CHAPTER 15
Concurrency

Introduction

In this chapter we're going to take a look at the control of multiple execution strands within a single Lisp image. We'll cover their creation and management, communication between them, shared and private state, interlocks, and critical sections. The central message is of the chapter is that Lisp implementations typically support much or all of this at a high level, leaving you to get on with developing your application without having to know too much about the innards of how threads work.

The terms used to describe concurrency are not consistent between Lisp implementations. Concurrency is typically implemented by multiple OS (or *native*) threads but that's not universally the case. On the other hand it's almost certain that your Lisp image is running within a single OS process. However in this chapter I'm going to reflect common usage within the Lisp community and refer to:

- the use of multiple execution strands as *multi-processing* (abbreviated as *MP*)
- the objects with which MP deals as *processes*.

In some Lisps processes can run in parallel on separate cores—see the notes in "Concurrent Processes" on page 13 towards the end of the chapter.

The ANSI standard does not hint at any of the concepts underlying MP. Nevertheless these days a Lisp implementation which did not support MP would be regarded as somewhat deficient. Certainly all the Lisps discussed in this book support it. MP is frequently mooted as a prime candidate for standardization; although that has never happened the features discussed in this chapter can be regarded as "core" and you'd expect to see most of them supported in some form or other by any MP system. These forms will be recognizably similar to each other and you won't have much trouble adapting from one to another. Whichever Lisp you're using, *read the manual*.

You should also be aware that a number of portable libraries are available which cover much of this common ground. You'll find these listed at:

http://www.cliki.net/thread

The examples in this chapter make extensive use of `lambda` forms and in particular closures (you'll find these described in Chapter 6). The `#.` reader macro (Chapter 5) is a convenient ally when debugging processes and appears in several examples. There's a short section at the end on the application of MP to AllegroCache; if you didn't read the last two chapters then skim it or skip it.

Sources

This chapter is an adaptation for ACL of material which I contributed a few years back to the Common Lisp Cookbook:

http://cl-cookbook.sourceforge.net/process.html

That piece drew its examples from the LispWorks MP implementation of the time (version 4.2) and LW users might prefer to refer to that as well. (But note that it doesn't mention some important features such as concurrent MP which we'll come to later on.)

Why Bother?

The first question to consider is: why bother with MP? Sometimes there's no need to bother: your application is so straightforward that you need not concern yourself with processes at all. But in many other cases it's difficult to imagine how a sophisticated application can be written without MP. How will you get it to respond to events in real time? For example:

- your application is computationally intensive and you want to take advantage of multiple CPUs;
- you might be writing a server which needs to be able to respond to more than one user or connection at a time (for instance: a web server, as in Chapter 24);
- you might want to perform some background activity, without halting the main application while this is going on (for example, in an interactive application you might want to read the contents of a file without halting the user interface);
- you might want your application to be notified when a certain time interval has elapsed;
- you might want to keep the application running and active while waiting for some system resource to become available;
- you might need to interface with some other system which requires multi-threading (for example, windows under Windows which generally run in their own threads);
- you might want to associate different contexts (e.g. different dynamic bindings) with different parts of the application;
- you might even have the simple need to do two things at once.

Emergency Exits

It’s fairly likely when you’re working with processes that—every now and again—one of them will run away from you and either fall motionless at your feet or make a CPU-intensive break for freedom. If you’ve managed to get your Lisp tangled you may be short on options for halting a runaway process, short of terminating the Lisp session altogether. So before you go any further, you should learn how to kill processes in your Lisp development environment. This is easiest if you’re working in an IDE.

- In ACL on Linux or Windows, select *Processes* from the *View* menu (or press Alt-F9) and a *processes dialog* will open; to kill a process which won’t respond to polite coercion select it and press the *Delete* button.
- In LW on any platform, it’s a similar story: *Process Browser* on the *Tools* menu or (aptly enough) press the podium icon of somebody running away. Click the “skull and crossbones” icon to kill the selected process.

In either case, open the dialog before you get into trouble.

