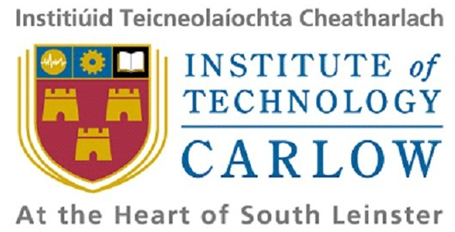
****

Student ID: C00131026

Student Name: Guanting Su

Supervisor: Joseph Kehoe

Date of submission: 26.11.2010

**Research Manual**

Contents

[1. Concurrency 1](#_Toc278361641)

[1.1. Issues 1](#_Toc278361642)

[1.2. Threads vs. Processes 1](#_Toc278361643)

[Processes 1](#_Toc278361644)

[Threads 2](#_Toc278361645)

[2. Locking 3](#_Toc278361646)

[2.1. Mutual Exclusion 3](#_Toc278361649)

[2.2. Semaphores 3](#_Toc278361653)

[2.3. Monitors 4](#_Toc278361654)

[2.4. Deadlock and Livelock 6](#_Toc278361655)

[2.5. Disadvantages of Locking 7](#_Toc278361656)

[3. What is Transaction? 7](#_Toc278361657)

[4. Transactional memory 8](#_Toc278361658)

[3.1. Hardware Transactional Memory 8](#_Toc278361660)

[3.2. Software Transactional Memory 9](#_Toc278361661)

[3.2.1. Advantages 11](#_Toc278361662)

[3.2.2. Disadvantages 11](#_Toc278361663)

[5. Lua 12](#_Toc278361664)

[5.1. Overview 12](#_Toc278361675)

[5.2. History 12](#_Toc278361676)

[5.3. Simple of Code 13](#_Toc278361677)

[5.4. Interacting Lua & C/C++ 14](#_Toc278361678)

[Sample of code 14](#_Toc278361679)

[5.5. LuaRocks 15](#_Toc278361680)

[5.5.1. Installation 16](#_Toc278361681)

[5.5.2. Using LuaRocks 16](#_Toc278361682)

[6. Lua libraries for concurrency 17](#_Toc278361692)

[6.1. Lanes 17](#_Toc278361694)

[6.1.1. Installing Lua Lanes 18](#_Toc278361702)

[6.1.2. Sample of code 18](#_Toc278361711)

[6.2. Coroutine 22](#_Toc278361712)

[6.3. Lua task 24](#_Toc278361713)

[6.3.1. Buliding Luatask 25](#_Toc278361723)

[6.3.2. Using of Luatask 26](#_Toc278361734)

[6.4. Lua Threads 26](#_Toc278361744)

[6.4.1. Locking by Lua 27](#_Toc278361745)

[6.5. Copas 27](#_Toc278361746)

[6.5.1. Installing 28](#_Toc278361758)

[6.5.2. Sample of copas 28](#_Toc278361759)

[6.6. Rings 29](#_Toc278361760)

[6.6.1. Installation 29](#_Toc278361761)

[6.6.2. Sample of Rings 29](#_Toc278361762)

[6.7. Compare the libraries 30](#_Toc278361775)

[7. Conclusion 32](#_Toc278361776)

[8. Reference 33](#_Toc278361777)

# Concurrency

*“In computer science, concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other. The computations may be executing on multiple cores in the same chip, preemptively time-shared threads on the same processor, or executed on physically separated processors. A number of mathematical models have been developed for general concurrent computation including Petri nets, process calculi, the Parallel Random Access Machine model and the Actor model.”* [1]

A concurrent program is a set of sequential programs that can be executed in parallel. We use the word process for the sequential programs that comprise a concurrent program and save the term program for this set of processes.

Increase the efficient of CPU and make it keep busy. Help human done more work use less time.

## Issues

*“Because computations in a concurrent system can interact with each other while they are executing, the number of possible execution paths in the system can be extremely large, and the resulting outcome can be indeterminate. Concurrent use of shared resources can be a source of indeterminacy leading to issues such as deadlock, and starvation.*

*The design of concurrent systems often entails finding reliable techniques for coordinating their execution, data exchange, memory allocation, and execution scheduling to minimize response time and maximize throughput.”* [1]

## Threads vs. Processes

### Processes

*“It is a program in execution. A process is the unit of work in a modern time-sharing system.”* [2] *“It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.”* [3]

* Process comprises
  + - Process ID
    - Text Section – for code
    - Register – Storage available on CPU
    - Program Counter – current instruction/activity
    - Stack – temporary data
    - Data Section – global data

The process is created, and the process creates a new process call child process and it is called parent process. Every child process also can create new process. The children processes can share the resource what is belong to parents. Sharing resources prevents process from overloading the system. In the execution of process, the parent and children can execute concurrently and parent waits for some or all children to terminate. When child process terminates usually returns status value to its parent process, and process will terminate after executing its final statement.

* Current processes can be
  + Independent processes

Cannot affect or be affected by other processes executing in the system

Any process that does not share data with other process

* + Cooperating processes

Can affect or be affected by other processes executing in the system

Any process that does shares data with any other process

Cooperating processes require Interprocess communication

### Threads

*“It is called the Lightweight process (LWP). It is a basic unit of CPU utilization.”* [2] *“In most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share the latter's instructions (its code) and its context (the values the various variables have at any given moment). To give an analogy, multiple threads in a process are like multiple cooks reading off the same cook book and following its instructions, not necessarily from the same page.”* [4]

* Threads Comprises
  + - Thread ID
    - Register
    - Program counter
    - Stack
    - Shares with threads belonging to same process
      * Code Section
      * Data Section
      * OS Resources

Compare multi threads and child processes. The threads can share data section so no need for interprocess communication and context witching quicker because there is less to save and load up because memory, share code section, data section, etc.

# Locking

*“In computer science, a lock is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. Locks are one way of enforcing concurrency control policies.”*[5]

The lock is help system can process multiple processes or threads to increase efficiency of the computer and avoid the computer errors due to concurrent programming.

Two types of lock:

* + Advisory locks

 Where each thread cooperates by acquiring the lock before accessing the corresponding data.

* + Mandatory locks

Where attempting unauthorized access to a locked resource will force an exception in the entity attempting to make the access.



## Mutual Exclusion

*“Mutual exclusion (often abbreviated to mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections. A critical section is a piece of code in which a process or thread accesses a common resource. The critical section by itself is not a mechanism or algorithm for mutual exclusion. A program, process, or thread can have the critical section in it without any mechanism or algorithm which implements mutual exclusion.”* [6]

Mutual exclusion allows the only one process to use the same resource at a same time.Lock, semaphore and monitor depend on this mechanism to build.



