# Übungsblatt 1

Ausgabe: 22.04.2008
Abgabe: 07.05.2008, 16:00 Uhr

## Aufgabe 1 (4 P)

In einem sequenziellen Programm können Abschnitte mittels semantikerhaltender Transformation durch andere Abschnitte ersetzt werden, die dieselbe Wirkung auf den Programmzustand haben. Zum Beispiel:

Ersetze `ADD(i,2);` durch `INC(i); INC(i);`
Ersetze `SQR(i);` durch `t = i; MUL(t,i); i = t;` (SQR: von englisch "to square" - quadrieren)

Beschreiben Sie etwaige Probleme, die sich bei dem Einsatz dieser Transformationen im Kontext nebenläufiger Ausführung ergeben können!

Wenden Sie die Transformationen auf folgendes Beispiel an:

```
i = 10;
co ADD(i,2); || SQR(i); oc
```

Wie lauten alle möglichen Werte von i?
Begründen Sie Ihre Antwort unter Einsatz einer grafischen Darstellung der möglichen Ablauffolgen!

Hinweis: Setzen Sie voraus, dass die Operationen `INC, MUL` unteilbar und die Variablen einfach sind.

## Aufgabe 2 (6 P)

Die Gabelungsanweisung zum Abzweigen eines eigenständig ablaufenden Prozesses **`fork`** soll in Java als neues Sprachkonstrukt eingeführt werden:

```
fork { Anweisung1; Anweisung2; ... }
```

Hierbei sind `Anweisung1; Anweisung2; ...` beliebige an solcher Stelle erlaubte Anweisungen. Das Konstrukt erzeugt einen neuen Prozeß, der die eingeschlossenen Anweisungen ausführt, während das Programm hinter der schließenden Klammer bereits fortgesetzt wird.

Betrachten Sie nun folgendes Programmfragment, in dem das neue Sprachkonstrukt erstmalig verwendet wird:

```
void printSquare(int i) {
  System.out.println(i + " " + i*i);
}

void tabulate (Iterator<Integer> it) {
  int x;
  while (it.hasNext()) {
    x = it.next();
    fork { printSquare(x); }
  }
```

```
}

...

// mylist = { 1,2,3,4 }
tabulate(mylist.listIterator());
System.out.println("fertig!");
```

a) (2P) Das Ergebnis ist nicht ganz so wie erwartet. Unter anderem wird folgende Ausgabe beobachtet:

```
1 1
3 9
3 9
4 16
fertig!
```

Warum kann so etwas zustande kommen?

b) (2P) Damit nicht genug, gelegentlich werden auch Ausgaben mit völlig unerwarteten Werten beobachtet, wie zum Beispiel:

```
...
-2135137970 260846532
...
```

Was ist die Ursache dafür?

c) (2P) Aufgrund der vielen Probleme wird eine alternative Version für die Gabelungsanweisung vorgeschlagen:

```
FORK P ( p1, p2, ...);
```

Hierbei ist `P` der Name einer Prozedur und `p1, p2, ...` sind gültige aktuelle Parameter für diese Prozedur. Die Anweisung wird so ausgeführt, daß zunächst die Parameterübergabe stattfindet. Nach der Parameterübergabe wird die Prozedur asynchron, d.h. in einem separaten Prozeß ausgeführt, während das Programm hinter der Anweisung fortgesetzt wird.
Löst dieser Ansatz die in a) und b) genannten Probleme? Begründen Sie Ihre Antwort.