Exercise

In the ACL IDE, type

```
(mp:process-run-function "foo" (lambda () (loop)))
```

into a listener and use the process dialog to kill the process.

If you aren’t in an IDE, let’s hope you can find (or fire up) a listener which isn’t totally hosed and into which you can type your implementation’s equivalent of a `process-kill` command (see “Basics” on page 4 below).

You might want to think about designing your application so that these situations can be dealt with cleanly by the end user.

Let’s Get Going

We’ll use ACL for the examples.

Getting Started

Typically, you don't have to do anything to obtain, load or start up your Lisp's MP system: if you're running the Lisp then you're running MP with it. This isn't always the case (e.g. LW on Unix workstations). If you're at all uncertain, read the manual for your implementation and use whatever facility has been provided for starting MP in your image.

In ACL, MP is started *lazily* the first time any process operation is invoked. If you're using the IDE, this will happen long before you get to the Lisp prompt.

Packages

ACL exports its MP utilities from the `MULTIPROCESSING` package, nicknamed `MP`. We'll assume here that you're `cl:use`ing this:

```
(use-package :mp)
```

Basics

The general idea is simple enough: pass a function to `process-run-function` and it'll run in its own *process*. (This may or may not be represented by a real thread at the OS level.)

In the following example, you create a process called `"Incrementing *foo*"`. The function `foo` is invoked (with no arguments) in that process.

```
(defvar *foo* 0)
(defun foo () (incf *foo*))

(process-run-function "Incrementing *foo*" 'foo)
 =>
#<PROCESS Incrementing *foo*(3) @ #x40689562>

*foo*
 =>
1

;; By the time you've typed this, the new process has almost certainly
;; terminated. Note that its printed representation has (subtly) changed.

**
 =>
#<PROCESS Incrementing *foo* @ #x40689562>
```

- The first argument to `process-run-function` is typically a string naming the process. You don't have to make the names unique, but it's a smart move if you do so in actual code, as it's a great way to tell your processes apart when you're debugging them. When playing around in the listener, you'll often find no harm in using an empty string here.

- Alternatively the first argument can be a list of process initialization keywords (one of which is `:name`, so you haven't lost anything if you use this form). See the documentation for `make-process` for details and "Per-process State" on page 6 below for an example.
- The second argument is the function to invoke in the new process. This can be any function designator, for example, an `fboundp` symbol or a `lambda` form.
- Any remaining arguments to `process-run-function` are passed to your function. So calls to `process-run-function` look like calls to `funcall` with an additional argument at the beginning.
- A call to `process-run-function` returns immediately (the return value is of type `process`), while the new process executes in the background (*asynchronously*).
- You'll find the process object on the list `*all-processes*`.
- When the process run-function (`foo` in the above) exits, the process itself no longer has any work to do and is terminated and removed from `*all-processes*`.

To kill a process programmatically (typically: from another process), call `process-kill` with that process as its sole argument. ACL causes the process to *unwind* (by `cl:throw`ing out of its execution stack) and then removes it from `*all-processes*`.

```
CL-USER(1): (process-kill
              (process-run-function "Don't forget to say goodbye"
                                     (lambda ()
                                       (unwind-protect
                                           (loop)
                                         (print "Goodbye"
                                                #.*standard-output*)))))
NIL
CL-USER(2):
"Goodbye"
```

The `#.` notation forces evaluation in the context of the Lisp reader and so returns the value of `*standard-output*` for the listener process. See "Where's my output?" on page 8 below.

If you didn't already have a handle on the process, use `process-name-to-process`. This function has signature

```
(process-name-to-process name &key abbrev (error t))
```

If `abbrev` is set, then the string name only has to match the beginning of a process name for that process to be returned. If `error` is true then failure to match the name to any process will signal an error.

Closures! Warning!