## Semaphores

*“In computer science, a semaphore is a protected variable or abstract data type that constitutes a classic method of controlling access by multiple processes to a common resource in a parallel programming environment. A semaphore generally takes one of two forms: binary and counting. A binary semaphore (flag) is a simple "true/false" (locked/unlocked) variable that controls access to a single resource. A counting semaphore is a counter for a set of resources. Either type may be employed to prevent a race condition. On the other hand, a semaphore is of no value in preventing resource deadlock, such as illustrated by the dining philosophers’ problem.”* [7]

The semaphore can control the multi processes states. Every process depends on the semaphore to change states (i.e. wait, running, and block).

*“The semaphore was introduced to provide a synchronization primitive that does not require busy waiting. Using semaphores we have given solutions to common concurrent programming problems. However, the semaphore is a low-level primitive because it is unstructured. If we were to build a large system using semaphores alone, the responsibility for the correct use of the semaphores would be diffused among all the implementers of the system. If one of them forgets to call signal(S) after a critical section, the program can deadlock and the cause of the failure will be difficult to isolate.”* [8]

The simple example of the semaphore show as the follow:

// allow process gets the resource to run the code. If it is not, the process should wait the semaphore.

Lock.wait()

//code

.

.

.

//releases the resource and resets the semaphore.

Lcok.signal()

## Monitors

*“In concurrent programming, a****monitor****is an object intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel.*

*Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.”* [9]

*“Monitors provide a structured concurrent programming primitive that concentrates the responsibility for correctness into modules. Monitors are a generalization of the kernel or supervisor found in operating systems, where critical sections such as the allocation of memory are centralized in a privileged program. Applications programs request services which are performed by the kernel. Kernels are run in a hardware mode that ensures that they cannot be interfered with by applications programs.”* [8]

The simple example of the monitor

|  |  |
| --- | --- |
| monitor CS  integer n <- 0  operation increment  integer temp  temp <- n  n <- temp + 1 | |
| p | q |
| p1: CS.increment | q1: CS.increment |

If the two process p1 and p2 are mutual execute, when they terminate the value of n should be 2.

The example semaphore simulated with monitor:

|  |  |
| --- | --- |
| monitor Sem  integer s <- k  condition notZero  operation wait  if s = 0  waitC(notZero)  s <- s - 1  operation signal  s <- s + 1  signalC(notZero) | |
| p | q |
| loop forever  non-critical section | loop forever  non-critical section |
| p1: Sem.wait  critical section  p2: Sem.signal | q1: Sem.wait  critical section  q2: Sem.signal |

Follow is the differences between semaphore and monitor:

| **Semaphore** | **Monitor** |
| --- | --- |
| wait may or may not block | waitC always blocks |
| signal always has an effect | signalC has no effect if queue is empty |
| signal unblocks an arbitrary blocked process | signalC unblocks the process at the head of the queue |
| a process unblocked by signal can resume execution immediately | a process unblocked by signalC must wait for the signaling process to leave monitor |

## Deadlock and Livelock

* Deadlock

What will cause deadlock? The lock put in the wrong order. The two or more processes are waiting for each other process to release resource. Hence the processes never get the resource forever and cause the computer stop working.

* Livelock

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

## Disadvantages of Locking

Although the locking is very useful method to solve the problem of concurrency to avoid use the same resource at a same time, but there is a big problem that is deadlock to the locking. The deadlock will make a program always use the resource what is the other programs waiting for, and then the other programs will wait the resource forever. Using the locking to solve concurrency is hard to avoid the deadlock, especially the big program. Because when the developers coding the program, they is really hard to control when it should release the resource and when it should wait the resource.

The locking is increase the efficient of the computer, but sometime the process will use the resource for a long time. And the other one process just will use the resource for a while. That is leading to computer is low-level efficient. But we can available the resource depend on the priority of process. But it is also hard to complete the problem.

# What is Transaction?

Before we start to understand what is the software transactional memory and how does it work. We need to know what transaction, its theory is and how to work. We also need study the transactional memory.

*“A transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer. A database transaction has four specific attributes: failure atomicity, consistency, isolation, and durability—collectively known as the ACID properties.”[13]*

* *Atomicity* requires that all constituent actions in a transaction complete successfully, or that none of these actions appear to start executing.
* Consistency is entirely application dependent.
* *Isolation* requires that each transaction produce a correct result, regardless of which other transactions are executing concurrently.
* *Durability* requires that once a transaction commits, its result be permanent (i.e., stored on a durable media such as disk) and available to all subsequent transactions.

# Transactional memory

Above I introduce achieve parallelism programs by locking. Although the locking solve the problem of concurrency, but the locking there are some disadvantages. And then people find out the mew technique what the idea is com form transaction of databases. The new technique is called Transactional memory. Transactional memory (TM) is view as a promising synchronization mechanism suited for emerging parallel architectures. (Rajwar, Transactional Memory, 2007)

***“Transactional memory*** *attempts to simplify parallel programming by allowing a group of load and store instructions to execute in an atomic way. It is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing.* *Transactional memory lies in the programming interface of parallel programs. The goal of a transactional memory system is to transparently support the definition of regions of code that are considered a transaction, that is, that have atomicity, consistency and isolation requirements.”* [10]

Although the idea of transactional memory is come from database transaction, but they have some differences and the transactional memory require some new implementation techniques. Follow the difference are among the most important:

* Data in a database resides on a disk, rather than in memory. And transactional memory accesses main memory. Hardware support is more attractive for transactional memory than for database systems.
* Transactional memory is not durable since data in memory does not survive program termination.
* Transactional memory is a retrofit into a rich, complex world full of existing programming languages, programming paradigms, libraries, programs, and operating system.

The TM’s greatest opportunity lies with those situations that are poorly served by combinations of pre-existing mechanisms.



