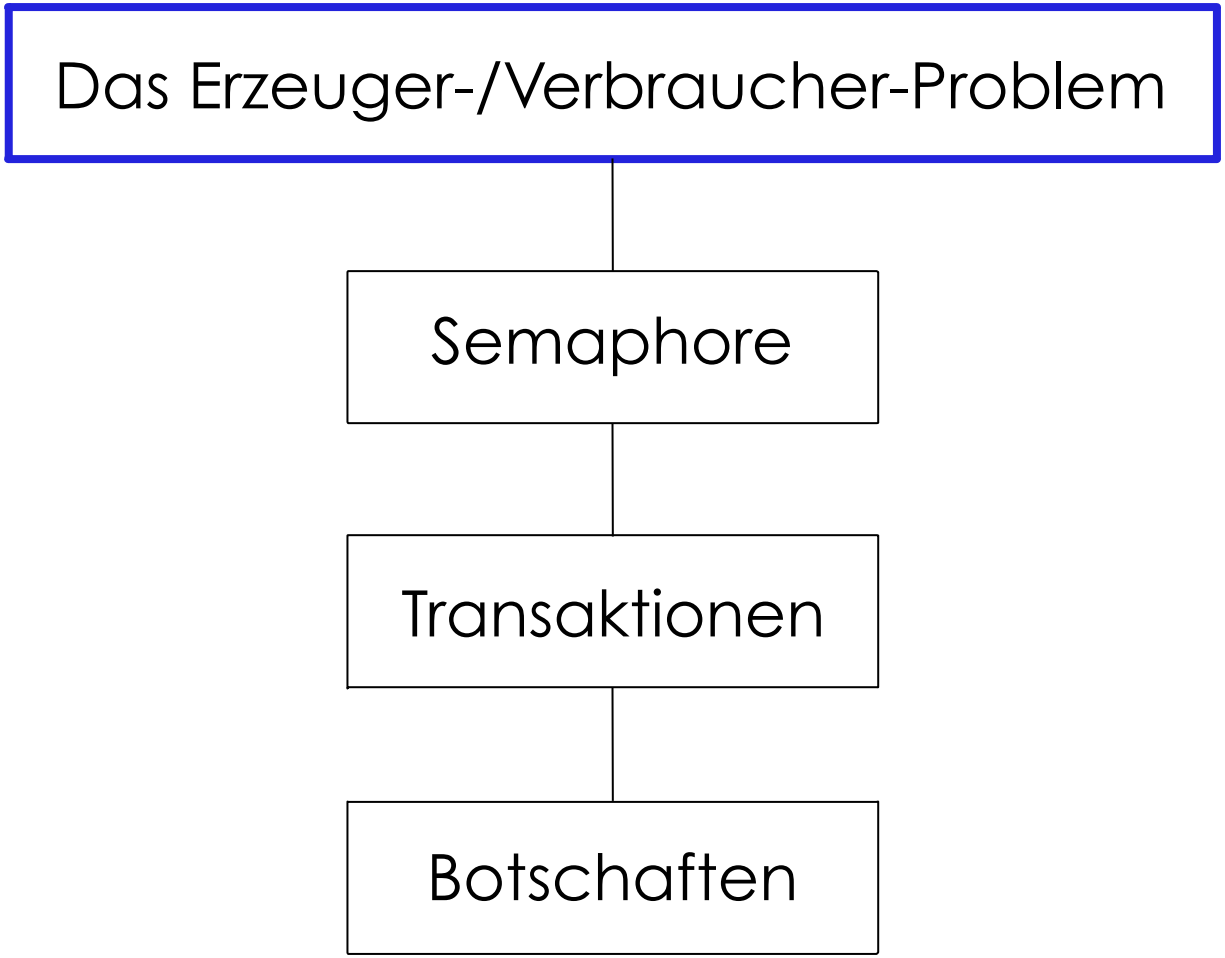
Wegweiser


Das Erzeuger-/Verbraucher-Problem
Semaphore
Transaktionen
Botschaften

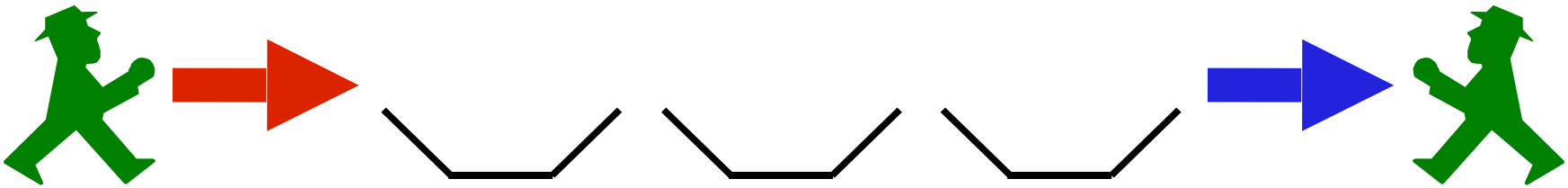
Erzeuger-/Verbraucher-Probleme

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 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