Note the difference between:

```
(let ((table (make-hash-table)))
  (loop for i from 1 to 20 do
        (process-run-function "One closure"
                              (lambda () (setf (gethash i table) t))))
  table)
```

and:

```
(let ((table (make-hash-table)))
  (loop for i from 1 to 20 do
        (process-run-function "Twenty different bindings"
                              (lambda (j) (setf (gethash j table) t))
                              i))
  table)
```

Exercise

Run these forms in the REPL; they both return empty tables. When you evaluate * the number of entries in each table jumps. Why is that?

In the first case, all twenty processes share the same closure variable `i`. The processes execute asynchronously: there's no way to tell exactly when or in what order they'll execute, or how they'll interleave with the process which created them. In this case (try it!) you will observe that all 20 processes are typically initialized before any of them have a chance to start running. By far the easiest way of dealing with this is to ensure that variables which need to be private to each process are bound on a per-process basis.

Exercise

Get two or three new processes running simultaneously, and convince yourself they're all there.

Per-process State

Every process in your Lisp system has its own private state. The following aspects of state will vary between different processes:

- dynamic bindings of special variables;
- catch tags;
- condition handlers, `unwind-protect`, etc.

On the other hand, the following are globally *set* rather than *bound* in a Lisp system and so will not vary between processes:

- definitions of functions, methods, classes, conditions, packages;

- contents of any *compound* objects, such as cons cells, arrays, CLOS objects, hash tables;
- values of closure variables;
- the state of any stream;
- symbol property-lists.

To see how per-process variable bindings can lead you astray, consider the following examples:

```
CL-USER(1): (defvar *foo* nil)

*FOO*
CL-USER(2): (process-run-function "Global set *foo*"
                                  (lambda () (setf *foo* 1)))

#<PROCESS Global set *foo*(7) @ #x20eb96d2>
CL-USER(3): *foo*
1
CL-USER(4): (process-run-function
             "Bind in new process, invisible elsewhere."
             (lambda ()
               (let ((*foo* 2))
                 (sleep 5)              ; affects just this process
                 (setf *foo* 3)
                 (print *foo* #.*standard-output*))))
#<PROCESS Bind in new process, invisible elsewhere.(7) @ #x20ec651a>
CL-USER(5): *foo*                       ; during the 5 second sleep
1
CL-USER(6):
3

CL-USER(7): *foo*
1
CL-USER(8): (let ((*foo* 4))
              (process-run-function
               "Bind in old process, can't see elsewhere."
               (lambda () (print *foo* #.*standard-output*))))
#<PROCESS Bind in old process, can't see elsewhere.(7) @ #x20ed1802>
CL-USER(9):
1
```

In both lines 1 and 2, `*foo*` is set globally. This means that every process shares the symbol's value. In line 4, `*foo*` is bound only within the new process; the original process (i.e. the listener) does not see either this binding or the result of a setf within the binding. Similarly, in line 8 the binding is present only in the original process.

To create a binding which will be present at the beginning of a new process, specify an `:initial-bindings` value:

```
CL-USER(10): (process-run-function '(:name "Bind around new process."

                                     ;; Note the "dotted list" format - an
                                     ;; unpleasant trap for the unwary.
```

```
                                  :initial-bindings ((*foo* . 5))
                                  )
                                (lambda ()
                                  (print *foo* #.*standard-output*)))
#<PROCESS Bind around new process.(7) @ #x20fc5e52>
CL-USER(11):
5

CL-USER(12):
```

ACL also provides `sys:global-symbol-value` for reading or setting a symbol's global value (outside of any bindings) and `symeval-in-process` for accessing its bound value within a given process.

Exercise

Explain from examination of `(process-initial-bindings *current-process*)` why calling `in-package` in one listener does not affect the package in another.

Where's my output?

