21

Multiple Processes

The previous chapter showed how continuations allow a running program to get hold of its own state, and store it away to be restarted later. This chapter deals with a model of computation in which a computer runs not one single *program*, but a collection of independent *processes*. The concept of a process corresponds closely with our concept of the state of a program. By writing an additional layer of macros on top of those in the previous chapter, we can embed multiprocessing in Common Lisp programs.

21.1 The Process Abstraction

Multiple processes are a convenient way of expressing programs which must do several things at once. A traditional processor executes one instruction at a time. To say that multiple processes do more than one thing at once is not to say that they somehow overcome this hardware limitation: what it means is that they allow us to think at a new level of abstraction, in which we don't have to specify exactly what the computer is doing at any given time. Just as virtual memory allows us to act as though the computer had more memory than it actually does, the notion of a process allows us to act as if the computer could run more than one program at a time.

The study of processes is traditionally in the domain of operating systems. But the usefulness of processes as an abstraction is not limited to operating systems. They are equally useful in other real-time applications, and in simulations.

Much of the work done on multiple processes has been devoted to avoiding certain types of problems. Deadlock is one classic problem with multiple pro-

cesses: two processes both stand waiting for the other to do something, like two people who each refuse to cross a threshold before the other. Another problem is the query which catches the system in an inconsistent state—say, a balance inquiry which arrives while the system is transferring funds from one account to another. This chapter deals only with the process abstraction itself; the code presented here could be used to test algorithms for preventing deadlock or inconsistent states, but it does not itself provide any protection against these problems.

The implementation in this chapter follows a rule implicit in all the programs in this book: disturb Lisp as little as possible. In spirit, a program ought to be as much as possible like a modification of the language, rather than a separate application written in it. Making programs harmonize with Lisp makes them more robust, like a machine whose parts fit together well. It also saves effort; sometimes you can make Lisp do a surprising amount of your work for you.

The aim of this chapter is to make a language which supports multiple processes. Our strategy will be to turn Lisp into such a language, by adding a few new operators. The basic elements of our language will be as follows:

Functions will be defined with the `=defun` or `=lambda` macros from the previous chapter.

Processes will be instantiated from function calls. There is no limit on the number of active processes, or the number of processes instantiated from any one function. Each process will have a priority, initially given as an argument when it is created.

Wait expressions may occur within functions. A wait expression will take a variable, a test expression, and a body of code. If a process encounters a wait, the process will be suspended at that point until the test expression returns true. Once the process restarts, the body of code will be evaluated, with the variable bound to the value of the test expression. Test expressions should not ordinarily have side-effects, because there are no guarantees about when, or how often, they will be evaluated.

Scheduling will be done by priority. Of all the processes able to restart, the system will run the one with the highest priority.

The default process will run if no other process can. It is a read-eval-print loop.

Creation and deletion of most objects will be possible on the fly. From running processes it will be possible to define new functions, and to instantiate and kill processes.

```
(defstruct proc  pri state wait)

(proclaim '(special *procs* *proc*))

(defvar *halt* (gensym))

(defvar *default-proc*
        (make-proc :state #'(lambda (x)
                              (format t "~%>> ")
                              (princ (eval (read)))
                              (pick-process))))

(defmacro fork (expr pri)
  `(prog1 ',expr
          (push (make-proc
                  :state #'(lambda (,(gensym))
                             ,expr
                             (pick-process))
                  :pri   ,pri)
                *procs*)))

(defmacro program (name args &body body)
  `(=defun ,name ,args
     (setq *procs* nil)
     ,@body
     (catch *halt* (loop (pick-process)))))
```

Figure 21.1: Process structure and instantiation.

Continuations make it possible to store the state of a Lisp program. Being able to store several states at once is not very far from having multiple processes. Starting with the macros defined in the previous chapter, we need less than 60 lines of code to implement multiple processes.

21.2 Implementation

Figures 21.1 and 21.2 contain all the code needed to support multiple processes. Figure 21.1 contains code for the basic data structures, the default process, initialization, and instantiation of processes. Processes, or `procs`, have the following structure:

`pri` is the priority of the process, which should be a positive number.

