Concurrent Communicating Sequential Processes

Module CSP

The CSP module of the Lua Virtual Machine (*lvm*) extends Lua with a concurrent and parallel process model based on a set of process constructors that can be nested, Among the process constructors, there is a set of synchronisation objects. Except concurrency, the process constructors and the channel IPC object matches exactly the process and communication behaviour of the original CSP model by Hoare (and the syntax of the OCCAM language).

The *CSP* module supports two levels of parallelism:

1. Threads executing process functions in a separate VM instance. Data can be shared by different VM instances and processes by a global memory store and data serialisation. Userdata objects can be shared by reference, all other data is shared by copy.
2. Fibers (**coroutines**) executing process functions in scheduled fibers in the same VM instance. Data can be shared by all fibers by reference. There is no parallel execution.

Process Constructors

Seq

`Seq(function {},shared?:{})`

Sequential process constructor executing a list of process functions strict sequentially in the current VM instance. The *Seq* process terminates if the last sequential process terminates. Following statements are executed after the *Seq* process terminates. The *Seq* constructor is trivial as Lua statements are already executed strict sequentially. All process functions share the same local and global scope.

Co

`Co(function {},shared?:{})`

Coroutine process constructor executing a list of process functions pseudo parallel in the current VM instance. Each process function is executed in a fiber (non-parallel but scheduled processes). The *Co* process terminates if the last process terminates, and following statements are executed after the *Co* process terminated. All process functions share the same local and global scope.

Par

`Par(function {},shared?:{})`

Parallel process constructor executing a list of process function parallel in different VM instances. Each process function has its own local and global scope. The *Par* process performs joining of (waiting for) all sub-processes and terminates if all parallel sub-processes terminated. Following statements are executed after the *Par* process terminated. The process function can access free variables and functions, but only with a local scope (except user-data tables). All free variables and functions will be serialised and deserialised in each sub-process.

Fork

`Fork(function {},shared?:{})`

Parallel process constructor executing a list of process function parallel in different VM instances. The *Fork* process performs no joining of (waiting for) sub-processes and following statements are executed immediately. The process function can access free variables and functions, but only with a local scope (except user-data tables). All free variables and functions will be serialised and deserialised in each sub-process.

Alt

`Alt(processes:function {},shared?:{})`

`Alt(processes:function {} {},shared?:{})`

Alternative choice process constructor executing one of the conditional processes. A conditional process have to execute a blocking operation, e.g., channel read. The first process that is ready is executed. Each conditional process is executed in a separate fiber (coroutine).

Interprocess Communication

Interprocess Communication (IPC) is used to synchronise processes for:

1. Cooperation
2. Coordination (competition)

IPC between threads (parallel processes) and fibers (coroutines) has to be distinguished. Thread IPC blocks entire threads and all contained fibers, whereas fiber IPC only blocks one fiber of a thread,

Channel

`Channel(depth:number,fiber?:boolean) -> channel`

Creates a new communication channel with the given FIFO depth. A *depth* = 0 creates an handshake channel (i.e., reader and writer rendezvous). If the *fiber* argument is true, a channel for scheduled coroutines (fibers) is created.

channel:read

```
channel:read() -> *
```

Reads data from the channel. If there is no data available, the operation suspends (blocks) the process execution until data is available.

channel:write

```
channel:write(data:*)
```

Writes data to the channel. If the queue is full (or there is no reader for *depth*=0), the operation suspends (blocks) the process execution until a read operation is performed.

Mutex

```
Mutex() -> mutex
```

Create a mutual exclusion lock.

mutex:lock

```
mutex:lock()
```

Acquires and locks a mutex. If the mutex is already locked, the operation suspends (blocks) the process execution and enqueues the process until the mutex lock is released and the current process gets the lock.

mutex:unlock

```
mutex:unlock()
```

Unlocks a previously acquired mutex lock. If there are waiting processes, the next process is dequeued and resumed.

Semaphore

```
Semaphore(init:number,fiber?:boolean) -> semaphore
```

Create a semaphore object with an initial counter value. If the *fiber* is set true, a semaphore for scheduled coroutines is created (cannot be used across parallel processes).

semaphore:down

`semaphore:down()`

Decrements the semaphore counter value by one. If the semaphore counter is already zero, the operation enqueues and suspends (blocks) the process until an *up* operation was performed.

semaphore:up

`semaphore:up()`

Increments the semaphore counter by one. If there are waiting processes, the next process is dequeued and resumed.

semaphore:level

`semaphore:level() -> number`

Return current counter value.

