Software Transactional Memory

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 19. Februar 2021

Übersicht

1. Einleitung
2. ACID-Eigenschaften
3. Operationen von TM-Systemen
4. Merkmale von TM-Systemen
5. Korrektheitsbegriffe für STM-Systeme
6. Der TL2-Algorithmus

Transactional Memory

- Relativ neuer Programmieransatz (erste Erwähnung ca. 1995, intensivere Forschung ca. 2005)
- Ziel: Programmierer schreibt mehr oder weniger sequentiellen Code
- System garantiert korrekten nebenläufigen Ablauf (z.B. keine ungewollten Deadlocks, keine Race Conditions, etc.)

Motivation: Beispiel

- Überweisungen zwischen Konto A und B
- Lösung: Sperre Konten entsprechend einer totalen Ordnung

Motivation: Beispiel

- Überweisungen zwischen Konto A und B
- Lösung: Sperre Konten entsprechend einer totalen Ordnung
- Erweiterung: Buche nur dann ab, wenn Konto gedeckt

Motivation: Beispiel

- Überweisungen zwischen Konto A und B
- Lösung: Sperre Konten entsprechend einer totalen Ordnung
- Erweiterung: Buche nur dann ab, wenn Konto gedeckt
- Lösung 1: Abort und Restart: Wenn Konto nicht gedeckt, starte von neuem. Birgt die Gefahr, immer wieder neu zu starten.

Motivation: Beispiel

- Überweisungen zwischen Konto A und B
- Lösung: Sperre Konten entsprechend einer totalen Ordnung
- Erweiterung: Buche nur dann ab, wenn Konto gedeckt
- Lösung 1: Abort und Restart: Wenn Konto nicht gedeckt, starte von neuem. Birgt die Gefahr, immer wieder neu zu starten.
- Lösung 2: Warte bis Konto gedeckt (Sperren müssen aufgehoben werden!)

Motivation: Beispiel

- Überweisungen zwischen Konto A und B
- Lösung: Sperre Konten entsprechend einer totalen Ordnung
- Erweiterung: Buche nur dann ab, wenn Konto gedeckt
- Lösung 1: Abort und Restart: Wenn Konto nicht gedeckt, starte von neuem. Birgt die Gefahr, immer wieder neu zu starten.
- Lösung 2: Warte bis Konto gedeckt (Sperren müssen aufgehoben werden!)
- Fazit: Kleine Erweiterungen erfordern große Änderungen

Lock-basierte Programmierung ist schlecht . . .

(Argumente von S. L. Peyton Jones)

- Setzen zu weniger Locks: Programmierer vergisst eine Sperre zu setzen

  Folge: Race Condition

- Setzen zu vieler Locks: Programmierer übervorsichtig:

  Folgen:
  - Programm unnötig sequentiell (Best-Case)
  - Deadlock (Wort-Case)

- Setzen der falschen Locks:
  - Beziehung zwischen Sperren und Daten kennt nur der Programmierer.
  - Compiler oder andere Programmierer kennen die Beziehung evtl. nicht.

Lock-basierte Programmierung ist schlecht ... (2)

- Setzen von Locks in der falschen Reihenfolge:
  - Total-Order-Theorem kann nur eingehalten werden, wenn jeder Programmierer weiß, wie die Ordnung der Locks aussieht.
- Fehlerbehandlung schwierig, da bei Fehlerbehandlung Locks entsperrt werden müssen.
- Vergessene `signal`-Operationen oder vergessenes erneutes Prüfen von Bedingungen führt zu fehlerhaften Systemen.

Ein schlagendes Argument

Lock-basierte Programmierung ist nicht modular

Beispiel:

- Buche von Konto A_1 oder A_2 auf Konto B, je nachdem welches Konto gedeckt ist.

Ein schlagendes Argument

Lock-basierte Programmierung ist nicht modular

Beispiel:

- Buche von Konto A_1 oder A_2 auf Konto B, je nachdem welches Konto gedeckt ist.
- kann nicht aus den bestehenden Programmen zusammengesetzt werden
- sondern erfordert neues Programm

Transactional Memory: Idee

- Datenbanksysteme sind nebenläufig
- Aus Anwendersicht jedoch: Kein Sperren o.ä. nötig
- Anwender schreibt Anfragen, die als Transaktionen auf der Datenbank ausgeführt werden
- Idee: Übertrage dieses Modell für die nebenläufige Programmierung
- Transaktionen auf dem gemeinsamen Speicher

Transactional Memory: Stand der Technik

- Es gibt einige (prototypische) Implementierungen
- Es gibt Software Transactional Memory aber auch Hardware Transactional Memory
- Jede Menge Designunterschiede
- Transaktionsverwaltung ist Zeit- und Platzintensiv

ACID-Eigenschaften

Zunächst betrachten wir Datenbanktransaktionen.

Diese müssen die ACID-Eigenschaften erfüllen:

- Atomicity: Alle Operationen einer Transaktion werden durchgeführt, oder keine Operationen wird durchgeführt.

  Verboten: Operation schlägt fehl, aber Transaktion erfolgreich

  Verboten: fehlgeschlagene Transaktion hinterlässt beobachtbare Unterschiede

  Eine erfolgreiche Transaktion commits, eine fehlgeschlagene Transaktion aborts

ACID-Eigenschaften (2)

- Consistency: Eine Transaktion verändert den Zustand der Datenbank.

  Diese Änderung muss konsistent sein.

  Konsistenz einer commited Transaktion hängt von der jeweiligen Anwendung ab. (z.B. der Kontostand eines Bankkontos darf nicht beliebig groß negativ sein, oder ein neu hinzugefügtes Konto muss eine eindeutige Kontonummer erhalten, usw.).

