Freie Universität Berlin

Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf

Freie Universität Berlin

Überblick

Überblick

- Sperrsynchronisation
  - Sperrsynchronisation in Datenbanken
- Bedingungssynchronisation
  - Verzögerte Monitore
  - Ereignissynchronisation
  - Nachrüsten von Synchronisation

3.2.2.3 Ereignissynchronisation in Java

Freie Universität Berlin

- 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 (signal-and-continue)
  - aufgeweckter Thread wartet, bis er die Sperre wiederbekommt
- void notifyAll()
  - entsprechend für *alle* blockierten Threads

Ereignissynchronisation in Java

Freie Universität Berlin

- Mit anderen Worten:

- Für jedes Objekt (Monitor) gibt es nur ein „Standard-Ereignis“ – das notify-Ereignis

- *Fairness:*
  - *keinerlei* Garantien!
  - Insbesondere konkurrieren nach notifyAll *alle* aufgeweckten Threads *und* die von außen hinzukommenden Threads um die Sperre.
- *Warten mit Timeout:*
- void wait(long msecs)
  - wie wait, mit dem Zusatz, daß der blockierte Thread nach der angegebenen Zeit geweckt wird.

Beispiel **Betriebsmittel** (Ressource, *resource)*

Freie Universität Berlin

- wie z.B. *Drucker* (3.2.2.1🡐) oder *Sperre*:

```
class Resource {
  private boolean busy;

  public synchronized void request() {
    if (busy) wait();
    busy = true;
  }
  public synchronized void release() {
    busy = false;
    notify();
  }
}
```

Beispiel Betriebsmittel-Pool

- Beispiel **Betriebsmittel-Pool** (z.B. Betriebsmittel = Speicherblock)
- *hier: nur die Synchronisationsstruktur*

```
class Resources {
   private int available;
   ...
   public synchronized void request(int claim) {
      while (claim > available) wait();
      available -= claim;
   }
   public synchronized void release(int free) {
     available += free;
     notifyAll();
   }
}
```

Beispiel *Puffer* (3.2.2.2🡐)

Freie Universität Berlin

```
public synchronized void send(Msg m) {
      if (count==size) wait();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
      notify();
}

public synchronized Msg recv() {
      if (count==0) wait();
      Msg m = cell[front];
      count--;
      front = (front+1)%size;
      notify();
      return m;
}
```

ist nicht korrekt !

2. Versuch

Freie Universität Berlin

```
public synchronized void send(Msg m) {
      if (count==size) wait();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
      if (count==1) notify();
}

public synchronized Msg recv() {
      if (count==0) wait();
      Msg m = cell[front];
      count--;
      front = (front+1)%size;
      if (count==size-1) notify();
      return m;
}
```

ist nicht korrekt !

3. Versuch

Freie Universität Berlin

```
public synchronized void send(Msg m) {
      while (count==size) wait();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
      if (count==1) notify();
}

public synchronized Msg recv() {
      while (count==0) wait();
      Msg m = cell[front];
      count--;
      front = (front+1)%size;
      if (count==size-1) notify();
      return m;
}
```

ist nicht korrekt !

4. Versuch – **korrekt**

Freie Universität Berlin

```
public synchronized void send(Msg m) {
      while (count==size) wait();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
      if (count==1) notifyAll();
}

public synchronized Msg recv() {
      while (count==0) wait();
      Msg m = cell[front];
      count--;
      front = (front+1)%size;
      if (count==size-1) notifyAll();
      return m;
}
```

Beispiel *Puffer-Variante* (3.2.2.2🡐)

Freie Universität Berlin

```
private boolean urgent;
public synchronized void send(Msg m) {
      while (count==size) wait();
      cell[rear] = m;
      if (m.isUrgent()) urgent = true;
      else              { count++; rear = (rear+1)%size; }
      notifyAll();
}
public synchronized Msg recvUrgent() {
      while (!urgent) wait();
      urgent = false;
      notifyAll();
      return cell[rear];
}
public synchronized Msg recv() {
      while (count==0 || urgent) wait();.....}
```

Ausnahmen

- **Merke:**
- Verzögerte Operation mit Ausnahmemeldung *(nicht Java)*

```
op(...) PRE  C
    WHEN G
    {.....}
```

- **in Java** realisieren durch das Idiom

```
op(...) throws NotC {
    if (!C) throw new notC();
    while  (!G) wait();
    .....  notifyAll(); }

others(...) .. notifyAll(); }
```

