Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

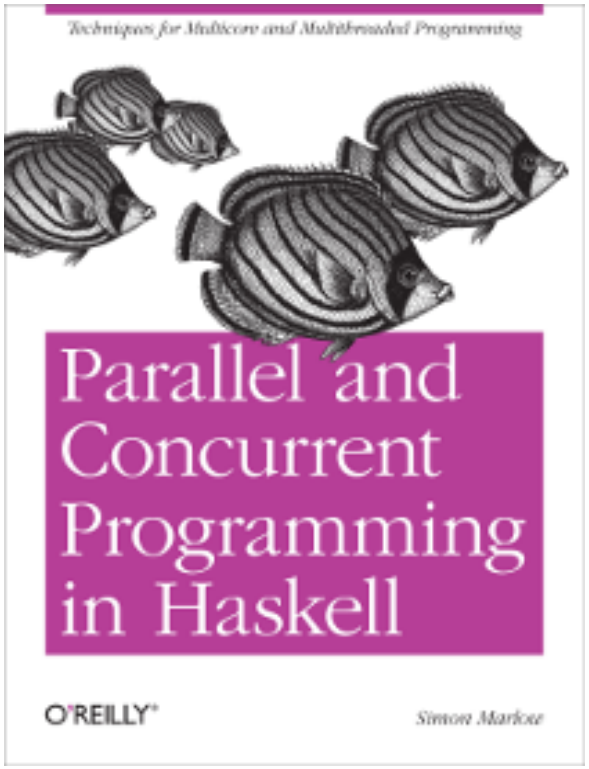
Outline I

Übersicht für Heute:

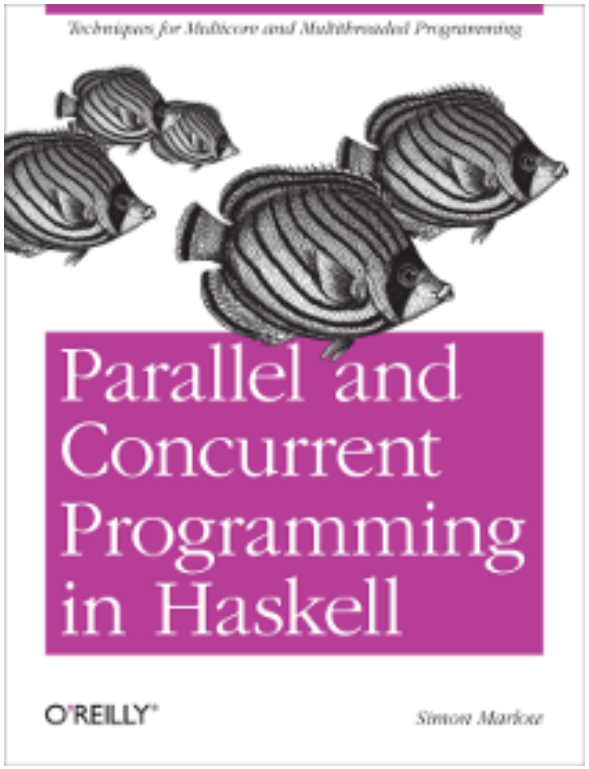
1. Wiederholung
2. Threads, MVars, etc.
   - forkIO und MVars
   - Deadlock Detection
3. Software Transactional Memory
   - Motivation
   - Beispiel: Banksoftware
4. Ein simpler Chat-Server

Wiederholung

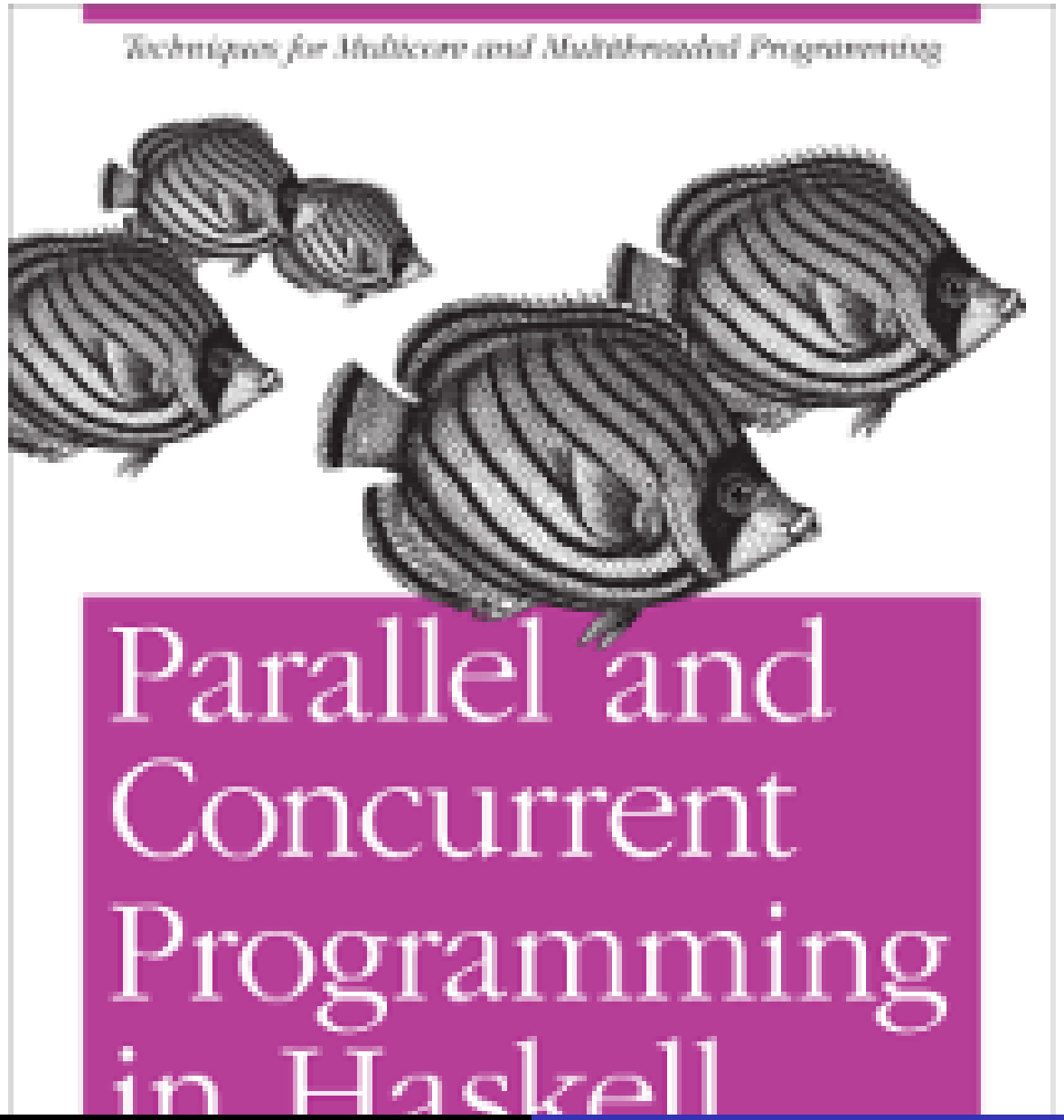
Leseempfehlung:



Leseempfehlung:



. . . srsly!


