Kapitel 3
Prozess-Synchronisation

1. Einführung und Grundlagen
2. Aktives Warten
3. Semaphore
4. Monitore
5. Message Passing
6. UNIX
7. Windows
8. Deadlocks

Einführung

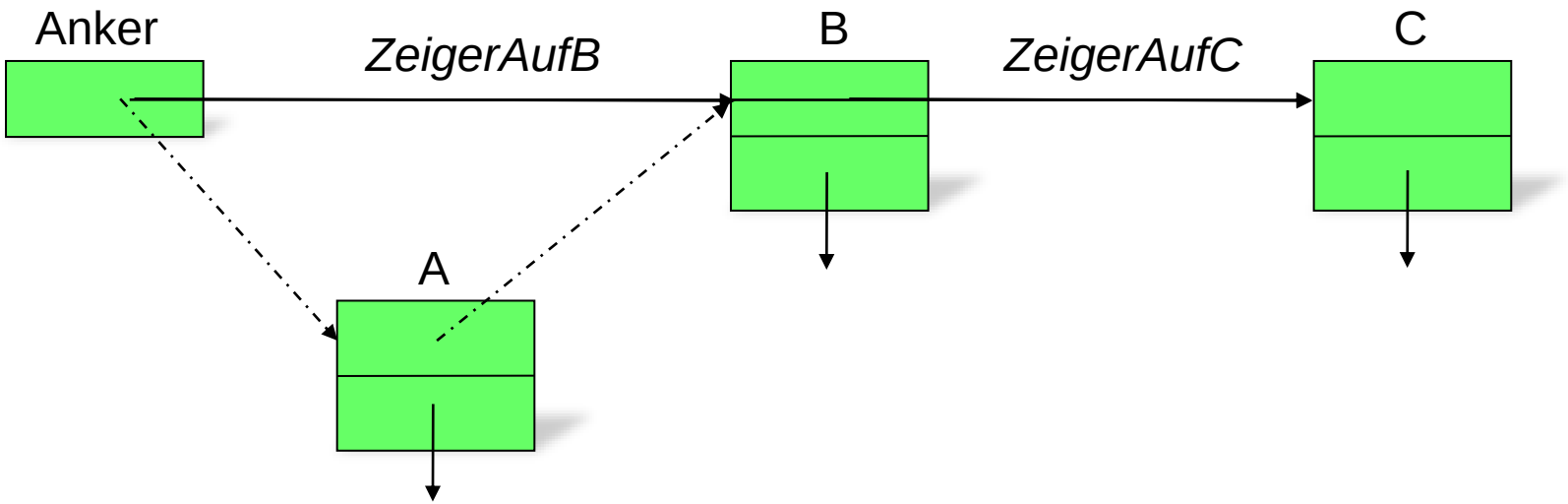
- Prozesse / Threads sind **konkurrierend / nebenläufig / parallel** ablaufend, wenn sie zu derselben Zeit existieren.
- Die Prozesse / Threads können zugeordnet sein
  - demselben Prozessor
  - verschiedenen Prozessoren mit gemeinsamem Hauptspeicher
  - voneinander weitgehend unabhängigen Prozessoren in verteilten Systemen.

Problemstellungen

- **Prozess-Synchronisation**: Herstellen einer zeitlichen Reihenfolge zwischen Prozessen
  - Wechselseitiger Ausschluss (z.B. für Zugriff auf gemeinsam benutzte Betriebsmittel)
  - Ablaufsteuerung bei Abhängigkeiten (Einhalten von Reihenfolgebedingungen)
- **Prozess-Kommunikation**: Expliziter Austausch von Daten zwischen Prozessen
  - Kommunikation innerhalb eines Systems
  - Kommunikation in verteilten Systemen

Probleme und Lösungen gelten ebenso für Threads

Beispiel („Pleiten, Pech und Pannen“): Gemeinsamer Zugriff auf eine Liste



Prozess 1:

Einhängen von Block A

2

(1) Lesen des Ankers: ZeigerAufB

(2) Setzen des NextZeigersA=ZeigerAufB

(3) Setzen des Ankers=ZeigerAufA

Prozess 2:

Aushängen von Block B

1

(1) Lesen des Ankers: ZeigerAufB

(2) Lesen des NextZeigersB: ZeigerAufC

(3) Setzen des Ankers=ZeigerAufC

3

➔ **Block A ist ebenfalls weg!!**

Kritische Abschnitte

Die Nebenläufigkeit führt zu der Notwendigkeit, wechselseitigen Ausschluss (mutual exclusion) bzgl. des Zugriffs auf bestimmte Objekte zu implementieren

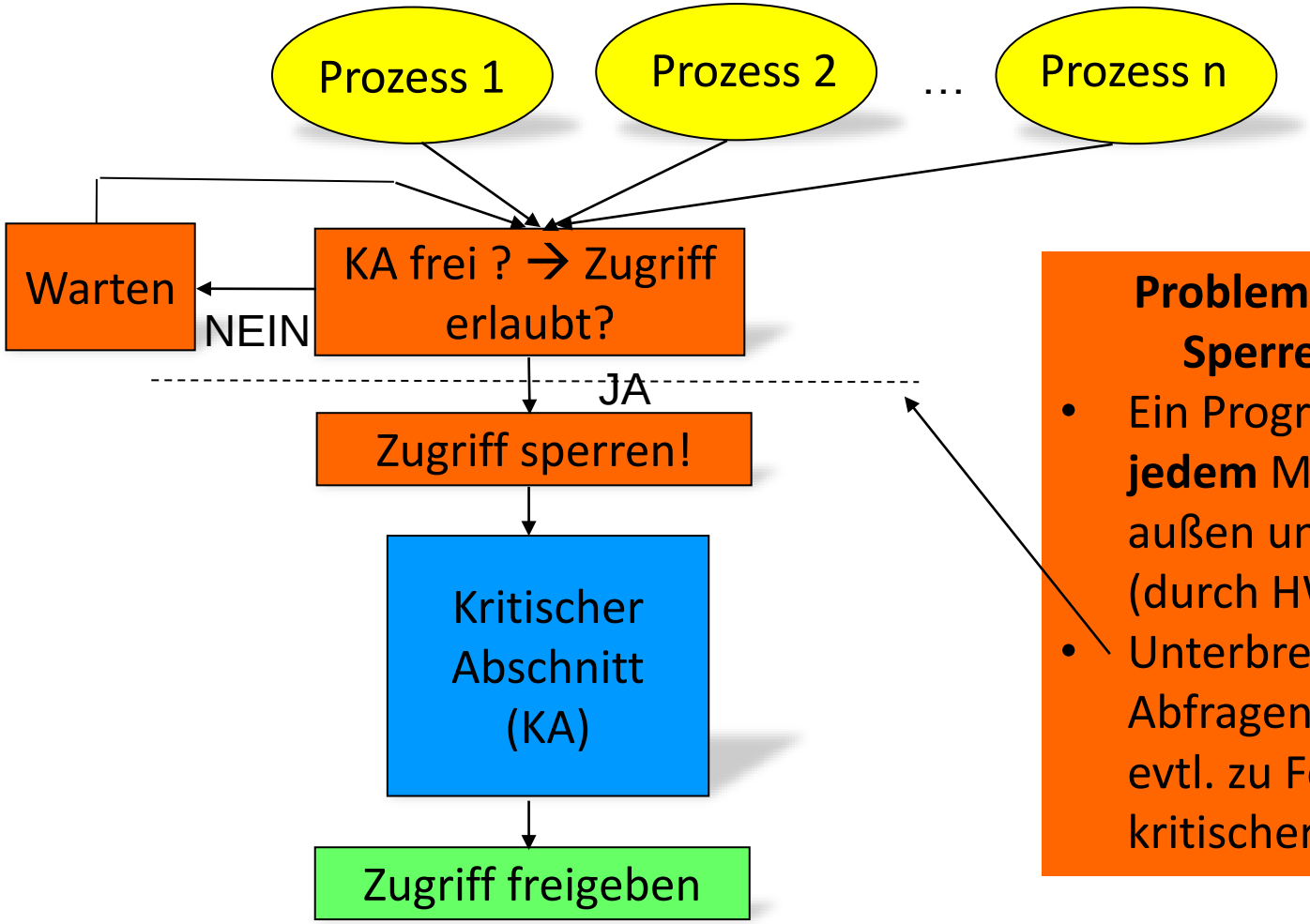
→ Definition von **kritischen Abschnitten** (critical sections):

- Ein kritischer Abschnitt ist ein Codeabschnitt, der zu einer Zeit nur durch einen Prozess bzw. Thread durchlaufen und in dieser Zeit nicht durch andere nebenläufige Prozesse bzw. Threads betreten werden darf
  → *keine Unterbrechung im kritischen Abschnitt erlaubt!*
- Im Beispiel: Der gesamte Code "Einhängen von Block A" (Prozess 1) und "Aushängen von Block B" (Prozess 2) bildet zusammen einen kritischen Abschnitt *(weil auf dieselben Objekte zugegriffen wird)*!

Anforderungen an Lösungen für wechselseitigen Ausschluss

1. Keine zwei Prozesse dürfen gleichzeitig im kritischen Abschnitt sein
2. Prozesse, die sich nicht im kritischen Abschnitt aufhalten, dürfen den Prozess im kritischen Abschnitt nicht beeinflussen (blockieren)
3. Kein Prozess, der einen kritischen Abschnitt betreten möchte, darf endlos zum Warten gezwungen werden
4. Es dürfen keine Annahmen über die Geschwindigkeit, mit der ein Prozess ausgeführt wird oder die Anzahl der Prozessoren gemacht werden (keine Wettrennen, „Race Conditions“)

Allgemeine Synchronisationslösung für wechselseitigen Ausschluss



Problem beim Abfragen / Sperren des Zugriffs:

- Ein Programm kann nach **jedem** Maschinenbefehl von außen unterbrochen werden (durch HW-Interrupt)!!
- Unterbrechung zwischen Abfragen und Sperren führt evtl. zu Fehlern → ist eigener kritischer Abschnitt!

Einfache Hardwarelösung: Ausschalten von Interrupts

- Ein Prozess kann im Prinzip durch Ausschalten der Interrupts (Interrupt-Bit) einen kritischen Abschnitt sichern:

```
DisableInterrupts();
/*kritischer Abschnitt */
EnableInterrupts();
```

- Probleme?
  - Funktioniert nur bei Einprozessor-Systemen
  - Es geht nur im Kernel-Mode
  - Interrupts könnten verloren gehen
  - Erfüllung der Anforderungen??

