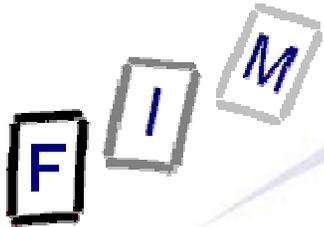


Peter Rene Dietmüller, Michael Sonntag

KV Betriebssysteme Synchronisation

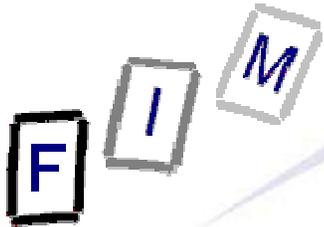
Institut für Informationsverarbeitung und
Mikroprozessortechnik (FIM)
Johannes Kepler Universität Linz, Österreich

E-Mail: sonntag@fim.uni-linz.ac.at
<http://www.fim.uni-linz.ac.at/staff/sonntag.htm>



Synchronisation – Was ist das?

- Sobald auf einem Rechner mehr als ein Prozess läuft, kann es zu Problemen kommen:
- Wenn zwei Prozesse auf dieselben 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 Kommunikationsebene verlagert (=gemeinsamer interner Puffer)!

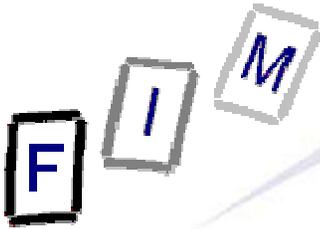


Synchronisation – Was ist das?

- 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 wie wenn zuerst A gelesen und geschrieben hätte, wie wenn zuerst B alle Aktionen durchgeführt hätte!

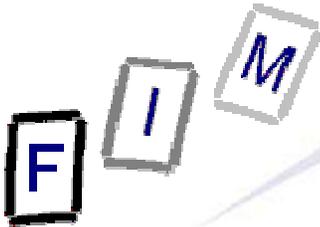
- Beispiel:

→ 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}



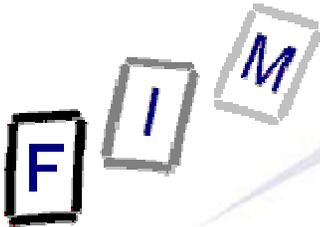
Race Conditions

- 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
 - Beim debuggen 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!



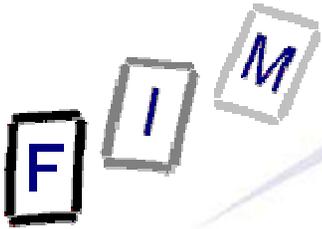
"Critical Region"

- 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 einziger Prozess in einer critical region befinden
 - Alle anderen müssen am Beginn dieses Programmteils solange aufgehalten werden, bis dieser Teil "frei" wird
 - Diese Region sollte so klein wie möglich gehalten werden!
 - » Effizienzgründe, da Parallelität leidet
- Ist garantiert, dass sich immer nur ein Prozess in einer critical region befindet, so sind race conditions ausgeschlossen!



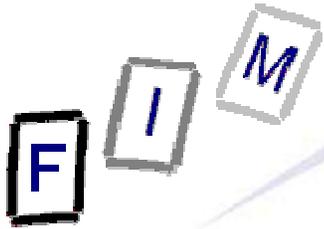
"Critical Region"

- Critical regions beheben nur das Problem der race conditions
 - Die Reihenfolge von critical regions ("Lost update"; 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 (fast) immer gleiche Ergebnisse** liefert
 - "Nur" keine Überlappung von Instruktionen
 - Deren Reihenfolge wird **nicht** garantiert



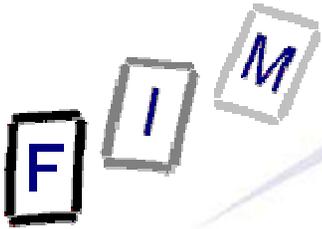
Lösung für "Gegenseitigen Ausschluss"

