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.

Musterlösung: Hier wurden nur die Definitionen aus dem Skript abgefragt.

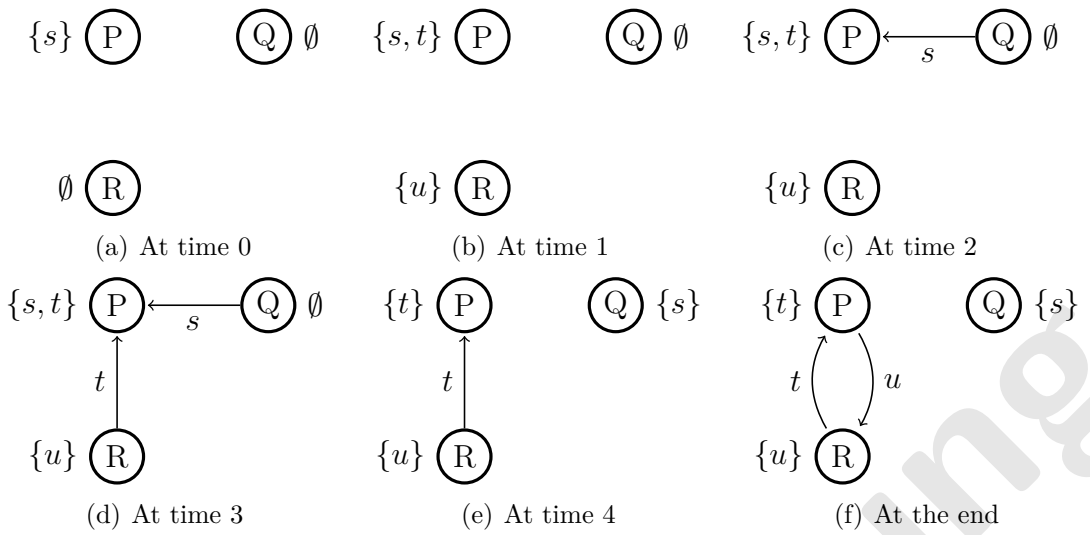
Aufgabe 2 (10 Punkte) Betrachten Sie die folgenden Anforderungen an die gemeinsamen binären Semaphore s, t, und u von den Prozessen P, Q und R.

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.

Zu 1. Der Anforderungsgraph am Ende des Laufes ist in Abbildung 1 wiedergegeben. Nur die letzte Abbildung 1(f) musste abgegeben werden.



(a) At time 0 (b) At time 1 (c) At time 2

(d) At time 3 (e) At time 4 (f) At the end

Abbildung 1: Anforderungsgraphen zu Aufgabe 2

Zu 2. Ja, es kommt zu einer Verklemmung. Prozess P wartet auf das Semaphor u, welches von R gehalten wird. Prozess R wartet auf das Semaphor t, welches von P gehalten wird. Damit schliesst sich der Kreis.

Aufgabe 3 (10 Punkte) 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.

Musterlösung *Zu 1.*

```
1  int x = 0;
2  cobegin
3    x1 = x;
4    x = 2 * x1;
5    x2 = x;
6    x = 2 * x2;
7  ||
```

```
8     x = 1;
9     x3 = x;
10    x = x3 + 1;
11  coend
```

Zu 2. Da nur das Lesen und nur das Schreiben auf eine Variable jeweils separate atomare Aktionen ist, ist das Verhalten des Ursprungsprogramms durch das Verhalten des Programms zu 1. beschrieben. Es spielt also keine Rolle, für welches Programm die Lösung entwickelt wird.

Die möglichen Werte lauten: $\{0, 1, 2, 3, 4, 5, 6, 8\}$

Die 7 ist nicht möglich. Wir müssten dafür ein Interleaving angeben, welches eine 7 erzeugt. Dazu könnte man die 1 verdoppeln und 1 addieren, um eine 3 zu generieren. Diese kann wieder verdoppelt werden, um eine 6 zu erhalten. Jetzt gibt es aber nicht mehr die Möglichkeit, den Wert von x um 1 zu erhöhen, da dies gemacht werden muss, um die 7 zu erzeugen.

Falsch ist es, jede Anweisung des ursprünglichen Programms als atomar zu betrachten, weil die atomare Anweisung dann das Lesen und das Schreiben in eine atomare Aktion verschmelzt. Unter dieser (falschen) Annahme würde x zum Schluss die Werte $\{2, 3, 5, 6, 8\}$ annehmen.

