12. SC und TSO

Beobachtung: Die bisherigen Betrachtungen gingen von einem Speicher aus, auf dem Schreib- und Lesebefehle atomar ausgeführt werden.
Das ist bei realen Prozessoren nicht der Fall.

Ansatz: Ein Speicherkonsistenzmodell definiert, ob/wann was (Befehle) für wen (Thread) sichtbar ist.

Der atomare Speicher erfüllt ein Modell namens Sequential-Consistency.
↳ Lamport '79.

Die Ausführungen der parallelen Threads lassen sich in eine totale Ordnung bringen → Interleaving-Semantik.

Ziel: Stelle Total-Store-Ordering vor,
das tatsächliche Speichermodell von x86-Prozessoren.
Zeige (nächstes Kapitel) die DRF-Garantie:
Falls Programm c unter SC keine Data-Races hat,
ist das Verhalten unter TSO dasselbe wie unter SC.

Das erklärt die Bedeutung von SC.
Das Modell existiert zwar nicht in der Praxis,
aber wir dürfen es annehmen.
Die DRF-Garantie ist unglaublich wichtig.
Unter TSO können wir Expertinnen und Experten programmieren.
SC verstehen alle.

12.1 Total-Store-Ordering — informell

Erstmalig formalisiert im SPARC-Manual '92.
Bouajjani, M., Möhlmann 2011.

Ansatz:
- Speichekonsistenzmodelle lassen sich operationell und axiomatisch beschreiben, ähnlich zu Beschreibung kontextfreier Sprachen über Pushdowns oder Grammatiken
- Es gibt Abwandlungen für beide Varianten. Generell sind axiomatische Modelle flexibler.
- Die Beziehungen sind nicht verstanden.

TSO als operationelles Modell:
- Jede Thread hat einen lokalen Store-Puffer, der von den anderen Threads nicht gesehen wird.
- Der Store-Puffer puffert die Schreibbefehle und führt sie zu einem späteren Zeitpunkt als Batch auf dem Hauptspeicher aus.
- Loads / Lesebefehle prüfen zunächst den lokalen Store-Puffer, ob ein Schreibbefehl für die gesuchte Adresse vorliegt.
- Falls Schreibbefehle für die gesuchte Adresse vorliegen, wird der Wert des aktuellsten Schreibbefehls gelesen. Man nennt das Store-Forwarding oder Early-Read.
- Falls keine Schreibbefehle für die gesuchte Adresse im lokalen Store-Puffer vorliegen, wird der Wert der Adresse aus dem Hauptspeicher gelesen.

Mehrkosten:
- Durch den lokalen Store-Buffer wird die Ausführungszeit von Schreibbefehlen reduziert.
- Durch das Batch-Processing wird die gesamt Zugriffszeit auf den Hauptspeicher reduziert.
- Durch Early-Reads benutzen sequentielle Programme (ein Thread) keinen Unterschied zwischen TSO und SC.

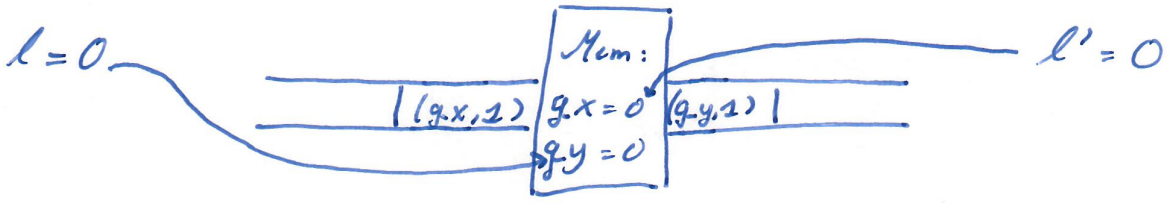
Beispiel: Dekkers Mutex

```
int g_x = 0;
int g_y = 0;
```

```
int l;
g_x = 1;
l = g_y;
if (l == 0) then
    critical section 1;
fi;
```

```
int l';
g_y = 1;
l' = g_x;
if (l' == 0) then
    critical section 2;
fi;
```

Nach den beiden Schreib- und Lesebefehlen ist unter TSO folgende Situation möglich:



Es wurden also beide Threads den kritischen Abschnitt betreten.

Mutual-Exclusion gilt nicht.

Siehe cpp-Implementierung.

12.2 Total-Store-Ordering — Formalisierung

Ansatz: Wir formalisieren das TSO-Modell indirekt,
als Einfluss auf die operationelle Semantik
paralleler Programme.
Wir geben also einen Begriff von TSO-Konfigurationen
und bestimmten Transitionen zwischen diesen Konfigurationen an.