## Hardware Transactional Memory

*“Hardware transactional memory (HTM) is a hardware system that supports implementing nondurable ACI (failure atomicity, consistency, and isolation) properties for threads manipulating shared data. HTM systems typically have a moderate software component and a larger hardware component. Their goal is to achieve the best performance with low overheads. HTMs introduce new hardware features to track transactional accesses, buffer tentative updates, and detect conflicts, while STMs use software to do the same.*

*Although STM are more flexible than HTM and offer some advantages, HTM systems have several key advantages over STM systems:*

*• HTMs can typically execute applications with lower overheads than a.*

*HTMs can have better power and energy profiles and higher performance.*

*HTMs execute independent of compiler technology and independent of memory access characteristics.*

*• HTMs are minimally invasive in an existing execution environment as they can potentially accommodate transactions that call functions that are not transaction aware, transaction-safe legacy libraries, and third-party libraries.*

*• HTMs can provide high-performance strong isolation without requiring applicationwide changes.*

*• HTMs are better suited for unmanaged and unsafe environments. ” [15]*

## Software Transactional Memory

*“In computer science,* ***software transactional memory*** *(STM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. It is an alternative to lock-based synchronization. A transaction in this context is a piece of code that executes a series of reads and writes to shared memory. These reads and writes logically occur at a single instant in time; intermediate states are not visible to other (successful) transactions.”* [11]

*“Software transactional memory (STM) is a software system that implements nondurable transactions with the ACI (failure atomicity, consistency, and isolation) properties for threads manipulating shared data. STMsystems run, for the most part, on conventional processors; though some early systems postulated more aggressive synchronization primitives than implemented by current processors.”[14]*

Software transactional memory (STM) offer advantages over than hardware transactional memory (HTM):

* Software is more flexible than hardware and permits the implementation of a wider variety of more sophisticated algorithms.
* Software is easier to modify and evolve than hardware.
* STMs can integrate more easily with existing (software) systems and language features, such as garbage collection.
* STMs have fewer intrinsic limitations imposed by fixed-size hardware structures, such as caches.

*“It is an abstraction for concurrent communication. The main benefits of STM are composability and modularity. That is, using STM you can write concurrent abstractions that can be easily composed with any other abstraction built using STM, without exposing the details of how your abstraction ensures safety. This is typically not the case with other forms of concurrent communication, such as locks or MVars*.” [12]

The following is a sample showing STM in action:

|  |
| --- |
| Process A |
| BeginTransaction  Atomic{  …<code>…  }  EndTransaction |

STM can be implemented as a lock-free algorithm or it can use locking. Follow is a sample of STM locking. The process A get the resource to run the code. After finish the code, if no other processes using, the process A will use lock to lock the resource to avoid the resource is available to others. If the lock is work, update the new value to resource and then reseals the resource. But if lock won’t work means other process uses it, the process A will restarts the transaction.

|  |
| --- |
| Process A |
| BeginTransaction  Atomic{  …<code>…  }  EndTransaction    Lock(ture)  If Lock works  Update resouces  Else  Goto BeginTransaction |

If the multi processes use the resource, the process A finish and want to update it, but maybe a few minutes ago, but the process A continue update resource and then the value of resource should be wrong. Therefore to prevent this situation happen, resource there are a lot of versions. If the resource has been updated, the number of version will be plus as well. But it is also need lock.

|  |
| --- |
| Process A |
| BeginTransaction  Atomic{  …<code>…  }  EndTransaction    Lock(true)  If Lock works  If the version doesn’t change  Update resource  Version No. ++  Else  Goto BeginTransaction |

### Advantages

Compare with lock, the STM is simpler. Because:

* + The transaction can be view in isolation and it looks like a single-threaded.
  + Deadlock and livelock can be completely avoided.
  + More efficient.

### Disadvantages

Although the STM has many advantages, but it also has limitations.

*“One troublesome point in dealing with any literature on transactional memory or non-blocking synchronization is that there is little in the way of formal analysis. Most proposals present proofs for their claims, but none use a common language, since — as far as the author is aware — there are few good formal frameworks for reasoning about concurrency and parallelism. ” [16]*

*“They cannot perform any operation that cannot be undone, including most I/O. Such limitations are typically overcome in practice by creating buffers that queue up the irreversible operations and perform them at a later time outside of any transaction. In Haskell, this limitation is enforced at compile time by the type system.”*[11]

# Lua



## Overview

*“Lua is a powerful, fast, lightweight, embeddable scripting language. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.”*[17] Now more and more programs need Lua to design. Lua has been use in many industrial applications (e.g. Adobe’s Photoshp lightroom), with an emphasis on embedded and games (e.g. World of Warcraft). Maybe there are not completely designed by Lua, but they use Lua to improve their functions of program. And now Lua is the leading scripting language in games. And so far Lua has been published several version and some books about it. Why these developers of programs choose Lua. Next are advantages of Lua:

Lua is fast, several researches show Lua is the fastest language in the scripting languages. Lua is portable, Lua is distributed in a small package and builds out-of-box in all platforms that have an ANSI/SO C compiler. Lua can run on all kinds of UNIX and Windows, and mobile devices as well (e.g. handheld computers and mobile phone and so on) and embedded microprocessors (such as ARM and Rabbit) for applications. Lua is embeddable, Lua iis fast language engine with small footprint that you can embed easily into your application. Lua has a simple and well API document and it allows strong integration with code written in other language. It is easy to extend libraries of Lua or extend programs written in other language with Lua. Lua is not only used to extend program in C/C++, it also can be in Java, C#, Smalltalk and so on, and even in other scriting languages, such as Perl and Ruby. Lua is powerful (but simple), A fundamental concept in the design of Lua is to provide meta-mechanisms for implementing features, instead of providing a host of features directly in the language. Lua is small, the new version of Lua contains source code, document, and examples, takes 212k compressed and 860k uncompressed. Lua is free open-source software, distributed under a very liberal licence (the well-know MIT license). Just download it and use it.

## History

*“Lua was created in 1993 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, members of the Computer Graphics Technology Group (Tecgraf) at thePontifical Catholic University of Rio de Janeiro, in Brazil. From 1977 until 1992, Brazil had a policy of strong trade barriers (called a 'market reserve') for computer hardware and software. In that atmosphere, Tecgraf's clients could not afford, either politically or financially, to buy customized software from abroad. Those reasons led Tecgraf to implement from scratch the basic tools it needed. Lua's historical 'father and mother' were data-description/configuration languages SOL (Simple Object Language) and DEL (data-entry language).”[18]*

## Simple of Code

Lua is scripting language, it can directory run without compile. So the first Lua example is “Hello World”. In lua just need

print(“Hello World”)

The following example is more complex, we define a calculate factorial function.

