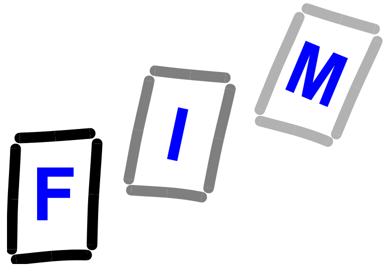
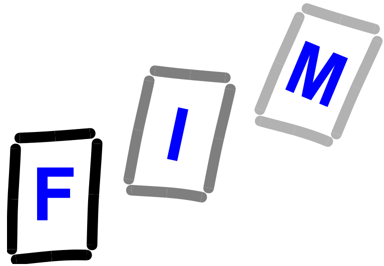



# Betriebssysteme

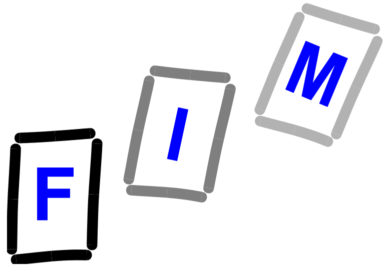
## G: Parallele Prozesse

### (Teil B: Klassische Problemstellungen, Mutual Exclusion, kritische Regionen)



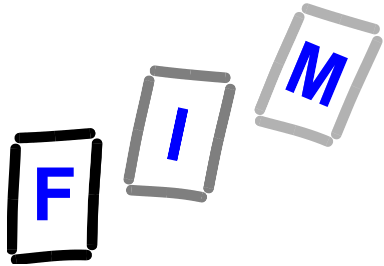

# Allgemeine Synchronisationsprobleme

- **Wir verstehen ein BS als eine Menge von parallel laufenden Systemprozessen.**
- **Diese Prozesse arbeiten auf gemeinsamen Daten (engl.: shared data).**
- **Das führt für parallele Prozesse zu besonderen Problemen bzw. Aufgaben:**
  - ➡ **Synchronisation**
  - ➡ **Wechselseitiger Ausschluss**



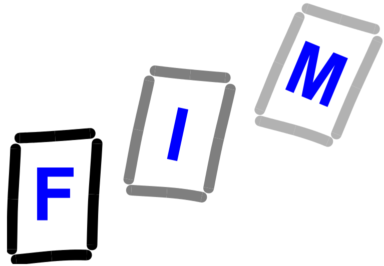

# Erzeuger-Konsumenten Problem

- **Bezeichnungen:**
  - „Bounded Buffer“: Puffer mit endlicher Kapazität
  - „Producer / Consumer Problem“
- **Betrachten wir eine Anwendung abstrakt:**
  - Produzent P erzeugt Daten und stellt sie in einen Puffer
  - Konsument K nimmt die Einträge aus dem Puffer heraus

