Implementation of Lua coroutines

N. Ramsey and N. Cooper, for COMP 250RTS

November 6, 2017

Instructor's commentary

I think we got most of the value last time when we built the model of thread states and thread execution. The last big question in my own mind—and it's an important one—is to understand exactly what's happening in the C stack (that is, in the run-time system) as all the other stuff is going on. Unfortunately, that understanding would seem to require intensive study of the code, which is not on offer.

Noah and I both found the coroutine code difficult to study. My objection is that most jobs are spread over more than one procedure, and many procedures do more than one job. I also found it difficult to understand all of the representations and their invariants.

For class, we've planned to do three simple comparisons, then declare victory.

What we learned last time

Threads are first-class values and can be found in the heap. A thread may be *running*, *suspended*, *normal*, or *dead* (R, S, N, or D). At any point in execution, exactly one thread is running. Initially it is the *main* thread.

Let us extend the state model by tracking the number of frames on each thread's call stack. A thread is dead if and only if its call stack is empty.

The `resume` and `yield` primitives put threads on a stack for evaluation (hereafter the "thread-evaluation stack"). These threads can *also* be visible from the heap, and additional threads not on the stack may be visible from the heap. The thread-evaluation stack looks like $N_1 \cdots N_k R$, where R is the currently running thread, N_k is the thread that resumed R, N_{k-1} resumed N_k, and so on. Thread N_1 is the main thread, and no thread resumed it. If $k = 0$, there are no normal threads, only the main (running) thread.

Here are some state transitions:

- When R calls a function, it gets an additional frame on its call stack.
- When R returns from a function, it loses a frame from its call stack. If this transition makes the number of frames go to zero, the R becomes D, and the thread N_k becomes the new running thread, receiving R's results as results from `coroutine.resume`.

  If, when R returns from a function $k = 0$, the program terminates.
- When R calls `coroutine.yield`, this action pops the thread-evaluation stack. Thread R transitions to state S. Thread N_k becomes the new running thread, receiving R's arguments to `yield` as results from `resume` (following `true`). Thread N_k transitions to state R.

  If $k = 0$, an error occurs.
- When R calls `coroutine.resume`, its argument must be a thread t in state S; otherwise an error occurs. Because t's state is S, we know t is not on the thread-evaluation stack. Thread R goes into state N. Thread t transitions to state R and is pushed onto the thread-evaluation stack.
  - If t arrived at S via `coroutine.create(f)`, the arguments to `resume` are passed to `f`.
  - If t arrived at S via `coroutine.yield(...)`, the arguments to `resume` become the results returned from `coroutine.yield`.
- When R calls `coroutine.create(f)`, a new thread t is allocated on the heap, and its initial state is S. Nothing happens to the thread-evaluation stack.

Recommended reading

Readings from the last week may be relevant. In addition, please read

- Section 4.7 of the Lua 5.3 Reference Manual, Handling Yields in C

The points covered in this section are arcane, but unless you know about them, you'll have a hard time making sense of the implementation.

If you've taken COMP 105 at Tufts, cast your mind back to the continuation-passing Boolean-formula solver. Function `lua_pcallk` is passing a closure that acts as a continuation. The closure's representation is split over two variables: `k` is the code, and `ctx` is the associated environment. Function `lua_yieldk` is a tail call that passes a continuation to `yield`, which never returns.

Key API functions

Here are some API functions that bear on coroutines:

```
lua_tothread
lua_resume
lua_yield
lua_yieldk
lua_callk
lua_error
```

When you come to class, please bring an understanding of these functions.

(We've chosen to ignore `lua_newthread` and its companion function `luaB_cocreate`, which combine to implement `coroutine.create`. These functions are important, but they appear uninteresting.)

Notes:

- `lua_resume` expects arguments on the register-window stack (`ldo.c` line 648).
- `lua_yield` is a control operator, and as such, it is to be used in tail position only—if you need to control what happens on the subsequent `resume`, you pass a continuation. (The API documentation below says, "usually, this function does not return.")

Official documentation

API function `lua_newthread`

```
lua_State *lua_newthread (lua_State *L);
```

Creates a new thread, pushes it on the stack, and returns a pointer to a **lua_State** that represents this new thread. The new thread returned by this function shares with the original thread its global environment, but has an independent execution stack. There is no explicit function to close or to destroy a thread. Threads are subject to garbage collection, like any Lua object.

API function `lua_tothread`

```
lua_State *lua_tothread (lua_State *L, int index);
```

Converts the value at the given index to a Lua thread (represented as `lua\_State\*`). This value must be a thread; otherwise, the function returns `NULL`.

