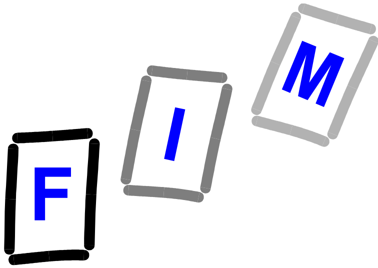
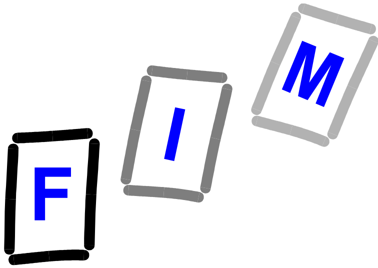


SS 2003

KV Betriebssysteme

(Peter René Dietmüller, Michael Sonntag)

Synchronisation

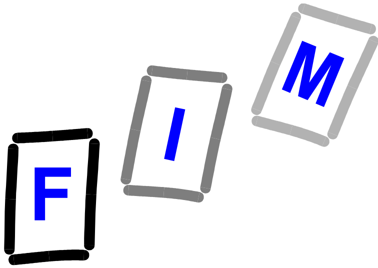


Synchronisation – Was ist das?

- Sobald auf einem Rechner mehr als ein Prozess läuft, kann es zu Problemen kommen:
- Wenn zwei Prozesse auf die selben Daten zugreifen, können Inkonsistenzen entstehen!
  - Beide Lesen: Unproblematisch!
  - Einer liest und einer schreibt: Vorher oder nachher lesen?
    - » Scheduling, Prioritäten, ...
  - Beide schreiben: Wer darf nachher schreiben (d. h. ändern)?
    - » Schlecht entworfenes Programm, Transaktionen, ...
- Das Problem entsteht NUR bei "shared memory!
  - Kein gemeinsamer Speicher: Nachrichten, Blackboards, ...
    - » Das Problem existiert auch dort, wird aber auf die Kommunikation verlagert!

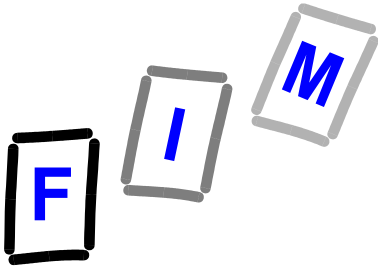
Synchronisation – Was ist das?

- **In der Praxis ist es jedoch so, daß beide Prozesse lesen und gleichzeitig beide schreiben wollen!**
  - A liest, B liest, A schreibt, B schreibt. Ergebnis: Komplett falsch!
    - » Weder gleich zu dem wie wenn zuerst A gelesen und geschrieben hätte, wie wenn zuerst B alle Aktionen durchgeführt hätte!
- **Beispiel:**
  - Start: a=0; Prozess 1: a=a+2; Prozess 2: a=a+3;
  - Ablauf: P1: $register_x$=a {$register_x$=0}
  - P2: $register_y$=a {$register_y$=0}
  - P1: $register_x$=$register_x$+2 {$register_x$=2}
  - P2: $register_y$=$register_y$+3 {$register_y$=3}
  - P1: a=$register_x$ {a=2}
  - P2: a=$register_y$ {a=3}



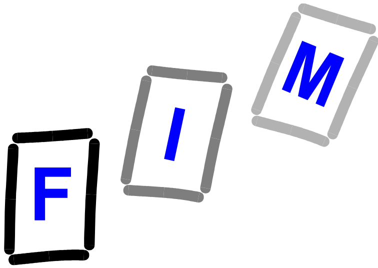
Race Conditions

- Diese Art von Problemen nennt man "race condition"
- Diese Fehler sind besonders schwer zu finden
  - Sie treten nicht immer sondern nur ab und zu auf
  - Die Probleme treten dann an vielen verschiedenen Stellen auf
  - Debuggt man das Programm, tritt der Fehler meist gar nicht auf
  - Fügt man Log-Statements ein, tritt der Fehler oft gar nicht auf
  - Keine Reproduzierbarkeit (winzige Zeitunterschiede wichtig)
- Unbedingt von vornherein vermeiden:
  - Immer wenn mehrere Prozesse auf dieselben Daten zugreifen
  - Irgendeine Art von Synchronisierung einbauen
    - » Das Programm wird langsamer, auch wenn sie tatsächlich nicht gebraucht würde
    - » Das Programm ist SEHR viel sicherer!



