**ALP4 SoSe 2013, Di. 16-18**

# Lösung Übungsblatt 5

**Christoph van Heteren-Frese (Matr.-Nr.: 4465677)**
**Sven Wildermann (Matr.-Nr.: 4567553)**
Tutor: Alexander Steen, eingereicht am 24. Mai 2013

---

## Aufgabe 1

1. Mit Hilfe des Tiefensuche-Algorithmus kann ein Graph auf Zyklenfreiheit untersucht werden. Hierzu überprüft man mit diesem Algorithmus, ob der Graph G eine Rückwärtskante besitzt oder nicht. Methode: Erweitere DFS um Speicherung der Rückwärtskanten (Back-Kante / B-Kante), sobald man von "v" aus auf einem markierten Knoten "u" trifft, hat man eine B-Kante gefunden, falls "u" nicht der direkte Vorgänger von "v" ist.

2. Algorithmus:

```
1        function ISACYCLIC(GRAPH G=(V,E)): bool {
2            B:={}
3            for all v in V do { marked[v]:=false; p(v):=nil}
4            for all v in V do {
5              if not marked[v] then
6              DFS-ACYCLIC(v)
7            }
8            if B:={} then return false else return true
9         }
10
11        procedure DFS-ACYCLIC(Node v){
12             marked[v]:=true
13             for all w in N(v) do
14                if not marked[w] then
15                       p(w):=v
16                       DFS-ACYCLIC(w)
17                else if p(v) =/ w then
18                      B:=B + (v,w)    // + ist hier die Mengenvereinigung
19        }
```

   Komplexität: Da jede Kante und jeder Knoten genau einmal besucht wird, beträgt die Laufzeit von Tiefensuche O(#V+#E), wobei #V = Anzahl der Knoten und #E = Anzahl der Kanten.

## Aufgabe 2

Die hier vorgestellten Implementierungnen ensprechen in etwa dem Algorithmus, den Horare in [1] vorgestellt hat.

Der Monitor hat vier Zugriffsfunktionen:

1. `start_read` wird durch reader aufgerufen, der lesen möchte

2. `end_read` wird durch reader aufgerufen, der lesen beendet
3. `start_write` wird durch writer aufgerufen, der schreiben möchte
4. `end_write` wird durch writer aufgerufen, der schreiben beendet

Die Variable `rc` zählt die aktiven Lerser, `rq` die wartenden Leser und `wq` die wartenden Schreiber. `busy` zeigt an, ob gerade ein Schreiber am seine Arbeit verrichtet.

a) Implemetierung in C

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdbool.h>
4
5  unsigned int rc = 0;
6  unsigned int rq = 0;
7  unsigned int wq = 0;
8  bool busy = false;
9  pthread_mutex_t mutex =
10   PTHREAD_MUTEX_INITIALIZER;
11 pthread_cond_t OKtowrite =
12   PTHREAD_COND_INITIALIZER;
13 pthread_cond_t OKtoread =
14   PTHREAD_COND_INITIALIZER;
15
16 void *reader_start() {
17   pthread_mutex_lock(&mutex);
18   if (busy || wq != 0) {
19     pthread_cond_wait(&OKtoread,
20       &mutex);
21   }
22   rc ++;
23   pthread_cond_signal(&OKtoread);
24   pthread_mutex_unlock(&mutex);
25 }
26 void *reader_end() {
27   pthread_mutex_lock(&mutex);
28   rc--;
29   if (rc ==  0) {
30     pthread_cond_signal(&OKtowrite);
31   }
32   pthread_mutex_unlock(&mutex);
33 }
34 void *writer_start() {
35   pthread_mutex_lock(&mutex);
36   if ( rc != 0 || busy) {
37     pthread_cond_wait(&OKtowrite, &mutex);
38   }
39   busy = true;
40   pthread_mutex_unlock(&mutex);
41 }
42
43 void *writer_end() {
44   pthread_mutex_lock(&mutex);
45   busy = false;
46   if ( rq != 0) {
47     pthread_cond_signal(&OKtoread);
48   } else {
49     pthread_cond_signal(&OKtowrite);
50   }
51   pthread_mutex_unlock(&mutex);
52 }
53
54 main () {
55   pthread_t p, q;
56   void *f (void *x) {
57     reader_start();
58     return NULL;
59   }
60   void *g(void *y)  {
61     writer_start();
62     return NULL;
63   }
64   pthread_create(&p, NULL, &f, NULL);
65   pthread_create(&q, NULL, &g, NULL);
66   (void) pthread_join(p,NULL);
67   (void) pthread_join(q,NULL);
68 }
```

b) Implementierung in Go