Prozess-Synchronisation: Lösungskonzepte

- Prozess-Synchronisation
  - Aktives Warten
  - Semaphore (Mutexe)
  - Signale / Events
  - Monitore
  - (Atomare) Transaktionen
  - ...
- Prozess-Kommunikation (Austausch von Nachrichten)
  - Shared Memory
  - Message Passing

Kapitel 3
Prozess-Synchronisation

1. Einführung und Grundlagen
2. Aktives Warten
3. Semaphore
4. Monitore
5. Message Passing
6. UNIX
7. Windows
8. Deadlocks

Was ist „Aktives Warten“?

- **Aktives Warten** (“Busy Waiting”)
  - Ein Prozess prüft ständig, ob er einen kritischen Abschnitt betreten darf *(z.B. in einer “while”-Schleife)*
  - Der Prozess ist aktiv nur mit dem Abfragen der Zugriffssperre beschäftigt und kann nichts Produktives durchführen
  - Der Prozess verbraucht durch das Abfragen aber CPU-Zeit!
- Eine Sperre, die aktives Warten verwendet, heißt „Spinlock“

Bewertung: Softwarelösungen für Aktives Warten

- Vorteil: Es gibt eine allgemeine Softwarelösung, welche die Anforderungen erfüllt (Peterson 1981)
- Nachteile:
  - **Aktives Warten verschwendet unnötig Prozessorzeit**
  - Anzahl der Prozesse muss vorher bekannt sein
  - Aufwändige Programmierung

➔ Keine Verwendung von „Aktivem Warten“ in Anwendungsprogrammen!

Hardwareunterstützung für unterbrechungsfreies Sperren: „Test and Set“ - Befehl

- Anforderung: Es muss eine Variable in **einem** Maschinenbefehl abgefragt und – wenn frei (Wert 0) – gesetzt werden (auf Wert 1) (→ nicht unterbrechbar!)
- Der Rückgabewert muss erkennen lassen, ob die Variable **vor Aufruf** frei war (TRUE) oder besetzt (FALSE)
- Pseudo-C-Code für den Maschinenbefehl:

```
boolean TestAndSet(int* i) {
  if (*i == 0) {
    *i = 1;
    return TRUE;
  } else {
    return FALSE;
  }
}
```

Aktives Warten mit TestAndSet: Abfragen und Sperren in einem Befehl

```
int lock = 0;
…
/* Kreisen in der while-Schleife, bis TestAndSet
   den Wert TRUE liefert
   Bei FALSE hat ein anderer Prozess
   die Sperre gesetzt */
while (TestAndSet (&lock) == FALSE){};
/* kritischer Abschnitt */
lock = 0;          /* Freigabe */
```

Bewertung: Hardwarelösung für Aktives Warten

- Vorteile:
  - Nicht-Unterbrechbarkeit kann auf einfache und sichere Weise auf Hardwareebene unterstützt werden
- Nachteile:
  - Aktives Warten verschwendet unnötig Prozessorzeit
  - Wettrennen (Race Conditions) werden nicht verhindert
  - Verhungern eines Prozesses ist möglich
  - Endloses Warten (Deadlock) ist ebenfalls möglich

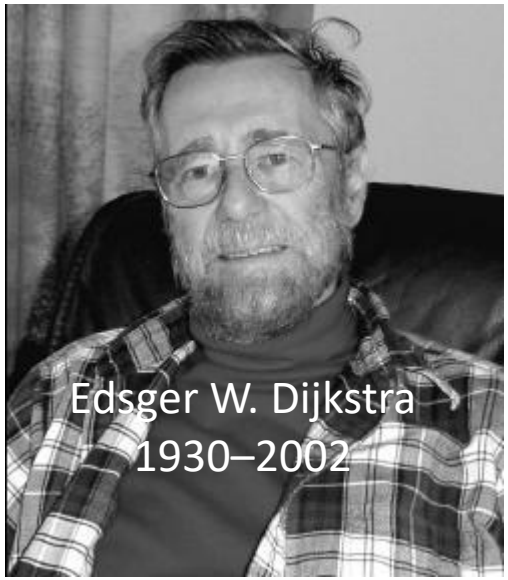
➔ Anwendung von hardwareunterstütztem „Aktiven Warten“ nur durch Systemprogramme für sehr kurze kritische Abschnitte

Kapitel 3
Prozess-Synchronisation

1. Einführung und Grundlagen
2. Aktives Warten
3. Semaphore
4. Monitore
5. Message Passing
6. UNIX
7. Windows
8. Deadlocks

Semaphore

- E. W. Dijkstra schlug 1965 Semaphore als allgemeines Konstrukt für den wechselseitigen Ausschluss und allgemeinere Synchronisationsprobleme vor
- Ein Semaphor ist ein Sperrmechanismus, der Prozesse unter bestimmten Bedingungen blockiert oder wieder freigibt
- Semaphore werden i.d.R. vom Betriebssystem zur Verfügung gestellt



Semaphor-Analogie: Regelung des Zugangs zu Geschäften

- Um Gedränge zu vermeiden, ist die **Anzahl der Kunden im Laden beschränkt**.
- Durchgesetzt wird dies mit einem **Stapel von Einkaufskörben am Eingang**.
- Jeder Kunde muss beim Betreten des Ladens **einen Einkaufskorb nehmen, ihn mit sich führen und beim Verlassen wieder abgeben**.
- Sind **keine Körbe mehr vorhanden, so muss er warten, bis ein anderer Kunde das Geschäft verlässt und seinen Korb wieder abgibt**.
- Es bildet sich so an der Tür eine mehr oder weniger lange **Warteschlange**, denn es können ja nie mehr Leute im Geschäft sein, als Körbe vorhanden sind.

Semaphor - Struktur

- Eine Integer-Zählvariable **S** ≥ 0 *(→ Anzahl freier Körbe)*
- Eine Warteschlange für Prozesse *(→ für Kunden vor dem Geschäft, wenn kein Einkaufskorb bereit steht)*
- Atomare (nicht-unterbrechbare) Zugriffsfunktionen
  - **P(S)** *("Passieren")* versucht, die Erlaubnis zum Passieren zu erhalten. Bei S == 0 in die Warteschlange einreihen, ansonsten S-- und eintreten *(→ Korb vom Stapel nehmen – falls vorhanden – und Geschäft betreten).*
  - **V(S)** *("Verlassen")* teilt das Verlassen des kritischen Abschnitts mit, daher S++ und weiter *(→ Korb abgeben und Geschäft verlassen).*
    Wenn ein Prozess V(S) ausführt und die Warteschlange nicht leer ist, wird genau einer dieser Prozesse „geweckt" und kann seine P-Operation beenden (→ *Irgendein Kunde aus der Warteschlange darf passieren und sich den Korb nehmen).*

Semaphor-Implementierung durch das Betriebssystem

- Ein in einer P-Operation wartender Prozess ist im „Blockiert“-Zustand, bis Wecken durch ein Signal / Ereignis erfolgt ➔ **kein** aktives Warten nötig!
- Implementierung der Nicht-Unterbrechbarkeit von P und V mittels Hardwareunterstützung *(z.B. durch “TestAndSet”-Befehl)*
- P(S) und V(S) bilden gemeinsam einen eigenen kritischen Abschnitt!

Allgemeines Semaphor

S wird zu Beginn mit einem Wert n ≥ 0 initialisiert

```
P(S):        while (S == 0 ) {
                <blockieren und warten>;
                }
             S-- ;
```

```
V(S) :       S++ ;
             if (< mindestens ein Prozess wartet auf S >) {
                        < wecke einen wartenden Prozess >;
             }
```

Typische Anwendung von allgemeinen Semaphoren:
Verwaltung von Pufferspeichern (Einhalten von Reihefolgebedingungen)
→ „Erzeuger / Verbraucher“ - Problem

Andere Bezeichungen:

- **P(S)** ≅ down(S) ≅ wait(S) ≅ **acquire(S)**
- **V(S)** ≅ up(S) ≅ signal(S) ≅ **release(S)**

Beispiel-Lösung mit JAVA und Semaphoren

Voraussetzung: Klasse „Semaphore“

Package: java.util.concurrent

```
public class Semaphore {

   // Konstruktor: Initialisierung von S
   public Semaphore(int permits) { }

   public void acquire() { } // P
   public void release() { } // V
}
```

➔ Beispiel: **Shop-Simulation**

Binäres Semaphor („Mutex“)

S wird zu Beginn mit 1 initialisiert (nur ein Prozess darf in den KA)

```
P(S):       while (S == 0 ) {
               <blockieren und warten>;  /* bis S == 1 */
               }
            S = 0;
```

```
V(S) :      S = 1;
            if (< mindestens ein Prozess wartet auf S >) {
                     < wecke einen wartenden Prozess >;
            }
```

Typische Anwendung von binären Semaphoren:

Wechselseitiger Ausschluss („Mutual Exclusion Problem” → **Mutex**)

Initialisierung: S = 1 („Kritischer Abschnitt frei für einen Prozess“)

```
P(S);
<kritischer Abschnitt>;
V(S);
```

Andere Bezeichungen: P(S) ≅ lock(S), V(S) ≅ unlock(S)

Beispiel-Lösung mit JAVA und Semaphoren

Voraussetzung: Klasse „ReentrantLocks“

Wiedereintrittsfähige Mutex-Implementierung,
d.h. derselbe Thread darf P evtl. mehrfach aufrufen

Package: java.util.concurrent.locks

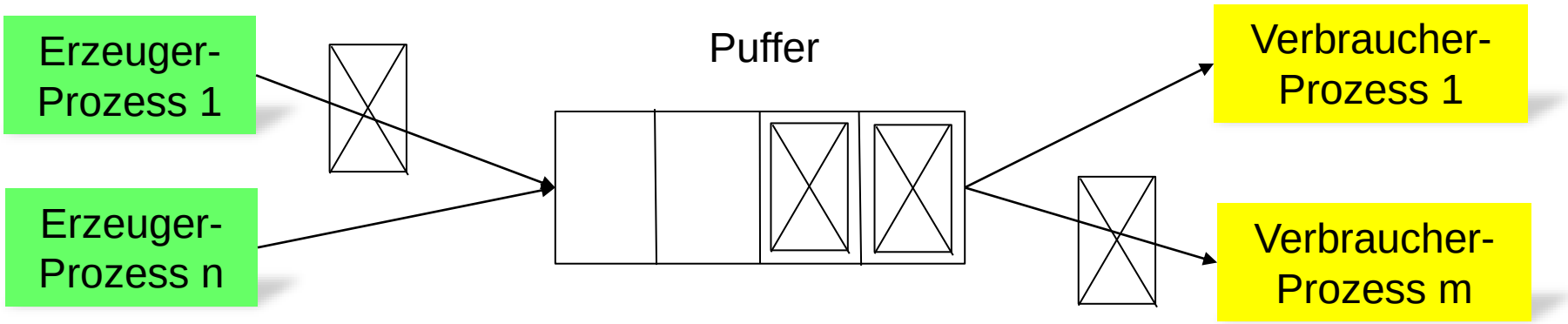
```
public class ReentrantLock {

   // Konstruktor: Initialisierung von S = 1
   public ReentrantLock() { }

   public synchronized void lock() { }    // P
   public synchronized void unlock() { } // V
}
```

