Kritischer Abschnitt

Beispiel: Geld abbuchen

- Problem:
  Wettläufe zwischen den Threads

  „race conditions“

➔ Den Programmteil, in dem auf gemeinsamem Speicher gearbeitet wird, nennt man

„Kritischer Abschnitt“

```
int Kontostand;
//Variable im gemeinsamen
//Speicher

bool abbuchen(int Betrag) {

  if (Betrag <= Kontostand){

    Kontostand -= Betrag;
    return true;

  } else return false;
}
```

Beispiel für einen möglichen Ablauf

```
Kontostand = 20;
COBEGIN
  T1: abbuchen(1);
  T2: abbuchen(20);
COEND
```

```
T1:
     ...
```

```
T2:
     ...
```

Beispiel für einen möglichen Ablauf

```
Kontostand = 20;
COBEGIN
  T1: abbuchen(1);
  T2: abbuchen(20);
COEND
```

```
if (Betrag <= Kontostand){
 Kontostand -= Betrag;
 return true;
} else return false;
```

```
if (Betrag <= Kontostand){
 Kontostand -= Betrag;
 return true;
} else return false;
```

- Resultat: T1 darf abbuchen, T2 nicht, **`Kontostand == 19`**

Mögliche Ergebnisse

```
Kontostand = 20;
COBEGIN
  T1: abbuchen(1);
  T2: abbuchen(20);
COEND
```

	Kontostand	Erfolg T1	Erfolg T2
1	19	True	False
2			
3			
4			
5			

Variante 2

```
Kontostand = 20;
COBEGIN
  T1: abbuchen(1);
  T2: abbuchen(20);
COEND
```

```
if (Betrag <= Kontostand){
```

```
if (Betrag <= Kontostand){
 Kontostand -= Betrag;
 return true;
} else return false;
```

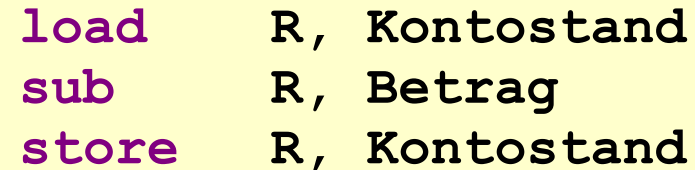
```
 Kontostand -= Betrag;
 return true;
} else return false;
```

- Resultat: T1 und T2 buchen ab, **`Kontostand == -1`**

Subtraktion unter der Lupe

Hochsprache | **Maschinensprache**

```
int Kontostand;
//Variable im gemeinsamen
//Speicher

bool abbuchen(int Betrag) {

 if (Betrag <= Kontostand){

  Kontostand -= Betrag;

  return true;

 } else return false;
}
```



Variante 3 – Die kundenfreundliche Bank

T1 : abbuchen(1);

T2 : abbuchen(20);

```
T1:
load    R, Kontostand
```

```
T2:
load    R, Kontostand
sub     R, Betrag
store   R, Kontostand
```

```
T1:
sub     R, Betrag
store   R, Kontostand
```

- Resultat: T1 und T2 buchen ab, **`Kontostand == 19`**

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)

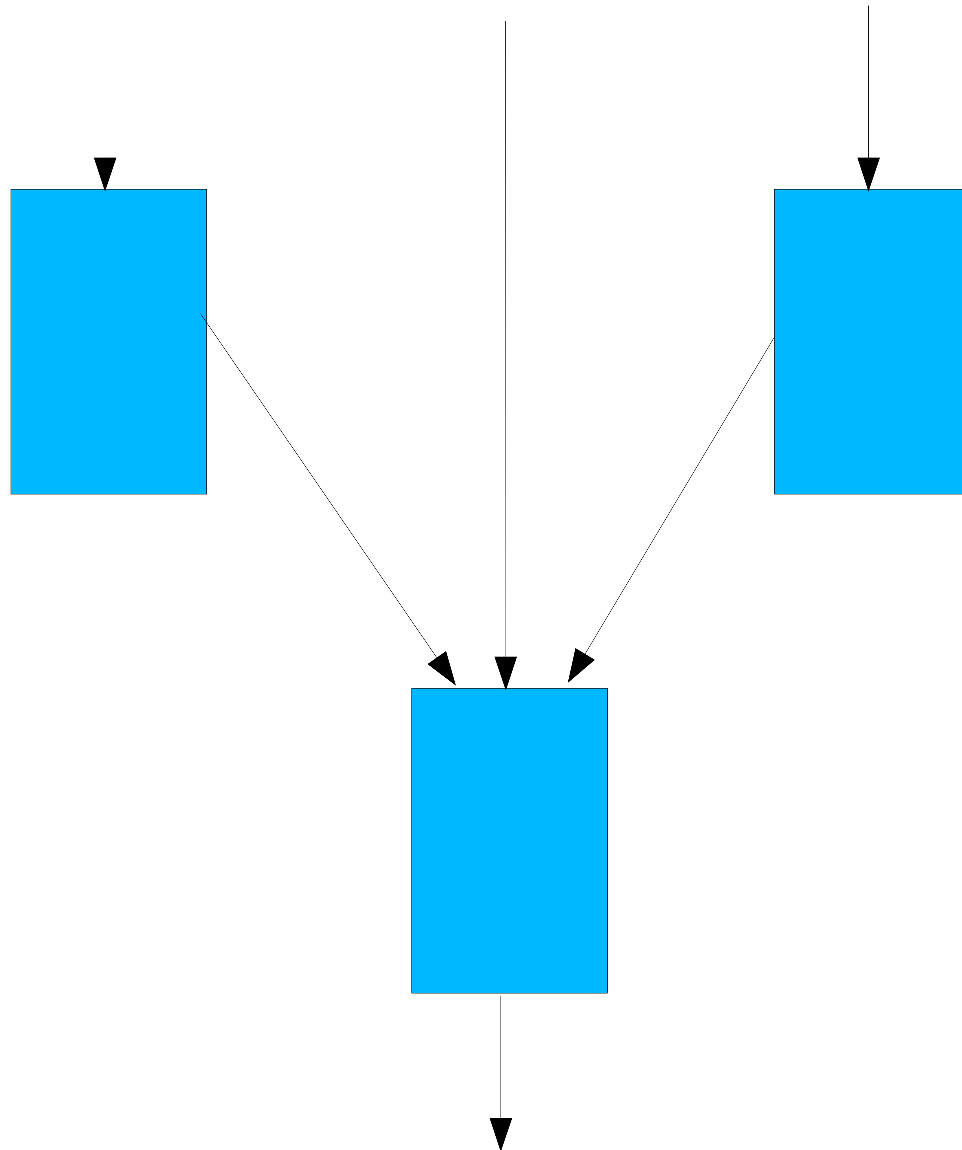
```
int K_S; lock_T Konto_Lock;

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)

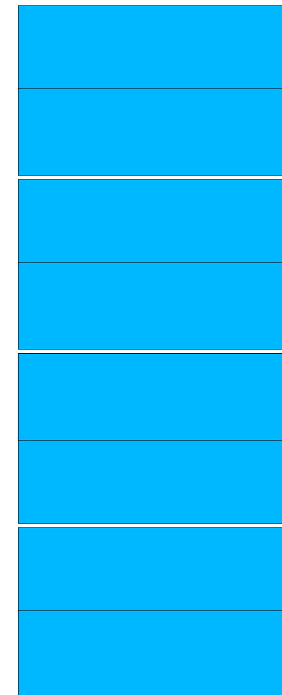
```
int K_S; lock_T Konto_Lock;

