Wegweiser

Zusammenspiel/Kommunikation mehrerer Threads

Nutzung gemeinsamen Speichers

- race conditions

➔ Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluß und dessen Durchsetzung

- mit HW-Unterstützung

Ein globaler Kritischer Abschnitt (1)

```
Globale Daten


  lock();

     Arbeite mit globalen Daten

  unlock();
```

Nur ein einziger Thread kann im kritischen Abschnitt sein.
(Lock ohne Parameter).

Ein globaler Kritischer Abschnitt (2)

```
int Kontostand;

bool abbuchen(int Betrag) {

  lock();

  if (Betrag <= Kontostand) {

     Kontostand -= Betrag;

     unlock();
     return true;

  } else {

        unlock();
        return false;
  }
}
```

Kritischer Abschnitt (3)

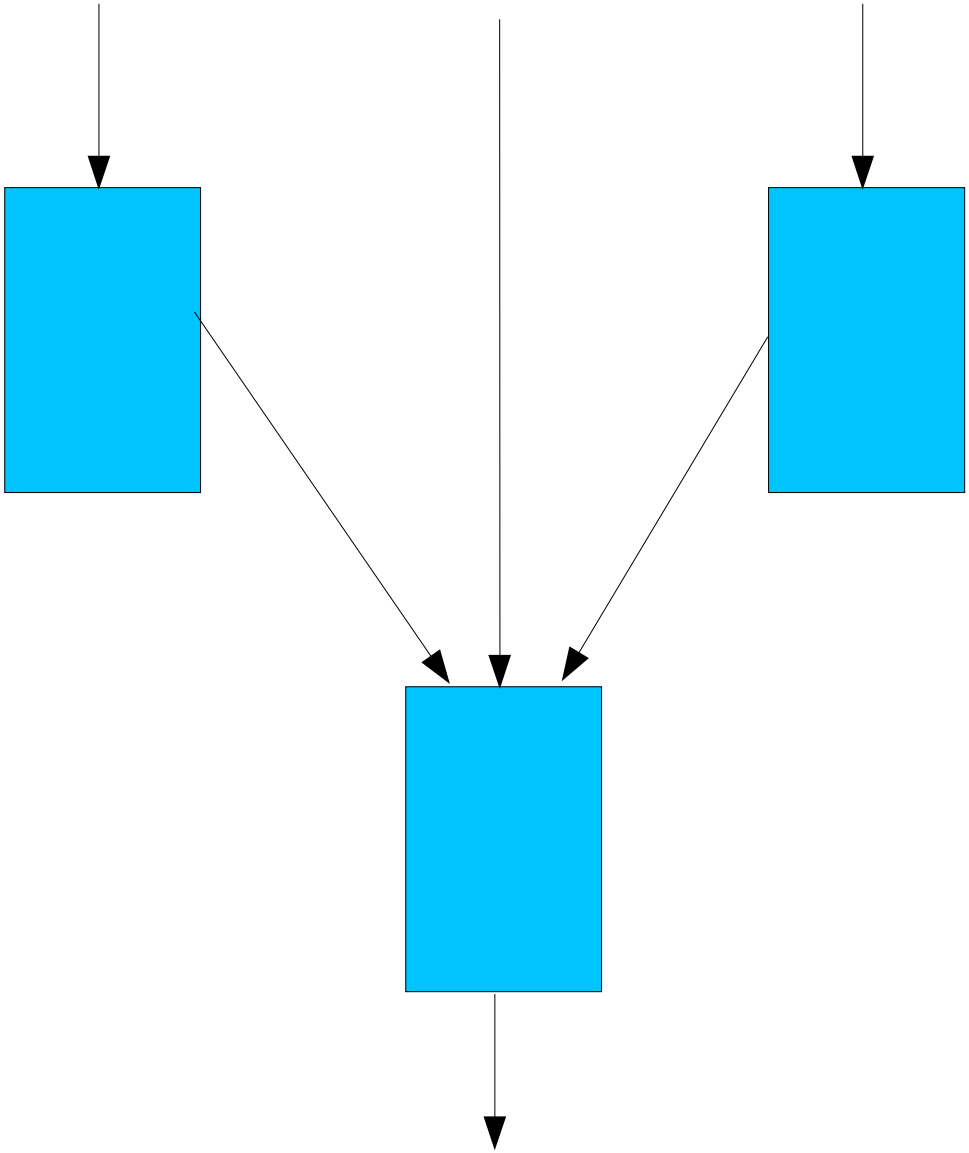
```
int K_S; lock_T Konto_Lock;

bool abbuchen(int Betrag, int * K_S, lock_T *K_Lock) {

  lock(K_lock);

  if (Betrag <= Kontostand) {

     Kontostand -= Betrag;

     unlock(K_lock);
     return true;

  } else {

        unlock(K_Lock);
        return false;
  }
}
```

Kritischer Abschnitt (4)

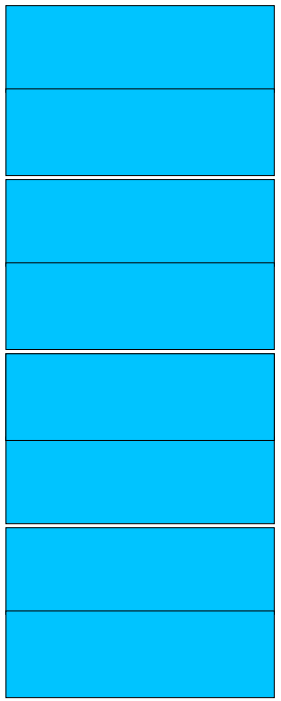
```
int K_S; lock_T Konto_Lock;

bool überweisen(..., lock_T *K1_Lock, lock_T *K2_Lock ) {

  lock(K1_lock); lock(K2_lock);

        ...

  unlock(K1_lock);  unlock(K2_lock);

}
```

→ Transaktionen

Auftreten kritischer Abschnitte

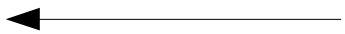


Tabellen:



Lösungsversuch 1 – mit Unterbrechungssperre

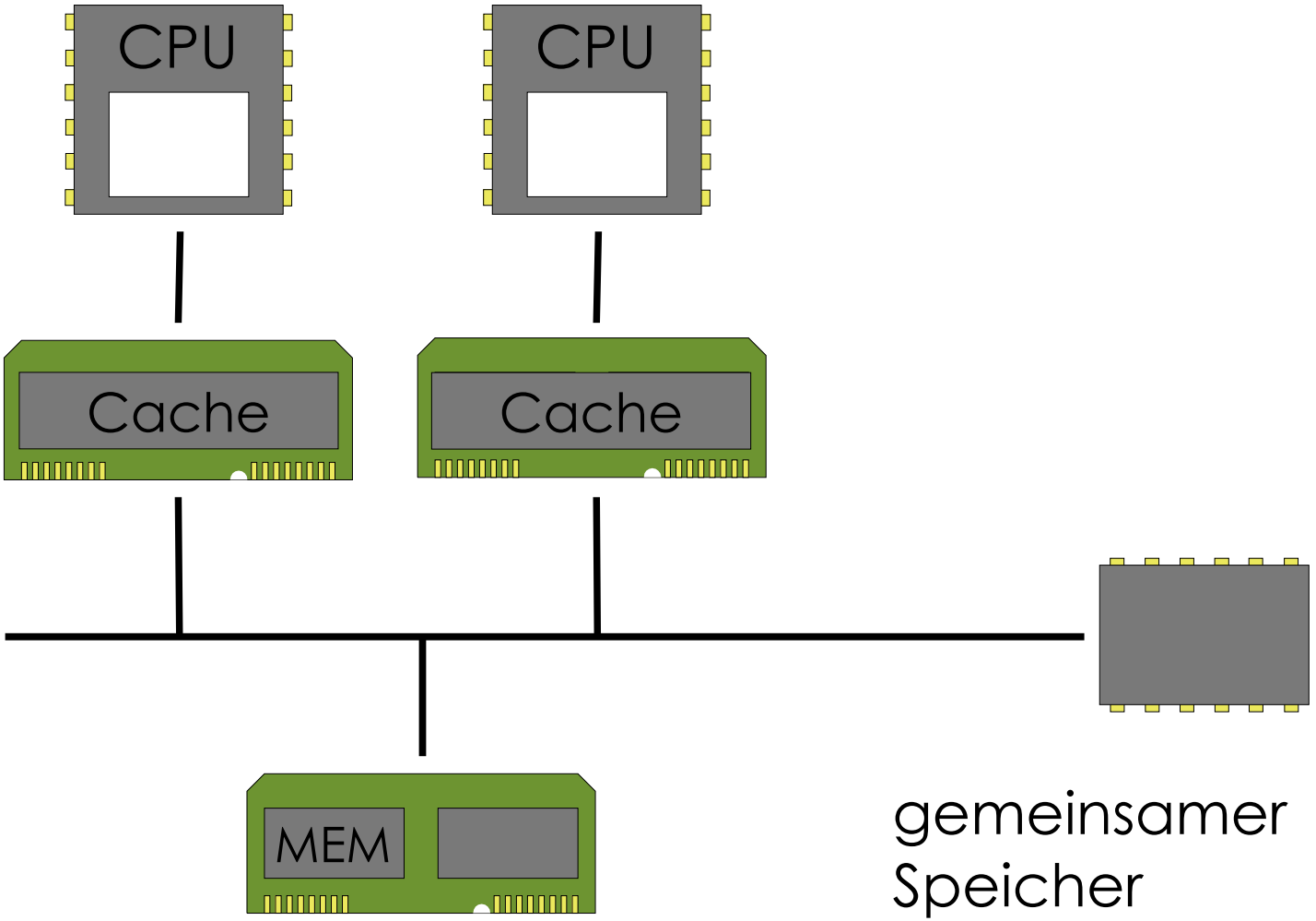
```
T1: ...
  pushf
  cli

  ld    R, Kontostand
  sub   R, Betrag
  sto   R, Kontostand

  popf
```



Globales Lock()

Die Unterbrechungssperre auf der CPU verhindert, dass **T1** im kritischen Abschnitt preemptiert wird.

Einfaches Modell einer Dual-CPU Maschine



CPUs

Bus mit
Arbiter und
Cache-Logik

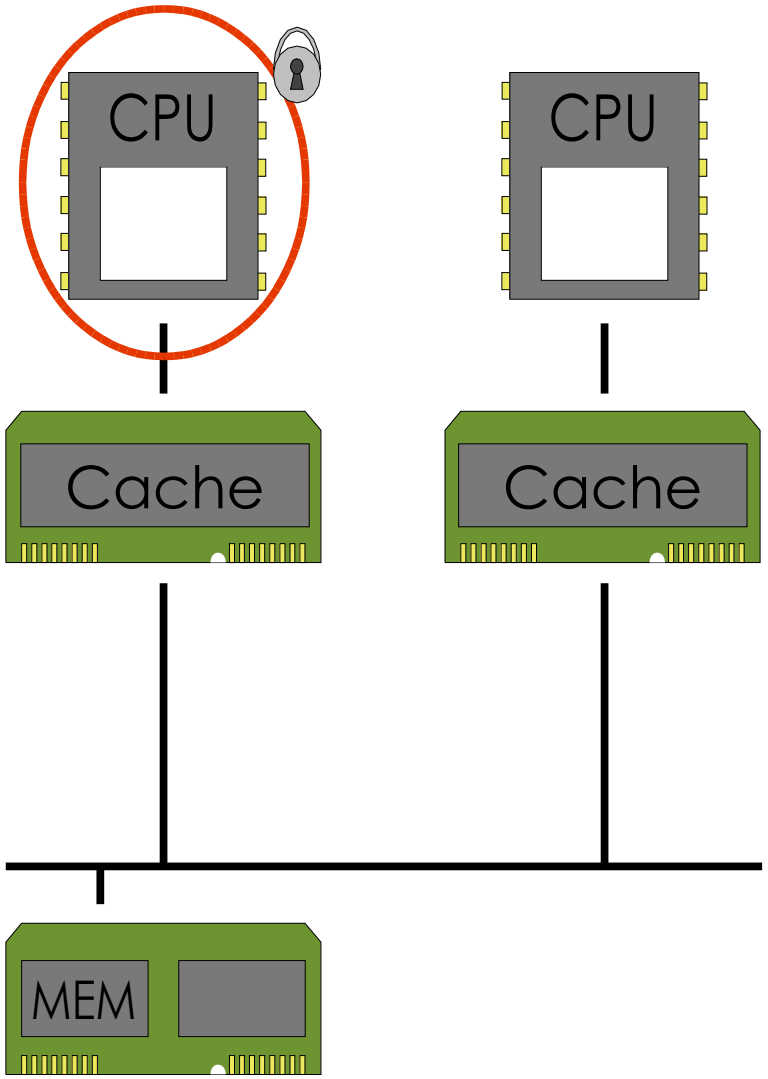
gemeinsamer
Speicher

Lösungsversuch 1 – mit Unterbrechungssperre

```
T1: ...
   pushf
   cli

   ld     R, Kontostand
   sub    R, Betrag
   sto    R, Kontostand

   popf
```

Die Unterbrechungssperre auf der CPU verhindert, dass **T1** im kritischen Abschnitt preemptiert wird.

