A Fast Mutual Exclusion Algorithm

Paper: Leslie Lamport

Vortrag: Gunnar Biederbeck

Seminar: Prof. deRoever, SS 2005

Überblick

- Der „Fast Mutual Exclusion Algorithm“ ist eine neue Lösung für den gegenseitigen Ausschluss von Prozessen

Überblick

- Der „Fast Mutual Exclusion Algorithm“ ist eine neue Lösung für den gegenseitigen Ausschluss von Prozessen
- Lösungen für dieses Problem sind seit Jahren bekannt,
  - diese haben aber einen Linearen Aufwand in Bezug auf die Anzahl der Prozessoren

Überblick

- Der „Fast Mutual Exclusion Algorithm“ ist eine neue Lösung für den gegenseitigen Ausschluss von Prozessen
- Lösungen für dieses Problem sind seit Jahren bekannt,
  - diese haben aber einen Linearen Aufwand in Bezug auf die Anzahl der Prozessoren
- Es sind bei bestimmten Vorraussetzungen nur 7 Speicherzugriffe notwendig

Überblick

- Diese Vorraussetzungen sind:
  - Kein konkurrierender Versuch in die CS zu gelangen
  - Atomares Lesen und Schreiben auf Register

Überblick

- Diese Vorraussetzungen sind:
  - Kein konkurrierender Versuch in die CS zu gelangen
  - Atomares Lesen und Schreiben auf Register
- Lamports Algorithmus hat konstanten Aufwand: 5 Schreibzugriffe, 2 Lesezugriffe
  - Nur unter den genannten Vorraussetzungen
  - Der Algorithmus ist nicht fair

Überblick

- Diese Vorraussetzungen sind:
  - Kein konkurrierender Versuch in die CS zu gelangen
  - Atomares Lesen und Schreiben auf Register
- Lamports Algorithmus hat konstanten Aufwand: 5 Schreibzugriffe, 2 Lesezugriffe
  - Nur unter den genannten Vorraussetzungen
  - Der Algorithmus ist nicht fair
- Der Algorithmus stellt bei konkurrierendem Zugriff trotzdem den gegenseitigen Ausschluss sicher
  - Dann aber mit höherem Aufwand

Anforderungen und Erfahrungen

- CS-Algorithmen sollten Prozesse nicht länger warten lassen, als unbedingt notwendig
  - Selbst, wenn der Algorithmus einen Prozess ewig lange warten lassen könnte, ist dieser Praxis-tauglich, da eine solche Situation kaum vorkommt

Anforderungen und Erfahrungen

- CS-Algorithmen sollten Prozesse nicht länger warten lassen, als unbedingt notwendig
  - Selbst, wenn der Algorithmus einen Prozess ewig lange warten lassen könnte, ist dieser Praxis-tauglich, da eine solche Situation kaum vorkommt
- Ein „Glaube“ (v.a. Betriebssystemprogrammierer) ist, dass Konkurrenzsituationen selten sind
  - Experimente aus der Praxis bestätigen dies

Anforderungen und Erfahrungen

- CS-Algorithmen sollten Prozesse nicht länger warten lassen, als unbedingt notwendig
  - Selbst, wenn der Algorithmus einen Prozess ewig lange warten lassen könnte, ist dieser Praxis-tauglich, da eine solche Situation kaum vorkommt
- Ein „Glaube“ (v.a. Betriebssystemprogrammierer) ist, dass Konkurrenzsituationen selten sind
  - Experimente aus der Praxis bestätigen dies
- Algorithmen werden in der Regel nach der Geschwindigkeit bei nicht-gleichzeitigem Zugriff von verschiedenen Prozessen beurteilt

Anforderungen und Erfahrungen

- CS-Algorithmen sollten Prozesse nicht länger warten lassen, als unbedingt notwendig
  - Selbst, wenn der Algorithmus einen Prozess ewig lange warten lassen könnte, ist dieser Praxis-tauglich, da eine solche Situation kaum vorkommt
- Ein „Glaube“ (v.a. Betriebssystemprogrammierer) ist, dass Konkurrenzsituationen selten sind
  - Experimente aus der Praxis bestätigen dies
- Algorithmen werden in der Regel nach der Geschwindigkeit bei nicht-gleichzeitigem Zugriff von verschiedenen Prozessen beurteilt
- Trotzdem darf kein Algorithmus bei gleichzeitigem Zugriff zu einem Deadlock führen

Geschwindigkeit der Algorithmen

- Zugriff auf gemeinsame Speicherbereiche braucht viel Zeit (im Gegensatz zu Registern)
- Ein gutes Maß für die Ausführungsgeschwindigkeit ist daher die Anzahl der Zugriffe auf den Speicher

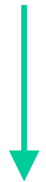
Geschwindigkeit der Algorithmen

- Zugriff auf gemeinsame Speicherbereiche braucht viel Zeit (im Gegensatz zu Registern)
- Ein gutes Maß für die Ausführungsgeschwindigkeit ist daher die Anzahl der Zugriffe auf den Speicher
- Alle dem Autor bekannten Algorithmen haben einen linearen Zeitaufwand O(N) bei N-Prozessen und keiner Konkurrenzsituation
- Die hier präsentierte Lösung benötigt nur 5 Schreib und 2 Lesezugriffe auf gemeinsamen Speicher, hat also **konstanten Zeitaufwand**

Der Algorithmus

- Vorraussetzungen
  - Jeder Prozess hat eine eindeutige ID (positive Integerzahl)
  - Der Zugriff auf einzelne Speicherzellen ist atomar
  - Kein Prozess verändert je die Variablen des Algorithmus von anderen Prozesse

Idee



```
repeat
   await <x=0>;
   <x:=i>
   delay;
until <x=i>;
critical section;
x:=0;
```

