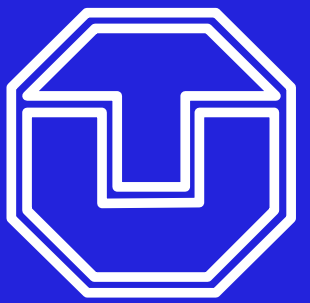
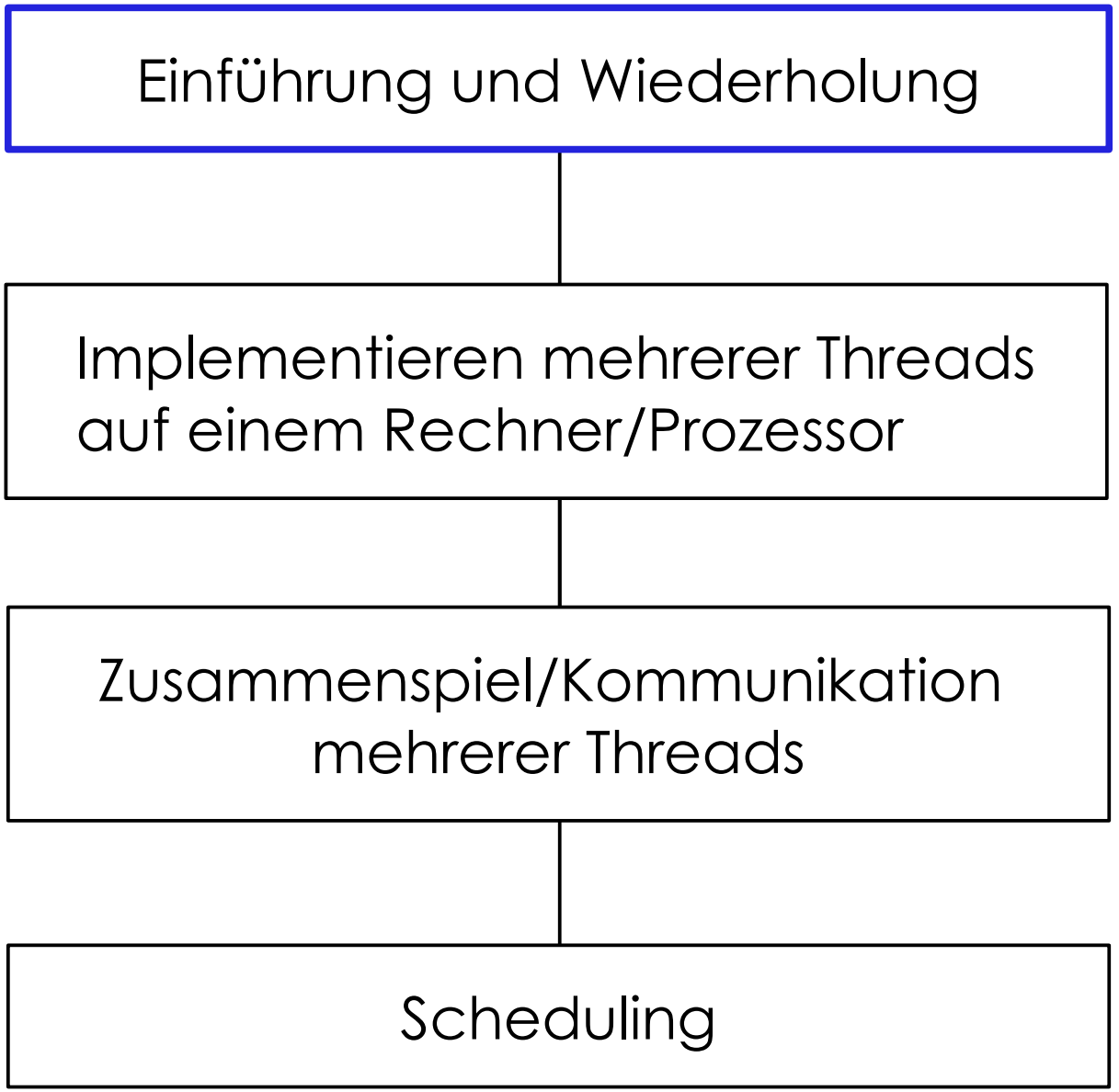
Threads

