# Using a waiting protocol to separate concerns in the mutual exclusion problem

Frode V. Fjeld
frodef@cs.uit.no
Department of Computer Science, University of Tromsø
Technical Report 2003-46

November 21, 2003

**Abstract**

How to implement process synchronization in a general-purpose software library while incurring a minimum of policy decisions on the system as a whole? We propose that in dealing with the problem of mutual exclusion in concurrent systems, a separation of concerns between the mechanism of detecting contention and the policy decision of what to do when such contention is detected is appropriate. We show with code how a waiting protocol can guide and aid the design and implementation of both lower-level primitives and the higher-level, policy-making parts of a system. We also show how the technique can be extended so as to help preventing some deadlock situations.

## 1 Introduction

The mutual exclusion problem is that of ensuring that at most one out of any number of concurrent processes are allowed to execute some critical section of code at any point in time. This problem is well researched and understood in terms of particular solutions and limitations on the space of all solutions [3]. To our knowledge, all solutions specify that each process upon attempting to enter the critical section follow a particular protocol, which follows schematically this outline:

1. Attempt to gain the privilege of entering the critical section.
2. If unsuccessful, go back (eventually) to step 1.
3. Execute and then leave the critical section.

This text is concerned with step 2 of this procedure. In other words, we propose an answer to the question "what to do when we are denied immediate entry to a critical section?". This is not to say that this question is necessarily a difficult one. When studying the mutual exclusion problem in isolation, "go back to step 1" is a perfectly good answer. The straight-forward implementation of that answer amounts to what is known as "busy-waiting", and this overall solution to the mutual exclusion problem is known as a "spin-lock". As an alternative to spin-locking, one may chose to have the process yield control—presumably to another, better-to-do process—in order to be awoken at some later time when the privilege of entering the critical section is expected to be available. This is known as a "sleep-lock". While these two solutions in principle are equivalent, their practical implications can be quite different. And interestingly, neither solution is appropriate for every situation. To the contrary, for either solution there are realistic circumstances when that solution is highly inappropriate. So the programmer must make a choice based on the particular circumstances of the situation. But sometimes the circumstances are obscure, for example because the critical section is part of a primitive operation where sufficient semantic context to make an informed choice between spinning and sleeping is not available. Consequently, current practice is

often to employ a hybrid approach, where one first busy-waits for perhaps a few tens of iterations before yielding control just like a sleep-lock would do. But this is still problematic, because not only must we somehow decide the number of iterations to busy-wait before going to sleep, but also the implementation of performing "go to sleep" has substantial policy implications.

We proceed as follows. Section 2 discusses the general idea of separating concerns in the mutual exclusion problem. In Section 3 we show how this separation may be manifest in actual code. Section 4 presents very briefly how the mechanisms of the previous section may be used to help prevent deadlocks. Finally, Section 5 addresses some practical concerns, before a brief conclusion is made in Section 6.

## 2 Separation of concerns

Dijkstra introduced the notion that modularization and "separation of concern" is necessary in order to "deal with the difficulties" of constructing software systems by taking the difficulties on one by one [2]. We propose that the mutual exclusion problem, and similar locking problems, should be separated into two concerns of mechanism and policy [4].

**Mechanism:** Detection of potential violations of concurrency invariants. That is, to answer correctly each process that asks, "may I enter this critical section?"

**Policy:** Deciding what to do if the answer to the previous question was "no."

The motivation for this separation is that each relates to contextual circumstances of quite different scope and nature. The former depends on highly time-critical information about very specific system state (i.e. "is any other process inside this critical region right now?"), while the latter is often based on a combination of a priori assumptions or knowledge (e.g. "if this critical section is busy, it is likely to remain so for a long time") and relatively slowly fluctuating system state (e.g. "Is the system's resources overloaded?").

## 3 Manifesting separation of concerns using signals and restarts

From this section onwards we use certain words according to their definition in the Common Lisp programming language specification [1], and these are emphasized LIKE THIS. Also, the following discussion assumes that a process that wants to enter a critical section looks essentially like this:

```
(unwind-protect
    (progn (acquire mutex)
           (do-critical-section))
  (release mutex))
```

where the operator `acquire` only returns when the process has successfully acquired the mutex. UNWIND-PROTECT ensures that `release` is called when the process exits the scope of the critical section. We also assume that it is safe (i.e. a no-op) to call `release` on a mutex the process does not currently hold.

We propose the following strategy in order to manifest the separation of concerns of the mutual exclusion problem in software systems. When attempting to enter a critical section, establish a "retry" control transfer target. In the event that the process is denied immediate entry to the critical section, SIGNAL this as an exceptional situation in order to enable anyone at higher abstraction levels to HANDLE this situation according to any policy they wish [5]. We refer to this scheme as a "waiting protocol". Assuming we have an operator `may-i-enter?` that answers correctly that question[1], such a waiting protocol may be expressed in Common Lisp as shown in Listing 1.

