



# Nebenläufigkeit

## Probleme



## Probleme

- **Race Conditions:**
  - Situationen, in denen zwei oder mehrere Threads/Prozesse, die gleichen geteilten Daten lesen oder schreiben und das Resultat davon abhängt, wann genau welcher Prozess ausgeführt wurde, werden Race Conditions genannt.
  - Lösung: Einführung von **kritischen Bereichen** und **wechselseitiger Ausschluss**.
- **Starvation (Aussperrung):**
  - Situation, in der ein Prozess unendlich lange auf ein Betriebsmittel wartet. Wichtig: sinnvolle Realisierung von Warteschlangen bei der Betriebsmittelvergabe, z.B. Prioritätenbasierte Warteschlangen
- **Priority Inversion (Prioritätsinversion):**
  - Wichtige Prozesse können durch unwichtigere Prozesse, die Betriebsmittel belegt haben verzögert werden, genaue Problemstellung siehe Kapitel Scheduling

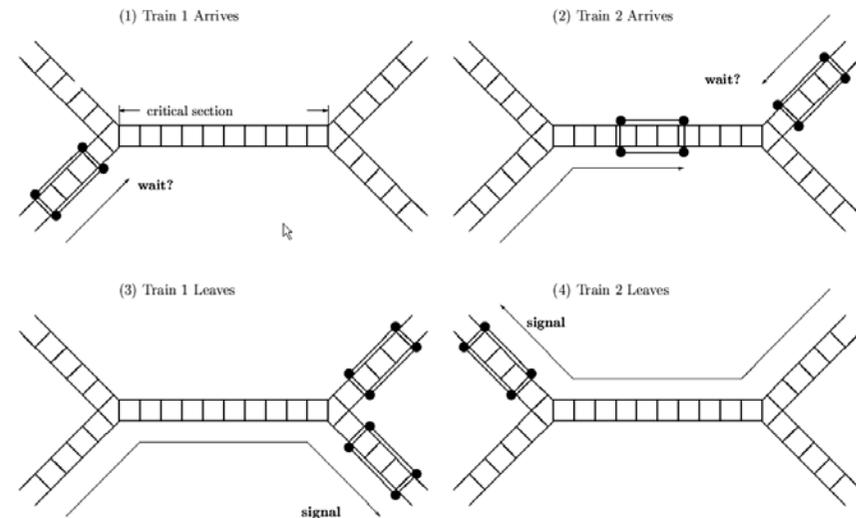


## Bedingungen an Lösung für wechselseitigen Ausschluss

- An eine gute Lösung für den wechselseitigen Ausschluss (WA) können insgesamt vier Bedingungen gestellt werden:
  1. Es dürfen niemals zwei Prozesse gleichzeitig im kritischen Bereich sein.
  2. Es dürfen keine Annahmen über die Geschwindigkeit oder Anzahl der Prozessoren gemacht werden.
  3. Kein Prozess darf außerhalb von kritischen Regionen andere Prozesse blockieren.
  4. Kein Prozess soll unendlich auf das Eintreten in den kritischen Bereich warten müssen.

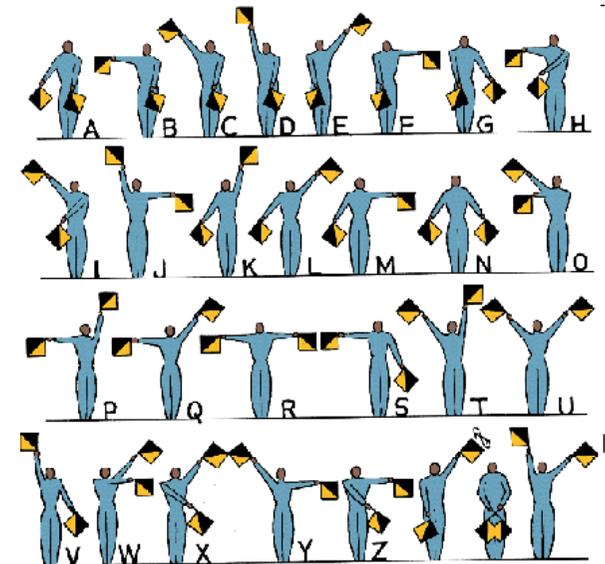
## Kritische Bereiche

- Um einen kritischen Bereich zu schützen, sind Mechanismen erforderlich, die ein gleichzeitiges Betreten verschiedener Prozesse bzw. Prozeßklassen dieser Bereiche verhindern.
  - Darf maximal nur ein Prozess gleichzeitig auf den kritischen Bereich zugreifen, so spricht man vom **wechselseitigen Ausschluss**.
  - Wird verhindert, daß mehrere (unter-schiedlich viele) Instanzen unter-schiedlicher Prozeßklassen auf den Bereich zugreifen, so entspricht dies dem **Leser-Schreiber-Problem** (so dürfen beispielsweise mehrere Instanzen der Klasse Leser auf den Bereich gleichzeitig zugreifen, Instanzen der Klasse Schreiber benötigen den exklusiven Zugriff).