Betriebssysteme

Hermann Härtig
TU Dresden



Wegweiser

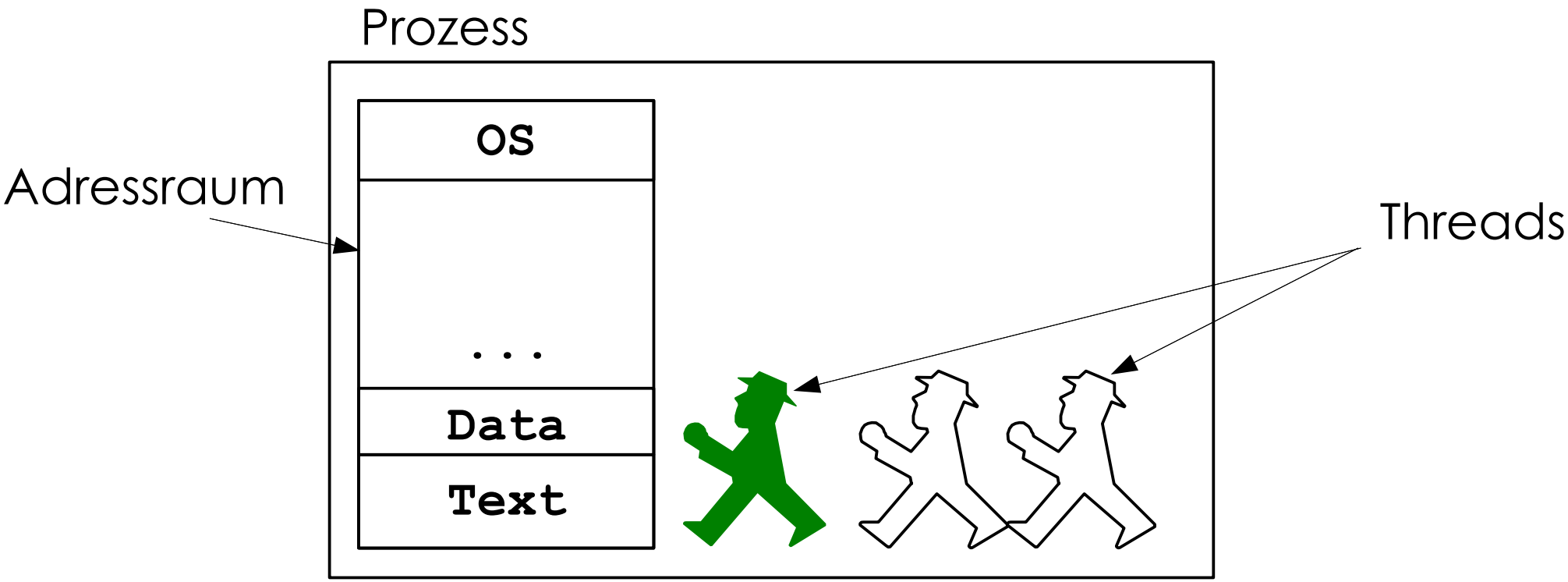

Einführung und Wiederholung
Implementieren mehrerer Threads auf einem Rechner/Prozessor
Zusammenspiel/Kommunikation mehrerer Threads
Scheduling

Definition: Thread

Eine selbständige

- ein sequentielles Programm ausführende
- zu anderen Threads parallel arbeitende
- von einem Betriebssystem zur Verfügung gestellte

Aktivität.



Notationen

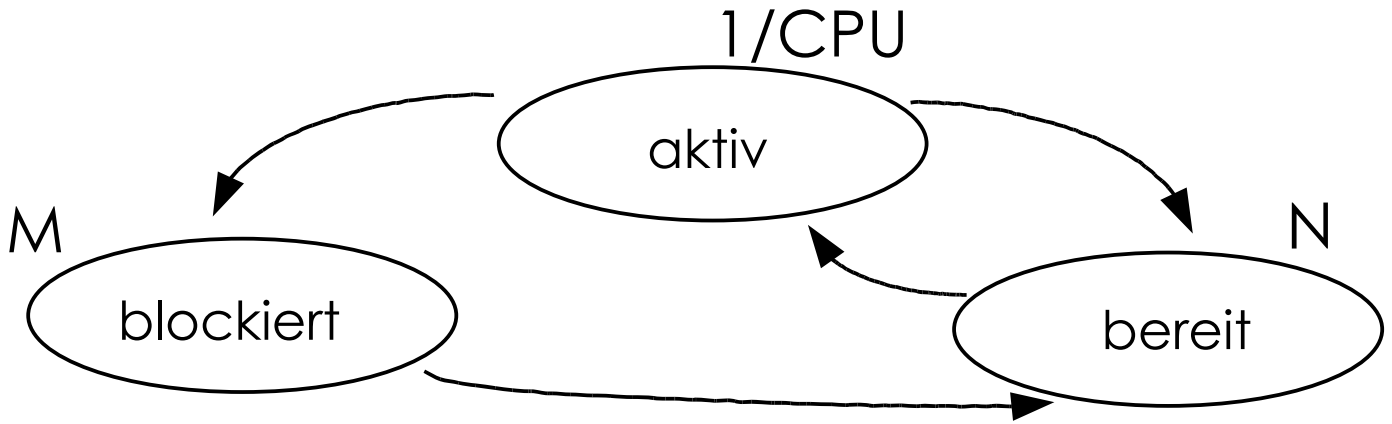
- P||Q

- ```
  create(Prozedur,Stack);
  join(thread);
  ```

