

# Rechnernetze und Betriebssysteme

## 3. Synchronisation

Prof. Dr. Martin Becke  
SS2015 – Version 1.0

# 1. 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.

## 2. 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**



# 3. Race conditions

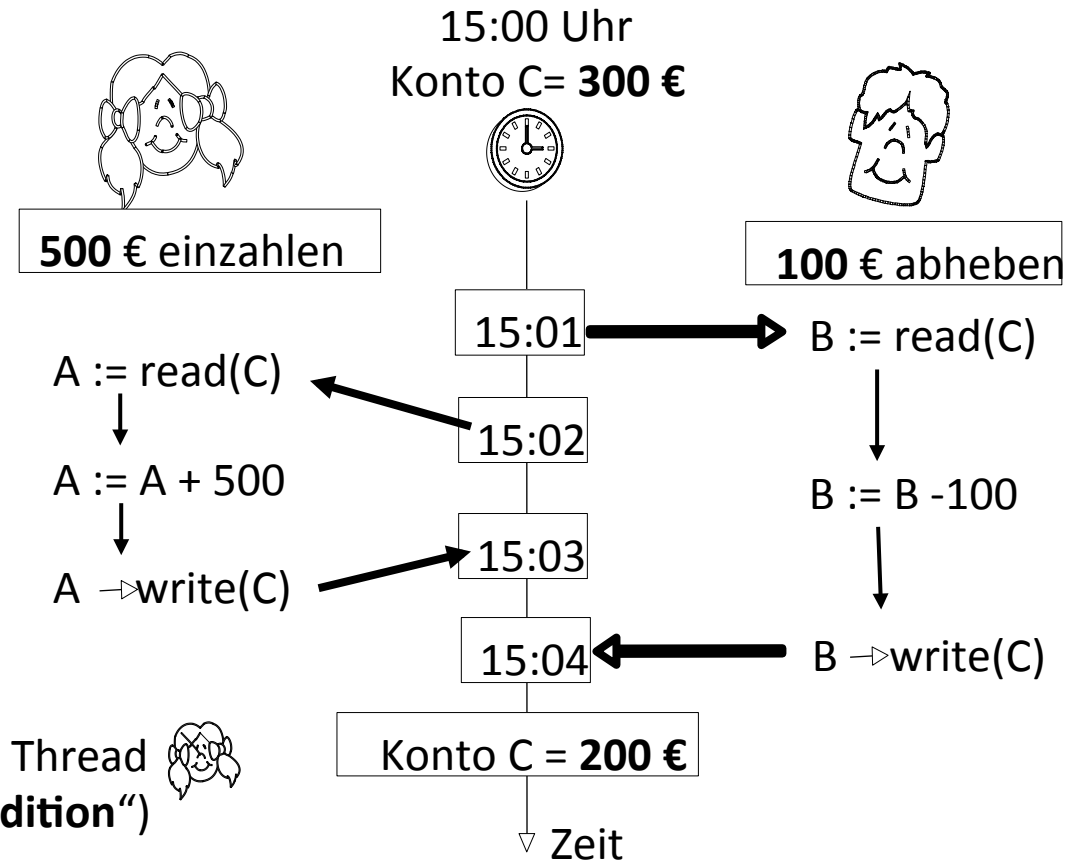
## Beispiel: Kontoführung

### Paralleler Zugriff auf globale Variable C mit 2 Threads

- Parallele Prozesse von vielen Benutzern arbeiten gleichzeitig auf den Daten.
- Falls **gleichzeitig** zwei Benutzer auf das **gleiche Konto** einzahlen möchten, so kann es sein, dass der Kontostand hinterher nicht stimmt.

Ein solcher Fehler wird von keinem Kunden toleriert!

Eine Fehlbuchung tritt nur auf, wenn Thread  den Thread  überholt („**race condition**“)



## 4. 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): Befehlsfolgen in “kritischen Abschnitten” dürfen nicht unterbrochen werden
  - Nur ein Prozess zur Zeit darf einen kritischen Abschnitt betreten
  - *Beispiel: Nur ein Prozess zur Zeit darf eine Buchung durchführen*

# 5. Anforderungen an 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 (fairness condition)
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“)