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:



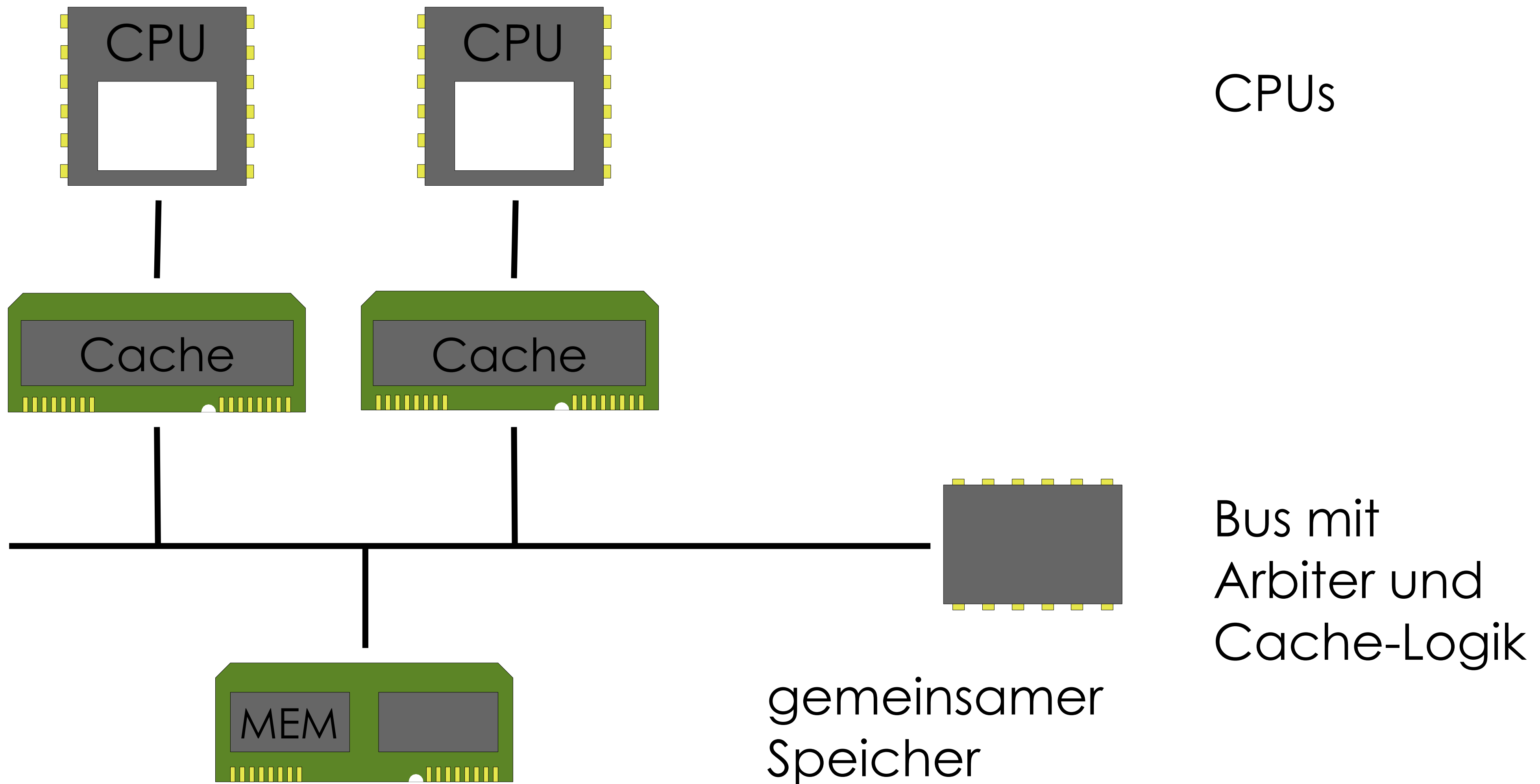
Lösungsversuch 1 – mit Unterbrechungssperre

```
T1: ...
  pushf
  cli

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

  popf
```

Globales Lock()