Event

`Event(fiber?:boolean) -> event`

Create an event object used to synchronise parallel processes or scheduled coroutines (*fiber* is true),.

event:await

`event:await()`

Waits for an event signal. The operation suspends (blocks) the process execution and enqueues the process until the event was raised by the *wakeup* operation.

event:wakeup

`event:wakeup()`

Signals an event and resumes all waiting processes.

Barrier

`Barrier(group:number,fiber?:boolean) -> barrier`

Create an event object used to synchronise processes

barrier:await

`event:await()`

Waits for a group of processes entering the barrier. The operation suspends (blocks) the process execution and enqueues the process until the last process (*i=group*) enters the barrier by the *await* operation.#

Timer

`Timer(interval:number,once:boolean,fiber?:boolean) -> timer`

Creates a timer object to synchronise processes temporarily. This timer can only be used in *Seq*, *Sched* , or *Alt* processes if the *fiber* argument is set. A fiber timer cannot be shared by *Par* processes (only with *fiber*=nil|false). The timer must be started after creation.

timer:await

`timer:await()`

Waits for a timer signal. The operation suspends (blocks) the process execution and enqueues the process until the event was raised by the timer..

timer:start

`timer:start()`

Starts the timer.

timer:stop

`timer:stop()`

Stops the timer.

Matrix

`Matrix:new(dims:number [],init?:number|function,type?:string) -> matrix`

`Matrix.double(dims:number [],init?:number|function) -> matrix`

`Matrix.float(dims:number [],init?:number|function) -> matrix`

`Matrix.int(dims:number [],init?:number|function) -> matrix`

Creates a new shared matrix object (supported dimensions: 1,2,3) that can be accessed in parallel processes concurrently. Supported data types are: *float*, *double* (default), *int*. Concurrent access is serialised and protected by a mutex. The index order is *i=column* (1), *i=row*,*j=column* (2), and *i=level*,*j=row*,*k=column* (3).

matrix:ones

`matrix:ones() -> matrix`

Assigns value 1 to all diagonal elements of the matrix.

matrix:random

`matrix:random(a?,b?) -> matrix`

Assigns random values to all elements of the matrix (default in the range 0,1, or alternatively in range *a*,*b*).

matrix:print

`matrix:print(format?:string) -> string`

Formatted printer (default format "%2.2f").

matrix:read

`matrix:read(i,j,k) -> number`

Reads the value of an element of the matrix.

matrix:write

`matrix:write(value:number,i,j,k)`

Writes new value to an element of the matrix.

Examples

```
require('Csp')
local counter=1
Seq({
  function () counter=counter+1 end,
  function () counter=counter+1 end
  })
print('Waiting for counter='..counter)
```

Seq Constructor Example

```
require('Csp')
Par({
  function (id)
```

```
    log('Process A starting');
    local x=ch:read();
    log('hello '..x);
  end,
  function (id)
    ch:write('world');
  end
  },{
    ch=Channel(0)
  })
print('Waiting ..')
```

Par Constructor Example

```
require('Csp')
Par({
  function (id)
    local x
    Alt({
      function () print(1); x=ch1:read(); print('got ch1') end,
      function () print(2); x=ch2:read(); print('got ch2') end,
    })
    log('hello '..x);
  end,
  function (id)
    ch2:write('world');
  end
  },{
    ch1=Channel(1),
    ch2=Channel(1),
  })
```

Alt Constructor Example

```
require('Csp')
local sema = Semaphore(0)
Par(
 {function (pid)
    Seq({
      function (pid2) sema:down(); print('1.1 hello pid='..pid..'.'..pid2)
end,
      function (pid2) print('1.2 hello pid='..pid..'.'..pid2) end
    })
  end,
  function (pid)
```

```
    Par({
      function (pid2) sema:up(); print ('2.1 hello pid='..pid..'.'..pid2)
end,
      function (pid2) print ('2.2 hello pid='..pid..'.'..pid2) end
    })
  end},{
})
print('Waiting ..')
```

Semaphore Example

```
require('Csp')
Fork(
 {function (pid)
    Seq({
      function (pid2) print('1.1 hello pid='..pid..'.'..pid2) end,
      function (pid2) print('1.2 hello pid='..pid..'.'..pid2) end
    })
  end,
  function (pid)
    Par({
      function (pid2) print ('2.1 hello pid='..pid..'.'..pid2) end,
      function (pid2) print ('2.2 hello pid='..pid..'.'..pid2) end
    })
  end},{
})
print('Not waiting ..')
loop.start()
```

Fork Example

Meta Data

```
Author: Stefan Bosse
Version: 24.5.2019
```