Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Dipl.-Inf. K. Stahl

# Nebenläufige Programmierung

Sommersemester 2003 — **Serie MST** — 21. Mai 2003

**Diese Serie soll in Einzelarbeit gelöst werden!**

## Aufgabe 1 (4 Punkte)

*n-Process Barrier.* Man kann eine wiederbenutzbare Barriere für $n$ Prozesse unter Benutzung von zwei Semaphoren und einer Zählvariable programmieren. Entwickle solch eine Lösung (Tip: Benutze die Idee "passing the baton").

## Aufgabe 2 (4 Punkte)

*Atomic Broadcast.* Gegeben sei ein Producer und $n$ Consumer die auf einen Buffer zugreifen. Der Producer schreibt Nachrichten in den Buffer, die Consumer liest sie. Jede Nachricht soll von allen $n$ Consumer Prozessen gelesen werden, bevor der Producer eine neue Nachricht in den Buffer schreiben darf.

(a) Entwickle eine Lösung für dieses Problem mit Semaphoren zur Synchronisation.

(b) Nimm an dass der Buffer $b$ Plätze hat. Der Producer kann nur in freie Plätze schreiben, und jede Nachricht muss von allen $n$ Consumer Prozessen gelesen sein, bevor der Platz wiederverwendet werden darf. Verschiedene Consumer dürfen aber unterschiedlich viele Nachrichten gelesen haben. Löse dieses generellere Problem.

## Aufgabe 3 (4 Punkte)

*Search/Insert/Delete.* Gegeben seien drei Arten von Prozessen die Zugriff haben auf eine gemeinsame verkettete Liste: Searchers, Inserters sowie Deleters. Searchers sind reine Leseprozesse die nebenläufig arbeiten dürfen. Inserters fügen neue Daten am Ende der Liste an; dies darf immer nur ein Prozess zur Zeit tun, allerdings können parallel beliebig viele Searchers arbeiten. Deleters löschen Daten an beliebigen Stellen der Liste, so dass immer nur einer pro Zeit aktiv sein darf, und zudem keine Searchers oder Inserters gleichzeitig arbeiten dürfen.
Entwickle eine Monitorlösung für diese Art von Synchronisation. Spezifiziere zunächst eine Monitorinvariante. Benutze Signal and Continue.

## Aufgabe 4 (4 Punkte)

*Memory Allocation.* Gegeben seien zwei Operationen zur Speicherverwaltung namens `request(amount)` und `release(amount)`, wobei `amount` ein positiver Integerwert ist. Wenn ein Prozess `request` aufruft, wird er verzögert bis genug freier Speicher verfügbar ist. Ein Prozess gibt Speicher wieder frei durch den Aufruf von `release`, wobei ein Prozess auch nur kleinere Teile des angeforderten Speichers freigeben kann.

(a) Entwickle einen Monitor der `request` und `release` implementiert. Spezifiziere zunächst eine globale Invariante. Kümmere Dich nicht um die Reihenfolge in der die Anforderungen bearbeitet werden. Benutze Signal and Continue.

(b) Modifiziere die Lösung zu (a) derart, dass die shortest-job-next Strategie (SJN) verfolgt wird, also kleinere Anforderungen Vorrang vor größeren Speicheranforderungen bekommen.

(c) Modifiziere die Lösung zu (a) derart, dass eine first-come, first-served Strategie (FCFS) verfolgt wird.

(d) Nimm an dass `request` und `release` nur zusammenhängenden Speicher bekommen bzw. zurückgeben. Beispielsweise wartet also ein Prozess der zwei Seiten anfordert bis zwei benachbarte Seiten verfügbar sind. Gib eine Monitorlösung an die diese Version von `request` und `release` implementiert. Wähle zunächst eine geeignete Repräsentierung für den Status der Speicherseiten und spezifiziere eine Monitorinvariante.

**Ausgabe:** Mittwoch, 21. Mai 2003
**Abgabe:** Mittwoch, 28. Mai 2003