

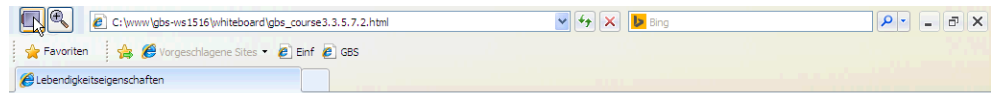
# Script generated by TTT

Title: Grundlagen\_Betriebssysteme (09.11.2015)

Date: Mon Nov 09 13:45:39 CET 2015

Duration: 86:59 min

Pages: 31



Man ist daran interessiert zu erkennen, ob es in einem System zu Blockierungen kommen kann, so dass Teile des Systems blockiert sind oder der gesamte Ablauf zum Stillstand kommt.

## Netzdarstellung

aktive Systemelemente als Transitionen (Prozessor, Maschine, etc.)

passive Systemteile als Stellen (Speicher, Lager, etc.)

veränderliche Objekte als Marken

Für Lebendigkeitsuntersuchungen sind Netzteile interessant, die niemals markiert werden oder die niemals ihre Marken verlieren.

## Definition

[Beispiel: Lebendiges Netz](#)

[Beispiel: Verklemmung](#)

Generated by Targeteam



## Beispiel: Lebendiges Netz

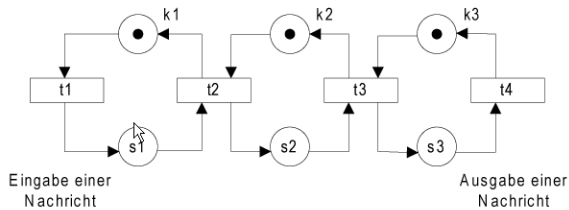


## Beispiel: Verklemmung



**Aufgabe:** Modellierung eines FIFO-Puffers mit Kapazität 3.

**Lösung:** Das System besteht aus 3 Zellen, die jeweils eine Nachricht aufnehmen können (die Stellen repräsentieren die Speicherzellen). Die Transition  $t_1$  modelliert das Eingeben einer neuen Nachricht und die Transition  $t_4$  modelliert die Ausgabe der Nachricht. Mit den Transitionen  $t_i$  werden die Nachrichten von Zelle  $s_{i-1}$  zur Zelle  $s_i$  weitergereicht. Voraussetzung ist, dass die entsprechende Zelle leer ist. Der Zustand "Zelle  $s_i$  ist leer", wird durch die markierte Stelle  $k_i$  modelliert.



Generated by Targeteam

2 Studenten benötigen ein 2-bändiges Lehrbuch. Student 1 leiht sich zunächst nur Band 1 aus und Student 2 leiht sich vorsorglich den noch vorhandenen Band 2 aus. Bevor Student 1 seinen ersten Band zurückgibt, möchte er noch den zweiten ausleihen. Auch Student 2 gibt seinen ausgeliehenen Band nicht zurück, sondern versucht, den ersten Band auszuleihen.

[Vor der Ausleihe](#)

[Nach der Ausleihe](#)

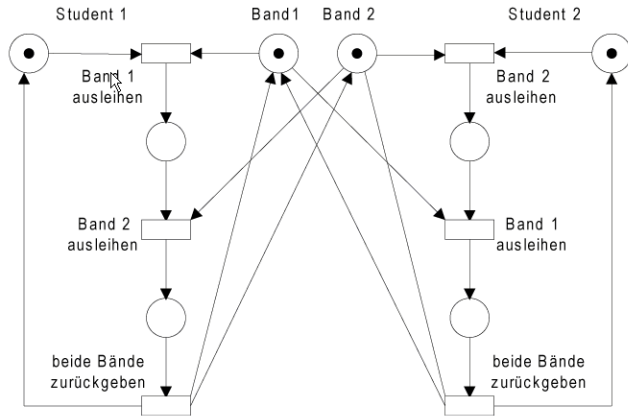
Generated by Targeteam



## Vor der Ausleihe



Anfangszustand des Netzes vor der Ausleihe.