`state` is a continuation representing the state of a suspended process. A process is restarted by funcalling its `state`.

`wait` is usually a function which must return true in order for the process to be restarted, but initially the `wait` of a newly created process is `nil`. A process with a null `wait` can always be restarted.

The program uses three global variables: `*procs*`, the list of currently suspended processes; `*proc*`, the process now running; and `*default-proc*`, the default process.

The default process runs only when no other process can. It simulates the Lisp toplevel. Within this loop, the user can halt the program, or type expressions which enable suspended processes to restart. Notice that the default process calls `eval` explicitly. This is one of the few situations in which it is legitimate to do so. Generally it is not a good idea to call `eval` at runtime, for two reasons:

1. It's inefficient: `eval` is handed a raw list, and either has to compile it on the spot, or evaluate it in an interpreter. Either way is slower than compiling the code beforehand, and just calling it.

2. It's less powerful, because the expression is evaluated with no lexical context. Among other things, this means that you can't refer to ordinary variables visible outside the expression being evaluated.

Usually, calling `eval` explicitly is like buying something in an airport gift-shop. Having waited till the last moment, you have to pay high prices for a limited selection of second-rate goods.

Cases like this are rare instances when neither of the two preceding arguments applies. We couldn't possibly have compiled the expressions beforehand. We are just now reading them; there is no beforehand. Likewise, the expression can't refer to surrounding lexical variables, because expressions typed at the toplevel are in the null lexical environment. In fact, the definition of this function simply reflects its English description: it reads and evaluates what the user types.

The macro `fork` instantiates a process from a function call. Functions are defined as usual with `=defun`:

```
(=defun foo (x)
  (format t "Foo was called with ~A.~%" x)
  (=values (1+ x)))
```

Now when we call `fork` with a function call and a priority number:

```
(fork (foo 2) 25)
```

a new process is pushed onto `*procs*`. The new process has a priority of 25, a `proc-wait` of `nil`, since it hasn't been started yet, and a `proc-state` consisting of a call to `foo` with the argument 2.

The macro `program` allows us to create a group of processes and run them together. The definition:

```
(program two-foos (a b)
  (fork (foo a) 99)
  (fork (foo b) 99))
```

macroexpands into the two `fork` expressions, sandwiched between code which clears out the suspended processes, and other code which repeatedly chooses a process to run. Outside this loop, the macro establishes a tag to which control can be thrown to end the program. As a gensym, this tag will not conflict with tags established by user code. A group of processes defined as a `program` returns no particular value, and is only meant to be called from the toplevel.

After the processes are instantiated, the process scheduling code takes over. This code is shown in Figure 21.2. The function `pick-process` selects and runs the highest priority process which is able to restart. Selecting this process is the job of `most-urgent-process`. A suspended process is eligible to run if it has no `wait` function, or its `wait` function returns true. Among eligible processes, the one with the highest priority is chosen. The winning process and the value returned by its `wait` function (if there is one) are returned to `pick-process`. There will always be some winning process, because the default process always wants to run.

The remainder of the code in Figure 21.2 defines the operators used to switch control between processes. The standard wait expression is `wait`, as used in the function `pedestrian` in Figure 21.3. In this example, the process waits until there is something in the list `*open-doors*`, then prints a message:

```
> (ped)
>> (push 'door2 *open-doors*)
Entering DOOR2
>> (halt)
NIL
```

A `wait` is similar in spirit to an `=bind` (page 267), and carries the same restriction that it must be the last thing to be evaluated. Anything we want to happen after the `wait` must be put in its body. Thus, if we want to have a process wait several times, the wait expressions must be nested. By asserting facts aimed at one another, processes can cooperate in reaching some goal, as in Figure 21.4.

```
(defun pick-process ()
  (multiple-value-bind (p val) (most-urgent-process)
    (setq *proc*  p
          *procs* (delete p *procs*))
    (funcall (proc-state p) val)))

(defun most-urgent-process ()
  (let ((proc1 *default-proc*) (max -1) (val1 t))
    (dolist (p *procs*)
      (let ((pri (proc-pri p)))
        (if (> pri max)
            (let ((val (or (not (proc-wait p))
                           (funcall (proc-wait p)))))
              (when val
                (setq proc1 p
                      max   pri
                      val1  val))))))
    (values proc1 val1)))

(defun arbitrator (test cont)
  (setf (proc-state *proc*) cont
        (proc-wait *proc*)  test)
  (push *proc* *procs*)
  (pick-process))

(defmacro wait (parm test &body body)
  `(arbitrator #'(lambda () ,test)
               #'(lambda (,parm) ,@body)))

(defmacro yield (&body body)
  `(arbitrator nil #'(lambda (,(gensym)) ,@body)))

(defun setpri (n) (setf (proc-pri *proc*) n))

(defun halt (&optional val) (throw *halt* val))

(defun kill (&optional obj &rest args)
  (if obj
      (setq *procs* (apply #'delete obj *procs* args))
      (pick-process)))
```

