

# Koordination und Synchronisation von Prozessen



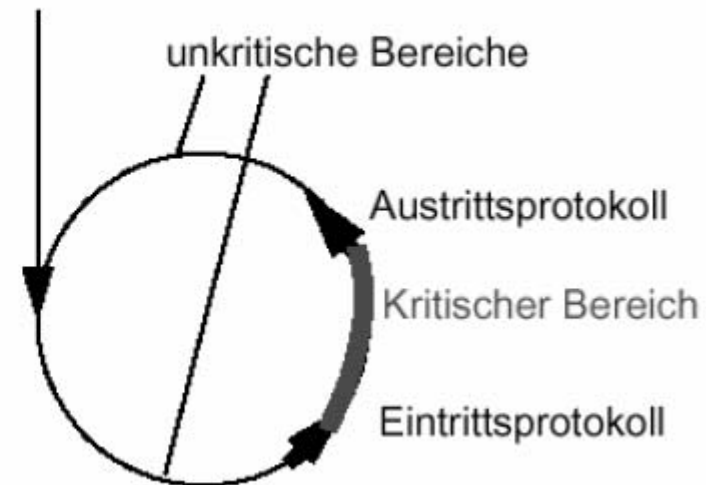
# Definitionen

- **Gemeinsames Betriebsmittel**
  - Ein Betriebsmittel (Speicher, Prozessor, E/A-Gerät), auf welches mehrere Prozesse gleichzeitig zugreifen können, ist ein gemeinsames Betriebsmittel dieser Prozesse.
- **Kritischer Bereich (engl. *critical region*)**
  - Synonym: Kritischer Abschnitt
  - Ein kritischer Bereich *bezüglich eines gemeinsamen Betriebsmittels H* ist ein Abschnitt eines Prozesses, in welchem dieser zu H zugreift.

# Protokoll für die Verwendung Kritischer Bereiche

```
<Initialisierungen>  
WHILE <not finished> DO  
  <Unkritischer Bereich>  
  <Eintrittsprotokoll>  
  <Kritischer Bereich>  
  <Austrittsprotokoll>  
END
```

Initialisierungsphase



# Anforderungen an die Behandlung Kritischer Bereiche (Dijkstra, 1965)

1. Zwei Prozesse dürfen *nicht gleichzeitig in ihren kritischen Bereichen bezüglich des gleichen Betriebsmittels sein* (mutual exclusion / gegenseitiger Ausschluss, GA).
2. Jeder Prozess, der am Eingang eines kritischen Bereichs wartet, muss diesen irgendwann auch betreten dürfen: *kein ewiges Warten* darf möglich sein (fairness condition).
3. Ein Prozess darf *ausserhalb eines kritischen Abschnitts* einen anderen Prozess *nicht blockieren*.
4. Es dürfen keine Annahmen über die *Abarbeitungsgeschwindigkeit* oder *Anzahl* der Prozesse bzw. Prozessoren getroffen werden.

# Versuch 1

```
                                VAR frei := TRUE;

P1:                                P2:
LOOP                                LOOP
    <unkritischer Bereich 1>;        <unkritischer Bereich 2>;
    REPEAT UNTIL frei;              REPEAT UNTIL frei;
    frei := FALSE;                  frei := FALSE;
    <kritischer Bereich 1>;          <kritischer Bereich 2>;
    frei := TRUE;                   frei := TRUE;
END                                  END
```

Der gegenseitige Ausschluss (GA) ist nicht gewährleistet

# Versuch 2

```
VAR freifür: INTEGER := 1;
```

```
P1:
```

```
LOOP
```

```
  <unkritischer Bereich 1>;
```

```
  REPEAT UNTIL freifür = 1;
```

```
  <kritischer Bereich 1>;
```

```
  freifür := 2;
```

```
END
```

```
P2:
```

```
LOOP
```

```
  <unkritischer Bereich 2>;
```

```
  REPEAT UNTIL freifür = 2;
```

```
  <kritischer Bereich 2>;
```

```
  freifür := 1;
```

```
END
```

GA gewährleistet, aber „Verhungern“ möglich

## Versuch 3

```
VAR
    besetzt1: BOOLEAN := FALSE;
    besetzt2: BOOLEAN := FALSE;

P1:
    LOOP
        <unkritischer Bereich 1>;
        REPEAT UNTIL not besetzt2;
        besetzt1 := TRUE;
        <kritischer Bereich 1>;
        besetzt1 := FALSE;
    END

P2:
    LOOP
        <unkritischer Bereich 2>;
        REPEAT UNTIL not besetzt1;
        besetzt2 := TRUE;
        <kritischer Bereich 2>;
        besetzt2 := FALSE;
    END
```