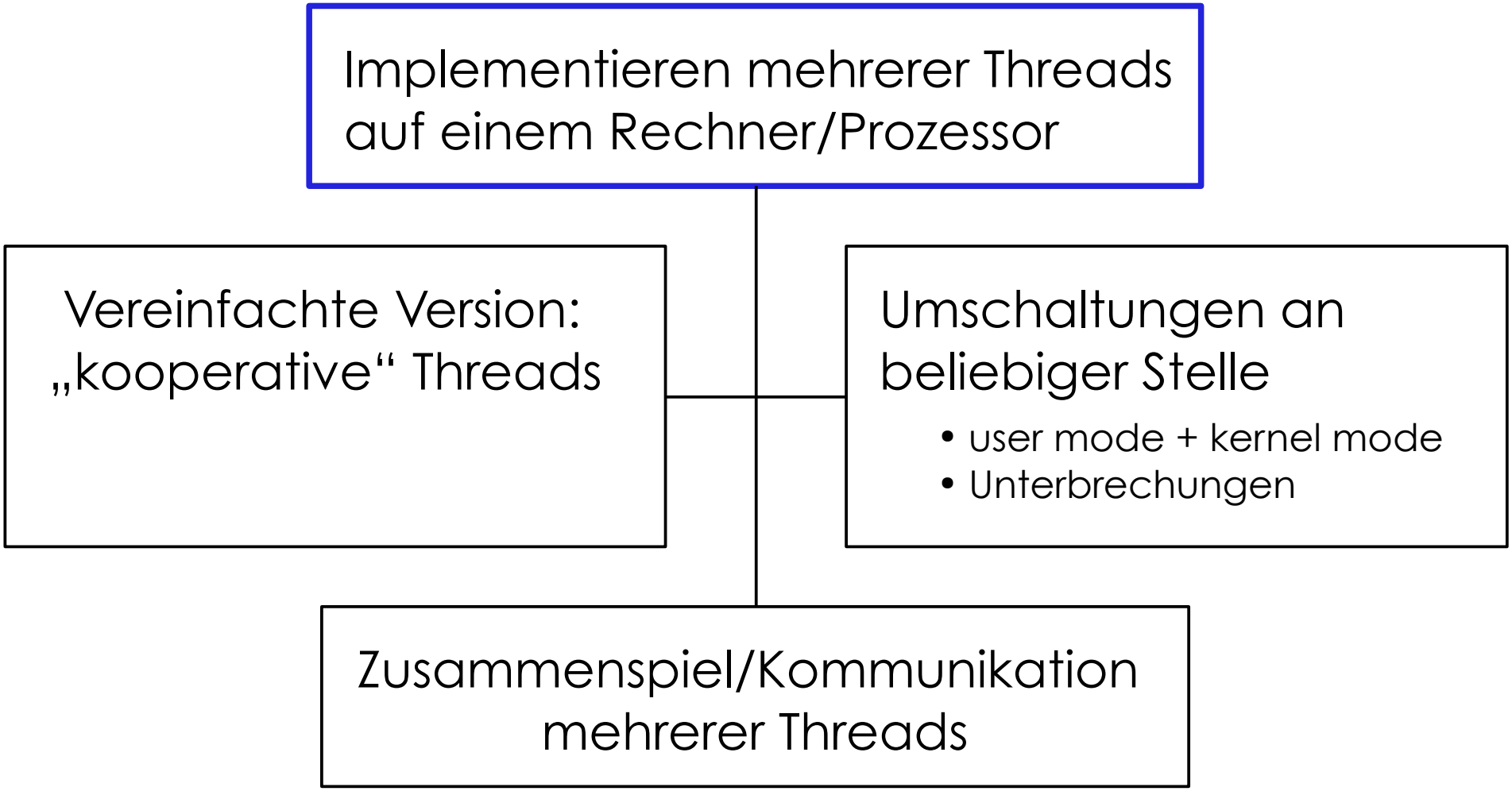
- ```
  COBEGIN
     P(Params); Q();
  COEND
  ```

Thread-Zustände

- Threads können gerade auf der CPU ausgeführt werden: „aktiv“ oder „rechnend“ / „running“
- Threads können „blockiert“ sein: sie warten auf ein Ereignis (z. B. Botschaft), um weiterarbeiten zu können.
  Laufen mehrere Threads auf einem Rechner, muss dann die CPU für andere Threads freigegeben werden.
- Threads können „nicht blockiert“, aber auch „nicht rechnend“ sein – also „bereit“ / „ready“



Wegweiser


Implementieren mehrerer Threads auf einem Rechner/Prozessor
Vereinfachte Version: „kooperative“ Threads
Umschaltungen an beliebiger Stelle
• user mode + kernel mode
• Unterbrechungen
Zusammenspiel/Kommunikation mehrerer Threads

Randbedingungen

- zu jeder Zeit ist höchstens ein Thread (pro CPU) aktiv
- ein aktiver Thread ist genau einer CPU zugeordnet (zu einem bestimmten Zeitpunkt)
- nur die bereiten Threads erhalten CPU (werden „aktiv“)
- „fair”: jeder Thread erhält angemessenen Anteil CPU-Zeit; kein Thread darf CPU für sich allein beanspruchen
- Wohlverhalten von Threads darf bei der Implementierung von Threads keine Voraussetzung sein
  z. B.: `while (true) {bla();}` darf nicht dazu führen, dass andere Threads nie wieder „drankommen“

