Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf
Basiert auf ALP IV, SS 2003
Klaus-Peter Löhr
Freie Universität Berlin

Überblick

Überblick

- Bedingungssynchronisation
  - Verzögerte Monitore
  - Ereignissynchronisation
  - Nachrüsten von Synchronisation

3.2 Bedingungssynchronisation

- *Motivation:*
- Wenn für eine Operation auf einem Objekt die *Voraussetzung nicht erfüllt* ist, ist es häufig angebracht, auf deren Erfüllung zu **warten**, anstatt false zu liefern oder eine *Ausnahme* zu melden.
- *Beispiele:*
  Virtueller Drucker (3.1.1←), Puffer(3.1.2 ←)

3.2.1 Verzögerte Monitore

- Verzögerter Monitor *(delayed monitor)* =

- Monitor, in dem auf die *Erfüllung einer Bedingung* gewartet wird
- Beim Blockieren wird der *Ausschluss aufgegeben*, so daß ein anderer Prozess in den Monitor eintreten kann – dessen Aktionen vielleicht dazu führen, daß die Bedingung erfüllt ist
- Ein im Monitor blockierter Prozess kann *frühestens* dann fortfahren, wenn ein anderer Prozess den Monitor verlässt – oder selbst blockiert.
- Blockierte Prozesse haben *Vorrang* vor Neuankömmlingen.

3.2.1.1 Warteanweisungen

- Syntax (Nicht Java!)
  - Statement = . . . | AwaitStatement
  - AwaitStatement = AWAIT Condition ;
- Diese Warteanweisung ist nur im statischen Kontext eines Monitors erlaubt, und die gemeinsamen Variablen in der *Condition* müssen Monitorvariable sein.
- *Semantik:*
  - Wenn nach der Warteanweisung fortgefahren wird, ist die angegebene Bedingung garantiert erfüllt.
  - Muß gewartet werden, wird der Monitor freigegeben.
  - *Fairness:* blockierte Prozesse haben Vorrang (s.o.)

Beispiel *Drucker* (3.1.1←)

- hier nur mit request und vereinfachtem release:

```
MONITOR Printer {
private Thread owner;

public void request() {
      AWAIT owner==null;
      owner = Thread.currentThread();
}
public void release() {
    owner = null;
}
}
```

- Statt Thread owner würde auch boolean busy reichen ➔

Beispiel *Implementierung einer Sperre* (3.1.5←)

```
MONITOR Lock {
  private boolean lock;
  public void lock() {
     AWAIT !lock;
     lock = true;
  }
  public void unlock() {
     lock = false;
  }
}
```

- Wohlverhalten der Benutzer vorausgesetzt, d.h. alternierende x.lock(), x.unlock()!

Bsp. *Implementierung einer Lese/Schreibsperre* (3.1.5←)

```
MONITOR RWlock {
private int readers;
private int writers;

public void Rlock() {
    AWAIT writers==0;
    readers++; }
public void Wlock() {
    AWAIT readers==0 && writers==0;
    writers++; }
public void Runlock() {
    readers--; }
public void Wunlock() {
    writers--; }
}
```


Beispiel *Puffer* (3.1.2 ←)

```
public void send(M m) {
      AWAIT count<size;
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
}
public M recv() {
      AWAIT count>0;
      M m = cell[front];
      count--;
      front = (front+1)%size;
    return m;
}
```


3.2.1.2 Wachen

- *Deklarative Variante* der Warteanweisung:
  - Wartebedingung als **Wache** *(guard) vor* einem Operationsrumpf eines Monitors (*gesperrt* wird allerdings *vor* Auswertung der Wache)
- *Vorteil:*
  Voraussetzung *(precondition) bewirkt* Synchronisation
- Beispiel (**nicht Java!**):

```
public void send(M m) WHEN count<size {
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
}
```

