LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prinzipien, Modelle und Algorithmen der Nebenläufigen Programmierung

Wintersemester 2019/20

Programmierprimitiven Teil III

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 28. November 2019

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:

```
loop forever
(1)  erzeuge e (vom Typ τ);
(2)  ch ⇐ e;
end loop
```

Verbraucher:

```
loop forever
(1)  ch ⇒ y;
(2)  verbrauche y;
end loop
```

Beispiel: Erzeuger / Verbraucher ohne Pufferung

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

Erzeuger:

```
loop forever
(1)   erzeuge e (vom Typ τ);
(2)   ch ⇐ e;
end loop
```

Verbraucher:

```
loop forever
(1)   ch ⇒ y;
(2)   verbrauche y;
end loop
```

Beispiel: Erzeuger / Verbraucher ohne Pufferung

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

Erzeuger:

```
loop forever
(1)   erzeuge e (vom Typ τ);
(2)   ch ⇐ e;
end loop
```

Verbraucher:

```
loop forever
(1)   ch ⇒ y;
(2)   verbrauche y;
end loop
```

Auswertung: Kommunikation nicht möglich, Erzeuger ist blockiert

Beispiel: Erzeuger / Verbraucher ohne Pufferung

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

Erzeuger:

```
loop forever
(1)  erzeuge e (vom Typ τ);
(2)  ch ⇐ e;
end loop
```

Verbraucher:

```
loop forever
(1)  ch ⇒ y;
(2)  verbrauche y;
end loop
```

Auswertung: Kommunikation möglich,
Kommunikation geschieht in einem Schritt

Beispiel: Erzeuger / Verbraucher ohne Pufferung

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

Erzeuger:

```
loop forever
(1)   erzeuge e (vom Typ τ);
(2)   ch ⇐ e;
end loop
```

Verbraucher:

```
loop forever
(1)   ch ⇒ y;
(2)   verbrauche e;
end loop
```

Auswertung: Beide Programmzeiger springen direkt weiter!

Beispiel: Erzeuger / Verbraucher ohne Pufferung

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

Erzeuger:

```
loop forever
(1)  erzeuge e (vom Typ τ);
(2)  ch ⇐ e;
end loop
```

Verbraucher:

```
loop forever
(1)  ch ⇒ y;
(2)  verbrauche ;
end loop
```

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?

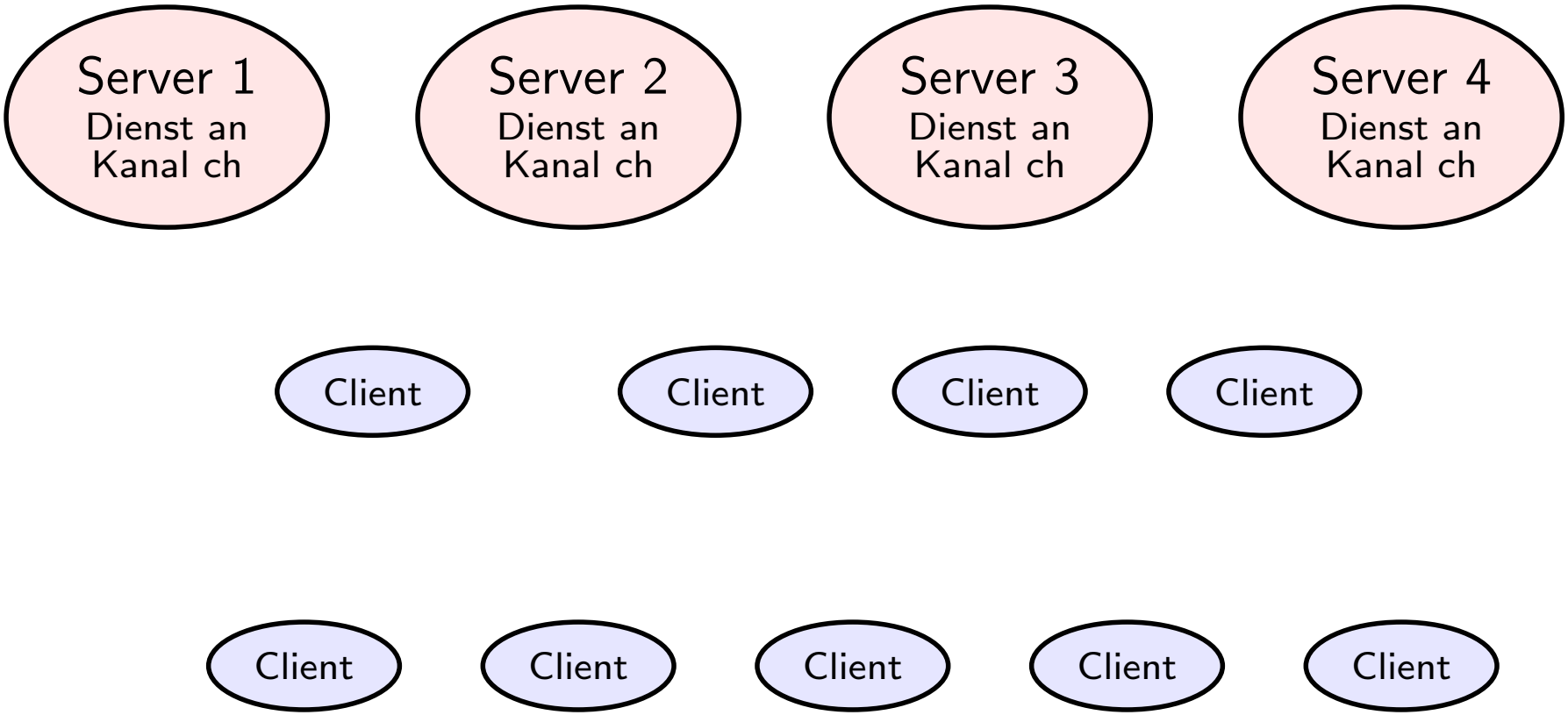
Analog

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

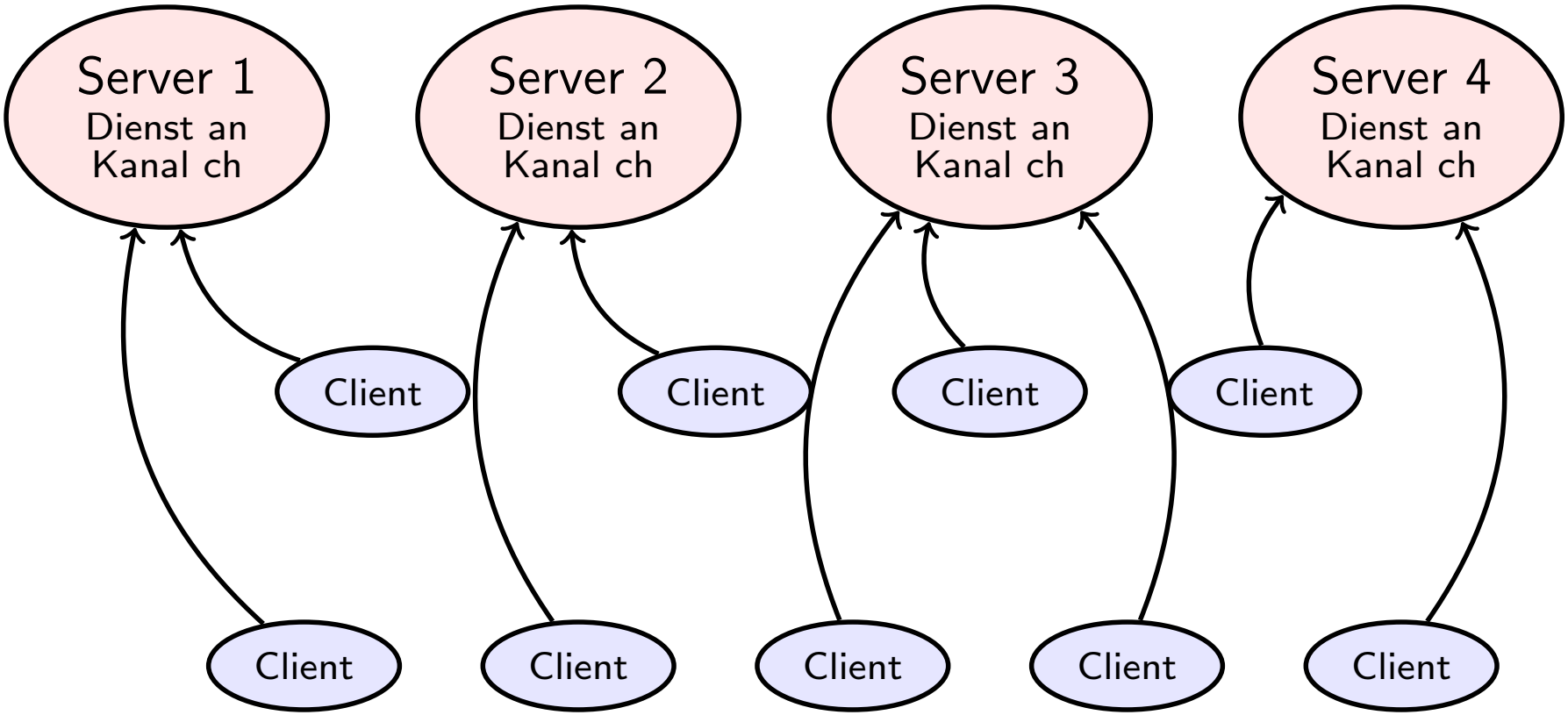
Wer druckt True aus?

