FH Bielefeld
University of
Applied Sciences

Vorlesung

# Betriebssysteme

## Teil 5

## Interprozesskommunikation und Synchronisation

Dozent

**Prof. Dr.-Ing.**

**Martin Hoffmann**

martin.hoffmann@fh-bielefeld.de

FH Bielefeld
University of
Applied Sciences

# Inhalt

- Interprozesskommunikation
  - Pipes, IPC, Shared Memory
- Mutexe und Semaphore
  - Schutz von kritischen Abschnitten
- Verklemmungen
  - Modellieren
  - Erkennen und Beheben
  - Verhindern (Banker Algorithm)

FH Bielefeld
University of
Applied Sciences

# IPC - Ziele

- Interprozesskommunikation kennenlernen
  - Beispiel Pipes
  - Shared memory
  - Sockets
- Race Condition erklären können
  - Mutex und Semaphore erklären und anwenden können

FH Bielefeld
University of
Applied Sciences

# Interprozesskommunikation

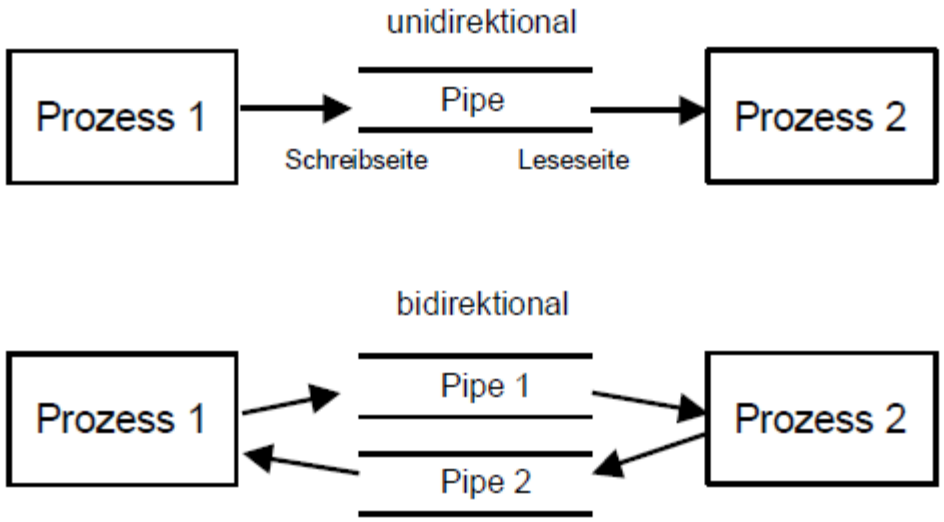
- Unter Unix (je nach Derivat) und Windows gibt es verschiedene IPC-Mechanismen:
  - **Pipes und FIFO‘s** (Named Pipes) als Nachrichtenkanal
  - **Nachrichtenwarteschlangen** (Message Queues)
  - **Gemeinsam genutzter Speicher** (Shared Memory)
  - **Sockets** mit IP-Loopback-Mechanismus

- Ggf. Synchronisationsmechanismen erforderlich:
  - Semaphore und Signale

FH Bielefeld
University of
Applied Sciences

# Pipes

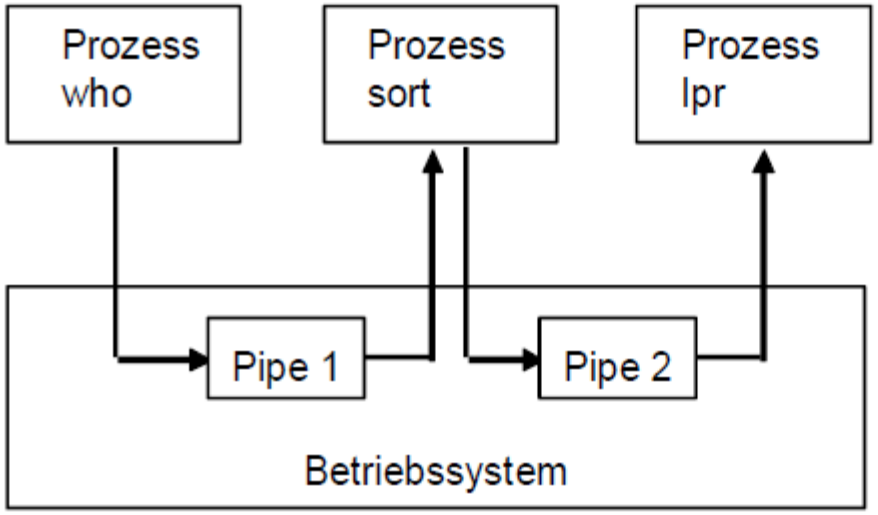
- Pipes: Spezieller unidirektionaler Mechanismus
- Unidirektionale und bidirektionale Kommunikation durch Nutzung mehrerer Pipes
- Bidirektionale Kommunikation über zwei Pipes kann sowohl halb- als auch vollduplex betrieben werden



FH Bielefeld
University of
Applied Sciences

# Pipes unter Linux

- Pipes werden u.a. genutzt, um die Standardausgabe eines Prozesses mit der Standardeingabe eines weiteren Prozesses zu verbinden
- Beispiel in Unix: who | sort | lpr




