

Shared-memory-Sprachen

- Synchronisationsmittel
 - Locks
 - Semaphoren
 - Condition Locks
- Probleme:
 - Sehr fehleranfällig
 - Programmstruktur gibt Kommunikation zwischen Threads oft nur unzureichend wieder
 - Läßt sich nicht auf verteilte Systeme erweitern

Message-Passing-Sprachen

Grundidee: Kein gemeinsamer Speicher, Austausch über Nachrichten

Programme versenden und empfangen Nachrichten über Kanäle (*Channels*)

- Synchronisation von Speicherzugriff entfällt
- Kausale Ordnung im Programm: Nachricht muss gesendet werden, bevor sie empfangen werden kann

Implementierungen

- Erlang
- Concurrent ML (CML): Erweitert Standard ML um Threads mit Message-Passing
- ...

Erlang

- Funktionale Sprache (keine Zuweisungen)
- Nebenläufig: Parallele Prozesse kommunizieren ausschließlich über Message-Passing
- Verteilt, robust, für große Anwendungen gedacht

Erlang in action

Beispiele für Erlang-Anwendungen:

- Telefonvermittlungssoftware von Ericsson
- SSL-Accelerator von Nortel Networks
- Verteilte Datenbank bei Amazon
- ejabberd

Unser Plan

- Implementieren Grundzüge von Message-Passing selbst
- CML-Erweiterung für scsh
- GUI-Programmierung mit CML

Threads, Prozesse und Zustand

Terminologie: Prozesse teilen sich keinen Zustand, Threads schon

- Erlang hat nur Prozesse

Bei CML ist geteilter Zustand möglich (über globale Variablen)

- Programmieren ohne geteilten Zustand ist in CML Stilfrage und Konvention
- Sprechen bei CML weiterhin über "Threads"

Buch

John Reppy, Concurrent Programming in ML, Cambridge University Press

Steht im Semesterapparat

Grundlegende Überlegungen

- Benennung der Channels
 - Hier: Channels sind Werte
- Wie viele Kommunikationspartner?
 - Erstmal 1 zu 1
- Senden und Empfangen synchron oder asynchron
 - Empfangen ist (fast) immer synchron
 - Senden ist synchron und asynchron sinnvoll

Asynchrone Channel

Analogie: Brief versenden

```
(define-record-type :async-channel
  (really-make-async-channel
    lock messages receive-waiting)
  async-channel?
  (lock async-channel-lock)
  (messages async-channel-messages)
  (receive-waiting async-channel-receive-waiting))
```

`messages` ist Queue mit gesendeten aber noch nicht empfangenen Nachrichten

`receive-waiting` ist Queue mit Threads die auf Nachricht warten

`lock` regelt Zugriff auf Queues

Smart Constructor

```
(define (make-async-channel)
  (let ((lock (make-lock)))
    (really-make-async-channel lock
                               (make-queue)
                               (make-queue))))
```

Einschub: Zellen

Eine Zelle speichert genau einen Wert

Wie ein Array mit einem Element

`(make-cell v)` erzeugt Zelle mit Wert `v`

`(cell-ref c)` liefert Wert, der in Zelle `c` gespeichert ist

`(cell-set! c v)` setzt den Wert von Zelle `c` auf `v`

Nachricht empfangen

Wenn `messages` Nachricht enthält, diese aus der Queue entfernen und zurückgeben

```
(define (receive-async channel)
  (let ((lock (async-channel-lock channel)))
    (obtain-lock lock)
    (if (queue-empty? (async-channel-messages channel))
        ...
        (let ((message (dequeue! (async-channel-messages channel))))
          (release-lock lock)
          message))))))
```

Nachricht empfangen

Andernfalls müssen wir warten

- Lock erzeugen und darauf blockieren
- Zelle erzeugen und zusammen mit Lock in `receive-waiting` ablegen

```
(let ((waiting-lock (make-lock))
      (cell (make-cell #f)))
  (obtain-lock waiting-lock)
  (enqueue! (async-channel-receive-waiting channel)
            (cons waiting-lock cell))
  (release-lock lock)
  (obtain-lock waiting-lock)
  (cell-ref cell))
```

Senden

Wenn kein Thread auf Nachricht wartet, in `messages` einreihen

```
(define (send-async channel message)
  (let ((lock (async-channel-lock channel)))
    (obtain-lock lock)
    (if (queue-empty? (async-channel-receive-waiting channel))
        (begin
          (enqueue! (async-channel-messages channel) message)
          (release-lock lock))
        ...)))
```

Senden

Andernfalls ersten Thread aufwecken und Nachricht übergeben

```
(let* ((pair (dequeue! (async-channel-receive-waiting channel)))
      (waiting-lock (car pair))
      (cell (cdr pair)))
  (cell-set! cell message)
  (release-lock waiting-lock)
  (release-lock lock))
```

Producer/Consumer mit Asynchronen Channels

Naiver Ansatz: Producer sendet an Channel, Consumer empfängt daraus

D.h. Channel ist effektiv der Puffer.

