

Shared State Concurrency

This document is the specification of the Shared State Concurrency programmer interface for Common Lisp. It is dedicated to the public domain.

Abstract

This document defines the Shared State Concurrency (SSC) programmer interface for Common Lisp. The SSC features a standardized, portable interface to work with multiple threads of control, namely: creation, execution and destruction of threads; locks for mutual exclusion of execution in critical sections of code; and synchronization between threads of control.

Overview

A **thread** is a single sequential flow of control in the execution of a program. In a single thread program there is exactly one thread. In a multithreaded program there are two or more threads. A multithreaded Common Lisp implementation has support for the concurrent execution of several threads in the same Lisp world. The execution of threads is **concurrent** in the sense that at any given time, the computer can be in the middle of the execution of several threads. In a multiprocessor machine there can actually be several threads running simultaneously, each one in a different processor. We say that multiple threads are **running in parallel** when they run at the same time, in multiple processors.

Each thread comprises a program counter, an execution (or “call”) stack, and a special variable binding environment; we call these the thread’s **execution context**, and is the information necessary to execute a sequence of instructions independently of all other threads.

There are several **resources** that are shared by all threads in an environment: packages, global function bindings, global values of special variables, and stream, for instance. Because there are shared resources and concurrent execution, several threads can be accessing a shared resource at the same time. It is necessary to ensure that the operations that each one executes does not *conflict* with the concurrent execution of the others. A section of code where a shared resource is accessed is called a **critical section** of that resource.

Thread execution is largely **asynchronous**, which means that each thread executes its code independently of other threads in the Lisp world. There are mechanisms to **synchronize** threads, to enforce dependency on the actions of the other threads. One example of where this is needed is in the execution of critical sections.

All the symbols presented in this document are available as external symbols in a package named or nicknamed **SSC**.

TD: How are the following sections organized?

About the arguments

In the following, it must be observed that parameters named

- *thread* must be thread objects;
- *lock* must be mutual exclusion lock objects;
- *rwlock* must be read/write lock objects;
- *condvar* must be condition variable objects;
- *semaphore* must be semaphore objects;
- *barrier* must be barrier objects;
- *name* must be either strings or `nil`;
- *timeout* must be non-negative real numbers.

It is an error, that must be reported by the implementation, if some argument does not respect these rules.

Additionally, in macros, the parameter named *body* can be empty.

Threads

A thread is useful only if it used to execute some code. The code that a thread executes is given by a parameterless function; that functions is the thread's **associated function**.

`make-thread function &key name &allow-other-keys` *Function*

This creates and returns a new thread and starts execution of *function* in it.

At any instant, a thread can be in one of four states: running, ready, blocked, or dead. A **running thread** is a thread that is currently running. There may be at most one thread running in one processor. This means that in a single processor machine, there may be at most one running thread; in a multiprocessor machine, there may be more than one.

A **ready thread** is a thread that can run, but it is not currently running because it is waiting for a processor. A thread is in this state if it has just started or become unblocked, or because it was preempted by another thread.

A **blocked thread** is a thread that cannot run, because it is “waiting for resources.” This may happen to a thread when it is waiting to acquire a lock or waiting for a condition variable, for example, or when it is waiting for some I/O operation.

A **dead thread** is a thread that has already exited its associated function.

A thread may alternate between ready and running states at any instant. This is controlled by the scheduler, and is more-or-less independent from the thread (if we ignore some hints, like `yield-processor`).

TD: Complete the explanation.

A thread that is not dead is **alive**. After a thread becomes dead, it cannot be made alive again.

`thread-alive-p thread` *Function*

This is a predicate that is true if *thread* is alive. A thread is alive from the moment it is created (just before `make-thread` returns) until sometime after it terminates running its associated function.

It is possible to terminate the execution of thread before it's associated function returns.

`kill-thread thread` *Function*

This terminates the execution of the associated function of *thread*, if it is alive. If *thread* is already dead, this function does nothing.