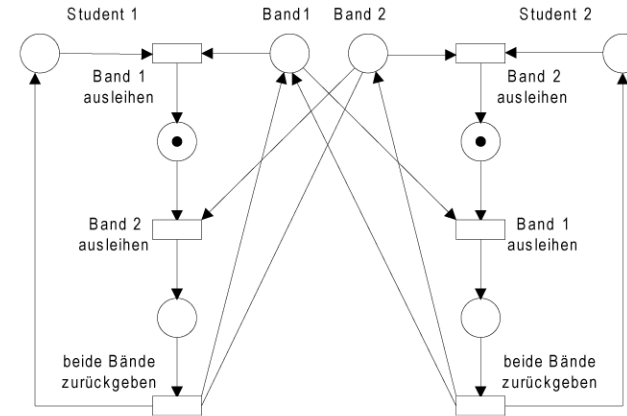


Generated by Targeseam

## Nach der Ausleihe



Zustand des Netzes nach dem Schalten Transitionen, d.h. nach der Ausleihe.



Generated by Targeseam



## Eigenschaften von Netzen



Ausgehend von einer Anfangsmarkierung können Eigenschaften wie Erreichbarkeit und Lebendigkeit eines Netzes bestimmt werden.

### Erreichbarkeit

### Lebendigkeitseigenschaften

Weitere Eigenschaften

Weitere interessante Eigenschaften - nur ganz informell - sind

### Fairness

Gegeben sei ein Netz  $N$  mit Anfangsmarkierung  $M$ . Das Netz ist **unfair** für eine Transition  $t$ , wenn es eine unendliche Sequenz gibt, in der  $t$  nur endlich oft auftritt, obwohl  $t$  unendlich oft transitionsbereit ist.

### Verhungern

$t$  verhungert (engl. Starvation): Es gibt eine unendliche Sequenz, in der die Transition  $t$  **niemals** auftritt.

Generated by Targeseam



## Petri-Netze



Im folgenden werden **Petri-Netze** vorgestellt, die eine graphen-orientierte Beschreibung verteilter Systeme und deren Abläufen ermöglicht.

### Allgemeines

### Definition: Petri-Netz

### Markierung und Schaltregeln

Zur Erfassung des dynamischen Verhaltens erweitern wir die Definition eines Petri-Netzes zunächst um Markierungen und geben dann die Schaltregeln an.

### Markierung

### Schaltregeln

### Animation Petrinetz

### Nebenläufigkeit

### Eigenschaften von Netzen

Generated by Targeseam



Aus BS-Sicht ist ein Prozess definiert durch einen Adressraum, eine darin gespeicherte Handlungsvorschrift in Form eines sequentiellen Programms (Programmcode), einen oder mehreren Aktivitätsträgern, die dynamisch die Handlungsvorschrift ausführen ⇒ Threads.

**Motivation**

Ein Thread ist die Abstraktion eines physischen Prozessors; er ist ein Träger einer sequentiellen Aktivität. Gründe für die Einführung von Threads:

- mehrere Threads ermöglichen Parallelität innerhalb eines Prozesses unter Nutzung des gemeinsamen Adressraums.
- Aufwand für Erzeugen und Löschen von Threads ist geringer als für Prozesse.
- Verbesserung der Performanz der Applikationsausführung durch Nutzung mehrerer Threads.
- Threads ermöglichen bei einem Multiprozessor-Rechensystem echte Parallelität innerhalb einer Applikation.

Prozess vs. Thread

Beispiel: Web-Server

Generated by Targeteam



Prozesse und Threads haben unterschiedliche Zielsetzungen:

**Prozesse** gruppieren Ressourcen,

**Threads** sind Aktivitätsträger, die zur Ausführung einer CPU zugeteilt werden.

**Threadspezifische Information**

Jeder Thread umfasst eine Reihe von Informationen, die seinen aktuellen Zustand charakterisieren:

Befehlszähler, aktuelle Registerwerte, Keller, Ablaufzustand des Thread.

**Prozessspezifische Information**

Die nachfolgende Information/Ressourcen wird von allen Threads eines Prozesses geteilt. Jeder Thread des Prozesses kann sie verwenden bzw. darauf zugreifen.