➔ Beispiel: **TestThread7**

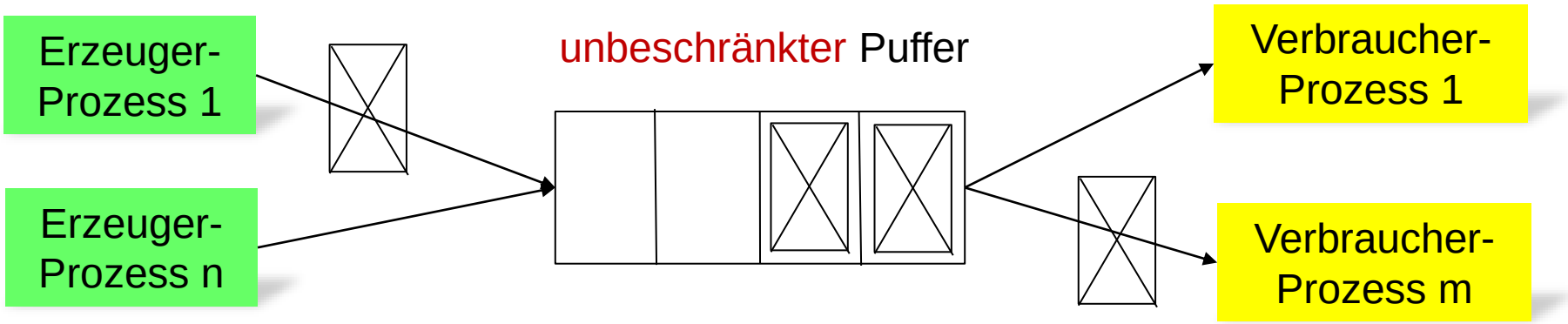
Verwaltung von Pufferspeichern



Erzeuger-Verbraucher-Problem mit Puffer:

- Ein oder mehrere Erzeuger-Prozesse generieren einzelne Datenpakete und speichern diese in einem Puffer (falls nicht voll)
- Ein oder mehrere Verbraucher-Prozesse entnehmen einzelne Datenpakete aus dem Puffer (falls nicht leer) und verbrauchen diese
- Zu jedem Zeitpunkt darf nur ein Prozess (Erzeuger oder Verbraucher) auf den Puffer zugreifen (→ Kritischer Abschnitt)

Verwaltung von unbeschränkten Pufferspeichern



Synchronisationselemente

- Mutex (binäres Semaphor): **S** (Synchronisation des Pufferzugriffs)
  → Zu jedem Zeitpunkt darf nur ein Prozess auf den Puffer zugreifen
- Allgemeines Semaphor: **B** (Anzahl belegter Pufferplätze)
  → Verbraucher müssen warten, wenn Puffer leer ist

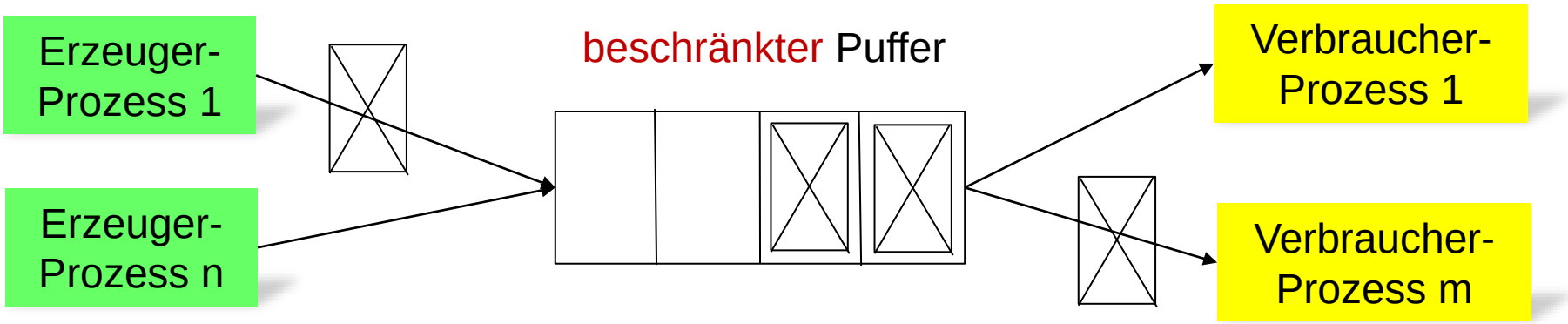
B wird mit 0 initialisiert (zu Beginn ist kein Platz belegt)
Bevor ein Element aus dem Puffer genommen wird: **`P(B)`** (B--)
Nachdem ein Element in den Puffer gelegt wurde: **`V(B)`** (B++)

Lösung: Erzeuger-Verbraucher-Problem mit unbeschränktem Puffer

- Initialisierung:
  - Mutex **S** = 1 /* Synchronisation des Pufferzugriffs */
  - Allg. Semaphor **B** = 0 /* Anzahl belegter Pufferplätze */
- Pseudo-Code:

Erzeuger	Verbraucher
`<Erzeuge Element e>;`	`P(B);  /* acquire */`
`P(S);  /* lock */`	`P(S);  /* lock */`
`<speichere e im Puffer>`	`<nimm e aus Puffer>`
`V(S);  /* unlock */`	`V(S);  /* unlock */`
`V(B);  /* release */`	`<verarbeite e>;`

Verwaltung von beschränkten Pufferspeichern



Synchronisationselemente

- Mutex (binäres Semaphor): **S** (Synchronisation des Pufferzugriffs)
  → Zu jedem Zeitpunkt darf nur ein Prozess auf den Puffer zugreifen

- Allgemeines Semaphor: **F** (Anzahl freier Pufferplätze) → Erzeuger müssen warten, wenn Puffer voll ist
  F wird mit der Pufferkapazität initialisiert (zu Beginn sind alle Plätze frei)
  Bevor ein Element in den Puffer gelegt wird: **`P(F)`** (F--)
  Nachdem ein Element aus dem Puffer genommen wurde: **`V(F)`** (F++)

Allgemeines Semaphor: **B** (Anzahl belegter Pufferplätze) → Verbraucher müssen warten, wenn Puffer leer ist

B wird mit 0 initialisiert (zu Beginn ist kein Platz belegt)
Bevor ein Element aus dem Puffer genommen wird: **`P(B)`** (B--)

Nachdem ein Element in den Puffer gelegt wurde: **`V(B)`** (B++)

Lösung: Erzeuger-Verbraucher-Problem mit beschränktem Puffer

- Initialisierung (N Plätze im Puffer):
  - Mutex **S** = 1 /* Synchronisation des Pufferzugriffs */
  - Allg. Semaphor **B** = 0 /* Anzahl belegter Pufferplätze */
  - Allg. Semaphor **F** = N /* Anzahl freier Pufferplätze */

Erzeuger	Verbraucher
`<Erzeuge Element e>;`	`P(B);  /* acquire */`
`P(F);  /* acquire */`	`P(S);  /* lock */`
`P(S);  /* lock */`	`<nimm e aus Puffer>`
`<speichere e im Puffer>`	`V(S);  /* unlock */`
`V(S);  /* unlock */`	`V(F);  /* release */`
`V(B);  /* release */`	`<verarbeite e>;`

Beispiel-Lösung Erzeuger-Verbraucher-Problem mit JAVA und Semaphoren

- Klasse **`BoundedBufferServer`**
  - Erzeugt eine Simulationsumgebung für ein Erzeuger/Verbrauchersystem (Erzeugen eines Puffers, Erzeugen und Beenden der Erz./Verbr.-Threads)
- Klasse **`BoundedBuffer<E>`**
  - Stellt einen generischen Datenpuffer (für Elemente vom Typ E) mit synchronisierten Zugriffsmethoden **`enter`** und **`remove`** zur Verfügung
- Klasse **`Producer extends Thread`**
  - Erzeuger-Thread: Erzeuge Date-Objekte und lege sie in den Puffer. Halte nach jeder Ablage für eine Zufallszeit an.
- Klasse **`Consumer extends Thread`**
  - Verbraucher-Thread: Entnimm Date-Objekte einzeln aus dem Puffer. Nach jeder Entnahme für eine Zufallszeit anhalten.

Beispiel: Erzeuger-/ Verbraucher-Problem in JAVA: Erzeuger-Code (Semaphor-Lösung)

Erzeuger führen die `enter`-Methode der Klasse `BoundedBuffer<E>` aus:

```
public void enter(E item) throws InterruptedException {
 sem_F.acquire();  // Freier Pufferplatz vorhanden?
 mutex_S.lockInterruptibly(); // Zugriff sperren

 buffer.add(item);    // Datenpaket in den Puffer legen

 mutex_S.unlock();    // Zugriff freigeben
 sem_B.release();     // ggf. Verbraucher wecken
}
```

Beispiel: Erzeuger-/ Verbraucher-Problem in JAVA: Verbraucher-Code (Semaphor-Lösung)

Verbraucher führen die `remove`-Methode der Klasse `BoundedBuffer<E>` aus:

```
public E remove() throws InterruptedException {
   E item;

   sem_B.acquire();  // Mindestens ein Platz belegt?
   mutex_S.lockInterruptibly(); // Zugriff sperren

   /* Datenpaket aus dem Puffer holen */
   item = buffer.removeFirst();

   mutex_S.unlock();  // Zugriff freigeben
   sem_F.release();   // ggf. Erzeuger wecken
   return item;
}
```

➔ Erzeuger-Verbraucher (Semaphore)

Kapitel 3
Prozess-Synchronisation

1. Einführung und Grundlagen
2. Aktives Warten
3. Semaphore
4. Monitore
5. Message Passing
6. UNIX
7. Windows
8. Deadlocks

Monitore

- Schwierigkeit mit der „Semaphor“-Lösung ?
  - Paarweise Benutzung P/V notwendig
  - Fehleranfällig
- Hoare, Hansen [74] schlugen als Programmiersprachenkonzept den Monitor vor
  - Ist in z.B. Concurrent Pascal, Ada, Modula-2/3 und **JAVA** implementiert

