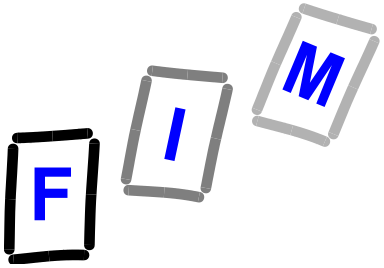
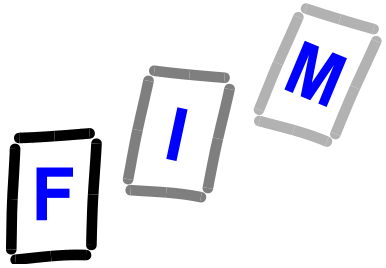


Betriebssysteme

G: Parallele Prozesse

(Teil C: SpinLock, Semaphore, Monitore)

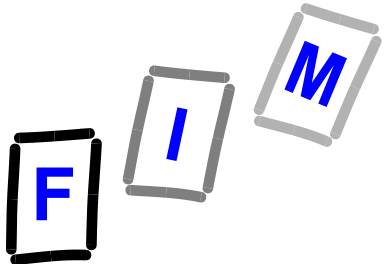


Hardwareunterstützung Uniprozessor-System

- **Verbiete Interrupts während des Aufenthalts in einer CR**

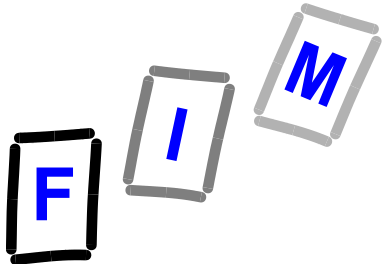
```
disable interrupt
  CR(bzw: Zugriff auf shared variable)
enable interrupt
```

- **Diese Lösung ist in einer Umgebung mit mehreren Prozessoren nicht zweckmäßig**
  - ➡ **Man müsste die Interrupt - Register *aller* Prozessoren gleichzeitig sperren**
  - ➡ **und sicherstellen, dass andere Prozesse nicht in die CR gelangen, sie also u.U. sogar anhalten**
    - » **Scheduler deaktivieren!**
- **Und bei einer Endlosschleife in der CR?**



Hardwareunterstützung Multiprozessor-System

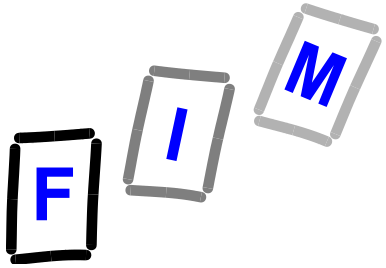
- **Moderne Architekturen stellen eine spezielle „TestAndSet“-Anweisung zur Verfügung.**
  - ➡ **Die ist ein unteilbarer (engl.: atomic) Befehl auf Hardwareebene**
  - ➡ **Ein Registerinhalt kann innerhalb eines Instruktionszyklus ohne jegliche Unterbrechung getestet und gesetzt werden**
  - ➡ **Werden zwei „TestAndSet“-Anweisungen gleichzeitig von zwei CPUs aktiviert, so laufen sie sequentiell in einer beliebigen / zufälligen Reihenfolge ab**
    - » **Aber niemals gleichzeitig/verzahnt!**
    - » **Wird durch spezielle Hardware-Kommunikation garantiert**



TestAndSet (Pseudo-Code)

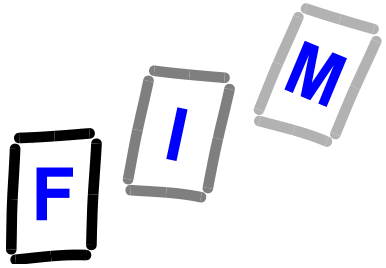
- **Unteilbarkeit wird hier vorausgesetzt!**

```
boolean TestAndSet (boolean flag)
{  /* versuche, flag von true auf false zu setzen */
   /* return: ob Wechsel von true auf false stattfand */
   if (flag) {
         flag = false;
         return true;
   } else {
         return false;
   }
}
```



TestAndSet C-Variante

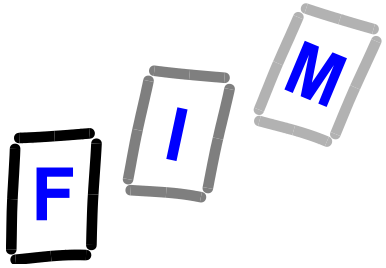
```
int TestAndSet (int  *flag) {
// mit  *call by reference
// 0 = false, 1 = true
// retval=1, wenn Wechsel von 1 auf 0 stattfand
   int retval = *flag;
   *flag = 0;
   return retval;
}
//Unteilbarkeit angenommen
```



Lösung für kurze CR mit busy waiting

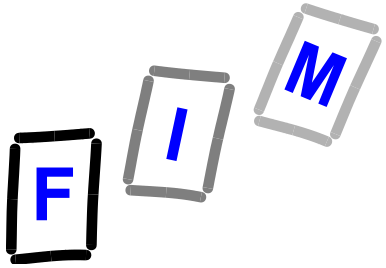
```
boolean free;
free = true;
...
do {
   while (!TestAndSet(free)) ; /*busy waiting */
   { Critical Region }
   free = true;
   { Code außerhalb der CR }
} while (1);
```

Achtung Zusatzannahme:
Diese Zuweisung ist atomar!