ACID-Eigenschaften (3)

- Isolation: Eine Transaktion liefert ein korrektes Resultat unabhängig davon, wieviele weitere nebenläufige Transaktionen durchgeführt werden.
- Durability: Das Ergebnis einer commited Transaktion ist permanent. D.h. es wird auf der Festplatte oder ähnliches permant geschrieben, bevor die Transaktion als commited gekennzeichnet werden darf.

Speichertransaktionen

- A,C,I wünschenswert
- Durability nicht möglich, da Hauptspeicher flüchtig.
- Atomarität nur gewährleistet, wenn alle Operationen als Transaktionen durchgeführt werden.

Weitere Anforderungen an TM:

- Datenbanken können Zugriffszeiten auf Festplatten mit Rechenzeit gegenrechnen, im Hauptspeicher geht das nicht, da Zugriffszeiten viel kürzer
- TM muss in bestehende Programmiersprachen integriert werden.

Basisoperationen von TM

`atomic`-Blöcke:

```
atomic {
    Code der Transaktion
}
```

- Code wird als Transaktion durchgefüh,rt
- Vorteil gegenüber Monitoren: Variablen müssen nicht explizit aufgezählt werden.
- Problem: Was passiert wenn gleiche Variable auch von außerhalb geändert wird?

Atomarität ist nicht garantierbar

Transaktionen müssen abbrechen oder comitten, aber:

```
atomic {
    while True do skip;
}
```

Deswegen andere Semantik (anders als bei Datenbanken):

Die Ausführung einer Transaktion (atomaren Blocks) hat drei mögliche Ergebnisse:

- commit, wenn die Transaktion erfolgreich war
- abort, wenn die Transaktion abgebrochen wurde
- undefiniert, wenn die Transaktion nicht terminiert

Abbrechen von Transaktionen

Transaktionen brechen ab, wenn

- ein Konflikt auftritt und der Transaktionsmanager auf Abbruch (und Roll-Back) entscheidet.

  Verhalten normalerweise: Manager versucht die Transaktion erneut durchzuführen

- ein expliziter `abort`-Befehl aufgerufen wird.

  Verhalten normalerweise: Transaktion wird abgebrochen

```
atomic {
    Guthaben := Guthaben + Zins;
    if Guthaben < 0 then abort;
}
```

Semantik von atomic

- Programm verhält sich, alsob für alle `atomic`-Blöcke ein globaler Lock verwendet wird.
- Zugriff auf Transaktionsvariablen außerhalb von Transaktionen kann beliebigen Unsinn anstellen
- Schreibender Zugriff: Variable wird verändert
- Lesender Zugriff: Kann Werte beobachten, die noch nicht comitted sind!

Basisoperationen (2)

Der `retry`-Befehl

- Ermöglicht es Transaktionen zu koordinieren
- `retry`: Transaktion wird abgebrochen (Roll-back) und erneut gestartet

Beispiel: Bounded Buffer:

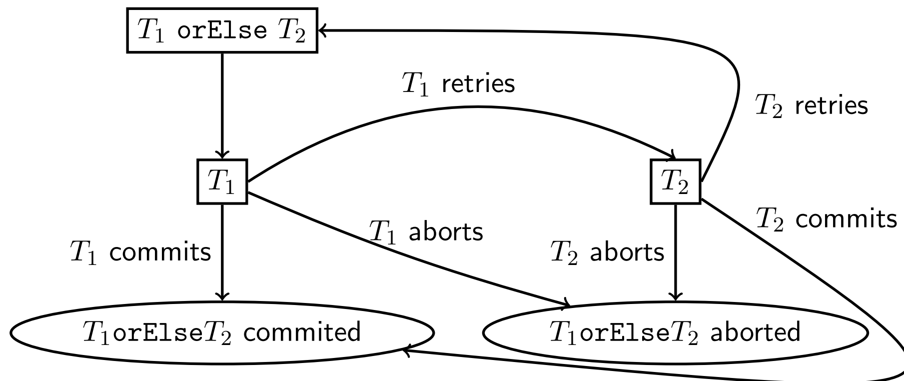
```
atomic{
  if isEmpty(buffer) then retry;
  Lese erstes Element des Puffers usw.
}
```

Basisoperationen (3)

Der `orElse`-Befehl

- Gibt Alternativen vor, wenn Transaktionen abbrechen
- T_1 `orElse` T_2.
  - Zunächst wird T_1 durchführt. Falls T_1 commited oder T_1 explizit via Befehl abgebrochen wird, dann wird T_2 nicht ausgeführt. Die Transaktion T_1`orElse`T_2 commited oder aborts, je nachdem was die Ausführung von T_1 macht.
  - Falls T_1 durch den Transaktionsmanager abgebrochen wird oder `retry` ausgeführt wird, dann startet dieser T_1 nicht neu, sondern versucht T_2 durchzuführen.
  - Falls T_2 explizit aborted oder commited, dann ist die gesamte Transaktion beendet.
  - Falls T_2 ein `retry` durchführt (oder vom System abgebrochen wird), dann wird wieder mit T_1`orElse`T_2 gestartet.

orElse-Schaubild



orElse-Beispiel

```
atomic {
  {
    B := B + Betrag;
    A1 := A1 - Betrag;
    if A1 ≤ 0 then retry;
  }
  orElse
  {
    B := B + Betrag;
    A2 := A2 - Betrag;
    if A2 ≤ 0 then retry;
  }
}
```

Merkmale von TM-Systemen: Weak / Strong Isolation

Weak Isolation:

- Speicherplätze können auch außerhalb von Transaktionen manipuliert werden

Strong Isolation:

- Trennung in Transaktionsvariablen und andere Variablen.
- System schützt den Zugriff auf Transaktionsvariablen: Z.B. durch
  - Automatisches Einfügen von `atomic`-Blöcken durch den Compiler
  - Typsystem verbietet Zugriff auf Transaktionsvariablen von außerhalb (z.B. in Haskell)

Merkmale von TM-Systemen: Geschachtelte Transaktionen

```
x := 1;
atomic {
  x := 2;
  atomic {
    x := 3;
    abort;
  }
}
```

Was soll eine solche Schachtelung bedeuten?