[1] Because the mutual exclusion problem is solved many times over, this is a safe assumption.

```
(define-condition acquisition-failed ()
  ((mutex :initarg :mutex :accessor mutex)))

(defun acquire (mutex)
  (loop until (may-i-enter? mutex)
        do (with-simple-restart
               (retry "Retry acquisition of ~S." mutex)
             (error 'acquisition-failed :mutex mutex))))
```

Listing 1: Our proposed implementation of the protocol for entering a critical section.

The version of `acquire` presented in Listing 1 will by default consider any denied immediate access to a critical section to be an ERROR, so unless some higher level module handles the signal, the debugger will be entered. However, the debugger will make the "retry" RESTART option available as an interactive option in the debugger, so this is not to say that the acquisition must necessarily fail. This default behavior may be appropriate for example in a system where some overall aspect of the design implies that concurrent access to shared resources should never occur. On the other hand, if the final call to ERROR was replaced with an identical call to SIGNAL, the default behavior of `acquire` would be that of busy-waiting.

At the higher spheres of abstraction, we are now in a position to explicitly decide the waiting policy for mutexes within some dynamic scope. For example, the system specified in Listing 2 will run `do-all-work` with spin-locking exclusively and indiscriminately, and regardless of the default signaling mode of `acquire`. This extends naturally to any sleeping policy that can be

```
(handler-bind ((acquisition-failed
                (lambda (condition)
                  (declare (ignore condition))
                  (invoke-restart 'retry))))
  (do-all-work))
```

Listing 2: The policy handler of a system whose policy is to always busy-wait.

programmatically expressed. For example, Listing 3 expresses a system where mutexes should behave like sleep-locks except when the system is in a state of overload, and except that the subclass of mutexes called `spin-mutex` should always behave like spin-locks.

```
(defvar *system-is-overloaded?* nil)

(defmethod wait-mutex ((mutex mutex))
  (when *system-is-overloaded?*
    (thread-yield mutex))
  (invoke-restart 'retry))

(defmethod wait-mutex ((mutex spin-mutex))
  (invoke-restart 'retry))

(handler-bind ((acquisition-failed
                (lambda (condition)
                  (wait-mutex (mutex condition)))))
  (do-all-work))
```

Listing 3: A more fine-tuned system.

The waiting protocol may also be used to collect statistics about the system's behavior for purposes of profiling or introspection and adaptive behavior. Such functionality can be integrated with the policy handling. It can also be added separately. If a condition handler is written in a "filtering" fashion, so that it is located between the `acquire` mechanism and the policy handler, any `acquisition-failed` condition can be quietly counted. When the statistics handler DECLINES to HANDLE the condition, it will be signaled further to any outer policy handlers as usual.

# 4 Deadlock prevention

The waiting protocol described in Section 3 may also play a role in preventing deadlocks. Deadlocks can result when a process waits for one lock while holding another. Sometimes it makes sense to obtain two or more locks in an "all or nothing" fashion, where the process either successfully obtains every lock, or none. Under our proposal, this is conveniently expressed as shown in Listing 4. This technique requires no change of code at the abstraction level of the policy handler. That is, the same code that determines how to wait for a single mutex, can determine how to wait for a set of mutexes collectively.

```
(define-condition combined-acquisition-failed
    (acquisition-failed)
  ((mutexes :initarg :mutexes :accessor mutexes)))

(defun acquire-atomically (&rest mutexes)
  (loop (with-simple-restart
            (retry "Retry combined acquisition of ~S." mutexes)
          (handler-case (mapc 'acquire mutexes)
            (acquisition-failed (condition)
              (mapc 'release mutexes)
              (error 'combined-acquisition-failed
                :mutex (mutex condition)
                :mutexes mutexes))))))
```

Listing 4: Locking of a number of mutexes in a semi-atomical fashion can help preventing deadlocks.

# 5 Some practical concerns

In Section 3 we described our proposed mechanism without regard to efficiency and or the availability of certain control structures (there are still programming languages in use that do not offer e.g. restarts or even in-context exception handling). In this section we address some such concerns.

## 5.1 A hybrid approach

Much like the traditional hybrid approach mentioned in Section 1 where the process busy-waits a finite number of iterations before sleeping, it might be appropriate to busy-wait a few iterations before signaling the `acquisition-failed` condition. But now another issue of separation of concerns arises, namely the question of how many iterations exactly? It turns out that this concern is easily piggy-backed on the mechanisms we showed in Listings 1 and 2 (although the syntactic sugar of WITH-SIMPLE-RESTART no longer suffices), as shown in Listing 5.

