Vorlesung "Modellierung nebenläufiger Systeme"
Sommersemester 2012
Universität Duisburg-Essen

Barbara König
Übungsleitung: Henning Kerstan

Anwendung: Google Go

Go ist eine Programmiersprache, die von Google entwickelt wird (`http://golang.org/`)

Eigenschaften von Go

- Typ- und Speichersicherheit, statisches Typsystem
- Unterstützung von Nebenläufigkeit und Kommunikation
- Garbage Collection
- Schnelle Kompilierung
- Unterstützung von Objekt-Orientierung (keine Klassen, aber Typen können Methoden zugeordnet werden)

Anwendung: Google Go

Unterstützung von Nebenläufigkeit und Kommunikation

- Funktionen können in separaten Prozessen ausgeführt werden (die Prozesse sind leichtgewichtig und verwenden denselben Adressraum)
- Kanäle (ähnlich zu Kanälen des π-Kalküls)

Philosophie: "Do not communicate by sharing memory. Instead share by communicating."

Synchronisation sollte mit Hilfe von Kommunikation erreicht werden, nicht mit Hilfe von Locks und Semaphoren.

Nebenläufigkeit und Kommunikation in Go

Goroutinen

Eine Funktion kann als eigener leichtgewichtiger Prozess (Thread) ausgeführt werden:

```
go f(x,y,z)
```

Wenn eine Funktion terminiert, so werden auch die aus ihr aufgerufenen Goroutinen beendet.

Nebenläufigkeit und Kommunikation in Go

Kanäle

Kanäle sind getypt, d.h., man kann nur Nachrichten eines bestimmten Typs verschicken. Außerdem ist "Kanal" (`chan`) selbst auch ein Datentyp.

Erzeugen eines (Integer-)Kanals:

```
ch := make(chan int)
```

Den Wert der Variable `v` über den Kanal `ch` versenden:

```
ch <- v
```

Wert über den Kanal `ch` empfangen, dieser Wert wird `v` zugewiesen:

```
v := <- ch
```

Nebenläufigkeit und Kommunikation in Go

Synchrone/asynchrone Kommunikation

Kommunikation ist im Standardfall synchron. Asynchrone Kommunikation kann durch Puffer in den Kanälen (d.h., Kanäle bestimmter Kapazität) erreicht werden.

Folgender Kanal `ch` hat Kapazität 100. Senden blockiert erst dann, wenn der Kanal voll ist.

```
ch := make(chan int, 100)
```

Nebenläufigkeit und Kommunikation in Go

Warten auf mehreren Kanälen

Es ist möglich, auf mehreren Kanälen gleichzeitig auf eine Nachricht zu warten. Liegt auf keinem Kanal eine Nachricht an, wird entweder der Default-Fall genommen oder – wenn es diesen nicht gibt – blockiert, bis eine Nachricht verfügbar ist.

```
select {
   case v := <- ch1:
      command1
   case v := <- ch2:
      command2
   default:
      command3
}
```

Bei mehreren Möglichkeiten, wird eine davon zufällig ausgewählt.

Nebenläufigkeit und Kommunikation in Go

Arten von Nachrichte auf Kanälen

Auf Kanälen können auch folgende Objekte verschickt werden:

- Funktionen
- Strukturen (structs)
- Andere Kanäle (wie bei der Mobilität im π-Kalkül)

```
var ChanOfChan = make(chan chan int)
```