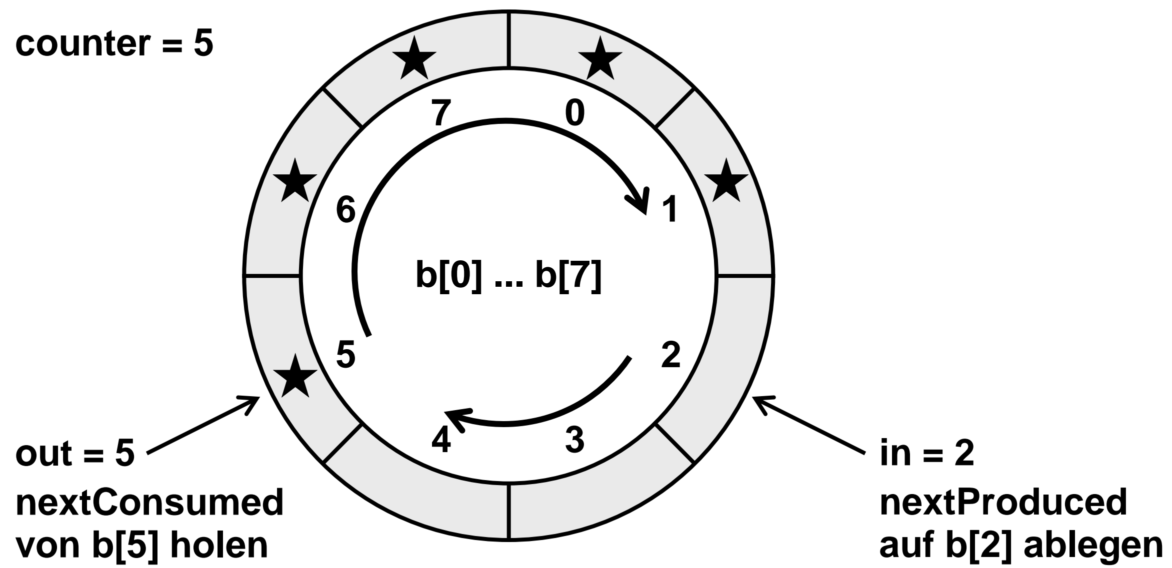


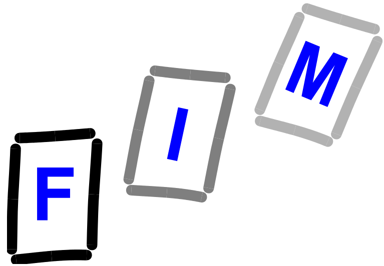

# Ringpuffer

```
#define BUFFER_SIZE 8

typedef struct {...} item;

item b[BUFFER_SIZE ]; // Puffer
int in = 0;
int out = 0;
counter = 0;
```




# Ring - Puffer



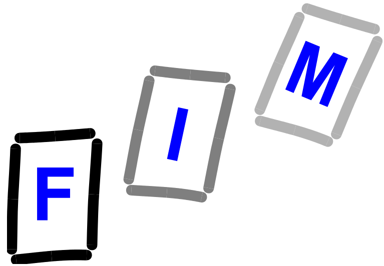



# Grundsätzliche Überlegungen

- Die Variable *in* zeigt auf den nächsten freien Platz im Puffer *b*.
- Die Variable *out* zeigt auf die erste belegte Position (so diese existiert)
- Zustand bei

  leerem Puffer *b*: *counter = 0*

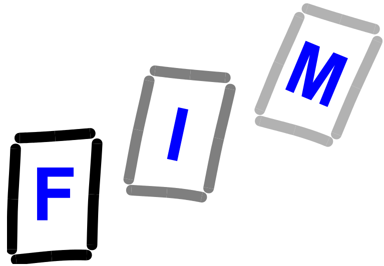
  vollem Puffer *b*: *counter = BUFFER_SIZE*




# Code für Produzent

- Produzent verwendet die lokale Variable ***nextProduced*** vom Typ ***item***

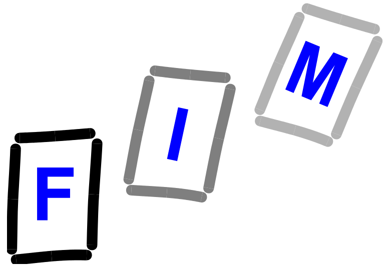
```
b[in] = nextProduced;
in = (in+1) % BUFFER_SIZE;
counter++;
```