The program in Listing 5 also indicates the general pattern we expect to see when extending the waiting protocol to support practical sleeping policies: Extend the condition object with more contextual information, and have the "retry" restart accept more options for its continuation. It is

```
(define-condition hybrid-acquisition-failed
    (acquisition-failed)
  ((iterations :initarg :iterations :accessor iterations)))

(defun acquire (mutex &optional (busy-wait 1))
  (loop until (dotimes (i busy-wait)
                (when (may-i-enter? mutex)
                  (return t)))
        do (restart-case (error 'hybrid-acquisition-failed
                                :mutex mutex
                                :iterations busy-wait)
             (retry (&optional (new-busy-wait 1))
                :report "Retry acquisition."
               (check-type new-busy-wait (integer 1 *))
               (setf busy-wait new-busy-wait)))))

(handler-bind ((acquisition-failed
                (lambda (condition)
                  (when (> (iterations condition) 1000)
                    (thread-yield (mutex condition)))
                  (invoke-restart 'retry 1000))))
  (do-all-work))
```

Listing 5: A hybrid approach that supports busy-waiting in combination with signaling.

an interesting topic for future work to identify precisely which information to provide and accept in order to enable informed policy decisions in general.

## 5.2 Alternative approaches to a waiting protocol

Listings 1 and 2 implement the fundamental aspects of a waiting protocol fully, using Common Lisp restarts and signaling. However, we may implement approximately the same protocol using simpler mechanisms:

```
(defvar *handle-failed-acquisition*)

(defun simpler-acquire (mutex)
  (loop until (may-i-enter? mutex)
        do (funcall *handle-failed-acquisition* mutex)))
```

And then at the policy-handling level:

```
(let ((*handle-failed-acquisition*
          (lambda (mutex)
            (thread-yield mutex))))
  (do-all-work))
```

This code maps easily to any normal systems programming language.[2] Here, invocation of the "retry" restart occurs implicitly when the handler function returns. This implementation is almost equally powerful to the one based on signaling and restarts. What is lost, however, is the interactive aspects of restarts and the natural integration with the lisp system, which to some degree is what facilitates the construction of bigger abstractions such as the deadlock prevention mechanism outlined in Section 4. But all this may be simulated by building the waiting protocol from simpler

[2]With the possible exception of dynamic bindings, which can be approximated with global variables.

primitives, such as shown above, or what is available in for example the C programming language or indeed assembly.[3] After all, what is any programming language construct but mere syntactic sugaring of any number of lower-level primitives?

We would however like to point out that an implementor of the compiler and run-time system might be able to achieve the best of both worlds by providing special-cased, efficient *and* well-integrated support for this particular usage of restarts and signaling.

# 6 Conclusion

The problem we have addressed is how to implement the mutual exclusion aspects of a general-purpose run-time library while incurring a minimum of policy decisions on whatever is built on top of it. We believe the ideas and mechanisms presented here are appropriate for some such situations.

The question that triggered these ideas was this: "Should the primitive data-types in a Common Lisp run-time be thread-safe or not?" The question is not specific to Common Lisp, but applies to any programming language system or software library in general that offers concurrency and mutable, compound data-structures. The question is interesting because there is a trade-off between safety and simplicity of use on the one hand, and efficiency on the other: While you don't want a simple modification to a hash-table to potentially destroy basic safety invariants, you also do not want to pay the price of process synchronization at every access to any hash-table just to guard against a situation that might never actually occur, even potentially. None of what is presented here resolves this dilemma. However, we believe our proposal decreases the cost of performing synchronization in a general-purpose library in terms of policy implications. This decreased cost may affect the balance of the trade-off when designing a library or a system.

We intend to experiment with the ideas presented here in actual systems, but this remains as future work. It also remains to be investigated if a waiting protocol is a useful way to express other waiting situations besides locking, in which case what we have just presented is only a special case of a more general concept.

# References


[1] ANSI X3.226:1994 programming language Common Lisp, 1994. Also available in hypertext form at `http://www.lispworks.com/reference/HyperSpec/`. 2, 7

[2] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall series in automatic computation. Prentice-Hall, 1976. Chapter 27, "In Retrospect". 2

[3] Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987. 1

[4] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the fifth symposium on Operating systems principles*, pages 132–140. ACM Press, 1975. 2

[5] Kent M. Pitman. Exceptional Situations in Lisp. In *Proceedings for the First European Conference on the Practical Application of Lisp (EUROPAL'90)*, March 1990. 2


[3] Or—obviously—with more complicated and general primitives, such as first-class continuations.

**Appendix**

# A Experimenting code

The following code can be loaded into any ANSI compliant Common Lisp system [1], and demonstrates several variants of a waiting protocol. Try the functions `busy-system`, `fine-tuned-system`, `hybrid-system`, or `simpler-system`. The line-numbers refer to the source file that you might find distributed with this document.