- Um critical 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 ein anderer Prozess seinen Wunsch dafür angemeldet hat
 - » Alle kommen irgendwann einmal dran (Keine starvation)



Grundlegende Konzepte

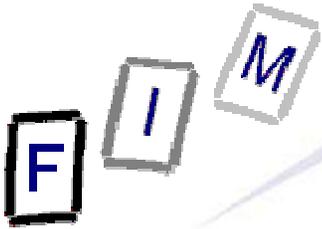
- Es gibt viele verschiedene Arten, Synchronisation zu erreichen. Bekannte und übliche Grundkonzepte sind:
 - **Semaphor: Integer-Variable**
 - » Analogie: (Mehrere) Züge auf einem Eisenbahngleis müssen vor dem Signal warten, bevor sie weiterfahren dürfen
 - **Critical region: Boolesche Variable**
 - » 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 absolut äquivalent, d.h. sie können jeweils durch eines der Anderen implementiert werden
 - In der Benutzung sind aber vielfach bestimmte einfach einfacher/klarer verständlich!



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)



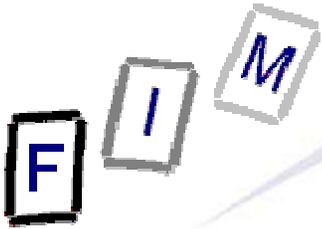
- Definition:

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

- Producer:

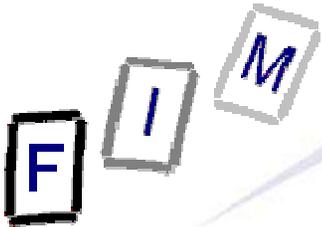
```
while(true) {  
    while(count==BUFFER_SIZE) ;  
    buffer[in]=nextProduced;  
    in=(in+1)%BUFFER_SIZE;  
    count++;  
}
```

"Bounded Buffer": Ohne Synchronisation (1)



