Klausur „Nichtsequentielle Programmierung“

Prof. Dr. Marcel Kyas

22. Juli 2009

Freie Universität Berlin

Nachname:		Bachelor ☐	Magister ☐
Vorname:		Master ☐	Lehramt ☐
Matr.Nr.:		Diplom ☐	☐

Hinweise zur Klausur

- Bitte überprüfen Sie, dass Sie alle Seiten dieser Klausuraufgaben erhalten haben, bevor Sie die Aufgaben bearbeiten.
- Lesen Sie jede Aufgabe vollständig und aufmerksam durch, bevor Sie sie bearbeiten.
- Schreiben Sie bitte Ihren *Namen* und Ihre *Matrikelnummer* auf jedes Blatt.
- Geben Sie bitte alle Blätter ab. Nummerieren Sie diese.
- Schreiben Sie Ihre Lösung entweder in die dafür vorgesehenen Felder auf den Aufgabenblättern oder auf den von der Klausuraufsicht ausgegebenen Blätter. Die Klausuraufsicht wird Ihnen auf Anfrage neue Blätter geben. *Das Benutzen von selbst mitgebrachten Blättern wird als Täuschungsversuch gewertet!*
- Es sind keine Hilfmittel zugelassen. Insbesondere dürfen Sie keine Bücher, Rechner, Notebooks oder Mitschriften benutzen. *Mobiltelefone* müssen ausgeschaltet und in der Tasche verstaut werden.
- Benutzen Sie bitte nur *nicht-radierbare* Schreiber, die *blau* oder *schwarz* schreiben. Klausuren, die mit Bleistift geschrieben wurden, werden nicht angenommen.
- Halten Sie bitte Ihren Studierendenausweis und einen Lichtbildausweis bereit.
- Die Klausur dauert 90 Minuten.
- Diese Klausur hat *drei* Aufgabenteile. Sie müssen aus dem Teil I jede Aufgabe bearbeiten und aus den Teilen II und III je eine Aufgabe. Sollten Sie beide Aufgaben bearbeiten, streichen Sie eine durch, sonst wird nur die schlechtere Lösung gewertet. Zum Bestehen der Klausur brauchen Sie mindestens 45 von 90 erreichbaren Punkten.

Wertung (wird vom Institut ausgefüllt)

Teil	I				II		III		Σ	Note
Aufgabe	1	2	3	4	5	6	7	8		
Punktzahl	3	10	10	12	20		35		90	
Erreichte Punkte										

Aufgabe 1

3 Punkte

1. Bitte kreuzen Sie nur die wahre Aussage an:

   ☐ Ein Algorithmus zum gegenseitigen Ausschluss heißt *sicher,* wenn zu jedem Zeitpunkt mindestens ein *Thread* in der kritischen Sektion ist.

   ☐ Ein Algorithmus zum gegenseitigen Ausschluss heißt *sicher,* wenn zu jedem Zeitpunkt höchstens ein *Thread* in der kritischen Sektion ist.

   ☐ Ein Algorithmus zum gegenseitigen Ausschluss heißt *sicher,* wenn jeder *Thread,* der Zugang zur kritischen Sektion anfragt, diesen auch schließlich bekommt.

2. Bitte kreuzen Sie nur die wahre Aussage an:

   ☐ Ein Algorithmus zum gegenseitigen Ausschluss heißt *fair (engl. starvation-free),* wenn von mehreren wartenden Threads immer ein *Thread* schließlich Zugang zur kritischen Sektion bekommt.

   ☐ Ein Algorithmus zum gegenseitigen Ausschluss heißt *fair (engl. starvation-free),* wenn jeder *Thread,* der Zugang zur kritischen Sektion anfordert, diesen auch schließlich bekommt.

   ☐ Ein Algorithmus zum gegenseitigen Ausschluss heißt *fair (engl. starvation-free),* wenn zu jedem Zeitpunkt höchstens ein *Thread* in der kritischen Sektion ist.