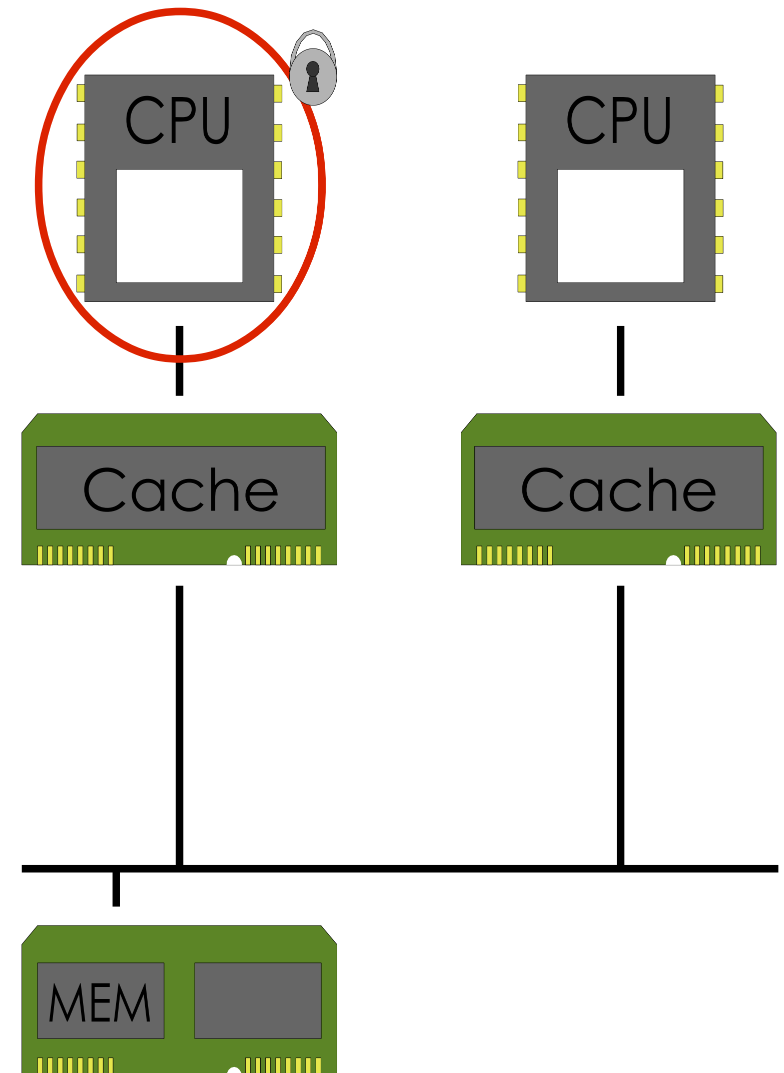
Die Unterbrechungssperre auf der CPU verhindert, dass **T1** im kritischen Abschnitt preemptiert wird.

Einfaches Modell einer Dual-CPU Maschine



Lösungsversuch 1 – mit Unterbrechungssperre

```
T1: ...
  pushf
  cli

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

  popf
```



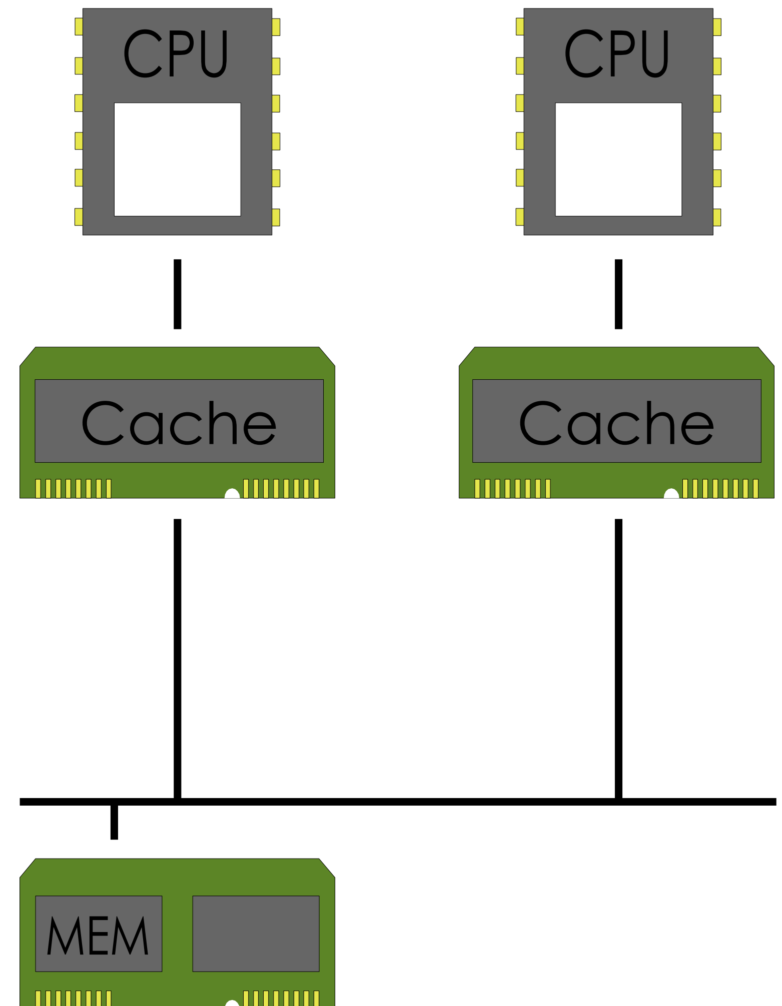
Die Unterbrechungssperre auf der CPU verhindert, dass **T1** im kritischen Abschnitt preemptiert wird.