## Programmtechnik zum Schutz kritischer Bereiche

- Aus dem Alltag sind diverse Mechanismen zum Schutz kritischer Bereiche bekannt:
  - *Signale* im Bahnverkehr
  - *Ampeln* zum Schutz der Kreuzung
  - *Schlösser* für einzelne Räume
  - *Vergabe* von Tickets
- Erstes Mittel zum Schutz kritischer Bereiche in der Softwaretechnik: der/das Semaphor, griech. für „Zeichenträger“ (siehe später und rechts)



## Falsche Lösung: Verwendung einer globalen Variable

*Prozeß A*

```
bool block = false; //global  
variable
```

```
...  
while(block){}; //busy wait  
block=true;  
... critical section A ...  
block=false;  
...
```

*Prozeß B*

```
...  
while(block){}; //busy wait  
block=true;  
... critical section A ...  
block=false;  
...
```

- Die obige Implementierung ist nicht korrekt,
  - da der Prozess direkt nach dem while-Abschnitt unterbrochen werden könnte und evtl. dann fortgesetzt wird, wenn block bereits durch einen anderen Prozess belegt ist.
  - Zudem ist die Lösung ineffizient (busy wait)



## 1. Möglichkeit: Peterson 1981 (Lösung für zwei Prozesse)

```
int turn=0;      Deklaration globale Variablen
boolean ready[2];
ready[0]=false;
ready[1]=false;
```

```
...                                Prozess 0
ready[0]=true;
turn = 1;
while(ready[1]
    && turn==1){}; //busy waiting
... critical section ...
ready[0]=false;
...
```

```
...                                Prozess 1
ready[1]=true;
turn = 0;
while(ready[0]
    && turn==0){}; //busy waiting
... critical section ...
ready[1]=false;
...
```

- Ausschluß ist garantiert, aber „busy waiting“ verschwendet immer noch Rechenzeit
- Die Realisierung für N Prozesse ist als „Lamport's Bakery Algorithmus“ bekannt:  
[http://en.wikipedia.org/wiki/Lamport's\\_bakery\\_algorithm](http://en.wikipedia.org/wiki/Lamport's_bakery_algorithm)

## 2. Möglichkeit: Ausschalten von Unterbrechungen zum WA

- Prozesswechsel beruhen immer auf dem Eintreffen einer Unterbrechung (z.B. neues Ereignis, Ablauf einer Zeitdauer)
- Die einfachste Möglichkeit einen Kontextwechsel zu verhindern ist das Ausschalten von Unterbrechungen bevor ein Prozess in den kritischen Bereich geht.
- Vorteile:
  - einfach zu implementieren, keine weiteren Konzepte sind nötig
  - schnelle Ausführung, Schreiben von Bits in Register
- Nachteile:
  - Für Multiprozessorsysteme ungeeignet
  - Keine Gerätebehandlung während der Sperre
  - Lange Sperren kritisch bei Echtzeitanwendungen

### 5.4.5 Interrupt Enable Clear register (VICIntEnClear - 0xFFFF F014)

This is a write only register. This register allows software to clear one or more bits in the Interrupt Enable register (see [Section 5.4.4 "Interrupt Enable register \(VICIntEnable - 0xFFFF F010\)" on page 52](#)), without having to first read it.

Table 42: Software Interrupt Clear register (VICIntEnClear - address 0xFFFF F014) bit allocation  
Reset value: 0x0000 0000

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	-	-	-	-	-	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	23	22	21	20	19	18	17	16
Symbol	-	-	AD1	BOD	I2C1	AD0	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	15	14	13	12	11	10	9	8
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C0	PWM0
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	7	6	5	4	3	2	1	0
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO



### 3. Möglichkeit: Semaphor

- Semaphor (griechisch von Zeichenträger, Signalmast) wurden von Edsger W. Dijkstra im Jahr 1965 eingeführt.
- Ein Semaphor ist eine Datenstruktur, bestehend aus einer Zählvariable  $s$ , sowie den Funktionen `down()` oder `wait()` (bzw.  $P()$ , von probeer te verlagen) und `up()` oder `signal()` (bzw.  $V()$ , von verhogen).

```
Init(Semaphor s, Int v)    V(Semaphor s)    P(Semaphor s)
{                          {                          {
  s = v;                   s = s+1;          while (s <= 0) {} ; // Blockade, unterschiedliche Implementierungen
}                          }                          s = s-1 ;          // sobald s>0 belegt eine Ressource
                           }                          }
```

- Bevor ein Prozess in den kritischen Bereich eintritt, muss er den Semaphor mit der Funktion `down()` anfordern. Nach Verlassen wird der Bereich durch die Funktion `up()` wieder freigegeben.
- **Wichtige Annahme:** die Ausführung der Funktionen von `up` und `down` darf nicht unterbrochen werden (atomare Ausführung), siehe Realisierung
- Solange der Bereich belegt ist (Wert des Semaphors  $\leq 0$ ), wird der aufrufende Prozess blockiert.