Unix-Kommandos:
who | sort | lpr

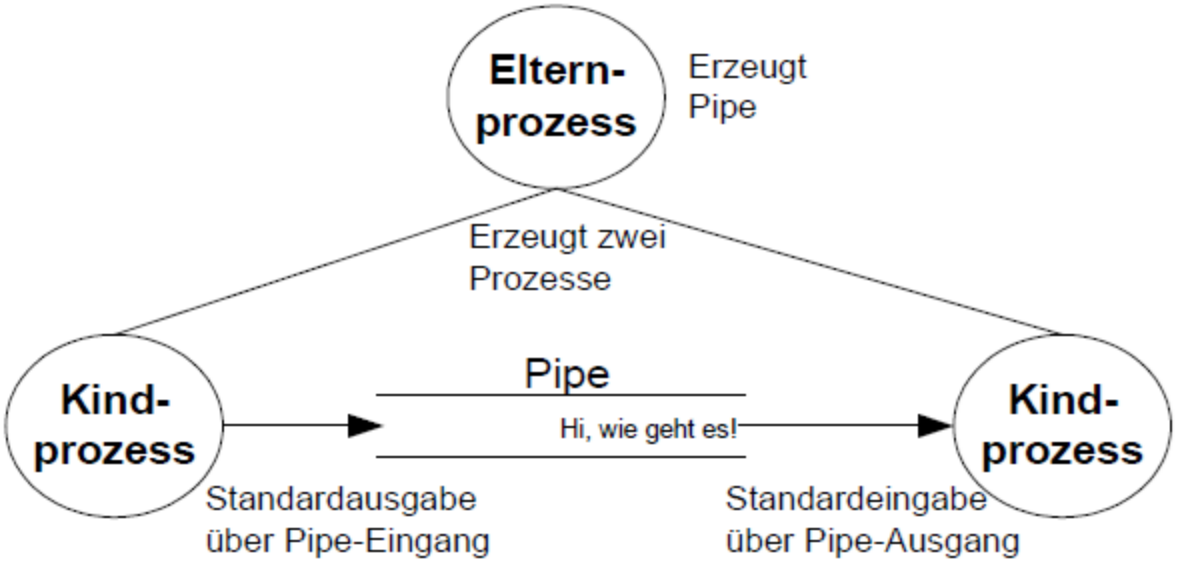
# Pipes: Programmierung

- Erzeugen einer Pipe:
  - Unter Unix mit dem Systemaufruf pipe() oder popen()
  - Unter Windows NT mit CreatePipe()
- Schließen einer Pipe:
  - Unter Unix mit dem Systemaufruf close() oder pclose()
  - Unter Windows NT mit closeHandle()
- Elternprozess erzeugt Pipe und vererbt sie an den Kindprozess
- Man kann Pipes blockierend (Normalmodus) und nicht blockierend einsetzen. Blockierend bedeutet:
  - Wenn die Pipe voll ist blockiert der Sendeprozess
  - Wenn die Pipe leer ist blockiert der Leseprozess
  - Sinnvoll für Erzeuger-Verbraucher-Problem

FH Bielefeld
University of
Applied Sciences

# Pipes: Beispiel (Teil 1)

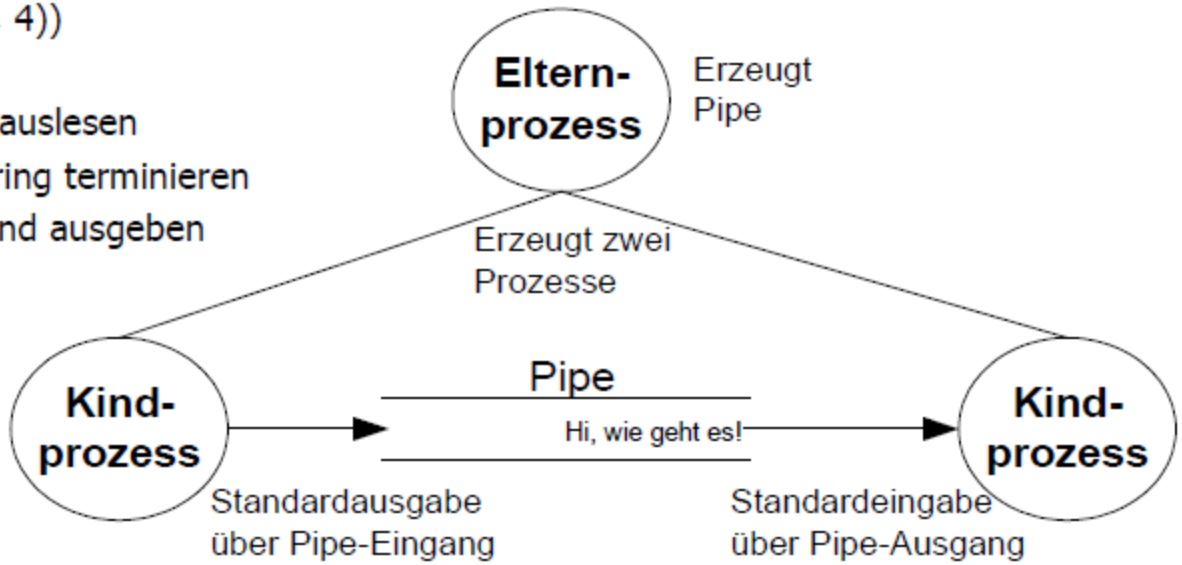
```
int fds[2] //Filedescriptoren für Pipe
...
pipe(fds);
if (fork() == 0) {
      // 1. Kindprozess, Standardausgabe auf Pipe-Schreibseite (Pipe-Eingang) legen
      // und Pipe-Leseseite (Pipe-Ausgang) schließen (wird nicht benötigt)
      dup2(fds[1], 1);  // 1 = Standardausgabe
      close(fds[0]);
      write (1, text, strlen(text)+1);
}
else{ ...
```



FH Bielefeld
University of
Applied Sciences

# Pipes: Beispiel (Teil 2)

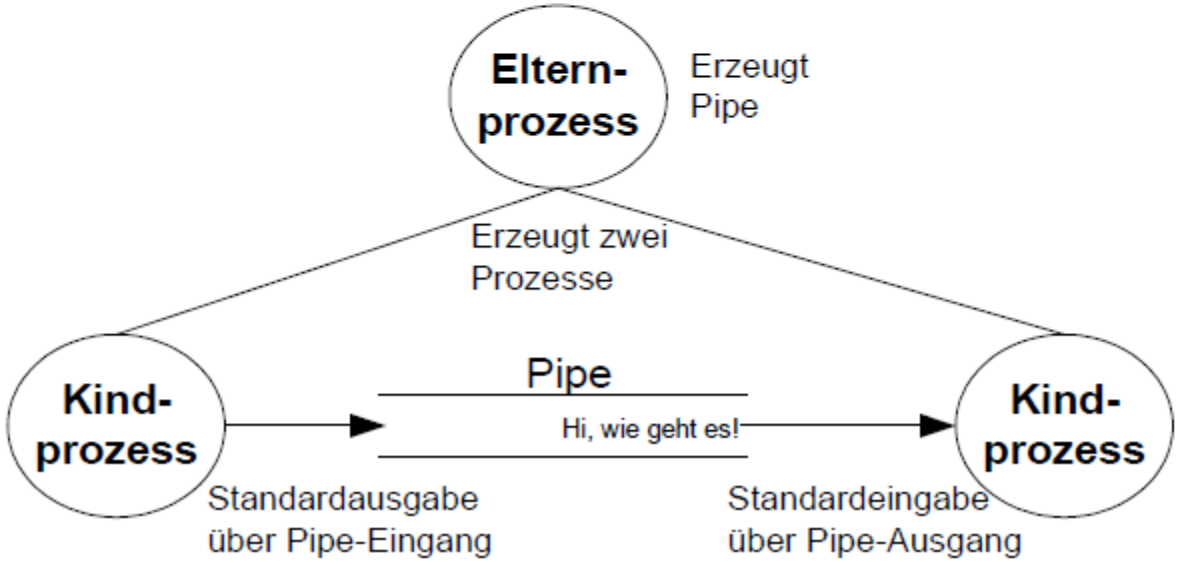
```
else{

   if (fork() == 0) {
        // 2. Kindprozess, Pipe-Leseseite (Pipe-Ausgang) auf
        // Standardeingabe umlenken und Pipe-Schreibseite
        // (Pipe-Eingang) schließen
        dup2(fds[0], 0); // 0 = standardeingabe
        close(fds[1]);
        while (count = read(0, buffer, 4))
        {
              // Pipe in einer Schleife auslesen
              buffer[count] = 0; // String terminieren
              printf("%s", buffer) // und ausgeben
        }
   }
```



