Klausur „Nichtsequentielle Programmierung“

Prof. Dr. Marcel Kyas

5. Oktober 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 die 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	10	5	15	15	18		27		90	
Erreichte Punkte										

Aufgabe 1 Sind die folgenden Definitionen zu den Definitionen aus der Vorlesung äquivalent?

10 Punkte

1. Ein Algorithmus heißt genau dann *verklemmungsfrei (deadlock-free)*, wenn, falls ein Thread niemals seinen Eintrittscode verlässt, es eine unendliche Folge von kritischen Sektionen, die von anderen Threads ausgeführt werden, gibt.

2. Ein Algorithmus heißt genau dann *fair (starvation-freedom)* wenn jede Ausführung des Eintrittcodes terminiert.

Begründen Sie Ihre Antworten.

5 Punkte

Aufgabe 2 Impliziert Fairness (*starvation-freedom*) die Abwesenheit von Verklemmungen (*deadlocks-freedom)*? Begründen Sie Ihre Antwort.

15 Punkte

Aufgabe 3 Der inkorrekte Algorithmus von Harris Hyman in Abbildung 1 soll gegenseitigem Ausschluss sicher stellen. Er wurde in der Januar-Ausgabe 1966 der „Communication of the ACM“ veröffentlicht. Erklären Sie, warum der Algorithmus fehlerhaft ist.

Der Algorithmus gilt für zwei Prozesse; seien $i \in \{0, 1\}$ deren Identitäten. Er benutzt drei gemeinsame Variablen `turn`, `flag[0]` und `flag[1]`. Initial gelte `flag[0]=0` und `flag[1]=0`. Der Wert von `turn` ist Anfangs 0 oder 1. Die Anweisung `await c` kann wie immer mit `while (!c) yield();` implementiert werden.

```
process P[i = 0,1] {
  for (;;) {
    // Remainder
    flag[i] = 1;
    while (turn == 1 - i) {
      await flag[1-i] == 0;
      turn = i;
    }
    // Critical section
    flag[i] = 0;
  }
}
```

Abbildung 1: Hyman's Algortithmus

Aufgabe 4 Nehmen Sie an, es gäbe die Semaphoroperationen

15 Punkte

$$P(S_1, S_2) \triangleq \langle await\ (S_1 > 0 \wedge S_2 > 0)\ S_1 = S_1 - 1; S_2 = S_2 - 1; \rangle$$
$$V(S_1, S_2) \triangleq \langle S_1 = S_1 + 1; S_2 = S_2 + 1; \rangle \quad .$$

Die Operationen manipulieren also zwei Semaphore gleichzeitig und atomar. Betrachten Sie nun die nachfolgende Lösung des Philosophenproblems:

```
semaphore fork[0 to 4] = {1, 1, 1, 1, 1};

process Philosopher[i = 0..4] {
  for (;;) {
    // Think
    P(fork[i], fork[(i+1)%5]);
    // Eat
    V(fork[i], fork[(i+1)%5]);
  }
}
```

1. Erklären Sie kurz, warum diese Lösung frei von Verklemmung (*deadlocks*) ist.
2. Ist diese Lösung fair (*starvation-free*), wenn wir starke Fairness annehmen? Begründen Sie Ihre Antwort.

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

18 Punkte

Aufgabe 5 In einer Bar arbeitet ein Barkeeper, der in ein Glas zwei Portionen eines Getränks einfüllt, wartet, bis ein Kunde austrinkt und dann ein neues Glas befüllt. Dem Barkeeper stehen zwei Getränke zur Verfügung: Orangensaft und Vodka. Er wählt die Portion, die er ausschänkt, zufällig. In etwa führt der Barkeeper also folgenden Code aus:

```
enum { empty, vodka, orange } glas[2]; // Shared variable.

process Barkeeper {
  for (;;) {
    // some coordination code
    glas[0] = vodka or orange;
    glas[1] = vodka or orange;
    // some coordination code
  }
}
```

An der Bar stehen drei Kunden, von denen der Erste nur Vodka (zwei Portionen Vodka), der Zweite Screwdriver (eine Portion Vodka und eine Portion Orangensaft) und der Dritte nur Orangensaft (zwei Portionen Orangensaft) trinkt.