3. Bitte kreuzen Sie nur die wahre Aussage an:

   ☐ Ein Algorithmus zum gegenseitigen Ausschluss heißt *verklemmungsfrei (engl. deadlock-free),* wenn von mehreren wartenden Threads immer ein *Thread* schließlich Zugang zur kritischen Sektion bekommen kann.

   ☐ Ein Algorithmus zum gegenseitigen Ausschluss heißt *verklemmungsfrei (engl. deadlock-free),* wenn zu jedem Zeitpunkt mindestens ein *Thread* in der kritischen Sektion ist.

   ☐ Ein Algorithmus zum gegenseitigen Ausschluss heißt *verklemmungsfrei (engl. deadlock-free),* wenn jeder *Thread,* der Zugang zur kritischen Sektion anfragt, diesen auch schließlich bekommen kann.

10 Punkte

Aufgabe 2 Betrachten Sie die folgenden Anforderungen an die gemeinsamen Semaphore s, t, und u von den Prozessen P, Q und R.

Zeit	P	Q	R
0	P(s)		
1	P(t)		P(u)
2		P(s)	
3			P(t)
4	V(s)		
5	P(u)		

1. Zeichnen Sie den *Anforderungsgraphen,* der am Ende dieses Laufes besteht.
2. Kommt es am Ende dieser Anforderungen zu einer Verklemmung? Begründen Sie Ihre Antwort.

10 Punkte

Aufgabe 3 Betrachten Sie das folgende Programm:

```
int x = 0;
cobegin
  x = 2x;
  x = 2x;
||
  x = 1;
  x = x + 1;
coend
```

Nehmen Sie an, dass das Lesen und das Schreiben auf eine Variable atomar ist, und dass `x` vor dem Inkrementieren in ein lokales Register geladen werden muss. Bearbeiten Sie die folgenden Aufgaben:

1. Schreiben Sie das Programm so um, dass es die *at-most-once* Eigenschaft erfüllt.
2. Zählen Sie alle möglichen Werte für x auf, nachdem das Programm terminiert ist.

12 Punkte

Aufgabe 4 Gegeben sei folgendes Programm:

```
int x = 0;
boolean c = true;

cobegin
  ⟨ await x >= 9; ⟩; c = false;
‖ while (c) ⟨x = x + 1⟩;
coend
```

1. Terminiert das Programm immer unter der Annahme, dass die Ablaufsteuerung *weakly fair* ist? Begründen Sie Ihre Antwort.
2. Terminiert das Programm immer unter der Annahme, dass die Ablaufsteuerung *strongly fair* ist? Begründen Sie Ihre Antwort.

Betrachten Sie nun das neue Programm

```
int x = 0;
boolean c = true;

cobegin
  ⟨ await x >= 9; ⟩; c = false;
‖ while (c) ⟨x = (x + 1) % 10⟩;
coend
```

Beantworten Sie jetzt die beiden Fragen zur Terminierung für dieses veränderte Programm.

Sie brauchen die folgende Aufgabe *nicht* bearbeiten, wenn Sie Aufgabe 6 bearbeiten.

20 Punkte

Aufgabe 5 Man kann eine wiederverwendbare Barriere für n Prozesse unter Benutzung von zwei *binären* Semaphoren und einer Zählvariable programmieren, die so deklariert werden:

```
int count = 0;
semaphore arrive(1);
semaphore go(0);
```

Entwickeln Sie eine solche Lösung. Erklären Sie *kurz* wie Ihre Lösung funktioniert.
Tipp: Benutzen Sie die Idee aus *passing the baton* für Ihre Lösung. Unter Umständen müssen Sie Ihre Lösung dann optimieren. Sie dürfen annehmen, dass die Semaphore *first-come-first-serve* garantieren.

Sie brauchen die folgende Aufgabe *nicht* bearbeiten, wenn Sie Aufgabe 5 bearbeiten.