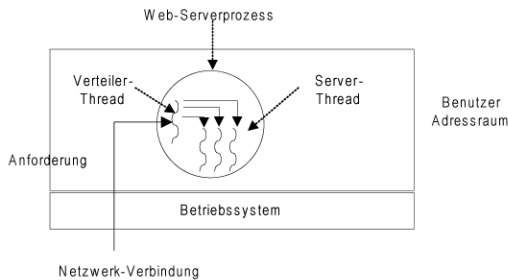
Adressraum, globale Variable, offene Dateien, Kindprozesse (z.B. erzeugt mit fork), eingetretene Alarmer bzw. Interrupts, Verwaltungsinformation



Generated by Targeteam



Ein Prozess kann mehrere Threads umfassen, die unterschiedliche Aufgaben übernehmen. Beispielsweise kann ein Web-Server einen Verteiler-Thread ("dispatcher") und mehrere Server-Threads ("worker-thread") umfassen.



Der Verteiler-Thread dient zur Annahme von Service-Anforderungen und gibt sie weiter an einen der Server-Threads zur Bearbeitung.  
 Alle Server-Threads nutzen den gemeinsamen Web-Cache.

Generated by Targeteam



Java unterstützt die Programmierung mit Threads, um nebenläufige Abläufe innerhalb einer Applikation zu ermöglichen. Java Threads können sowohl in Java-Programmen als auch in Java-Applets verwendet werden.

**Definition**

Threads können durch Implementierung der Schnittstelle **Runnable** realisiert werden.

Thread Implementierungen überschreiben die run-Methode der Schnittstelle Runnable.

```
public class CallbackDigest implements Runnable {
    ....
    public void run() { .... }
}
```

Thread Instanzen werden durch den Aufruf der start-Methode der Schnittstelle Runnable gestartet.

```
public static void main(String[] args) {
    ....
    CallbackDigest cb = new CallbackDigest(...);
    Thread t = new Thread(cb);
    t.start();
}
```

Die Anweisung `t.start()` ruft die run-Methode der Klasse CallbackDigest auf.

Ergebnisrückgabe

Generated by Targeteam



## Ergebnisrückgabe



Der Ablauf der Threads ist asynchron, d.h. die Ausführungsreihenfolge einer Menge von Threads ist nicht fest vordefiniert. Dies ist insbesondere bei kooperierenden Threads von Bedeutung, z.B. bei der Ergebnisrückgabe eines Thread an einen anderen Thread.

[Direkter Ansatz](#)

[Callback Ansatz](#)

Generated by Targeteam



## Direkter Ansatz



Angenommen jeder Thread liest eine Datei und erzeugt daraus die zugehörige Hash-Information (z.B. verwendet für verschlüsselte Datenübertragung).

```

public class ReturnDigest implements Runnable {
    private File input;
    private byte[] digest;
    public ReturnDigest(File input) { this.input = input; }
    public void run() {
        .....
        digest = .....
    }
    public byte[] getDigest() { return digest; }
}

public class ReturnDigestUserInterface {
    .....
    public static void main(String[] args) {
        .....
        for (int i = 0; i < args.length; i++) {
            File f = new File(args[i]);
            ReturnDigest dr = new ReturnDigest(f);
            Thread t = new Thread(dr); t.start();
            .....
            byte[] digest = dr.getDigest();
            ....
        }
    }
}

```



## Direkter Ansatz



```

.....
    digest = .....
}
public byte[] getDigest() { return digest; }
}

public class ReturnDigestUserInterface {
    .....
    public static void main(String[] args) {
        .....
        for (int i = 0; i < args.length; i++) {
            File f = new File(args[i]);
            ReturnDigest dr = new ReturnDigest(f);
            Thread t = new Thread(dr); t.start();
            .....
            byte[] digest = dr.getDigest();
            ....
        }
    }
}

```

*erzeugt Hashcode für Datei*

Die Ausführung führt zu dem Fehler

Exception in thread "main" java.lang.NullPointerException at ReturnDigestUserInterface.main.

Die Variable digest wird vor der Thread-Initialisierung benutzt.