→ 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**

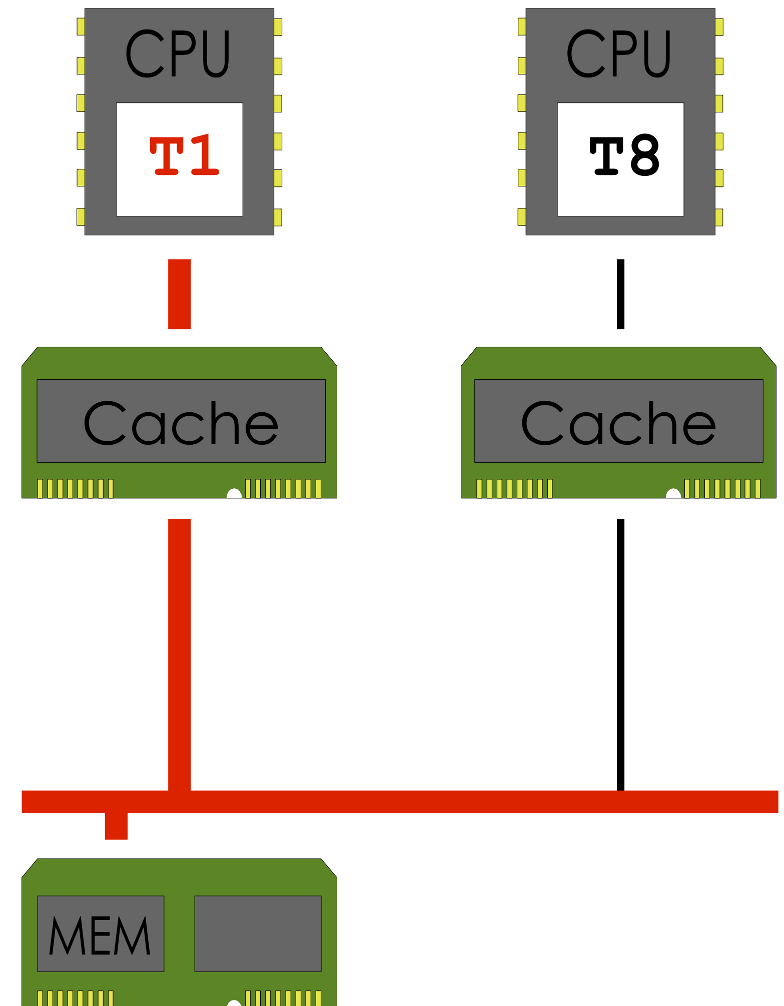
Lösungsversuch 2 – KA mit einer Instruktion

```
Ti: ...

  ld      R, Betrag

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

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

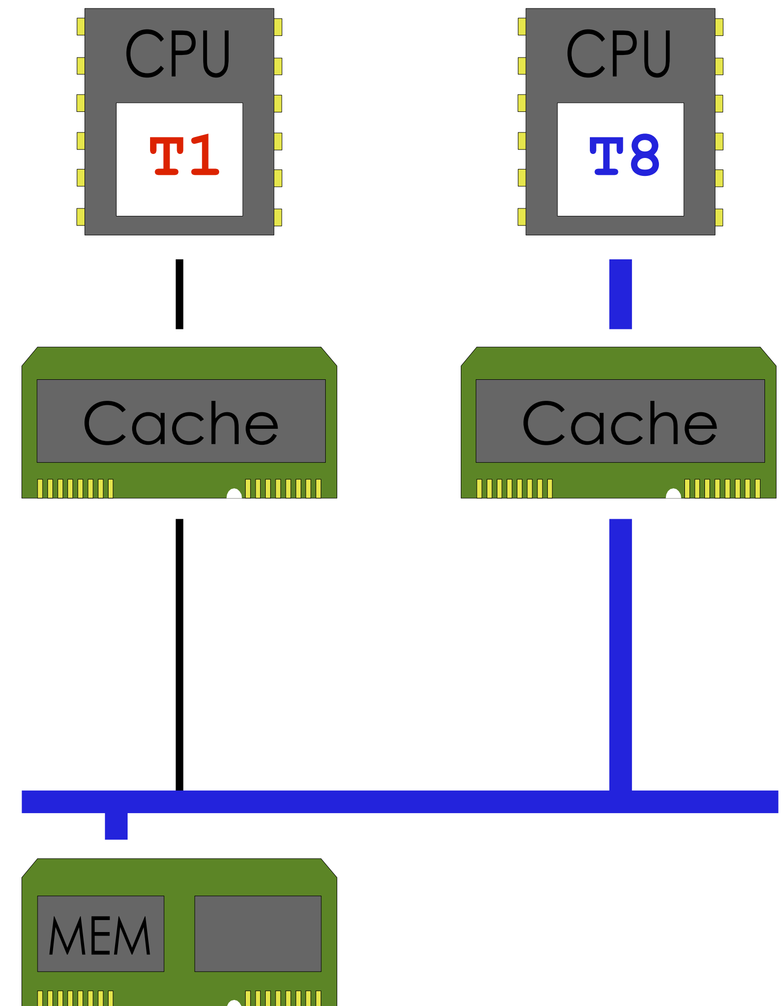
- Einzelne Buszyklen sind die atomare Einheit !

➔ ...



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 !

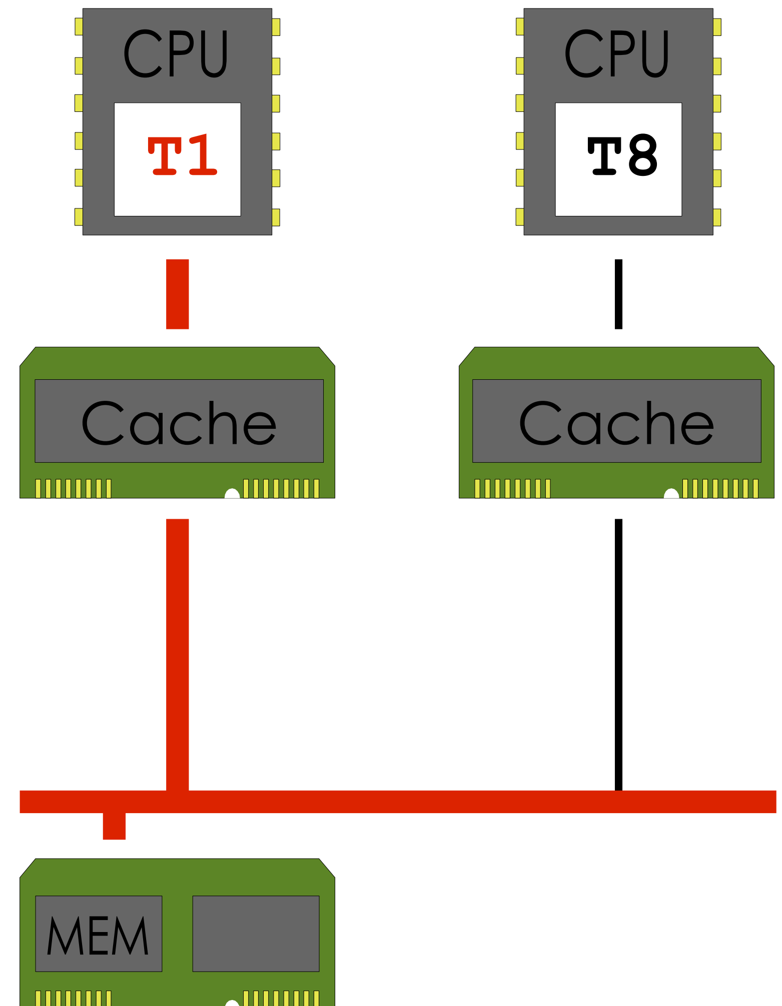
→ funktioniert so nicht !

gemeinsamer Speicher

- **Kontostand**

Lösungsversuch 2 – KA mit einer Instruktion