--define a factorial function

function fact(n)

if n==0 then

return 1

else

return n\*fact(n-1)

end

end

print("enter a number:")

a=io.read("\*number") --read a number

print(fact(a))

In this example, use recursion to calculate the factorial of number. For example, if I input 5, the code the first return should be 5\*fact(5-1), and then function invoke itself, the second time return is 5\*4\*fact(4-1). As before repeating many time until n==0, the final result should be 5\*4\*3\*2\*1=120.

*“Each piece of code that Lua executes, such as a \_le or a single line in interactive mode, is called a chunk. A chunk is simply a sequence of commands (or statements).”*[19]

Lua does not need separator between consecutive statements, but if you want, you also can use semicolon. For example:

a=1

b=a\*2

a=1;

b=a\*2;

Both of them are fine, no error and can be executed.

The code also can be typed likes this:

a=1; b=a\*2

a=1 b=a\*2 --Ugly, but valid

The second one is valid as well. Above four examples are all valid and equivalent.

## Interacting Lua & C/C++

Lua is scripting language, it means Lua is not only can be independent developing a compete program and Lua can be developed to embed other program as well. Lua is implemented in pure ANSI C, you can build Lua in an ANSI C compiler and Lua also can compiles cleanly as C++. And Lua can be invoked by C/C++, the Lua can invoke C/C++ as well.

***“ANSI C*** *is the standard published by the American National Standards Institute (ANSI) for the C programming language. Software developers writing in C are encouraged to conform to the requirements in the document, as it encourages easily portable code.”[20]*

ANSI C now can be supported almost compilers. Nowadays most of the C code being written is based on ANSI C.

*“These two views of Lua (as an extension language and as an extensible language) correspond to two kinds of interaction between C and Lua. In the first kind, C has the control and Lua is the library. The C code in this kind of interaction is what we call application code. In the second kind, Lua has the control and C is the library. Here, the C code is called library code. Both application code and library code use the same API to communicate with Lua, the so-called C API.” [21]*

C API is completely is a standard C language, when you use C API, you also need care about the type of values, memory allocation and so on.

### Sample of code

#include <stdio.h>

#include "lua.h"

#include "lauxlib.h"

#include "lualib.h"

int main (void) {

char buff[256];

int error;

lua\_State \*L = luaL\_newstate(); /\* opens Lua \*/

luaL\_openlibs(L); /\* opens the standard libraries \*/

while (fgets(buff, sizeof(buff), stdin) != NULL) {

error = luaL\_loadbuffer(L, buff, strlen(buff), "line") ||

lua\_pcall(L, 0, 0, 0);

if (error) {

fprintf(stderr, "%s", lua\_tostring(L, -1));

lua\_pop(L, 1); /\* pop error message from the stack \*/

}

}

lua\_close(L);

return 0;

}

This is a sample of a stand-alone Lua interpreter. The header file lua.h defines the basic functions provided by Lua. It includes functions to create a new environment, to invoke Lua functions (such as lua\_pcall), to read and write global variables in the Lua environment, to register new functions to be called by Lua, and so on. Everything defined in lua.h has a lua-prefix. The header file luaxlib.h defines the functions provided by the auxiliary library (auxlib). All its definitions start with luaL\_ (e.g. luaL\_loadbuffer). The auxlib library uses the basic API provided by lua.h to provide a higher abstraction leve; all Lua standard libraries use the auxlib.

You can compile Lua both as C and as C++ code; lua.h does not include this typical adjustment code that is present in several other C libraries:

#ifdef \_\_cplusplus

extern "C" {

#endif

...

#ifdef \_\_cplusplus

}

#endif

If you have compiled Lua as C code (the most common case) and are using it in

C++, you can include lua.hpp instead of lua.h. It is de\_ned as follows:

extern "C" {

#include "lua.h"

}

## LuaRocks

*“This is****LuaRocks****, a deployment and management system for Lua modules.*

*LuaRocks allows you to install Lua modules as self-contained packages called "rocks", which also contain version dependency information. This information is used both during installation, so that when one rock is requested all rocks it depends on are installed as well, and at run time, so that when a module is required, the correct version is loaded. LuaRocks supports both local and remote repositories and multiple local rocks trees. You can download and install LuaRocks on UNIX and Windows.*

*LuaRocks is free software and uses the same license as Lua 5.1.”*[22]

There are several types of rocks, and when packed they are identified by their filename extensions. These are:

* **Source rocks** (.src.rock): these contain the rockspec and the source code for the Lua modules provided by the rock. When installing a source rock, the source code needs to be compiled.
* **Binary rocks** (.*system*-*arch*.rock: .linux-x86.rock, .macosx-powerpc.rock): these contain the rockspec and modules in compiled form. Modules written in Lua may be in source .lua format, but modules compiled as C dynamic libraries are compiled to their platform-specific format.
* **Pure-Lua rocks** (.all.rock): these contain the rockspec and the Lua modules they provide in .lua format. These rocks are directly installable without a compilation stage and are platform-independent.

LuaRocks handle dependencies on Lua modules, rocks can specify other rocks it depends on, and attempts to fulfill those dependencies at install time. A rock will only be installed if all its dependencies can be fulfilled. LuaRocks also supports verification of dependencies on external libraries. A rock can specify an external package it depends on (for example, a C library), and give to LuaRocks hints on how to detect if it is present, typically as C header or library filenames.

### Installation

In UNIX system, install Lua first, using the appropriate methods for your system. Download and unpack the LuaRocks tarball. Run “./configure”. Run “make’ and as superuse, run”make install”.

But if your system is Windows. The Lua for Windows, which there are all-in one package. In this package include many different libraries and even LuaRocks. You also can down laod the LuaRocks package for Windows is all in one package which includes everything you need to run LuaRocks, Including helper binaries and a Lua interpreter. The package includes a n installer script, INSTALL.BAT, which provides a number of options for customizing your installation. Run INSTALL/? for detail.

### Using LuaRocks

After finish the installation of LuaRocks. LuaRocks installs some command-line which are your interface for managing you rocks: luarocks and luarocks-admin. Make sure the directory where they are located is in your PATH.