A thread may not terminate immediately on a call to `kill-thread`, i.e., it is not guaranteed that *thread* has terminated when `kill-thread` returns. On the other hand, `kill-thread` does not return if it is called with the current thread.

`threadp object` *Function*

This is a predicate that is true if *object* is a thread.

`thread-name thread` *Function*

This returns the name of *thread*.

`current-thread` *Function*

This returns the object for the current thread.

`alive-threads` *Function*

This returns a list of threads that are currently alive. One thread that is certain to be a member of (`alive-threads`) is (`current-thread`).

`yield-processor` *Function*

This is an hint to the scheduler that this is a good time to change state of the current thread from running to ready and allow other thread to run in the processor that the current thread is running.

TD: How does the function `cl:sleep` behave?

TD: What about the scheduler?

TD: How does the garbage collector behaves in relation to unreachable and blocked threads?

```
(make-thread #'(lambda ()
                 (let ((m (make-lock)))
                     (acquire-lock m)
                     (condvar-wait (make-condvar m))))))
```

Each thread has a **priority**, a number between -5 and $+5$, that is an indication to the scheduler of how important it is that the thread be chosen to run when there are several threads ready and not enough CPUs for them all. Larger priority numbers represent more important threads. This can be ignored by the implementation.

`thread-priority thread` *Function*

This returns the priority of *thread*; it is a value between -5 and $+5$. This can be changed with `setf`. The argument for the `setf` form must be an integer between -5 and $+5$.

Note that the value you specify with `setf` might not be the one you get back: in an implementation that does not support priorities, `thread-priority` might always return 0, for example.

Mutual exclusion locks

One way to protect a thread from interference from other threads when it is running in a critical region for some resource is to disallow any other thread to enter a critical region for the same resource.

A mutual exclusion lock, or simply lock, is an device that supports sharing of resources common to several threads by mutual exclusion. They protect critical regions of code where only one thread at a time can be running. It is thus guaranteed that there are no concurrent accesses to the resource.

A lock is an object that a thread can **acquire**, in order to claim exclusive access to a shared resource. We then say that the lock is **owned** by the thread that acquires it. A lock not owned by any thread is **unowned**. After the thread is done with the resource, it **releases** the lock. An attempt to acquire a lock only succeeds if the lock is unowned. When a thread attempts to acquire a owned lock, it blocks until the lock is released by other thread.

A lock is **fair** when the order by which threads manage to acquire it is the same as the temporal order that the request is done. Although implementations are encouraged to implement fair locks, this specification does not require that they do.

There are two major kinds of locks: recursive and non-recursive. A recursive lock is one that can be recursively locked by any thread. When a thread recursively acquires the lock, it must be released an equal number of times to really release the lock. A non-recursive lock is one that does not allow recursive locking: if a thread tries to acquire a lock that it is already owned by itself the thread either deadlocks or the acquiring fails.

`make-lock &key name (kind :errorcheck) &allow-other-keys` *Function*

This creates and returns a new, unowned lock. The argument *kind* can be one of `:errorcheck`, `:no-errorcheck`, or `recursive`; it is an error if it is any other object.

`acquire-lock lock` *Function*

This acquires *lock*. If *lock* is already owned by other thread, then the current thread blocks until *lock* becomes unowned. If *lock* is already owned by the current thread, what happens depends on the kind of *lock*. In a recursive lock, the lock is ‘re-acquired’, and it must be released an equal number of times; in a `:errorcheck` lock, a condition `lock-deadlock-error` is signaled; in a `:no-errorcheck` lock the current thread deadlocks. This function returns an unspecified value.

`try-acquire-lock lock` *Function*

This tries to acquire *lock*. If *lock* is unowned, it acquires *lock* and returns `t`; otherwise, it returns `nil`. In a recursive lock that is already owned by the current thread, this function succeeds in re-acquiring the lock and it returns `t`. In a non-recursive lock that is owned this function returns `nil`.