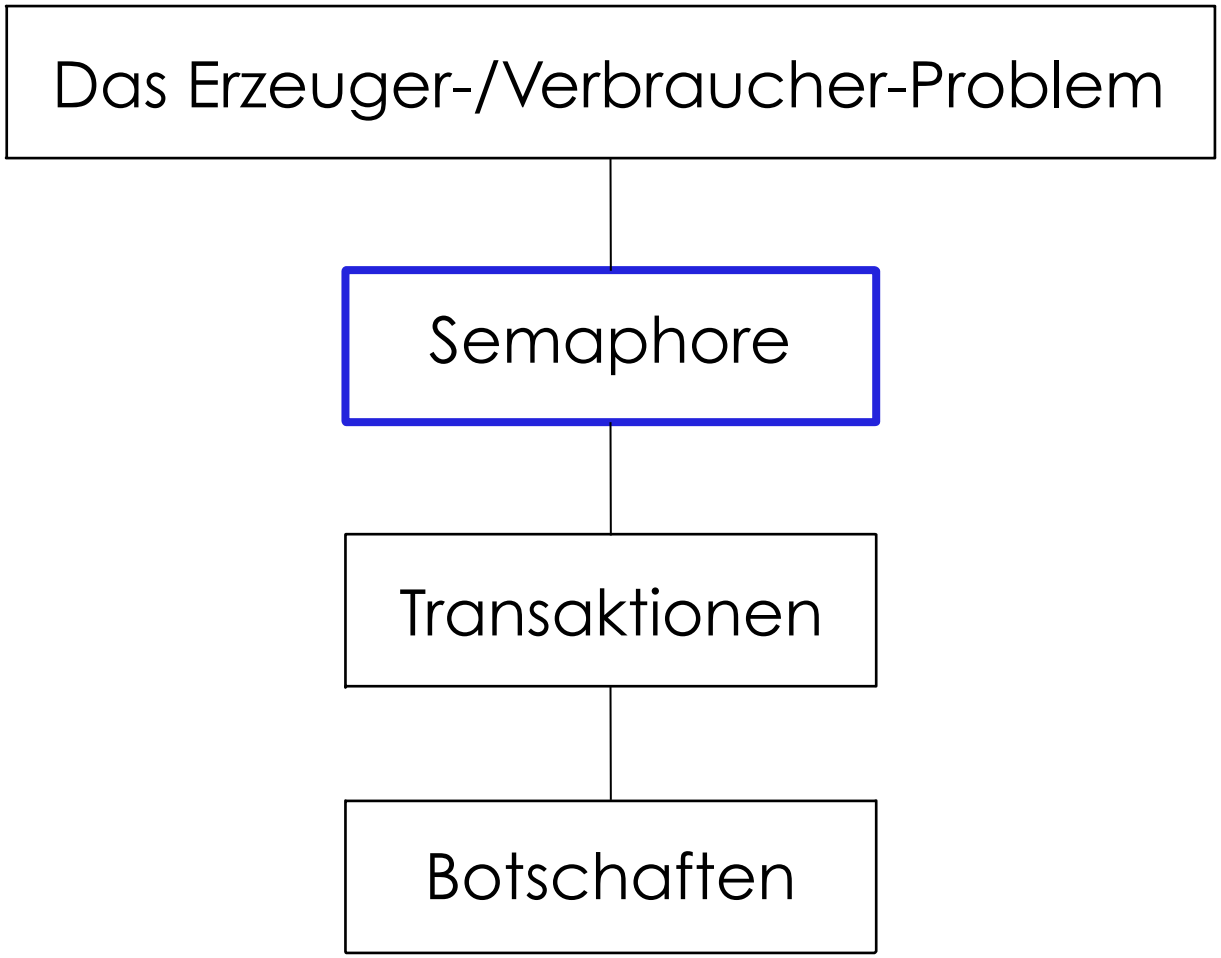
```
int count = 0;
void produce() {

  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();
//Kritischer Abschnitt

mutex.up();
```

```
mutex.down();

do_critical_stuff();
//Kritischer Abschnitt

mutex.up();
```