The command to run luarocks:

Luarocks

You can get help on any command by using the help command:

Luarocks help install

Installing packages is done by typng command, such as:

Luarocks install luasocket

 If you wrote a Lua package (containing one or more modules) and want to make it available to users through LuaRocks. The process consists essentially of the following steps:

* Writing a rockspec file
* Publishing your code online
* Submitting a rockspec for inclusion in the rocks server



# Lua libraries for concurrency



## Lanes

*“Lua Lanes is a Lua extension library providing the possibility to run multiple Lua states in parallel. It is intended to be used for optimizing performance on multicore CPU's and to study ways to make Lua programs naturally parallel to begin with.*

*Lanes is included into your software by the regular require "lanes" method. No C side programming is needed; all APIs are Lua side, and most existing extension modules should work seamlessly together with the multiple lanes.”[23]*

Its features are lanes have separated data. Shared data is possible with Linda objects. Communications is separate of threads, using Linda objects. Data passing uses fast inter-state copies (no serialization required). “Deep userdata” concept, for sharing userdata over multiple lanes. Millisecond level timers, integrated with the Linda system. Threads can cancellable, with proper cleanup. No application level locking – ever!

The Lua Lanes can run in several systems, such as Mac OS X PowerPC/Intel (10.4 and later), Linux86 and Windows 2000/XP and later (MinGW or Visual C++ 2005/2008). The underlying threading code can be compiled Win32 API or Pthreads. But if the threading codes are compiled under the Pthread, and the threading codes can run in only some systems not any system. Pthread require OS specific tweaks and guessing undocumented behavior. But other features of the Lua Lanes should be portable to any modern platform.

Although the Lua Lanes provides a good technique to achieve running multiple Lua states in parallel, but it there are some limitations. Coroutines are not passed between states. Sharing full userdata between states needs special C side preparations. It doesn’t include network level parallelism.



### Installing Lua Lanes

Lua Lanes is built simply by “make” command on the support platform (make-vc for Visual C++). In Windows, users can open “Command Prompt” from Programs of Start. In the “Command Prompt”, types “make” command to install Lua Lanes.

To install Lanes, you should need the lanes.lua and lua51-lanes.so|dll files to be reachable by Lua. Or use Lua Rocks package management.

>luarocks search lanes

--output listing Lua Lanes is there—

>luarocks install lanes

--output—

But there is a Lua for Windows, it include many libraries what include Lua Lanes. Hence if you use Windows system and you do not want to install libraries one by one. You can install Lua for Windows to quickly start to code Lua.



### Sample of code

#### Creation

Now we see the lanes how to work, and its some sample code. The first one is let’s see how to create lanes.

require "lanes"

f= lanes.gen( function(n) return 2\*n end )

a= f(1)

b= f(2)

print( a[1], b[1] ) -- 2 4

In this sample, invoke a function for parallel calling, and calling it with several arguments. Each of the two results is calculated in a separate OS thread, parallel to the calling one.

#### Status

The Lua Lanes also there are different states. You can use status function to check the state of the current execution of a lane. This is similar to coroutine.status, which there are suspended, running, normal and dead four states.

str= lane\_h.status

#### Results and errors

There are two ways to read the result of a lane that will be waited by simply reading. But before the result reading, make sure lane has finished and pass its first (may be only) return value. Other return will be available in other lane\_h indices.

[val]= lane\_h[1]

If the lane ended in an error, it is propagate state at this place.

[...]|[nil,err,stack\_tbl]= lane\_h:join( [timeout\_secs]

The lanes will wait until the lane finishes. Or timeout, returns nil on timeout. If the lane meet an error, then the nil, err, stack\_tbl are return values of the lane. Unlike in reading the results in table fashion, errors are not propagated.

If you use join function, make sure your lane main function return a non-nil value so you can tell timeout and error cases apart from successful (using the .status property may be risky, since it might change between a timed out join and the moment you read it).

require "lanes"

f= lanes.gen( function() error "!!!" end )

a= f(1)

--print( a[1] ) -- propagates error

v,err= a:join() -- no propagation

if v==nil then

error( "'a' faced error"..tostring(err) )

-- manual propagation

end

If you want to wait for multiple lanes to finish, use Linda object. Give each lane a specific id, and send that id over a Linda once that thread is done (as the last thing you do). The Linda will be introduced later in this paper.

#### Cancelling

Send a cancellation request to the lane. If timeout\_secs is non-zero, waits for the request to be processed, or a timeout to occur. Returns true if the lane was already done (in "done", "error" or “cancelled" status) or if the cancellation was fruitful within timeout period.

bool=lane\_h:cancel([timeout\_secs=0.0,][force\_kill\_bool=fals)

If the lane is still running and force\_kill is true, the OS thread running the lane is forcrfully killed.

#### Lindas

*“Communications between lanes is completely detached from the lane handles themselves. By itself, a lane can only provide return values once it's finished, or throw an error. Needs to communicate during runtime are handled by Linda objects, which are deep userdata instances. They can be provided to a lane as startup parameters, upvalues or in some other Linda's message.*

*Access to a Linda object means a lane can read or write to any of its data slots. Multiple lanes can be accessing the same Linda in parallel. No application level locking is required; each Linda operation is atomic.”[23]*

require "lanes"

local linda= lanes.linda()

local function loop( max )

for i=1,max do

print( "sending: "..i )

linda:send( "x", i ) -- linda as upvalue

end

end

a= lanes.gen("",loop)( 10000 )

while true do

local val= linda:receive( 3.0, "x" ) -- timeout in seconds

if val==nil then

print( "timed out" )

break

end

print( "received: "..val )

end

Characteristics of the Lanes implementation of Lindas are:

* keys can be of number, string or boolean type
* values can be any type supported by inter-state copying (same limits as for function parameters and upvalues)
* consuming method is :receive (not in)
* non-consuming method is :get (not rd)
* two producer-side methods: :send and :set (not out)
* send allows for sending multiple values -atomically- to a given key
* receive can wait for multiple keys at once
* individual keys' queue length can be limited, balancing speed differences in a producer/consumer scenario (making :send wait)

h= lanes.linda()   
  
bool= h:send( [timeout\_secs,] key, ... )   
[val, key]= h:receive( [timeout\_secs,] key [, ...] )   
  
= h:limit( key, n\_uint )

The send and receive methods use Linda keys as FIFO stacks (first in first out). If using numbers as the first Lina key, one must explicitly give nil as the timeout parameter to avoid ambiguities.

#### Timer

=lanes.timer(linda\_h,key,date\_tbl|first\_secs[period\_sec])

Timer can be run once, or in a reoccurring fashion (period\_secs>0). The first occurrence can be given either as a date or as a relative delay in seconds. The date table is like what os.date("\*t")returns, in the local time zone. Once a timer expires, the key is set with the current time (in seconds, same offset as os.time() but with millisecond accuracy). The key can be waited upon using the regular Linda: receive () method. A timer can be stopped simply by first\_secs=0 and no period.

