

Lösung Übungsblatt 2

Christoph van Heteren-Frese (Matr.-Nr.: 4465677),

Sven Wildermann (Matr.-Nr.: 4567553)

Tutor: Alexander Steen, eingereicht am 26. April 2013

Aufgabe 1

a)

Bei dieser Implementierung ist es möglich, dass zunächst $a[0] = a[1] = false$ gilt (das Prinzip der Unteilbarkeit der Abfrage eines Zustandes und seiner Veränderung ist nicht eingehalten [vgl. 1, S. 39]). Im anschließenden Schleifendurchlauf kann durch die gleiche Ablaufreihenfolge $a[0] = a[1] = true$ gelten, wodurch beide Prozesse gleichzeitig ihren kritischen Bereich betreten würden. Der gegenseitige Ausschluss ist hier also nicht gewährleistet.

b)

Bei dieser Version ist eine Ablaufreihenfolge möglich, bei der `interested[0]=true` und `interested[1]=true` gilt. Wenn beide Prozesse den kritischen Abschnitt betreten möchten, wird einer dem anderen immer den Vortritt lassen. Zwar ist es wahrscheinlich, dass sich bei einem der nächsten Schleifendurchläufe eine andere Situation einstellt, aber grundsätzlich ist so eine Implementierung zu verwerfen, da die Gefahr der Verklemmung (hier: *livelock*) besteht. Dijkstra schreibt dazu:

„If the two processes are about to enter their critical sections, it must be impossible to devise for them such finite speeds, that the decision which one of the two is the first to enter its critical section is postponed to eternity“
[vgl. 2, S. 80]

c)

Bei dieser Implementierung fällt auf, dass der Variablen `favoured` kein Wert zugewiesen wird. Demnach kann – abhängig von der Initialisierung – nur einer der beiden Prozesse seinen kritischen Bereich betreten. Die Warteschleife des nicht (initial) favorisierten Prozesses kann so nie verlassen werden.

Aufgabe 2

a)

Gegenseitiger Ausschluss:

Behauptung: Der Algorithmus von Dekker erfüllt den wechselseitigen Ausschluss.

Beweis: Angenommen, die Aussage ist falsch. Dann gibt es eine Ablaufreihenfolge bei der sich beide Prozesse in ihrem kritischen Abschnitt befinden. Daraus folgt, dass beide Prozesse die Austrittsbedingung der äußeren *for*-Schleife erfüllt haben. Drei Fälle können unterschieden werden, die dazu führen könnten:

1. **Keiner der Beiden Prozesse läuft durch den Schleifenkörper.** Beiden Prozesse müssten dann die jeweilige Schleifenabbruchbedingung von Anfang an erfüllen ($\neg i_1$ für *P0* und $\neg i_0$ für *P1*). Das ist aber nicht möglich. Sei *P1* der zweite Prozess, der die *for*-Bedingung prüft. Dann hat *P0* vorher durch die Anweisung `interested[p]` dafür gesorgt, dass *P1* in die Schleife eintreten muss. Aus Symmetriegründen gilt dies auch für die andere Reihenfolge.
2. **Beide Prozess durchlaufen den Schleifenkörper.** Dann muss einer der beiden Prozesse in der inneren *for*-Schleife hängen bleiben, da `favoured` erst nach Durchlaufen des kritischen Abschnitts (durch *Unlock*) geändert wird.
3. **Einer der Prozesse hat den Schleifenkörper durchlaufen während sich der andere bereits im kritischen Abschnitt befindet.** Dass geht auch nicht. Sei *P1* der Prozess, der seinen kritischen Bereich betreten möchte, obwohl *P0* bereits seinen betreten hat. Dann muss *P1* wegen $p_0 = \text{true}$ seine äußere Warteschleife betreten, die er erst wieder verlassen kann, wenn *P0* seine Interessensbekundung in seinem Austrittsprotokoll mit `interested[p]=false` zurücksetzt.

Diese Fälle zeigen, dass die Annahme falsch ist. Somit garantiert der Dekker-Algorithmus den wechselseitigen Ausschluss. \square

b)

Verklemmungsfreiheit:

„Eine Verklemmung ist nur möglich, wenn bei Erfüllung der Vorbedingungen i_0 und i_1 für den Eintritt in die Schleifen in beiden Eintrittsprotokollen keinen von ihnen Terminiert [...]“ [vgl. 1, S. 44]. Denn: Wenn nur ein Prozess Interesse bekundet, wird an Stelle des Schleifenrumpfes sofort der kritische Bereich betreten. Bekunden aber beide Prozesse Interesse, zieht der nicht favourisierte Prozess sein Interesse solange zurück, bis er selbst favourisiert wird und lässt dem anderen Prozess somit den Vortritt.

Fairness und Behinderungsfreiheit:

Die Anweisung `favoured=1-p` im Austrittsprotokoll sorgt dafür, dass nach Verlassen des kritischen Bereiches eines Prozesses immer der Andere favourisiert wird und so zum

Zuge kommt. Es ist aber trotzdem sichergestellt, dass dieser – sollte er als Einziger Interesse bekunden – den kritischen Bereich erneut betreten kann, da dann der gesamte Schleifenrumpf übersprungen wird und **favoured** gar nicht zum tragen kommt.