Je nach Ablauf (quasi zufällig)

Kann sinnvoll sein . . .

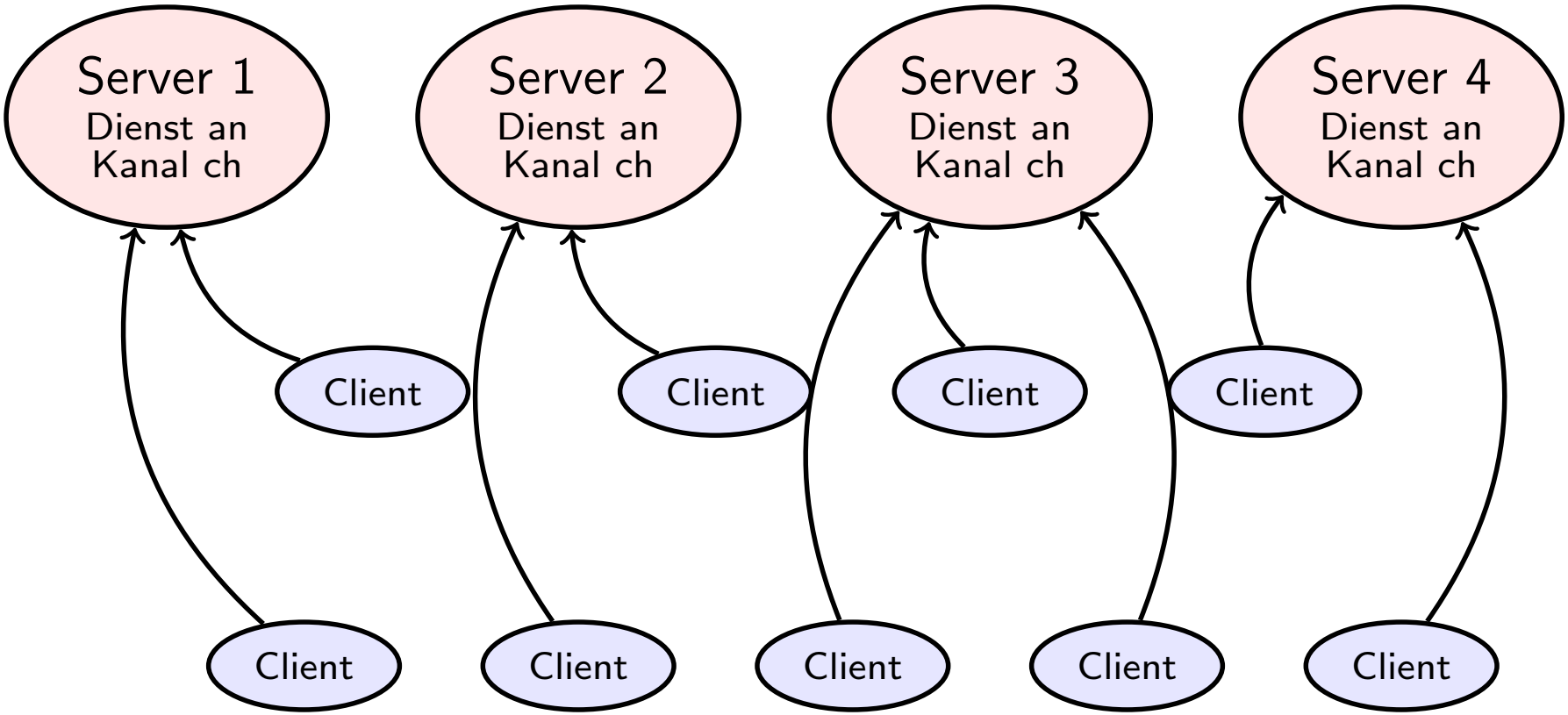


Kann sinnvoll sein . . .



"Lastverteilung automatisch"

