value for the thread.

Components

Each configuration is formed of 11 components:

- Q is the queue (the ordered set) of active thread ids (see Section 4);
- Q^W is the queue of waiting threads ids; if $Q^W \neq \emptyset$ its first element is the doyen (4.4);
- R^T is a stack set associating to each thread its current region (2.1);
- R^M associates to each mutex its (immutable) region (2.1);
- R^R is a set of region pairs (r_1, r_2) such that $r_1 < r_2$; R^R is such that relation $<$ is the transitive closure of this set (2.1);
- A is a stack set associating to each thread the set of mutexes that the thread is explicitly authorized to lock, *i.e.*, the prelock set of the thread's current critical section or $\emptyset$ if the thread is not in a critical section (2.2);
- N is a stack set associating to each thread the last attributed mutex identifier (> 0) when the thread entered its current critical section, or 0 if the thread is not in a critical section (2.2.1);
- O is a set associating to each thread a stack that contains all the mutexes currently owned by the thread and reflects the nesting of the critical sections the thread is currently in;
- W is a stack set associating to each thread id the set of mutexes that the thread is currently waiting to lock;
- n is the (integer) value of the last mutex to have been instantiated (2.2.1);
- T is the (unordered) set of threads.

Chemical style

To simplify a little the semantic rules, and to represent the concurrent execution of the various components, we shall use the following syntax for configurations:

$\Gamma ::= Q \mid Q^W \mid R^T \mid R^M \mid R^R \mid A \mid N \mid O \mid W \mid n \mid t \mid (\Gamma \mid\mid \Gamma)$
where t is a single thread ($t \in T$).

We assume that parallel composition is commutative and associative, so that the rules can be expressed following the "chemical style" of [1], specifying local "reactions" of the form $\Gamma \rightarrow \Gamma'$ that can take place anywhere in the configuration.

Syntactic sugar and other notation

- $x : l$ means that x and l are respectively the head and tail of a list or pile
- $q_1 \cdot q_2$ is the concatenation of queues q_1 and q_2
- $x * S \triangleq \{x\} \cup S$
- $x_i \triangleq (i, x)$
- If $(i, (x : l)) \in S$ then $S(i) \triangleq x$

A.4 Rules

The rules of the semantics are presented in Figure 4.

Configurations that would raise errors in our implementation, namely, unauthorized acquisitions, are irreducible.

B. Proof of starvation-freedom

B.1 Indefinite progression

During an execution of a program, we consider that the possible evolutions of a thread t fall into 3 cases:

- t terminates, *i.e.*, eventually terminates;
- t ends up blocked forever (never making progress);
- t is indefinitely making progress, *i.e.*, either ends up never being blocked anymore, or enters infinitely many critical sections.

Let t be a thread. We say that "if t makes progress it *eventually* does X" if t cannot indefinitely make progress *or terminate* and not do X.

B.2 Property

We now prove that the algorithm presented in Section 4 is starvation-free (see Section 3.3), *i.e.*, that for any execution of any program, **if** each thread making progress eventually releases the mutexes it owns **and if** the number of ascendancies of no waiting thread tends towards infinity **then** each thread eventually makes progress.

For this proof we assume structured locking with no condition variables. This proof is based on the fact that all queuing, effective taking and unlocking operations are done sequentially under a global lock.

B.3 Proof

Let us assume that each thread making progress eventually releases all the mutexes it owns.

B.3.1 Lemma

If a given doyen never makes progress, then the number of its ascendancies tends towards infinity.

B.3.2 Proof of the lemma

Let us assume that t_d never makes progress. Then we algorithmically construct an infinite family (S_k) of non-empty and disjoint sets of ascendancies of t_d.

At any instant, let n be the number of S_k already computed. $S_0..S_{n-1}$ are fixed and they only contain threads that are blocked forever, while S_n is currently being computed.

At any instant, for any $p \leq n$, let $S^p_\infty = \{t \mid t \prec^* t_d\} \setminus \bigcup_{0..p} S_k$. Note that $\forall\, p, q.\ p < q \Rightarrow S^q_\infty \subset S^p_\infty$.

At any instant, we will show that for all $p \leq n$ the following invariant holds:

$(K_p) \qquad \forall\, t \in S^p_\infty.\ \exists\, t' \in S_p.\ t \prec^* t'$

At $n = 0$ we set $S_0 = \{t_d\}$. By hypothesis, t_d will never make progress. (K_0) will clearly be true at any future instant.