Beispiel Betriebsmittel-Pool

- Beispiel Betriebsmittel-Pool
  (z.B. Betriebsmittel = Speicherblock)
- *hier: nur die Synchronisationsstruktur*

```
MONITOR Resources {        // this is NOT JAVA
  private final int max = ...;
  private int available = max;

  public void request(int claim)
     PRE  claim <= max
     WHEN claim <= available {
     available  -= claim;    }
  public void release(int free)  {
     available += free;      }
}
```

Beispiel Betriebsmittel-Pool

- Beispiel Betriebsmittel-Pool
  (z.B. Betriebsmittel = Speicherblock)
- *hier: nur die Synchronisationsstruktur*

```
class Resources {        // this IS JAVA
  private final int max = ...;
  private int available = max;
  public synchronized void request(int claim) ... {
      if   (claim > max) throw new ...
      while (claim > available) wait();
      available  -= claim;
  }
  public synchronized void release(int free) {
      available += free;
      notifyAll();
  }
}
```

Unabhängige Ereignisse

- Unabhängige Ereignisse (d.h. nicht an Daten gebunden):
  - `Object x = new Object();` „Ereignis x“
  - `synchronized(x){x.wait();}` „warte auf x“
  - `synchronized(x){x.notifyAll();}` „x tritt ein“

- *Achtung:*
  Ereignis „geht verloren“, wenn keiner wartet

Condition Objekte in java.util.concurrent

Freie Universität Berlin

- An Objektsperren gibt es nur eine einzige Bedingung auf die mit wait() gewartet werden kann
- Condition Objekte machen diese Bedingung explizit
- Ein Sperrobjekt kann mehrere Bedingungsobjekte haben
- In der Lock Schnittstelle weiterhin:
  Condition newCondition()
  Erzeugt und bindet ein Condition-Objekt an Sperre
- Methoden
  - void await()
  - boolean await(long time, TimeUnit unit)
  - boolean awaitUntil(Date deadline)
    Auf Signal warten
  - void signal()
    Entspricht notify()
  - void signalAll()
    Entspricht notifyAll()
- Können fair sein

Beispiel


```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
```

Beispiel

```
public Object take() throws InterruptedException {
 lock.lock();
 try {
   while (count == 0)
     notEmpty.await();
   Object x = items[takeptr];
   if (++takeptr == items.length) takeptr = 0;
   --count;
   notFull.signal();
   return x;
 } finally {
   lock.unlock();
 }
}
}
```

Implementierungsexkurs: Locks und Waitsets in Java (1.4)

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.)

Sperren in Java

- Abschnitt 8.5 in http://java.sun.com/docs/books/vmspec/2nd-edition/html/Threads.doc.html
  - 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 Zähler (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 (signal-and-continue)
  - 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)

