alp4 - Übungszettel 2

Johannes Rohloff

29. April 2013

Aufgabe 1

Untersuchen Sie die Tauglichkeit zur Sperrsynchronisation der folgenden Implementierungen von Lock/Unlock. Hierbei untersuche Ich die gegebenen Implementierungen bezüglich auf Gegenseitigen Ausschluss, Behinderungsfreiheit, Verklemmungsfreiheit und Fairness.

a)

Der gegebene Code:

```
func Lock (p uint) {
      for {
            a[1-p] = a[p]
            if a[p] && ! a[1-p] { break }
      }
}


funcUnlock (p uint) {
     a[1-p] = true
}
```

Gegenseitigen Ausschluss : Dei gegeben Implemtierung garantiert keinen gegnseitigen Ausschluss! Treten beide Prozesse gleichzeitig in die Funktion Lock unter der Voraussetzung, das a[0] und a[1] verschiedene Werte haben und beide lesen bevor einer von ihnen den neuen Wert Überschreiben kann, so haben a[0] und a[1] nach dem schreiben ebenfalls verscheidene Werte. Dies ermöglicht beiden Prozessen gemeinsam in den geschützten Bereich einzutringen.

Behinderungsfreiheit : Behinderungsfreihet ist gegeben, da keine Bestimmte Konfiguration von Variablen vorausgestzt ist, um den Kritischen Abschnitt zu betreten. Die Beiden Variablen müssen entgegengestzte Werte haben und dies wird durch die Zuweisung garantiert, sofern kein weiterer Prozess Lock aufruft.

Verklemmungsfreiheit : Auch die Verklämmungsfreihei der Implementierung ist nicht gegeben. Zwei Aufrufende Prozesse können, im umgeschickten Fall, die Beiden Variablen so überschreiben, dass sie den gleichen Wert haben und so kann niemals ein Prozess in den kritischen bereich eintreten.

Fairness : Über die Fairness kann nicht so viel geagt werden. Da die Implementierung die wichtigsten merkmale eines Schlosses nicht erfüllt.

b) Der gegebene Code:

```
func Lock (p uint) {
      for {
            interested[p] = true
            if interested[1-p] {
              interested[p] = false
            }
            if interested[p] { break }
      }
}

func Unlock (p uint) {
      interested[p] = false
}
```

Gegenseitigen Ausschluss : Der Gegenseitige Auschluss it bei dieser Implementierung garantiert.

Behinderungsfreiheit : Ist gegeben. Wenn nur ein Prozess in den kritischen Bereich vordringen will, so kann er dies direkt tun, da die zweite Bedingung nun nicht eintritt (i_{1-p} ist false).

Verklemmungsfreiheit : Diese Implementierung ist nicht Verklemmungsfrei. Wenn beide in den kritischen Abschnitt kommen behindern sie sich gegenseitig. Keiner der beiden Prozesse gibt nach.

Fairness : Diese Implemntierung ist Fair. (sofern man von einem verklemmenden Prozess sagen kann das er fair ist.)

c) Der gegebene Code:

```
func Lock (p uint) {
      interested[p] = true
      forinterested[1-p] && favoured != p {
         Null()
      }
}

func Unlock (p uint) {
      interested[p] = false
}
```

Gegenseitigen Ausschluss : Die Implementierung sichert den Gegensietigen Ausschluss. Sollte der andere Prozess seine intersse signalisiert haben, so kann ich nur wenn ich favourisiert bin nicht in die Schleife eintreten (alos in den kritischen Abschnitt eintreten).

Behinderungsfreiheit : Ist gegeben. Wenn kein anderer Prozess interesse bekundet, so kann ich nicht in die schleife kommen. Dies ermöglicht ein dirketes eintretn in die kritische Sektion.

Verklemmungsfreiheit : Die Implementierung ist verklemmungsfrei. Sie favourisiert einen Prozess und lässt diesen immer eintreten. Somit kann sich nichts verklemmen.

Fairness : Diese Implementierung ist nicht fair. Sie Ändert den Favourisierten Prozess nicht. Dieser ist immer auf den initialen Wert gesetzt und der entsprechende Prozess kann im zweifelsfall die ganze Zeit ím kritischen Abschnitt bleiben.

Aufgabe 2

a)

Beweisen Sie, dass der Algorithmus von Dekker gegenseitigen Ausschluss garantiert.