→ genügt im MP-Fall nicht!



gemeinsamer Speicher

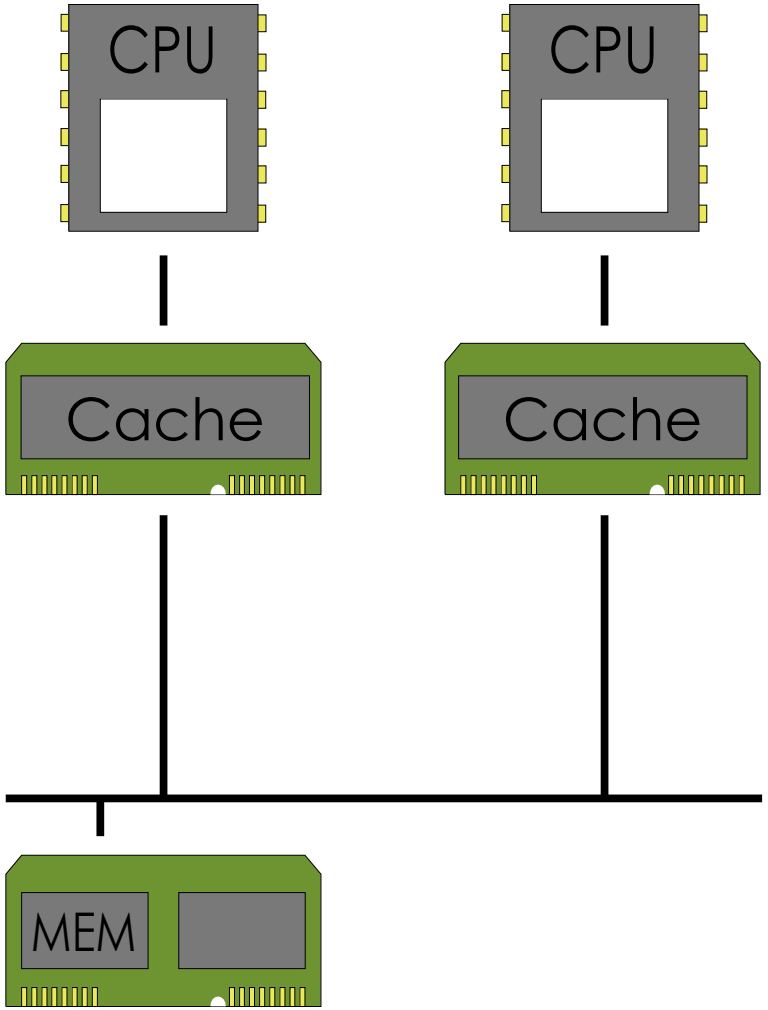
- **Kontostand**

Lösungsversuch 2 – KA mit einer Instruktion

```
Ti: ...

   ld    R, Betrag

   sub   Kontostand, R
```

sub ist nicht unterbrechbar

aber: funktioniert im
MP-Fall auch nicht!
Begründung:
folgende Folien



gemeinsamer Speicher

- **Kontostand**

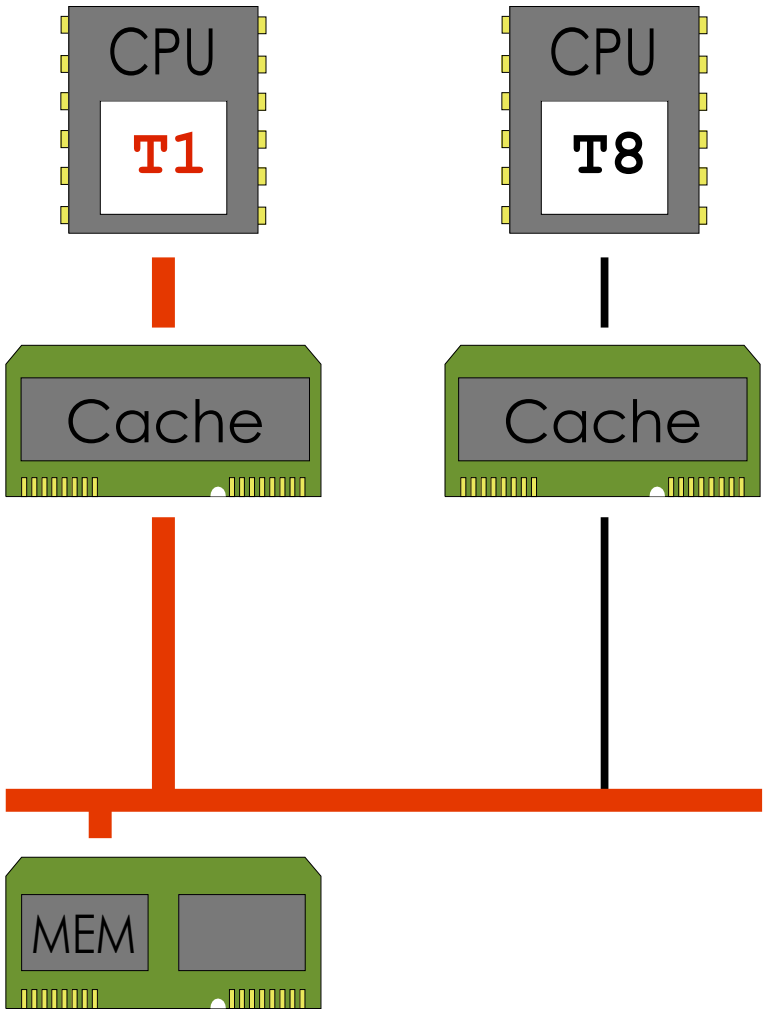
Lösungsversuch 2 – KA mit einer Instruktion

```
Ti: ...

   ld     R, Betrag

   µload     TempR, Kontostand
   µsub      TempR, R
   µstore    TempR, Kontostand
```

- Die Instruktion
  `sub Kontostand, R`
  hat zwei Buszyklen

- Einzelne Buszyklen sind die atomare Einheit !

→ ...



gemeinsamer Speicher
- **`Kontostand`**

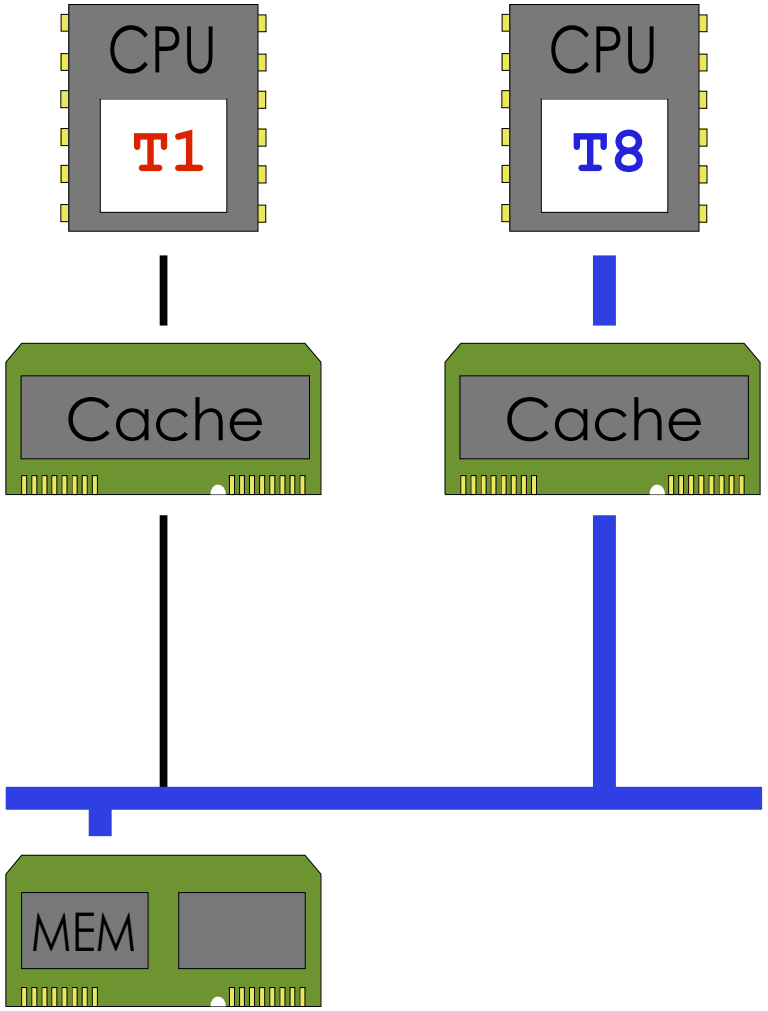
Lösungsversuch 2 – KA mit einer Instruktion

```
Ti: ...

   ld     R, Betrag

   µload     TempR, Kontostand
   µsub      TempR, R
   µstore    TempR, Kontostand
```

- Die Instruktion
  `sub Kontostand, R`
  hat zwei Buszyklen

- Einzelne Buszyklen sind die atomare Einheit !

→ funktioniert so nicht !



gemeinsamer Speicher
- **`Kontostand`**

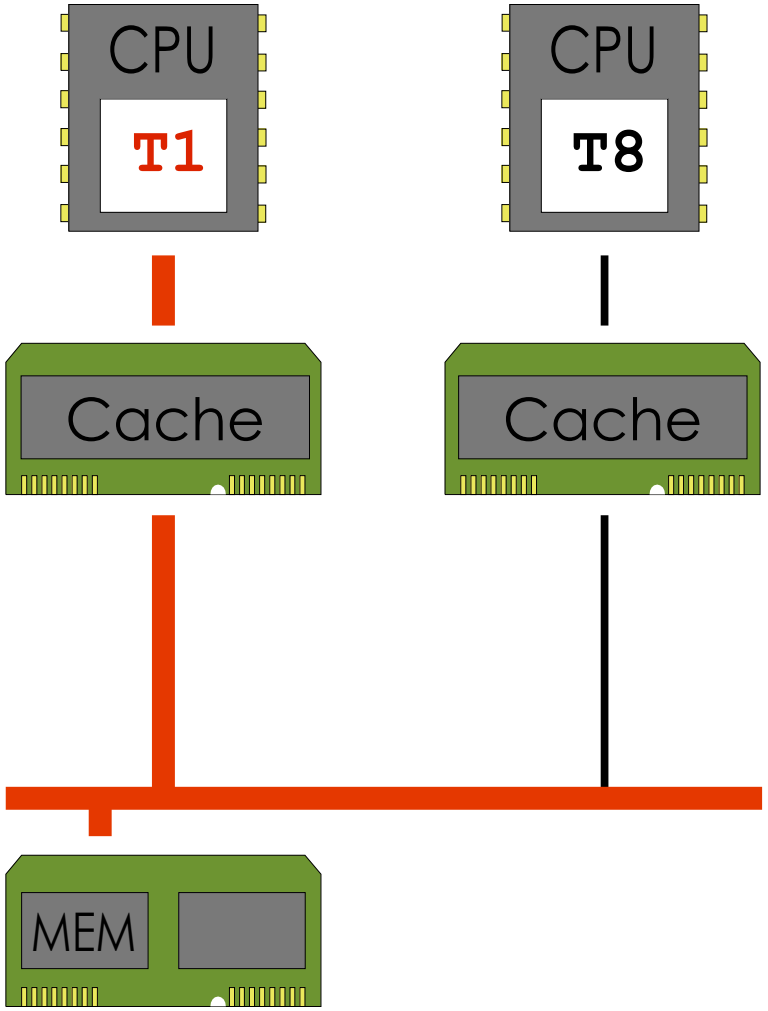
Lösungsversuch 2 – KA mit einer Instruktion

```
Ti: ...

   ld     R, Betrag

   µload      TempR, Kontostand
   µsub       TempR, R
   µstore     TempR, Kontostand
```