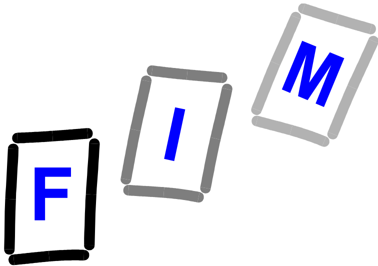
"Critical Region"

- **Programme die Synchronisation benötigen besitzen sogenannte "critical regions"**
- **Dies sind die Teile, in denen auf dem gemeinsamen Speicher gearbeitet wird**
- **Es darf sich als Definition immer nur ein einzige Prozess in einer critical region befinden**
  - **Alle anderen müssen am Beginn dieses Programmteils solange aufgehalten werden, bis dieser Teil verlassen wurde**
  - **Diese Region sollte so klein wie möglich gehalten werden!**
- **Ist garantiert, daß sich immer nur ein Prozess in einer critical region befindet, so sind race conditions ausgeschlossen**



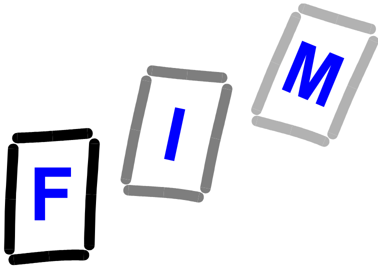
"Critical Region"

- **Critical regions beheben nur das Problem der race conditions**
  - ➢ **Die Reihenfolge von critical regions (siehe oben; zwei Prozesse wollen schreiben: wer darf nachher schreiben) wird nicht geregelt oder beeinflußt**
- **Die restlichen Probleme müssen selbst durch die Programmlogik behoben werden!**
  - ➢ **Je nach der Bedeutung, Priorität der Aktionen oder Benutzer, ...**
- **Critical regions sorgen nur dafür, daß das Programm bei gegebener Eingabe immer die gleichen Ergebnisse liefert**
  - ➢ **"Nur" keine Überlappung von Instruktionen; deren Reihenfolge wird nicht garantiert**



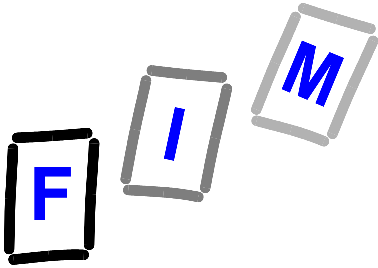
Lösung für "Gegenseitigen Ausschluß"

- **Um critial regions zu implementieren müssen drei Probleme gelöst werden:**
  - **Mutual exclusion: Keine zwei Prozesse dürfen sich gleichzeitig in einer critical region befinden**
    - » **Die Basis-Voraussetzung und das Ziel**
  - **Progress: Ist kein Prozess in einer critical region so wird in endlicher Zeit genau einer der Prozesse als berechtigt ausgewählt, diese zu betreten**
    - » **Irgendwann kommt der nächste Prozess dran (Kein Deadlock)**
  - **Bounded waiting: Es gibt einen Grenzwert wie oft andere Prozesse ihre critical region betreten dürfen nachdem eine anderer Prozess seinen Wunsch dafür angemeldet hat**
    - » **Alle kommen irgendwann einmal dran (Keine Starvation)**