An obvious way to test whether processes are behaving as you imagine they ought is to get them to print messages to the listener. For example, you might feel justified in trying something like:

```
CL-USER(12): (process-run-function "Where's my output?"
                                (lambda () (print 99)))
#<PROCESS Where's my output?(7) @ #x20e7027a>
CL-USER(13):
```

Where indeed is your output? The answer is that your new process has a different `*standard-output*` to the listener, and that's where your output has gone to. Here is how you might find out where precisely that is:

```
CL-USER(13): (process-run-function "Where indeed?"
                                (lambda ()
                                  (print *standard-output*
                                         #.*standard-output*)))
#<PROCESS Where indeed?(7) @ #x20e820ea>
CL-USER(14):
#<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x201184aa>
```

Exercise

(In the ACL Windows IDE) open the Windows console (i.e. `*terminal-io*`) by evaluating `(cg.base:show-console)`. Now you can prove directly where process output goes (by sending something there).

Exercise

The following behaves differently to the `"Where indeed?"` process above. Why? What's the significance of the "`#.`" in these two examples?

```
(process-run-function "Why not here?"
                      'print
                      *standard-output*
                      #.*standard-output*)
```

Where's my input?

An entirely more annoying situation occurs when two or more processes share the same `*standard-input*`. This is can happen if you're running Lisp from a Unix command line. Consider this interaction (ACL on FreeBSD):

```
CL-USER(1): *current-process*
#<PROCESS Initial Lisp Listener(2) @ #x4052e332>
CL-USER(2): (process-run-function "foo" 'error "bar")
#<PROCESS foo(3) @ #x405f757a>
CL-USER(3):
Error: bar

Restart actions (select using :continue):
 0: Abort entirely from this (lisp) process.
[Current process: foo]
[1] CL-USER(1): *current-process*
#<PROCESS Initial Lisp Listener(2) @ #x4052e332>
CL-USER(4): *current-process*
#<PROCESS foo(3) @ #x405f757a>
[Current process: foo]
[1] CL-USER(2):
```

Two processes (the original Lisp listener and your own `"foo"`) have the same `*standard-input*`. The Lisp scheduler appears to be switching between them each time it generates a prompt. (Things could be worse. Another Lisp in which I tried that out reliably switched contexts part-way through reading lines of input from the prompt, which made bailing out somewhat hit and miss.)

The best approach to this particular hazard is to avoid it altogether. If you're compelled to debug a multi-processing application at the command line, consider using *SLIME* (Chapter 18).

Interactions Between Processes

Waiting

In all the above examples, a process is created to run a simple function and then halt. In a typical application at least some of your processes will run an event loop of some sort. An event loop is a function which repeatedly waits for an event external to that process to occur. When an event is noticed, it is dispatched (maybe to another process) for processing and the event loop cycles back to its waiting state. The "other process" here might already exist (perhaps running an event loop of its own) or might be created specifically to perform this task (i.e. handle a single event) and then terminate.

It might be tempting to construct an event loop using `cl:sleep`. For example:

```
(defun bogus-event-loop ()
  (loop
   (sleep 1)                          ; THIS IS WRONG
   (when (something-has-happened)
     (act-on-that-thing))))

(process-run-function "Bogus event loop"
                      'bogus-event-loop)
```

This is a poor choice, because you're condemned to wait for the `sleep` to return before you can perform the wake-up test. (Also, it's conceivable that your implementation cannot sleep one single process without sleeping the whole Lisp executable. In such cases, any processing required before the predicate `something-has-happened` can return true will never happen.)

Consider instead:

```
(defun improved-event-loop ()
  (loop
   (process-wait "Waiting for something to happen"
                 'something-has-happened)
   (act-on-that-thing)))
```

The arguments to `process-wait` are a string (which you should use for describing what this process is waiting for), a function and optionally arguments to that function. The call to `process-wait` will not return until this *wait-function* returns true.

Keep the wait-function simple: don't use it to perform expensive calculations and be wary of using it to mess with MP. In particular, don't call `process-wait` recursively (in ACL for example this causes the process to hang).

If you're waiting for input on an external stream, the call `(wait-for-input-available stream)` will ensure that your Lisp session notices the input when it arrives.

Exercise

Read the ACL documentation on `wait-for-input-available`. (If you're using the IDE, type the symbol into a listener, select it and press the F1 key.)

