

www.Mcours.com

Site N°1 des Cours et Exercices Email: contact@mcours.com



LuaTask 1.6 Manual

Introduction

Who has to read this

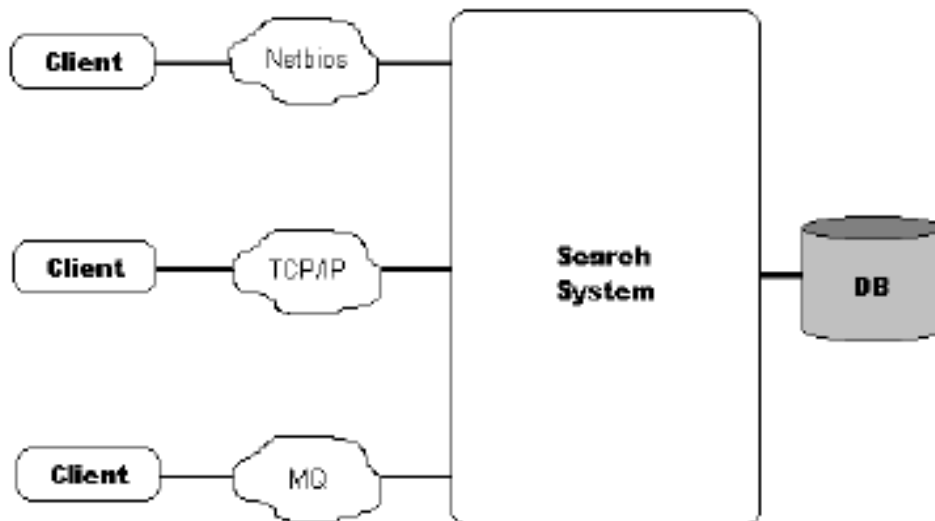
Target audience is the programmer wishing to have multiple Lua universes, each running into a independent OS thread.

The name

We choose the "task" name to avoid confusion with "lua threads".

The idea

Imagine a data base search system, waiting for requests from clients with different communication mechanism: Netbios, TCP/IP and Websphere MQ:

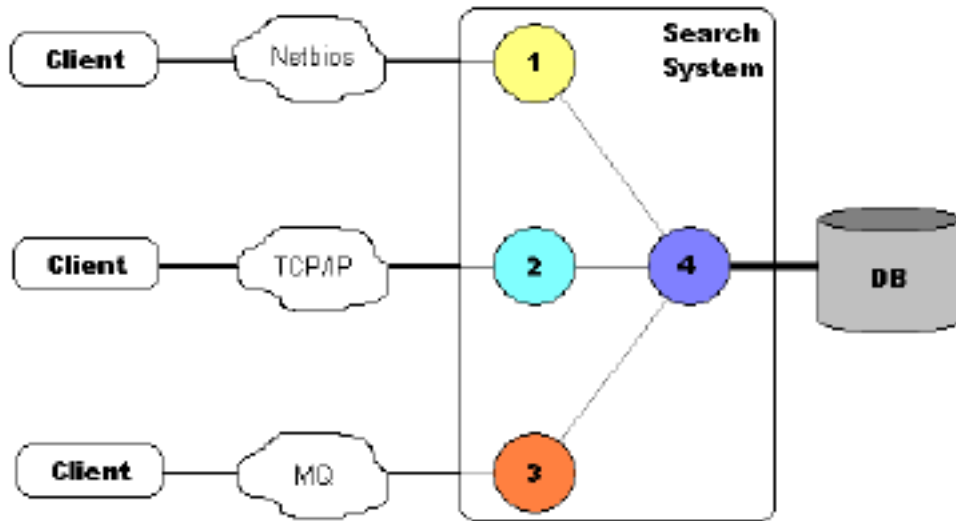


We will assume that the Data Base Server outperform our needs.

Now, we can identify at least four specific functions inside the system:

1. Netbios communications.
2. TCP/IP communications.
3. MQ communications.
4. Search and retrieval.

Let me redraw the previous figure:

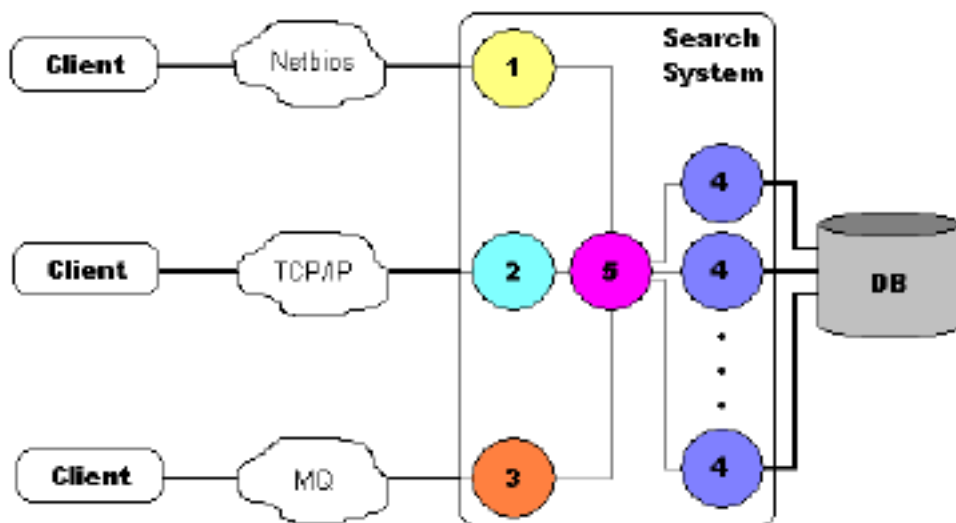


With a moderated load, our system can do functions 1 to 4 in a sequential manner.

But when clients number grows, we must try to overlap unrelated functions in order to speed up the whole system.

LuaTask helps you to accomplish that by using multiple threads of execution.

Dealing with a heavy load...



adds some complexity and, probably, a new task:

5. The dispatcher.

Tasks 1 to 5 run independently and communicate each other through message queues.

Using LuaTask, it is possible the following sequence of events:

1. A communication task receives a message from a client.
2. It passes the message to the dispatcher.
3. The dispatcher chooses a search task, and puts the message in its queue.
4. The search task does its job and sends the response directly to the communication task.
5. The communication task sends the response to the client.

Each task has a message queue, implemented as a memory fifo linked list.

Each entry in a message queue has a dynamically allocated data area, and a 32 bit general purpose number.

All this can be easily programmed in plain Lua, LuaTask and libraries like LuaSocket, LuaSQL, etc.

Building

Step 1

Expand src directory from tgz inside src directory of your Lua installation.

Step 2

Win32

You must select the threading support:

1. define NATV_WIN32 : Original code without cancellation.
2. not define NATV_WIN32 : Pthreads-Win32 code dependent. (You must have Pthreads-Win32 SNAPSHOT 2004-06-22 or later)

Static

Adapt your LibLuaLib.dsp using build/win32/static/LibLuaLib.dsp as an example.

Build lua.exe

Loadable module

Put build/win32/module/task.dsp inside src/LuaTask of your installation.

Add it to Lua.dsw

Build task.dll

Linux/BSD

Static

Adapt your config using build/ix/static/config as an example.

Adapt your src/lib/Makefile using build/ix/static/Makefile as an example.

Build lua binary.

Loadable module

Adapt your config using build/ix/module/config as an example.

Put build/ix/module/Makefile inside src/LuaTask of your installation.

Build libtask.so

Programming

How to use it

If you statically linked LuaTask, you must put a call to `luaopen_task()` after calling to `lua_open()` in your

program. This is the only thing to code in C language.

If you are using the LuaTask dynamic library, you must include the following code in the main task:

```
require 'task'
```

Initialization code inside LuaTask creates the global tasks list, gets a thread specific key, and creates the

"task" namespace.

Now, you can use the functions inside "task" namespace to manipulate tasks and messages.

Look at the "LuaTask Reference Guide" for functions syntax.

A very simple example with two tasks

1. A main task showing a prompt and getting a command.
2. A secondary echo task.

The main task (t1.lua):

```
require 'task'

local myid = task.id() -- gets the current task id

local tsk, err = task.create( 't2.lua', { myid} ) -- creates a new task

while true do

    io.stdout:write( '\necho> ' ) -- shows prompt

    local cmd = io.stdin.read( '*1' ) -- gets command

    if cmd == 'quit' then

        break -- if command is quit, terminates

    else

        task.post( tsk, cmd, 0 ) -- sends command text to echo task

        local buf, flags, rc = task.receive( -1 ) -- waits for answer

        io.stdout:write( buf ) -- shows answer

    end

end

task.post( tsk, '', 1 ) -- sends dummy message with "1" as stop flag

task.receive( 1000 ) -- waits (1 sec) for stop acknowledge from echo task
```

The "echo" task (t2.lua):

```
local main_id = arg[1] -- gets the main task id from arg[1]
while true do
    local buf, flags, rc = task.receive( -1) -- waits for message
    if flags == 1 then
        break -- if flags is stop, terminates
    else
        task.post( main_id, 'Echo: {' .. buf .. '}', 0) -- sends echo
    end
end
task.post( main_id, '', 1) -- sends acknowledge for stop message
```

The data base search system example

1. A main task dispatching requests from input tasks to worker tasks.
2. A Netbios input task.
3. A TCP/IP input task.
4. A MQ input task.
5. Five SQL worker tasks.