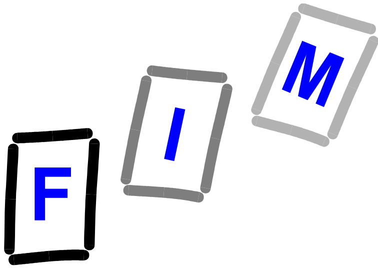
Grundlegende Konzepte

- **Es gibt viele verschiedene Arten, Synchronisation zu erreichen. Bekannte Grundkonzepte sind:**
  - **Semaphore: Integer-Variable**
    - » **Analogie: Züge auf einem Eisenbahngleis müssen vor dem Signal warten, bevor sie weiterfahren dürfen**
  - **Critical region: Boolsche Variable**
    - » **Analogie: Wer den "Stab des Sprechers" hält, darf reden**
  - **Monitor: "Objektorientierte critical region"**
    - » **Analogie: Immer nur eine Person darf mit der Geisterbahn fahren; ist aber niemand da, fährt der Wagen eben leer**
- **Die Grundkonzepte sind äquivalent, d. h. sie können jeweils durch die anderen implementiert werden**



Das "Bounded Buffer" Problem

- **Zur Erläuterung der verschiedenen Konstrukte wird ein übliches Problem verwendet**
- **Ein Buffer, in den ein Prozess schreibt und aus dem ein anderer Prozess liest**
- **Dieser Buffer ist in seiner Größe begrenzt ("bounded")**
  - **Ist er voll, muß der Produzent warten**
  - **Ist er leer, muß der Konsument warten**



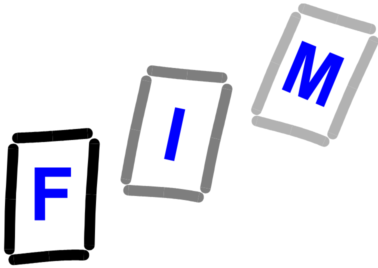
"Bounded Buffer": Ohne Synchronisation (1)

- **Definition:**

```
final static int BUFFER_SIZE=...;
Object[] buffer=new Object[BUFFER_SIZE];
int count=0;
int in=0;
int out=0;
```

- **Producer:**

```
while(true) {
    while(count==BUFFER_SIZE) ;
    buffer[in]=nextProduced;
    in=(in+1)%BUFFER_SIZE;
    count++;
}
```

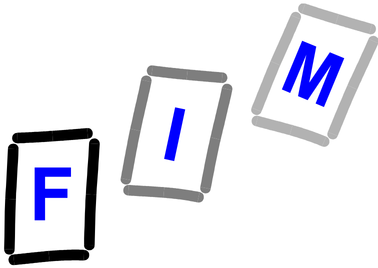


"Bounded Buffer": Ohne Synchronisation (1)

- **Consumer:**

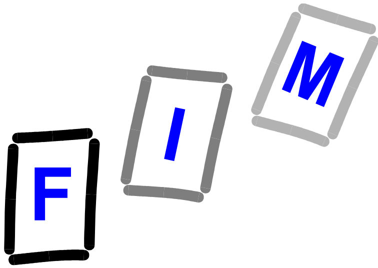
```
while(true) {
    while(count==0) ;
    nextConsumed=buffer[out];
    out=(out+1)%BUFFER_SIZE;
    count--;
}
```

- **Wird jeweils ein ganzer Schleifendurchlauf von Producer oder Consumer ausgeführt, so ist diese Lösung korrekt**
  - ➢Aber was passiert, wenn count++ und count-- gleichzeitig ausgeführt werden?
  - ➢Oder wenn count vor dem Hineinschreiben des neuen Wertes erhöht wird?



Semaphor

- Eine Integer-Variable, auf die nur über zwei spezielle Funktionen zugegriffen werden kann:
  - s.wait(): Warten bis der Wert größer als 0 ist und dann um eines verringern
  - s.signal(): Wert um eins erhöhen. Wird der Wert größer als 0, so wird ein wartender Prozess gestartet.
  - Wert<=0: Anzahl der wartenden Prozesse
  - Wert>0: Anzahl der Prozesse die ein wait ausführen können ohne blockiert zu werden
- Diese Aktionen sind nicht teilbar und werden auf jeden Fall zur Gänze ausgeführt
  - Dies wird vom Betriebssystem/Hardware garantiert!