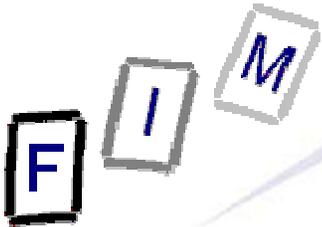
- Consumer:

```
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++` (vorige Seite; P) und `count--` (diese Seite; C) gleichzeitig ausgeführt werden?
 - Oder wenn `count` vor dem Hineinschreiben des neuen Wertes erhöht wird?



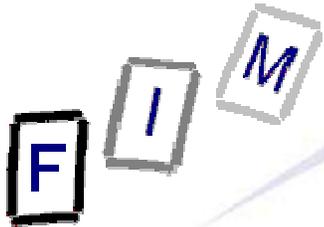
Semaphor

- 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 eins verringern
 - `s.signal()`: Wert um eins erhöhen. Wird der Wert größer als 0, so wird ein wartender Prozess freigesetzt
 - $\text{Wert} \leq 0$: Anzahl der wartenden Prozesse
 - $\text{Wert} > 0$: Anzahl der Prozesse die ein wait ausführen können, ohne blockiert zu werden
- Diese Aktionen sind **unteilbar** und werden auf jeden Fall zur Gänze und nicht-überlappend (pro Semaphor) ausgeführt
 - Dies wird vom Betriebssystem/Hardware garantiert!



Semaphor

- Ein Semaphor kann "auf Vorrat" mit signals versehen werden, kann aber auch 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 höchstens N Prozesse sich gleichzeitig in der critical region befinden
- Andere Verwendungen (wait oder signal alleine) sind auch möglich, aber gut zu überlegen (Deadlocks, ...)
 → Hauptanwendungsgebiete für Semaphore!

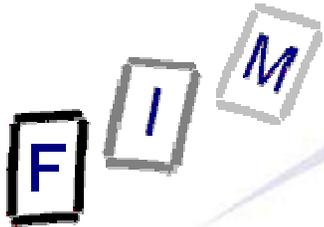


Semaphor Beispiel

- Lösung von Bounded Buffer mit Semaphoren
- Initialisierung:

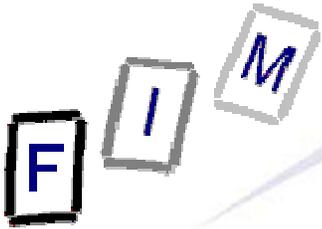
```
Semaphor empty=new Semaphor(BUFFER_SIZE);  
Semaphor full=new Semaphor(0);  
Semaphor mutex=new Semaphor(1);
```
- Producer:

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



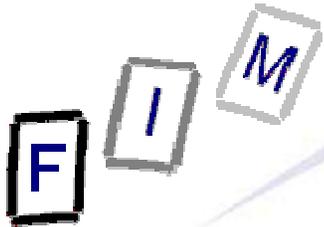
Semaphor Beispiel

- Consumer:
`full.wait();`
`mutex.wait();`
`nextConsumed=buffer[out];`
`out=(out+1)%BUFFER_SIZE;`
`count--;`
`mutex.signal();`
`empty.signal();`
- "Mutex" ist nötig um sicherzustellen, dass sich immer nur ein Prozess in der critical region befindet
- Full und empty dienen dazu, den Füllstand abzubilden und gleichzeitig die Prozesse stillzulegen, bis Platz frei bzw. wieder 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 Programmiersprache/Betriebssystem garantiert
 - Alternative: region *v* when *boolean_expression* do *Statement*;
 - » Zusätzlich muss noch der boolesche Ausdruck "Wahr" ergeben
 - » Wenn nicht, wird solange mit der Ausführung gewartet, bis er es ist (d. h. ein anderer Prozess muss diese "wahr-machen")!



Critical region Beispiel

- Lösung von Bounded Buffer mit critical regions:

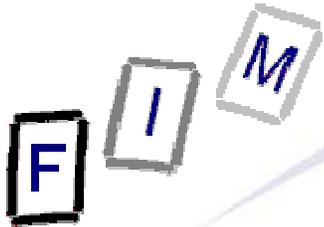
- Initialisierung:

```
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++;  
}
```

→Achtung: Nicht wirklich Java-Code!

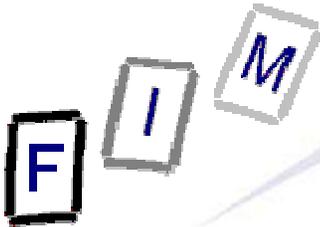


Critical region Beispiel

- Consumer:

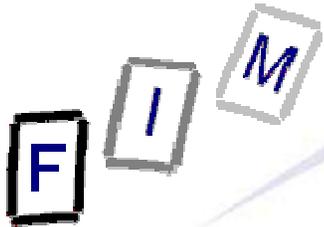
```
region myBuffer when (count>0) {  
    nextConsumed=buffer[out];  
    out=(out+1)%BUFFER_SIZE;  
    count--;  
}
```

- Hier entweder busy-waiting, oder auch nicht
 - Je nach konkreter Implementierung!
 - » Nicht im Synchronisations-Konzept festgelegt!
 - Kommt darauf an, wie die Bedingung geprüft wird



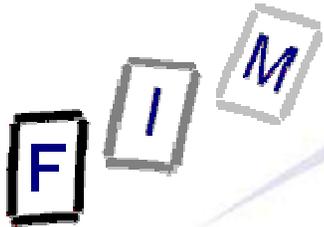
Monitor

- Bei einem Monitor wird eine Klasse als solche definiert
- Es dürfen dann nur Klassenmethoden auf die eigenen Variablen zugreifen (d.h. alles ist private)
 - 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 (irgend-)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 von verschiedenen Threads aufgerufen → NEIN!**



Monitor Operationen

- `x.wait()`
 - Der derzeitige Prozess wird 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 Semaphore: Dort hat "signal" immer eine Auswirkung!
- `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

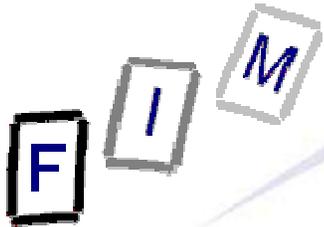
- Lösung von Bounded Buffer mit Monitor:

- Initialisierung:

```
monitor BoundedBuffer {
```

- Producer:

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

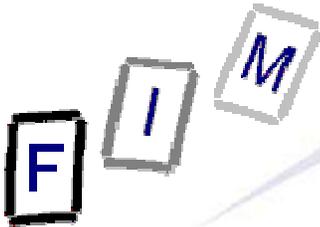


- Consumer

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

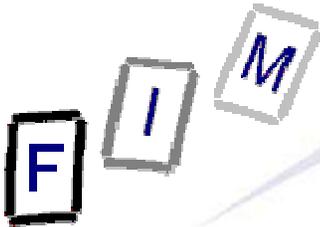
- "Überschüssige" signal sind kein Problem
→ Sie sind ohnehin wirkungslos
- Bei signal nichts prüfen: Entweder es warten NUR producer oder NUR consumer (oder gar niemand)
→ Gemischtes Warten (≥ 1 Producer UND ≥ 1 Consumer warten ist unmöglich!)

Implementation in Java Monitor



- Das einzige Synchronisationskonzept das Java direkt unterstützt
 - Seit Java 1.5: Mehr in der Klassenbibliothek (siehe später)
- Leichte Modifikation: Es wird nicht eine Klasse als "Monitor" definiert, sondern eine frei wählbare Teilmenge aller Methoden
 - Diese können auch auf andere (externe) Variablen oder Objekte zugreifen (Aber Vorsicht dabei!)
 - 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 möglich!

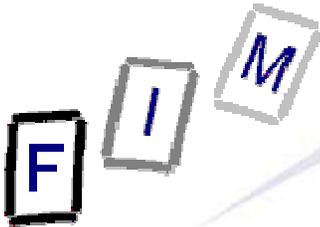
Implementation in Java Monitor



- 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(); ... }`
 - Entspricht "`this.notify();`"

Implementation in Java

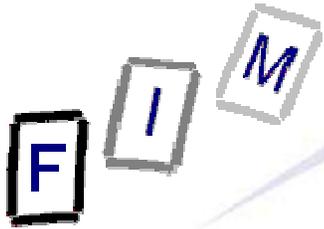
Critical region



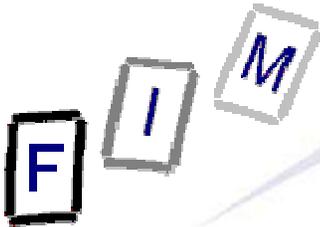
- Meistens in der Form von "Mutex" (=mutual exclusion)
- lock(): Beginn einer kritische Region
- unlock(): Ende einer kritischen Region
- Es kann immer nur ein Prozess den lock besitzen und daher immer nur einer in der critical region sein

- Java 1.5: Package `java.util.concurrent.locks`
 - **Zusätzlich "tryLock": Lock holen, wenn möglich**
 - » **D.h. kein Warten wenn nicht frei!**
 - **Zusätzlich "lockInterruptible": Kann unterbrochen werden**
 - **Implementierungen:**
 - » **ReentrantLock, ReentrantReadWriteLock**

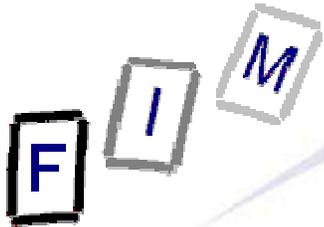
Implementation in Java Semaphore



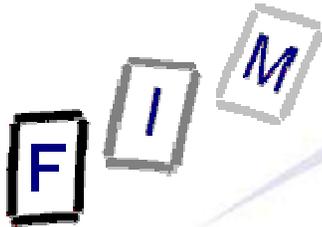
- Semaphore-Implementation in Java ist nicht besonders kompliziert (siehe auch Beispiele)
 - Darauf achten, was bei interrupts passiert
 - » Java ermöglicht es, wartende Threads auch zu unterbrechen
 - InterruptedException wird ausgeworfen
 - Nicht möglich: "Abschießen" von Threads
 - » Wäre zwangsweises beenden von einem anderen Thread aus
 - » Thread-Kommunikation stattdessen erforderlich
 - Thread muss sich freiwillig selbst beenden
