Universität Bielefeld
Technische Fakultät

R|V|S

Rechnernetze und
Verteilte Systeme

Vorlesung 10

Mutual Exclusion

Peter B. Ladkin
ladkin@rvs.uni-bielefeld.de

Sequentielle Operationen in Logik

- Sei es, wir möchten eine Sequenz von Operationen spezifizieren:-
- Also $\pi == (\alpha; \beta; \gamma)$
- Logik hat keine Sequentializierungs-Operator
- Wir definieren

$$A: \wedge pc = `a' \wedge \alpha \wedge pc' = `b' \qquad B: \wedge pc = `b' \wedge \beta \wedge pc' = `c' \qquad \Gamma: \wedge pc = `c' \wedge \gamma \wedge pc' = `d'$$

und

$$\Pi == \Box (A \vee B \vee \Gamma \vee NoOp)$$

Sequentielle Operationen in Logik

- Ausser der Werte des Program-Counters haben beide Programme das gleiche Verhalten
- Für andere Constructs gibt es Tricks:
  - Für loop(α; β; γ) endloop
    definiert man lieber
    Γ: $\wedge$ pc = `c'
    $\wedge$ γ
    $\wedge$ pc′ = `a'
    da nach Γ könnte nur A ausgeführt werden

Sequentielle Operationen in Logik

- Wie könnte man also das folgende Programm in Logik schreiben?

```
var integer   x,y  = 0;
    semaphore sem  = 1;

cobegin loop  P(sem);
              <x <- x+1>;
              V(sem)      endloop
[]
        loop  P(sem);
              <y <- y+1>
              V(sem)      endloop
```

Sequentielle Operationen in Logik

- Also:

$$\begin{aligned}\mathbf{Init} == & \wedge \mathbf{pc1} = \text{`a'} \\ & \wedge \mathbf{pc2} = \text{`a'} \\ & \wedge \mathbf{x} = 0 \\ & \wedge \mathbf{y} = 0 \\ & \wedge \mathbf{sem} = 1\end{aligned}$$

- **ProgVars == <x,y>**
- **Unchanged Φ == $(\Phi' = \Phi)$**

Sequentielle Operationen in Logik

- $\alpha 1 == \wedge$ pc1 = `a'
  $\wedge$ pc1′ = `b'
  $\wedge$ 0 < sem
  $\wedge$ sem′ = sem - 1
  $\wedge$ Unchanged ProgVars
- $\gamma 1 == \wedge$ pc1 = `c'
  $\wedge$ pc1′ = `d'
  $\wedge$ sem′ = sem + 1
  $\wedge$ Unchanged ProgVars

Sequentielle Operationen in Logik

- $\beta 1 == \land$ pc1 = `b'
  $\land$ pc1′ = `c'
  $\land$ x′ = x+1
  $\land$ Unchanged <y,sem>
- Ähnlich für den zweiten Thread für y, sagen wir $\alpha 2, \beta 2, \gamma 2$
- $N1 == \alpha 1 \lor \beta 1 \lor \gamma 1$
  $N2 == \alpha 2 \lor \beta 2 \lor \gamma 2$
  $N == N1 \lor N2 \lor NoOp$
- $\Pi == Init \land \Box NN$

Übung

- Schreib $\alpha 2$, $\beta 2$, $\gamma 2$ in voll

Bemerkungen

- Die volle Aufschreibung in Logik ist lästig
- Aber nicht so lästig wie ein Programm, die unerwartete Nebeneffeckten hat
- Es geht leider nicht anders. Spezifikation muss genau sein und das bedeutet u.a., dass man sagen muss, was mit allen betroffenen Variablen bei jedem Schritt passieren muss
- Das Semaphore-Verfahren hat aber das Mutex-Problem vereinfacht.