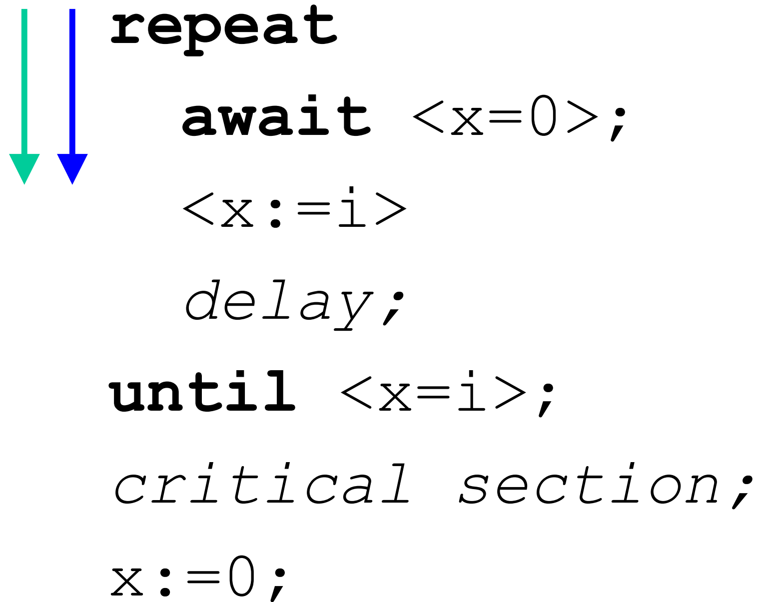
Zunächst ohne delay

Prozess i

Prozess j

x=0

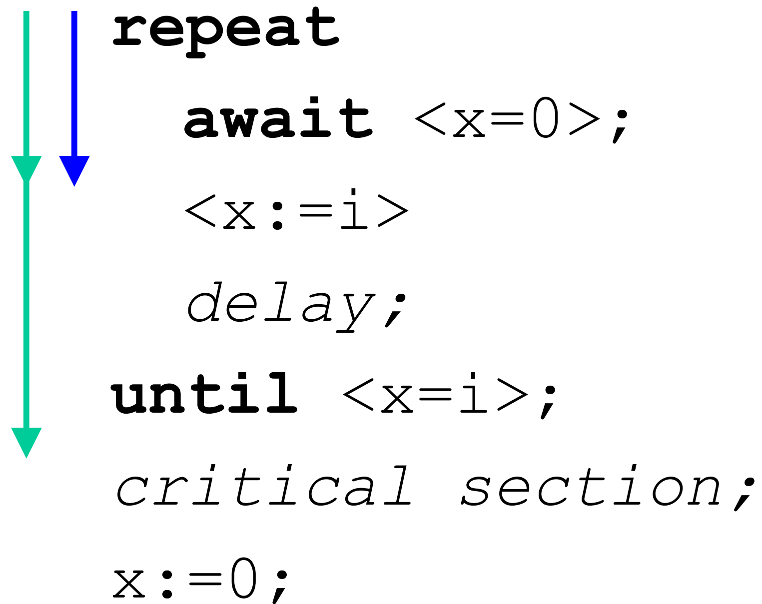
Idee



Zunächst ohne delay

- Prozess i
- Prozess j
- x=0

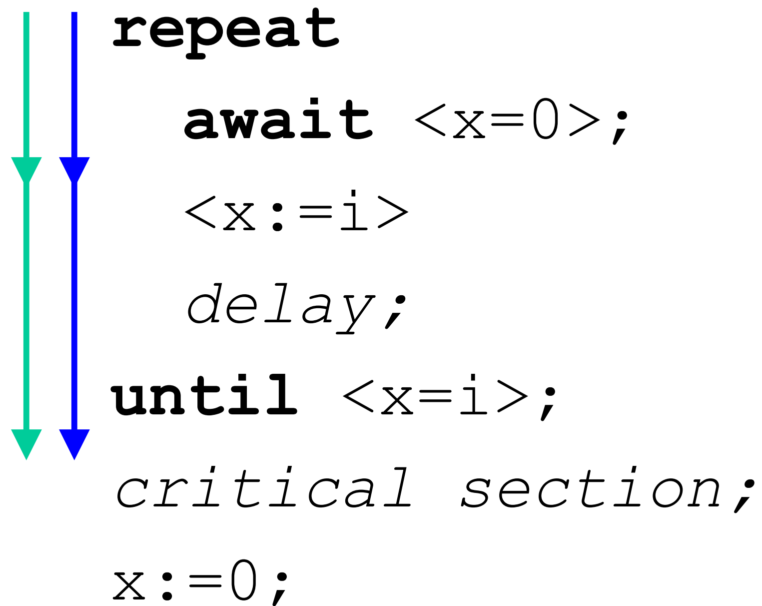
Idee



Zunächst ohne delay

- Prozess i
- Prozess j
- x=i

Idee



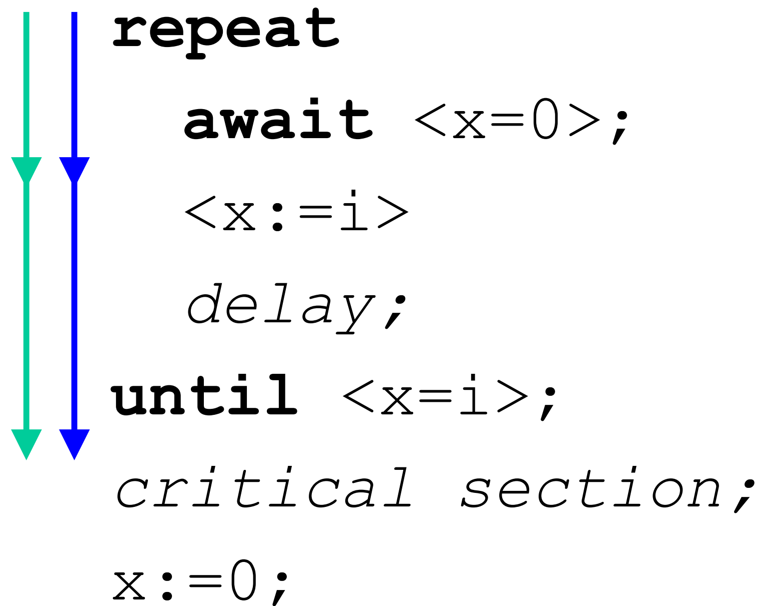
Zunächst ohne delay

Prozess i

Prozess j

x=j

Idee



Zunächst ohne delay

- Prozess i
- Prozess j
- x=j

CRASH !

Idee

```
repeat
   await <x=0>;
   <x:=i>
   delay;
until <x=i>;
critical section;
x:=0;
```

Falls ein Prozess j das x=0 in await ausliest, bevor ein anderer Prozess i <x:=i> ausgeführt hat, sorgt das delay dafür, dass Prozess i lange genug wartet, so dass Prozess j <x:=j> ausgeführt hat, bevor Prozess i zum until <x=i> kommt.

Idee

```
repeat
   await <x=0>;
   <x:=i>
   delay;
until <x=i>;
critical section;
x:=0;
```

