Klausur „Nichtsequentielle Programmierung“

Prof. Dr. Marcel Kyas

20. Juli 2011

Freie Universität Berlin

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

Hinweise zur Klausur

- Überprüfen Sie, dass Sie alle Seiten dieser Klausuraufgaben erhalten haben, bevor Sie die Aufgaben bearbeiten. Dieser Aufgabensatz hat *13 Seiten,* diese eingeschlossen.
- Lesen Sie jede Aufgabe vollständig und aufmerksam durch, bevor Sie sie bearbeiten.
- Schreiben Sie Ihre Lösung 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!*
- Schreiben Sie Ihren *Namen* und Ihre *Matrikelnummer* auf jedes Blatt.
- Geben Sie *alle* Blätter ab und nummerieren Sie diese. Tragen Sie die Anzahl der zusätzlichen Blätter unten ein.
- Benutzen Sie nur *nicht-löschbaren* Schreiber, die *blau* oder *schwarz* schreiben. Klausuren, die mit Bleistift geschrieben wurden, werden nicht angenommen. Korrekturband und Korrekturflüssigkeit darf nicht benutzt werden.
- Es sind keine Hilfsmittel zugelassen. Insbesondere dürfen Sie keine Bücher, Rechner, Notebooks oder Mitschriften benutzen. *Mobiltelefone* müssen ausgeschaltet und in der Tasche verstaut werden.
- Halten Sie Ihren Studierendenausweis und einen Lichtbildausweis bereit.
- Die Klausur dauert 90 Minuten.

Diese Klausur hat _____ zusätzliche Blätter.

Wertung (wird vom Institut ausgefüllt)

Aufgabe	1	2	3	4	5	6	7	Σ	Note
Maximale Punktzahl	9	6	15	6	12	24	18	90	
Erreichte Punktzahl									

Aufgabe 1 (9 Punkte) Betrachten Sie das Programm in Listing 1 und beantworten Sie die folgenden Fragen:

Listing 1: A concurrent program

```
PROGRAM interleaving IS
  x: INTEGER := 0;
BEGIN
  COBEGIN
    x = ⟨x⟩ + 3; || x = ⟨x⟩ + 2;
  COEND
END interleaving;
```

(a) Schreibe Sie das Programm in Listing 1 so um, dass es die at-most-once Eigenschaft erfüllt.

(b) Geben Sie die Formel an, mit der man die Zahl der Berechnungen eines nichtsequentiellen Programms bestimmt. Führen Sie alle Bezeichner ein und erklären Sie diese.

(c) Benutzen Sie Ihre Formel aus (b), um die Zahl der Berechnungen des Programms zu bestimmen.

(d) Zählen Sie alle möglichen Werte für `x` nach terminieren des Programms auf.

(e) Ist das Programm *deterministisch (deterministic)*? Begründen Sie Ihre Antwort.

(f) Ist das Programm *determiniert (determined)*? Begründen Sie Ihre Antwort.

Lösung

(a) Ein äquivalentes Programm mit der at-most-once Eigenschaft ist:

```
PROGRAM interleaving IS
  x: Integer := 0;
  t1, t2: Integer;
BEGIN
  COBEGIN
      t1 := x + 3; x := t1;
   || t2 := x + 2; x := t2;
  COEND
END interleaving;
```

(b) Sei N die Zahl der Prozesse und p_i die Zahl der atomaren Anweisungen im Prozess i für $1 \leq i \leq N$. Dann ist die Zahl der Berechnungen $\frac{(\sum_{i=1}^{N} p_i)!}{\prod_{i=1}^{N}(p_i!)}$.

(c) 6

(d) $2, 3, 5$

(e) Das Programm ist nicht deterministisch, weil es mehr als eine Berechnung gibt.

(f) Es ist nicht determiniert, da es mehr als ein Ergebnis gibt.

