Software Transactional Memory

Robert Kupferschmied, Matrikel 59112

December 12, 2013

Inhaltsverzeichnis

Inhaltsverzeichnis

Prozess, Thread, Spark

- Prozess: Algorithmisch ablaufende Informationsverarbeitung
  Prozesse können unterbrochen werden

Bereit:	Besitzt alle Ressourcen, wartet auf Prozessorzuordnung
Laufend:	Ist aktuell dem Prozessor zugeordnet
Wartend:	Ist durch Betriebssystem unterbrochen wartet auf Betriebsmittel

- Thread: Leichtgewichtiger Prozess
  Teilt sich Betriebsmittel mit anderen Threads,
  besitzt eigenen Stack
- Spark: leichtgewichtiger Thread, wird vom Laufzeitsystem gestartet

Verschränkung

Threads können gleichzeitg auf gleiche Betriebsmittel zugreifen.

Java Code	Pseudo Assembler
i++;	mov a,0x00 //Adresse der Variable inc a mov 0x00,a

Anzahl möglicher Verschränkung

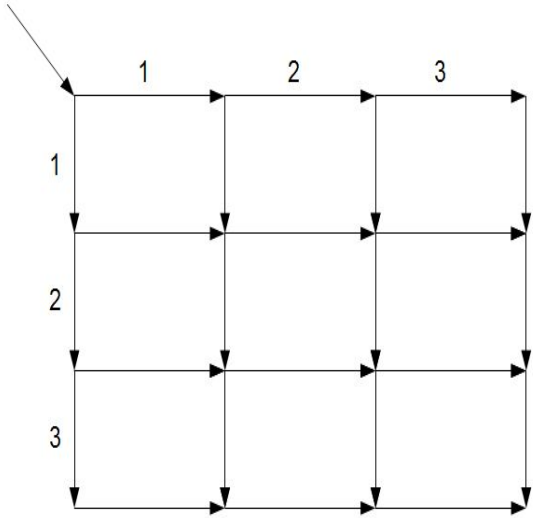

Figure: Mögliche Wege

- $6! = 24$ mögliche Wege
- Testen ist sehr aufwendig!!!

Threadsynchronisation

Blockierung

- Implementation durch Semaphore
- von Dijkstra konzipiert

Semaphor I -> 2 Operationen:

1. I.lock
2. I.unlock

In Java:

```
Object lock;
Synchronized(lock){ ... };
```

Mutex Variablen in Haskell

Mutex = mutual exclusion = Wechselseitiger Ausschluss

- takeMVar :: MVar a -> IO a – Block when MVar is empty
- putMVar :: MVar a -> a -> IO () –Block when MVar is full
- tryTakeMVar :: MVar a -> IO (Maybe a) –Noneblocking
- tryPutMVar :: MVar a -> a -> IO Bool –Noneblocking

Deadlock / Dining Philosophers

Atomare Operationen/Feingranular

- atomar = unteilbar

Atomic-Bibliothek in Java

Bsp:

```
AtomicLong al = new AtomicLong(0);
al.getAndIncrement(); //ist ein atomarer Maschinenbefehl
```

Software Transactional Memory

- Technik zur Vereinfachung von nebenläufiger Programme
- Erlaubt Operationen die den Status von Variablen ändern
- Diese Änderungen werden atomar ausgeführt
- Man kann mehrere Operationen zu atomaren Blöcken zusammenfassen
- Die Transaktionen werden durch Transaktionsmanager durchgeführt

Ablauf einer Transaktion

- Transaktionen sind atomare Blöcke
- Es gibt einen Transaktionsmanager
- Transaktionsmanager besitzt einen Log
- Daten werden in Log geschrieben
- Im Log werden die Veränderungen gespeichert
- Nach Transaktion werden die Änderungen im Log im Hauptspeicher atomar übernommen ("commit", vgl. SQL)
- Es wird nur ein "commit" ausgeführt wenn Daten mit Log konsistent mit HS, ansonsten muss Transaktion wiederholt werden

Die Bibliothek - Control.Concurrent.STM

- atomically:: STM a -> IO a
- newTVar:: a -> STM (TVar a)
- readTVar:: TVar a -> STM a
- writeTVar:: TVar a -> a -> STM ()
- retry:: STM a
- orElse:: STM a -> STM a -> STM a
- throwSTM:: Exception e => e -> STM a
- catchSTM:: Exception e => STM a -> (e -> STM a) -> STM a