Anmerkung: Fairness impliziert Verklemmungsfreiheit. Denn wenn garantiert ist, dass der Prozess, der in den kritischen Bereich eintreten will, dies auch schließlich tut, dann ist auch garantiert, dass von den Prozessen, die in die kritische Sektion eintreten wollen, dies auch einer tut.

Aufgabe 3

Der Algorithmus ist nicht korrekt, da es Situationen gibt, in denen mehr als ein Prozess im kritischen Abschnitt ist. Ein möglicher Ablauf für den erweiterten PETERSON-Algorithmus, der zu einer Situation führt, bei der zwei der drei Prozesse im kritischen Bereich sind, sieht wie folgt aus:

- Alle drei Prozesse (1,2,3) führen die Lock-Funktion bis zur dritten Zeile aus und setzen damit `interested` auf `true` und die globale Variable `favoured` auf $(p + 1) \% 3$
- Zuletzt hat Prozess 3 `favoured` auf 1 gesetzt.
- Damit springt Prozess 1 in den kritischen Abschnitt, da die Bedingungen für die (warte-) for-Schleife nicht mehr erfüllt sind $(1 \& 0 || 1 \& 0) = 0$.
- Nachdem der kritische Abschnitt abgearbeitet wurde setzt der Prozess 1 sein `interested` auf `false`.
- Prozess 2 prüft die Warte-Bedingung mit dem Ergebnis $(1 \& 0 || 0 \& 1) = 0$, so dass Prozess 2 die Warte-Schleife verlässt und seinen kritischen Abschnitt betritt.
- Während Prozess 2 nun seinen kritischen Abschnitt ausführt, prüft Prozess 3 die Warte-Bedingung:
- **Die Überprüfung der Warte-Bedingung ergibt $(0 \& 1 || 1 \& 0) = 0$, so dass Prozess 3 nun ebenfalls den kritischen Abschnitt betritt.**
- Prozess 2 und Prozess 3 befinden sich nun gleichzeitig im kritischen Abschnitt.

Damit ist der gegenseitige Ausschluss nicht gewährleistet und die Erweiterung des Algorithmusses nicht korrekt.

Aufgabe 4

a)

Die Begründung der Korrektheit der ersten Version des Bäckerei-Algorithmus von Lamport erfolgt informal und abgekürzt. Der formale Beweis kann im offiziellen Paper nachgelesen werden [vgl. 3].

Jeder “Kunde” (oder Thread) zieht nacheinander (garantiert durch die Variable “drawing”) eine Nummer, welche um 1 höher sein sollte als die bisher am höchsten vergebene Nummer. Der “Verkäufer” (Prozess) ruft dann die Kunden nacheinander auf - also immer die nächst höhere natürliche Zahl. Sobald zwei Kunden gleichzeitig eine Nummer gezogen haben und damit die gleiche Nummer erhalten haben, entscheidet der Zufall darüber wer von den Kunden mit der gleichen Nummer zuerst bedient wird. Weil nur n Kunden gleichzeitig eine Zahl “ziehen” und somit die gleiche Zahl erhalten können, ist nach $(n-1)$ -Schritten (endlich vielen!) auch der letzte Kunde mit der selben Zahl dran. Außerdem wird von jedem Verkäufer immer nur ein Kunde bedient und die Bedienung ist nach endlicher Zeit erledigt. Somit sind gegenseitiger Ausschluss, Behinderungsfreiheit, Verklemmungsfreiheit und Fairness garantiert. Des weiteren fällt “busy waiting” weg, da immer “der Nächste” aufgerufen wird und die Kunden nicht dauernd fragen müssen, ob Sie “der Nächste” sind.

b)

Der Unterschied zwischen dem ersten und zweiten Algorithmus besteht darin, dass der Fall des “gleichzeitigen Ziehens einer Nummer” nun anders behandelt wird. Während vorher dieser Bereich durch die Variable “drawing” abgesichert wurde (-und alle Kunden in eine Warteschlange geschickt wurden, bis diese dann eine Nummer ziehen konnten-) , gibt es nun eine initiale Nummer (1), die jeder “Kunde” erhält noch bevor er seine richtige Nummer zieht. Somit kann verhindert werden, dass die Nummer eines eintrittswilligen Kunden nicht durch Unterbrechnung eines anderen auf 0 stehen bleibt und dieser somit garnicht bedient wird.

Literatur

- [1] Christian Maurer. *Nichtsequentielle Programmierung mit Go 1 Kompakt*. Springer Vieweg, 2012. ISBN 978-3642299681.
- [2] Edsger W. Dijkstra. The origin of concurrent programming. chapter Cooperating sequential processes, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002. ISBN 0-387-95401-5.
- [3] Leslie Lamport. A new solution of dijkstra’s concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974. ISSN 0001-0782. doi: 10.1145/361082.361093. URL <http://doi.acm.org/10.1145/361082.361093>.