Aufgabe 2 (6 Punkte) Betrachten Sie die folgende Vorbedingung und Zuweisung

$$\{x \geq 4\}\ \langle x := x - 3\rangle \quad .$$

Zeigen Sie für jedes der folgenden Tripel, ob die obige Zuweisung mit dem Tripel interferiert.

(a) $\{x \geq 0\}\ \langle x := x + 5\rangle\ \{x \geq 5\}$

(b) $\{x \geq 42\}\ \langle x := x + 5\rangle\ \{x \geq 42\}$

(c) $\{x \geq 11\}\ \langle x := x + 5\rangle\ \{x \geq 13\}$

(d) $\{y$ ist ein vielfaches von $5\}\ \langle x := y\rangle\ \{x$ ist ein vielfaches von $5\}$

Lösung

(a) Die Anweisung interferiert mit der Nachbedinung, denn $\{x \geq 5 \wedge x \geq 4\}\ \langle x := x - 3\rangle\ \{x \geq 5\}$ ist nicht gültig. Gegenbeispiel: $x = 5$

(b) Die Anweisung interferiert mit beiden Bedingungen, denn $\{x \geq 42 \wedge x \geq 4\}\ \langle x := x - 3\rangle\ \{x \geq 42\}$ ist nicht gültig. Gegenbeispiel: $x = 42$

(c) Die Anweisung interferiert mit beiden Bedingungen, denn weder $\{x \geq 11 \wedge x \geq 4\}\ \langle x := x - 3\rangle\ \{x \geq 11\}$ noch $\{x \geq 13 \wedge x \geq 4\}\ \langle x := x - 3\rangle\ \{x \geq 13\}$ sind gültig. Gegenbeispiel: $x = 13$

(d) Die Anweisung interferiert mit der Nachbedingung, denn $\{x \geq 4 \wedge x$ ist ein vielfaches von $5\}\ \langle x := y\rangle\ \{x$ ist ein vielfaches von $5\}$ ist nicht gültig. Gegenbeispiel: $x = 5$.

Aufgabe 3 (15 Punkte) In dieser Aufgabe geht es um *deadlock*-Vermeidung.

Ein Restaurant möchte vier Dinnerpartys $P1$, $P2$, $P3$ und $P4$ bedienen. Das Restaurant hat insgesamt 9 Teller und 10 Schüsseln. Das Restaurant benutzt den *Bankiersalgorithmus* als sein *deadlock*-Vermeidungsalgorithmus.

Nimm an, dass jede Gruppe mit Essen aufhören wird und dann auf den Kellner wartet, damit dieser den nächsten Gang serviert (auf Tellern oder in Schüsseln). Nimm weiter an, dass die Gäste geduldig sind.

Der maximale Bedarf jeder Party ist: 7 Teller und 7 Schüsseln für $P1$, 6 Teller und 10 Schüsseln für $P2$, 1 Teller und 2 Schüsseln für $P3$ und 2 Teller und 2 Schüsseln für $P4$.

Zur Zeit hat jede Party: 2 Teller und 3 Schüsseln für $P1$, 3 Teller und 5 Schüsseln für $P2$, 0 Teller und 1 Schüsseln für $P3$ und 1 Teller und 0 Schüsseln für $P4$.

Beantworte die folgenden zwei Fragen:

(a) Wird das Restaurant es schaffen, alle 4 Parties erfolgreich zu bedienen? Begründen Sie ihre Antwort.

(b) Nehmen Sie an, dass eine Party $P5$ in das Restaurant kommt. Insgesamt benötigt diese Party 5 Teller und 3 Schüsseln. Zu Beginn bringt der Kellner dieser Gruppe 2 Teller. Wie viele Teller und Schüsseln muss das Restaurant mindestens kaufen, um alle fünf Gruppen erfolgreich zu bedienen?

