9

Coroutines

A *coroutine* is similar to a thread (in the sense of multithreading): a line of execution, with its own stack, its own local variables, and its own instruction pointer; but sharing global variables and mostly anything else with other coroutines. The main difference between threads and coroutines is that, conceptually (or literally, in a multiprocessor machine), a program with threads runs several threads concurrently. Coroutines, on the other hand, are collaborative: A program with coroutines is, at any given time, running only one of its coroutines and this running coroutine only suspends its execution when it explicitly requests to be suspended.

Coroutine is a powerful concept. As such, several of its main uses are complex. Do not worry if you do not understand some of the examples in this chapter on your first reading. You can read the rest of the book and come back here later. But please come back. It will be time well spent.

9.1 Coroutine Basics

Lua offers all its coroutine functions packed in the `coroutine` table. The `create` function creates new coroutines. It has a single argument, a function with the code that the coroutine will run. It returns a value of type `thread`, which represents the new coroutine. Quite often, the argument to `create` is an anonymous function, like here:

```
co = coroutine.create(function ()
       print("hi")
     end)

print(co)   --> thread: 0x8071d98
```

A coroutine can be in one of three different states: suspended, running, and dead. When we create a coroutine, it starts in the suspended state. That means that a coroutine does not run its body automatically when we create it. We can check the state of a coroutine with the `status` function:

```
print(coroutine.status(co))   --> suspended
```

The function `coroutine.resume` (re)starts the execution of a coroutine, changing its state from suspended to running:

```
coroutine.resume(co)   --> hi
```

In this example, the coroutine body simply prints "`hi`" and terminates, leaving the coroutine in the dead state, from which it cannot return:

```
print(coroutine.status(co))   --> dead
```

Until now, coroutines look like nothing more than a complicated way to call functions. The real power of coroutines stems from the `yield` function, which allows a running coroutine to suspend its execution so that it can be resumed later. Let us see a simple example:

```
co = coroutine.create(function ()
       for i=1,10 do
         print("co", i)
         coroutine.yield()
       end
     end)
```

Now, when we resume this coroutine, it starts its execution and runs until the first `yield`:

```
coroutine.resume(co)    --> co   1
```

If we check its status, we can see that the coroutine is suspended and therefore can be resumed again:

```
print(coroutine.status(co))   --> suspended
```

From the coroutine's point of view, all activity that happens while it is suspended is happening inside its call to `yield`. When we resume the coroutine, this call to `yield` finally returns and the coroutine continues its execution until the next yield or until its end:

```
coroutine.resume(co)    --> co   2
coroutine.resume(co)    --> co   3
...
coroutine.resume(co)    --> co   10
coroutine.resume(co)    -- prints nothing
```

During the last call to `resume`, the coroutine body finished the loop and then returned, so the coroutine is dead now. If we try to resume it again, `resume` returns **false** plus an error message:

```
print(coroutine.resume(co))
--> false   cannot resume dead coroutine
```

Note that `resume` runs in protected mode. Therefore, if there is any error inside a coroutine, Lua will not show the error message, but instead will return it to the `resume` call.

A useful facility in Lua is that a pair resume–yield can exchange data between them. The first `resume`, which has no corresponding `yield` waiting for it, passes its extra arguments as arguments to the coroutine main function:

```
co = coroutine.create(function (a,b,c)
       print("co", a,b,c)
     end)
coroutine.resume(co, 1, 2, 3)    --> co  1  2  3
```

A call to `resume` returns, after the **true** that signals no errors, any arguments passed to the corresponding `yield`:

```
co = coroutine.create(function (a,b)
       coroutine.yield(a + b, a - b)
     end)
print(coroutine.resume(co, 20, 10))  --> true  30  10
```

Symmetrically, `yield` returns any extra arguments passed to the corresponding `resume`:

```
co = coroutine.create (function ()
       print("co", coroutine.yield())
     end)
coroutine.resume(co)
coroutine.resume(co, 4, 5)     --> co  4  5
```

Finally, when a coroutine ends, any values returned by its main function go to the corresponding `resume`:

```
co = coroutine.create(function ()
       return 6, 7
     end)
print(coroutine.resume(co))   --> true  6  7
```

We seldom use all these facilities in the same coroutine, but all of them have their uses.

For those that already know something about coroutines, it is important to clarify some concepts before we go on. Lua offers what I call *asymmetric coroutines*. That means that it has a function to suspend the execution of a coroutine and a different function to resume a suspended coroutine. Some other languages offer *symmetric coroutines*, where there is only one function to transfer control from any coroutine to another.