Drei Varianten: geglättete, geschlossene, offene Transaktionen

Merkmale von TM-Systemen: Geschachtelte Transaktionen (2)

Geglättete Transaktionen:

Verhalten, alsob das innere atomic Statement fehlt.

```
x := 1;
atomic {
   x := 2;
   atomic {
      x := 3;
      abort;
   }
}
```

verhält sich wie:

```
x := 1;
atomic {
   x := 2;
   x := 3;
   abort;
}
```

Das Beispiel ergibt abort (x=1).

Merkmale von TM-Systemen: Geschachtelte Transaktionen (3)

Geschlossene Transaktionen:

- Verhalten wie bei geglätteten Transaktionen, aber:
- Innerer Abbruch führt nicht zum Abbruch der äußeren Transaktion.

```
x := 1;
atomic {
  x := 2;
  atomic {
    x := 3;
    abort;
  }
}
```

verhält sich wie

```
x := 1;
atomic {
  x := 2;
}
```

(da innere Transaktion abbricht)

Beispiel ergibt x=2.

Merkmale von TM-Systemen: Geschachtelte Transaktionen (4)

Offene Transaktionen:

- Innere Transaktionen sind eigenständig.
- Sobald innere Transaktion committed, ist deren Effekt für alle sichtbar und bleibt sichtbar, selbst dann, wenn die äußere Transaktion abbricht

```
x := 1
atomic {
  x:=2;
  atomic {
    x:= 3;
  }
  abort;
}
```

Ergebnis: x = 3!

Merkmale von TM-Systemen: Granularität

- Welche Einheiten werden auf Konflikte überwacht?
- Wort-/ Blockgranularität: Wenn paralleler Zugriff auf Speicherworte, dann Konflikt
- Objektgranularität: Paralleler Zugriff auf ein Objekt führt zum Konflikt

Beachte bei Objektgranularität: Konflikt auch dann, wenn unterschiedliche Objektattribute geändert werden

Merkmale von TM-Systemen: Direktes / verzögertes Update

Direktes Update

- Ausführende Transaktionen modifizieren Objekte sofort.
- Erfordert Schutz vor gleichzeitigem Zugriff (Concurrency Control)
- Alte Werte werden in einem Log gespeichert
- Abbruch der Transaktion:
  Roll-Back und Recovery zum Wiederherstellen der Daten aus dem Log

Merkmale von TM-Systemen: Direktes / verzögertes Update (2)

Verzögertes Update

- Transaktionen erhalten lokale Kopien der Objekte
- Modifikation zunächst an den Kopien
- Beim Commit: Originale werden durch lokale Kopien ersetzt
- Logging: Wer hat welche Kopien, wer darf committen?

Merkmale von TM-Systemen: Direktes / verzögertes Update (2)

- Direktes Update ist **optimistisch**, verzögertes Update **pessimistisch**
- Optimistisch: Annahme, dass Konflikte selten auftreten
- Pessimistisch: Annahme, dass Konflikte häufig auftreten

Merkmale von TM-Systemen: Zeitpunkt der Konflikterkennung

- **Early:** Beim ersten Zugriff, der zum Konflikt führt
  (Notwendig dafür: Logging, wer wann zugegriffen hat)
- **Late:** Erst beim committen wird geprüft, ob ein Konflikt vorliegt
- **Irgenwann:** Iteratives Prüfen (in Zeitabständen)

Merkmale von TM-Systemen: Konfliktmanagement

- Beim Konflikt: Welche Transaktion wird abgebrochen?
- Viele verschiedene Strategien
- Ziel: Fortschritt des Gesamtsystems
- Dadurch verboten: Breche alle Transaktionen ab

Korrektheitskriterien für STM-Systeme

- Was muss gelten, damit ein TM-System als „korrekt“ gilt?
- Eher umstritten, viele Forschungsarbeiten
- Z.B. Dziuma,Fatourou,Kanellou 2015: Überblick mit 16 verschiedenen Korrektheitskriterien und Beziehungen zwischen diesen:
  - strict-serializability
  - serializability
  - opacity
  - causal-consistency
  - causal-serializability
  - virtual-world-consistency
  - strong-virtual-world-consistency
  - snapshot-isolation

  jeweils in einer eager und einer deferred-update Variante

Korrektheitskriterien für STM-Systeme (2)

Formale Betrachtung der Ausführung:

- Transaktionsausführung:
  Schrittweise Abarbeitung, wobei jeder Schritt
  - einem Aufruf auf einem nebenläufigen Objekt (z.B. atomares Register, CAS-Objekt, ...) und
  - lokalen Berechnungen

  entspricht
- Wir schränken uns auf atomare Register als Objekte ein
- Ausführungen werden als Historien notiert

Historien

Definition:

Eine Historie H ist eine Folge von Ereignissen, wobei ein Ereignis sein kann:

- Transaktion T_i führt Aufruf einer Operation durch:
  - T_i.read(x) (Lesen der transaktionalen Variablen x)
  - T_i.write(x, v) (Beschreiben der transaktionalen Variablen x mit dem Wert v)
  - T_i.commit (T_i möchte committen)
  - T_i.abort (T_i möchte abbrechen)
- Transakton T_i erhält Rückgabe für eine Operation:
  - $T_i.w$ bei read (w ist der gelesene Wert)
  - T_i.Ok bei write, commit
  - $T_i.A_{T_i}$ zeigt Abbruch der Transaktion T_i an

