

Programmierprimitiven: Tuple-Spaces

Prof. Dr. David Sabel

LFE Theoretische Informatik



Übersicht

- 1 Einleitung
- 2 Tuple Space
- 3 pSpaces und goSpace
- 4 Beispiele

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 mehreren Sprachen implementiert, z.B. JavaSpaces, TSpaces (IBM), pSpaces)
- Gute Idee, aber nie richtig zum Trend geworden
- Koordinationssprache (für Nebenläufigkeit / Parallele Programmierung)
- Wir betrachten für die Beispiele die pSpaces-Implementierung in Go (goSpace)

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

$\text{out}(N, v_1, \dots, v_n)$:

- 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

$\text{out}(N, v_1, \dots, v_n)$:

- 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) $\text{out}(\text{"Tupel 1"}, 10, \text{True})$
- (2) $x := 50;$
- (3) $b := \text{True};$
- (4) $\text{out}(\text{"Tupel 2"}, x, 100, b)$



Operationen auf dem Tuple Space

$\text{out}(N, v_1, \dots, v_n)$:

- 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) $\text{out}(\text{"Tupel 1"}, 10, \text{True})$

(2) $x := 50;$

(3) $b := \text{True};$

(4) $\text{out}(\text{"Tupel 2"}, x, 100, b)$

$(\text{"Tupel 1"}, 10, \text{True})$

Operationen auf dem Tuple Space

$\text{out}(N, v_1, \dots, v_n)$:

- 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

$\text{out}(N, v_1, \dots, v_n)$:

- 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

$\text{out}(N, v_1, \dots, v_n)$:

- 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)

$\text{in}(N, x_1, \dots, x_n)$:

- 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 Tuple eingefügt wird
- Ansonsten: Variablen werden durch das Entfernen an die Werte gebunden

Operationen auf dem Tuple Space (2)

`in(N, x1, ..., xn):`

- Entfernen eines Tupels aus dem Tuple Space
- `N`: Name (String), `xi`: 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 Tuple 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)

`in(N, x1, ..., xn):`

- Entfernen eines Tupels aus dem Tuple Space
- `N`: Name (String), `xi`: 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 Tuple 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)

`in(N, x1, ..., xn):`

- Entfernen eines Tupels aus dem Tuple Space
- `N`: Name (String), `xi`: 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 Tuple 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)

`in(N, x1, ..., xn):`

- Entfernen eines Tupels aus dem Tuple Space
- `N`: Name (String), `xi`: 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 Tuple 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)

$\text{in}(N, x_1, \dots, x_n)$:

- 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 Tuple eingefügt wird
- Ansonsten: Variablen werden durch das Entfernen an die Werte gebunden

- (1) $\text{in}(\text{“Tupel 1“}, x, y)$
- (2) `print 10; print True;`
- (3) *blockiert*

(“Tupel 2“, 50, 100, True)

Operationen auf dem Tuple Space (3)

`read(N, x_1, \dots, 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, x1, ..., xn):`

- 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, x1, ..., xn):`

- 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, x1, ..., xn):`

- 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, x1, ..., xn):`

- 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, x1, ..., xn):`

- 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, x1, ..., xn):`

- 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`

- 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

- Homepage: <https://github.com/pSpaces>
- Projekt der Technischen Universität Dänemark und der Universität Camerino
- Enthält Bibliotheken für Java, C#, Go, JavaScript und Swift
- Wir zeigen kurz die Verwendung in Go

Go und pSpaces = goSpace

Import der Bibliothek:

```
import ("github.com/pspaces/gospace")
```

Neuen Tuple Space erstellen mit NewSpace, z.B.

```
meinTupleSpace := gospace.NewSpace("Name des Tuple Space")
```

Tupel sind über den Typ `Tuple` verfügbar.

```
var tuple Tuple = CreateTuple("Milch", 1)
```

Zugriff auf Komponenten mit `GetFieldAt`:

```
tuple.GetFieldAt(i)
```

Typen der Werte müssen gecasted werden:

```
var mtuple Tuple = CreateTuple("Milch", 1)
var was string
was = (mtuple.GetFieldAt(0)).(string)
var wieviel int
wieviel = (mtuple.GetFieldAt(1)).(int)
```