Aufgabe 4 (12 Punkte) 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.

Zu 1. Das erste Programm terminiert, wenn wir schwache Fairness annehmen, denn wenn x erst einmal 9 ist, bleibt es größer und die Bedingung bleibt jetzt dauerhaft erhalten.

Das erste Programm terminiert, wenn wir starke Fairness annehmen, denn wenn x wird immer wieder größer als 9, also muss auch hier die Anweisung `c = false` ausgeführt werden.

Zu 2. Das zweite Programm terminiert nicht, wenn wir schwache Fairness annehmen, denn wenn x 9 wird, kann es danach auf 0 zurückgesetzt werden. Da x nicht dauerhaft größer 9 ist, braucht die Anweisung nicht ausgeführt werden.

Das zweite Programm terminiert, wenn wir starke Fairness annehmen, denn wenn x wird immer wieder 9, also muss auch hier die Anweisung `c = false` ausgeführt werden.

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

Aufgabe 5 (20 Punkte) 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.

Lösung In dieser Lösung sollen Semaphore zur Synchronisation benutzt werden, also insbesondere keine Monitore oder busy-waiting Schleifen.

Die grundlegende Idee der Barriere ist folgende:

```
void wait() {
  ⟨count = count + 1; ⟩
  ⟨await count == n; ⟩
  ⟨count = count - 1; ⟩
  ⟨await count == 0;⟩
}
```

Diese Skizze einer Barriere ist wiederverwendbar.

Wir benutzen nun die Idee von *passing the baton* um die Semaphor-Lösung zu konstruieren. Der letzte ankommende Prozess weckt dann alle wartenden Prozesse auf.

```
void wait() {
  arrive.acquireUninterruptibly();
  count = count + 1;
  if (count < n) arrive.release() else go.release();
  go.acquireUninterruptibly();
  count = count - 1;
  if (count > 0) go.release() else arrive.release();
```

Wir benutzen hier, dass der *delay counter* den gleichen Wert wie die Variable `count`. Wichtig ist hier, dass nie mehr Aufrufe auf `release()` erfolgen, als auch `acquireUninterruptibly()` erfolgt sind, damit das Paar `arrive` und `go` ein *split-binary* Semaphor bleiben.

Nur der letzte Prozess, der an der Barriere ankommt, gibt das Signal auf `go`, und der letzte Prozess, der die Barriere verlässt, gibt sein Signal auf `arrive`, was den letzten Thread in der Barriere freigibt und die Barriere in den initialen Zustand zurück setzt, was sie mit einem `arrive.release()` quittieren.

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

Aufgabe 6 (20 Punkte) [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.

Lösung

Zu 1. Wir benutzen eine *split binary semaphore* T, die Anfangs 1 ist, und ein array von Semaphoren a, die alle Anfangs 0 sind.

```
Semaphore T = new Semaphore(1);
Semaphore[] a = { new Semaphore(0), new Semaphore(0), new
  Semaphore(0) };
```

Der Kellner wählt zufällig eine Zutat von den dreien, die er nicht auf den Tisch legt und weckt dann den wartenden Raucher auf.

```
process Kellner {
  for (;;) {
    P(T);
    x = random(3); // x represents the _missing_ ingredient
    V(a[x]);
  }
```

Der Raucher wartet, bis der Kellner ihn weckt, nimmt die Zutaten, und raucht dann. Danach weckt er den Kellner.

```
process Raucher[x] {
  for (;;) {
    P(a[x]);
    // Make cigarette and smoke
    V(T);
  }
}
```

Zu 2. Die zu 1. beschriebene Lösung ist verklemmungsfrei, da hier eine split-binary semaphore benutzt wird, und der Prozess, der die Semaphore hält, diese dann an genau einen weiteren abgibt. Dadurch wird *hold-and-wait* vermieden.

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

Aufgabe 7 (35 Punkte) 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.

Lösung

Zu 1. Der Monitor repräsentiert den Zustand der Brücke. Autos, die aus dem Norden kommen werden in `north` gezählt und die aus dem Süden in `south`. Der Monitor ist sicher, solange `north == 0 || south == 0` gilt.

Für die Autos aus dem Norden spezifizieren wir die Methoden `enterFromNorth()` und `leaveFromNorth`, um der Brücke mitzuteilen, dass die Brücke betreten und danach verlassen wird. Ähnliche Methoden definieren wir für die Autos aus dem Süden.