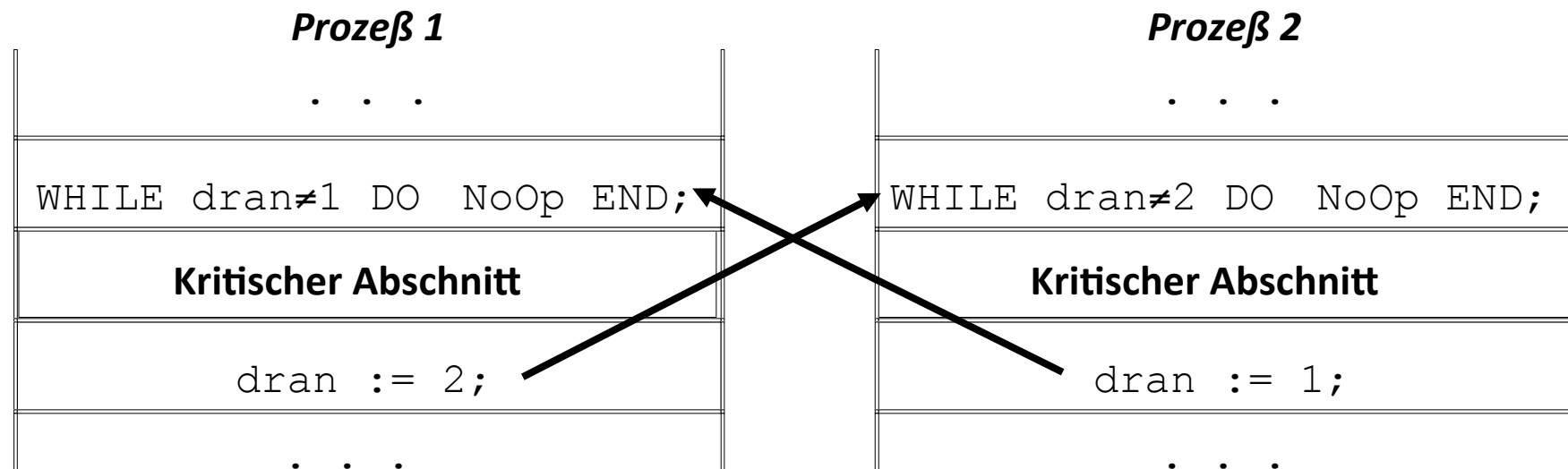
***(Dijkstra 1965)***

# 6. Synchronisierung

## Naiver Ansatz:

*mutual exclusion mit globaler Variablen*

Wartesperre errichten, bis kritischer Abschnitt frei      Initial dran=1



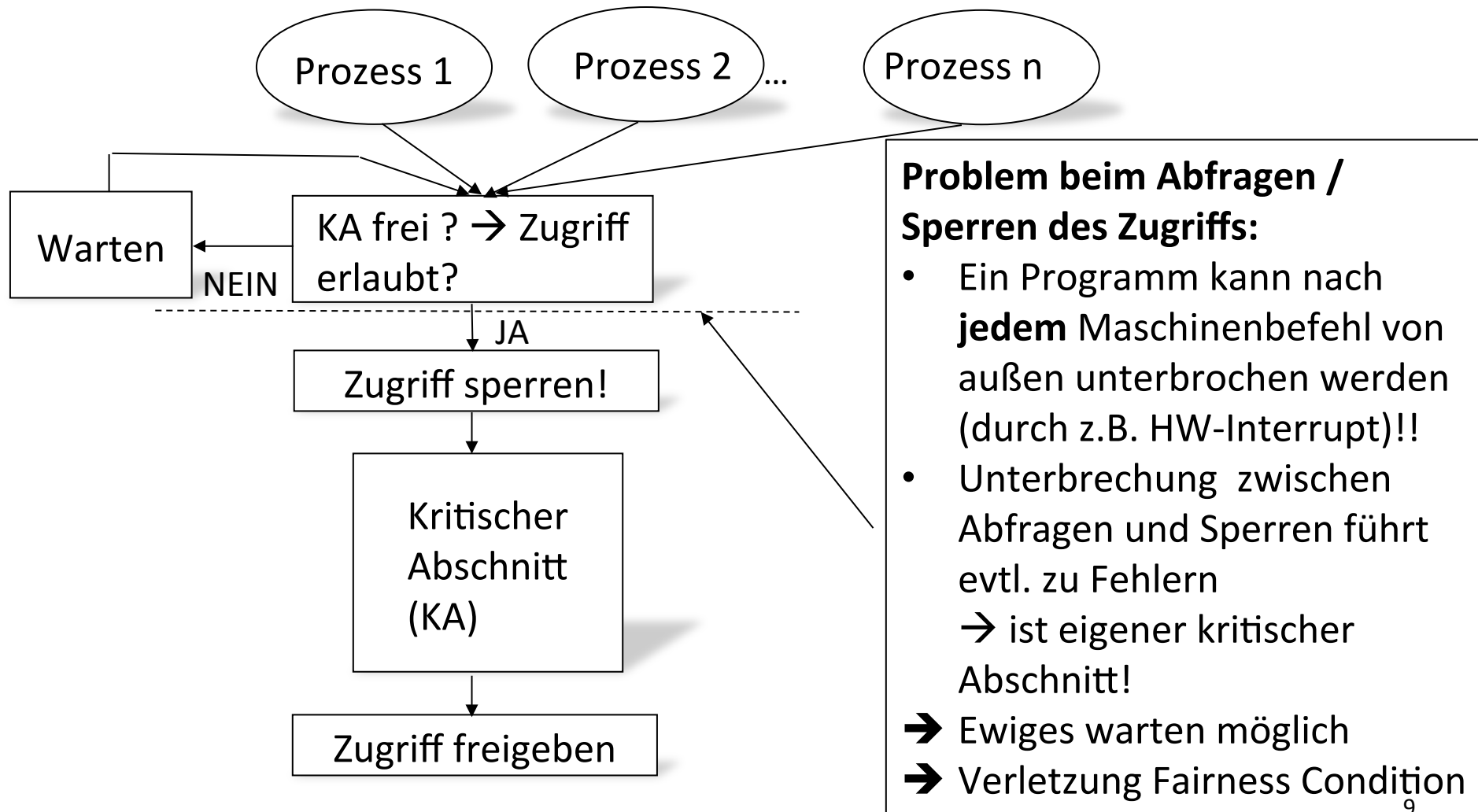
**Diese Herangehensweise wird auch „Aktives Warten“ genannt! (busy wait)**

