Klausur „Nichtsequentielle Programmierung“

Prof. Dr. Marcel Kyas

14. Juli 2010

Freie Universität Berlin

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

Musterlösung

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 in die dafür vorgesehenen Felder auf den Aufgabenblättern, auf die Rückseiten der Aufgabenblätter 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.

Wertung (wird vom Institut ausgefüllt)

Aufgabe	1	2	3	4	5	6	7	Σ	Note
Punktzahl	15	6	3	6	15	27	18	90	
Erreichte Punkte									

Aufgabe 1 (15 Punkte)

Betrachten Sie das folgende Programm:

```
cobegin
  ⟨x = x + 3;⟩ || ⟨x = x + 5;⟩
coend
```

Beantworten Sie die folgenden Fragen:

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

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

Bearbeiten Sie folgende Aufgaben:

(c) Geben Sie die stärkste Nachbedingung (d.h. jene, die alle anderen impliziert) für das Programm an, die für die Vorbedingung $\{x = 0\}$ gilt.

(d) Geben Sie eine *interferenzfreie Beweisskizze,* die Ihre Behauptung beweist. Führen Sie den Interferenzfreiheitstest beispielhaft für *nur eine* Anweisung und *eine* kritische Bedingung durch.

(e) Wie viele Interferenzfreiheitstests müssten Sie insgesamt durchführen, um das Programm korrekt zu beweisen?

Lösung

Zu (a): Das Programm ist nicht deterministisch, weil es zwei verschiedene Ausführungen gibt.

Zu (b): Das Programm ist determiniert, weil der einzige Wert für x bei Terminierung $x = 8$ ist.

Zu (c): Die Nachbedingung lautet $\{x = 8\}$

Zu (d): Die Beweisskizze sieht wie folgt aus:

$\{x = 0\}$
cobegin
  $\{x = 0 \vee x = 5\}$ ⟨`x = x + 3;`⟩ $\{x = 3 \vee x = 8\}$
||
  $\{x = 0 \vee x = 3\}$ ⟨`x = x + 5;`⟩ $\{x = 5 \vee x = 8\}$
coend
$\{x = 8\}$

Aus der Parallelitätsregel unter Annahme der Interferenzfreiheit folgt die Behauptung aus $(x = 0 \vee x = 5) \wedge (x = 0 \vee x = 3) \iff x = 0$ und $(x = 3 \vee x = 8) \wedge (x = 5 \vee x = 8) \iff x = 8$.

Interferenzfreiheit: Eine kritische Bedingung von ⟨`x = x + 3;`⟩ ist $\{x = 0 \vee x = 5\}$, die Vorbedingung von ⟨`x = x + 5;`⟩ ist $\{x = 0 \vee x = 3\}$. Wir müssen also zeigen, dass $\{(x = 0 \vee x = 5) \wedge (x = 0 \vee x = 3)\}$ ⟨`x = x + 5;`⟩ $\{x = 0 \vee x = 5\}$ gültig ist. Beweis:

1. $(x = 0 \vee x = 5) \wedge (x = 0 \vee x = 3) \iff x = 0$
2. $x = 5 \implies x = 0 \vee x = 5$
3. $\{x = 0\}$ ⟨`x = x + 5; `⟩ $\{x = 5\}$ (Zuweisungsaxiom)
4. $\{(x = 0 \vee x = 5) \wedge (x = 0 \vee x = 3)\}$ ⟨`x = x + 5;`⟩ $\{x = 0 \vee x = 5\}$ Konsequenzregel mit 1, 2, 3.

Es müssten 4 Tests durchgeführt werden.

Musterlösung

Aufgabe 2 (6 Punkte)

Betrachten Sie das Programm

```
int x = 0;
cobegin
  x = ⟨x⟩ + 3; || x = ⟨x⟩ + 5;
coend
```

und bearbeiten Sie nachfolgende Aufgaben:

(a) Schreiben Sie das Programm so um, dass es die *at-most-once*-Eigenschaft erfüllt.

(b) Geben Sie die Formel an, mit der man die Zahl der Berechnungen eines nicht-sequentiellen Programms bestimmt. Erklären Sie alle von ihnen benutzten Bezeichner.

(c) Benutzen Sie die Formel unter (b), um die Zahl der Berechnungen Ihrer Antwort zu (a) zu bestimmen.

(d) Nennen Sie alle möglichen Werte für x nach dem Terminieren Ihrer Antwort zu (a).