FH Bielefeld
University of
Applied Sciences

# Pipes: Beispiel (Teil 3)

```
...
      else {
         // Im Vaterprozess: Pipe an beiden Seiten schließen und
         // auf das Beenden der Kindprozesse warten
         close(fds[0]);
         close[fds[1]);
         wait(&status);
         wait(&status);
      }
      exit(0);
   }
}
```



FH Bielefeld
University of
Applied Sciences

# Interprozesskommunikation

- *Parallele* Prozesse:
  - Rechner mit **mehreren CPUs** oder Netzwerke aus unabhängigen Rechnern (Multiprozessorsystemen) können mehrere Prozesse **zeitgleich** ausführen
  - Prozesse laufen **unabhängig** von einander (*parallel*)
- *Nebenläufige* Prozesse:
  - Rechner mit **einer CPU** können immer nur einen Prozess bearbeiten
  - Prozesse laufen hintereinander in **beliebiger Reihenfolge** (*nebenläufig*)
  - Parallelität (*Pseudoparallelität*) wird durch Multitasking realisiert

FH Bielefeld
University of
Applied Sciences

# Nebenläufige Prozesse

**Kommunikation** zwischen Prozessen:

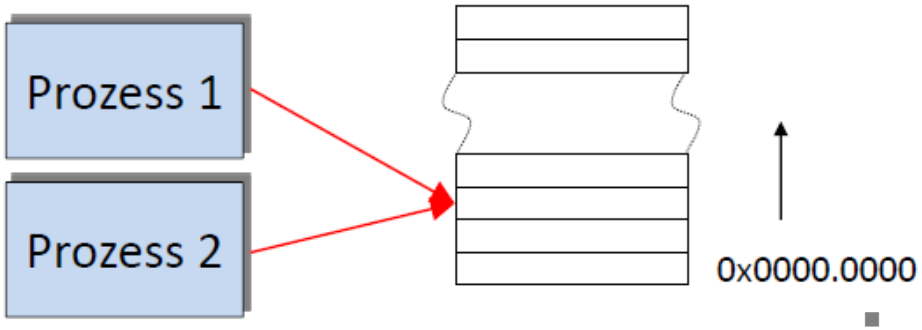
- Prozesse arbeiten oft zusammen, um ihre Aufgabe zu erfüllen.
- Dabei stellen sich folgende Fragen:
  - Wie findet der **Austausch** der Daten zwischen den Prozessen statt?
    - Über **gemeinsame Variablen** (gemeinsame Speicherbereiche)?
    - Über **Nachrichtenaustausch** (*Message Passing*)?
  - Wie wird die **Konsistenz** gemeinsam genutzter Daten sichergestellt?
  - Wie wird die richtige **Reihenfolge** beim Zugriff auf gemeinsame Daten sichergestellt?
- Die beiden letzten Fragen führen zum Problem der *Prozess**synchronisation***
  - ***Scheduling beeinflußt Abarbeitungsreihenfolge von Maschinenbefehlen***
  - ***Außerhalb der Kontrolle des Anwendungsentwicklers***

FH Bielefeld
University of
Applied Sciences

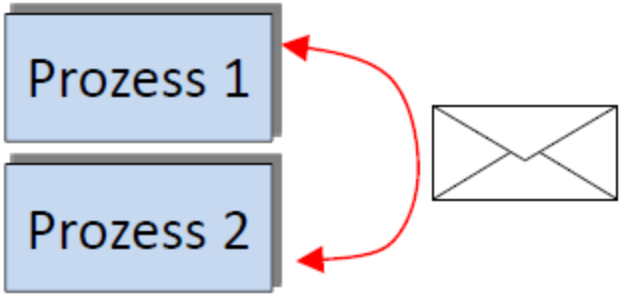
# Interprozesskommunikation

Kommunikationsformen:

- Gemeinsam benutzter **Speicher**:




- **Nachrichten**-basierte Kommunikation:




- Wird bei Ein- oder Multiprozessorsystemen mit gemeinsamen physikalischen Speicher eingesetzt

- Die Prozesse können auf *beliebigen* Maschinen laufen!
- Kommunikation kann auch *über Rechnergrenzen* hinweg erfolgen.

FH Bielefeld
University of
Applied Sciences

# Beispiele

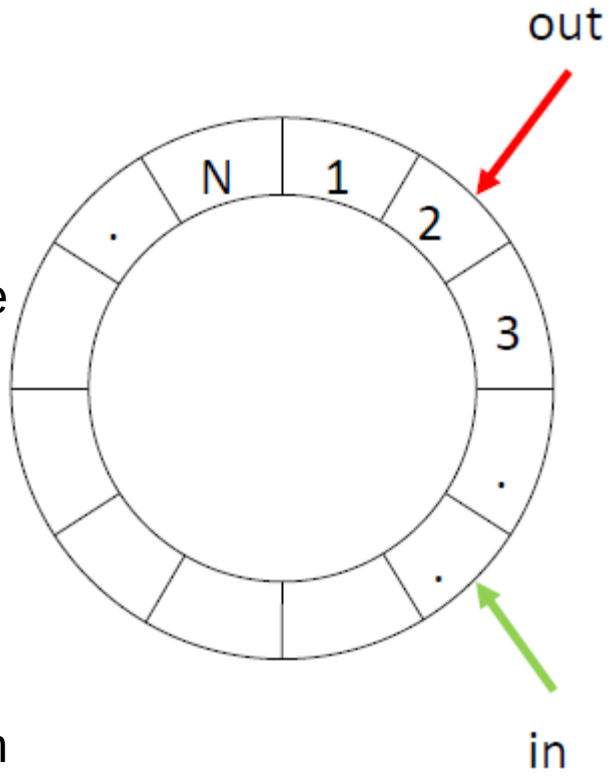
- Low-Level (POSIX)
  - shmget()
  - msgget()
- D-Bus (Desktop-Bus)
  - http://freedesktop.org/wiki/Software/dbus/
  - Linux Desktops GNOME/KDE
- Apache ActiveMQ
  - http://activemq.apache.org/
  - Verteilte Systeme

FH Bielefeld
University of
Applied Sciences

# Erzeuger-Verbraucher-Problem

Beispiel: *Erzeuger-Verbraucher Problem*

- **Szenario**: zwei Prozesse besitzen einen gemeinsamen Puffer mit fester, endlicher Länge.
  - Der **Erzeuger** (*producer*) Prozess schreibt Daten in den Puffer.
  - Der **Verbraucher** (*consumer*) Prozess holt die Daten aus dem Puffer.
- **Problem**:
  - Der Erzeuger darf **keine Daten in den vollen Puffer** schreiben,
  - der Verbraucher darf **keine Daten aus dem leeren Puffer** holen.
- Damit ist eine **Prozesssynchronisation** zwischen dem Erzeuger und dem Verbraucher notwendig.



