

Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf
Freie Universität Berlin

[1] © Robert Tolksdorf, Berlin

Überblick

[2] © Robert Tolksdorf, Berlin

Überblick

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

[3] © Robert Tolksdorf, Berlin

Speichermodell in Java

[4] © Robert Tolksdorf, Berlin

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;
```

[5] © Robert Tolksdorf, Berlin

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;           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
```

[6] © Robert Tolksdorf, Berlin

Semantikerhaltung

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

[7] © Robert Tolksdorf, Berlin

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

[8] © Robert Tolksdorf, Berlin

Semantikerhaltung

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

<code>l=l+257;</code>	<code>k=l;</code>	<code>r1=l[0];</code>	
		<code>r2=l[1];</code>	
		<code>incr r1;</code>	
		<code>incr r2;</code>	
		<code>l[0]=r1;</code>	
		<code>l[1]=r2;</code>	
			<code>k[0]=l[0];</code>
			<code>k[1]=l[1];</code>

[9] © Robert Tolksdorf, Berlin

Synchronisierung

- Abhilfe: Synchronisierung

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

[10] © Robert Tolksdorf, Berlin

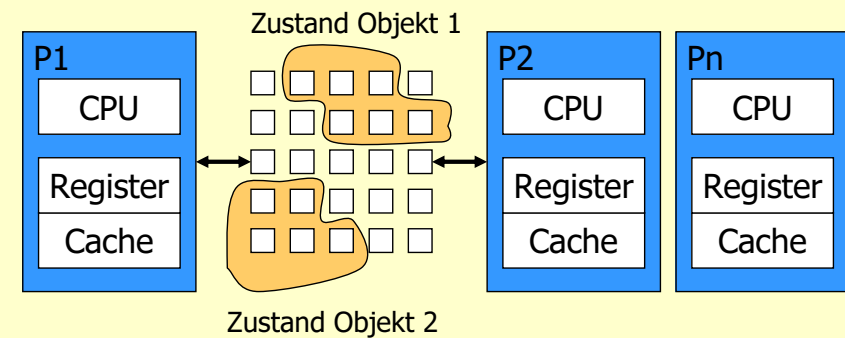
Synchronisierung

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

[11] © Robert Tolksdorf, Berlin

Speichermodell in Java

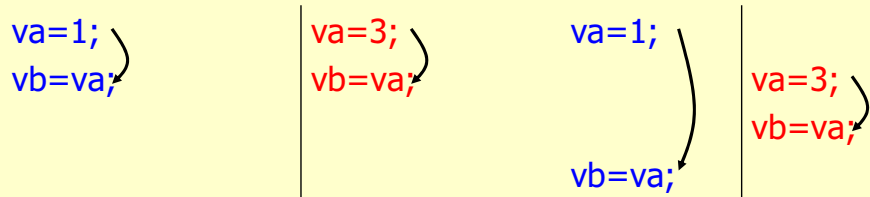
- Speichermodell: Idealer Mehrprozessorbetrieb
 - Jeder Prozess arbeitet auf einem Prozessor
 - Alle Prozesse teilen sich Speicher
 - Jeder Prozessor hat lokale Kopien des gemeinsamen Speichers (Register, Cache)



[12] © Robert Tolksdorf, Berlin

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

[13] © Robert Tolksdorf, Berlin

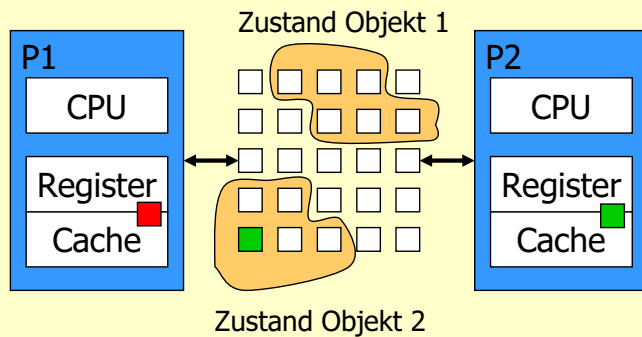
Atomizität

- 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

[14] © Robert Tolksdorf, Berlin

Synchronisierung und Sichtbarkeit

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



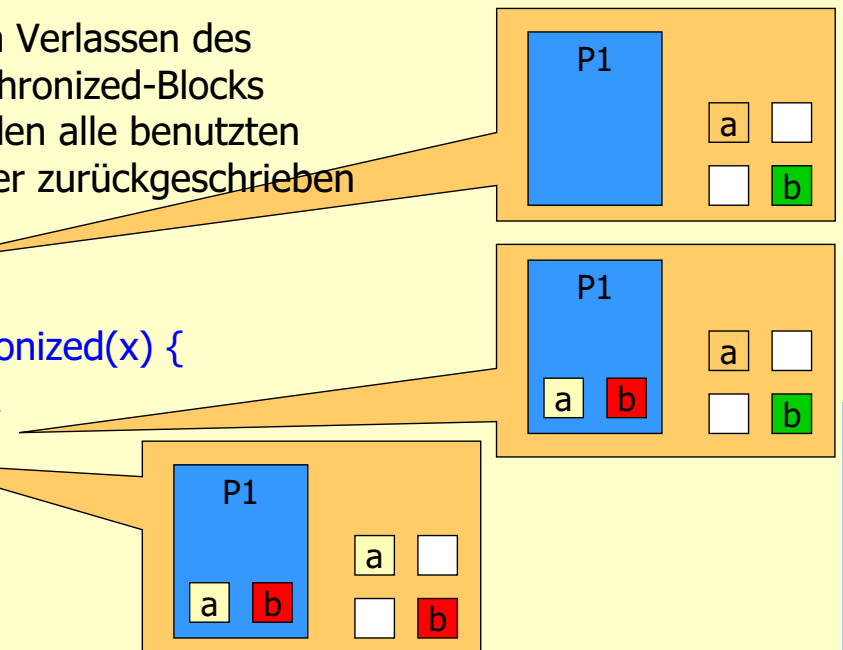
- Wann „weiß“ P2 von dem neuen Wert?
- Sichtbarkeitsregeln machen einzige Zusicherungen