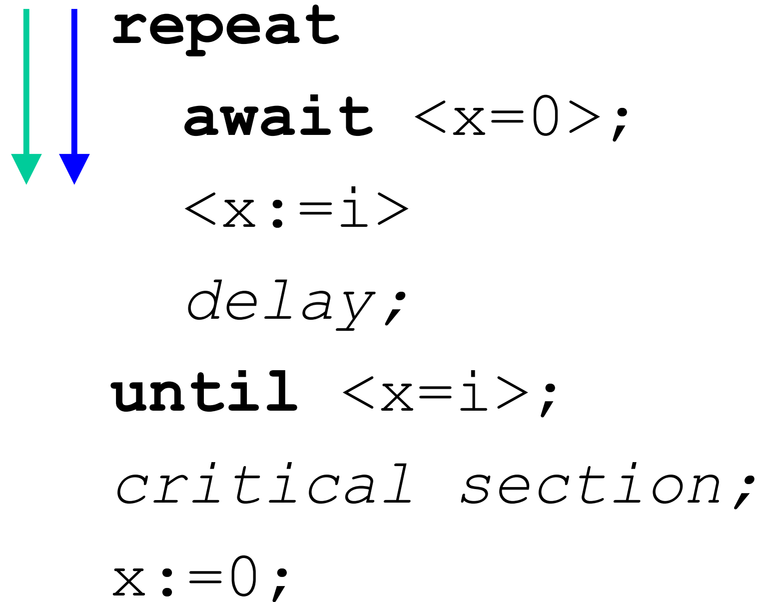
Jetzt mit delay

Prozess i

Prozess j

x=0

Idee



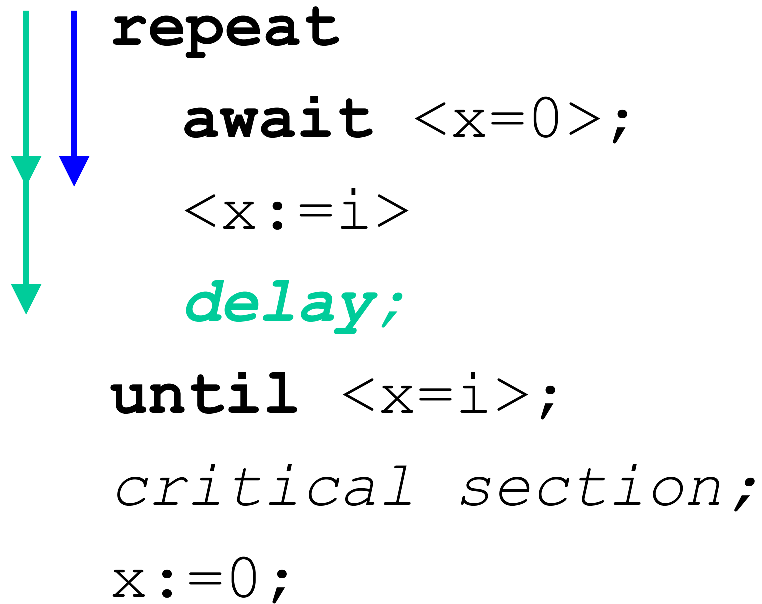
Jetzt mit delay

Prozess i

Prozess j

x=0

Idee



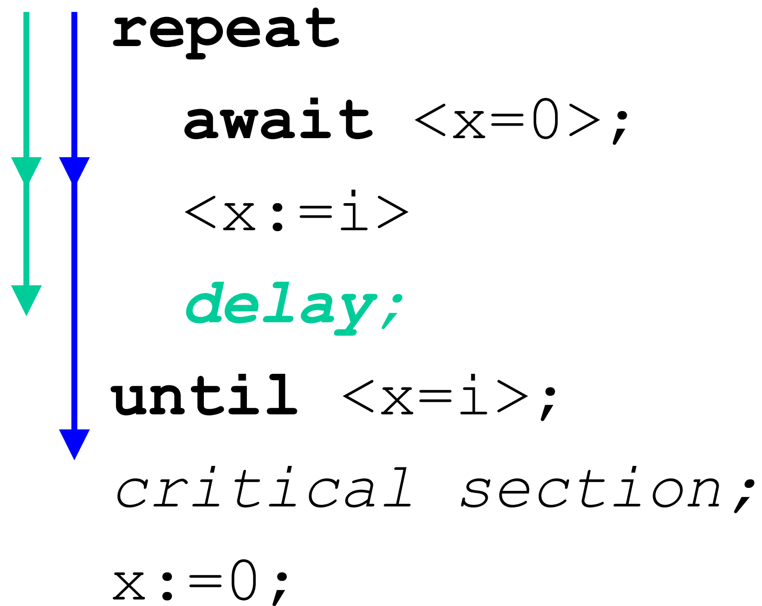
Jetzt mit delay

Prozess i

Prozess j

x=i

Idee



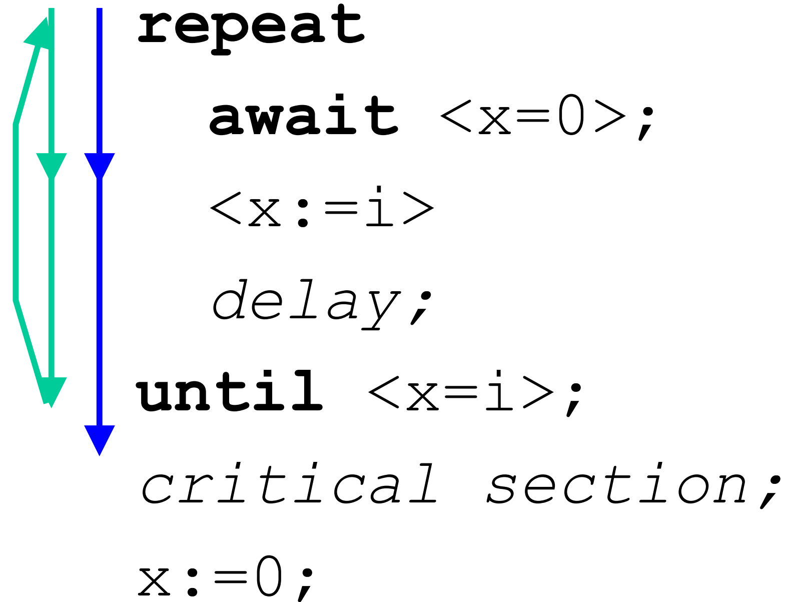
Jetzt mit delay

Prozess i

Prozess j

x=j

Idee



Jetzt mit delay

Prozess i

Prozess j

x=j

Idee

- Wir könnten annehmen, dass das „delay“ funktioniert
- Dann müsste das delay die schlechtestmögliche Ausführungszeit berücksichtigen
- Bei N-1 Prozessen würde dies zu O(N) führen

Resultat und Konstruktion

- O(N) wollen wir vermeiden, also kein delay!
- Überlegung, wie auf den Speicher zugegriffen werden muss:

Resultat und Konstruktion

- O(N) wollen wir vermeiden, also kein delay!
- Überlegung, wie auf den Speicher zugegriffen werden muss:
- S_i sei die Sequenz von Schreib- und Lesezugriffen auf den Speicher von Prozess i

Resultat und Konstruktion

- O(N) wollen wir vermeiden, also kein delay!
- Überlegung, wie auf den Speicher zugegriffen werden muss:
- S_i sei die Sequenz von Schreib- und Lesezugriffen auf den Speicher von Prozess i
- Forderung: Sequential Consistent Memory:
  - Jedes Lesen in S_i gibt entweder den Initialen Wert des Speichers oder den Wert, der vorher in S_i geschrieben wurde

Konstruktion

- Es macht keinen Sinn, dass S_i hierfür einen Wert schreibt der von anderen Prozessen S_j nicht gelesen wird (ein solches Schreiben kann nicht entscheidend sein beim Zugang zur CS)
- das S_i sieht für alle Prozesse gleich aus:
  - Die Anzahl der Speicherzugriffe ist dann fest, unabhängig von der Prozessanzahl
  - Jeder Prozess greift in der gleichen Reihenfolge auf den Speicher zu

Konstruktion

$S_i = r_x$ ...

Macht keinen Sinn, da jeder Prozess den initialen Wert von X lesen würde

Konstruktion

$S_i = w_x , w_x \ldots$

$S_i = w_x , w_y \ldots$

Zwei Schreiboperationen nacheinander in die selbe Variable macht keinen Sinn

Zwei Schreiboperationen in verschiedene Variablen könnte man durch eine längere Schreiboperation ersetzen

Konstruktion

$S_i = w_x , r_x \ldots$

$S_i = w_x , r_y \ldots$

Die Variable X zu lesen, nachdem diese gerade geschrieben wurde mach keinen Sinn, jeder Prozess liest exakt den Wert, der vorher geschrieben wurde

Es macht keinen Sinn eine Variable zu lesen, die nie geschrieben wird

Die letzte Operation vor der CS sollte kein Write sein, denn Schreiben von einer Variablen kann dem Prozess bei einer Entscheidung zum Ausführen der CS nicht helfen

Konstruktion

$S_i = w_x , r_y , w_y , r_x , ...$

1. Schreiben von x
2. Lesen von y
3. Wenn y den initialen Wert hat, dann schreiben von y
4. Lesen von x, wenn x den Wert hat, der in 1. geschrieben wurde, dann CS ausführen

Konstruktion

$S_i = w_x , r_y , w_y , r_x , CS , w_y$

- Nach der CS muss der Prozess eine Variable schreiben, um den anderen Prozessen das Ende der CS anzuzeigen
- Dies kann kein Schreiben von x sein, denn jeder Prozess schreibt x an erster Stelle

Algorithmus

```
start:      <x:=i>
            if <y != 0> then goto start fi;
            <y:=i>
            if <x != i> then delay;
                   if <y != i> then goto start fi;
            fi;
            CS;
            <y:=0>
```

X=i

Y=0

- Das delay, muss lang genug sein für folgenden Fall:
  falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

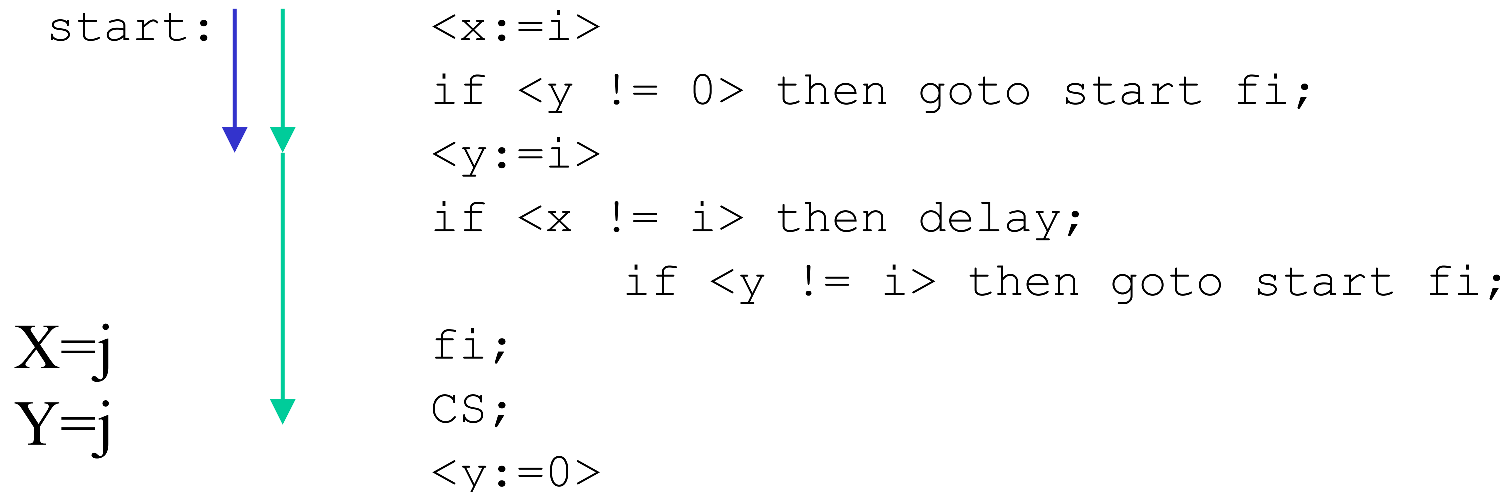
Algorithmus

```
start:    <x:=i>
          if <y != 0> then goto start fi;
          <y:=i>
          if <x != i> then delay;
                  if <y != i> then goto start fi;
          fi;
          CS;
          <y:=0>
```