- Die Instruktion
  `sub Kontostand, R`
  hat zwei Buszyklen

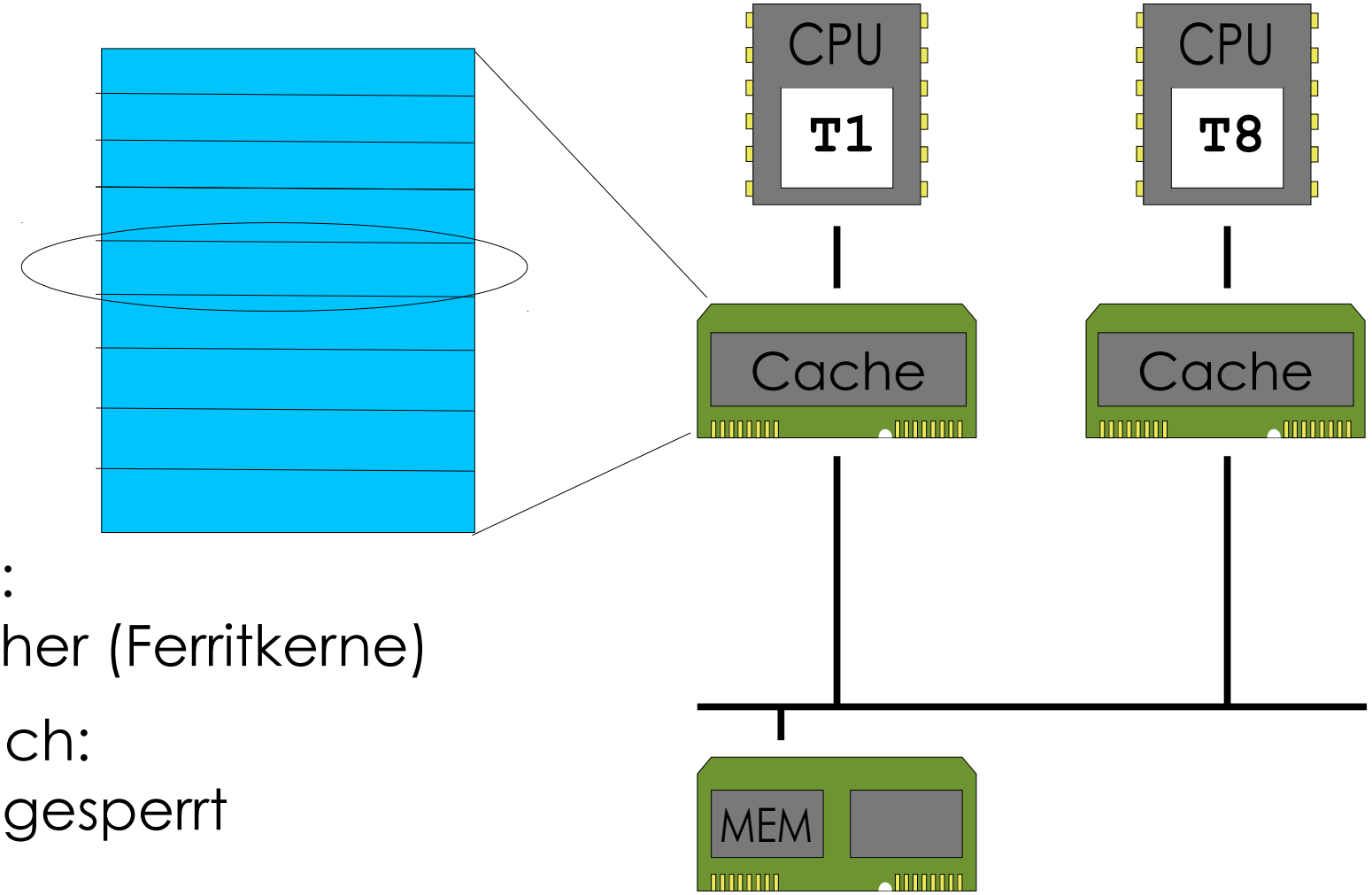
- Einzelne Buszyklen sind die atomare Einheit !

→ funktioniert so nicht !



gemeinsamer Speicher
- **`Kontostand`**

HW-Implementierung Atomare Instruktion



- Ganz alt:
  im Speicher (Ferritkerne)
- Alt/einfach:
  Bus wird gesperrt
- Aktuell:
  Cache-Line wird gesperrt

gemeinsamer Speicher

- **`Kontostand`**

Kritischer Abschnitt: Anforderungen

Bedingungen/Anforderungen

- Keine zwei Threads dürfen sich zur selben Zeit im selben kritischen Abschnitt befinden.
  - ➔ Wechselseitiger Ausschluss ***Sicherheit***

- Jeder Thread, der einen kritischen Abschnitt betreten möchte, muss ihn auch irgendwann betreten können. ***Lebendigkeit***

- Es dürfen keine Annahmen über die Anzahl, Reihenfolge oder relativen Geschwindigkeiten der Threads gemacht werden.

Im Folgenden verschiedene Lösungsansätze ...

Implementierung mittels Unterbrechungssperre

Vorteile

- einfach und effizient

Nachteile

- nicht im User-Mode
- manche KA sind zu lang
- funktioniert nicht auf Multiprozessor-Systemen

Konsequenz

- wird nur in BS für 1-CPU-Rechner genutzt und da nur im Betriebssystemkern

```
lock():
  pushf
  cli       //disable irqs


unlock():
  popf      //restore
```

Implementierung mittels Sperrvariable

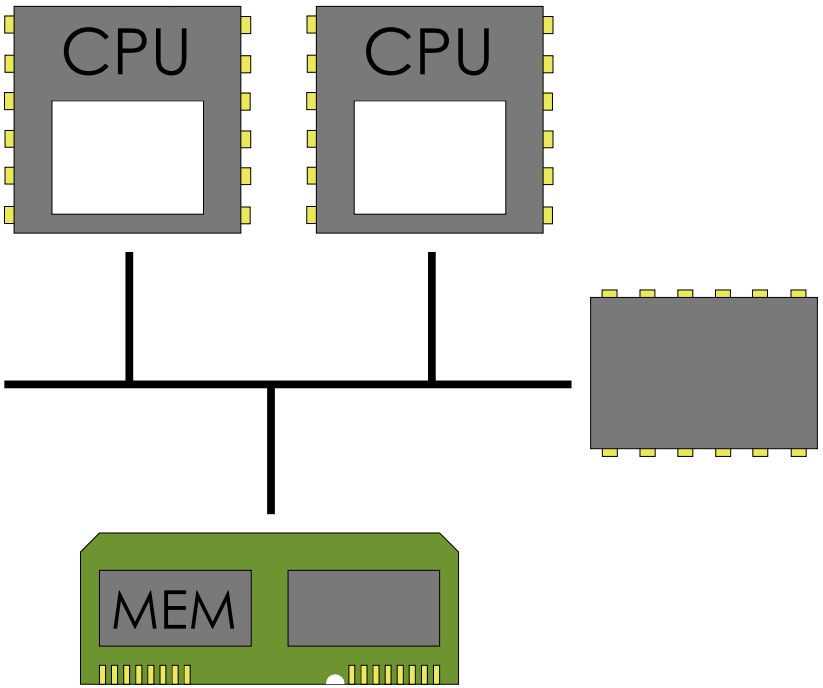
- **funktioniert nicht!**
- warum?

➔ Frage:
welche unteilbare Einheit stellt die HW zur Verfügung?

```
int Lock = 0;
//Lock == 0: frei
//Lock == 1: gesperrt

void lock(int *lock) {

  while (*lock != 0);
  //busy waiting

  *Lock = 1;
}

void unlock(int *lock) {

  *lock = 0;
}
```

RA: unteilbare Operationen der HW

Mehrprozessormaschine



CPUs

Bus/Cache
Logik

gemeinsamer
Speicher

Bus/Cache - Logik implementiert Atomarität für eine Adresse:

- Lesen
- Schreiben
- Lesen und Schreiben

```
test_and_set  R, lock
//R = lock; lock = 1;

exchange   R, lock
//X = lock; lock = R; R = X;
```

Implementierung mit HW Unterstützung

Vorteile

- funktioniert im MP-Fall
- es können mehrere KA so implementiert werden

Nachteile

- busy waiting
- hohe Busbelastung
- ein Thread kann verhungern
- nicht für Threads auf demselben Prozessor

Konsequenz

- geeignet für kurze KA bei kleiner CPU-Anzahl

```
lock:
  test_and_set R, lock
  //R = lock; lock = 1;

  cmp       R, #0
  jnz       lock
  //if (R != 0)
  //  goto lock
  ret


unlock:
  mov       lock, #0
  ret
```

Implementierung für eine CPU im User-Mode

- Unterbrechungssperren:

  sind nicht sicher

  ein Kern-Aufruf (cli) pro „lock“ ist zu teuer

- busy waiting usurpiert die CPU

➔ also .... Kernaufruf nur bei gesperrtem KA (selten)

➔ jedoch race condition: owner u. U. noch nicht richtig gesetzt

```
pid_t owner;
lock_t lock = 0;

void lock(lock_T *lock) {

  while(test_and_set(*lock)){

    Kern.Switch_to (owner);

  }

  owner = AT;
  //aktueller Thread
}

void unlock(lock_T *lock) {

  *lock = 0;
}
```

Alternative: Atomare Operation CAS

```
int cas(lock_t *lock,
    lock_t old, lock_t new){

//echte Implementierung

  unsigned value_found;

  asm volatile(
   "lock; cmpxchgl %1, (%2)"
   : "=a" (value_found)
   : "q"(new), "q"(lock),
     "0"(old)
   : "memory");

  return value_found;
}
```

Alternative: Atomare Operation CAS

```
int cas(lock_t *lock,
    lock_t old, lock_t new){

//echte Implementierung

  unsigned value_found;

  asm volatile(
   "lock; cmpxchgl %1, (%2)"
   : "=a" (value_found)
   : "q"(new), "q"(lock),
     "0"(old)
   : "memory");

  return value_found;
}
```

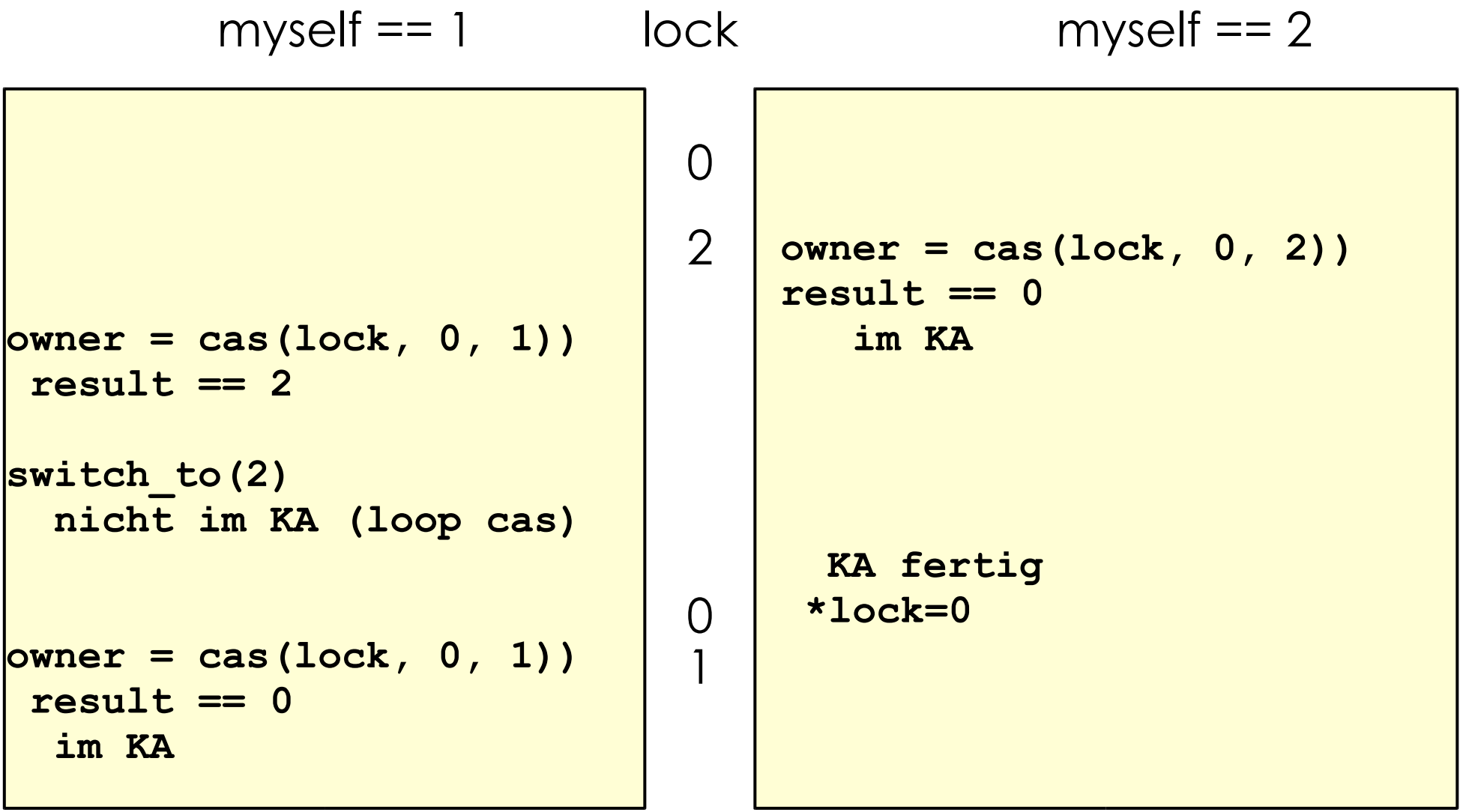
```
lock_t cas(lock_t *lock,
    lock_t old, lock_t new){

// Semantik
// atomar !!

  if (*lock == old) {

     *lock = new;
     return old;

  } else {

     return *lock;
  }
}
```

Alternative: Atomare Operation CAS

```
int cas(lock_t *lock,
   lock_t old, lock_t new){

//anschauliche Variante

  if (*lock == old) {

     *lock = new;
     return old;

  } else {

     return *lock;
  }
}
```

```
void lock(
    lock_t *lock) {

  int owner;

  while (owner =
       cas(lock, 0, myself)){

     Kern.Switch_to (owner);

  }
}

void leavesection(
    lock_t *lock) {

  *lock = 0;
}0
```

Alternative: Atomare Operation CAS



KA: kritischer Abschnitt

Wegweiser

Zusammenspiel/Kommunikation mehrerer Threads

Nutzung gemeinsamen Speichers

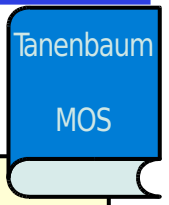
- race conditions

➔ Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluß und dessen Durchsetzung

- ohne HW-Unterstützung möglich ????