Operationen:

`Put(x_1, x_2, \dots, x_n)` entspricht `out(x_1, \dots, x_n)`

`Get(x_1, x_2, \dots, x_n)` entspricht `in(x_1, \dots, x_n)`

`Query(x_1, x_2, \dots, x_n)` entspricht `read(x_1, \dots, x_n)`

Nicht-blockierende Varianten: `GetP`, `QueryP`

Die Rückgabe der Operation ist immer ein Paar (`Tuple`, `error`)

goSpace: Beispiel zu Put

Put erzeugt direkt ein Tupel aus den Argumenten und fügt es ein

```
// Tuple Space erzeugen
meinTupelSpace := NewSpace("MeinTupelSpace")
var tuple Tuple = CreateTuple("Milch", 1)
// Tupel einfügen
meinTupelSpace.Put(tuple.GetFieldAt(0), tuple.GetFieldAt(1))
meinTupelSpace.Put("Butter", 2)
meinTupelSpace.Put("Saft", "Orange", 3)
meinTupelSpace.Put("Saft", "Apfel", 2)
meinTupelSpace.Put
```

Beachte: `meinTupelSpace.Put(tuple)`

fügt ein Einer-Tupel ein, das aus einem Paar besteht:

Es wird `((Milch,1))` und **nicht** `(Milch,1)` eingefügt.

goSpace: Get und GetP

Ein Tupel herausnehmen und ausdrucken:

```
tuple2, _ := meinTupelSpace.Get("Butter",2)
fmt.Println(tuple2)
```

Get ist blockierend, daher führt ein zweites

```
meinTupelSpace.Get("Butter",2)
```

zu einem Deadlock (es gab nur ein ("Butter",2)-Tupel).

Mit GetP wird nicht blockiert:

```
_, err:= meinTupelSpace.GetP("Butter",2)
fmt.Println(err)
```

ergibt

```
Space.main("Butter", 2): operation on this space failed.
```


goSpace: Get und GetP (2)

Casten ist notwendig:

```
tuple3, _ := meinTupelSpace.Get("Milch",1)
var number int = tuple3.GetFieldAt(1).(int) // casting zu int
```

goSpace: Get mit Variablen

Sollen die Werte von Variablen durch Get oder Query gebunden werden, dann

- Statt der Variablen müssen Adressen der Variablen übergeben werden
- D.h. statt `x` wird `&x` übergeben

Falsch:

```
var numberOSaft int = 0
_, err2 := meinTupelSpace.GetP("Saft", "Orange", numberOSaft)
// Das sucht nach dem Tupel ("Saft", "Orange", 0) !
```

Richtig:

```
var numberOSaft int = 0
t, _ := meinTupelSpace.GetP("Saft", "Orange", &numberOSaft)
numberOSaft = t.GetFieldAt(2).(int) // notwendig!
```