- Java 1.5: Package **java.util.concurrent**
 - Methoden heißen "acquire" und "release"
 - Zusätzlich "tryAcquire": Acquire sofern möglich
 - Fairness möglich (FIFO; mit Ausnahme: tryAcquire!)



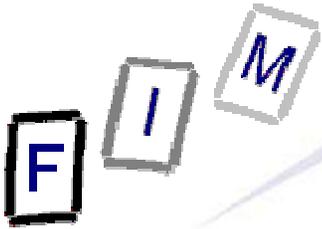
- Java unterstützt Threads
 - Threads besitzen Prioritäten
 - » Diese sind Betriebssystem-unabhängig!
 - "Ungefähre" Abbildung auf OS-Prioritäten
 - Kein eigener Scheduler (→ OS-Threads verwendet)
 - Threads können hierarchisch gruppiert werden
 - » Verwaltungs-Vereinfachung; keine Auswirkung auf Scheduling
- Zustandsmodell wie bei Prozessen, aber Methoden zur Überführung stark eingeschränkt!
 - Stop, suspend, resume, destroy: Deprecated und wohl bald endgültig beseitigt
 - » Diese Methoden können zu enormen Problemen führen
 - "destroy" war z.B. überhaupt nie implementiert!



- Unabhängig vom Prozess-Modell (running, waiting, suspended, ...), können Java Threads weitere Zustände besitzen:
 - Alive: Gestartet und noch nicht beendet
 - » =running, waiting oder suspended
 - Daemon: "Hintergrund-Thread"
 - » Die JVM wird beendet, sobald nur noch Daemon-Threads vorhanden sind
 - Interrupted: Der Thread war im Wartezustand und wurde unterbrochen
 - » Achtung: Die Abfrage **setzt den Status zurück!**

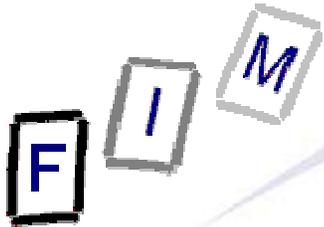


- Erzeugen: Jeder Thread benötigt einen Namen
 - Keiner festgelegt → automatisch Eindeutiger erzeugt
- `Thread.currentThread()`: Derzeit ausgeführter Thread
- `join`: Warten (ev. bestimmte Zeit) auf Ende des Thread
- `Thread.sleep`: Wartezustand für bestimmte Zeitspanne
 - Achtung: Monitore werden nicht freigegeben!
- `Thread.yield()`: Neu-Scheduling auslösen, wobei dieser Thread (hoffentlich) übergangen wird
 - D.h. freiwillig die CPU abgeben
 - Von Bedeutung ev. für Optimierungen
 - » Keinerlei Garantien!
- Hauptmethode: `public void run()`
 - Ausführen der eigentlichen Arbeit



Java Threads wait vs. sleep

- Beides legt den aktuellen Thread "schlafen"
- Monitore:
 - sleep behält alle Monitore
 - wait verliert den "eigenen", behält aber alle anderen
- Zeitablauf:
 - Wait kann sofort, nach festgelegter Zeit, oder auch überhaupt nie zurückkehren
 - Sleep ist immer zeitlich begrenzt
- Beide Wartezustände können durch "interrupt" unterbrochen werden
 - Sofortige Rückkehr mit "InterruptedException"



Java Threads Beispiel