Beispiele

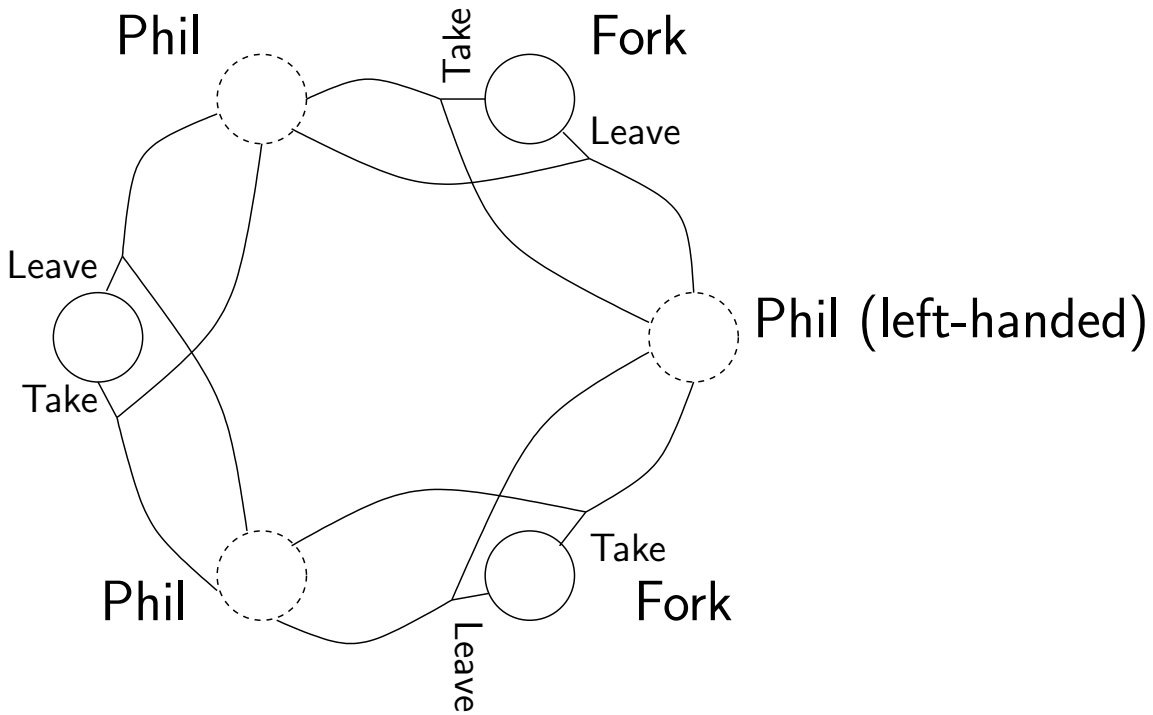
Wir betrachten nun vier Code-Beispiele:

- Dining Philosophers
- Alternating Bit Protocol
- Versenden von Kanälen (Mobilität)

Dabei werden nur die interessantesten Code-Ausschnitte angegeben.

Beispiel: Dining Philosophers

Es werden n Philosophen (einer linkshändig, alle anderen rechtshändig) mit n Gabeln modelliert. Die Kommunikationsstruktur sieht für den Fall $n = 3$ wie folgt aus:



Beispiel: Dining Philosophers

Eine Gabel wird durch eine Struktur `Fork` beschrieben, die neben Attributen auch zwei Kanäle beinhaltet, über die mit der Gabel kommuniziert werden kann:

```
type Fork struct {
   name string          //  fork name
   free bool            //  is it free?

   // (public) channels
   Take chan int        //  request of taking
   Leave chan int       //  release
}
```