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 keine 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
```

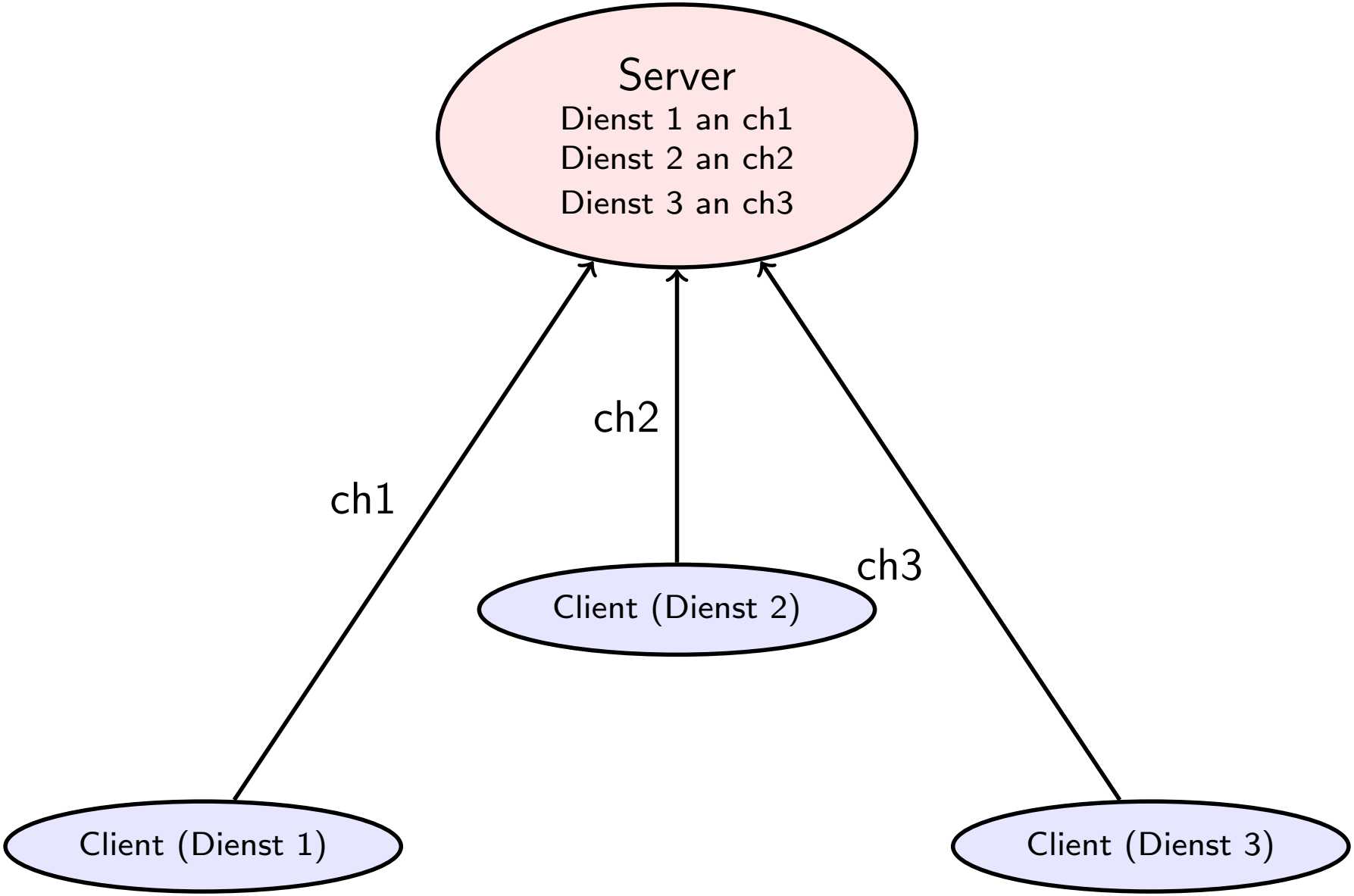
Selective Input (2)

Prozess 1	Prozess 2	Prozess 3	Prozess 4
`either`	ch1 $\Leftarrow$ e1	ch2 $\Leftarrow$ e1	ch3 $\Leftarrow$ e1
ch1 $\Rightarrow$ var1			
`or`			
ch2 $\Rightarrow$ var2			
`or`			
ch3 $\Rightarrow$ var3			

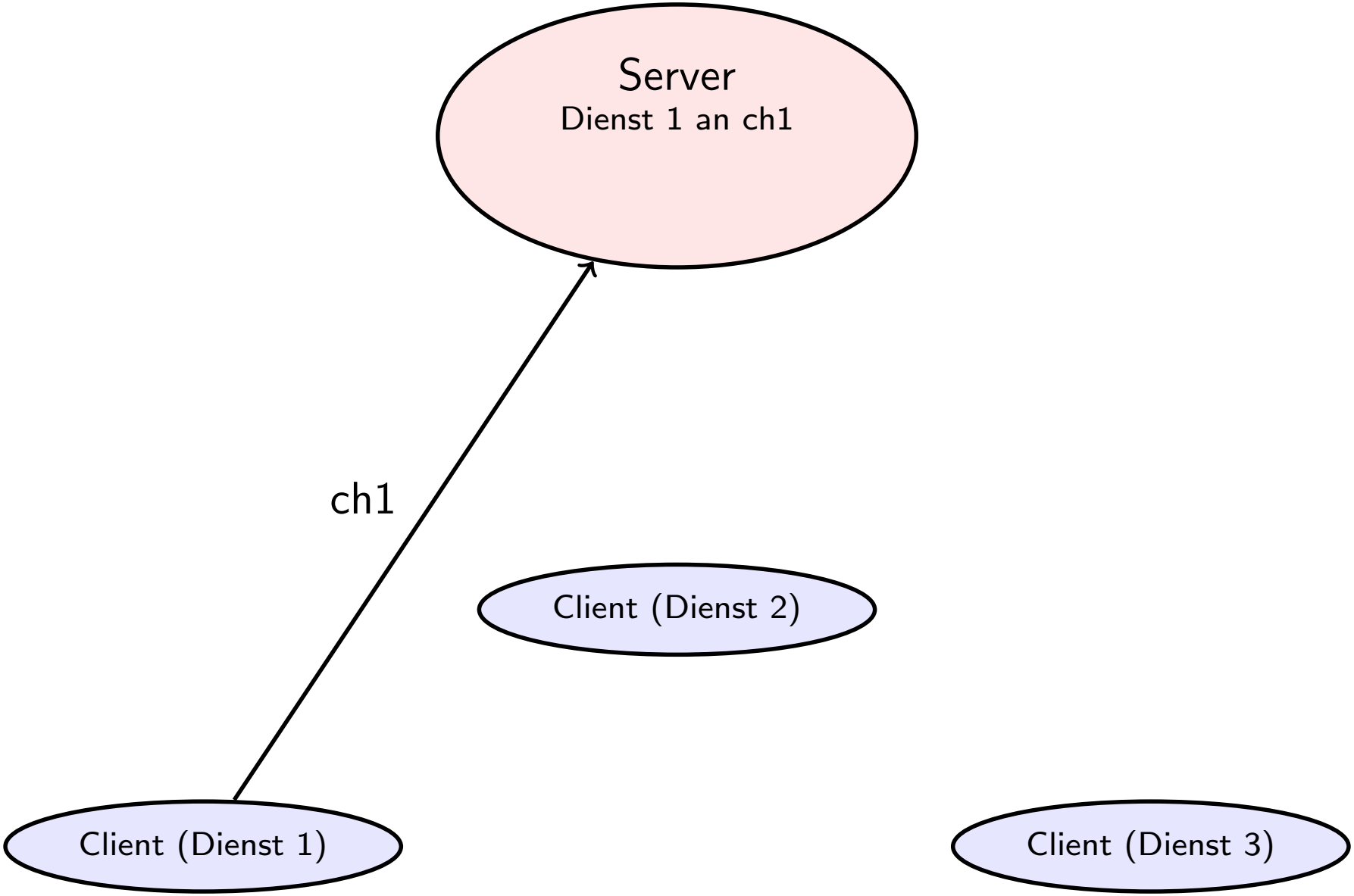
3 Möglichkeiten danach

Prozess 1	Prozess 2	Prozess 3	Prozess 4
(var1 := e1)		ch2 $\Leftarrow$ e2	ch3 $\Leftarrow$ e3
Prozess 1	Prozess 2	Prozess 3	Prozess 4