Lösung in JAVA mit Klasse util.concurrent.atomic.AtomicBoolean

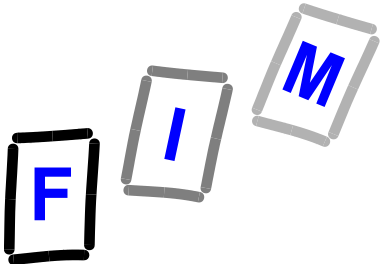
```
import  java.util.concurrent.atomic.AtomicBoolean;

class AtomicBoolean free = new AtomicBoolean(true);
……..
do {
while (!free.getAndSet(false));
        { Critical Region CR }
        free.set(true);
        { Outside of CR }
} while (true);
```



Sperrvariablen (engl.: spin lock)

- **Sperrvariablen können als eine besondere Art von „schwachen“ Semaphoren angesehen werden**
  - ➡ **Semaphore werden später besprochen**

- **Eine Sperrvariable stellt folgende Funktionen für die Variable S zur Verfügung:**
  - ➡ **Eine unteilbare Operation zum Dekrementieren: „lock“**
  - ➡ **Eine unteilbare Operation zum Inkrementieren: „unlock“**
  - ➡ **Beim Warten wird „busy waiting“ verwendet: mit lock(S) wartet Prozess bei S<=0**

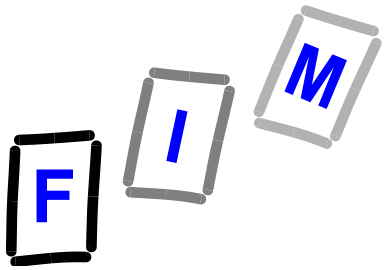


Spin lock

- **Der Name leitet sich aus Folgendem ab: Während (vor einer CR) gesperrt wird, führt die CPU weiterhin ihre Zyklen aus („spins“) und das verweist auf beschäftigtes Warten (busy waiting)**
- **Beachte auch: Beim spin lock ist nicht gewährleistet, dass ein vor einer CR wartender Prozess auch fair bedient wird, also nach endlichem Warten die CR betreten darf!**

Spin lock

- **Praktische Verwendung:**
- **Intern in Betriebssystemen, wenn**
  - ➡ **man davon ausgehen kann, dass die Critical Region ohnehin sehr bald frei wird**
    - » **Nur sehr kurzer Code wird damit abgesichert**
    - » **Prozesswechsel hin & zurück wäre länger als Wartezeit**
  - ➡ **sich nur wenige Prozesse darum streiten bzw. der Blockierungsfall nur selten eintritt**
  - ➡ **es mehrere CPUs gibt**
    - » **Einzelne CPU: P1 in CR → Taskwechsel → P2 wartet im Spinlock, obwohl der ohne neuen Taskwechsel (zurück zu P1) sicher nie frei werden wird!**
    - » **„Adaptiver Mutex“: Spinlock für Warten auf von derzeit laufendem Thread benutzter Ressource, ansonsten Taskswitch (Ressource in Besitz von schlafendem/wartendem Thread)**

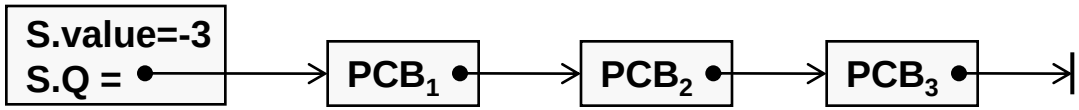


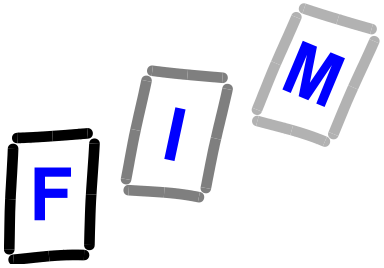
Sperrvariablen und CR

- Sei “flag”die Sperrvariable
  - ➡ flag ist „shared“; wird mit 1 (true) initialisiert
  - ➡
```
lock(flag);
        CR    // kritische Region
unlock (flag);
// weiteres Codestück
```
- Nachteil: „busy waiting“
- Nur bei „kurzen“ CR zweckmäßig
  - ➡ Bzw. wenn zu vermuten ist, dass nur sehr kurz gewartet werden muss (2*Kontextswitch würde länger dauern)