Ohne HW-Unterstützung, 2 alternierende Threads

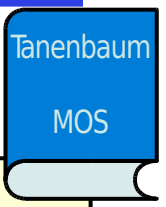



```
CPU 0:

forever {
  do something;

  entersection(0);

    //kritischer Abschnitt

  Leavesection (0)

}
```

```
CPU 1:

forever {
  do something;

  entersection(1);

    //kritischer Abschnitt

  Leavesection (1)

}
```

In dieser Vorlesung nur sehr eingeschränkte Lösungen:
für 2 alternierende Threads
(allgemeine Lösung kompliziert)

Schlechte Lösung




```
CPU 0:

Entersection(0){
  while (blocked == 0){};
}


leavesection(0){
  blocked = 0;
}
```

```
CPU 1:

Entersection(1){
  while (blocked == 1){};
}


leavesection(1){
  blocked = 1;
}
```

Lösung nach Peterson (Vorbetrachtung)

```
CPU0:
entersection(0){


  blocked = 0;

  while (

    (blocked == 0)){};
}
leavesection(0){

}
```

```
CPU1:
entersection(1){


  blocked = 1;

  while (

    (blocked == 1)){};
}
leavesection(1){

}
```

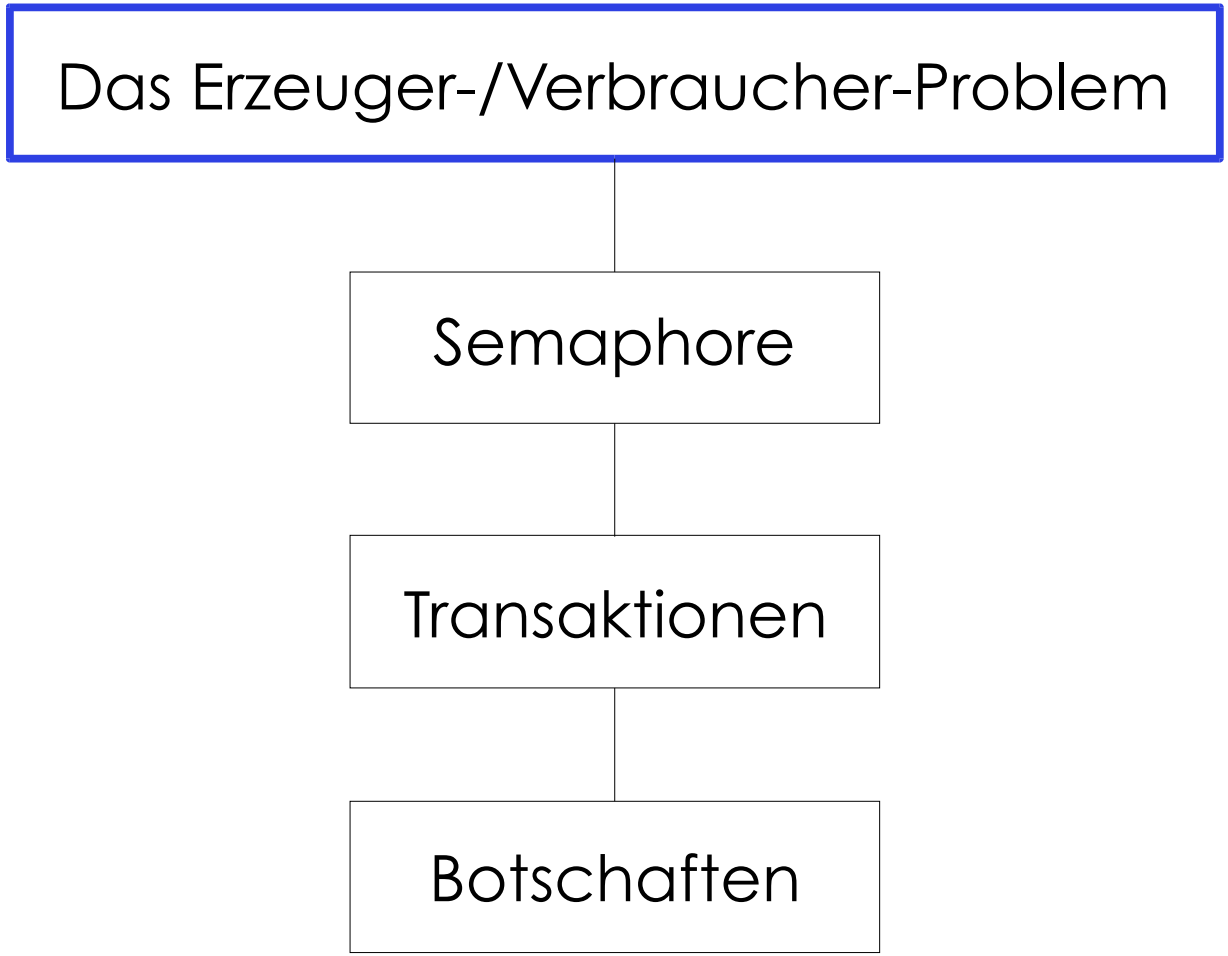
Lösung nach Peterson

```
CPU0:

entersection(0){

  interested[0] = true;
  //Interesse bekunden
  blocked = 0;

  while (
   (interested[1]==true)&&
   (blocked == 0)){};
}
leavesection(0){
  interested[0] = false;
}
```

```
CPU1:

entersection(1){

  interested[1] = true;
  //Interesse bekunden
  blocked = 1;

  while (
   (interested[0]==true)&&
   (blocked == 1)){};
}
leavesection(1){
  interested[1] = false;
}
```

- „Peterson“ funktioniert **nicht** in Prozessoren mit „weak memory consistency“ (→ Vert. BS)

Wegweiser


Das Erzeuger-/Verbraucher-Problem
Semaphore
Transaktionen
Botschaften

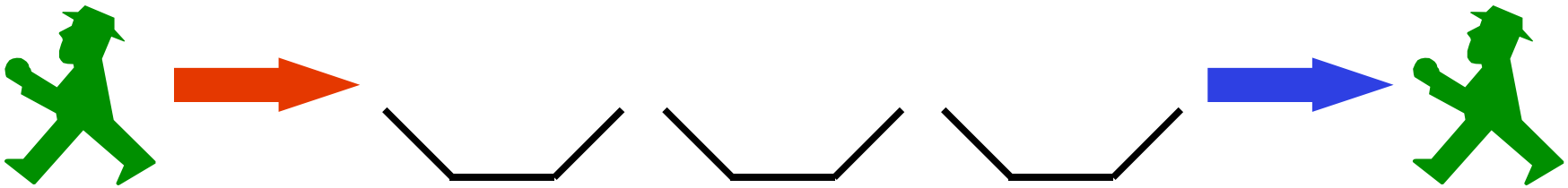
Erzeuger-/Verbraucher-Probleme

Beispiele

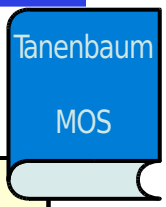
- Betriebsmittelverwaltung
- Warten auf eine Eingabe (Terminal, Netz)

Reduziert auf das Wesentliche

Erzeuger-Thread ***endlicher*** Puffer Verbraucher-Thread



Ein Implementierungsversuch



Erzeuger

```
void produce() {
  while(true) {
     item = produce_item();



     enter_item(item);



  }
}
```

Verbraucher

```
void consume() {
  while(true) {



     item = remove_item();



     consume_item(item);
  }
}
```

Blockieren und Aufwecken von Threads

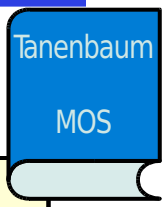
- Busy Waiting ist bei Erzeuger-/Verbraucher-Problemen sinnlos.

Daher:

- **`sleep(queue)`**
  TCBTAB[AT].Zustand=blockiert //TCB des aktiven Thread
  queue.enter(TCBTAB[AT])
  schedule

- **`wakeup(queue)`**
  TCBTAB[AT].Zustand=bereit
  switch_to(queue.take)

Ein Implementierungsversuch



Erzeuger

```
int count = 0;

void produce() {

  while(true) {

     item = produce_item();

     if (count == N)
        sleep(ProdQ);

     enter_item(item);
     count++;

     if (count == 1)
        wakeup(ConsQ);
  }
}
```

Verbraucher

```
void consume() {

  while(true) {

     if (count == 0)
        sleep(ConsQ);

     item = remove_item();
     count--;

     if (count == N - 1)
        wakeup(ProdQ);

     consume_item(item);
  }
}
```

Ein Implementierungsversuch

Tanenbaum MOS

Erzeuger

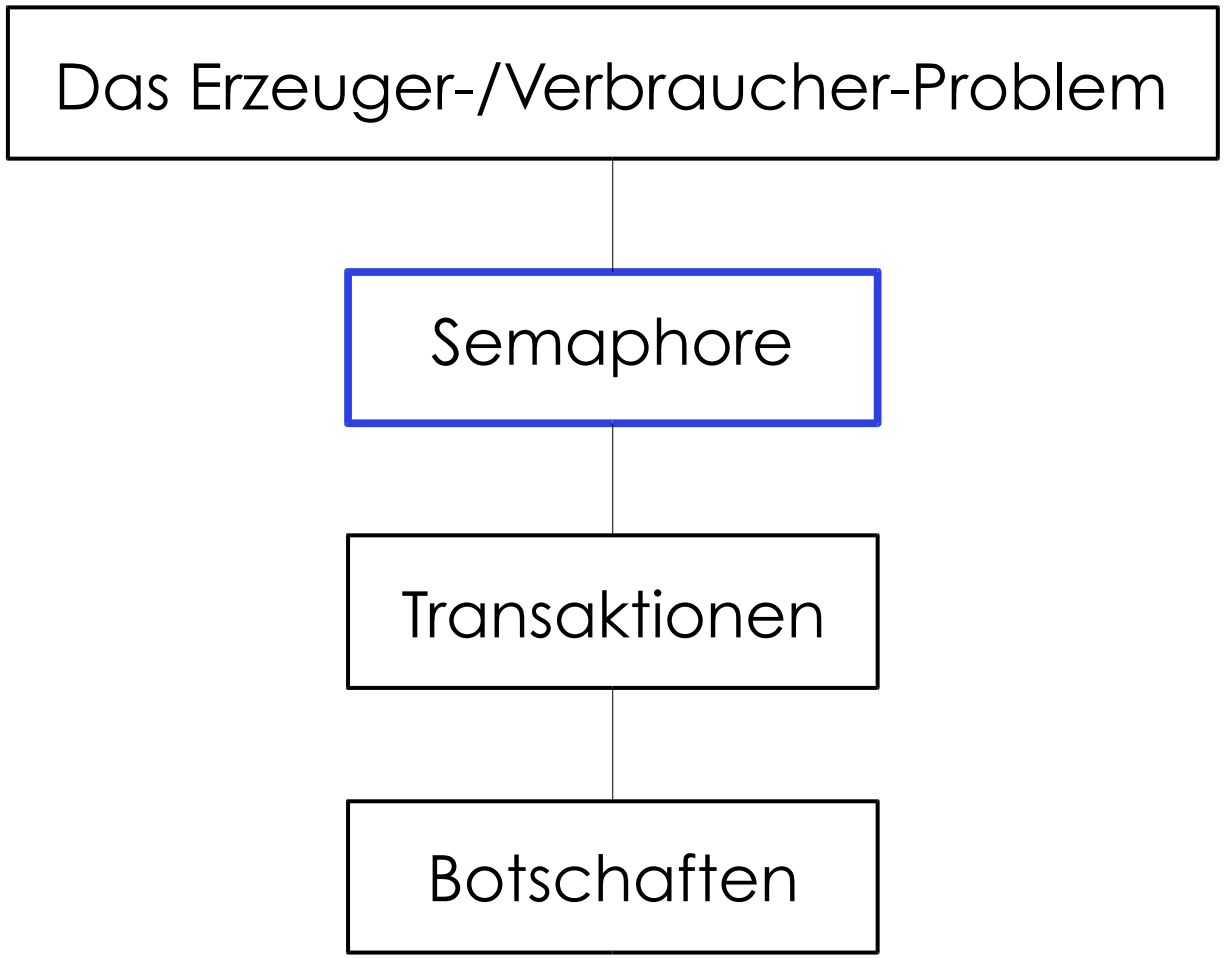
```
int count = 0;

void produce() {

  while(true) {

     item = produce_item();

     if (count == N)
        sleep(ProdQ);

     enter_item(item);
     count++;

     if (count == 1)
        wakeup(ConsQ);
  }
}
```

Verbraucher

```
void consume() {

  while(true) {

     if (count == 0)
        sleep(ConsQ);

     item = remove_item();
     count--;

     if (count == N - 1)
        wakeup(ProdQ);

     consume_item(item);
  }
}
```

FALSCH!!!