At $n > 0$, assuming that $S_0..S_{n-1}$ are fixed and only contain threads that will never make progress, and assuming that $(K_0)..(K_{n-1})$ will always hold, let us assign

$S_n := \{t \mid \exists\, t' \in S_{n-1}.\ t \prec t'\}$

and let S_n evolve as follows: whenever $\exists\ t \in S_n.\ \neg(\exists\ t' \in S_{n-1}.\ t \prec t')$, then $S_n := S_n \setminus \{t\}$.

Proof of (K_n)

Let us prove that (K_n) holds when S_n is initialized and at any subsequent instant, by induction over time:

When S_n is initialized, $\forall\, t \in S^n_\infty.\ t \in S^{n-1}_\infty$ therefore since (K_{n-1}), $\exists\, t'' \in S_{n-1}.\ t \prec^* t''$, therefore, by definition of S_n, (K_n) holds.

Subsequently, (K_n) could become false in 3 cases:

1. A new $t \in S^n_\infty$ appears. Let us then show that $\exists\, t' \in S_n.\ t \prec^* t'$. The present case can arise in 2 ways:
   (a) t becomes an ascendancy of t_d. Since (J) holds, it cannot be because t acquires $\mathtt{m}$ waited by $t_1 \prec^* t_d$. Thus it must be because $t_1 \prec^* t_d$ starts waiting for $\mathtt{m}$ owned by t. $t_1 \in S^{n-1}_\infty$ because other ascendancies of t_d are blocked. Let us distinguish between 2 cases:
      i. $t_1 \in S_n$. Then $t' = t_1$. ✓
      ii. $t_1 \in S^n_\infty$. Then by induction hypothesis, *i.e.*, (K_n) applied to t_1, $\exists\, t' \in S_n.\ t \prec t_1 \prec^* t'$. ✓
   (b) t leaves S_n but $t \prec^* t_d$ still holds. Then because of (K_{n-1}), $\exists\, t'' \in S_{n-1}.\ t \prec^* t''$, but not $t \prec t''$ because t is leaving S_n, therefore $t \prec t_1 \prec^+ t''$ and we conclude as before. ✓
2. For t, t' as in (K_n), t' leaves S_n. This means that t' is unlocking mutexes, *i.e.*, is making progress, which contradicts $t \prec^* t'$. ✓
3. For t, t' as in (K_n), $t \prec^* t'$ becomes false but $t \in S_\infty$ still holds. This means that t is unlocking one or more mutexes. Because of (K_{n-1}), $\exists\, t'' \in S_{n-1}.\ t \prec^* t''$. There are 2 cases:
   (a) $t \prec t_1 \prec^+ t''$ and we conclude as before. ✓
   (b) $t \prec t''$. We distinguish between 2 sub-cases:
      i. If $t \prec t''$ was true since S_n was initialized or before, then we could have $t \in S_n$, which is not the case by hypothesis. ✓
      ii. Otherwise, when $t \prec t''$ became true, $t \in S^n_\infty$ held, thus by induction hypothesis we had $\exists\, t' \in S_n.\ t \prec^* t'$. Then this relation still holds. Indeed, since we assume structured locking, t did not yet release the mutexes it owns that are involved in this relation, since it just released the mutexes it acquired afterwards, and the other threads involved in the relation did not make progress either. ✓

Thus (K_n) holds and will hold at any future instant. □ (K_n)

Let $t_m = min_<\bigcup_{0..n-1} S_k$. Then,

$$
\begin{aligned}
S_n = \emptyset \quad &\overset{(K_n)}{\Longrightarrow} \quad S_n \cup S^n_\infty = \emptyset \\
&\Longrightarrow \quad \forall\, t \prec^* t_d.\ t \in \textstyle\bigcup_{0..n-1} S_k \\
&\overset{(J)}{\Longrightarrow} \quad \nexists\, t.\ t < t_m \\
&\overset{(I)}{\Longrightarrow} \quad t_m\ makes\ progress
\end{aligned}
$$

Yet by induction (over n) hypothesis, all threads in $\bigcup_{0..n-1} S_k$ are blocked forever; therefore S_n will never be empty.

Yet S_n can only lose threads, therefore at some point it will stop evolving forever. After this point, if at least one thread $t \in S_n$ kept making progress indefinitely, t would eventually release all the mutexes it owns and leave S_n, which is impossible. Therefore at some point all threads in S_n will be blocked forever.

We then proceed to computing S_{n+1}.

This constructs an infinite family of non-empty and disjoint sets of ascendancies of t_d, therefore the number of t_d's ascendancies tends towards infinity. □ (Lemma)

Thus by contraposition, if for no thread t the number of ascendancies of t tends towards $+\infty$, then t_d eventually makes progress.

Then if some threads never made progress, it is clear that one of them would end up being the doyen, and thus would eventually make progress. (Contradiction.)

Therefore each thread eventually makes progress. □