Assignment 1: RPC Library

Due: Wednesday, October 4, 2017 at 4:20pm

Overview

Serialization, or the conversion of data into bytestreams and back, is an eminently useful tool in data processing and remote communication. Formats like JSON, Protobuf, and pickle are commonplace in web requests, data pipelines, databases, and so on. However, serialization is notoriously difficult to attain in statically typed, non-reflective languages like C++. In this assignment, we will demonstrate the power and flexibility of Lua by implementing a small library for serializing Lua datatypes.

To put your implementation into practice, you will also write a library to do inter-process remote procedure calls (RPC). This library will take existing class-like interfaces and create a new implementation with an identical interface but backed by a separate process, permitting concurrent evaluation of Lua functions.

Setup

Assignment 1 lives at `/afs/ir/class/cs242/assignments/assign1`. Copy `assign1` into your current directory by logging into Rice and executing

```
cp -r /afs/ir/class/cs242/assignments/assign1 assign1
```

Alternatively, you can use `scp` and copy it into your local machine.

Submitting

You must implement the three parts of the RPC library in `starter/rpc.lua`:

- `serialize`
- `deserialize`
- `rpcify`

When you're finished, make sure to `scp` the `assign1` directory to Rice, if you worked on your local machine. When logged onto Rice, submit by first navigating to the root of the assign1 directory (you should see the directories `starter` and `common`). From there, execute the command

```
python /afs/ir/class/cs242/bin/submit.py 1
```

In order to verify that your assignment was submitted, execute

```
ls /afs/ir/class/cs242/submissions/assign1/<SUnet ID>
```

You should see the timestamps of your submissions.

Part 1: Serialization

First, you are going to implement the `serialize` and `deserialize` functions. These functions are conceptually simple: `serialize` takes as input a Lua value and returns a string, and `deserialize` takes a string and returns a Lua value. These two functions should fulfill the property that for all serializable inputs `t`, `deserialize(serialize(t)) == t`.

> *Note:* here, equality (or `==`) has the same semantics as the Lua equality test for all serializable values *except* tables. In Lua, table equality is defined by their location in memory, not their contents, so we instead use a *structural* definition of equality between tables. That is, two tables `t1 == t2` iff they have the same keys mapping to the same values. See `table.equals` in `common/tests.lua` for a codification of this idea.

Not all Lua values can be easily serialized like closures or pointers to C values, so your functions only need to be defined for *serializable* Lua values. A serializable Lua value is either a number, a string, a boolean, nil, or a table containing both keys and values that are serializable Lua values. For example:

```
-- serializable values
1
3.14
"hello"
true
nil
{foo = "bar", [1] = {["two"] = 2}}
{1, 2, nil, false}

-- non-serializable values
function() print 'hi' end
{[function() print 'hi' end] = 0}
"(parenthesis string)"
```

If your `serialize` receives an non-serializable Lua value or if your `deserialize` receives an corrupted serialized string, you can do absolutely whatever you want. We won't test those cases.

It's entirely up to you how to implement your serialization—that's part of the fun! However, it should be from scratch. Don't find a JSON implementation online and copy it into your code. To simplify your life (the goal is not to learn how to make an amazingly robust parser), you may assume that any string values you are asked to serialize will never contain parentheses (neither `(` nor `)`) in the string, so you can more easily use parentheses as delimeters in your serialized string.

To test your code, run `lua common/tests.lua` from the assignment directory. We recommend adding more tests to the `serialize_tests` function to thoroughly test your functions!

Some useful tips:

- The `type()` function will give you the type of a Lua value.
- Look at the Lua string library for tools on generating and parsing strings. You may specifically be interested in `string.format` and `string.gmatch`.
- We have provided you a few helper functions in `common/util.lua`. Check them out!

