****

**Student ID: C00131026**

**Student Name: Guanting Su**

**Supervisor: Joseph Kehoe**

**Date of submission: 19.12.2010**

**Functional Specification**

**<STM Lua>**

**(CW228)**

Content

[1. What is my project? 1](#_Toc280324619)

[2. Project can do 1](#_Toc280324620)

[3. Functions description 2](#_Toc280324621)

[3.1. Create Transaction 2](#_Toc280324625)

[3.2. Add valuables to Transaction 2](#_Toc280324626)

[3.3. Start Transaction 2](#_Toc280324627)

[3.4. Commit transaction 3](#_Toc280324628)

[3.4.1. Check version 3](#_Toc280324629)

[3.4.2. Update 3](#_Toc280324630)

[4. Potential Users 3](#_Toc280324631)

[5. Metrics 3](#_Toc280324632)

[6. Tests 4](#_Toc280324633)

[6.1. Single Transaction and single thread 4](#_Toc280324640)

[6.2. Single Transaction and Multiple threads 4](#_Toc280324641)

[6.3. Multiple Transactions and single thread 4](#_Toc280324642)

[6.4. Multiple Transactions and Multiple threads 5](#_Toc280324643)

# What is my project?

The concurrency programming is more and more popularly and impotent. The hard ware now is building more and more multi-cores in one CPU. To increase the efficient of the data process. On the other sides, the software also supports ways to achieve the concurrency programming. Use Locking, STM (Software Transaction Memory) and so on. And this project is implemented by STM and Lua. Why choose STM and not Locking.

Because the STM is lock-free. Only after process, it need lock to let resource cannot be available to others. If it finds other one use it, it will restart process the data. Compare STM with Locking, the STM can avoid dead lock and live lock and most of time is more efficient than Locking.

Lua is a powerful, fast, lightweight, simple, embeddable scripting language. Lua can run on all kinds of UNIX and Windows, and mobile devices as well. I build my project to a Lua library, and then any one use Lua can require this library to achieve concurrency.

# Project can do

My project will be developed into a library of Lua. And then if the other developers need this library, they can require the library in their Lua code to achieve concurrency programming base on STM. It supports some functions to let developers can develop easier, and save the time of developers. After create a Transaction, the developers can add valuable into the transaction. And then start the transaction to process the valuable. At last finish the process, the transaction commit update the new value to valuable. In STM, the developers don’t remember where need set a lock and where they should release the resource to avoid deadlock or livelock. Because use STM will not need to consider about that, it is able to completely avoid deadlock and livelock.

# Functions description

This section is description of project’s functions. And the process steps as following graphic show.



1.
2.
3.

## Create Transaction

Create a new transaction to store the users need process data, because a transaction runs in isolation, meaning it executes as if it’s the only operation running on the system and as if all other threads suspended while it runs. Hence the effects of a memory transaction’s stores are not visible outside the transaction until the transaction commits; it also means that there are no other conflicting stores by other transactions while runs.

## Add valuables to Transaction

The programmer can write some code in a transaction what also is called atomic. And then my project calculates it.

## Start Transaction

When start a transaction, the transaction will check the version number of valuable. Save the version number. When the transaction commits, the transaction will match the current version. Depend on the version number to update the valuable or not.

## Commit transaction

When the transaction need commit, the transaction will lock the valuable and check version and then update version and valuable.

### Check version

The transaction will check the current version of valuable and the version what is transaction got before. If they the same, means no one use valuable before, and make sure this is right value to update to valuable. And the version number of valuable will plus one. If they are not the same, the transaction gets the current version number and value of valuable, restart transaction.

### Update

After the check version, if pass the check update the new value to the valuable. And release the valuable.

# Potential Users

Today the concurrency programming widely use. It can help other developer easier t achieve concurrency. So my project main users should be developer who is developing a program by Lua and need achieve multithreading or something like that. Lua is scripting language; other programming language also can use Lua to achieve this purpose.

* Better than Locking. Only when it is finish will use lock for a moment.
* More efficient than others.
* It is small.
* And so on.

# Metrics

This section is describing after the project finish develop, if the project can achieve the below purposes means the project is successful.

* Develop project into library of Lua, and everyone who needs concurrency can require it in their Lua code. They can invoke the methods and functions to achieve the concurrency.
* My project can support single transaction, multiple transactions and the single transaction or the multiple transactions can be handled by single thread or multiple threads.
* When the transaction commit can check the version of valuable and update it successfully.
* When the transaction commit, using lock to avoid valuable is share with multiple transaction in the mean time. Let the valuable cam get the right value.

# Tests

This section is introducing how to test the project in the final test. Following are the examples of test code.

Require “STM” --my developing STM library

Require “lanes” –support create thread and control

1.
2.
3.
4.
5.
6.

## Single Transaction and single thread

--create new Transaction

newT = (function --code—return n end)

--create a thread

f = lanes.gen (function (n) return newt (n) end)

a=f (n)

print (a)

## Single Transaction and Multiple threads

--create new Transaction

newT = (function --code—return n end)

--create a thread

f1 = lanes.gen (function (n) return newt (n) end)

a=f1 (n)

f2 = lanes.gen (function (n) return newt (n) end)

b=f2 (n)

print (a)

print (b)

## Multiple Transactions and single thread

--create new Transaction

newT1 = (function –user type code-- return n end)

newT2 = (function –user type code-- return n end)

--create a thread

f = lanes.gen (function (n) return newT1 (n) end)

a=f (n)

end

f = lanes.gen (function (n) return newT2 (n) end)

b=f (n)

end

print (a)

print (b)

## Multiple Transactions and Multiple threads

--create new Transaction

newT1 = (function –user type code-- return n end)

newT2 = (function –user type code-- return n end)

--create a thread

f1 = lanes.gen (function (n) return newT1 (n) end)

a=f1 (n)

f2 = lanes.gen (function (n) return newT1 (n) end)

b=f2 (n)

f3 = lanes.gen (function (n) return newT2 (n) end)

c=f3 (n)

f4 = lanes.gen (function (n) return newT2 (n) end)

d=f4 (n)

print (a)

print (b)

print (c)

print (d)