Generated by Targeteam



## Callback Ansatz



Nicht das main-Programm holt die Ergebnisse ab, sondern die aufgerufenen Threads rufen jeweils eine Methode des main-Programms auf, um die Ergebnisse zu übergeben => **Callback**.

```

public class CallbackDigest implements Runnable {
    private File input;
    public CallbackDigest(File input) { this.input = input; }
    public void run() {
        .....
        byte[] digest = .....;
        CallbackDigestUserInterface.receiveDigest(digest);
    }
    public byte[] getDigest() { return digest; }
}

public class CallbackDigestUserInterface {
    .....
    public static void receiveDigest(byte[] digest) {
        .....
    }
}

public static void main(String[] args) {
    .....
    for (int i = 0; i < args.length; i++) {
        File f = new File(args[i]);
        CallbackDigest dr = new CallbackDigest(f);
        Thread t = new Thread(dr); t.start();
    }
}

```



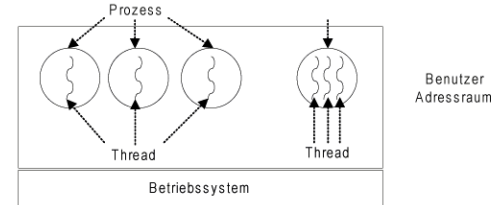
```

public class CallbackDigest implements Runnable {
    private File input;
    public CallbackDigest(File input) { this.input = input; }
    public void run() {
        .....
        byte[] digest = ..... erzeugt Hashcode
        CallbackDigestUserInterface.receiveDigest(digest);
    }
    public byte[] getDigest() { return digest; }
}

public class CallbackDigestUserInterface {
    .....
    public static void receiveDigest(byte[] digest) {
        .....
    }
    public static void main(String[] args) {
        .....
        for (int i = 0; i < args.length; i++) {
            File f = new File(args[i]);
            CallbackDigest dr = new CallbackDigest(f);
            Thread t = new Thread(dr); t.start();
        }
    }
}

```

Threads sind ein BS-Konzept für die Modellierung und Realisierung von nebenläufigen Aktionen in einem Rechensystem.

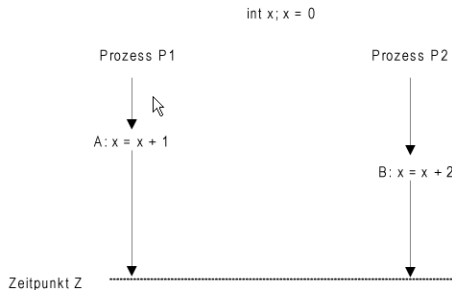


[Charakterisierung von Threads](#)  
[Threads in Java](#)

Generated by Targeteam



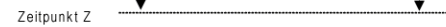
P1 und P2 sind nebenläufige Prozesse, die in einem Multiprozessorsystem parallel ablaufen. Die Variable x ist gemeinsame Variable. Prozess P1 läuft auf CPU 0 mit Prozess P2 auf CPU 1 gleichzeitig ab.



Das Ergebnis ist vom zeitlichen Ablauf abhängig. Es sind folgende Fälle möglich:

- Fall 1**  
P1 liest x = 0, erhöht, speichert x = 1;  
P2 liest x = 1, erhöht, speichert x = 3; => Wert von x = 3
- Fall 2**  
P2 liest x = 0, erhöht, speichert x = 2;  
P1 liest x = 2, erhöht, speichert x = 3; => Wert von x = 3
- Fall 3**  
P1 und P2 lesen x = 0;

*(oder Thread)*



Das Ergebnis ist vom zeitlichen Ablauf abhängig. Es sind folgende Fälle möglich:

- Fall 1**  
P1 liest x = 0, erhöht, speichert x = 1;  
P2 liest x = 1, erhöht, speichert x = 3; => Wert von x = 3
- Fall 2**  
P2 liest x = 0, erhöht, speichert x = 2;  
P1 liest x = 2, erhöht, speichert x = 3; => Wert von x = 3
- Fall 3**  
P1 und P2 lesen x = 0;  
P1 erhöht, speichert x = 1;  
P2 erhöht, speichert x = 2; => Wert von x = 2
- Fall 4**  
P1 und P2 lesen x = 0;  
P2 erhöht, speichert x = 2;  
P1 erhöht, speichert x = 1; => Wert von x = 1

