Verteilte Sensornetzwerke

Mit Datenaggregation und Sensorfusion

PD Stefan Bosse

Universität Bremen - FB Mathematik und Informatik

Synchronisation und Kommunikation

Wie kann mit Coroutinen (Fibers) geschachtelt mehrere Aufgaben aus der Sicht des Programmiermodells "parallel" ausführen?

Welche Synchronisation wird benötigt?

Einführung eines universellen Synchronisationsobjekts für Coroutine und Implementierung von Stanadards

Coroutinen

- Eine Coroutine ist in Lua ein separates Unterprogramm
- Coroutinen werden in Lua niemals parallel (gleichzeitig) ausgeführt
- Eine Coroutine kann eine andere nicht unterbrechen
- Aber: Eine Coroutine kann an beliebiger Stelle im Programmfluss zu einem Scheduler verzweigen (*yield*), der eine andere ruhende Coroutine (re)aktiviert (*resume*)

Coroutinen

```
local co1 = coroutine.create(function ()
  ..
  coroutine.yield()
  ..
end
local co2 = coroutine.create(function ()
  ..
  coroutine.yield()
  ..
end
coroutine.resume(co1)
coroutine.resume(co2)
```

Coroutinen Synchronisation

- Verschachtelte Ausführung von Coroutinen ist ergibt noch keine Synchronisation
- Synchronisation zwischen Coroutinene wäre z.B.:
  - Warten auf Daten
  - Zeitliche Synchronisation (Barrierer, Event)
  - Produzenten-Konsumenten Abstimmung (Semaphore, Collector)
  - Schutz von zusammengesetzten (nicht atomaren) Datenstrukturen (Mutex)

Universelles Synchronisationobjekt

- Einführung eines universellen Synchronisationsbjekts dessen Verhalten durch drei Funktionen gegeben sind:
  - *oninit*: Initialisierung eines Zählers
  - *onawait*: Berechnung eines neues Zählerwertes, Warten und Freigeben von Coroutinen
  - *onsignal*: Berechnung eines neues Zählerwertes, Freigeben von Coroutinen (und ggf.s Datenspeicherung in einer Queue)

Universelles Synchronisationobjekt

- Das universelle *cosync* Objekt bietet mit der *await* Operation die Möglichkeit eine Art Belegung einzuleiten
- Die *signal* Operation ermöglicht eine Art von Freigabe
- Es gibt einen optionalen Zähler (wenn es eine *init* Funktion gibt)
- Die *onawait* und *onsignal* Funktionen bestimmen die Blockierung und Freigabe von Coroutinen (intern mit *yield* und *resume* umgesetzt)
  - Diese beiden Funktionen aktualisieren auch den Zähler (wenn vorhanden)
  - Es werden Tupel aus Rückgabewerte verwendet: $\langle counter, wait, release \rangle$
  - Beiden Funktionen erhalten den aktuellen Zählerwert und die Anzahl bereits wartender Coroutinen als Argumente

Universelles Synchronisationobjekt

```
local co = cosync:new(
oninit? :
  function ()
    return value -- sets counter value
  end |
  value |
  nil
onawait :
  function (counter,waiters)
    return newcounter,wait?,release?
  end |
  function (waiters)
    return wait?,release?
  end
onsignal :
  function (counter,waiters)
    return newcounter,release?
  end |
  function (waiters)
    return release?
  end |
  nil
)
```

Barriere mit Cosync

Eine Barriere dient zur zeitlichen Synchronosation einer Gruppe von Prozessen / Coroutinen.

- Es handelt sich um eine selbstauslösende Synchronisation nur unter Verwendung der +await* Operation
- Die Gruppengröße ist vorher festgelegt
- Jeder Prozess tritt der Barriere und wird blcokiert bis der N-te Prozess beitritt und alle Prozesse freigibt.

Implementierung

```
Barrier = function (N)
  return cosync:new(
    nil,
    function (waiters)
      return waiters~=(N-1),waiters==(N-1)
    end
  )
end
```

Beispiel

```
local ba = Barrier(4)
for i = 1,3 do
  local co = coroutine.create(function ()
    -- wait for new input data
    ba:await()
  end); coroutine.resume(co)
end
.. prepare input data for jobs ..
ba:await()
```

Mutex mit Cosync

- Schutz "kritischer" Codeabschnitte (um Datenkonsistenz zu gewährleisten)
- Immer nur ein Prozess kann einen Mutex Lock besitzen/halten (Invariante)
- Die Operationen *await* und *signal* müssen immer paarweise verwendet werden.
- Ist der Lock vergeben, führen *await* Aufrufe zur Prozessblockierung (oder einem *yield*)

Implementierung

```
Mutex = function ()
  return cosync:new(
    function () return 1 end,
    function (counter,waiters)
      if counter == 0 then return counter,true
      else return 0,false end
    end,
    function (counter,waiters)
      if waiters then return 0,1
      else return 1,0 end
    end
  )
end
```