```
public class MyThread extends Thread {  
    public void run() {  
        while(!Thread.interrupted()) {  
            doSomething();  
        }  
    }  
}
```

```
MyThread t=new MyThread();
```

```
t.start();
```

...

```
t.interrupt();
```

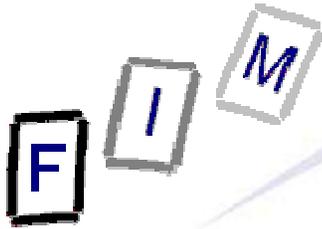
...

```
t.join(5000);
```

Ausführung von run() beginnt

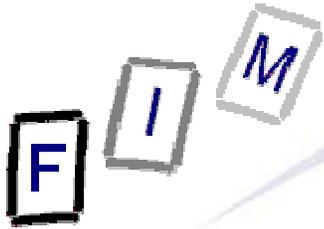
"Signal", dass Thread beendet werden soll (läuft aber noch weiter!)

Warten auf tatsächliches Ende (5 Sekunden lang)



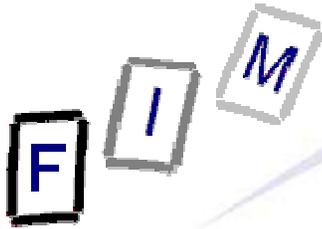
Java Threadgroups

- Hierarchische Gruppierung von Threads
 - Baumstruktur
- Dient einfacherer Verwaltung
- Sicherheits-Unterstützung
 - Ein Thread kann nur auf seine eigene ThreadGroup, sowie auf hierarchisch untergeordnete Gruppen zugreifen
- Geringe Bedeutung, außer zur inhaltlichen Strukturierung (interne Verwaltung bei vielen Threads)



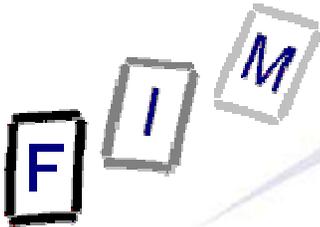
ThreadLocal Variablen

- Einmal deklariert, werden für jeden Thread separat angelegt
 - Normal: `private static Variable`
- Jeder Thread der darauf zugreift, sieht seinen eigenen Wert
 - Achtung: `1 Variable` \Leftrightarrow `x Werte!`
 - "Lazy initialization": Ohne Zugriff darauf wird nichts angelegt
- Verwendung: Klassen reentrant machen
 - "Stateful class" soll "thread-safe" werden
 - Alternative: Daten in den Thread hineinverlagern
 - » Ist in diesen Fällen aber meist schwierig!
- Bsp: Ein Objekt wird von mehreren Threads verwendet
 - Für jeden Thread aber ein anderer Zustand

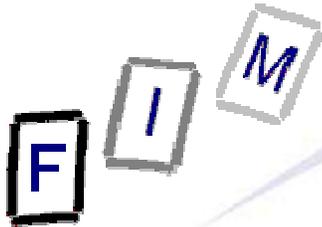


Java 1.5: Synchronisation

- Spezielle Synchronisationsklassen neu eingeführt
 - `java.util.concurrent`: Allgemein verwendbare Klassen
 - `java.util.concurrent.atomic`: Klassen für atomaren Zugriff auf einzelne Variablen bzw. Arrays
 - `java.util.concurrent.locks`: Hilfsklassen und grundlegende Primitive
- Enthalten viele nützliche Klassen, teilweise auch mit Erweiterungen
 - Beispiel: Semaphore unterstützt auch "fair waiting"
 - » Wer als erster wartet, kommt als erster heraus
 - » "Normales" `wait()`: Keine derartige Garantie ("irgendeiner")!
 - Wie auch in einfacherer Beispiels-Semaphore Implementierung!



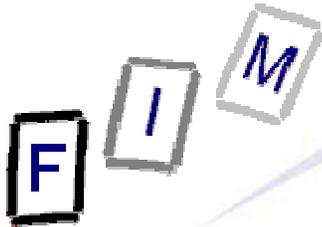
- Synchronisierte Queues, Hashmaps, Lists, Sets
- Unterstützung für Thread-Pooling und Scheduling
 - **Executor: Ähnlich einem Thread**
 - » Entkopplung der Aufgabe davon, wie sie tatsächlich ausgeführt wird (wann, welcher Thread, ...)
 - **Können auch asynchron ausgeführt werden**