Eine Transaktion

```
atomInc :: Num a => TVar a -> STM ()
atomInc x = do
    i <- readTVar x – lesen des aktuellen Status
    writeTVar x (i+1) – Veränderung im Log
main = do
    shared <- atomically $ newTVar 0
    atomically $ atomInc shared
```

- atomically ruft den Transaktionsmanager auf
- Keine IO Aktion in Transaktion möglich

Blocking – Retry

retry :: STM a

- retry blockiert den aktuellen Spark bis sich die die Variablen ändern

Bsp:

```
takeForks f1 f2 = atomically $ do
    b1 <- readTVar f1
    b2 <- readTVar f2
    if(b1 && b2) then( do
    writeTVar f1 False
    writeTVar f2 False) else retry
```

orElse

```
orElse :: STM a -> STM a -> STM a
```

Wirkung:

1. Die erste Transaktion wird durchgeführt. Falls sie ein Ergebnis zurückliefert, endet orElse
2. Falls die erste Transaktion retry aufruft. Wird die zweite Transaktion durchgeführt

Yesod und STM

- STM ist eine eigenständige Bibliothek
- Webserver sind parallele Programme
- Jeder Client bekommt sein eigenen Thread
- Vorteilhaft gemeinsamen Speicher über STM zu synchronisieren:
  1. Keine/kaum Deadlocks
  2. Keine Verschränkungseffekte

Vor- und Nachteile

STM bietet eine Reihe von Vorteilen.

1. zusammensetzbare Atomare Blöcke
2. nicht Blockierend

Frage: Warum brauche ich noch MVars?

- takeMVar ist schneller als takeTMVar
- Code wird aber nicht automatisch schneller durch die Benutzung von MVars!!!

Fairness

- MVars sind fair
- Mehrere Threads greifen auf MVar zu, dann werden die Threads im Fifo Prinzip abgearbeitet
- STM hat diese Möglichkeit nicht
- Hier werden alle wartenden Threads "aufgeweckt", wenn sich die Variable ändert
- Keine Echtzeitanwendungen!!!

Kosten von Transaktionen

writeTVar	übernimmt nur Änderung im Log
readTVar	muss das Protokoll durchgehen -> Prüfung ob es von früherem writeTVar beschrieben wurde readTVar hat Kosten von O(n), n = Anz. der Log Einträge
retry	bracht den Log, hier werden readTVars ausgewertet
commit	1.(Nicht Konsistent) Verwerfen einer Transaktion ist billig 2.(Konsistent) Sperren aller beteiligten TVars und schreiben des Logs

Faustregeln zur Verbesserung

1. Immer eine feste Anz. von TVars lesen. Sonst enstehen Kosten von $O(n^2)$.
2. Transaktionen sollten gleich lang sein.

Bsp:
atomically \$ mapM takeTMVar ts //Kosten $O(n^2)$
mapM (atomically . takeTMVar) ts //Kosten n*O(n)

Hardware Transactional Memory

Problem: Prozessor-Cache behindert Parallelität.

- Transaktionsmanager muss Log in HS auswerten.

Idee:

- jeder Prozessor erhält 2 Caches, einen für Transaktionen und einen regulären, diese sind exklusiv
- transaktionale Cache erhält zusätzliche Logik, die commit und retry erleichtern
- vorläufiges Schreiben nicht im HS sichtbar.

Quellen

- http://chimera.labs.oreilly.com/books/1230000000929/ch10.html#sec_stm-cost
- https://github.com/simonmar/parconc-examples
- https://github.com/yesodweb/yesod/wiki/Keeping-(in-memory)-state-with-yesod
- http://de.wikipedia.org/wiki/Transactional_Memory
- http://rosettacode.org/wiki/Dining_philosophers
- http://www.cs.brown.edu/ mph/HerlihyM93/herlihy93transactional.pdf

Zusammenfassung

Pro:

- Umfangreiche Bibliothek zur Threadsynchronisierung
- einfache Benutzung, ohne Deadlocks oder Verschränkungseffekte
- Möglichkeit atomare Blöcke zu definieren

Contra:

- STM ist nicht fair
- Lange Transaktionen committen nie.

weitere Beispiele

- STM Channel
- Raucherproblem (TMVars)
- STM-Scala