Nebenläufige Programmierung in Haskell: Abstraktionen

Und das erleben Sie heute:

- Letztes Mal: Grundlagen der nebenläufigen Programmierung
  - Threads in Haskell
  - `MVar`
- Heute: darauf aufbauende Abstraktionen
- Längeres Beispiel: Netzwerkprogrammierung

Abstraktionen

Mit `MVars` mächtigere Synchronisationskonzepte implementieren:

- Puffernde Kanäle
- Semaphoren (allgemein, quantitativ)
- Monitore und Mutexe

Buffer, Kanäle, Semaphoren schon im Modul `Concurrent`

Abstraktion I: Kanäle

- Ein Kanal besteht aus Strom mit einem Lese- und Schreibende:

```
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
```

  - Hier `MVar`, damit Lesen/Schreiben nicht unterbrochen wird
- Ein Strom ist `MVar (ChItem a)`:
  - entweder leer,
  - oder enthält Werte aus Kopf a und Rest.

```
type Stream a = MVar (ChItem a)
data ChItem a = ChItem a (Stream a)
```

- Schnittstelle:

```
newChan   :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan  :: Chan a -> IO a
```

- Neuen Kanal erzeugen:

```
newChan :: IO (Chan a)
newChan =
  do hole  <- newEmptyMVar
     read  <- newMVar hole
     write <- newMVar hole
     return (Chan read write)
```

  ◦ NB: Leseende = Schreibende

In einen Kanal schreiben

- Neues Ende (`hole`) anlegen
- Wert in altes Ende schreiben
- Zeiger auf neues Ende setzen

```
writeChan (Chan _ write) val =
  do new_hole <- newEmptyMVar
     old_hole <- takeMVar write
     putMVar write new_hole
     putMVar old_hole (ChItem val new_hole)
```

- Kann nicht blockieren — `write` immer gefüllt.

Aus Kanal lesen

- Anfang auslesen, Anfangszeiger weitersetzen
- Kann blockieren (wenn Kanal leer)

```
readChan (Chan read _) =
  do read_end <- takeMVar read
     (ChItem val new_read_end)
                 <- readMVar read_end
     putMVar read new_read_end
     return val
```

  ◦ `readMVar :: MVar a-> IO a` liest `MVar`, schreibt Wert zurück.
  ◦ `readMVar` statt `takeMVar`, um Duplikation zu ermöglichen

Weitere Kanalfunktionen

- Zeichen wieder vorne einhängen:

```
unGetChan :: Chan a -> a -> IO ()
```

- Kanal duplizieren (Multicast):

```
dupChan :: Chan a -> IO (Chan a)
```

- Kanalinhalt als (unendliche) Liste:

```
getChanContents :: Chan a -> IO [a]
```

  - Auswertung terminiert nicht, sondern blockiert

Abstraktionen II: Semaphoren

- Abstrakter Datentyp `QSem`
- Betreten kritischer Abschnitt (P): `waitQSem :: QSem -> IO ()`
- Verlassen kritischer Abschnitt (V): `signalQSem :: QSem -> IO ()`
- Semaphore: Zähler plus evtl. wartende Threads
  - P erniedrigt Zähler, blockiert ggf. aufrufenden Thread
  - V erhöht Zähler, gibt ggf. blockierte Threads frei
- Implementierung von Semaphoren mit `MVar`: eigenes Scheduling
- Variation: Quantitative Semaphoren
  - Zähler kann um Parameter n erhöht/erniedrigt werden

Semaphoren: die P-Operation

```
data QSem = QSem (MVar (Int, [MVar ()]))
```

- `MVar` .. für die ganze Semaphore, darin:
- Zähler der Prozesse im kritischen Abschnitt
- Liste von wartenden Prozessen (`MVar ()`)

```
newQSem :: Int -> IO QSem
newQSem n = do m <- newMVar (n, [])
               return (QSem m)
```

Semaphoren: die P-Operation

Eintritt in kritischen Abschnitt:

- Wenn Eintritt möglich, Zähler erniedrigen
- Ansonsten blockieren (Reihenfolge!)

```
waitQSem :: QSem -> IO ()
waitQSem (QSem sem) = do
   (avail,blocked) <- takeMVar sem
   if avail > 0 then
     putMVar sem (avail-1,[])
    else do
     block <- newEmptyMVar
     putMVar sem (0, blocked++[block])
     takeMVar block
```