Techniques for Multicore and Multithreaded Programming
Parallel and
Concurrent
Programming
in Haskell

Überblick:

Was war nochmal der Unterschied zwischen Paralllelism und Nebenläufigkeit?

Überblick:

Was war nochmal der Unterschied zwischen Paralllelism und Nebenläufigkeit?

Parallelism:

- Mehrere Hardwareelemente

Überblick:

Was war nochmal der Unterschied zwischen Paralllelism und Nebenläufigkeit?

Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen

Überblick:
Was war nochmal der Unterschied zwischen Paralllelism und Nebenläufigkeit?

Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)

Überblick:
Was war nochmal der Unterschied zwischen Paralllelism und Nebenläufigkeit?

Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

Überblick:
Was war nochmal der Unterschied zwischen Paralllelism und Nebenläufigkeit?

Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

Concurrency:

- Mehrere Threads

Überblick:
Was war nochmal der Unterschied zwischen Paralllelism und Nebenläufigkeit?

Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

Concurrency:

- Mehrere Threads
- Dinge gleichzeitig tun

Überblick:
Was war nochmal der Unterschied zwischen Paralllelism und Nebenläufigkeit?

Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

Concurrency:

- Mehrere Threads
- Dinge gleichzeitig tun
- nichtdeterministisch

Überblick:
Was war nochmal der Unterschied zwischen Paralllelism und Nebenläufigkeit?

Parallelism:

- Mehrere Hardwareelemente
- Antwort schneller kriegen
- deterministisch (i.d.R.)
- oft deklarativ

Concurrency:

- Mehrere Threads
- Dinge gleichzeitig tun
- nichtdeterministisch
- oft impertativ

Die Basics: Threads, MVars, etc.

Wir beginnen mit der Funktion, die einen neuen Thread erstellt:

```
forkIO :: IO () -> IO ThreadId
```

Wir beginnen mit der Funktion, die einen neuen Thread erstellt:

```
forkIO :: IO () -> IO ThreadId
```

Threads interagieren notwendigerweise mit der Welt, ergo ist die Berechnung, die wir übergeben vom Typ `IO ()`.

Wir beginnen mit der Funktion, die einen neuen Thread erstellt:

```
forkIO :: IO () -> IO ThreadId
```

Threads interagieren notwendigerweise mit der Welt, ergo ist die Berechnung, die wir übergeben vom Typ `IO ()`.

Die `ThreadId` kann später benutzt werden um z.B. den Thread vorzeitig zu töten oder ihm eine Exception zuzuschmeißen.

Ein kleines Beispiel:

```
import Control.Concurrent
import Control.Monad
import System.IO

main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  forkIO (replicateM_ 100000 (putChar 'A'))
  replicateM_ 100000 (putChar 'B')
```

Ein kleines Beispiel:

```
import Control.Concurrent
import Control.Monad
import System.IO

main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  forkIO (replicateM_ 100000 (putChar 'A'))
  replicateM_ 100000 (putChar 'B')
```

...Output?

Ein kleines Beispiel:

```
import Control.Concurrent
import Control.Monad
import System.IO

main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  forkIO (replicateM_ 100000 (putChar 'A'))
  replicateM_ 100000 (putChar 'B')
```

...Output?

```
AAAAAAAAABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABAB
```

Aber...

Aber. . . wie kriegen wir jetzt Ergebnisse aus der Berechnung raus?
Der Typ ist nur `IO ()`, das liefert nichts (interessantes) zurück!

Aber. . . wie kriegen wir jetzt Ergebnisse aus der Berechnung raus? Der Typ ist nur `IO ()`, das liefert nichts (interessantes) zurück!

Das gleiche Problem hatten wir schon in der `Par`-Monade. Lösung damals waren `IVars`:

```
data IVar a  -- instance Eq

new :: Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a
```

Introducing: . . .

Introducing: ...MVars!

```
data MVar a  -- abstract

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()

readMVar     :: MVar a -> IO a
```

Introducing: ...MVars!

```haskell
data MVar a  -- abstract

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()

readMVar     :: MVar a -> IO a
```

Wir brauchen hier keine eigene Monade wie `Par`. Da Concurrency so oder so effektvoll ist, reicht `IO` vollkommen aus.

Unterschied zwischen `IVars` und `MVars`: erstere sind ***i***mmutable, letztere sind ***m***utable.

Ein Beispiel zu MVars:

```
main :: IO ()
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r <- takeMVar m
  print r
  r <- takeMVar m
  print r
```

Ein Beispiel zu `MVars`:

```
main :: IO ()
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r <- takeMVar m
  print r
  r <- takeMVar m
  print r
```

Wie wir sehen kann die gleiche `MVar` über Zeit mehrere Zustände annehmen und erfolgreich zur Kommunikation zwischen Threads benutzt werden.

Generell haben `MVars` drei Hauptaufgaben:

Generell haben `MVars` drei Hauptaufgaben:

- **Channel mit nur einem Slot**
  Eine `MVar` kann als Nachrichtenkanal zwischen Threads benutzt werden, allerdings maximal eine Nachricht auf einmal halten.

Generell haben `MVars` drei Hauptaufgaben:

- **Channel mit nur einem Slot**
  Eine `MVar` kann als Nachrichtenkanal zwischen Threads benutzt werden, allerdings maximal eine Nachricht auf einmal halten.
- **Behältnis für shared mutable state**
  In Concurrent Haskell brauchen oft mehrere Threads Zugriff auf einen shared state. Ein beliebtes Designpattern ist, das dieser State als normaler (immutable) Haskell-Datentyp repräsentiert und in einer `MVar` verpackt wird.

