Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf
Freie Universität Berlin

Überblick

Überblick

- Java Speichermodell
- Abbruch von Java-Threads
- Locks und Waitsets

Speichermodell in Java

Speichermodell und Optimierung

- Exakte Semantik der Manipulation von Daten im Speicher: Speichermodell
- Semantikerhaltende Umsetzung?
- Sequenz der Ausführung von Code kann anders sein als programmiert
- Umordnen durch Compiler:

```
a=a+1;        a=a+2;
b=3;          b=3;
a=a+1;
```

- Parallelisieren durch Prozessor

```
a=a+2;        (a,b)=(a+2,3);
b=3;
```


Optimierung

- Datenhaltung in Registern oder Cache

```
a=a+1;        r1=a+1;       r1=a+1;
b=3;          b=3;          r1++;
a=a+1;        r1++;         a=r1;
              a=r1;         b=3;
```

- Handhabung „langer" Daten
  (Annahme l ist 16 Bit lang, Register sind 8 Bit lang)

```
l=l+257;      r1=l[0];
              r2=l[1];
              incr r1;
              incr r2;
              l[0]=r1
              l[1]=r2
```


Semantikerhaltung

- Entsprechende Optimierungen sind semantikerhaltend *für die sequentielle Ausführung* (as-if-sequential)
- Sie sind sinnvoller Stand der Kunst in der Code- und Ausführungsoptimierung

Semantikerhaltung

- Optimierungen sind so nicht semantikerhaltend *für die nichtsequentielle Ausführung*
- Umordnung:

```
a=a+1;        |             a=a+2;        |
b=3;          |             b=3;          |
              | c=a;                      | c=a;
a=a+1;        |             |             |
```

- Zwischenspeicherung von Daten:

```
r1=a+1;       |             r1=a+1;       |
b=3;          |             r1++;         |
              | c=a;        a=r1;         |
r1++;         |             b=3;          |
a=r1;         |             |             | c=a;
```

Semantikerhaltung

- „Lange Daten“
  (Annahme l ist 16 Bit lang, Register sind 8 Bit lang)

l=l+257;	k=l;	r1=l[0];	
		r2=l[1];	
		incr r1;	
		incr r2;	
		l[0]=r1;	
			k[0]=l[0];
		l[1]=r2;	
			k[1]=l[1];

Synchronisierung

- Abhilfe: Synchronisierung

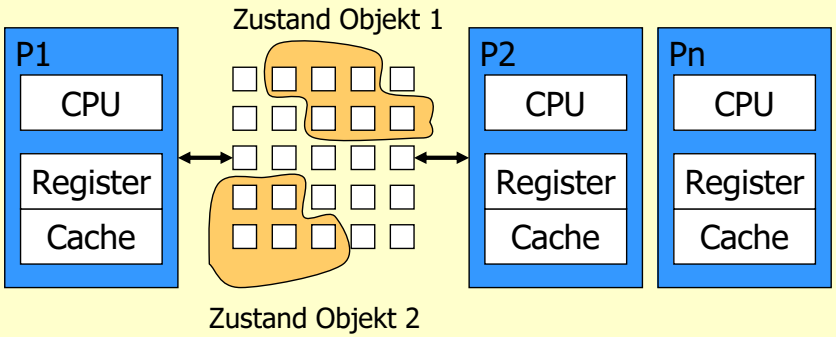
synchronized (x) {l=l+257;}	synchronized(x) {k=l;}	r1=l[0];	
		r2=l[1];	
		incr r1;	
		incr r2;	
		l[0]=r1;	
		l[1]=r2;	
			k[0]=l[0];
			k[1]=l[1];

Synchronisierung

- Probleme:
  - Verbietet optimalere Programmausführungen
  - Erzeugt Mehrarbeit
  - Code ist nicht zugänglich
- Lösung:
  - Code anhand der minimalen Garantien des Speichermodells denken und überprüfen