Verhinderung des Nichtdeterminismus nur dadurch möglich, dass man garantiert, dass die Veränderung von x in den beiden Prozessen unter wechselseitigem Ausschluss (engl. mutual exclusion) erfolgt



## Beispiel: gemeinsame Daten



## Synchronisation



Zeitpunkt Z

(oder Thread)

Wert von x

Das Ergebnis ist vom zeitlichen Ablauf abhängig. Es sind folgende Fälle möglich:

### Fall 1

P1 liest x = 0, erhöht, speichert x = 1;  
P2 liest x = 1, erhöht, speichert x = 3; => Wert von x = 3

### Fall 2

P2 liest x = 0, erhöht, speichert x = 2;  
P1 liest x = 2, erhöht, speichert x = 3; => Wert von x = 3

### Fall 3

P1 und P2 lesen x = 0;  
P1 erhöht, speichert x = 1;  
P2 erhöht, speichert x = 2; => Wert von x = 2

### Fall 4

P1 und P2 lesen x = 0;  
P2 erhöht, speichert x = 2;  
P1 erhöht, speichert x = 1; => Wert von x = 1

Verhinderung des Nichtdeterminismus nur dadurch möglich, dass man garantiert, dass die Veränderung von x in den beiden Prozessen unter wechselseitigem Ausschluss (engl. mutual exclusion) erfolgt.

Generated by Targem

Eine wichtige Systemeigenschaft betrifft die Synchronisation paralleler Ereignisse. Mehrere Prozesse konkurrieren um eine gemeinsame Ressource (CPU), oder greifen auf gemeinsame Daten zu.

### Beispiele

Die beiden Beispiele basieren auf der speicherbasierten Prozessinteraktion, d.h. Prozesse (oder auch Threads) interagieren über gemeinsam zugreifbare Speicherzellen.

#### Beispiel: gemeinsame Daten

#### Erzeuger-Verbraucher-Problem

#### Definition: Wechselseitiger Ausschluss

#### Modellierung

#### Synchronisierungskonzepte

#### Semaphore

#### Synchronisierung von Java Threads

Generated by Targem



## Definition: Wechselseitiger Ausschluss



## Modellierung



Gegeben sei ein Petri-Netz (S, T, F) und eine Anfangsmarkierung M0. Wenn zwei Transitionen  $t_1, t_2 \in T$  **wechselseitig ausgeschlossen** sind, dann ist keine Markierung M' erreichbar, so dass  $t_1$  und  $t_2$  unter dieser Markierung gleichzeitig transaktionsbereit sind. Wir sagen, dass Transitionen, die wechselseitig ausgeschlossen auszuführen sind, **kritische Abschnitte** (engl. critical section, critical region) eines Systemablaufs modellieren.

### Beispiel: gemeinsame Daten

Der Zugriff auf gemeinsame Ressourcen, z.B. auf gemeinsame Variable, muss koordiniert werden. Bei exklusiven Ressourcen wird die Nutzung sequenzialisiert.

```

Prozess P1:      Prozess P2:
main () {      main () {
.....          .....
region x do    region x do
  x = x + 1;    x = x + 2;
end region     end region
.....         .....
}              }

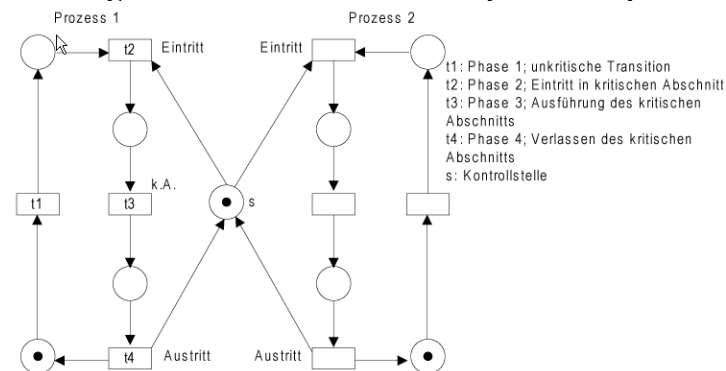
```

Generated by Targem

Modelliert man parallele Einheiten, die kritische Abschnitte besitzen, durch Petri-Netze, so sind vier Phasen dieser parallelen Aktivitäten von Interesse:

1. Ausführen unkritischer Abschnitte/Transaktionen
2. Betreten eines kritischen Abschnitts
3. Ausführen der Transaktion(en) des kritischen Abschnitts
4. Verlassen des kritischen Abschnitts.

