



# Betriebssysteme

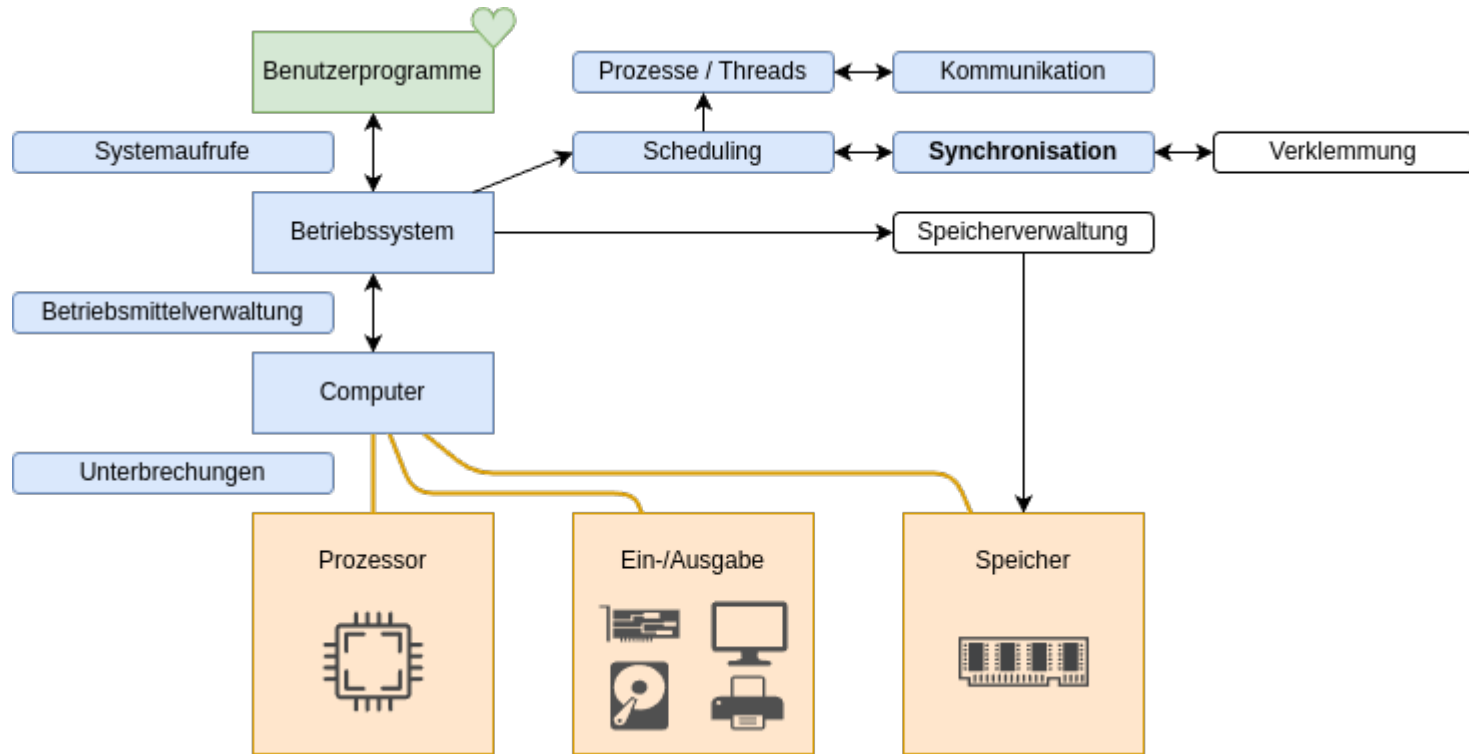
Stage 7 – Synchronisation

# Was bisher geschah ...

---

- > Geschichte der Rechner und Betriebssysteme
- > Von-Neumann-Architektur
- > Bootprozess und Sicherheitskonzept
- > Interaktion mit dem Betriebssystem
  - Systemaufrufe (*system calls*)
  - Unterbrechungen (*interrupts*)
- > Aufbau von Betriebssystemen
- > Betriebssystemstrukturen
  
- > Prozesse & Threads
  
- > Scheduling
  
- > Prozesskommunikation
  - Shared memory
  - Signale
  - Pipes

# Übersicht



# Wiederholung

---

## > Problemstellung

- Mehrere Prozesse bearbeiten gemeinsam eine Aufgabe
  - Verkürzung der Bearbeitungszeit durch Nutzung mehrerer CPUs
  - Verbergen der Bearbeitung durch Ausführung im Hintergrund
- Die Bearbeitung koordiniert werden
  - Geregelter Zugriff auf gemeinsame Datenstrukturen
  - Reihenfolge der Bearbeitung kann eine Rolle spielen
  - Austausch von Daten zwischen Prozessen

## > Mechanismen zur Kommunikation und Synchronisation notwendig

- Kommunikation durch gemeinsamen Speicher
- Kommunikation durch Nachrichtenaustausch
  - Spezialfall: Kommunikation von Ereignissen
- Synchronisation des Prozessablaufs

## > Das Betriebssystem stellt Prozessen diese Funktionalitäten zur Verfügung

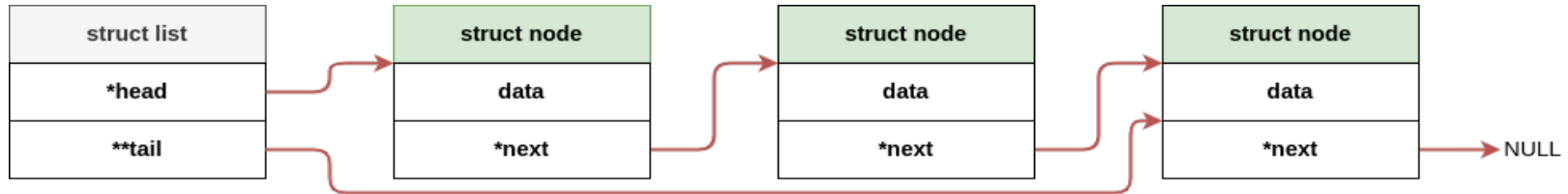
# Inhalt: Prozesssynchronisation

---

- > Synchronisation
  - Problemstellung & Lösungsansatz am Beispiel FIFO-Liste (und Bankkonto)
  - Naive Vorgehensweise
    - Mittels Softwareunterstützung
    - Mittels Hardwareunterstützung
  - Valide Vorgehensweise
    - Mittels Softwareunterstützung
    - Mittels Hardwareunterstützung
- > Betriebssystemunterstützung: Semaphore & Mutex
- > Programmiersprachenunterstützung: Monitore
- > Beispiel: Klassische Synchronisationsprobleme