Sir C.A.R. Hoare

Per Brinch Hansen

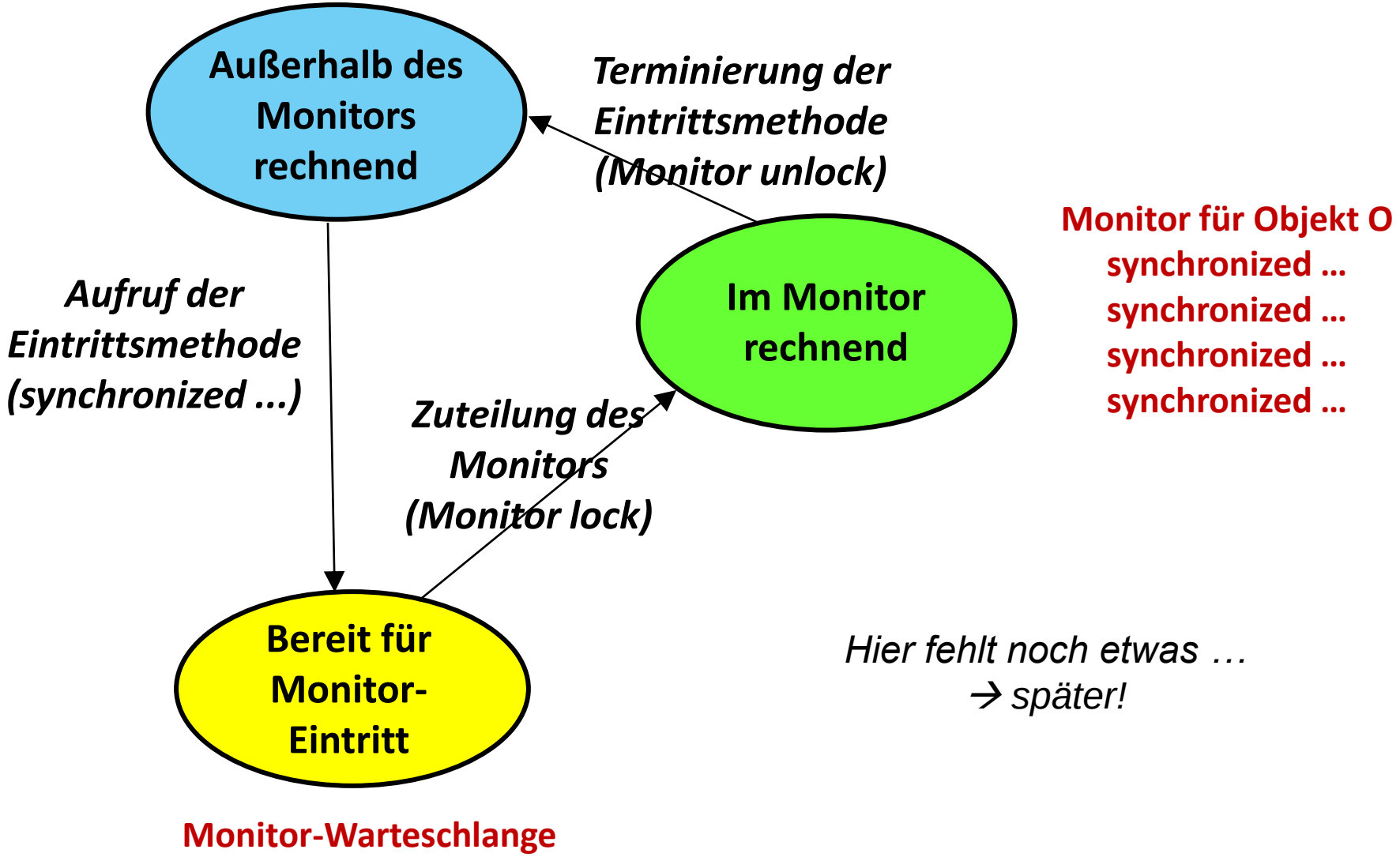
Was ist ein „Monitor“?

- Ein **Monitor überwacht den Aufruf bestimmter Methoden**
- Ein Thread „betritt“ den überwachten Monitorbereich durch den Aufruf **einer** der Methoden und „verlässt“ ihn mit dem Ende dieser Methode
- Nur ein Thread zur Zeit kann sich innerhalb der überwachten Monitor-Methoden aufhalten *(Sperre des kritischen Abschnitts)*, die übrigen müssen warten
- Die wartenden Threads werden von dem Monitor in einer Monitor-Warteschlange verwaltet (sind blockiert)
- Es gibt zusätzliche Synchronisationsfunktionen innerhalb des Monitors *(zum Einhalten von Reihenfolgebedingungen → später)*

Monitore in Java

- **Jedes** JAVA-Objekt besitzt einen eigenen Monitor!
- Der Monitor eines Objekts überwacht **alle** Methoden / Blöcke des Objekts, die mit **`synchronized`** bezeichnet sind.
- Eintritt in den Monitorbereich über Aufruf **irgendeiner** **`synchronized`** – Methode des Objekts
- Beim Eintritt in eine **`synchronized`**-Methode wird der Monitorbereich des entsprechenden Objekts für andere Threads gesperrt und nach dem Austritt wieder freigegeben
- Ist der Monitorbereich eines Objektes gesperrt
  - … kann kein anderer Thread eine synchronisierte Methode dieses Objektes ausführen (→ ab in die Monitor-Warteschlange!)
  - Eine unsynchronisierte Methode lässt sich dagegen ausführen!

JAVA-Synchronisation: Thread-Zustandsdiagramm



Angabe eines Monitors in Java

- Synchronisation von Blöcken
  - Es können beliebige Code-Blöcke synchronisiert werden
  - Angabe eines Synchronisationsobjekts (welcher Monitor?) nötig
  - Syntax: `synchronized ( <Synchr.-Objekt> ) { ... }`
- Synchronisation von Methoden einer Klasse
  - Schlüsselwort **`synchronized`** im Methodenkopf angeben
  - Wirkung ist identisch mit **`synchronized(this) { ... }`** am Anfang der Methode
- Synchronisation über Klassen
  - Wie Objekte, besitzt auch jede Klasse genau einen Monitor
  - Eine Klassenmethode, die die Attribute **`static synchronized`** trägt, fordert somit den *Monitor der Klasse* an *(Blocksynchronisation:* **`getClass()`** *als Objektreferenz verwenden)*

Wechselseitiger Ausschluss über Monitor

Monitor-Lösung *(ThreadTest8)*:

```
public synchronized void showOutput(Object output) {
    /* Kritischer Abschnitt */
}
```

Im Vergleich: Semaphor-Lösung *(ThreadTest7)*:

```
private ReentrantLock mutex = new ReentrantLock();
public void showOutput(Object output) {
    mutex.lock();      // P(mutex)
    /* Kritischer Abschnitt */
    mutex.unlock();    // V(mutex)
}
```

Reihenfolgebedingungen im Monitor

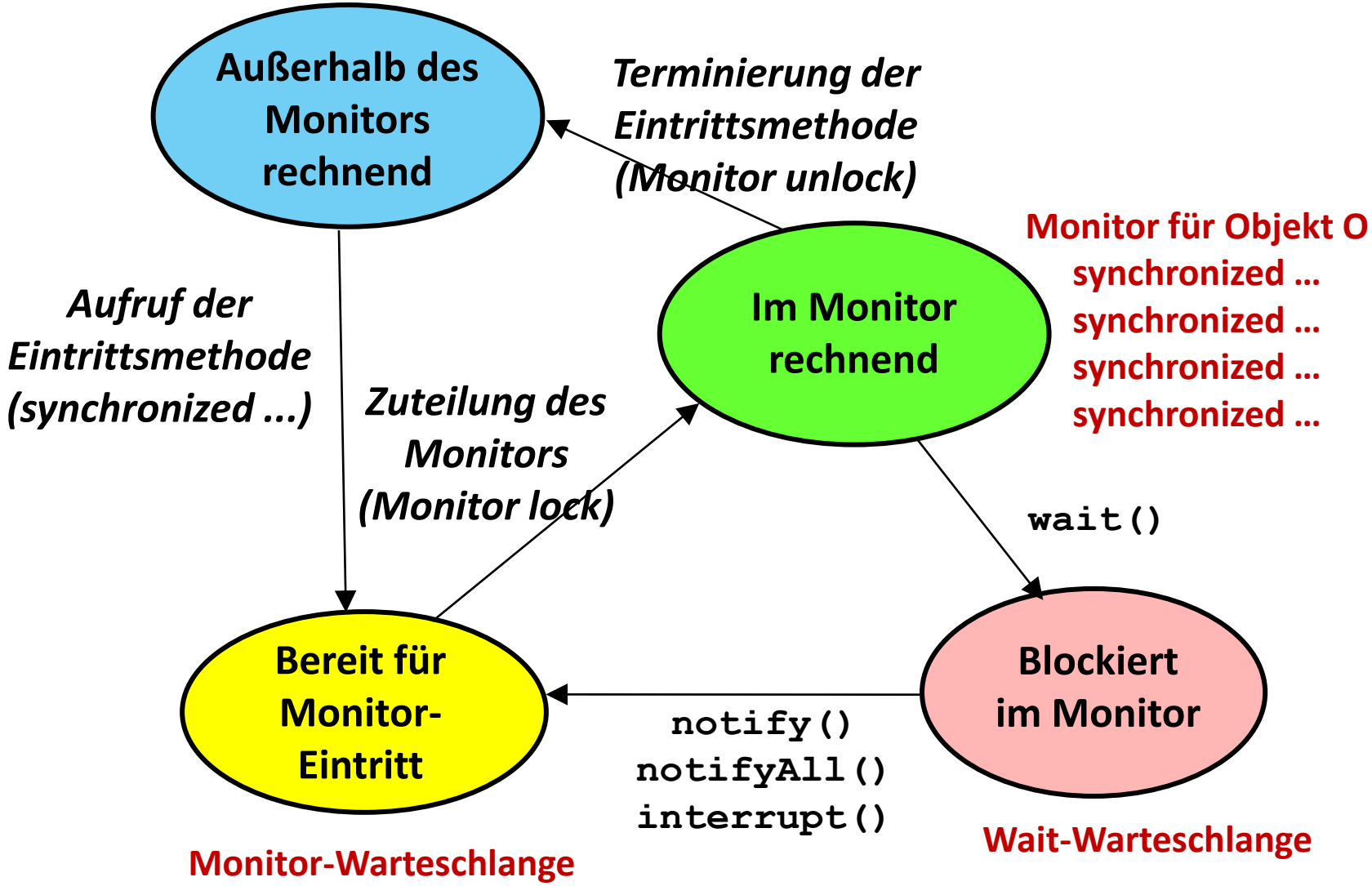
- Bisher gelöst: Wechselseitiger Ausschluss ➔ TestThread8
- Neue Problemstellung: Einhalten von Reihenfolgebedingungen
  - Ein Thread X befindet sich im kritischen Abschnitt (= in einem **`synchronized`**-Block/ -Methode = im Monitor)
  - Er kann den kritischen Abschnitt erst verlassen, nachdem ein anderer Thread im selben kritischen Abschnitt (Monitor) ein Ereignis ausgelöst hat

  ➔ Was tun???