`timed-acquire-lock lock timeout` *Function*

This tries to acquire *lock* until it manages to acquire it or until *timeout* seconds elapse, whichever comes first. If it manages to acquire the lock, the function returns `t`; otherwise it returns `nil`.

The accuracy of *timeout* is unspecified, but implementations are encouraged to keep it under one second. One possible, although not the best, implementation of this function is

```
(defun timed-acquire-lock (lock timeout)
  (loop until (try-acquire-lock lock)
        until (<= timeout 0)
        do (sleep 1)
        do (decf timeout))
  (eq (current-thread) (lock-owner lock)))
```

`release-lock` *lock* *Function*

This releases *lock*. If the lock is recursive, the release depends on the number of times that the lock has been acquired and released by the current thread. If *lock* is not owned by the current thread, a `lock-use-error` condition is signaled, if the lock is recursive or error checking; otherwise, the result is unspecified. This function returns an unspecified value.

`with-lock` (*lock*) &body *body* *Macro*

This executes *body* forms with *lock* owned by the current thread. It returns the values returned by the last form in *body*.

It is something almost like this:

```
(defmacro with-lock ((lock) &body body)
  (let ((lock-sym (gensym "LOCK")))
    `(let ((,lock-sym ,lock))
      (unwind-protect
        (progn
          (acquire-lock ,lock-sym)
          (locally ,@body))
        (release-lock ,lock-sym))))))
```

The problem with this definition is that `with-lock` must not try to release the lock, when unwinding, if the current thread does not own the lock—because acquiring failed, for example.

`lockp` *object* *Function*

This is a predicate that is true if *object* is a lock object.

`lock-name` *lock* *Function*

Returns the name of *lock*. The name of a lock can be altered with `setf`. *object*.

`lock-owner` *lock* *Function*

Returns the thread object of the owner of *lock*. If *lock* is unowned, returns `nil`.

Read/Write Lock

A read/write lock is a lock that protects access to shared resources, like a mutual exclusion lock does, but that distinguishes between threads that want to modify data and threads that merely want to read them. While a mutual exclusion lock does not allow concurrent access to data, a read/write lock allows it as long as none the threads need to change the data.

A read/write lock has a **fairness property**, which is its behavior when there are reader and writer threads trying to lock the read/write lock. A **fair** or **unbiased** lock is one in which readers and writers have access to the lock in the order that that they ask for it: readers wait for earlier writers; writers wait for earlier readers and writers. A **reader biased** lock is one in which preference is given to the readers, allowing a new readers to join a group of current readers even if there are writers waiting. A reader biased lock minimizes the delay for readers at the expense of readers reading possible outdated data. A **writer biased** lock is one in which preference is given to the writers, making new readers wait for any current or waiting writer. A writer biased lock ensures that updates are seen as soon as possible at the expense of less throughput of readers.

`make-rwlock &key name (fairness :fair) &allow-other-keys` *Function*
Creates and returns a new, unlocked read/write lock. The argument *fairness* can be one of `:fair`, `:read-bias`, or `:write-bias`. This argument is an hint to the implementation, and can be ignored. It hints if the lock should be fair, reader biased or writer biased. It is an error if *fairness* is any other object.

`acquire-rwlock rwlock access-kind` *Function*
This locks *rwlock* for the current thread. The argument *access-kind* can be either `:read` or `:write`; it is an error if it is any other object. It specifies which kind of operation the thread intends to do in the resource. If the lock cannot be locked, the current thread blocks until it can. The function returns an unspecified value.

If the current thread has already locked the read/write lock, it deadlocks.

`try-acquire-rwlock rwlock access-kind` *Function*
This is similar to `lock-rwlock` but instead of blocking the current thread if *rwlock* is not available, it returns `nil`. If it succeeds in locking *rwlock*, it returns `t`.

