

## SSC: How it is being Done

This document provides a review of the current state of the art of multithreading API in current CL implementations and in other Lisp dialects. It is dedicated to the public domain.

### Status

This is an initial draft. Most of what is written in this document is based in the documentation of the implementations reviewed. It is assumed that the documents reflect what is implemented.

The reviewed implementations are Allegro, Corman, LispWorks, OpenMCL, and SBCL. Missing implementations include ABCL, CMUCL, ECL, MCL, and SCL.

### Thread management

In Allegro and LispWorks a thread can be either active or inactive. What determines this are two lists that each thread has: one list is the thread's *run-reasons* and the other list its *arrest-reasons*. A thread is active if it has at least one *run-reason* and no *arrest-reasons*. An active thread can run and is considered by the scheduler to run, unless some synchronization mechanism prevents it. A thread is inactive if it has some *arrest-reasons* or if it has no *run-reason*. An inactive thread does not run; it is not considered by the scheduler to run. Any kind of object can be in either of these lists.

### Life cycle

To get a thread running some code, you need to create the thread object, specify the function that the thread should run, and finally instruct it to start running. In all CL implementations you have a function that can do all this in one go. Usually, this function accept arguments for configuring the created thread. One necessary argument is the function that the new thread should run. In all implementations except Corman, the function returns a Lisp object that represents the thread; in Corman, the function returns a thread ID, an integer that is used with all functions that need a thread argument.

In Allegro and OpenMCL you can do each of the steps (creating a thread object, specifying the function to run and start it) separately. Again, the function that creates the thread object accepts several arguments for configuring the new thread. In this case, the function that creates the thread does not take a function-to-run argument.

In Corman and SBCL a thread dies when the function that it was running returns. In these implementations you can use a thread to run a function one time. After the function ends running, the thread cannot run another function. In the other implementations, a thread can be reset and run another function. In Allegro and LispWorks, when a thread is reset, it cancels its present computation, by throwing and unwinding the stack, and restarts running its initial function. In OpenMCL, the present computation is also canceled, but its initial function is not implicitly rerun; also, you use the reset function, with an optional argument, to kill

the thread. In Allegro you can use an optional argument to instruct the thread not to unwind its stack.

In Allegro, a thread can be configured to automatically restart its initial function when it terminates.

In some of the implementations that allow you to reset a thread (for reuse), there is also a preset function that allows you to set the initial function for a thread. This is the same function used to set the initial function for a thread after it is created. In Allegro, this function also resets the thread. In LispWorks you cannot change the initial function of a thread.

In all implementations there are function to throw out of a thread, unwinding its stack. Some implementations also have a brute-force thread termination function: it kill the thread in a way that stack unwinding is not possible.

In Corman and OpenMCL you can suspend a thread, preventing it from running, and stopping it if it is already running. You can also resume a thread that has been previously suspended. In OpenMCL, a thread cannot suspend itself.

## Thread Properties

**TD:** Threads in Allegro can be configured in a number of ways: you can give it a priority, quantum, etc. Each thread has a property list, etc.

## Interrupting

In Allegro, LispWorks, OpenMCL, and SBCL a thread can be interrupted. This is done with something like

*(interrupt-process thread function)*

What this does is to force the execution of *function* in *thread* at some point in the future: *thread* interrupts what it was doing, executes *function* and then resumes execution at the point it was interrupted. The interruption usually happens when *thread* is next scheduled.

In Allegro and OpenMCL a thread can be interrupted while waiting; it resumes waiting after running *function*.

In Allegro an interrupt function can be interrupted by additional thread interrupts. Also, *function* must return normally; otherwise it will not continue its computation. If *thread* is not active when it is interrupted, it is made active to run *function* and made inactive when *function* returns.

LispWorks has a function that forces a thread to enter the debugger.

In OpenMCL a thread can interrupt itself. If *thread* is blocking in a system call, the system call is aborted. The thread must have been started in order to respond to a process interrupt request.

OpenMCL's `without-interrupts` macro defers these kinds of interruptions in a section of code. Allegro and LispWorks also have `without-interrupts` macro. In these two implementations the macro defers these kinds of interruptions, but also disables other kinds of asynchronous interrupts, effectively guaranteeing that the body of the macro is executed without any other thread running, unless the code in body does something to block or to activate scheduling.