#### Locks

Lanes does not generally require locks or critical sections to be used, at all. If necessary, a limited queue can be used to emulate them. lanes.lua offers some sugar to make it easy:

lock\_func= lanes.genlock( linda\_h, key [,N\_uint=1] )

lock\_func( M\_uint ) -- acquire

..

lock\_func( -M\_uint ) -- release

The generated function acquires M entries from the N available, or releases them if the value is negative. The acquiring call will suspend the lane, if necessary. Use M=N=1 for a critical section lock (only one lane allowed to enter).

#### Limitations on data passing

Data passed between lanes (either as starting parameters, return values, upvalues or via Lindas) must conform to the following:

* Booleans, numbers, strings, light userdata, Lua functions and tables of such can always be passed.
* Cyclic tables and/or duplicate references are allowed and reproduced appropriately, but only within the same transmission.
* using the same source table in multiple Linda messages keeps no ties between the tables
* Objects (tables with a metatable) are copyable between lanes.
  + metatables are assumed to be immutable; they are internally indexed and only copied once per each type of objects per lane
* C functions (lua\_CFunction) referring to LUA\_ENVIRONINDEX or LUA\_REGISTRYINDEX might not work right in the target
  + rather completely re-initialize a module with require in the target lane
* Full userdata can be passed only if it's prepared using the deep userdata system, which handles its lifespan management
  + in particular, lane handles cannot be passed between lanes
* coroutines cannot be passed

## Coroutine

*“Coroutines allow us to execute several tasks at once. This is done in a controlled manner by passing control to each routine and waiting until the routine says it has finished. We can reenter the routine to continue at a later time and by doing this repeatedly we achieve multi-tasking.”[24]*

*“A coroutine is similar to a thread (in the sense of multithreading): it is a line of execution, with its own stack, its own local variables, and its own instruction pointer; but sharing global variables and mostly anything else with other coroutines. The main difference between threads and coroutines is that, conceptually (or literally, in a multiprocessor machine), a program with threads runs several threads concurrently. Coroutines, on the other hand, are collaborative: at any given time, a program with coroutines is running only one of its coroutines, and this running coroutines suspends its execution only when it explicitly requests to be suspended.”[25]*

Each task runs in a thread which is independent away from other threads. If there are several tasks running at once is often called multi-thread. But in multi-thread there are different ways to be implemented. Some systems allocate a fix amount of time to each thread and take control away when the time is up, passing the control to next one. This is called pre-emptive multi-thread. In other systems, a thread is allocated time how long it is taking. The thread knows it must pass control to other thread so that they can function as well. This is call cooperative or collaborative.

Coroutines (also called collaborative multithreading) in Lua are not threads or process of operating system. Coroutines are block of Lua code what are reated within Lua, and have their own flow of control like threads. Actually coroutines save state of Lua by stack to simulate thread. Because it is a fake thread, switching thread cost will be small, and it is lightweight for create a new thread. New\_thread just create a new stack in momory used to store the coroutine variable, also kown as lua\_state. Only one coroutine ever runs at a time, and it runs until it activates others, or yield (return to the coroutine that invoked it). Coroutines are a way to express multiple cooperating threads of control in a convenient and natural way, but unfortunately do not execute in parallel. However, since coroutines switch much faster than operating system threads and do not typically require complex and sometimes expensive locking mechanisms, using coroutines is typically faster than the equivalent program using full OS threads.

A coroutine in Lua represents an independent thread of execution and unlike threads in multithread systems. A coroutine there are four different states: suspended, running, dead and normal. A coroutine only suspends its execution by explicitly calling a yield function. When we create a coroutine, it starts in the suspended.

Create a new coroutine, it supports create function.

co = coroutine.create(function () print("hi") end)

The function coroutine.resume(re) strats the execution of a coroutine, changing its state form suspended to running:

coroutine.resume(co) --> hi

If the coroutine body prints “hi” and terminate, leaving the corotine in the dead state.

print(coroutine.status(co)) --> dead

The real power of coroutines stems from the yield function, which allows a running coroutine to suspend its own execution so that be resumed later.

co = coroutine.create(function ()

for i=1,10 do

print("co", i)

coroutine.yield()

end

end)

## Lua task

The Luatask is aim to build multiple Lua universes, each running into an independent OS thread. Imagine a data base search system, waiting for requests from clients with different communication mechanism: Netbios, TCP/IP and Websphere MQ.

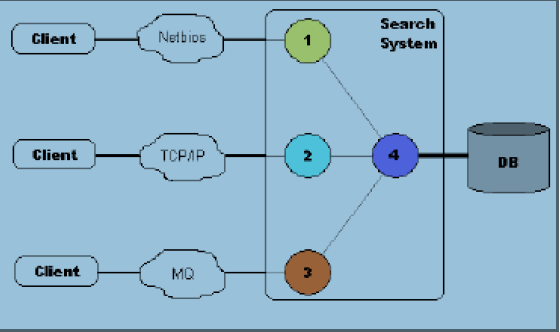
We can identify at least four specific functions inside the system:

1. Netbios communications.

2. TCP/IP communications.

3. MQ communications.

4. Search and retrieval.



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.

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 dispacher 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.



### Buliding Luatask

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

**In 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



### Using of Luatask

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.



## Lua Threads

*“ANSI C, the standard to which Lua complies, has no mechanism for managing multiple threads of execution. Threads and the synchronization objects used to control them are provided by the underlying operating system. You'll need to use these in order to implement threading in Lua. You will not need to modify the Lua distribution to do this. Threads are tricky to get right even in the simplest of circumstances such as a pure C application. An application which embeds or extends Lua must cope with the additional complexity of coordinating threads with the Lua library. If your multitasking needs can be met with Lua's single-threaded coroutines, you would be well-advised to choose that route. Read CoroutinesTutorial for more details. If you do choose to implement multiple pre-emptive threads to your Lua project, the following guidelines may help. Each thread in C which interacts with Lua will need its own Lua state. Each of these states has its own runtime stack. When a new C thread is started, you can create its Lua state in one of two ways. One way is to call lua\_open. This creates a new state which is independent of the states in other threads. In this case, you'll need to initialize the Lua state (for example, loading libraries) as if it was the state of a new program. This approach eliminates the need for mutex locks (discussed below), but will keep the threads from sharing global data. The other approach is to call lua\_newthread. This creates a child state which has its own stack and which has access to global data. When a new C thread is started, you can create its Lua state in one of two ways. One way is to call lua\_open. This creates a new state which is independent of the states in other threads. In this case, you'll need to initialize the Lua state (for example, loading libraries) as if it was the state of a new program. This approach eliminates the need for mutex locks (discussed below), but will keep the threads from sharing global data. The other approach is to call lua\_newthread. This creates a child state which has its own stack and which has access to global data. This approach is discussed here.”[26]*

### Locking by Lua

Your biggest concern when working with threads is to prevent them from corrupting one another’s environments.

