

Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

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

[1] © Peter Lühr, Robert Tolksdorf, Berlin

Überblick

[2] © Peter Lühr, Robert Tolksdorf, Berlin

Überblick

- Semaphore
 - Operationen
 - Sperren
 - Puffer
- Nebenläufigkeit und Objektorientierung
 - Vererbungsanomalien
 - Abhilfen
 - Aktive Objekte

[3] © Peter Lühr, Robert Tolksdorf, Berlin

3.3 Semaphore (Dijkstra 1968)

- sind einfache Synchronisationsobjekte, ähnlich wie Ereignisse, aber unabhängig von Monitoren:
 - (das) Semaphore: altes Flügelsignal – wie bei der Bahn
 - (engl. semaphore: auch Flaggensignal bei der Marine)



- Operationen:
 - P() steht für (holl.) passieren
 - V() steht für (holl.) verhogen oder vrijgeven
- Synonyme für P/V:
 - up/down ,
 - wait/signal

[4] © Peter Lühr, Robert Tolksdorf, Berlin

3.3.1 Abstrakte Definition

- Die Anzahl der abgeschlossenen **P/V**-Operationen auf einem bestimmten, mit n initialisierten Semaphor sei p bzw. v .
- Dann gilt die Invariante:
$$P \leq v+n$$
- Mit anderen Worten:
 - V kann stets ausgeführt werden,
 - P aber nur dann, wenn $p < v+n$

[5] © Peter Lühr, Robert Tolksdorf, Berlin

Definition/2

- des Typs Semaphor mit Hilfe eines verzögerten Monitors:

```
MONITOR Semaphore {
  private int signals;
  public Semaphore(int init)
    PRE  init >= 0 {signals = init;}
    „Vorschuss von Signalen“

  public void P()
    WHEN signals > 0 {signals--;}
    „verbraucht Signal“

  public void V()  {signals++;}
  }
  „erzeugt Signal“
```

- Fairness: FCFS

[6] © Peter Lühr, Robert Tolksdorf, Berlin

Semaphore

- Beachte: **P/V** entsprechen einfachen Operationen request/release auf einem Ressourcen-Pool
- Achtung!
Semaphore nicht mit Ereignissen verwechseln!
- Weitere Terminologie – entsprechend dem aktuellen Verhalten:
 - Boolesches Semaphor:
 - hat Invariante $signals \leq 1$,
 - verhält sich wie Sperrvariable (3.1.5 ←)
 - privates Semaphor:
 - nur ein Prozess führt **P**-Operationen darauf aus
 - (Konsequenz: Frage nach Fairness ist irrelevant)

[7] © Peter Lühr, Robert Tolksdorf, Berlin

Verallgemeinerte Semaphore

- Verallgemeinerte Semaphor-Operationen:

```
void P(int n)
void V(int n)
```

- (vgl. request/release)

```
static void P(s1,s2,..)
static void V(s1,s2,..)
```

- (vgl. Mehrobjekt-Sperren (3.1.6 ←))

[8] © Peter Lühr, Robert Tolksdorf, Berlin

3.3.2 Sperren mit Semaphoren

- Ein Semaphor kann als Sperrvariable verwendet werden:
- der Effekt von
`x.lock(); ; x.unlock();`
- wird erzielt durch
`mutex.P(); ; mutex.V();`
- mit `Semaphore mutex = new Semaphore(1);`
- (= Boolesches Semaphor)

[9] © Peter Lühr, Robert Tolksdorf, Berlin

Sperren mit Semaphoren

- Ist der zu schützende Abschnitt nur „halbkritisch“, kann das Semaphor entsprechend initialisiert werden.
- Beispiel:
 - interaktives nichtsequentielles Programm schützt Benutzer vor Informationsüberflutung durch Begrenzung der Anzahl der zu einem Zeitpunkt auf dem Bildschirm erscheinenden Dialogfenster:
 - `Semaphore sema = new Semaphore(3);`

```
sema.P();
open dialogue;
display output;
accept input;
close dialogue;
sema.V();
```

