Programmierprimitiven: Kanäle

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 27. November 2020

Übersicht

Shared Memory vs. Message Passing

- Die bisherigen Programmierkonstrukte Semaphore und Monitore werden im gemeinsamen Speicher benutzt, um Prozesskommunikation bzw Synchronisation zu ermöglichen
- In diesem Abschnitt: Kanäle (Channels)
- Diese benötigen keinen gemeinsamen Speicher
- Synchronisation und Kommunikation nur über das Empfangen und Senden von Nachrichten
- Implementierung in gemeinsamen Speicher möglich
- Aber: Auch verwendbar in verteilten Systemen

Synchron vs. Asynchron

- Synchrone Kanäle:
  Empfangen und Senden geschieht in einem Schritt, d.h.
  - Sender wird blockiert bis Empfänger da ist
  - Empfänger wird blockiert bis Sender da ist
- Asynchrone Kanäle: Empfangen und Senden kann zu unterschiedlichen Schritten geschehen
- Wir betrachten jetzt: Synchrone Kanäle
- Solche Kanäle wurden von C.A.R Hoare eingeführt im sog. Communicating sequential processes-Formalisum (CSP)
- Kanäle sind oft als Bibliotheken für Programmiersprachen implementiert.
- Google Go: Kanäle sind nativ eingebaut

Kanäle

- Ein Kanal verbindet einen sendenden Prozess mit einem empfangenden Prozess
- Oft erlaubt: ein Kanal verbindet mehrere sendende und empfangende Prozesse
- Kanäle sind typisiert: Nur Elemente (Nachrichten) gleichen Typs können über den Kanal verschickt werden

Kanäle: Operationen

- Sei ch ein Kanal
- ch $\Leftarrow$ w
  - entspricht: "sende w über den Kanal ch"
  - dabei ist w ein Wert vom passenden Typ
  - wir schreiben auch ch $\Leftarrow$ x, für eine Programmvariable x

  Semantik: Sende den Wert der Variablen x über Kanal ch
- ch $\Rightarrow$ x
  - entspricht "empfange über den Kanal ch und setze Variable x auf den empfangenen Wert"
  - Hier: Nur Variablen erlaubt!

Beispiel: Erzeuger / Verbraucher ohne Pufferung

ch: Kanal über dem Typ τ
y: Programmvariable vom Typ τ

Erzeuger:	Verbraucher:
`loop forever`	`loop forever`
(1) erzeuge e (vom Typ τ);	(1) ch $\Rightarrow$ y;
(2) ch $\Leftarrow$ e;	(2) verbrauche ey;
`end loop`	`end loop`

Auswertung: Kommunikation nicht möglich, Erzeuger ist blockiert
Kommunikation möglich,
Kommunikation geschieht in einem Schritt Beide Programmzeiger springen direkt weiter!
Sender und Verbraucher blockieren,
solange kein "Gegenstück" vorhanden ist

Ein Kanal – mehrere Sender / Empfänger

Prozess 1:	Prozess 2:	Prozess 3:
ch $\Leftarrow$ True	ch $\Leftarrow$ False	ch $\Rightarrow$ x
		print x;

Was druckt Prozess 3 aus?

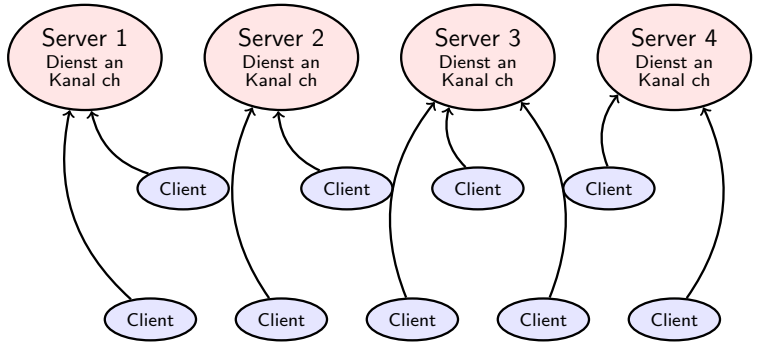
Analog

Prozess 1:	Prozess 2:	Prozess 3:
ch $\Rightarrow$ x	ch $\Rightarrow$ x	ch $\Leftarrow$ True
print x;	print x;	

Wer druckt True aus?

Je nach Ablauf (quasi zufällig)

Kann sinnvoll sein ...



"Lastverteilung automatisch"

Nachteil alles läuft über einen Kanal!