With some help from you, Lua will prevent its internal data structures from being corrupted. When Lua enters into an operation which must not be pre-empted, it calls lua\_lock. When the critical operation is complete, it calls lua\_unlock. In the default distribution these two functions do nothing. When using threads in Lua, they should be replaced with OS-dependent implementations. In a POSIX environment you'll use an object of type pthread\_mutex\_t. In Windows, you'll use either a handle which is returned from CreateMutex, or, more optimally, an opaque data structure of type CRITICAL\_SECTION.

All coroutines within a particular lua universe must share the same mutex. Avoid the mistake of associating the mutex with a specific Lua state and then failing to find it again when a different coroutine within the same universe is locked. A simple example for Win32 follows. A custom header file luauser.h might contain:

#define lua\_lock(L) LuaLock(L)

#define lua\_unlock(L) LuaUnlock(L)

#define lua\_userstateopen(L) LuaLockInitial(L)

#define lua\_userstatethread(L,L1) LuaLockInitial(L1) // Lua 5.1

void LuaLockInitial(lua\_State \* L);

void LuaLockFinal(lua\_State \* L);

void LuaLock(lua\_State \* L);

void LuaUnlock(lua\_State \* L);

## Copas

*“Copas is a dispatcher based on coroutines that can be used by TCP/IP servers. It uses****LuaSocket****as the interface with the TCP/IP stack.*

*A server registered with Copas should provide a handler for requests and use Copas socket functions to send the response. Copas loops through requests and invokes the corresponding handlers. For a full implementation of a Copas HTTP server you can refer to****Xavante****as an example.*

*Copas is free software and uses the same****license****as Lua 5.1”[27]*



### Installing

You can install Copas using **[LuaRocks](http://www.luarocks.org)**:

Luarocks install copas

### Sample of copas

Copas is a dispatcher that can help a lot in the creation of servers based on **LuaSocket**. Here we present a quick introduction to Copas and how to implement a server with it.

Assuming you know how to implement the desired server protocol, the first thing you have to do in order to create a Copas based server is create a server socket to receive the client connections. To do this you have to bind a host and a port using LuaSocket:

Server = socket.bind(host, port)

You have to create a handler function that implements the server protocol. The handler function will be called with a socket for each client connection and you can use copas.send() and copas.receive() on that socket to exchange data with the client.

For example, a sample echo handler would be:

Function echoHandler(skt)

Skt=copas.wrap(skt)

While true do

Local data=skt:receive()

If data==”quit” then

break

end

skt:send(data)

end

end

To register the server socket with Copas and associate it with the corresponding habdler we do:

Copas.addserver(server, echoHandler)

Finally, to start Copas and all the registered servers we just call:

Copas.loop()

As long as every handler uses Copas's send and receive, simultaneous connections will be handled transparently by Copas for every registered server.

## Rings

*“Rings is a library which provides a way to create new Lua states from within Lua. It also offers a simple way to communicate between the creator (master) and the created (slave) states.*

*Rings is free software and uses the same****license****as Lua 5*.1.

*Rings also offers****Stable****, a very simple API to manage a shared table at the master state.*”[ (Rings, 2008)]

### Installation

If you are using LuaRocks, just type:

Luarocks install rings

If you prefer to install manually, the compiled binary file should be copied to a directory in your **C path**. The file stable.lua should be copied to a directory in your **Lua path**.

### Sample of Rings

**rings.new (env)**

Returns a newly created Lua state. Takes an optional environment to be used by [**remotedostring**](http://keplerproject.github.com/rings/manual.html#rings_remotedostring). If the environment is nil, it defaults to the master\_M or \_G tables.

state**:close ()**

Closes the state.

state**:dostring (string, ...)**

Executes a string in the slave state. The arguments could be accessed exactly as in a **[vararg function](http://www.lua.org/manual/5.1/manual.html" \l "2.5.9)**. Valid types of arguments and return values are: number, string, boolean, nil and userdata (which are converted to lightuserdata).   
Returns a boolean indicating the status of the operation, followed by the returned values or an error message in case of error.

Slave function

The following function is registered in the newly created slave state.

**remotedostring (string, ...)**

Executes a string in the master state. Behaves exactly as the method dostring except that it acts in the master state.

Stable

Stable is a simple API which provides a way for a slave state to store and retrieve data to and from its master state. This library is not opened automatically in a slave state.

**stable.get (key)**

Returns the value of a given key.

**stable.set (key, value)**

Stores a value associated to a key. Returns nothing.

The following sample shows how to execute code in another state passing arguments and returning values:

require”rings”

S=rings.new()

data={12,13,14}

print（S:dostring([[

aux={}

for I, v in ipairs({…}) do

table.insert(aux,1,v)

end

return unpack(aux)]], unpack(data))) –true, 14,13,12

S: close



## Compare the libraries

Above introduce the several libraries of Lua for concurrency. And introduce how to install and their sample of codes. Now this paper will compare them, to understand what are different of them, they are work independent for the same purpose or they are work tighter to achieve concurrency program and so on.

1. Lua Lanes is a Lua extension library providing the possibility to run multiple Lua states in parallel. It is regular Lua 5.1 module. Completely separate Lua states, one per OS thread. No application level locking. Threads are cancellable (with complete cleanup). Timeout on all pending operations. Thread contents are given as regular Lua functions. Standard libraries opened to subthreads can be granually selected. Fast stack-to-stack copies, via hidden “keeper states”. No serialization needed. Protects calls to “require”, allowing wide compatibility with existing modules (and all, with minor changes). But the Lanes requires OS threads, currently 1:1 mapping to OS threads and not utilizing network parallelism.
2. Rings is a library which provides a way to create new Lua states and offers a simple way to communicate between the creator (master) and the created (slave) states. Rings also offers separate Lua state, but no multithreading. This makes it simple, but it won’t use more than one CPU core. Open all Lua standard libraries for subthreads, but Lanes opens the needed ones. Marshalls numbers, string, Booleans, userdata, Lanes marshalls also non-cyclic tables. “remotedostring” allows executing code in the master state, and the Lanes does not allow subthreads to trouble/modify master automatically to allow effective sandboxing. The same can be achieved by sending code between in Lanes, but master needs to explicitly allow receive a function and executer it. The advantage of Rings is it offers stable, a very simple API to manage a shared table at the master state. The weakness is thread contents define as tring, not Lua souce as such; does not give syntax check at file parsing, and does not allow syntax highlight.
3. LuaThread provides thread creation, mutexes, condition variables, and inter-thread queues to the Lua scripts. It takes a C-kind of approach, where Lua globals are shared by the threads running, and need therefore to be guarded against multithreading conflicts.