X=j
Y=0

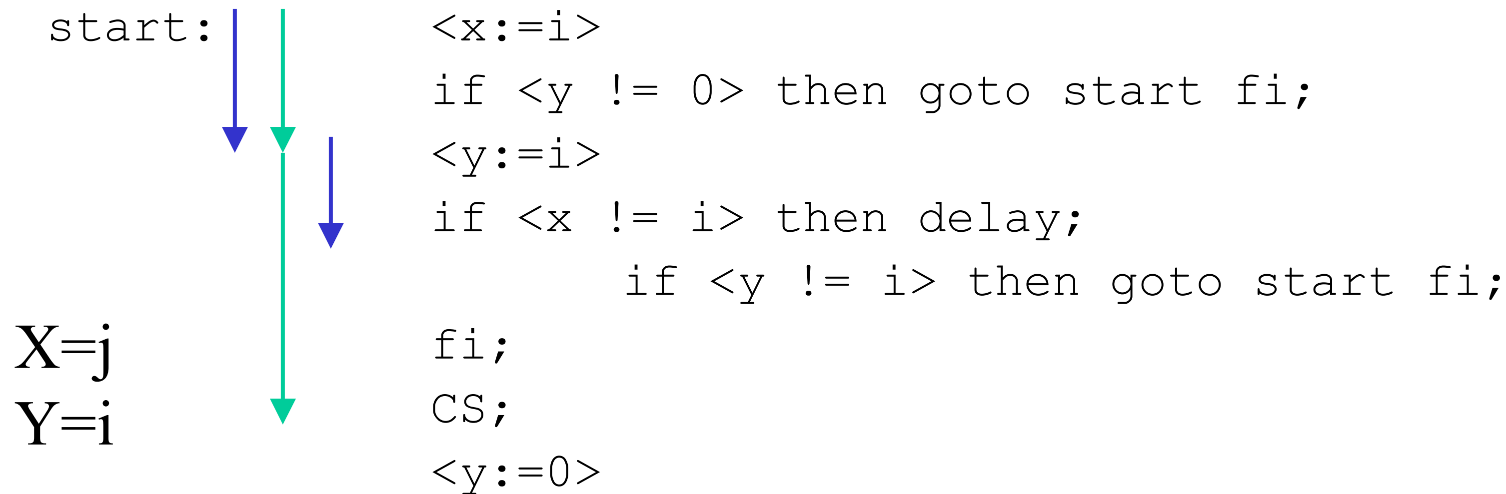
- Das delay, muss lang genug sein für folgenden Fall:
  falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

Algorithmus



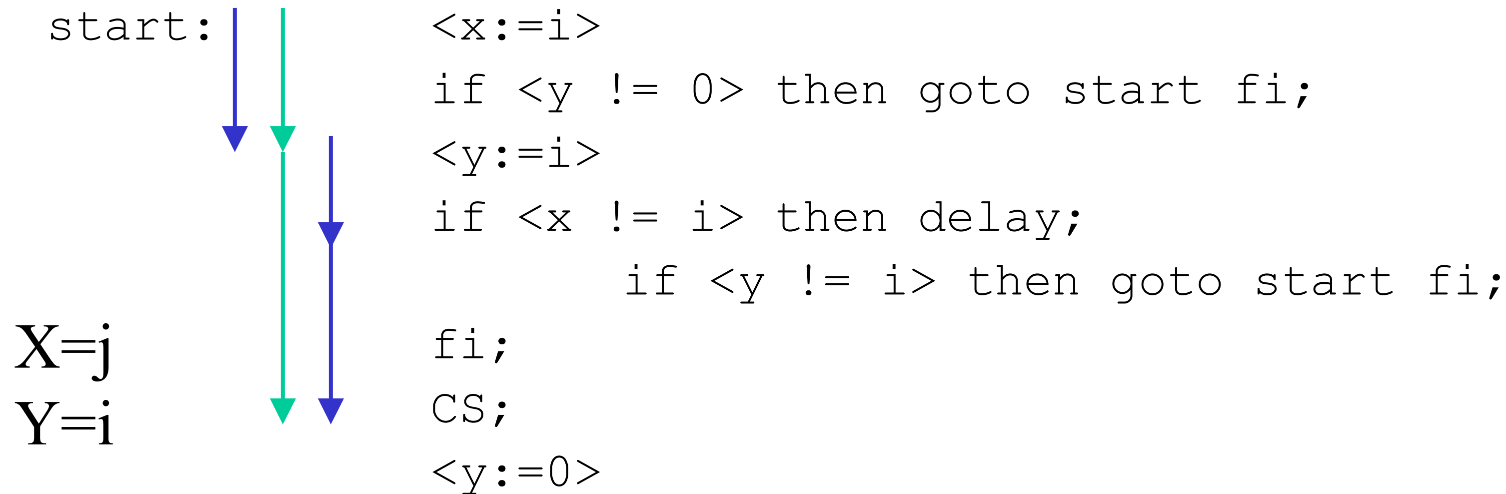
- Das delay, muss lang genug sein für folgenden Fall:
  falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

Algorithmus



- Das delay, muss lang genug sein für folgenden Fall:
  falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

Algorithmus



- Das delay, muss lang genug sein für folgenden Fall:
  falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

Algorithmus

```
start:        <x:=i>
              if <y != 0> then goto start fi;
              <y:=i>
              if <x != i> then delay;
                     if <y != i> then goto start fi;
              fi;
              CS;
              <y:=0>
```

X=i

Y=0

- Das delay, muss lang genug sein für folgenden Fall: falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

Algorithmus

```
start:      <x:=i>
            if <y != 0> then goto start fi;
            <y:=i>
            if <x != i> then delay;
                    if <y != i> then goto start fi;
            fi;
            CS;
            <y:=0>
```

X=j
Y=0

- Das delay, muss lang genug sein für folgenden Fall:
  falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

Algorithmus

```
start:    <x:=i>
          if <y != 0> then goto start fi;
          <y:=i>
          if <x != i> then delay;
                 if <y != i> then goto start fi;
X=j       fi;
Y=j       CS;
          <y:=0>
```

- Das delay, muss lang genug sein für folgenden Fall:
  falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

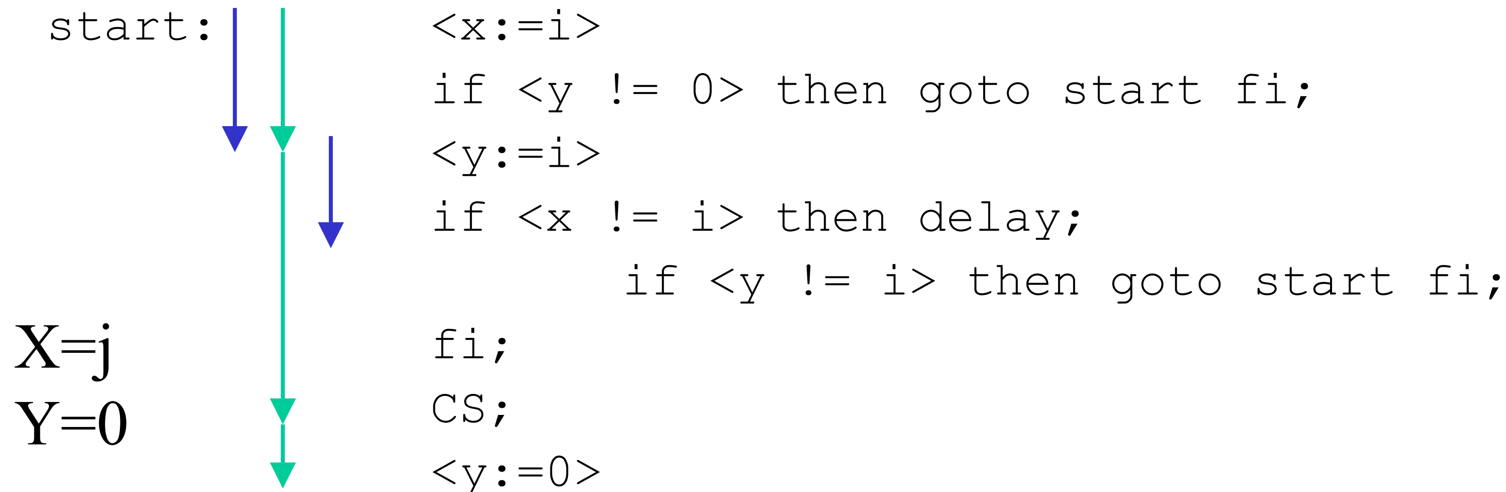
Algorithmus

```
start:      <x:=i>
            if <y != 0> then goto start fi;
            <y:=i>
            if <x != i> then delay;
                  if <y != i> then goto start fi;
            fi;
            CS;
            <y:=0>
```