Generell haben `MVars` drei Hauptaufgaben:

- **Channel mit nur einem Slot**
  Eine `MVar` kann als Nachrichtenkanal zwischen Threads benutzt werden, allerdings maximal eine Nachricht auf einmal halten.
- **Behältnis für shared mutable state**
  In Concurrent Haskell brauchen oft mehrere Threads Zugriff auf einen shared state. Ein beliebtes Designpattern ist, das dieser State als normaler (immutable) Haskell-Datentyp repräsentiert und in einer `MVar` verpackt wird.
- **Baustein für kompliziertere Strukturen**

Mehr Leckerlis:

Was passiert, wenn wir folgenden Code ausführen?

```
main :: IO ()
main = do m <- newEmptyMVar
          takeMVar m
```

Mehr Leckerlis:

Was passiert, wenn wir folgenden Code ausführen?

```
main :: IO ()
main = do m <- newEmptyMVar
          takeMVar m
```

Wir bekommen eine Fehlermeldung, dass das Programm hängt, statt einfach nur ein hängendes Programm.

```
$ ./mvar3
mvar3: thread blocked indefinitely in an MVar operation
```

Deadlock detection:

Threads und `MVars` sind Objekte auf dem Heap. Das `RTS` (i.e. der Garbage collector) durchläuft den Heap um alle lebendigen Objekte zu finden, angefangen bei den laufenden Threads und ihren Stacks.

Alles was so nicht erreichbar sind (z.B. ein Thread der auf eine `MVar` wartet, die nirgendwo sonst referenziert wird), blockiert und bekommt eine Exception geschmissen.

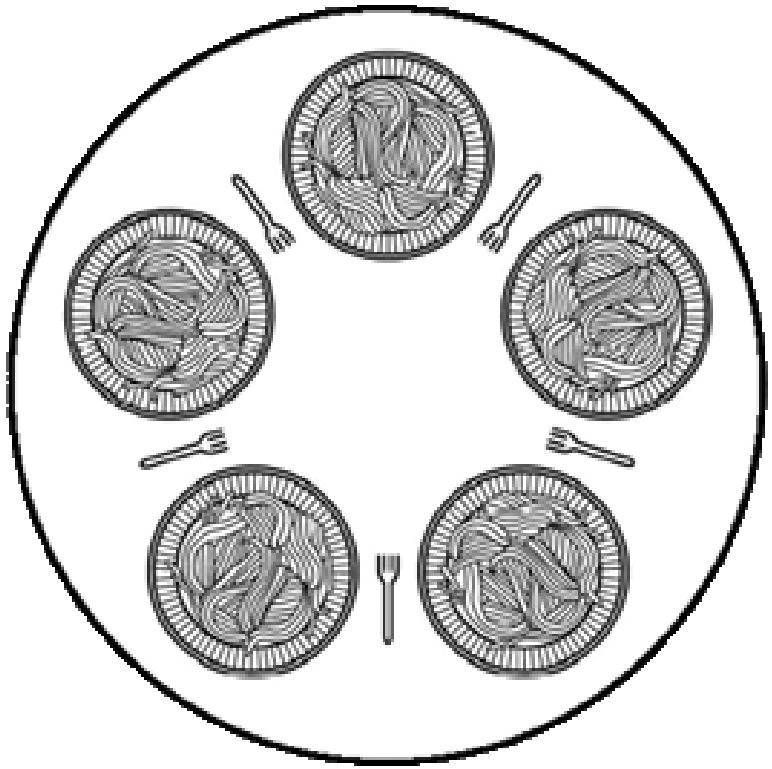


Abbildung: dining philosophers

Deadlock detection:

Dieses Vorgang funktioniert allerdings nicht immer wie man zunächst denkt. Beispiel: Was passiert mit diesem Code?

```
main :: IO ()
main = do
  lock <- newEmptyMVar
  complete <- newEmptyMVar
  forkIO $ takeMVar lock `finally` putMVar complete ()
  takeMVar complete
```

Deadlock detection:

Dieses Vorgang funktioniert allerdings nicht immer wie man zunächst denkt. Beispiel: Was passiert mit diesem Code?

```
main :: IO ()
main = do
  lock <- newEmptyMVar
  complete <- newEmptyMVar
  forkIO $ takeMVar lock `finally` putMVar complete ()
  takeMVar complete
```

Da nicht nur der geforkte Thread sondern auch der ursprüngliche gedeadlocked sind, wird hier die Fehlermeldung geprintet, statt die rettende Exception an das Kind zu senden.

Software Transactional Memory (STM)

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

Beliebte Fehler:

- **Races** (vergessene Locks)

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

Beliebte Fehler:

- **Races** (vergessene Locks)
- **Deadlocks** (Locks in falscher Reihenfolge genommen)

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

Beliebte Fehler:

- **Races** (vergessene Locks)
- **Deadlocks** (Locks in falscher Reihenfolge genommen)
- **Lost wakeups** (Conditional-Variable nicht bescheid gesagt)

Motivation:

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)

„The Future is Parallel, and the Future of Parallel is Declarative “
https://www.youtube.com/watch?v=hlyQjK1qjw8

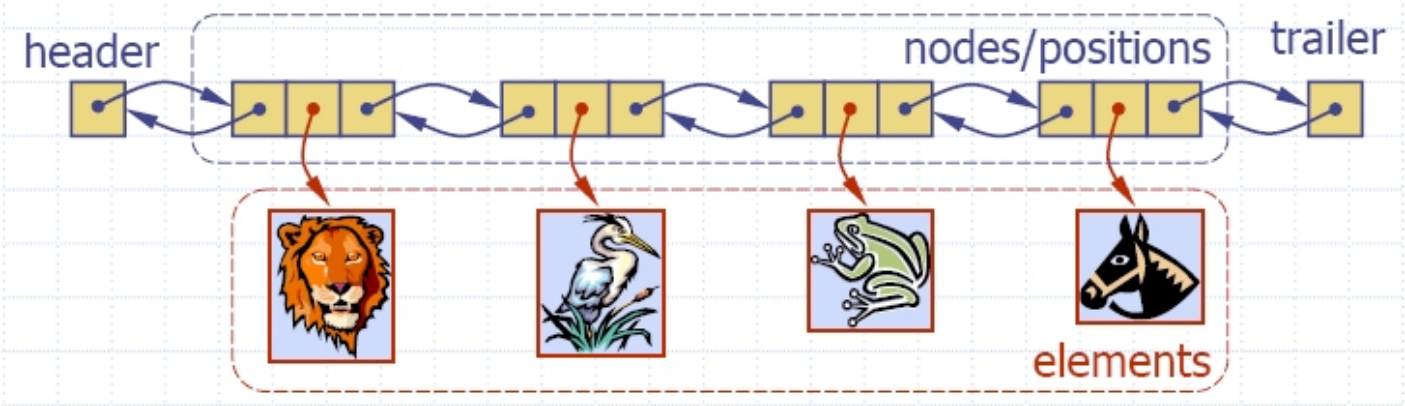
Beliebte Fehler:

- **Races** (vergessene Locks)
- **Deadlocks** (Locks in falscher Reihenfolge genommen)
- **Lost wakeups** (Conditional-Variable nicht bescheid gesagt)
- **Error Recovery** (Exceptionhandler müssen Locks freigeben und teilweise Ursprungszustand restaurieren)

Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

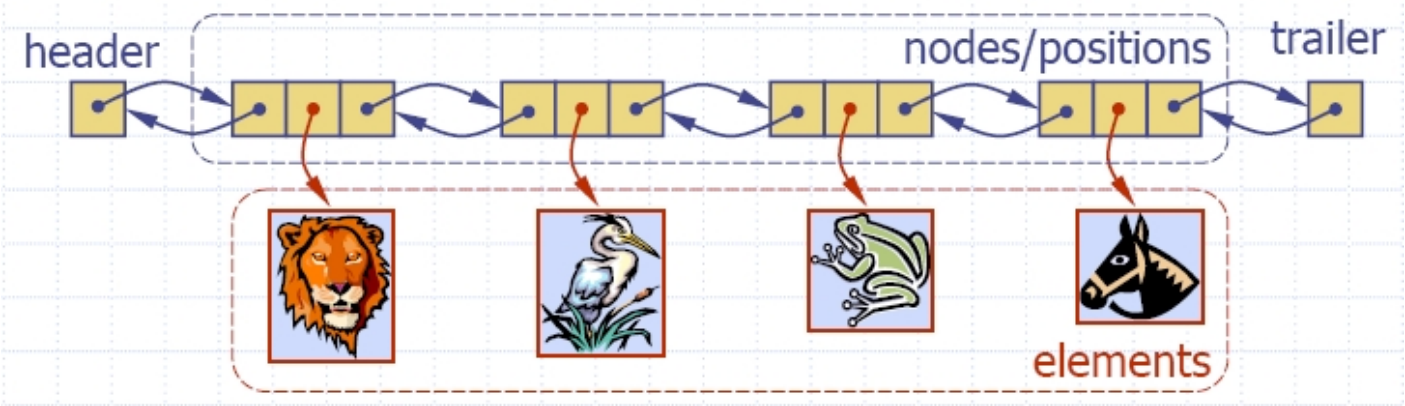
Bildquelle: http://www.math.bas.bg/∼nkirov/2015/NETB201/slides/ch04/ch04.html



Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

Bildquelle: http://www.math.bas.bg/~nkirov/2015/NETB201/slides/ch04/ch04.html

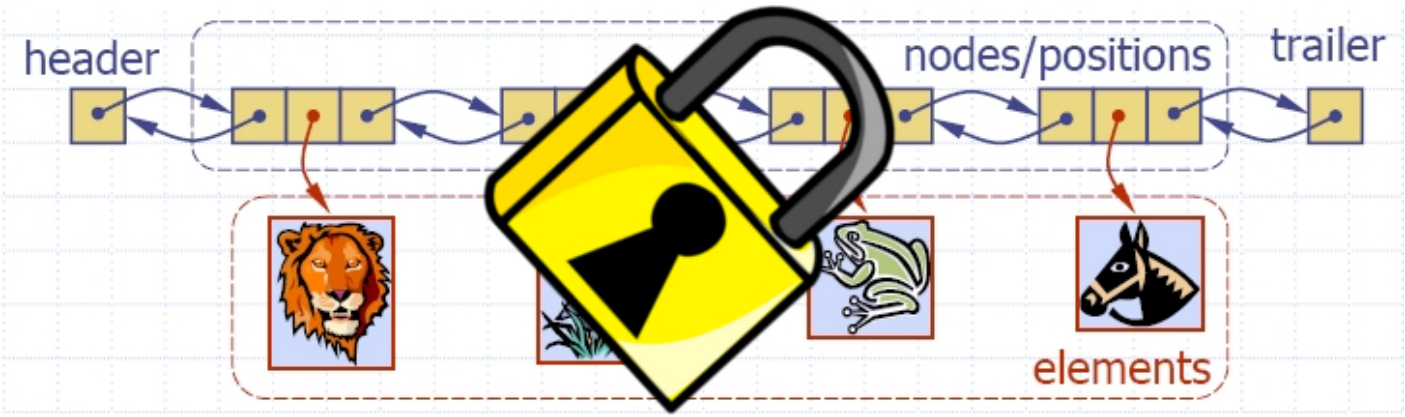


Problem: offensichtlich, race conditions etc.

Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

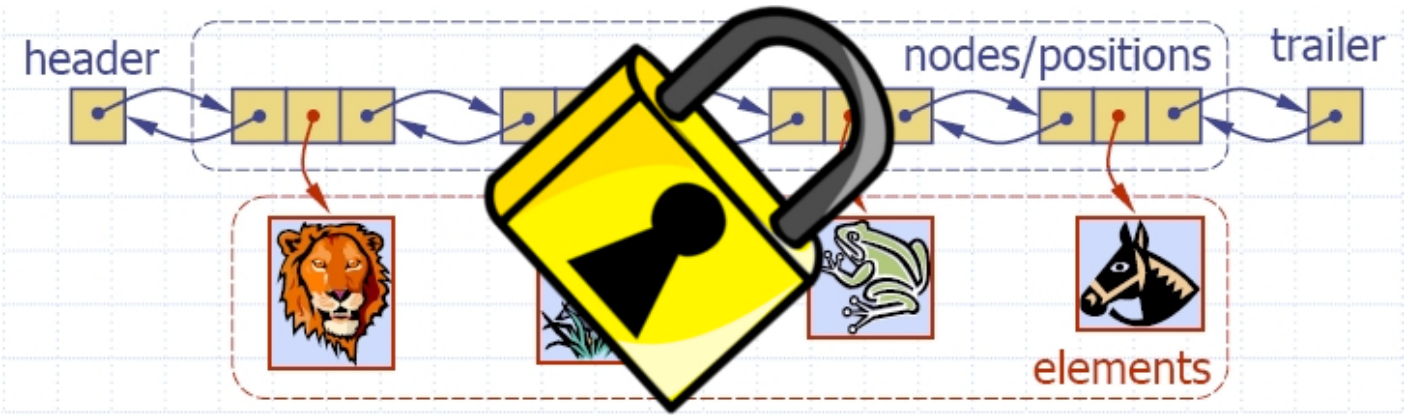
Bildquelle: http://www.math.bas.bg/∼nkirov/2015/NETB201/slides/ch04/ch04.html



Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

Bildquelle: http://www.math.bas.bg/∼nkirov/2015/NETB201/slides/ch04/ch04.html

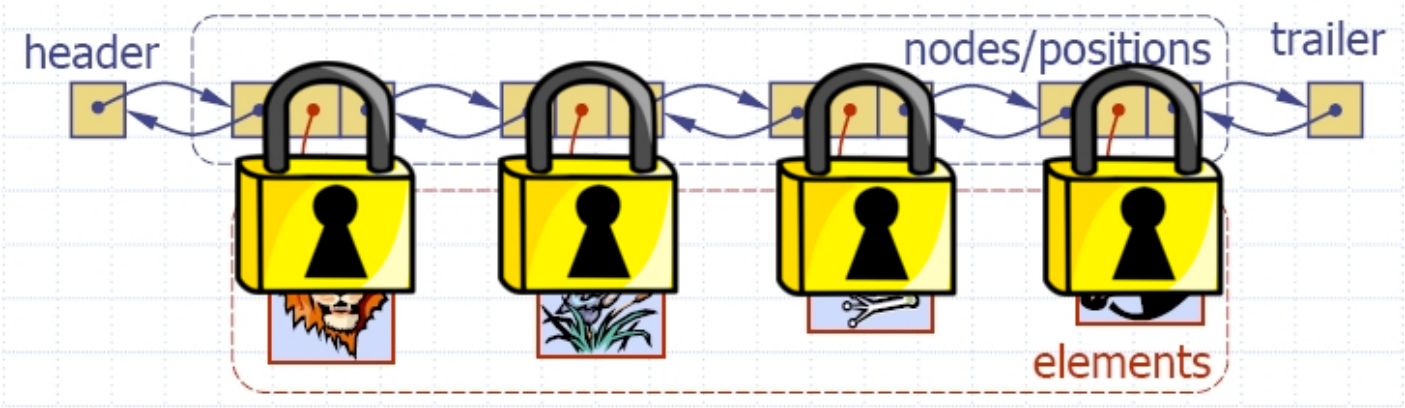


Problem: Nicht gerade nebenläufig. . .

Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

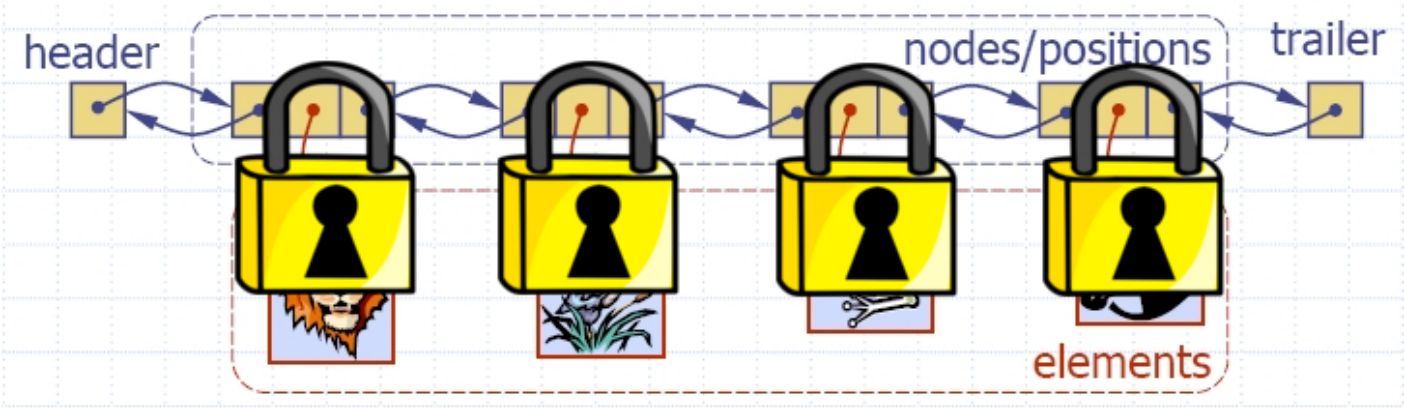
Bildquelle: http://www.math.bas.bg/∼nkirov/2015/NETB201/slides/ch04/ch04.html



Beispiel:

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

Bildquelle: http://www.math.bas.bg/~nkirov/2015/NETB201/slides/ch04/ch04.html



Problem: Fehleranfälligkeit bei kleinen Listen

Aufgabe vs. Schwierigkeit: Zweiendige Liste

Problemstellung	Schwierigkeit
sequentiell	Gymnasium oder Bachelor

Aufgabe vs. Schwierigkeit: Zweiendige Liste

Problemstellung	Schwierigkeit
sequentiell	Gymnasium oder Bachelor
Locks et al.	(gerade nicht mehr) publizierbar auf internationalen Konferenzen

Aufgabe vs. Schwierigkeit: Zweiendige Liste

Problemstellung	Schwierigkeit
sequentiell	Gymnasium oder Bachelor
Locks et al.	(gerade nicht mehr) publizierbar auf internationalen Konferenzen
atomic blocks (`STM`)	Bachelor

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren (d.h. nicht unterbrochenen) Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade auch (`do`-Notation etc.).

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren (d.h. nicht unterbrochenen) Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade auch (`do`-Notation etc.).

- Das bedeutet, dass Deadlocks unmöglich werden *weil es keine Locks mehr gibt*!

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren (d.h. nicht unterbrochenen) Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade auch (`do`-Notation etc.).

- Das bedeutet, dass Deadlocks unmöglich werden *weil es keine Locks mehr gibt*!
- Automatisierte error recovery. `STM` stellt den Ausgangszustand von selbst wieder her.

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren (d.h. nicht unterbrochenen) Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade auch (`do`-Notation etc.).

- Das bedeutet, dass Deadlocks unmöglich werden *weil es keine Locks mehr gibt*!
- Automatisierte error recovery. `STM` stellt den Ausgangszustand von selbst wieder her.
- `TVars` (Transaction Variables). Wie `IVars` und `MVars`, nur in der `STM`-Monade.

STM auf einen Blick:

```
data STM a                                  -- abstract
instance Monad STM                          -- among other things

atomically :: STM a -> IO a

data TVar a                                 -- abstract
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
```

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus
- . . . und zwar in einem Rutsch, ohne Unterbrechung!

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus
- . . . und zwar in einem Rutsch, ohne Unterbrechung!
- Stellt bei Fehlschlag Ursprungszustand wieder her.

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus
- . . . und zwar in einem Rutsch, ohne Unterbrechung!
- Stellt bei Fehlschlag Ursprungszustand wieder her.
- Deshalb: Kein `IO` in Transaktionen!

Ein kurzer Blick auf `atomically`:

```
atomically :: STM a -> IO a
```

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus
- . . . und zwar in einem Rutsch, ohne Unterbrechung!
- Stellt bei Fehlschlag Ursprungszustand wieder her.
- Deshalb: Kein `IO` in Transaktionen!
- Ebenfalls: Keine genesteten `atomicallys` (Typen!)

Ein kurzer Blick auf `retry`:

```
retry :: STM a
```

Ein kurzer Blick auf `retry`:

```
retry :: STM a
```

- *Kein* Sprachkonstrukt!