(var2 := e2)	ch1 $\Leftarrow$ e1		ch3 $\Leftarrow$ e3
Prozess 1	Prozess 2	Prozess 3	Prozess 4
(var3 := e3)	ch1 $\Leftarrow$ e1	ch2 $\Leftarrow$ e2	

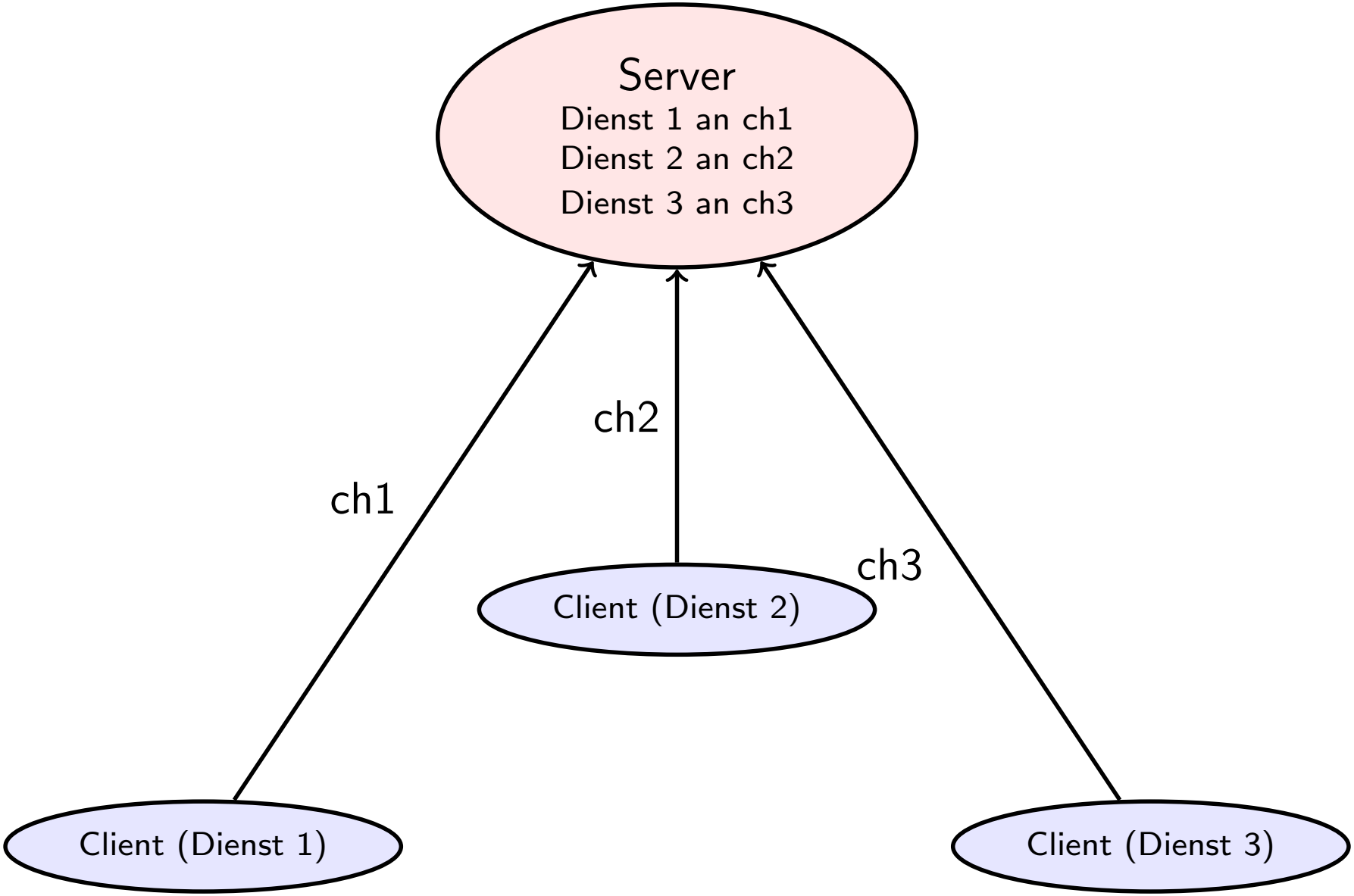
Z.B. nützlich bei ...



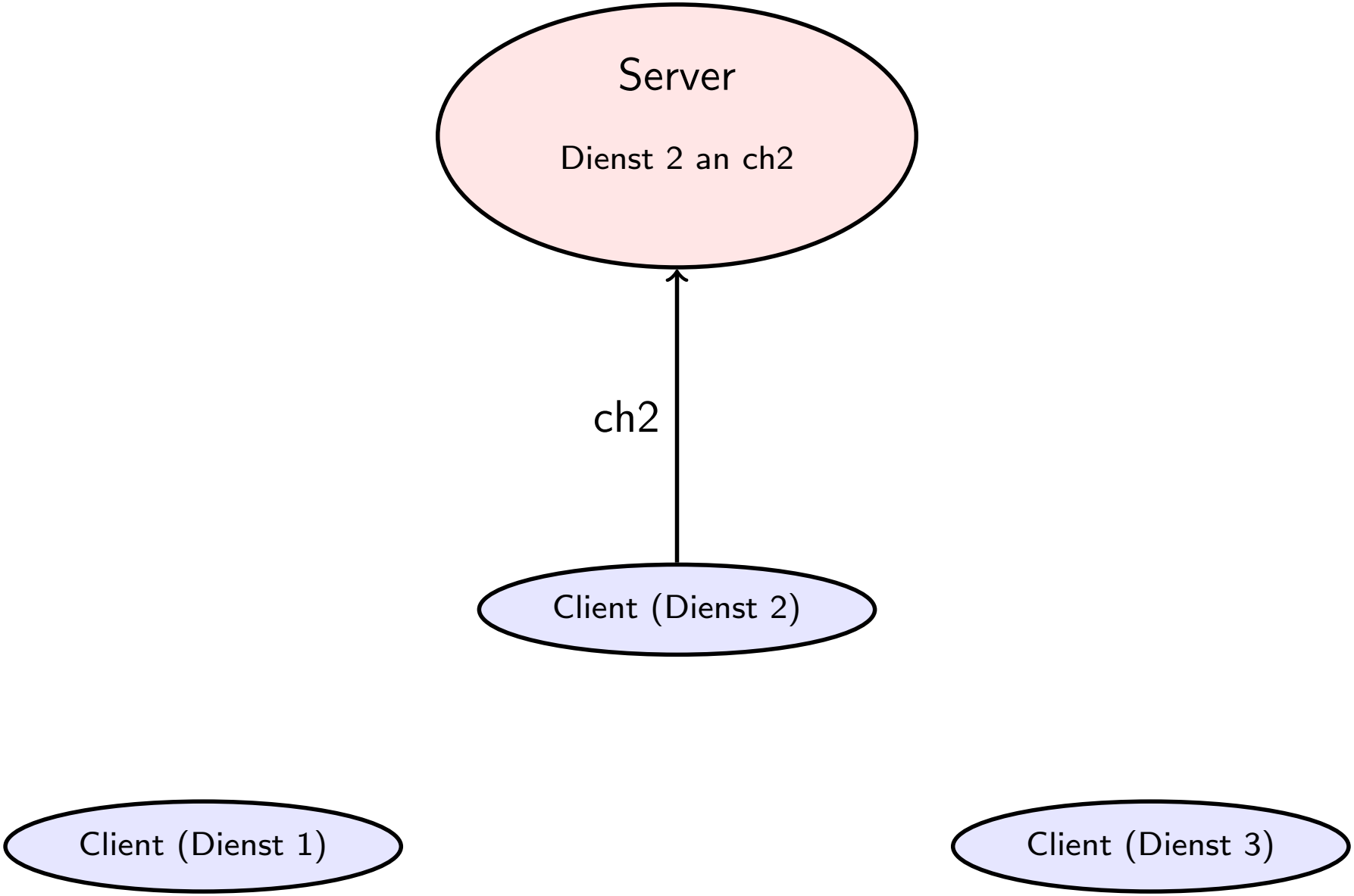
Z.B. nützlich bei . . .



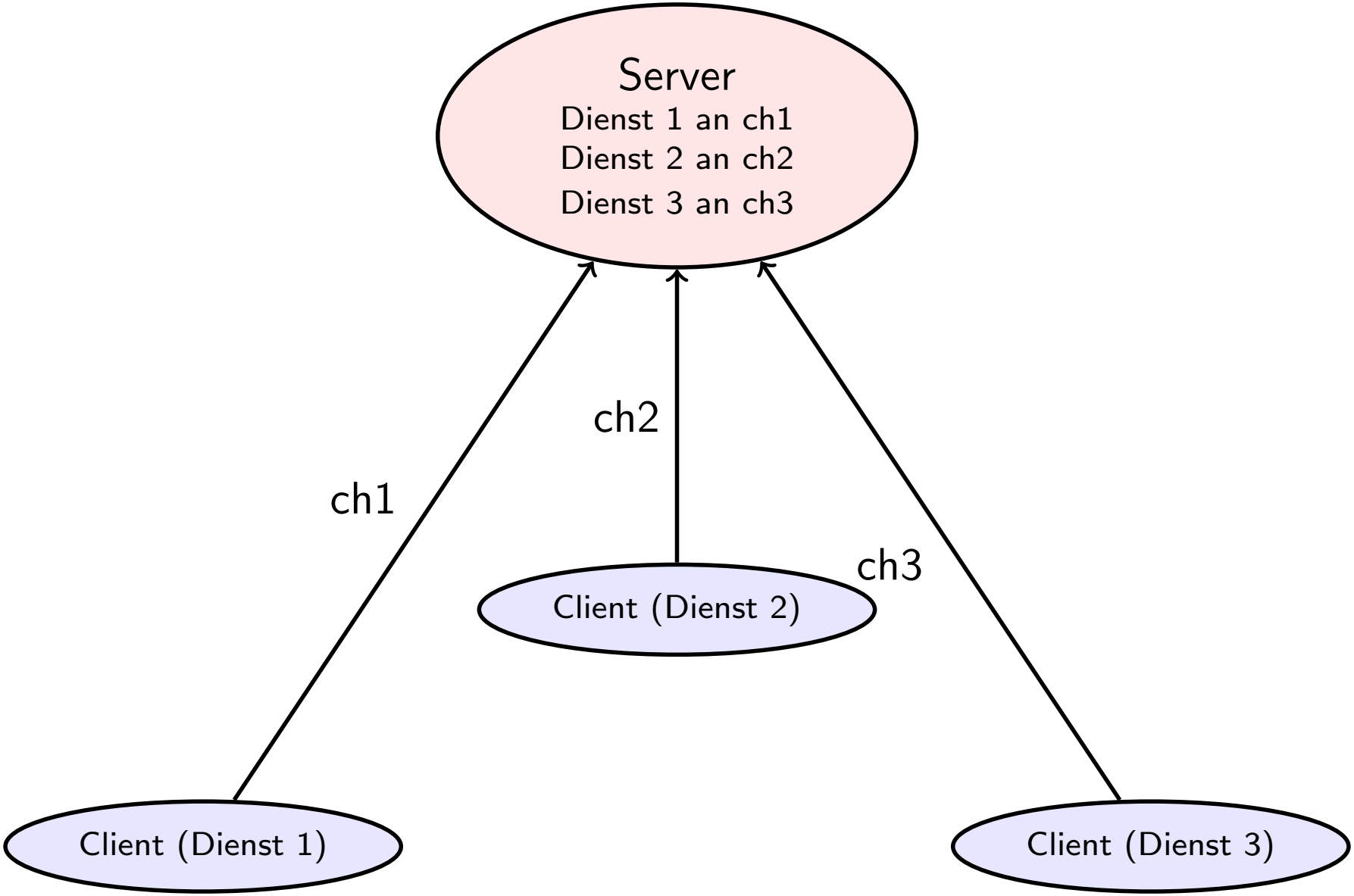
Z.B. nützlich bei ...



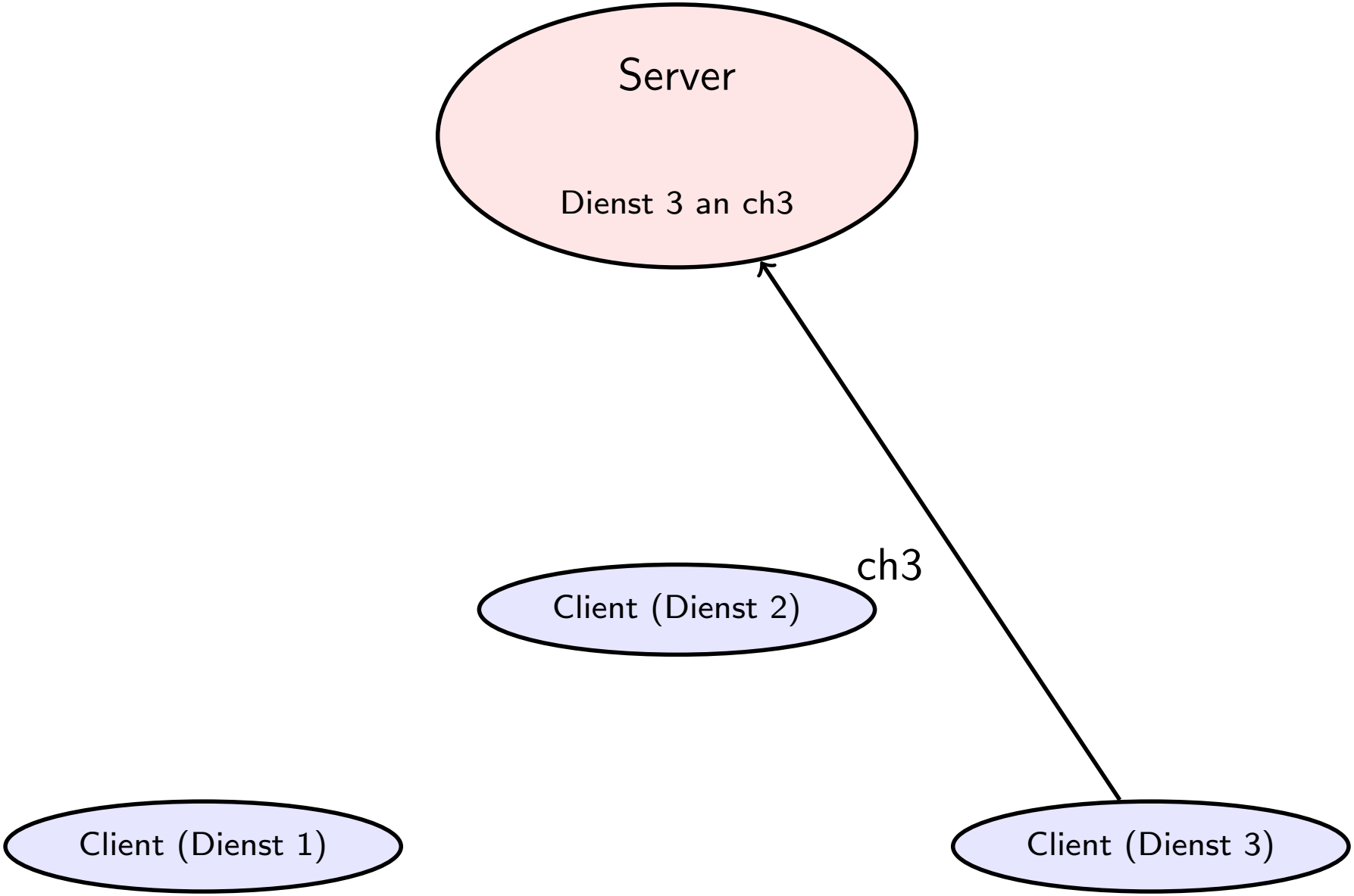
Z.B. nützlich bei . . .



Z.B. nützlich bei ...



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 := <- ch2:
    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)
   }}}
```