# Beispiel Synchronisation: FIFO-Liste

```
1 void push (struct list *l, int data) {  
2     // Neues element erzeugen  
3     struct node *el = (struct node *) malloc (sizeof (struct node));  
4     el->data = data;  
5     el->next = NULL;  
6  
7     // Neues element in liste einfuegen  
8     *l->tail = el;  
9     l->tail = &el->next;  
10  
11     return;  
12 }
```



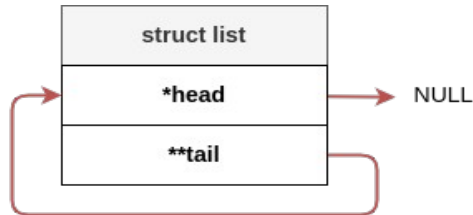
# Beispiel Synchronisation

## Thread A

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```

## Thread B

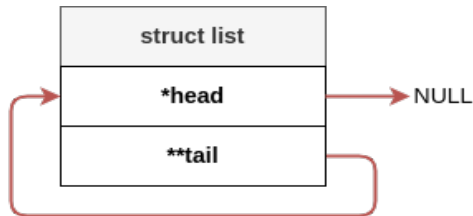
```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```



# Beispiel Synchronisation

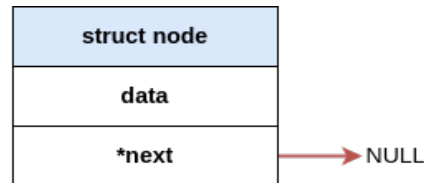
## Thread A

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```



## Thread B

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```



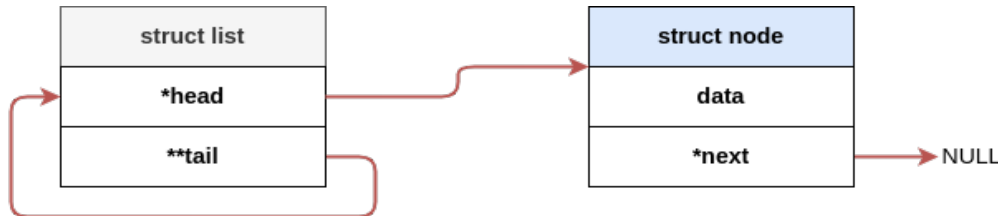
# Beispiel Synchronisation

## Thread A

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```

## Thread B

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```



# Beispiel Synchronisation

## Thread A

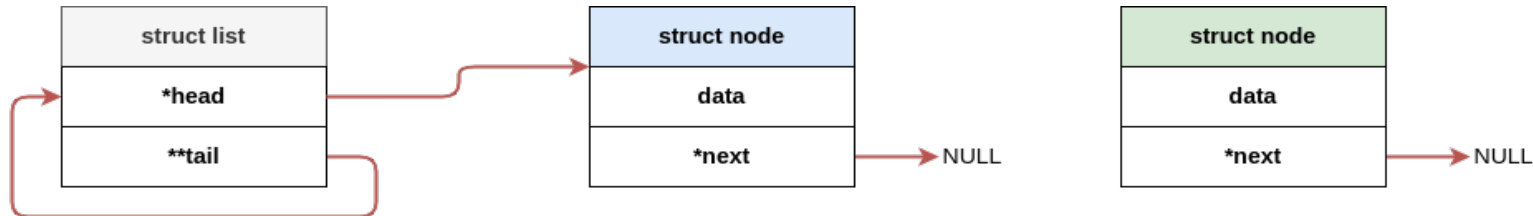
```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```