Bemerkungen

- Mutex könnte für eine beliebige Anzahl von Prozessen benötigt werden, z.B., für Drucker-Services
- Das Semaphore-Verfahren hat es aber auf zwei Prozessen reduziert: P(sem) und V(sem)
- Die Änderungen auf sem sind relative einfach zu implementieren. Was nicht so einfach ist, ist das "Unchanged" Bedingung festzustellen

Algorithmen für zweier Mutex

- Versuch 1 (V1)

flag: 0..1

P0:loop

while flag ≠ 0 do NoOp;
<CritSect0>;
flag = 1 endloop

P1:loop

while flag ≠ 1 do NoOp;
<CritSect1>;
flag = 0 endloop

Kommentare zu V1

- Die Programme könnte nur im strikten Austausch ausgeführt werden
- Im Fall, dass Prozess P0 stolpert, könnte Prozess P1 nicht weiter ausgeführt werden
- Das gleiche für Prozess P1 gegenüber Prozess P0
- V1 benutzt Busy-Waiting
- Gleichzeitiges Lesen/Schreiben von **flag** ist aber kein Problem

Übung

- Beweise die Aussagen auf der letzten Folie
- Hinweis:
  - In jedem Programm gibt es drei Befehle
  - Die Befehle könnten in einer beliebigen Reihenfolge ausgeführt werden
  - Es gibt nur wenige Befehl-Sequenzen, die wir betrachten müssen
- Behauptung: nur die Interleaving muss betrachtet werden; man kann das Verfahren, in dem zwei Befehle überlappen, ignorieren

Algorithmen für zweier Mutex

- Versuch 2 (V2)

flag: array [0..1] of 0..1

```
P0:loop
        flag[0] = 1;
        while flag[1] = 1 do NoOp;
        <CritSect0>;
        flag[0] = 0   endloop

P1:loop
        flag[1] = 1;
        while flag[0] = 1 do NoOp;
        <CritSect1>;
        flag[1] = 0   endloop
```

Kommentare zu V2

- Prozessen P0 und P1 könnte beliebig ausgeführt werden; sie müssen nicht im strikten Austausch ausgeführt werden
- Sei es, dass Prozess P0 setzt **flag[0]= 1** und direkt danach Prozess P1 setzt **flag[1]= 1.** Es folgt, dass beide Prozessen ewig auf einander warten

  Dieses Verfahren, dass zwie oder mehrere Prozessen auf einander ewig warten, heisst **Deadlock**

Übung

- Beweise die Aussagen auf der letzten Folie

Algorithmen für zweier Mutex

- Versuch 3 (V3)

  flag: array [0..1] of 0..1
  turn: {a,b}

Algorithmen für zweier Mutex

- V3

```
P0:loop
        flag[0] = 1;
        turn = b;
        while flag[1]=1 and turn=b
            do wait;
        <CritSect0>;
        flag[0] = 0   endloop
```

Algorithmen für zweier Mutex

- V3

```
P1:loop
        flag[1] = 1;
        turn = a;
        while flag[1]=1 and turn=a
            do wait;
        <CritSect0>;
        flag[1] = 0   endloop
```

Kommentare zu V3

- **turn** ist ein Shared-Variabel
- **Sei es, dass P0 und P1 versuchen, turn** gleichseitig zu setzen. Wir müssen sicher sein, dass **turn** entweder Wert **a** oder Wert **b** bekommt
- **Dies ist relative einfach, falls turn** ein einfaches Bit ist
- **Ein Hardware-Stück, dass dies sichert, heisst ein Arbiter**

Arbiter

- Ein perfektes Arbiter kann nicht existieren
- Jedes Arbiter hat einen Bereich, in dem es nicht sichergestellt werden kann, dass 0 oder 1 definitiv auskommt
- Diese Phänomen heisst **Metastability**
- **Die Existenz von Metastability ist in den 70en Jahren von Lamport und Palais bewiesen**
- **Gleichzeitig und davon unabhängig hat Charles Molnar ein Beispiel von Metastability bemerkt**
- **Beide Papers sind von Zeitschriften abgelehnt!**

Kommentare zu V3

- Da **turn** ein einfaches Bit sein muss, könnte V3 nicht ohne weiteres zu mehreren Prozessen generalisiert werden

Kommentare zu V1-V3

- V1 und V2 sind in einem Paper von Dijkstra (1965) diskutiert
- Die erste Mutex-Lösung kommt auch von Dijkstra in 1965
- V3 ist ein bekannter Algorithmus, Peterson's Algorithmus (G.L. Peterson, 1981)
- Das Mutex-Problem bleibt: Man versucht, die Voraussetzungen sowie Behauptungen für jeden Algorithmus schwächer zu machen. Ausserdem ist effizienz nach wie vor ein Problem

Übung

- Versuch zu beweisen, dass V3 Mutex guarantiert
- Versuch zu beweisen, dass kein Deadlock bei der Ausführung des V3 entstehen kann