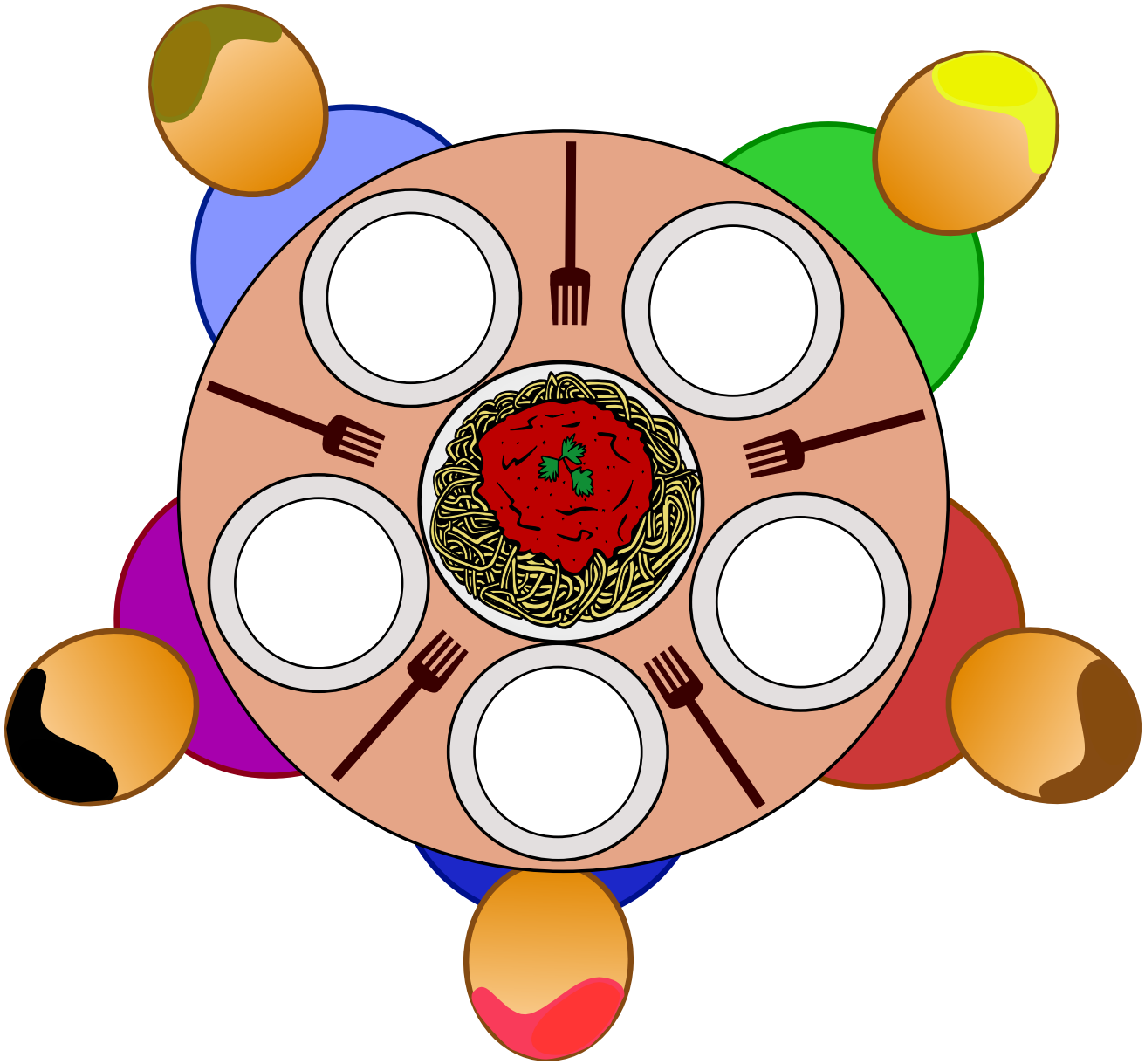
Auch beim Output ist select erlaubt

- 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: 1 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 i

```
loop forever
(1)  Denke;
(2)  forks[i] ⇒ x
(3)  forks[i+1]⇒ x
(4)  Esse;
(5)  forks[i] ⇐ True
(6)  forks[i+1] ⇐ True
end loop
```

Gabel i

```
loop forever
(1)  forks[i] ⇐ True
(2)  forks[i] ⇒ x
end loop
```

Speisende Philosophen mit Kanälen (3)

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

Philosoph i

```
loop forever
(1)  Denke;
(2)  forks[i] ⇒ x
(3)  forks[i+1]⇒ x
(4)  Esse;
(5)  forks[i] ⇐ True
(6)  forks[i+1] ⇐ True
end loop
```

Gabel i

```
loop forever
(1)  forks[i] ⇐ True
(2)  forks[i] ⇒ x
end loop
```

Nicht Deadlockfrei

Speisende Philosophen mit Kanälen (3)

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

Philosoph n

```
loop forever
(1)  Denke;
(2)  forks[i+1] ⇒ x
(3)  forks[i]⇒ x
(4)  Esse;
(5)  forks[i+1] ⇐ True
(6)  forks[i] ⇐ True
end loop
```

Philosoph i $<$ n

```
loop forever
(1)  Denke;
(2)  forks[i] ⇒ x
(3)  forks[i+1]⇒ x
(4)  Esse;
(5)  forks[i] ⇐ True
(6)  forks[i+1] ⇐ True
end loop
```

Gabel i

```
loop forever
(1)  forks[i] ⇐ True
(2)  forks[i] ⇒ x
end loop
```

Deadlockfrei, nicht Starvation-frei

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')
}
```

Tuple Spaces: Das Linda Modell

Nachteile von Kanälen

- Kanalname muss Sender und Empfänger bekannt sein
- Gerade bei Server/Client Architekturen schlecht: Server muss die Kanalnamen seiner Dienste exportieren
- Weiterer Nachteil: Nachrichten können nur zwischen aktiven Prozessen verschickt werden
- Im folgenden: Das Linda Modell (in mehrenen Sprachen implementiert, z.B. JavaSpaces, TSpaces (IBM),pSpaces)
- Gute Idee, aber nie richtig zum Trend geworden
- Koordinationssprache (für Nebenläufigkeit / Parallele Programmierung)

Tuple Space

(“Tupel 2“,False, 1.0, True, 50)

(20,30)

(“Tupel 1“,1.0, True)

(40)

(“Tupel 1“,1.0, True)

(1,1,1,1,1,1,1,1)

Tuple Space (2)

- Annahme: Es gibt einen Tuple Space: “Platz” für Tupel, alle Prozesse können darauf zugreifen
- Tupel sind getypt, d.h. jede Komponente hat einen festen Typ
- z.B. (“Tupel 1“:String,1.0:Float, True:Bool)
- Gleiche Tupel sind erlaubt
- Konvention oft: Erste Komponente muss ein String sein (Name/Beschreibung des Tupels)

Operationen auf dem Tuple Space

$\texttt{out}(N, v_1, \ldots, v_n)$:

- Einfügen eines Tupels in den Tuple Space
- N: Name (String)
- v_i: Programmvariable oder Wert
- Wenn v_i Programmvariable,
  dann wird der aktuelle Wert von v_i eingefügt

Operationen auf dem Tuple Space

$\texttt{out}(N, v_1, \ldots, v_n)$:

- Einfügen eines Tupels in den Tuple Space
- N: Name (String)
- v_i: Programmvariable oder Wert
- Wenn v_i Programmvariable,
  dann wird der aktuelle Wert von v_i eingefügt

(1) out("Tupel 1", 10, True)
(2) x := 50;
(3) b := True;
(4) out("Tupel 2", x, 100, b)

Operationen auf dem Tuple Space

$\texttt{out}(N, v_1, \ldots, v_n)$:

- Einfügen eines Tupels in den Tuple Space
- N: Name (String)
- v_i: Programmvariable oder Wert
- Wenn v_i Programmvariable,
  dann wird der aktuelle Wert von v_i eingefügt

