



# Betriebssysteme

---

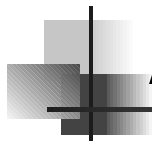
## Prozess-Synchronisation



# Überblick

---

- Arten der Synchronisation
- Kritische Abschnitte
- Kriterien für Synchronisationslösungen
- Hardware-Unterstützung
- Semaphore
- Klassische Synchronisationsprobleme
- Monitore



# Arten der Synchronisation

- ( )  
,  
Konkurrenz um die Nutzung gemeinsamer,  
aber nur exklusiv nutzbarer Betriebsmittel  
(Variable, Dateien, E/A-Geräte usw.)
- /  
Abstimmung der Ausführung der  
Verarbeitungsschritte aufgrund von  
Datenabhängigkeiten (z. B. Produzenten-  
Konsumenten-Synchronisation)



# Race Conditions und kritische Abschnitte

## Prozess/Thread 1

```
...  
i = leseZaehler();  
i = i + 10;  
schreibeZaehler( i );  
...
```

## Prozess/Thread 2

```
...  
j = leseZaehler();  
j = j - 5;  
schreibeZaehler( j );  
...
```

( ):  
Programmabschnitte, in denen sich zu jedem Zeitpunkt  
maximal ein Prozess/Thread befinden darf

# Bedingungen für eine gute Lösung (nach Dijkstra)

1. Zu jedem Zeitpunkt befindet sich  
Prozess im kritischen Bereich  
( )
2. Es werden über die  
oder die  
der Prozesse getroffen
3. des kritischen Bereichs darf ein  
Prozess andere Prozesse
4. Wartezeit vor dem  
Eintritt in den kritischen Bereich ( )

Uwe Neuhaus

BS: Prozess-Synchronisation

5

## Erster Lösungsversuch

dran = 1;

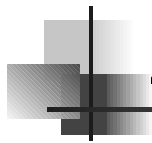
### Prozess/Thread 1

```
i = leseZaehler();  
i = i + 10;  
schreibeZaehler( i );
```

### Prozess/Thread 2

```
j = leseZaehler();  
j = j - 5;  
schreibeZaehler( j );
```

- Probleme:
- Busy-Wait
  - Ein Prozess blockiert sich selbst



## Zweiter Lösungsversuch

bereit1 = bereit2 = false;

### Prozess/Thread 1

```
i = leseZaehler();  
i = i + 10;  
schreibeZaehler( i );
```

### Prozess/Thread 2

```
j = leseZaehler();  
j = j - 5;  
schreibeZaehler( j );
```

- Probleme:
- Busy-Wait
  - Prozesse/Threads können sich gegenseitig blockieren (Verklemmung/Deadlock)



## Dritter Lösungsversuch

bereit1 = bereit2 = false;

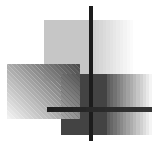
dran = 0;

### Prozess/Thread 1

```
i = leseZaehler();  
i = i + 10;  
schreibeZaehler( i );
```

### Prozess/Thread 2

```
j = leseZaehler();  
j = j - 5;  
schreibeZaehler( j );
```



## Hardware-Unterstützung

- Interrupts ausschalten
- Atomare Maschineninstruktionen:
  - :  
Auslesen und Setzen einer Speicherzelle
  - :  
Tauschen des Wertes zweier Speicherzellen
  - :  
Auslesen einer Speicherzelle und Erhöhen der Zelle um einen angegebenen Wert



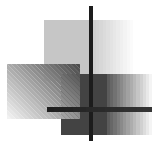
## Synchronisation mit TestAndSet bzw. Swap

```
belegt = false;  
Prozess/Thread
```

```
i = leseZaehler();  
i = i + 10;  
schreibeZaehler( i );
```

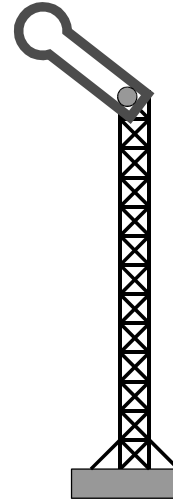
```
belegt = false;  
Prozess/Thread
```

```
j = leseZaehler();  
j = j - 5;  
schreibeZaehler( j );
```



# Semaphore (Dijkstra, 1965)

- „  
“
- zum Signalisieren des Zustands eines kritischen Abschnitts
- Meist verbunden mit einer zugehörigen
- Atomare (unteilbare, ununterbrechbare) Operationen:
  - : der Semaphore ( )
  - : der Semaphore ( )