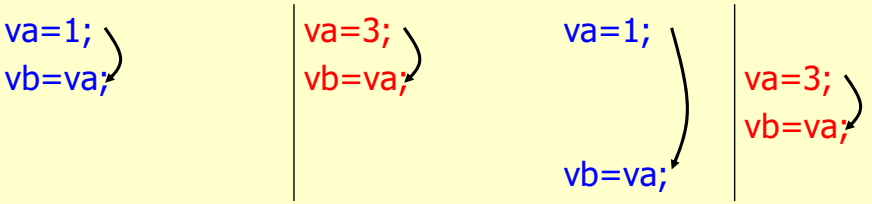
Speichermodell in Java

- Speichermodell: Idealer Mehrprozessorbetrieb
  - Jeder Prozess arbeitet auf einem Prozessor
  - Alle Prozesse teilen sich Speicher
  - Jeder Prozessor hat lokale Kopien des gemeinsamen Speichers (Register, Cache)



Ordnung

- Innerhalb eines Threads herrscht as-if-sequential Semantik
- Zwischen unsynchronisierten Threads gibt es *keinerlei Garantien* mit den Ausnahmen
  - Relative Ordnung zwischen synchronized Methoden und Blöcken bleibt erhalten
  - Relative Ordnung von Operationen auf volatile Feldern bleibt erhalten



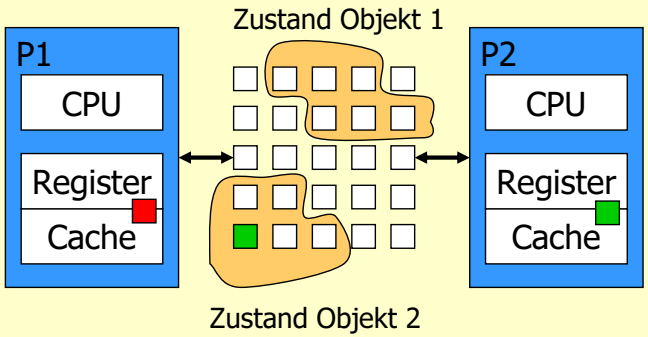
- Stört as-if-sequential Semantik

Atomizität

- Atomizität
  - Atomarer Zugriff:
    - Alle einfachen Typen außer long und double
    - volatile long
    - volatile double
  - Atomarer Zugriff nicht garantiert:
    - long
    - double
- Garantiert, dass atomarem Zugriff liefert
  - den initialen Wert oder
  - den Wert, den ein Prozess komplett geschrieben hat
  - aber kein Zwischenwert konkurrierender Threads
  - aber nicht notwendig den letzten Wert

Synchronisierung und Sichtbarkeit

- Prozessor lädt Wert und ändert Kopie in Register oder Cache:

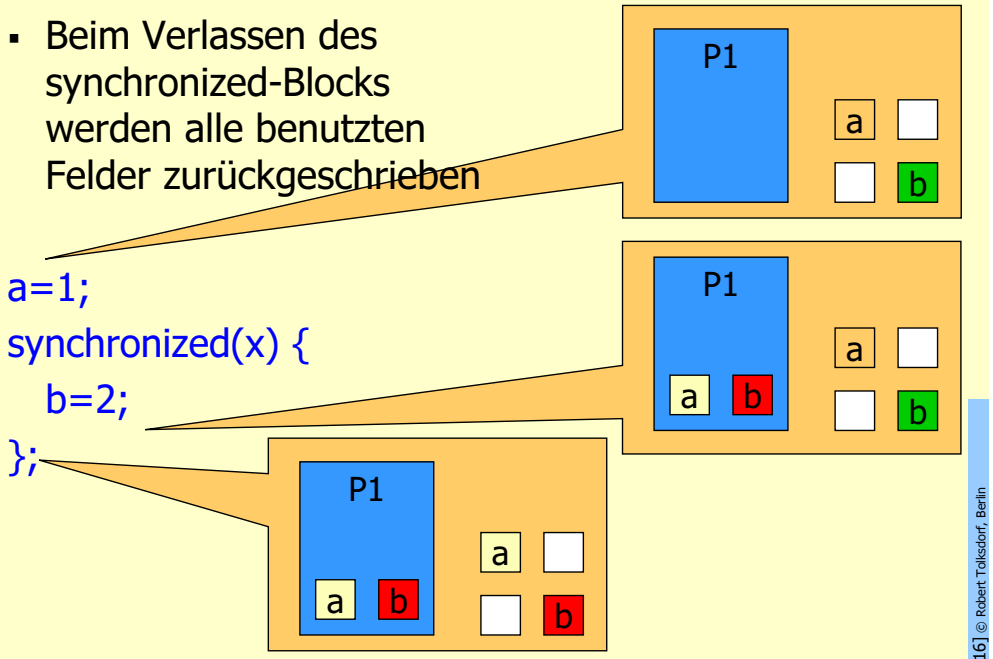


- Wann „weiß" P2 von dem neuen Wert?
- Sichtbarkeitsregeln machen einzige Zusicherungen