```
Ti: ...

  ld      R, Betrag

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



gemeinsamer Speicher
- **Kontostand**

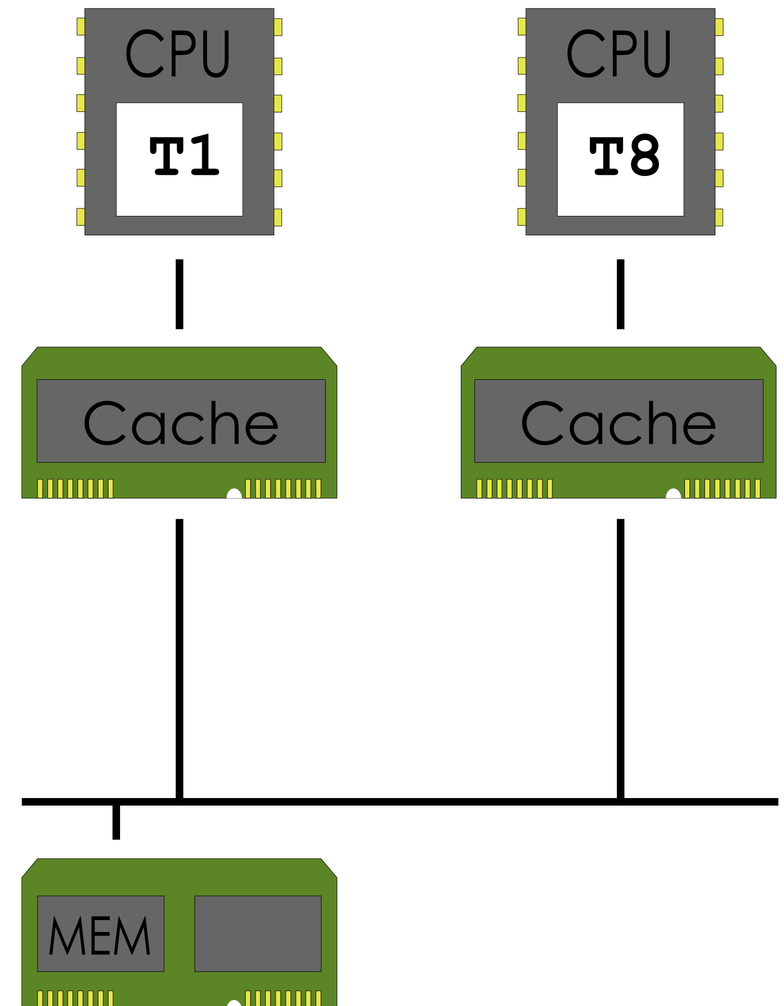
- Die Instruktion
  sub Kontostand, R
  hat zwei Buszyklen

- Einzelne Buszyklen sind die atomare Einheit !

→ funktioniert so nicht !

Lösungsversuch 3 – atomare Instruktion

```
Ti: ...

 ld            R, Betrag

 atomic_sub    Kontostand, R
```



- Die Instruktion **`atomic_sub Kontostand, R`** ist nicht unterbrechbar.
- Die Speicherzelle bleibt während Instruktion für den Zugriff gesperrt

➔ funktioniert.

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
  ret

unlock():
  popf        //restore
  ret
```

Implementierung mittels Sperrvariable

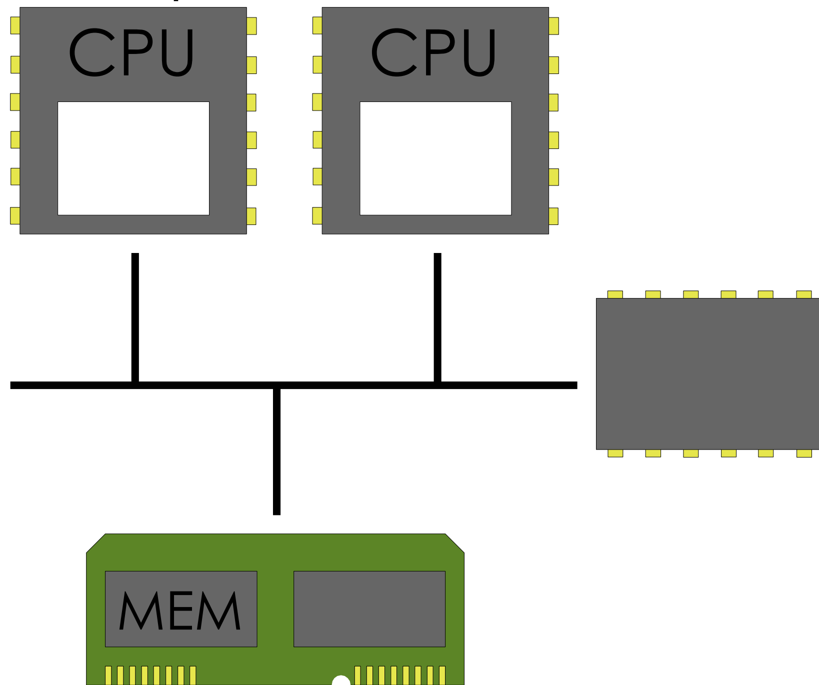
- **funktioniert nicht!**
- warum?

➔ 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 pro „lock“ ist zu teuer
- busy waiting usurpiert die CPU

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

```
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;
}
```

Implementierung für eine CPU im User-Mode

- Unterbrechungssperren sind nicht sicher
- Ein Kern-Aufruf pro „entersection“ 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
```

Wegweiser

Zusammenspiel/Kommunikation mehrerer Threads

Nutzung gemeinsamen Speichers

- race conditions
- → Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluß und dessen Durchsetzung

- ohne HW-Unterstützung möglich ????