Modellierung jeder Phase durch eine Transition; Koordinierung des wechselseitigen Ausschluss durch Kontrollstelle s.



### Beispiel: Leser-Schreiber-Problem

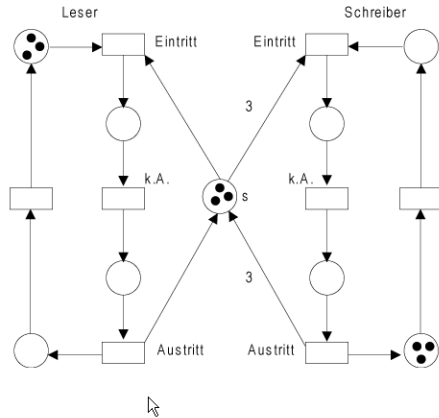
Generated by Targem



Das System umfasst **Leser-Aktivitäten**, die in ihrem kritischen Abschnitt lesend auf eine gemeinsame Ressource zugreifen und **Schreib-Aktivitäten**, die in ihrem kritischen Abschnitt schreibend auf die gemeinsame Ressource zugreifen. Wir fordern:

1. Leser-Aktionen im kritischen Abschnitt können parallel stattfinden, wobei die Anzahl der parallelen Leser begrenzt sei, z.B. auf drei.
2. Leser- und Schreib-Aktionen sind wechselseitig ausgeschlossen.
3. auch Schreib-Aktionen sind untereinander wechselseitig ausgeschlossen.

Die Abbildung zeigt eine mögliche Modellierung mittels eines Petri-Netzes mit drei Lesern und drei Schreibern.



Generated by Targeseam



Eine wichtige Systemeigenschaft betrifft die Synchronisation paralleler Ereignisse. Mehrere Prozesse konkurrieren um eine gemeinsame Ressource (CPU), oder greifen auf gemeinsame Daten zu.

**Beispiele**

Die beiden Beispiele basieren auf der speicherbasierten Prozessinteraktion, d.h. Prozesse (oder auch Threads) interagieren über gemeinsam zugreifbare Speicherzellen.

**Beispiel: gemeinsame Daten**

**Erzeuger-Verbraucher-Problem**

**Definition: Wechselseitiger Ausschluss**

**Modellierung**

**Synchronisierungskonzepte**

**Semaphore**

**Synchronisierung von Java Threads**

Generated by Targeseam



**Ziel:** Einführung wichtiger Realisierungskonzepte zur Synchronisation paralleler Abläufe. Dazu Konkretisierung des Prozessbegriffs.

**Prozess - Konkretisierung**

**Konzepte für wechselseitigen Ausschluss**

Die Netz-Modellierung hat bereits gezeigt, dass man zur Synchronisation von Prozessen spezifische **Kontrollkomponenten** benötigt (z.B. zusätzliche Stellen im Petri-Netz, oder Kapazitätsbeschränkungen, die implizit durch die abstrakte Kontrollkomponente, die die Transitionsbereitschaft von Transitionen prüft, kontrolliert werden).

**Anforderungen**

Jede Realisierung des w.A. benötigt eine Basis, mit festgelegten **atomaren**, d.h. nicht teilbaren Operationen.

**Unterbrechungssperre**

**Test-and-Set Operationen**

**Dienste mit passivem Warten**

**Semaphor-Konzept**

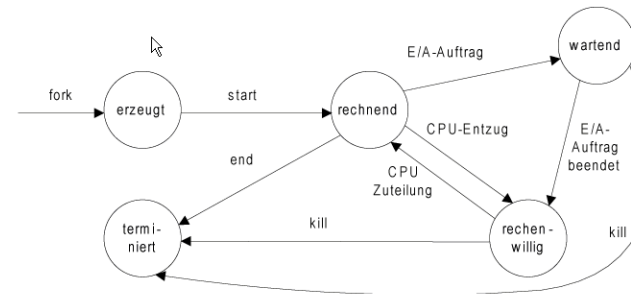
Das Semaphor-Konzept ermöglicht die Realisierung des w.A. auf einem höheren Abstraktionslevel als die bereits angesprochenen Hardwareoperationen.