Beachte: Binden des Wertes muss noch mal extra geschehen

goSpace: Gesamte Datei

```
package main
import (
    "fmt"
    . "github.com/pspaces/gospace")
func main() {

    var tuple Tuple = CreateTuple("Milch", 1)
    fmt.Println(tuple)

    // Tupel zerlegen
    fmt.Print(tuple.GetFieldAt(0))
    fmt.Println(tuple.GetFieldAt(1))

    // Tuple Space erzeugen
    meinTupelSpace := NewSpace("MeinTupelSpace")

    // Tupel einfügen
    meinTupelSpace.Put(tuple.GetFieldAt(0),tuple.GetFieldAt(1))
    meinTupelSpace.Put("Butter",2)
    meinTupelSpace.Put("Saft","Orange",3)
    meinTupelSpace.Put("Saft","Apfel",2)
    meinTupelSpace.Put("Saft","Kirsche",1)

    // Tupel rausnehmen
    tuple2, _ := meinTupelSpace.Get("Butter",2)
    fmt.Print("Bekam ")
    fmt.Println(tuple2)
```

```
// nicht blockierende Variante
_, err:= meinTupelSpace.GetP("Butter",2)
fmt.Print("Der Fehler ist ")
fmt.Println(err)

tuple3, _ := meinTupelSpace.Get("Milch",1)
var number int = tuple3.GetFieldAt(1).(int) // casting zu int
fmt.Print("Milch hatte die Zahl ")
fmt.Println(number)

// falsch, da nicht die Adresse übergeben wird
var numberOSaft int = 0
_, err2 := meinTupelSpace.GetP("Saft","Orange",numberOSaft)
// Das sucht nach dem Tupel ("Saft","Orange",0) !
fmt.Print("Der Fehler war ")
fmt.Println(err2)

// richtiger Aufruf:
meinTupelSpace.Get("Saft","Orange",&numberOSaft)

// aber numberOSaft ist nicht gesetzt!
fmt.Print("OSaft hatte die Zahl ")
fmt.Println(numberOSaft)

// richtig daher
var numberASaft int = 0
tuple4,_ := meinTupelSpace.GetP("Saft","Apfel",&numberASaft)
numberASaft = tuple4.GetFieldAt(2).(int)
fmt.Print("Apfelsaft hatte die Zahl ")
fmt.Println(numberASaft)
```

Initial: Tupel ("MUTEX") im Tuple Space

Prozess i:

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

(“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
```

(“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
```



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
```



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
```




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
```



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
```



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
```

(“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
```



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
```



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
```

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

Mutex mit goSpace

```
package main
import ("fmt" ; . "github.com/pspaces/gospace"; "strconv"; "bufio"; "os")
func worker(id int, space Space) {
    for {
        space.Get("Mutex")
        fmt.Println("a) Worker " + strconv.Itoa(id)+ " im kritischen Abschnitt.")
        fmt.Println("b) Worker " + strconv.Itoa(id)+ " im kritischen Abschnitt.")
        fmt.Println("c) Worker " + strconv.Itoa(id)+ " im kritischen Abschnitt.")
        space.Put("Mutex")
    }
}
func main() {
    // Tuple Space erzeugen
    space := NewSpace("MeinTupelSpace")
    space.Put("Mutex")
    for i:=0; i < 10; i++ {
        go worker(i,space)
    }
    // Auf Benutzereingabe warten
    reader := bufio.NewReader(os.Stdin)
    reader.ReadString('\n')
}
```


- Tuple Space: Raum von Tupeln, Tupel haben getypte Komponenten, erste Komponente = Name des Tupels
- `out(Name, v1, ..., vn)`: Füge Tupel ein, wobei v_i Werte bzw. Programmvariablen sind, eingefügt werden dann die aktuellen Werte der Variablen.
- `in(Name, x1, ..., xn)`: Nehme "matching Tuple" aus dem Tuple Space heraus, und binde die Werte an die Variablen x_1, \dots, x_n . Blockiere, wenn es kein matching Tuple gibt
- `read(Name, x1, ..., xn)`: Wie `in`, aber das Tuple wird im Tuple Space belassen.

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();  
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)

("SEM")

("SEM")

("SEM")

"S.V = 3"

"S.M = \emptyset "

Prozess P:

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

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 = \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 = {P}"

Prozess P:

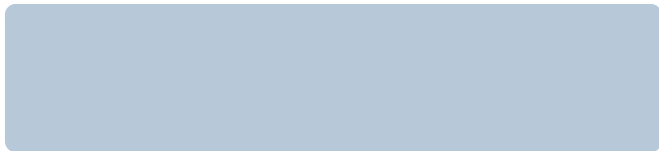
```
wait();  
wait();  
wait();  
signal();  
wait();  
wait();  
wait();  
wait();  
wait();  
wait();
```

blockiert

Prozess Q:

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

Semaphore mit Tuple Spaces (2)