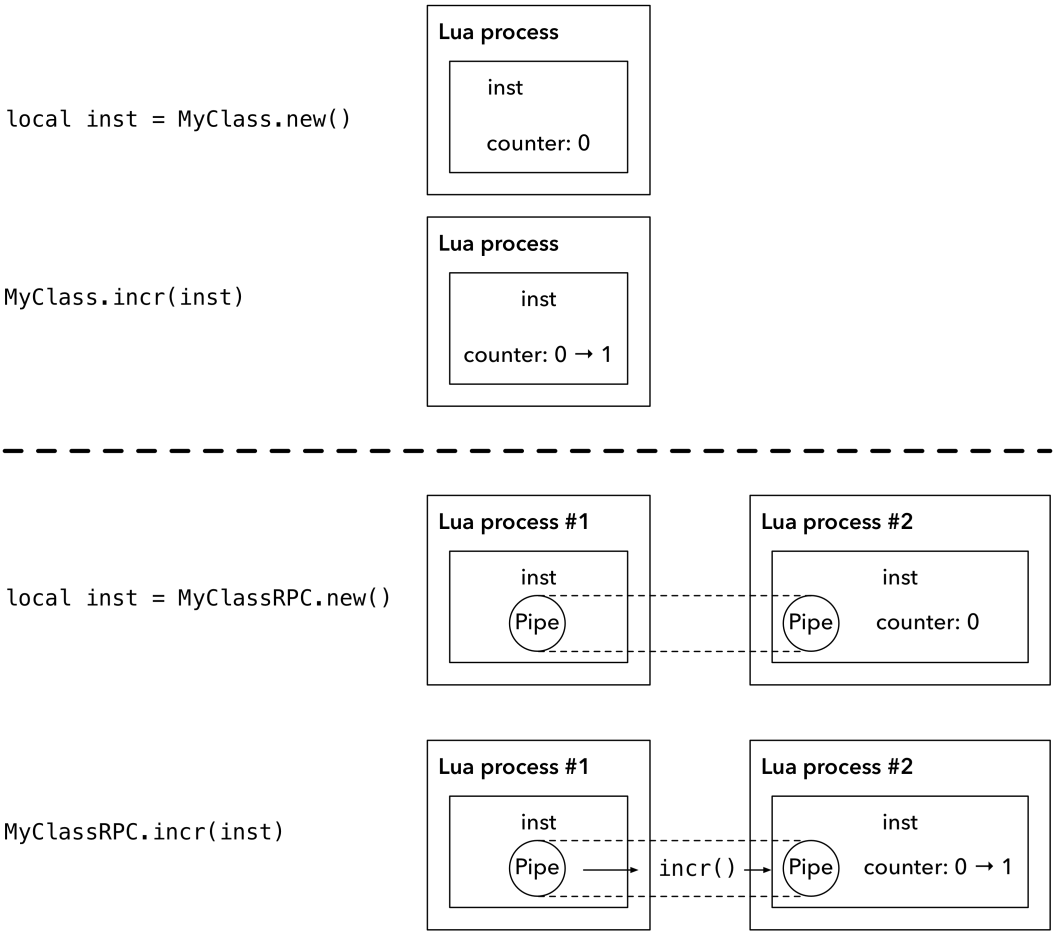
Part 2: RPC

Now, you are going to implement a function `rpcify` that takes in a class and returns a new class which represents the RPC-ified version of the original. Here's a simple example:

```
local MyClass = {}

function MyClass.new()
    return {counter = 0}
end

function MyClass.incr(t)
    t.counter = t.counter + 1
    return t.counter
end

local normal_inst = MyClass.new()
assert(MyClass.incr(normal_inst) == 1)

local MyClassRPC = rpc.rpcify(MyClass)

local remote_inst = MyClassRPC.new()
assert(MyClassRPC.incr(remote_inst) == 1) -- same as before
```

```lua
local future = MyClassRPC.incr_async(remote_inst)
assert(future() == 2) -- rpcify adds asynchronous versions of each method

MyClassRPC.exit(remote_inst) -- it also adds an exit method to cleanup the process
```

Here, a "class" is essentially just a table with function values. We will discuss a more refined notion of classes next Monday, but this is our definition for now. Here is an illustration of a subset of the above process.



Specification

On a high level, the way our RPC should work is that we use operating system *processes* to parallelize computation. Invoking `new()` on an RPC-ified class forks off a new Lua process (child) which runs independently of the original (parent). The parent invokes methods on the child by passing serialized messages over an input pipe, and receives serialized results on an output pipe. Within the parent process, the instance handle represents a pointer to the child process.

More formally:

- `rpcify` takes a table `Class` with string keys mapping to function values. The table `class` will have a function `new(...)` that returns a table, and `Class` will not have an `exit` key. Every other function in `Class` takes a class instance returned by `new` as the first argument, and the remaining arguments and return values are serializable.
- `rpcify` returns a table `NewClass` that satisfies the following properties:
  - `NewClass` has a function `new` that forks off a child process and returns to the parent a handle `inst` (a proxy) to the child object. You may decide the representation of `inst` (it should probably hold at least a pipe for communication).
  - For each key `k` in `Class`, `NewClass` contains two keys `NewClass.k(inst, ...)` and `NewClass.k_async(inst, ...)` that invoke the method `k` on the child process. In the synchronous case, `NewClass.k(inst, ...)` will block until the process returns a result. In the asynchronous case, `NewClass.k_async(inst, ...)` will return a function that takes no arguments, and when called will block until the method `k` returns.
  - `NewClass` contains a method `NewClass.exit(inst)` that, when called, will signal the child process to exit and `posix.wait(...)` on the child until it dies.
  - For simplicity, your implementation may assume that a given instance will never have more than one call, synchronous or asynchronous, active to it from the parent process at any given time. For example, you do not need to support the parent

calling two functions asynchronously and then retrieving both results.