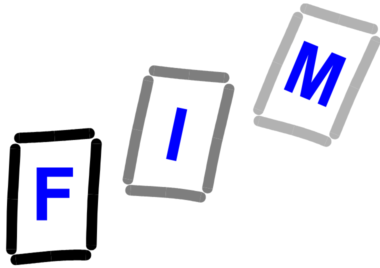


Semaphor

- Ein Semaphor kann "auf Vorrat" mit signals versehen werden und kann viele Prozesse gleichzeitig im Wartezustand haben
- Für critical regions wird der Wert auf 1 gesetzt: Ein Prozess kann hinein, alle anderen müssen warten!

```
s.wait();
.... critical region ....
s.signal();
```

- Bei einem Startwert von N wird garantiert, daß sich höchstens N Prozesse gleichzeitig in der critical region befinden
- Andere Verwendungen (wait oder signal alleine) sind auch möglich, aber gut zu überlegen (Deadlocks, ...)!



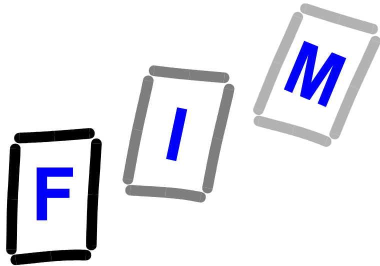
Semaphor Beispiel

- Lösung von Bounded Buffer mit Semaphoren
- Initialisierung:

```
Semaphor empty=new Semaphor(BUFFER_SIZE);
Semaphor full=new Semaphor(0);
Semaphor mutex=new Semaphor(1);
```

- Producer:

```
empty.wait();
mutex.wait();
buffer[in]=nextProduced;
in=(in+1)%BUFFER_SIZE;
count++;
mutex.signal();
full.signal();
```

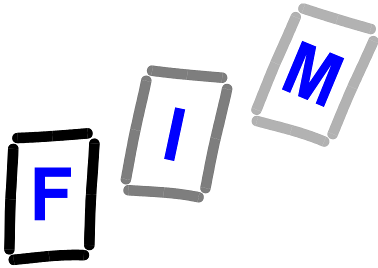


Semaphor Beispiel

- Consumer:

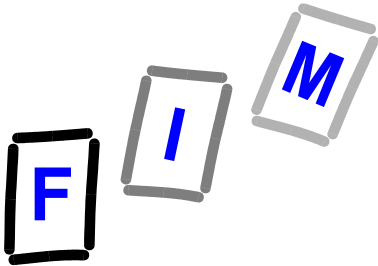
```
full.wait();
mutex.wait();
nextConsumed=buffer[out];
out=(out+1)%BUFFER_SIZE;
count--;
mutex.signal();
empty.signal();
```

- "Mutex" ist nötig um sicherzustellen, daß immer nur ein Prozess sich in der critical region befindet
- Full und empty dienen dazu den Füllstand abzubilden und gleichzeitig um Prozesse stillzulegen, bis Platz frei/Daten vorhanden sind



Critical region

- Eine globale Variable (=Token), die jeweils nur von einem Prozess "besessen" werden kann
  - Nur innerhalb einer critical region ist Zugriff darauf möglich
- Beispiel:
  - v: shared *Type*;
  - region v do *Statement*;
    - » Nur innerhalb von "Statement" kann auf v zugegriffen werden
    - » Immer nur ein einziger Prozess kann für eine bestimmte "shared"-Variable in einem region-Statement sein
    - » Dies wird von der Programmiersprache/Betriebssystem garantiert
  - Alternative: region v when *boolean_expression* do *Statement*;
    - » Zusätzlich muß noch der boolsche Ausdruck "Wahr" ergeben
    - » Wenn nicht, wird solange mit der Ausführung gewartet, bis er es ist (d. h. ein anderer Prozess muß diese "wahrmachen")!