The Dispatcher (dispatcher.lua):

```
require 'task'

-- sets number of workers
local workers = 5

-- saves task id
local me = task.id()

-- creates workers task id table
local WTable = {}

-- starts workers
for w = 1, workers do
    WTable[w] = task.create( 'worker.lua', { me} )
end

-- starts input tasks
task.create( 'netbios.lua', { me} )
task.create( 'mq.lua', { me} )
task.create( 'tcpip.lua', { me} )

local w = 0

while true do
    -- receives request
    local request, flags, err = task.receive( -1)

    -- does a simple round-robin dispatching
    if w == workers then
        w=1
    else
        w=w+1
    end

    -- forwards the request to selected worker
    task.post( WTable[w], request, flags)
end
```


The Netbios input task (netbios.lua):

```
-- The Netbios interface has been simplified a lot
-- Session startup and shutdown are hidden.
-- Sorry, we'll use polling for simplicity
-- Adds a Netbios name
NB_AddName( 'MyNetbiosName' )
-- saves dispatcher task id
local dispatcher = arg[1]
-- saves task id
local me = task.id()
while true do
    -- Receives with timeout
    local request, lsn, err = NB_ListenAndReceive( 100)
    if not err then
        -- sends request if no error
        -- flags contains this task id and the lsn origin of the
        -- message ( 8 bits) shifted 4 bits left
        task.post( dispatcher, request, me + lsn * 16)
    end
    local response, flags, err = task.receive( 100)
    if response then
        -- sends response
        err = NB_Send( math.floor( flags / 16), response)
    end
end
end
```

The TCP/IP input task (tcpip.lua):

```
-- The Tcp interface has been simplified a lot
-- Session startup and shutdown are hidden.
-- Sorry, we'll use polling for simplicity
-- Creates a server socket

local svrskt = CreateServerSocket( '*', 12321)

-- saves dispatcher task id
local dispatcher = arg[1]

-- saves task id
local me = task.id()

while true do

    -- Receives with timeout

    local request, cidx, err = AcceptAndReceive( svrskt, 100)

    if not err then

        -- sends request if no error

        -- flags contains this task id and the socket index

        -- origin of the message ( 16 bits) shifted 4 bits left

        task.post( dispatcher, request, me + cidx * 16)

    end

    local response, flags, err = task.receive( 100)

    if response then

        -- sends response

        err = SendAndClose( math.floor( flags / 16), response)

    end

end

end
```

The MQ input task (mq.lua):

```
-- The MQ interface has been simplified a lot
-- Sorry, we'll use polling for simplicity
-- Connects to the Queue Manager

local qmhandle, compcode, reason = MQCONN( 'MyQMName' )

-- saves dispatcher task id

local dispatcher = arg[1]

-- saves task id

local me = task.id()

while true do

    -- Receives with timeout

    local request, msgobjidx, err = MQ_GetWithTimeout( 100)

    if not err then

        -- sends request if no error

        -- flags contains this task id and the message object

        -- index origin of the message ( 24 bits)

        -- shifted 4 bits left

        task.post( dispatcher, request, me + msgobjidx * 16)

    end

    local response, flags, err = task.receive( 100)

    if response then

        -- sends response

        err = MQ_PutByIndex( math.floor( flags / 16), response)

    end

end

end
```

The SQL Worker task (worker.lua):

```
-- Loads ODBC support
local ENV = SQLOpen( 'odbc' )

-- Connects to SQL Server
local CON = ENV:Connect( { dsn = 'srch', uid = 'srch', pwd = 'srch' } )

while true do

    -- gets request

    local request, flags, err = task.receive( -1 )

    -- obtains which input task did send this

    local returnto = math.mod( flags, 16 )

    -- invokes a stored procedure

    local STMT = CON:Execute( "exec sp_srch( '" .. request .. "'" ) )

    -- gets the response

    local response = STMT:Fetch()

    -- closes the cursor

    STMT:Close()

    -- sends response

    task.post( returnto, response, flags )

end
```

Copyright

Copyright (c) 2003–2007 Daniel Quintela. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