  , (für alle Operationen möglich, einzige mögliche Rückgabe von abort)

Beispiel

Z.B. $x = y = z = 1$ am Anfang

Transaktion 1: $x := y + 1$

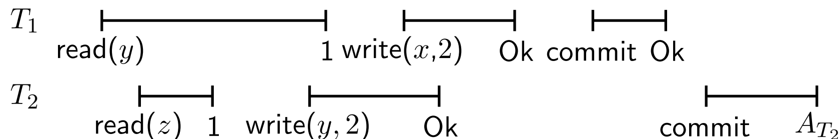
Transaktion 2: $y := z + 1$

Eine mögliche Historie ist:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
$T_2.A_{T_2}$

Beispiel (2)

Darstellung der Historie mit Intervallen entlang der Zeitachse



Vorgehen

- Zunächst: Begriffe und notwendige Notation für Historien
- Danach: Auswahl von Korrektheitsbegriffen mithilfe dieser Formalismen
- Wir stellen nur einige wenige Kriterien vor
  (aus Guerraoui & Kapalka, PPoPP 2008)

Restriktion

Sei H Historie.

- Für Transaktion T ist $H|T$ die Restriktion von H auf T:
  Lösche alle Ereignisse aus H, die nicht zu T gehören.
- Für $T_1, \ldots, T_n$ ist $H|\{T_1, \ldots, T_n\}$ die Restriktion von H auf $T_1, \ldots, T_n$:
  Lösche alle Ereignisse aus H, die nicht zu $T_1, \ldots, T_n$ gehören.
- Für eine transaktionale Variable x ist $H|x$ die Restriktion von H auf x:
  Lösche aller Ereignisse, die nicht zur Variablen x gehören.

Beispiele

Historie H_0:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
$T_2.A_{T_2}$

Beispiele

<u>Historie H_0</u>:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
$T_2.A_{T_2}$

<u>Historie $H_0|T_2$</u>:

~~T_1.read(y)~~
T_2.read(z)
T_2.1
T_2.write(y,2)
~~T_1.1~~
T_2.Ok
~~T_1.write(x,2)~~
~~T_1.Ok~~
~~T_1.commit~~
~~T_1.Ok~~
T_2.commit
$T_2.A_{T_2}$

Beispiele

Historie H_0:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
$T_2.A_{T_2}$

Historie $H_0|T_2$:

T_2.read(z)
T_2.1
T_2.write(y,2)
T_2.Ok
T_2.commit
$T_2.A_{T_2}$

Beispiele

Historie H_0:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
T_2.A_{T_2}

Historie $H_0|T_2$:

T_2.read(z)
T_2.1
T_2.write(y,2)
T_2.Ok
T_2.commit
T_2.A_{T_2}

Historie $H_0|\{T_1, T_2\}$:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
T_2.A_{T_2}

Beispiele

Historie H_0:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
$T_2.A_{T_2}$

Historie $H_0|T_2$:

T_2.read(z)
T_2.1
T_2.write(y,2)
T_2.Ok
T_2.commit
$T_2.A_{T_2}$

Historie $H_0|\{T_1,T_2\}$:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
$T_2.A_{T_2}$

Historie $H_0|y$:

T_1.read(y)
~~T_2.read(z)~~
~~T_2.1~~
T_2.write(y,2)
T_1.1
T_2.Ok
~~T_1.write(x,2)~~
~~T_1.Ok~~
~~T_1.commit~~
~~T_1.Ok~~
~~T_2.commit~~
~~$T_2.A_{T_2}$~~

Beispiele

Historie H_0:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
$T_2.A_{T_2}$

Historie $H_0|T_2$:

T_2.read(z)
T_2.1
T_2.write(y,2)
T_2.Ok
T_2.commit
$T_2.A_{T_2}$

Historie $H_0|\{T_1, T_2\}$:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
$T_2.A_{T_2}$

Historie $H_0|y$:

T_1.read(y)
T_2.write(y,2)
T_1.1
T_2.Ok

Wohl-geformte Historien

Historie H ist wohl-geformt, wenn für jede Transaktion T gilt:

- $H|T$ besteht abwechselnd aus Aufrufen und Rückgaben
- $H|T$ enthält keinerlei Ereignisse nach dem Ereignis $T.A_T$
- Wenn $H|T$ das Ereignis T.commit enthält,
  dann folgt entweder nichts mehr, oder einmal $T.A_T$ oder einmal T.Ok
- Wenn $H|T$ das Ereignis T.abort enthält,
  dann folgt entweder nichts oder einmal $T.A_T$

Wir betrachten fortan nur wohl-geformte Historien!

Beendete, abgebrochene, vollständige, laufende Transaktionen

Klassifizierungen einer Transaktion T in einer Historie H:

- T ist committed in H, wenn $H|T$ mit T.commit, T.Ok endet.
- T ist aborted in H, wenn $H|T$ mit $T.A_T$ endet.
- T ist vollständig in H, wenn T committed oder aborted in H ist.
- T ist laufend in H, wenn T nicht-vollständig in H ist.

Restriktion auf committete Transaktionen:

- $comm(H) := H|\{T_1, \ldots, T_n\}$,
  wobei $T_1, \ldots, T_n$ alle Transaktionen in H sind, die committed sind

Sequentielle Historie

Eine Historie ist sequentiell, wenn alle Ereignisse einer Transaktion hintereinander in einer Teilsequenz stehen.