Alternative: Atomare Operation CAS

myself == 1	lock	myself == 2
	0	

	2	`owner = cas(lock, 0, 2))`
		`result == 0`
		`im KA`

`owner = cas(lock, 0, 1))`
`result == 2`
 	
`switch_to(2)`
`nicht im KA (loop cas)`		

		`KA fertig` `*lock=0`
	0	

`owner = cas(lock, 0, 1))`
`result == 0`
`im KA`
	1	

KA: kritischer Abschnitt

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 consistency“
- mehr dazu in fortgeschrittenen Vorlesungen

Wegweiser

Das Erzeuger-/Verbraucher-Problem

Semaphore

Transaktionen

Botschaften

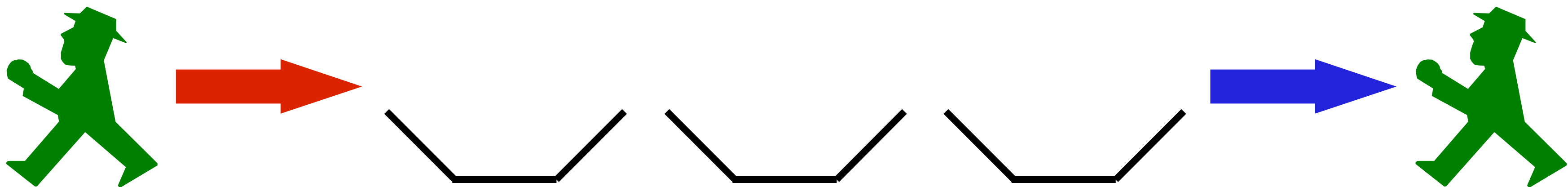
Erzeuger-/Verbraucher-Probleme

Beispiele

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

Reduziert auf das Wesentliche

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



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

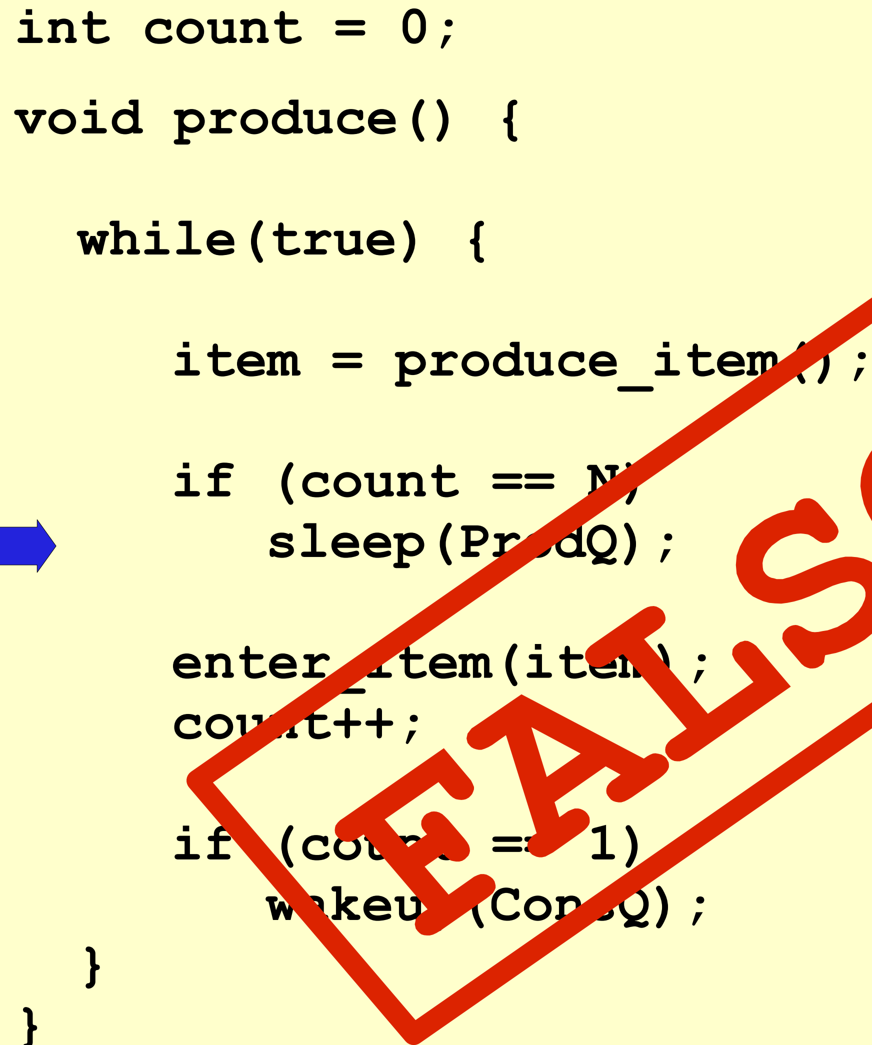
```
int count = 0;
void produce() {

  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

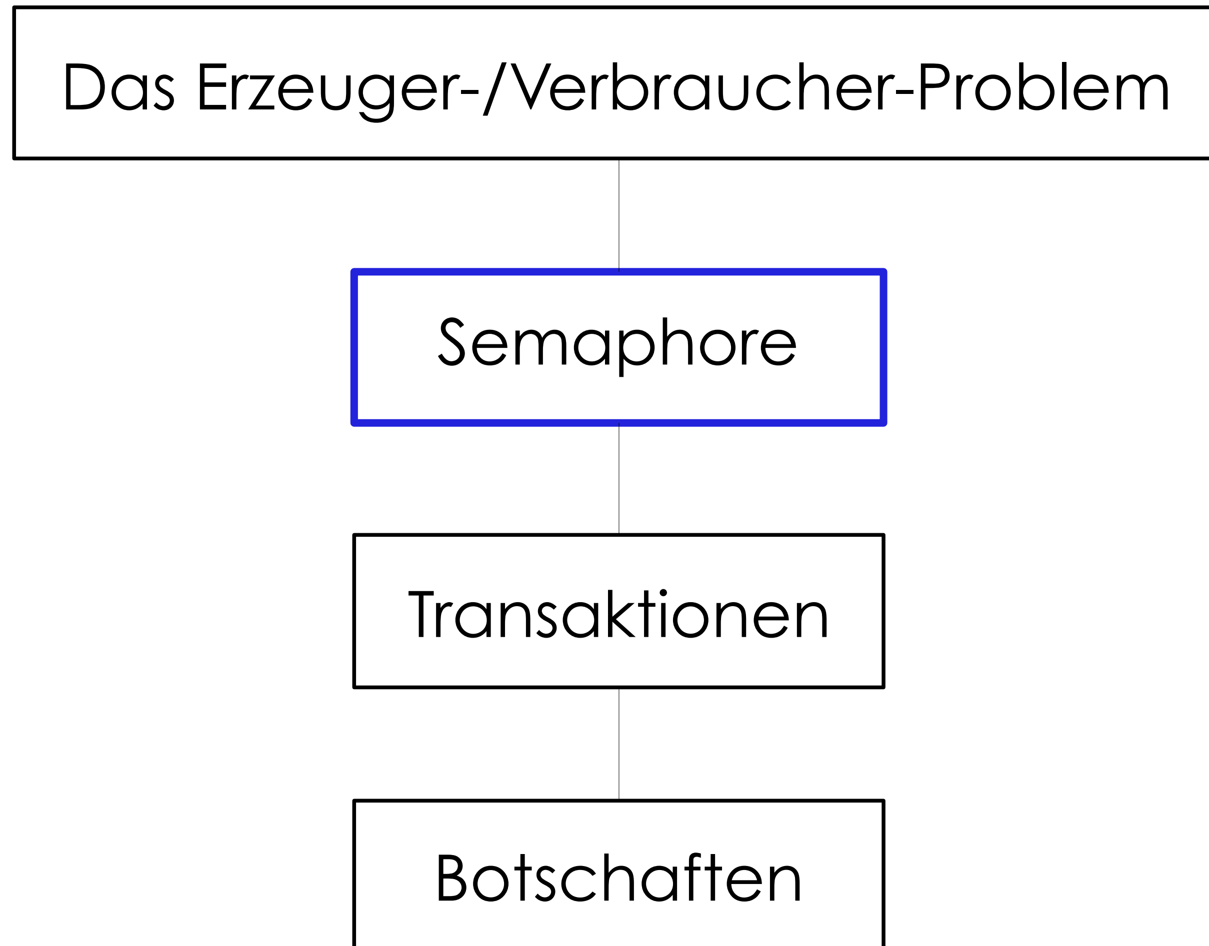
Erzeuger



Verbraucher