Kontextwechsel



## Thread B

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```



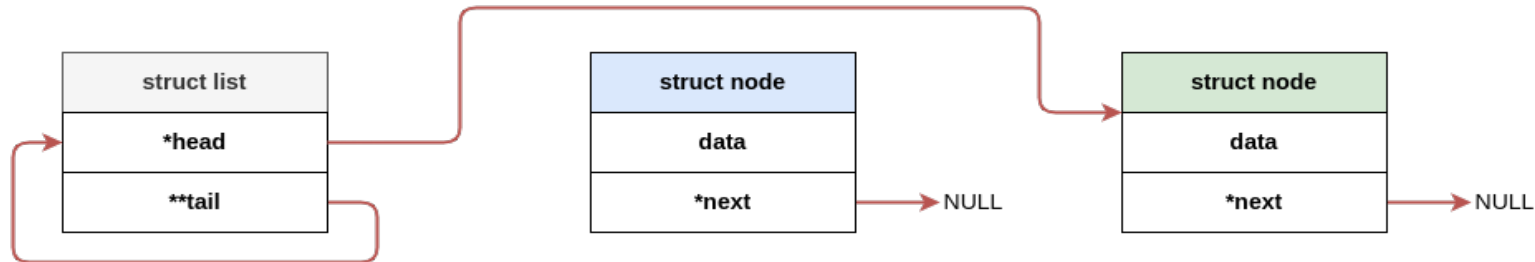
# Beispiel Synchronisation

## Thread A

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```

## Thread B

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```



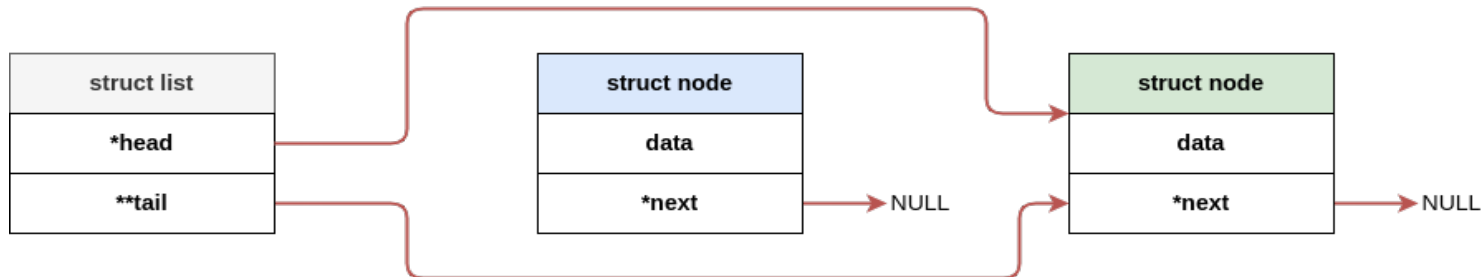
# Beispiel Synchronisation

## Thread A

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```

## Thread B

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```



# Beispiel Synchronisation

## Thread A

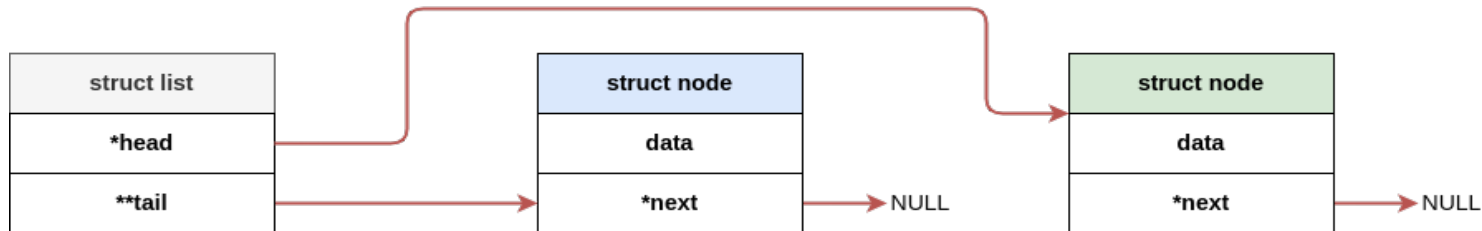
```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```

Kontextwechsel



## Thread B

```
void push (struct list *l, int data) {  
    // Neues element erzeugen  
    struct node *el = malloc (...);  
    el->data = data;  
    el->next = NULL;  
  
    // Head / Next Pointer anpassen  
    *l->tail = el;  
  
    // Tail Pointer anpassen  
    l->tail = &el->next;  
  
    return;  
}
```



# Beispiel Synchronisation: Erkenntnis

---

- > Das gezeigte Problem
  - kann auftreten, muss aber nicht auftreten!
  - ist nicht zuverlässig reproduzierbar (→ u.a. abhängig von der Systemlast, Anzahl CPUs, ...)
- > Man spricht von einer **Race Condition**
  - *“Situationen, in denen zwei oder mehr Prozesse einen gemeinsamen Speicher lesen oder beschreiben und das Endergebnis davon abhängt, wer wann genau läuft, werden Race Conditions genannt”* [Tanenbaum]
  - Lesen ist unproblematisch. Das Problem entsteht, wenn mindestens ein Prozess Daten verändert
- > Kritische Regionen
  - In der Regel ist der größte Teil der Arbeiten eines Prozesses nicht kritisch
  - Lediglich der schreibende Zugriff auf die gemeinsamen Daten muss koordiniert erfolgen
    - Man spricht von **kritischen Regionen**
- > Lösung
  - Zu jedem Zeitpunkt darf sich nur ein einziger Prozess im kritischen Abschnitt befinden
    - Man spricht von **wechselseitigem Ausschluss**

# Wechselseitiger Ausschluss

---

## > Forderungen an eine Implementierung für den wechselseitigen Ausschluss

- Zu jedem Zeitpunkt darf sich nur ein Prozess in seinem kritischen Abschnitt befinden (Basisforderung)
- Es dürfen keine Annahmen über die Ausführungsgeschwindigkeiten, die Anzahl der unterliegenden Prozessoren oder den Scheduling-Algorithmus getroffen werden
- Kein Prozess, der sich nicht in seinem kritischen Abschnitt befindet, darf andere Prozesse blockieren (Fortschritt)
- Kein Prozess darf unendlich lange warten müssen, bis er in seinen kritischen Abschnitt eintreten kann (Verhungern)

# Idee für wechselseitigen Ausschluss: Sperrvariable

- > Verwendung einer “Sperrvariable”
- > Prolog-/Epilog-Paar um jeden kritischen Abschnitt
  - Aktives Warten auf Eintritt in den kritischen Abschnitt (engl. *busy waiting*)
- > Herausforderungen
  - Alle Prozesse müssen sich an das Vorgehen halten
  - Aktives Warten für einen längeren Zeitraum verschwendet Prozessorzeit

```
1  Lock lock;
2
3  void push (struct list *l, int data) {
4      // Neues element erzeugen
5      struct node *el = malloc (...);
6      el->data = data;
7      el->next = NULL;
8
9      // Kritischen Abschnitt betreten
10     enter (&lock);
11
12     // Neues element in liste einfuegen
13     *l->tail = el;
14     l->tail = &el->next;
15
16     // Kritischen Abschnitt verlassen
17     leave (&lock);
18
19     return;
20 }
```

# Naive Implementierung

- > Umsetzung: Sperren und öffnen kritischer Abschnitte mittels **Schloß-Variable**

```
1  /* Datentyp fuer lock-Variable
2  */
3  typedef unsigned char Lock;
4
5  /* Kritischen Abschnitt betreten
6   * Warten, bis lock "offen", dann selbst "verschließen"
7   */
8  void enter (Lock* lock) {
9      while (*lock)
10         ;
11
12     *lock = 1;
13 }
14
15 /* Kritischen Abschnitt verlassen
16 * Lock auf "offen" setzen
17 */
18 void leave (Lock* lock) {
19     *lock = 0;
20 }
```

# Diskussion: Naive Implementierung

- > Problem: enter() ist selbst kritisch!

```
1  /* Datentyp fuer lock-Variable
2  */
3  typedef unsigned char Lock;
4
5  /* Kritischen Abschnitt betreten
6   * Warten, bis lock "offen", dann selbst "verschließen"
7   */
8  void enter (Lock* lock) {
9      while (*lock)
10         ;
11
12     *lock = 1;
13 }
14
15 /* Kritischen Abschnitt verlassen
16 * Lock auf "offen" setzen
17 */
18 void leave (Lock* lock) {
19     *lock = 0;
20 }
```

Erfüllt die Anforderung #1 an  
Wechselseitigen Ausschluß nicht!

Hier keine Unterbrechung erlaubt!

# Idee: Vermeidung von Kontextwechseln

- > Das Problem entsteht durch die Unterbrechung während kritischer Tätigkeiten
- > Lösung
  - Unterbrechungen vermeiden → Deaktivieren von Interrupts!
    - Jeder Prozess sperrt vor Eintritt in seinen kritischen Abschnitt alle Unterbrechungen
    - Jeder Prozess lässt die Unterbrechungen am Ende seines kritischen Abschnitts wieder zu