Beispiele:

Historie H_0:	Historie H_1:	Historie H_2:
T_1.read(y)	T_1.read(y)	T_2.read(z)
T_2.read(z)	T_1.1	T_2.1
T_2.1	T_1.write(x,2)	T_2.write(y,2)
T_2.write(y,2)	T_1.Ok	T_2.Ok
T_1.1	T_1.commit	T_2.commit
T_2.Ok	T_1.Ok	T_2.Ok
T_1.write(x,2)	T_2.read(z)	T_1.read(y)
T_1.Ok	T_2.1	T_1.2
T_1.commit	T_2.write(y,2)	T_1.write(x,2)
T_1.Ok	T_2.Ok	T_1.Ok
T_2.commit	T_2.commit	T_1.commit
$T_2.A_{T_2}$	$T_2.A_{T_2}$	T_1.Ok
nicht sequentiell	sequentiell	sequentiell

Äquivalenz von Historien

Definition

Zwei Historien H, H' sind äquivalent (geschrieben als $H \sim H'$),
wenn für jede Transaktion T gilt: $H|T = H'|T$
(die Historien pro Transaktion sind identisch).

Beispiel

Historie H_1:	Historie H_2:	Historie H_3:	Historie H_4:
T_1.read(x)	T_1.read(x)	T_1.read(x)	T_2.read(x)
T_2.read(x)	T_2.read(x)	$T_1.v$	$T_2.v$
$T_1.v$	$T_2.v$	T_1.write(x,v')	T_2.read(y)
$T_2.v$	$T_1.v$	T_1.Ok	$T_2.v''$
T_2.read(y)	T_1.write(x,v')	T_1.commit	T_2.commit
T_1.write(x,v')	T_1.Ok	T_1.Ok	T_2.Ok
T_1.Ok	T_1.commit	T_2.read(x)	T_1.read(x)
T_1.commit	T_2.read(y)	$T_2.v'$	$T_1.v$
$T_2.v''$	T_1.Ok	T_2.read(y)	T_1.write(x,v')
T_1.Ok	$T_2.v''$	$T_2.v''$	T_1.Ok
T_2.commit	T_2.commit	T_2.commit	T_1.commit
T_2.Ok	T_2.Ok	T_2.Ok	T_1.Ok

Beispiel

Historie H_1:	Historie H_2:	Historie H_3:	Historie H_4:
T_1.read(x)	T_1.read(x)	T_1.read(x)	
		$T_1.v$	
$T_1.v$		T_1.write(x,v')	
	$T_1.v$	T_1.Ok	
	T_1.write(x,v')	T_1.commit	
T_1.write(x,v')	T_1.Ok	T_1.Ok	
T_1.Ok	T_1.commit		T_1.read(x)
T_1.commit			$T_1.v$
	T_1.Ok		T_1.write(x,v')
T_1.Ok			T_1.Ok
			T_1.commit
			T_1.Ok

Beispiel

Historie H_1:	Historie H_2:	Historie H_3:	Historie H_4:
			T_2.read(x)
T_2.read(x)	T_2.read(x)		$T_2.v$
	$T_2.v$		T_2.read(y)
$T_2.v$			$T_2.v''$
T_2.read(y)			T_2.commit
			T_2.Ok
		T_2.read(x)	
	T_2.read(y)	$T_2.v'$	
$T_2.v''$		T_2.read(y)	
	$T_2.v''$	$T_2.v''$	
T_2.commit	T_2.commit	T_2.commit	
T_2.Ok	T_2.Ok	T_2.Ok	

Beispiel

Historie H_1:	Historie H_2:	Historie H_3:	Historie H_4:
T_1.read(x)	T_1.read(x)	T_1.read(x)	T_2.read(x)
T_2.read(x)	T_2.read(x)	$T_1.v$	$T_2.v$
$T_1.v$	$T_2.v$	T_1.write(x,v')	T_2.read(y)
$T_2.v$	$T_1.v$	T_1.Ok	$T_2.v''$
T_2.read(y)	T_1.write(x,v')	T_1.commit	T_2.commit
T_1.write(x,v')	T_1.Ok	T_1.Ok	T_2.Ok
T_1.Ok	T_1.commit	T_2.read(x)	T_1.read(x)
T_1.commit	T_2.read(y)	$T_2.v'$	$T_1.v$
$T_2.v''$	T_1.Ok	T_2.read(y)	T_1.write(x,v')
T_1.Ok	$T_2.v''$	$T_2.v''$	T_1.Ok
T_2.commit	T_2.commit	T_2.commit	T_1.commit
T_2.Ok	T_2.Ok	T_2.Ok	T_1.Ok

- H_1 und H_2 sind äquivalent
- H_1 und H_3 sind nicht äquivalent, H_1 und H_4 sind äquivalent

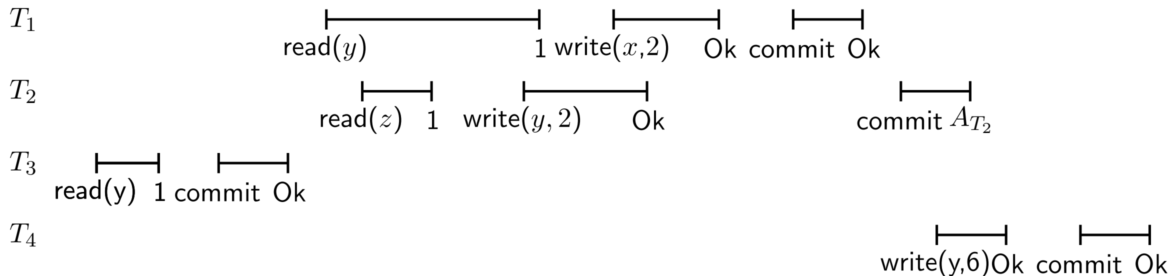
Realzeitordnung der Transaktionen

Definition

Für eine Historie H ist die Realzeit-Ordnung der Transaktionen $\prec_H$ die partielle Ordnung, die definiert ist durch: Wenn T_i vollständig in H ist und das erste Ereignis von T_j liegt nach dem letzten Ereignis von T_i in H, dann gilt $T_i \prec_H T_j$.