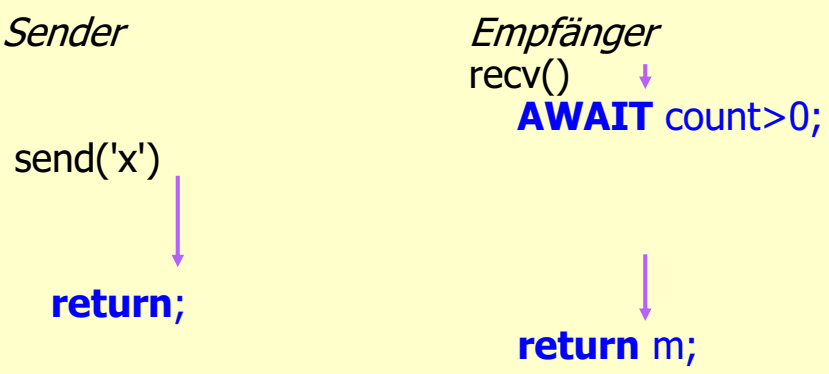

Implementierung?

- **Effiziente Übersetzung** von **AWAIT/WHEN** ist schwierig !

1. Nach jeder Zuweisung bei allen wartenden Prozessen die Wartebedingungen zu überprüfen ist undenkbar.
2. Daher die Forderung, daß die gemeinsamen Variablen in diesen Bedingungen nur Monitorvariable sein dürfen: damit kann sich die Überprüfung auf den Monitor, bei dem zugewiesen wird, beschränken (sofern – weitere Forderung! – es sich nicht um Verweisvariable handelt!).
3. Es genügt, die Wartebedingungen dann zu überprüfen, wenn der Monitor freigegeben wird, also beim Verlassen des Monitors oder bei einem Warten bei **AWAIT**.
   Das kann immer noch ineffizient sein.

3.2.1.3 Korrektheit

- Serialisierbarkeit:
  wie in 3.1.7, Def. 2, auch bei Blockaden
- Beispiel Puffer:



- produziert gleiches Ergebnis ("",'x') wie send recv

Spezifikation und Verifikation

- Spezifikation und Verifikation fallen bei Wache leichter als bei *Warteanweisung*:

- { P } op(...); { Q } mit P = C & G & U , d.h.
- Bezug auf Implementierung :
  - Nichterfüllung von C(ondition) : löst Ausnahmemeldung aus
  - Nichterfüllung von G(uard) : löst Blockade aus
  - Nichterfüllung von U(ndef) : bewirkt undefiniertes Verhalten

- *Programmiersprachliche Umsetzung* – erfordert *deklarative* Auslösung von Ausnahmemeldungen:
- op(...) **PRE** C **WHEN** G **{.....}**

Spezifikation und Verifikation

- **Spezifikation** bei *Warteanweisung:*

```
{ if(!C) throw ...
  ......;     // op1 mit Effekt 1 (problematisch)
  AWAIT G;
  ......;  // op2 mit Effekt 2
}
```

- (entsprechend 3,4,...)

- *Spezifikation*
  - { P } op(...); { Q }
  - { P1 } op1 { Q1 ∧ ¬ G }
  - { P2 ∧ G } op2 { Q2 }

- Bei mehrfachen **AWAIT**s nicht mehr praktikabel !

3.2.2 Ereignissynchronisation

- Prozesse warten auf Ereignisse *(events)*,
  die von anderen Prozessen ausgelöst werden.

- *Mehrere* Prozesse können auf *ein* Ereignis warten;
  - beim Eintreten des Ereignisses werden entweder
    - *alle* Prozesse oder
    - nur *ein* Prozess aufgeweckt;
  - das Ereignis „geht verloren", wenn *keiner* wartet.

3.2.2.1 Ereignisobjekte

- Systemklasse EVENT (auch Condition , Signal),
- ausschließlich zur Benutzung *in Monitoren* (**nicht Java!**):

```
class EVENT {...
public void WAIT()   {...}
    // releases monitor and
    // waits for this.SIGNAL()
public void SIGNAL() {...}
    // awakes one process blocked
    // in this.WAIT(), if any
public int WAITING() {...}
    // number of processes blocked
    // in this.WAIT()
}
```


Fairness

- *Fairness* bei Monitor-Freigabe:
- im Monitor blockierte Prozesse haben Vorrang vor eintrittswilligen Prozessen;
- bei SIGNAL: wer länger wartet hat Vorrang (FCFS, First-Come-First-Served).

