7.2 Sprachkonstrukte zum gegenseitigen Ausschluss

Wie wir gesehen haben, ist die Sicherstellung des gegenseitigen Ausschlusses eine der zentralen Aspekte bei der nebenläufigen Programmierung. Hierfür wurden verschiedene Konzepte entwickelt, die wir im Folgenden näher betrachten wollen.

7.2.1 Semaphore

Zur Vermeidung von Busy Waiting wurde schon sehr früh das Konzept von **Semaphoren** eingeführt.

Definition 7.1 (Semaphore (Dijkstra 1968))**.** *Ein Semaphor ist eine nichtnegative ganzzahlige Variable mit einer Prozesswarteschlange und zwei atomaren Operationen.*

`P` ***(„passeren") oder „wait":*** *Falls Semaphorwert > 0, erniedrige diesen um 1, sonst stoppe die Ausführung des aufrufenden Prozesses und trage diesen in die Prozesswarteschlange des Semaphors ein.*

`V` ***(„vrijgeven") oder „signal":*** *Falls Prozesse in der Warteschlange warten, dann aktiviere einen, sonst erhöhe den Semaphorwert um 1.*

*Der Initialwert eines Semaphors ist 1 (dann sprechen wir auch von einem **binären Semaphor**) oder $n > 1$ (falls n Prozesse in den kritischen Bereich gleichzeitig eintreten können sollen).*

Wichtig ist, dass `P`/`V` **unteilbare** (atomare) Operationen sind, die durch das Betriebssystem realisiert werden. Mit Semaphoren können wir kritische Bereiche absichern, indem wir `P` zu Beginn und `V` am Ende benutzen.

Beispiel 7.4 (Nutzung eines Semaphors)**.** Sei `s` ein binärer Semaphor. Dann kann ein kritischer Bereich wie folgt abgesichert werden:

```
⋮
P(s);
x := x+1;  // kritischer Bereich
V(s);
⋮
```

In Programmiersprachen können Semaphore auf unterschiedliche Weise verwendet werden:

- Algol 68: Dies war die erste Programmiersprache mit eingebauten Semaphoren.
- C: Die Programmiersprache C nutzt Unix-Bibliotheken für Semaphore (`sys/sem.h`). Die Erzeugung neuer Prozesse erfolgt durch Kopieren (`fork()`) eines existierenden

Prozesses, d.h. im Prinzip läuft in beiden Prozessen der gleiche Code ab. Um dann trotzdem unterschiedliche Aufgaben in den Prozessen zu bearbeiten, liefert `fork()` als Ergebnis die Zahl `0` im neuen Prozess und eine positive Zahl (Process Identifier) im Elternprozess, sodass dann eine Verzweigung auf Grund dieses Wertes vorgenommen werden kann.

Die direkte Benutzung von Semaphoren hat aber einige Nachteile:

- Es ist ein sehr niedriger Ansatz zur Synchronisation. Man könnte auch sagen, dass Semaphore der Assembler für die Synchronisation sind, d.h. sie sind die Grundlage, um höhere Konzepte zu realisieren.
- Die Benutzung von Semaphore ist sehr fehleranfällig (wie auch die Benutzung von Assembler), da man z.B. darauf achten muss, dass zu jeder `P`-Operation irgendwann immer eine `V`-Operation ausgeführt wird. Ebenso kann die Synchronisation mit Semaphoren bei der Programmierung von komplexen, gemeinsam benutzten Datenstrukturen sehr komplex sein.

7.2.2 Monitore

Die Nachteile der Semaphore führte Hoare 1974 zur Idee, eine höhere Abstraktion zur Synchronisation zu entwickeln.