Figure 21.2: Process scheduling.

```
(defvar *open-doors* nil)

(=defun pedestrian ()
  (wait d (car *open-doors*)
    (format t "Entering ~A~%" d)))

(program ped ()
  (fork (pedestrian) 1))
```

Figure 21.3: One process with one wait.

Processes instantiated from `visitor` and `host`, if given the same `door`, will exchange control via messages on a blackboard:

```
> (ballet)
Approach DOOR2. Open DOOR2. Enter DOOR2. Close DOOR2.
Approach DOOR1. Open DOOR1. Enter DOOR1. Close DOOR1.
>>
```

There is another, simpler type of wait expression: `yield`, whose only purpose is to give other higher-priority processes a chance to run. A process might want to yield after executing a `setpri` expression, which resets the priority of the current process. As with a `wait`, any code to be executed after a `yield` must be put within its body.

The program in Figure 21.5 illustrates how the two operators work together. Initially, the barbarians have two aims: to capture Rome and to plunder it. Capturing the city has (slightly) higher priority, and so will run first. However, after the city has been reduced, the priority of the `capture` process decreases to `1`. Then there is a vote, and `plunder`, as the highest-priority process, starts running.

```
> (barbarians)
Liberating ROME.
Nationalizing ROME.
Refinancing ROME.
Rebuilding ROME.
>>
```

Only after the barbarians have looted Rome's palaces and ransomed the patricians, does the `capture` process resume, and the barbarians turn to fortifying their own position.

Underlying wait expressions is the more general `arbitrator`. This function stores the current process, and then calls `pick-process` to start some process

```
(defvar *bboard* nil)

(defun claim   (&rest f) (push f *bboard*))

(defun unclaim (&rest f) (pull f *bboard* :test #'equal))

(defun check   (&rest f) (find f *bboard* :test #'equal))

(=defun visitor (door)
  (format t "Approach ~A. " door)
  (claim 'knock door)
  (wait d (check 'open door)
    (format t "Enter ~A. " door)
    (unclaim 'knock door)
    (claim 'inside door)))

(=defun host (door)
  (wait k (check 'knock door)
    (format t "Open ~A. " door)
    (claim 'open door)
    (wait g (check 'inside door)
      (format t "Close ~A.~%" door)
      (unclaim 'open door))))

(program ballet ()
  (fork (visitor 'door1) 1)
  (fork (host 'door1) 1)
  (fork (visitor 'door2) 1)
  (fork (host 'door2) 1))
```

Figure 21.4: Synchronization with a blackboard.

(perhaps the same one) running again. It will be given two arguments: a test function and a continuation. The former will be stored as the `proc-wait` of the process being suspended, and called later to determine if it can be restarted. The latter will become the `proc-state`, and calling it will restart the suspended process.

The macros `wait` and `yield` build this continuation function simply by wrapping their bodies in lambda-expressions. For example,

```
(wait d (car *bboard*) (=values d))
```

```
(=defun capture (city)
  (take city)
  (setpri 1)
  (yield
    (fortify city)))

(=defun plunder (city)
  (loot city)
  (ransom city))

(defun take (c)    (format t "Liberating ~A.~%" c))
(defun fortify (c) (format t "Rebuilding ~A.~%" c))
(defun loot (c)    (format t "Nationalizing ~A.~%" c))
(defun ransom (c)  (format t "Refinancing ~A.~%" c))

(program barbarians ()
  (fork (capture 'rome) 100)
  (fork (plunder 'rome) 98))
```

Figure 21.5: Effect of changing priorities.

expands into:

```
(arbitrator #'(lambda () (car *bboard*))
            #'(lambda (d) (=values d)))
```

If the code obeys the restrictions listed in Figure 20.5, making a closure of the `wait`'s body will preserve the whole current continuation. With its `=values` expanded the second argument becomes:

```
#'(lambda (d) (funcall *cont* d))
```

Since the closure contains a reference to `*cont*`, the suspended process with this wait function will have a handle on where it was headed at the time it was suspended.