Semaphore

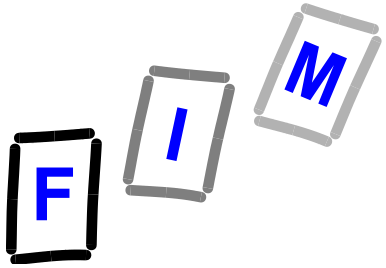
- **Erweiterung um eine Warteschlange Q**
  - ➡ **Sobald ein Prozess P warten muss, wird er in Q eingereiht**
  - ➡ **Damit wechselt der Zustand von P von „running“ zu „waiting“**
  - ➡ **Wenn ein P_j aus Q genommen wird, ändert sich sein Zustand von „waiting“ zu „ready“ (oder gleich zu rechnend, hängt vom Scheduling Konzept ab)**





Semaphore

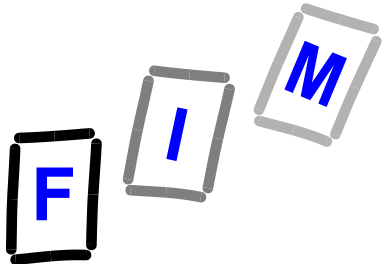
- Eine **Semaphore S** ist ein systemnahes Hilfsmittel zur Synchronisation:
- S ist eine Integer-Variable, auf die neben der Initialisierung S.Init(n) ausschließlich durch zwei **atomare** Operationen zugegriffen wird:
  - ➡ **Wait** („p-operation, p = proberen“)
  - ➡ **Signal** („v-operation, v = verhogen“)
- Zu S ist eine **Warteschlange** Q zugeordnet.



Semaphore-Operationen: Verbale Beschreibung

S.Init(n)

- **Semaphore S wird mit dem Wert n initialisiert.**
- **Falls S für die Implementierung eines kritischen Bereichs (CR) verwendet wird: init(S,1) → S=1**
- **Spezialfall: *Binäre Semaphore:* S.Init(true) bzw. S.Init(false)**



Semaphore-Operationen: WAIT

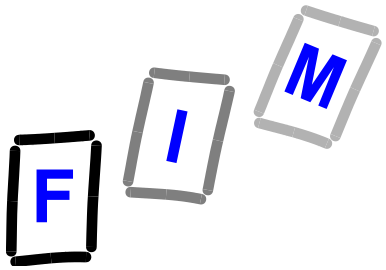
S.Wait()

- Prozess P prüft ob S > 0.
- Ist dies der Fall, wird (unteilbar, atomar!) S=S-1 ausgeführt und P läuft ungehindert weiter.
- Andernfalls: P „legt sich schlafen“
  - ➡ P.Status := wartend
  - ➡ P wird in die zu S gehörende Warteschlange Q eingetragen
- Beachte: war bei P: S.Wait() der Wert S=1, so gilt nachher: S=0
- Vergleiche Implementierungsvariante später

Semaphore-Operationen: SIGNAL

S.Signal() (Teil 1 von 2)

- Der Prozess P prüft damit, ob ein oder mehrere Prozesse in der zu S gehörenden Warteschlange Q „schlafen“.
- Wenn dies der Fall ist:
  Ein Prozess P_Q wird aus Q ausgewählt und in den Zustand „bereit“ versetzt
  - ➡ „Aufwecken“ von P_Q, P_Q.Status := ready
  - ➡ Die Auswahl aus Q muss fair sein, um das Problem Verhungern (Starvation) zu vermeiden



Semaphore-Operationen: SIGNAL

S.Signal() (Teil 2 von 2)