Beispiel *Drucker* (3.2.1.1←)

```
MONITOR Printer {
private Thread owner = null;
private final EVENT free = new EVENT();
    // transition to owner==null

public void request() {
    if(owner!=null) free.WAIT();
    // { owner==null }
     owner = Thread.currentThread(); }
public void release() {
    owner = null;
    // { owner==null }
    free.SIGNAL(); }
}
```

wichtige Dokumentation!

3.2.2.2 Signal-Varianten

1. *signal-and-wait:*
   - aus WAIT aufgeweckter Prozess übernimmt Monitor, und aufweckender Prozess *blockiert in* SIGNAL (!).
   - *Begründung:* Monitorübergabe ohne Zustandsänderung.
2. *signal-and-continue:*
   - aus WAIT aufgeweckter Prozess übernimmt Monitor erst dann, wenn aufweckender Prozess ihn freigibt.
   - *Begründung:* Effizienz.
3. *signal-and-return:*
   - SIGNAL ist mit Verlassen des Monitors verbunden.
   - *Begründung:* Vorteile von 1) und 2), begrenzter Nachteil.

Beispiel *Puffer* (3.2.1.1←)

```
private final EVENT notfull  = new EVENT ();
private final EVENT notempty = new EVENT ();
public void send(Msg m) {
      if(count==size) notfull.WAIT();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
      notempty.SIGNAL();
}
public Msg recv() {
      if(count==0) notempty.WAIT();
      Msg m = cell[front];
      count--;
      front = (front+1)%size;
      notfull.SIGNAL();
      return m;
}
```


Beispiel *Puffer* (3.2.1.1←)

```
private final EVENT notfull  = new EVENT ();
private final EVENT notempty = new EVENT ();
public void send(Msg m) {
      if(count==size) notfull.WAIT();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
      notempty.SIGNAL(); }
public Msg recv() {
      if(count==0) notempty.WAIT();
      Msg m = cell[front];
      count--;
      front = (front+1)%size;
      notfull.SIGNAL();
      return m;
}
```

wäre nur mit 2) korrekt !

Variante mit drei Ereignissen

- Variante mit drei Ereignissen (korrekt für 2)):

```
private final EVENT urgent   = new EVENT ();
private final EVENT notfull  = new EVENT ();
private final EVENT notempty = new EVENT ();
public void send(Msg m) {
      if(count==size) notfull.WAIT();
      cell[rear] = m;
      if(m.isUrgent()) urgent.SIGNAL();
      else {count++;
            rear = (rear+1)%size;
            notempty.SIGNAL(); }
}
public Msg recvUrgent() {
      urgent.WAIT();
      notfull.SIGNAL();
      return cell[rear];
}
```


Beispiel *Zwei Drucker*

```
MONITOR TwoPrinters {
      // "normal" and "special"
      // special includes normal capabilities
private boolean normalbusy, specialbusy;
private final EVENT printerfree = new EVENT ();
private final EVENT specialfree = new EVENT ();

public boolean request(boolean special) {.....}

public void    release(boolean special) {.....}
}
```

Beispiel *Zwei Drucker,* Implementierung

```
public boolean request(boolean special) {
      if(special){
         if(specialbusy) specialfree.WAIT();
           else          specialbusy = true;
     else /* normal */ {
         if(normalbusy & specialbusy) printerfree.WAIT();
           if(normalbusy) specialbusy = true;
           else {normalbusy  = true; return false;} }
      return true;
}

public void release(boolean special) {
     if(special)
         if(specialfree.waiting()>0)
               specialfree.SIGNAL();
           else {specialbusy = false;
                 printerfree.SIGNAL();
     else /* normal */ {
           normalbusy = false; printerfree.SIGNAL(); }
}
```

- (ist korrekt für jede der 3 SIGNAL-Varianten)

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

Ereignissynchronisation in Java

- 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)*

- 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←)

```
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

```
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

```
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**

```
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←)

```
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
- *Achtung, Achtung:*
  kein Ersatz für *Events* gemäß 3.2.2.1 !

```
synchronized void op(){ .....
          synchronized(x){x.wait();} ... }
```

- führt zu **Verklemmung** !

3.2.3 Nachrüsten von Synchronisation (vgl. 3.1.3)

- 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
  - Nachrüsten von Synchronisation
    - Vererbung
    - Delegation