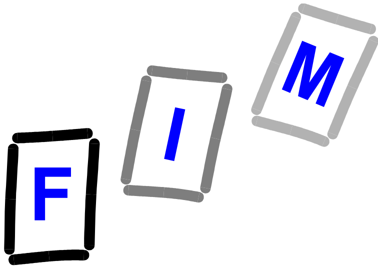
Critical region Beispiel

- Lösung von Bounded Buffer mit critical regions:
- Initialisierung:

```
class BufferObj {
   Object buffer[]=new Object[BUFFER_SIZE];
   int count=0; int in=0; int out=0;
};
shared BufferObj myBuffer=new BufferObj();
```

- Producer:

```
region myBuffer when (count<BUFFER_SIZE) {
   buffer[in]=nextProduced;
   in=(in+1)%BUFFER_SIZE;
   count++;
}
```

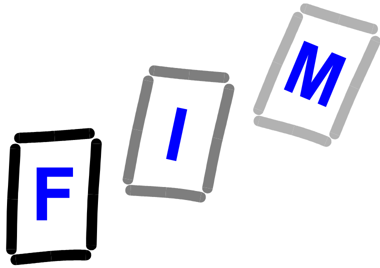


Critical region Beispiel

- **Consumer:**

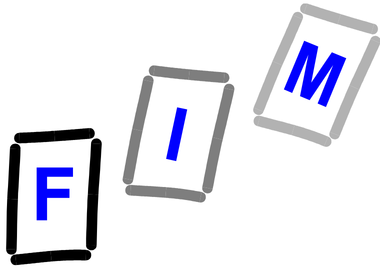
```
region myBuffer when (count>0) {
    nextConsumed=buffer[out];
    out=(out+1)%BUFFER_SIZE;
    count--;
}
```

- **Hier entweder busy-waiting oder nicht, je nach Implementierung!**
  - ➢ **Kommt darauf an, wie die Bedingung geprüft wird**



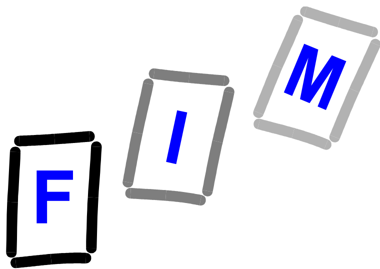
Monitor

- **Bei einem Monitor wird eine Klasse als solche definiert**
- **Es dürfen dann nur Klassenmethoden auf die eigenen Variablen zugreifen**
  - **Und diese dürfen nur auf die eigenen Variablen und Parameter zugreifen, nicht jedoch auf externe Daten!**
- **Zusätzlich besitzt ein Monitor eine Warteschlange für Threads, die gerne eine Methode des Monitors ausführen möchten:**
- **Es darf immer nur ein einziger Thread eine Methode der Klasse ausführen**
  - **Ganz egal welche, rekursiver Aufruf, etc.**
  - **Verschiedene Methoden durch verschiedene Threads → NEIN!**



Monitor Operationen

- x.wait()
  - Der derzeitige Prozess wird dauerhaft suspendiert (in den Wartezustand gesetzt)
- x.signal()
  - Genau ein Prozess wird aus dem Wartezustand herausgeholt und auf "runnable" gesetzt
  - Wartet kein Prozess, ergibt sich keine Änderung
    - Unterschied zu Semaphor: Dort hat signal IMMER eine Auwirkung
- x ist entweder der Monitor selber oder eine spezielle Variable (z. B. "condition x;")
  - Je nach Implementierung; bei speziellen Variablen gibt es dann jeweils eine gesonderte Warteschlange



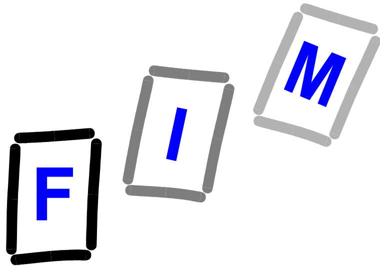
Monitor Beispiel

- Lösung von Bounded Buffer mit Monitor:
- Initialisierung:

```
monitor BoundedBuffer {
```

- Producer:

```
{
    if(count==BUFFER_SIZE)
        wait();
    buffer[in]=nextProduced;
    in=(in+1)%BUFFER_SIZE;
    count++;
    signal();
}
```

