Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion
CAU Kiel, Informatik und Prakt. Mathematik
Prof. Dr. M. Hanus, Dr. Frank Huch

24118 Kiel
Christian-Albrechts-Platz 4
Tel.: 0431 / 880-7277

12. Übung „Prinzipien von Programmiersprachen“

Bearbeitung bis zum 8. Februar 2004

Aufgabe 1 (2 Punkte)
In dieser Aufgabe wollen wir die Semantik nebenläufiger Programme formalisieren. Hierzu betrachten wir eine kleine nebenläufige Programmiersprache (KNPS), deren Anweisungen wie folgt definiert sind:

$$\begin{array}{lll} \mathit{Stm} & ::= & \mathit{Var} := \mathit{Exp} \mid \texttt{fork}(\mathit{label}) \mid \mathit{label} : \mathit{Stm} \mid \texttt{lock}(\mathit{Var}) \mid \\ & & \texttt{unlock}(\mathit{Var}) \mid \texttt{print}(\mathit{Exp}) \mid \texttt{if } \mathit{Exp} \texttt{ then } \mathit{Stm} \mid \\ & & \texttt{while } \mathit{Exp} \texttt{ do } \mathit{Stm} \mid \texttt{begin } \mathit{StmList} \texttt{ end} \\ \mathit{StmList} & ::= & \mathit{Stm}; \mathit{StmList} \mid \mathit{Stm} \end{array}$$

In KNPS können mit Hilfe der `fork`-Anweisung neue Prozesse gestartet werden. Der neue Prozess beginnt seine Ausführung mit der durch *label* (eine beliebige Zeichenfolge) markierten Anweisung. Alle Variablen seien global bekannt und werden zwischen den Prozessen geteilt. Die Anweisungen `lock` und `unlock` verhalten sich wie in der Vorlesung beschrieben.

a) Programmieren Sie die in der Vorlesung vorgestellten dinierenden Philosophen in KNPS. Überlegen Sie insbesondere, wie die Philosophen gestartet werden können.

b) Modifizieren Sie Ihr Programm so, dass maximal drei Philosophen gleichzeitig essen dürfen. Ist das entstehende Programm Deadlockfrei? Sind noch Livelocks möglich? Ist Fairness gewährleistet?

c) Nun wollen wir eine formale Semantik in Form einer small-step Interleaving-Semantik definieren. Hierzu müssen wir zunächst den Zustandsraum definieren. Ein Programmzustand (*KNPS-State*) besteht aus einer Menge von Prozessen (*StmList*), einer Speicherbelegung der verwendeten Variablen und einer Lock-Belegung für jede Variable. Formalisieren Sie die Programmzustände. Überlegen Sie sich hierzu insbesondere eine elegante Darstellung der Lock-Belegung. Wie sieht der Startzustand (q_0) für eine gegebenes KNPS-Programm p aus?

d) Definieren Sie die Einzelschrittsemantik

$$\Rightarrow \subseteq \mathit{KNPS\text{-}State} \times \mathit{KNPS\text{-}State}$$

Die Semantik ist für Zustände mit mehreren Prozessen nicht-deterministisch, da bei der Interleaving-Semantik davon ausgegangen wird, dass die Prozesse Ihre Anweisungen verzahnt ausführen. Interessant wird die Modellierung der

Anweisungen `lock` und `unlock`. Achten Sie darauf, dass Ihre Modellierung blockierte Prozesse wirklich warten lässt. Ein Deadlock entspricht dann einem Zustand ohne Nachfolger. Gehen Sie davon aus, dass die Berechnung des R-Wertes eines Ausdrucks (*Expr*) atomar erfolgt.

e) Zeigen Sie, dass die von Ihnen definierte Semantik nicht konfluent ist, d.h. es gibt Programme mit folgendem Verhalten:

$$q_0 \Rightarrow^* q_1 \not\Rightarrow \text{ und } q_0 \Rightarrow^* q_2 \not\Rightarrow \text{ mit } q_1 \neq q_2$$

f) Geben Sie für Ihr Philosophen-Programm (mit nur drei Philosophen) einen Lauf Ihrer Semantik in einen Deadlock an.

g) Ist Ihre Behandlung von Ausdrücken realistisch? Wie müsste diese abgeändert werden um das Verhalten wirklicher nebenläufiger Implementierungen zu modellieren?

Aufgabe 2
In dieser (und der nächsten) Aufgabe sollen Sie ein paar kleine nebenläufige Java-Programme schreiben.

a) Implementieren Sie eine Klasse `Semaphore` in Java.

b) Zeigen Sie mit Hilfe eines Beispielprogramms, dass sich auch nebenläufige Java-Programme nicht-deterministisch verhalten können, sprich bei zwei Läufen unterschiedliche Ergebnisse berechnet (bzw. ausgegeben) werden können. In der Praxis ist es manchmal schwierig, unterschiedliche Schedulings zu erzeugen. Deshalb dürfen Sie hierzu mit der `Thread`-Methode `sleep(log millisec)` das Scheduling Ihres Programms beeinflussen.

Zusatzaufgabe 3 (1 Sonderpunkt)
Implementieren Sie einen einelementigen Puffer mit den Methoden `put` und `take`. Threads, die ein `put` (bzw. `take`) auf einen vollen (bzw. leeren) Puffer ausführen, sollen suspendieren, bis ein anderer Prozess ein `take` (bzw. `put`) auf dem Puffer ausführt. Als Werte sollen in dem Puffer beliebige Objekte gespeichert werden können.
Können Sie in ihrer Implementierung "busy-waiting" der Form "`while` *Bed.* `wait();`" (siehe Puffer-Implementierung der Vorlesung) verhindern und lesende bzw. schreibende Threads gezielt aufwecken?