Konzepte der Nebenläufigkeit

• *Thread (lightweight process)*	vs.	*Prozess*
Programmiersprache (z.B. Java, Concurrent Haskell, ...)		Betriebssystem
Gemeinsamer Speicher		getrennter Speicher
Mehrere pro Programm		einer pro Programm

- Multitasking:

  präemptiv: Kontextwechsel kann erzwungen werden

  kooperativ: Kontextwechsel nur freiwillig

Zur Erinnerung: *Threads* in Java

- Erweiterung der Klassen `Thread` oder `Runnable`
- Gestartet wird Methode `run` — durch eigene überladen
- Starten des Threads durch Aufruf der Methode `start()`
- Kontextwechsel mit `yield()`
- **Kooperativ**, möglicherweise (!) **präemptiv**.

Threads in Haskell: Concurrent Haskell

- Sowohl `ghc` als auch `hugs` implementieren Threads auf Haskell-Ebene.
- `ghc`: *präemptiv*, Hugs: *kooperativ*
- Modul `Concurrent` enthält Basisfunktionen
- Wenige Basisprimitive, darauf aufbauend Abstraktionen

Threads in Haskell: Concurrent Haskell

- Schnittstelle: Jeder Thread hat einen Identifier (`ThreadId`)

  `forkIO:: IO()-> IO ThreadId` erzeugt neuen nebenläufigen Thread.

  `killThread :: ThreadId -> IO ()` stoppt Thread.

  Kontextwechsel mit `yield :: IO ()`

  Eigener Thread ist `myThreadId :: IO ThreadId`.

  `threadDelay :: Int -> IO ()` wartet `n` Mikrosekunden (Auflösung $\cong 50ms$)
- Blockierung:

  Systemaufrufe blockieren *alle Threads* (den gesamten Haskell-Prozeß)

  Aber: Haskell Standard-IO blockiert *nur den aufrufenden Thread*

Concurrent Haskell — erste Schritte

Ein einfaches Beispiel

```
import Concurrent

write c = forever (putChar c)

main :: IO ()
main = forkIO (write 'X') >> write 'O'
```

Übersetzen: `ghc -package concurrent simple.hs`

Mit Hugs keine besonderen Optionen nötig.

Synchronisation mit `MVar`s

- `MVar a` veränderbare Variable: entweder leer gefüllt mit einem `a`
- `newEmptyMVar :: IO (MVar a)`, `newMVar :: a -> IO (MVar a)`
  erzeugt neue, leere bzw. mit initialem Wert gefüllte `MVar`
- `takeMVar :: MVar a -> IO a`
  liest Wert aus `MVar`, hinterläßt sie leer, blockiert wenn leer
- `putMVar :: MVar a -> a -> IO ()`
  schreibt neuen Wert in `MVar`; wenn `MVar` voll, gibt es *Fehler*
- `tryTakeMVar :: MVar a -> IO (Maybe a)`
  liest `MVar` nicht-blockierend
- `isEmptyMVar :: MVar a -> IO Bool`
  liefert `True` wenn `MVar` leer. Achtung: Zustand verändert sich!

Ein einfaches Beispiel ohne Synchronisation

Nebenläufige Eingabe von der Tastatur ohne Synchronisation:

```
echo :: String-> IO ()
echo p = do putStrLn ("Please enter line "++p)
            line <- getLine
            randomRIO (1,100) >>= \n-> forN n (putStr (p++ line))
            echo p
main :: IO ()
main = forkIO (echo "1") >> echo "2"
```

Problem: beide Prozesse fordern gleichzeitig zur Eingabe auf

Lösung: `MVar` synchronisiert Eingabe

Ein einfaches Beispiel mit Synchronisation

`MVar` voll $\Leftrightarrow$ Eingabe möglich

Typ des Inhalts der `MVar` irrelevant (also `MVar ()`)

```
echo :: MVar ()-> String-> IO ()
echo flag p = do takeMVar flag
                 putStrLn ("Please enter line "++p)
                 line <- getLine
                 putMVar flag ()
                 randomRIO (1,100) >>= \n-> forN n (putStr (p++ line))
                 yield  -- noetig, da sonst keine Fairness.
                 echo flag p
main :: IO ()
main = do flag <- newMVar ()
          forkIO (echo flag "1") >> echo flag "2"
```

Das Standardbeispiel: die speisenden Philosophen

Philosoph i: vor dem Essen i-tes und $(i+1)$ mod n-tes Stäbchen nehmen, nach dem Essen wieder zurücklegen

Stäbchen modelliert als `MVar ()`

```
philo :: [MVar ()] -> Int-> IO ()
philo chopsticks i =
   let num_phil = length (chopsticks)
   in do putStrLn ("Phil #"++(show i)++" thinks...")
         takeMVar (chopsticks !! i)
         takeMVar (chopsticks !! ((i+1) `mod` num_phil))
         putStrLn ("Phil #"++(show i)++" eats...")
         putMVar (chopsticks !! i) ()
         putMVar (chopsticks !! ((i+1) `mod` num_phil)) ()
         philo chopsticks i
```


Das Standardbeispiel: die speisenden Philosophen (2)

Hauptfunktion: n Stäbchen erzeugen

Anzahl Philosophen in der Kommandozeile

```
main :: IO ()
main = do num_phil <- getArgs >>= \a-> return (read (head a))
          chopsticks <- mapM (const (newMVar ())) [0.. num_phil-1]
          mapM_ (forkIO . (philo chopsticks)) [0.. num_phil-1]
          block
```

Hilfsfunktion: blockiert aufrufenden Thread

```
block :: IO ()
block = newEmptyMVar >>= takeMVar
```


Abstraktionen I: Puffer

Leser-Schreibe-Problem — kein Überschreiben.