X=j
Y=i

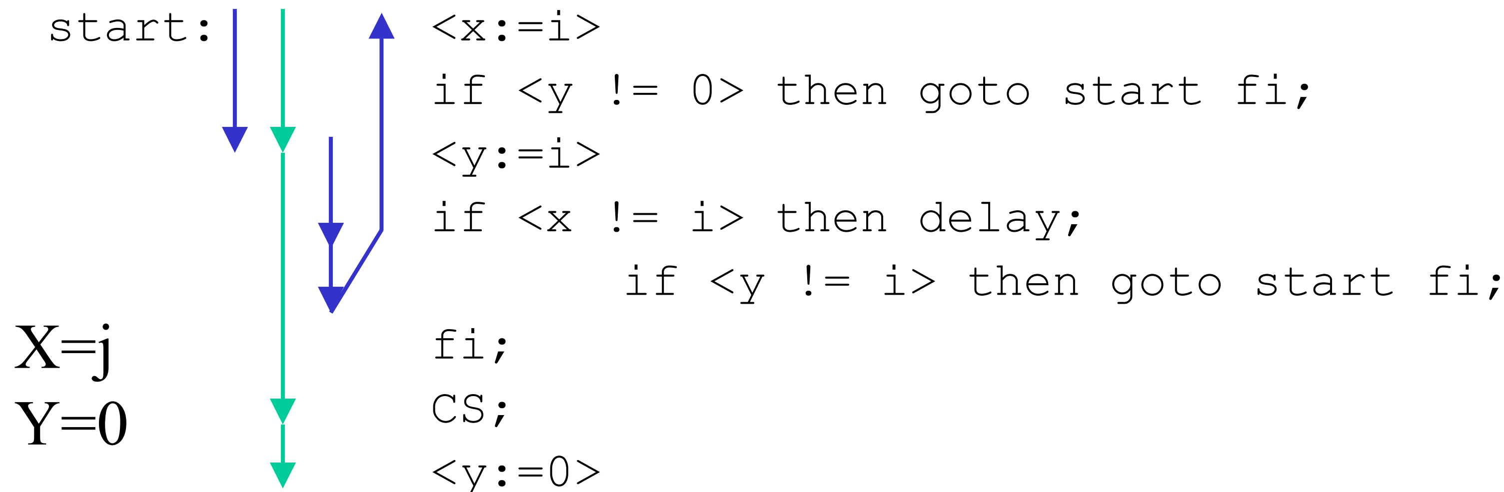
- Das delay, muss lang genug sein für folgenden Fall:
  falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt", da es nur bei Konkurrenzsituationen ausgeführt wird

Algorithmus



- Das delay, muss lang genug sein für folgenden Fall: falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

Algorithmus



- Das delay, muss lang genug sein für folgenden Fall:
  falls Prozess j y=0 liest bevor Prozess i y=i setzt, und j CS ausführt, dann muss j y=0 setzen, bevor i das delay beendet hat
- Dieses delay ist „erlaubt“, da es nur bei Konkurrenzsituationen ausgeführt wird

Algorithmus: Problem

```
start:        <x:=i>
              if <y != 0> then goto start fi;
              <y:=i>
              if <x != i> then delay;
                     if <y != i> then goto start fi;
              fi;
              CS;
              <y:=0>
```

X=2
Y=0
w2-x

Algorithmus: Problem

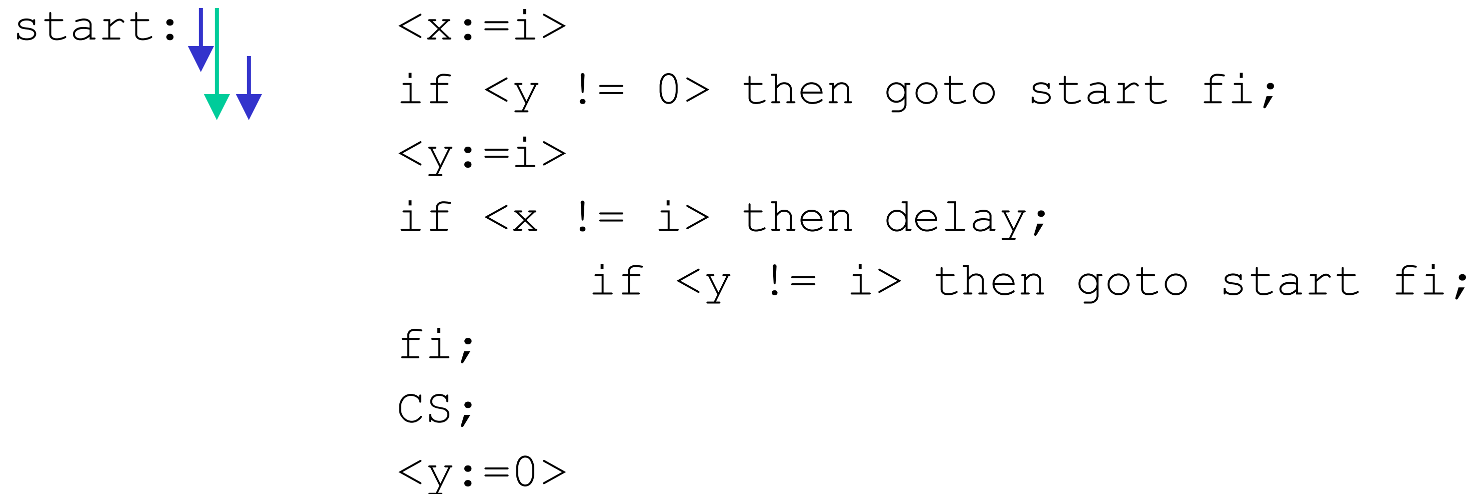
```
start:          <x:=i>
                if <y != 0> then goto start fi;
                <y:=i>
                if <x != i> then delay;
                        if <y != i> then goto start fi;
                fi;
                CS;
                <y:=0>
```