Ein kurzer Blick auf `retry`:

```
retry :: STM a
```

- *Kein* Sprachkonstrukt!
- Rollt zurück und versucht die gleiche Transaktion erneut durchzuführen

Ein kurzer Blick auf `retry`:

```
retry :: STM a
```

- *Kein* Sprachkonstrukt!
- Rollt zurück und versucht die gleiche Transaktion erneut durchzuführen
- . . . allerdings erst zu einem angebrachten Zeitpunkt.
  CPU wird nicht unnützerweise zur Heizung.

Ein kurzer Blick auf `orElse`:

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

Ein kurzer Blick auf `orElse`:

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

- *Kein* Sprachkonstrukt!

Ein kurzer Blick auf `orElse`:

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

- *Kein* Sprachkonstrukt!
- `orElse a b` führt `b` aus, wenn `a` `retry` aufruft.

Ein kurzer Blick auf `orElse`:

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

- *Kein* Sprachkonstrukt!
- `orElse a b` führt `b` aus, wenn `a` `retry` aufruft.
- Komposition von `STM`-Berechnungen:
  `>>=` ist `AND`; `orElse` ist `OR`

Ein kurzer Blick auf `orElse`:

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

- *Kein* Sprachkonstrukt!
- `orElse a b` führt `b` aus, wenn `a` `retry` aufruft.
- Komposition von `STM`-Berechnungen:
  `>>=` ist `AND`; `orElse` ist `OR`

Beispiel:

```
takeEitherTMVar :: TMVar a -> TMVar b -> STM (Either a b)
takeEitherTMVar ma mb =
  fmap Left (takeTMVar ma)
    `orElse`
  fmap Right (takeTMVar mb)
```

Stellen wir uns vor, wir wollen eine simplifizierte Bankensoftware schreiben, die in der Lage sein soll, Konten und Überweisungen zu repräsentieren.

Stellen wir uns vor, wir wollen eine simplifizierte Bankensoftware schreiben, die in der Lage sein soll, Konten und Überweisungen zu repräsentieren.

Eine naive Implementation wäre die folgende:

```
type Account = IORef Integer

transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    fromVal <- readIORef from
    toVal   <- readIORef to
    writeIORef from (fromVal - amount)
    writeIORef to (toVal + amount)
```

Diese Implementation hätte jedoch in einem Nebenläufigen Setting einige Probleme. Man beachte folgende Zeile:

```
fromVal <- readIORef from
```

Diese Implementation hätte jedoch in einem Nebenläufigen Setting einige Probleme. Man beachte folgende Zeile:

```
fromVal <- readIORef from
```

Finden nun mehrere Aktionen gleichzeitig statt, so könnte es sein, dass mehrere Threads denselben Wert als `fromVal` lesen, bevor die jeweils andere Transaktion durchgeführt wurde.
Dies hätte zur Folge, dass später ein inkorrekter Kontostand berechnet und gesetzt würde.

Führen wir also Locks ein (durch Benutzung von `MVars`), um diese race condition zu verhindern.

Führen wir also Locks ein (durch Benutzung von `MVars`), um diese race condition zu verhindern.

```haskell
type Account = MVar Integer

credit :: Integer -> Account -> IO ()
credit amount account = do
    current <- takeMVar account
    putMVar account (current + amount)

debit :: Integer -> Account -> IO ()
debit amount account = do
    current <- takeMVar account
    putMVar account (current - amount)
```

In dieser Implementation sähe eine Funktion für Überweisungen etwa so aus:

```
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

In dieser Implementation sähe eine Funktion für Überweisungen etwa so aus:

```
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Diese verhindert, dass durch fehlerhaftes Zusammenspiel von Threads Geld geschaffen oder vernichtet wird.

In dieser Implementation sähe eine Funktion für Überweisungen etwa so aus:

```
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Diese verhindert, dass durch fehlerhaftes Zusammenspiel von Threads Geld geschaffen oder vernichtet wird.

Es existiert allerdings immer noch eine race condition: Der Thread, der die Überweisung ausführt, könnte direkt nach dem `debit`-Schritt von der CPU verdrängt werden und die Bankensoftware dadurch in einem inkonsistenten Zustand zurück lassen.

Wie sähe diese Software mit STM aus?

```
type Account = TVar Integer

credit :: Integer -> Account -> STM ()
credit amount account = do
    current <- readTVar account
    writeTVar account (current + amount)

debit :: Integer -> Account -> STM ()
debit amount account = do
    current <- readTVar account
    writeTVar account (current - amount)

transfer :: Integer -> Account -> Account -> STM ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Vergleich zur Variante mit MVars:

```
type Account = MVar Integer

credit :: Integer -> Account -> IO ()
credit amount account = do
    current <- takeMVar account
    putMVar account (current + amount)

debit :: Integer -> Account -> IO ()
debit amount account = do
    current <- takeMVar account
    putMVar account (current - amount)

transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

```
transfer :: Integer -> Account -> Account -> IO ()
```

Diese Funktion führt die Überweisung vollkommen in der echten Welt durch, mit allen möglichen Fehlern, die dabei auftreten können.

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

```
transfer :: Integer -> Account -> Account -> IO ()
```

Diese Funktion führt die Überweisung vollkommen in der echten Welt durch, mit allen möglichen Fehlern, die dabei auftreten können.

```
transfer :: Integer -> Account -> Account -> STM ()
```

Diese Funktion stellt uns nur eine Berechnung in der `STM`-Monade bereit, die wir später entweder direkt oder auch als Baustein einer größeren Transaktion ausführen können.

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

```
transfer :: Integer -> Account -> Account -> IO ()
```

Diese Funktion führt die Überweisung vollkommen in der echten Welt durch, mit allen möglichen Fehlern, die dabei auftreten können.

```
transfer :: Integer -> Account -> Account -> STM ()
```

Diese Funktion stellt uns nur eine Berechnung in der `STM`-Monade bereit, die wir später entweder direkt oder auch als Baustein einer größeren Transaktion ausführen können.

Der Vorteil ist, dass wir nur einen Weg haben, die Berechnung auszuführen:

```
atomically :: STM a -> IO a
```

Wenn wir doch jetzt dieses tolle `STM` haben, brauchen wir dann überhaupt noch `MVars`?

Wenn wir doch jetzt dieses tolle `STM` haben, brauchen wir dann überhaupt noch `MVars`?

Ja! Hier sind einige Gründe:

Wenn wir doch jetzt dieses tolle `STM` haben, brauchen wir dann überhaupt noch `MVars`?

Ja! Hier sind einige Gründe:

- **Performance:** `STM` ist eine Abstraktion, und wie (fast) alle Abstraktionen hat es Laufzeitkosten.

Wenn wir doch jetzt dieses tolle `STM` haben, brauchen wir dann überhaupt noch `MVars`?

Ja! Hier sind einige Gründe:

- **Performance:** `STM` ist eine Abstraktion, und wie (fast) alle Abstraktionen hat es Laufzeitkosten.
- **Fairness:** Blockieren mehrere Threads auf einer `MVar`, werden sie garantiert FIFO wieder aufgeweckt. `STM` hingegen hat keine Garantie für Fairness.

Hands on:
Ein einfacher Chat-Server in Haskell

Wir wollen uns einen simplen Chat-Server basteln, der Verbindungen mit mehreren Clients (verbunden über `telnet`) gleichzeitig offen halten und bearbeiten kann.

Wir wollen uns einen simplen Chat-Server basteln, der Verbindungen mit mehreren Clients (verbunden über `telnet`) gleichzeitig offen halten und bearbeiten kann.

Insbesondere sollen einem Client folgende Befehle offen stehen:

1. */tell name* Schickt eine private Nachricht an den User *name*

Wir wollen uns einen simplen Chat-Server basteln, der Verbindungen mit mehreren Clients (verbunden über `telnet`) gleichzeitig offen halten und bearbeiten kann.

Insbesondere sollen einem Client folgende Befehle offen stehen:

1. */tell name* Schickt eine private Nachricht an den User *name*
2. */kick name* Disconnectet den User *name*

Wir wollen uns einen simplen Chat-Server basteln, der Verbindungen mit mehreren Clients (verbunden über `telnet`) gleichzeitig offen halten und bearbeiten kann.

Insbesondere sollen einem Client folgende Befehle offen stehen:

1. */tell name* Schickt eine private Nachricht an den User *name*
2. */kick name* Disconnectet den User *name*
3. */quit* Disconnected den aktuellen Client selbst

Wir wollen uns einen simplen Chat-Server basteln, der Verbindungen mit mehreren Clients (verbunden über `telnet`) gleichzeitig offen halten und bearbeiten kann.

Insbesondere sollen einem Client folgende Befehle offen stehen:

1. */tell name* Schickt eine private Nachricht an den User *name*
2. */kick name* Disconnectet den User *name*
3. */quit* Disconnected den aktuellen Client selbst
4. *message* Alle anderen Strings werden an alle verbundenen Clients gebroadcastet.

Ein sinnvoller erster Schritt ist oft, zu überlegen, wie man seine grundlegenden Datentypen repräsentieren möchte.

Bei uns sind das inbesondere Clients und Nachrichten:

Ein sinnvoller erster Schritt ist oft, zu überlegen, wie man seine grundlegenden Datentypen repräsentieren möchte.

Bei uns sind das inbesondere Clients und Nachrichten:

```
type ClientName = String -- zum besseren Verständnis

data Client = Client
  { clientName     :: ClientName
  , clientHandle   :: Handle
  , clientKicked   :: TVar (Maybe String)
  , clientSendChan :: TChan Message
  }
```

Ein sinnvoller erster Schritt ist oft, zu überlegen, wie man seine grundlegenden Datentypen repräsentieren möchte.

Bei uns sind das inbesondere Clients und Nachrichten:

```
type ClientName = String -- zum besseren Verständnis

data Client = Client
  { clientName     :: ClientName
  , clientHandle   :: Handle
  , clientKicked   :: TVar (Maybe String)
  , clientSendChan :: TChan Message
  }
```

`TChans` sind FIFO Channel zur Kommunikation zwischen Threads, ebenfalls ein `STM` primitive.

Nachrichten sind ebenfalls schnell implementiert:

Nachrichten sind ebenfalls schnell implementiert:

```
data Message = Notice String
             | Tell ClientName String
             | Broadcast ClientName String
             | Command String
```

Wobei `Notice` eine Nachricht vom Server, `Tell` eine private Nachricht, `Broadcast` eine öffentliche Nachricht und `Command` ein Befehl vom Nutzer ist.

Neue Clients zu erstellen ist für's Erste ziemlich einfach (vorausgesetzt `Handle` und `ClientName` werden übergeben):

```
newClient :: ClientName -> Handle -> STM Client
newClient name handle = do
  c <- newTChan
  k <- newTVar Nothing
  return Client { clientName     = name
                , clientHandle   = handle
                , clientSendChan = c
                , clientKicked   = k
                }
```

Neue Clients zu erstellen ist für's Erste ziemlich einfach (vorausgesetzt `Handle` und `ClientName` werden übergeben):

```
newClient :: ClientName -> Handle -> STM Client
newClient name handle = do
  c <- newTChan
  k <- newTVar Nothing
  return Client { clientName     = name
                , clientHandle   = handle
                , clientSendChan = c
                , clientKicked   = k
                }
```

Man beachte, dass der Rückgabewert ein `Client` in `STM` ist!

Neue Clients zu erstellen ist für's Erste ziemlich einfach (vorausgesetzt `Handle` und `ClientName` werden übergeben):

```
newClient :: ClientName -> Handle -> STM Client
newClient name handle = do
  c <- newTChan
  k <- newTVar Nothing
  return Client { clientName     = name
                , clientHandle   = handle
                , clientSendChan = c
                , clientKicked   = k
                }
```

Man beachte, dass der Rückgabewert ein `Client` in `STM` ist!

```
sendMessage :: Client -> Message -> STM ()
sendMessage Client{..} msg =
  writeTChan clientSendChan msg
```

Die `{..}`-Syntax nennt sich *record wildcard*-Syntax (benötigt die Extension `RecordWildCards`). Diese holt alle Felder des Records mit ihren respektiven Namen in scope.

Ein Server ist in unserem Falle eine Key-Value-Map von `ClientNames` zu `Clients`. Das bedeutet jedem Namen wird genau ein Client zugeordnet.
(importiert aus `Data.Map`, nicht zu verwechseln mit `map` für Listen)

Diese Map wird in einer `TVar` vorgehalten.

```
data Server = Server
  { clients :: TVar (Map ClientName Client)
  }
```

Eine Funktion für neue Server ist schnell erstellt. Hier ist es nicht notwendig, in `STM` zu bleiben.

```
newServer :: IO Server
newServer = do
  c <- newTVarIO Map.empty
  return Server { clients = c }
```

Wollen wir nun also eine Nachricht über einen ganzen `Server` verschicken, (Broadcast), nehmen wir uns einfach alle Elemente der Map einzeln.

```
broadcast :: Server -> Message -> STM ()
broadcast Server{..} msg = do
  clientmap <- readTVar clients
  mapM_ (\client -> sendMessage client msg)
                            (Map.elems clientmap)
```

Wollen wir nun also eine Nachricht über einen ganzen `Server` verschicken, (Broadcast), nehmen wir uns einfach alle Elemente der Map einzeln.

```
broadcast :: Server -> Message -> STM ()
broadcast Server{..} msg = do
  clientmap <- readTVar clients
  mapM_ (\client -> sendMessage client msg)
                            (Map.elems clientmap)
```

`mapM_` ist das `map` auf Listen, nur dass das (monadische) Ergebnis jeweils weggeschmissen wird.