Schreiben aller gepufferten Veränderungen

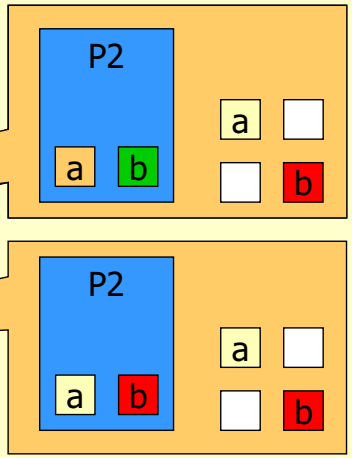
- Beim Verlassen des synchronized-Blocks werden alle benutzten Felder zurückgeschrieben

```
a=1;
synchronized(x) {
  b=2;
};
```



Neuladen aller gepufferten Werte

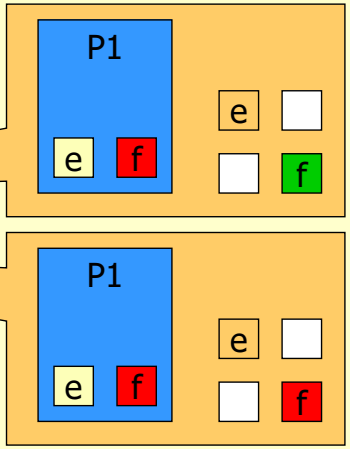
- Beim Betreten des synchronized-Blocks werden alle benutzten Felder neu geladen




```
c=a;
synchronized(x) {
   d=b;
};
```


volatile

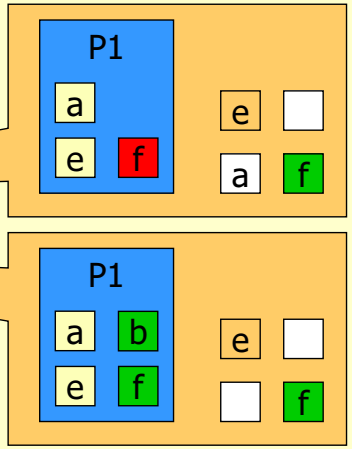
- Sofortiges „Durchschreiben“ nach Änderungen




```
nonvolatilee=3;
volatilef=4;
```


volatile

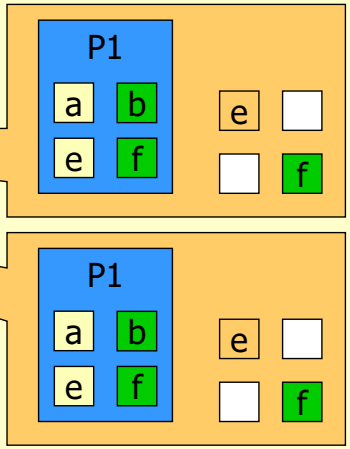
- Jeweiliges Neuladen vor Benutzung




```
a=nonvolatilee;
b=volatilef;
```


Termination

- Bei Termination werden alle Variablen zurückgeschrieben




```
...;
return;
```

- Wenn Thread t_1 ein t_2.join() macht, ist garantiert, dass die Änderungen von t_2 für t_1 sichtbar sind

volatile

- Volatile-Deklaration

```
volatile float f;
```

- Gleiches Datum durch Ausschluss geschützt:

```
class VolatileFloat  {
private float v;
final synchronized void set(float f) {v=f;}
final synchronized float get() {return v;}
final synchronized void inc() {v++;}
}
fo=new VolatileFloat();
```


volatile

f	fobject
gleich bezüglich Ordnung, Sichtbarkeit	
f++; nicht atomar	fo.incr(); atomar
wenig Overhead per Zugriff	mehr Aufwand per Zugriff
mehr Overhead als synchronisierte Zugriffsfolge	Ein Block mit vielen Zugriffen mit einer Synchronisation