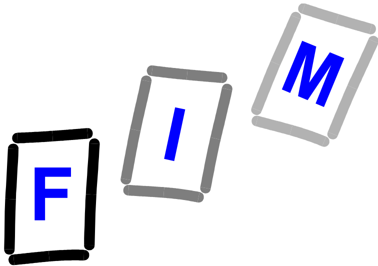


Monitor Beispiel

- Consumer

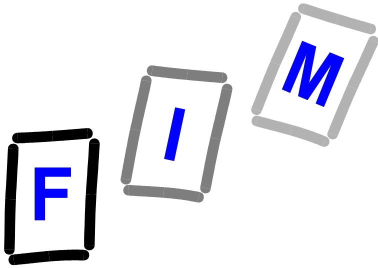
```
{
    if(count==0)
        wait();
    nextConsumed=buffer[out];
    out=(out+1)%BUFFER_SIZE;
    count--;
    signal();
}
```

- "Überschüssige" signal sind kein Problem: Sie sind ohnehin wirkungslos
- Bei signal nichts prüfen: Entweder es warten NUR producer oder NUR consumer (oder gar niemand)
  - Gemischtes Warten (>=1 Producer UND >=1 Consumer warten ist unmöglich!)



Implementation in Java Monitor

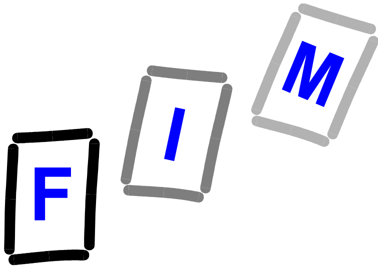
- Das einzige Synchronisationskonzept das Java direkt unterstützt
- Leichte Modifikation: Es wird nicht eine Klasse als "Monitor" definiert sondern eine frei wählbare Teilmenge der Methoden
  - Diese können auch auf andere (externe) Variablen oder Objekte zugreifen (Aber Vorsicht!)
  - Kennzeichnen einer Methode als "synchronized"
  - synchronized (obj) {} Blöcke um explizit zu synchronisieren
  - Jeweils nur eine Warteschlange (=1 Monitor) pro Objekt
    - » ALLE synchronized Methoden eines Objektes besitzen mutual exclusion; keine "geteilten Subsets" davon sind möglich!



Implementation in Java Monitor

- **wait und wait(long milliSecondsTimeout)**
  - **Thread wartet bis er aufgeweckt wird bzw. das Timeout abläuft**
- **notify() zum Aufwecken (=signal(); nur anderer Name)**
- **notifyAll() um ALLE wartenden Threads aufzuwecken**
- **wait und notify können nur aufgerufen werden, wenn der Monitor im Besitz des Threads ist**
  - **D. h. in synchronized Methoden oder in Blöcken synchronized(obj), wobei obj das zu bearbeitende Objekt ist**
    - » **Beispiel: synchronized(obj) { ... obj.wait(); ... }**
    - » **Beispiel: public synchronized void put(...); { ... notify(); ... }**

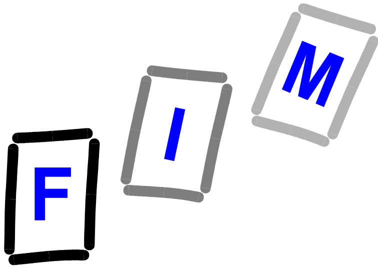
      Enspricht "this.notify();"



Implementation in Java Critical region

- Meistens in der Form von "Mutex" (=mutual exclusion)
- lock(): Beginn einer kritische Region
- unlock(): Ende einer kritischen Region
- Es kann immer nur ein Prozess den lock besitzen und daher immer nur einer in der critical region sein
- Implementation trivial, bis auf ein extrem trickreiches Problem (sonst ev. mehrere Threads gleichzeitig drin!)

Mutex.java



Implementation in Java Semaphor

- **Sempahor-Implementation in Java ist nicht besonders kompliziert**
  - **Darauf achten, was bei interrupts passiert**
    - **Java ermöglicht es, wartende Threads auch abzubrechen**

Semaphor.java