Für sequentielle Historien H deren Transaktionen alle vollständig in H sind, ist die Realzeit-Ordnung stets eine totale Ordnung.

Beispiel zur Realzeitordnung



Es gilt:

$$T_3 \prec_H T_1 \qquad T_3 \prec_H T_2 \qquad T_3 \prec_H T_4 \qquad T_1 \prec_H T_4$$

Legale Historien, Teil 1

Definition:

Eine sequentielle Historie H, wobei alle Transaktionen committed sind bis auf möglicherweise die letzte Transaktion ist **legal**, wenn für alle transaktionale Variablen gilt:

$H|x$ stimmt überein mit der sequentiellen Beschreibung von atomaren Registern (die read- und write-Operationen und deren Rückgaben verhalten sich, alsob man sie in einem sequentiellen Programm ausgeführt hätte).

Beispiele

Historie H_5:	Historie $H_5\|x$:	Historie $H_5\|y$:
T_1.read(y)	T_1.write(x,4)	T_1.read(y)
$T_1.3$	T_1.Ok	$T_1.3$
T_1.write(x,4)	T_1.read(x)	T_2.read(y)
T_1.Ok	$T_1.4$	$T_2.3$
T_1.read(x)	T_2.read(x)	
$T_1.4$	$T_2.2$	
T_1.commit		
T_1.Ok		
T_2.read(x)		
$T_2.2$		
T_2.read(y)		
$T_2.3$		

H_5 ist nicht legal!

Legale Historien, Teil 2

Wenn abgebrochene Transaktionen in der sequentiellen Historie enthalten sind, passt der bisherige legal-Begriff nicht.

Daher werden die Historien pro Transaktion zusammen mit den vorherigen erfolgreichen betrachtet

Definition:

Eine sequentielle Historie H, deren Transaktionen allesamt vollständig in H sind, ist legal, wenn für jede Transaktion T_i in H die Historie H_i legal ist, wobei H_i aus H entsteht, indem genau jene Ereignisse $T_j.e$ aus H in H_i übernommen werden mit

- $j = i$, oder
- T_j ist committed in H und $T_j \prec T_i$ in H.

Beachte: H_i erfüllt die Anforderungen an den zuvor definierten Begriff der Legalität: Alle Transaktionen außer evtl. die Transaktion T_i sind in H_i committed.

Beispiel

Historie H_6	Historie $H_{6,1}$	Historie $H_{6,2}$	Historie $H_{6,3}$
T_1.write$(x,3)$	T_1.write$(x,3)$	T_3.read(x)	T_3.read(x)
T_1.Ok	T_1.Ok	$T_3.4$	$T_3.4$
T_1.read(x)	T_1.read(x)	T_3.write$(y,2)$	T_3.write$(y,2)$
$T_1.3$	$T_1.3$	T_3.Ok	T_3.Ok
T_1.abort	T_1.abort	T_3.commit	T_3.commit
$T_1.A_{T_1}$	$T_1.A_{T_1}$	T_3.Ok	T_3.Ok
T_3.read(x)		T_2.read(y)	
$T_3.4$		$T_2.2$	
T_3.write$(y,2)$		T_2.read(x)	
T_3.Ok		$T_2.4$	
T_3.commit			
T_3.Ok			
T_2.read(y)			
$T_2.2$			
T_2.read(x)			
$T_2.4$			

Da $H_{6,1}$, $H_{6,2}$ und $H_{6,3}$ legale sequentielle Historien von committed Transaktionen sind, ist H_6 eine legale vollständige, sequentielle Historie.

Wesentliche Anforderungen einer korrekten STM-Implementierung

- Transaktionen, die committed sind, erscheinen,
  alsob sie unteilbar in einem Schritt durchgeführt wurden
- abgebrochene Transaktionen erscheinen, alsob sie gar nicht durchgeführt wurden.

Weitere Forderungen: Erhaltung der Realzeit-Ordnung

- Der Zeitpunkt zudem die Effekte einer Transaktion erscheinen, liegt irgendwo in der Laufzeit der Transaktion.
- D.h. die Transaktion selbst sollte keine veralteten Speicherzustände lesen.
- Effekt kann z.B. dadurch auftreten, dass durch die Verwendung von Caches noch alte Werte der transaktionalen Variablen gespeichert sind.
- Es sollte gelten: Wenn eine Transaktion T_1 ein Objekt x modifiziert und committed, und danach eine Transaktion T_2 startet und x liest, dann liest T_2 den von T_1 geschriebenen Wert und nicht einen älteren Wert.

Formal definiert man für Historien:

Eine Historie H' erhält die Realzeitordnung einer Historie H, wenn gilt $\prec_H \subseteq \prec_{H'}$.