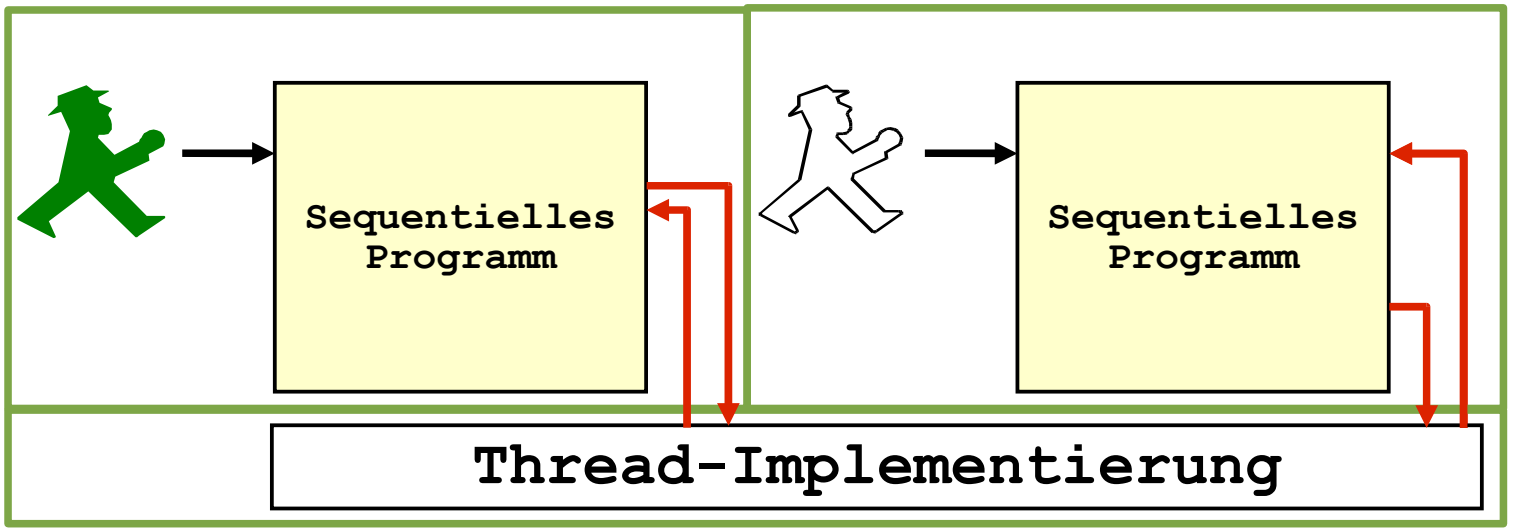
Kooperative vs. preemptive Umschaltung

Umschaltung zwischen kooperativen Threads

Alle Threads rufen zuverlässig in bestimmten Abständen eine Umschaltoperation der Thread-Implementierung auf

Umschaltung ohne Kooperation an beliebigen Stellen

Thread wird zur Umschaltung gezwungen - „preemptiert“



Kooperative Umschaltung: Beispiel

Langlaufender Thread

```
Thread1 {

  Raytrace(Bildbereich[0]);
  schedule();

  Raytrace(Bildbereich[1]);
  schedule();

  Raytrace(Bildbereich[2]);
  schedule();

  Raytrace(Bildbereich[3]);
  schedule();
}
```

Periodischer Thread

```
Thread2 {

  while (true) {

     receive(mesg);
     // blockiert Thread2
     // bis zum Eintreffen
     // der Nachricht;
     // impliziert schedule

     handle(mesg);
  }
}
```

Preemptive Umschaltung: Beispiel

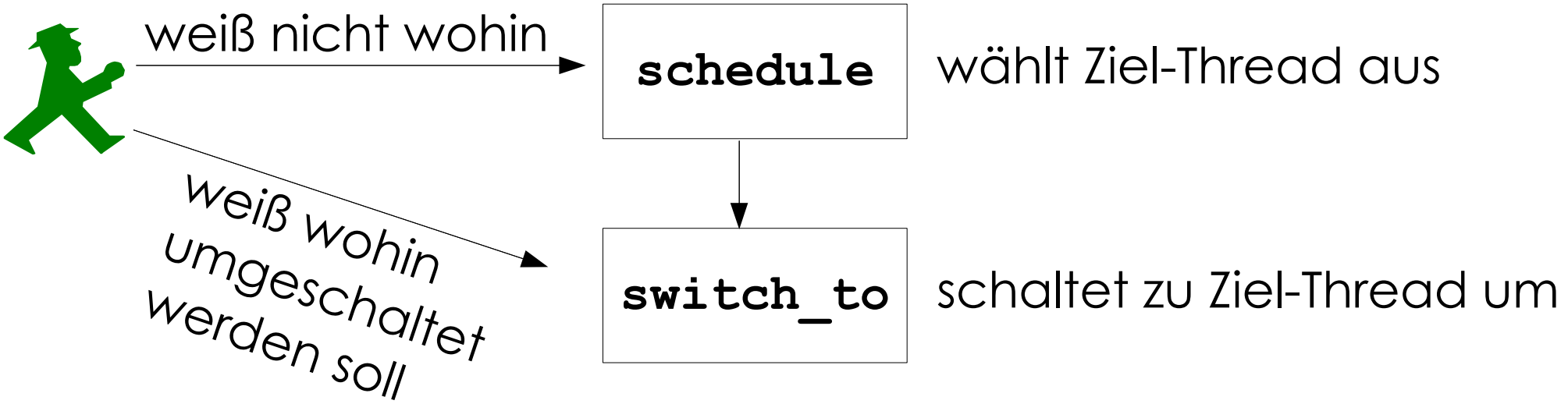
Langlaufender Thread

```
Thread1 {

  Raytrace(Bildbereich[0]);

  Raytr ...
  // durch Betriebssystem
  // erzwungene Umschaltung
  // weil Rechenzeit zu
  // Ende oder wichtigerer
  // Thread bereit wird;
  // später Fortsetzung an
  // alter Stelle
  ... ace(Bildbereich[1]);

  Raytrace(Bildbereich[2]);
  Raytrace(Bildbereich[3]);
}
```