# Aufgabe 3 (4 P)

Die folgende Klasse realisiert Objektlisten als einfach verkettete Listen; Die Eintragung eines Objekts erfolgt wie bei einer push-Operation auf einem Keller:

```
class ObjectList<E> {
        private Cell<E> head;
        private static class Cell<E> {
                E head;
                Cell<E> tail;
        }
        public void add(E x) {
                Cell<E> cell = new Cell<E>();
                cell.head = x;
                cell.tail = head;
                head = cell;
        }
        ...
}
```

Zwei nebenläufige Prozesse erzeugen Objekte x, y und wollen sie in einer Objektliste objects ablegen:

```
ObjectList<E> objects = new ObjectList<E>();
co ... objects.add(x); .......
|| ....... objects.add(y); ...
oc
.....
```

Das geht nicht immer gut! Was kann passieren? (Bitte mit Zeitschnitten verdeutlichen!)

# Aufgabe 4 (4 P)

Wenn zwei Prozesse asynchron gemeinsame Variablen verändern, kann das zu unerwünschten und damit inkorrekten Ergebnissen führen. Abschnitte, die auf in dieser Hinsicht kritische gemeinsame Variablen zugreifen, werden daher *kritische Abschnitte* genannt. Ein solches fehlerhaftes Verhalten lässt sich verhindern, indem man gewährleistet, dass kritische Abschnitte in *gegenseitigem Ausschluss* ausgeführt werden. Dies bedeutet, dass sicherzustellen ist, dass sich immer höchstens ein Prozess in einem kritischen Abschnitt befinden kann.

Sobald sich also ein Prozess A in einem kritischen Abschnitt befindet, müssen alle anderen Prozesse, die ebenfalls einen kritischen Abschnitt betreten wollen, warten, bis A seinen Abschnitt verlässt. Im folgenden werden zwei Ansätze zur Gewährleistung des gegenseitigen Ausschluss vorgestellt, die *aktives Warten* ("busy waiting") einsetzen. *Aktives Warten* wird hier nur zu Lernzwecken eingesetzt, ist aber in der realen Praxis ein sehr schlechter Programmierstil und ineffizient!
Überprüfen Sie für jeden Ansatz, ob dieser korrekt ist!
Korrektheit ist hier als Gewährleistung des gegenseitigen Ausschlusses und Lebendigkeit definiert. Geben Sie bei inkorrekten Ansätzen fehlerhafte Beispiele unter Einsatz von Zeitschnitten an.

**Ansatz A:** (2 P)

```
static volatile boolean crit = false;

// Prozess 1                  // Prozess 2
while(true) {                 while(true) {
 while (crit);                 while (crit);
 crit = true;                  crit = true;
 do_critical_work();           do_critical_work();
 crit = false;                 crit = false;
}                             }
```

**Ansatz B:** (2 P)

```
static volatile boolean crit1 = false;
static volatile boolean crit2 = false;

// Prozess 1                  // Prozess 2
while(true) {                 while(true) {
 crit1 = true;                 crit2 = true;
 while (crit2);                while (crit1);
 do_critical_work();           do_critical_work();
 crit1 = false;                crit2 = false;
}                             }
```

**Hinweise zur Abgabe:**

Die Abgabe der Lösungen erfolgt auf zwei Wegen:

- Abgabe auf Papier:

  Bitte schreiben Sie zur jeder Aufgabe eine Dokumentation, in der Sie anhand von *relevanten* Codefragmenten Ihre Implementierung erläutern. Die Codefragmente sollten möglichst kurz gehalten werden und nur der Orientierung für den Leser dienen; entscheidend sind Ihre Erläuterungen, was diese Fragmente machen und wieso Sie sich für diese Art der Implementierung entschieden haben. Die Dokumentation ist *bis Mittwoch 16:00 Uhr in den Tutorenfächern* abzugeben .

- Abgabe per E-Mail:

  Schicken Sie bei Programmieraufgaben ein ZIP-Archiv an Ihren Tutor, welches den vollständigen Quellcode sowie ausführbare JAR-Dateien für die jeweiligen Aufgabenteile enthält. Der Betreff der E-Mail sollte wie folgt aussehen: "[ALP4] Blatt X - Namen Y".

Zum Bestehen des Übungsblattes müssen 50% der Punkte erreicht werden.