Der im Buch gegebene Algorithmus:

```
func Lock(p uint) { // p ¡ 2
      interested[p] = true
      for interested[1-p] {
          if favoured == 1-p {
            interested[p] = false
            for favoured == 1-p { Null() }
            interested[p] = true
          }
      }
}

func Unlock(p uint) { // p ¡ 2
      favoured = 1-p
      interested[p] = false
}
```

Analog zum Beweis im Buch wird eine Notation für die Variablen verwendet.

i_p : intrested[p]

f_p : favoured[p]

$p, i_p, f_p \in \{0, 1\}$

Außerdem werden einige verscheidene Zustände Unterscheiden:

1. Ein Prozess hat sein Interesse noch nicht Signalisiert, also weder $i_p = 0$
2. Ein Prozess befindet sich in der ersten schleife, also $i_0 = i_1 = 1$
3. Ein Prozess befindet sich im kritischen Abschnitt

Beweis des gegnseitigen Ausschlusses : Um den Gegenseitigen Auschluss zu garantieren dürfen in keiner Situation (egal wie / wer und wann) beide Prozesse im kritischen abschnitt sein. Ich agumentiere hier so, dass ich Prozzess 1 in einen zustand versetzte und zeige, dass der andere Prozess nicht in den kritischen Abschnitt gelangen kann. In die andere Richtung müssen nur die Indezes getauscht werden.

Annahme: P1 ist in kritischen Abschnitt. Somit folgt sofort: $i_1 = 1$. Ist der Prozess P0 in 1. so setzt er $i_0 = 1$ und gelangt in die schleife. Ist $f_1 = 0$ so bleibt er in der Schleife gefangen bis $i_1 = 0$ (hier gilt die Invariante $i_0 \wedge \neg i_1 \wedge f_0$, damit der Prozess 1 in den kritischen Abschnitt eintreten kann). Falls $f_1 = 1$ so bleibt P1 in der zweiten Schleife hängen bis $f_1 = 1$.

Annahme: P1 ist eintrittswillig und favorisiert. Es gilt $i_0 = i_1 = 1$ und $f_1 = 1$. Beide Prozesse erreichen die erste Schleife. Hier folgt für P0 $i_0 = 0$ und abwarten auf $f_1 = 0$. Somit ist der Weg für P1 frei, der als einziger Prozess in der kritischen Sektion ist

Annahme: P1 ist eintrittswillig und nicht favorisiert. Hier folgt das gleiche wie im vorherigen Fall, nur mit umgedrehten Indices.

b)

Begründen sie das der gegebene Algorithmus die weiteren Anforderungen an Sperrsynchronisation erfüllt.

Behinderungsfreiheit : Wenn nur ein Prozess am eintritt interessiert ist, so spielt das favourt Bit keine Rolle. Er kann direkt eintreten, da $i_{p-1} = 0$ ist. Die Implementation ist also behinderungsfrei.

Verklemmungsfreiheit : Der Algorithmus kann nicht verklemmen. Wenn beide Prozesse in den kritischen Abschnitt eintreten wollen, so kann durch die Favorisierung nur einer der Beiden in den gesperrten Bereich eintreten. Durch Unlock wird dieses Bit immer geflippt wenn ein Prozess aus dem kritischen Abschnitt austritt.

Fairness : Das Verfahren ist fair. Nach jedem Austritt favourisiert ein Prozess den anderen und lässt ihm beim nächsten Durchlauf den Vortritt. Im Fall von beidseitigem konstanten Interesse, können also beide Prozesse gleich oft auf den kritischen Abschnitt zugreifen.

Aufgabe 3

Zeigen sie, dass diese Implemtierung für drei Prozesse nicht korrekt ist.

```
func Lock (p uint) { // p ¡ 3
      interested[p] = true
      q, r:= (p + 1) % 3, (p + 2) % 3
      favoured = q
      for interested[q] && favoured == q interested[r] && favoured == r {
          Null()
      }
}

func Unlock (p uint) { // p ¡ 3
      interested[p] = false
}
```

Gegenseitigen Ausschluss : Dies ist nicht gegeben. Ein Gegenbeispiel. Alle drei Prozesse wollen in die kritische Sektion eintreten. P1 setzt $f = 2$, allerdings wird er sofort von P0 unterbrochen und somit ist $f = 1$. Nun Prüft P1 seine Eintritts Bedingung und tritt ein (P2 hat noch kein interesse bekundet). Im nächsten schritt setzt P3 $f = 0$. Damit kann auch P0 in den kritischen Bereich einteten. Somit ist kein gegnseitiger Ausschluss gegeben.