Wegweiser


Das Erzeuger-/Verbraucher-Problem
Semaphore
Transaktionen
Botschaften

Semaphore

```
class SemaphoreT {

 public:

   SemaphoreT(int howMany);
   //Konstruktor

   void Down();
   //P: passeren = betreten

   void Up();
   //V: verlaten = verlassen

}
```

Kritischer Abschnitt mit Semaphoren

```
SemaphoreT mutex(1);
```

```
mutex.Down();

do_critical_stuff();
//Kritischer Abschnitt

mutex.Up();
```

```
mutex.Down();

do_critical_stuff();
//Kritischer Abschnitt

mutex.Up();
```

Ein Thread kann kritischen Abschnitt betreten

Beschränkte Zahl

```
SemaphoreT mutex(3);
```

```
mutex.Down();

do_critical_stuff();


mutex.Up();
```

```
mutex.Down();

do_critical_stuff();


mutex.Up();
```

Drei Threads können kritischen Abschnitt betreten

Auf dem Weg Erzeuger/Verbraucher

```
SemaphoreT mutex(3);
```

```
mutex.Down();

  enter_item(item)
```

```
  remove_item(item)

mutex.Up();
```

Maximal 3 item können in den Puffer

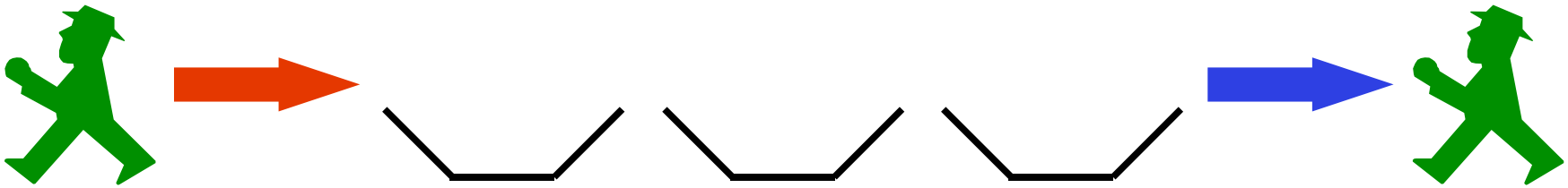
Semaphor-Implementierung

```
class SemaphoreT {

  int count;
  QueueT queue;

 public:

  SemaphoreT(int howMany);

  void Down();
  void Up();

}

SemaphoreT::SemaphoreT(
         int howMany) {

  count = howMany;
}
```

```
SemaphoreT::Down() {

  if (count <= 0)
     sleep(queue);

  count--;
}

SemaphoreT::Up() {

  count++;

  if (!queue.empty())
     wakeup(queue);
}

//Alle Methoden sind als
//kritischer Abschnitt
//zu implementieren !!!
```

Intuition (aber falsch)

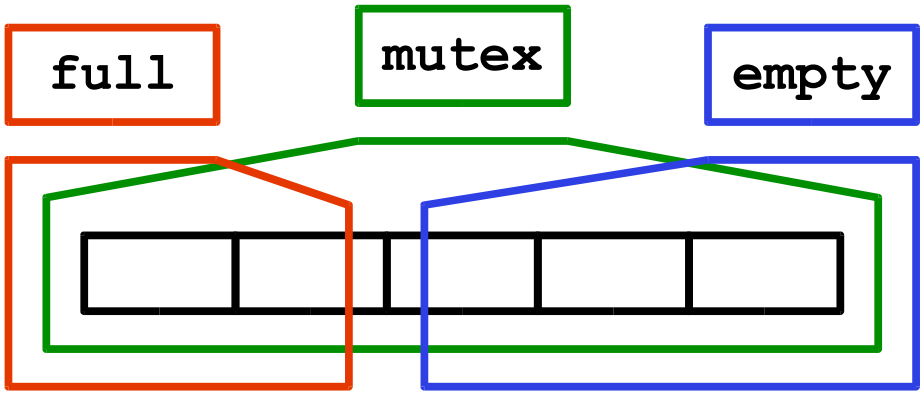
Erzeuger-Thread ***3-Elemente*** Puffer Verbraucher-Thread



```
SemaphoreT E_V(3);

E_V.Down();

enter_item();
```

```
remove_item();

E_V.Up();
```

EV-Problem mit Semaphoren



Tanenbaum MOS

```
#define N 100
//Anzahl der Puffereinträge

SemaphoreT mutex(1);
//zum Schützen des KA

SemaphoreT NOFempty(N);
//Anzahl der freien
//Puffereinträge

SemaphoreT NOFfull(0);
//Anzahl der belegten
//Puffereinträge
```

EV-Problem mit Semaphoren

Erzeuger

```
void produce() {
  while (true) {
    item = produce_item();


    enter_item(item);


  }
}
```

Verbraucher

```
void consume() {
  while (true) {


    item = remove_item();


    consume_item(item);
  }
}
```

EV-Problem mit Semaphoren

Tanenbaum MOS

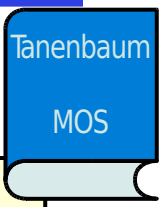
Erzeuger

```
void produce() {
  while (true) {
    item = produce_item();
    NOFempty.Down();

    enter_item(item);



  }
}
```

Verbraucher

```
void consume() {
  while (true) {



    item = remove_item();

    NOFempty.Up();
    consume_item(item);
  }
}
```

EV-Problem mit Semaphoren



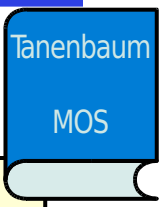
Erzeuger

```
void produce() {
  while (true) {
    item = produce_item();
    NOFempty.Down();

    enter_item(item);

    NOFfull.Up();
  }
}
```

Verbraucher

```
void consume() {
  while (true) {
    NOFfull.Down();

    item = remove_item();

    NOFempty.Up();
    consume_item(item);
  }
}
```

EV-Problem mit Semaphoren



Erzeuger

```
void produce() {

  while (true) {

    item = produce_item();

    NOFempty.Down();
    mutex.Down();

    enter_item(item);

    mutex.Up();
    NOFfull.Up();

  }
}
```

Verbraucher

```
void consume() {

  while (true) {

    NOFfull.Down();
    mutex.Down();

    item = remove_item();

    mutex.Up();
    NOFempty.Up();

    consume_item(item);

  }
}
```

Versehentlicher Tausch der Semaphor-Ops

Tanenbaum MOS

Erzeuger

```
void produce() {

  while (true) {

    item = produce_item();

    NOFempty.Down();
    mutex.Down();

    enter_item(item);

    mutex.Up();
    NOFfull.Up();

  }
}
```

Verbraucher

```
void consume() {

  while (true) {

    mutex.Down();
    NOFfull.Down();

    item = remove_item();

    mutex.Up();
    NOFempty.Up();

    consume_item(item);

  }
}
```

DEADLOCK

Monitore

- Sprachkonstrukt – ein abstrakter Datentyp (Klasse), der alle Operationen (Methoden) eines kritischen Abschnitts sammelt, welche dann unter gegenseitigem Ausschluss ablaufen
- Condition-Variable **b** mit zwei Operationen
  wait(b)
  signal(b)
  zur Koordination der Threads

Tanenbaum MOS

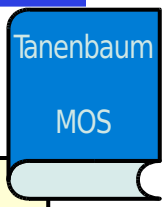
```
monitor MyMonitorT{

   condition sync_queue;

   ...

   void atomar_insert(x);

   void atomar_delete(x);

   void atomar_calc_sum();

}
```

EV-Problem mit einem Monitor

Tanenbaum MOS

```
monitor ProdCons {

  condition empty, full;
  int count;

  ProdCons() {

     count = 0;
  }

  void enter(itemT item);
  itemT remove();

}
```

EV-Problem mit einem Monitor



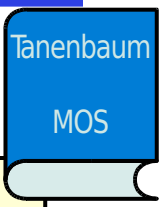
Erzeuger

```
void ProdCons::enter(
        itemT item) {




  enter_item(item);
  count++;




}
```

Verbraucher

```
itemT ProdCons::remove() {

  itemT tmp;




  tmp = remove_item();
  count--;




  return tmp;
}
```

EV-Problem mit einem Monitor



Erzeuger

```
void ProdCons::enter(
        itemT item) {

  if (count == N)
     wait(full);

  enter_item(item);
  count++;




}
```

Verbraucher

```
itemT ProdCons::remove() {

  itemT tmp;




  tmp = remove_item();
  count--;

  if (count == N - 1)
     signal(full);

  return tmp;
}
```

EV-Problem mit einem Monitor

Erzeuger

```
void ProdCons::enter(
        itemT item) {

  if (count == N)
     wait(full);

  enter_item(item);
  count++;

  if (count == 1)
     signal(empty);

}
```

Verbraucher

```
itemT ProdCons::remove() {

  itemT tmp;

  if (count == 0)
     wait(empty);

  tmp = remove_item();
  count--;

  if (count == N - 1)
     signal(full);

  return tmp;
}
```

Bei mehreren beteiligten Objekten

Problem

Jedes Konto als Monitor bzw. mit Semaphoren implementiert

- Konto2 nicht zugänglich/verfügbar
- Abbruch nach 1. Teiloperation

Abhilfe

➔ Alle an einer komplexen Operation beteiligten Objekte vorher sperren

```
void ueberweisen(
      int Betrag,
      KontoT Konto1,
      KontoT Konto2) {

  Konto1.abbuchen(Betrag);

  Konto2.gutschreib(Betrag);

}
```

Bei mehreren beteiligten Objekten

Vorgehensweise

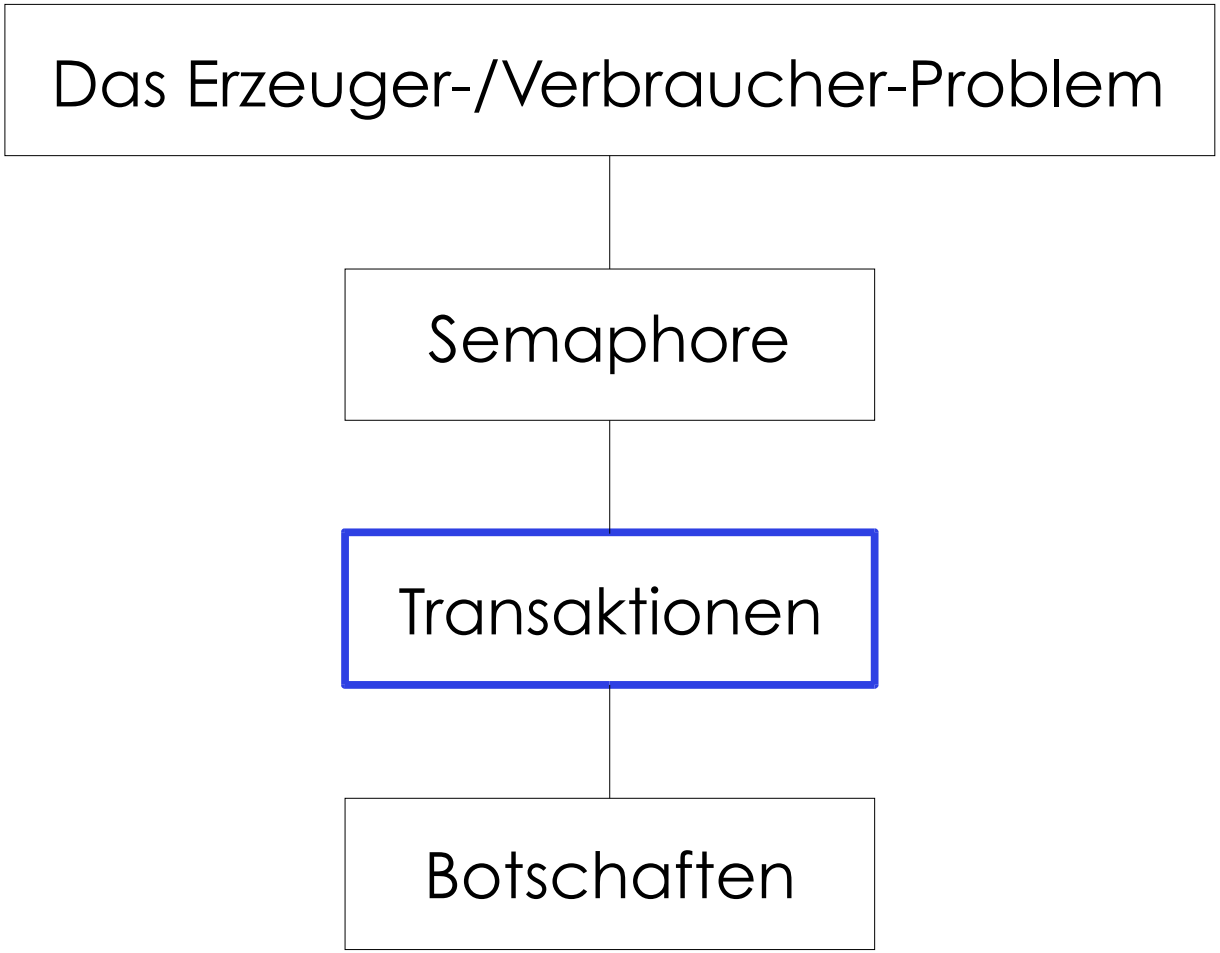
- beteiligte Objekte vor dem Zugriff sperren
- erst nach Abschluss der Gesamtoperation entsperren

```
void ueberweisen(
     int Betrag,
     KontoT Konto1,
     KontoT Konto2) {

 lock(Konto1);
 Konto1.abbuchen(Betrag);


 lock(Konto2);
 if NOT (Konto2.
  gutschreib(Betrag)) {

   Konto1.gutschreib(Betrag);
 }

 unlock(Konto1);
 unlock(Konto2);
}
```