20 Punkte

Aufgabe 6 [Problem der Kettenraucher [Patil 1971, Parnas 1975]] Nehmen Sie an, dass *drei* Kettenraucher an einem Tisch sitzen. Zum Rauchen braucht jeder Raucher *drei* Zutaten:

1. Tabak,
2. Papier und
3. Streichhölzer.

Von diesen Zutaten hat jeder Raucher einen unendlichen Vorrat von einer Zutat: der erste Raucher hat Tabak, der zweite Raucher hat Papier und der dritte Raucher hat Streichhölzer. Zusätzlich sitzt ein Kellner am Tisch, der zufällig *zwei* verschiedene Zutaten auf den Tisch legt. Der Raucher mit der dritten Zutat nimmt dann die anderen beiden Zutaten, macht eine Zigarette und raucht Sie danach. Der Kellner wartet, bis der Raucher fertig ist. Dann wiederholt sich der Zyklus.

1. Modellieren Sie den Kellner und jeden Raucher als einen Prozess. Benutzen Sie Semaphore zur Synchronisation. Vergessen Sie nicht, Ihre Semaphore zu initialisieren. Vielleicht müssen Sie weitere Variablen einführen. Erklären Sie *kurz* wie Ihre Lösung funktioniert. Sollten Sie eine Zufallszahl brauchen, benutzen Sie die Funktion `random(n)`, die zufällig eine Zahl x mit $0 \leq x < n$ zurück gibt.
2. Ist Ihre Lösung verklemmungsfrei? Begründen Sie Ihre Antwort.

Sie brauchen die folgende Aufgabe *nicht* bearbeiten, wenn Sie Aufgabe 8 bearbeiten.

35 Punkte

Aufgabe 7 Autos aus dem Norden und dem Süden kommen an einer einspurigen Brücke an. Alle Autos, die aus einer Richtung kommen, können gemeinsam über die Brücke fahren. Autos, die in die andere Richtung fahren, müssen warten.

1. Entwickeln Sie eine Lösung für dieses Problem. Pseudocode reicht aus. Modellieren Sie Autos als Prozesse und benutzen Sie einen Monitor zur Synchronisierung. Spezifizieren Sie zuerst eine Invariante für den Monitor und entwickeln Sie dann den Rumpf des Monitors. Kümmern Sie sich nicht um *Fairness* und bevorzugen Sie keine Sorte von Autos. Benutzen Sie *signal-and-continue.* Nehmen Sie echte *condition variables* an und ignorieren Sie die Ausnahmebehandlung. Beschreiben Sie ganz kurz wie Ihre Lösung funktioniert.

2. Modifizieren Sie Ihre Lösung zu 1. so, dass Sie *fair* wird, d.h. jedes Auto, dass über die Brücke fahren möchte, fährt schließlich auch über die Brücke. *Hinweis:* Lassen Sie Autos abwechseln.

Sie brauchen die folgende Aufgabe *nicht* bearbeiten, wenn Sie Aufgabe 7 bearbeiten.

35 Punkte

Aufgabe 8 Schreiben Sie einen Monitor, der es einem Paar von Prozessen erlaubt, Werte auszutauschen. Dieser Monitor hat genau eine Operation: `exchange(int& value)` (d.h., `value` wird als *Referenzparameter* übergeben). Nachdem zwei Prozesse die Operation `exchange(...)` aufgerufen haben, tauscht der Monitor die beiden Werte aus und gibt sie den aufrufenden Prozessen zurück. Der Monitor soll wiederverwendbar sein, d.h. er tauscht erst die Werte der ersten zwei Aufrufenden, dann die der zweiten Aufrufenden und so weiter.

Sie dürfen entweder *signal-and-continue* oder *signal-and-wait* benutzen, aber schreiben Sie auf, welche Disziplin Sie benutzen.

Beschreiben Sie mit wenigen Worten wie Ihre Lösung funktioniert.