Problem: Wenn Producer schneller ist als Consumer, geht irgendwann der Speicher aus

Idee: Producer muss warten, bis Platz ist

Umsetzung: Producer wartet auf "frei"-Nachricht, Puffer sendet diese, wenn Consumer gelesen hat

Hier: Einelementiger Puffer

Implementierung

```
(define-record-type :buffer
  (really-make-buffer empty-channel
                      insert-channel remove-channel
                      remove-ack-channel)

  buffer?
  (empty-channel buffer-empty-channel)
  (insert-channel buffer-insert-channel)
  (remove-channel buffer-remove-channel)
  (remove-ack-channel buffer-remove-ack-channel))
```

- Lesen aus `empty-channel` möglich, wenn Puffer leer
- Producer sendet Wert an `insert-channel`
- Consumer liest Wert aus `remove-channel`
- Consumer teilt über `remove-ack-channel` mit, dass Wert gelesen (senden ist asynchron!)

Puffer erzeugen

Channel erzeugen und Puffer-Thread starten

Puffer-Thread verwaltet Zustand und synchronisiert

```
(define (make-buffer)
  (let ((empty-channel (make-async-channel))
        (insert-channel (make-async-channel))
        (remove-channel (make-async-channel))
        (remove-ack-channel (make-async-channel)))
    (letrec
      ((empty
        (lambda ()
          (send-async empty-channel #f)
          (full (receive-async insert-channel))))
        (full
        (lambda (value)
          (send-async remove-channel value)
          (receive-async remove-ack-channel)
          (empty))))
      (spawn empty))
    ...))
```

Einfügen

- Warten, bis aus `empty-channel` gelesen werden kann
- Dann (asynchron) senden

```
(define (insert buffer value)
  (receive-async (buffer-empty-channel buffer))
  (send-async (buffer-insert-channel buffer) value))
```

Lesen

- Aus `remove-channel` lesen (blockiert eventuell)
- Beliebigen Wert an `remove-ack-channel` schicken, damit `full` weiß, dass Wert gelesen wurde

```
(define (remove buffer)
  (let ((value
        (receive-async (buffer-remove-channel buffer))))
    (send-async (buffer-remove-ack-channel buffer) #f)
    value))
```

Synchrone Channel

Analogie: Telefonieren

`send` blockiert, bis `receive` auf dem Channel aufgerufen wird

Implementierung: ÜA

Producer/Consumer mit synchronen Channels

Producer und Consumer direkt über synchronen Channel verbinden:

- Producer würde beim Senden blockieren, wenn Consumer noch nicht liest
- (Puffergröße wäre 0)

```
(define-record-type :buffer
  (really-make-buffer insert-channel remove-channel)
  buffer?
  (insert-channel buffer-insert-channel)
  (remove-channel buffer-remove-channel))
```

- Producer schreibt in `insert-channel`
- Consumer liest aus `remove-channel`
- Puffergröße 1

Erzeugen des Puffers

Einfacher als bei asynchronen Channels

Channel übernehmen bereits Synchronisation

```
(define (make-buffer)
  (let ((insert-channel (make-sync-channel))
        (remove-channel (make-sync-channel))))
    (letrec
      ((empty
        (lambda ()
          (full (receive-sync insert-channel))))))
      (full
        (lambda (value)
          (send-sync remove-channel value)
          (empty))))
      (spawn empty))
    (really-make-buffer insert-channel
                        remove-channel)))
```

Einfügen und Empfangen

Einfach aus Channels des Puffers lesen und hineinschreiben

```
(define (insert buffer value)
  (send-sync (buffer-insert-channel buffer) value))

(define (remove buffer)
  (receive-sync (buffer-remove-channel buffer)))
```

Vergleich

Asynchrone Channels

- Vorteil: Weniger Overhead. Aber: in der Praxis nicht so relevant
- Nachteil: Keine Synchronisation, damit schwerer zu handhaben
- Typischer Fehler: Pufferüberlauf (schwer zu erkennen)
- Eingebauter Puffer kann auch nützlich sein