Semaphoren: die V-Operation

- Falls wartende threads, einen aufwecken.
- Alternatives Scheduling:
  - am Anfang hinzufügen, vom Anfang nehmen (einfacher, unfair)
  - am besten: zufällige Auswahl

```
signalQSem :: QSem -> IO ()
signalQSem (QSem sem) = do
   (avail,blocked) <- takeMVar sem
   case blocked of
     [] -> putMVar sem (avail+1,[])
     (block:blocked’) -> do
           putMVar sem (0,blocked’)
           putMVar block ()
```

Abstraktion III: Monitore und Mutexe

- Monitore schützen Aktionen.
- Modellierung als Typklasse:

```
class Synchronize v where
   synchronize :: v -> IO a -> IO a
```

  - Parameter `v`: Datenstruktur, über der synchronisiert wird
  - Beispiel: Semaphoren

```
instance Synchronize QSem where
   synchronize s a = do waitQSem s
                        r<- a
                        signalQSem s
                        return r
```

Reentrante Mutexe

- Problem:
  - Semaphoren (u.a.) nicht reentrant
  - keine Rekursion möglich
- Lösung: reentrantes Mutex (mutual exclusion)
- Operationen: `acquire` und `release`
- Mutex enthält:
  - `ThreadId` des besitzenden Threads, Zähler für Rekursionstiefe
  - Liste von blockierten Threads (`Id`, `MVar`)

```
data Mutex = Mutex (MVar (Maybe (ThreadId,Int),
                          [(ThreadId, MVar ())]))
```

Eintritt in den kritischen Abschnitt

- Mutex akquirieren:

```
acquire :: Mutex -> IO ()
```

  - Wenn Thread Mutex besitzt, dann Zähler erhöhen;
  - sonst neue `MVar ()` erzeugen, ThreadId und MVar in Liste einhängen, auf MVar warten;

Eintritt in den kritischen Abschnitt

```
acquire :: Mutex -> IO ()
acquire (Mutex mvar) =
   do st <- takeMVar mvar
      current <- myThreadId
      case st of
         (Nothing,[]) -> putMVar mvar (Just (current,1),[])
         (Just (holder,n),pnd) ->
            if current == holder
              then putMVar mvar (Just (holder,n+1),pnd)
              else do bsem <- newEmptyMVar
                      putMVar mvar (Just (holder,n),
                                    (current,bsem):pnd)
                      takeMVar bsem
```

Verlassen des kritischen Abschnitts

- Mutex freigeben:

```
release :: Mutex -> IO ()
```

  - Prüfen, ob Thread Mutex besitzt (sonst Fehler)
  - Wenn Zähler 1 und Liste leer, wird Mutex leer
  - Wenn Zähler 1, dann ersten Thread aus Liste aufwecken
  - Wenn Zähler größer 1, dann erniedrigen

Verlassen des kritischen Abschnitts

```
release :: Mutex -> IO ()
release (Mutex mvar) =
   do st <- takeMVar mvar
      current <- myThreadId
      case st of
         (Just (h,n),pnd) | current == h ->
            release' mvar h n pnd
         _ -> do putMVar mvar st
                 error "Illegal lock release"  where
    release' mvar _ 1 [] = putMVar mvar (Nothing,[])
    release' mvar _ 1 ((h',sem):pnd') =
         do putMVar mvar (Just (h',1),pnd'); putMVar sem ()
    release' mvar h n pnd = putMVar mvar (Just (h,n-1), pnd)
```