Kanäle in der Programmiersprache Go

- Initialisieren eines Kanals: `make(chan type)` öffnet einen Kanal mit Inhalt vom Typ `type`, z.B.

```
kanal := make(chan string)
```

- Senden: Anstelle von ch $\Leftarrow$ w, schreibt man in Go

```
ch <- w
```

z.B.

```
kanal <- "Hallo"
```

- Empfangen: Anstelle von ch $\Rightarrow$ x, schreibt man in Go

```
x := <- ch
```

z.B.

```
x := <- kanal
```

Bzw. wenn man das empfangene Element nicht benötigt:

```
<- kanal
```


Beispiel in Go

```
// Quelle:  https://de.wikipedia.org/wiki/Go_(Programmiersprache)
package main
import "fmt"

func zehnMal(kanal chan string) {
 sag := <- kanal                    // Argument empfangen
 for i := 0; i < 10; i++ {          // Zehnmal senden
   kanal <- sag          }
 close(kanal)                       // Kanal schliessen
}

func main() {
 kanal := make(chan string) // synchronen Kanal oeffnen
 go zehnMal(kanal)          // Starten der parallelen Go-Routine zehnMal
 kanal <- "Hallo"           // Senden eines Strings
 // Empfangen der Strings, bis der Channel geschlossen wird
 for s := range kanal { fmt.Println(s) }
 fmt.Println("Fertig!")
}
```

Mutual-Exclusion mit Kanälen

- Idee: Ein Prozess als Wächter, der den kritischen Abschnitt bewacht
- Nur wer eine Nachricht vom Wächter erhält, darf in den kritischen Abschnitt

Mutual-Exclusion mit Kanälen (2)

mutex:	Kanal über dem Typ Bool
local:	lokale Variable, Initialwert egal

Prozess i

```
loop forever
(1)  Restlicher Code;
(2)  mutex ⇒ local;
(3)  Kritischer Abschnitt;
(4)  mutex ⇐ True;
end loop
```

Wächter

```
loop forever
(1)  mutex ⇐ True;
(2)  mutex ⇒ local;
end loop
```


Mutual-Exclusion mit Kanälen

- erfüllt Mutual-Exclusion
- ist Deadlock-frei (ein wartender Prozess erhält immer die Nachricht)
- nicht Starvation-frei

Mutual-Exclusion in Go

```
package main
import ("fmt"; "strconv"; "bufio"; "os")

func worker(mutex chan bool,  work string) {
 for { // Endlosschleife
  <- mutex // empfange
  fmt.Println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
  fmt.Println("> Thread " + work + " jetzt im Kritischen Abschnitt")
  fmt.Println("> Thread " + work + " noch im Kritischen Abschnitt")
  fmt.Println("> Thread " + work + " gleich aus dem Kritischen Abschnitt")
  fmt.Println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
  mutex <- true // sende  }
}

func guard(mutex chan bool) {
 for { // Endlosschleife
  mutex <- true
  <- mutex               }
}
```

Mutual-Exclusion in Go (2)

```
func main() {
    // synchronen Kanal öffnen
    mutex := make(chan bool)
    // Waechter starten
    go guard(mutex)
    // Zehn Worker starten
    for i := 0; i < 10; i++ {
        go worker(mutex,strconv.Itoa(i))
    }
    // Auf Eingabe warten, damit Programm nicht sofort endet
    reader := bufio.NewReader(os.Stdin)
    reader.ReadString('\n')
}
```


Modellierung von gemeinsamen Speicher

- Selbst wenn kein gemeinsamer Speicher vorhanden ist, kann dieser mit Kanälen modelliert werden.
- Idee: Ein Prozess stellt den Speicher dar und "verwaltet" ihn.
- Zugriff auf den Speicher mittels Senden und Empfangen von Nachrichten

Eine Speicherzelle

- Wir betrachten eine Speicherzelle
- Operationen: read und write
- Implementierung benutzt zwei Kanäle:
  - requestChannel:
    Über diesen Kanal werden read- oder write-Anfragen an den Verwalter geschickt
  - replyChannel:
    Über diesen Kanal antwortet der Verwalter
- Typ des Zelleninhalts: CellType
- Typ des requestChannel: (Bool, CellType)
- Typ des replyChannel: CellType

Eine Speicherzelle (2)

Variablen:
x: Lokale Variable des Server-Prozesses vom Typ CellType
dummy: Irgendeine Variable vom Typ CellType

Server-Prozess für die Zelle:

```
loop forever
  requestChannel ⇒ r;
  if fst(r) then// write-Operation
    x := snd(r);
  else // read-Operation
    replyChannel ⇐ x;
end loop
```