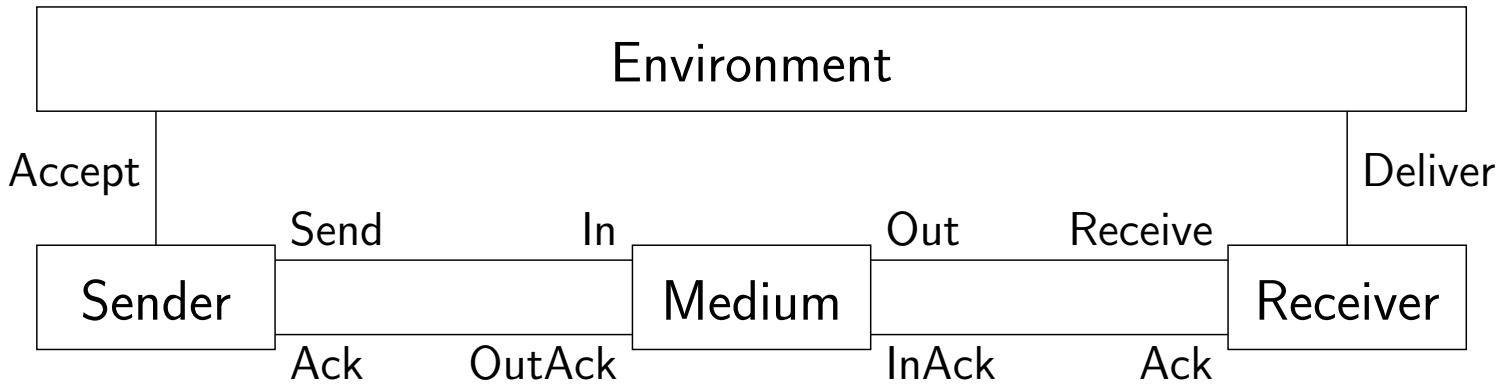
Beispiel: Dining Philosophers

Zum Typ `Fork` gibt es eine Funktion `Run`: in einer Schleife werden Signale aus den Kanälen ausgelesen und der Zustand geändert.

```
func (self *Fork) Run () {
  for {
    if self.free {
      <- self.Take
      self.free = false
      fmt.Printf("Fork %s taken\n", self.name)
    } else {
      <- self.Leave
      self.free = true
      fmt.Printf("Fork %s released\n", self.name)
    }
  }
}
```

Beispiel: Alternating Bit Protocol

Wir modellieren nun das Alternating Bit Protocol – ähnlich wie in CCS – mit Hilfe von Google Go. Dabei sieht die Kommunikationsstruktur schematisch wie folgt aus:



Beispiel: Alternating Bit Protocol

Der Sender im Alternating Bit Protocol wird mit Hilfe von drei Kanälen modelliert:

```
type Sender struct {
  msg int          //  which message?

  Accept chan int   // channel for accepting messages
  Send chan [2]int  //  channel for sending messages
  Ack chan int      //  channel for acknowledgements
}
```

Beispiel: Alternating Bit Protocol

Wie bei der Modellierung in CCS wird solange eine Nachricht verschickt, bis eine Bestätigung ankommt. Dann wird zum nächsten Bit gewechselt.

```
func (self *Sender) RunSender () {
  var ack int
  var content int

  ack = 1

  ...
```

Beispiel: Alternating Bit Protocol

```
  ...

  for {
    content = <- self.Accept
    for ack!=self.msg {
      self.Send <- [2]int{self.msg,content}
      select {
        case ack = <- self.Ack:
        default:
      }
    }
    self.msg = 1 - self.msg
  }
}
```

Beispiel: Alternating Bit Protocol

Das Medium verliert zufällig Nachrichten:

```
func (self *Medium) ForwardMsg () {
  for {
    arr := <- self.In
    if (rand.Float32() <= p) {
      fmt.Printf("Msg not lost\n")
      self.Out <- arr
    } else {
      fmt.Printf("Msg lost\n")
    }
  }
}
```