JAVA-Monitorfunktionen

- Für die Signalisierung der Threads untereinander können die Methoden der Klasse **`Object`**:
  `wait()`, `notify()`, `notifyAll()` benutzt werden
- **`wait()`**: Monitor freigeben und in zusätzlicher wait-Warteschlange warten
- **`notify()`**: Einen (beliebigen) Thread in der wait-Warteschlange „wecken“
- **`notifyAll()`**: Alle Threads in wait-Warteschlange „wecken“
- Der Aufruf dieser Methoden muss aus dem Monitor heraus erfolgen (innerhalb einer **`synchronized`**-Methode oder eines **`synchronized`**-Blocks)!

JAVA-Synchronisation: Thread-Zustandsdiagramm



Wirkung eines wait()-Aufrufs durch Thread X

```
<Synchr-Objekt>.wait()
```

- Thread X muss sich im Monitor des synchronisierten Objekts befinden!
- Thread X gibt anschließend den Monitor frei und wartet in der wait-Warteschlange des synchronisierten Objekts
- Thread X nimmt erst wieder am „normalen“ Scheduling in der Monitor-Warteschlange teil, wenn eine der folgenden Bedingungen eintritt:
  - Ein anderer Thread ruft **`notify()`** auf und Thread X wird aus der wait-Warteschlange ausgewählt
  - Ein anderer Thread ruft **`notifyAll()`** auf
  - Ein anderer Thread ruft **`interrupt()`** für Thread X auf
- Wenn Thread X den Monitor wieder betreten darf, wird die Ausführung nach dem **`wait()`**-Befehl fortgesetzt

Wirkung eines notify()-Aufrufs durch Thread Y

```
<Synchr-Objekt>.notify()
```

- Thread Y muss sich im Monitor des synchronisierten Objekts befinden!
- Ein (beliebiger) Thread X aus der wait-Warteschlange des synchronisierten Objekts wird in die allgemeine Monitor-Warteschlange des synchronisierten Objekts gestellt
- Thread X konkurriert ggf. anschließend wieder mit anderen Threads um den Zugang zum Monitor
- Thread Y behält den Monitor solange, bis er beendet ist, d.h. er darf ungestört zuende rechnen („signal and continue“)
- Es können bei notify() keine Parameter (Bedingungen, „conditional variables“) übergeben werden!

Beispiel:
Eine eigene Semaphor-Implementierung in JAVA

```
public final class Semaphore {
   private int count;

   public Semaphore(int value) {
      count = value;    }

   public synchronized void P()
                        throws InterruptedException {
      while (count <= 0) {
            this.wait();
      }
      count--;    }

   public synchronized void V() {
      ++count;
      this.notify();    }
}
```

Beispiel: Erzeuger-/ Verbraucher-Problem in JAVA: Erzeuger-Code (Monitor-Lösung)

Erzeuger führen die `enter`-Methode der Klasse `BoundedBuffer<E>` aus:

```
public synchronized void enter(E item)
                        throws InterruptedException {
  while (buffer.size() == bufferMaxSize) {
    /* Puffer ist voll */
    this.wait();  // Ausführenden Thread blockieren
   }
  buffer.add(item);    // Datenpaket in den Puffer legen
  this.notifyAll();    // ggf. Verbraucher wecken
}
```

Beispiel: Erzeuger-/ Verbraucher-Problem in JAVA: Verbraucher-Code (Monitor-Lösung)

Verbraucher führen die `remove`-Methode der Klasse `BoundedBuffer<E>` aus:

```
public synchronized E remove()
                throws InterruptedException {
   E item;
   while (buffer.size() == 0) {
     /* Puffer ist leer */
     this.wait(); // Ausführenden Thread blockieren
    }
   /* Datenpaket aus dem Puffer holen */
   item = buffer.removeFirst();
   this.notifyAll();                  // ggf. Erzeuger wecken
   return item;
}
```

➔ Erzeuger-Verbraucher (Monitor)

Weitere Synchronisationsmöglichkeiten in JAVA

Vorteil: Flexiblerer Einsatz
Nachteil: kompliziertere Anwendung

Packages:

java.util.concurrent

- allg. Semaphore, Barrieren, ThreadPools, …

java.util.concurrent.atomic

- **`boolean compareAndSet(expectedValue,updateValue);`**
- *Atomare Veränderung von Variablenwerten*

java.util.concurrent.locks

- Locks (Mutexe) + Conditions (Bedingungen)
- *Implementierungsalternative zu* **`synchronize`** *und* **`wait/notify`**
- Idee: jede Condition hat ihre eigene Wait-Warteschlange („**Condition Queue**“)
- Methoden: ***`lock`*`.lock()` / ***`lock`*`.unlock()`**
  `condition.await()` / `condition.signal()`

➔ Erzeuger-Verbraucher (Condition Queues)

Anwendung des Erzeuger-/Verbraucher-Modells für JAVA-GUIs

- Der AWT-Event-Thread *("event dispatching thread")* ist ein *Hintergrund*-Thread, der **alle** Ereignisse *("events")* im Zusammenhang mit GUI-Objekten (AWT/Swing) ausführt
  (z.B. Fenster anzeigen/schließen, Behandlung von Mausklicks und Tastatureingaben, …)
- Auftragserteilung findet statt über die AWT-Event-Queue (ein Puffer zur Speicherung von Events)
  - durch "Event-Erzeuger" (das BS bei Maus-/Tastatur-Events oder andere Threads)
  - **Einziger** "Event-Verbraucher": der AWT-Event-Thread
    (→ Ausführen des "Listener"-Codes und Veränderungen von GUI-Objekten)

Swing und Threads: Synchronisation

- Problem: Wenn ein Anwendungsthread ein Swing-Objekt verändern möchte, muss dies synchronisiert werden *(sonst evtl. Konflikt mit dem AWT-Event-Thread)*
- Es ist aber kein synchronisierter Zugriff auf Swing-Elemente vorgesehen, weil nur der AWT-Event-Thread Änderungen vornehmen darf!
- Lösung: Beauftragung des AWT-Event-Threads durch den Anwendungsthread über die EventQueue *(mit synchronisiertem Puffer-Zugriff)*
  → Erzeuger-/ Verbraucher-System mit AWT-Event-Thread als einzigem Verbraucher!
- Spezieller Event-Typ für das Ausführen von Anwendungsthreads: `InvocationEvent`

Methoden zur Erzeugung eines InvocationEvent (Klasse: `java.awt.EventQueue`)

- **`static void invokeLater (Runnable runnable)`**

  Ein InvocationEvent wird der EventQueue hinzugefügt. Nach Abarbeitung aller vorherigen AWT-Events führt der AWT-Event-Thread die **`run`**-Methode des übergebenen Objekts aus

- **`static void invokeAndWait (Runnable runnable) throws InterruptedException,InvocationTargetException`**

  Wie **`invokeLater(..)`** Unterschied: Der aufrufende Thread wird blockiert (**`wait`**), bis der **`InvocationEvent`** abgearbeitet ist, und erst danach fortgesetzt

➔ TestThread9

Kapitel 3
Prozess-Synchronisation und –Kommunikation

1. Einführung und Grundlagen
2. Aktives Warten
3. Semaphore
4. Monitore
5. Message Passing
6. UNIX
7. Windows
8. Deadlocks

Inter Process Communication (IPC)

- IPC via „**Shared Memory**”
  - Kommunikation über privaten gemeinsamen Speicher
  - Benutzung gemeinsamer Datenbereiche wie Semaphore, Mutex, …
- IPC via „**Message Passing**”
  - Kommunikation über Nachrichtenaustausch
  - Die Nachrichten können lokal oder über ein Netzwerk gesendet werden (→ verteilte Systeme!)
  - Nachrichten dienen zum Datenaustausch
  - Mit Nachrichten kann auch gegenseitiger Ausschluss realisiert werden

Message Passing

- Die grundlegenden Funktionen im Zusammenhang mit Nachrichten sind:
  - `send (destination, &message)`
  - `receive (source, &message)`

- Wichtige Eigenschaften von Message Passing Systemen:
  - Synchronisationsverhalten
  - Adressierung

Message Passing: Synchronisationsverhalten

Fundamentale Bedingung: Der Empfänger kann eine Nachricht erst empfangen, nachdem der Sender sie abgeschickt hat

- **Blockierendes Senden:** Die **`send`**-Operation blockiert, bis der Empfang der Nachricht bestätigt wird.
- **Nicht-blockierendes Senden:** Die **`send`**-Operation ist beendet, sobald die Nachricht abgeschickt wurde.
- **Blockierendes Empfangen:** Wenn zuvor eine Nachricht gesendet wurde, wird diese empfangen und die Ausführung fortgesetzt. Ansonsten blockiert die **`receive`**-Operation, bis eine Nachricht eingetroffen ist.
- **Nicht blockierendes Empfangen:** Wenn zuvor eine Nachricht gesendet wurde, wird diese empfangen und die Ausführung fortgesetzt. Ansonsten wird die Ausführung mit einer entsprechenden Fehlermeldung fortgesetzt.

Adressierung (1)

Direkte Adressierung

- Sender:
  - der Sender kennt die Adresse des Empfängers P
    ```
    send(P,&message)   /* sende Nachricht an P */
    ```
- Empfänger:
  - Der Empfänger kennt die Adresse des Senders Q und wartet auf eine Nachricht dieses Senders
    ```
    receive(Q,&message)  /* empfange Nachricht von Q */
    ```
    oder
  - Der Empfänger wartet auf eine Nachricht und erkennt aus der Nachricht, wer der Sender war (Absenderinformation in der Message)
    ```
    receive(&message)  /* empfange Nachricht von
                          beliebigem Sender */
    ```

Keine Zwischenspeicherung notwendig!