Some people call asymmetric coroutine *semi-coroutines* (because they are not symmetrical, they are not really *co*). However, other people use the same term *semi-coroutine* to denote a restricted implementation of coroutines, where a coroutine can only suspend its execution when it is not inside any auxiliary function, that is, when it has no pending calls in its control stack. In other words, only the main body of such semi-coroutines can yield. A *generator* in Python is an example of this meaning of semi-coroutines.

Unlike the difference between symmetric and asymmetric coroutines, the difference between coroutines and generators (as presented in Python) is a deep one; generators are simply not powerful enough to implement several interesting constructions that we can write with true coroutines. Lua offers true, asymmetric coroutines. Those that prefer symmetric coroutines can implement them on top of the asymmetric facilities of Lua. It is an easy task. (Basically, each transfer does a yield followed by a resume.)

9.2 Pipes and Filters

One of the most paradigmatic examples of coroutines is in the producer–consumer problem. Let us suppose that we have a function that continually produces values (e.g., reading them from a file) and another function that continually consumes these values (e.g., writing them to another file). Typically, these two functions look like this:

```
function producer ()
  while true do
    local x = io.read()     -- produce new value
    send(x)                 -- send to consumer
  end
end

function consumer ()
  while true do
    local x = receive()        -- receive from producer
    io.write(x, "\n")          -- consume new value
  end
end
```

(In that implementation, both the producer and the consumer run forever. It is an easy task to change them to stop when there is no more data to be handled.) The problem here is how to match `send` with `receive`. It is a typical case of a who-has-the-main-loop problem. Both the producer and the consumer are active, both have their own main loops, and both assume that the other is a callable service. For this particular example, it is easy to change the structure of one of the functions, unrolling its loop and making it a passive agent. However, this change of structure may be far from easy in other real scenarios.

Coroutines provide an ideal tool to match producers and consumers, because a resume–yield pair turns upside-down the typical relationship between caller and callee. When a coroutine calls `yield`, it does not enter into a new function; instead, it returns a pending call (to `resume`). Similarly, a call to `resume` does not start a new function, but returns a call to `yield`. This property is exactly what we need to match a `send` with a `receive` in such a way that each one acts as if it were the master and the other the slave. So, `receive` resumes the producer so that it can produce a new value; and `send` yields the new value back to the consumer:

```
function receive ()
  local status, value = coroutine.resume(producer)
  return value
end

function send (x)
  coroutine.yield(x)
end
```

Of course, the producer must now be a coroutine:

```
producer = coroutine.create(
  function ()
    while true do
    local x = io.read()     -- produce new value
      send(x)
    end
  end)
```

In this design, the program starts calling the consumer. When the consumer needs an item, it resumes the producer, which runs until it has an item to give to the consumer, and then stops until the consumer restarts it again. Therefore, we have what we call a *consumer-driven* design.

We can extend this design with filters, which are tasks that sit between the producer and the consumer doing some kind of transformation in the data. A filter is a consumer and a producer at the same time, so it resumes a producer to get new values and yields the transformed values to a consumer. As a trivial example, we can add to our previous code a filter that inserts a line number at the beginning of each line. The complete code would be like this:

```
function receive (prod)
  local status, value = coroutine.resume(prod)
  return value
end

function send (x)
  coroutine.yield(x)
end
```

```
function producer ()
  return coroutine.create(function ()
    while true do
      local x = io.read()     -- produce new value
      send(x)
    end
  end)
end

function filter (prod)
  return coroutine.create(function ()
    local line = 1
    while true do
      local x = receive(prod)   -- get new value
      x = string.format("%5d %s", line, x)
      send(x)      -- send it to consumer
      line = line + 1
    end
  end)
end

function consumer (prod)
  while true do
    local x = receive(prod)   -- get new value
    io.write(x, "\n")          -- consume new value
  end
end
```

The final bit simply creates the components it needs, connects them, and starts the final consumer:

```
p = producer()
f = filter(p)
consumer(f)
```

Or better yet:

```
consumer(filter(producer()))
```

If you thought about Unix pipes after reading the previous example, you are not alone. After all, coroutines are a kind of (non-preemptive) multithreading. While in pipes each task runs in a separate process, with coroutines each task runs in a separate coroutine. Pipes provide a buffer between the writer (producer) and the reader (consumer) so there is some freedom in their relative speeds. This is important in the context of pipes, because the cost of switching between processes is high. With coroutines, the cost of switching between tasks is much smaller (roughly the same cost of a function call), so the writer and the reader can go hand in hand.

9.3 Coroutines as Iterators

We can see loop iterators as a quite specific example of the producer–consumer pattern. An iterator produces items to be consumed by the loop body. Therefore, it seems appropriate to use coroutines to write iterators. Actually, coroutines provide a powerful tool for this task. Again, the key feature is their ability to turn upside-down the relationship between caller and callee. With this feature, we can write iterators without worrying about how to keep state between successive calls to the iterator.