- P_Q.state := ready

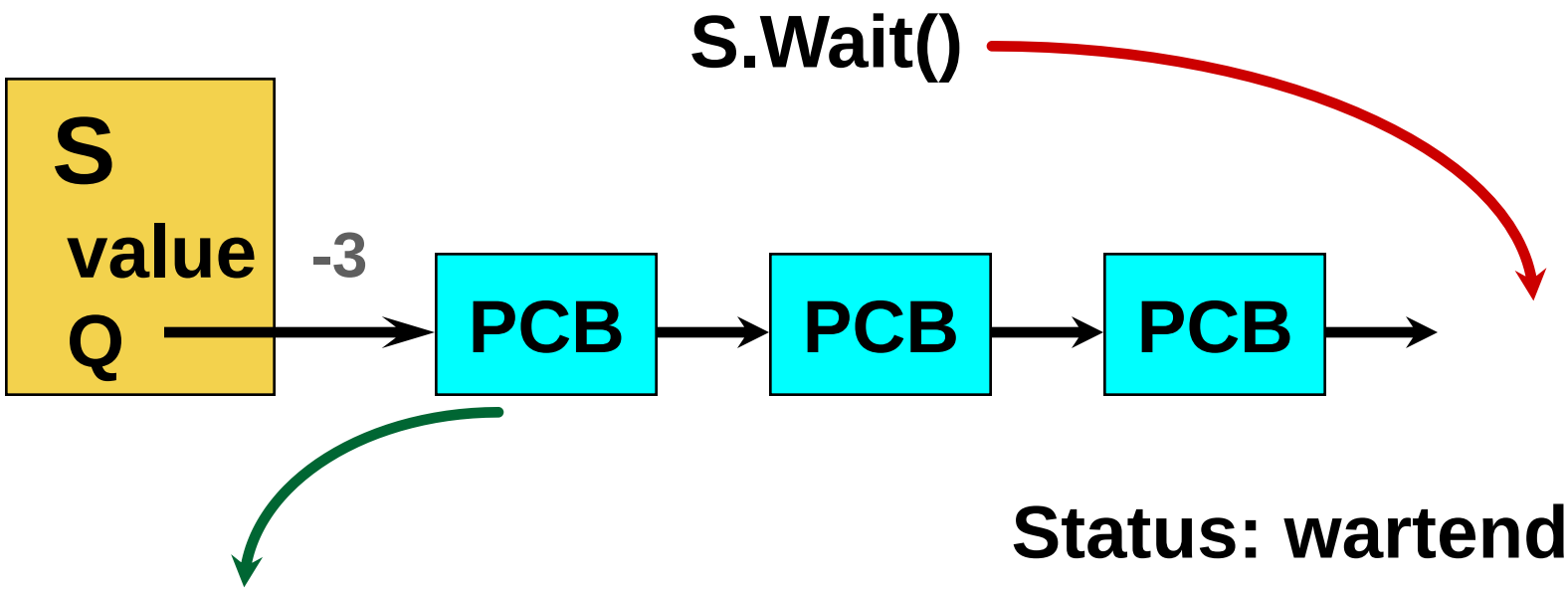
  ist implementierungsabhängig:

  ➡ Entweder P läuft weiter oder

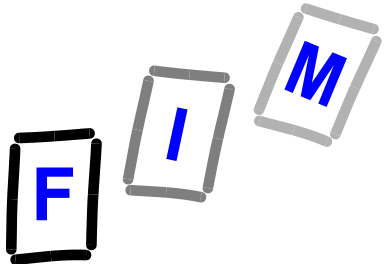
  ➡ der Scheduling Algorithmus entscheidet, ob entweder P oder P_Q laufen soll.

- Wenn Q leer war, wird S (unteilbar, atomar!) um 1 erhöht: S = S+1

Semaphore schematisch



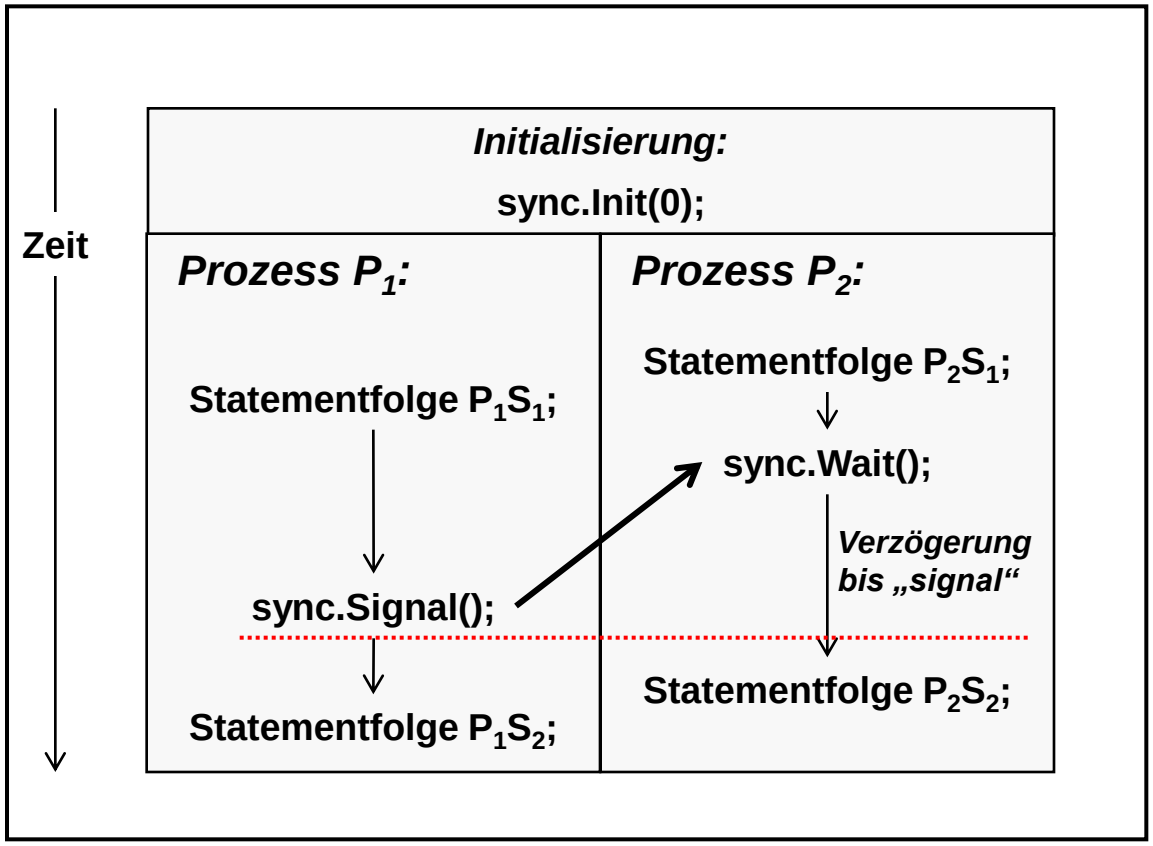
Achtung: Hier Variante mit negativem Wert bei wartenden Prozessen!