“S.V = 0” “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();
```

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

- 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: $\text{in}(N, x_1, \dots, x_n)$: 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: $\text{in}(N, x_1, \dots, x_n)$: 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=);`

matcht nur Tupel der Form ("Tupel 1", 10)

lässt x unverändert

`x:=10;`

`in("Tupel 1", x);`

matcht alle Tupel der Form ("Tupel 1", $z:\text{Int}$) wobei z eine Zahl

überschreibt Wert von x durch z

Erweiterung der in-Operation in goSpace

In goSpace wird auf die Variablenwerte gematcht, wenn die Variablennamen (statt der Adressen) übergeben werden, d.h.

- `in(Name, x)` entspricht `Get(Name, &x)`
- `in(Name, x=)` entspricht `Get(Name, x)`

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:

$\text{in}(N + \text{“Dienst1“}, \dots); \dots$

Server 2:

$\text{in}(N + \text{“Dienst2“}, \dots); \dots$

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=,...); ...`

- 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");
```


Beispiel in Go

```
func send(space Space, content int) {
    space.Get("channel1", "get")
    space.Put("channel2", content)
    space.Get("channel3", "got")
}

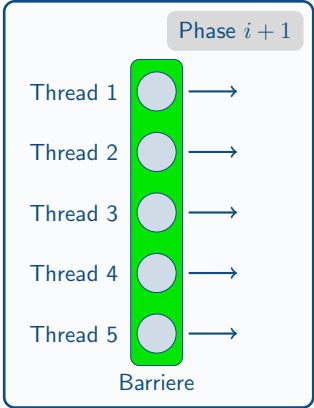
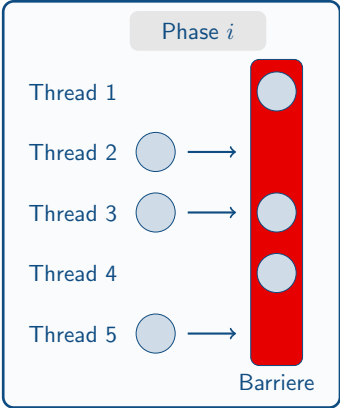
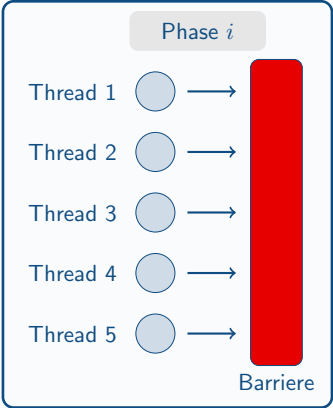
func receive(space Space) int {
    space.Put("channel1", "get")
    var content int = 0
    t, _ := space.Get("channel2", &content)
    content = t.GetFieldAt(1).(int)
    space.Put("channel3", "got")
    return content
}
```

```
func sender(id int, space Space) {
    for {
        time.Sleep(time.Duration(rand.Intn(10000)) * time.Millisecond)
        fmt.Println("Worker " + strconv.Itoa(id) + " will seine Id senden")
        send(space, id)
        fmt.Println("Worker " + strconv.Itoa(id) + " hat seine Id gesendet")
    }
}

func receiver(id int, space Space) {
    for {
        time.Sleep(time.Duration(rand.Intn(10000)) * time.Millisecond)
        fmt.Println("Worker " + strconv.Itoa(id) + " will empfangen")
        var msg int = receive(space)
        fmt.Println("Worker " + strconv.Itoa(id) + " hat "
            + strconv.Itoa(msg) + " empfangen")
    }
}

func main() {
    // Tuple Space erzeugen
    space := NewSpace("MeinTupelSpace")
    for i:=0; i < 100; i++ {
        go sender(i, space) // Sender und Empfaenger erzeugen
        go receiver(100+i, space)
    }
    reader := bufio.NewReader(os.Stdin)
    reader.ReadString('\n')
}
```

Barrieren



Barriere

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 der Barriere
  barrier()
  Code nach der Barriere
end loop
```

Barrienbeispiel in Go