`timed-acquire-rwlock rwlock access-kind timeout` *Function*
This tries to acquire *rwlock* until it manages to acquire it or until *timeout* seconds elapse, whichever comes first. If it manages to acquire the lock, the function returns `t`; otherwise it returns `nil`.

`release-rwlock rwlock` *Function*
This unlocks *rwlock*. It is an error if the current thread does not own *rwlock*. It returns an unspecified value.

`with-rwlock (rwlock access-kind) &body body` *Macro*
This executes the *body* of the macro with *rwlock* locked.

`rwlockp object` *Function*
This is a predicate that is true if *object* is a read/write lock.

`rwlock-name` *rwlock*

Returns the name of *rwlock*.

Function

Condition Variables

A condition variable is a synchronization mechanism used by threads to wait for and signal changes in the state of shared data. Two operations are defined: a thread can wait on a condition variable, blocking itself until a change in some shared data happens; and a thread can signal a condition variable, unblocking some thread that is waiting on it. Usually, a wait is done by a thread that wants to change the shared data, and a signal is done by a thread that has just changed the data.

A condition variable is always connected to the change of state of some shared data. Because of this, it has an **associated lock** that is used to protect the critical sections of those shared data.

`make-condvar lock &key name &allow-other-keys` *Function*

This creates and returns a new condition variable with *lock* as its associated lock.

`wait-condvar condvar` *Function*

This atomically unlocks *condvar*'s associated lock and blocks the current thread on the condition variable *condvar*. Before returning, after the current thread is unblocked, the function relocks *condvar*'s associated lock. This function returns an unspecified value. If this function is called by a thread that does not own *condvar*'s associated lock, a **lock-not-locked-error** condition is signaled.

`wait-condvar/timeout condvar timeout` *Function*

This is similar to `wait-condvar`, but if the condition variable is not signaled before *timeout* seconds pass, this function returns `:timeout`. It returns an unspecified value, different from `:timeout`, if *timeout* seconds have not yet passed.

`signal-condvar condvar &key allp` *Function*

This signals that something changed in the state of the shared resource protected by the lock associated to *condvar*. This means that if some threads are waiting for this condition, then one (if *allp* is `nil`) or all (if *allp* is not `nil`) of the blocked threads should wake up. If there are no threads blocked on *condvar* then this has no effect. This function returns an unspecified value. If this function is called by a thread that does not own *condvar*'s associated lock, a **lock-not-locked-error** condition is signaled.

A **spurious wakeup** is the unblocking of a thread waiting on a condition variable that happens without other thread signaling the condition variable. To allow greater flexibility, implementations are allowed to have this behavior. Because of this, the condition variable's predicate must always be checked.

`condvar-lock condvar` *Function*

This returns the mutual exclusion lock associated with *condvar*. It is an error if *condvar* is not a condition variable.

`condvarp object` *Function*

This is a predicate that is true if *object* is a condition variable.

`condvar-name condvar` *Function*

This return the name of *condvar*.

An idiomatic use of of the operations on a condition variable is

```
(defun enqueue (queue element)
  (with-lock ((condvar-lock (queue-condvar queue)))
    (%enqueue queue element)
    (when (queue-was-empty-p queue)
      (signal-condvar (queue-condvar queue))))))
(defun dequeue (queue)
  (with-lock ((condvar-lock (queue-condvar queue)))
    (loop while (queue-empty queue)
      do (wait-condvar (queue-condvar queue)))
    (%dequeue queue)))
```

Semaphores

Semaphores are synchronization objects with an internal integer counter. Two operations are defined in a semaphore and both act on the semaphore's internal counter: the **down operation** decreases the counter value by 1 and the **up operation** increases it by 1. In addition, and depending on the value of the internal counter, the current thread can block during a down operation or it can unblock a blocked thread during an up operation.

