

# Algorithmen und Programmierung IV

SS 2006

29. Mai 2006

## Musterlösung zum Aufgabenblatt 4

### Aufgabe 1

- a) Bei der `send`-Methode findet die Überprüfung der Anzahl der Elemente des Puffers vor dem `synchronized`-Block statt, so dass mehrere Aufrufe dieser Methode diese Überprüfung passieren können, auch wenn nur noch ein Element eingefügt werden kann.

```
// Annahme: count() == size - 1
-----
if (count() == size) throw new Overflow();
-----
                if (count() == size) throw new Overflow();
                cell[rear] = m;
                rear = (rear + 1) % (size + 1);
                // Nun gilt: count() == size
-----
cell[rear] = m;
rear = (rear + 1) % (size + 1);
// Jetzt aber: count() == (size + 1) % (size + 1) == 0 (!)
```

- b) Die Variable `m`, deren Wert zurückgegeben werden soll, wurde innerhalb des `synchronized`-Block deklariert und ist damit auch nur dort bekannt. Der Fehler kann behoben werden, indem `m` vor dem `synchronized`-Block vereinbart wird.

```
public M recv() throws Unterflow {
    M m;
    synchronized (cell) { // receiver exclusion
        if (count() == 0) throw new Unterflow();
        m = cell[front];
        front = (front + 1) % (size + 1);
    }
    return m;
}
```

- c) Die Methode `urgent` legt eine dringende Nachricht vorne im Puffer ab. Als Grundlage für `send` und `recv` werden die Methoden aus Abschnitt 3.1.2 von Foliennummer 27 verwendet.

```
public void urgent(M m) throws Overflow {
    synchronized (this) { // send exclusion
        if (count() == size) throw new Overflow();
        synchronized (cell) { // "receiver" exclusion
            front = (front + size) % (size + 1); // d.h. "front - 1" ohne Unterlauf
            cell[front] = m;
        }
    }
}
```

Erklärung: Die Ausschlusssynchronisation mit dem Bezug `this` ist notwendig, um dem Szenario aus Aufgabenteil a) vorzubeugen. Dadurch wird verhindert, dass durch parallele Ausführung von `send` und `urgent` die Kapazität des Puffers überschritten wird. Die Ausschlusssynchronisation mit dem Bezug `cell` ist notwendig, um unerwünschte Effekte in Kombination mit dem Auslesen und Schreiben von `front` in der Methode `recv` zu verhindern. Das heißt, durch diesen Ausschluss verhalten sich die Zuweisungen an `front` in `recv` und `urgent`, als wären sie unteilbar.

## Aufgabe 2

Bei jedem Aufruf der Methode `incr` werden die Variablen `i` und `n` sowohl gelesen, als auch geschrieben. Um unerwarteten Effekten in Bezug auf `i` vorzubeugen, würde sich eine Ausschlusssynchronisation mit dem Bezug `this` anbieten, da jedes Exemplar von `Funny` ein eigenes Attribut `i` besitzt.

Da aber das Attribut `n` statisch ist, wird als Bezug ein Objekt benötigt, das alle Exemplare von `Funny` gemeinsam haben. Dafür verwendet man gewöhnlich das zu einer Klasse gehörige Klassenobjekt; in diesem Fall `Funny.class`.

```
void incr() {
    synchronized (Funny.class) {
        n += i++;
    }
}
```

Eine zusätzliche Synchronisation auf `this` ist nicht notwendig, weil durch die Synchronisation auf `Funny.class` bereits der Eintritt mehrere Fäden in den kritischen Block jeglicher Aufrufe von `incr` ausgeschlossen ist.

### Aufgabe 3

Die Technik zum Nachrüsten des Ausschlusses in der Klasse `Collections` ist Delegation. Es wird mit Hilfe statischer Methoden ein synchronisierter Adapter für eine Datenstruktur erzeugt, der Methodenaufrufe an die eigentliche Datenstruktur weiterleitet und dabei den wechselseitigen Ausschluss (der meisten) seiner Methodenaufrufe garantiert.

Ein Vorteil dieser Lösung ist, dass beliebige Datenstrukturen, welche die Schnittstellen des `Collection`-Frameworks implementieren, mit Ausschlussynchronisation nachgerüstet werden können, denn der Adapter erzeugt das Exemplar der darunterliegenden Datenstruktur nicht selber, sondern verwendet ein ihm übergebenes Exemplar.

Ein Nachteil dieser Lösung besteht darin, dass die Ausschlussynchronisation (u.U. versehentlich) umgangen werden kann, wenn der Zugriff auf die Datenstruktur nicht über den Adapter, sondern direkt über die dem Adapter übergebene Datenstruktur erfolgt (Anm.: Bei geschickter Anwendung kann dies jedoch wieder ein Vorteil sein; siehe Aufgabe 4).

Sollte eine Datenstruktur bereits synchronisiert gewesen sein, führt die zusätzliche Synchronisierung – abgesehen von einer gewissen Effizienzminderung – zu keinen Problemen. Dadurch dass immer zuerst der kritische Bereich des Adapter und dann derjenige der Datenstruktur betreten wird, kann keine Verklemmung eintreten.

## Aufgabe 4

Laut Spezifikation hat der Benutzer bei einer Traversierung eines durch die Klasse `Collections` generierten Adapters selbst dafür Sorge zu tragen, dass sowohl die Erzeugung als auch die Verwendung eines Iterators synchronisiert wird. Das liegt daran, dass einerseits der Adapter *nicht* vollständig thread-safe ist und dass über den Iterator ein *nicht* synchronisierter Zugriff auf die Datenstruktur möglich ist, so dass eine nebenläufige Modifikation zu unerwünschten Effekten führen kann. Folgerichtig finden wir in der Dokumentation von `Iterator` diese Passage (unter `remove`):

„The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.“

Der Iterator „merkt sich“ die jeweils nächste Position in der zu traversierenden Datenstruktur, und diese Positionsinformation ist nach einer nebenläufigen Veränderung der Menge möglicherweise nicht mehr korrekt. (Vergleiche dies mit den Erläuterungen zum internen Iterator in der Vorlesung, 3.1.7, Seite 22.)

Die von der Klasse `Collections` generierten Adapter hätten die Möglichkeit, synchronisierte Iteratoren zu erzeugen, deren Verwendung vom Benutzer hätte nicht synchronisiert werden müssen. Es sind folgende Gründe denkbar, die gegen synchronisierte Iteratoren sprechen:

- Eine Synchronisierung der Methoden `hasNext` und `next` hätte zu Folge, dass für jedes zu traversierende Element der Datenstruktur zweimal ein kritischer Bereich betreten und wieder verlassen werden müsste, was mit unnötigen Effizienzeinbußen verbunden wären.
- Die Synchronisierung der Methoden `hasNext` und `next` könnte nicht verhindern, dass die Datenstruktur zwischen diesen beiden Methodenaufrufen verändert wird. Beispielsweise könnte das letzte Element aus der Datenstruktur gelöscht werden, nachdem `hasNext` die Existenz eines weiteren Elements bestätigt hat, aber bevor es mit `next` abgerufen worden ist.

Anm.: Der zweite Fall ist natürlich auch bei sequenzieller Programmausführung denkbar, jedoch müsste man einen solchen Fall böswillig konstruieren. Bei nicht-sequenzieller Ausführung kann ein solcher Fall sehr einfach eintreten.