```
1  /* Kritischen Abschnitt betreten
2  * Interrupts deaktivieren
3  */
4  void enter (void) {
5      asm ("cli");
6  }
7
8  /* Kritischen Abschnitt verlassen
9  * Interrupts aktivieren
10 */
11 void leave (void) {
12     asm("sti");
13 }
```

Mittels “cli” und “sti” können x86-CPU  
angewiesen werden, Interrupts  
zu deaktivieren bzw. aktivieren

# Diskussion: Vermeidung von Kontextwechseln

---

- > Das Problem entsteht durch die Unterbrechung während kritischer Tätigkeiten
- > Lösung
  - Unterbrechungen vermeiden → Deaktivieren von Interrupts!
    - Jeder Prozess sperrt vor Eintritt in seinen kritischen Abschnitt alle Unterbrechungen
    - Jeder Prozess lässt die Unterbrechungen am Ende seines kritischen Abschnitts wieder zu
- > Probleme
  - Unbrauchbar für allgemeine Benutzerprozesse, da nicht zugesichert werden kann, dass sie die Interrupts auch wieder zulassen (z.B. wegen Programmierfehler)
  - Lösung unbrauchbar im Falle eines Multiprozessor-Systems, da sich die Interrupt-Sperre i.d.R. nur auf einen Prozessor auswirkt
- > Diese Lösung wird häufig innerhalb des Betriebssystemkerns selbst eingesetzt, um wechselseitigen Ausschluss zwischen Kernroutinen zu gewährleisten (z.B. im alten Einprozessor-UNIX-Kern, um wechselseitigen Ausschluss mit einem Interrupt-Handler sicherzustellen)

# Idee: Striktes Alternieren

- > Umsetzung: Die Prozesse wechseln sich strikt ab (A, B, A, B, ...)
  - Kritisch ist nur eine einzige Zuweisungsoperation

## Thread 0 (A)

```
while (1) {  
    while (turn != 0)  
        ;  
  
    do_critical_things ();  
  
    /* Gestatte dem anderen Thread  
     * (Thread 1) die Ausführung.  
     */  
    turn = 1;  
}
```

## Thread 1 (B)

```
while (1) {  
    while (turn != 1)  
        ;  
  
    do_critical_things ();  
  
    /* Gestatte dem anderen Thread  
     * (Thread 0) die Ausführung.  
     */  
    turn = 0;  
}
```

# Diskussion: Striktes Alternieren

---

- > Umsetzung: Die Prozesse wechseln sich strikt ab (A, B, A, B, ...)
    - Kritisch ist nur eine einzige Zuweisungsoperation
  - > Problem
    - Skaliert nicht wirklich gut
    - Das Warten wird aktiv durchgeführt (→ Verschwenden von CPU-Zeit)
    - Die Fortschrittsbedingung (#3) kann verletzt sein, falls einer der Prozess wesentlich langsamer ist als der andere
- keine ernsthafte Lösung

Valide Lösungen

# Valide Lösungen in Software

- > Algorithmus von Dekker (1965)
  - Erste bekannte korrekte Lösung für 2 Prozesse
- > Algorithmus von Peterson (1981)
  - Einfachere, elegante Lösung
- > Algorithmus von Lamport (“Bäckerei”-Algorithmus)
- > Weitere Lösungen von Dijkstra, Knuth, Eisenberg/McGuire, ...

Dekker

```
// Thread 0
wants_to_enter[0] = true;
while (wants_to_enter[1]) {
    if (last_thread == 0) { //°
        wants_to_enter[0] = false;
        while (last_thread == 0)
            ;
        wants_to_enter[0] = true;
    }
}

// hier kritischen Abschnitt einfügen
last_thread = 0;
wants_to_enter[0] = false;
```

```
// Thread 1
wants_to_enter[1] = true;
while (wants_to_enter[0]) {
    if (last_thread == 1) { //°
        wants_to_enter[1] = false;
        while (last_thread == 1)
            ;
        wants_to_enter[1] = true;
    }
}

// hier kritischen Abschnitt einfügen
last_thread = 1;
wants_to_enter[1] = false;
```

Peterson

```
1  #define N 2
2
3  volatile int turn;
4  volatile int interested[N];
5
6  void enter_region (int process) {
7      int other;
8      other = 1 - process;
9      interested[process] = 1;
10     turn = other;
11
12     // Aktives warten
13     while (interested[other] == 1 && turn == other)
14         ;
15 }
16
17 void leave_region (int process) {
18     interested[process] = 0;
19 }
```

# Valide Lösungen mit Hardwareunterstützung

- > Viele CPUs unterstützen **atomare (unteilbare) Operationen** zur Realisierung von „Schloß-Algorithmen“

- **Motorola 68K: Test-and-Set**