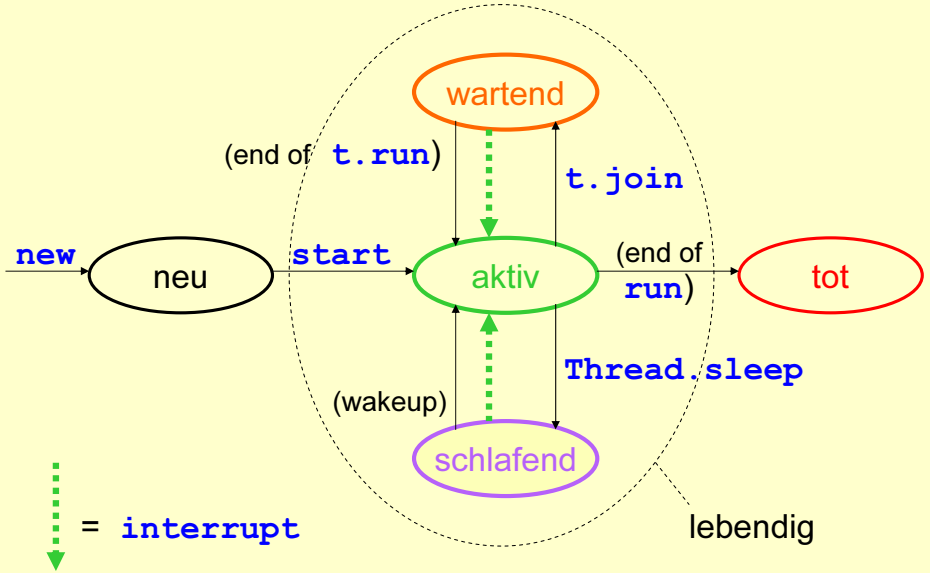
- volatile dann, wenn
  - Synchronisation nicht aus anderen Gründen gebraucht wird (Atomizität mehrerer Zugriffe, Konsistenz mit anderen Feldern, Schreibaktionen von einander unabhängig etc.)
  - Ein Schreiber, viele nebenläufige Leseprozesse
  - „Leichte Synchronisation" konsistente Sichtbarkeit erfordert

Java Threads abbrechen

Kommunikation zwischen Threads

- Threads „kommunizieren"
  - indirekt über wait() und notify()/notifyAll()
  - direkt über start(), join(), interrupt()
- public void interrupt()
  - setzt ein verborgenes Boolesches Attribut interrupt des Threads (das nach der Erzeugung des Threads zunächst false ist) auf true ;
- public boolean isInterrupted()
  - fragt den Zustand dieses Attributs ab
- (Früher gab es ein stop(), das ist deprecated!)

Zustandsübergänge im Bild



Unterbrechung

- t.interrupt()
  - Wenn im Zustand
    - wartend (durch wait())
    - schlafend (durch sleep())

    Wurf von InterruptedException
  - Anderenfalls setzen eines Flags
    - Boolsches Flag ist intern
    - kann mit interrupted() abgefragt werden

Reaktion auf interrupt()

- t schlafend (oder wartend in wait() oder join()):

```
try {
for(;;) {
   sleep(Long.MAX_VALUE);
}
catch (InterruptedException e) {
   System.out.println("I'm interrupted");
}
```

- t weder schlafend noch wartend:

```
for(;;) {
   if (isInterrupted()) break;
}
System.out.println("I'm interrupted");
```


interrupt() ist Aufforderung zur Terminierung

- Exception fangen und terminieren:

```
run() {…
   try {
     for(;;) {
                sleep(Long.MAX_VALUE);
     } catch (InterruptedException e) {
                System.out.println("I'm interrupted but do not care");
     return();
   }…
```

- Exception fangen und eventuell nicht:

```
run() {…
   try {
   for(;;;) {
     sleep(Long.MAX_VALUE);
   }
   catch (InterruptedException e) {
     if (iCare()) {
                System.out.println("I'm interrupted");
                return();
     }}…
```

interrupt() ist Aufforderung zur Terminierung

- Test des Interrupted-Flags
  - muß aktiv geschehen
  - muß verteilt im Thread-Code geschehen
- Zwei Methoden
  - isInterrupted(): Läßt Flag unverändert

```
for(;;) {
  if (isInterrupted()) break;
}
if !(isInterrupted()) System.out.println("Impossible");
```