```
void consume() {

  while(true) {

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

    item = remove_item();
    count--;

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

    consume_item(item);
  }
}
```

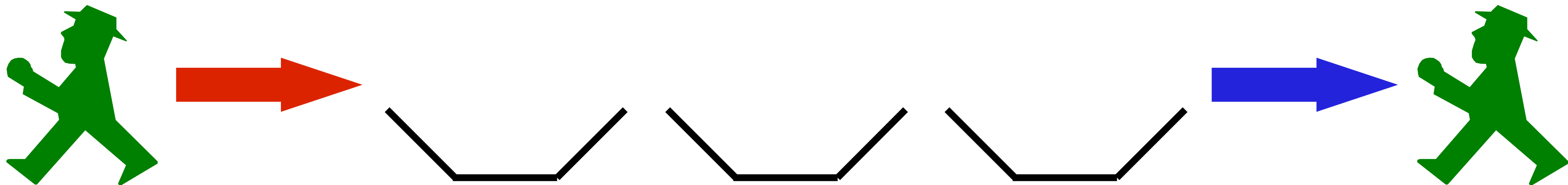
Wegweiser



Semaphore

```
class SemaphoreT {

 public:

   SemaphoreT(int howMany);
   //Konstruktor

   void P();
   //passeren = betreten

   void V();
   //verlaten = verlassen

}
```

Intuition (aber falsch)




```
SemaphoreT E_V(3);

E_V.P();

enter_item();
```

```
remove_item();


E_V.V();
```

Kritischer Abschnitt mit Semaphoren

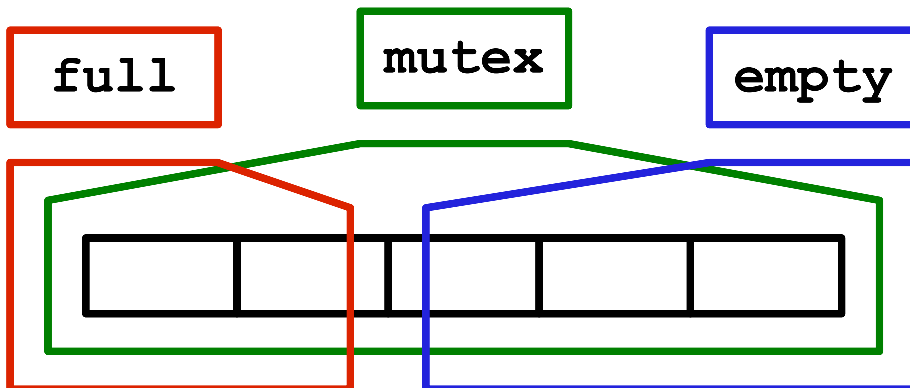
Wechselseitiger Ausschluss