```
acquire TAS lock
      BNE acquire
```

- **Intel x86: XCHG**

```
      mov ax,1
acquire: chg ax,lock
      cmp ax,0
      jne acquire
```

- **PowerPC: LL/SC**

- ...

Aktives und Passives Warten

# Aktives Warten

---

- > Alle bisher gezeigten Lösungen (außer Interrupts deaktivieren) nutzten **Aktives Warten**
  - Dies ist problematisch, da
    - der Prozess selbst nicht zur Lösung beiträgt, sondern nur wartet
    - andere Prozesse durch den Wartenden behindert werden
    - es zu Verklemmung kommen kann (→ Alle warten aufeinander)
    - Es zu Prioritätsumkehr kommen kann
- > Innerhalb des Betriebssystemkerns wird zum Teil aktives warten genutzt
  - In Linux ist sog. **futex (fast user-space locking)** verfügbar
    - Genutzt von allen gängigen Synchronisationsprimitiven in Linux → Mutex, Semaphor, Barrier, wsw.
    - Falls ein Futex frei ist, kann er gesperrt werden, ohne dass dazu ein Kontextwechsel nötig ist

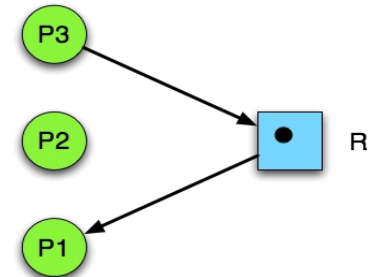
# Problem: Prioritätsumkehr

## > Prioritätsumkehr

- Kann trotz korrekter Lösungen mit aktivem Warten (Peterson, Test-and-Set) auftreten
- Kann unter Umständen auch in Lösungen mit passivem Warten auftreten

## > Vereinfachtes Beispiel

- **Prozess P1-P3:** P3: hohe Priorität, Prozess P2: mittlere Priorität, Prozess P1: niedere Priorität
  - **Ressource R:** Prozesse P1 und P3 benötigen Zugriff auf die
  - **Prioritätsscheduling:** Falls ein höher priorisierter Prozess rechnen will, erhält er Vorrang
  - Annahme: P1 rechnet innerhalb seines krit. Abschnitt (belegt Resource R) und P3 wird rechenwillig
    - Der Scheduler verdrängt P1 nicht, bevor dieser seinen kritischen Abschnitt verlassen hat
  - Annahme: P1 rechnet innerhalb seines krit. Abschnitt (belegt Resource R) und P2 wird rechenwillig
    - Der Scheduler verdrängt P1 sofort und lässt P2 rechnen
    - Falls nun P3 rechenwillig wird, kann der Scheduler ihn nicht einplanen da P1 noch innerhalb des krit. Abschnitts die Resource R blockiert
    - Er kann aber auch P1 nicht einplanen, da dies die Regeln des Prioritätsscheduling verletzen würde
- > Insbesondere in Echtzeitbetriebssystemen ist eine Lösung hierfür extrem wichtig. Bis heute nicht zufriedenstellend gelöst



Prioritäten: P3 > P2 > P1

# Passives Warten

---

- > Es wird eine Lösung benötigt, die kein aktives warten voraussetzt
- > **Passives Warten**
  - Prozesse geben die Kontrolle an das Betriebssystem ab, während sie warten
    - Prozess blockiert sich selbst (wie im Falle eines I/O-Stoßes)
  - Scheduler berücksichtigt den Prozess nicht in der Ablaufplanung, bis das Ereignis eintritt, auf das gewartet wird
  - Andere Prozesse können in dieser Zeit ausgeführt werden
  - Unter Umständen sind irgendwann alle Prozesse blockiert

Betriebssystemunterstützung

# Betriebssystemunterstützung

---

- > Passives warten mit Betriebssystemunterstützung
- > Primitive
  - **SLEEP()** blockiert den ausführenden Prozess, bis er von einem anderen Prozess geweckt wird
  - **WAKEUP(process)** weckt den Prozess process. Der ausführende Prozess wird dabei nie blockiert
- > Diese Primitive können auch der allgemeinen ereignisorientierten Kommunikation dienen

# Mutex-Locks

---

- > Der Begriff **Mutex** ist von *mutual exclusion* abgeleitet
- > Ein Mutex offeriert die Operationen
  - **Lock** als Prolog-Operation zum Betreten des kritischen Abschnitts
  - **Unlock** als Epilog-Operation beim Verlassen des kritischen Abschnitts
- > Gelegentlich wird angenommen, dass unlock alle wartenden Prozesse entblockiert und sich diese dann erneut um das Betreten des kritischen Abschnitts bewerben
- > Mutexe können als **Spezialfall von Semaphoren** angesehen werden

# Beispiel: Pthread Mutex

---

- > **pthread\_mutex\_t fastmutex = PTHREAD\_MUTEX\_INITIALIZER;**
  - Anlegen einer Mutex-Variablen (mehrere Varianten)
- > **pthread\_mutex\_init (pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr);**
  - Initialisieren einer Mutex-Variablen
- > **int pthread\_mutex\_lock (pthread\_mutex\_t \*mutex);**
  - Lock anfordern
- > **int pthread\_mutex\_trylock (pthread\_mutex\_t \*mutex);**
  - Lock anfordern, falls ohne Blockieren möglich
- > **int pthread\_mutex\_unlock (pthread\_mutex\_t \*mutex);**
  - Lock freigeben
- > **int pthread\_mutex\_destroy (pthread\_mutex\_t \*mutex);**
  - Lock zerstören

# Beispiel: Mutex

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <pthread.h>
4
5  /* A global mutex variable
6   */
7  pthread_mutex_t lock;
8
9  /* The function executed by each thread
10 */
11 void* tfunc (void*) {
12     pthread_mutex_lock (&lock);
13     // do critical work here ...
14     pthread_mutex_unlock (&lock);
15 }
16
17 int main (int argc, char **argv) {
18     /* Initialize the mutex
19     */
20     if (pthread_mutex_init (&lock, NULL) != 0) {
21         perror ("mutex_init");
22         exit (EXIT_FAILURE);
23     }
24
25     // Start threads, do stuff ...
26     // [...]
27 }
```

Globale Mutex Variable

Kritischen Abschnitt betreten

Kritischen Abschnitt verlassen

Mutex initialisieren

# Semaphore

---

- > Ein Semaphore  $s$  ist ein “Objekt” das eine lokale Integer-Variable  $count$  enthält, die mit einer nicht-negativen Zahl  $n$  initialisiert wird, und das die folgenden atomaren (unteilbaren) Operationen offeriert
  - **P** (niederländ. *prolaag*, „versuche zu erniedrigen“; auch *down*, *wait*)
    - Falls  $count = 0$ : Der laufende Prozess wird blockiert
    - Falls  $count > 0$ : Der Wert des Semaphore wird um 1 dekrementiert
  - **V** (niederländ. *verhoog*, „erhöhe“; auch *up*, *signal*)
    - Der Wert des Semaphore wird um 1 inkrementiert
    - Ein ggf. auf den Semaphore blockierter Prozess wird deblockiert
- > Betriebssystemabstraktion, für den Austausch von Synchronisationssignalen
  - Statt der Nutzung von aktivem Warten können Prozesse passiv Warten (P-Operation)
  - Andere Prozesse signalisieren, wenn sie z.B. einen krit. Abschnitt freigeben (V-Operation)
- > Abgrenzung zu Mutex
  - Mutex ist ein Spezialfall von Semaphore
  - Wert des Mutex wechselt nur binär (wechselseitiger Ausschluß)
  - Mutex hat einen Besitzer-Kontext: Nur der Prozess, der einen Mutex sperrt, kann ihn entsperren