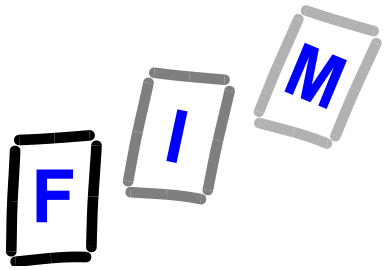
Semaphore-Operationen Beispiel

- **Annahme: S=3**
- **Dann können mit P1: S.Wait(); P2: S.Wait() und P3: S.Wait() drei Prozesse ohne anzuhalten an S „vorbei“.**
- **Erst der nächste Prozess P_4 wird beim Aufruf von P4: S.Wait() in die Schlange Q eingefügt.**
- **Er muss dort solange warten, bis ein anderer Prozess P_i ein S.Signal() auslöst.**

Synchronisation

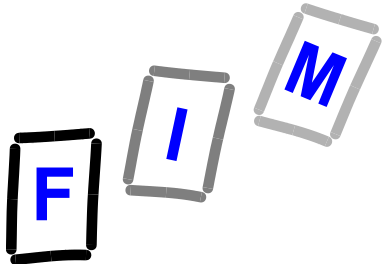


- Statementfolge P_2S_2 wird nach dem wait(sync) ausgeführt.
- Dieses muss auf signal(sync) warten, welches auf P_1S_1 folgt.
- Die prozessübergreifene zeitliche Reihenfolge „P_1S_1 und erst dann P_2S_2“ ist damit sichergestellt.



Semaphoren: Zusammenfassung auf Basis Pseudo-Code

- **Zähler S**
  - Zählt gesendete und erhaltene Signale
- **Atomare (unteilbare) Operationen**
  - **S.Signal()**
    - S := S-1, unteilbare Operation
      und prüft, ob ein Prozess aus der zu S assoziierten Warteschlange zu entnehmen und „ready“ zu setzen ist
  - **S.Wait()**
    - Überprüft zuerst ob S > 0
    - IF S > 0 THEN S := S-1, unteilbare Operation
    - IF S <= 0 THEN Prozess in Wartezustand versetzen
- **Probleme:**
  - ➡Aufwecken (wake-up)
  - ➡Beschäftigtes Warten (busy waiting)



Init();
Wait(); Signal()

- **Alle Operationen auf Semaphore sind unteilbar (atomar)**

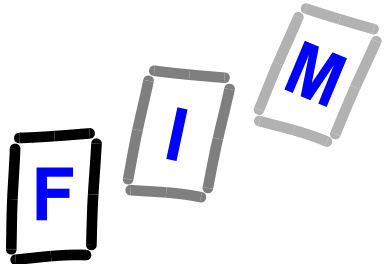
LOCK;

auszuführende Operationen
um das Semaphore-Konzept umzusetzen;

UNLOCK;

Betrachte die Operationen auf eine Semaphore damit als sehr kurzen CR !

Man kann zB *spinlocks* zur Implementierung verwenden.