```
// Als dritte Komponente wird stets die
// Gesamtkapazität gespeichert
func newBarrier(space Space, capacity int) {
    space.Put("ankommen", capacity, capacity)
}
func sync(space Space) {
    var capacity int
    var x int
    t, _ := space.Get("ankommen", &x, &capacity)
    x = t.GetFieldAt(1).(int)
    capacity = t.GetFieldAt(2).(int)
    x = x-1
    if (x > 0) { space.Put("ankommen", x, capacity)
    } else { space.Put("verlassen", capacity, capacity) }
    var y int
    t, _ = space.Get("verlassen", &y, capacity)
    y = t.GetFieldAt(1).(int)
    y = y-1
    if y > 0 { space.Put("verlassen", y, capacity)
    } else { space.Put("ankommen", capacity, capacity) }
}
```

```
func worker(id int, space Space) {
    var count int = 1
    for {
        fmt.Println("Worker "
            + strconv.Itoa(id)
            + " arbeitet in Phase "
            + strconv.Itoa(count))
        time.Sleep(time.Duration(rand.Intn(10000))
            * time.Millisecond)
        fmt.Println("Worker " + strconv.Itoa(id) + " an sync")
        sync(space)
        fmt.Println("Worker " + strconv.Itoa(id) + " nach sync")
        count++
    }
}

func main() {
    // Tuple Space erzeugen
    space := NewSpace("MeinTupelSpace")
    newBarrier(space, 5)
    for i:=0; i < 5; i++ {
        go worker(i, space)
    }
    // Auf Benutzereingabe warten
    reader := bufio.NewReader(os.Stdin)
    reader.ReadString('\n')
}
```

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  $\neq$  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)

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

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

Code des Erzeugers:

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

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

Code des Verbrauchers:

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


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

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

Code des Erzeugers:

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

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

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("buffer", tindex', e);  
  out("FIFO-Queue", "tail", tindex');  
  out("notEmpty");  
}
```

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

Erzeuger-Verbraucher in Go

```
func makeBuffer(space Space, capacity int) {
    for i:=1; i <= capacity; i++ { space.Put("notFull")}
    space.Put("FIFO-Queue", "tail",0)
    space.Put("FIFO-Queue", "head",0)
}
func produce(space Space, element int) {
    space.Get("notFull")
    var tindex int
    t,_ := space.Get("FIFO-Queue","tail",&tindex)
    tindex = t.GetFieldAt(2).(int)
    tindex = tindex +1
    space.Put("buffer",tindex,element)
    space.Put("FIFO-Queue","tail",tindex)
    space.Put("notEmpty")
}
func consume(space Space) int {
    space.Get("notEmpty")
    var hindex int
    t,_ := space.Get("FIFO-Queue","head",&hindex)
    hindex = t.GetFieldAt(2).(int)
    hindex = hindex +1
    var result int
    t2,_ := space.Get("buffer",hindex,&result)
    result=t2.GetFieldAt(2).(int)
    space.Put("FIFO-Queue", "head",hindex)
    space.Put("notFull")
    return result
}
```

```
func producer(id int,space Space) {
    for {
        time.Sleep(time.Duration(rand.Intn(10000)) * time.Millisecond)
        fmt.Println("Producer " + strconv.Itoa(id)+ " will produzieren")
        produce(space,id)
        fmt.Println("Producer " + strconv.Itoa(id)+ " hat produziert")
    }
}
func consumer(id int,space Space) {
    for {
        time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
        fmt.Println("Consumer " + strconv.Itoa(id)+ " will konsumieren")
        t := consume(space)
        fmt.Println("Consumer " + strconv.Itoa(id)+ " bekam " + strconv.Itoa(t))
    }
}
func main() {
    // Tuple Space erzeugen
    space := NewSpace("MeinTupelSpace")
    makeBuffer(space,3)
    for i:=0; i < 10; i++ {
        go consumer(i,space)
    }
    for i:=0; i < 10; i++ {
        go producer(i,space)
    }
    reader := bufio.NewReader(os.Stdin)
    reader.ReadString('\n')
}
```

Readers & Writers

Reader:

startRead();
Kritischer Abschnitt;
stopRead();

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

Writer:

startWrite();
Kritischer Abschnitt;
stopWrite();

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);  
}
```

```
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');  
}
```

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

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