Manuscript submitted to **Roy Ratcliffe** *Blog*
Article

Lua Thread Pool

Roy Ratcliffe*

*Correspondence: roy@ratcliffe.me

ABSTRACT Lua co-routines usefully support multi-threading without preemption. This article presents a simple but powerful `thread` module that unmasks some of the hidden subtleties and adds a call-back feature that allows for multiple overlapping responders to threaded results for improved software decomposition.

INTRODUCTION

This article starts with the implementation and works back through the design along with explanations to account for the details, and some examples.

LUA MODULE

Cutting a long story short, the Lua thread-pool module's 'undressed source' lists below. No comments or documentation, just the naked code appears.

Sometimes seeing only the trees proves useful, and quickly lets the developer assess the pros and cons. Software always has positives and negatives; no perfect engineering solution exists except perhaps as myth.

```
local thread = {}

local threadpoolt = {
  __index = {}
}

function threadpoolt.__index:fork(forked)
  local co = coroutine.create(forked)
  self[co] = {}
  return co
end

function threadpoolt.__index:on(co, callback)
  table.insert(self[co], callback)
  return co
end

function threadpoolt.__index:success(co, callback)
  return self:on(co, function(status, ...)
      if status then callback(...) end
    end)
end

function threadpoolt.__index:failure(co, callback)
  return self:on(co, function(status, ...)
      if not status then callback(...) end
    end)
end

local function call(callbacks, ...)
  local indices = {}
  for i, callback in ipairs(callbacks) do
    if not pcall(callback, ...) then
      table.insert(indices, 1, i)
    end
  end
  for _, i in ipairs(indices) do
    table.remove(callbacks, i)
  end
  return ...
end

function threadpoolt.__index:call(co, ...)
  return call(self[co], coroutine.resume(co, ...))
end

function threadpoolt.__index:continue()
  for co, callbacks in pairs(self) do
    call(callbacks, coroutine.resume(co))
  end
end

function threadpoolt.__index:reap()
  for co, _ in pairs(self) do
    if coroutine.status(co) == "dead" then
      self[co] = nil
    end
  end
  return next(self) == nil
end

function thread.pool()
  return setmetatable({}, threadpoolt)
end

return thread
```

Listing complete, super simple. Function `thread.pool` builds a new pool of threads.

THREAD POOL DESIGN

Object design below in Universal Modelling Language (UML).

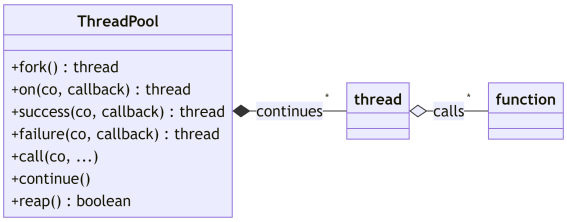

Figure 1: Thread pool class diagram

The order of the methods generally follows the typical usage ordering; that of:

(1) fork first,
(2) set up call-backs,
(3) make the initial call with arguments,
(4) continue and reap.

Notice the class diagram's composition and aggregation association markers. The pool *composes* threads; the call-back functions *aggregate* with those threads. The latter association applies more loosely. A pool's threads cannot exist without the pool but the call-backs can, conceptually speaking.

LUA DETAILS

Some important details require expansion notwithstanding:

- the module definition,
- first use of the `call` method,
- last result by returning not by yielding, and finally
- protected call-back functions.

Local Table for Module

The implementation uses the preferred module definition prologue and epilogue forms. Other dependent modules, as well as the global application sources, can access the module using Lua's `require` as normal, assigning the result locally, using any arbitrary local name. The module names and returns the module table as `thread` internally, rather than use a pseudonym such as `_M`.

If a typical usage applies the statement:

```
local thread = require "thread"
```

Given that, then a new pool constructs using a `thread.pool()` call.

First Call

The first call invokes the call-back functions *before* the first pool continue operation. This important fact requires that responding call-backs need setting up *before* the first call.

Last Result

The last result does not matter if the thread continues indefinitely. It only matters if the thread yields some useful results and subsequently returns, i.e. exits the thread. Such cases should remain aware that the final return counts as a successful continuation; function `coroutine.resume(co)` returns `true` for returning co-routine `co`, in technical terms. Hence the final return triggers the success call-backs.

Success and Failure Call-Back Functions

Methods `on`, `success` and `failure` answer their thread argument in order to allow for chaining. This allows the following chain.

```
pool:call(pool:success(pool:fork(function(s)
        for a in string.gmatch(s, "%a") do
          coroutine.yield(a)
        end
        return "world"
      end), print), "hello")
```

Fork a function as a co-routine; in this contrived example, match and yield letters from string 'hello' then finally resulting in "world" string. Connect a success call-back function, standard-library `print`. Make the first call with arguments, the string to match against.

Call-back functions run as protected calls and disappear from the thread call-back association if and when the call fails with error. Protection is essential; it implies that pool continuation runs without error despite responder errors. But it also implies that the responders cannot themselves yield.

Next Nil

Lua's `#` operator does *not* answer the size of a given table. Rather, it gives the highest numerical index currently present within a given table. Sometimes that result matches the size of the table but sometimes not.

For example,

```
#{"a"}
```

correctly answers 1 as one might expect. However,

```
#{["a"] = 1}
```

unexpectedly answers 0 because there is no "highest numerical index."

Lua does nevertheless let you test for an empty table using `next`; a `nil` answer occurs for empty tables, non-`nil` for non-empty.

CAVEATS AND IMPLEMENTATION DETAILS

Thread pool users *must* create a Lua co-routine using the `fork` method, even though it primarily just creates a

co-routine using `coroutine.create` from the Lua standard library. Forking does however add an extra dimension: it registers the resulting thread with the pool. You cannot bind a call-back to a thread that has *not* registered with the pool; attempting to run `on` with a non-forked thread or a thread created by some other pool will fail. Applications may have a need for more than one pool of threads, one for servicing socket requests, another for periodic continuations for instance; applications might even construct a tree of pools under some scenarios. So call this 'fork only feature' a design constraint limitation.

Internally the module utilises `co` as the symbol for a co-routine, a thread; this approach disambiguates a thread instance from the usage of `thread` as the name of the module. Doing so also immitates the Lua `coroutine` standard library which also employs `co` for thread types.

USAGE

Example usage below.

```
local thread = require "thread"

local pool = thread.pool()

local co = pool:fork(function(a, b)
    for i = a, b - 1 do
      coroutine.yield(i)
    end
    return b
  end)

pool:on(co, function(status, ...)
    error("oops")
  end)

pool:success(co, print)
pool:failure(co, function(...)
    print("failure")
  end)

pool:call(co, 1, 10)

repeat
  pool:continue()
until pool:reap()
```

It outputs 1 through 10 inclusive but some important features require explanation.

- Calling `pool:call(co, 1, 10)` both passes the arguments (1 and 10) and also runs the thread until its first co-routine yield. In short, it starts.
- Returning succeeds for one final resume. Notice the for-loop yields for all except the last. The function *returns* the last result using `return` and not by yielding.
- The call-back function that raises an 'oops' error fails and disappears from the call-back chain.

CONCLUSIONS

Connection from co-routine to call-back is a useful feature though strictly not necessary in many scenarios. Co-routines often yield nothing. Yielding itself is a useful act; it allows other threads to continue. Connecting a sequence of call-back functions lets an application further decompose and simplify arbitrary behaviours—always a good thing. Instead of interlacing behaviours with yields, the application architect can functionally decompose the yielder function from its result responder functions based on success and failure along with the results themselves.