Whether this is exactly what you want, or whether a more loosely implemented multithreading (s.a. Lanes) would be better, is up to you. One can argue that a loose implementation is easier for the developer, since no application level lockings need to be considered。

* Pros:
  + No marshalling overhead, since threads share the same Lua state
* Cons:
  + requires a modified Lua core
  + application level locking required

1. Lua coroutines is an integral part of Lua 5 itseft. Th Coroutines allow us to execute several tasks at once. This is done in a controlled manner by passing control to each routine and waiting until the routine says it has finished. It is listed here, since it should be the first multitasking mechanism to consider. It can also be used together with Lanes, or Lua thread and Rings. Coroutines are very usable in provider/consumer situations, allowing you to queue Lua functions on an as-needed dataflow basis with each other.

* Pros:
  + works with plain Lua (no extensions)
  + works on any platform
  + lightweight (no OS level threading or locking involved)
* Cons:
  + co-operative, meaning your code will need to decide, who gets to run
  + does not utilize multiple CPUs/cores

*“The main difference between threads and coroutines is that, conceptually (or literally, in a multiprocessor machine), a program with threads runs several threads concurrently. Coroutines, on the other hand, are collaborative: at any given time, a program with coroutines is running only one of its coroutines, and this running coroutines suspends its execution only when it explicitly requests to be suspended.”[25]*

# Conclusion

From this research, I get clearer to understand what STM Lua is, and the STM Lua work for what. Know the Lua is scripting programming language. It is fast, small, easy to learn and free. The STM (software transactional memory) is instead of locking to achieve running the parallel program. Using this technique, developers do not need worry about deadlock or livelock, and it will increase the effciency of the parallel system. The Lua support some libraries to build the control multi-thread and something like that. This paper introduces the libraries for concurrency, and shows them how to install and work. And compare them, what are their different.

# Reference

[1]Wikipedia. (last modified 2010, September 24). *Concurrency*. Retrieved October 20, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Concurrency\_(computer\_science)

[2]Abraham Silberschatz Peter Baer Galvin and Greg Gagne (2005). Operating System Concepts. JOHN WILEY & SONS.

3]Wikipedia. (last modified 2010, November 3). *Process (computing)*. Retrieved Octobor 20, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Process\_(computing)

[4]Wikipedia. (last modified 2010, November 3). *Thread (computer science)*. Retrieved Octobor 20, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Thread\_(computer\_science)

[5]Wikipedia. (last modified 2010, October 10). *Lock (computer science)*. Retrieved October 17, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Lock\_(computer\_science)

[6]Wikipedia. (last modified 2010, October 29). *Mutual exclusion*. Retrieved October 18, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Mutual\_exclusion

[7]Wikipedia. (last modified 2010, November 1). *Semaphore (programming)*. Retrieved Octobor 18, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Semaphore\_(programming)

[8]Ben-Ari, M. (2006). *Principles of Concurrent and Distributed Programming.* Addison-Wesley.

[9]Wikipedia. (last modified 2010, November 3). *Monitor (synchronization)*. Retrieved Octobor 18, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Monitor\_(synchronization)

[10]Wikipedia. (last modified 2010, August 18). *Transactional memory*. Retrieved Octobor 30, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Transactional\_memory

[11]Wikipedia. (last modified 2010, October 31). *Software transactional memory*. Retrieved October 30, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Software\_transactional\_memory

[12]HaskellWiki. (last modified 2008, March 18). *Software transactional memory*. Retrieved Octobor 30, 2010, from HaskellWiki: http://www.haskell.org/haskellwiki/Software\_transactional\_memory

[13] Rajwar, James R. Larus and Ravi (2007). In *Transactional Memory* (p. 53). Morgan & Claypool.

[14] Rajwar, James R. Larus and Ravi (2007). In *Transactional Memory* (p. 6). Morgan & Claypool.

[15] Rajwar, James R. Larus and Ravi (2007). *Transactional Memory* (p. 131). Morgan & Claypool.

[16]Marshall, C. (2005, December 1). *Software Transactional Memory* (p. 10).

[17]Lua. (Last update 2010, October 12). *About of Lua*. Retrieved October 20, 2010, from Lua: http://www.lua.org/about.html

[18]Wikipedia. (last modified 2010 , November 6). *Lua (programming language)*. Retrieved November 1, 2010 , from Wikipedia: http://en.wikipedia.org/wiki/Lua\_(programming\_language)

[19]Ierusalimschy, R. (2006). *Programming in Lua (p. 4)*. Roberto Ierusalimschy.

[20]Wikipedia. (last modified on 2010 , October 11 ). *ANSI C*. Retrieved November 10, 2010 , from Wikipedia: http://en.wikipedia.org/wiki/ANSI\_C

[21]Ierusalimschy, R. (2006). *Programming in Lua (p. 217)*. Roberto Ierusalimschy.

[22]LuaRocks. (n.d.). *LuaRocks*. Retrieved November 10, 2010, from LuaRocks: http://luarocks.org/en

[23]Lua Lanes. (2009, Jan). *Lua Lanes*. Retrieved November 11, 2010, from Lua Lanes: http://kotisivu.dnainternet.net/askok/bin/lanes/index.html#description

[24]*Coroutines Tutorial*. (Last edited 2007, January 5). Retrieved November 16, 2010, from Lua-users: http://lua-users.org/wiki/CoroutinesTutorial

[25]Ierusalimschy, R. (2006). *Programming in Lua (p. 73)*. Roberto Ierusalimschy.

[26]*Threads Tutorial*. (Last edited 2007 , January 5). Retrieved November 16, 2010, from Lua-users: http://lua-users.org/wiki/ThreadsTutorial

[27]*Copas*. (2009, April 7). Retrieved November 13, 2010, from Lua Copas: http://keplerproject.github.com/copas/

[28]*Rings*. (2008, June 30). Retrieved November 15, 2010, from Lua Rings: http://keplerproject.github.com/rings/