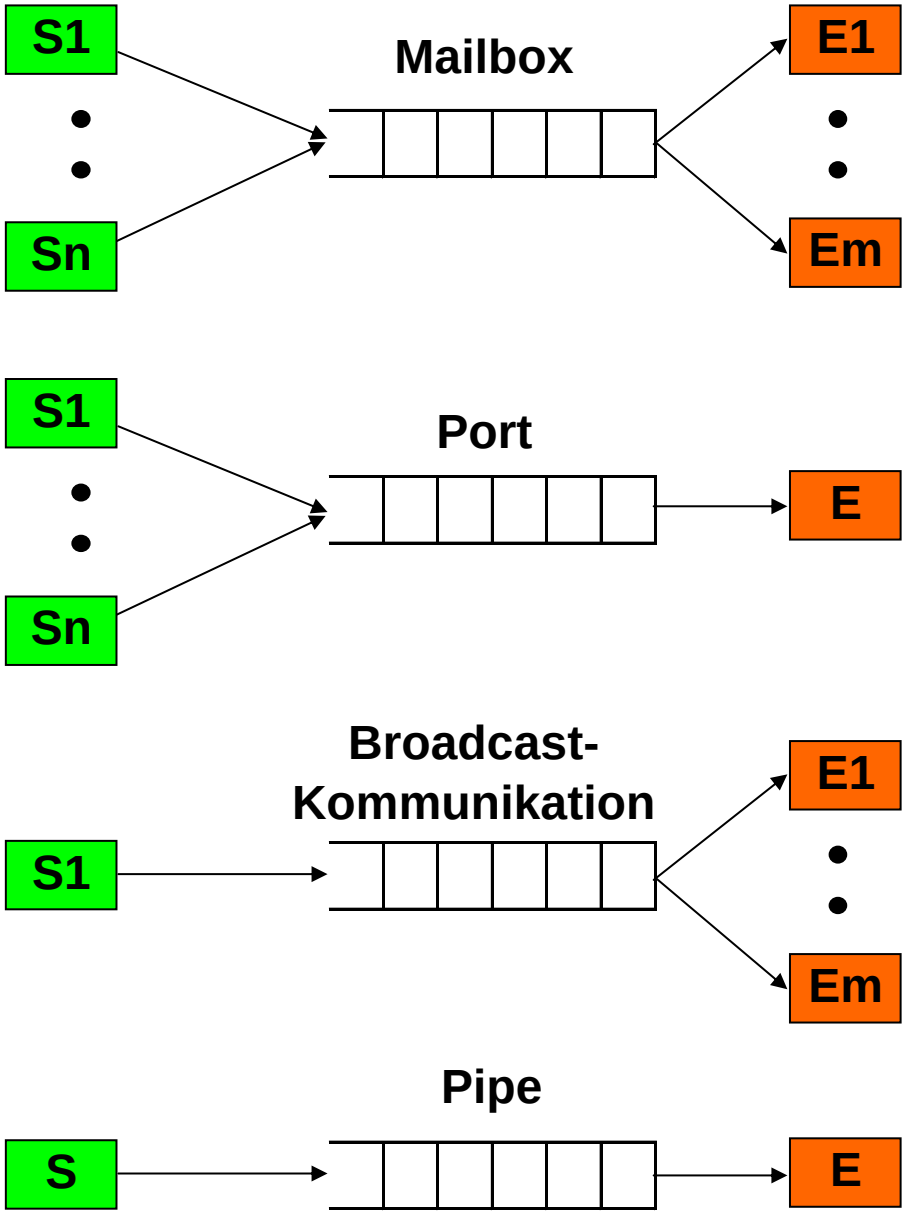
Adressierung (2)

Indirekte Adressierung („Message Queueing“)

- Nachrichten werden zu einer allgemein zugänglichen Warteschlange (Queue) gesendet
- Entkopplung von Sender und Empfänger
  - n Sender - m Empfänger (n:m – Beziehung): allgemeine Mailbox mit beliebigem Zugriff
  - n:1-Beziehung: nur ein Empfänger darf auf die Mailbox zugreifen („Port“) → typisch für Client-/ Serverkommunikation
  - 1:m-Beziehung: ein Sender sendet an viele Empfänger („Broadcast“-Kommunikation)
  - 1:1-Beziehung: ein Sender und ein Empfänger mit Warteschlange: „Pipe“ (Message = Bytestrom!)
- In allen 4 Fällen: Empfänger-Adressen müssen nicht bekannt sein!

Indirekte Adressierung: Übersicht

Implementierung über Erzeuger-/Verbraucher-Modell:
Sender = Erzeuger
Empfänger = Verbraucher
MessageQueue = Puffer



Beispiel: Pipes in JAVA

- `public class PipedInputStream extends InputStream`
- `public int read();`
  - liest genau ein Byte aus der Pipe
  - entspricht „receive“-Funktion
  - blockiert, wenn Pipe leer
- `public class PipedOutputStream extends OutputStream`
- `public void write(int b);`
  - schreibt genau ein Byte in die Pipe
  - entspricht „send“-Funktion
  - blockiert, wenn Pipe voll

➔ Beispiel-Code: Erzeuger-/ Verbraucherproblem in JAVA unter Verwendung von Pipes

Kapitel 3
Prozess-Synchronisation

1. Einführung und Grundlagen
2. Aktives Warten
3. Semaphore
4. Monitore
5. Message Passing
6. UNIX
7. Windows
8. Deadlocks

Synchronisations- und Kommunikationsmechanismen in UNIX: Überblick

- Signale
- Pipes
- Semaphore
- Shared Memory
- Message Queues
- Sockets

Kommunikation zwischen Prozessen!

UNIX: Signale

- Signale sind Software-Mechanismen, durch die Prozesse von eingetretenen Ereignissen („Events“) informiert werden.
- Gleichzeitig auftretende Signale werden ohne vorhersagbare Reihenfolge dem Prozess präsentiert.
- Signale können von einem Prozess zu einem anderen oder vom Kern intern versendet werden.
- Soll bei Empfang eines bestimmten Signals eine definierte Aktion stattfinden, so kann ein Prozess einen „Signal Handler“ definieren (SW-Interrupthandler).
- Das „Versenden“(Eintrag in den PCB) und „Empfangen“ (beim Prozesswechsel) von Signalen findet durch den Kern statt.

UNIX: Pipes

- Eine Pipe ist ein FIFO-Puffer, in den ein Prozess hineinschreiben und aus dem ein anderer Prozess lesen kann.
- Bei ihrer Erzeugung bekommt eine Pipe eine feste Länge in Bytes *(Linux: 4 KiB)*
- Ein schreibender Prozess wird sofort ausgeführt, wenn noch genügend Platz in der Pipe ist. Andernfalls wird der Prozess blockiert.
- Ein lesender Prozess wird dann blockiert, wenn er mehr Bytes lesen will, als aktuell in der Pipe sind.
- UNIX gewährleistet über gegenseitigen Ausschluss, dass nur ein Prozess zur Zeit auf die Pipe zugreifen kann.
- „Unnamed Pipes“ können nur von verwandten Prozessen benutzt werden (`pipe`-Systemaufruf), sonst: „Named Pipe“

UNIX: Semaphore

- Semaphor-Systemaufrufe sind allgemeine P- und V-Operationen
- Mehrere Operationen auf verschiedenen Semaphoren können gleichzeitig ausgeführt werden (Blockierung des Prozesses, wenn **eine** P-Operation warten muss)
- Die Semaphor-Systemaufrufe beziehen sich daher jeweils auf eine Semaphor-Gruppe.
- Das Inkrement bzw. Dekrement kann größer als Eins sein.
- Der UNIX-Kern sorgt dafür, dass Setzen und Zurücksetzen der Semaphore „atomar“ geschieht.

UNIX: Semaphor-Zugriffsfunktionen

- **Erzeugung einer Semaphor-Gruppe +**
  Zugriff auf eine Semaphor-Gruppe:
  `semid = semget (key, count, flag)`
  - key = definierter Schlüssel (long) oder Konstante IPC_PRIVATE
  - count = Anzahl Semaphore
  - flag: Neu erzeugen? (IPC_CREAT), sonst existierende ID
    + Setzen von Zugriffrechten
- **Kontrolloperationen: `semctl(semid, …)`**
  - Lesen / Setzen der Semaphorwerte, Löschen der Semaphore
- **P/V-Operationen: `semop(semid,&oplist,num_ops)`**
  - akzeptiert ein Array von Semaphor-Operationen, die jeweils durch eine Semaphor-Nummer und einen Wert sem_op spezifiziert sind
  - Der Wert von sem_op wird zur Semaphor-Zählvariablen hinzuaddiert!

UNIX: Shared Memory

- Die Kommunikation zwischen Prozessen findet über einen gemeinsamen Speicherbereich statt. Lesen und Schreiben sind normale Speicherzugriffe.
- Anlegen/Suchen einer „Shared Memory Region“ :
  - `shmid = shmget(key, size, flag)`
- Zuordnung zu einem Prozess durch Einbinden in den Adressraum („Shared Memory Attach“):
  - `virt_addr = shmat(shmid,&addr,flag)`
    - addr : Wunschadresse oder 0
    - virt_addr : wirkliche virtuelle Anfangsadresse
    - flag : z.B read/write oder read-only
- Für jeden Prozess kann festgelegt werden, ob er nur lesend oder auch schreibend zugreifen darf

UNIX: Message-Queues

- Mit dem **`msgget`**-Systemaufruf wird eine Message-Queue erzeugt oder der Zugriff erlangt (analog **`semget`** und **`shmget`**)
- Kommunikationsfunktionen:
  - Send: **`msgsnd (msgqid,&msg,length,flag)`**
    - msg: Struktur mit Message-Typ (integer) + Nutzdaten
  - Receive: **`msgrcv(msgqid,&msg,maxlength,type,flag)`**
    - Die Message-Queue wird nach einer Nachricht des angegebenen Message-Typs (**`type`**) durchsucht.
    - Wird keine passende Nachricht gefunden, so wird je nach Wert von **`flag`** der Empfänger blockiert oder **`msgrcv`** gibt eine Fehlermeldung zurück.

UNIX: Sockets

- Ein Socket ist eine Kommunikationsschnittstelle („Steckdose“), die durch einen Deskriptor repräsentiert wird und mit einem symbolischen Namen verbunden sein kann.
- Zweck: Zugang für Prozesse (Anwendungsschicht) zu tieferliegenden Kommunikationsprotokollen (z.B. TCP/IP)
- Grundlage: Client / Server-Modell
  - Server bieten im Netz jeweils einen speziellen „Dienst" an (warten auf Anfragen)
  - Clients benutzen diese Dienste

Zusatzinfos:
➔ JAVA-Klasse: **Socket** (Package: java.net)
➔ Sockets werden in der Vorlesung „Rechnernetze“ ausführlich behandelt!