Weitere Forderungen: Keine inkonsistenten Zustände lesen

- Dürfen laufende Transaktionen, die weder committed noch abgebrochen sind, inkonsistente Speicherzustände lesen, wenn sichergestellt ist, dass sie später sowieso abgebrochen werden?
- Für Datenbanktransaktionen ist sowas erlaubt und unproblematisch
- Aber: STM-Transaktionen laufen nicht in einer völlig abgekapselten Umgebung

Weitere Forderungen: Keine inkonsistenten Zustände lesen

Beispiel:

- Seien x und y zwei (transaktionale) Variablen, welche die Invariante $y = x^2$ und $x \geq 2$ stets erfüllen sollen.
- Der Programmierer erfüllt die Invariante, indem darauf achtet, dass alle Transaktionen sie erfüllen
- Sei $x = 4$ und $y = 16$
- Betrachte nun die Transaktion T_1: $x := 2; y := 4$.
- Wenn eine andere Transaktion T_2 den alten Wert von x (4) und den neuen Wert von y (4) beobachten kann, könnte diese $1/(y - x)$ berechnen und damit einer Laufzeitfehler verursachen.
- Das Abbrechen von T_2 aufgrund des Lesen inkonsistenter Werte verhindert diesen Fehler nicht.

Korrektheitskriterien (1)

Linearisierbarkeit:

- Jede Transaktion T wirkt wie ein atomarer Funktionsaufruf auf den transaktionalen Variablen, der in einem Schritt ausgeführt wird.
- Die erfolgreichen Transaktionen sind dann eine Sequenz solcher Schritte.

Korrektheitskriterien (2)

Sequentialisierbarkeit

Sequentialisierbarkeit fordert, dass für jede Historie H des TM-Systems gilt: Es gibt eine sequentielle und legale Historie S mit $S \sim comm(H)$.

D.h. die Historie der erfolgreichen Transaktionen ist äquivalent zur Historie eines sequentiellen Ablaufs der Transaktionen.

Korrektheitskriterien (3)

Strikte Sequentialisierbarkeit

Strikte Sequentialisierbarkeit fordert, dass für jede Historie H des TM-Systems gilt: Es gibt eine sequentielle und legale Historie S mit $S \sim comm(H)$ und $\prec_{comm(H)} \subseteq \prec_S$.

D.h. zusätzlich zur Sequentialisierbarkeit: Realzeit-Ordnung wird eingehalten, d.h. der gesuchte sequentielle Ablauf muss der Realzeit-Ordnung entsprechen.

Korrektheitskriterien (4)

Alle Begriffe bisher betrachten nur erfolgreiche Transaktionen.

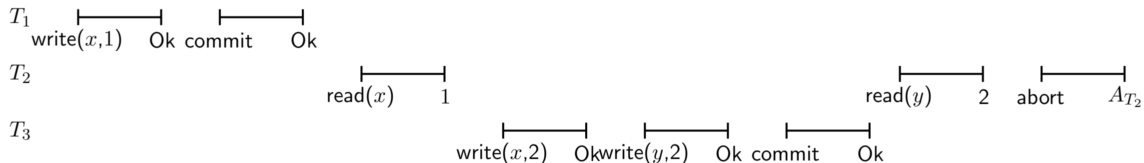
Recoverability

- Wenn eine Transaktion T_i eine transaktionale Variable beschreibt, dann darf keine andere Transaktion T_j die selbe transaktionale Variable lesen, bevor T_i erfolgreich oder abgebrochen ist.

Probleme:

- Erzwingt manchmal zuviel Sequentialisierung der Transaktionsausführung.
- Verbietet nicht, dass abbrechende Transaktionen inkonsistente Zustände sehen.

Beispiel



Obwohl Recoverability gilt, liest T_2 inkonsistente Werte

Korrektheitskriterien (5)

Opazität (opacity): Opazität fordert, dass jede Historie des TM-Systems opak ist:

History H ist opak, wenn es eine vollständige, sequentielle, legale Historie S gibt, die äquivalent zu einer History $H' \in complete(H)$ ist, so dass S die Realzeitordnung von H respektiert.

Dabei ist $complete(H)$ Menge von Historien H', so dass alle Transaktionen die in H vorkommen in H' vollständig sind, wobei

- H' wohl-geformt ist,
- H' enthält gegenüber H nur zusätzliche commit und abort-Ereignisse, und zwar nur für Transaktionen, die laufend sind in H.
- laufende Transaktionen T aus H, für die T.commit in H vorkommt sind committed oder aborted in H'
- laufende Transaktionen T aus H, für die kein T.commit in H vorkommt sind aborted in H'.

Opazität impliziert...

- Die Operationen jeder erfolgreichen Transaktion werden wie atomar in einem Schritt durchgeführt. D.h. insbesondere: Entfernt man alle Schritte von abgebrochenen oder laufenden Transaktionen, so ist die (nebenläufige) Ausführung der erfolgreichen Trransaktionen äquivalent zu einer sequentiellen Ausführung dieser Transaktionen.
  Zusätzlich muss gelten, dass die sequentielle Ausführung die Realzeit-Ordnung einhält.
- Effekte von abgebrochenen Transaktionen sind niemals sichtbar für andere Transaktionen.
- Jede Transaktion (egal ob erfolgreich oder abgebrochen) sieht nur konsistente Zustände, d.h. solche die von committeten Transaktionen entstanden sind, und einem Zustand in der sequentiellen Ausführung entsprechen.

Der TL2-Algorithmus

- Vorgeschlagen von Dice, Shalev und Shavit, 2006
- TL = Transactional Locking
- Implementiert den Transaktionsmanager
- Kein `retry` und `orElse`
- Keine verschachtelten Transaktionen
- Verwendet einen globalen Zähler gc (fetch-and-increment-Objekt)
- Erfüllt Opacity.

TL2: Gemeinsame Datenstrukturen

Die Transaktionalen Variablen $x_1, x_2, \ldots$ werden verwaltet durch:

- Feld TVar, sodass TVar$[i]$ den Inhalt von x_i enthält
- Feld C, sodass C$[i]$ ein Compare-and-Swap-Objekt ist, mit Inhalt (ver,lock) wobei:
  - ver ist Zeitstempel (des globalen Zählers) des letzten Schreibzugriffs
  - lock ist ein Wahrheitswert, der anzeigt, ob der Speicherplatz als gesperrt gilt.

