Institut für Informatik

Arbeitsgruppe Autonome Intelligente Systeme

Freiburg, 12. Dezember 2017

Systeme I: Betriebssysteme
Übungsblatt 7

Aufgabe 1 (1+1 Punkte)

In der Vorlesung haben Sie den *Peterson-Algorithmus* für den wechselseitigen Ausschluss für zwei Prozesse kennengelernt. Folgend ist eine vermeintliche Erweiterung auf drei Prozesse aufgeführt.

Initialisierung

```
1  f[0] := 0;
2  f[1] := 0;
3  f[2] := 0;
4  turn := 0;
```

Prozess 0

```
5  wiederhole
6  {
7    f[0] := 1;
8    turn := 0;
9    solange((f[1]=1 oder f[2]=1)
10           und turn=0)
11   {
12       tue nichts;
13   }
14
15   Anweisung 1
16   Anweisung 2      } kritische Region
17   ...
18
19   f[0] := 0;
20
21   Anweisung 3
22   Anweisung 4      } nichtkritische Region
23   ...
24 }
```

Prozess 1

```
wiederhole
{
  f[1] := 1;
  turn := 1;
  solange((f[0]=1 oder f[2]=1)
          und turn=1)
  {
      tue nichts;
  }

  Anweisung 5
  Anweisung 6      } kritische Region
  ...

  f[1] := 0;

  Anweisung 7
  Anweisung 8      } nichtkritische Region
  ...
}
```

Prozess 2

```
wiederhole
{
  f[2] := 1;
  turn := 2;
  solange((f[0]=1 oder f[1]=1)
          und turn=2)
  {
      tue nichts;
  }

  Anweisung 9
  Anweisung 10     } kritische Region
  ...

  f[2] := 0;

  Anweisung 11
  Anweisung 12     } nichtkritische Region
  ...
}
```

a) Ist bei der gezeigten Variante für drei Prozesse der wechselseitige Ausschluss gewährleistet? Begründen Sie Ihre Antwort.

b) Ist der wechselseitige Ausschluss garantiert, wenn man in der oben gezeigten Variante in den `solange`-Schleifen bei den Vergleichen mit `turn` statt Gleichheit Ungleichheit fordert (für Prozess `i` `turn` $\neq$ `i`, also beispielsweise für Prozess 0: `turn` $\neq$ `0`)? Begründen Sie Ihre Antwort.

Aufgabe 2 (2 Punkte)

Zwei Vektoren `vektor_a`, `vektor_b` sollen elementweise mit einer komplizierten Funktion verrechnet werden und das Ergebnis im Vektor `ergebnis` gespeichert werden. Dazu existiert bereits folgender Code:

Initialisierung

```
1  vektor_a := [2, 6, 3, 7, 9];
2  vektor_b := [1, 8, 2, 5, 4];
3
4  ergebnis := [0, 0, 0, 0, 0];
5  i := 0;
6  n := 5;
```

Prozess

```
7  solange (i < n) {
8      a := vektor_a[i];
9      b := vektor_b[i];
10     s := komplizierte_funktion(a, b);
11     ergebnis[i] := s;
12     i := i + 1;
13 }
```

Da `komplizierte_funktion()` sehr viel Rechenzeit in Anspruch nimmt, soll die `solange`-Schleife in Zeilen 7–13 von mehreren Prozessen parallel auf mehreren Kernen der CPU ausgeführt werden.

Der angegebene Code funktioniert jedoch nur unter der Annahme, dass nur ein Prozess die `solange`-Schleife ausführt. Ändern Sie den Code so ab, dass er von mehreren Prozessen gleichzeitig ausgeführt werden kann. Es muss sichergestellt sein, dass das Ergebnis korrekt ist und `komplizierte_funktion()` für jedes $i \in \{0, 1, \ldots, n-1\}$ genau einmal aufgerufen wird.

Die Variablen `vektor_a`, `vektor_b`, `ergebnis`, `i` und `n` werden dabei von allen Prozessen geteilt. Die Variablen `a`, `b` und `s` sind lokale Variablen (d.h. jeder Prozess hat eine eigene Kopie der Variable).

Bemerkung. Benutzen Sie die Funktionen `mutex_lock()` und `mutex_unlock()`, die in der Vorlesung vorgestellt wurden. Sie dürfen außerdem die Reihenfolge der Codezeilen ändern und neue Variablen definieren.