## 7. 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“



# 8. Spinlock



## 8. Einfache Hardwarelösung: Ausschalten von Interrupts

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

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

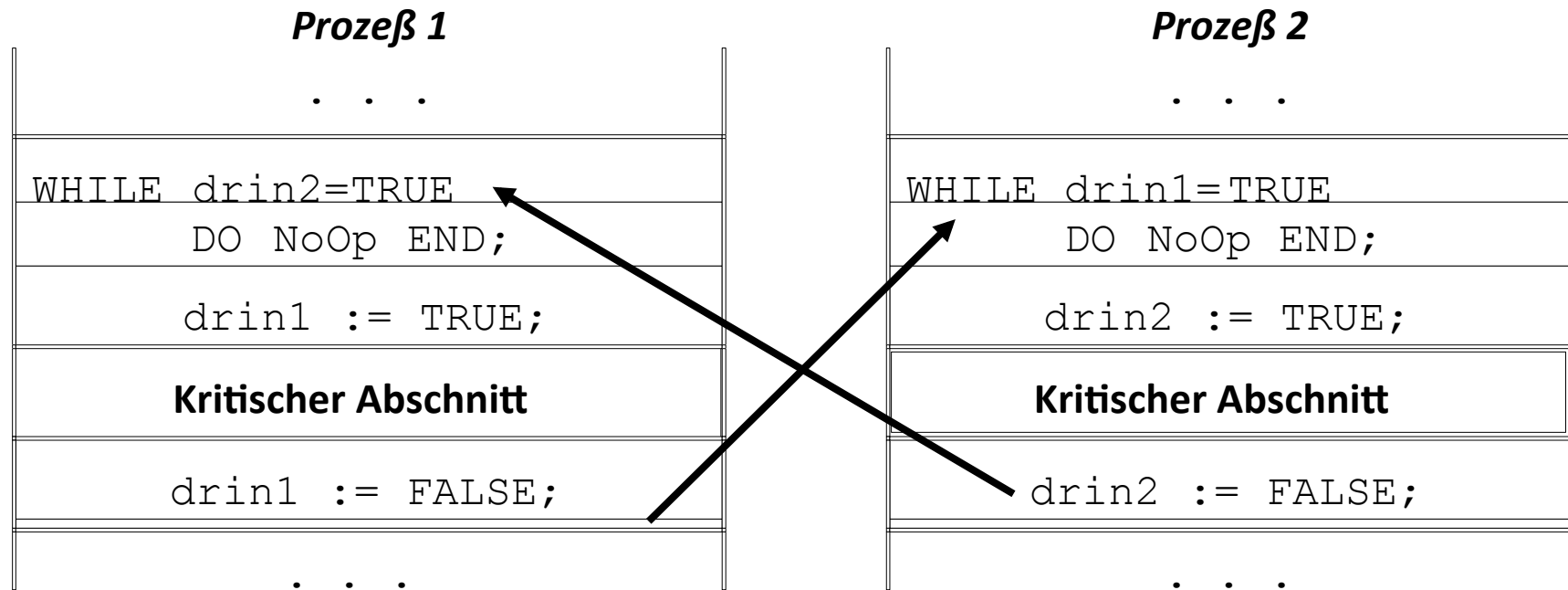
- Aber:
  - Funktioniert nur bei Einprozessoren
  - Es geht nur im Kernel-Mode
  - Interrupts könnten verloren gehen
  - Erfüllung der Anforderungen??

**Nicht praktikabel!!!**

# 9. Synchronisierung

## Verbesserung:

Zugang nur blockiert beim Aufenthalt in krit. Abschnitt



**Problem:** Initialisierung

drin1=drin2=False  $\Rightarrow$  keine *mutual exclusion*!

drin1=True, drin2=False  $\Rightarrow$  P2 ist blockiert ohne Sinn

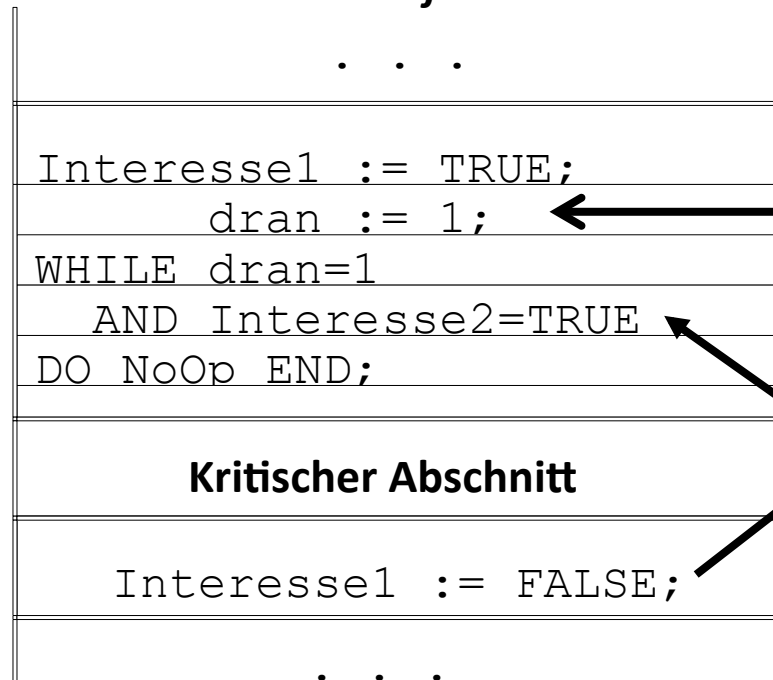
# 10. Aktives Warten (Lösung)