To illustrate this kind of use, let us write an iterator to traverse all permutations of a given array. It is not an easy task to write directly such iterator, but it is not so difficult to write a recursive function that generates all those permutations. The idea is simple: Put each array element in the last position, in turn, and recursively generate all permutations of the remaining elements. The code is as follows:

```
function permgen (a, n)
  if n == 0 then
    printResult(a)
  else
    for i=1,n do

      -- put i-th element as the last one
      a[n], a[i] = a[i], a[n]

      -- generate all permutations of the other elements
      permgen(a, n - 1)

      -- restore i-th element
      a[n], a[i] = a[i], a[n]

    end
  end
end
```

To see it working, we should define an appropriate `printResult` function and call `permget` with proper arguments:

```
function printResult (a)
  for i,v in ipairs(a) do
    io.write(v, " ")
  end
  io.write("\n")
end

permgen ({1,2,3,4}, 4)
```

After we have the generator ready, it is an automatic task to convert it to an iterator. First, we change `printResult` to `yield`:

```
function permgen (a, n)
  if n == 0 then
    coroutine.yield(a)
  else
  ...
```

Then, we define a factory that arranges for the generator to run inside a coroutine, and then create the iterator function. The iterator simply resumes the coroutine to produce the next permutation:

```
function perm (a)
  local n = table.getn(a)
  local co = coroutine.create(function () permgen(a, n) end)
  return function ()   -- iterator
    local code, res = coroutine.resume(co)
    return res
  end
end
```

With that machinery in place, it is trivial to iterate over all permutations of an array with a **for** statement:

```
for p in perm{"a", "b", "c"} do
  printResult(p)
end
  --> b c a
  --> c b a
  --> c a b
  --> a c b
  --> b a c
  --> a b c
```

The `perm` function uses a common pattern in Lua, which packs a call to resume with its corresponding coroutine inside a function. This pattern is so common that Lua provides a special function for it: `coroutine.wrap`. Like `create`, `wrap` creates a new coroutine. Unlike `create`, `wrap` does not return the coroutine itself; instead, it returns a function that, when called, resumes the coroutine. Unlike the original `resume`, that function does not return an error code as its first result; instead, it raises the error in case of errors. Using `wrap`, we can write `perm` as follows:

```
function perm (a)
  local n = table.getn(a)
  return coroutine.wrap(function () permgen(a, n) end)
end
```

Usually, `coroutine.wrap` is simpler to use than `coroutine.create`. It gives us exactly what we need from a coroutine: a function to resume it. However, it is also less flexible. There is no way to check the status of a coroutine created with `wrap`. Moreover, we cannot check for errors.

9.4 Non-Preemptive Multithreading

As we saw earlier, coroutines are a kind of collaborative multithreading. Each coroutine is equivalent to a thread. A pair yield–resume switches control from one thread to another. However, unlike "real" multithreading, coroutines are non preemptive. While a coroutine is running, it cannot be stopped from the outside. It only suspends execution when it explicitly requests so (through a call to `yield`). For several applications this is not a problem, quite the opposite. Programming is much easier in the absence of preemption. You do not need to be paranoid about synchronization bugs, because all synchronization among threads is explicit in the program. You only have to ensure that a coroutine only yields when it is outside a critical region.

However, with non-preemptive multithreading, whenever any thread calls a blocking operation, the whole program blocks until the operation completes. For most applications, this is an unacceptable behavior, which leads many programmers to disregard coroutines as a real alternative to conventional multithreading. As we will see here, that problem has an interesting (and obvious, with hindsight) solution.

Let us assume a typical multithreading situation: We want to download several remote files through HTTP. Of course, to download several remote files, we must know how to download one remote file. In this example, we will use the *LuaSocket* library, developed by Diego Nehab. To download a file, we must open a connection to its site, send a request to the file, receive the file (in blocks), and close the connection. In Lua, we can write this task as follows. First, we load the LuaSocket library:

```
require "luasocket"
```

Then, we define the host and the file we want to download. In this example, we will download the HTML 3.2 Reference Specification from the World Wide Web Consortium site:

```
host = "www.w3.org"
file = "/TR/REC-html32.html"
```

Then, we open a TCP connection to port 80 (the standard port for HTTP connections) of that site:

```
c = assert(socket.connect(host, 80))
```

The operation returns a connection object, which we use to send the file request:

```
c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")
```

Next, we read the file in blocks of 1 KB, writing each block to the standard output:

```
while true do
  local s, status = c:receive(2^10)
  io.write(s)
  if status == "closed" then break end
end
```

The `receive` method always returns a string with what it read plus another string with the status of the operation. When the host closes the connection we break the receive loop.