Synchrone Channels

- Vorteil: Synchronisation implizit
- Programme sind leichter zu verstehen
- Typischer Fehler: Deadlock (Leicht zu erkennen)
- Simulation von asynchronem Senden möglich

⇒ CML hat synchrone Channels

Programmierparadigmen für Message-Passing

- Prozessnetzwerke
- Client/Server

Prozessnetzwerke

Strukturiere Programm, so dass die Daten von einem Thread zum nächsten fließen

Prozessnetzwerke bilden Graph:

- Knoten sind die Threads
- Kanten sind Channels
- Threads lesen Eingabe aus Channel und geben Ergebnis in Channel aus

Channels repräsentieren *Streams*: unendlich lange Folgen von Werten

Einfacher Knoten

Liefere Channel mit allen natürlichen Zahlen ab n :

```
(define (make-int-channel n)
  (let ((channel (make-sync-channel)))
    (spawn
      (lambda ()
        (let loop ((i n))
          (send-sync channel i)
          (loop (+ 1 i))))))
  channel))
```

Sieb des Eratosthenes

Stream aller Primzahlen:

- Erzeuge Stream aller Zahlen
- Entferne die Vielfachen der Primzahlen
- übrig bleibt Stream der Primzahlen

Vielfache entfernen

Realisiert als Prozessnetzwerkknoten

```
(define (remove-multiples factor channel)
  (let ((out-channel (make-sync-channel)))
    (spawn
      (lambda ()
        (let loop ()
          (let ((i (receive-sync channel)))
            (if (not (zero? (remainder i factor)))
                (send-sync out-channel i))
            (loop))))))
    out-channel))
```

Entferne aus dem Eingabestrom `channel` alle Vielfachen von `factor`

(`remainder` liefert Rest einer Ganzzahlendivision)

Stream aller Primzahlen

- Lies nächste Primzahl aus
- Entferne die Vielfachen

```
(define (prime-channel)
  (let ((primes (make-sync-channel)))
    (spawn
      (lambda ()
        (let loop ((channel (make-int-channel 2)))
          (let ((factor (receive-sync channel)))
            (send-sync primes factor)
            (loop (remove-multiples factor channel))))))
      primes)))
```

Client/Server-Paradigma

Clienten stellen über Channel Anfragen an Server, die dieser ggf. über anderen Channel beantwortet

Server ist typischerweise als Endlosschleife implementiert

Server hat dabei mehrere Zustände

Beispiel: Zellen

Clients können über `request-channel` 2 Arten von Nachrichten an Server senden

- `:get-request`: Inhalt der Zelle auslesen
- `:put-request`: Inhalt der Zelle neu setzen

Server antwortet bei `:get-request` auf `reply-channel`

Records für Requests und Zelle

```
(define-record-type :get-request
  (make-get-request)
  get-request?)
```

```
(define-record-type :put-request
  (make-put-request value)
  put-request?
  (value put-request-value))
```

```
(define-record-type :cell
  (really-make-cell request-channel reply-channel)
  cell?
  (request-channel cell-request-channel)
  (reply-channel cell-reply-channel))
```

Erzeugen der Zelle

Erzeuge Channels, starte Server-Thread

```
(define (make-cell value)
  (let ((request-channel (make-sync-channel))
        (reply-channel (make-sync-channel)))
    (spawn
      (lambda ()
        (let loop ((value value))
          (let ((request (receive-sync request-channel))
                (cond
                 ((get-request? request)
                  (send-sync reply-channel value)
                  (loop value))
                 ((put-request? request)
                  (loop (put-request-value request)))))))
      (really-make-cell request-channel reply-channel)))
```

Zelle auslesen

`:get-request` senden, auf `reply-channel` auf Antwort warten

```
(define (cell-ref cell)
  (send-sync (cell-request-channel cell)
             (make-get-request))
  (receive-sync (cell-reply-channel cell)))
```

Neuen Wert für Zelle setzen

`:put-request` senden, enthält neuen Wert

```
(define (cell-set! cell value)
  (send-sync (cell-request-channel cell)
             (make-put-request value)))
```

Zusammenfassung

- Bei Message-Passing tauschen Threads Nachrichten explizit anstatt über gemeinsamen Speicher aus
- Wesentlich klarer, wann Threads Daten austauschen
- Empfangen ist immer synchron
- Synchrones Senden ist besser als asynchrones Senden
- Programmierparadigmen:
 - Prozessnetzwerke
 - Client/Server