Drei Threads können kritischen Abschnitt betreten

Auf dem Weg zu Erzeuger/Verbraucher

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

```
mutex.down();

  enter_item(item)
```

```
  remove_item(item)

mutex.up();
```

Maximal 3 item können in den Puffer

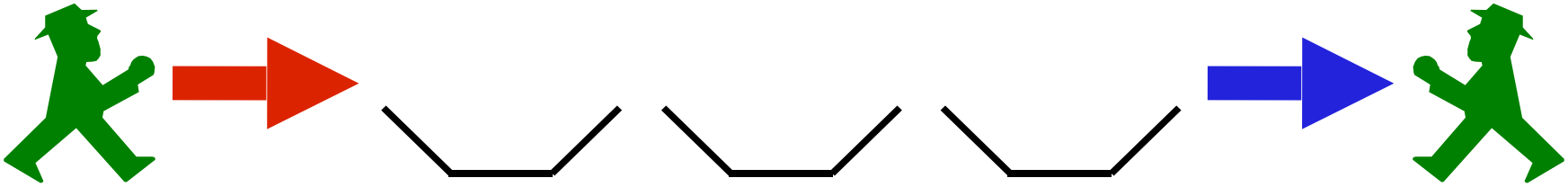
Semaphor-Implementierung

```
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)

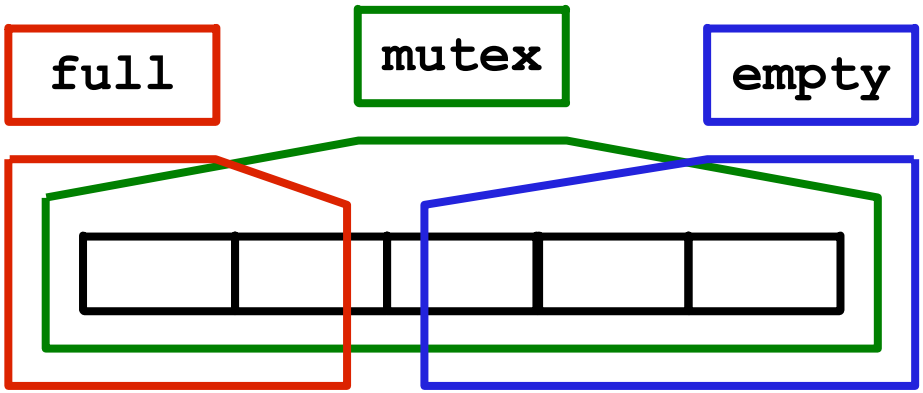
Erzeuger-Thread ***3-Elemente*** Puffer Verbraucher-Thread



```
SemaphoreT E_V(3);

E_V.down();

enter_item();
```

```
remove_item();

E_V.up();
```

EV-Problem mit Semaphoren




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

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

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

SemaphoreT full(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();
    empty.down();

    enter_item(item);



  }
}
```

Verbraucher

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



    item = remove_item();

    empty.up();

    consume_item(item);

  }
}
```

EV-Problem mit Semaphoren

Erzeuger

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

    enter_item(item);

    full.up();
  }
}
```

Verbraucher

```
void consume() {
  while (true) {
    full.down();

    item = remove_item();

    empty.up();
    consume_item(item);
  }
}
```

EV-Problem mit Semaphoren

Erzeuger

```
void produce() {

  while (true) {

    item = produce_item();

    empty.down();
    mutex.down();

    enter_item(item);

    mutex.up();
    full.up();

  }
}
```

Verbraucher

```
void consume() {

  while (true) {

    full.down();
    mutex.down();

    item = remove_item();

    mutex.up();
    empty.up();

    consume_item(item);

  }
}
```

Versehentlicher Tausch der Semaphor-Ops

Erzeuger

```
void produce() {

  while (true) {

    item = produce_item();

    empty.down();
    mutex.down();

    enter_item(item);

    mutex.up();
    full.up();

  }
}
```

Verbraucher

```
void consume() {

  while (true) {

    mutex.down();
    full.down();

    item = remove_item();

    mutex.up();
    empty.up();

    consume_item(item);

  }
}
```

DEADLOCK

Bedingungsvariablen

- Semaphore verbinden Zähler und Warteschlange
- Bedingungsvariablen stellen nur eine Warteschlange bereit
- Operationen: **`wait()`** und **`signal()`**
- Bedingung frei implementierbar
- Test der Bedingung muss mit Lock geschützt werden
- dieses Lock wird bei **`wait()`** atomar freigegeben und beim Aufwachen wieder gesperrt