[10] © Peter Lühr, Robert Tolksdorf, Berlin

Ressourcen-Verwaltung (3.2.2.3←)

```
class Resources<Resource> {
private final Stack<Resource> pool;
private final Semaphore available;
public Resources(Stack<Resource> pool){
    this.pool = pool;
    available = new Semaphore(pool.length());
}

public Resource request() {
    available.P();
    synchronized(pool){return pool.pop();}
}

public void release(Resource r) {
    synchronized(pool){pool.push(r);}
    available.V();
}
}
```

[11] © Peter Lühr, Robert Tolksdorf, Berlin

3.3.3 Puffer-Varianten

- Delegation an LinearQueue aus 3.2.3:

```
class Buffer {
private final Queue<Object> queue;
private final Semaphore frame;
private final Semaphore messg;

public Buffer(int size) {
    queue = new LinearQueue(size);
    frame = new Semaphore(size);
    messg = new Semaphore(0);
}
}
```

[12] © Peter Lühr, Robert Tolksdorf, Berlin

Buffer/2

```
public void send(Object m) {
    frame.P(); // außerhalb des kritischen Abschnitts !
    synchronized(queue){queue.append(m);}
    messg.V();
}
public Object rcv() {
    Object m;
    messg.P();
    synchronized(queue){m = queue.remove();}
    frame.V();
    return m;
}
public int length() {
    synchronized(queue){return queue.length();}
}
```

[13] © Peter Lehr, Robert Tolksdorf, Berlin

Spezialfall: Puffer mit einem Platz:

```
class Box {
    protected double box;
    private final Semaphore empty = new Semaphore(1);
    private final Semaphore full = new Semaphore(0);

    public void send(double m) {
        empty.P();
        box = m; // kein Ausschluß erforderlich !
        full.V();
    }
    public double rcv() {
        full.P();
        double m = box;
        empty.V();
        return m;
    }
}
```

[14] © Peter Lehr, Robert Tolksdorf, Berlin

3.3.4 Äquivalenz von Semaphoren und Monitoren

- Semaphore können mit Monitoren implementiert werden (3.3.1 ←)
- Die Umkehrung gilt ebenfalls – siehe unten !
- Somit sind Semaphore und Monitore hinsichtlich ihrer Synchronisationsfähigkeiten **gleich mächtig**.

[15] © Peter Lehr, Robert Tolksdorf, Berlin

Beispiel – Monitor mit signal-and-return

```
class Monitor { // "Sema" means "Semaphore"
    protected final Sema mutex = new Sema(1);
    protected class Event {
        private int waiting = 0;
        private Sema ready = new Sema(0);
        public void wait() {
            waiting++;
            mutex.V();
            ready.P();
            waiting--; }
        public void signal() {
            if(waiting != 0) ready.V();
            else mutex.V();
            break Monitor; } //almost Java!
    }
}
```

- Für andere Monitor-Varianten ähnlich

[16] © Peter Lehr, Robert Tolksdorf, Berlin

Beispiel/2

- Damit z.B.

```
class X extends Monitor {
    private final Event e = new Event();
    ...
    ...
    public void op1() { // if result, modify ...
        mutex.P();
        ... e.wait(); ...
        mutex.V();
    }
    public void op2() { // no result possible
        mutex.P();
        ... e.signal();
        mutex.V(); // disposable
    }
}
```

[17] © Peter Löhr, Robert Tolksdorf, Berlin

3.4 Nebenläufigkeit und Objektorientierung

- Hinterfragen: wie verhalten sich die Konzepte **Prozesse** und **Synchronisation** zu den Konzepten **Klassen** und **Vererbung** ?
- Übersichtsartikel *Briot/Guerraoui/Löhr 1998* (ACM Computing Surveys, Sept. 1998)
- Als Technischer Bericht: *Briot/Guerraoui/Löhr 1997*
- <ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-97-07.ps.gz>

[18] © Peter Löhr, Robert Tolksdorf, Berlin

3.4.1 Vererbungsanomalien

- (*inheritance anomalies*)
- Eine **Vererbungsanomalie** liegt vor, wenn
 - in einer Unterklasse einer synchronisierten Klasse *allein aus Synchronisationsgründen* das Umdefinieren (*overriding*) einer ererbten Operation erforderlich wird.