Lösung

Zu (a):

```
cobegin
  x1 = x;
  x = x1 + 3;
||
  x2 = x;
  x = x2 + 5;
coend
```

Zu (b): Sei n die Zahl der Prozesse, und c_i die Zahl der atomaren Anweisungen des Prozess i, $0 \leq i < n$. Die Zahl der Berechnungen ergibt sich als

$$\frac{(\sum_{i=0}^{n-1} c_i)!}{\prod_{i=0}^{n-1} c_i!}$$

Zu (c): $\frac{(2+2)!}{2!\cdot 2!} = 6$
Zu (d): $\{3, 5, 8\}$.

Aufgabe 3 (3 Punkte)

Erklären Sie, warum ein Semaphor ein *abstrakter Datentyp* ist. Wie ist ein Semaphore spezifiziert?

Lösung

Ein Semaphor ist ein abstrakter Datentyp, da ein Semaphor nur initialisiert werden kann und danach nur durch die Operationen `P()` und `V()` manipuliert und inspiziert werden kann.

Ein Semaphor kann als nicht-negative ganze Zahl spezifiziert werden, auf denen die folgenden Operationen definiert sind:

$$P(s) \triangleq \langle \textbf{await } (s > 0)\ s := s - 1 \rangle \quad V(s) \triangleq \langle s := s + 1; \rangle \quad .$$

Musterlösung

Aufgabe 4 (15 Punkte)

Ein System hat 4 Prozesse, von 0 bis 3 nummeriert, und 3 zuteilbare Betriebsmittel. Die derzeitige Zuteilung, der noch austehende maximalen Bedarfe (d.h. *Maximum* = *Allocated*+ *Need*) und die verfügbaren Betriebsmittel sind wie folgt:

$$\mathit{Allocated} = \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \end{pmatrix} \quad \mathit{Need} = \begin{pmatrix} 7 & 4 & 2 \\ 1 & 2 & 2 \\ 6 & 0 & 3 \\ 4 & 3 & 1 \end{pmatrix} \quad \mathit{Available} = \begin{pmatrix} 2 & 3 & 2 \end{pmatrix}$$

(a) Benutzen Sie den Bankiersalgorithmus, um zu bestimmen, ob dieser Zustand sicher ist. Begründen Sie Ihre Antwort, z.B. in dem sie einen Ablaufplan angeben.

(b) Welche Betriebsmittel und in welcher Menge müssen zusätzlich besorgt werden, damit in diesem Zustand ein weiterer Prozess mit dem maximalen Bedarf $(4, 6, 6)$ ohne Verklemmung eingeplant werden kann?

(c) Welche vier Bedingungen müssen für eine Verklemmung notwendigerweise erfüllt sein?

Lösung

Zu (a): Der Zustand ist sicher. Ein Ablaufplan lautet $2 \cdot 4 \cdot 3 \cdot 1$
Zu (b): $(0, 1, 1)$
Zu (c): Die vier Bedingungen sind:

1. Wechselseitiger Ausschluß
2. zirkuläres Warten
3. hold-and-wait
4. no preemption

Aufgabe 5 (6 Punkte)

Gegeben sei folgendes Programm:

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

cobegin
  ⟨await x >= 42⟩; ⟨c = false;⟩
  || while (c) { ⟨x = x + 1⟩; }
  || while (c) { if (x > 42) ⟨x = 0;⟩ }
coend
```

Beantworten Sie folgende Fragen und begründen Sie Ihre Antworten:

(a) Terminiert das Programm immer unter der Annahme, dass die Ablaufsteuerung gerecht (*weakly fair, just*) ist?

(b) Terminiert das Programm immer unter der Annahme, dass die Ablaufsteuerung mitfühlend (*strongly fair, compassionate*) ist?

Musterlösung

(c) Ist jede mitfühlende Ablaufsteuerung gerecht, d.h. ist jede gerechte Berechnung auch eine Mitfühlende?

(d) Ist jede gerechte Ablaufsteuerung mitfühlend, d.h. ist jede mitfühlende Berechnung auch eine Gerechte?

Lösung

Zu (a): Ein terminieren des Programms ist unter einer gerechten Ablaufsteuerung nicht garantiert, da die Bedingung `x >= 42` der wählbaren await-Anweisung immer wieder falsifiziert wird. Deswegen muss die Anweisung nicht ausgeführt werden.

Zu (b): Ein terminieren des Programms ist unter einer mitfühlenden Ablaufsteuerung garantiert, da die Bedingung `x >= 42` der wählbaren await-Anweisung immer wieder wahr wird, und die Anweisung muss deswegen ausgeführt werden. Danach muss das Programm terminieren.

Zu (c): Eine mitfühlende Ablaufsteuerung ist gerecht. Jede gerechte Berechnung eines Programms P ist auch eine mitfühlende, da eine Bedingung, die von einem Zeitpunkt an für immer wahr wäre auch immer wieder wahr ist.

Zu (d): Eine gerechte Ablaufsteuerung ist nicht mitfühlend. Im Teil (b) und (c) haben wir festgestellt, dass das Programm unter einer gerechten Ablaufsteuerung nicht terminiert, wohl aber unter einer mitfühlenden.

Kommentar zu (c) und (d): Eine gerechte Ablaufsteuerung führt garantiert alle gerechten Berechnungen zu, d.h. jene, die der Definition einer gerechten Berechnung entsprechen. Weitere Berechnungen müssen nicht ausgeführt werden.

Eine mitfühlende Ablaufsteuerung führt garantiert alle mitfühlenden Berechnungen aus, d.h. jene, die der Definition einer gerechten Berechnung entsprechen. Weitere Berechnungen müssen nicht ausgeführt werden.

Nun gilt, dass jede *gerechte* Berechnung eine mitfühlende ist, aber nicht umgekehrt.

Ein Programm, dass unter einer gerechten Ablaufsteuerung terminiert wird also auch unter einer mitfühlenden terminieren, aber nicht umgekehrt.

Aufgabe 6 (27 Punkte)

In einem Ferienlager gibt es einen Duschraum mit n Duschkabinen, $n > 0$. Jede Duschkabine kann höchstens von einer Person benutzt werden. Eine Person kann entweder eine Duschkabine benutzen, auf eine freie Duschkabine warten oder etwas anderes machen, z.B. Sport treiben. Keine Person besetzt für immer eine Duschkabine, aber niemand ist gezwungen, zu duschen. Höchstens n Frauen beziehungsweise Männer können gleichzeitig duschen. Aus sittlichen Gründen dürfen aber nie ein Mann und eine Frau gleichzeitig duschen.

(a) Geben Sie eine Invariante an, die die Anforderung an den Duschraum ausdrückt, d.h. höchstens n Personen duschen gleichzeitig und es duschen nie Männer und Frauen gemeinsam.

(b) Schreiben Sie Pseudocode, der dieses Problem modelliert. Modellieren Sie Männer und Frauen als Prozesse. Benutzen Sie *ausschließlich Semaphore*, um den Zutritt zum Duschraum mit seinen Duschkabinen zu koordinieren. Kümmern Sie sich noch nicht um *fairness,* achten Sie aber auf Verklemmungsfreiheit.

(c) Erklären Sie kurz, warum Ihre Lösung unter (b) sicher ist.

Musterlösung

(d) Erklären Sie kurz, warum Ihre Lösung unter (b) frei von Verklemmung ist.

(e) Modifizieren Sie Ihre Lösung zu (b) so, dass sie *fair* ist, also wenn ein Mann duschen möchte, er schließlich auch duscht und wenn eine Frau duschen, sie schließlich auch duscht. Sie brauchen nur die Änderungen beschreiben.

(f) Welches ist das schwächste Semaphore, das Sie in Ihrer Lösung zu (e) annehmen müssen?

Lösung 1

Dieses Problem ist ähnlich zu der *einspurigen Brücke,* beschränkt aber zusätzlich die Zahl der Prozesse, die in den kritischen Bereich eintreten dürfen. Es kann mit der Staffelstabtechnik gelöst werden.

Zu (a): Seien `men` die Zahl der Männer und `women` die Zahl der Frauen in der Dusche. Die Invariante lautet:

$$(men = 0 \vee women = 0) \wedge men \leq n \wedge women \leq n$$

Zu (b): Die Lösung ist in Abbildung 1 gezeigt. Diese Lösung ist nicht fair.

Zu (c): Die Staffelstabtechnik erhält die Invariante, die schon Sicherheit ausdrückt. Die Bedingungen, z.B. `(women == 0 && men < n)` entstehen dabei als schwächste Vorbedingung der Invariante zu `men++`.

Zu (d): Duschende Männer müssen die Dusche schließlich verlassen, und geben dann einen Platz für den nächsten Mann frei, d.h. führen `men--` aus.

Zu (e): Um die Lösung fair zu machen, führen wir eine Variable `turn` ein, und folgen dem Prinzip von Dekker's Algorithmus. `turn==0` bedeutet, dass Männer duschen dürfen und `turn==1` bedeutet, das Frauen duschen dürfen. `wm` zählt die wartenden Männer im Vorraum und `ww` die wartenden Frauen. Als Vereinfachung soll das Eintreten und Austreten atomar ausgeführt werden; das manipulieren der `w`-Zähler muss nicht atomar geschehen, wir erhalten dadurch aber mehr Code.

Die Lösung ist Verklemmungsfrei, weil `turn` im Zweifel entscheidet, welche Person als nächstes duschen darf.

Wir generieren mit der Staffelstabtechnik eine Lösung mit Semaphoren, die in Abbildung 3 gezeigt wird.

Wir könnten weiterhin zeigen, dass die Invariante `dw == ww && dm == wm` vor jedem Aufruf von `P()` und `V()` gilt. Letztere Optimierung ist in der Lösung nicht berücksichtigt.

Zu (f): Es werden mindestens starke Semaphore benötigt.

Lösung 2

Diese Lösung wurde dem "Little book of semaphores" entnommen und angepasst.

Zu (a): Siehe Lösung 1.

Musterlösung