API function `lua_resume`

```
int lua_resume (lua_State *L, lua_State *from,
                int nargs);
```

Starts and resumes a coroutine in the given thread `L`. To start a coroutine, you push onto the thread stack the main function plus any arguments; then you call **lua_resume**, with `nargs` being the number of arguments. This call returns when the coroutine suspends or finishes its execution. When it returns, the stack contains all values passed to **lua_yield**, or all values returned by the body function. **lua_resume** returns **LUA_YIELD** if the coroutine yields, **LUA_OK** if the coroutine finishes its execution without errors, or an error code in case of errors (see **lua_pcall**). In case of errors, the stack is not unwound, so you can use the debug API over it. The error message is on the top of the stack. To resume a coroutine, you remove any results from the last **lua_yield**, put on its stack only the values to be passed as results from `yield`, and then call **lua_resume**. The parameter `from` represents the coroutine that is resuming `L`. If there is no such coroutine, this parameter can be `NULL`.

API function `lua_yield`

```
int lua_yield (lua_State *L, int nresults);
```

This function is equivalent to **lua_yieldk**, but it has no continuation (see **continuations**). Therefore, when the thread resumes, it continues the function that called the function calling `lua\_yield`.

API function `lua_yieldk`

```
int lua_yieldk (lua_State *L,
                int nresults,
                lua_KContext ctx,
                lua_KFunction k);
```

Yields a coroutine (thread). When a C function calls **lua_yieldk**, the running coroutine suspends its execution, and the call to **lua_resume** that started this coroutine returns. The parameter `nresults` is the number of values from the stack that will be passed as results to **lua_resume**. When the coroutine is resumed again, Lua calls the given continuation function `k` to continue the execution of the C function that yielded (see **continuations**). This continuation function receives the same stack from the previous function, with the `n` results removed and replaced by the arguments passed to **lua_resume**. Moreover, the continuation function receives the value `ctx` that was passed to **lua_yieldk**. Usually, this function does not return; when the coroutine eventually resumes, it continues executing the continuation function. However, there is one special case, which is when this function is called from inside a line hook (see **debugI**). In that case, `lua\_yieldk` should be called with no continuation (probably in the form of **lua_yield**), and the hook should return immediately after the call. Lua will yield and, when the coroutine resumes again, it will continue the normal execution of the (Lua) function that triggered the hook. This function can raise an error if it is called from a thread with a pending C call with no continuation

function, or it is called from a thread that is not running inside a resume (e.g., the main thread).

API function `lua_callk`

```
void lua_callk (lua_State *L,
                int nargs,
                int nresults,
                lua_KContext ctx,
                lua_KFunction k);
```

This function behaves exactly like **lua_call**, but allows the called function to yield (see **continuations**).

API function `lua_error`

```
int lua_error (lua_State *L);
```

Generates a Lua error, using the value at the top of the stack as the error object. This function does a long jump, and therefore never returns (see **luaL_error**).

API function `lua_pcall`

```
int lua_pcall (lua_State *L, int nargs,
               int nresults, int msgh);
```

Calls a function in protected mode. Both `nargs` and `nresults` have the same meaning as in **lua_call**. If there are no errors during the call, **lua_pcall** behaves exactly like **lua_call**. However, if there is any error, **lua_pcall** catches it, pushes a single value on the stack (the error message), and returns an error code. Like **lua_call**, **lua_pcall** always removes the function and its arguments from the stack. If `msgh` is 0, then the error message returned on the stack is exactly the original error message. Otherwise, `msgh` is the stack index of a *message handler*. (This index cannot be a pseudo-index.) In case of runtime errors, this function will be called with the error message and its return value will be the message returned on the stack by **lua_pcall**. Typically, the message handler is used to add more debug information to the error message, such as a stack traceback. Such information cannot be gathered after the return of **lua_pcall**, since by then the stack has unwound. The **lua_pcall** function returns one of the following constants (defined in `lua.h`):

- `LUA_OK` **(0):** success.
- `LUA_ERRRUN`**:** a runtime error.
- `LUA_ERRMEM`**:** memory allocation error. For such errors, Lua does not call the message handler.
- `LUA_ERRERR`**:** error while running the message handler.
- `LUA_ERRGCMM`**:** error while running a `__gc` metamethod. (This error typically has no relation with the function being called.)

First round of discussion questions:

Here are two questions that can be addressed just based on the public interfaces.

(1) What do `lua_resume` and `lua_pcall` have in common? Are the differences interesting?