X=1

Y=0

w2-x, w1-x, r1-y

Algorithmus: Problem

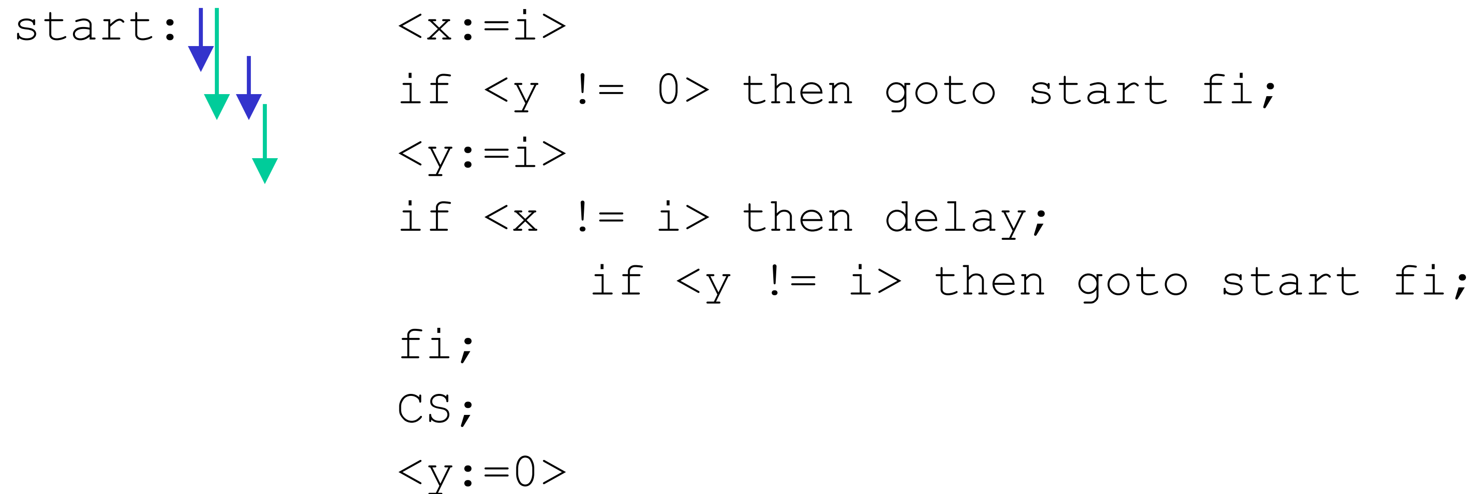


X=1

Y=0

w2-x, w1-x, r1-y, r2-y

Algorithmus: Problem

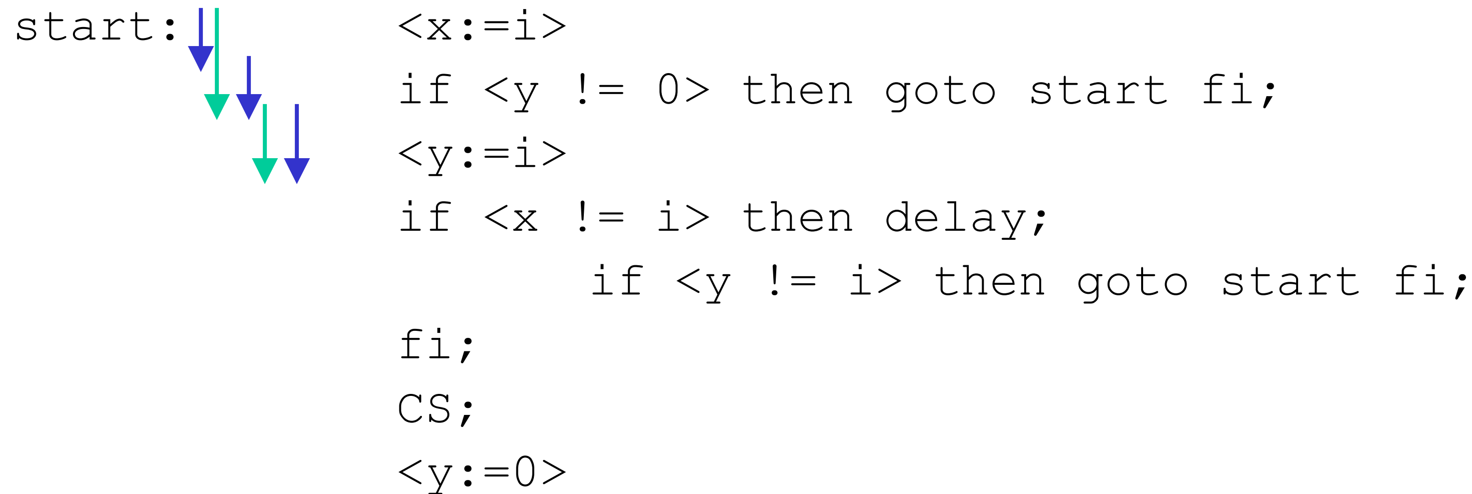


X=1

Y=1

w2-x, w1-x, r1-y, r2-y, w1-y

Algorithmus: Problem

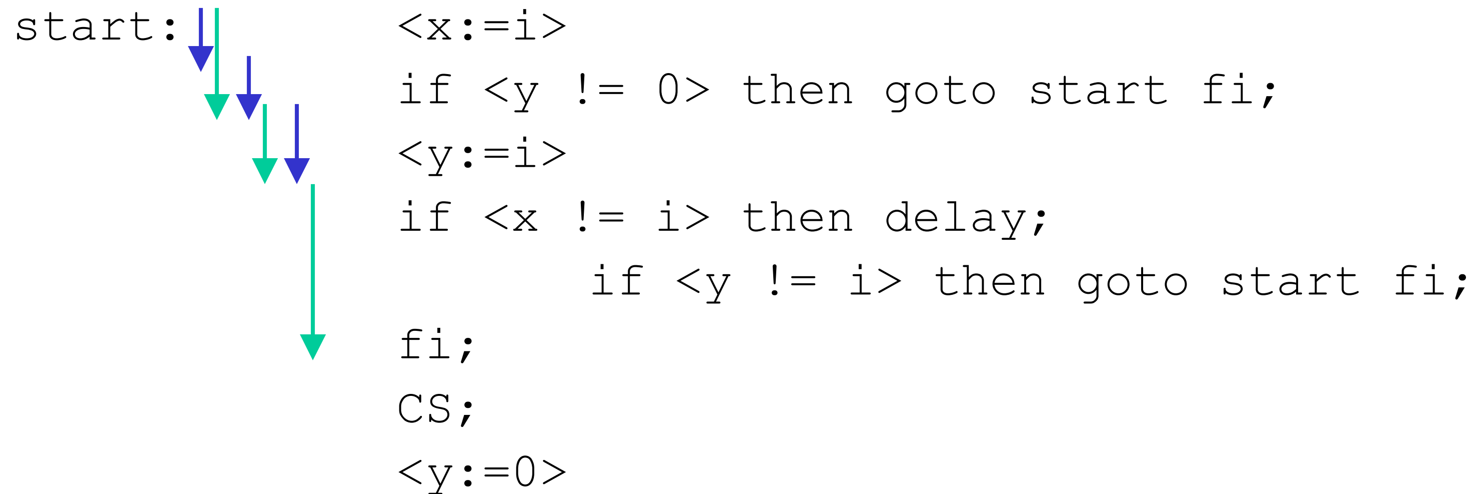


X=1

Y=2

w2-x, w1-x, r1-y, r2-y, w1-y, w2-y

Algorithmus: Problem

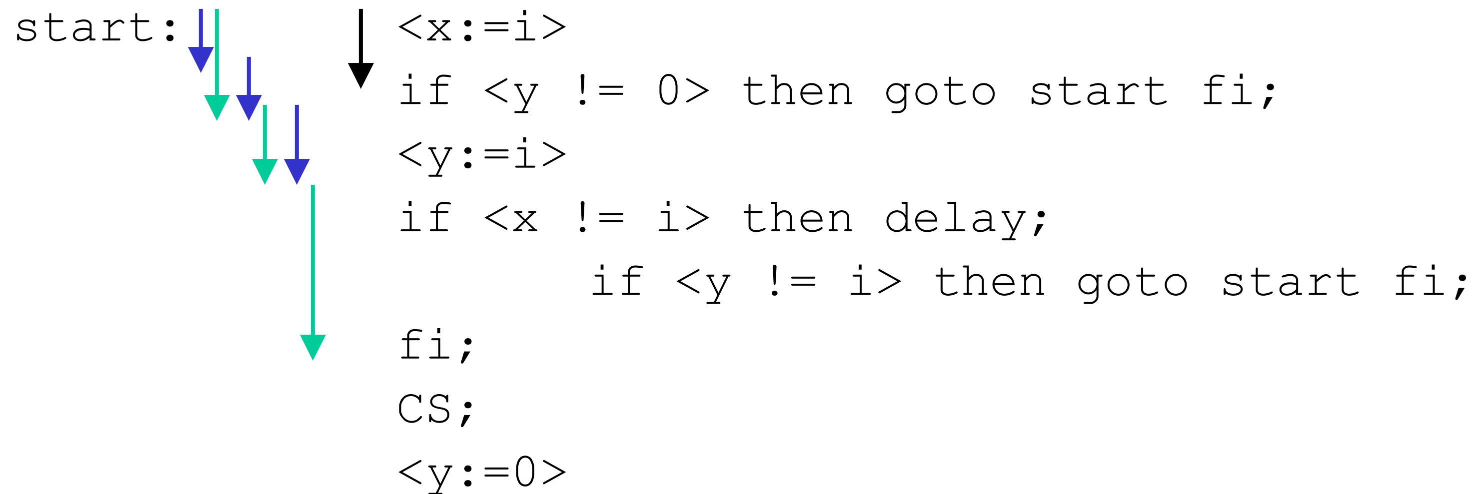


X=1

Y=2

w2-x, w1-x, r1-y, r2-y, w1-y, w2-y, r1-x

Algorithmus: Problem

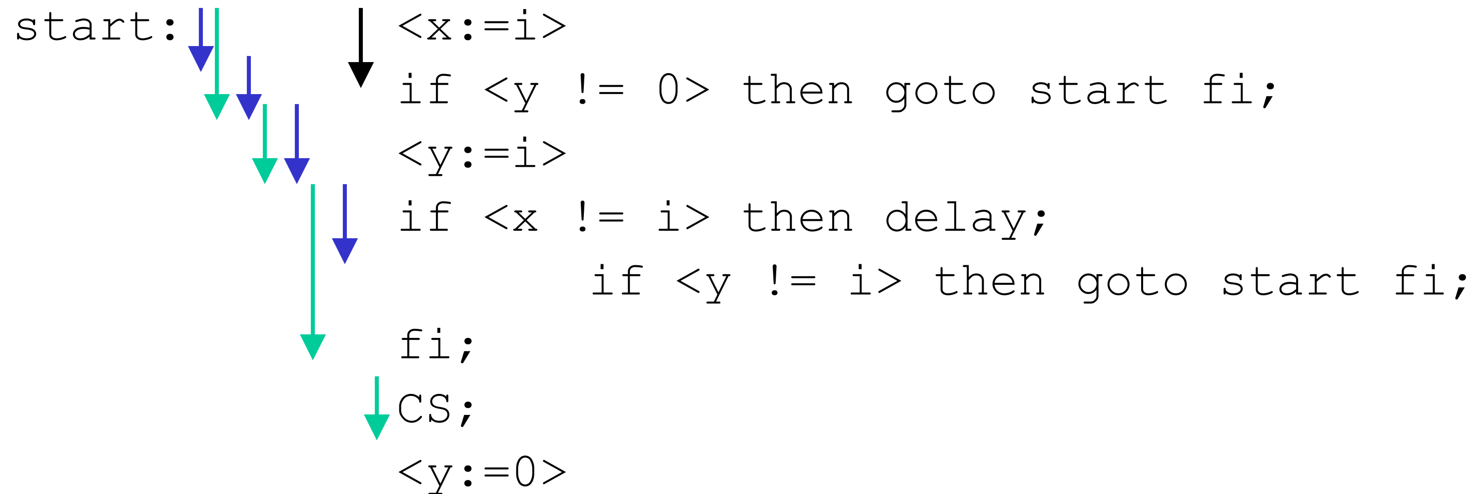


X=3

Y=2

w2-x, w1-x, r1-y, r2-y, w1-y, w2-y, r1-x, w3-x

Algorithmus: Problem



X=3

Y=2

w2-x, w1-x, r1-y, r2-y, w1-y, w2-y, r1-x, w3-x, r2-x

Algorithmus: Problem

- Wenn Prozess 1 in der CS ist, wurden X und Y überschrieben
  - d.h. der Programmstatus ist der gleiche, als wenn 1 niemals ausgeführt worden wäre
  - Prozess 2 merkt, dass eine Konkurrenzsituation da ist, weiß aber nicht, dass 1 in der CS ist

Algorithmus: Problem

- Wenn Prozess 1 in der CS ist, wurden X und Y überschrieben
  - d.h. der Programmstatus ist der gleiche, als wenn 1 niemals ausgeführt worden wäre
  - Prozess 2 merkt, dass eine Konkurrenzsituation da ist, weiß aber nicht, dass 1 in der CS ist
- Außerdem: Keine Annahmen, wie lange ein Prozess in der CS ist

Algorithmus: Problem

- Wenn Prozess 1 in der CS ist, wurden X und Y überschrieben
  - d.h. der Programmstatus ist der gleiche, als wenn 1 niemals ausgeführt worden wäre
  - Prozess 2 merkt, dass eine Konkurrenzsituation da ist, weiß aber nicht, dass 1 in der CS ist
- Außerdem: Keine Annahmen, wie lange ein Prozess in der CS ist
- Lösung: Eine Zusätzliche Variable (Flag), die anzeigt, ob ein Prozess in der CS ist

Algorithmus 2: Lösung

```
start:          <b[i]:=true>;
                <x:=i>;
                if <y != 0> then <b[i]:=false>;
                       await <y=0>;
                       goto start fi;
                <y:=i>;
                if <x != i> then <b[i]:=false>;
                       for j:=1 to N do await <!b[j]> od;
                       if <y != i> then await <y=0>;
                              goto start
                       fi;
                fi;
                CS;
                <y:=0>
                <b[i]:=false>;
```

Beweis

1. Mutual Exclusion
2. Deadlock Freiheit

Generischer Algorithmus

```
         α:     <x:=i>
         β:     if <y != 0> then goto α fi;
         γ:     <y:=i>
{P_i^δ}  δ:     if <x != i> then achieve P_i^ε;
        {P_i^ε} ε:     if <y != i> then goto α fi;
                fi;
{P_i^CS}        [ζ:CS];
{P_i^CS} η:     <y:=0>
```

i. Label hinzugefügt
ii. Annahmen in {} -Klammern hinzugefügt
iii. Critical Section in [ ] –Klammern