Kapitel 3
Prozess-Synchronisation

1. Einführung und Grundlagen
2. Aktives Warten
3. Semaphore
4. Monitore
5. Message Passing
6. UNIX
7. Windows
8. Deadlocks

Synchronisations- und Kommunikationsmechanismen in Windows

- Synchronisationsobjekte:
  - Process
  - Thread
  - Event
  - Mutex
  - Semaphore
  - Waitable timers
  - …
- Benutzung von Synchronisationsfunktionen über **Windows-API** (Funktionsbibliothek)

Kommunikation zwischen Threads!

Windows-API: Wait-functions

- Verallgemeinertes Konzept: Die „Wait“-Funktionen prüfen, ob die spezifizierten Bedingungen vorliegen. Wenn nicht, wird der Thread blockiert.
- Die „Wait“- Funktionen kehren erst zurück, wenn ein angegebenes Synchronisationsobjekt (oder alle) im Zustand „**signalisiert**“ ist (sind) oder Timeout eintritt.
- Nach Rückkehr der wait-Funktion wird i.d.R. der Zustand des Synchronisationsobjekts verändert (ggf. wieder zu „**nicht-signalisiert**“).
- **`WaitForSingleObject/WaitForMultipleObjects`**
- **`SignalObjectAndWait`**
  - Mit SignalObjectAndWait kann gleichzeitig (atomar) ein anderes Objekt auf „signalisiert“ gesetzt werden.

Wait-functions Beispiele

Die **Implementierung** der "Wait"-Funktionalität ist abhängig vom Synchronisations-Objekt *(vgl. "Interface"-Konzept)*

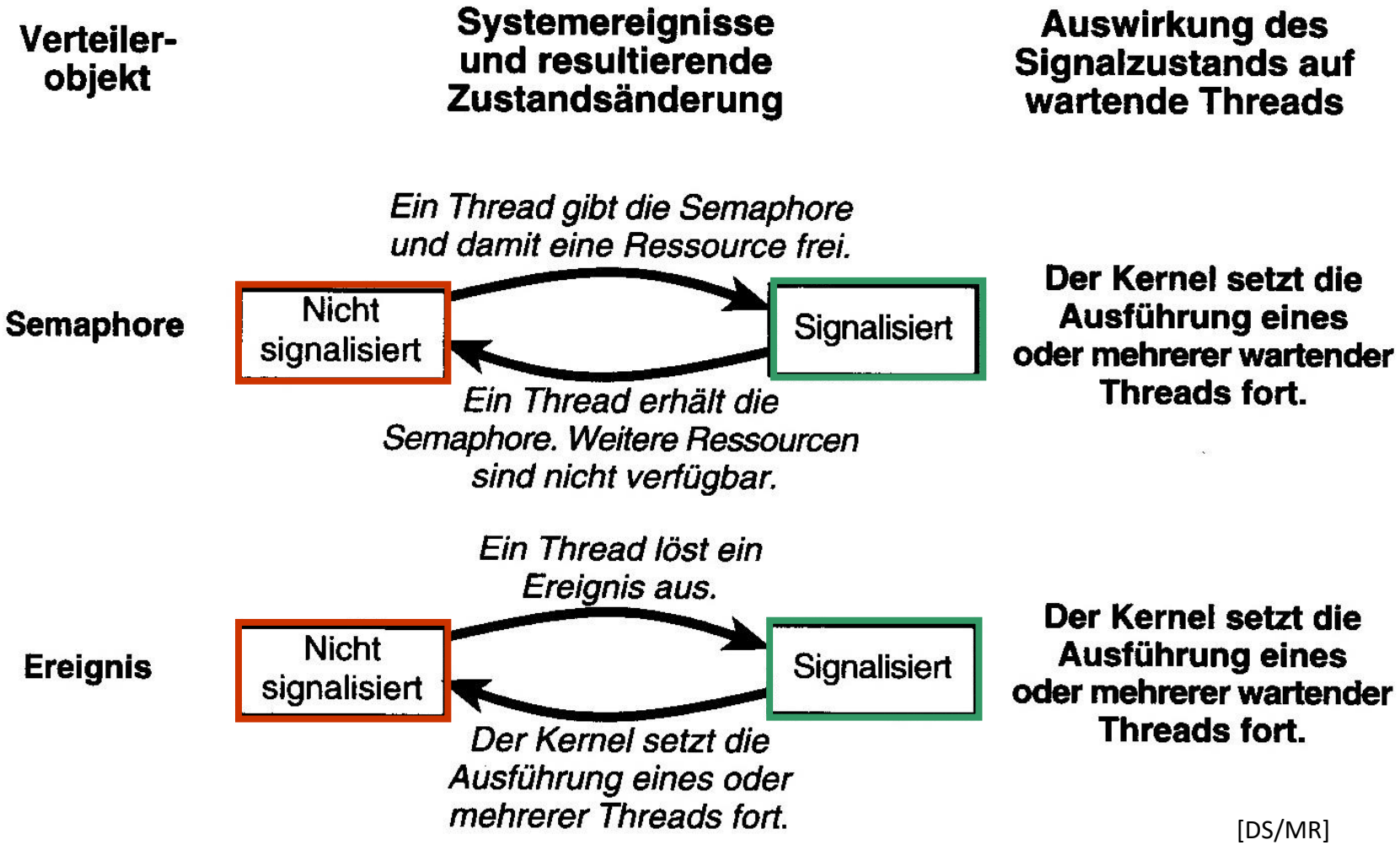
Beispiel Semaphore:

- **`WaitForSingleObject`** (bei Semaphor-Objekten = P-Funktion) dekrementiert den Semaphor-Zähler
- **`ReleaseSemaphore`** (= V-Funktion) inkrementiert den Semaphor-Zähler
- Beispielcode: WindowsSemaphoreTest.c

Beispiel Thread:

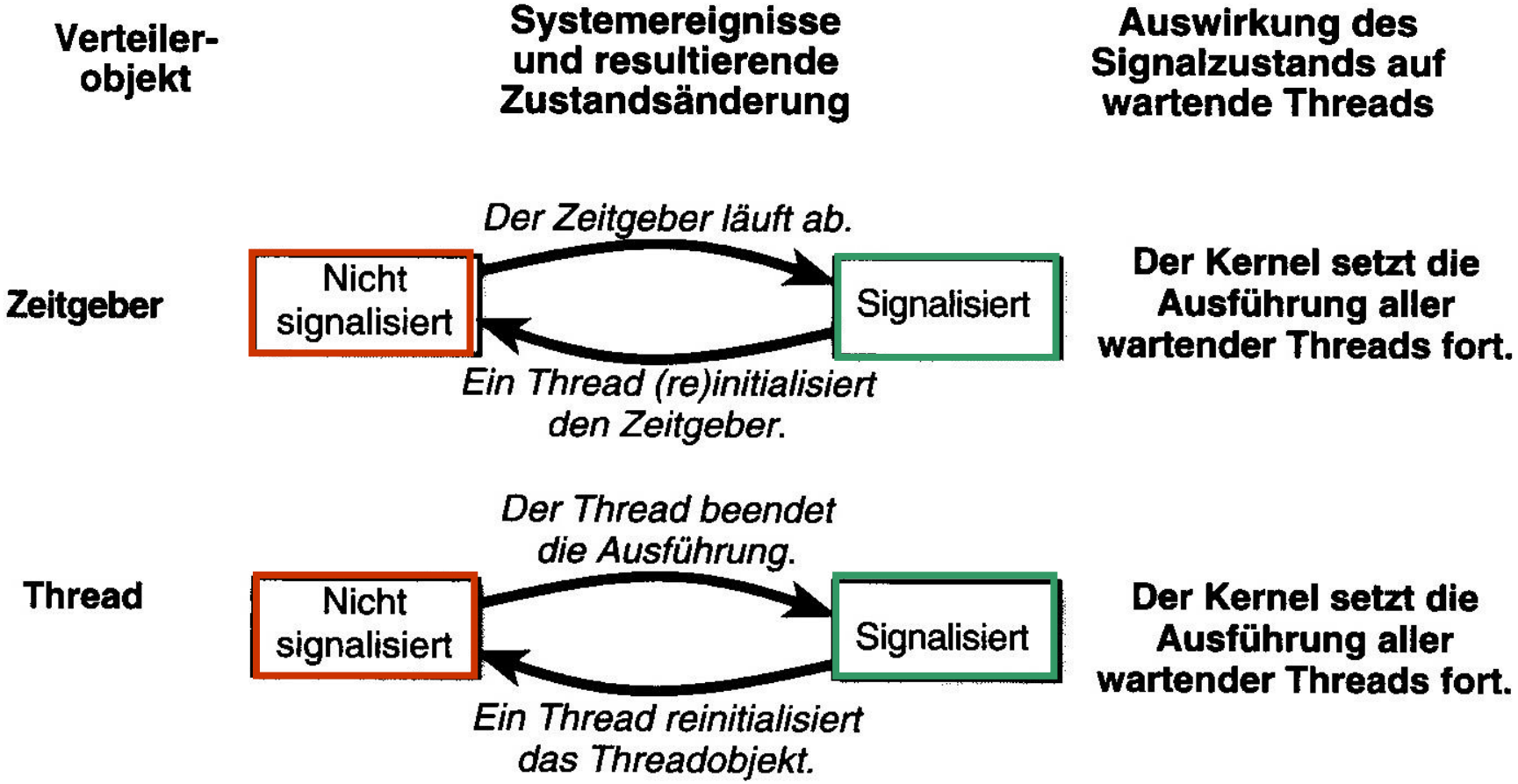
- **`WaitForSingleObject`** (bei Thread-Objekten = join-Funktion) wartet, bis der Thread beendet ist
- Beispielcode: WindowsThreadTest.c

Windows-Synchronisationsobjekte: Zustandsdiagramme (1)



[DS/MR]

Windows-Synchronisationsobjekte: Zustandsdiagramme (2)



