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 (10 Punkte) Sind die folgenden Definitionen zu den Definitionen aus der Vorlesung äquivalent?

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.

Lösung

1. Die Definition ist äquivalent zur Definition der Vorlesung. Eine Definition der Vorlesung lautet: „Ein Algorithmus ist verklemmungsfrei, falls von allen Threads, die in die kritische Sektion eintreten wollen, ein Thread in die kritische Sektion eintreten kann.“ Wenn ein Thread im Eintrittscode ist, und es eine unendliche Folge von kritischen Sektionen gibt, die von anderen Threads ausgeführt werden, dann treten andere Threads in die CS ein.

2. Die Definition ist äquivalent zur Definition der Vorlesung. Wenn ein Thread, der in die CS eintreten will, diese auch erreicht, dann muss der Eintrittscode terminiert haben. Umgekehrt: Wenn der Eintrittscode terminiert, erreichen wir die kritische Sektion.

Aufgabe 2 (5 Punkte) Impliziert Fairness (*starvation-freedom*) die Abwesenheit von Verklemmungen (*deadlocks-freedom)*? Begründen Sie Ihre Antwort.

Lösung *Starvation-freedom* impliziert *deadlock freedom.* Wenn garantiert ist, dass der Thread, der in die kritische Sektion eintreten will, dies auch schließlich tut, dann ist auch garantiert, dass von den Threads, die in die kritische Sektion eintreten wollen, dies auch einer tut.

Aufgabe 3 (15 Punkte) 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.

Lösung: Beide Prozesse können in die kritische Sektion eintreten.
Sei o.B.d.A. `turn=0`.

1. Prozess 1 setzt zu erst `flag[1]=1`.

2. Da `turn==0` gilt, tritt dieser Prozess nun in die While-Schleife ein.

3. Weil `flag[0]` den Wert `0` hat, terminiert die Await-Anweisung.

4. Nun setzte der Prozess 0 die Variable `flag[0]=1`.

```
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

5. Weil noch `turn==0` gilt, tritt Prozess 0 nicht in die While-Schleife ein, sondern geht direkt in die kritische Sektion.

6. Jetzt setzt Prozess 1 die Variable `turn=1` und tritt in die kritische Sektion ein.

Damit sind beide Threads in der kritische Sektion.

Aufgabe 4 (15 Punkte) Nehmen Sie an, es gäbe die Semaphoroperationen

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

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.

Lösung: Zu 1. Diese Lösung ist Verklemmungsfrei. Jeder Philosoph nimmt beide Gabeln gleichzeitig und in einem atomaren Schritt. Dadurch kommt es nie vor, dass der Philosoph eine Resource (also eine Gabel) hält und gleichzeitig auf eine andere wartet.

Zu 2. Es gibt *keine* fairness-Annahme, unter der diese Lösung *absence of starvation* garantiert. Betrachte (o.B.d.A.) den Lauf, in dem Philosoph 1 essen möchte, während Philosoph 0 gerade isst. Nun kann auch Philosoph 2 essen (es sind also die Gabeln $0, 1, 2, 3$ in Benutzung. Nehmen wir nun an, dass sich Philosoph 0 und 2 mit dem Essen abwechseln. Dann ist mindestens immer eine der Gabeln 1 und 2 in Benutzung. Entsprechend terminiert das Essen eines jeden einzelnen Philosophen, aber die Bedingung, auf die der Philosoph 1 wartet, wird von nun an niemals wahr. Damit gilt sowohl die Vorbedingung von schwacher und von starker fairness, und dennoch wird Philosoph 1 verhungern.

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

Aufgabe 5 (18 Punkte) 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 Bartender {
  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.

Lösung: Dieses Problem löst man am Besten mit *passing the baton*. Wir geben als Vorstufe eine Lösung an, die await benutzt.

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

process Bartender {
  for (;;) {
    await (glas[0] == empty && glas[1] == empty);
    glas[0] = vodka or orange;
    glas[1] = vodka or orange;
  }
}
```

Wir geben nun den Screwdriverkunden an, die anderen sind ähnlich:

```
process Barkeeper {
  for (;;) {
    await ((glas[0] == orange && glas[1] == vodka) ||
           (glas[0] == vodka && glas[1] == orange));
    glas[0] = empty;
    glas[1] = empty;
  }
}
```