FH Bielefeld
University of
Applied Sciences

# Erzeuger-Verbraucher-Problem: Lösung 1

Erster Versuch:

- Synchronisation über eine **globale** Variable **count**:

```
void producer()
{
  while (true)
  {
    produce(&item);
    while (count == N)
      doNothing();
    buffer[in] = item;
    in = (in + 1) % N;
    count = count + 1;
  }
}
```

```
void consumer()
{
  while (true)
  {
    while (count == 0)
      doNothing();
    item = buffer[out];
    out = (out + 1) % N;
    count = count - 1;
    consume(item);
  }
}
```

- Diese Lösung ist **fehlerhaft** und kann zu unvorhergesehenen Ergebnissen führen! Warum?

FH Bielefeld
University of
Applied Sciences

# Erzeuger-Verbraucher-Problem: Lösung 2

- Darstellung in Pseudo-Maschinencode:

```
; void producer()
; …
; count = count + 1;
P1: mov Register1, count
P2: inc Register1
P3: mov count, Register1
;…
```

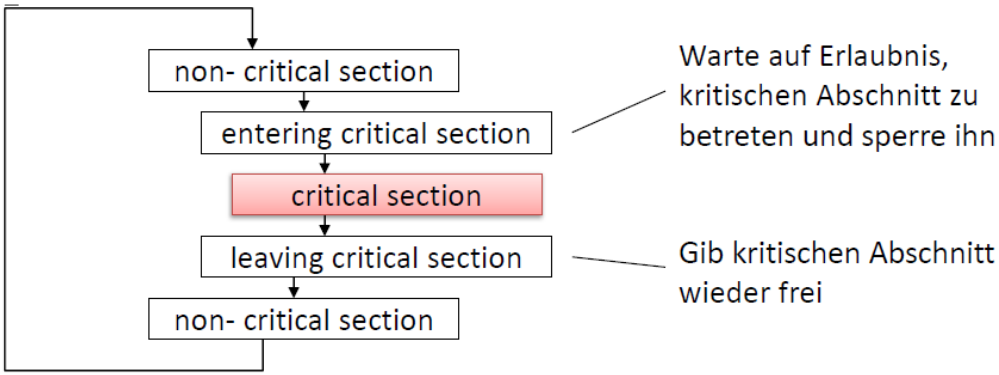
```
; void consumer()
; …
; count = count - 1;
C1: move Register2, count
C2: dec Register2
C3: move count, Register2
; …
```

- Annahme:
  - **count** sei 5 und die Maschinenbefehle werden in der **Reihenfolge** P1, P2, C1, C2, **C3, P3** abgearbeitet. Welchen Wert hat **count**?
  - **count** sei 5 und die Maschinenbefehle werden in der **Reihenfolge** P1, P2, C1, C2, **P3, C3** abgearbeitet. Welchen Wert hat **count**?
- Der Wert von **count** ist abhängig vom Augenblick der Prozessumschaltung. Es liegt eine sog. *race condition* vor.

FH Bielefeld
University of
Applied Sciences

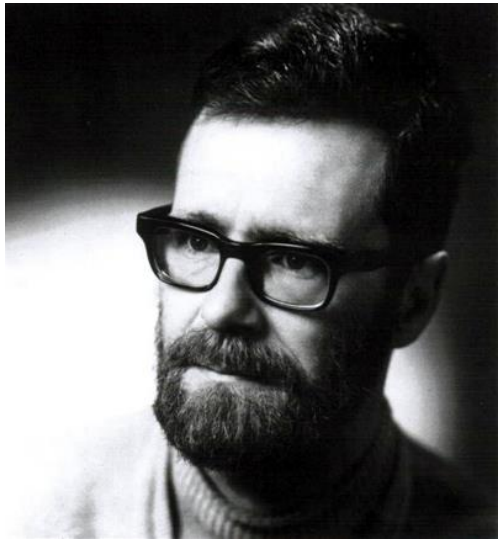
# Kritischer Abschnitt

- Ein kritischer Abschnitt (*critical section*) ist ein Codebereich, in dem das Ergebnis der Ausführung in Abhängigkeit von der **Ausführungsreihenfolge** der Prozesse oder Threads variieren kann, da der Zugriff auf globale Daten **nicht deterministisch** ist.
- Ein Prozess kann einen anderen „überholen“, es liegt eine ***race condition*** vor.
- race conditions zwischen zwei Prozessen sind im allgemeinen nur sehr **schwer zu finden**, da sie **nicht reproduzierbar** auftreten.
- Um race conditions zu verhindern, müssen kritische Abschnitte **gegen Mehrfachzugriff geschützt** werden.



# Dijkstra

- Dijkstra-Algorithmus zur Berechnung eines kürzesten Weges in einem Graphen
- erstmalige Einführung von Semaphoren zur Synchronisation zwischen Threads
- Philosophenproblem
- Bankieralgorithmus