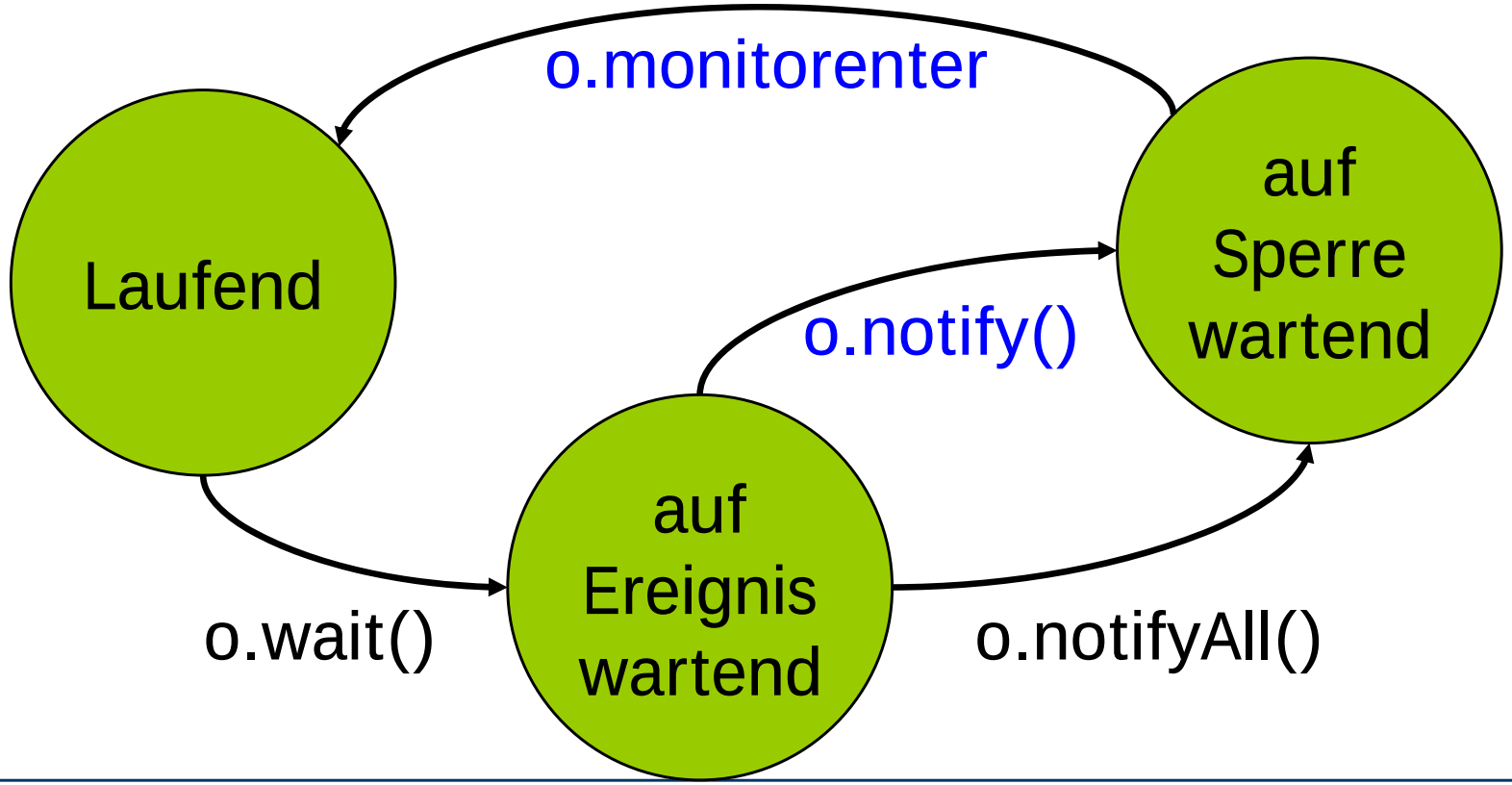
- T macht wait() auf O
  - Voraussetzung: T hat Sperre (Zähler ist n)
  - T wird in das Waitset eingefügt
  - T wird blockiert (ist nicht zur Ausführung bereit)
  - n unlock-Operationen werden durchgeführt

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



3.2.3 Nachrüsten von Synchronisation (vgl. 3.1.3)

Freie Universität Berlin

- am Beispiel
  Schlange mit exceptions ➔ *Puffer ohne exceptions*

```
class LinearQueue<Message>
      implements Queue<Message> {
  public final int size;
  protected   int count;
  .....
  public void append(Message m)
       PRE count < size {.....}
  public Message remove()
       PRE count > 0    {.....}
  public int length() {.....}
}
```

mit Vererbung

- … mit Vererbung: umdefinierte Operationen
  (*ohne exceptions*!)

```
MONITOR Buffer<Message>
        extends LinearQueue<Message> {
  public Buffer(int size)  {super(size);}
  public void append(Message m)
      WHEN count < size {super.append(m);}
  public Message remove()
      WHEN count > 0    {return super.remove();}
  public int length()  {return super.length();}
 }
```

- *Achtung!*
  Bei Einführung anderer Namen – z.B. send statt append – würden die unsynchronisierten Operationen sichtbar bleiben!

mit Delegation

- … mit Delegation: (ebenfalls *ohne exceptions*)

```
MONITOR Buffer<Message>
        implements Queue<Message> {
  private final Queue<Message> q;

  public  Buffer(int size) {
        q = new LinearQueue<Message>(size);

  public void append(Message m)
      WHEN q.length() < q.size  {q.append(m);}
  public Message remove()
      WHEN q.length() > 0 {return q.remove();}
  public int length() {return q.length();}
}
```

Zusammenfassung

Zusammenfassung

- Bedingungssynchronisation
  - Verzögerte Monitore
    - Warteanweisung AWAIT
    - Wachen WHEN
  - Ereignissynchronisation
    - Ereignisobjekte WAIT/SIGNAL
    - Signalvarianten
    - Ereignisse in Java
    - Locks und Waitsets in Java
      - monitorenter und monitorexit in JVM
      - Waitset für wait()
      - Zweistufige Konkurrenz um Ereignis und Sperre
  - Nachrüsten von Synchronisation
    - Vererbung
    - Delegation