The `halt` operator stops the whole program, by throwing control back to the tag established by the expansion of `program`. It takes an optional argument, which will be returned as the value of the program. Because the default process is always willing to run, the only way programs end is by explicit `halts`. It doesn't matter what code follows a `halt`, since it won't be evaluated.

Individual processes can be killed by calling `kill`. If given no arguments, this operator kills the current process. In this case, `kill` is like a wait expression

which neglects to store the current process. If `kill` is given arguments, they become the arguments to a `delete` on the list of processes. In the current code, there is not much one can say in a `kill` expression, because processes do not have many properties to refer to. However, a more elaborate system would associate more information with processes—time stamps, owners, and so on. The default process can't be killed, because it isn't kept in the list `*procs*`.

21.3 The Less-than-Rapid Prototype

Processes simulated with continuations are not going to be nearly as efficient as real operating system processes. What's the use, then, of programs like the one in this chapter?

Such programs are useful in the same way that sketches are. In *exploratory programming* or *rapid prototyping*, the program is not an end in itself so much as a vehicle for working out one's ideas. In many other fields, something which serves this purpose is called a sketch. An architect could, in principle, design an entire building in his head. However, most architects seem to think better with pencils in their hands: the design of a building is usually worked out in a series of preparatory sketches.

Rapid prototyping is sketching software. Like an architect's first sketches, software prototypes tend to be drawn with a few sweeping strokes. Considerations of cost and efficiency are ignored in an initial push to develop an idea to the full. The result, at this stage, is likely to be an unbuildable building or a hopelessly inefficient piece of software. But the sketches are valuable all the same, because

1. They convey information briefly.
2. They offer a chance to experiment.

The program described in this chapter is, like those in succeeding chapters, a sketch. It suggests the outlines of multiprocessing in a few, broad strokes. And though it would not be efficient enough for use in production software, it could be quite useful for experimenting with other aspects of multiple processes, like scheduling algorithms.

Chapters 22–24 present other applications of continuations. None of them is efficient enough for use in production software. Because Lisp and rapid prototyping evolved together, Lisp includes a lot of features specifically intended for prototypes: inefficient but convenient features like property lists, keyword parameters, and, for that matter, lists. Continuations probably belong in this category. They save more state than a program is likely to need. So our continuation-based implementation of Prolog, for example, is a good way to understand the language, but an inefficient way to implement it.

This book is concerned more with the kinds of abstractions one can build in Lisp than with efficiency issues. It's important to realize, though, that Lisp is a language for writing production software as well as a language for writing prototypes. If Lisp has a reputation for slowness, it is largely because so many programmers stop with the prototype. It is easy to write fast programs in Lisp. Unfortunately, it is *very* easy to write slow ones. The initial version of a Lisp program can be like a diamond: small, clear, and very expensive. There may be a great temptation to leave it that way.

In other languages, once you succeed in the arduous task of getting your program to work, it may already be acceptably efficient. If you tile a floor with tiles the size of your thumbnail, you don't waste many. Someone used to developing software on this principle may find it difficult to overcome the idea that when a program works, it's finished. "In Lisp you can write programs in no time at all," he may think, "but boy, are they slow." In fact, neither is the case. You can get fast programs, but you have to work for them. In this respect, using Lisp is like living in a rich country instead of a poor one: it may seem unfortunate that one has to work to stay thin, but surely this is better than working to stay alive, and being thin as a matter of course.

In less abstract languages, you work for functionality. In Lisp you work for speed. Fortunately, working for speed is easier: most programs only have a few critical sections in which speed matters.