Aufgabe 3 (1+3+1+1 Punkte)

Ein bekanntes Problem der Informatik ist das Problem der speisenden Philosophen: In einem Elfenbeinturm sitzen fünf Philosphen an einem runden Tisch und machen nichts außer denken und essen. Jeder der Philosophen hat einen nie leer werdenden Teller Spaghetti vor sich stehen. Zwischen jedem Paar nebeneinanderstehender Teller liegt eine Gabel, insgesamt also 5 Stück. Da es sich um eine sehr schwierig zu essende Sorte Spaghetti handelt, braucht jeder Philosoph 2 Gabeln, um von den Spaghetti zu essen: Je die Gabel rechts und links von seinem Teller.

Jeder Philosoph kann folgende Aktionen ausführen:

- `nimm_linke_Gabel`: Der Philosoph nimmt die linke Gabel in die Hand.

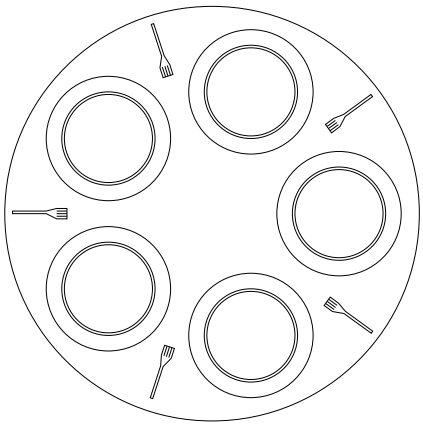

Abbildung 1: Der Tisch der Philosophen

- `nimm_rechte_Gabel`: Der Philosoph nimmt die rechte Gabel in die Hand.
- `lege_linke_Gabel_hin`: Der Philosoph legt die linke Gabel wieder auf den Tisch.
- `lege_rechte_Gabel_hin`: Der Philosoph legt die rechte Gabel wieder auf den Tisch.
- `esse`: Der Philosoph isst.
- `denke`: Der Philosoph denkt.

Die Philosophen tauschen nie die Plätze und die Gabeln kommen stets an den Platz zurück, von dem sie genommen wurden. Ist eine Gabel nicht da, so blockieren die Aktionen `nimm_..._Gabel`, bis die Gabel wieder an ihrem Platz ist und aufgenommen werden kann. Kein Philosoph denkt oder isst unendlich lange.

a) Angenommen jeder Philosoph führt folgenden Algorithmus aus:

Philosoph i

```
wiederhole {
    denke;
    nimm_linke_Gabel;
    nimm_rechte_Gabel;
    esse;
    lege_linke_Gabel_hin;
    lege_rechte_Gabel_hin;
}
```

Ist es möglich, dass bei dieser Lösung ein Deadlock auftritt? Liefern Sie ggf. ein Beispiel.

b) Entwickeln Sie eine neue Lösung mit folgenden Merkmalen:

- Neben den Philosophen-Prozessen existiert ein zentraler Prozess, der die Verwaltung der Gabeln übernimmt.
- Wenn ein Philosoph essen möchte oder fertig mit Essen ist, muss er das dem zentralen Prozess bekanntgeben (siehe Pseudocode unten).

- Der zentrale Prozess soll eine Warteschlange hungriger Philosophen verwalten (**F**irst **I**n, **F**irst **O**ut).
- Bei einem Essenswunsch wird der Philosoph in die Warteschlange eingetragen und muss auf eine Freigabe des zentralen Prozesses warten.
- Der zentrale Prozess holt kontinuierlich Philosophen aus der Warteschlange. Falls die beiden Nachbarn des Philosophen gerade nicht essen, erhält der Philosoph eine Freigabe, andernfalls wird er erneut hinten in die Warteschlange eingereiht.

Die Philosophen führen dabei folgenden Code aus:

Initialisierung

```
1  Freigabe := [0, 0, 0, 0, 0];
2  Q := neue_Warteschlange();
```

Philosoph i für alle $i \in \{0, ..., 4\}$

```
3  wiederhole
4  {
5    denke;
6    Füge i in Warteschlange Q ein
7    solange (Freigabe[i] = 0) {  // aktives Warten auf Freigabe
8      tue nichts;
9    }
10   nimm_linke_Gabel;
11   nimm_rechte_Gabel;
12   esse;
13   lege_linke_Gabel_hin;
14   lege_rechte_Gabel_hin;
15   Freigabe[i] := 0;   // Philosoph signalisiert, dass er fertig ist
16  }
```

Schreiben Sie den zentralen Prozess in gut erläutertem Pseudocode. Denken Sie sich insbesondere eine Lösung dafür aus, wie der zentrale Prozess erkennen kann, ob ein Philosoph eine Freigabe erhalten sollte oder nicht.