```
int men = 0, women = 0;
semaphore e(1), cm(0), cw(0);
int dm = 0, dw = 0;

process Man {
  P(e);
  if (women > 0 || men >= n) {
    dm++; V(e); P(cm);
  }
  men++;
  SIGNAL();
  // Take shower
  P(e);
  men--;
  SIGNAL();
}

process Woman {
  P(e);
  if (men > 0 || women >= n) {
    dw++; V(e); P(cw);
  }
  women++;
  SIGNAL();
  // Take shower
  P(e);
  women--;
  SIGNAL();
}

void SIGNAL() {
  if ((women == 0 && men < n) && dm > 0) { dm--; V(cm); }
  else if ((men == 0 && women < n) && dw > 0) { dw--; V(cw); }
  else V(e);
}
```

Abbildung 1: Eine unfaire Lösung

```
int men = 0, women = 0, turn, wm = 0, ww = 0;
process Man {
  for (;;) {
    // Exercise
    ⟨wm++; await ((women == 0 && men < n) && (turn == 0 || ww == 0)) men++; wm--;⟩
    // Take shower
    ⟨ men--; turn = 1;⟩
  }
}

process Women {
  for (;;) {
    // Exercise
    ⟨ww++; await (men == 0 && women < n && (turn == 1 || wm == 0)) women++; ww--;⟩
    // Take shower
    ⟨ women--; turn = 0;⟩
  }
}
```

Abbildung 2: Eine faire Lösung mit await-Anweisungen

Musterlösung

```
int men = 0, women = 0, turn, wn = 0, ws = 0;
int dm = 0, dw = 0;
semaphore e(1), cm(0), cw(0);

process Man {
  for (;;) {
    // Exercise
    P(e);
    wm++; if (!(women == 0 && men < n && (turn == 0 || ww == 0))) { dm++; V(e); P(cm); }
    men++; wm--;
    SIGNAL();

    // Take shower
    P(e); men--; turn = 1; SIGNAL();
  }
}

process Women {
  for (;;) {
    // Exercise
    P(e);
    ws++; if (!(men == 0 && women < n && (turn == 1 || wm == 0))) { dw++; V(e); P(cw); }
    women++; ww--;
    SIGNAL();
    // Take shower
    P(e); women--; turn = 0; SIGNAL();
  }
}

void SIGNAL() {
  if (women == 0 && men < n && (turn == 0 || ww == 0) && dm > 0) { dm--; V(cm); }
  else if (men == 0 && women < n && (turn == 1 || wm == 0) && dw > 0) { dw--; V(cw); }
  else V(e);
}
```

Musterlösung

Abbildung 3: Eine faire Lösung

```
int men = 0, women = 0;
semaphore man(1), woman(1), empty(1), room(N);

process Man {
    for (;;) {
        // Turn on the light
        P(man);
        men++;
        if (men == 1) {
            P(empty);
        }
        V(man);

        P(room); // Limit the numer of men in the room.
        // Shower
        V(room);