TL2: Lokale Datenstrukturen der Transaktion

- Menge readset: Adressen der gelesenen Speicherplätze
- Menge writeset: Adressen der geschriebenen Speicherplätze
- Feld writelog[i]: enthält neuen Wert für x_i
  Schreiboperationen werden zunächst nur lokal durchgeführt!
- Zeitstempel readver: Zeitstempel des ersten Lesezugriffs der Transaktion
- Menge lockset: Adressen der gesperrten Speicherplätze

TL2: Pseudo-Code (1)

Gemeinsame Datenstrukturen:

gc	(global counter), fetch-and-increment-Objekt, initial gc=0
TVar[1..]	Feld von atomaren Registern, initial TVar[i] $= 0$
C[1..]	Feld von Compare-and-Swap-Objekten mit Paaren als Inhalt, initial C[i]=(0,false)

Lokale Datenstrukturen:

readver	lokaler Zählerstand, initial readver$= \bot$
readset, writeset, lockset	Mengen von Adressen, initial alle $= \emptyset$
writelog[1..]	Feld von Werten

TL2: Pseudo-Code (2)

Operation T_k.read(x_m)

```
if readver = ⊥ then readver := read(gc); // readver beim ersten Lesen setzen
if m ∈writeset then return writelog[m]; // falls lokal geschrieben, diese Version nehmen
(ver1,locked1) := read(C[m]); // CAS-Objekt lesen
result := read(TVar[m]); // Inhalt lesen
(ver2,locked2) := read(C[m]); // CAS-Objekt lesen
 // Abbrechen, wenn lesen nicht atomar, TVar gesperrt, oder Version neuer
if ver1 ≠ ver2 or locked2 or ver2 > readver then Tk.abort();
readset := readset ∪{m}; // Adresse ins readset aufnehmen
return result;
```

TL2: Pseudo-Code (3)

Operation T_k.write(x_m, val)

writeset := writeset $\cup\{m\}$; // Adresse ins writeset aufnehmen
writelog[m] := val; // lokal schreiben
`return` Ok;

Operation T_k.abort()

// alle gesperrten Objekte entsperren
`for all` $m \in$ lockset do
  (ver,locked) := read(C[m]);
  write(C[m],(ver,`False`));
// Initialzustand herstellen
readset,writeset,lockset := $\emptyset, \emptyset, \emptyset$;
readver := $\perp$;
`return` A_{T_k};

TL2: Pseudo-Code (4)

Operation T_k.commit()

```
for all m ∈ writeset do // alle zu schreibenden TVars sperren, und im lockset vermerken
    (ver,locked) := read(C[m]);
    if locked then T_k.abort();
    lock := compare-and-swap(C[m],(ver,False),(ver,True));
    if not lock then T_k.abort(); // Abbruch, wenn Sperren nicht erfolgreich
    lockset := lockset ∪{m};
writever := 1+fetch-and-increment(gc);// counter hochzählen für neue Schreibeversion
if writever ≠ readver+1 then// readset validiieren (für readver+1=writever nicht notwendig)
    for all m ∈ readset do
        (ver,locked) := read(C[m]);
        if ver > readver or (locked and m ∉ writeset) then T_k.abort();
for all m ∈ writeset do // Neuen Inhalt schreiben und entsperren
    write(TVar[m],writelog[m]);
    write(C[m],(writever,False))
readset,writeset,lockset := ∅, ∅, ∅;
readver := ⊥;
return Ok;
```

TL2: Bemerkungen

- Prüfen des readset und Sperren des writeset stellt sicher, dass der Effekt der Transaktion wie eine Funktion wirkt
- TL2 verwendet wenige Sperren: Lock wird nur beim Committen gesetzt.
- Erzeugen neuer Speicherplätze: Einfach durch ein weiteres Set.

TL2: Bemerkungen (2)

Nur lesende Transaktionen

- Können auch auf das Readset verzichten
- Sie validiieren die Gültigkeit der Lesezugriff nur durch den Zeitstempel readver und den Vergleich von readver mit ver_2 bei jedem Lesezugriff.
- Optimierung: Behandele jede Transaktion zunächst wie eine nur lesende, beim ersten Schreiben: Neustart und lese/schreibe Transaktion

TL2: Semantischer Fehler bei bedingter Nichtterminierung

Sei `x` = True am Anfang

<table>
<tr><td><u>Transaktion 1</u></td><td><u>Transaktion 2</u></td></tr>
<tr><td>

```
if x then
  loop forever
else return
```

</td><td>

`x` := False;

</td></tr>
</table>

Wenn Transaktion 1 zuerst läuft, wird diese in eine Endlosschleife laufen, obwohl nach Ablauf von Transaktion 2, keine Endlosschleife notwendig.

Mögliche Abhilfe: Iteratives Prüfen des readset auf Konflikt.

Fazit

- Leichtere Programmierung mit TM
- Problem: Transaktionen können nur Operationen sicher verarbeiten, die umkehrbar sind
- Z.B. nicht umkehrbar: Drucke „Hallo“, „Launch Missiles“ etc.
- Weitere Probleme: Code in Transaktionen zerlegen:
  Kleine Transaktionen sind besser als große (wegen Konfliktpotential)
  Man muss herausfinden:
  Welcher Codeabschnitt muss atomar durchgeführt werden?
  Datenstrukturen: Z.B. ein Baum in einer transaktionalen Variable vs. ein Knoten in einer transaktionalen Variablen
- Wir sehen noch Programmierbeispiele beim Thema Haskell STM