*Peterson 1981*

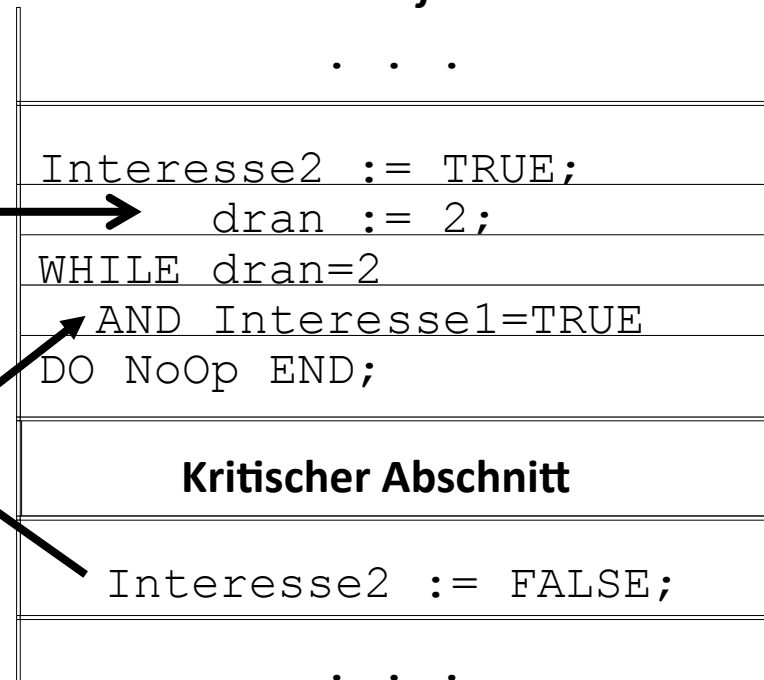
**Verbesserung:** - eine globale Variable,  
- zwei Prozeß-Zustandsvariable.

Initial `Interessel = Interesse2 = FALSE`

**Prozeß 1**



**Prozeß 2**



# 9. Diskussion: Aktives Warten

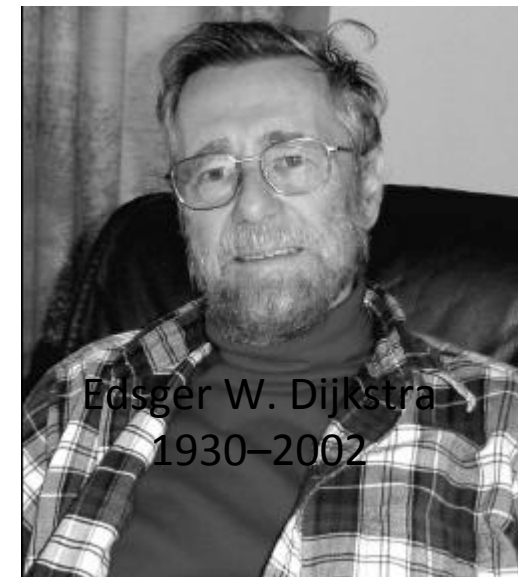
- Vorteil:
    - Es gibt eine allgemeine Softwarelösung, welche die Anforderungen erfüllt (Peterson 1981)
  - Nachteile:
    - **Aktives Warten verschwendet unnötig Prozessorzeit**
    - Anzahl der Prozesse muss vorher bekannt sein
      - Sonst „Fainess condition“ in Gefahr
    - Aufwändige Programmierung
- ➔ **Ziel: Generalisierte Form und bessere Nutzung der Prozessorzeit**

# 10. Aktives Warten mit Liste

- Idee Verwaltung der Prozesse in einer Datenstruktur
- Die Datenstruktur besteht aus einem Zähler und einer Prozessliste
  - Struktur wird als Semaphore bezeichnet
  - Zur Initialisierung ist Zähler 0 und die Liste leer

# 11. 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 ("aufweckt")
- Semaphore werden i.d.R. vom Betriebssystem zur Verfügung gestellt



Edsger W. Dijkstra  
1930–2002

## 12. Passives Warten mit Semaphore

- Idee Verwaltung der Prozesse in einer Datenstruktur
- Die Datenstruktur besteht aus
  - einem Zähler und
  - einer Prozessliste
- ➔ Liste der Prozesse erlaubt passives warten.
  - Prozess wartet nicht aktiv auf Ressource
  - Prozess wird aktiviert, wenn Ressource frei



## 13. 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.



# 13. Semaphore - Struktur

- Eine Integer-Zählvariable  $s \geq 0$  ( $\rightarrow$  Anzahl freier Körbe)
- Eine Warteschlange für Prozesse ( $\rightarrow$  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 ( $\rightarrow$  Korb vom Stapel nehmen – falls vorhanden – und Geschäft betreten).
  - **V(S)** (“Verlassen”) teilt das Verlassen des kritischen Abschnitts mit, daher  $S++$  und weiter ( $\rightarrow$  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 ( $\rightarrow$  Irgendein Kunde aus der Warteschlange darf passieren und sich den Korb nehmen).