Behinderungsfreiheit : Diese Kriterium erfüllt die Implementierung. Wenn ich alleine zugriefen will sind die anderen beiden Werte von $i_p = 0$. Dies erlaubt es erst gar nicht in die Schleife einzutreten.

Verklemmungsfreiheit : Durch das Favourisiern wird garantiert, das mindesten einer der 3 Prozesse zum Zug kommt. Somit ist diese Erweiterung Verklemmungsfrei.

Fairness : Die Implementierung ist fair. Sie verschiebt das favoured Byte pro durchlauf. Somit wird der Zugriff im Zweifelsfall immer weiter gegeben.
Insgesammt folgt somit: kein Gegenseitiger Ausschluss, damit ist die Implementierung nicht korrekt.

Aufgabe 4

a)

Begründen Sie die Korrektheit der ersten Version von Laports Bäckerei Algorithmus

Grundidee: Der Bäckerei Algorithmus nutzt das Konzept eines Ticket Systmes um mehrere Prozesse zu sperrsynchronisieren. Dabei orientieren sich die Prozesse an einer Ticket-Nummer, die die Reihenfolge des Zugriffs der verschiedenen Prozesse darstellt. Die Prozesse wählen dabei selber ihre Nummer.

Der Algorithmus hat folgende Form.

```
func Lock (p uint) { // p ¡ P
      drawing[p] = true
      number[p] = maximum() + 1
      drawing[p] = false
      for a:= uint(0); a ¡ P; a++ {
          for drawing[a]
              { Null() }
          for number[a] ¿ 0 && less(a, p)
              { Null() }
      }
}
func Unlock (p uint) {
      number[p] = 0
}
```

Gegenseitigen Ausschluss : Der gegenseitige Ausschluss ist gegeben, da Über die Ticket Operation eine klare Reihenfolge zur bearbeitung erzwungen wird. Wer zu erst kommt darf auch zu erst auf den kritischen Abschnitt zugreiefen. Das Problem (das in solche Situationen immer auftritt), das die zuweisung der Wartenummer nicht atomar ist wird ducrh zwei Maßnahemen neutralisiert. Erstens, bestimmen die Prozesse selber ihre Nummer, es gibt somit keinen zentralen Zähler bei dem ein Problem mit dem Übreschreiben auftreten könnte. Dies führt allerdings dazu, dass mehrere Prozesse sich die gleich Nummer überlegen können. Dies wird durch eine zweite Maßnahme abgefangen: bei gleicher Wartenummer entscheidet die Prozessnummer. Somit kann immer nur ein Prozess in den kritischen abschnitt eintreten: der Prozess muss die aktuelle nummer (drawing) haben und zeitgleich die niedrigste Prozessnummer.

Behinderungsfreiheit : Es gilt Behinderungsfreiheit. Wenn nur ein Prozess zieht, so kriegt er immer die niedrigste Nummer und ist sofort an der Reihe.

Verklemmungsfreiheit : Ein solcher Algorithmus kann nicht verklemmen. Die Ordnung der Prozessnummern erzwingt, dass ein Prozess immer an der Reihe ist. Es kann keine Situation geben, in der kein Prozess als nächstes in die Kritische Sektion eintreten kann.

Fairness : Der Algorithmus ist fair. Nachdem man den kritischen Bereich verlassen hat, so kann man nur eine neue Nummer ziehen. Somit kann man nicht 2 mal direkt hintereinander an der Reihe sein oder einen anderen Prozess ausschneiden.

b)

Erklären Sie den Schritt von der ersten zur zweiten Version

In der vereunfachten Version hat die Eintrittsfunktion folgenden Aufgabe:

```
func Lock(p uint) { // p ¡ P
        number[p] = 1
        number[p] = max()+1
        for a:= uint(1); a ¡ P; a++ {
            if a != p {
              for number[a] ¿ 0 && less(a, p) { Null() }
            }
        }
}
```

In dieser Version ist die Blockierung für das zieghen herausgenommen. Prozesse müssen nicht mehr darauf warten ob ein andere Prozess grade seine nummer zieht. Diese Änderung funktioniert, da das zeiehen der Nummer nicht atomar abgeschirmt sein muss. Sollte ein Prozess ziehen so kriegt er im schlimmsten Fall die gleiche nummer wie der aktuelle Prozess. Somit kommt er auf jeden Fall dran. Die Änderung verändert somit das Verhalten der Implementierung nicht.