GA nicht gewährleistet, da beide Prozesse ungefähr gleichzeitig die REPEAT Schleife passieren können

# Versuch 4

```
VAR
    besetzt1: BOOLEAN := FALSE;
    besetzt2: BOOLEAN := FALSE;

P1:
    LOOP
        <unkritischer Bereich 1>;
        besetzt1 := TRUE;
        REPEAT UNTIL not besetzt2;
        <kritischer Bereich 1>;
        besetzt1 := FALSE;
    END

P2:
    LOOP
        <unkritischer Bereich 2>;
        besetzt2 := TRUE;
        REPEAT UNTIL not besetzt1;
        <kritischer Bereich 2>;
        besetzt2 := FALSE;
    END
```

GA gewährleistet, aber "Verklemmung" möglich



# Versuch 5

```
VAR
    besetzt1: BOOLEAN := FALSE;
    besetzt2: BOOLEAN := FALSE;

P1:
    LOOP
        <unkritischer Bereich 1>;
        besetzt1 := TRUE;
        WHILE besetzt2 DO
            besetzt1 := FALSE;
            besetzt1 := TRUE;
        END;
        <kritischer Bereich 1>;
        besetzt1 := FALSE;
    END

P2:
    LOOP
        <unkritischer Bereich 2>;
        besetzt2 := TRUE;
        WHILE besetzt1 DO
            besetzt2 := FALSE;
            besetzt2 := TRUE;
        END;
        <kritischer Bereich 2>;
        besetzt2 := FALSE;
    END
```

Praktisch gangbare Lösung, sofern ein synchroner Lauf der beiden Prozesse unwahrscheinlich ist

# Versuch 6: Dekker's Algorithmus

```
VAR
    besetzt1: BOOLEAN := FALSE;
    besetzt2: BOOLEAN := FALSE;
    VortrittFür: INTEGER := 1;

P1:
LOOP
    <unkritischer Bereich 1>;
    besetzt1 := TRUE;
    WHILE besetzt2 DO
        IF VortrittFür = 2 THEN
            besetzt1 := FALSE;
            REPEAT UNTIL VortrittFür =
                1;
            besetzt1 := TRUE;
        END
    END;
    <kritischer Bereich 1>;
    besetzt1 := FALSE;
    VortrittFür := 2;
END

P2:
LOOP
    <unkritischer Bereich 2>;
    besetzt2 := TRUE;
    WHILE besetzt1 DO
        IF VortrittFür = 1 THEN
            besetzt2 := FALSE;
            REPEAT UNTIL VortrittFür =
                2;
            besetzt2 := TRUE;
        END
    END;
    <kritischer Bereich 2>;
    besetzt2 := FALSE;
    VortrittFür := 1;
END
```

„Die Lösung des Theoretikers“

# Der Semaphor: Definition

- Semaphor  $v$ : Abstrakter Datentyp mit den Operationen  $\text{Signal}(v)$  und  $\text{Wait}(v)$
- Def.  $r(v)$ : Anzahl vollständig ausgeführte  $\text{Wait}$ -Operationen auf  $v$  seit Zeitbeginn
- Def.  $s(v)$ : Anzahl vollständig ausgeführte  $\text{Signal}$ -Operationen auf  $v$  seit Zeitbeginn
- $t=0$ :  $r(v)=0$ ,  $s(v)=0$
- für  $t \geq 0$  gilt:
  - I:  $0 \leq r(v) \leq s(v)$
  - II:  $s(v) - r(v) \leq \max$

## Implikationen für die Implementation

1. Die Operationen  $\text{Signal}(v)$  und  $\text{Wait}(v)$  müssen als kritische Bereiche bezüglich  $v$  implementiert sein.
2. Die Ausführung von  $\text{Wait}(v)$  zu einer Zeit, wenn  $s(v)=r(v)$  ist, verzögert den die Operation ausführenden Prozess, bis  $s(v)<r(v)$  ist; anschliessend kann  $\text{Wait}(v)$  vollständig ausgeführt werden.
  - Der Prozess wird in  $\text{Wait}(v)$  blockiert!
3. Die Ausführung von  $\text{Signal}(v)$  zu einer Zeit, wenn  $s(v)-r(v) = \max$  ist, wird als Ausnahmesituation behandel (Programmierfehler  $\rightarrow$  exception).