Finally, we close the connection:

```
c:close()
```

Now that we know how to download one file, let us return to the problem of downloading several files. The trivial approach is to download one at a time. However, this sequential approach, where we only start reading a file after finishing the previous one, is too slow. When reading a remote file, a program spends most of its time waiting for data to arrive. More specifically, it spends most of its time blocked in the call to `receive`. So, the program could run much faster if it downloaded all files simultaneously. Then, while a connection has no data available, the program can read from another connection. Clearly, coroutines offer a convenient way to structure those simultaneous downloads. We create a new thread for each download task. When a thread has no data available, it yields control to a simple dispatcher, which invokes another thread.

To rewrite the program with coroutines, let us first rewrite the previous download code as a function:

```
function download (host, file)
  local c = assert(socket.connect(host, 80))
  local count = 0    -- counts number of bytes read
  c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")
  while true do
    local s, status = receive(c)
    count = count + string.len(s)
    if status == "closed" then break end
  end
  c:close()
  print(file, count)
end
```

Because we are not interested in the remote file contents, this function only counts the file size, instead of writing the file to the standard output. (With several threads reading several files, the output would intermix all files.) In this new code, we use an auxiliary function (`receive`) to receive data from the connection. In the sequential approach, its code would be like this:

```
function receive (connection)
  return connection:receive(2^10)
end
```

For the concurrent implementation, this function must receive data without blocking. Instead, if there is not enough data available, it yields. The new code is like this:

```
function receive (connection)
  connection:timeout(0)   -- do not block
  local s, status = connection:receive(2^10)
  if status == "timeout" then
    coroutine.yield(connection)
  end
  return s, status
end
```

The call to `timeout(0)` makes any operation over the connection a non-blocking operation. When the operation status is "`timeout`", it means that the operation returned without completion. In this case, the thread yields. The non-false argument passed to `yield` signals to the dispatcher that the thread is still performing its task. (Later we will see another version where the dispatcher needs the timed-out connection.) Notice that, even in case of a timeout, the connection returns what it read until the timeout, so `receive` always returns `s` to its caller.

The next function ensures that each download runs in an individual thread:

```
threads = {}    -- list of all live threads
function get (host, file)
  -- create coroutine
  local co = coroutine.create(function ()
    download(host, file)
  end)
  -- insert it in the list
  table.insert(threads, co)
end
```

The table `threads` keeps a list of all live threads, for the dispatcher.

The dispatcher is simple. It is mainly a loop that goes through all threads, calling one by one. It must also remove from the list the threads that finish their tasks. It stops the loop when there are no more threads to run:

```
function dispatcher ()
  while true do
    local n = table.getn(threads)
    if n == 0 then break end   -- no more threads to run
    for i=1,n do
      local status, res = coroutine.resume(threads[i])
      if not res then    -- thread finished its task?
        table.remove(threads, i)
        break
      end
    end
  end
end
```

Finally, the main program creates the threads it needs and calls the dispatcher. For instance, to download four documents from the W3C site, the main program could be like this:

```
host = "www.w3.org"

get(host, "/TR/html401/html40.txt")
get(host,"/TR/2002/REC-xhtml1-20020801/xhtml1.pdf")
get(host,"/TR/REC-html32.html")
get(host,
    "/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt")

dispatcher()   -- main loop
```

My machine takes six seconds to download those four files using coroutines. With the sequential implementation, it takes more than twice that time (15 seconds).

Despite the speedup, this last implementation is far from optimal. Everything goes fine while at least one thread has something to read. However, when no thread has data to read, the dispatcher does a busy wait, going from thread to thread only to check that they still have no data. As a result, this coroutine implementation uses almost 30 times more CPU than the sequential solution.

To avoid this behavior, we can use the `select` function from LuaSocket. It allows a program to block while waiting for a status change in a group of sockets. The changes in our implementation are small. We only have to change the dispatcher. The new version is like this:

```
function dispatcher ()
  while true do
    local n = table.getn(threads)
    if n == 0 then break end   -- no more threads to run
    local connections = {}
    for i=1,n do
      local status, res = coroutine.resume(threads[i])
      if not res then    -- thread finished its task?
        table.remove(threads, i)
        break
      else    -- timeout
        table.insert(connections, res)
      end
    end
    if table.getn(connections) == n then
      socket.select(connections)
    end
  end
end
```

Along the inner loop, this new dispatcher collects the timed-out connections in table `connections`. Remember that `receive` passes such connections to `yield`;

thus `resume` returns them. When all connections time out, the dispatcher calls `select` to wait for any of those connections to change status. This final implementation runs as fast as the first implementation with coroutines. Moreover, as it does no busy waits, it uses just a little more CPU than the sequential implementation.