Periodischer Thread

```
Thread2 {

  while (true) {

     receive(mesg);
     // blockiert Thread2
     // bis zum Eintreffen
     // der Nachricht;
     // impliziert schedule

     handle(mesg);
  }
}
```

Umschaltmechanismen



Umschaltmechanismen

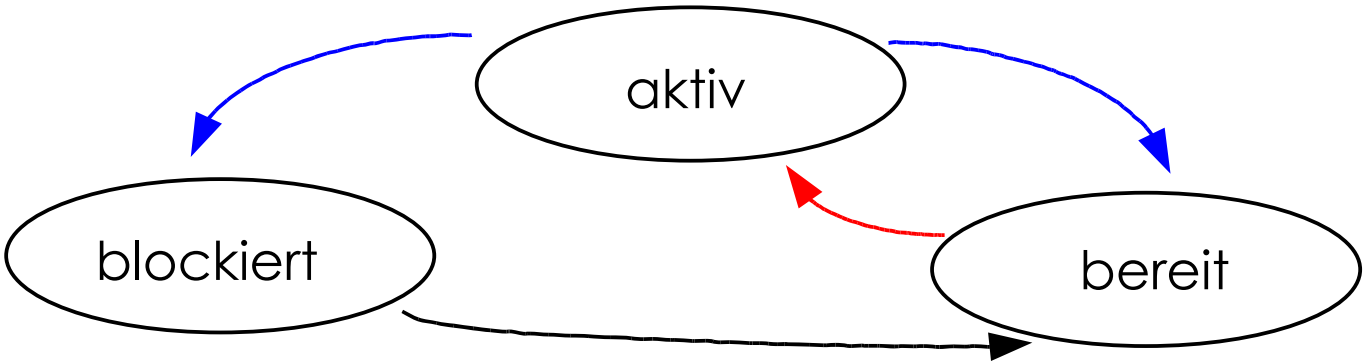
`Ziel-Thread = schedule()`

- Auswahl eines bereiten Threads, falls Ziel-Thread unbekannt
- ruft `switch_to` auf, um zum ausgewählten Thread umzuschalten

`switch_to(Ziel-Thread)`

- wird direkt aufgerufen, wenn Ziel-Thread bekannt
- schaltet vom aktiven Thread zum Ziel-Thread um
- wenn ein Thread wieder aktiv wird, wird er an der Stelle fortgesetzt, an der von ihm weggeschaltet wurde

Zustandsänderung bei Umschaltung



- aktiver Thread ändert Zustand auf
  - „blockiert“: wartet auf ein Ereignis (z. B. Nachricht)
  - „bereit“: Rechenzeit zu Ende oder wichtigerer Thread ist „bereit“ geworden
- danach Aufruf der Funktion: **`switch_to(Ziel-Thread)`**
- Ziel-Thread ändert Zustand von „bereit“ auf „aktiv“

Bewertung der kooperativen Umschaltung

- Nicht kooperierende Threads können System lahm legen
- Sehr aufwendig, Umschaltstellen im Vorhinein festzulegen

➔ heute sehr geringe Bedeutung, praktisch keine mehr in Betriebssystemen, Echtzeitsystemen, ...

Verbreitete Zwischenform

- Threads im Betriebssystemkern sind nicht preemptierbar, man vertraut darauf, dass Kern-Konstrukteure `switch_to()` einfügen.
- Threads, die Benutzerprogramme ausführen, sind jederzeit preemptierbar.

Betriebsmittel eines Threads

- Kern-Stack
- CPU State: CPU-Register, FPU-Register

➔ ein Nutzer eines (nicht kooperativen) Thread-Systems muss sich darauf verlassen können, dass nicht ein anderer Thread einen Teil seiner Register zerstört hat

- Thread-Zustand (aktiv, bereit, blockiert)
- Verweis auf Adressraum
- Scheduling-Attribute

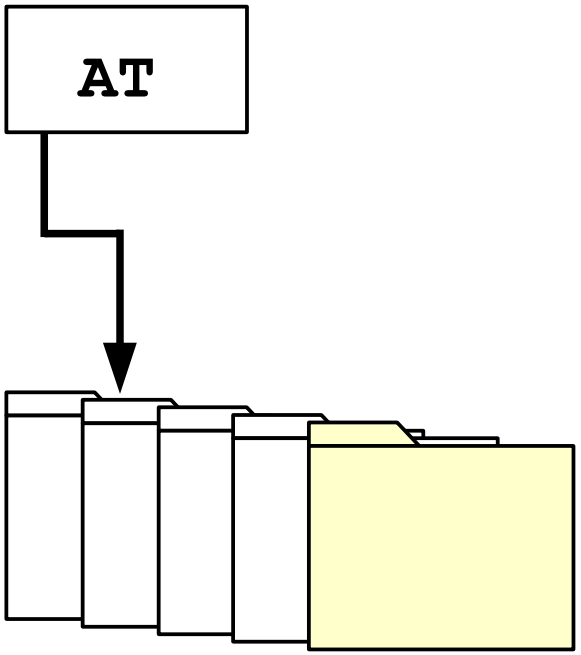
➔ **Thread Control Block (TCB)**

zentrale Struktur des Kerns zur Verwaltung eines Threads
(in den folgenden Graphiken lassen wir „Kern-Stack“ weg)