Foundations

To achieve the above specification, we will need a way to create new Lua processes and communicate between them. This involves two core utilities provided by your operating system, wrapped in the luaposix library:

1. **fork**: invoking this function causes a new child process to fork off of the existing one, copying the entire contents of memory from the parent into the child. After the fork, both processes contain the same environment, state, variables, and so on–the only difference is that the `fork` function returns a value of 0 in the child process, and returns the child's process ID in the parent. For example:

```
local posix = require("posix")

local some_var = 1
local pid = posix.fork()
if pid == 0 then
    print("I am the child, and some_var is: ", some_var)
    os.exit()
else
    print("I am the master, my child is ", pid, " and some_var is: ", some_var)
    posix.wait(pid) -- Hangs until child process exits
end
```

   Both child and parent will print the same value (1) for `some_var` because they both have access to the pre-fork environment. This mechanism will allow your RPC-ified `new` function to create a new process and share values like a pipe (below) between them.

   > *Note*: when the child process is done with its computations, you should call `os.exit()` to make sure that no other code is executed from the child process.

2. **Pipes**: pipes allow data to pass between OS processes. They have a POSIX implementation, but we've abstracted that away from you into a `Pipe` class. Pipes should be treated as unidirectional–one side writes the input, and the other side reads the output. For example:

```
local util = require("common.util")
local posix = require("posix")
local Pipe = util.Pipe

local pipe = Pipe.new()
local pid = posix.fork()

if pid == 0 then
   -- Child process
   Pipe.write(pipe, "I am the child!")
   os.exit()
else
   -- Parent process
   print("Child says: " .. Pipe.read(pipe))
   posix.wait(pid)
end
```

   The `write` function takes a string of arbitrary length and writes it to the pipe, returning immediately. The `read` function blocks until a string is written to the pipe being read from. This mechanism will allow your parent process to communicate with your RPC-ified instance's forked process.

   > *Note*: you may assume serialized messages you send over pipes will never be larger than 1024 bytes, and subsequently a) all writes and reads are performed atomically, and b) a message will never need to be split up into parts.

Additionally, you'll need one piece of Lua magic. You will need to pass arbitrary argument lists from the parent to the child. To access all the arguments to a function, you can use the ellipsis syntax:

```
local function return_args(...)
    return {...}
end

local t = return_args(1, "foo")
```

```
print(t[1], t[2]) -- prints 1, "foo"
print(unpack(t))  -- prints the same thing
```

The ellipsis represents the the list of arguments given to the function. You can explicitly turn it into a list by wrapping it with a table (`{...}`). You can turn a table back into an argument list by using the built-in `unpack` function as shown above.

Getting started

If you don't have any ideas on how to get started after reading the documentation above, here's a recommended exercise to understand the core RPC ideas of this section.

```
local util = require("common.util")
local posix = require("posix")
local Pipe = util.Pipe

local in_pipe = Pipe.new()
local out_pipe = Pipe.new()

function child()
  local t = {
    counter = 0,
    incr = function(self)
      self.counter = self.counter + 1
    end,
    value = function(self)
      return self.counter
    end
  }

  -- TODO: What to do in the child? Fill this in yourself.
end

function parent(pid)
  Pipe.write(in_pipe, "incr")
  Pipe.read(out_pipe) -- return is nil, don't care

  Pipe.write(in_pipe, "value")
  print(Pipe.read(out_pipe)) -- should be 1

  Pipe.write(in_pipe, "exit") -- child should exit now
  posix.wait(pid)
end

local pid = posix.fork()
if pid == 0 then
  child()
else
  parent(pid)
end
```

Read through the code to understand what it's trying to do—we want to send commands to a table over a pipe and read results on another pipe. Think about how you would implement the `child` function in this case. Once you've done that, you'll want to think about how to make this solution *generic* over any kind of input class (or any such table `t`).

Testing

To test your code, it's the same as before: run `lua common/tests.lua` . Look for the `rpc_tests` function.