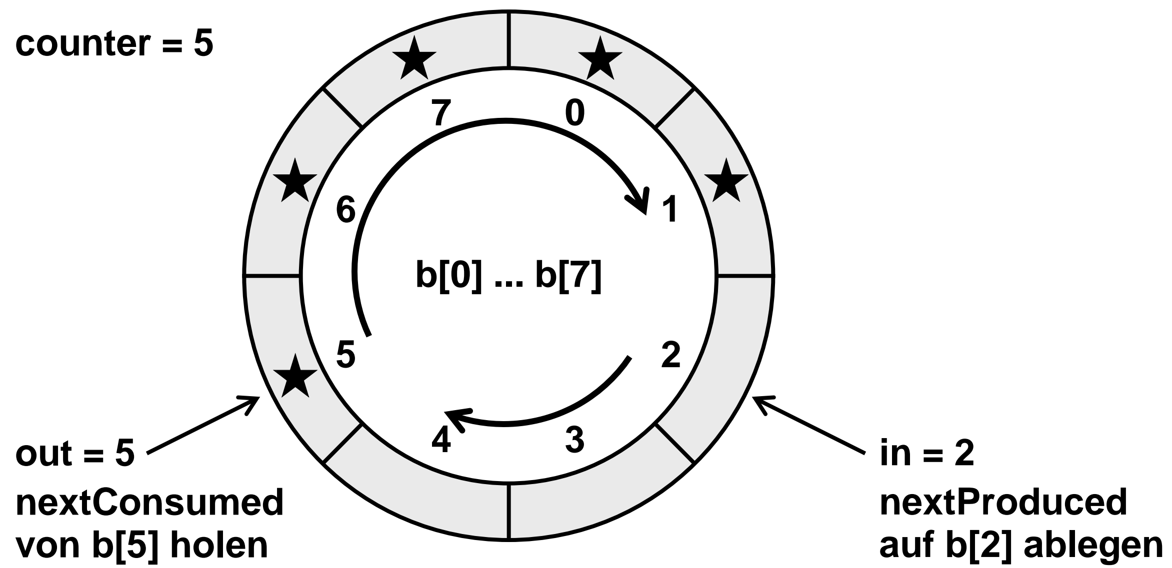

# Code für Konsument

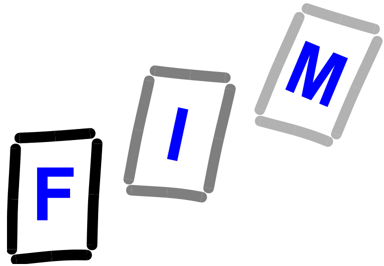
- **Konsument verwendet die lokale Variable *nextConsumed* vom Typ *item***

```
nextConsumed = b[out] ;
out = (out+1) % BUFFER_SIZE;
counter- - ;
```



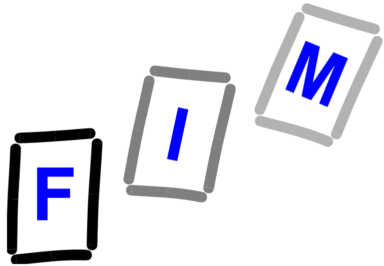

# Ring - Puffer





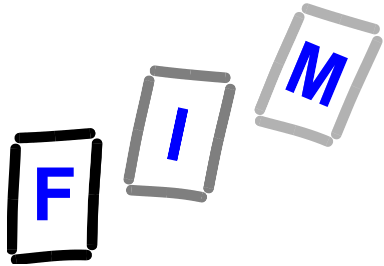

# Zusatzbedingungen

- **Kapazität des Puffer *b* ist begrenzt.**
  - ➡ P darf nicht in den vollen Puffer einfügen
  - ➡ C kann nicht aus dem leeren Puffer Einträge entnehmen
  - ➡ Einträge (Nachrichten) sollen in der gleichen Reihenfolge entnommen werden, wie sie produziert (gesendet) wurden
- **Puffer *b* glättet Geschwindigkeitsschwankungen von / zwischen P und C.**




# Busy waiting? ("Beschäftigtes Warten")

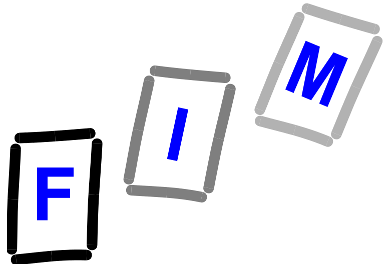
- P darf nicht in vollen Puffer schreiben
  while (counter == BUFFER_SIZE) ; // no-op
- C kann nicht aus leerem Puffer lesen
  while (counter == 0 ); // no-op
- Problem: P und C verbrauchen CPU-Zeit während des Wartens
  - ➡ Lösung durch Strategie beim Scheduling,
  - ➡ oder spezielles „Aktivierungs- / Aufweck-Konzept" (engl.: wake-up) notwendig.




# Ein Problem bleibt!

Obgleich sowohl Produzent P als auch Konsument C (für sich alleine) korrekt arbeiten, gibt es ein Problem, wenn beide gemeinsam parallel laufen.

```
while (1) {
      while (counter == 0 );
      nextConsumed = b[out] ;
      out = (out+1) % BUFFER_SIZE;
      counter- - ;
}
```




# Gemeinsame (engl.: shared) Daten!

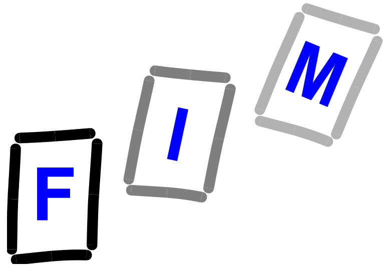
- „counter“ wird von P *und* C verwendet!!
  - P: counter = counter + 1;
  - C: counter = counter - 1;