Bemerkung. Nehmen Sie an, dass die Warteschlange Q ausschließlich die Operationen `push(item)` (fügt das Element an das Ende der Warteschlange), `pop()` (entfernt ein Element vom Beginn der Liste und liefert es zurück) sowie `isEmpty()` (liefert `true`, wenn die Liste leer ist) unterstützt.

Sie können davon ausgehen, dass die Methoden der Warteschlange *thread-safe* sind, d.h. simultaner Zugriff aus verschiedenen Prozessen nicht zu Problemen führt.

c) Kann es bei Ihrer Lösung aus b) passieren, dass ein Philosoph verhungert, er also unendlich lange darauf warten muss, essen zu dürfen (etwa weil er auf andere Philosophen warten muss, die noch längere Zeit denken)? Liefern Sie ggf. ein Beispiel.

d) Kann es bei Ihrer Lösung passieren, dass ein Philosoph warten muss, obwohl seine Nachbarn gerade nicht essen wollen und seine Gabeln eigentlich frei wären? Liefern Sie ggf. ein Beispiel.

Begründen Sie bitte alle Antworten.

Aufgabe 4 (1+1+1 Punkte)

In der Vorlesung wurden mehrere Algorithmen vorgestellt, mit denen der wechselseitige Ausschluss für eine kritische Region garantiert werden kann. Alle Softwarelösungen hatten jedoch den Nachteil, dass in einer Warteschleife auf das Eintreten eines Ereignisses gewartet wird, wodurch unnötig Rechenzeit verschwendet wird („aktives Warten “).

Um dieses Problem zu vermeiden, stellt das Betriebssystem *Semaphore* zur Verfügung. Semaphore werden verwendet, um Prozesse „schlafen zu legen“ und beim Eintreten eines Ereignisses wieder „aufzuwecken“. Dazwischen wird den Prozessen die CPU nicht zugeteilt, sodass keine Rechenzeit verschwendet wird.

a) Zwei Prozesse A, B verwenden einen[1] gemeinsamen Semaphor S. Unter welchen Bedingungen wird

   1) Prozess A schlafen gelegt?
   2) Prozess A aufgeweckt?

   Geben Sie jeweils allgemein an, was der Zählerstand des Semaphors und der Inhalt der Warteschlange des Semaphors sein muss und welcher Prozess welche Methode aufruft, damit das jeweilige Ereignis eintritt.

b) Drei Prozesse A, B, C verwenden einen gemeinsamen Semaphor S, der anfänglich den Zählerstand 0 aufweist. Erst führt Prozess C und dann Prozess A die Methode `down()` aus, der Zählerstand ändert sich auf -2 und beide Prozesse blockieren. Prozess B führt jetzt die Methode `up()` aus. Der resultierende Zählerstand ist -1 (also kleiner 0). Wird Prozess C trotzdem aufgeweckt und kann fortfahren?

c) Vier Prozesse A, B, C, D verwenden einen gemeinsamen Semaphor S, dessen Zähler auf 2 initialisiert wird. Die Warteschlange des Semaphors arbeitet nach dem FIFO-Prinzip. Vervollständigen Sie in folgender Tabelle den Zustand des Semaphors und der Prozesse (rechenbereit bzw. blockiert) nach dem Ausführen der einzelnen Anweisungen:

Anweisung	Semaphor S Zähler	Semaphor S Warteschl.	Prozess A	Prozess B	Prozess C	Prozess D
1. Initialisierung	2	leer	bereit	bereit	bereit	bereit
2. Prozess A: S.`down()`						
3. Prozess B: S.`down()`						
4. Prozess C: S.`down()`						
5. Prozess D: S.`down()`						
6. Prozess A: S.`up()`						
7. Prozess B: S.`up()`						
8. Prozess C: S.`up()`						

Abgabe: Als PDF-Datei über Ilias bis 18. Dezember 2017, 23:59 Uhr

[1] das Substantiv Semaphor ist maskulin oder ein Neutrum `http://www.duden.de/node/696332/revisions/1109960/view`