        // Leave room and turn off the light.
        P(man);
        men--;
        if (men == 0) {
            V(empty);
        }
        V(man);
    }
}

process Women {
    for (;;) {
        // Turn on the light
        P(woman);
        women++;
        if (men == 1) {
            P(empty);
        }
        V(woman);

        P(room); // Limit the numer of women in the room.
        // Shower
        V(room);

        // Leave room and turn off the light.
        P(woman);
        women--;
        if (women == 0) {
            V(empty);
        }
        V(woman);
    }
}
```

Musterlösung

Abbildung 4: Eine unfaire Lösung ohne Passing the Baton

Zu (b): Eine unfaire Lösung ist in Abbildung 4 gezeigt.

Zu (c): Damit die Lösung sicher ist, muss gezeigt werden, dass sie die Invariante erhält. Wir führen ein informelles, operationales Argument. Es wird gezählt, wie viele Männer oder Frauen in den Duschraum eintreten wollen oder im Duschraum sind. Will ein Mann eintreten, erhöht er den Zähler `men` um `1`. Ist er der Erste, so versucht er das Semaphor `empty` zu nehmen um Frauen auszuschließen. Sollten Frauen duschen, halten sie das Semaphor und der Mann muss warten, dass alle Frauen den Duschraum verlassen haben, während die nachfolgenden Männer noch auf das Semaphor `man` warten. Danach reguliert das zählende Semaphor `room` die Zahl der Personen im Duschraum, so dass in dem Duschraum höchstens `N` Personen sind. Beachte, dass wenn ein Mann im Duschraum ist, keine Frau mehr duscht und umgekehrt. Das Austrittsprotokol ist analog. Der letzte Mann, der die Dusche verlässt, gibt das Semaphor `empty` wieder frei.

Zu (d): Diese Lösung ist frei von Verklemmungen. Für eine Verklemmung ist zirkuläres Warten notwendig. Wir beobachten, dass niemals ein Mann und eine Frau auf `empty` gemeinsam warten. Im folgenden betrachten wir o.B.d.A. Frauenprozesse, denn das Argument ist für Männerprozesse wieder symmetrisch. Eine Frau wartet auf `empty` nur dann, wenn sie die erste Frau ist und während sie `woman` hält. Dann kann keine andere Frau `empty` schon haben. Ein zirkuläres Warten unter Frauen ist daher nicht möglich. Also besitzt ein Mann das `empty`-Semaphor. Männerprozesse warten allenfalls auf das `men`-Semaphor und nie auf das `woman`-Semaphor. Also kommt es auch hier nicht zu zirkulären warten.

Zu (e): Eine faire Lösung ist in Abbildung 5 gezeigt.

Zu (f): Es werden mindestens starke Semaphore benötigt.

Aufgabe 7 (18 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 bzw. die Hungrige den Koch und wartet, bis der Koch den Topf aufgefüllt hat. Das Verhalten der Eingeborenen und des Kochs wird durch den folgenden Code spezifiziert.

```
process Savage[1:n] is
  while true do
    Pot.get_serving();
    -- Eat
  end;
end;
```

```
process Cook is
  while true do
    Pot.sleep_and_put_servings();
  end;
end;
```

(a) Entwickeln Sie Pseudocode für die Aktionen der Eingeborenen und des Kochs. Benutzen Sie einen Monitor zur Synchronisation. Die Lösung soll frei von Verklemmungen sein und den Koch nur wecken, wenn der Topf leer ist. Benutzen Sie *signal and continue.*

Musterlösung

```
int men = 0, women = 0;
semaphore man(1), woman(1), empty(1), room(N);
semaphore turnstile(1); // This must be a strong semaphore!

process Man {
    for (;;) {
        // Turn on the light
        P(turnstile);
            P(man);
            men++;
            if (men == 1) {
                P(empty);
            }
            V(man);
        V(turnstile);

        P(room); // Limit the numer of men in the room.
        // Shower
        V(room);

        // Leave room and turn off the light.
        P(man);
        men--;
        if (men == 0) {
            V(empty);
        }
        V(man);
    }
}

process Women {
    for (;;) {
        // Turn on the light
        P(turnstile);
            P(woman);
            women++;
            if (men == 1) {
                P(empty);
            }
            V(woman);
        V(turnstile);

        P(room); // Limit the numer of women in the room.
        // Shower
        V(room);