**P**

$reg_1$ = counter; — p1
$reg_1$ = $reg_1$ + 1; — p2
counter = $reg_1$; — p3

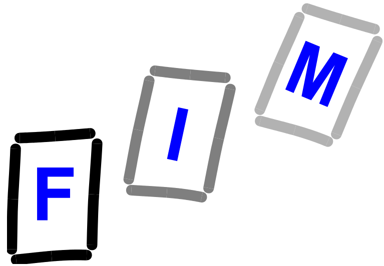
**C**

$reg_2$ = counter; — c1
$reg_2$ = $reg_2$ - 1; — c2
counter = $reg_2$; — c3




# Beispiel

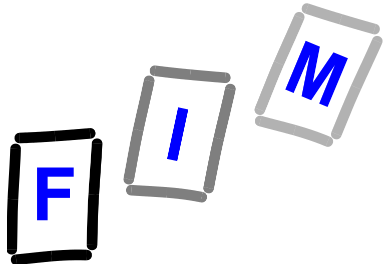
- Status am Anfang (Annahme):
  counter = 5
- Operationen:
  P erzeugt einen Eintrag
  C konsumiert einen Eintrag
- Erwartetes Ergebnis nach Ausführung:
  counter = 5




# Beispiel für "verzahnten" Ablauf

| Op | Operation | $reg_1$ | $reg_2$ | counter |
|---|---|---|---|---|
| | Anfangszustand | . | . | 5 |
| p1 | $reg_1$ = counter | 5 | . | 5 |
| p2 | $reg_1 = reg_1 + 1$ | 6 | . | 5 |
| c1 | $reg_2$ = counter | 6 | 5 | 5 |
| c2 | $reg_2 = reg_2 - 1$ | 6 | 4 | 5 |
| p3 | counter = $reg_1$ | 6 | 4 | 6 |
| c3 | counter = $reg_2$ | 6 | 4 | 4 |

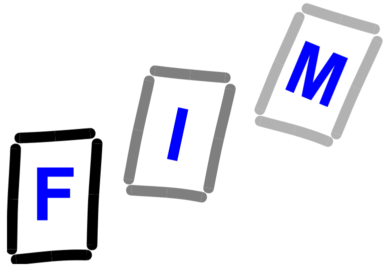
Könnte auch 6 (oder 5) sein!




# „Wettlaufbedingung“

**Wenn mehrere parallele Prozesse auf shared data auch modifizierend zugreifen und das Ergebnis von der jeweiligen Zugriffsreihenfolge oder relativen Geschwindigkeit der Prozesse abhängt, so nennt man diese Situation eine**

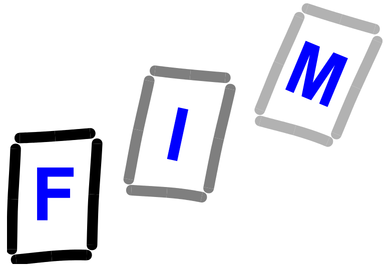
***race condition***




# Wechselseitiger Ausschluss

**Problem der „race condition“beim Producer/Consumer (shared counter!):**

- ➡ **Erfordert eine entsprechende Synchronisation der parallelen Prozesse**
- ➡ **Zu gewährleisten, dass *zu jedem Zeitpunkt* nur *ein Prozess* die Variable *counter* ändert**
  - » **Es gibt einen Codebereich in C und P, in dem *beide* Prozesse die gemeinsame (engl.: shared) Variable *counter* ändern können.**
  - » **Weder P noch C dürfen diesen „kritischen Bereich“ ausführen, während der jeweils andere Prozess dies (bereits) macht: Wechselseitiger Ausschluss**




# Kritischer Abschnitt (engl.: critical region)

**Eine Folge von Operationen auf gemeinsame (engl.: shared) Daten, welche sich beim Zugriff wechselseitig zeitlich ausschließen müssen, heißt**