`make-semaphore` *value* &`key` *name* &`allow-other-keys` *Function*

This creates and returns a new semaphore with *value* as the initial value of its internal counter. It is an error if *value* is not an integer.

`down-semaphore` *semaphore* *Function*

This checks the semaphore's internal counter. If the counter is zero, the current thread blocks until some other thread does an up operation in the semaphore, thus increasing the counter. When the counter is greater than zero, the function decreases it by 1 and returns the new value.

`try-down-semaphore` *semaphore* *Function*

This checks the semaphore's internal counter and, if it is greater than zero, decreases it by 1 and returns the new value. If the internal counter is not greater than zero, the function returns `nil`, not changing the semaphore in any way.

This function is similar to `down-semaphore`, but where the `down-semaphore` would block the current thread, this function returns `nil` instead.

`timed-down-semaphore` *semaphore* *timeout* *Function*

This tries to do a down operation on *semaphore* until it manages to do it or until *timeout* seconds elapse, whichever comes first. If it manages to do the down operation, the function returns the value new value of the semaphore; otherwise it returns `nil`.

`up-semaphore` *semaphore* *Function*

This functions increases the semaphore's internal counter value by 1 and returns the new value. This can have the side effect of waking a thread that is blocked in `down-semaphore`.

`with-semaphore` (*semaphore*) &`body` *body* *Macro*

This does a down operation on *semaphore*, executes *body*, and then does an up operation on the *semaphore*. The up operation is done even if *body* exits non-locally. It returns whatever results from evaluating *body*.

`semaphorep` *object* *Function*

This is a predicate that is true if *object* is a semaphore.

`semaphore-name` *semaphore* *Function*

This returns the name of *semaphore*.

Barriers

A Barrier, is a synchronization mechanism used by a fixed number of threads. As each thread reaches the barrier it blocks. When the last thread reach the barrier, all blocked threads are unblocked and the barrier is reset to be used in a new iteration.

`make-barrier` *count* &`key` *name* &`allow-other-keys` *Function*

This creates and returns a barrier with *count* participant threads. It is an error if *count* is not a positive integer.

`pass-barrier` *barrier* *Function*

This blocks the current thread until the required number of threads have called `pass-barrier` on *barrier*. When that occurs, two things happen: the barrier is reset to be used again, in a new iteration; all threads that were blocked are unblocked and this function returns `t` in one of the threads and `nil` in all the others.

`pass-barrier/timeout` *barrier* *timeout* *Function*

This is similar to `pass-barrier`, but it returns `:timeout` if *timeout* seconds pass and some of the other participant threads have not passed yet.

`barrierp` *object* *Function*

This is a predicate that is true if *object* is a cyclic barrier.

`barrier-name` *barrier* *Function*

This return the name of *barrier*. The name of a barrier can be altered with `setf`.

Thread-mailbox

FIX: Isn't there a better term than thread-mailbox?

Any thread can send messages to any thread. A message has a **sender**, which is the thread that sends the message and a **receiver**, which is the recipient thread. The **message** itself can be any lisp object. Each thread has a thread-mailbox, which stores the messages sent to the thread. The messages are stored in a queue, in the order that they were sent. The **first message** in a thread-mailbox is the oldest message that has not it been received by the thread; it is the first message in the queue . The **last message** in a thread-mailbox is the newest message; it is the last message in the queue.

thread-send *thread message* *Function*

This stores *message* as the last message in *thread*'s thread-mailbox. It returns an unspecified value.

If *thread* is dead, then a **message-delivery-error** condition is signaled in the thread that called **thread-send**.

thread-receive *Function*

This returns the first message in the current thread's thread-mailbox. If no message is available, the current thread blocks until one is available.

thread-try-receive *Function*

This is similar to **thread-receive** and it has the same effect in the current thread's thread-mailbox if it is not empty. When no message is available, this function does not block the current thread. It returns two values: if a message is available , it returns (values (thread-receive) t); if a message is not available, it returns (values nil nil).