Lösung

(a) Nein, es ist nicht möglich. Der Bankiersalgothmus wird zu erst $P3$ und $P4$ einplanen, um deren Betriebsmittel verfügbar zu machen. Dann sind 4 Teller und 2 Schüsseln verfügbar. Damit gibt es 1 Teller und 2 Schüsseln zu wenig für $P1$ und 3 Schüsseln zu wenig für $P2$, weshalb die Prozesse verklemmen.

(b) Das Restaurant muss noch 1 Teller und 2 Schüsseln kaufen.

Aufgabe 4 (6 Punkte) Gegeben sei folgendes Programm:

```
int x = 10; boolean c = true;

cobegin
  ⟨await x == 0⟩; c = false; || while (c) ⟨x = x - 1⟩;
coend
```

Ignorieren Sie für Ihre Antworten auf nachfolgende Fragen Bereichsüberläufe und -unterläufe.

(a) Terminiert das Programm immer unter der Annahme, dass die Ablaufsteuerung *weakly fair* ist? Begründen Sie Ihre Antwort.

(b) 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 = 10; boolean c = true;

cobegin
  ⟨await x == 0⟩; c = false; || while (c) ⟨x = x - 1⟩ || if (x < 0) ⟨x = 10⟩
coend
```

Ignorieren Sie wieder für Ihre Antworten auf nachfolgende Fragen Bereichsüberläufe und -unterläufe.

(c) Terminiert das Programm immer unter der Annahme, dass die Ablaufsteuerung *weakly fair* ist? Begründen Sie Ihre Antwort.

(d) Terminiert das Programm immer unter der Annahme, dass die Ablaufsteuerung *strongly fair* ist? Begründen Sie Ihre Antwort.

Lösung

(a) Nein, denn die Bedingung der Await-Anweisung gilt genau ein Mal und bleibt danach falsch. Sie wird also nicht wahr und bleibt danach wahr.

(b) Nein, denn die Bedingung der Await-Anweisung gilt genau ein Mal und bleibt danach falsch. Sie wird also nicht immer wieder wahr.

(c) Nein, denn die Bedingung der Await-Anweisung gilt höchstens zwei Mal und bleibt danach falsch. Sie wird also nicht wahr und bleibt danach wahr.

(d) Nein, denn die Bedingung der Await-Anweisung gilt höchstens Mal und bleibt danach falsch. Sie wird also nicht immer wieder wahr.

Aufgabe 5 (12 Punkte) *The Dining Savages.* Ein Stamm Eingeborener isst sein gemeinsames Abendessen aus einem großen Topf, der `M` Portionen gekochten Missionar enthält. Wenn ein Eingeborener oder eine Eingeborene essen möchte, nimmt er bzw. sie sich eine Portion, es sei denn der Topf ist leer. Wenn der Topf leer ist, weckt der Hungrige oder die Hungrige den Koch und wartet, bis der Koch den Topf aufgefüllt hat.

(a) Entwickeln Sie Pseudocode für die Aktionen der Eingeborenen und des Kochs. Benutzen Sie ausschließlich Semaphore zur Synchronisation. Die Lösung soll frei von Verklemmungen sein und den Koch nur wecken, wenn der Topf leer ist, sie muss aber nicht fair sein. *Hinweis:* Benutzen Sie *passing the baton*.

(b) Unter welchen Bedingungen ist Ihre Lösung fair, d.h. jeder Hungrige wird immer etwas zu Essen bekommen?

Erste Lösung

(a) In Listing 2 zeige ich zuerst eine (falsche) Lösung mit `await`-Anweisungen, die ich danach mit dem *passing-the-baton* Prinzip in eine Lösung mit Semaphoren umwandle. Der Inhalt des Topfes wird in der gemeinsamen Variablen `servings` gespeichert. Es soll die Invariante `0 <= servings and servings <= M` gelten.

Listing 2: Await-Lösung für Dining Savages