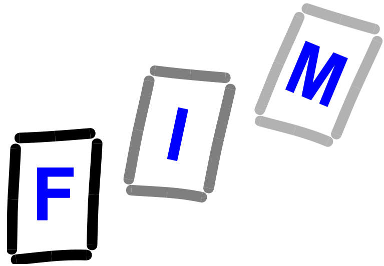
## CR = Kritischer Abschnitt

**(engl.: *critical region*/critical section wir verwenden das Kürzel *CR*).**

**Die Ausführung eines kritischen Bereiches ist nur immer einem Prozess zu einem Zeitpunkt möglich**

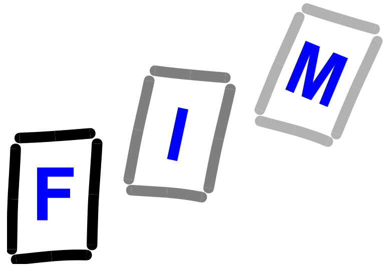
**=**

## Wechselseitiger Ausschluss



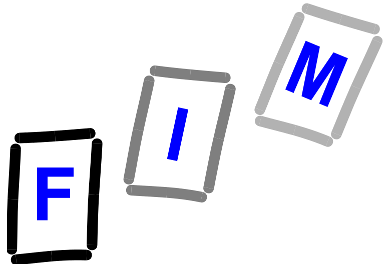

# CR schematisch

```
do {
        ............................
    EINTRITT IN CR
      KRITISCHER ABSCHNITT
    AUSTRITT AUS CR
        ............................
} while (1)
```



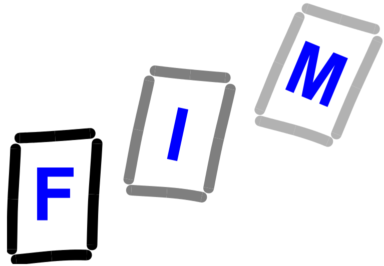

# Anforderungen an eine Lösung

- **Wechselseitiger Ausschluss (Mutual exclusion)**
  **Höchstens ein Prozess ist in einer CR.**
- **Fortschritt (Progress)**
  **P bleibt nur für eine begrenzte Zeit in der CR.**
  **Kein Prozess außerhalb seiner CR darf einen anderen am Eintreten in eine solche blockieren.**
- **Beschränktes Warten (Fairness)**
  **Wenn P eine CR betreten will, so wird dies innerhalb endlicher Zeit gestattet.**
- **Keine Annahmen (No assumptions)**
  **Relative Geschwindigkeit der Prozesse und Anzahl der Prozesse bzw. Prozessoren ist beliebig**



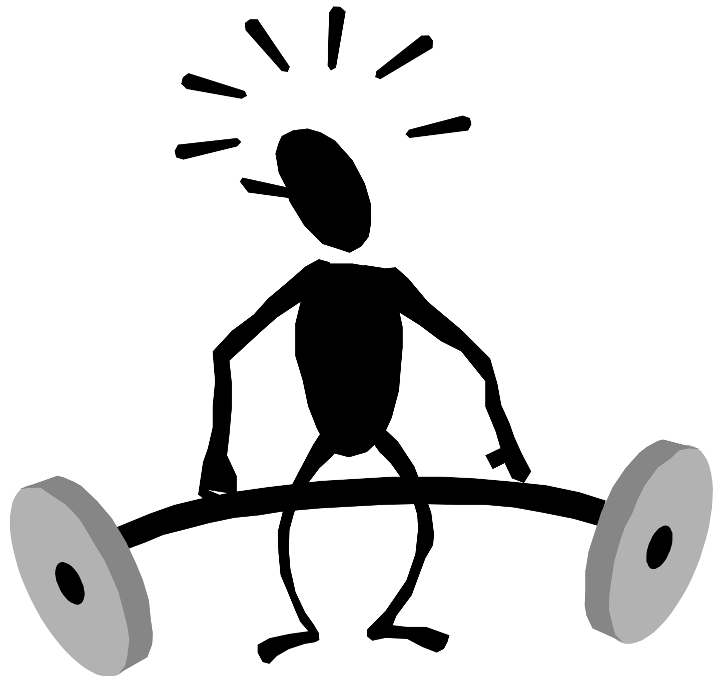

# Klassiker!

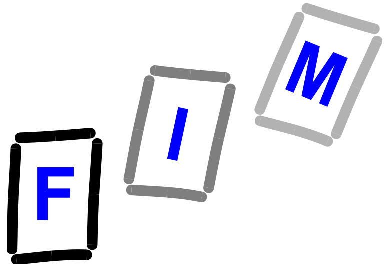
- E. Dijkstra zeigte in seiner „klassischen“ Veröffentlichung (1965) „Cooperating Sequential Processes“, welche typischen Probleme es gibt:
  - ➡ Verletzung von Mutual Exclusion
  - ➡ Erzwungene Reihenfolge
  - ➡ Verletzung von Progress (vgl.: deadlock)