```
(1)  out("Tupel 1", 10, True)
(2)  x := 50;
(3)  b := True;
(4)  out("Tupel 2", x, 100, b)
```

("Tupel 1", 10, True)

Operationen auf dem Tuple Space

$\texttt{out}(N, v_1, \ldots, v_n)$:

- Einfügen eines Tupels in den Tuple Space
- N: Name (String)
- v_i: Programmvariable oder Wert
- Wenn v_i Programmvariable,
  dann wird der aktuelle Wert von v_i eingefügt

```
(1)  out("Tupel 1", 10, True)
(2)  x := 50;
(3)  b := True;
(4)  out("Tupel 2", x, 100, b)
```

("Tupel 1", 10, True)

Operationen auf dem Tuple Space

$\texttt{out}(N, v_1, \ldots, v_n)$:

- Einfügen eines Tupels in den Tuple Space
- N: Name (String)
- v_i: Programmvariable oder Wert
- Wenn v_i Programmvariable,
  dann wird der aktuelle Wert von v_i eingefügt

(1) out(“Tupel 1“, 10, True)
(2) x := 50;
(3) b := True;
(4) out(“Tupel 2“, x, 100, b)

(“Tupel 1“, 10, True)

Operationen auf dem Tuple Space

$\texttt{out}(N, v_1, \ldots, v_n)$:

- Einfügen eines Tupels in den Tuple Space
- N: Name (String)
- v_i: Programmvariable oder Wert
- Wenn v_i Programmvariable,
  dann wird der aktuelle Wert von v_i eingefügt

(1) out(“Tupel 1“, 10, True)
(2) x := 50;
(3) b := True;
(4) out(“Tupel 2“, x, 100, b)

(“Tupel 1“, 10, True)

(“Tupel 2“, 50, 100, True)

Operationen auf dem Tuple Space (2)

$\texttt{in}(N, x_1, \ldots, x_n)$:

- Entfernen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- Im Tuple Space muss ein "Matching Tuple" vorhanden sein
- Matching Tuple = gleiche Länge, gleiche Typen, gleiche Werte
- Falls kein passendes Tuple vorhanden: Prozess blockiert, bis ein passendes Tupel eingefügt wird
- Ansonsten: Variablen werden durch das Entfernen an die Werte gebunden

Operationen auf dem Tuple Space (2)

$\texttt{in}(N, x_1, \ldots, x_n)$:

- Entfernen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- Im Tuple Space muss ein "Matching Tuple" vorhanden sein
- Matching Tuple = gleiche Länge, gleiche Typen, gleiche Werte
- Falls kein passendes Tuple vorhanden: Prozess blockiert, bis ein passendes Tupel eingefügt wird
- Ansonsten: Variablen werden durch das Entfernen an die Werte gebunden

```
(1)  in("Tupel 1", x, y)
(2)  print x;print y;
(3)  in("Tupel 1", x, y)
```

("Tupel 1", 10, True)

("Tupel 2", 50, 100, True)

Operationen auf dem Tuple Space (2)

$\texttt{in}(N, x_1, \ldots, x_n)$:

- Entfernen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- Im Tuple Space muss ein "Matching Tuple" vorhanden sein
- Matching Tuple = gleiche Länge, gleiche Typen, gleiche Werte
- Falls kein passendes Tuple vorhanden: Prozess blockiert, bis ein passendes Tupel eingefügt wird
- Ansonsten: Variablen werden durch das Entfernen an die Werte gebunden

```
(1)  in("Tupel 1", x, y)
(2)  print x;print y;
(3)  in("Tupel 1", x, y)
```

("Tupel 1", 10, True)

("Tupel 2", 50, 100, True)

Operationen auf dem Tuple Space (2)

$\texttt{in}(N, x_1, \ldots, x_n)$:

- Entfernen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- Im Tuple Space muss ein "Matching Tuple" vorhanden sein
- Matching Tuple = gleiche Länge, gleiche Typen, gleiche Werte
- Falls kein passendes Tuple vorhanden: Prozess blockiert, bis ein passendes Tupel eingefügt wird
- Ansonsten: Variablen werden durch das Entfernen an die Werte gebunden

(1) `in`("Tupel 1", x, y)
(2) print 10; print True;
(3) `in`("Tupel 1", x, y)

("Tupel 2", 50, 100, True)

Operationen auf dem Tuple Space (2)

$\texttt{in}(N, x_1, \ldots, x_n)$:

- Entfernen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- Im Tuple Space muss ein "Matching Tuple" vorhanden sein
- Matching Tuple = gleiche Länge, gleiche Typen, gleiche Werte
- Falls kein passendes Tuple vorhanden: Prozess blockiert, bis ein passendes Tupel eingefügt wird
- Ansonsten: Variablen werden durch das Entfernen an die Werte gebunden

(1) in("Tupel 1", x, y)
(2) print 10; print True;
(3) in("Tupel 1", x, y)

("Tupel 2", 50, 100, True)

Operationen auf dem Tuple Space (2)

$\texttt{in}(N, x_1, \ldots, x_n)$:

- Entfernen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- Im Tuple Space muss ein "Matching Tuple" vorhanden sein
- Matching Tuple = gleiche Länge, gleiche Typen, gleiche Werte
- Falls kein passendes Tuple vorhanden: Prozess blockiert, bis ein passendes Tupel eingefügt wird
- Ansonsten: Variablen werden durch das Entfernen an die Werte gebunden

(1) `in`("Tupel 1", x, y)
(2) print 10; print True;
(3) *blockiert*

("Tupel 2", 50, 100, True)

Operationen auf dem Tuple Space (3)

`read`$(N, x_1, \ldots, x_n)$:

- Lesen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- wie `in` nur das Tupel wird nicht entfernt

Operationen auf dem Tuple Space (3)

`read`$(N, x_1, \ldots, x_n)$:

- Lesen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- wie `in` nur das Tupel wird nicht entfernt

```
(1)  read("Tupel 1", x, y);
(2)  print x;print y;
(3)  read("Tupel 1", x, y);
(4)  read("Tupel 3", x, y);
```

("Tupel 1", 10, True)

("Tupel 2", 50, 100, True)

Operationen auf dem Tuple Space (3)

`read`$(N, x_1, \ldots, x_n)$:

- Lesen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- wie `in` nur das Tupel wird nicht entfernt

```
(1)  read("Tupel 1", x, y);
(2)  print x;print y;
(3)  read("Tupel 1", x, y);
(4)  read("Tupel 3", x, y);
```

("Tupel 1", 10, True)

("Tupel 2", 50, 100, True)

Operationen auf dem Tuple Space (3)

`read`$(N, x_1, \ldots, x_n)$:

- Lesen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- wie `in` nur das Tupel wird nicht entfernt

(1) `read`(“Tupel 1“, x, y);
(2) print 10; print True;
(3) `read`(“Tupel 1“, x, y);
(4) `read`(“Tupel 3“, x, y);

(“Tupel 1“, 10, True)

(“Tupel 2“, 50, 100, True)

Operationen auf dem Tuple Space (3)

`read`$(N, x_1, \ldots, x_n)$:

- Lesen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- wie `in` nur das Tupel wird nicht entfernt

(1) `read`(“Tupel 1“, x, y);
(2) print 10; print True;
(3) `read`(“Tupel 1“, x, y);
(4) `read`(“Tupel 3“, x, y);

(“Tupel 1“, 10, True)

(“Tupel 2“, 50, 100, True)

Operationen auf dem Tuple Space (3)

`read`$(N, x_1, \ldots, x_n)$:

- Lesen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- wie `in` nur das Tupel wird nicht entfernt

```
(1)  read("Tupel 1", x, y);
(2)  print 10; print True;
(3)  read("Tupel 1", x, y);
(4)  read("Tupel 3", x, y);
```

("Tupel 1", 10, True)

("Tupel 2", 50, 100, True)

Operationen auf dem Tuple Space (3)

`read`$(N, x_1, \ldots, x_n)$:

- Lesen eines Tupels aus dem Tuple Space
- N: Name (String), x_i: Programmvariable
- wie `in` nur das Tupel wird nicht entfernt

(1) `read`(“Tupel 1“, x, y);
(2) print 10; print True;
(3) `read`(“Tupel 1“, x, y);
blockiert

(“Tupel 1“, 10, True)

(“Tupel 2“, 50, 100, True)

Operationen auf dem Tuple Space (4)

Manchmal noch weitere Operationen:

- `inp`: Wie `in`, nur dass beim Fehlversuch nicht blockiert wird, sondern eine Exception auftritt
- `readp`: Wie `read`, , nur dass beim Fehlversuch nicht blockiert wird, sondern eine Exception auftritt
- `eval`: Tupelkomponenten werden durch neue Prozesse nebenläufig berechnet, danach wie `out`

Nichtdeterminismus

- Gibt es mehrere passende Tupel zu ein `in`- oder `read`-Operation, wird irgendeiner gewählt.
- Gibt es mehrere wartende `in`- oder `read`-Operationen und ein passendes Tupel wird eingefügt, so wird irgendein wartender Prozess entblockiert

Mutual-Exclusion mit Tuple Spaces

Initial: Tupel (“MUTEX“) im Tuple Space

Prozess i:

```
loop forever
(1)   Restlicher Code;
(2)   in(“MUTEX“);
(3)   Kritischer Abschnitt;
(4)   out(“MUTEX“);
end loop
```

Beispiel

(“MUTEX“)

Prozess 1:

```
loop forever
(1)  Restlicher Code;
(2)  in(“MUTEX“);
(3)  Kritischer Abschnitt;
(4)  out(“MUTEX“);
end loop
```

Prozess 2:

```
loop forever
(1)  Restlicher Code;
(2)  in(“MUTEX“);
(3)  Kritischer Abschnitt;
(4)  out(“MUTEX“);
end loop
```

Beispiel

(“MUTEX“)

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in(“MUTEX“);
(3)   Kritischer Abschnitt;
(4)   out(“MUTEX“);
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in(“MUTEX“);
(3)   Kritischer Abschnitt;
(4)   out(“MUTEX“);
end loop
```

Beispiel

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Beispiel

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Beispiel

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Beispiel

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Beispiel

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Beispiel

(“MUTEX“)

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in(“MUTEX“);
(3)   Kritischer Abschnitt;
(4)   out(“MUTEX“);
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in(“MUTEX“);
(3)   Kritischer Abschnitt;
(4)   out(“MUTEX“);
end loop
```

Beispiel

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Beispiel

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Beispiel

Prozess 1:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

Prozess 2:

```
loop forever
(1)   Restlicher Code;
(2)   in("MUTEX");
(3)   Kritischer Abschnitt;
(4)   out("MUTEX");
end loop
```

- Erfüllt wechselseitigen Ausschluss und ist Deadlock-frei
- Bei 2 Prozessen: Starvation-frei, bei mehr als 2 nicht

Semaphore mit Tuple Spaces

Initial: k-Tupel (“SEM“) im Tuple Space

```
wait(){
  in(“SEM“);
}

signal(){
  out(“SEM“);
}
```

Semaphore mit Tuple Spaces (2)

("SEM") ("SEM") ("SEM")

("SEM") ("SEM") ("SEM")

"S.V = 6" "S.M = $\emptyset$"

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

("SEM") ("SEM") ("SEM")

("SEM") ("SEM")

"S.V = 5" "S.M = $\emptyset$"

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

(“SEM“) (“SEM“)

(“SEM“) (“SEM“)

“S.V = 4” “S.M = $\emptyset$”

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

(“SEM“) (“SEM“)

(“SEM“)

“S.V = 3” “S.M = $\emptyset$”

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

("SEM") ("SEM") ("SEM")

("SEM")

"S.V = 4" "S.M = $\emptyset$"

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

("SEM")

("SEM")

"S.V = 2" "S.M = $\emptyset$"

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

(“SEM“)

“S.V $= 1$” “S.M $= \emptyset$”

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

"S.V = 0" "S.M = ∅"

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

"S.V = 0" "S.M = $\{P\}$"

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

blockiert

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

"S.V = 0" "S.M = ∅"

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Semaphore mit Tuple Spaces (2)

("SEM")

"S.V = 1" "S.M = $\emptyset$"

Prozess P:

```
wait();
wait();
wait();
signal();
wait();
wait();
wait();
wait();
wait();
```

Prozess Q:

```
signal();
signal();
```

Atomares Update von Tupeln

```
(1)   in("Tupel",x);
(2)   x := f(x);
(3)   out("Tupel",x);
```

- Tupel ist nach Zeile (1) nicht mehr im Tupel Space
- Tupel erscheint erst wieder nach Zeile (3) im Tupel Space
- Aktion ist atomar, da kein Prozess während des Updates auf Tupel zugreifen kann

Synchrone Kommunikation

- Simulation von synchronen Kanälen.
- ch $\Leftarrow$ e:
  `in(“ch1“,“get“);`
  `out(“ch2“,e);`
  `in(“ch3“,“got“);`
- ch $\Rightarrow$ x:
  `out(“ch1“,“get“);`
  `in(“ch2“,x);`
  `out(“ch3“,“got“);`

Synchrone Kommunikation

- Simulation von synchronen Kanälen.
- ch $\Leftarrow$ e:
  `in`(“ch1“,“get“);
  `out`(“ch2“,e);
  `in`(“ch3“,“got“);
- ch $\Rightarrow$ x:
  `out`(“ch1“,“get“);
  `in`(“ch2“,x);
  `out`(“ch3“,“got“);

Falsch: geht so nicht: da `in` nur Variablen erhalten darf!

Erweiterung der in-Operation

- bisher: $\texttt{in}(N, x_1, \ldots, x_n)$: x_i formale Parameter (Variablen die durch die `in`-Operation gebunden werden)
- jetzt auch erlaubt: x_i aktuelle Parametervariablen, wir schreiben diese als x=.
- Semantik: Wert der Variablen x muss die Tupelkomponente matchen.

Erweiterung der in-Operation

- bisher: $\texttt{in}(N, x_1, \ldots, x_n)$: x_i formale Parameter (Variablen die durch die `in`-Operation gebunden werden)
- jetzt auch erlaubt: x_i aktuelle Parametervariablen, wir schreiben diese als x=.
- Semantik: Wert der Variablen x muss die Tupelkomponente matchen.

Beispiel:

x:=10; in(“Tupel 1“, x=);	x:=10; in(“Tupel 1“, x);
matcht nur Tupel der Form (“Tupel 1“, 10)	matcht alle Tupel der Form (“Tupel 1“, z:Int) wobei z eine Zahl
lässt x unverändert	überschreibt Wert von x durch z

Rückblick: Tuple Spaces

- Tuple Space: Raum von Tupeln, Tupel haben getypte Komponenten, erste Komponente = Name des Tupels
- `out`(Name, $v_1, \ldots, v_n$): Füge Tupel ein, wobei v_i Werte bzw. Programmvariablen sind, eingefügt werden dann die aktuellen Werte der Variablen.
- `in`(Name, $x_1, \ldots, x_n$): Nehme "matching Tuple" aus dem Tuple Space heraus, und binde die Werte an die Variablen $x_1, \ldots, x_n$. Blockiere, wenn es kein matching Tuple gibt
- `read`(Name, $x_1, \ldots, x_n$): Wie `in`, aber das Tuple wird im Tuple Space belassen.
- Erweitertes `in` und `read`: Als Parameter ist auch $x =$ zugelassen, Semantik: der Wert von x muss mit dem Tuple im TupleSpace übereinstimmen.

Vorteile der Erweiterung

- Man kann auf mehreren Komponenten matchen
- Programmierung kann vereinheitlicht werden

Vorteile der Erweiterung

- Man kann auf mehreren Komponenten matchen
- Programmierung kann vereinheitlicht werden

Z.B. Server 1 und Server 2 bieten verschiedene Dienste an

Ohne Erweiterung: Dienst im Namen fest kodiert

Server 1:
`in`(N + “Dienst1“,...); . . .

Server 2:
`in`(N + “Dienst2“,...); . . .

Vorteile der Erweiterung

- Man kann auf mehreren Komponenten matchen
- Programmierung kann vereinheitlicht werden

Z.B. Server 1 und Server 2 bieten verschiedene Dienste an

Ohne Erweiterung: Dienst im Namen fest kodiert

Server 1:
`in`(N + "Dienst1",...); ...

Server 2:
`in`(N + "Dienst2",...); ...

Mit Erweiterung

Server 1:
dienst := "Dienst1";
`in`(N,dienst=,...); ...

Server 2:
dienst := "Dienst2";
`in`(N,dienst=,...); ...

Synchrone Kommunikation

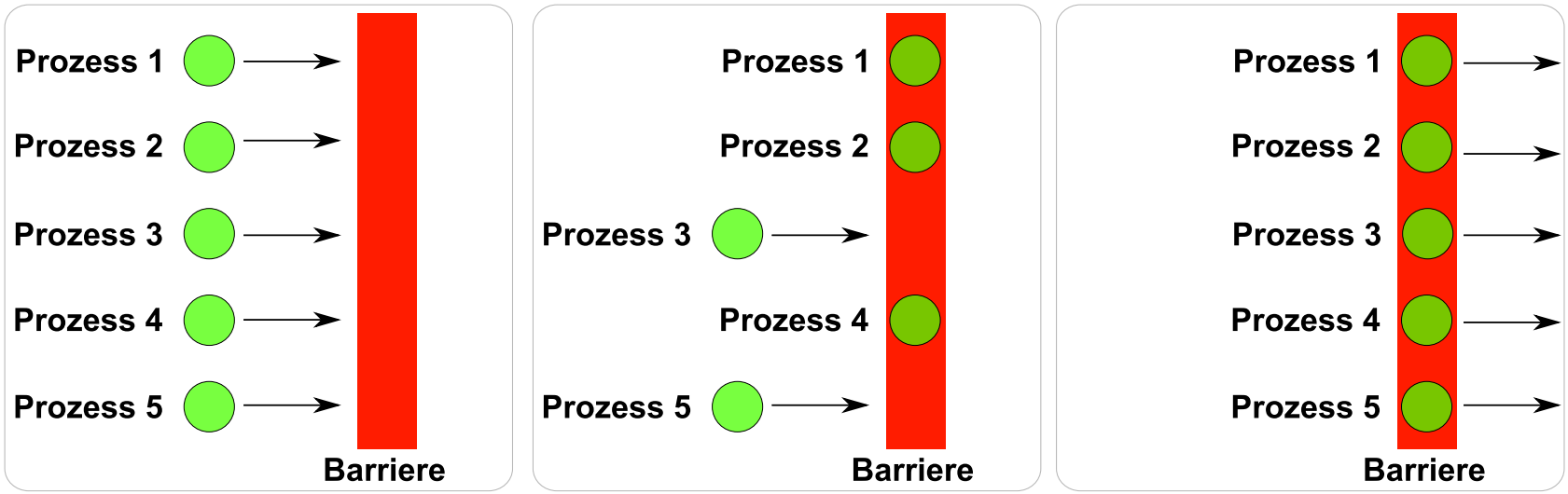
- Simulation von synchronen Kanälen
- ch $\Leftarrow$ e:

```
get := “get“;
got := “got“;
in(“ch1“,get=);
out(“ch2“,e);
in(“ch3“,got=);
```

- ch $\Rightarrow$ x:

```
out(“ch1“,“get“);
in(“ch2“,x);
out(“ch3“,“got“);
```

Barrier


Prozess 1
Prozess 2
Prozess 3
Prozess 4
Prozess 5
Barriere
Prozess 1
Prozess 2
Prozess 3
Prozess 4
Prozess 5
Barriere
Prozess 1
Prozess 2
Prozess 3
Prozess 4
Prozess 5
Barriere

Barrier

Initial: Ein Tupel (“ankommen“,N);

```
barrier (){
//Ankommen
  in(“ankommen“,x);
  x := x-1;
  if x > 0
    {out(“ankommen“,x);}
  else
    {out(“verlassen“,N);} //Letzter Ankommer
  in(“verlassen“,y);
  y := y-1;
  if y > 0
    {out(“verlassen“,y);}
  else
    {out(“ankommen“,N);} //Letzter Verlasser
}
```

Prozess i:

```
loop forever
  Code vor dem Barrier
  barrier()
  Code nach dem Barrier
end loop
```

Speisende Philosophen

```
philosoph(i) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := i+1 mod N;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

```
initialize() {
  for i=1 to N do
    out("Gabel",i);
    if i ≠ N then out("Raum");
  for i=1 to N do
    Erzeuge Prozess philosoph(i);
}
```

Deadlockfrei und Starvationfrei

Speisende Philosophen (2)

```
initialize() {
   for i=1 to 5 do
      out("Gabel",i);
      if i ≠ 5 then out("Raum");
   for i=1 to 5 do
      Erzeuge Prozess philosoph(i);
}
```

("Raum") ("Raum") ("Gabel",1) ("Gabel",2) ("Gabel",3)

("Raum") ("Raum") ("Gabel",4) ("Gabel",5)

Speisende Philosophen (3)

```
philosoph(3) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := 4;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

("Raum") ("Raum") ("Gabel",1) ("Gabel",2) ("Gabel",3)

("Raum") ("Raum") ("Gabel",4) ("Gabel",5)

Speisende Philosophen (3)

```
philosoph(3) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := 4;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

("Raum") ("Gabel",1) ("Gabel",2) ("Gabel",3)

("Raum") ("Raum") ("Gabel",4) ("Gabel",5)

Speisende Philosophen (3)

```
philosoph(3) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := 4;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

("Raum") ("Gabel",1) ("Gabel",2)

("Raum") ("Raum") ("Gabel",4) ("Gabel",5)

Speisende Philosophen (3)

```
philosoph(3) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := 4;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

("Raum") ("Gabel",1) ("Gabel",2)

("Raum") ("Raum") ("Gabel",4) ("Gabel",5)

Speisende Philosophen (3)

```
philosoph(3) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := 4;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

("Raum") ("Gabel",1) ("Gabel",2)

("Raum") ("Raum") ("Gabel",5)

Speisende Philosophen (3)

```
philosoph(3) {
  loop forever
    Denke;
    in(“Raum“);
    in(“Gabel“,i=);
    j := 4;
    in(“Gabel“,j=);
    Esse;
    out(“Gabel“,i);
    out(“Gabel“,j);
    out(“Raum“);
  end loop
}
```

(“Raum“) (“Gabel“,1) (“Gabel“,2)

(“Raum“) (“Raum“) (“Gabel“,5)

Speisende Philosophen (3)

```
philosoph(3) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := 4;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

("Raum") ("Gabel",1) ("Gabel",2) ("Gabel",3)

("Raum") ("Raum") ("Gabel",5)

Speisende Philosophen (3)

```
philosoph(3) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := 4;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

("Raum") ("Gabel",1) ("Gabel",2) ("Gabel",3)

("Raum") ("Raum") ("Gabel",4) ("Gabel",5)

Speisende Philosophen (3)

```
philosoph(3) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := 4;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

("Raum") ("Raum") ("Gabel",1) ("Gabel",2) ("Gabel",3)

("Raum") ("Raum") ("Gabel",4) ("Gabel",5)

Erzeuger / Verbraucher

Initial: N Tupel (“notFull“) im Tuple-Space

```
produce(Element e) {
    in(“notFull“);
    out(“buffer“,e);
    out(“notEmpty“);
}
```

```
consume() {
    in(“notEmpty“);
    in(“buffer“,x);
    out(“notFull“);
    return(x);
}
```

Code des Erzeugers:

```
loop forever
(1)   erzeuge e
(2)   produce(e)
end loop
```

Code des Verbrauchers:

```
loop forever
(1)   e := consume();
(2)   verbrauche e;
end loop
```

Erzeuger / Verbraucher

Initial: N Tupel (“notFull“) im Tuple-Space

```
produce(Element e) {
    in(“notFull“);
    out(“buffer“,e);
    out(“notEmpty“);
}
```

```
consume() {
    in(“notEmpty“);
    in(“buffer“,x);
    out(“notFull“);
    return(x);
}
```

Code des Erzeugers:

```
loop forever
(1)   erzeuge e
(2)   produce(e)
end loop
```

Code des Verbrauchers:

```
loop forever
(1)   e := consume();
(2)   verbrauche e;
end loop
```

Keine FIFO-Reihenfolge!

Erzeuger / Verbraucher: FIFO

Initial: N Tupel (“notFull“) im Tuple-Space
(“FIFO-Queue“,“tail“,0)
(“FIFO-Queue“,“head“,0)

```
produce(Element e) {
    in(“notFull“);
    in(“FIFO-Queue“,“tail“, tindex);
    tindex’ = tindex +1;
    out(“FIFO-Queue“, tindex’, e);
    out(“FIFO-Queue“, “tail“, tindex’);
    out(“notEmpty“);
}
```

```
consume() {
    in(“notEmpty“);
    in(“FIFO-Queue“, “head“, hindex);
    hindex’ = hindex+1;
    in(“FIFO-Queue“, hindex=, x);
    out(“FIFO-Queue“, “head“, hindex’);
    out(“notFull“);
    return(x);
}
```

Readers & Writers

Reader:
startRead();
Kritischer Abschnitt;
stopRead();

Writer:
startWrite();
Kritischer Abschnitt;
stopWrite();

- Ein Tupel wird benutzt der Form:
  (“RW“,Anzahl Reader, Anzahl Writer)
- Initial: Ein Tupel (“RW“,0,0)

Readers & Writers (2)

```
startRead() {
  in("RW",countR,0);
  countR' := countR + 1;
  out("RW",countR',0);
}

startWrite() {
  in("RW",0,0);
  out("RW",0,1);
}
```

```
stopRead() {
  in("RW",countR,0);
  countR' := countR - 1;
  out("RW",countR',0);
}

stopWrite() {
  in("RW",0,1);
  out("RW",0,0);
}
```

Readers & Writers: Priorität für Writers

Initial: Ein Tupel (“RW“,0,0,0)

```
startRead() {
  in(“RW“,countR,0,0);
  countR’ := countR + 1;
  out(“RW“,countR’,0,0);
}
```

```
stopRead() {
  in(“RW“,countR,0,waitingW);
  countR’ := countR - 1;
  out(“RW“,countR’,0,waitingW);
}
```

```
startWrite() {
  in(“RW“,countR,countW,waitingW);
  w’ := waitingW+1;
  out(“RW“,countR,countW,w’);
  in(“RW“,0,0,waitingW);
  w’ := waitingW-1;
  out(“RW“,0,1,w’);
}
```

```
stopWrite() {
  in(“RW“,0,1,waitingW);
  out(“RW“,0,0,waitingW);
}
```