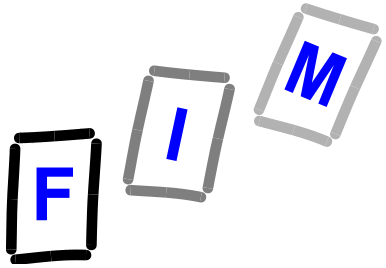


Init(); Wait(); Signal()

- **Alle Operationen auf Semaphore sind unteilbar (atomar)**

```
{  LOCK;              // Einsatz Sperrvariable
     auszuführende Operationen;
   UNLOCK;            // Einsatz Sperrvariable
```

```
typedef struct {
    int value;
    struct process *Q;
} semaphore
```



Init(); Wait(); Signal()

semaphore **S;**

S.Init(n) ::

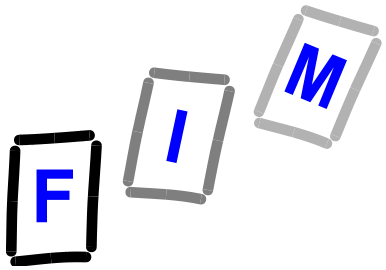
```
LOCK; S.value:=n; UNLOCK;
```

S.Wait() ::

```
LOCK; _WAIT(S); UNLOCK;
```

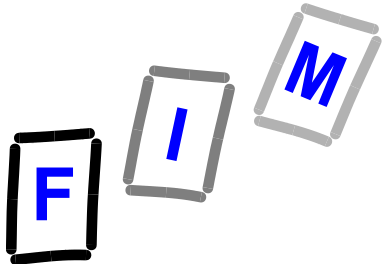
S.Signal() ::

```
LOCK; _SIGNAL(S); UNLOCK;
```



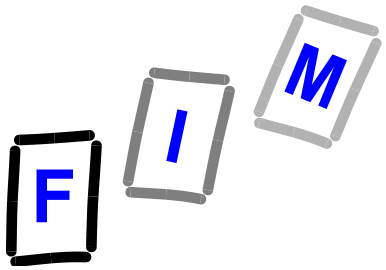
Wait(): Schematisch, Pseudocode

```
LOCK;
  S.value:=S.value-1; (*DEC(S.value)*)
  IF S.value < 0
    THEN
      Enqueue(S.Q, P.PCB);
      block P (* P muss warten *)
  END; (*if*)
UNLOCK;
```



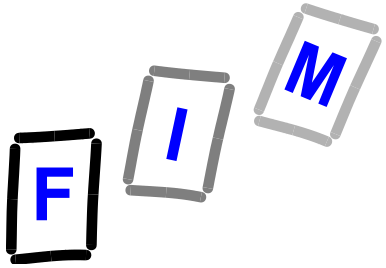
Wait()
Schematisch, C-Stil

```
LOCK;
  void Wait(semaphore S) {
     S.value--;
     if(S.value < 0) {
          //Prozess in die Warteschlange S.Q aufnehmen:
          enqueue(S.Q,P->PCB);
          block(P);
     }
  }
UNLOCK;
```



Signal()
Schematisch, Pseudocode

```
LOCK;
  S.value:=S.value+1;(*INC(S.value)*)
  IF S.value <= 0
    THEN
      Dequeue(S.Q,aPCB);(*fair!*)
      wakeup(aPCB)  (* nun ready *)
  END; (*if*)
UNLOCK;
```

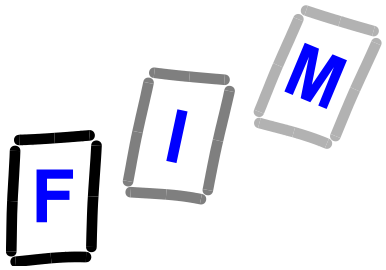


Signal()
Schematisch, C-Stil

LOCK;

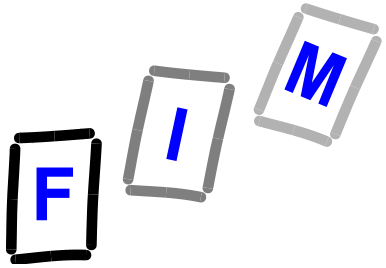
```
void Signal(semaphore S) {
    S.value++;
    if(S.value <= 0) {
        //einen Prozess PQ aus S.Q entfernen
        dequeue(S.Q,aPCB);
        wakeup(aPCB);
    }
}
```

UNLOCK;



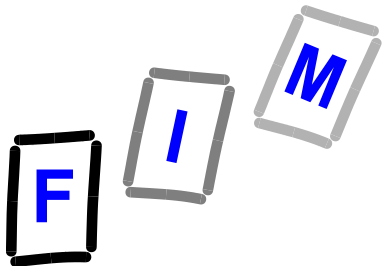
S.value
Schematisch, Pseudocode

- **Anmerkung:**
  - ➡ **In den Implementierungen wird S.value negativ, falls Prozesse auf ein SIGNAL warten.**
  - ➡ **S.value zählt damit die Anzahl der vor S wartenden Prozesse.**
- **Die Codesequenzen für Init, Wait und Signal können auch jeweils als (kurzer) kritischer Bereich angesehen werden.**
- **`Dequeue` muss „fair“ sein, um „Verhungern“ (engl.: Starvation) zu verhindern.**
  - ➡ **Trivial: First Come – First Served**