EV-Problem mit einem Monitor

Erzeuger

```
void produce() {


  enter_item();
  count++;



}
```

Verbraucher

```
void consume() {


  remove_item();
  count--;



}
```

EV-Problem mit einem Monitor

Erzeuger

```
void produce() {

  lock(mutex);




  enter_item();
  count++;




  unlock(mutex);

}
```

Verbraucher

```
void consume() {

  lock(mutex);




  remove_item();
  count--;




  unlock(mutex);

}
```

EV-Problem mit einem Monitor

Erzeuger

```
void produce() {

  lock(mutex);

  if (count == N)
     wait(not_full,mutex);

  enter_item();
  count++;




  unlock(mutex);

}
```

Verbraucher

```
void consume() {

  lock(mutex);




  remove_item();
  count--;

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

  unlock(mutex);

}
```

EV-Problem mit einem Monitor

Erzeuger

```
void produce() {

  lock(mutex);

  if (count == N)
     wait(not_full,mutex);

  enter_item();
  count++;

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

  unlock(mutex);

}
```

Verbraucher

```
void consume() {

  lock(mutex);

  if (count == 0)
     wait(not_empty,mutex);

  remove_item();
  count--;

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

  unlock(mutex);

}
```

Monitore

- Sprachkonstrukt
- Datentyp der von mehreren Threads genutzte Daten und darauf definierte Operationen bereitstellt
- Operationen können kritische Abschnitte enthalten, welche unter gegenseitigem Ausschluss ablaufen
- Implementierung mittels Locks oder Bedingungsvariablen

```
monitor MyMonitorT{

   condition sync_queue;

   ...

   void atomic_insert(x);

   void atomic_delete(x);

   void atomic_calc_sum();

}
```

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.gutschrift(Betrag);