```
1  package rw
2  import . "sync"
3
4  type Imp struct {
5    rc, rq, wq uint
6    mutex Mutex
7    OKtowrite *Cond
8    OKtoread *Cond
9    busy bool
10   }
11
12 func New() *Imp {
13   x:= new(Imp)
14   x.OKtowrite = NewCond(&x.mutex)
15   x.OKtoread = NewCond(&x.mutex)
16   x.busy = false
```

```
17    x.rc, x.rq, x.wq = 0, 0 ,0
18    return x
19  }
20
21  func (x *Imp) Reader_start() {
22    x.mutex.Lock()
23    if x.busy || x.wq != 0 {
24      x.OKtoread.Wait()
25      x.rq ++
26    }
27    x.rc ++
28    x.OKtoread.Signal()
29    x.mutex.Unlock()
30  }
31
32  func (x *Imp) Reader_end() {
33    x.mutex.Lock()
34    x.rc --
35    if x.rc == 0 { x.OKtowrite.Signal() }
36    x.mutex.Unlock()
37  }
```

```
38
39  func (x *Imp) Writer_start() {
40    x.mutex.Lock()
41    if x.rc != 0 || x.busy {
42      x.OKtowrite.Wait()
43      x.wq ++
44    }
45    x.busy = true
46    x.mutex.Unlock()
47  }
48
49  func (x *Imp) Writer_end() {
50    x.mutex.Lock()
51    x.busy = false
52    if x.rq != 0 {
53      x.OKtoread.Signal()
54    } else {
55      x.OKtowrite.Signal()
56    }
57    x.mutex.Unlock()
58  }
```

## Aufgabe 3

Diese Lösung des Leser-Schreiber-Problems funktioniert nicht. Ein Gegenbeispiel:

- R1 möchte auf die Datei F1 lesen und führt daher die Funktion ReaderIn (erfolgreich) aus.
- Nun möchte (während R1 liest) W1 auf die Datei F1 schreibend zugreifen. Da nun aber die Bedingung (nR größer 0) erfüllt ist, wartet W1 indem es c.Wait() ausführt.
- Hinweis: Diese Wait-Anweisung befindet sich innerhalb des Blocks, in dem m.Lock aktiv ist.
- m.Unlock wird von W1 also solange nicht ausgeführt wie R1 noch liest.
- Jetzt möchte R1 den Lesevorgang auf F1 wieder beenden, in dem es ReaderOut() ausführt.
- ReaderOut() kann allerdings nicht ausgeführt werden, da m.Lock() nicht ausgeführt werden kann - weil W1 diesen Lock nicht wieder freigegeben hat.
- **Damit ist ein Livelock entstanden. Der Lesezugriff auf die Datei kann ohne Neustart des Systems nicht mehr geschlossen werden. Eine Dateiveränderung ist mit dieser Lösung nicht mehr möglich, sobald auch nur ein Thread lesenden Zugriff auf die Datei hat.**

## Aufgabe 4

Das Krümelmonsterproblem kann im Prinzip durch den im Buch gegebenen Algorithmus für das Konto gelöst werden [vgl. 2, S. 135].

```
1  package kruemel
2  import . "sync"
3
4  type Imp struct {
5    cookies uint
6    mutex Mutex
7    credit *Cond
8    }
9
10 func New() *Imp {
11   box:= new(Imp)
12   box.credit = NewCond(&box.mutex)
13   return box
14 }
15
16 func (box *Imp) Bake(c uint) {
17   box.mutex.Lock()
18   box.cookies += c
19   box.credit.Signal()
20   box.mutex.Unlock()
21 }
22
23 func (box *Imp) Eat(c uint) {
24   box.mutex.Lock()
25   for box.cookies < c {
26     box.credit.Wait()
27   }
28   box.cookies -= c
29   box.credit.Signal()
30   box.mutex.Unlock()
31 }
32
33 func (box *Imp) Balance() uint {
34   return box.cookies
35 }
```

**Listing 1:** Krümelmonsteproblem in Go.

## Literatur


[1] CAR Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10), 1974. URL `http://dl.acm.org/citation.cfm?id=361161`.

[2] Christian Maurer. *Nichtsequentille Programmierung mit Go 1 Kompakt.* Springer Vieweg, 2012. ISBN 978-3642299681.