# Beispiel: Semaphore

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <semaphore.h>
4
5  /* A global semaphore variable
6   */
7  sem_t s;
8
9  int main (int argc, char **argv) {
10     /* Initialize the semaphore
11     */
12     if (sem_init (&s, 0, 1) != 0)
13         exit (EXIT_FAILURE);
14
15     // Start threads, do stuff ...
16
17     if (sem_destroy (&s) != 0)
18         exit (EXIT_FAILURE);
19
20     // [...]
21 }
22
23 /* The function executed by each thread
24 */
25 void* tfunc (void*) {
26     sem_wait (&s);
27     // do critical work here ...
28     sem_post (&s);
29 }
```

Globale Variable für den Semaphore

- Eine globale Variable eignet sich bei Threads
- Synchronisation von Prozessen würde erfordern, dass sich Der Semaphore z.B. in einem *shared memory* Segment befindet

Initialisierung des Semaphore auf 1 (3ter Parameter = 1)

- Es kann sich genau 1 Prozess im kritischen Abschnitt befinden
- Der 2te Parameter gibt an, dass es sich um einen Semaphore für die Synchronisation von Threads handelt.

Entfernen des Semaphore, falls dieser nicht mehr benötigt wird.

Kritischen Abschnitt betreten.

- der erste Prozess der den krit. Abschnitt betritt dekrementiert Zähler des Semaphore auf 0. Alle weiteren Prozesse blockieren.

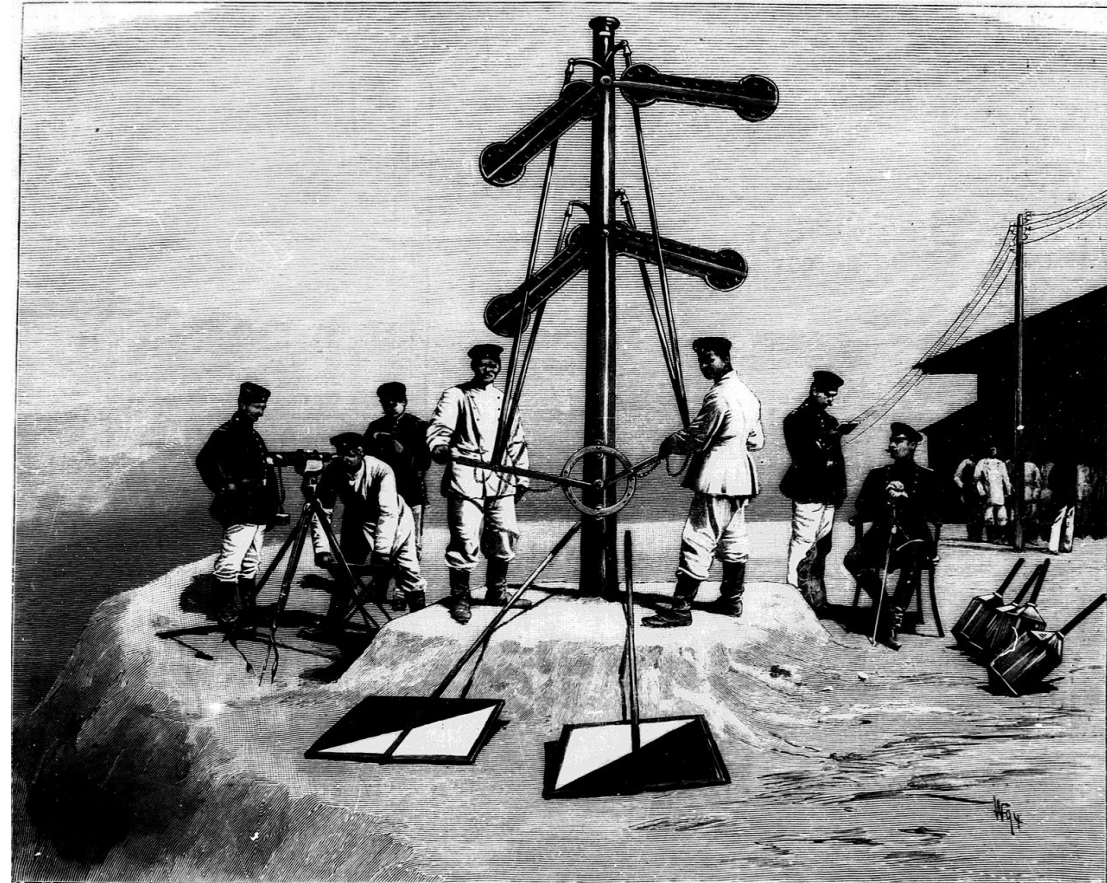
Kritischen Abschnitt verlassen

- dabei wird entweder ein wartender Prozess geweckt, oder der Zähler inkrementiert.

# Semaphore: Anwendung

## > Anwendungsfälle

- Wechselseitiger Ausschluss
- Betriebsmittelvergabe
- Einseitige Synchronisation
- “Verstecken von Interrupts”
- Durchsetzen einer Vorrangrelation



Die optische Telegraphie und der Signaldienst in den europäischen Armeen: Semaphore-Apparat in der preussischen Armee.  
Originalzeichnung von Adolf Seib.

# Anwendungsfall: Betriebsmittelvergabe

- > Es seien  $n$  Exemplare vom Betriebsmitteltyp  $bm$  vorhanden
  - Es dürfen maximal  $n$  Prozesse einen kritischen Abschnitt betreten, der das Betriebsmittel  $bm$  erfordert

- > Idee

- Dem Betriebsmittel-Typ wird ein Semaphore  $bm$  zugeordnet
- Ein Prozess beantragt das Betriebsmittel durch  $P(bm)$