## Lösung mit Semaphor

### Prozess/Thread 1

```
i = leseZaehler();  
i = i + 10;  
schreibeZaehler( i );
```

### Prozess/Thread 2

```
j = leseZaehler();  
j = j - 5;  
schreibeZaehler( j );
```

S ist ein Semaphoren-Objekt mit den Methoden  
wait() (möchte passieren) und signal() (verlassen)

# Realisierung eines binären Semaphors

## **S.wait():**

```
if ( ) {  
    Prozess in Warteschlange W einreihen;  
    Prozess in Zustand „wartend“ versetzen;  
}
```

## **S.signal():**

```
if ( ) {  
    Einen Prozess aus Warteschlange W lösen;  
    Prozess in Zustand „bereit“ versetzen;  
}  
else { belegt = false; }
```

# Realisierung eines Zähl-Semaphors

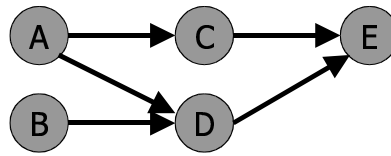
## **S.wait():**

```
if ( ) {  
    Prozess in Warteschlange W einreihen;  
    Prozess in Zustand „wartend“ versetzen;  
}
```

## **S.signal():**

```
if ( ) {  
    Einen Prozess aus Warteschlange W lösen;  
    Gelösten Prozess in Zustand „bereit“ versetzen;  
}
```

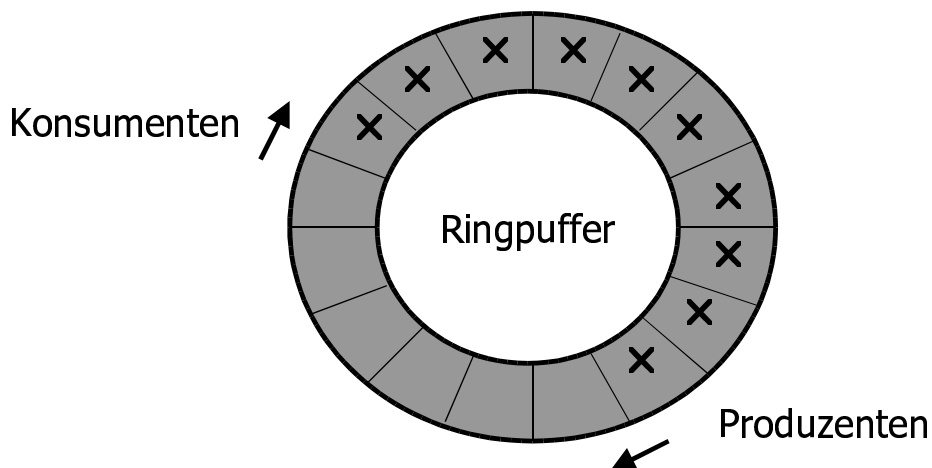
# Semaphore zur Synchronisation bei Präzedenzen



Binäre Semaphore c, d1, d2, e1 und e2 mit false initialisieren

|                          |        |                          |
|--------------------------|--------|--------------------------|
| A:                       | BodyA; | c.signal(); d1.signal(); |
| B:                       | BodyB; | d2.signal();             |
| C: c.wait();             | BodyC; | e1.signal();             |
| D: d1.wait(); d2.wait(); | BodyD; | e2.signal();             |
| E: e1.wait(); e2.wait(); | BodyE; |                          |


# Produzenten/Konsumenten-Problem (mit Ringpuffer)






# Semaphore zur Produzenten/ Konsumenten-Synchronisation

## Produzent

```
while (true) {  
    ware = produziere();  
  
      
}
```

## Konsument

```
while (true) {  
  
      
    konsumiere(ware);  
}
```

Initialisierung: mutex.zaehler = 1; frei.zaehler = max;  
belegt.zaehler = 0;

# Weitere klassische Synchronisationsprobleme

- - Einige Prozesse/Threads wollen einen Datenbereich lesen, einige wollen ihn verändern.
  - Gleichzeitiger Lesezugriff ist erlaubt
  - Schreibzugriffe müssen exklusiv erfolgen
- - Fünf Philosophen sitzen um einen runden Tisch, denken nach und essen Reis mit Stäbchen.
  - Zwischen den Tellern liegt jeweils ein Stäbchen, zum Essen braucht man aber zwei.

# Monitore

- funktional äquivalent zu Semaphoren
- werden in einer Klasse mit einem zugehörigen Semaphor kombiniert.
- Unterstützung der Synchronisation durch

## Schematischer Aufbau eines Monitors