There's often a risk that the wait-function will never return true, meaning that the waiting process might wait forever. If you don't want that to happen, consider instead using `process-wait-with-timeout`. This function has signature

```
(process-wait-with-timeout reason timeout wait-function &rest arguments)
```

where `timeout` can be any non-negative real number. If the wait-function succeeds before that number of seconds has passed, it returns true; otherwise it stops waiting and returns `nil`. For example:

```
(defun check-wait-for-file (file timeout)
  (or (probe-file file)
      (process-wait-with-timeout (format nil "Waiting for \"~a\"" file)
                                 timeout
                                 'probe-file file)
      (error "File \"~a\" still not found" file)))
```

Wait-functions are run when the system attempts to switch into the waiting processes. If there are several active processes, Lisp will switch between them many times per second. How often that is depends on implementation and OS platform: if you need to know your system's granularity, experiment!

Exercise

Adapt `check-wait-for-file` to count how many times the wait function was called. This will give you a rough estimate of how often the scheduler switches between processes.

Related to `process-wait-with-timeout` is the macro `with-timeout`:

```
(with-timeout (seconds &rest timeout-forms) &body body)
```

The body is run as an implicit `progn`. If it takes longer than `seconds` to do this, execution throws out and the timeout-forms are run, returning the result of the last form.

Data-driven Interrupts

An important mode of communication between processes is the *interrupt*: a message sent by one process to another, to ask that process to perform some task. This might come in handy if:

- you need certain actions to be performed *synchronously*, for example you need your web server to flush some cache before it accepts any further connections; or
- you need actions to be performed in the dynamic context of another process (so as to access its variable bindings, perhaps).

The safest approach is to use a *message queue*. One process "owns" the queue and other processes can "post" tasks to it; the owning process deals with them one at a time when it's good and ready to do so.

ACL supports this paradigm with the class `queue` (a *first in*, *first out* data structure), atomic operators `enqueue` and `dequeue`, and the predicate `queue-empty-p`. The accessor `process-property-list` is also useful here.

```
;; Call (event-handler) to set up a handling process which processes events
;; from its queue, always completing one before going on to the next. Use
;; handle-in-process below to add events to the queue.

(defun event-handler ()
  (process-run-function "Event handler"
                        'handle-events))

(defun handle-events ()
  (let ((queue (make-instance 'queue)))

    ;; Put the queue where client processes can find it

    (setf (getf (process-property-list *current-process*) 'queue)
          queue)

    (loop ;; Wait for a request to arrive (when one does the queue
          ;; will stop being empty)

          (process-wait "Waiting for request in queue"
                        (lambda () (not (queue-empty-p queue))))

          ;; Now we know a request has arrived, peel it off the queue
          ;; and run it

          (let ((request (dequeue queue)))
            (funcall request)))))

(defun handle-in-process (process function &rest arguments)
  (let* ((queue (getf (process-property-list process) 'queue))
         (no-result (cons nil nil))
         (result no-result))

    ;; Send function to other process
    (enqueue queue
             (lambda ()
               (setf result (apply function arguments))))

    ;; Wait for the value of RESULT to change, and then return it

    (process-wait (format nil "Waiting for response from ~a"
                          (process-name process))
                  (lambda () (not (eq result no-result))))
    result))
```

Exercise

What is interesting about the value of `no-result`?

LispWorks uses objects called `mailbox`es which support this queues attached to processes. These are documented in the LW Reference Manual.

Atomic Operators

When you're working on a data structure you sometimes need to be certain that whatever you're doing to it will be complete before any other process touches that data. Such an operation is referred to as a *critical section*, or as being *atomic*.

You *cannot assume* that any operator is atomic unless your implementation's documentation has explicitly told you that this is the case. Some implementations support atomic versions of common operators, for example `atomic-push`; when they do that's a firm warning that the CL counterparts (in this case `cl:push`) are not *thread-safe*.

Exercise

What might happen if two different processes were to `cl:push` to the same location at approximately the same time?