# 13. Semaphore- Implementierung

- Ein in einer P-Operation wartender Prozess ist im „Blockiert“-Zustand, bis Wecken durch ein Signal / Ereignis erfolgt → **kein** aktives Warten nötig!
- $P(S)$  und  $V(S)$  bilden gemeinsam einen eigenen kritischen Abschnitt
- $P$  und  $V$  operieren auf  $S$
- Implementierung der Nicht-Unterbrechbarkeit von  $P$  und  $V$  mittels Hardwareunterstützung im OS durch atomare Operationen

# 14. HW- Implementierung: Atomare Aktionen

- **Atomare Instruktionsfolgen** (können nicht unterbrochen werden)

- **Test And Set** (*test and set lock*)

```
PROCEDURE TestAndSet(VAR target:BOOLEAN): BOOLEAN
VAR tmp:BOOLEAN;
    tmp:=target;    target:= TRUE;    RETURN tmp;
END TestAndSet;
```

- **Swap**

```
PROCEDURE swap(VAR source,target: BOOLEAN)
VAR tmp:BOOLEAN;
    tmp:=target; target:=source; source:=tmp;
END swap;
```

- **Fetch And Add**

```
PROCEDURE fetchAndAdd(VAR a, value:INTEGER): INTEGER
VAR tmp:INTEGER;
    tmp:=a; a:=tmp+value;    RETURN tmp;
END fetchAndAdd;
```

# 13. Passives Warten mit Semaphor

## Software Pseudo-Code

## Datenstruktur

```
PROCEDURE P (VAR s:Semaphor)
```

```
BEGIN
```

```
  s.value:=s.value-1;
```

```
  IF s.value < 0 THEN
```

```
    einhängen(MyID,s.list); sleep;
```

```
  END;
```

```
END P;
```

```
PROCEDURE V (VAR s:Semaphor)
```

```
VAR PID: ProcessId;
```

```
BEGIN
```

```
  IF s.value < 0 THEN
```

```
    PID:=aushängen(s.list); wakeup(PID);
```

```
  END;
```

```
  s.value:=s.value +1;
```

```
END V;
```

```
TYPE Semaphor =
```

```
  RECORD
```

```
    value: INTEGER;
```

```
    list : ProcessList;
```

```
  END;
```

```
Initial: s.value:=1
```

# 15. Aktives Warten mit TestAndSet

```
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 */
```

# 16. Beispiel-Lösung mit JAVA und Semaphoren

**Voraussetzung: Klasse „Semaphore“**

**Package: java.util.concurrent**

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

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

## 52. Beispiel:

### Eine eigene Semaphor-Implementierung in JAVA

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

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

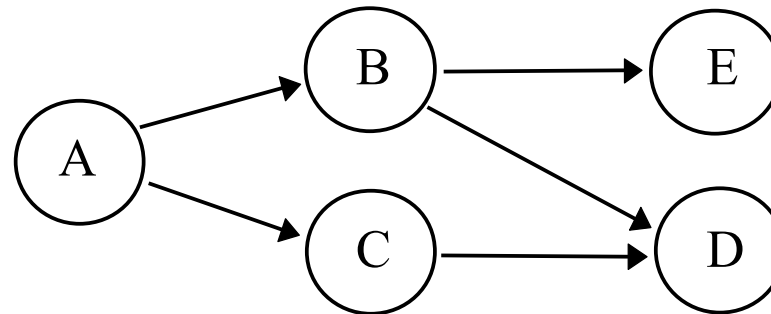
    public synchronized void P()
        throws InterruptedException {
        while (count <= 0) {
            this.wait();
        }
        count--;    }

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



# 17. Beispiel Synchronisation Prozesse

- Gemeinsamer Zugriff auf Ressourcen ist das eine (both write, both read), bildet aber keine Abhängigkeit wieder (create, consume)
- Beispiel:



Versuch mit Semaphore:  
(Globale Variable b,c,d1,d2,e mit 0 initialisieren.)