- Weitere Prozesse können ebenfalls  $P(bm)$  aufrufen, bis der Semaphor den Wert 0 erreicht
- Falls alle Betriebsmittel blockiert sind (Semaphor  $bm = 0$ ) blockiert auch ein  $P(bm)$  aufrufender Prozess
- Nach der Benutzung gibt der Prozess das Betriebsmittel per  $V(bm)$  wieder frei

- > Hinweis

- $P() == \text{down}() == \text{wait}() == \text{sem\_wait}()$
- $V() == \text{up}() == \text{signal}() == \text{sem\_post}()$

Betriebssystemetheorie

Auch zu finden  
sowie `notify()`, `uvm`.

POSIX (IEEE Std 1003.1)

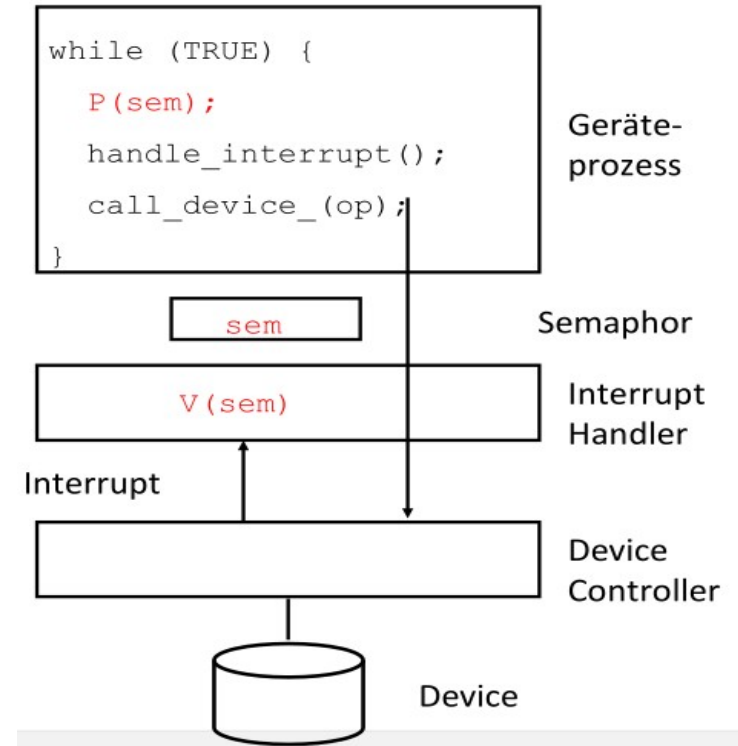
```
1  #include <semaphore.h>
2
3  #define NUM_RESOURCE 5
4
5
6  sem_t bm = NUM_RESOURCE;
7
8  int main (void) {
9      // [...]
10
11      P (&bm);
12
13      // Use resource bm
14
15      V (&bm);
16
17      // [...]
18 }
```

# Anwendungsfall: Verstecken von Interrupts

- > Interrupt-Behandlungsroutine wird blockiert bis ein Interrupt eingeht
  - Passives Warten auf ein externes Ereignis mittels Semaphore

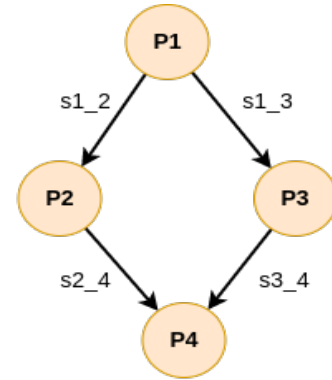
- > Idee

- Jedem I/O-Gerät wird ein Geräteprozess und ein mit 0 initialisiertes Semaphore zugeordnet
- Nach dem Start des Geräteprozesses führt dieser eine P-Operation auf dem Semaphore aus (und blockiert)
- Im Falle eines Interrupts des Gerätes führt der Interrupt Handler eine V-Operation auf dem Semaphore aus.
- Der Geräteprozess wird entblockiert (und dadurch rechenwillig) und kann das Gerät bedienen



# Anwendungsfall: Durchsetzen einer Vorrangrelation

- > Mehrere Prozesse müssen in einer koordinierten Reihenfolge arbeiten
  - Es sei  $P$  eine Menge kooperierender Prozesse und  $<$  eine partielle Ordnung auf der Menge  $P$
  - $P1 < P2$  := Prozess  $P1$  muss vor Prozess  $P2$  ausgeführt werden
  - Hinweis
    - $<$  wird **Vorrangrelation** genannt
    - In einem Graph wird  $P1 < P2$  häufig durch  $P1 \rightarrow P2$  dargestellt
- > Idee
  - Jeder Vorrangbeziehung  $P1 < P2$  wird ein Semaphore  $s$  zugeordnet, auf der  $P1$  die Operation  $V(s)$  und  $P2$  die Operation  $P(s)$  aufruft
  - Alle Semaphore werden mit  $0$  initialisiert
  - Alle Prozesse können gleichzeitig gestartet werdenDie Vorrangrelation wird korrekt durchgesetzt