Reentrante Monitore

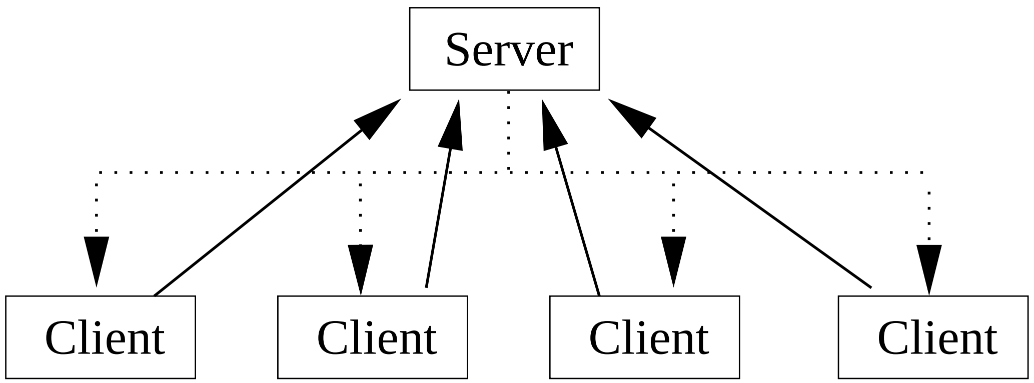
Mit Mutex Implementierung reentranter Monitore:

```
instance Synchronize Mutex where
  synchronize m a =
    do acquire m
       r<- a
       release m
       return r
```

Rekursion möglich (aber teuer).

Längeres Beispiel: It's good to talk

- Ziel: ein Programm, um sich über das Internetz zu unterhalten (`talk`, `IRC`, etc.)
- Verteilte Architektur:



- Hier: Implementierung des Servers
  - Netzverbindungen durch Socket

Socketprogrammierung

- Socket erzeugen, an Namen binden, mit `listen` Verbindungsbereitschaft anzeigen
- Zustandsbasierte Verbindung:
  - Serverseite: mit `accept` auf eingehende Verbindungen warten
  - Jede Verbindung erzeugt neuen Filedescriptor
    $\Rightarrow$ inhärent nebenläufiges Problem!
  - Clientseite: Mit `connect` Verbindung aufnehmen.
- Zustandslose Verbindung: `sendTo` zum Senden, `recvFrom` zum Empfangen.
- GHC-Modul `Network`
  - Low-level Funktionen in `Network.Socket`

Das Modul `Network`

```
type Socket
data PortID = Service String -- z.B. "ftp"
            | PortNumber PortNumber
            | UnixSocket String -- Socket mit Namen (im Dateisystem)
type Hostname = String
instance Num PortNumber
-- Zustandsbasiert:
listenOn   :: PortID-> IO Socket
accept     :: Socket-> IO (Handle, Hostname, PortNumber)
connectTo  :: Hostname -> PortID -> IO Handle
-- Zustandslos:
sendTo   :: HostName -> PortID -> String -> IO ()
recvFrom :: HostName -> PortID -> IO String
```

Serverarchitektur

- Ein Kanal zur Nachrichtenverbreitung:
  - eine Nachricht, viele Empfänger (broadcast)
  - Realisierung mittels `dupChan`
- Zentraler Scheduler
- Für jede ankommende Verbindung neuer Thread:
  - Nachrichten vom Socket auf den Kanal schreiben
  - Nachrichten vom Kanal in den Socket schreiben
- Problem: Wie aus Socket oder Kanal lesen wenn beide blockieren?
- Lösung: Zwei Threads
- Clientarchitektur: `telnet`

Talk 0.1: Hauptprogramm

```
main :: IO ()
main =
  do portNum <- getArgs >>= return . read . head
     s <- listenOn (PortNumber (fromInteger portNum))
     ch<- newChan
     loop s ch
```

Talk 0.1: Hauptschleife

```
loop :: Socket -> Chan String -> IO ()
loop s ch =
  do (handle, wh, p) <- accept s
     hSetBuffering handle NoBuffering
     putStrLn ("New connection from "++ wh++ " on "++ show p)
     ch2    <- dupChan ch
     forkIO (newUser handle ch2)
     loop s ch
```

Talk 0.1: Benutzerprozess

```
newUser :: Handle-> Chan String -> IO ()
newUser socket msgch =
  forkIO read >> write where
    read :: IO ()
    read  = hGetLine socket >>= writeChan msgch >> read
    write :: IO ()
    write = readChan msgch >>= hPutStrLn socket >> write
```

Zeigen!

Talk 0.1: Zusammenfassung

Übersetzen mit `-package concurrent -package net`

Nachteile:

- Nachrichten stauen sich im Kanal
- Serverprozess fällt um, wenn Benutzer Verbindung beendet
- Keine Fehlerbehandlung
- Benutzer anonym

Zusammenfassung

- Abstraktionen über einfachen `MVars`:
  - Kanäle
  - Semaphoren
  - Monitore
- Haskell: the world's finest concurrent programming language?
- Längeres Beispiel: Talk.