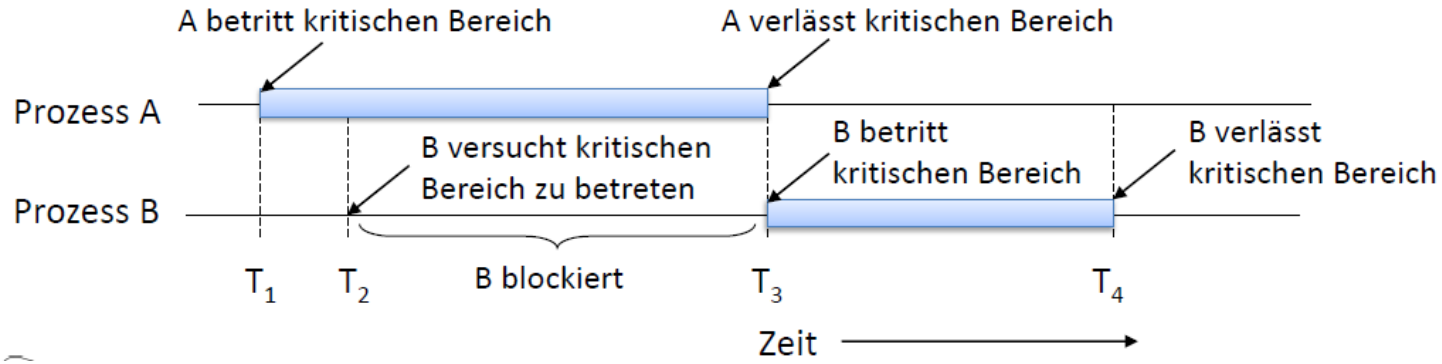
E.W. Dijkstra 1930-2004

FH Bielefeld
University of
Applied Sciences

# Kritischer Abschnitt: Vermeidung

An eine Lösung zur **Vermeidung von Race Conditions** werden **vier Bedingungen** gestellt (*E.W. Dijkstra, 1965*):

- 1. Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Abschnitten sein (gegenseitiger Ausschluss, *mutual exclusion*)
- 2. Es dürfen keine Annahmen über Abarbeitungsgeschwindigkeiten oder Anzahl der Prozesse bzw. Prozessoren gemacht werden.
- 3. Kein Prozess darf außerhalb eines kritischen Abschnitts einen anderen Prozess blockieren.
- 4. Kein Prozess sollte ewig darauf warten müssen, in seinen kritischen Abschnitt eintreten zu können (*Fairness*).



FH Bielefeld
University of
Applied Sciences

# Schutz von kritischen Abschnitten: Lösung 1, Interrupts sperren

- Erster Ansatz: **Sperren von Unterbrechungen** (Interrupts) in kritischen Abschnitten
- Durch das Sperren von Unterbrechungen können **keine Prozesswechsel** mehr stattfinden
- Diese Lösung hat folgende **Nachteile**:
  - Funktioniert nur in **Ein-Prozessor Systemen** (siehe Forderung 2).
  - Es ist nicht ratsam, Benutzerprozessen die Erlaubnis zu geben, Interrupts zu sperren. Ein fehlerhafter Prozess kann das gesamte **System lahm legen**.
- Deshalb findet diese Lösung nur
  - Anwendung *innerhalb* des Betriebssystems in Ein-Prozessor Lösungen oder
  - in Embedded Systemen mit klar definierten Benutzer-Prozessen.

FH Bielefeld
University of
Applied Sciences

# Schutz von kritischen Abschnitten: Lösung 2, „taking turns“

Zweiter Ansatz: über **Variablen sperren** (*Softwarelösung, Abb. Tanenbaum*):

```
while (TRUE) {
    while (turn != 0)        /* loop */;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

**Figure 2-23.** A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

- Verletzt Kriterium 3 „Kein Prozess darf außerhalb eines kritischen Abschnitts einen anderen Prozess blockieren.“

FH Bielefeld
University of
Applied Sciences

# Schutz von kritischen Abschnitten: Lösung 3, Peterson 1981

Dritter Ansatz: nach Peterson (1981) (*Softwarelösung*):

```
/* global data */
int turn;
boolean interest_1 = false;
boolean interest_2 = false;
```

```
void process_1()
{
  /* enter critical section 1 */
  interest_1 = true;
  turn = 2;
  while ((turn == 2)
    && interest_2 == true))
    doNothing();
  /* critical section 1 */
  interest_1 = false;
}
```

```
void process_2()
{
  /* enter critical section 2 */
  interest_2 = true;
  turn = 1;
  while ((turn == 1)
    && interest_1 == true))
    doNothing();
  /* critical section 2 */
  interest_2 = false;
}
```

FH Bielefeld
University of
Applied Sciences

# Schutz von kritischen Abschnitten: Lösung 4, TSL

Dritter Ansatz: TSL-Anweisung (Hardwarelösung):

- Die **TSL- Anweisung** (*Test and Set Lock*): **TSL RX, LOCK** speichert den Wert der Variable **LOCK** in **RX** und speichert einen ***Wert ungleich 0*** (hier: 1) in **LOCK**
- Die Operation ist **atomar**, kann also nicht unterbrochen werden.
- Aufzurufende Funktionen beim *Betreten* und *Verlassen* der kritischen Abschnitte:

```
enter_section:
  TSL RX, LOCK       ; kopiere LOCK und sperre mit 1
  CMP RX, #0         ; war die Sperrvariable 0?
  JNE enter_section  ; Wenn nein, dann ist CS gesperrt
  RET                ; Wenn ja, Ruecksprung und betritt CS

leave_section:
  MOV LOCK, #0       ; speichere 0 in Sperrvariable
  RET                ; Ruecksprung
```

FH Bielefeld
University of
Applied Sciences

# Prozesssynchronisation: Aktives Warten

- Sowohl Petersons Lösung wie auch die TSL-Anweisung arbeiten mit **aktiven Warteschleifen** (*Spinlocks, busy waiting*).
- **Probleme** beim aktiven Warten sind:
  - **CPU-Zeit wird verschwendet**, die andere Prozesse für sinnvolle Aufgaben benötigen.
    - Aktives Warten kann durch **Systemaufrufe**
    - **sleep()** (Prozess blockiert sich) und
    - **wakeup()** (weckt einen blockierten Prozess) umgangen werden.
  - **Prioritätsumkehrproblem**: Ist ein **niedrigpriorisierter** Prozess *L* in einem **kritischen Abschnitt** und will ein **hochpriorisierter** Prozess *H* diesen Betreten, so wird in einem System mit **striktem Prioritätsscheduling** *L* nie mehr durch den Scheduler aktiviert (*H* ist immer lauffähig).
    - ***L*** kann den kritischen Abschnitt **nicht mehr verlassen**.
    - ***H*** kann den kritischen Abschnitt **nicht mehr betreten**.

# Prozesssynchronisation: Semaphore

**Semaphore**:

- Ein Semaphor ist eine geschützte ganzzahlige **Zähler-Variable** (Integer),
- Unterstützt die beiden **unteilbaren** (*atomaren*) **Operationen**:
  - **down()**: kann aufrufenden Prozess schlafen legen
  - **up()**: weckt ggf. einen Prozess auf
- Bemerkung: **atomare Operation** stellt selbst kritischen Abschnitt dar!
- Realisierung per Systemaufruf mit Sperrung von Interrupts, Verwendung von **Warteschlange**.