Thread Control Block (TCB) und TCB-Tabelle

- TCB-Tabelle (**`TCBTAB`**)
- aktiver Thread (**`AT`**)
  globale Variable der Thread-Implementierung

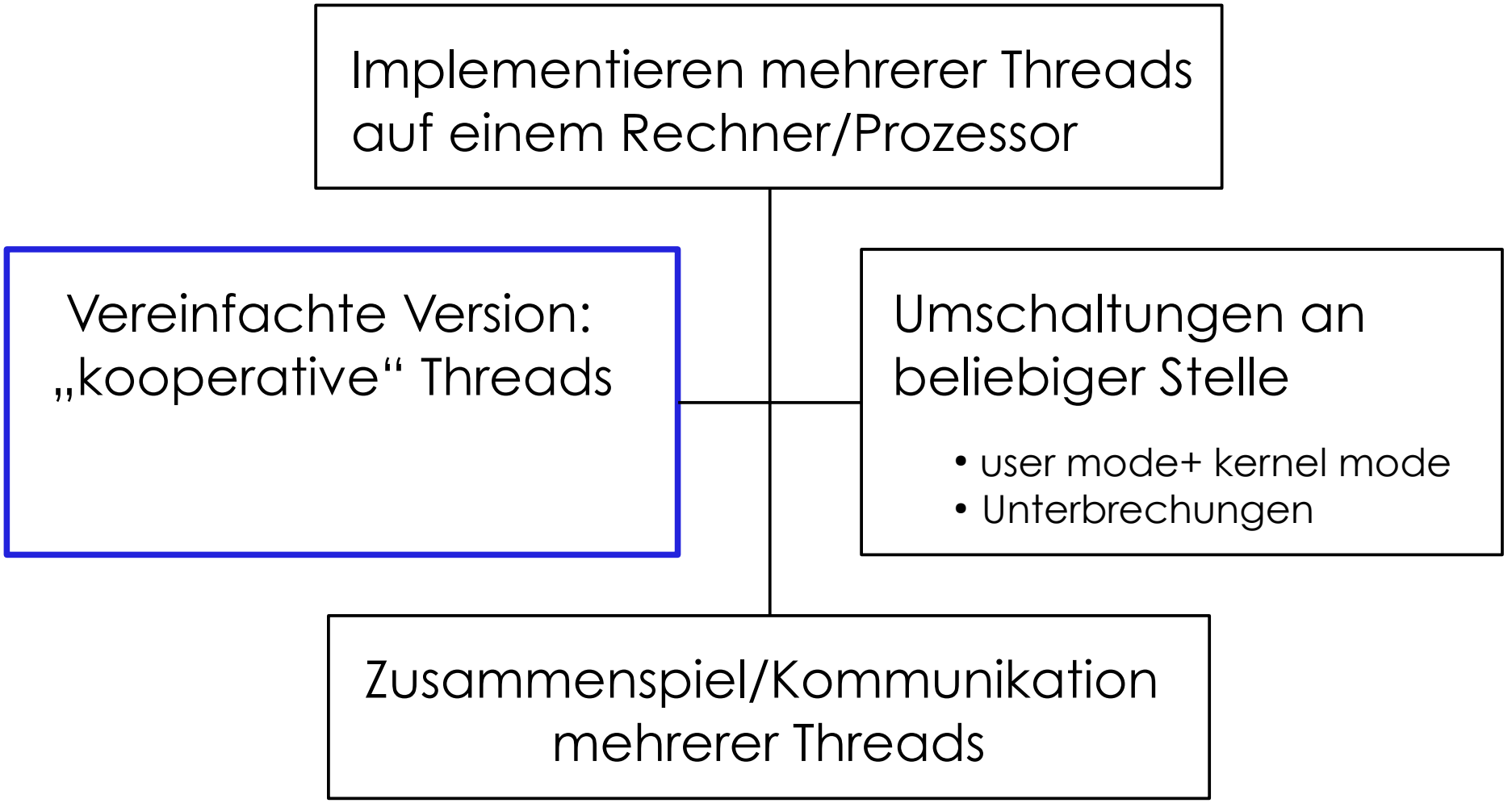
- Thread Control Block



Thread1

- Thread-Zustand:
  aktiv, bereit, blockiert
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

Wegweiser


Implementieren mehrerer Threads auf einem Rechner/Prozessor
Vereinfachte Version: „kooperative“ Threads
Umschaltungen an beliebiger Stelle
• user mode+ kernel mode
• Unterbrechungen
Zusammenspiel/Kommunikation mehrerer Threads

Ein paar Bezeichner

Im Folgenden:

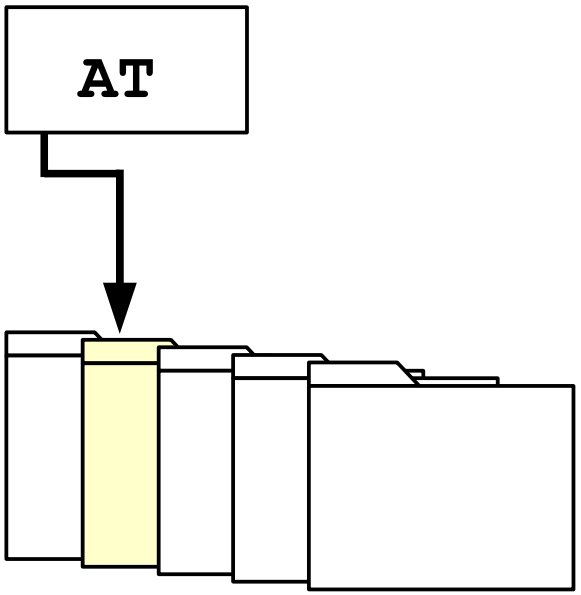
- **`TCBTAB[A]`** Eintrag in der TCB-Tabelle für Thread A
- **`AT`** aktiver Thread
- **`ZT`** Ziel-Thread

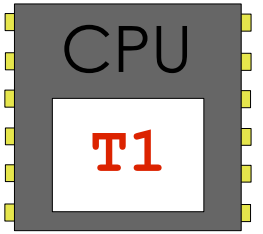
- **`store_CPU_state`** speichert Register in aktuellen TCB
- **`load_CPU_state`** restauriert Register aus aktuellem TCB

- **`SP`** Stack Pointer – Zeiger an die aktuelle Position im Stack
- **`PC`** Program Counter –
  Zeiger auf den nächsten auszuführenden Befehl

Thread-Umschaltung

```
switch_to(ZT)
{
  store_CPU_state();
  TCBTAB[AT].SP = SP;
  AT = ZT;
  SP = TCBTAB[AT].SP;
  load_CPU_state();
}
```



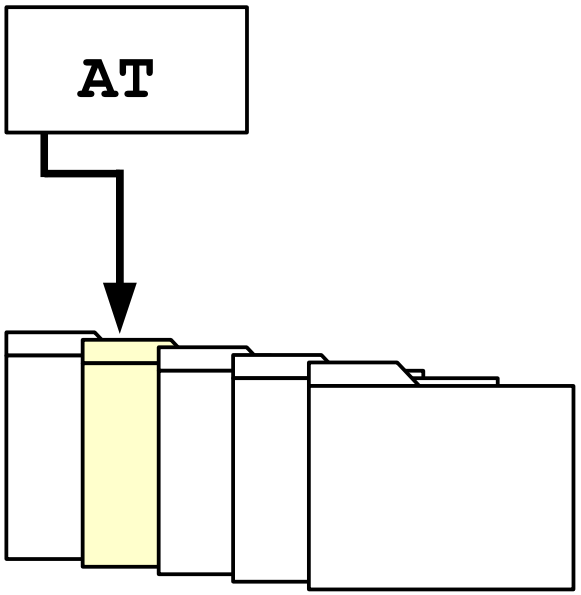


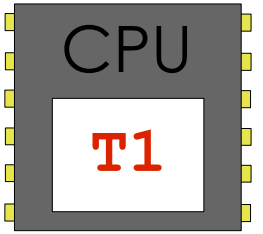
Thread1

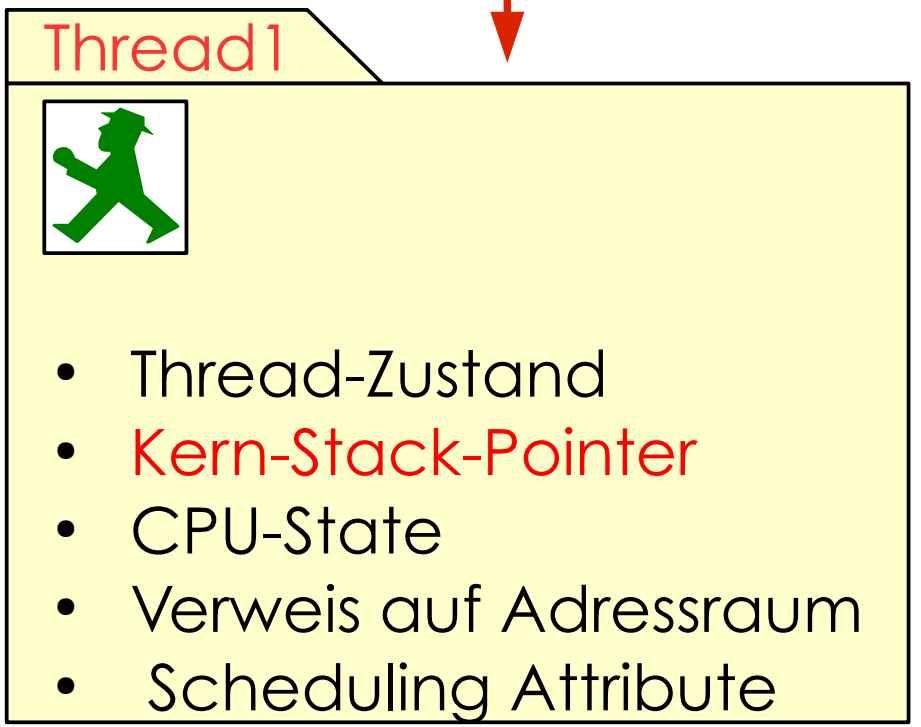
- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