```
1 PROGRAM Savages IS
2   servings: Integer := M;
3   awaken: Boolean := False;
4
5   PROCESS Savage IS
6   BEGIN
7     WHILE true DO
8       ⟨AWAIT (servings > 0) servings := servings - 1;⟩
9   END WHILE;
10 END Savage;
11
12 PROCESS Cook IS
13 BEGIN
14   WHILE true DO
15     ⟨AWAIT (servings = 0) servings := M;⟩
16   END WHILE;
17 END Cook;
```

Der Code zur Lösung des Problems ist in Listing 3 abgebildet. Sie wurde mit der *passing the baton* Methode entwickelt.

Es werden drei Semaphore `entry`, `savage` und `cook` benutzt, von denen `entry` mit 1 und alle anderen mit 0 initialisiert werden. Weiterhin werden zwei Zähler `dsavage` und `dcook`, beide Anfangs 0 benutzt.

Weil `servings = 0 or servings > 0` für nicht-negative Zahlen gültig ist, ist das Programm frei von Verklemmung. Es kann also immer ein Koch oder ein Wilder fortsetzen. Nebenbei kann Listing 3 optimiert werden. Die optimierte Lösung ist in Listing 4 gegeben.

Listing 3: Dining Savages

```
1  PROCESS Savage IS
2  BEGIN
3    WHILE true DO
4      P(entry);
5      IF servings == 0 THEN
6        dsavage := dsavage + 1;
7        V(entry);
8        P(savage);
9      END IF;
10     servings := servings - 1;
11     SIGNAL;
12   END WHILE;
13 END Savage;
14
15 PROCESS Cook IS
16 BEGIN
17   WHILE true DO
18     P(entry);
19     IF servings > 0 THEN
20       dcook := dcook + 1;
21       V(entry);
22       P(cook);
23     END IF;
24     servings := M;
25     SIGNAL;
26   END WHILE;
27 END Cook;
28
29 PROCEDURE SIGNAL IS
30 BEGIN
31   IF servings = 0 and dcook > 0 THEN
32     dcook := dcook - 1;
33     V(cook);
34   ELSIF servings > 0 and dsavage > 0 THEN
35     dsavage := dsavage - 1;
36     V(savage);
37   ELSE
38     V(entry);
39   END IF;
40 END SIGNAL;
```


Wie Anfangs behauptet, ist diese Lösung nicht korrekt, denn hier weckt der Wilde den Koch, der den Topf leert, und nicht der Wilde, der den leeren Topf vorfindet. Dazu muss Listing 4 angepasst werden, um dem neuen Ablaufplan gerecht zu werden. Diese Anpassung geschieht durch umordnen der Synchronisationsmuster, siehe Listing 5. Das Wecken des Kochs wird in den Vorspann des Wildens gezogen. Der Test auf den vollen Topf im Vorspann muss nun in einer Schleife durchgeführt werden, da ein Wilder durch Eintretende überholt werden kann, und somit der Topf leer ist, wenn der Wilde aus dem Topf isst.

Listing 4: Optimierte Lösung für Dining Savages

```
1 PROCESS Savage IS
2 BEGIN
3   WHILE true DO
4     P(entry);
5     IF servings == 0 THEN
6       dsavage := dsavage + 1;
7       V(entry);
8       P(savage);
9     END IF;
10    servings := servings - 1;
11    IF servings = 0 and dcook > 0 THEN
12      dcook := dcook - 1;
13      V(cook);
14    ELSIF servings > 0 and dsavage > 0 THEN
15      dsavage := dsavage - 1;
16      V(savage);
17    ELSE
18      V(entry);
19    END IF;
20  END WHILE;
21 END Savage;
22
23 PROCESS Cook IS
24 BEGIN
25   WHILE true DO
26     P(entry);
27     IF servings > 0 THEN
28       dcook := dcook + 1;
29       V(entry);
30       P(cook);
31     END IF;
32     servings := M;
33     IF dsavage > 0 THEN
34       dsavage := dsavage - 1;
35       V(savage);
36     ELSE
37       V(entry);
38     END IF;
39   END WHILE;
40 END Cook;
```


(b) Die Lösung ist fair, wenn wir starke Semaphore annehmen. Dann kann ein hungrider Wilder am Semaphore `savage` nur beschränkt oft überholt werden.

Zweite Lösung

(a) In Listing 6 zeige ich eine Lösung, die nicht mittels *passing the baton* generiert wurde. Diese Lösung wird auch im Little Book of Semaphores für das Problem angegeben.

Listing 5: Lösung für Dining Savages