Concurrent Processes

Traditionally Lisp implements MP by emuluating multi-tasking. When it comes down to it, only one task is being performed at a time. Forthcoming releases of both ACL and LW will support *symmetric* multiprocessing: *concurrent* threads running in parallel on multiple cores. This will allow for considerable performance gains but will have implications for the enforcement of atomicity. Both implementations will introduce new operators for controlling concurrency as well as a number of primitives for atomic read-modify-write operations.

Some other implementations already provide symmetric MP, albeit with less sophisticated controls. In all cases, you are strongly advised to read your documentation with regard to concurrency. For ACL users an upgrade note is available here:

> *http://www.franz.com/support/documentation/current/doc/smp.htm*

Locks

Sometimes it's important to control access to a resource, so that only one process is operating on it at a time. You could do this is by restricting access from your code to that resource and only allowing one specialized process to touch it, using an event handler to enforce this. However there are two potential drawbacks to this approach:

- frequently you'll need to wait in the invoking process until the operation on the resource is complete;
- this is becoming a somewhat heavyweight mechanism for handling what should be a fairly simple operator.

A *lock* is a simple Lisp object, which can be *held* by no more than one process at a time. A process attempting to hold a lock which is already in use will hang (i.e. be forced to wait) until the lock is *freed*. In the following example, the locking mechanism is reduced to its most minimal form.

```
;; Create a lock and remember it.

(defvar *lock* (make-process-lock))

(defun use-resource-when-free (stream)

  ;; Callers must wait at this point for the lock to be freed,
  ;; before they can proceed with the body of this form.

  (with-process-lock (*lock*)
    (use-resource-anyway stream)

    ;; When we exit this form, the lock is freed automatically
    ;; and other processes will be allowed to claim it.
    ))

(defun use-resource-anyway (stream)
  (let ((name (process-name *current-process*)))
    (format stream "~&Starting ~a." name)
    (sleep 1)
    (format stream "~&Ending ~a." name)))

(defun test (lock-p)
  (let ((run-function (if lock-p
                          'use-resource-when-free
                        'use-resource-anyway)))
    (dotimes (id 3)
      (process-run-function (format nil "process ~a" id)
                            run-function *standard-output*))))
```

Your implementation may also support other process-friendly structures, such as `semaphores`. Check your documentation.

Allegrocache

Recall that AllegroCache connections are not *thread-safe*: you should not allow two processes to "use the same connection" at the same time. We're now in a position to consider potential solutions to this problem.

Exercise

One option is to use a lock to guard all unsafe operations. Implement this, in part at least. You won't want dozens of `(with-lock ...)` forms scattered through your code. How will you address this?

Another option is to maintain a *pool* of several connections from which processes can request a connection and to which they can be guaranteed to return it:

```
(defun process-run-with-connection (name-or-options function &rest args)
  (process-run-function name-or-options
                        (lambda ()

                          ;; Per-process binding of *allegrocache* keeps
                          ;; the connection private

                          (let* ((*allegrocache* (request-connection)))
                            (unwind-protect
                                (apply function args)
                              (return-connection *allegrocache*))))))

;; Use of a QUEUE ensures atomicity (we don't want two processes picking
;; up the same connection)

(defvar *connections* (make-instance 'queue))

(defun request-connection ()
  (or (dequeue *connections*)
      (open-network-database ...)))

(defun return-connection (connection)
  (enqueue *connections* connection))
```

Exercise

Adapt this example to enforce an upper limit on the number of connections.

Problems?

We should end this chapter with a word of caution. MP errors can be very hard to debug. Your only symptom may be “unexplainable” data corruption or “illogical” program flow. You may find the problem hard or impossible to reproduce, even though you have incontrovertible proof of it happening once. Any of this should point you to examine your MP usage. Pay attention to anecdotal evidence. I have in mind a *race condition* whose outcome flipped after one of my users defragmented his hard drive thus speeding up processes which accessed the disk. Ask yourself: “clearly this is impossible, but how could this have happened?”, “can I document the order on which I depend upon things happening?”, “is anything else unexpected happening?”