## Beispiel: Bankkonto

- Durch Verwendung eines gemeinsamen Semaphors `semAccount` kann das Bankkonto `account` auch beim schreibenden Zugriff von zwei Prozessen konsistent gehalten werden:

### Prozess A

```
P ( semAccount ) ;  
x=readAccount ( account ) ;  
x=x+500 ;  
writeAccount ( x , account ) ;  
V ( semAccount ) ;
```

### Prozess B

```
P ( semAccount ) ;  
y=readAccount ( account ) ;  
y=y-200 ;  
writeAccount ( y , account ) ;  
V ( semAccount ) ;
```

- Zur Realisierung des wechselseitigen Ausschlusses wird ein binärer Semaphor mit zwei Zuständen: 0 (belegt), 1 (frei) benötigt. Binäre Semaphore werden auch *Mutex* (von *mutal exclusion*) genannt.



## Erweiterung: zählender Semaphore

- Nimmt ein Wert auch einen Wert größer eins an, so wird ein solch ein Semaphor auch als **zählender Semaphor** (counting semaphore) bezeichnet.
- Beispiel für den Einsatz von zählenden Semaphoren: In einem **Leser-Schreiber-Problem** kann die Anzahl der Leser aus Leistungsgründen z.B. auf 100 gleichzeitige Lesezugriffe beschränkt werden:

```
semaphore sem_reader_count ;  
init(sem_reader_count, 100) ;
```

- Jeder Leseprozess führt dann folgenden Code aus:

```
P(sem_reader_count) ;  
read() ;  
V(sem_reader_count) ;
```

- Leser-Schreiber-Probleme sind vielfältig modifizierbar, je nach Priorität der prozesse. LS-Problem: Keine Prioritäten. Erstes LS-Problem: Leserpriorität. Zweites LS-Problem: Schreiber-Priorität.



## Realisierungen von Semaphoren

- Die Implementierung eines Semaphors erfordert spezielle Mechanismen auf Maschinenebene; der Semaphor ist für sich ein kritischer Bereich.  
⇒ Die Funktionen  $up()$  und  $down()$  dürfen nicht unterbrochen werden, da sonst der Semaphor selbst inkonsistent werden kann.
- Funktionen die nicht unterbrechbar sind, werden **atomar** genannt.
- Realisierungsmöglichkeiten:
  1. Kurzfristige Blockade der Prozeßwechsel während der Bearbeitung der Funktionen  $up()$  und  $down()$ . Implementierung durch Verwendung einer Interrupt-Sperre, denn sämtliche Prozesswechsel werden durch **Unterbrechungen (Interrupts)** ausgelöst.
  2. **Spinlock**: Programmieretechnik auf der Basis von Busy Waiting. Vorteil: Unabhängig vom Betriebssystem und auch in Mehrprozessorsystemen zu implementieren, jedoch massive Verschwendung von Rechenzeit. Im Gegensatz dazu können die Lösungen von 1 und 2 mit Hilfe von Warteschlangen sehr effizient realisiert werden.
  3. **Test&Set**-Maschinenbefehl: Die meisten Prozessoren verfügen heute über einen Befehl „**Test&Set**“ (oder auch Test&SetLock). Dieser lädt atomar den In-halt (typ. 0 für frei, 1 für belegt) eines Speicherwortes in ein Register und schreibt ununterbrechbar einen Wert (typ.  $\neq 0$ , z.B. 1 für belegt) in das Speicherwort.



# Realisierungen von Semaphoren

## Verwendung des Test&Set Maschinenbefehls

*enter\_region: ; A "jump to" tag; function entry point.*

```
tsl reg, flag           ; Test and Set Lock; flag is the
                        ; shared variable; it is copied
                        ; into the register reg and flag
                        ; then atomically set to 1.
cmp reg, #0             ; Was flag zero on entry?
jnz enter_region       ; Jump to enter_region if
                        ; reg is non-zero; i.e.,
                        ; flag was non-zero on entry.
ret                    ; Exit; i.e., flag was zero on
                        ; entry. If we get here, tsl
                        ; will have set it non-zero; thus,
                        ; we have claimed the resource as-
                        ; sociated with flag.
```

## Test&Set-Maschinenbefehl bei Mehrprozessorsystemen

- Problem: gemeinsamer Zugriff von mehreren Prozessoren auf einen Speicherbereich
- Für die Test&Set Operation muss für eine CPU der exklusive Zugriff auf den Speicherbereich garantiert sein.
  - Bus Locking
- Mechanismen im Intel Pentium II für den atomaren Zugriff auf Speicherbereiche: Multiple Processor Management Abschnitt 7.1:  
<http://download.intel.com/design/PentiumII/manuals/24319202.pdf>