```
P1 () {  
    // do work  
  
    V (s1_2);  
    V (s1_3);  
  
    exit (0);  
}
```

```
P2 () {  
    P (s1_2);  
  
    // do work  
  
    V (s2_4);  
  
    exit (0);  
}
```

```
P3 () {  
    P (s1_3);  
  
    // do work  
  
    V (s3_4);  
  
    exit (0);  
}
```

```
P4 () {  
    P (s2_4);  
    P (s3_4);  
  
    // do work  
  
    exit (0);  
}
```

# Diskussion: Semaphore

---

- > Nutzung mittels Systemaufrufen blockieren/signalisieren
  - Linux: futex – u.a. kurzes aktives warten, dann blockieren
- > Wesentlich ist die Unteilbarkeit in der Implementierung von P() und V():
  - Auf Einprozessorsystemen: Unteilbarkeit kann durch Sperren aller Unterbrechungen während der Ausführung von P() und V() erreicht werden. Zulässig, da nur wenige Maschineninstruktionen zur Implementierung nötig sind.
  - Auf Multiprozessorsystemen: Jedem Semaphor wird eine mittels Test-and-Set realisierte Sperrvariable mit Aktivem Warten vorgeschaltet. Hierdurch kann zu jedem Zeitpunkt nur höchstens ein Prozessor das Semaphor manipulieren.
  - Beachte: Es besteht ein großer Unterschied zwischen Aktivem Warten auf den Zugang zum Semaphor, das einen kritischen Abschnitt schützt (wenige Instruktionen, Mikrosekunden) und Aktivem Warten auf den Zugang zum kritischen Abschnitt selbst (problemabhängig, Zeitdauer nicht vorab bekannt oder begrenzt).
- > Nachteile / Probleme
  - Es besteht die Gefahr der Verklemmung
  - Hohe Komplexität in der Implementierung
  - Semaphorbenutzung wird nicht erzwungen
    - Alle kooperierenden Prozesse müssen die Protokolle exakt einhalten
    - Semaphoroperationen können in der gesamten Anwendung verstreut sein

# Einschub: Windows

---

## > Windows Critical Sections

- Wechselseitiger Ausschluss von Threads eines Prozesses mittels passivem Warten  
(mehr dazu: <https://learn.microsoft.com/en-us/windows/win32/sync/critical-section-objects>)

## > Windows Mutex

- Systemobjekte für den wechselseitigen Ausschluss zwischen Threads beliebiger Prozesse  
(mehr dazu: <https://learn.microsoft.com/en-us/windows/win32/sync/mutex-objects>)

## > Windows Semaphore

- Semaphore Objekte unter Windows (nächstes Thema)  
(mehr dazu: <https://learn.microsoft.com/en-us/windows/win32/sync/semaphore-objects>)

# Monitore

---

- > Ein **Monitor** (*Dijkstra (1971), Hansen (1972); formal definiert von Hoare (1974)*)  
stellt einen **abstrakten Datentyp** dar, der **explizite Synchronisationseigenschaften** besitzt
- > **Aufbau und Funktionsweise**
  - Prozeduren bzw. Funktionen und Datenstrukturen
    - vergleichbar mit einer Klasse (z.B. in Java)
    - Prozesse können die Prozeduren des Monitors nutzen, aber nicht direkt auf dessen Datenstrukturen zugreifen
  - Monitor stellt sicher, dass zu jedem Zeitpunkt jeweils nur ein Prozess die Prozeduren/Funktionen benutzt
- > **Monitore sind ein spezielles Konstrukt einer Programmiersprache**
  - Der Compiler erkennt sie als solche
  - Durchsetzung des wechselseitigen Ausschluß durch den Compiler,
  - Weniger Anfällig für Fehler durch Programmierer:in
- > **Implementierungen**
  - Concurrent Pascal, Modula-3, Java, (Rust), ...

# Synchronisationsprobleme

(Lösung mittels Semaphoren)

# Erzeuger-Verbraucher Problem

- > Ein *Erzeuger* generieren Daten, ein *Verbraucher* konsumiert diese



- > Problem

- Der Erzeuger kann nicht arbeiten, falls die Queue voll ist
- Der Verbraucher kann nur dann arbeiten, wenn die Queue nicht leer ist

- > Lösung

- Falls keine Arbeit vorhanden oder die Queue voll ist, legt sich der jeweilige Prozess schlafen
- Es gibt viele Lösungsansätze für dieses Problem: Semaphore, Monitore, Nachrichtenaustausch, usw.

- > Erweiterung: mehrere Erzeuger, mehrere Verbraucher

# Erzeuger-Verbraucher Problem

## > Beispielhafte Implementierung

- Verkettete Liste als gemeinsame Datenstruktur in einem shared memory Bereich begrenzter Größe (Platz für 5 Items)
- Funktionen `enter_item()` und `remove_item()` wie im Beispiel der FIFO-Liste (enthält kritischen Abschnitt beim einfügen und entfernen)

## > Die folgenden Semaphore werden genutzt

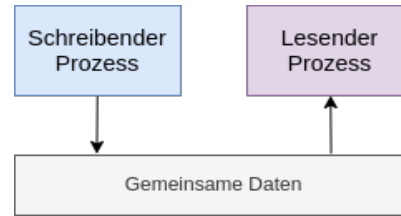
- `empty` Synchronisation von Erzeuger und Verbraucher  
Erzeuger blockiert an `down(&empty)`, falls keine freien Speicherplätze mehr vorhanden sind
- `inuse` Synchronisation von Erzeuger und Verbraucher  
Verbraucher blockiert an `down(&full)`, falls sich keine Elemente in der Liste befinden
- `mutex` Durchsetzung des wechselseitigen Ausschlusses beim Zugriff auf die FIFO-Liste

```
1  #define N 5
2
3  semaphore mutex = 1;
4  semaphore empty = N;
5  semaphore inuse = 0;
6
7  void producer (void) {
8      int item;
9
10     while (1) {
11         produce_item (&item);
12         P (&empty);
13         P (&mutex);
14         enter_item (item);
15         V (&mutex);
16         V (&inuse);
17     }
18 }
19
20 void consumer (void) {
21     int item;
22
23     while (1) {
24         P (&inuse);
25         P (&mutex);
26         remove_item (&item);
27         V (&mutex);
28         V (&empty);
29         consume_item (item);
30     }
31 }
```

# Leser-Schreiber Problem

---

- > Mehrere *Leser* und *Schreiber* greifen auf gemeinsame Daten zu
  - Zu jedem Zeitpunkt dürfen entweder **mehrere Leser oder genau ein Schreiber** zugreifen



- > Beispiel Datenbankserver
  - Viele Leser-Threads sollen gleichzeitig SELECT-Anfragen beantworten können
  - Falls ein Schreiber-Thread eine INSERT-Operation durchführt darf sich kein anderer im krit. Abschnitt befinden

# (Erstes) Leser-Schreiber Problem

- > Beispielhafte Implementierung (Leser haben Vorrang)
  - Der Zugriff auf die gemeinsame Datenstruktur wird durch den Semaphor `daten` geschützt. Falls ein Leser den Semaphor bereits hält, können weitere Leser hinzukommen. Leser haben Vorrang gegenüber Schreibern.
- > Die folgenden Semaphore werden genutzt
  - `mutex` Durchsetzung des wechselseitigen Ausschlusses bei der Verfolgung bzw. Anpassung des `reader_count`
  - `daten` Synchronisation des Zugriffs auf die Daten

```
1 semaphore mutex = 1;
2 semaphore daten = 1;
3 int reader_count = 0;
4
5 void reader (void) {
6     while (1) {
7         P (&mutex);
8         reader_count++;
9         if (reader_count == 1)
10            P (&daten);
11        V (&mutex);
12
13        read_data ();
14
15        P (&mutex);
16        reader_count--;
17        if (reader_count == 0)
18            V (&daten);
19        V (&mutex);
20
21        use_data ();
22    }
23 }
24
25 void writer (void) {
26     while (1) {
27         create_data();
28
29        P (&daten);
30        write_data ();
31        V (&daten);
32    }
33 }
```

# Zusammenfassung

---

- > Es besteht ein **grundsätzlicher Bedarf für Synchronisierung** / Wechselseitigen Ausschluss
  - **Aktives Warten**
    - Funktionierende Softwarelösungen existieren (Dekker, Peterson, usw.)
    - Hardwareunterstützung für atomare Operationen existiert (TAS, XCHG, usw.)
    - Nachteil: Verbrennen von CPU-Zeit, Kein Beitrag zur Lösung des Problems
  - **Passives Warten**
    - Betriebssysteme stellen Systemaufrufe für Synchronisation und Signalisierung bereit
    - **Semaphore** (Spezialfall **Mutex**)  
(Viele Einsatzszenarien: Wechselseitiger Ausschluss, Betriebsmittelvergabe, Einseitige Synchronisation, Durchsetzen einer Vorrangrelation)
    - **Monitore** (Implizite Synchronisationseigenschaften als Teil der Programmiersprache)
- > Das Betriebssystem selbst nutzt *aktives Warten* (sehr vorsichtig und sehr kurz), um zum Beispiel die kritischen Abschnitte von Semaphoren zu sichern
- > Beispiele
  - Erzeuger-Verbraucher-Problem
  - Leser-Schreiber-Problem