## Scheduling

In Allegro, LispWorks and OpenMCL, each threads has a priority, an integer value.

When the scheduler considers which of the available (runnable) thread will run next, it always chooses one that has a priority that is not smaller than any of the other available threads. For any available thread with priority P, no thread with priority less than P will run.

There are functions to get and change the priority of a thread

In LispWorks processes with the same priority are treated equally and fairly: the scheduler uses a round-robin scheme to select the next thread to run.

In Allegro, LispWorks, and OpenMCL there is a function that allows the current thread to yield the processor. This resumes the scheduler and gives other threads a chance to run. The thread that yields does not block or wait; it will run again, as soon as it is selected by the scheduler to run.

## Special Variables

In Allegro, Corman Lisp, and SBCL the global value of special variables is shared by all threads, but each thread has its own (local) special variable bindings. A thread does not inherit dynamic bindings from the thread that created it: when a thread is created, all special variables are bound to the global value of the variable.

## Exclusion

### Mutual Exclusion Locks

All implementations have mutual exclusion locks. In all there is a function to acquire the lock and one to release it, as well as a macro that executes its body with the lock owned. In all implementations except Corman the lock can have a name; it is used for debugging purposes: it shows, for example, in the whostate of the thread that are waiting to acquire it.

In Allegro and LispWorks you can specify a timeout when acquiring the lock. If the timeout is reached and the lock is not free, the function returns `nil`.

In Allegro and SBCL a lock has a value, which normally is the thread that owns the lock; if value is `nil` it is not owned by any thread. At least in SBCL the value is not of much use and in fact some functions will not work right if you set it to a value that is not the thread that owns the lock.

In LispWorks and OpenMCL locks are recursive.

In Allegro you can use an argument with the release function that specifies the value of the lock. This argument is optional and its default value is the current thread. In this way, if the argument is not provided, the function will fail when a thread tries to release a lock that it does not own. On the other hand, this argument can be used to release a lock that is owned by other thread. The `with-process-lock` macro, which acquires the lock before executing its body, has a key parameter to allow recursive locks.

In Corman locks are called critical-sections.

The LispWorks the function `release` will signal an error if the current process does not own the thread, but you can pass an argument to tell it not to. There are functions to retrieve a lock's owner thread, the number of times it has been recursively locked, its name, and whether it is locked or not. When creating a lock, you can use an argument to say that the lock is *important*. You can then use the `free-important-locks` function to free important locks.

**TD:** What is the use of important locks?

In OpenMCL there is a function that tries to acquire the lock and returns `nil` instead of blocking the current thread if the lock is owned by other thread.

In SBCL when you acquire the lock you can pass an argument that specifies that the current thread should not block: if the lock is not available, the function returns `nil`. Locks are not recursive, but there is a `with-recursive-lock` macro that correctly handles recursive locks.

### Read/Write Locks

OpenMCL has read/write locks. There are no functions to acquire and release the lock, only macros that execute their body with the lock acquired for reading or for writing. A thread can recurse in a lock, but it cannot acquire it for reading if it is writing, nor for writing if it is reading, as it would deadlock the thread.

## Synchronization

In Allegro, LispWorks, and OpenMCL, a thread can wait until a given predicate returns true. You call a function with the predicate as an argument and the function returns only when predicate is true. What happens is that the predicate is run periodically (normally when the thread is about to be scheduled to run) and if it returns `nil`, the thread does not run and other thread is scheduled. There is also a function to wait with a timeout: the function returns `nil` if the predicate is not true and the timeout elapses.

In some implementations the wait predicate is checked in different threads (and consequently with different stacks): the current thread (the one that calls the wait function) applies it one time, before yielding to the scheduler if it is false. Then, each time the predicate is checked it is applied in the thread in which the scheduler is running. For example, in

```
(defvar *a* :s1)
(make-thread
  #'(lambda ()
      (let ((*a* :s2))
        (process-wait #'(lambda () (eq *a* value))))))
```

the new thread will not wait if *value* is `:s2`, because of the test made using the new thread's stack (and dynamic environment). It will not wait if *value* is `:s1`, because of the test made in the scheduler thread's stack (assuming that this other thread's code has not rebound `*a*`.) It will wait if *value* is neither `:s1` nor `:s2`. In Allegro, a non-local exit from the predicate is equivalent to the predicate being true, as is if the predicate signals a condition.