- Semaphore: Klassisch, zusätzlich Fairness möglich
- CountdownLatch: Einmaliges warten, bis eine gewisse Anzahl von Threads einen Punkt erreicht haben; dann werden alle gleichzeitig freigesetzt ("Rendezvous")
- CyclicBarrier: Wie CountdownLatch, aber rücksetzbar
- Exchanger: Synchronisationspunkt für zwei Threads zum Austauschen zweier Datenobjekte (→ Bounded Buffer!)



java.util.concurrent.atomic: Überblick

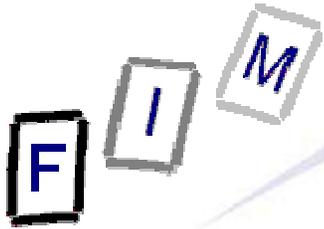
Achtung: Hier werden teilweise generische Typen verwendet
(auch neu in Java 1.5)!

- Atomar: Zugriff darauf ist "unteilbar"
 - Race conditions können innerhalb nicht auftreten
 - » Zwischen verschiedenen Zugriffen jedoch immer noch!
- Atomare Datenelemente: Boolean, Integer, Long
- Atomare Arrays: Integer, Long
 - Atomizität betrifft nur den Zugriff auf einzelne Elemente!
- Atomare Referenz, Referenz-Array und Referenz mit einem Markierungsbit oder einem Markierungs-Integer
 - Referenz auf einen beliebigen Datentyp
 - » Typsicher aufgrund generischen Typs
 - D.h. **nicht** auf "Object" → Generischer Typ erforderlich!



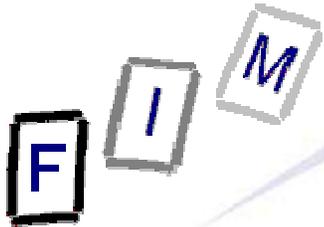
java.util.concurrent.atomic: Überblick

- Über Reflection und Spezialklassen Atomizität für Zugriffe auf Elemente existierender Klassen möglich
 - » `AtomicReferenceFieldUpdater<T,V>`
 - Element muß als *volatile* definiert sein
 - Funktioniert nur für Zugriffe über diese Klasse
 - » Dann **immer** diese Zugriffsart verwenden; sonst Probleme!
- Beispiele (AtomicInteger):
 - `addAndGet`: Wert dazuaddieren, Ergebnis zurückgeben
 - `get`, `set`, `compareAndSet`, `decrementAndGet`, `incrementAndGet`, `getAndAdd`, `getAndSet`, ...
 - `weakCompareAndSet`: Wenn der Wert gleich einem erwarteten Wert ist, wird er auf einen neuen Wert gesetzt
 - » Kann jederzeit (auch ohne sichtbaren Grund) fehlschlagen
 - » Implementierung viel effizienter



java.util.concurrent.locks: Überblick

- ReentrantLock: Wie der interne Monitor bei Verwendung von *synchronized*, aber mit Zusatzfunktionen
 - Eigene Queue für Fairness
- ReentrantReadWriteLock: Beliebig viele gleichzeitige Lesezugriffe möglich, aber zu jedem Zeitpunkt nur ein einziger Schreibzugriff (und währenddessen keine Lesezugriffe)
 - Reentrant, Fairness möglich, zurückschalten von Write auf Read (downgrade) möglich
 - » Um Werte während der Änderung abfragen zu können
 - Read → Write unmöglich
 - » Read lock freigeben, dann neuen Write lock anfordern
- Interne Hilfsklassen

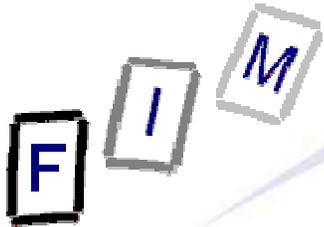


Synchronisation: Fallstricke (1)