Definition 7.2 (Monitor)**.** *Ein **Monitor** ist ein Modul/Objekt, welches den synchronisierten Zugriff auf Daten organisiert. Ein Monitor besteht aus*

- *lokalen Daten,*
- *Operationen auf diesen Daten,*
- *einem Initialisierungsteil und*
- *einem Mechanismus zur Suspension („delay“) und zum Aufwecken („continue“) von Operationen/Prozessen.*

Wichtig sind zudem die folgenden Eigenschaften:

- *Der Zugriff auf lokale Daten erfolgt nur über die Operationen des Monitors (somit entspricht ein Monitor einem ADT).*
- *Zu jedem Zeitpunkt kann nur ein Prozess in den Monitor eintreten (d.h. eine Operation aufrufen). Während dieser Zeit müssen Aufrufe von anderen Prozessen warten.*

Beispiel 7.5 (Monitor)**.** Wir betrachten einen Puffer in Concurrent-Pascal, einer Erweiterung der Programmiersprache Pascal um Nebenläufigkeit mit Monitoren:

```
type buffer =
  monitor
    var contents: array[1..n] of ...;
        num: 0..n;  { number of elements }
        sender, receiver: queue;

    procedure entry append (item: ...);
    begin
      if num = n then delay(sender); { buffer is full }
      ... { insert in buffer }
      continue(receiver);
    end;

    procedure entry remove (var item: ...);
    begin
      if num = 0 then delay(receiver);
      ... { take one item }
      continue(sender);
    end;
  begin
    num := 0; ...
  end;
```

Wie man sehen kann, gehört zu Monitoren auch das Konzept von Prozesswarteschlangen (`queue`), um im Monitor befindliche Prozesse zu suspendieren, falls bestimmte Bedingungen nicht erfüllt sind.

- `delay(Q)` fügt den aufrufenden Prozess in die Warteschlange `Q` ein.
- `continue(Q)` aktiviert einen Prozess aus der Warteschlange `Q`.
- `entry` markiert eine von außen aufrufbare Monitor-Operation.

Monitore sind prinzipiell ein elegantes Konzept (alle kritischen Bereiche und Datenstrukturen sind in einem ADT gekapselt), aber trotzdem wurde dieses eher selten explizit in Programmiersprachen verwendet (z.B. in Concurrent Pascal, Mesa). Man findet aber konzeptuell ähnliche Konzepte in anderen Programmiersprachen. Z.B. bietet Java ein Monitor-ähnliches Konzept an (siehe Kapitel 7.3).

7.2.3 Rendezvous-Konzept

Ein weiteres höheres Synchronisationskonzept ist das sogenannte **Rendezvous-Konzept**.

Ein **Rendezvous** bezeichnet die Kommunikation und Synchronisation von Prozessen durch koordiniertes Abarbeiten einer Prozedur. Dieses Synchronisationskonzept findet man in den Sprachen Ada und auch in Concurrent C. Die Grundidee des Rendezvous ist eine Client-Server-Kommunikation:

Server: bietet einen oder mehrere Einstiegspunkte mit Parametern zum Aufruf an und wartet auf einen Aufruf.

Client: Ruft diese Einstiegspunkte auf und wartet bei dem Aufruf (d. h. er blockiert), bis dieser akzeptiert und abgearbeitet ist (daher die Bezeichnung *Rendezvous*).

Beispiel 7.6 (Rendezvous)**.** Die Sprachkonstrukte für das Rendezvous-Konzept in der Sprache Ada sind:

- Definition eines Einstiegspunktes:

```
accept <entryname + parameter> do
  <body>
end
```

- Aufruf eines Einstiegspunktes:

```
<servername>.<entryname + parameter>
```

Dadurch ergibt sich der folgende zeitliche Ablauf:

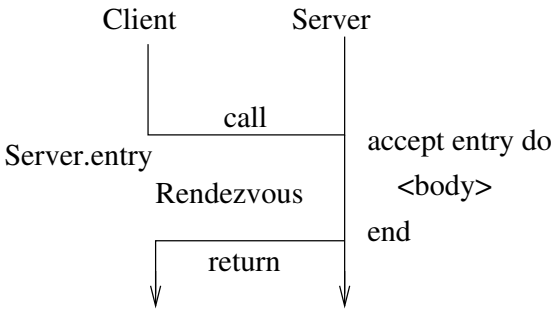

Abbildung 7.3: Ablauf Rendezvous.