In LispWorks you can retrieve the predicate that a thread uses to determine whether it waits or not.

## Condition variables

SBCL has condition variables. There are the usual wait, signal, and broadcast functions. The signal function takes an optional argument with the number of threads to wake up. The default is 1.

## Semaphores

OpenMCL has semaphores. A newly created semaphore always has a count of 0. There are function to perform the usual up and down operations. The up function returns an integer that is “an error identifier that was returned by the underlying OS call.” There is also a timed down operation that returns `nil` if the down operation cannot be performed in the semaphore without blocking it and the timeout elapses.

## Mailboxes

LispWorks has mailboxes. Any thread can receive and send messages from any mailbox. Mailboxes are not bounded. If there are no messages in a mailbox, receiving can either block or timeout. There is also a peek function: it fetches a message but does not remove it from the mailbox. Each thread has a mailbox; you can use the `process-mailbox` function to access it.

## Miscellanea

In Allegro, LispWorks, and OpenMCL you can do a series of default initial bindings in a new thread. Some implementations use a special variable to specify this; others use an argument to the function that creates threads. In either case, this is given as an alist of (`symbol . valueform`) pairs: the valueform is used to compute the value of new binding of symbol in the execution environment of the newly-created thread.

In Allegro and LispWorks there is a function that must be used to start/enable multithreading.

In Corman Lisp you must (`require 'threads`).

LispWorks has simple threads, which have not stack. These threads can be run in the context of any non-simple thread, so they have some limits: they cannot be blocked nor preempted. The function that creates simple processes needs two functions as argument: one is the function the thread will run. The other is a wait-function, a predicate: the function that the thread is supposed to run is only started when this predicate returns true.

Most implementations use a threads name only to help debugging, but LispWorks gives you the ability to find a process by name.

OpenMCL has the `with-terminal-input` that executes its body with exclusive read access to the terminal.

## Debugging

Threads in Allegro, LispWorks, and OpenMCL have a *whostate*. This is a string used to indicate something about the status of the thread. The whostate of a thread changes as the thread changes states (as it blocks, unblocks, dies, etc.) This is where the names of the synchronization objects can be useful. For example, a thread waiting to acquire a mutual exclusion lock named "foo" may have "Blocked on the lock foo." as its whostate.

Some functions allow for explicitly changing the whostate of the current thread. For example, the `process-lock` function, used to wait until a given predicate is true, takes a string argument that becomes the whostate of the thread while the thread is waiting.

In Allegro you can change the whostate of a process with `setf`.

## Other Lisp dialects

### EuLisp

When EuLisp creates a new thread, it installs a default signal handler that handles any signal not handled by the code that the thread runs. What this default handler does is to *abort* the thread.

You can access the value returned by the function running in a thread with the function `thread-value`. The thread that calls `thread-value` waits until the other thread terminates running its function. If the thread was aborted, which can only occur as a result of a signal handled by the default handler, then `thread-value` will signal the condition that aborted the thread on the thread accessing the value.

There is a function that allows a thread to force the signaling of a condition in any other thread. The condition is signaled no later than when the specified thread is rescheduled for execution. Conditions are delivered to the thread in order of receipt.

### Lisp Machine

Lisp Machine Lisp has the `store-conditional` function that provides a low-level mechanism to implement some synchronization mechanisms. The call `(store-conditional location old-value new-value)` stores *new-value* into *location* iff *location* currently contains *old-value*. It returns `t` iff the cell was changed. To get a suitable *location* to use with `store-conditional` you can use the special form `locf`.

Lisp Machine Lisp has process queues. A process queue is a mutex lock with a queue. It guarantees first-in first-out semantics for waiting threads for a certain number of threads. If that number is exceeded, the order is not guaranteed for the excess.