Methoden für den Zugriff auf die Zelle:

```
read(requestChannel, replyChannel) {
  requestChannel ⇐ (False,dummy);
  replyChannel ⇒ x;
  return(x);
}

write(reqCh, replyCh, x) {
  requestChannel ⇐ (True,x)
}
```

Eine Speicherzelle (3)

Variante: Server als rekursive Funktion

Server-Prozess für die Zelle:

```
cell(x) {
  requestChannel ⇒ r;
  if fst(r) then// write-Operation
    cell(snd(r));
  else // read-Operation
    replyChannel ⇐ x;
    cell(x); }
```

Methoden für den Zugriff auf die Zelle:

```
read(requestChannel, replyChannel) {
  requestChannel ⇐ (False,dummy);
  replyChannel ⇒ x;
  return(x);
}

write(reqCh, replyCh, x) {
  requestChannel ⇐ (True,x)
}
```


Selective Input

- Erweiterung um ein weiteres Konstrukt
- ermöglicht wartenden Empfang auf mehreren Kanälen
- Sobald Nachricht auf einem Kanal empfangen wird, werden andere Kanäle nicht mehr berücksichtigt
- Nichtdeterministische Auswahl bei mehreren Möglichkeiten+

```
either
  ch1 ⇒ var1
or
  ch2 ⇒ var2
or
  ch3 ⇒ var3
```

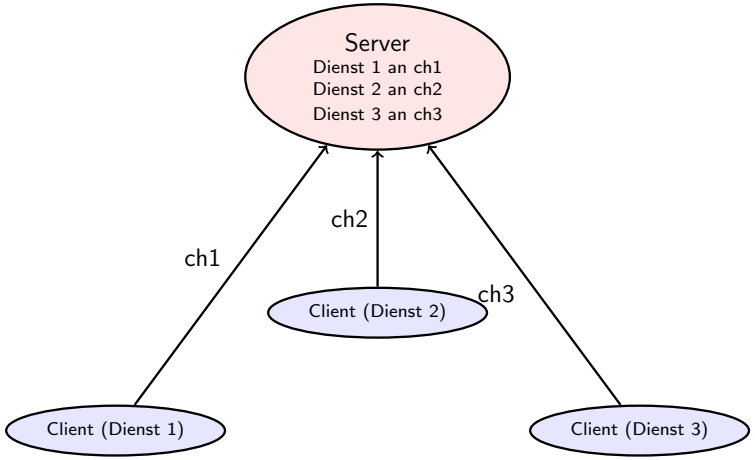

Selective Input (2)

Prozess 1	Prozess 2	Prozess 3	Prozess 4
either ch1 ⇒ var1 or ch2 ⇒ var2 or ch3 ⇒ var3	ch1 ⇐ e1	ch2 ⇐ e1	ch3 ⇐ e1

3 Möglichkeiten danach

- Prozess 1 (var1 := e1) | Prozess 2 | Prozess 3 ch2 ⇐ e2 | Prozess 4 ch3 ⇐ e3
- Prozess 1 (var2 := e2) | Prozess 2 ch1 ⇐ e1 | Prozess 3 | Prozess 4 ch3 ⇐ e3
- Prozess 1 (var3 := e3) | Prozess 2 ch1 ⇐ e1 | Prozess 3 ch2 ⇐ e2 | Prozess 4

Z.B. nützlich bei . . .



Selective Input in Go

- Das Schlüsselwort `select` stellt in Go die Möglichkeit bereit, an mehreren Kanälen gleichzeitig zu warten.

Anstelle von

```
either
  ch1 ⇒ var1
or
  ch2 ⇒ var2
or
  ch3 ⇒ var3
```

In Go

```
select {
  case var1 := <- ch1:
    code1
  case var2 := <- ch2:
    code2
  case var3 := <- ch3:
    code3
}
```


Go-Beispiel mit select

```
import ("fmt";"time";"math/rand")

func sleepAndWriteToChannel(c chan string,content string)   {
 var n = rand.Intn(1000)
 time.Sleep(time.Duration(n) * time.Millisecond) // warte
 c <- content                                    // schreibe }

func main() {
 c1 := make(chan string)
 c2 := make(chan string)
 // 2 Go-Routinen starten:
 go sleepAndWriteToChannel(c1,"one")
 go sleepAndWriteToChannel(c2,"two")
 for i := 0; i < 2; i++ {
 // Gleichzeitiges Lauschen an Kanaelen c1 und c2
 select {
   case msg1 := <-c1:  fmt.Println("received", msg1)
   case msg2 := <-c2:  fmt.Println("received", msg2)
   }}}
```