Nun benutzen wir *passing the baton:*

```
enum { empty, vodka, orange } glas[2]; // Shared variable.
semaphore table = 1, vodka = 0, screwdriver = 0, orange = 0;

process Bartender {
  for (;;) {
    P(table);
    glas[0] = vodka or orange;
    glas[1] = vodka or orange;
    if (glas[0] == vodka && glas[1] == vodka) {
      V(vodka);
    } else if ((glas[0] == orange && glas[1] == vodka) ||
               (glas[0] == vodka && glas[1] == orange)) {
      V(screwdriver);
    } else if (glas[0] == vodka && glas[1] == vodka) {
      V(orange);
    } else {
      V(e); // can't happen!
    }
  }
}
```

Wir geben nun den Screwdriverkunden an, die anderen sind ähnlich:

```
process Screwdriver {
  for (;;) {
    P(screwdriver);
    glas[0] = empty;
    glas[1] = empty;
    V(table);
  }
}
```

Hier haben wir den SIGNAL code optimiert, da nach dem Trinken nur die Bedingung `glas[0] == empty && glas[1] == empty` gelten kann.

Zu 2.: Das Prinzip *passing the baton* erhält die Korrektheit der await Lösung und garantiert dazu Deadlockfreiheit. Zur Korrektheit: Jeder Kunde wartet auf den Inhalt seinen Glases und bekommt sein Signal, wenn sein Glas gefüllt ist. Das wird durch die await Bedingungen garantiert. Der Barkeeper befüllt das Glas erst, nachdem es vollständig geleert wurde und signalisiert den Kunden erst dann, wenn das Glas richtig befüllt ist.

Zu 3.: Das Prinzip *passing the baton* garantiert Deadlockfreiheit, denn es wird eine *split binary semaphore* benutzt, wodurch immer ein Prozess weiter machen kann.

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

Aufgabe 6 (18 Punkte) Implementieren Sie eine Prozedur `exchange(value: in out Integer)`, welche von zwei Prozessen zum Tauschen von Werten aufgerufen wird. Die Notation **`in out`** 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 von `value` 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.

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.

Die Lösung wird in Abbildung 2 gezeigt. Wir benutzen ein binäres Semaphor `e`, um `exchange` unter wechselseitigen Ausschluss auszuführen. Wir benutzen ein zweites Semaphore `w` als Signalgeber. Die Variable `frist` zeigt an, ob der aufrufende Prozess der erste Prozess ist. Die Variable `buffer` wird als gemeinsamer Speicher zur Kommunikation der zu tauschenden Werte benutzt. Der initiale Wert von `buffer` ist unwichtig.

Der erste Aufrufer speichert den Wert von `value` in `buffer`, gibt `e` frei, damit ein zweiter Thread eintreten kann und wartet dann auf `w`. Der Zweite tauscht seinen Wert mit dem von `buffer` aus. Dann wird, während `e` gehalten wird, ein Signal auf `w` gegeben. Dadurch kann der Erste den Wert des Zweitens in `buffer` abholen, ohne dass ein dritter Prozess den Tausch stören kann. Der Erste gibt schließlich `e` wieder frei, damit das nächste Paar tauschen kann.

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

Aufgabe 7 (27 Punkte) 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.

```
semaphore e(1), /* Mutual exclusion between callers. */
          w(0); /* Signal the first caller that the second finished. */
boolean first = true; /* Whether caller is the first one. */
int buffer; /* To pass value between processes */

void exchange(int& value) {
  P(e);
  if (first) { /* First caller */
    buffer = value;
    first = false; /* Indicate that first did his part. */
    V(e);
    P(w); /* When w is signalled, e is locked. */
    value = buffer; /* Get second's value */
    V(e); /* Unlock; was locked by second. */
  } else { /* Second caller */
    int tmp = value; value = buffer; buffer = tmp; /* swap values */
    first = true; V(w);
  }
}
```

Abbildung 2: `exchange` mit Semaphoren

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?

Lösung Wir nehmen hier an, dass signalisierte Prozesse Priorität über neu eintretende Prozesse haben.