```
down(sem)
{
  sem = sem - 1;
  if (sem < 0) queue_this_process_and_block();
}
up(sem)
{
  sem = sem + 1;
  if (sem <= 0) wakeup_first_process_in_queue();
}
```

FH Bielefeld
University of
Applied Sciences

# Prozesssynchronisation: Generelle Nutzung Semaphore

| Aktion | Wert Semaphor s | Aktiver Prozess | Inhalt Warteschlange |
|---|---|---|---|
| | 1 | % | [] |
| Prozess A tritt in kritischen Abschnitt eintreten → `down(s)` | 0 | A | [] |
| Prozess B will in kritischen Abschnitt eintreten → `down(s)` | -1 | A | [B] |
| Prozess C will in kritischen Abschnitt eintreten → `down(s)` | -2 | A | [B, C] |
| Prozess A verlässt kritischen Abschnitt → `up(s)` | -1 | B | [C] |
| Prozess B verlässt kritischen Abschnitt → `up(s)` | 0 | C | [] |
| Prozess C verlässt kritischen Abschnitt → `up(s)` | 1 | % | [] |

FH Bielefeld
University of
Applied Sciences

# Prozesssynchronisation: Beispiel zu Semaphoren

Erzeuger-Verbraucher Problem mit drei Semaphoren:

```
/* global Data */
#define N 100              /* number of slots in buffer  */
typedef int semaphore;     /* semaphores are integers    */
semaphore mutex = 1;       /* access to critical section */
semaphore unused = N;      /* counts unused slots        */
semaphore used  = 0;       /* counts used slots          */
```

```
void producer()
{
  while (true)
  {
    produce(&item);
    down(&unused);
    down(&mutex);
    buffer[in] = item;
    in = (in + 1) % N;
    up(&mutex);
    up(&used);
  }
}
```

```
void consumer()
{
  while (true)
  {
    down(&used);
    down(&mutex);
    item = buffer[out];
    out = (out + 1) % N;
    up(&mutex);
    up(&unused);
    consume(item);
  }
}
```

FH Bielefeld
University of
Applied Sciences

# Prozesssynchronisation: Mutexe

**Mutexe**:

- Ein Mutex (Kurzform von *mutual exclusion*, gegenseitiger Ausschluss) ist ein **binärer** (nicht zählender) **Semaphor**.
- Ein Mutex kann **zwei Werte** annehmen: *gesperrt* und *nicht gesperrt*.
  - **mutex_lock()** sperrt den Mutex.
  - **mutex_unlock()** gibt den Mutex wieder frei.
- Ein Mutex ist im allgemeinen *einfacher* zu realisieren als ein zählender Semaphor:
  - er kann aber immer auch durch einen zählenden Semaphor ersetzt werden.
  - andererseits: ein binärer Semaphor kann auch immer zur Implementierung eines zählenden Semaphors verwendet werden.

FH Bielefeld
University of
Applied Sciences

# Prozesssynchronisation: Mutexe & Semaphore

**Programmierfehler** können zu Verklemmungen oder falschen Ergebnissen führen:

- Ein Mutex-Semaphor wird am Ende des kritischen Abschnitts **nicht** wieder **freigegeben**.
- **Sprünge** in kritische Abschnitte ohne das Mutex-Semaphor zu setzen.
- **Vertauschen der Reihenfolge** von Mutex und zählendem Semaphor (z.B. Erzeuger-Verbraucher-Problem.)

**Fazit**:

- Semaphore und Mutexe erfordern eine **hohe Disziplin** vom Programmierer, eventuelle *Fehler sind nur schwer zu lokalisieren*.

- Beispiel: Praktikum „Multi-Threading und Synchronisation“

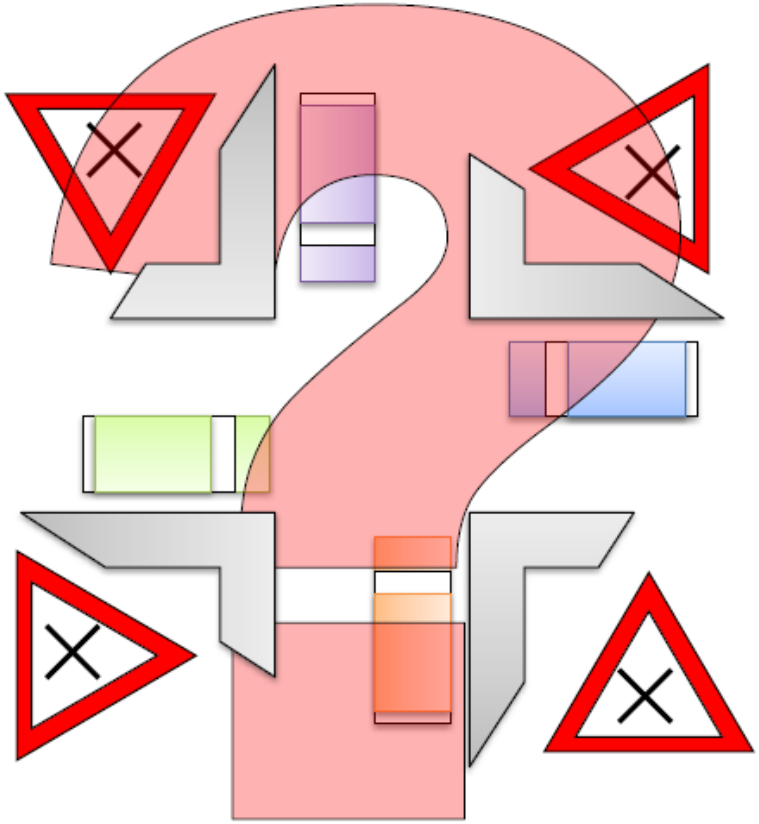
FH Bielefeld
University of
Applied Sciences

# Inhalt

- Interprozesskommunikation
  - Pipes, IPC, Shared Memory
- Mutexe und Semaphore
  - Schutz von kritischen Abschnitten
- Verklemmungen
  - Modellieren
  - Erkennen und Beheben
  - Verhindern (Banker Algorithm)

# Verklemmungen (Deadlocks)

- Deadlock im täglichen Leben: Straßenverkehr



FH Bielefeld
University of
Applied Sciences

# Verklemmungen (Deadlocks)

**Deadlock** in Computersystemen:

- Zur Erfüllung seiner Aufgaben benötigt ein Prozess **Betriebsmittel** (Ressourcen). Das können sein:
  - Drucker,
  - Scanner,
  - Speicher (vom Betriebssystem verwaltet),
  - aber auch z.B. ein gesperrter Datensatz in einer Datenbank.
- Beispiel: zwei Prozesse A und B arbeiten auf einer Datenbank
  - A reserviert Datensatz 1, B reserviert Datensatz 2
  - Nun möchte A Datensatz 2 reservieren und B Datensatz 1 – die Datensätze sind aber schon belegt.
  - Die entstehende Situation ist eine **Verklemmung** (Deadlock), da die **Prozesse nun für immer blockieren**.

FH Bielefeld
University of
Applied Sciences

# Arten von Ressourcen

Es wird zwischen zwei Arten von Ressourcen unterschieden:

- **Unterbrechbare** Ressourcen (*preemptable resources*):
  - Die Ressource kann ohne größere Problem dem Prozess entzogen werden. Beispiel: *Arbeitsspeicher*.
- **Ununterbrechbare** Ressourcen (*nonpreemptable resources*):
  - Die Ressource kann dem Prozess nicht entzogen werden, ohne dass die Ausführung fehlschlägt. Beispiel: *DVD-Brenner*.

Ressourcenanforderung:

- Ressourcen können vom Betriebssystem (*fremdverwaltet*) oder dem Prozess *selbstverwaltet* werden.
  - Beispiel für selbstverwaltete Ressourcen: Datensätze in einer Datenbank, mit anderen Prozessen geteilte Variablen, ...
- Bei selbstverwalteten Ressourcen sind Deadlocks wahrscheinlicher!

FH Bielefeld
University of
Applied Sciences

# Wiederholung Semaphore

**Semaphore**:

- Ein Semaphor ist eine geschützte ganzzahlige **Zähler-Variable** (Integer),
- Unterstützt die beiden **unteilbaren** (*atomaren*) **Operationen**:
  - **down()**:
    - legt aufrufenden Prozess schlafen
    - vermerkt den Prozess in der Warteschlange
  - **up()**:
    - weckt ggf. einen Prozess auf
    - entfernt Prozess aus Warteschlange

FH Bielefeld
University of
Applied Sciences

# Beispiel für Ressourcenanforderung