(2) What do `lua_yield` and `lua_error` have in common? Are the differences interesting?

Guide to studying the internals

Here are some recommendations about what to study (and what not to study) in the implementation.

The state of a Lua thread

The representation of the state of a Lua thread is shocking (Figure 1).

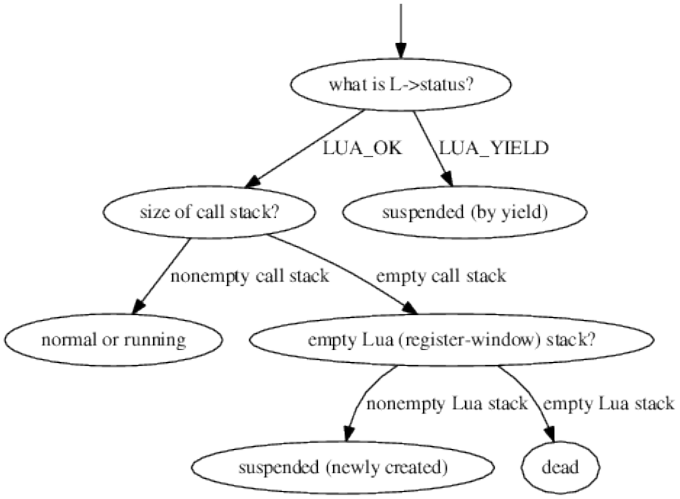

Figure 1: Decision tree for thread status

Functions named `luaB_*`

Functions that begin with the `luaB_` prefix are part of Lua's "base library." Unlike the other prefixed functions, they don't appear in any `.h` file. They are mostly found in `lbaselib.c`, but the coroutine ones are in `lcorolib.c`. *All* these functions use only the public interfaces; they are not aware of any of the internals.

Things that are safe to ignore

Please ignore `lua_lock` and `lua_unlock`.

Please ignore `luai_userstateresume`, along with any other function whose name begins with `luai_user`. (These functions default to no-ops. They mark state transitions, and they enable a client to extend the thread abstraction.)

As noted above, we're choosing to ignore `lua_newthread`, plus the implementations of `coroutine.create`, `coroutine.resume`, and `coroutine.yield`. These "base library" functions are nice illustrations of how to use the Lua API to create new Lua functions, but the story of coroutines is revealed primarily through the API functions and the internals.

Ignore *hooks*. That is, whenever you're reading code, assume you take the path that says "not in a hook."

Things to take on faith

Macro `errorstatus` does what you think. Macros `savestack` and `restorestack` convert locations (in the register-window stack) to and from integers. Macro `savestack` converts a pointer to an array index, and `restorestack` converts back. The "saved" index is invariant under changes in the stack—which would be necessary if the stack needs to grow. The original pointer is not.

Things to review

Please review functions `luaD_precall`, `luaD_poscall`, and `luaD_rawrunprotected`.

I think we had a quick look at internal function `luaD_throw` (line 110 of file `ldo.c`). If not, it just throws to the current exception handler. Most of the code is error checking (in the event there is no handler).

Core things to study

Internal function `unroll` (`ldo.c`, line 540):

- The last operation of `resume`, it continues the execution of a suspended thread that has just transitioned to *running*. Function `unroll` seems to do the actual running.

Internal function `resume` (file `ldo.c` line 612):

- The meat of `coroutine.resume`, which looks as much like a call as possible
- Line 630 restores `ci->func`, which determines what part of the register-window stack the current activation points to. It corresponds to the save on line 705.

API function `lua_resume` (file `ldo.c` line 648):

- Like all API functions, it does a good deal of checking.
- Uses internal functions `resume` and `unroll` to do its dirty work.

API function `lua_yieldk` (`ldo.c` line 692), which is also `yield` (with no continuation):

- When you sweep away all the checking, the key parts are the updates to `L->status`, `ci->extra`, and `ci->u.c` (the continuation, if any), followed by `luaD_throw`.

Other things to study

`recover` (`ldo.c` line 581): recovers from an error in a coroutine.

Agenda for class

Technical agenda

(3) Compare `lua_yieldk` with `lua_resume` side by side.
   (a) Match up the matching parts.
   (b) Explain the parts that don't match.

(4) Compare `lua_yieldk` and `lua_error` side by side. Identify similarities and differences.

(5) Compare `lua_resume` and `lua_pcallk` side by side. Identify similarities and differences.

Pedagogical agenda

(6) What did we learn from looking at the code?

(7) In light of our experience, how do we want to handle the Erlang case study?