# 17. Beispiel Synchronisation Prozesse



**PROCESS A: TaskBodyA; V(b) ; V(c) ; END A;**

**PROCESS B: P(b) ; TaskBodyB; V(d1) ;V(e) ; END B;**

**PROCESS C: P(c) ; TaskBodyC; V(d2) ; END C;**

**PROCESS D: P(d1) ; P(d2) ; TaskBodyD; END D;**

**PROCESS E: P(e) ; TaskBodyE; END E;**

## 18. Binäres Semaphor („Mutex“)

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

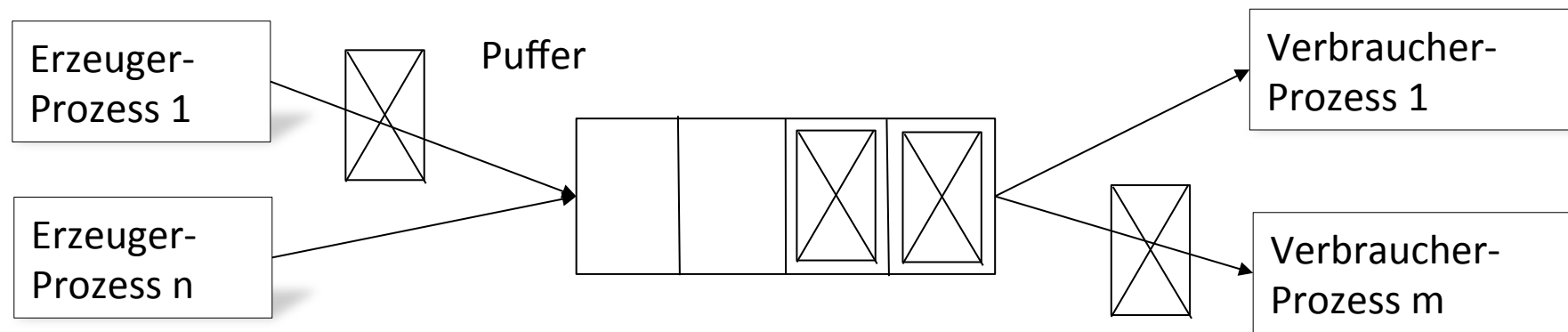
```
P(S):      while (S == 0 ) {  
              <blockieren und warten>; /* bis S == 1 */  
            }  
            S = 0;
```

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

# 18. Binäres Semaphor („Mutex“)

- 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 Bezeichnungen:  $P(S) \equiv \text{lock}(S)$ ,  $V(S) \equiv \text{unlock}(S)$

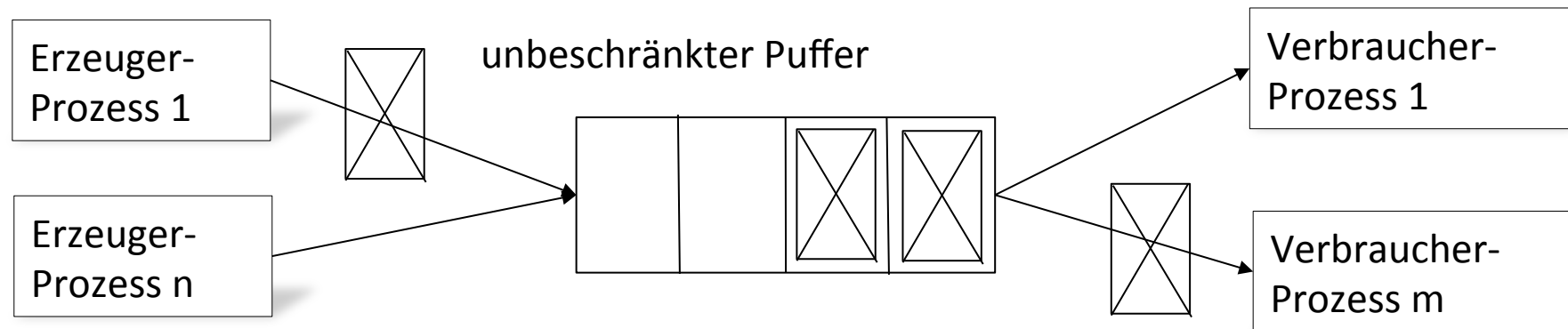
# 18. Beispiel Mutex bei Verwaltung von Pufferspeichern



## 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)

# 19. 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)** ( $B++$ )

## 20. Lösung: Erzeuger-Verbraucher-Problem mit unbeschränktem Puffer

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

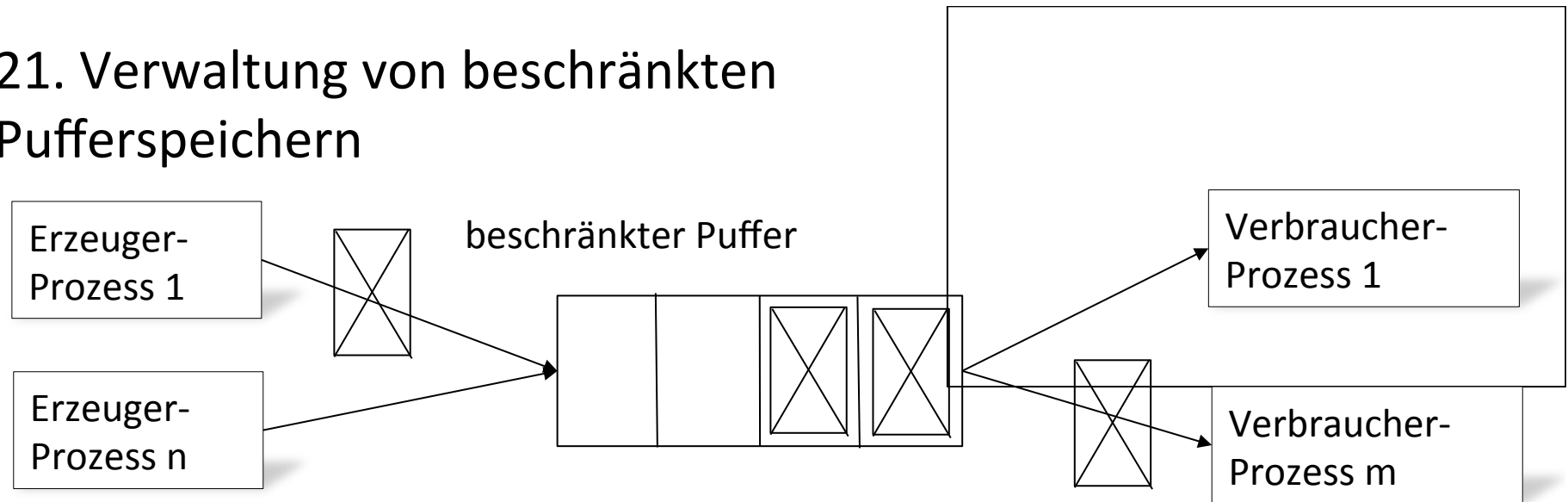
### Erzeuger

```
<Erzeuge Element e>;  
P(S); /* lock */  
<speichere e im Puffer>  
V(S); /* unlock */  
V(B); /* release */
```