```
semaphore resource_1;
semaphore resource_2;

void process_A(void)
{
   down(&resource_1);
   down(&resource_2);
   use_resources();
   up(&resource_2);
   up(&resource_1);
}
void process_B(void)
{
   down(&resource_1);
   down(&resource_2);
   use_resources();
   up(&resource_2);
   up(&resource_1);
}
```

a) Deadlock freier Code

```
semaphore resource_1;
semaphore resource_2;

void process_A(void)
{
   down(&resource_1);
   down(&resource_2);
   use_resources();
   up(&resource_2);
   up(&resource_1);
}
void process_B(void)
{
   down(&resource_2);
   down(&resource_1);
   use_resources();
   up(&resource_1);
   up(&resource_2);
}
```

b) Code mit möglichem Deadlock

FH Bielefeld
University of
Applied Sciences

# Deadlock: Definition

Definition eines Deadlocks:

> *Eine Menge von Prozessen befindet sich in einer Verklemmung (deadlock), wenn jeder Prozess der Menge auf ein Ereignis wartet, dass nur ein anderer Prozess aus der Menge auslösen kann.*

FH Bielefeld
University of
Applied Sciences

# Deadlocks: Bedingungen

Vier **Bedingungen**, die zum Auftreten einer Verklemmung *notwendig* sind (*Coffman / Elphick / Shoshani*, 1971):

1. *Wechselseitiger Ausschluss*:
   - Jede Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet.
2. *Hold-and-wait-Bedingung*:
   - Prozesse, die schon Ressourcen reserviert haben, können noch weitere Ressourcen anfordern.
3. *Nichtunterbrechbarkeit*:
   - Ressourcen, die einem Prozess bewilligt wurden, können diesem nicht wieder entzogen werden (*no preemption*).
4. *Zyklische Wartebedingung*:
   - Es muss eine zyklische Kette von Prozessen geben, von denen jeder auf eine Ressource wartet, die dem nächsten Prozess in der Kette gehört.

FH Bielefeld
University of
Applied Sciences

# Deadlocks: Modellierung

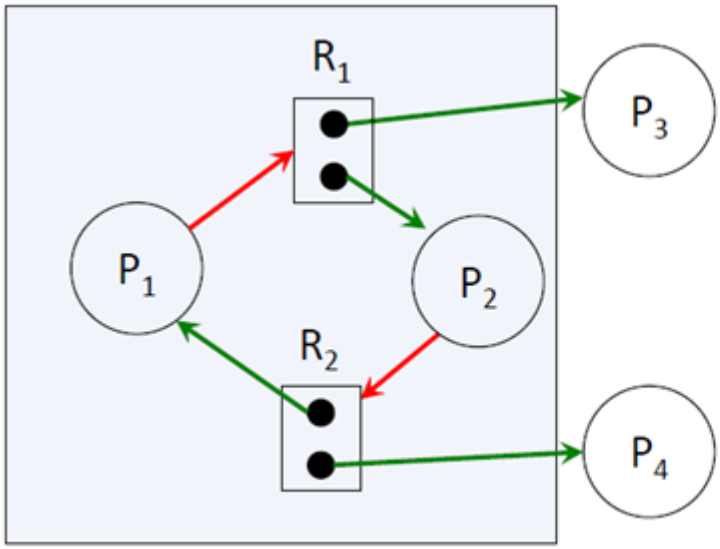
- Die vier Bedingungen für einen Deadlock können mit einem gerichteten Graphen dargestellt werden.
- **Belegungs-Anforderungs-Graph** (*resource allocation graph*):
  - Der Graph besteht aus einer Menge von **Prozessen** $P = \{P_1, P_2, P_3, \ldots, P_n\}$ und einer Menge von **Ressourcen** $R = \{R_1, R_2, R_3, \ldots, R_m\}$ sowie **gerichtete** Kanten.
  - Eine gerichtete Kante von einer Ressource $\mathbf{R_j}$ zu einem Prozess $\mathbf{P_i}$ bedeutet, dass die Ressource dem Prozess zugewiesen wurde (*assignment edge*).