```
SemaphoreT mutex(1);

mutex.P();

do_critical_stuff();
//Kritischer Abschnitt

mutex.V();
```

Beschränkung der Threadanzahl im KA

```
SemaphoreT mutex(42);

mutex.P();

do_critical_stuff();
//Kritischer Abschnitt

mutex.V();
```

EV-Problem mit Semaphoren




```
#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

Erzeuger

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

    enter_item(item);


  }
}
```

Verbraucher

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


    item = remove_item();

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

EV-Problem mit Semaphoren

Erzeuger

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

    enter_item(item);

    NOFfull.V();
  }
}
```

Verbraucher

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

    item = remove_item();

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

EV-Problem mit Semaphoren

Erzeuger

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

    NOFempty.P();
    mutex.P();

    enter_item(item);

    mutex.V();
    NOFfull.V();

  }
}
```

Verbraucher

```
void consume() {
  while (true) {
    NOFfull.P();
    mutex.P();

    item = remove_item();

    mutex.V();
    NOFempty.V();

    consume_item(item);

  }
}
```

Versehentlicher Tausch der Semaphor-Ops

Erzeuger

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

    NOFempty.P();
    mutex.P();

    enter_item(item);

    mutex.V();
    NOFfull.V();

  }
}
```

Verbraucher

```
void consume() {
  while (true) {
    mutex.P();
    NOFfull.P();

    item = remove_item();

    mutex.V();
    NOFempty.V();

    consume_item(item);

  }
}
```

DEADLOCK

Semaphor-Implementierung

```
class SemaphoreT {

  int count;
  QueueT queue;

 public:

  SemaphoreT(int howMany);

  void P();
  void V();

}

SemaphoreT::SemaphoreT(
        int howMany) {

  count = howMany;
}
```

```
SemaphoreT::P() {

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

  count--;
}

SemaphoreT::V() {

  count++;

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

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

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

```
monitor MyMonitorT{

  condition sync_queue;

  ...

  void atomar_insert(x);

  void atomar_delete(x);

  void atomar_calc_sum();

}
```

EV-Problem mit einem Monitor

```
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

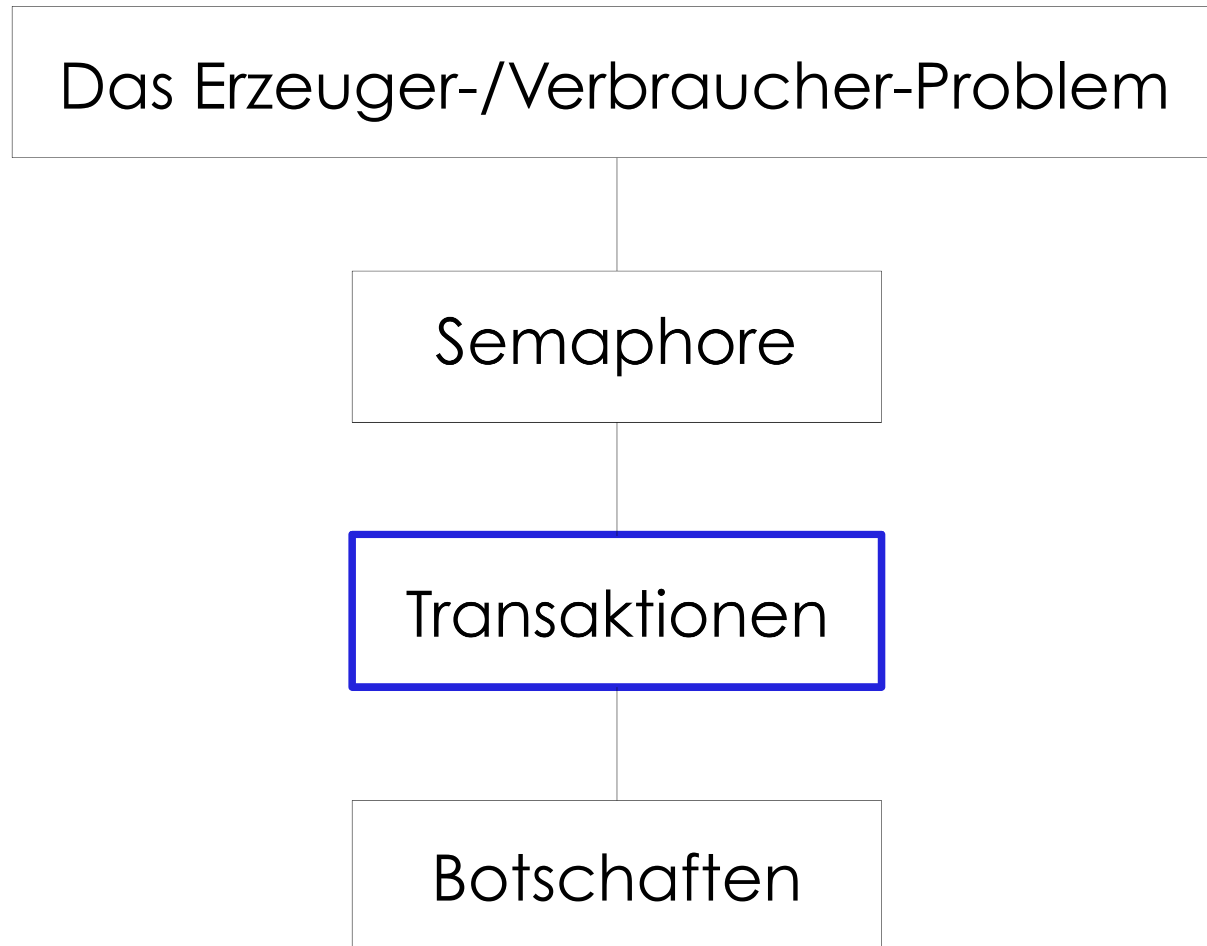
- beteiligte Objekte vor dem Zugriff sperren
- erst nach Abschluss der Gesamtoperation entsperren

```
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



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. Cray T3D