iv. Ersatz des *delay* durch ein *achieve*

Mutual Exclusion

- Jede Annahme ist mit einem Konrollpunkt verbunden
- Die Annahme P_i^{CS} ist mit jedem Kontrollpunkt innerhalb der CS verbunden
- Wir müssen zwei Dinge beweisen:
  - Sequential Correctness.
  - Interference Freedom.

Die Annahmen im einzelnen

$P_i^{\delta}: \quad x = i \supset y \neq 0$

$P_i^{\varepsilon}: \quad y = i \supset \forall j : \neg(at(\gamma_j) \vee at(\delta_j) \vee in(cs_j))$

$P_i^{CS}: \quad y \neq 0 \wedge \forall j \neq i : [\neg in(cs_j)] \wedge [(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j]$

Es gilt immer: $P_i^{cs} \wedge P_j^{cs} \equiv false$

Sequential Correctnes Proof

```
          α:      <x:=i>
          β:      if <y != 0> then goto α fi;
          γ:      <y:=i>
{P_i^δ}   δ:      if <x != i> then achieve P_i^ε;
             {P_i^ε} ε:     if <y != i> then goto α fi;
                  fi;
{P_i^CS}          [ζ:CS];
{P_i^CS}  η:      <y:=0>
```

$$P_i^\delta: \quad x = i \supset y \neq 0$$

Die Ausführung von γ lässt $P_i^\delta = \text{true}$

γ setzt $y = i, i \neq 0$

Sequential Correctnes Proof

```
          α:     <x:=i>
          β:     if <y != 0> then goto α fi;
          γ:     <y:=i>
{P_i^δ}   δ:     if <x != i> then achieve P_i^ε;
             {P_i^ε} ε:   if <y != i> then goto α fi;
                 fi;
{P_i^CS}         [ζ:CS];
{P_i^CS}  η:     <y:=0>
```

Falls der Test im Statement δ $x = i$ ergibt,

und Prozess i damit in die CS gelangt, dann wird $P_i^{cs} = true$

Sequential Correctnes Proof

```
        α:    <x:=i>
        β:    if <y != 0> then goto α fi;
        γ:|   <y:=i>
{P_i^δ} δ:|   if <x != i> then achieve P_i^ε;
           {P_i^ε} ε:   if <y != i> then goto α fi;
              fi;
{P_i^CS}      [ζ:CS];
{P_i^CS} η:   <y:=0>
```

$P_i^\delta: \quad x = i \supset y \neq 0$

Die Annahme $P_i^\delta = \text{true}$ impliziert, dass $y > 0$

Da $x = i \supset x \neq j \Rightarrow \forall j \neq i, [(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j] = \text{true}$

Sequential Correctnes Proof

```
        α:   <x:=i>
        β:   if <y != 0> then goto α fi;
        γ:   <y:=i>
{P_i^δ} δ:   if <x != i> then achieve P_i^ε;
                 {P_i^ε} ε:   if <y != i> then goto α fi;
             fi;
{P_i^CS}     [ζ:CS];
{P_i^CS} η:  <y:=0>
```

$$P_j^{cs}: \quad y \neq 0 \wedge \forall i \neq j : [\neg in(cs_i)] \wedge [(at(\gamma_i) \vee at(\delta_i)) \supset x \neq i]$$

Angenommen $at(\delta_i) = true \wedge in(cs_j) = true$ bevor i den Test ausführt

$\Rightarrow in(cs_j) = true \Rightarrow P_j^{cs} = true \Rightarrow x \neq i$

$\Rightarrow x \neq i \;\Rightarrow$ i tritt nicht in die CS ein.