```
mapM_ :: (Monad m, Foldable t) => (a -> m b) -> t a -> m ()
```

Wir brauchen einen Port, auf dem der Server lauschen soll. Eigentlich egal welcher, solange wir nicht einen beliebten Standardport nehmen.

```
port :: Int
port = 44444
```

```haskell
main :: IO ()
main = withSocketsDo $ do
  server <- newServer
  sock <- listenOn (PortNumber (fromIntegral port))
  printf "Listening on port %d\n" port
  forever $ do
    (handle, host, port) <- accept sock
    printf "Accepted connection from %s: %s\n" host (show port)
    forkFinally (talk handle server) (\_ -> hClose handle)
```

```haskell
main :: IO ()
main = withSocketsDo $ do
  server <- newServer
  sock <- listenOn (PortNumber (fromIntegral port))
  printf "Listening on port %d\n" port
  forever $ do
    (handle, host, port) <- accept sock
    printf "Accepted connection from %s: %s\n" host (show port)
    forkFinally (talk handle server) (\_ -> hClose handle)
```

accept blockiert bis eine neue Verbindung hergestellt wurde.

```haskell
main :: IO ()
main = withSocketsDo $ do
  server <- newServer
  sock <- listenOn (PortNumber (fromIntegral port))
  printf "Listening on port %d\n" port
  forever $ do
    (handle, host, port) <- accept sock
    printf "Accepted connection from %s: %s\n" host (show port)
    forkFinally (talk handle server) (\_ -> hClose handle)
```

`accept` blockiert bis eine neue Verbindung hergestellt wurde.

Mit `forkFinally` erstellen wir einen neuen Thread um die Interaktion mit diesem Client (über die noch zu schreibende Funktion `talk`) zu regeln und sobald das beendet ist den Handle sinnvoll zu beenden.

Clients werden atomar hionzugefügt, damit sich nicht zwei Clients gleichzeitig mit demselben Namen anmelden können.

```
checkAddClient :: Server -> ClientName -> Handle
                                       -> IO (Maybe Client)
checkAddClient server@Server{..} name handle = atomically $ do
  clientmap <- readTVar clients
  if Map.member name clientmap
    then return Nothing
    else do
       client <- newClient name handle
       writeTVar clients $ Map.insert name client clientmap
       broadcast server  $ Notice (name ++ " has connected")
       return (Just client)
```

Clients entfernen funktioniert ähnlich, ebenfalls atomar:

```
removeClient :: Server -> ClientName -> IO ()
removeClient server@Server{..} name = atomically $ do
  modifyTVar' clients $ Map.delete name
  broadcast server $ Notice (name ++ " has disconnected")
```

```
talk :: Handle -> Server -> IO ()
talk handle server@Server{..} = do
  hSetNewlineMode handle universalNewlineMode
      -- Swallow carriage returns sent by telnet clients
  hSetBuffering handle LineBuffering
  readName
 where
  readName = do
    hPutStrLn handle "What is your name?"
    name <- hGetLine handle
    if null name
      then readName
      else mask $ \restore -> do
             ok <- checkAddClient server name handle
             case ok of
               Nothing -> restore $ do
                  hPrintf handle
                      "The name %s is already in use.\n" name
                  readName
               Just client ->
                  restore (runClient server client)
                      `finally` removeClient server name
```

```
mask :: ((forall a. IO a -> IO a) -> IO b) -> IO b
```

```
mask :: ((forall a. IO a -> IO a) -> IO b) -> IO b
```

Keine Panik!

```
mask :: ((forall a. IO a -> IO a) -> IO b) -> IO b
```

Keine Panik!

`mask` ist eine Funktion, die eine Funktion als Argument nimmt, die wiederum eine Funktion nimmt...
Die Feinheiten sind nicht enorm wichtig (und können bei Neugier oder Bedarf gerne nachgelesen werden).
Die Idee ist, dass für kritische Bereiche verhindert wird, dass Exceptions von außen geblockt werden, sodass nicht unpassend dazwischen gefunkt werden kann.

```
mask :: ((forall a. IO a -> IO a) -> IO b) -> IO b
```

Keine Panik!

`mask` ist eine Funktion, die eine Funktion als Argument nimmt, die wiederum eine Funktion nimmt...
Die Feinheiten sind nicht enorm wichtig (und können bei Neugier oder Bedarf gerne nachgelesen werden).
Die Idee ist, dass für kritische Bereiche verhindert wird, dass Exceptions von außen geblockt werden, sodass nicht unpassend dazwischen gefunkt werden kann.

```
mask $ \restore -> do
    x <- acquire
    restore (do_something_with x) `onException` release
    release
```

```
runClient :: Server -> Client -> IO ()
runClient serv@Server{..} client@Client{..} = do
  race server receive
  return ()
 where
  receive = forever $ do
    msg <- hGetLine clientHandle
    atomically $ sendMessage client (Command msg)

  server = join $ atomically $ do
    k <- readTVar clientKicked
    case k of
      Just reason -> return $
        hPutStrLn clientHandle $
              "You have been kicked: " ++ reason
      Nothing -> do
        msg <- readTChan clientSendChan
        return $ do
            continue <- handleMessage serv client msg
            when continue $ server
```

```
race :: IO a -> IO b -> IO (Either a b)
```

`race` ist eine Funktion, die zwei `IO`-Aktionen nimmt und sie gleichzeitig startet.

```
race :: IO a -> IO b -> IO (Either a b)
```

`race` ist eine Funktion, die zwei `IO`-Aktionen nimmt und sie gleichzeitig startet.

Die erste Funktion, die Output produziert (oder eine Exception wirft), darf ihr Ergebnis nach oben weiter geben, die andere wird sofort abgebrochen.
Die beiden Argumente zu `race` befinden sich also in einem *Wettrennen* miteinander.

```
handleMessage :: Server -> Client -> Message -> IO Bool
handleMessage server client@Client{..} message =
  case message of
     Notice msg         -> output $ "*** " ++ msg
     Tell name msg      -> output $ "*" ++ name ++ "*: " ++ msg
     Broadcast name msg -> output $ "<" ++ name ++ ">: " ++ msg
     Command msg ->
       case words msg of
           ["/kick", who] -> do
               atomically $ kick server who clientName
               return True
           "/tell" : who : what -> do
               tell server client who (unwords what)
               return True
           ["/quit"] ->
               return False
           ('/':_):_ -> do
               hPutStrLn clientHandle $ "Unrecognized command: " ++ msg
               return True
           _ -> do
               atomically $ broadcast server $ Broadcast clientName msg
               return True
 where
   output s = do hPutStrLn clientHandle s; return True
```