Objekte mit Sperren

```
monitor Konto {

  int locked;
  condition queue;

  void lock() {

     if (locked)
        wait(queue);

     locked = true;
  }

  void unlock() {

     if (locked)
        signal(queue)
     else locked = false;
  }
}
```

Wegweiser


Das Erzeuger-/Verbraucher-Problem
Semaphore
Transaktionen
Botschaften

Verallgemeinerung: Transaktionen

Motivation

- Synchronisation komplexer Operationen mit mehreren beteiligten Objekten
- Rückgängigmachen von Teiloperationen gescheiterter komplexer Operationen
- Dauerhaftigkeit der Ergebnisse komplexer Operationen (auch bei Fehlern und Systemabstürzen)

Voraussetzungen

- Transaktionsmanager
- alle beteiligten Objekte verfügen über bestimmte Operationen

Konto-Beispiel mit Transaktionen

```
void ueberweisung(int Betrag,
                  KontoT Konto1, KontoT Konto2) {

  int Transaction_ID = begin_Transaction();

  use(Transaction_ID, Konto1);
     Konto1.abbuchen(Betrag);

  use(Transaction_ID, Konto2);
     if (!Konto2.gutschreiben(Betrag)) {

        abort_Transaction(Transaction_ID);
        //alle Operationen, die zur Transaktion gehören,
        //werden rückgängig gemacht
     }

  commit_Transaction(Transaction_ID);
  //alle Locks werden freigegeben
}
```

Transaktionen: „ACID“

Eigenschaften von Transaktionen

- **A**tomar:
  Komplexe Operationen werden ganz oder gar nicht durchgeführt.
- Konsistent (**C**onsistent):
  Es werden konsistente Zustände in konsistente Zustände überführt, Zwischenzustände dürfen inkonsistent sein.
- **I**soliert:
  Transaktionen dürfen parallel durchgeführt werden genau dann, wenn sichergestellt ist, dass das Ergebnis einer möglichen sequentiellen Ausführung der Transaktionen entspricht.
- **D**auerhaft:
  Nach dem Commit einer Transaktion ist deren Wirkung verfügbar, auch über Systemabstürze hinweg.

Transaktionsmanager

Sehr leistungsfähige Werkzeuge ...

- Mehr dazu in den Vorlesungen:

  (Verteilte Betriebssysteme)
  Datenbanken

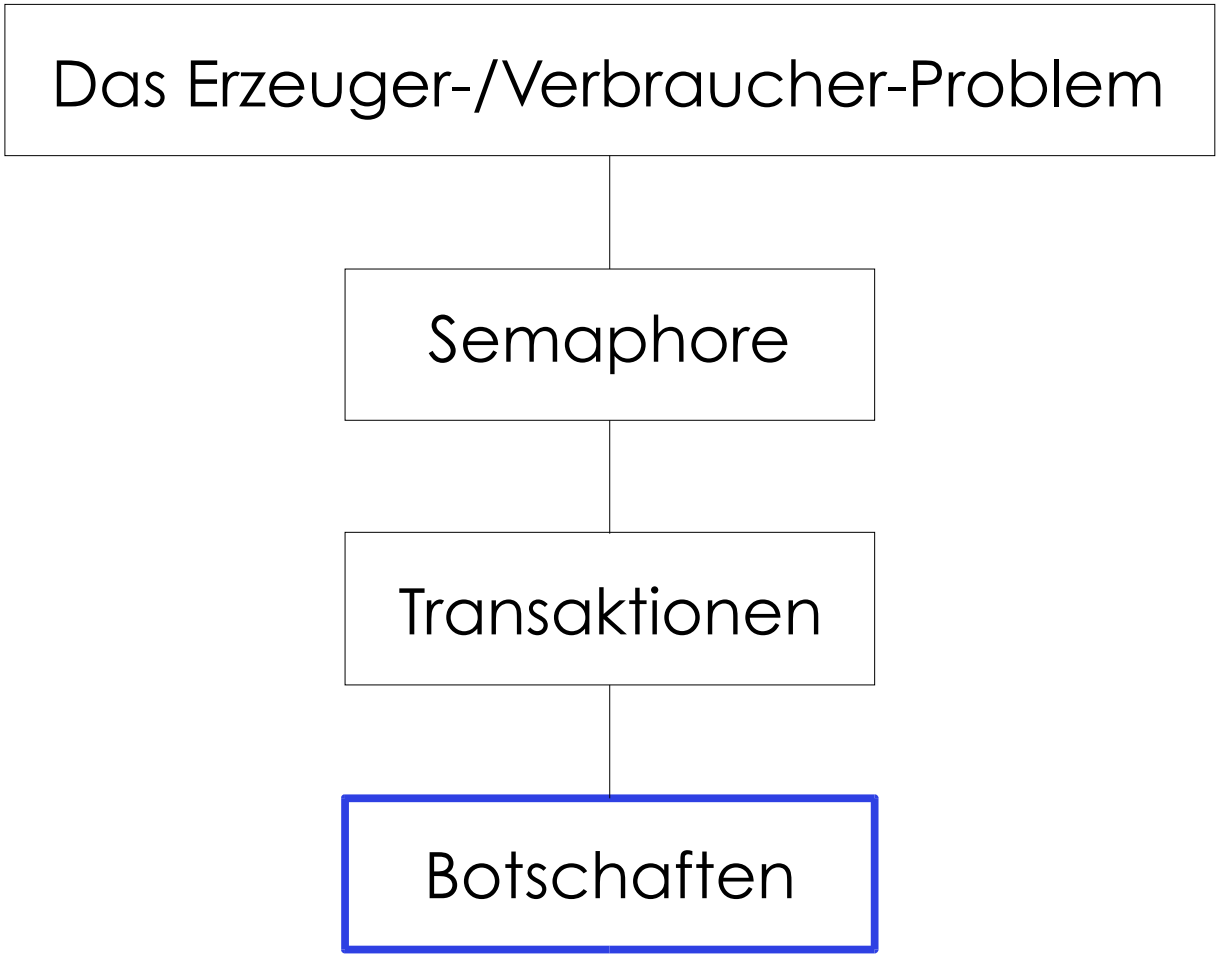
Nachteile von Semaphoren etc.

- Basieren auf der Existenz eines gemeinsamen Speichers

Jedoch häufig Kommunikation zwischen Threads ohne gemeinsamen Speicher, z. B.

- Threads in unterschiedlichen Adressräumen
- Threads auf unterschiedlichen Rechnern
- massiv parallele Rechner – jeder Rechenknoten mit eigenem Speicher, z. B. HPC Cray Jaguar

Wegweiser


Das Erzeuger-/Verbraucher-Problem
Semaphore
Transaktionen
Botschaften

Botschaften

Senden

- baue Botschaft auf
  (Daten, Daten, ..., Botschaft)
- sende Botschaft zum Ziel

```
void send(dest, message);

void receive(message);
```

Empfangen

- empfange Botschaft
- extrahiere
  (Daten, Daten, ..., Botschaft)

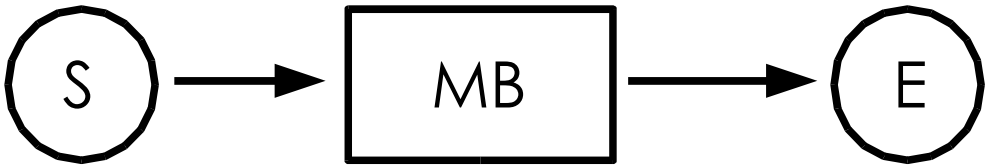
„marshalling“

Botschaften

Varianten des Grundkonzepts

- Adressat:

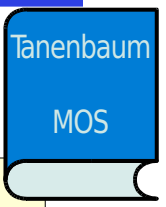
  Prozess, Thread oder Briefkasten (mail box)



- Pufferung/Synchronität:

  mit/ohne

EV-Problem mit Botschaften



Erzeuger

```
void producer() {

  while (true) {

     item = produce_item();




     send(item);
  }
}
```

Verbraucher

```
void consumer() {



 while (true) {

    receive(item);




    consume_item(item);

 }
}
```

EV-Problem mit Botschaften

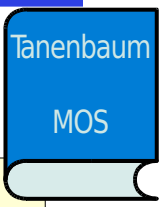
Erzeuger

```
void producer() {

  while (true) {

     item = produce_item();

     receive(empty);



     send(item);
  }
}
```

Verbraucher

```
void consumer() {


send(empty);

 while (true) {

    receive(item);



    send(empty);

    consume_item(item);

 }
}
```

EV-Problem mit Botschaften



Erzeuger

```
void producer() {

  while (true) {

     item = produce_item();

     receive(empty);



     send(item);
  }
}
```

Verbraucher

```
void consumer() {

 for(int i = 0; i < N; i++)
   send(empty);

 while (true) {

   receive(item);



   send(empty);

   consume_item(item);

 }
}
```

Synchrone Botschaften

Senden

- **`int send(message, timeout, TID);`**
- synchron, d. h. terminiert, wenn das korrespondierende **`receive`**/**`wait`** durchgeführt wurde
- Ergebnis : **`false`**, falls das Timeout greift

Warten

- **`int wait(message, timeout, &TID);`**
- akzeptiert eine Botschaft von irgendeinem Thread
- liefert die ID des Sender-Threads

Empfangen

- **`int receive(message, timeout, TID);`**
- akzeptiert eine Botschaft von einem bestimmten Sender-Thread

Binärer Semaphor mit Hilfe synchroner Botschaften

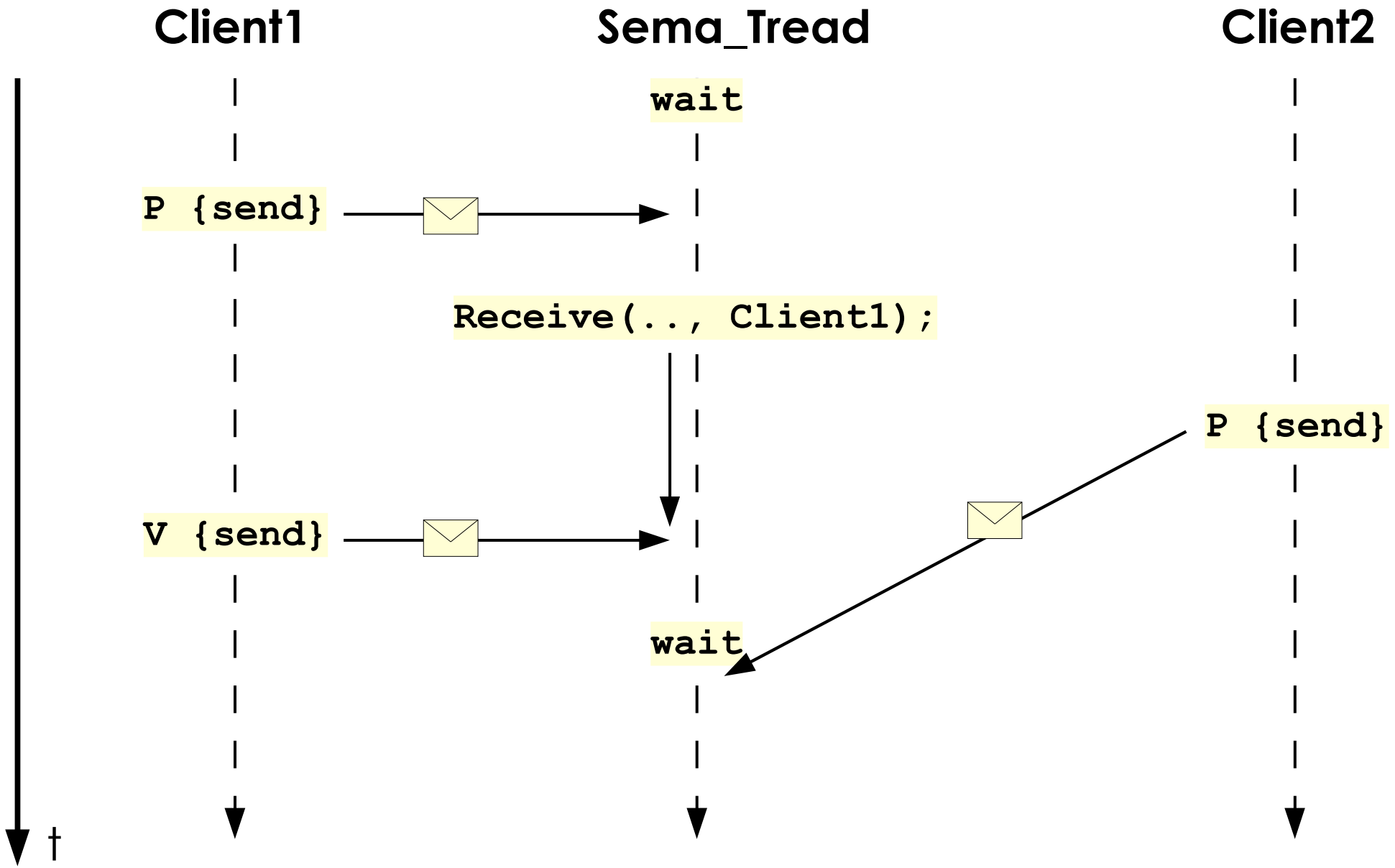
Semaphore-Thread

```
thread_T sema_thread;

while (1) {

 wait(message, threadId);
 receive(message,threadId);

}
```