[15] © Robert Tolksdorf, Berlin

Schreiben aller gepufferten Veränderungen

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

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

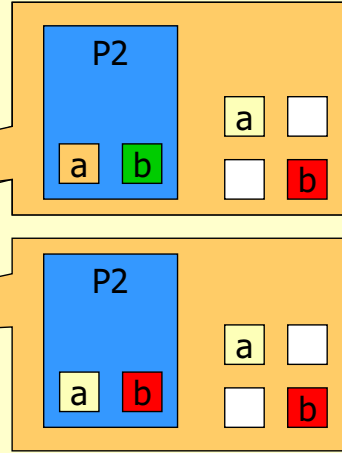


[16] © Robert Tolksdorf, Berlin

Neuladen aller gepufferten Werte

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

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

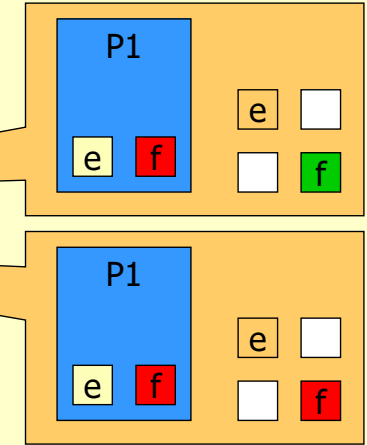


[17] © Robert Tolksdorf, Berlin

volatile

- Sofortiges „Durchschreiben“ nach Änderungen

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

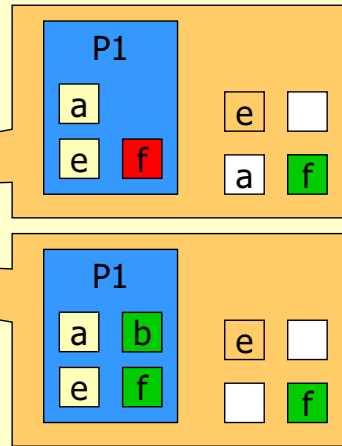


[18] © Robert Tolksdorf, Berlin

volatile

- Jeweiliges Neuladen vor Benutzung

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

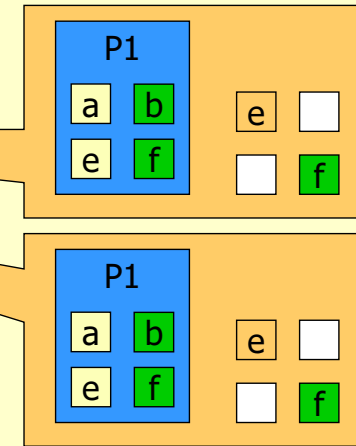


[19] © Robert Tolksdorf, Berlin

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

[20] © Robert Tolksdorf, Berlin

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();`

[21] © Robert Tolksdorf, Berlin

volatile

f	fobject
gleich bezüglich Ordnung, Sichtbarkeit	
<code>f++</code> ; nicht atomar	<code>fo.incr()</code> ; 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

[22] © Robert Tolksdorf, Berlin

Java Threads abbrechen

[23] © Robert Tolksdorf, Berlin

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!)

[24] © Robert Tolksdorf, Berlin

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");`

[29] © Robert Tolksdorf, Berlin

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

[30] © Robert Tolksdorf, Berlin

Locks und Waitsets

[31] © Robert Tolksdorf, Berlin

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
 - Sperrsynchrisation (locking) oder
 - Ausschlusssynchronisation (exclusion synchronization)

[32] © Robert Tolksdorf, Berlin

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)`

[33] © Robert Tolksdorf, Berlin

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

[34] © Robert Tolksdorf, Berlin

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`.

[35] © Robert Tolksdorf, Berlin

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`.

[36] © Robert Tolksdorf, Berlin

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 Zähler (siehe Def monitorenter und monitorexit)

[37] © Robert Tolksdorf, Berlin

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

[38] © Robert Tolksdorf, Berlin

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

[39] © Robert Tolksdorf, Berlin

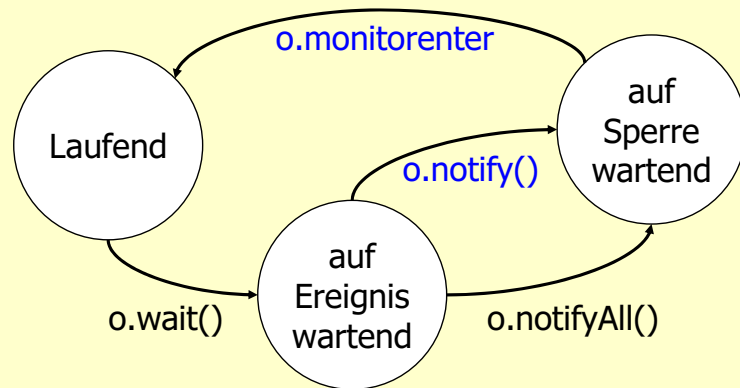
Ablauf von wait/notify/notifyAll

- 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 entfernt
 - `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

[40] © Robert Tolksdorf, Berlin

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



[41] © Robert Tolksdorf, Berlin

Zusammenfassung

[42] © Robert Tolksdorf, Berlin

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

[43] © Robert Tolksdorf, Berlin