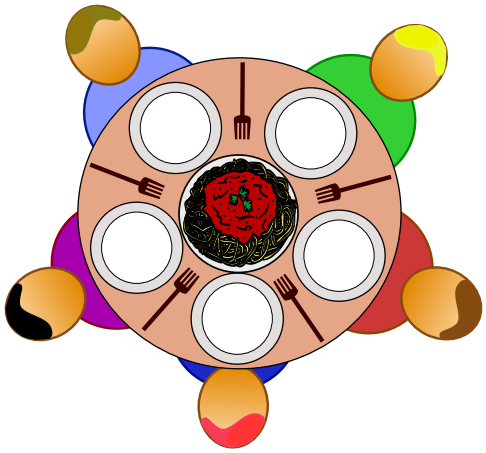

Auch beim Output ist select erlaubt

- In Go kann select verwendet werden, um eine von mehreren Sende- und Empfang-Operationen durchzuführen, bzw. darauf zu warten

Z.B.

```
...
 // Gleichzeitiges Lauschen und Schreiben an den Kanaelen c1 und c2
        select {
         case msg1 := <-c1:
            fmt.Println("received", msg1)
         case msg2 := <-c2:
            fmt.Println("received", msg2)
         case c2 <- "three":
            fmt.Println("send three on c2")
         case c1 <- "three":
            fmt.Println("send three on c1")
         }
```


Speisende Philosophen mit Kanälen



Speisende Philosophen mit Kanälen (2)

- pro Gabel: Ein Prozess, der via Kanal mit linken und rechten Philosophen verbunden ist
- Philosophenprozess: Versucht linke und rechte Gabel zu erhalten über Empfang von Nachrichten

Speisende Philosophen mit Kanälen (3)

forks : Feld von Kanälen über dem Typ Bool
x: lokale Variablen

Philosoph n	Philosoph i $<$ n	Gabel i
`loop forever`	`loop forever`	`loop forever`
(1) Denke;	(1) Denke;	(1) forks[i] $\Leftarrow$ True
(2) forks[i+1] $\Rightarrow$ x	(2) forks[i] $\Rightarrow$ x	(2) forks[i] $\Rightarrow$ x
(3) forks[i]$\Rightarrow$ x	(3) forks[i+1]$\Rightarrow$ x	`end loop`
(4) Esse;	(4) Esse;	
(5) forks[i+1] $\Leftarrow$ True	(5) forks[i] $\Leftarrow$ True	
(6) forks[i] $\Leftarrow$ True	(6) forks[i+1] $\Leftarrow$ True	
`end loop`	`end loop`	

Nicht Deadlockfrei

Speisende Philosophen in Go

```
package main
// Deadlock-freie Version:  10. Philosoph nimmt die Gabeln
// in umgekehrter Reihenfolge
import ("fmt";"strconv";"bufio";"os")

func fork (forks[](chan bool),i int) {
 for {
  forks[i] <- true
   <- forks[i]
  }
}
```


Speisende Philosophen in Go (2)

```
func philosopher (forks[](chan bool),i int) {
 for {
  fmt.Println("Philosoph: " + strconv.Itoa(i) + ": Denke...")
  if (i == 9) { <- forks[0]
                fmt.Println("Philosoph: "+strconv.Itoa(i)+": Habe rechte Gabel")
  } else      { <- forks[i]
                fmt.Println("Philosoph: "+strconv.Itoa(i)+": Habe linke Gabel") }
  if (i == 9) { <- forks[i]
                fmt.Println("Philosoph: "+strconv.Itoa(i)+": Habe linke Gabel")
  } else      { <- forks[i+1]
                fmt.Println("Philosoph: "+strconv.Itoa(i)+": Habe rechte Gabel") }
  fmt.Println("Philosoph: " + strconv.Itoa(i) + ": Esse...")
  if ( i == 9 ) { forks[0] <- true
                  forks[i] <- true
  } else        { forks[i] <- true
                  forks[i+1] <- true }
 }
}
```

Speisende Philosophen in Go (3)

```
func main() {
 //Gabel erstellen
 forks:=make([](chan bool),10)
 for i, := range forks{
  forks[i] = make(chan bool)
 }
 for i:=0; i < 10; i++ {
  go fork(forks,i)
 }
 //Philosophen erstellen
 for i:=0; i < 10; i++ {
  go philosopher(forks,i)
 }
 // Eingabe erwarten
 reader := bufio.NewReader(os.Stdin)
 reader.ReadString('\n')
}
```