Discussion questions for Lua call stacks

N. Ramsey, for COMP 250RTS

October 18, 2017

Overview of function prefixes and header files

Table 1: Internal function prefixes and header files

Prefix	Header	Abstraction
`luaL_`	`lauxlib.h`	Auxiliary library (extends public API)
`luaK_`	`lcode.h`	Code generator
N/A	`lctype.h`	Replacement for `ctype.h`
`luaG_`	`ldebug.h`	Debug hooks?
`luaD_`	`ldo.h`	Calls; register-window stack
`luaF_`	`lfunc.h`	Closures and free varibles (“upvalues”)
`luaC_`	`lgc.h`	Garbage collection
`luaX_`	`llex.h`	Lexical analysis
N/A	`llimits.h`	Sizes, limits, numeric-operation macros
`luaM_`	`lmem.h`	Allocation
`luaO_`	`lobject.h`	Key representations: Values, type tags, conversions
N/A	`lopcodes.h`	Instruction decoding; opcode table
N/A	`lparser.h`	Parser; classification of names
`luaE_`	`lstate.h`	Interpreter state; call stack
`luaS_`	`lstring.h`	Strings
`luaH_`	`ltable.h`	(Hash) tables
`luaT_`	`ltm.h`	Tag methods (metamethods)
`lua_`	`lua.h`	Public (C) API
N/A	`luaconf.h`	Configuration for portability (uninteresting)
`luaopen_`	`lualib.h`	Access to standard libraries
`luaV_`	`lvm.h`	Bytecode interpreter & operations
`luaZ_`	`lzio.h`	Buffered streams

Documentation to read

Before tackling any code, read this documentation:

- Lua manual sections 4.1 to 4.3 (the stack)
- The section on Lua Stack and Registers in the Lua 5.3 Bytecode Reference
- API documentation for these functions:
  - `lua_call`
  - `lua_gettop`
  - `lua_remove`
  - `lua_pushvalue`
  - `lua_tolstring`
  - `lua_pcall`

Code to read

Note: When reading source code, you may want to use the Unix "tags" facility, which gives you a fast way to jump to the definition of any identifier. If you use Emacs or vi, look up the Unix commands `etags` or `ctags`.

On line 160 of header file `lstate.h`, understand these fields of the `struct lua_State`:

- Fields `top`, `stack_last`, and `stack` (type `StkId` is `TValue *`)
- Fields `ci` and `nci`
- Field `base_ci`
- Field `errorJmp`
- Field `nCcalls`

On line 65 of header file `lstate.h`, understand the `CallInfo` structure. Ignore fields having to do with yields.

On line 97 of `lapi.c`, read the implementation of `lua_checkstack`. Note differences between `ci->top` and `L->top`.

On line 60 of `lapi.c`, read the first two cases in the implementation of `index2addr`.

On line 1130 of `lvm.c`, read the implementation of the `OP_CALL` opcode, and also the implementation of `luaD_precall` on line 413 of `ldo.c`. Be prepared to compare the two main cases of `luaD_precall`. Ignore the "hook" code.

Finally, the implementation of `pcall`:

- Starting on line 931 of `lapi.c`, read the implementation of `lua_pcallk`. Focus on the special case of `lua_pcall`, in which `ctx` is 0 and `k` is `NULL`. (Note: functions `savestack` and `restorestack` convert a stack index to and from an integer, which is then resilient to movement (reallocation) of the stack.)

- Notice function `f_call`; it uses `luaD_callnoyield` and `luaD_call` as a roundabout way of getting to `luaD_precall` and `luaV_execute`.
- On line 721 of `ldo.c`, read the implementation of `luaD_pcall`. Focus on the stack unwinding. Don't get distracted by `luaD_shrinkstack`.
- On line 136 of `ldo.c`, read the implementation of `luaD_rawrunprotected`. The `LUAI_TRY` macro calls `setjmp`, and it executes the body if `setjmp` returns 0, which happens on the first trip though. The outcome of the body is represented by the contents of field `lj.status`. During the execution of the protected call, this field is the same as `L->errorJmp->status`, which may be set to an error code by `luaD_throw` (just above). The final status is returned from `luaD_rawrunprotected`.

Discussion questions

(1) How is the Lua call stack represented?

(2) How does it present an interleaving of Lua activations with C activations? What's actually happening with the C stack?

(3) How does the Lua call stack work with the register-window stack (also known as "the Lua stack")?

(4) In the virtual machine, how does the `CALL` instruction work?

(5) In the API, how does function `lua_call` work?

(6) How is the Lua primitive `pcall` implemented?