Sequential Correctnes Proof

```
            α:   <x:=i>
            β:   if <y != 0> then goto α fi;
            γ:   <y:=i>
{P_i^δ}     δ:   if <x != i> then achieve P_i^ε;
         {P_i^ε} ε:   if <y != i> then goto α fi;
                 fi;
{P_i^CS}         [ζ:CS];
{P_i^CS}    η:   <y:=0>
```

$$P_i^{\varepsilon}: \quad y = i \supset \forall j : \neg(at(\gamma_j) \vee at(\delta_j) \vee in(cs_j))$$

Da das achieve P_i^{ε} terminiert $\Rightarrow P_i^{\varepsilon} = true$

Sequential Correctnes Proof

```
        α:    <x:=i>
        β:    if <y != 0> then goto α fi;
        γ:    <y:=i>
{P_i^δ} δ:    if <x != i> then achieve P_i^ε;
        {P_i^ε} ε:    if <y != i> then goto α fi;
              fi;
{P_i^CS}      [ζ:CS];
{P_i^CS} η:   <y:=0>
```

Falls der Test y = i ergibt, gilt also

$P_i^{\varepsilon}: \quad y = i \supset \forall j : \neg(at(\gamma_j) \vee at(\delta_j) \vee in(cs_j))$

y = i heißt insbesondere $y \neq 0$

$P_i^{CS}: \quad y \neq 0 \wedge \forall j \neq i : [\neg in(cs_j)] \wedge [(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j]$

Sequential Correctnes Proof

```
         α:   <x:=i>
         β:   if <y != 0> then goto α fi;
         γ:   <y:=i>
{P_i^δ}  δ:   if <x != i> then achieve P_i^ε;
     {P_i^ε} ε:   if <y != i> then goto α fi;
              fi;
{P_i^CS}      [ζ:CS];
{P_i^CS} η:   <y:=0>
```

$$P_i^{cs}: \quad y \neq 0 \wedge \forall j \neq i : [\neg in(cs_j)] \wedge [(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j]$$

Bei der Ausführung der CS bleibt $P_i^{cs} = true$

Interference Freedom Proof

$$P_i^{\delta}: \quad x = i \supset y \neq 0$$

Prozess i ist der einzige, der $x = i$ setzt.

Daher kann Prozess j P_i^{δ} nur im Fall $y = 0$ fälschen.

Dies passiert nur im Statement η.

Da die Annahme P_j^{CS} wahr sein muss, falls j η ausführt, gilt:

Falls Prozess i am Punkt δ ist, dann ist $x \neq i$, in welchem Fall das setzen von y auf 0 die Annahme P_i^{δ} nicht verfälscht.

Interference Freedom Proof

$$P_i^{\varepsilon}: \quad y = i \supset \forall j : \neg(at(\gamma_j) \vee at(\delta_j) \vee in(cs_j))$$

Prozess i ist der einzige, der y = i setzt.

Prozess j kann die Annahme nur beim Erreichen der Punkte γ oder δ oder dem Eintritt in die CS bei y = i fälschen.

γ und δ können jedoch nur nach β erreicht werden, in welchem dafür y = 0 sein muss.

Die CS kann nur nach dem Test in ε ausgeführt werden, indem y = j gefunden wird.

Diese beiden Möglichkeiten scheiden jedoch aus, wenn y = i ist.

Interference Freedom Proof

$$P_i^{cs}: \quad y \neq 0 \wedge \forall j \neq i : [\neg in(CS_j)] \wedge [(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j]$$

Die Annahme P_i^{CS} erlaubt nicht, dass ein anderer

Prozess am Punkt η ist. $\Rightarrow$ es gilt immer $y \neq 0$.

$in(cs_j) = true$ nur in folgenden Fällen möglich:

(i) der Test in δ ergibt $x = j$

(ii) der Test in ε ergibt $y = j$

(i) ist nicht möglich, da wegen P_i^{CS} gilt: falls $at(\delta_j) \Rightarrow x \neq j$

(ii) ist nicht möglich, da wegen P_j^{ε} gilt: falls $y = j \Rightarrow in(cs_i) = false$

Interference Freedom Proof

$$P_i^{cs}: \quad y \neq 0 \wedge \forall j \neq i : [\neg in(CS_j)] \wedge [(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j]$$

Zuletzt muss gezeigt werden, dass j nicht

$(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j$ verfälscht.

Dies könnte nur im Punkt γ erreicht werden, welcher nach β kommt.

In β müsste dafür aber $y = 0$ gefunden werden.

Dies ist nicht möglich, da $P_i^{CS} \rightarrow y \neq 0$.

Mutual Exclusion Algorithmus 2

```
start:          <b[i]:=true>;
        α:      <x:=i>;
        β:      if <y != 0> then <b[i]:=false>;
                       await <y=0>;
                       goto start fi;
        γ:      <y:=i>;
        δ:      if <x != i> then <b[i]:=false>;
                       for j:=1 to N do await <!b[j]> od;
            ε:         if <y != i> then await <y=0>;
                              goto start
                       fi;
                fi;
        ζ:      CS;
        η:      <y:=0>
                <b[i]:=false>;
```

Mutual Exclusion Algorithmus 2

z.Z. ist also:

falls i $at(\varepsilon_i)$ und $y = i \Rightarrow$ j ist nicht zwischen γ und η.

Falls y = i bei ε_i, dann auch in der for-Schleife vorher.

Es ist jedoch b[j] = true, falls j zwischen γ und η ist.

Um ε zu erreichen, muss b[j] jedoch false sein.

Daher kann j nicht zwischen γ und η sein, wenn y = i.

Deadlock Freedom Proof

S. $y = i \neq 0 \supset (in(\delta_i) \vee in(cs_i))$

- Beweis durch temporale Logik:
  - P = true, wenn P irgendwann true wird und dann für immer true bleibt
  - P⇝Q = true, wenn P=false ist (wird), oder Q=true ist oder wird.

Deadlock Freedom Proof

z.Z. ist $at(\alpha_i) \rightsquigarrow \exists j : in(cs_j)$

Annahmen: $at(\alpha_i) = true$ und $\Box(\forall j : \neg in(cs_j)) = true$

T. Das achieve Statement terminiert für $\Box(y = i \wedge \forall j : \neg in(cs_j)) = true$

$at(\alpha_i) \rightsquigarrow y \neq 0$ Im Statement β findet i entweder $y \neq 0$, oder setzt $y = i$ in γ

$\rightsquigarrow$

$y \neq 0 \supset \Box y \neq 0$ Wenn $y \neq 0$ ist, dann wird es erst durch Statement η auf null gesetzt. Statement η kann jedoch wegen $\Box \forall j : \neg in(cs_j)$ nicht ausgeführt werden.

Deadlock Freedom Proof

$\rightsquigarrow$

$(\Box y \neq 0) \rightsquigarrow \exists j : \Box y = j$ Wenn $y \neq 0$ ist, kann Statement γ nicht mehr erreicht werden.

Der letzte Prozess $at(\gamma_j)$ setzt $y = j$

$\rightsquigarrow$

$(\Box y = j) \rightsquigarrow at(\varepsilon_j)$ Wegen Invariante S gilt: $y = j \Rightarrow in(cs_j) \vee in(\delta_j)$

Wegen der Annahme $\neg in(cs_j) \Rightarrow in(\delta_j) = true \wedge x \neq j$.

Wegen Annahme T muss j irgendwann ε erreichen.

$\rightsquigarrow$

$(\Box y = j \wedge at(\varepsilon_j)) \rightsquigarrow false$ Process j führt eventuell den Test in ε aus und findet $y = j$

Process j tritt dann in die CS ein, was ein wiederspruch zur Annahme $\Box \neg in(cs_j)$

Deadlock Freedom Algorithmus 2

```
δ:    if <x != i> then <b[i]:=false>;
                for j:=1 to N do await <!b[j]> od;
```

Zu Beweisen ist, dass Annahme T korrekt ist, also

$\square(y = i \wedge \forall j : \neg in(cs_j))$ impliziert, dass die for-Schleife eventuell terminiert.

Da b[j] eventuell false wird und bleibt, terminiert die for-Schleife.

Vielen Dank für die Aufmerksamkeit