Clients

```
void P() {

 send(empty_message,
      sema_thread);
}

void V() {

 send(empty_message,
      sema_thread);
}
```

Binärer Semaphor mit Hilfe synchroner Botschaften



Botschaften

Aufbau

- feste Länge
- beliebig, aber am Stück
- zerfleddert

Umgang mit Fehlersituationen (z. B. Netz)

- Quittung für jede Nachricht
- Folgenummern
- gar nichts

Botschaften

Systemarchitektur:

- Betriebssystemkerne (Linux, Mikrokerne, ...)
- Netzwerk-“Stack“
- Bibliotheken (z.B., MPI für Hochleistungsrechnen)

Wegweiser

Zusammenspiel (Kommunikation) mehrerer Threads

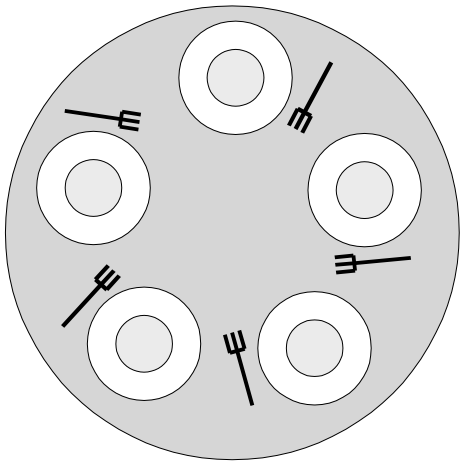
Einige typische Probleme

- Erzeuger / Verbraucher
- 5 Philosophen
- Leser/Schreiber

Das Zuschneiden von Threadsystemen
→ Verschiedene Schedulingverfahren

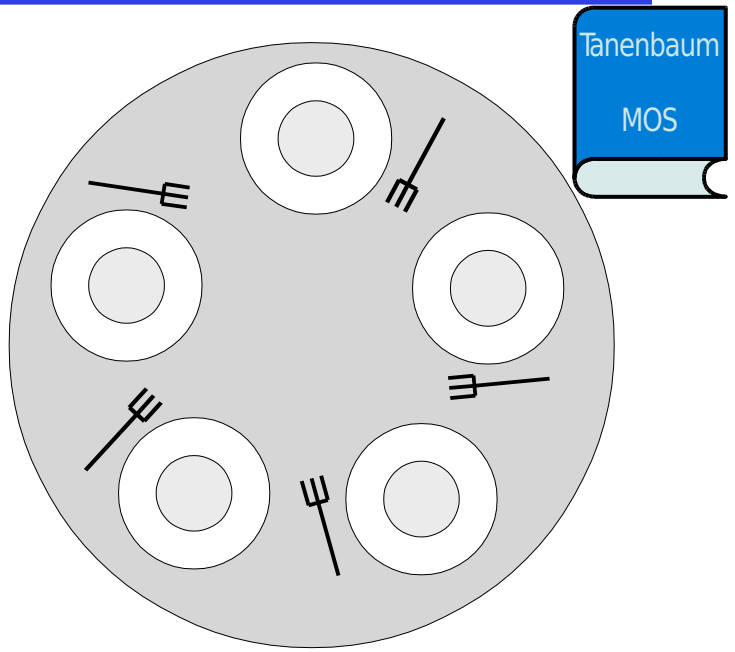
Dining Philosophers

- 5-Philosophen-Problem
- 5 Philosophen sitzen um einen Tisch, denken und essen Spaghetti
- zum Essen braucht jeder zwei Gabeln, es gibt aber insgesamt nur 5
- Problem: kein Philosoph soll verhungern



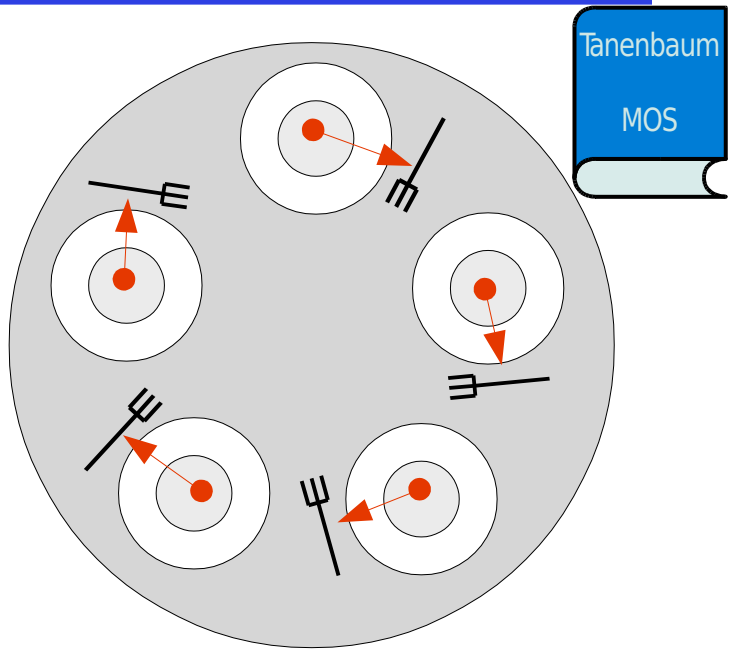
Die offensichtliche ( ) Lösung

```
void Philosoph(int i) {

  Denken();

  NimmGabel(i);

  NimmGabel((i+1)%5);

  Essen();

  LegeZurückGabel(i);
  LegeZurückGabel((i+1)%5);
}
```



Die offensichtliche (aber falsche) Lösung

```
void Philosoph(int i) {

  Denken();

  NimmGabel(i);

  NimmGabel((i+1)%5);

  Essen();

  LegeZurückGabel(i);
  LegeZurückGabel((i+1)%5);
}
```



Falsch

- kann zu Verklemmungen/Deadlocks führen
  (alle Philosophen nehmen gleichzeitig die linken Gabeln)
- zwei verbündete Phil. können einen dritten aushungern

Leser/Schreiber-Problem

Lesen und Schreiben von Daten (z. B. Datenbankeinträgen)

- mehrere Prozesse dürfen gleichzeitig lesen
- nur einer schreiben

Leser/Schreiber-Problem

Lesen und Schreiben von Daten (z. B. Datenbankeinträgen)

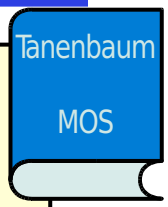
- mehrere Prozesse dürfen gleichzeitig lesen
- nur einer schreiben

```
SemaphoreT Mutex(1);
SemaphoreT DB(1);
int        RCount = 0;
```

Leser/Schreiber mit Semaphoren

```
void Reader() {

  //read

}
```

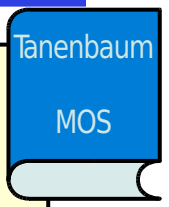
```
void Writer() {

  //write

}
```



Leser/Schreiber mit Semaphoren

```
void Reader() {

  DB.Down();

  //read

  DB.Up();
}
```

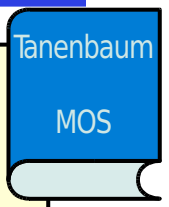
```
void Writer() {

  DB.Down();

  //write

  DB.Up();

}
```



Leser/Schreiber mit Semaphoren

```
void Reader() {


  Rcount++;
  if (Rcount==1) DB.Down();


  //read


  Rcount--;
  if (Rcount==0) DB.Up();

}
```

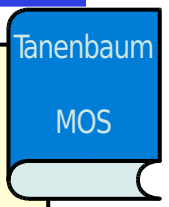
```
void Writer() {



  DB.Down();

  //write

  DB.Up();


}
```



Leser/Schreiber mit Semaphoren

```
void Reader() {

  Mutex.Down();
  Rcount++;
  if (Rcount==1) DB.Down();
  Mutex.Up();

  //read

  Mutex.Down();
  Rcount--;
  if (Rcount==0) DB.Up();
  Mutex.Up();
}
```

```
void Writer() {



  DB.Down();

  //write

  DB.Up();


}
```



Wegweiser

Zusammenspiel (Kommunikation) mehrerer Threads

Einige typische Probleme

Das Zuschneiden von Threadsystemen: Scheduling

- Gesichtspunkte für ...
- Round Robin
- Prioritäten
- Echtzeitsysteme

Scheduling

Aufgabe:

Entscheidung über Prozessorzuteilung

- an welchen Stellen (Zeitpunkte, Ereignisse, ...)
- nach welchem Verfahren

Scheduling

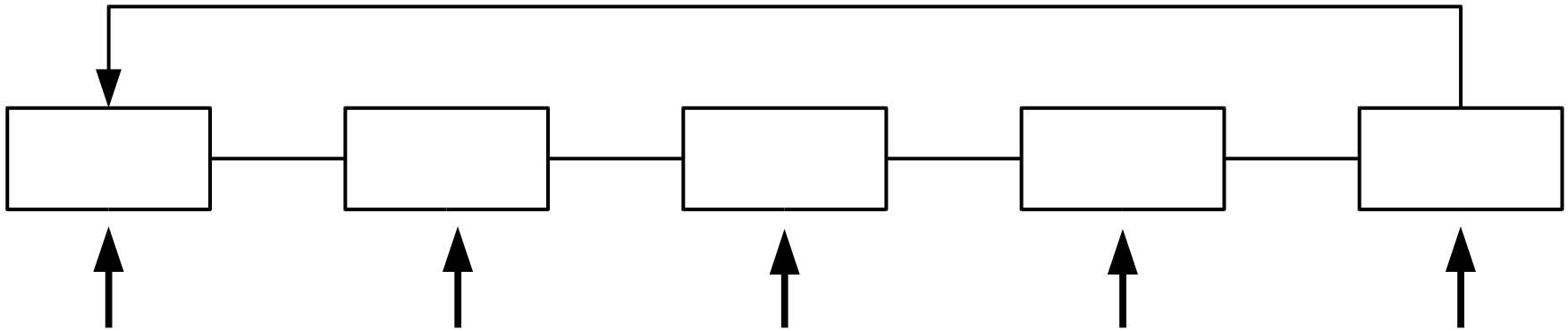
Kriterien

- Fairness: fairer Anteil an CPU für den Thread
- Effizienz: möglichst vollständige Auslastung der CPU
- Antwortzeiten: interaktiver Anwendungen
- Wartezeiten: von Hintergrundjobs
- Durchsatz: maximale Zahl von Aufgaben pro ZE
- Zusagen: ein Prozess muss spätestens *dann* fertig sein

Probleme

- Kriterien sind widersprüchlich
- Anwender-Threads sind in ihrem Verhalten oft unvorhersehbar

Round Robin mit Zeitscheiben



- jeder Thread erhält reihum eine ***Zeitscheibe*** (time slice), die er zu Ende führen kann

Wahl der Zeitscheibe

- zu kurz: CPU schaltet dauernd um
- zu lang: Antwortzeiten zu lang

Prioritätssteuerung

- Threads erhalten ein Attribut: Priorität
- höhere Priorität verschafft Vorrang vor niederer Priorität

Fragestellungen

- Zuweisung von Prioritäten
- Thread mit höherer Priorität als der rechnende wird bereit
  → Umschaltung: sofort, ... ? ("preemptive scheduling“)

➔ häufig: Kombination Round Robin und Prioritäten

Zuweisung von Prioritäten

- statisch ...
- dynamisch
  - Benutzer (Unix: “nice”)
  - Heuristiken im System
  - gezielte Vergabeverfahren