```
public synchronized void startIt() {  
    myLock.lock();  
    .....  
}
```

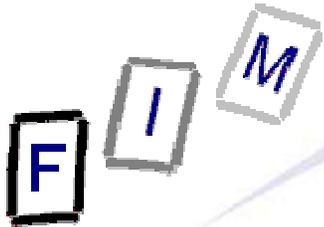
```
public synchronized void endIt() {  
    ....  
    myLock.unlock();  
}
```

- Deadlock ziemlich sicher!
- Warum?



Synchronisation: Fallstricke (1)

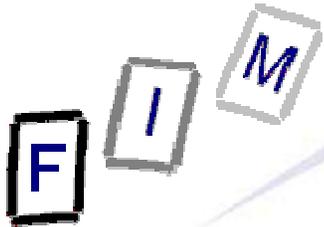
- Muss in `myLock.lock()` gewartet werden, da der lock derzeit belegt ist, wird der Monitor für die Klasse ("synchronized") nicht freigegeben
 - Wait (muss darüber implementiert werden!) gibt immer nur den eigenen Monitor frei, behält aber andere!
- Die Methode "endIt", welche ein anderer Thread aufrufen würde, um den von ihm belegten Lock wieder freizugeben, kann deswegen nicht betreten werden
- Ergebnis:
 - Thread 1 wartet in "`myLock.lock()`" auf den Lock
 - » Besitzt Monitor des Objektes
 - Thread 2 wartet in "synchronized" auf Monitor des Objektes
 - » Besitzt den Lock "`myLock`"



Synchronisation: Fallstricke (2)

```
public synchronized void dolt() {  
    myLock.lock();  
    myLock.unlock(); // Free lock to avoid deadlocks  
    doSomething();  
    myLock.lock(); // Reacquire lock  
    myLock.unlock();  
}
```

- Funktioniert ziemlich sicher nicht (Race conditions)!
- Warum?



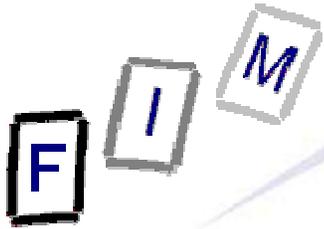
Synchronisation: Fallstricke (2)

```
public synchronized void doIt() {  
    myLock.lock();  
    myLock.unlock();  
    doSomething();  
    myLock.lock();  
    myLock.unlock();  
}
```

Konzeptuell: Beliebige lange Zeit

Theorie: Mehrere Stunden möglich!

- Zwischen "unlock" und "doSomething" kann beliebig lange Zeit liegen: Während dieses Zeitraums gilt **kein** gegenseitiger Ausschluss!



Synchronisation: Fallstricke (3)

- Synchronisierte Collections: Die Lösung!
 - Nicht immer! Beispiel: Hashtable
- Zwei Versionen: "Klassisch" und "Concurrent"
- Beide besitzen Einschränkungen!
 - `if(!table.containsKey(key)) table.put(key,obj);`
 - » Was, wenn zwischen den beiden Methoden entfernt?
 - Daher `putIfAbsent(key,obj);` verwenden, dies ist **atomar**!
- Nächstes Problem: Alle derzeitigen Elemente ausgeben
 - Schreiben und Lesen sind zwar "korrekt", d.h. keine falschen Werte, aber haben **keinen** gegenseitigen Ausschluss!
 - » Iterator: Zustand zu irgendeinem Zeitpunkt; spätere Änderungen sind **vielleicht** berücksichtigt!
- Locking für die gesamte Tabelle ev. nötig!