Zu 1. Der Monitor repräsentiert den Zustand des Duschraums. Wir führen zwei Zähler nm und nf ein, die die Zahl der Männer und Frauen im Duschraum angeben. Das der Duschraum nicht überfüllt ist, kann durch $nm + nf \leq n$ spezifiziert werden. Der gegenseitige Ausschluss kann wie immer durch $nm = 0 \vee nf = 0$ angegeben werden. Die Monitorinvariante ist dann die Konjunktion beider Prädikate, also:

$$nm + nf \leq n \wedge (nm = 0 \vee nf = 0)$$

Zu 2. Für die Männer spezifizieren wir die Methoden `enterMan()` und `leaveMan()`, um mitzuteilen, dass eine Duschkabine benötigt wird und danach verlassen wurde. Ähnliche Methoden definieren wir für die Frauen.

Der Männerprozess sieht dann wie folgt aus:

```
process Man {
  for (;;) {
    // Do something else ...
```

```
    Shower.enterMan();
    // Take a shower
    Shower.leaveMan();
  }
}
```

Der Frauenprozess sieht ähnlich aus.

```
process Woman {
  for (;;) {
    // Do something else ...
    Shower.enterWoman();
    // Take a shower
    Shower.leaveWoman();
  }
}
```

Der Duschenmonitor sieht damit wie folgt aus:

```
  monitor Shower {
    int nf = 0, nm = 0;
    condition waitMen, waitWomen;

    procedure enterMan() {
      while (nf > 0 || nm >= n) wait(waitMen);
      nm++;
    }

    procedure leaveMen() {
      nm --;
      if (nm == 0) signal_all(waitWomen);
      else if (nm < n) signal(waitMen);
    }

    procedure enterWoman() {
      while (nm > 0 || nf >= n) wait(waitWomen);
      nf++;
    }

    procedure leaveWoman() {
      nf--;
      if (nf == 0) signal_all(waitMen);
      else if (nf == n-1) signal(waitWomen);
    }
  }
```

Diese Lösung benutzt *signal-and-continue.*

Zu 3. Um den Monitor fair zu machen, wird gezählt, wie Person des jeweiligen Geschlechts warten. Sollte eine Person des anderen Geschlechts warten, werden alle Person des jetzt den Duschraum benutzenden Geschlechts ausgesperrt.

Der Duschmonitor sieht damit wie folgt aus:

```
monitor Shower {
  int nf = 0, nm = 0;
  condition waitMen, waitWomen;

  procedure enterMan() {
    if (nf > 0 || nm >= n || (nm > 0 && !empty(waitWomen))) {
      wait(waitMen);
    }
    nm++;
  }

  procedure leaveMen() {
    nm --;
    if (nm == 0) signal_all(waitWomen);
    else if (nm < n && !empty(waitWomen)) signal(waitMen);
  }

  procedure enterWoman() {
    if (nm > 0 || nf >= n || (nf > 0 && !empty(waitMen))) {
      wait(waitWomen);
    }
    nf++;
  }

  procedure leaveWoman() {
    nf--;
    if (nf == 0) signal_all(waitMen);
    else if (nf < n && !empty(waitMen)) signal(waitWomen);
  }
}
```


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

Aufgabe 8 (27 Punkte) 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.

Lösung Zu 1. Die Monitorinvariante ist in diesem Beispiel `true`.

```
monitor {
  boolean[m] printer;
  condition waiting;

  int request() {
    int i;
    for (;;) {
      i = 0;
      while (i < printer.length) {
        if (!printer[i]) {
          printer[i] = true;
          return i;
        }
        ++i;
      }
      // all in use...
      waiting.wait();
    }
  }

  void release(int i) {
    printer[i] = false;
    waiting.notifyAll();
  }
}
```

Zu 2.

```
monitor {
  boolean[] printer;
  PQ<condition> waiting;

  int request(int prio) {
    int i;
    Condition c = new Condition();
    for (;;) {
      i = 0;
      while (i < printer.length) {
        if (!printer[i]) {
          printer[i] = true;

          return i;
        }
        ++i;
      }
      // all in use...
      // Avoid a race on dequeuing and waiting.
      // Also, when we are awakened, the condition has
      // been removed from the queue.
```

```
        waiting.add(prio, c);
        c.wait();
      }
    }

    void release(int i) {
      printer[i] = false;
      if (!waiting.empty()) {
        waiting.get().notify(); // There is only one waiting.
      }
    }
  }
```