# Ein (falsches) Beispiel

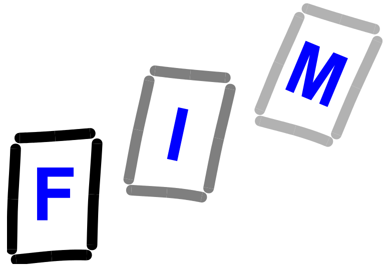
- Der folgende Algorithmus erfüllt „gegenseitigen Ausschluss“.
- Aber: Er unterstützt NICHT die Fortschritt-Bedingung
  - ➡ Lösung führt zu Verklemmung (engl.: deadlock)
  - ➡ „Beschäftigtes Warten“ (engl.: busy waiting) innerhalb der WHILE-Schleife





# Wechselseitiger Ausschluss, aber Verklemmung (1)

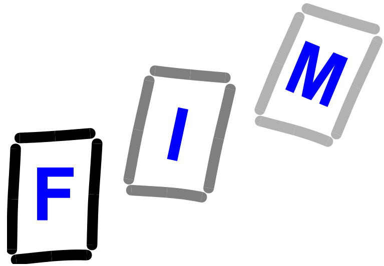
```
boolean flag [2];
flag[0]=false; flag[1]=false;
      COBEGIN
(*P0*)
do {
  flag[0]=true;
  while flag[1]; //no-op
  // Kritischer Bereich
   flag[0]=false;
}

(*P1*)
do {
  flag[1]:=true;
  while flag[0]; //no-op
  //Kritischer Bereich
   flag[1]=false;
}
      COEND
```




# Wechselseitiger Ausschluss, aber Verklemmung (2)

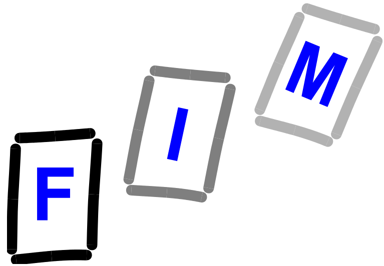
- Wechselseitiger Ausschluss erfüllt
- $P_0$ setzt zuerst flag[0] auf true und zeigt damit, dass er bereit ist, die CR zu betreten:

```
do {
  flag[0]=true;
```



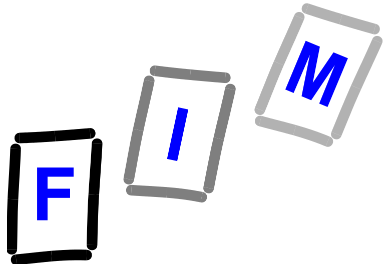

# Wechselseitiger Ausschluss aber Verklemmung (3)

- $P_0$ wartet dann bis flag[1] auf false steht.
  while flag[1]; //no-op
- Das ist erfüllt, sobald $P_1$ die CR verlassen hat.
- Beim Verlassen des CR setzt $P_1$ das flag[1] auf *false* und erlaubt damit dem anderem Prozess, die CR zu betreten:
  flag[0]=false;




# Wechselseitiger Ausschluss aber Verklemmung (4)

- Ist das nicht gleich wie im „counter“ Beispiel des Ringbuffers?
- Nein!
  - ➡ P1 liest flag[1] und schreibt flag[0]
  - ➡ P2 schreibt flag[1] und liest flag[0]
    - » Beides sind unterschiedliche Speicherzellen!
  - ➡ Aber: P1 und P2 (lesen &) schreiben auf counter
    - » Das ist dieselbe Speicherzelle!




# Wechselseitiger Ausschluss aber Verklemmung (5)

**ABER,**
**nehmen wir folgende Verzahnung an**

**T0: P0 setzt flag[0] =true;**

Hier Interrupt, CPU schaltet von P0 auf P1

**T1: P1 setzt flag[1] =true;**

**JETZT laufen P0 und P1 im while-loop ewig.**
**Die „Fortschritt-Bedingung“ ist nicht erfüllt!**