Implementing Lua Coroutines Using Semaphores

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

November 29, 2017

Introduction

This class has two goals:

- Deepen our understanding of Lua's coroutines
- Implement something more serious using threads with locks and shared memory

Locks and shared memory provide my least favorite model of concurrency—but they're in the hardware, and they are the foundation of most better models.

Recommended reading

All in Birrell (1989), "An Introduction to Programming with Threads." We'll focus on locks (there called "mutexes") and *condition variables.* If you're already familiar with this style of programming, pages 3 to 6 and 13 to 14 are probably sufficient for a refresher. More detailed recommendations follow.

- Pages 1 to 3 (the first two sections) provide context for the tutorial. They are optional.
- Pages 3 to 6 describe the basic primitives in the SRC thread facility: their types and their (informal) dynamic semantics. You will need to get solid on these primitives. Notes:
  - The key novelty here is the `Wait` primitive, which atomically unlocks a mutex and blocks on a condition variable.
  - The `LOCK m DO ... END` construct is a reliable way of using `p` and `v` correctly. It ensures that `v` is called on exit even if the exit is caused by an exception or by other abnormal control flow.
  - Type `REFANY` is analogous to C's `void *`.
- Pages 6 and 7 present "alerts," which are a nicely integrated extension to the standard model implemented by Cormack, but we're going to skip them.
- Pages 7 to 9 present the basics of programming with `LOCK`. Master the sections on "unprotected data" and "invariants."
- Skim or skip the sections on cheating, deadlocks, and lock conflicts on pages 9 to 12.
- Alert yourself to the issue of "releasing the mutex with in a LOCK clause" on pages 12 to 13.
- Pages 13 and 14 present the main novelty of this tutorial: condition variables. Master them. The subtleties on pages 14 to 21 can be skipped—but do read the two paragraphs headed "complexity" at the bottom of page 19.
- Pages 21 to 25 talk about where to find parallelism. For us, the only first three paragraphs are of any interest, and the main point is this:

  > *You will find no need to use older schemes for asynchronous operation (such as interrupts, Unix signals or VMS AST's). If you don't want to wait for the result of a device interaction, invoke it in a separate thread. If you want to have multiple device requests outstanding simultaneously, invoke them in multiple threads. If your operating system still delivers some asynchronous events through these older mechanisms, the runtime library supporting your threads facility should convert them into more appropriate mechanisms. See, for example, the design of the Topaz system calls (11) or the exception and trapping machinery included with Sun's lightweight process library (14,15).*

  That "runtime library" is us.

There's plenty else of interest in the tutorial, but if you come to class prepare to program with locks and condition variables, that will be all you need.

Our model of Lua coroutines (from Nov 6)

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.

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.

Class exercise

(1) Using locks and condition variables, design and implement `coroutine.create`, `coroutine.yield`, and `coroutine.resume`.

  (a) Design a preliminary data structure to represent a coroutine. This data structure will correspond roughly to Erlang's "process-control block." Start with

  - A value of type `Thread` (created with `Fork`)
  - Whatever representation you feel best represents the state of a coroutine and its relationships to other coroutines.

  (b) Complete your design by considering concurrency. Identify data structures or invariants that need to be protected by a lock, operations that need to execute atomically, and communications between threads that need to be synchronized. (At minimum, you will need the capability of synchronizing threads at `coroutine.yield` and `coroutine.resume`.)

  - Extend your process-control block with whatever variables of type `Mutex` and `Condition` you feel are needed to manage synchronization and concurrent execution.

  (c) Write code for each of the three coroutine operations. To simplify matters, assume that `coroutine.yield` and `coroutine.resume` each pass and return exactly one value, of Modula-2+ type `Value`.

  You will rely most on primitive operations `Fork`, `Wait`, and `Signal`, plus the `LOCK m DO ... END` form. As Birrell advises, avoid the explicit `Acquire` and `Release` (`p` and `v`).

(2) Birrell's threads provide true parallelism—unless it is blocked on a mutex or condition variable, a thread can run at any time. Lua's coroutines, by contrast, are highly constrained: at most one is ever running,[1] and they obey a strict LIFO discipline.

  (a) Verify that the coroutines in your implementation behave the same way as Lua coroutines: at most one is ever running, and when one coroutine yields, control is always transferred in the expected way.

[1] If no coroutine is running, then the main thread must be running.