Anmerkungen:

- Der Aufruf kann vor `accept` oder auch umgekehrt erfolgen, da beide Partner bis zur Synchronisation warten.
- Ein Server kann mehrere Clients bedienen.
- Ein Server kann mehrere Einstiegspunkte anbieten. Hierzu gibt es die allgemeinere Form:

```
select
  when <condition> => accept ... end; ...
or
  when <condition> => accept ... end; ...
end select;
```

Die Unterschiede zu Monitoren sind:

- Der Server ist kein eigenständiges Modul, das gemeinsame Datenstrukturen verwaltet.
- Jeder Prozess (in Ada: `task`) kann mittels einer `accept`-Anweisung zu einem Server werden.
- Die Server/Client-Funktion ist nicht global fixiert (im Gegensatz zu einem Monitor, der immer ein Server ist), sondern kann dynamisch variieren.
- Es gibt (implizit) eine Warteschlange pro Einstiegspunkt.

Beides sind daher unterschiedliche, aber hohe Konzepte (im Gegensatz zu Semaphoren) zur Synchronisation. Um die Benutzung des Rendezvous-Konzepts zu zeigen, betrachten wir als wieder einen Puffer.

Beispiel 7.7 (Puffer in Ada mittels Rendezvous-Konzept)**.**

```
task Buffer is
  entry Append(item: in INTEGER);
  entry Remove(item: out INTEGER);
end;

task body Buffer is
  contents: array (1..n) of INTEGER;
  num: INTEGER range 0..n := 0;
  ipos, opos: INTEGER range 1..n := 1;
 begin loop
  select
    when num<n =>
      accept Append(item: in INTEGER) do
        contents(ipos) := item;
      end;
    ipos := (ipos mod n)+1; num := num+1;
  or
    when num>0 =>
      accept Remove(item: out INTEGER) do
        item := contents(opos);
      end;
    opos := (opos mod n)+1; num := num-1;
  end select;
 end loop;
end Buffer;
```

Anmerkungen:

- Die Abarbeitung eines `accept` ist ähnlich zu einem Prozeduraufruf mit üblicher Parameterübergabe, aber der Prozeduraufruf des Client wird im Server abgearbeitet.
- Ein *Erzeuger-Client*wird durch den Server-Aufruf `Buffer.Append(v)` gestartet.
- Ein *Verbraucher-Client* wird durch den Server-Aufruf `Buffer.Remove(v)` gestartet.
- Das Rendezvous ist erst möglich, falls die `when`-Bedingung erfüllt ist.

7.2.4 Nachrichtenaustausch (Message Passing)

Bei der Kommunikation durch **Nachrichtenaustausch** (message passing) erfolgt die Kommunikation und Synchronisation zwischen Prozessen durch das Versenden von Nachrichten über Kommunikationskanäle, welche häufig vom Betriebssystem unterstützt werden. Unterschiede gibt es bei diesem Konzept bei

- der Struktur der Kanäle (symmetrisch/asymmetrisch), und
- der Synchronisation.

Vorteile des Nachrichtenaustausches:

- Es gibt keine kritischen Bereiche, da Prozesse konzeptuell getrennt sind, d.h. sie haben damit keinen gemeinsamen Speicherbereich.
- Es erlaubt flexible Kommunikationsstrukturen ($1:1$, $1:n$, $n:1$, $n:m$ Kommunikation).