1. Modellieren Sie den Barkeeper und die Kunden als Prozesse. Benutzen Sie ausschließlich Semaphore, um die Prozesse zu koordinieren.

2. Erklären Sie, warum Ihre Lösung korrekt ist, d.h. warum kein Kunde ein anderes als sein bevorzugtes Getränk trinkt und niemand das Glas zu früh, also mit nur einer Portion, leert.

3. Ist Ihre Lösung verklemmungsfrei? Begründen Sie Ihre Antwort.

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

18 Punkte

Aufgabe 6 Implementieren Sie eine Prozedur `exchange(int& value)`, welche von zwei Prozessen zum Tauschen von Werten aufgerufen wird. Die Notation `int&` bedeutet hier, dass die Variable `value` als *Referenzparameter* übergeben wird. Der erste Prozess, der `exchange(value)` aufruft, muss warten. Wenn ein zweiter Prozess `exchange(value)` aufruft, werden die Werte ausgetauscht und beide Prozeduren terminieren. Die Prozedur soll wiederverwendbar sein, d.h. sie tauscht erst die Werte der ersten zwei Aufrufenden, dann die der zweiten Aufrufenden und so weiter.

Geben Sie die Implementierung dieser Prozedur an, die diese Spezifikation erfüllt. Benutzen Sie ausschließlich Semaphore zur Synchronisation. Stellen Sie sicher, dass Sie alle benutzen Variablen und Semaphore deklarieren und initialisieren sowie alle gemeinsam genutzen Variablen außerhalb der Prozedur deklarieren.

Beschreiben Sie mit wenigen Worten, wie Ihre Lösung funktioniert.

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

27 Punkte

Aufgabe 7 In einem Ferienlager gibt es einen Duschraum mit n Duschkabinen. Männer und Frauen wollen gleichzeitig duschen. Eine Person kann entweder eine Duschkabine benutzen, auf eine freie Duschkabine warten oder etwas anderes machen, z.B. Sport treiben. Eine Duschkabine kann höchstens von einer Person benutzt werden. Aus sittlichen Gründen dürfen aber nie ein Mann und eine Frau zur selben Zeit duschen.

1. Geben Sie eine Monitorinvariante an, die die Korrektheit Ihrer Lösung ausdrückt. Sie brauchen die Korrektheit nicht beweisen.

2. Schreiben Sie Pseudocode, der dieses Problem modelliert. Modellieren Sie Männer und Frauen durch Prozesse und den Duschraum mit seinen Duschkabinen durch einen Monitor. Kümmern Sie sich noch nicht um *fairness.* Geben Sie die Signaldisziplin Ihres Monitors an.

3. Erklären Sie kurz, warum Ihre Lösung sicher ist.

4. Ist Ihre Lösung frei von Verklemmungen? Begründen Sie Ihre Antwort knapp.

5. Modifizieren Sie Ihre Lösung zu Punkt 2 so, dass sie *fair* ist, also wenn ein Mann duschen möchte, schließlich auch ein Mann duscht und wenn eine Frau duschen möchte schließlich auch eine Frau duscht. Welche Annahme machen Sie über die Warteschlangen?

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

27 Punkte

Aufgabe 8 Nehmen Sie an, dass n Prozesse sich m Drucker teilen. Bevor ein Prozess einen Drucker nutzen kann, muss er mit `request()` einen Drucker anfordern. Der Aufruf `printer = request()` gibt dabei die Identität eines freien Druckers zurück, die mit `printer` bezeichnet werden soll. Wenn der Prozess mit dem Drucken fertig ist, soll mit `release(printer)` der Drucker wieder frei gegeben werden.

- Entwickeln Sie einen Monitor, der `request` und `release` implementiert. Spezifizieren Sie die Monitorinvariante. Benutzen Sie *signal-and-continue*.
- Nehmen Sie an, dass die aufrufenden Prozesse eine Priorität angeben. Drucker sollen dann entsprechend der Priorität zugeteilt werden.

  Sie dürfen eine Prioritätswarteschlange benutzen. Die Operation `add(q,p,x)` fügt einen Wert `x` in die Warteschlange `q` mit Priorität p ein. Die Operation `get(q)` gibt das Element höchster Priorität zurück und entfernt es aus `q`. Das Prädikat `empty(q)` gibt an, ob die Warteschlange leer ist.