```
1  PROCESS Savage IS
2  BEGIN
3    WHILE true DO
4      P(entry);
5      WHILE servings = 0 THEN
6        dsavage := dsavage + 1;
7        IF dcook > 0 THEN
8          dcook := dcook - 1;
9          V(cook);
10       ELSE
11         V(entry);
12       END IF;
13       P(savage);
14     END IF;
15     servings := servings - 1;
16     IF servings > 0 AND dsavage > 0 THEN
17       dsavage := dsavage - 1;
18       V(savage);
19     ELSE
20       V(entry);
21     END IF;
22   END WHILE;
23 END Savage;
24
25 PROCESS Cook IS
26 BEGIN
27   WHILE true DO
28     P(entry);
29     IF servings > 0 THEN
30       dcook := dcook + 1;
31       V(entry);
32       P(cook);
33     END IF;
34     servings := M;
35     IF dsavage > 0 THEN
36       dsavage := dsavage - 1;
37       V(savage);
38     ELSE
39       V(entry);
40     END IF;
41   END WHILE;
42 END Cook;
```


(b) Die Lösung ist fair, wenn wir starke Semaphore annehmen. Dann kann ein hungrider Wilder am Semaphore `entry` nur beschränkt oft überholt werden.

Aufgabe 6 (24 Punkte) *Speicherzuteilung.* Nehmen Sie an, es gibt zwei Operationen: `request(amount)` und `release(amount)`, wobei `amount` eine positive, ganze Zahl ist. Wenn ein Prozess `request` aufruft, soll er so lange warten, bis `amount` Seiten Speicher frei sind.

Listing 6: Lösung für Dining Savages

```
1 PROCESS Savage IS
2 BEGIN
3   WHILE true DO
4     P(entry);
5     IF servings = 0 THEN
6       V(cook);
7       P(savage);
8     END IF;
9     servings := servings - 1;
10    V(entry);
11  END WHILE;
12 END Savage;
13
14 PROCESS Cook IS
15 BEGIN
16   WHILE true DO
17     P(cook);
18     servings := M;
19     V(savage);
20   END WHILE;
21 END Cook;
```


Speicher wird durch aufrufen von `release` freigegeben. Seiten müssen nicht in der gleichen Menge freigegeben werden, in der sie angefordert wurden.

(a) Entwickeln sie einen Monitor, der `request` und `release` implementiert. Geben Sie zuerst die globale Invariante an. Achten Sie vorerst nicht auf die Reihenfolge, in der Anfragen beantwortet werden. Benutzen Sie *signal and continue* und eine *covering condition*. Erklären Sie die Funktionsweise Ihrer Lösung.

(b) Modifizieren Sie Ihre Lösung zu (a) so, dass *shortest job next* als Zuteilungsstrategie benutzt wird. Insbesondere soll kleineren Anfragen vor Größeren entsprochen werden. Erklären Sie die Funktionsweise Ihrer Lösung.

(c) Modifizieren Sie Ihre Lösung zu (a) so, dass *first-come, first-serve* als Zuteilungsstrategie benutzt wird. Das bedeutet, dass eine Anfrage auch dann verzögert werden muss, wenn genügend Speicher vorhanden ist, um der Anfrage zu entsprechen. Erklären Sie die Funktionsweise Ihrer Lösung.

Lösung 1

(a) Der Monitor ist in Listing 7 abgebildet.

Dessen Invariante ist: `available >= 0`. Entsprechend ist die Covering Condition `available - amount >=` in request.

Ein Anfragender wartet, wenn seine Anfrage nicht erfüllt werden kann. Wenn Speicher zurückgegeben wird, werden alle wartenden Prozesse geweckt, damit sie prüfen, ob ihre Anfrage erfüllt werden kann.

(b) Der Monitor ist in Listing 8 abgebildet.

Listing 7: Memory allocation