Modelle zum Nachrichtenaustausch:

Punkt-zu-Punkt: Ein Sender schickt eine Nachricht an einen Empfänger (Prozess).

symmetrisch: Empfänger und Sender kennen sich gegenseitig mit Namen, dadurch kann der Empfänger direkt antworten (vgl. Telefon)

asymmetrisch: Nachrichtenfluss nur in eine Richtung (vgl. „pipes“ in Unix). Fall es viele Sender gibt, muss der Empfänger die eingehenden Nachrichten zwischenspeichern: Jeder Empfänger hat einen Empfangspuffer, welcher in diesem Zusammenhang auch als **mailbox** oder **port** bezeichnet wird.

synchrone Kommunikation: Der Sender wartet, bis der Empfänger die Nachricht akzeptiert hat. (z.B. bei ungepufferten Kanälen wie in Go, s.u.).

asynchrone Kommunikation: Der Sender arbeitet nach dem Versand einer Nachricht direkt weiter, die Empfängerbestätigung muss explizit programmiert werden (häufig bei Netzwerken und many-to-one-Kommunikation).

Rendezvous: s.o.; im Wesentlichen synchroner Nachrichtenaustausch zwischen Sender und Empfänger.

Prozedurfernaufrufe: Prozedurfernaufrufe (*remote procedure call*, RPC) bzw. Methodenfernaufrufe (*remote method invocation*, RMI (Java)) ähneln dem Rendezvous-Konzept, jedoch wird hier versucht, diese sprachlich analog zu lokalen Prozedur- bzw. Methodenaufrufen zu behandeln, was folgende Vorteile bietet:

- Einfache Einbettung in existierende Programmiersprachen, d.h. es muss keine neue Syntax hierfür eingeführt werden.
- „Einfache“ Portierung existierender sequentieller Programme in einen nebenläufigen Kontext. Allerdings sind dann komplexere Fehlersituationen möglich!

Gruppenkommunikation (broadcasting, multicasting): ein Prozess kann Nachrichten direkt an eine Menge von Empfängerprozessen schicken.

Diese Konzepte können in Programmiersprachen z.B. durch das Betriebssystem und/oder entsprechende Bibliotheken unterstützt werden, was im Wesentlichen keine Änderung der Basissprache verlangt. Es gibt aber auch Programmiersprachen, die spezielle Konstrukte anbieten, um mit Nachrichtenaustausch zu arbeiten. Als Beispiel betrachten wir die Konzepte der Programmiersprache Go zur nebenläufigen Programmierung etwas genauer.

Go unterstützt die einfache Erzeugung leichtgewichtiger Prozesse ("Goroutines"), die mittels Nachrichtenaustausch über Kanäle miteinander kommunizieren. Für diesen Zweck gibt es folgende Konstrukte in der Sprache Go:

1. Einfache Erzeugung eines Prozesses, genannt **Goroutine**. Ein solcher Prozess ist im Prinzip ein Prozeduraufruf, der nebenläufig zum Hauptprogramm abgearbeitet wird. Dieser wird mit dem Schlüsselwort `go` gestartet:

   ```
   go worker(···)
   ```

   Da `go` eine Anweisung ist, werden mögliche Ergebniswerte der Prozedur ignoriert.

2. Ein Datentyp für getypte Kanäle zum Austausch von Daten zwischen Prozessen. Hierfür gibt es das Schlüsselwort `chan`, wobei der Typ der Kanaldaten dahinter geschrieben wird und ein Pfeil vor dem Schlüsselwort für einen Eingabekanal und ein Pfeil dahinter für einen Ausgabekanal steht. Z.B. wird durch

   ```
   func worker(in <-chan int, out chan<- string) { ... }
   ```

   eine Prozedur deklariert, die ganze Zahlen aus einem Kanal `in` lesen und Zeichenketten auf einen Kanal `out` schreiben kann. Neue Kanäle werden mit

   ```
   make(chan τ)
   ```

   erzeugt.

