

Testatblatt

Das nächste Übungsblatt ist das erste Testatblatt, d.h.:

- 2 Wochen Bearbeitungszeit
- Ausgabe morgen auf der Homepage
- Abgabe in Teams bei Christoph
- Alle Teammitglieder müssen die komplette Abgabe erläutern können
- Testate finden am 12.7 statt; Termine morgen in der Ü-Stunde

Bisherige Vorgehensweise

Wie haben wir bisher programmiert?

- Race-Conditions um jeden Preis vermieden
- Nach Zugriffskonflikt: inkonsistenter Zustand nicht erkennbar
- Abstraktionen, die in der Implementierung wieder Locks verwenden

Programmieren mit Locks

Locks sind also unser grundlegendes Synchronisationsmittel.

Nachteile:

- Für höheres Maß an Nebenläufigkeit braucht man mehr Locks
- Höherer Speicherbedarf
- Komplexität des Programmes steigt

Ziemlich viel Aufwand zur Verhinderung einer Situation, die selten eintreten würde.

Idee

Neue Idee:

Mache Zugriffskonflikte im Programm erkenn- und behandelbar

Transaktionen

Transaktion fasst Zustandsänderungen zusammen, die atomar ausgeführt werden sollen.

Lebenslauf einer Transaktion:

- Beginn der Transaktion
- gewünschte Zustandsänderungen aufzeichnen, aber nicht ausführen
- Abschluß der Transaktion

Abschluß der Transaktion

Kritischer Moment: Abschluß der Transaktion (Commit)

Bei Abschluß:

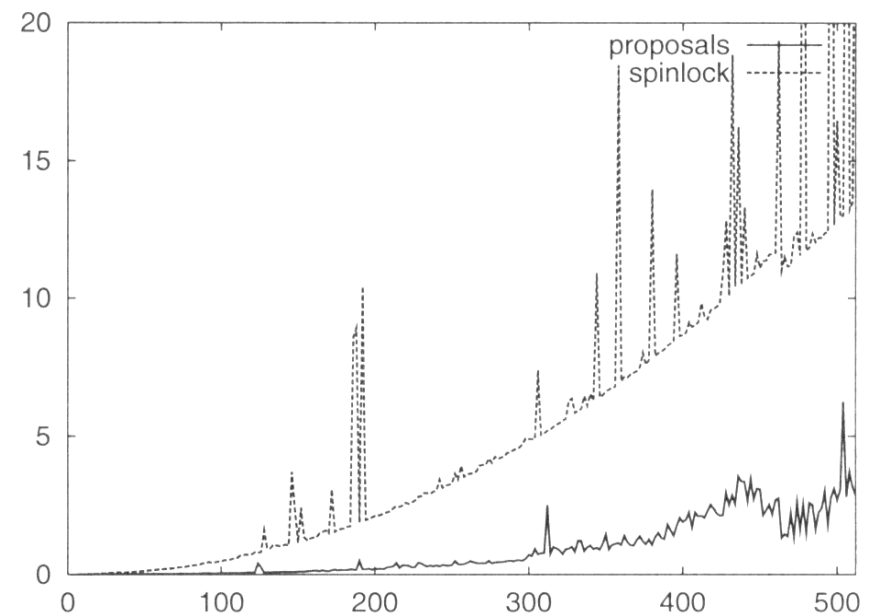
- Überprüfe ob sich relevante Zustandsänderungen (im Gesamtsystem) ergeben haben
- Wenn ja: Wiederholung der Transaktion
- Wenn nein: Mache die aufgezeichneten Änderungen aktiv

Vorteile

Vorteile optimistischer Nebenläufigkeit:

- Bessere Performance
- Transaktionen sind kombinierbar
- Keine explizite Synchronisation nötig

Performance



In der Praxis

Optimistische Nebenläufigkeit wird wichtiger:

- Alternative Namen: (software) transaction(al) memory
- Implementierungen für verschiedene Sprachen
- Beispiele: Scheme 48 1.0, GHC
- Unterschiedliche APIs
- Angekündigt: Implementierung in Hardware

Im folgenden: Scheme 48 API

Proposals

Ein Proposal registriert Lese- und Schreibzugriffe.