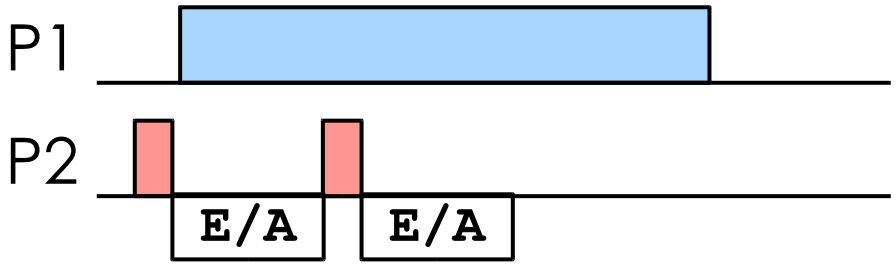
Beispiel für Heuristik

- hoher Anteil an Wartezeiten (z.B. E/A)
  → hohe Priorität

➔ bessere Auslastung von E/A-Geräten

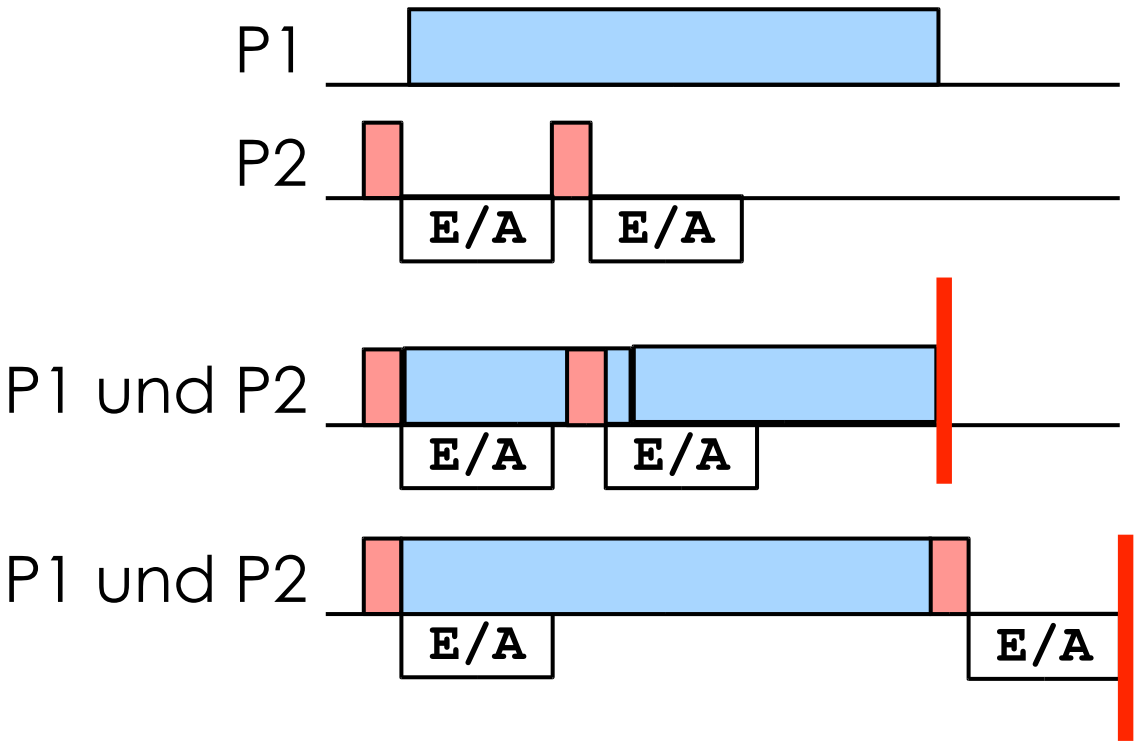
Beispiel

- Cobegin P2; P1 Coend
- P1 erst kurze Pause, dann lang CPU
- P2 kurz CPU, langsame EA



Beispiel

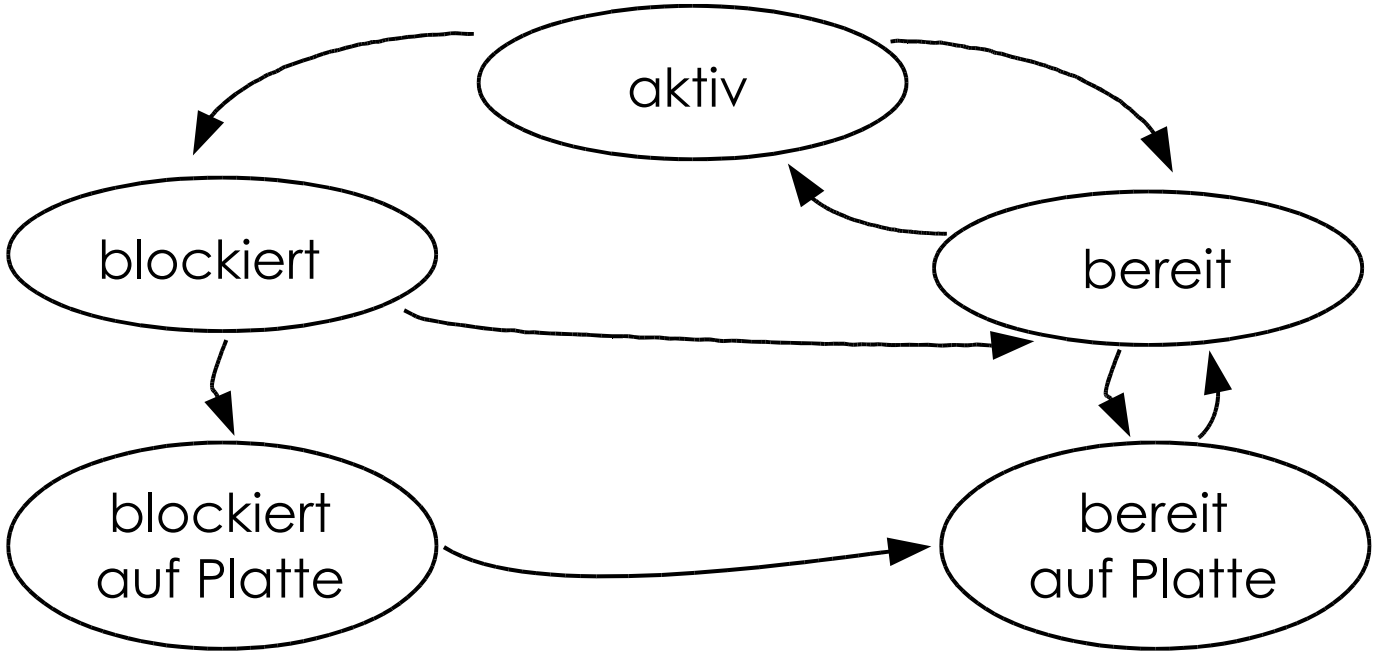
- Cobegin P2; P1 Coend
- P1 erst kurze Pause, dann lang CPU
- P2 kurz CPU, langsame EA



Zwei-Ebenen-Scheduling

Ausgangspunkt

- Prozesse benötigen nicht nur CPU, sondern auch Speicher, manchmal müssen Prozesse ganz aus Hauptspeicher verdrängt werden.
- Nicht nur die Zuteilung von CPU zu Thread, sondern auch die Verdrängung auf Platte wird Gegenstand des Schedulings.



„Prioritätsumkehr“ (Priority Inversion)

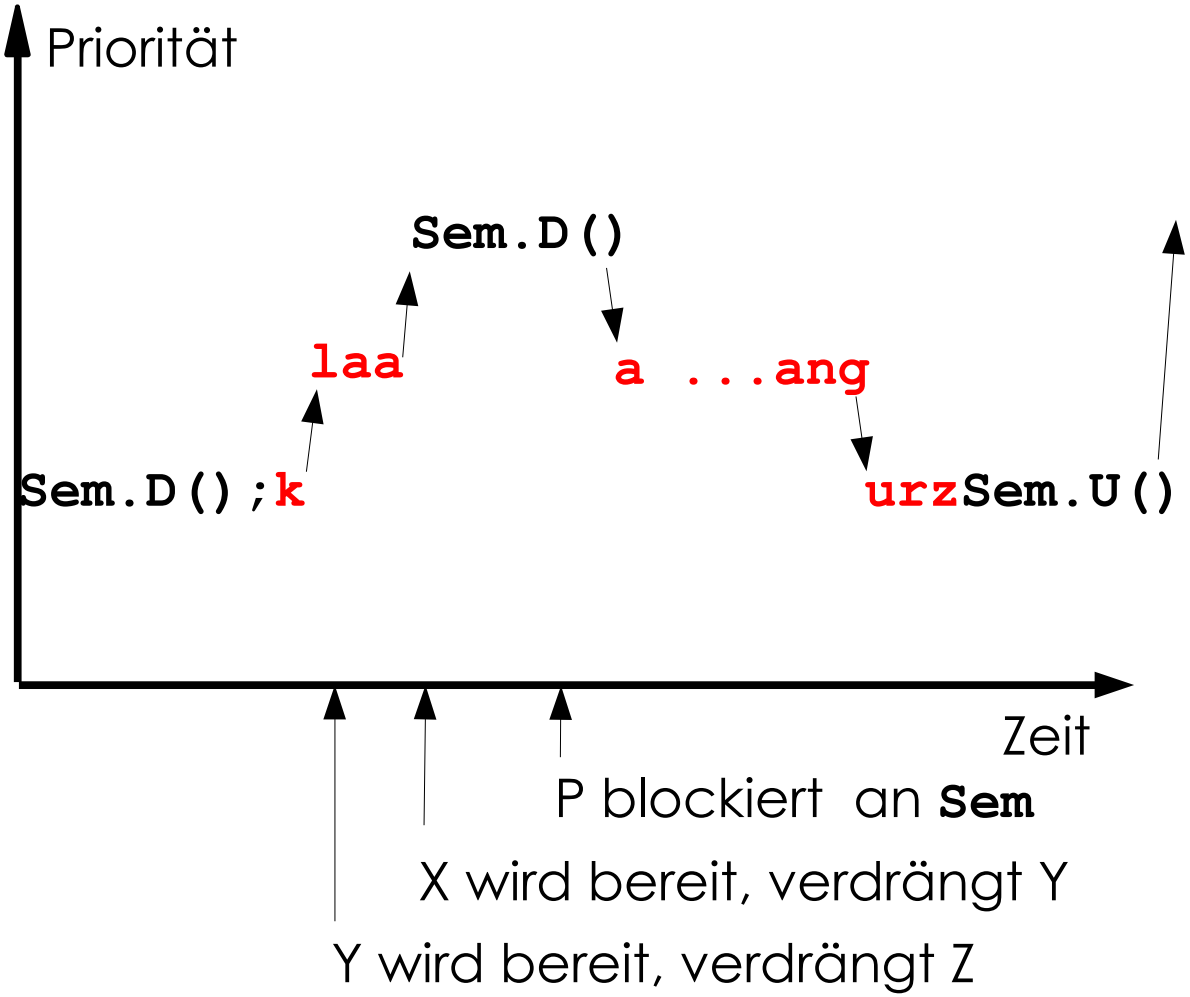
Beispielszenario:

- drei Threads X, Y, Z
- X höchste, Z niedrigste Priorität
- werden bereit in der Reihenfolge: Z, Y, X

```
cobegin
  //X:
  {Sem.D();kurz;Sem.U()};
  //Y:
  {laaaaaaaaaaaaaaaang};
  //Z:
  {Sem.D();kurz;Sem.U()};
coend
```

„Prioritätsumkehr“ (Priority Inversion)

```
cobegin
  //X:
  {Sem.D();kurz;Sem.U()};
  //Y:
  {laaaa ... aaaaaaaang};
  //Z:
  {Sem.D();kurz;Sem.U()};
coend
```



X hat höchste Priorität, kann aber nicht laufen,
weil es an von Z gehaltenem Semaphor blockiert und
Y läuft und damit Z den Semaphor nicht freigeben kann.

Zusammenfassung Threads/Prozesse

- (Prozesse) Threads sind ein mächtiges Konzept zur Konstruktion von Systemen, die mit asynchonen Ereignissen umgehen können müssen.
- Kooperierende parallele Threads müssen sorgfältig synchronisiert werden.
  Fehler dabei sind extrem schwer zu finden, da zeitabhängig.
- Bei vielen Entscheidungen sind Kriterien zu berücksichtigen, die einander widersprechen.
  Ein solider Entwurf muss die Diskussion der "Trade Offs" einschließen.

Bald in dieser Vorlesung:

- „Echtzeitsysteme“ und Scheduling
- Interaktion mit Speicher und Adressierung
- Kommunikation von Prozessen in einem verteilten System

Echtzeitsysteme

Ausgangspunkt

- Threads machen Zusagen über die Zeit bis zur Erledigung bestimmter Aufgaben.
- Dadurch entstehen Zeitschranken (deadlines), die von Prozessen eingehalten werden müssen.

Vereinfachtes Modell

- Threads periodisch
  - `P` Periode
  - `wcet` höchstens zu erwartende Rechenzeit in Periode (worst case execution time)
- Deadline = Ende der Periode