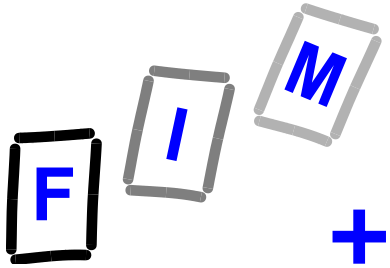
CR
„from inside out“

- **Was muss getan werden, um gegenseitigen Ausschluss zu erreichen, wenn nur Semaphore zur Verfügung stehen?**
- **Wie können Semaphore als „elementare“ Signale zwischen kooperierenden parallelen Threads verwendet werden?**
- **Wie gestaltet sich der Einsatz von Semaphoren als „Zähler“ mit unteilbaren Operationen *inc* (SIGNAL) und *dec* (WAIT)?**
- **Die folgenden Codeausschnitte sind teilweise Pseudocode.**



Kritischer Bereich (CR)
implementiert mittels Semaphoren

```
VAR mutex: semaphore;

Init(mutex,1);

{ Wait(mutex);
     CR KRITISCHER BEREICH
  Signal(mutex);
```



Nachrichtenpuffer + wechselseitiger Ausschluss

Einsatz von 3 Semaphoren:

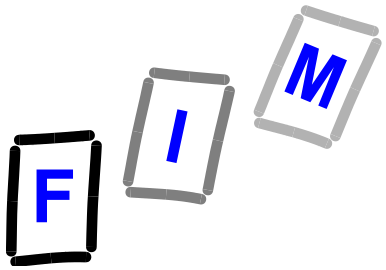
mutex: **Stellt den gegenseitigen Ausschluss beim Zugriff auf den Puffer sicher.**

***empty*:** **INIT(empty, Puffergröße);**
Gibt die Anzahl freier Puffereinträge an.

***full*:** **INIT(full, 0);**
Gibt die Anzahl belegter Pufferplätze an.

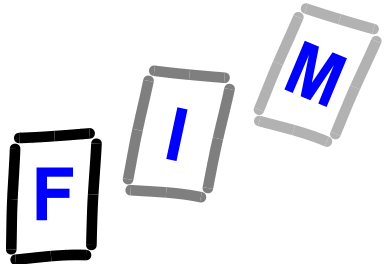
Nachrichtenpuffer Produzent

```
do {
   erzeuge einen Eintrag vom Typ item
   . . . . . .
   empty.Wait();
      mutex.Wait();
         (*CR*)
         trage Eintrag in den Puffer ein
      mutex.Signal();
   full.Signal();
}while(true); (*Endlosschleife*)
```



Nachrichtenpuffer Konsument

```
do {
    . . . . . .
   full.Wait();
     mutex.Wait();
        (*CR*)
        Eintrag aus dem Puffer entnehmen
     mutex.Signal();
   empty.Signal();
}while(true); (*Endlosschleife*)
```



Nachrichtenpuffer Erläuterung zur Lösung

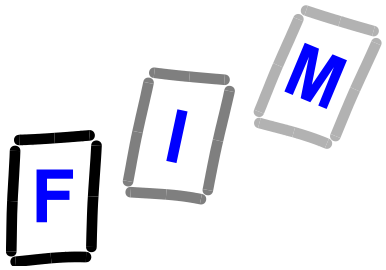
```
mutex.Wait();
    (*CR*)
     Eintrag aus/in dem Puffer
     entnehmen / eintragen
mutex.Signal();
```

- ***Wechselseitiger Ausschluss*** **(mutual exclusion) beim Zugriff auf den Puffer ist gewährleistet.**

Zweck von WAIT(full)

```
do {
     full.Wait();
     . . . . . .
} while(true); (*Endlosschleife*)
```

- Mit full.Wait() überprüft der Konsument, ob ein Element im Puffer b verfügbar ist.
- Ist kein Element vorhanden, so „legt er sich schlafen“.