**Monitor-Konzept**

Generated by Targeseam



Prozess = Ablauf eines Programms in einem Rechensystem; unterschiedliche Zustände eines Prozesses.



Generated by Targeseam



**Ziel:** Einführung wichtiger Realisierungskonzepte zur Synchronisierung paralleler Abläufe. Dazu Konkretisierung des Prozessbegriffs.

### Prozess - Konkretisierung

#### Konzepte für wechselseitigen Ausschluss

Die Netz-Modellierung hat bereits gezeigt, dass man zur Synchronisation von Prozessen spezifische **Kontrollkomponenten** benötigt (z.B. zusätzliche Stellen im Petri-Netz, oder Kapazitätsbeschränkungen, die implizit durch die abstrakte Kontrollkomponente, die die Transitionsbereitschaft von Transitionen prüft, kontrolliert werden).

#### Anforderungen

Jede Realisierung des w.A. benötigt eine Basis, mit festgelegten **atomaren**, d.h. nicht teilbaren Operationen.

#### Unterbrechungssperre

#### Test-and-Set Operationen

#### Dienste mit passivem Warten

#### Semaphor-Konzept

Das Semaphor-Konzept ermöglicht die Realisierung des w.A. auf einem höheren Abstraktionslevel als die bereits angesprochenen Hardwareoperationen.

#### Monitor-Konzept

Generated by Targeteam

Folgende Anforderungen sind an eine Realisierung des wechselseitigen Ausschlusses (**w.A.**) zu stellen:

Die kritischen Abschnitte der Prozesse sind wechselseitig auszuschließen.

Eine Realisierung des w.A. darf nicht von einer Annahme über die Reihenfolge, in der die Prozesse ihre kritischen Abschnitte ausführen, abhängen.

Eine Realisierung des w.A. darf nicht von Annahmen über die Ausführungszeit der Prozesse abhängen.

Unter der Annahme, dass Prozesse nach endlicher Zeit ihre kritischen Abschnitte wieder verlassen, muss gelten, dass kein Prozess einen anderen Prozess unendlich lange daran hindern darf, seinen kritischen Abschnitt auszuführen.

Generated by Targeteam



**Ziel:** Einführung wichtiger Realisierungskonzepte zur Synchronisierung paralleler Abläufe. Dazu Konkretisierung des Prozessbegriffs.

### Prozess - Konkretisierung

#### Konzepte für wechselseitigen Ausschluss

Die Netz-Modellierung hat bereits gezeigt, dass man zur Synchronisation von Prozessen spezifische **Kontrollkomponenten** benötigt (z.B. zusätzliche Stellen im Petri-Netz, oder Kapazitätsbeschränkungen, die implizit durch die abstrakte Kontrollkomponente, die die Transitionsbereitschaft von Transitionen prüft, kontrolliert werden).

#### Anforderungen

Jede Realisierung des w.A. benötigt eine Basis, mit festgelegten **atomaren**, d.h. nicht teilbaren Operationen.

#### Unterbrechungssperre

#### Test-and-Set Operationen

#### Dienste mit passivem Warten

#### Semaphor-Konzept

Das Semaphor-Konzept ermöglicht die Realisierung des w.A. auf einem höheren Abstraktionslevel als die bereits angesprochenen Hardwareoperationen.

#### Monitor-Konzept

Generated by Targeteam

Der Ablauf des Prozesses darf nicht unterbrochen werden.

In Ein-Prozessorsystemen (1 CPU) kann es ausreichend sein, mit **Enable** und **Disable Interrupt** Operationen dafür zu sorgen, dass der Prozess, der einen kritischen Abschnitt ausführt, dabei nicht unterbrochen wird.

Realisierung mit Unterbrechungssperre ist nützlich für den Betriebssystemkern, aber sollte nicht für allgemeine Benutzerprogramme zur Verfügung stehen.

Generated by Targeteam