- Proposal merkt sich Lese- und Schreibzugriffe auf Datenstrukturen
- Registrierung mit speziellen Selektoren und Mutatoren für Datenstrukturen
- Schreibzugriffe sind vor dem Commit nur im Proposal sichtbar
- Jeder Thread hat ein momentanes Proposal

Commit kann erfolgen, wenn gelesene und geschriebene Datenstrukturen gegenüber dem Zeitpunkt der Registrierung unverändert sind.

with-new-proposal

```
(with-new-proposal (lose) body ...)
```

- Speichert das momentane Proposal und installiert es nach Auswertung wieder
- Erzeugt neues Proposal für die Auswertung von `body`
- `lose` ist eine Prozedur, die `body` nochmal mit einem frischen Proposal auswertet

Beispiel:

```
(define (call-atomically! thunk)
  (with-new-proposal (try-again)
    (thunk)
    (or (maybe-commit) (try-again))))
(values))
```

maybe-commit und atomically!

```
(maybe-commit)
```

Prüft, ob das momentane Proposal des Threads umgesetzt werden kann, tut dies ggf. und gibt `#t` bzw. `#f` zurück.

```
(atomically! body ...)
```

Makro-Version von `call-atomically!`

```
(call-atomically thunk) und (atomically body ...)
```

Funktionieren wie die Versionen ohne Ausrufezeichen, geben aber den Rückgabewert des Thunks/letzten Ausdrucks zurück.

Paare

`(provisional-car p)` und `(provisional-cdr p)`

- Registrieren Leseoperation auf dem Paar `p` im momentanen Proposal
- Geben den `car` bzw. `cdr` zurück

`(provisional-set-car! p v)` und `(provisional-set-cdr! p v)`

- Registrieren den Schreibzugriff auf `p` im momentanen Proposal
- Setzen den `car` bzw. `cdr` von `p` auf `v`

`provisional-` Selektoren und Mutatoren gibt es auch für Strings, Vektoren und Zellen

Beispiel 1

Beispiel 1:

```
(let ((p (cons 0 #f)))
  (atomically
    (provisional-set-car! p 1)
    (provisional-car p)))
```

Beispiel 1

Beispiel 1:

```
(let ((p (cons 0 #f)))
  (atomically
    (provisional-set-car! p 1)
    (provisional-car p)))
```

⇒ 1

Beispiel 2

Beispiel 2:

```
(let ((p (cons 0 #f)))
  (atomically
    (provisional-set-car! p 1)
    (car p)))
```

Beispiel 2

Beispiel 2:

```
(let ((p (cons 0 #f)))  
  (atomically  
    (provisional-set-car! p 1)  
    (car p)))
```

⇒ 0

Beispiel 3

Beispiel 3:

```
(let ((p (cons 0 #f)))  
  (with-new-proposal (lose)  
    (provisional-set-car! p 1)  
    (or (maybe-commit) (lose))  
    (car p)))
```

Beispiel 3

Beispiel 3:

```
(let ((p (cons 0 #f)))  
  (with-new-proposal (lose)  
    (provisional-set-car! p 1)  
    (or (maybe-commit) (lose))  
    (car p)))
```

⇒ 1

invalidate-current-proposal!

```
(invalidate-current-proposal!)
```

- Erklärt das momentane Proposal für ungültig
- D.h. ein (maybe-commit) würde #f liefern

call-ensuring-atomicity

```
(call-ensuring-atomicity thunk)
```

Benutze momentanes Proposal, wenn vorhanden:

```
(define (call-ensuring-atomicity thunk)
  (if (current-proposal)
      (thunk)
      (call-atomically thunk)))
```