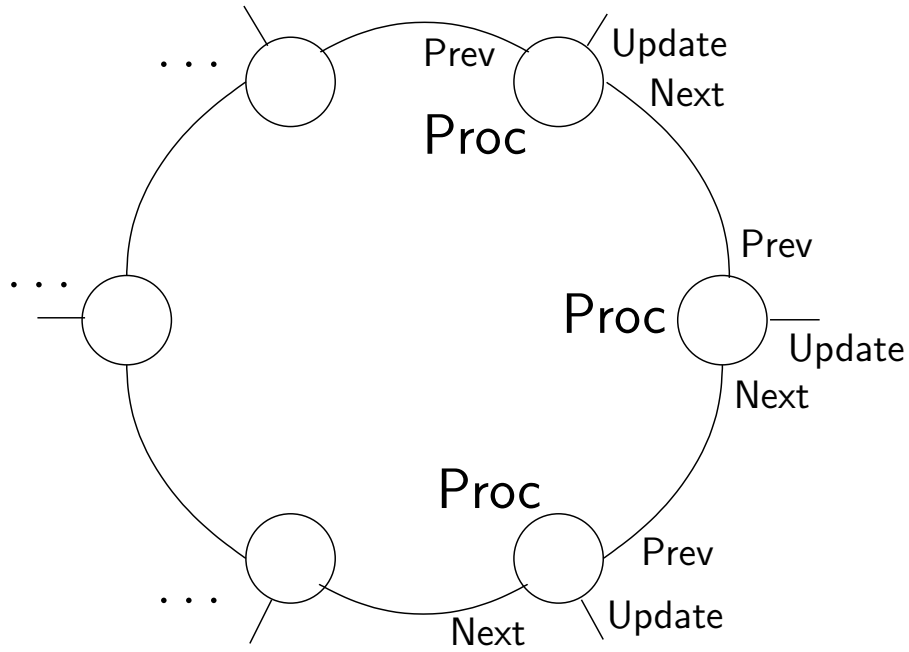
Beispiel: Alternating Bit Protocol

Der Empfänger wird ähnlich zum Sender modelliert.

Außerdem gibt es eine Umgebung, die dem Sender Nachrichten mit zufälligem Inhalt zum Verschicken bereitstellt und diese auch vom Empfänger abholt (accept, deliver).

Beispiel: Versenden von Kanälen – Mobilität

Um Mobilität von Kanälen zu modellieren, betrachten wir einen Ring von Prozessen, die sich gegenseitig reihum Nachrichten schicken. Nach einer bestimmten Zeit wird von außen über die Update-Kanäle die Ringkonfiguration geändert, so dass der Ring anschließend in die Gegenrichtung zeigt.



Beispiel: Versenden von Kanälen – Mobilität

Der Typ Prozess wird folgendermaßen beschrieben:

```
type Proc struct {
  nr int
  Prev chan int
  Next chan int
  Update (chan [2](chan int))
}
```

Beispiel: Versenden von Kanälen – Mobilität

```
func (self *Proc) Run () {
  if self.nr==0 { self.Next <- 1 }
  for {
    select {
      case upd := <- self.Update:
        self.Prev = upd[0]
        self.Next = upd[1]
      default:
    }
    <- self.Prev
    fmt.Printf("Message received: %d\n",self.nr)
    self.Next <- 1
  }
}
```

Beispiel: Versenden von Kanälen – Mobilität

Nun muss noch der Ring "zusammengesteckt" werden:

```
const NProc = 5

func main () {
  var c [NProc](chan int)
  var cm [NProc] (chan [2](chan int))
  var procs [NProc]Proc
  var i int

  for i=0; i<NProc; i++ {
    c[i] = make(chan int,10)
    cm[i] = make(chan [2](chan int),10)
  }

  ...
```

Beispiel: Versenden von Kanälen – Mobilität

```
...

for i=0; i<NProc; i++ {
  // create the ith process
  procs[i] = Proc {i,c[i],c[(i+1)%NProc],cm[i]}
  go procs[i].Run()
}
```

Beispiel: Versenden von Kanälen – Mobilität

Außerdem wird in bestimmten Zeitabständen der Ring umgedreht:

```
  for {
    time.Sleep(4000000000)
    fmt.Printf("Reverse\n")
    // reverse the ring
    for i=0; i<NProc; i++ {
      procs[i].Update <-
        [2](chan int){procs[i].Next,procs[i].Prev}
    }
  }
}
```

Verwandte Sprachen

Verwandte Sprachen:

Auch andere Sprachen versuchen, Nebenläufigkeit, Synchronisation und Kommunikation ähnlich wie Google Go zu realisieren.
Insbesondere:

- Erlang (seit 1986, Joe Armstrong, Ericsson)
- Scala (seit 2003, Martin Odersky, EPFL Lausanne)
- ...