7.7.3 Transaktionsbasierter Speicher

Die bisher vorgestellten Synchronisationsmechanismen hatten zum Ziel, kritische Bereiche durch verschiedene Mechanismen, wie Semaphore oder Monitore, zu schützen. Hierdurch können einerseits neue Probleme, wie z.B. Verklemmungen, auftreten. Andererseits kann durch zu viel Synchronisation die Effizienz des Gesamtsystems beeinträchtigt werden. Weiterhin ist ein generelles Problem von Synchronisationsmechanismen mit Sperren die *fehlende Komponierbarkeit*. Wenn wir z.B. Hash-Tabellen mit synchronisierten Einfüge- und Löschoperationen haben und ein Element aus einer Tabelle löschen und dann in eine andere Tabelle einfügen, dann ist das Element zwischen den beiden synchronisierten Operationen für andere Prozesse nicht sichtbar. Wenn wir es erst einfügen und dann löschen, ist es zwischen den beiden synchronisierten Operationen für andere Prozesse doppelt vorhanden. Somit müsste man diese kombinierte Operationen auf einer anderen Ebene synchronisieren, aber es ist nicht möglich, einfach beide synchronisierten Operationen zu einer größeren synchronisierten Operation zu kombinieren.

Eine Alternative zu diesen Synchronisationsmechanismen kann man im Bereich der Implementierung von Datenbanken finden, denn diese Implementierungen sind darauf ausgelegt, einen hohen Durchsatz auch bei vielen nebenläufigen Anfragen und Änderungen zu gewährleisten. Dies gelingt dort durch eine *optimistische Nebenläufigkeitskontrolle*: statt kritische Bereiche für alle anderen nebenläufigen Prozesse zu sperren, lässt man die Ausführung erst einmal zu und prüft dann am Ende der Ausführung, ob diese ohne Konflikte durchgelaufen ist. Statt kritischer Bereiche definiert man bei Datenbanken **Transaktionen**, d.h. eine Folge von Aktionen (Datenbankzugriffe und -änderungen), die vollständig ausgeführt werden sollen. Jede Transaktion soll dabei die **ACID**-Eigenschaften (atomicity, consistency, isolation, durability) erfüllen:

Atomarität Eine Transaktion wird entweder ganz oder gar nicht ausgeführt. Eine abgebrochene Transaktion verändert den Datenbankzustand nicht.

Konsistenz Nach Ausführung einer Transaktion ist die Datenbank in einem konsistenten oder gültigen Zustand, d.h. die Invarianten einer Datenbank sind erfüllt.

Isolation Transaktionen laufen isoliert voneinander ab, d.h. sie sehen nichts von anderen ablaufenden Transaktionen. Somit spielt es keine Rolle, ob sie nebenläufig oder sequenziell ablaufen. Dies ist eine der Hauptziele einer nebenläufigen Implementierung.

Dauerhaftigkeit Wenn eine Transaktion erfolgreich beendet ist, dann sind die Ergebnisse dauerhaft gespeichert (d.h. auch bei möglichen Systemfehlern).

Der Aspekt der Dauerhaftigkeit ist bei der nebenläufigen Programmierung weniger relevant. Daher ist das Ziel eines **transaktionsbasierten Speichers** (engl. *transactional memory*), die ACI-Eigenschaften sicherzustellen. Ein transaktionsbasierter Speicher kann sowohl in Hardware als auch in Software realisiert werden. Bei letzterem spricht man auch von **Software Transactional Memory** (**STM**), was z.B. durch Bibliotheken in verschiedenen Programmiersprachen bereitgestellt werden kann. Da es STM-Bibliotheken

für sehr viele Programmiersprachen gibt, wollen wir nachfolgend die Ideen von STM skizzieren.

Die wesentliche Grundidee von STM ist die Definition kritischer Abschnitte als Transaktion, was oft als *atomarer Block* bezeichnet wird, wie z.B.

```
atomic {
  z1 = x*2;
  z2 = y+1;
  x = z1+z2;
}
```

Hierbei sind `x` und `y` Variablen, die auch von anderen Threads gelesen und verändert werden. Diese werden auch *Transaktionsvariablen* genannt. Der optimistische Ansatz von STM ist, dass diese Transaktion ausgeführt und erst am Ende ein *commit* durchgeführt wird, wodurch die Transaktion entweder erfolgreich ist oder wiederholt werden muss.
Implementierungstechnisch kann man sich die Umsetzung einer solchen Transaktion wie folgt vorstellen:

- Die Zuweisungen erfolgen zunächst an lokale Variablen.
- Wenn globale Variablen gelesen werden, werden deren Werte gespeichert.
- Nur das *commit* am Ende einer Transaktion wird wie ein kritischer Bereich implementiert: hierbei werden die aktuellen Werte der gelesenen globalen Variablen zu diesem Zeitpunkt geprüft. Wenn diese unverändert sind, werden die lokalen Variablen in die entsprechenden globalen geschrieben und die Transaktion ist erfolgreich. Ansonsten gibt es einen Konflikt bei der Ausführung und die Transaktion wird abgebrochen und wiederholt.