### Verbraucher

```
P(B); /* acquire */  
P(S); /* lock */  
<nimm e aus Puffer>  
V(S); /* unlock */  
<verarbeite e>;
```

## 21. 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)** ( $B++$ )



## 36. 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

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

### Verbraucher

```
P(B);    /* acquire */  
P(S);    /* lock */  
<nimm e aus Puffer>  
V(S);    /* unlock */  
V(F);    /* release */  
<verarbeite e>;
```

# 37. Beispiel-Lösung

## Erzeuger-Verbraucher- Problem mit JAVA und Semaphoren



Voraussetzungen:

- Klasse „Semaphore“  
Package: `java.util.concurrent`
- Klasse „ReentrantLocks“ (Mutex-Implementierung)  
Package: `java.util.concurrent.locks`

# 38. Erzeuger-/ Verbraucher-Simulation: Entwurf

- **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.

### 39. 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  
}
```

## 40. 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;  
}
```

# 41. Monitore

- **Problem: Abhängigkeiten konnten bisher nicht dargestellt werden**
  - **Keine** syntaktische Kapslung der Bereiche
  - 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
  - Nicht Implementiert in C oder C++



Sir C.A.R. Hoare



Per Brinch Hansen

## 42. Was ist ein „Monitor“?

- Ein **Monitor überwacht den Aufruf bestimmter Methoden**
- Ein Thread „betritt“ den überwachten Monitorbereich (*kritischen Abschnitt*) 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 (*automatische 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*)

# 43. 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!



# 44. Wechselseitiger Ausschluss über Monitor

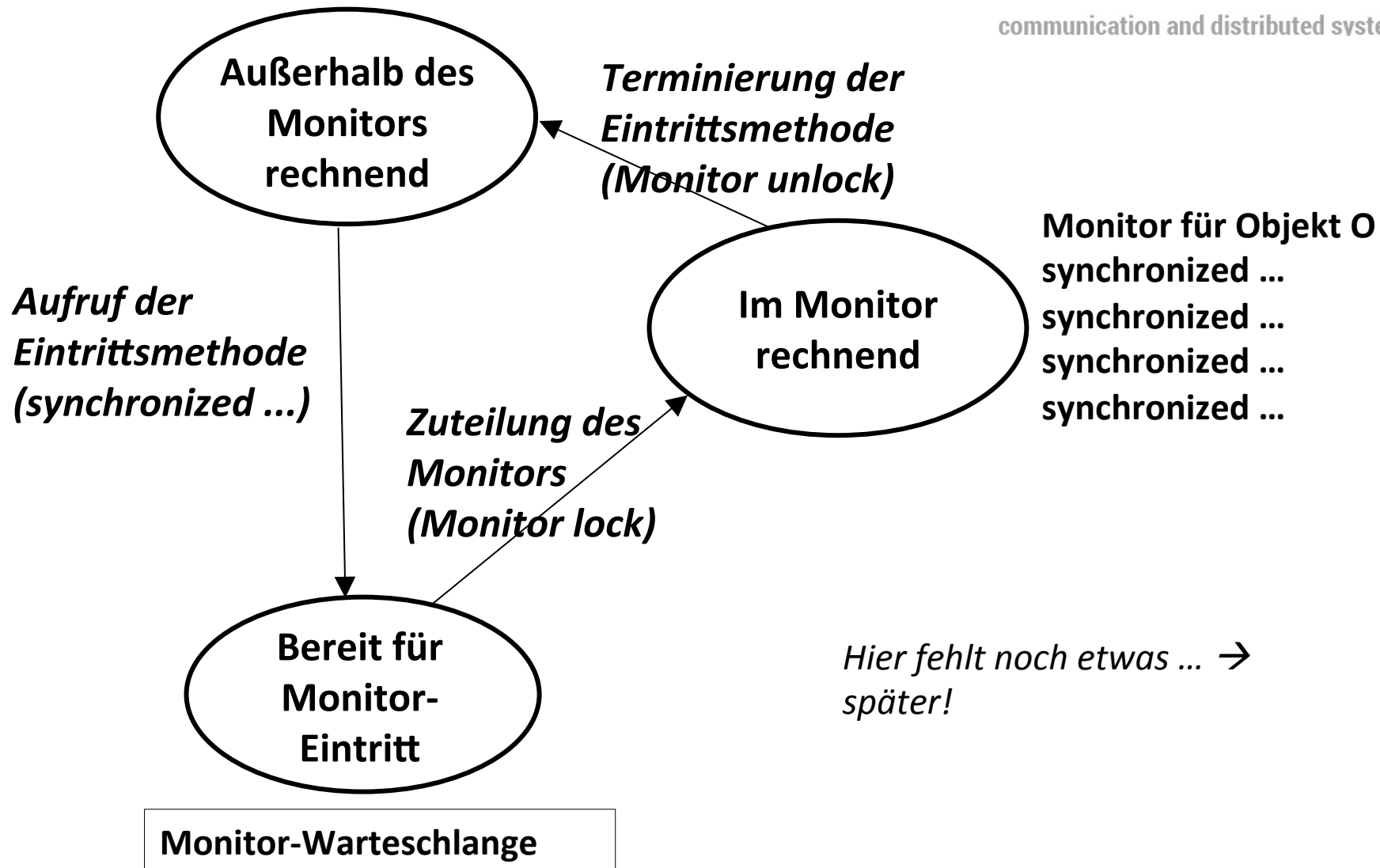
Monitor-Lösung :