  - Eine gerichtete **Kante** von einem Prozess $\mathbf{P_i}$ zu einer Ressource $\mathbf{R_j}$ bedeutet, dass der Prozess die Ressource beanspruchen möchte (*request edge*).
- Ein **Zyklus** im Graph bedeutet einen **Deadlock**.

FH Bielefeld
University of
Applied Sciences

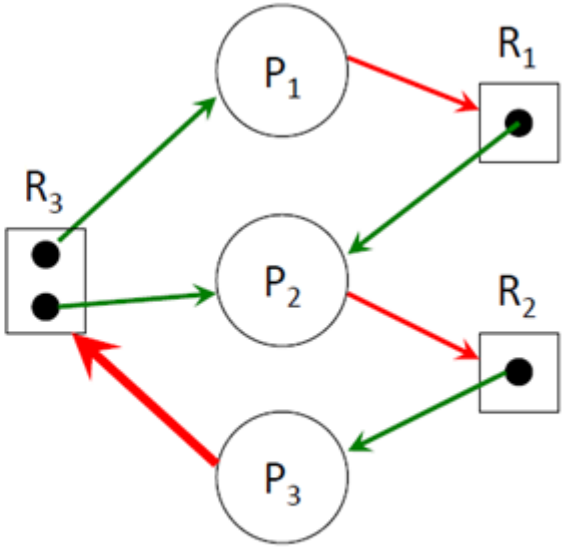
# Deadlocks: Modellierung

Belegungs-Anforderungs-Graph mit *mehreren Ressourcen* vom gleichen Typ.
Beispiel:

- Ein Rechenzentrum besitzt mehrere Drucker gleichen Typs.
- Für einen Prozess ist es transparent (und egal!) auf welchem Drucker er seine Ausgabe macht.




Trotz Zyklus **kein** Deadlock



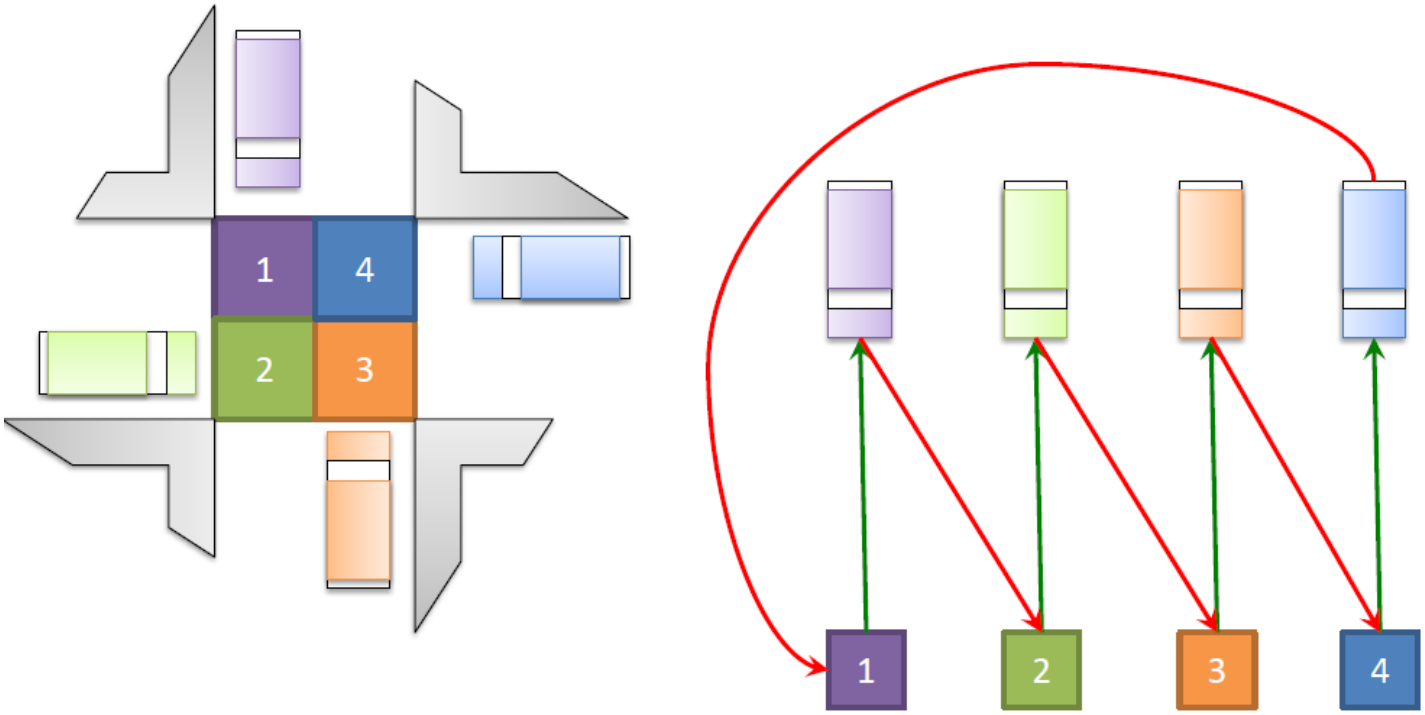

Hier existiert ein Deadlock

FH Bielefeld
University of
Applied Sciences

# Eigenschaften von Belegungs-Anforderungs-Graphen

- Enthält ein Belegungs-Anforderungs-Graph **keine Zyklen**, dann existiert auch **keine Verklemmung**.
- Besitzt ein Belegungs-Anforderungs-Graph **einen Zyklus** und existiert von jeder beteiligten **Ressource** nur genau **ein Exemplar**, dann existiert eine **Verklemmung**
- Besitzt ein Betriebsmittel-Zuweisungsgraph einen **Zyklus** und von den beteiligten Ressourcen existieren **mehrere Exemplare**, so ist eine Verklemmung möglich, aber **nicht unbedingt** auch eingetreten.

# Belegungs-Anforderungs-Graph: Modellierung Straßenkreuzung




- Jedes Auto **fährt ein Stück** auf die Kreuzung (Areal = *Ressource*). Ein weiteres Areal wird beansprucht, um die Kreuzung **gerade passieren** (= *Prozess*) zu können. Dies wird aber von einem **anderen Fahrzeug belegt** →*Deadlock*.

FH Bielefeld
University of
Applied Sciences

# Behandlung von Deadlocks

Dem Deadlock Problem kann auf **vier möglichen Weisen** begegnet werden:

1. **Ignorieren**: Erscheint unangemessen, aber:
   - Die meisten Betriebssysteme inklusive Windows und Unix verfahren so.
   - Kein zusätzlicher Overhead für selten auftretende Ereignisse.
2. **Erkennen und Beheben**: Lasse Deadlocks passieren und behebe sie dann.
3. **Dynamische Verhinderung:** durch vorsichtiges Ressourcenmanagement.
4. **Vermeidung von Deadlocks**: Eine der vier notwendigen Bedingungen muss prinzipiell unerfüllbar werden.

FH Bielefeld
University of
Applied Sciences

# Deadlocks ignorieren

FH Bielefeld
University of
Applied Sciences

## Vorlesung

**Vielen Dank für Ihre Aufmerksamkeit**

## Dozent

**Prof. Dr.-Ing.**
**Martin Hoffmann**
martin.hoffmann@fh-bielefeld.de