# Implementationskizze für Wait()

Wait(v):

*If*  $r(v) < s(v)$  *Then*

$r(v) = r(v) + 1$

*Else*

<der aufrufende Prozess wird verzögert>

*Endif*

# Implementationskizze für `Signal()`

`Signal(v):`

*If*  $s(v) - r(v) = \max$  *Then*

*<signalisiere eine Ausnahmebedingung>*

*Else*

$s(v) = s(v) + 1$

*If* *<ein Prozess wartet bei v>* *Then*

$r(v) = r(v) + 1;$

*<Prozess zur Ausführung bereit machen>*

*Endif*

*Endif*

# Gegenseitiger Ausschluss mittels Semaphor

`var v: semaphore;`    `v` schützt die kritischen Bereiche aller beteiligten Prozesse bezüglich eines Betriebsmittels

```
process p1;
BEGIN
  LOOP
    <unkritischer Bereich von p1>
    WAIT(v);
    <kritischer Bereich von p1>
    SIGNAL(v)
  END
END;

process p2;
BEGIN
  LOOP
    <unkritischer Bereich von p2>
    WAIT(v);
    <kritischer Bereich von p2>
    SIGNAL(v)
  END
END;

BEGIN (* Initialisierung, wird nur einmal ausgeführt *)
  SIGNAL(v)
END
```

**ETH**

Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

# Interprozesskommunikation





# Konzepte für die Interprozesskommunikation

- Verwendung eines gemeinsamen Speichers (shared memory) → gemeinsames Betriebsmittel der kommunizierenden Prozesse! → Semaphore!
- Austausch von Nachrichten (message passing)
- Gemeinsamer Speicher mit Konstrukten auf einer höheren Abstraktionsebene
  - Critical Region (Hoare, Brinch-Hansen 1972) [Brause S. 73-74]
  - Monitor (Brinch-Hansen 1973, Hoare 1974) [Brause S. 74-75]
- Rendez-vous Konzept der Programmiersprache ADA (entwickelt 1977-83 als Sprache für die Programmierung eingebetteter Systeme)

# Message Passing

- Prozess sendet eine Nachricht über einen Kanal zu einem oder mehreren Prozessen
  - `send(Ziel, Nachricht)`; `receive(*Quelle, Nachricht)`
  - unicast, multicast, broadcast (Punkt-Punkt vs. Punkt-Mehrpunkt)
- Varianten
  - Direkte Kommunikation zwischen Prozessen vs. Indirekte Kommunikation über Briefkästen (mailbox)
  - Empfangen von genau einer Quelle oder von beliebigen Quellen
  - Kanal mit oder ohne Speicherkapazität
- Message Passing benötigt keinen gemeinsamen Speicher, kann aber gut damit realisiert werden

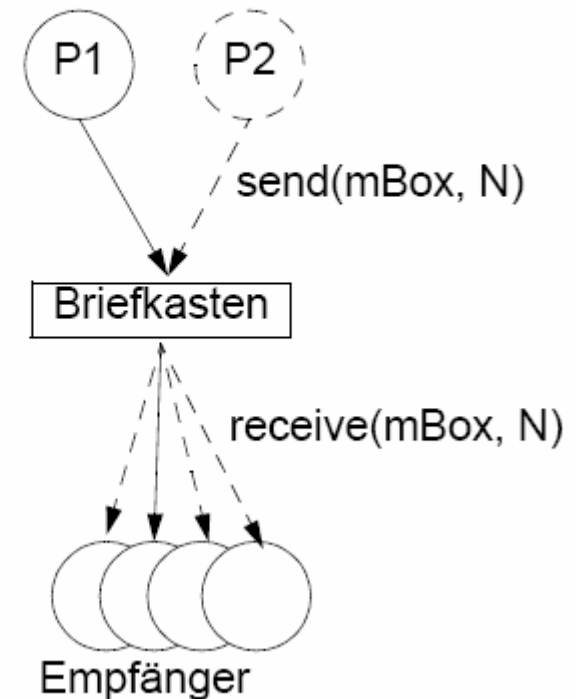
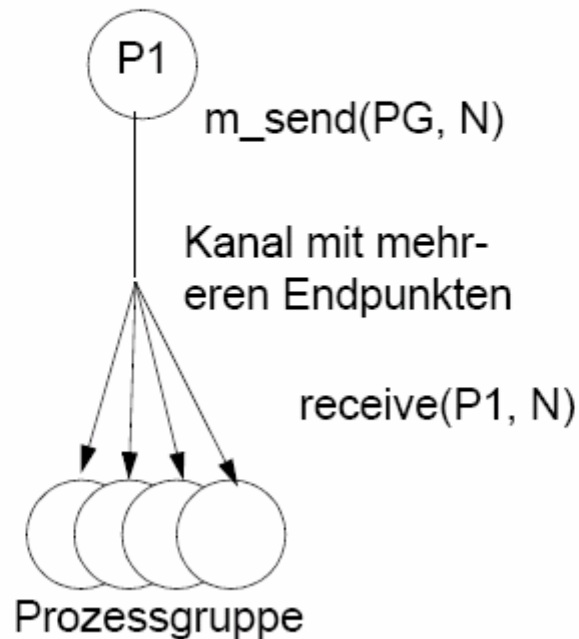
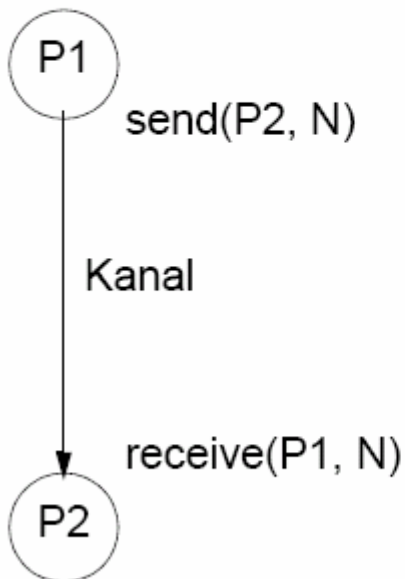
# Direkte vs. indirekte Kommunikation