Beispiel

```
local mu = Mutex()
local data = { ... }
for i = 1,3 do
  local co = coroutine.create(function ()
    mu:await()
    .. modifify non-atomic shared data structures ..
    mu:signal()
  end); coroutine.resume(co)
end
```

Semaphore mit Cosync

- Semaphoren dienen klassisch zur Synchronisation von Produzenten- und Konsumentenprozessen
- Der Semaphor besitzt einen Zähler der nicht negativ wird (Invariante)
- Die *await* Operation erniedrigt den Zähler um den Wert 1, es sei denn der Zähler ist Null, dann wird der Aufrufer blockiert
- Die *signal* Operation erhöht den Zählerwert um 1. War er vorher aud Null und gibt es wartende Prozesse, wird einer ausgewählt und freigegebn. Der Zähler bleibt auf Null

Implementierung

```
Semaphore = function (init)
  return cosync:new(
    function ()
      return init
    end,
    function (counter,waiters)
      if counter > 0 then
        return counter-1,false
      else
        return counter,true
      end
    end,
    function (counter,waiters)
      if waiters > 0 and counter == 0 then
        return 0,1  -- +1 consumed by release of waiter
      else
        return counter+1,0
      end
    end
  )
end
```

Beispiel

```
local sema = Semaphore(0)
for i = 1,3 do
  local co = coroutine.create(function ()
    .. produce data ..
    sema:signal()
  end); coroutine.resume(co)
end
for i = 1,3 do
  sema:await()
end
```

Collector mit cosync

- Der Semaphore muss bei einem Produzenten-Konsumenten System mit N Produzenten N-mal dekrementiert werden um auf die Terminierung aller Arbeitsprozesse zu warten
- Den Semaphore kann man einfach in einen Kollektor transformieren der nur noch einen *await* Aufruf benötigt
  - Fusion von Barriere und Semaphor

Implementierung

```
Collector = function (N)
  return cosync:new(
    function ()
      return 0
    end,
    function (counter,waiters)
      if counter == N then return counter,false,true
      else return counter,true,false end
    end,
    function (counter,waiters)
      counter=counter+1
      if counter == N then return counter,true
      else return counter,false end
    end
  )
end
```

Beispiel

```
local coll = Colelctor(3)
for i = 1,3 do
  local co = coroutine.create(function ()
    .. produce data ..
    coll:signal()
  end); coroutine.resume(co)
end
local status = coll:await()
```

Cosync und Timeouts

- Bisher wurde angenommen, dass das Synchronisationsereignus auch eintritt, z.B. dass die Barriere ausgelöst wird
- Gerade bei verteilter Kommunikation können Nachrcihten verloren gehen
  - Wenn der Nachrcihtenempfang mit einer *signal* Operation verknüpft ist, würde u.U. das Ereignes ausbleiben
- Daher führt man Timeouts als autosynchronisierendes (negatives) Ereignis ein

```
local o = cosync:new(...)
local status = o:await(TimeoutInMillseconds)
```

- Tritt das Synchronisationsereignis ein, liefert *await* true zurück, ansonsten false ("Interrupt")

Die Wahl des Zeitfensters bis zum Scheitern einer Synchronisation ist schwierig und kritisch.

- Es besteht immer die Gefahr dass ein Ereignis nach dem Timeout noch auftreten kann (z.B. der Empfang einer Nachricht)
- Ist der Timeout zu klein, steigt die Wahrscheinlichkeit ein Ereignis zu verpassen
- Ist der Timeout zu groß gewählt. bremst dass die Datenverarbeitung aus und es können ggfs. andere Ereignisse verpasst werden (Echtzeitfähigkeit leidet)

Cosync und Daten

- Cosync Objekte dienen zunächst der reinen Coroutinensynchronisation im Kontrollpfad
- Die *signal* Operation kann verwendet werden, um Daten in einer Queue (Liste) zu speichern
- *await* kann die Daten abrufen

```
local o = cosync:new(nil|counter0,..,..)
...
o:signal(mydata1)
o:signal(mydata2)
...
local status,data = o:await(TimeoutInMillseconds)
local status,counter,data = o:await(TimeoutInMillseconds) -- with counter
-- data={mydata1,mydata2}
```

Cosync Objekte Reinitialisieren

- Mittels der *reset* Operation können cosync Objekte zurückgesetzt werden:
  - Zähler wird auf *oninit* Wert gesetzt (optional durch Funktionsargument bestimmt)
  - Datenliste wird gelöscht

```
local o = cosync:new(...)
o:reset(counterinit?)
```

Coroutinen Kommunikation mit CoVNC

- Äquivalent zu *route* LuaOS API, hier aber lokal nur für Coroutinen
- Simulierter Nachrichtenverlust kann mittels optionalen *loss* Parameter eingestellt werden [0-1)

```
local vc = covnc:new({
  loss?=number,
  bidir?=boolean
})
local p = vc:port(id?) -- p.id
p:connect(to)
local data = p:read()
p:send({..},to?)
p:receiver(function (data) .. end)
```