        // Leave room and turn off the light.
        P(woman);
        women--;
        if (women == 0) {
            V(empty);
        }
        V(woman);
    }
}
```

Musterlösung

Abbildung 5: Eine faire Lösung ohne Passing the Baton

```
1  monitor Pot is
2    servings : Integer := M; -- Number of servings in pot.
3    cook, savages : Condition; -- Condition variables.
4
5    procedure get_serving() is
6      while servings = 0 do -- While there is no food
7        if not empty(cook) then -- and the cook is sleeping
8          signal(cook); -- Wake him up.
9        end;
10       wait(savages); -- Wait for the new servings.
11     end;
12     servings := servings - 1;
13   end;
14
15   procedure sleep_and_put_servings() is
16     while servings > 0 do -- While there is food in the pot
17       wait(cook); -- sleep.
18     end;
19     servings := M;
20     signal_all(savages);
21   end;
22 end;
```

Abbildung 6: Die Wilden mit *signal-and-continue*

(b) Funktioniert Ihre Lösung unter (a) auch mit *signal and wait?* Wenn ja, geben Sie ein überzeugende Begründung. Wenn nicht, ändern Sie Ihre Lösung so ab, dass sie jetzt mit *signal and wait* funktioniert.

Lösung

Die hier angegenenen Lösungen sind für Monitore entwickelt, in denen neu eintretende Prozesse die gleiche Priorität haben wie signalisierte.

Zu (a): Das Programm in Abbildung 6 löst das Problem mit *signal-and-continue.* Die `while`-Schleife in Zeile 7 implementiert eine *covering-condition,* denn wenn ein Wilder vom Koch geweckt wird, können schon `M` wilde neu eingetretens ein und der Topf ist wieder leer.

Die Boolesche Variable `waiting` markiert, ob der Koch geweckt wurde. Dadurch soll verhindert werden, dass der Koch mehr als einmal gekocht wird. Diese Variable ist nur notwendig, wenn signalisierte Prozesse die gleiche Priorität haben wie signalisierende.

Das Schlafen in `sleep_and_put_servings` wird ebenfalls durch eine `while`-Schleife geschützt (Zeile 19), um die folgende Verklemmung zu verhindern: Erst wird der Topf geleert und dann treten nacheinander die Wilden ein, wecken den Koch, der noch keinen Schritt getan hat und legen sich schlafen. Erst dann tritt der Koch in die Prozedur `sleep_and_put_servings` ein. Würde der Koch jetzt nicht den Topf prüfen, dann würde auch er warten, denn die Signale sind alle an eine leere Bedingung gesendet worden, wurden also verloren. Also schlafen jetzt alle Prozesse.

Es sei noch angemerkt, dass an dieser Stelle eine `if`-Abfrage ausreicht, da es genau einen Kochprozess gibt und die Bedingung durch die Eingeborenen nicht mehr geändert werden kann.

Musterlösung

```
1  monitor Pot is
2    servings : Integer := M; -- Number of servings in pot.
3    cook, savages : Condition; -- Condition variables.
4
5    procedure get_serving() is
6      while servings = 0 do
7        if not empty(cook) then -- Avoid multiple signalling of the cook.
8          signal(cook); -- Cook continues.
9        else
10         wait(savages); -- Wait until it is prepared.
11       end;
12     end;
13     servings := servings - 1;
14   end;
15
16   procedure put_servings() is
17     while servings > 0 do
18       wait(cook); -- Sleep.
19     end;
20     servings := M;
21     signal_all(savages);
22   end;
23 end;
```

Abbildung 7: Die Wilden mit *signal-and-wait*

Der Koch muss wartende Eingeborene mit `signal_all` in Zeile 25 wecken, um zu verhindern, dass Eingeborene "für immer" warten, wenn mehr als ein Eingeborener in Zeile 12 wartet. Das ist z.B. für Java-Monitore möglich, in denen alle Prozesse die gleiche Priorität haben.

Zu (b): Das Programm in Abbildung 7 löst das Problem mit *signal-and-wait.* Die einzige Änderung ist, dass jetzt nicht mehr unbedingt auf den Koch gewartet wird (Zeile 12), da nach einem Signal der Signalisierte Prozess fortsetzt. Wenn eintretende Prozesse die gleiche Priorität haben wie signalisierte Prozesse, kann es passieren, dass diese Prozesse vor dem Koch drankommen, also müssen sie warten.

Musterlösung