[19] © Peter Löhr, Robert Tolksdorf, Berlin

Beispiel 1

```
class Resources<R> {
    .....
    protected int available = ...;
    public synchronized R request() {
        while(available == 0) wait();
        available--;
        ..... // get resource
    }
    public synchronized void release(R r) {
        ..... // put resource
        available++;
        notify(); // NOT notifyAll !
    }
}
```

[20] © Peter Löhr, Robert Tolksdorf, Berlin

Beispiel 1

```
class Resources2 extends Resources {  
  
    public synchronized Pair<R> request2() {  
        while(available < 2) wait();  
        available -= 2;  
        ..... // get 2 resources  
    }  
  
    public synchronized void release(R r) {  
        super.release(r);  
        notifyAll();  
    }  
}
```

- Mögliches Szenario *ohne* undefiniertes **release**:
 1. Prozesse warten bei `request` und bei `request2`
 2. `release` weckt Prozess in `request2`

[21] © Peter Lehr, Robert Tolksdorf, Berlin

2 Alternativen zur Abhilfe

1. Bessere Programmiersprache:

```
MONITOR Resources<R> {  
    .....  
    public R request()  
        WHEN available > 0 {.....}  
  
    public void release(R r) {.....}  
    }  
  
MONITOR Resources2 extends Resources {  
    public Pair<R> request()  
        WHEN available > 1 {.....}  
    }  
}
```

[22] © Peter Lehr, Robert Tolksdorf, Berlin

2 Alternativen zur Abhilfe

1. Sorgfältige Planung bei der Konstruktion der Oberklasse – hier:

```
class Resources<R> {  
    .....  
    public synchronized void release(R r) {  
        ..... // put resource  
        available++;  
        notifyAll();  
    }  
}
```

- Weniger effizient, vermeidet aber die beschriebene Anomalie

[23] © Peter Lehr, Robert Tolksdorf, Berlin

Beispiel 2

- *Tabelle* mit Materialkonstanten, Atomgewichten o.ä.

```
class KeysToDoubles {  
    protected final int max = ...; // fixed capacity  
    protected final Vector keys = new Vector(max);  
    protected final double[] nums = new double[max];  
  
    public synchronized  
        void enter(Object key, double num) {  
            nums[keys.size()] = num;  
            keys.addElement(key);  
        }  
    public double lookup(Object key) { // NOT sync. !  
        return nums[keys.indexOf(key)];  
    }  
}
```

- (Der Einfachheit halber werden hier alle möglichen Ausnahmen ignoriert.)

[24] © Peter Lehr, Robert Tolksdorf, Berlin

Beispiel 2

```
class UpdatableDoubles extends KeysToDoubles {
```

```
    public synchronized
```

```
        void update(Object key, double num) {  
            nums[keys.indexOf(key)] = num;  
        }
```

```
    public synchronized double lookup(Object key) {
```

```
        return super.lookup(key);  
    }
```

- Mögliches Szenario *ohne* umdefiniertes **lookup** :
Überlappende Ausführung von **lookup** und **update** bewirkt,
daß ein **nums**-Element (das nicht unteilbar ist!) überlappend
gelesen und überschrieben werden kann.

[25] © Peter Lühr, Robert Tolksdorf, Berlin

Abhilfe?

