Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Prof. Dr. W.-P. de Roever
Jens Schönborn und Jan Waller

Nebenläufige und verteilte Programmierung

Wintersemester 2006/07 27.11.2006

Serie 7
Mittsemestertest

Ausgabetermin: 27.11.2006

Abgabe: 7.12. 2006 (12:00)

Diese Serie soll in Einzelarbeit gelöst werden!

Jede Aufgabe soll auf einem neuen Zettel begonnen werden. Verseht bitte alle Zettel die ihr abgebt mit eurem Namen.

Aufgabe 1 (6 Punkte) *Mutual exclusion.* Angenommen ein Computer hat einen *atomaren swap* Befehl. Dieser sei wie folgt definiert:
swap(var1, var2):<tmp = var1; var1 = var2; var2 = tmp; >
tmp ist ein internes Register. *swap* darf zur Lösung nicht geändert werden.

1. Benutze swap um eine Lösung für das critical section Problem für n Prozesse zu entwickeln. Die eventual entry Eigenschaft braucht nicht berücksichtigt werden. Beschreibe gründlich wie deine Lösung funktioniert und warum sie korrekt ist.

2. Modifiziere deine Antwort zu 1 so, daß die Lösung schnell auf einem Multiprocessorsystem mit shared memory und caches läuft. Erkläre welche und warum du Änderungen vorgenommen hast.

3. Gebe unter Verwendung des *swap* Befehls eine Lösung für das critical section Problem an. Sie soll eventual entry Eigenschaft bei weakly fair scheduling gewährleisten. Beachte, daß du nicht deine Lösung aus 1 verwenden kannst um die atomare Aktion im Ticket Algorithmus zu implementieren, da man, bezogen auf das scheduling, unter Verwendung von unfairen Komponenten keine faire Lösung erstellen kann. Du darfst annehmen, daß jeder Prozess eine eindeutige Identität, zum Beispiel Integers von 1 bis n hat. Erkläre deine Lösung, und argumentiere überzeugend warum sie korrekt ist und die eventual entry Eigenschaft bei weakly fair scheduling gewährleistet.

Aufgabe 2 (4 Punkte) *n-Process Barrier.* Man kann eine wiederbenutzbare Barriere für n Prozesse unter Benutzung von zwei Semaphoren und einer Zählvariable programmieren. Entwickle eine solche Lösung (Tip: Benutze die Idee "passing the baton").

Aufgabe 3 (10 Punkte) *Bakery Algorithmus.*

1. Zeige, dass die `turn[i]` Variable im Bakery Algorithmus (S. 111ff) unbeschränkt hohe Werte annehmen kann.

2. Modifiziere den Bakery Algorithmus (coarse grained, Fig. 3.10 auf S. 112) derart, dass die `turn[i]` Werte begrenzt werden.
   Hinweis: Die atomare Anweisung welche `turn[i]` setzt, darf nicht geändert werden, jedoch das await-Statement. Neue Variablen, z.B. ein `next`, dürfen eingeführt werden.

3. Modifiziere ebenfalls die fine-grained Variante von Fig. 3.11 auf S. 114.
   Hinweis: Es darf hier kein Fetch-and-Add verwendet werden. Die Aufgabe ist schwer. Man kann versuchen, beim Überschreiten der Grenze wieder von vorne anzufangen. Wähle hierfür eine passende der Grenze in Abhängigkeit von der Anzahl der Prozesse. Mache dann eine Fallunterscheidung darüber, in welchem Bereich die `turn` Werte der Prozesse liegen können. Und zuletzt eleminiere weiteres Fehlverhalten.