direkte Kommunikation

indirekte Kommunikation

Punkt-Punkt

Punkt-Mehrpunkt



# Speicherfähigkeit des Kanals

- Kanal kann keine Nachricht speichern
  - Sendeoperation blockiert, bis ein Empfängerprozess bereit ist, die Nachricht entgegenzunehmen (synchrones Senden)
  - Alternative: Nachricht geht verloren, wenn Empfänger nicht bereit ist
- Kanal kann eine begrenzte Zahl von Nachrichten zwischenspeichern
  - Blockierung wenn Puffer voll (asynchrones Senden)
- Kanal kann eine unbegrenzte Zahl von Nachrichten zwischenspeichern
  - Theoretisch: Sendeoperation blockiert nie
  - Praktisch: Überschreiten der realen Kanalkapazität ist ein Programmierfehler.

# Eigenschaften der ausgetauschten Nachricht

- Senden einer Kopie der Nachricht oder eines Zeigers auf die Nachricht
- Nachrichten mit fester oder variabler Länge
- Struktur der Nachrichten
  - Unstrukturierte Bytestrings
  - Vordefinierte Struktur (z.B. Typ, Länge, Inhalt)

# Beispiel: Interprozesskommunikation in Topsy

## Message Passing:

- Direkte Kommunikation: Prozesse werden über ihre Identifikation adressiert
- Genau ein Empfänger
- Asynchrones Senden mit “unbegrenztem” Pufferspeicher
- Nachrichten haben eine **feste** Länge und haben eine Struktur: (Messageld, Daten = 3 32-Bit Werte), s. TOPSY-Book S. 15, File Messages.h.
- Beim Empfangen: Selektion von Absendern und Typen von Nachrichten möglich

# System Calls für IPC in Topsy

- Senden von Nachrichten

```
SyscallError tmMsgSend(ThreadId to, Message *msg)
```

- Empfangen von Nachrichten

```
SyscallError tmMsgRecv(ThreadId* from, MessageId msgId, Message* msg, int timeout)
```

- from, to: Identifikation eines Prozesses
  - from == ANY: Empfangen einer Nachricht eines beliebigen Prozesses
- msg: Zeiger auf Nachricht, als Parameter übergeben
- msgId: Typ der Nachricht (anwendungsspezifisch)
- msgId == ANYMSGTYPE: beliebiger Typ einer Nachricht akzeptabel
- timeout: Maximale Wartezeit von tmMsgRecv

## Beispiele und klassische Anwendungen

- Implementation eines der behandelten Konstrukte mit einem anderen (Semaphore mit Message Passing, Message Passing mit gemeinsamem Speicher und Semaphoren)
- Semaphore mit einem internen Zähler (counting semaphore) durch binäre Semaphore
- Zirkulärer Zwischenspeicher (FIFO, ring buffer) [Brause S. 70]
- Reader/Writer – Problem [Brause S. 69]
- Problem der „dining philosophers“ [Brause S. 73]

<http://www.cs.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html>

(s. auch Link in Blackboard)