1. Bessere Programmiersprache:
statt **synchronized** deklarative Angaben zur wechselseitigen
Verträglichkeit (*compatibility*) von Operationen, z.B.

```
void enter(Object key, double num) {  
    ...}
```

```
double lookup(Object key) COMP lookup, enter {  
    ...}
```

```
void update(Object key, double num) {  
    ...}
```

2. Sorgfältige Planung ??

[26] © Peter Lühr, Robert Tolksdorf, Berlin

Beispiel 3

- Beispiel 3: Schlange → Puffer → erweiterter Puffer

```
class LinearQueue { // 3.1.2 ←, 3.2.3 ←  
    protected int size, front, rear;
```

```
    ...  
    public void append(Object x) throws ... {...}  
    public Object remove()    throws ... {...}  
}
```

- Bei dieser Repräsentation der Schlange (3.1.2)
verursacht eine Überlappung von **append** mit **remove**
keine Probleme!
- Daher:

[27] © Peter Lühr, Robert Tolksdorf, Berlin

Beispiel 3

```
class Buffer extends LinearQueue { // sync.  
    protected final Sema frame = new Sema(size);  
    protected final Sema messg = new Sema(0);  
    protected final Object sending = new Object();  
    protected final Object receiving = new Object();  
    public void append(Object x) {  
        frame.P();  
        synchronized(sending) {super.append(x);}  
        messg.V(); }  
    public Object remove() {Object x;  
        messg.P();  
        synchronized(receiving) {x = super.remove();}  
        frame.V();  
        return x; }  
}
```

[28] © Peter Lühr, Robert Tolksdorf, Berlin

Beispiel 3

```
class ClearableBuffer extends Buffer {  
public void clear() {
```

```
    ???  
    front = 0;  
    rear = 0;  
    ???  
}
```

Nicht implementierbar ohne
völliges Umschreiben von
append/remove !

- Abhilfe ?
 1. Mit **WHEN** und **COMP** !
 2. Sorgfältige Planung ??

[29] © Peter Lehr, Robert Tolksdorf, Berlin

3.4.2 Aktive Objekte (active objects)

- Zur Erinnerung:
 - 2.1.4: Prozess \approx Objekt/Klasse
 - Prozessaktivität = ausgezeichneter Block
- 2.2.1: Java:
 - Prozess („Thread“) = Unterklasse von **Thread**
 - Prozessaktivität = Rumpf von **run**

[30] © Peter Lehr, Robert Tolksdorf, Berlin

3.4.2.1 Autonome Operationen

- Wünschenswert:
Klasse kann **autonome Operationen** enthalten
- Syntax: (nicht Java!)
 - AutonomousMethodDecl = **AUTO** Identifier Block
- Eine solche Operation wird nach abgeschlossener Initialisierung *ohne expliziten Aufruf* automatisch gestartet und nach Beendigung automatisch *erneut* gestartet.
- Es gibt hier *weder* Modifizierer *noch* Argumente *noch* Ergebnis!

[31] © Peter Lehr, Robert Tolksdorf, Berlin

Beispiel

- (kommt ohne Synchronisation aus!):

```
class Moving { // in plane; Vector is 2-dim.  
public Moving(Vector start, double timeUnit) {  
    pos = start.clone(); // position  
    vel = new Vector(0,0); // velocity  
    time = timeUnit; } // time granule  
protected Vector pos;  
protected volatile Vector vel;  
protected final double time;  
  
    AUTO step {  
        pos = new Vector(pos.x + vel.x * time, pos.y + vel.y * time); }  
  
public void setVelocity(Vector v) {  
    vel = v.clone();  
}
```

[32] © Peter Lehr, Robert Tolksdorf, Berlin

Beispiel/2

- Erweiterung z.B. so:

```
class MovingAndBeeping extends Moving {  
public MovingAndBeeping(Vector s, double t,  
                        Speaker sound) {
```

```
    super(s,t);  
    this.sound = sound; }
```

```
protected volatile boolean beepOn;  
protected final Speaker sound;
```

```
AUTO beep WHEN beepon { sound.beep(); }
```

```
public void on(boolean b) { beepOn = b; }  
}
```

- (wiederum keine Synchronisation erforderlich)