}
```

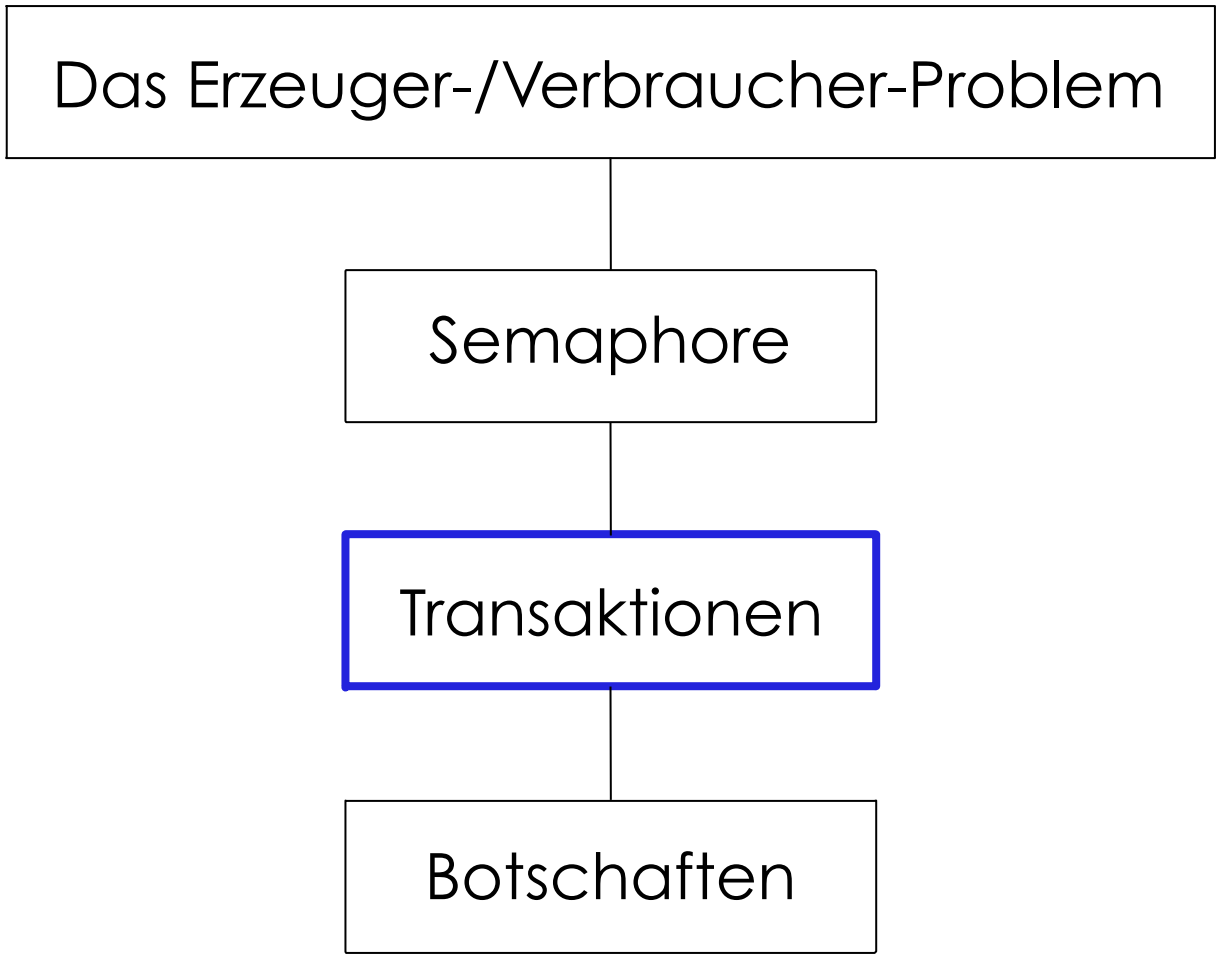
Bei mehreren beteiligten Objekten

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.
  gutschrift(Betrag)) {

   Konto1.gutschrift(Betrag);
 }

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

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:
  mehr dazu in der Datenbanken-Vorlesung
- 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.

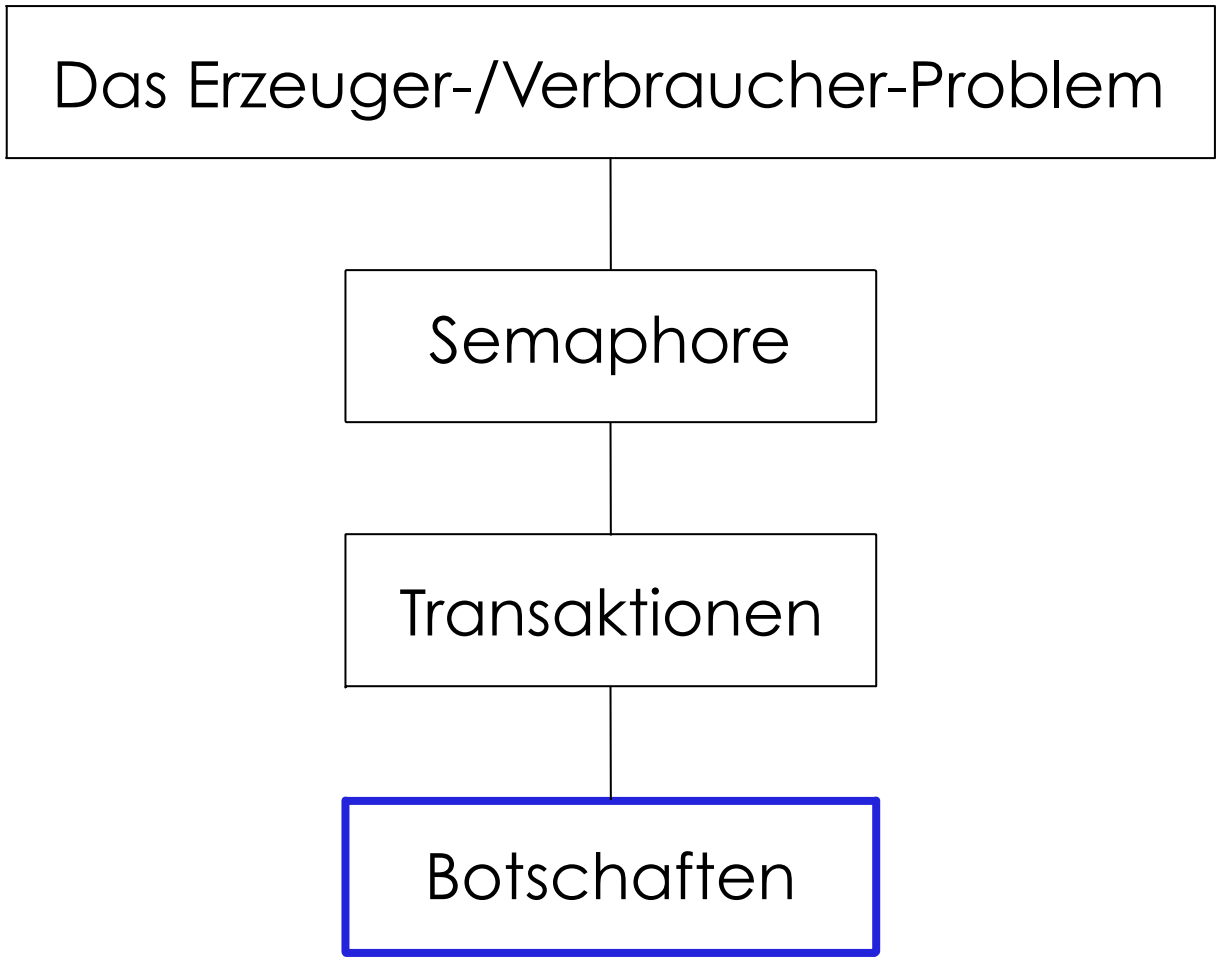
Nachteile von Semaphoren etc.

→ 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

Wegweiser


Das Erzeuger-/Verbraucher-Problem
Semaphore
Transaktionen
Botschaften

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

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() {



 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);

 }
}
```