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

Im Vergleich: Semaphore-Lösung

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

## 45. JAVA-Synchronisation: Thread-Zustandsdiagramm



## 46. 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*)

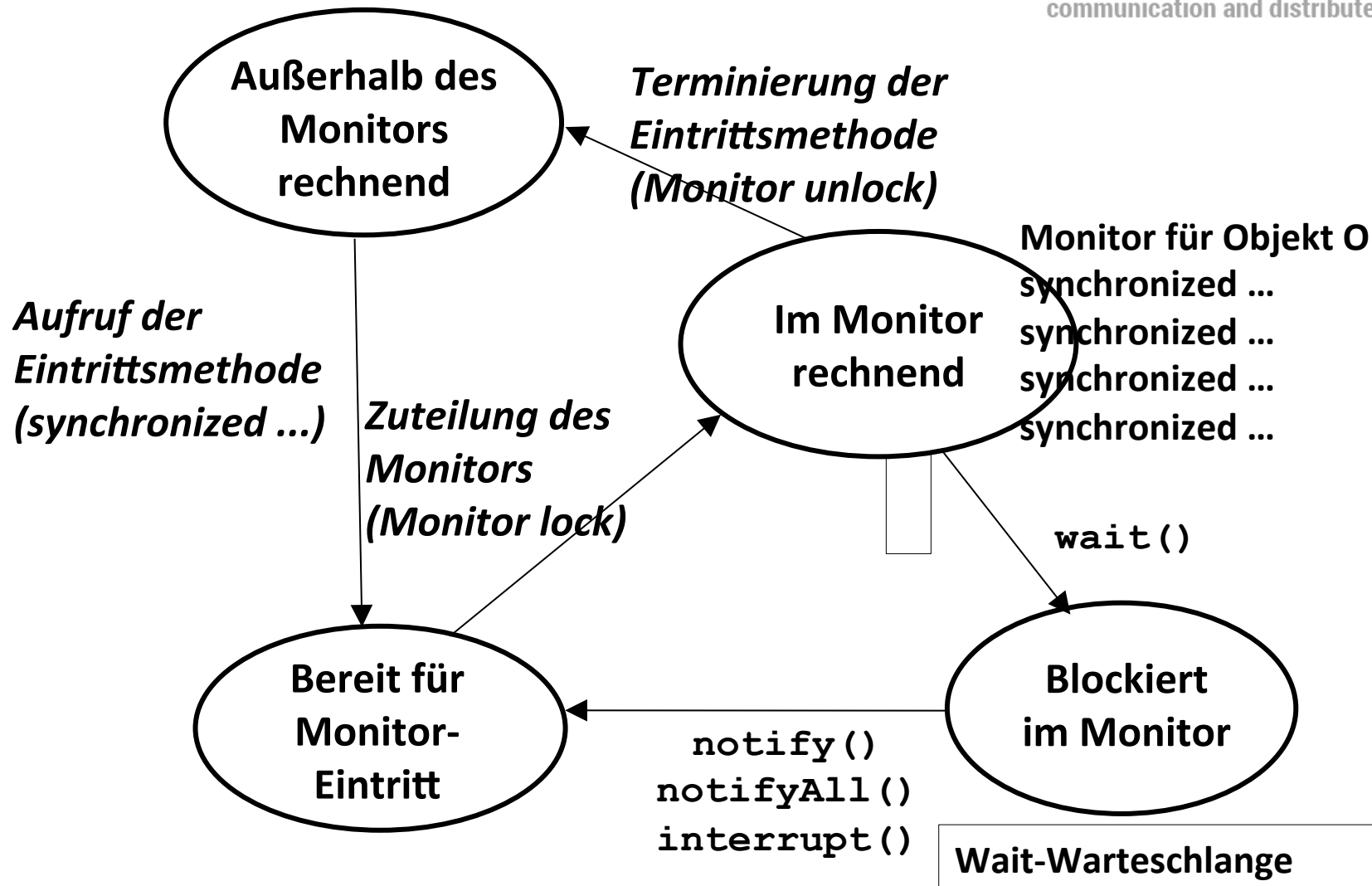
# 47. Reihenfolgebedingungen im Monitor

- Bisher gelöst: Wechselseitiger Ausschluss
  - 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???

# 48. 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)!

## 49. JAVA-Synchronisation: Thread-Zustandsdiagramm



## 50. 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

## 51. 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!



56. Ende des 3. Kapitels:  
Was haben wir geschafft?

## **Prozess-Synchronisation**

**3.1 Einführung und Grundlagen**

**3.2 Aktives Warten**

**3.3 Semaphore**

**3.4 Monitore**

**Next Live Coding: Philosophenproblem**