[33] © Peter Lehr, Robert Tolksdorf, Berlin

Beispiel/3

```
class Moving { // ... in Java – aber „Frühstart“-Gefahr (2.2.1)
```

```
public Moving(Vector start, double timeUnit) {  
    pos = start.clone(); // position  
    vel = new Vector(0,0); // velocity  
    time = timeUnit; // time granule  
    new Thread(new Runnable() {  
        public void run() { while(true) step();}).start();}
```

```
protected Vector pos;  
protected volatile Vector vel;  
protected final double time;
```

```
private void step() {  
    pos = new Vector(pos.x + vel.x * time, pos.y + vel.y * time);  
}
```

```
public void setVelocity(Vector v) {  
    vel = v.clone(); }  
}
```

[34] © Peter Lehr, Robert Tolksdorf, Berlin

3.4.2.2 Asynchrone Operationen

- Zur Erinnerung (2.1.2):
- Wenn eine ergebnislose Operation nach Aufruf und erfolgter Parameter-Übergabe **asynchron** ausgeführt werden soll (d.h. nebenläufig zum Aufrufer), kann das mit der Gabelungsanweisung

```
FORK Operation(Parameters)
```

erreicht werden.

- FORK** gibt *dem Aufrufer* die Möglichkeit, *zwischen Synchronie und Asynchronie zu wählen*.

[35] © Peter Lehr, Robert Tolksdorf, Berlin

Asynchrone Operation

- Alternative:
Asynchrone Operation (nicht Java!), d.h. Asynchronie ist Eigenschaft der Operation (und Gabelungsanweisung entfällt)
- Syntax:
 - AsynchronousMethodDecl = **ASYNC** MethodDecl
- ASYNC** ist ein Modifizierer (*modifier*).
- Vorteile:
 - Semantik!
Es ist i.a. *nicht gleichgültig*, ob eine Operation synchron oder asynchron abläuft.
 - Effizienz:
Übersetzer kann Threading optimieren.

[36] © Peter Lehr, Robert Tolksdorf, Berlin

Beispiel

- Beispiel:

```
class Printer {
    ...
    public ASYNC void print(File f) {
        ..... // do print f
    }
    public Status check() {
        .....
    }
}
```

- Benutzung:

```
... s = printer.check(); printer.print(f); ...
```

[37] © Peter Lehr, Robert Tolksdorf, Berlin

Beispiel/2

- In Java:

```
class Printer {
    ...
    public void print(final File f) { // async.
        new Thread(new Runnable(){
            public void run(){
                .../* do print f */...}})
            .start();
    }
    public Status check() {
        .....
    }
}
```

[38] © Peter Lehr, Robert Tolksdorf, Berlin

3.4.2.3 Verzögerte Synchronisation

- Asynchrone Operationen *mit Ergebnis (nicht Java!)*:

```
class Printer {
    ...
    public ASYNC Result print(File f) {
        ..... // do print f
    }
}
```

Verweistyp, nicht primitiver Typ!

- Aufruf in

```
Result result = printer.print(File f);
```

liefert einen Ergebnisvertreter („future“) result

[39] © Peter Lehr, Robert Tolksdorf, Berlin

Verzögerte Synchronisation

- Implizite Synchronisation mit der Operations-Beendigung bei erstem Zugriff auf den Ergebnisvertreter – verzögerte Synchronisation (*lazy synchronization, wait-by-necessity*)

```
Result result = printer.print(f);
```

```
...
```

```
... // do other business
```

```
...
```

```
Status s = result.getStatus();
```

```
// await termination of print,
```

```
// then access print result
```

[40] © Peter Lehr, Robert Tolksdorf, Berlin

Verzögerte Synchronisation in Java?

- Nachbau in Java ?
 - → Übung
- Verhalten von

ASYNCR op(A a)

PRE ...

WHEN ...

{.....}

→ ehrgeizigere Übung

Zusammenfassung

Zusammenfassung

- Semaphore
 - P() / V()
 - Sperren mit Semaphoren
 - Puffer
 - Semaphore und Monitore gleich mächtig
- Nebenläufigkeit und Objektorientierung
 - Vererbungsanomalien
 - Abhilfen
 - Aktive Objekte
 - Autonome Operationen
 - Asynchrone Operationen
 - Verzögerte Synchronisation