Wegweiser

Zusammenspiel (Kommunikation) mehrerer Threads

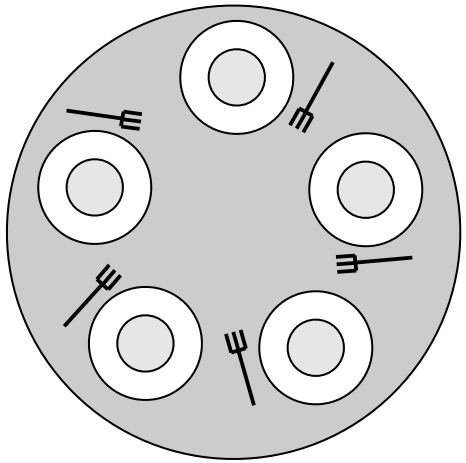
Einige typische Probleme

- Erzeuger / Verbraucher
- 5 Philosophen
- Leser/Schreiber

Scheduling

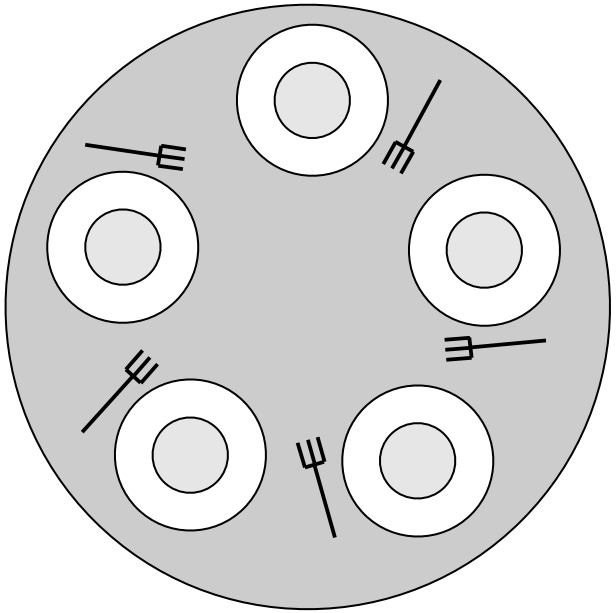
Dining Philosophers

- 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



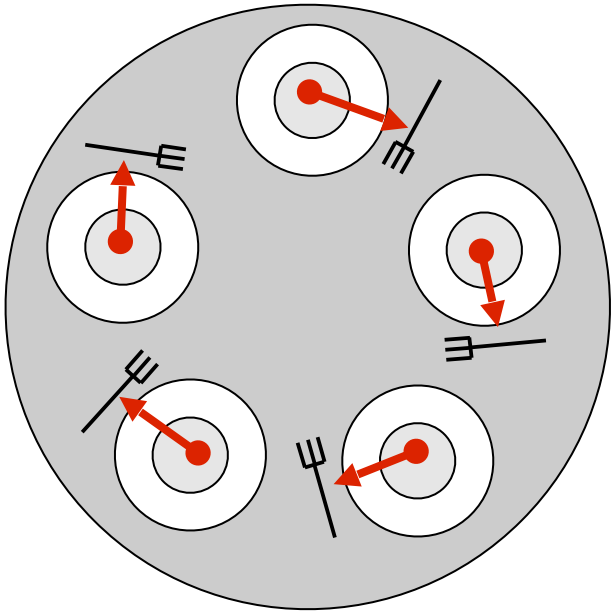
Die offensichtliche (...) Lösung

```
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

Scheduling

Aufgabe:

Entscheidung über Prozessorzuteilung

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