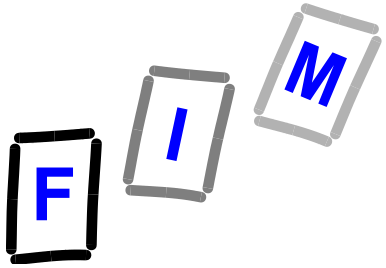


Zweck von SIGNAL(empty)

```
do {
     . . . . . .
     empty.Signal();
}while(true); (*Endlosschleife*)
```

Mit empty.Signal() wird

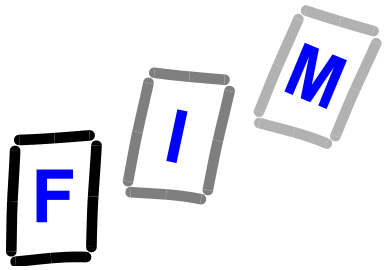
- **empty als Zähler für die Anzahl der freien Plätze verwendet (als INC(empty))**
- **bei Bedarf ein / der schlafende(r) Produzent P aktiviert.**



Die Lösung ist symmetrisch

full.Signal()

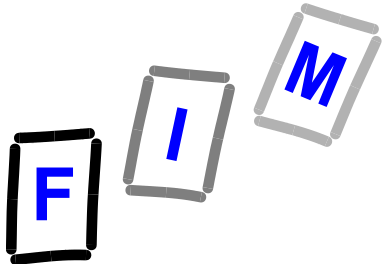
- Der Produzent P zählt, nachdem er einen neuen Eintrag in den Puffer gestellt hat, mit *full* die Anzahl der belegten Einträge hinauf: *INC(full)*.
- Falls vorher *full = 0* war, wird der schlafende Konsument geweckt.



Weitere Synchronisierungskonzepte

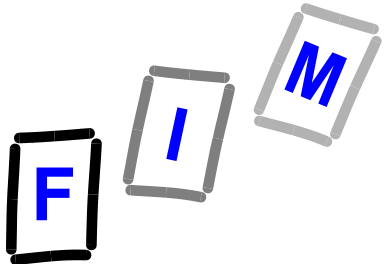
- **MONITORE**
  - ➡ Ähnlich wie abstrakte Datentypen („Datenkapselung“)
  - ➡ Übliche Zugriffsmethoden, unterstützen aber zusätzlich wechselseitigen Ausschluss

```
TYPE monitor_name = MONITOR
variable declarations
  PROCEDURE  entry P1(..);
  BEGIN . . . . . .
  END P1;
  PROCEDURE  entry P2(..);
  BEGIN . . . . .
  END P2;
BEGIN
  Initialisierungsteil
END (*monitor_name*);
```



JAVA und Monitore (1)

- Verwendet *synchronized* Schlüsselwort
- Jede Methode einer Klasse muss als *synchronisiert* definiert werden.
  - ➡ Jedes erzeugte Objekt ist implizit mit einem Monitor verbunden.
  - ➡ Eine *synchronized* Methode sperrt das Objekt, bevor sie auf das Objekt ausgeführt werden kann.
  - ➡ Jeder andere Thread, der versucht, ein bereits gesperrtes Objekt zu sperren, wird in eine zugehörige Warteschlange aufgenommen.

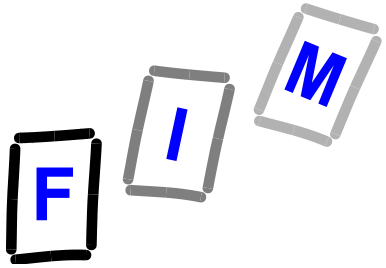


Beispiel für JAVA and Monitore (2)

```
public class BankAccount {
  private double dBalance;
```

Überlegen Sie: Warum nötig?

```
  public synchronized void
    Withdrawal (double dAmount) {
          dBalance = dBalance - dAmount;
  } /*Withdrawal*/

  public synchronized void
    Deposit(double dAmount) {
          dBalance = dBalance + dAmount;
  } /*Deposit*/

} /*BankAccount*/
```

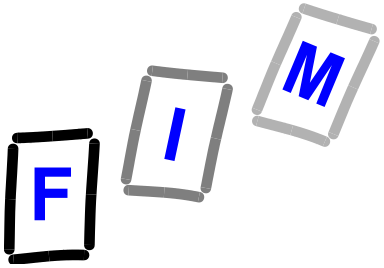


Wichtig bei JAVA und Monitore (3)

- Das *synchronized* Schlüsselwort gibt an, dass die gesamte Methode ein kritischer Bereich (CR) ist.
- Das angegebene Objekt wird gesperrt.
- Falls „static“ hinzugefügt wird:
  Das Sperren bezieht sich auf die Klasse (class method ↔ instance method)

```
public static synchronized void
   Withdrawal(double dAmount)
```

sperrt die gesamte Objekt-Klasse.



JAVA und Monitore (4)

- **Jede Ablaufreihenfolge von „synchronized“ Methoden ist erlaubt.**
- **Das Produzent-Konsument-Problem (engl.: producer-consumer problem) ist komplizierter !!!!!!!!!**
  - ➡ **Produzent darf nicht in den vollen Puffer schreiben.**
  - ➡ **Konsument darf nicht aus dem leeren Puffer lesen.**
  - ➡ **In beiden Fällen:**
    Sperre muss aufgehoben werden → notify
    Warten bis Sperre offen: → wait()
    + catch(InterruptedExeption e) {}