```
1 MONITOR Memory IS
2   available: Integer;
3   delay: Condition;
4 BEGIN
5   PROCEDURE request(amount: Integer) IS
6   BEGIN
7     WHILE amount > available DO
8       wait(delay);
9     END WHILE;
10    available := available - amount;
11  END request;
12
13  PROCEDURE release(amount: Integer) IS
14  BEGIN
15    available := available + amount;
16    signal_all(delay);
17  END release;
18 END Memory;
```

Listing 8: Memory allocation (SJN)

```
1 MONITOR Memory IS
2   available: Integer;
3   delay: Condition;
4 BEGIN
5   PROCEDURE request(amount: Integer) IS
6   BEGIN
7     WHILE amount > available DO
8       wait(delay, amount);
9     END WHILE;
10    available := available - amount;
11    signal(delay);
12  END request;
13
14  PROCEDURE release(amount: Integer) IS
15  BEGIN
16    available := available + amount;
17    signal(delay);
18  END release;
19 END Memory;
```

Hier wird ein priorisiertes `wait` benutzt, in dem die Anfragegröße als Priorität benutzt wird. Wir nehmen an, dass der Prozess mit der kleinsten Priorität zu erst geweckt wird.

Ein wesentlicher Unterschied ist, dass bei priorisierten Warteschlangen ein `signal_all` nicht sinnvoll benutzt werden kann, denn es werden alle Prozesse zwar in der Reihenfolge ihrer Prioritäten geweckt, aber es gibt keine Garantie, dass die Prozesse dann in dieser Reihenfolge auch in `request` geählt werden. Deshalb wird in release in Zeile 17 ein Prozess mit `signal` geweckt.

Nach dem ein Prozess seine Anfrage erfüllt bekommt, wird in Zeile 11 ein weiterer Prozess geweckt, da mehr Betriebsmittel zurück gegeben worden sein konnten als der Prozess angefragt hat und der Nächste seine Anfrage ebenfalls erfüllt bekommen kann.

(c) Der Monitor ist in Listing 9 abgebildet.

Listing 9: Memory allocation (FCFS)