---

```
;;;; Basic functionality that represents an actual system.

(defclass mutex ()
  ((name :initarg :name :reader name :initform (gensym))
   (lockedp :initform nil :accessor lockedp))
  (:documentation
   "This is only a mock-up mutex, it doesn't do actual synchronization."))

(defmethod print-object ((mutex mutex) stream)
  (print-unreadable-object (mutex stream :identity nil :type t)
    (format stream "~S ~:[free~;locked~]"
            (name mutex)
            (lockedp mutex)))
  nil)

(defgeneric wait-mutex (mutex)
  (:documentation "Implement the waiting policy for mutex."))

(defun may-i-enter? (mutex)
  (when (and (not (lockedp mutex))
             (progn (finish-output)
                    (y-or-n-p "Should we let someone enter ~S?" mutex)))
    (setf (lockedp mutex) t)))

(defun release (mutex)
  (setf (lockedp mutex) nil))

(defun thread-yield (reason)
  (warn "Process yielded control waiting for ~S, but we are insomniac."
        reason))

(defmacro with-mutex ((mutex &optional (acquire 'acquire)) &body body)
  ;; In a real system, there will only be one acquire, it will not be passed around as an argument.
  (let ((mutex-var (gensym)))
    `(let ((,mutex-var ,mutex))
       (unwind-protect (progn (funcall ,acquire ,mutex-var) ,@body)
         (release ,mutex-var)))))

(defun do-all-work (&optional (acquire 'acquire))
  "Pretend to do some work, with a particular version of acquire."
  (loop as i upfrom 0
        as mutex = (make-instance 'mutex :name i)
        do (with-mutex (mutex acquire)
             (format t "~&Working with ~S." mutex))))

;;;; Listing 1.

(define-condition acquisition-failed ()
  ((mutex :initarg :mutex :accessor mutex)))

(defun acquire (mutex)
  (loop until (may-i-enter? mutex)
        do (with-simple-restart
             (retry "Retry acquisition of ~S." mutex)
             (error 'acquisition-failed :mutex mutex))))

;;;; Listing 2.

(defun busy-system ()
  (handler-bind
      ((acquisition-failed
        (lambda (condition)
          (declare (ignore condition))
          (invoke-restart 'retry))))
    (do-all-work)))
```

```
;;;; Listing 3.

(defclass spin-mutex (mutex) ())

(defvar *system-is-overloaded?* nil)

(defmethod wait-mutex ((mutex mutex))
  (when *system-is-overloaded?*                                          90
    (thread-yield mutex))
  (invoke-restart 'retry))

(defmethod wait-mutex ((mutex spin-mutex))
  (invoke-restart 'retry))

(defun fine-tuned-system ()
  (handler-bind
      ((acquisition-failed
        (lambda (condition)                                              100
          (wait-mutex (mutex condition)))))
    (do-all-work)))

;;;; Listing 4.

(define-condition combined-acquisition-failed
    (acquisition-failed)
  ((mutexes :initarg :mutexes :accessor mutexes)))

(defun acquire-atomically (&rest mutexes)                                110
  (loop (with-simple-restart
            (retry "Retry combined acquisition of ~S." mutexes)
          (handler-case (map nil 'acquire mutexes)
            (acquisition-failed (condition)
              (map nil 'release mutexes)
              (error 'combined-acquisition-failed
                     :mutex (mutex condition)
                     :mutexes mutexes))))))

;;;; Listing 5.                                                          120

(define-condition hybrid-acquisition-failed (acquisition-failed)
  ((iterations :initarg :iterations :accessor iterations)))

(defun hybrid-acquire (mutex &optional (busy-wait 1))
  (loop until (dotimes (i busy-wait)
                (when (may-i-enter? mutex) (return t)))
        do (restart-case (error 'hybrid-acquisition-failed
                                :mutex mutex
                                :iterations busy-wait)                   130
             (retry (&optional (new-busy-wait 1))
               :report "Retry acquisition."
               (check-type new-busy-wait (integer 1 *))
               (setf busy-wait new-busy-wait)))))

(defun hybrid-system ()
  (handler-bind
      ((acquisition-failed
        (lambda (condition)
          (when (> (iterations condition) 3)                             140
            (thread-yield (mutex condition)))
          (invoke-restart 'retry
                          (progn (format t "~&how may iterations? ")
                                 (finish-output)
                                 (read)))))))
    (do-all-work 'hybrid-acquire)))

;;;; Simpler approach.

(defvar *handle-failed-acquisition*)                                     150

(defun simpler-acquire (mutex)
  (loop until (may-i-enter? mutex)
        do (funcall *handle-failed-acquisition* mutex)))

(defun simpler-system ()
  (let ((*handle-failed-acquisition*
         (lambda (mutex)
           (thread-yield mutex))))
    (do-all-work 'simpler-acquire)))                                     160
```