  - Thread.interrupted(): Löscht Flag

```
for(;;) {
  if (Thread.interrupted()) break;
}
if (isInterrupted()) System.out.println("Impossible");
```


Problem

- Das Interrupted-Flag wird zurückgesetzt durch
  - Thread.interrupted()
  - aber auch durch abgebrochenes sleep(), join(), wait()
- run() in Thread r:

```
while (!Thread.interrupted()) {
  doSomethingComplicated();
}
```

- An anderer Stelle: r.interrupt()
- r terminiert nicht, wenn in doSomethingComplicated ein sleep() ausgeführt wurde und InterruptedException abgefangen wurde!
- Interrupt ist kein gutes Mittel zur Kommunikation zwischen Treads

Locks und Waitsets

3.1.1 Kritische Abschnitte

- Syntax in Java
  Java Statement = ... | SynchronizedStatement
  SynchronizedStatement =
  synchronized (Expression) Block
- Der angegebene Ausdruck muß ein Objekt bezeichnen.
- Der angegebene Block heißt auch kritischer Abschnitt (critical section).
- Semantik:
  Zwischen kritischen Abschnitten, die sich auf das gleiche Objekt (nicht null) beziehen, ist wechselseitiger Ausschluss (mutual exclusion) garantiert.
- Die zu diesem Zweck praktizierte Synchronisation heisst
  - Sperrsynchronisation (locking) oder
  - Ausschlusssynchronisation (exclusion synchronization)

3.1.2 Monitore

- Java praktiziert einen Kompromiss: synchronized ist als Modifier einer Methode in einer Klasse verwendbar:
- Syntax in Java:
  synchronized otherModifiers MethodDeclaration
- Semantik bei Objektmethoden:
  - als wäre der Rumpf ein kritischer Abschnitt mit synchronized(this)
- Semantik bei Klassenmethoden (static):
  - als wäre der Rumpf ein kritischer Abschnitt mit synchronized(className.class)

Sperren in Java

- Abschnitt 8.5 in http://java.sun.com/docs/books/vmspec/2nd-edition/html/Threads.doc.html
  - A *lock* operation by *T* on *L* may occur only if, for every thread *S* other than *T*, the number of preceding *unlock* operations by *S* on *L* equals the number of preceding *lock* operations by *S* on *L*. (Less formally:
    - only one thread at a time is permitted to lay claim to a lock; moreover,
    - a thread may acquire the same lock multiple times and
    - does not relinquish ownership of it until a matching number of *unlock* operations have been performed.)
  - An *unlock* operation by thread *T* on lock *L* may occur only if the number of preceding *unlock* operations by *T* on *L* is strictly less than the number of preceding *lock* operations by *T* on *L*. (Less formally: a thread is not permitted to unlock a lock it does not own.)
- Java Virtual Machine hat Sperroperationen als Instruktionen:
  - *monitorenter*: implementiert lock-Operation
  - *monitorexit*: implementiert unlock-Operation

monitorenter

- **Operation** Enter monitor for object
- **Format** *monitorenter*
  Forms *monitorenter* = 194 (0xc2)
- **Operand Stack** ..., *objectref* ...
- **Description**
  The *objectref* must be of type reference.
  Each object has a monitor associated with it. The thread that executes *monitorenter* gains ownership of the monitor associated with *objectref*. If another thread already owns the monitor associated with *objectref*, the current thread waits until the object is unlocked, then tries again to gain ownership. If the current thread already owns the monitor associated with *objectref*, it increments a counter in the monitor indicating the number of times this thread has entered the monitor. If the monitor associated with *objectref* is not owned by any thread, the current thread becomes the owner of the monitor, setting the entry count of this monitor to 1.
- **Runtime Exception**
  If *objectref* is null, *monitorenter* throws a NullPointerException.

monitorexit

- **Operation** Exit monitor for object
- **Format** *monitorexit*
- **Forms** *monitorexit* = 195 (0xc3)
- **Operand Stack** ..., *objectref* ...
- **Description**
  The *objectref* must be of type reference.
  The current thread should be the owner of the monitor associated with the instance referenced by *objectref*. The thread decrements the counter indicating the number of times it has entered this monitor. If as a result the value of the counter becomes zero, the current thread releases the monitor. If the monitor associated with *objectref* becomes free, other threads that are waiting to acquire that monitor are allowed to attempt to do so.
- **Runtime Exceptions**
  If *objectref* is null, *monitorexit* throws a NullPointerException.
  Otherwise, if the current thread is not the owner of the monitor, *monitorexit* throws an IllegalMonitorStateException.
  Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in Section 8.13 and if the second of those rules is violated by the execution of this *monitorexit* instruction, then *monitorexit* throws an IllegalMonitorStateException.