```
1 MONITOR Memory IS
2   available: Integer;
3   ticket: Integer := 0;
4   next: Integer := 0;
5   delay: Condition;
6 BEGIN
7   PROCEDURE request(amount: Integer) IS
8     myticket: Integer;
9   BEGIN
10    myticket := ticket;
11    ticket := ticket + 1;
12    WHILE amount > available or next /= myticket DO
13      wait(delay);
14    END WHILE;
15    available := available - amount;
16    next := next + 1;
17    signal_all(delay);
18  END request;
19
20  PROCEDURE release(amount: Integer) IS
21  BEGIN
22    available := available + amount;
23    signal_all(delay);
24  END release;
25 END Memory;
```

Es werden Sequenznummern mit zwei Zählern `ticket` und `next` gebildet. Der Wert von `ticket` gibt an, wie viele Anfragen gestellt wurden und der Wert von `next` gibt an, welche Anfrage jetzt beantwortet werden muss. `ticket` wird erhöht, wenn eine Anfrage gestellt wird und `next` wird erhöht, wenn eine Anfrage erfüllt wird.

Grundsätzlich kann man auch die Lösung zu (b) mit den Ticketnummern als Priorität benutzen.

Aufgabe 7 (18 Punkte) Das *Sushi-Bar Problem*.

Nehmen Sie eine Sushi-Bar mit K ($K > 1$) Sitzplätzen und $M > K$ Kunden an. Wenn ein Kunde ankommt und es ist ein Sitzplatz frei, kann er sich sofort in diesen setzen. Wenn er ankommt und alle K Sitzplätze sind besetzt, dann nimmt der Kunde an, dass dies eine Gesellschaft ist, die gemeinsam isst, und er wartet, bis alle Plätze leer sind, bevor er sich setzen kann.

(a) Modellieren Sie die Kunden als Prozesse und geben Sie das Eintritts- und Austrittsprotokoll an, das diese Anforderungen erzwingt. Benutzen Sie ausschließlich Se-

maphore zum Synchronisieren. Achten Sie darauf, dass die Lösung nicht verklemmt, aber achten Sie nicht auf Fairness.

(b) Erklären Sie kurz, wie Ihre Lösung unter (a) funktioniert.

(c) Gibt es eine Semaphorimplementierung, unter der Ihre Lösung zu (a) fair ist? Wenn ja, benennen Sie diese und begründen Sie ihre Antwort. Wenn nicht, geben Sie einen Ablauf an, der nicht fair ist.

Solution Das besondere an diesem Problem ist, dass es nicht direkt mit passing the baton gelöst werden kann. Die Bedingung des Kunden bezieht sich auf zwei zeitlich getrennte Eigenschaften *alle Plätze belegt* und *alle Plätze frei*.

(a) Der Code ist in Listing 10 angegeben.

Listing 10: A Solution to the Sushi Bar Problem

```
1 PROCESS Customer IS
2 BEGIN
3   P(check);
4   IF customers = K THEN
5     waiting := waiting + 1;
6     V(check);
7     P(wait);
8   ELSE
9     customers := customers + 1;
10    V(check);
11  END IF;
12  -- eat
13  P(check);
14  customers := customers - 1;
15  IF customers = 0 THEN
16    WHILE customers < K and waiting > 0 DO
17      customers := customers + 1;
18      waiting := waiting - 1;
19      V(wait);
20    END WHILE;
21  END IF;
22  V(check);
23 END Customer;
```

Verklemmungsfreiheit wird in dieser Lösung garantiert, in dem kein Prozess auf einem Semaphor wartet, während er ein anderes Semaphor noch hält.

(b) Das Eintrittsprotokoll und Austrittsprotokoll ist mit dem Semaphor `check`, das auf 1 initialisiert wurde, gesichert. Kunden warten auf dem Semaphor `wait` darauf, dass alle Plätze frei werden. Weiterhin gibt es zwei Zähler `customers`, welches die Zahl der Kunden an der Bar angibt, und `waiting`, welches die Zahl der wartenden Kunden angibt. Im Prolog wird der Kunde warten, wenn die Bar voll ist. Der letzte Kunde, der die Bar verlässt, wird bis zu K wartende Kunden wecken, damit sie ihrerseits die Bar betreten können (Zeilen 16–20).

(c) Diese Lösung ist nicht fair. Wenn die Bar einmal voll ist, dann wird ein Kunde warten, bis alle Plätze wieder frei sind. Ein Lauf ist denkbar, in dem ein Kunde die Bar verlässt, so dass nun $K-1$ Kunden anwesend sind. Dann kann wieder ein Kunde die Bar betreten und essen. Dies kann sich beliebig oft wiederholen.

Eine alternative Lösung testet in Zeile 4 auf `customers = K or waiting > 0`, d.h. ein Kunde wartet, wenn die Bar voll ist oder ein Kunde noch wartet. Damit werden die Anforderungen verstärkt. So kann garantiert werden, dass die vollständige Party die Bar verlässt, bevor ein wartender Kunde nachrückt. Diese Lösung ist fair, wenn man starke Semaphore annimmt.