Errata et Addenda

Fehler beim Satz

Auf S. 106, 109, 125 und in Tab. 4.1, 10.1, 10.3, 10.4 ist der Dekrementierungsoperator falsch gesetzt: „-“ muss in diesen Fällen durch „--“ ersetzt werden.

Auf S. 172 ist die waagerechte Linie in der Mitte der Tab. 6.4 zu entfernen.

Schreibfehler

In Zeile 17 von unten auf S. 16 ist `Counter++` durch `n++` zu ersetzen.

In Zeile 3 von unten auf S. 24 ist `thread-` durch `pthread-` zu ersetzen.

In Zeile 12 auf S. 77 ist `inWaitingRomm` durch `inWaitingRoom` zu ersetzen.

In Zeile 9 auf S. 124 ist `type addSemaphrore struc` durch `type addSemaphore struct` zu ersetzen.

In Zeile 2 von unten auf S. 218 ist `nicht` durch `nichts` zu ersetzen.

In Zeile 13 auf S. 259 ist `c <- true` durch `c <- 0` zu ersetzen.

Algorithmus von Kessels

Das `Lock`-Protokoll im Algorithmus von Kessels auf S. 55 ist nicht korrekt.
Die Semantik von `turn` ist folgende:

- `favoured == 0` genau dann, wenn `turn[0] != turn[1]` und
- `favoured == 1` genau dann, wenn `turn[0] == turn[1]`,

Die Einführung der lokalen Variablen `local` war eine Schnapsidee. Damit lautet das Eintrittsprotokoll für Prozess `0`:

```
  Store (&interested[0], 1)
  Store (&turn[0], turn[1])
  for interested[1] && turn[0] == turn[1] {
    Nothing()
  }
}
```

und das für Prozess `1`:

```
  Store (&interested[1], 1)
  Store (&turn[1], 1 - turn[0])
  for interested[0] && turn[1] != turn[0] {
    Nothing()
  }
}
```

– zusammengefasst:

```
func Lock (i uint) { // i < 2
  Store (&interested[i], 1)
  Store (&turn[i], (i + turn[1-i]) % 2)
  for interested[1-i] == 1 && turn[i] == (i + turn[1-i]) % 2 {
    Nothing()
  }
}
```

Auch die Version für *n* Prozesse in der Version nach Taubenfeld (S. 70/71) ist nicht korrekt.

Hier die genaue Übersetzung aus der Originalarbeit von KESSELS nach Go:

```
package lockn

import (. "nU/obj"; . "nU/atomic")

type kessels struct {
  uint "number of processes"
  interested, turn [][2]uint
  e []uint // < 2
}

func newKessels (n uint) LockerN { // n is a power of 2.
  x := new(kessels)
  x.uint = uint(n)
  x.interested = make([][2]uint, n)
  x.turn = make([][2]uint, n)
  x.e = make([]uint, n)
  return x
}

func (x *kessels) Lock (p uint) {
  for n := x.uint + p; n > 1; n /= 2 {
    k, m := n / 2, n % 2
    Store (&x.interested[k][m], 1)
    m1 := 1 - m
```

```
    Store (&x.turn[k][m], (x.turn[k][m1] + m) % 2)
    for x.interested[k][m1] == 1 && x.turn[k][m] == (x.turn[k][m1] + m) % 2 {
      Nothing()
    }
    x.e[k] = m
  }
}

func (x *kessels) Unlock (p uint) {
  n := uint(1)
  for n < x.uint {
    n = 2 * n + x.e[n]
    Store (&x.interested[n/2][n%2], 0)
  }
}
```

Go-Scheduler

Der Zuteiler von Go ist nicht in der Lage, Go-Routinen zu unterbrechen, die in einer beschäftigten Warteschleife hängen (meines Erachtens ein Fehler im Go-System). Aus diesem Grunde ist es nicht nur „eventuell“ sinnvoll, sondern *zwingend erforderlich*, ihm die Möglichkeit zu geben, innerhalb solcher Warteschleifen auf eine andere Goroutine umzuschalten, damit Programme, die solche Schleifen benutzen, nicht in ihnen hängen bleiben.

Das Setzen solcher Unterbrechungspunkte kann mit einem Aufruf von `time.Sleep(1)` oder mit einem expliziten Hinweis an den Zuteiler, dass er umschalten kann, durch den Aufruf der Funktion `runtime.Gosched()` (s. https://golang.org/pkg/runtime/#Gosched) erreicht werden.

Der letzte Absatz auf S. 40 ist daher wie folgt zu ändern:

Dabei ist die Funktion `Nothing` zwar *im Prinzip* die „leere Anweisung“

```
func Nothing() {
}
```

Die beiden ersten Zeilen auf S. 41 sind wie folgt zu ersetzen:

Aber: Aus den genannten Gründen ist die leere Implementierung dieser Funktion entweder durch den Aufruf von `time.Sleep(1)` oder von `runtime.Gosched()` zu ersetzen.

Auch in den folgenden Algorithmen ist in jeder beschäftigten Warteschleife am Ende ihres Rumpfs noch ein Aufruf von `Nothing()` einzufügen:

- im Algorithmus von DEKKER auf S. 62,
- im Algorithmus von KNUTH auf S. 64,
- im Algorithmus von HABERMANN auf S. 65 und 66 und
- im Algorithmus von SZYMANSKI auf S. 77 und 78.

Zur Unteilbarkeit elementarer Wertzuweisungen

Die im vorletzten Absatz auf S. 20 angegebene Grundannahme, dass eine Wertzuweisung `y = x` an eine Variable `y` eines elementaren Typs unteilbar ist, wenn `x` eine Konstante oder eine Variable eines zuweisungsverträglichen Typs ist, war in Version 1.9 noch erfüllt. *Das gilt aber spätestens seit Version 1.12 nur noch, wenn der verwendete Rechner nur einen Prozessor hat oder wenn `runtime.GOMAXPROCS(1)` aufgerufen wurde*.

Aus diesem Grunde garantieren quasi sämtliche in den Abschnitten 2.4 und 2.5 angeführten hochsprachlichen Schlossalgorithmen in der Form, wie sie dort angegeben sind, für Rechner mit mehreren Prozessoren nicht mehr den gegenseitigen Ausschluss.

Die in den `Lock`- und `Unlock`-Funktionen verwendeten Wertzuweisungen `y = x` vom oben angegebenen Typ müssen also in diesen Protokollen *grundsätzlich* durch die Verwendung einer unteilbaren Wertzuweisung ersetzt werden.

Mit dem Ziel der Unabhängigkeit von der Busbreite der verwendeten Rechner, also von der Frage, ob der Typ `uint` als `uint32` oder `uint64` realisiert ist, verwenden wir die atomare Funktion

```
// *n = k.
func Store (n *uint, k uint)
```

mit den Implementierungen in Go-Assembler

```
TEXT ·Store(SB),NOSPLIT,$0
  MOVL n+0(FP), BX
  MOVL k+4(FP), AX
  XCHGL AX, 0(BX)
  RET
```

für 32-bit-Rechner und

```
TEXT ·Store(SB),NOSPLIT,$0
  MOVQ n+0(FP), BX
  MOVQ k+8(FP), AX
  XCHGQ AX, 0(BX)
  RET
```

für 64-bit-Rechner, die wir in unserem Paket `nU/atomic` unterbringen.

Die Slices `interested` und `critical` in diesen Protokollen müssen dementsprechend *auch* den Typ `[]uint` anstelle von `[]bool` haben (mit `0` für `false` und `1` für `true`).