Auch als Makro-Version vorhanden:

```
(ensuring-atomicity body ...)
```

Zähler mit Locks

Ein einfacher synchronisierter Zähler mit Locks:

```
(define (make-counter)
  (let ((value 0))
    (lambda ()
      (set! value (+ value 1))
      value)))

(define safe-counter
  (let ((lock (make-lock))
        (counter (make-counter)))
    (lambda ()
      (with-lock lock (counter))))))
```

Eine "kleine" Erweiterung

Aufgabe:

- Eine Liste von Zählern (beliebige Anzahl)
- **step-counters!** läuft atomar ab und erhöht alle Zähler in der Liste
- Wiederverwendung des implementierten Zählers

Eine "kleine" Erweiterung

Aufgabe:

- Eine Liste von Zählern (beliebige Anzahl)
- **step-counters!** läuft atomar ab und erhöht alle Zähler in der Liste
- Wiederverwendung des implementierten Zählers

Lösung: Ein Lock für die gesamte Liste.

Eine "kleine" Erweiterung

Aufgabe:

- Eine Liste von Zählern (beliebige Anzahl)
- `step-counters!` läuft atomar ab und erhöht alle Zähler in der Liste
- Wiederverwendung des implementierten Zählers

Lösung: Ein Lock für die gesamte Liste.

Nachteil: Grad der Nebenläufigkeit möglicherweise zu stark eingeschränkt

Beobachtung: Atomare Operationen können nicht kombiniert werden

Beispiel: Zähler

Ein synchronisierter Zähler mit Proposals

```
(define (make-counter)
  (let ((value (make-cell 0)))
    (lambda ()
      (ensure-atomicity
       (lambda ()
         (let ((v (+ (provisional-cell-ref value)
                    1)))
           (provisional-cell-set! value v)
           v)))))))
```

Proposals kombinieren

```
(define (step-counters! . counters)
  (ensure-atomicity
   (lambda ()
     (for-each (lambda (counter)
                 (counter))
               counters))))
```

Proposals kombinieren

```
(define (step-counters! . counters)
  (ensure-atomicity
   (lambda ()
     (for-each (lambda (counter)
                 (counter))
               counters))))
```

`ensure-atomicity` verwendet das momentane Proposal (falls vorhanden)

Damit wird diese Transaktion in eine größere Transaktion eingebettet

⇒ Transaktionen können miteinander kombiniert werden

Records mit Proposals

```
(define-synchronized-record-type ...)
```

- Definiert Record-Typ; sehr ähnlich zu `define-record-type`
- Felder automatisch mit momentanen Proposal verknüpfen

Beispiel:

```
(define-synchronized-record-type pare :pare
  (kons kar kdr)
  (kar kdr)
  pare?
  (kar kar set-kar!)
  (kdr kdr set-kdr!))
```

queue.scm (1)

```
(define-synchronized-record-type queue :queue
  (really-make-queue head tail)
  (head tail)
  queue?
  (head real-queue-head set-queue-head!)
  (tail queue-tail set-queue-tail!))
```

```
(define (make-queue)
  (really-make-queue '() '()))
```

```
(define (queue-empty? q)
  (null? (real-queue-head q)))
```

queue.scm (2)

```
(define (enqueue! q v)
  (ensure-atomicity!
   (let ((p (cons v '())))
     (cond ((null? (real-queue-head q))
            (set-queue-head! q p))
           ((null? (queue-tail q))
            (invalidate-current-proposal!))
           (else
            (set-cdr! (queue-tail q) p)))
     (set-queue-tail! q p))))
```

queue.scm (3)

```
(define (dequeue! q)
  (ensure-atomicity
   (let ((pair (real-queue-head q))
         (cond ((null? pair)
                (error "empty queue" q))
              (else
               (let ((value (car pair))
                     (next (cdr pair)))
                 (set-queue-head! q next)
                 (if (null? next)
                     (set-queue-tail! q '())
                     value)))))))
```