synchronized

- synchronized legt lock/unlock Klammern um Block:
  - synchronized void doit() {*Block*} mit o.doit:
    monitorenter o;
    Block;
    monitorexit o;
  - synchronized(x) {*Block*}
    monitorenter x;
    Block;
    monitorexit x;
- Dynamische Klammerung über Zahler (siehe Def monitorenter und monitorexit)

3.2.2.3 Ereignissynchronisation in Java

- mittels Operationen der Wurzelklasse Object –
- *sofern* der ausführende Thread das Objekt gesperrt hat (*andernfalls* IllegalMonitorStateException):
- void wait() throws InterruptedException
  - blockiert und gibt die Objektsperre frei
- void notify()
  - weckt einen blockierten Thread auf (sofern vorhanden), gibt aber noch nicht die Sperre frei (vgl. 3.2.2.2 2))
  - aufgeweckter Thread wartet, bis er die Sperre wiederbekommt
- void notifyAll()
  - entsprechend für *alle* blockierten Threads

Ablauf von wait/notify/notifyAll

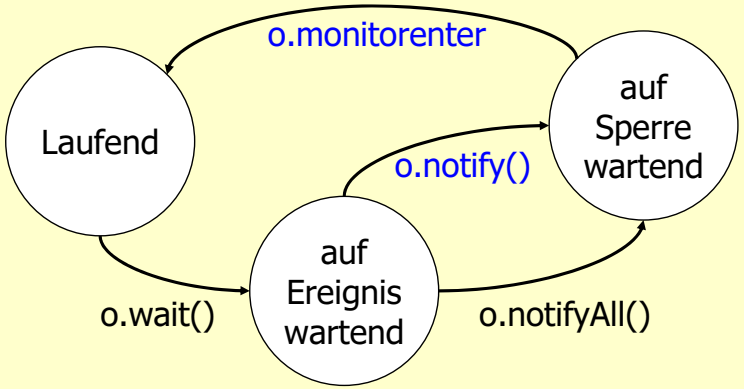
- Objekt O hat
  - Sperre (ca. Thread x Zähler)
  - Waitset (Menge von Threads)
- T macht wait() auf O
  - Voraussetzung: T hat Sperre (Zähler ist N)
  - T wird in das Waitset eingefügt
  - T wird blockiert (ist nicht zur Ausführung bereit)
  - N unlock-Operationen werden durchgeführt

Ablauf von wait/notify/notifyAll

- Drei Ereignisse können T beeinflussen
  - ein notify wird auf O gemacht und T wird zur Benachrichtigung ausgewählt
  - ein notifyAll wird auf O gemacht
- Dann:
  - wait war mit einem Timeout t versehen und dieser ist abgelaufen
  - T wird aus waitset enfernt
  - T wird als bereit zur Ausführung markiert
  - T führt lock-Operation aus (in Konkurrenz mit anderen)
  - N-1 zusätzliche lock-Operationen werden ausgeführt
  - Ein „Resume T" wird ausgeführt, wait() kehrt zurück, Zustand ist unverändert

Ablauf von wait/notify/notifyAll

- Thread S, der notify() und notifyAll auf O macht, hat Sperre auf O
- T kann erst dann versuchen, Sperre zu erhalten wenn S Sperre aufgibt
- T steht dann in Konkurrenz mit anderen



Zusammenfassung

Zusammenfassung

- Java Speichermodell
  - Optimierung kann Code umstellen und in Konflikt mit as-if-sequential Semantik stehen
  - Synchronisierung
  - Umgang mit gepufferten Werten
- Abbruch von Java-Threads
  - interrupt()
  - Nur Aufforderung zur Termination
- Locks und Waitsets
  - monitorenter und monitorexit in JVM
  - Waitset für wait()
  - Zweistufige Konkurrenz um Ereignis und Sperre