Um den Vergleich der gelesenen globalen Variablenwerte zu vermeiden, wird dies häufig durch Zeitstempel implementiert. Zur Sicherstellung der ACI-Eigenschaften sind folgende Dinge wichtig:

1. Transaktionsvariablen dürfen nur in atomaren Blöcken gelesen und verändert werden.
2. Transaktionen müssen wiederholbar sein, d.h. sie dürfen nur Seiteneffekte enthalten, die wiederholt werden können.

Diese optimistische Synchronisation kann einen höheren Durchsatz gegenüber einer pessimistischen Synchronisation mit Sperren liefern, wenn es wenig commit-Konflikte gibt. Es stellt ein einfaches Programmiermodell mit klarer Semantik dar und vermeidet die üblichen Nebenläufigkeitsprobleme wie Verklemmungen.

Wie schon erwähnt, kann das STM-Programmierkonzept durch Bibliotheken realisiert werden, sodass kaum Spracherweiterungen notwendig sind (die Sprache Clojure[5] hat

[5] `https://clojure.org/`

z.B. eine eingebaute Sprachunterstützung für STM). Allerdings hängt die erfolgreiche Verwendung von der disziplinierten Benutzung der Bibliotheksfunktionen ab. Interessant ist hierbei die STM-Implementierung von Haskell[6] (Harris et al., 2005). Diese erzwingt durch das Typsystem die korrekte Benutzung der STM-Bibliothek. Z.B. haben Transaktionsvariablen einen ausgezeichneten Typ (`TVar`) und es gibt spezielle Operationen darauf:

```
newTVar   :: a → STM (TVar a)
readTVar  :: TVar a → STM a
writeTVar :: TVar a → a → STM ()
```

Diese Operationen können nur innerhalb einer speziellen `STM`-Monade verwendet werden, wobei die Operation `atomically` Aktionen der `STM`-Monade zur Ausführung bringt:

```
atomically :: STM a → IO a
```

Darüberhinaus gibt es `STM`-Operationen zur Wiederholung der aktuellen Transaktion oder Definition einer alternativen Transaktion:

```
retry :: STM a
orElse :: STM a → STM a → STM a
```

Somit verbietet das Typsystem, dass normale I/O-Aktionen (z.B. Schreiben von Dateien) innerhalb von Transaktionen ausgeführt werden, oder Transaktionsvariablen außerhalb von Transaktionen gelesen oder geschrieben werden. Solche Garantien sind bei STM-Implementierungen in anderen Sprachen schwieriger oder gar nicht zu erzielen.

Das folgende Programm zeigt die Benutzung dieser STM-Bibliothek. Zwei Bankkonten werden als Transaktionsvariablen implementiert. Die Überweisung von einem Konto auf das andere wird mittels `atomically` atomar ausgeführt, sodass kein Geld verloren werden kann.

```
import Control.Concurrent
import Control.Concurrent.STM

type Account = TVar Int

-- Create new account as a TVar.
newAccount :: Int → STM Account
newAccount am = newTVar am

-- Get the balance of the account.
getBalance :: Account → STM Int
getBalance acc = readTVar acc
```

[6] `http://hackage.haskell.org/package/stm`

```
-- Deposit some money into the account.
deposit :: Account → Int → STM ()
deposit acc am = do
  bal <- readTVar acc
  writeTVar acc (bal + am)

-- Withdraw some money from the account if the balance is positive.
-- The return flag indicates whether the withdraw was possible.
withdraw :: Account → Int → STM Bool
withdraw acc am = do
  bal <- readTVar acc
  if bal >= am then do writeTVar acc (bal - am)
                       return True
               else return False

-- Withdraw some money from the account if the balance is positive.
-- The return flag indicates whether the withdraw was possible.
transfer :: Account → Account → Int → STM Bool
transfer from to am = do
  bal <- getBalance from
  if bal >= am then do withdraw from am
                       deposit to am
                       return True
               else return False

main = do
  k1 <- atomically (newAccount 100)
  k2 <- atomically (newAccount 100)
  ...
  atomically (transfer k1 k2 n)
  ...
  bal1 <- atomically (getBalance k1)
  print bal1
  bal2 <- atomically (getBalance k2)
  print bal2
```