Scheduling

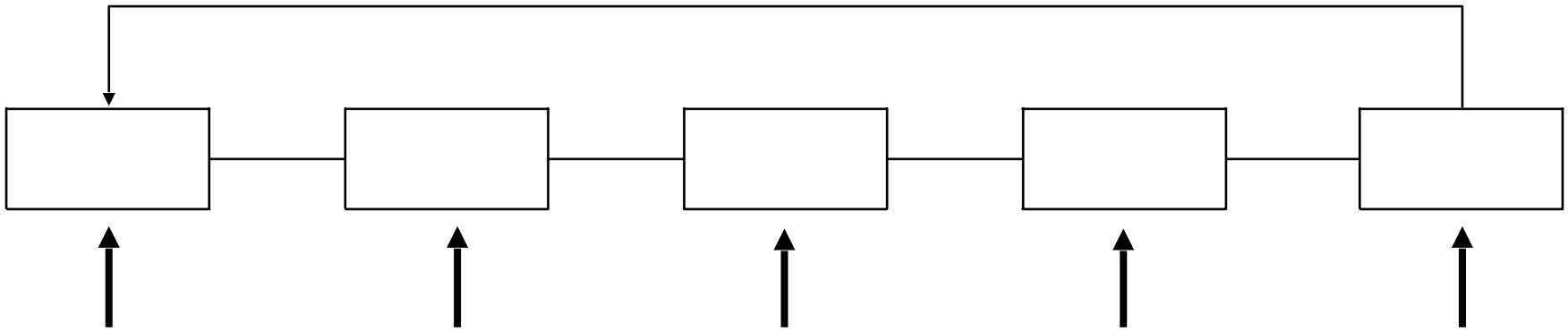
Kriterien

- 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
- Energie

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“)

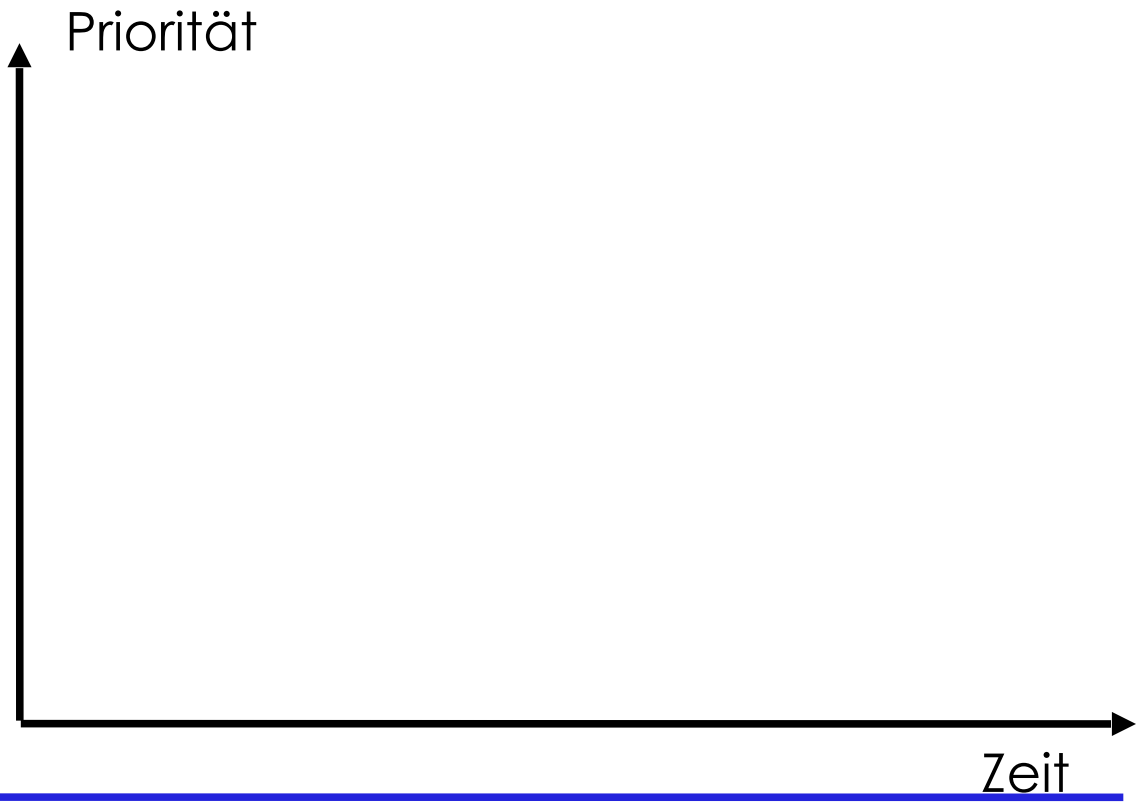
→ häufig: Kombination Round Robin und Prioritäten

„Prioritätsumkehr“ (Priority Inversion)

Beispielszenario:

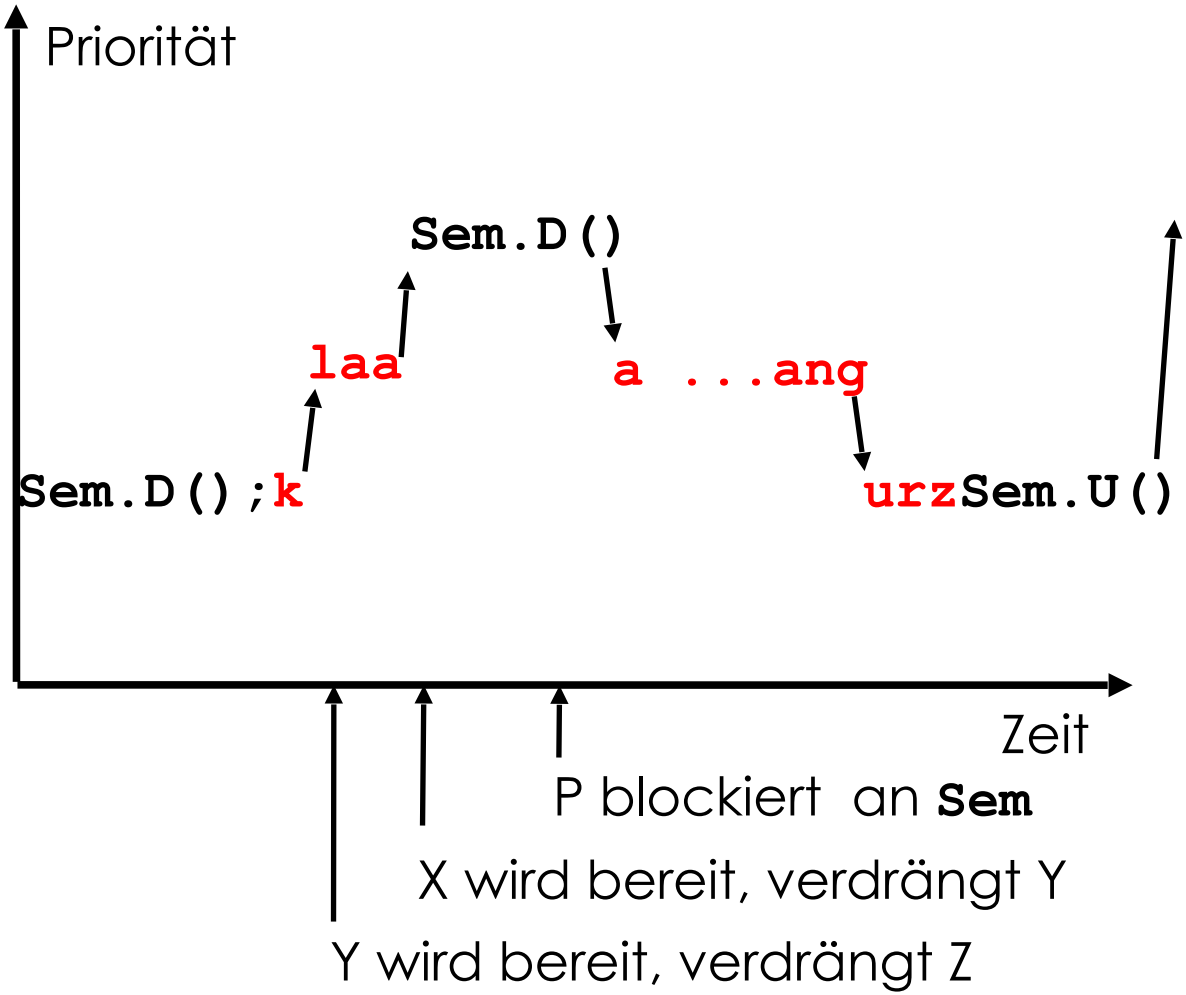
- drei Threads X, Y, Z
- X höchste, Z niedrigste Priorität
- werden bereit in der Reihenfolge: Z, Y, X

```
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 kann Z den Semaphor nicht freigeben.

Zusammenfassung Threads

- Threads sind ein mächtiges Konzept zur Konstruktion von Systemen, die mit asynchronen Ereignissen und Parallelität umgehen sollen.
- Kooperierende parallele Threads müssen sorgfältig synchronisiert werden. Fehler dabei sind extrem schwer zu finden, da zeitabhängig.
- Bei der Implementierung von Synchronisationsprimitiven auf die Kriterien Sicherheit und Lebendigkeit achten.
- Durch Blockieren im Kern (z.B. mittels Semaphoren) kann Busy Waiting vermieden werden.
- Standardlösungen für typische Probleme, wie das Erzeuger/Verbraucher-Problem