Eine `MVar a` für den Wert Schreiber→Leser,
eine `MVar ()` für die Synchronisation Leser→Schreiber

```
data CVar a = CVar (MVar a) (MVar ())

newCVar :: IO (CVar a)
newCVar = newEmptyMVar >>= \ datum ->
          newMVar ()   >>= \ ack ->
          return (CVar datum ack)
```


Abstraktionen I: Puffer

```
writeCVar :: CVar a -> a -> IO ()
writeCVar (CVar datum ack) val
 = takeMVar ack      >>
   putMVar datum val >>
   return ()

readCVar :: CVar a -> IO a
readCVar (CVar datum ack)
 = takeMVar datum >>= \ val ->
   putMVar ack () >>
   return val
```

Nachteil: immer nur einen Wert zur Zeit.

Besser: puffernder *Kanal*

Abstraktion I: Buffered Channels

Ein Kanal hat einen *Lesestrom* und einen *Schreibstrom*:

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

Ein Strom ist entweder leer, oder enthält Werte (Typ `ChItem a`) aus einem Kopf `a` und einen 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
```


Abstraktion I: In einen Kanal schreiben

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

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


Abstraktion I: Aus Kanal lesen

Anfang auslesen, Anfangszeiger weitersetzen

Kann blockieren (wenn Kanal leer)

```
readChan (Chan read _write) = 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
readMVar mvar = do val <- takeMVar mvar
                   putMVar mvar value
                   return value
```


Abstraktion I: Weitere Kanalfunktionen

- Zeichen wieder vorne einhängen:

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

  `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`
- `waitQSem :: QSem -> IO ()` (P) für Betreten des kritischen Abschnitts
- `signalQSem :: QSem -> IO ()` (V) beim Verlassen des kritischen Abschnitts
- Semaphore enthält intern Zähler plus evtl. wartende Threads;
  `waitQSem` erniedrigt Zähler, blockiert ggf. aufrufenden Thread
  `signalQSem` erhöht Zähler, gibt ggf. blockierte Threads frei
- Implementierung von Semaphoren mit `MVars`: eigenes Scheduling
- Variation: *binäre Semaphoren* — nur zwei Zustände
- Variation: *Quantitative Semaphoren* — Zähler kann um Parameter n erhöht/erniedrigt werden

Abstraktionen II: Semaphoren

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

newQSem :: Int -> IO QSem

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


Abstraktionen II: Semaphoren

```
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 ()
```

Alternatives Scheduling:

- am Anfang hinzufügen, vom Anfang nehmen (einfacher, kann aushungern)
- zufällige Auswahl

Abstraktion III: Monitore

Monitore können *Aktionen* schützen (festgelegte Auswertungsreihenfolge)

Modellierung als Typklasse:

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

Parameter `v`: Datenstruktur über der synchronisiert wird, z.B. Semaphoren:

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

Vgl. Java — Parameter `v` implizit.

Abstraktionen IV: Reentrante Wechselseitige Aussschlüsse

Problem: Semaphoren usw. nicht *reentrant* (kann sich selbst blockieren), dadurch *keine Rekursion* möglich.

Lösung: *reentrantes Mutex* (mutual exclusion)

Mutex enthält:
- `ThreadId` des besitzenden Threads
- Liste von blockierten Threads (`Id`, `MVar`)

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


Abstraktion IV: Mutex akquirieren

```
acquire :: Mutex -> IO ()
acquire (Mutex mvar) =
   do st <- takeMVar mvar
      current <- myThreadId
      case st of
         -- Mutex gehört keinem: in Besitz nehmen
         (Nothing,[]) -> putMVar mvar (Just (current,1),[])
         (Just (holder,n),pnd) ->     -- Mutex gehört uns: Zähler erhöhen
            if current == holder then putMVar mvar (Just (holder,n+1),pnd)
              else -- Mutex gehört uns nicht: in Warteschlange einfügen
                   do bsem <- newEmptyMVar
                      putMVar mvar (Just (holder,n),(current,bsem):pnd)
                      takeMVar bsem
```


Abstraktion IV: Mutex freigeben

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


Abstraktion IV: Reentrante Monitore

Mit Mutex Implementierung reentranter Monitore:

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

Verwendung (*schlechtes Beispiel*):

```
  output m h =
    do line<- ...
       synchronize m (hPutStrLn line1 >> output m h)
  m <- newMutex
  h<- openFile "output.txt" WriteMode
  ...
  output m h
  ...
```

Zusammenfassung

Concurrent Haskell bietet

- Threads auf Quellsprachenebene
- Synchronisierung mit `MVars`
- Durch schlankes Design einfache Implementierung im Compiler
- Funktionales Paradigma erlaubt Konstruktion weiterer Abstraktionen: Channels, Semaphoren, Monitore, ...