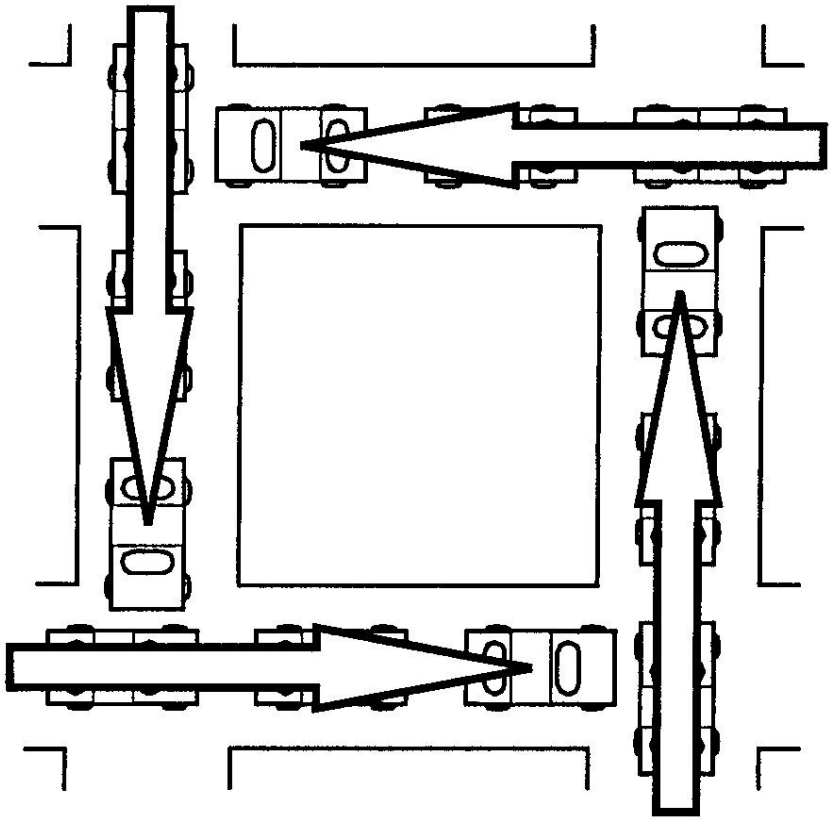
[DS/MR]

Kapitel 3
Prozess-Synchronisation

1. Einführung und Grundlagen
2. Aktives Warten
3. Semaphore
4. Monitore
5. Message Passing
6. UNIX
7. Windows
8. Deadlocks

Deadlocks (Verklemmungen)

Eine Menge von Prozessen befindet sich in einem **Deadlock-Zustand**, wenn jeder Prozess aus der Menge auf ein Ereignis wartet, das nur ein anderer Prozess aus der Menge auslösen kann.



[CV]

Beispiel: Möglicher Deadlock-Zustand

- **Prozess P1**

  Drucker.P

  Scanner.P

  Aufgabe bearbeiten

  Drucker.V

  Scanner.V

- **Prozess P2**

  Scanner.P

  Drucker.P

  Aufgabe bearbeiten

  Scanner.V

  Drucker.V

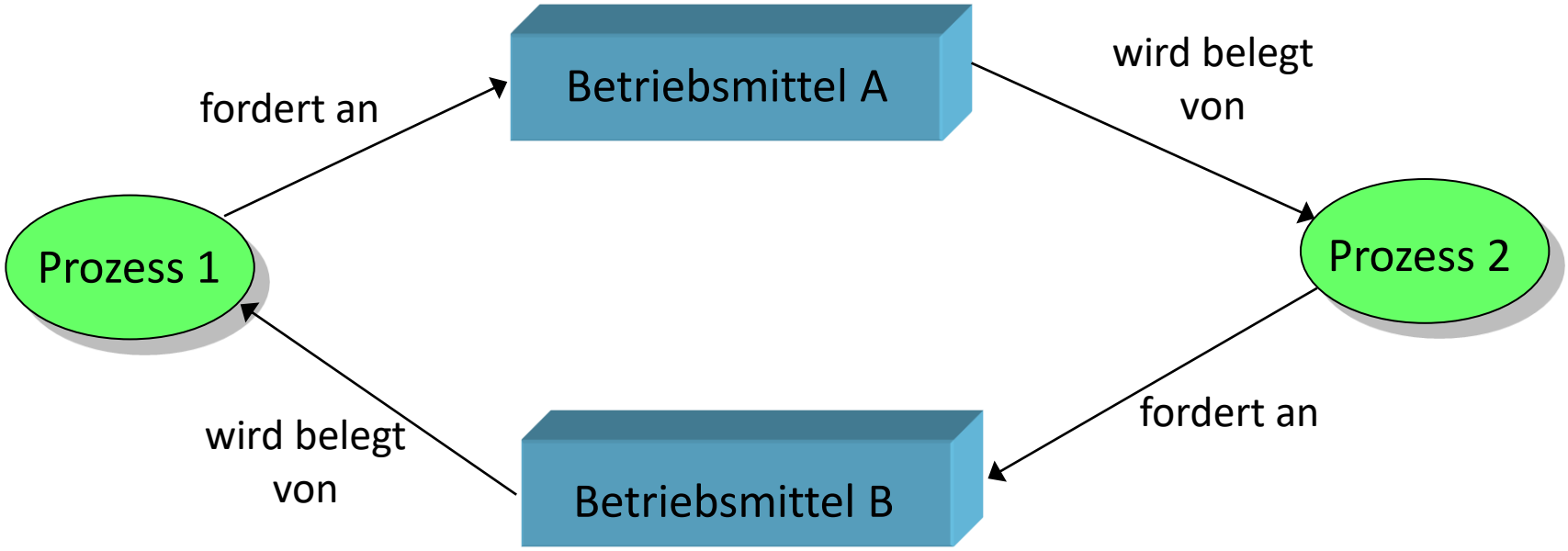
Notwendige Deadlock-Bedingungen

1. **Wechselseitiger Ausschluss**
   - Nur **ein** Prozess kann ein Betriebsmittel zu einem Zeitpunkt besitzen (exklusiver Besitz, exklusive Benutzung)
2. **Hold and Wait**
   - Ein Prozess, der bereits Betriebsmittel besitzt, kann noch weitere Betriebsmittel anfordern
3. **No Preemption**
   - Kein Betriebsmittel kann einem Prozess weggenommen werden (Betriebsmittel bleiben dem Prozess zugeordnet, bis er sie freiwillig abgibt)

Notwendige und hinreichende Deadlock-Bedingung

4. Cirular Wait

- Es existiert zusätzlich eine kreisförmig geschlossene Kette von Prozessen, bei der ein Prozess ein Betriebsmittel besitzt, auf das der nächste in der Kette wartet



Was kann gegen Deadlocks getan werden?

1. Deadlocks verhindern
   - Deadlock-Situationen aufgrund der Systemkonstruktion unmöglich machen
2. Deadlocks vermeiden
   - Deadlock-Situationen durch Überwachung zur Laufzeit vermeiden
3. Deadlocks entdecken und nachträglich beheben
   - Deadlock-Situationen zulassen und auflösen
4. Betriebssystem-Heuristiken einsetzen
   - Anwendungsspezifischen Algorithmen die Feinarbeit überlassen

1. Deadlock-Verhinderung

Zur Verhinderung von Deadlocks muss sichergestellt werden, dass **mindestens eine** der Deadlock-Bedingungen nicht erfüllt ist:

- Entweder „Kein wechselseitiger Ausschluss“
  - Betriebsmittel können nicht exklusiv reserviert werden
- oder „Kein Hold and Wait“
  - ein Prozess darf nur dann Betriebsmittel anfordern, wenn er **keine anderen** hat → jeder Prozess muss **alle** benötigten Betriebsmittel **vor** seiner Ausführung anfordern
- oder „Preemption“
  - Prozesse müssen die Betriebsmittel auf Anforderung zurückgeben
- oder „Kein Circular Wait“
  - Alle Betriebsmittel stehen in einer Reihenfolge (Nummerierung) und dürfen nur in dieser Reihenfolge angefordert werden

→ **Starke Einschränkungen im Multiprogramming!!**

2. Deadlock-Vermeidung

- Bei jeder Betriebsmittelanforderung wird überprüft, ob diese Anforderung potentiell zu einem Deadlock führen kann
  - Der Prozess, der Betriebsmittel so anfordert, dass ein Deadlock möglich ist, wird nicht gestartet
  - Das Betriebsmittel, das ein Prozess anfordert und dessen Zuweisung einen Deadlock möglich macht, wird dem Prozess nicht gegeben
- Es ist Wissen über die zukünftigen Betriebsmittel - Anforderungen der Prozesse notwendig!

Deadlock-Vermeidung: Bankier-Algorithmus (Dijkstra 1965)

- Der Bankier-Algorithmus überprüft bei Anforderung eines Betriebsmittels jeweils die Deadlock-Möglichkeit
- Der aktuelle Zustand ist der Zustand des Systems mit den zur Zeit den Prozessen zugewiesenen Betriebsmitteln
- Ein *sicherer Zustand* ist ein Zustand, bei dem es mindestens einen Ablauf gibt, der alle Prozesse beendet und zu keinem Deadlock führt
- Der Bankier-Algorithmus überprüft, ob mit der Vergabe des angeforderten Betriebsmittels ein unsicherer Zustand entsteht – dann wird das Betriebsmittel nicht gewährt

Deadlock-Vermeidung: Anwendung in der Praxis

- Probleme:
  - Prozesse wissen selten im voraus, wieviele Betriebsmittel sie brauchen werden
  - Freie Betriebsmittel können evtl. wegen eines potentiell unsicheren Zustand nicht genutzt werden
  - Prozesse werden zu unterschiedlichen Zeiten gestartet und terminiert
  - Betriebsmittel können verschwinden (z.B. USB-Geräte)

➔ **Geringe praktische Anwendungsmöglichkeit!**

3. Deadlock-Entdeckung und -Behebung

- Deadlock-Entdeckung
  - Analog Bankier-Algorithmus
- Deadlock-Behebung
  - einen oder mehrere Prozesse abbrechen und Betriebsmittel freigeben
  - einem oder mehreren Prozessen zwangsweise Betriebsmittel entziehen
  - System neu starten („Reboot tut gut!“)

➔**Geringe praktische Anwendungsmöglichkeit!**

4. Betriebssystem-Heuristiken einsetzen

- Keine allgemeine Betriebssystem-Strategie für Deadlocks einsetzen, sondern „Daumenregeln“ *(Heuristiken)*
  - Zugriff auf Betriebsmittel nur durch spezielle Prozesse (z.B. Drucker-Spooler)
  - Ressourcen nur zuweisen, wenn es unbedingt nötig ist (Belegungszeiten minimieren)
  - …
- Angepasste Algorithmen für spezielle Anwendungen verwenden
  - Datenbanken: Two-Phase-Locking
  - …

Ende des 3. Kapitels: Was haben wir geschafft?

3. Prozess-Synchronisation

- 3.1 Einführung und Grundlagen
- 3.2 Aktives Warten
- 3.3 Semaphore
- 3.4 Monitore
- 3.5 Message Passing
- 3.6 UNIX
- 3.7 Windows
- 3.8 Deadlocks