Autos kommen aus einer Himmelsrichtung, die zufällig gewählt wird.

```
enum direction { north, south };

process Car(enum direction dir) {
  for (;;) {
    if (dir == north) {
      bridge.enterFromNorth();
      // Drive over the bridge
      bridge.leaveFromNorth();
    } else {
      bridge.enterFromSouth();
      // Drive over the bridge
      bridge.leaveFromSouth();
    }
  }
}
```

Im folgenden nehmen wir an, dass wartende Prozesse Priorität über neu eintretende Prozesse haben. Der Brückenmonitor sieht damit wie folgt aus:

```
monitor Bridge {
  int north = 0, south = 0;
  condition waitNorth, waitSouth;

  void enterFromNorth() {
    if (south > 0) wait(waitNorth);
    north++;
  }

  void leaveFromNorth() {
    north --;
    if (north == 0) signal_all(waitSouth);
  }

  void enterFromSouth() {
    if (north > 0) wait(waitSouth);
    south++;
  }

  void leaveFromNorth() {
    south--;
    if (south == 0) signal_all(waitNorth);
  }
}
```


Zu 2. Im folgenden nehmen wir wieder an, dass wartende Prozesse Priorität über neu eintretende Prozesse haben (sonst muss eine `turn`-Variable benutzt werden). Sollte ein Auto aus einer Richtung warten, sperren wir alle weiteren Autos aus der anderen Richtung.

Der Brückenmonitor sieht damit wie folgt aus:

```
monitor Bridge {
  int north = 0, south = 0;
  Condition waitNorth, waitSouth;

  void enterFromNorth() {
    if (south > 0 || (north > 0 && !empty(waitSouth))) {
      wait(waitNorth);
    }
    north++;
  }

  void leaveFromNorth() {
    north --;
    if (north == 0) signal_all(waitingSouth);
  }

  void enterFromSouth() {
    if (north > 0 || (south > 0 && !empty(waitNorth))) {
      wait(waitSouth);
```

```
    }
    south++;
  }

  void leaveFromNorth() {
    south--;
    if (south == 0) signal_all(waitNorth);
  }
}
```

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

Aufgabe 8 (35 Punkte) 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.

Lösung *Zur Erinnerung:* Ein *Referenzparameter* ist ein Parameter einer Unterprozedur, dessen Modifikation im Methodenrumpf auch ausserhalb dessen sichtbar bleiben. Beachten Sie, dass Java und C keine Referenzparameter unterstützen (hier wird nur der Wert einer Referenz übergeben, es handelt sich also tatsächlich um *Wertparameter*), während sie in C++ durch ein & angezeigt werden.

Wir benutzen eine Variable `stage`, um anzuzeigen, was die verschiedenenen Threads gemacht haben. `stage == 0` bedeutet, dass der Wert in `buffer` frei ist, d.h. der aufrufende Thread ist der erste des Paars. `stage == 1` bedeutet, dass der erste Thread seinen Wert geschrieben hat und jetzt auf den Zweiten wartet. `stage == 2` bedeutet, dass der zweite Wert geschrieben hat und jetzt auf den ersten wartet. Nachdem der erste fertig ist, signalisiert er dem Zweiten, der wiederum allen Nachzügler ein Signal gibt.

```
monitor Exchange {
  int buffer;
  int stage = 0;
  Condition second, complete;

  procedure exchange(int& value) {
    // Wait until the swap is complete.
    while (stage == 2) complete.wait();

    if (stage == 0) {
      // The first caller
      buffer = value;
      stage = 1;
      second.wait();
      // Now, buffer contains the value of the second caller.
```

```
      value = buffer;
      // The exchange is done, wake up the other threads.
      stage = 0;
      complete.notifyAll();
    } else {
      // The second caller
      assert (stage == 1);
      // Swap the current value of buffer with value, buffer was
      // written last by the first partner.
      int t = value;
      value = buffer;
      buffer = t;
      // Establish a lock for all following threads.
      stage = 2;
      second.notify(); // Wake up our partner.
    }
  }
}
```

Diese Lösung benutzt *signal-and-continue* und nimmt keine Priorität für die Signalisierten an. Beachte, dass an der Bedingung `second` höchstens 1 Thread wartet, niemals mehr.