Wir betrachten Programme

$$c_1 \parallel \ldots \parallel c_n$$

mit c_i sequentielle Programme der Sprache $\mathcal{W}$,
die über einem Shared-Heap ausgeführt werden.
Das ist das Programmiermodell für CSL mit dem Operator $\parallel$ ganz oben.
Gegeben ein solches Programm, definieren wir
die Menge der Thread-IDs $TID := \{1, \ldots, n\}$.

Definition:
Die Menge der TSO-Konfigurationen ist

$$TSOCF := Heap \times L(conf)$$

mit

$$L(conf) := TID \to (\mathcal{W} \times Stack \times Buf),$$

wobei

$$Buf := (\underbrace{\mathbb{Z} \setminus \{nil\}}_{\text{Adressen}} \times \underbrace{\mathbb{Z}}_{\text{Werte}})^* \leftarrow \text{endliche Worte.}$$

Es gibt also einen Shared-Heap und
für jeden Thread
- das noch auszuführende Programm,
- den Stack,
- den Inhalt des Store-Buffers.

Die TSO-Transitionsrelation ist beschriftet

mit Aktionen

$$Act := TID \times (\{loc, isu\} \cup (\{ld, st\} \times (\mathbb{Z} \setminus \{init\}) \times \mathbb{Z}))$$

Die Relation

$$\rightarrow_{TSO} \subseteq TSOCF \times Act \times TSOCF$$

ist definiert wie folgt:

$$\text{(LOCAL)} \quad \frac{(c, s, h) \rightarrow (c', s', h') \text{ ohne Zugriff auf Heap}}{(h, buf[i \mapsto (c, s, b)]) \xrightarrow{(i, loc)} (h, buf[i \mapsto (c', s', b)])}$$

$$\text{(LOAD)} \quad \frac{[\![a]\!] s = adr \qquad b \downarrow adr = \varepsilon \qquad h(adr) = val}{(h, buf[i \mapsto (x = [a]; c, s, b)]) \xrightarrow{(i, ld, adr, val)} (h, buf[i \mapsto (c, s[x \mapsto val], b)])}$$

$$\text{(EARLY)} \quad \frac{[\![a]\!] s = adr \qquad b \downarrow adr = (adr, val).\, b'}{(h, buf[i \mapsto (x = [a]; c, s, b)]) \xrightarrow{(i, ld, adr, val)} (h, buf[i \mapsto (c, s[x \mapsto val], b)])}$$

$$\text{(ISSUE)} \quad \frac{[\![a_1]\!] s = adr \qquad [\![a_2]\!] s = val}{(h, buf[i \mapsto ([a_1] = a_2; c, s, b)]) \xrightarrow{(i, isu)} (h, buf[i \mapsto (c, s, b.(adr, val))])}$$

$$\text{(STORE)} \quad \frac{}{(h, buf[i \mapsto (c, s, b.(adr, val))]) \xrightarrow{(i, st, adr, val)} (h[adr \mapsto val], buf[i \mapsto (c, s, b)])}$$

Die Menge der TSO-Buchungen des Programms ist

$$[\![c_1 \| \ldots \| c_n]\!]_{TSO} := \{ \tau \in Act^* \mid (h, buf) \xrightarrow{\tau}_{TSO} (h', buf'), \text{ wobei } buf(i) = (c_i, s, \varepsilon) \text{ f.a. } i \}$$

Die Menge der SC-Berechnungen des Programms ist

$$[\![c_1 \| \dots \| c_n]\!]_{SC} := [\![c_1 \| \dots \| c_n]\!]_{TSO} \setminus Wrong \quad \text{mit}$$

$$Wrong := \bigcup \Pi_{cl}^* . a . b . \Pi_{cl}^*$$

$$a = (i, isu)$$

$$b \neq (i, rd, adr, vd)$$

$$f.a.\ adr, vd$$

Beispiel:

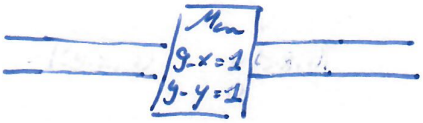
t_1

```
g_x = 1;
l = g_y;
if (l == 0) then
  cs 1;
fi
```

$\|$

t_2

```
g_y = 1;
l' = g_x;
if (l' == 0) then
  cs 2;
fi
```



Es gibt folgende TSO-Berechnung:

$$(1, isu).(1, ld, g\_y, 0).(2, su).(2, st, g\_y, 1).(2, ld, g\_x, 0).(1, st, g\_x, 1)$$

Das Delay macht den Mutex kaputt.

Bemerkung:

Bei den meisten Berechnungen gilt Mutex.

Wozu die Analyse?

Durch die starke Verbreitung von Libraries hilft herkömmliches Verhalten immer irgendwo auf.