3. Ein Element kann durch einen Pfeil links von einem Kanal gelesen und einen Pfeil rechts in einen Kanal geschrieben werden:

   ```
   a = <-in   // receive element from channel 'in'
   out <- e   // send element to channel 'out'
   ```

Somit ist das Kommunikationskonzept von Go stark beeinflusst durch das von Tony Hoare entwickelte theoretische Modell der "Communicating Sequential Processes" (**CSP** (Hoare, 1978)). Auf Grund der sehr effizienten Implementierung dieses Modells kann man in Go mit einer großen Anzahl an Prozessen arbeiten.

Als einfaches Beispiel betrachten wir die Berechnung von pythagoreischen Tripeln, d.h. ganzen positiven Zahlen a, b, c mit der Eigenschaft $a*a+b*b = c*c$. Gesucht ist ein Tripel

mit einer bestimmten Summe, z.B. 5000. Da diese Tripel sehr unregelmäßige Abstände haben, wollen wir ein Go-Programm schreiben, dass solche Tripel mit mehreren parallel ablaufenden Prozessen berechnet. Zur Darstellung der Tripel definieren wir `Triplet` als Typsynonym für ein 3-elementiges Feld:

```
type Triplet [3]int
```

Ein einzelner Berechnungsprozess sucht für eine Zahl a alle größeren Zahlen b und c, sodass a, b, c ein pythagoreisches Tripel ist. Weil diese Prozesse nebenläufig ablaufen, holen sie sich ihre Startwerte a aus einem Eingabekanal mit Zahlen und schreiben die Ergebnisse auf einen Ausgabekanal mit Tripeln:

```
func FindPythTriplets(in <-chan int, out chan<- Triplet, maxsum int) {
    i := <-in // Receive value from 'in'.
    for i < maxsum {
        for j := i+1; i+j < maxsum; j++ {
            for k := j+1; i+j+k <= maxsum; k++ {
                if i*i + j*j == k*k {
                out <- Triplet{i,j,k} // Send triplet to 'out'.
            }
            }
        }
        i = <-in  // Get next start value
    }
}
```

Um diese Prozesse mit geeigneten Startwerten zu versorgen, benötigen wir noch einen Kanal mit positiven ganzen Zahlen, den wir durch folgende Prozedur erzeugen:

```
// Send the sequence 1, 2, 3, 4, ... to channel 'ch'.
func Generate(ch chan<- int) {
    for i := 1; ; i++ {
        ch <- i
    }
}
```

Man beachte, dass dieser Kanal konzeptionell beliebig viele Elemente enthält, aber immer gewartet wird, bis Elemente gelesen sind, weil der Kanal nicht gepuffert ist.

Das Hauptprogramm besteht nun aus der Erzeugung der Kanäle und einer bestimmten Anzahl von Arbeitsprozessen für das Suchen der pythagoreischen Tripel:

```
func main() {
    const sum = 5000          // the sum we are searching
    const numworkers = 20     // number of workers
```

```
    numbers := make(chan int)      // create new channel with numbers
    go Generate(numbers)
    triplets := make(chan Triplet) // create new channel for results
    for i := 1; i <= numworkers; i++ { // create workers
        go FindPythTriplets(numbers, triplets, sum)
    }

    ts := Triplet{0,0,0}
    for ts[0]+ts[1]+ts[2] != sum {
       ts = <- triplets
    }
    fmt.Println(ts)
}
```

Lässt man dieses Programm mit unterschiedlich vielen Prozessoren laufen, kann man deutliche Laufzeitunterschiede erkennen.

Somit kann man sehen, dass Go es ermöglicht, mit wenig Aufwand nebenläufige Programme zu realisieren. Im unserem Beispiel sollte man beachten, dass die Arbeitsprozesse alle vom gleichen Kanal ihre Startwerte holen und die Ergebnisse ebenfalls in einen gemeinsamen Kanal schreiben. Weil die Lese- und Schreiboperationen auf Kanälen atomar sind, sind weitere Maßnahmen zur Synchronisation nicht erforderlich.