Thread-Umschaltung

```
switch_to(ZT)
{
   store_CPU_state();
   TCBTAB[AT].SP = SP;
   AT = ZT;
   SP = TCBTAB[AT].SP;
   load_CPU_state();
}
```

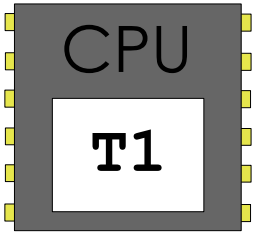


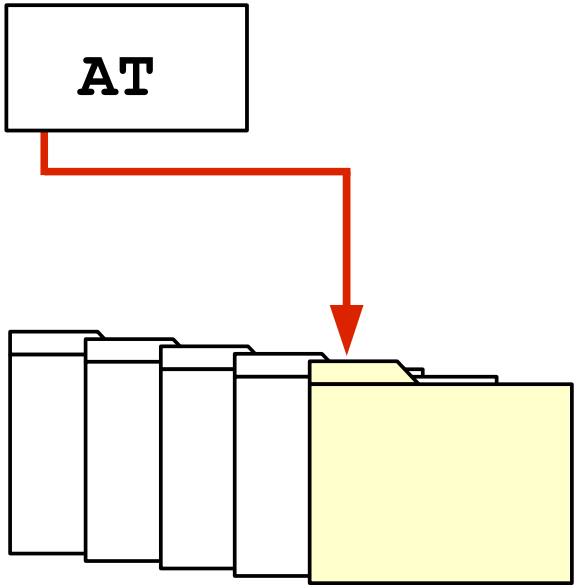




Thread-Umschaltung

```
switch_to(ZT)
{
   store_CPU_state();
   TCBTAB[AT].SP = SP;
   AT = ZT;
   SP = TCBTAB[AT].SP;
   load_CPU_state();
}
```



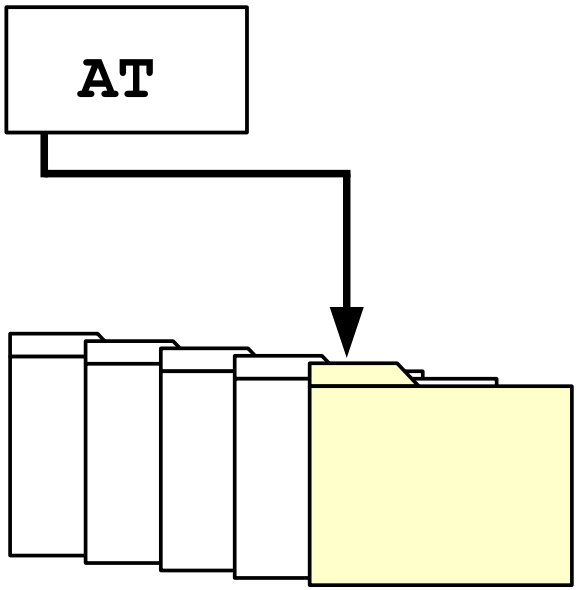


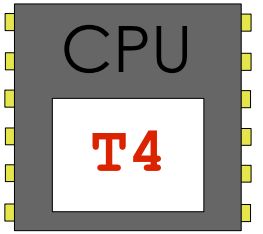
Thread4

- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

Thread-Umschaltung

```
switch_to(ZT)
{
   store_CPU_state();
   TCBTAB[AT].SP = SP;
   AT = ZT;
   SP = TCBTAB[AT].SP;
   load_CPU_state();
}
```



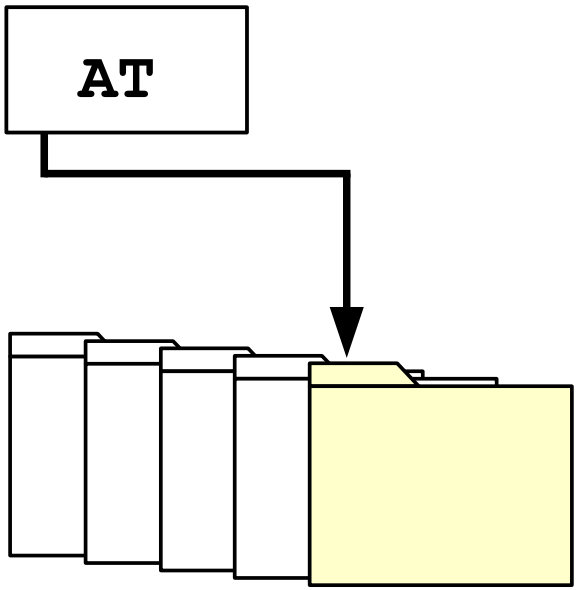


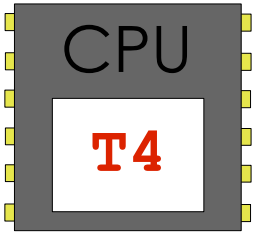
Thread4

- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

Thread-Umschaltung

```
switch_to(ZT)
{
   store_CPU_state();
   TCBTAB[AT].SP = SP;
   AT = ZT;
   SP = TCBTAB[AT].SP;
   load_CPU_state();
}
```



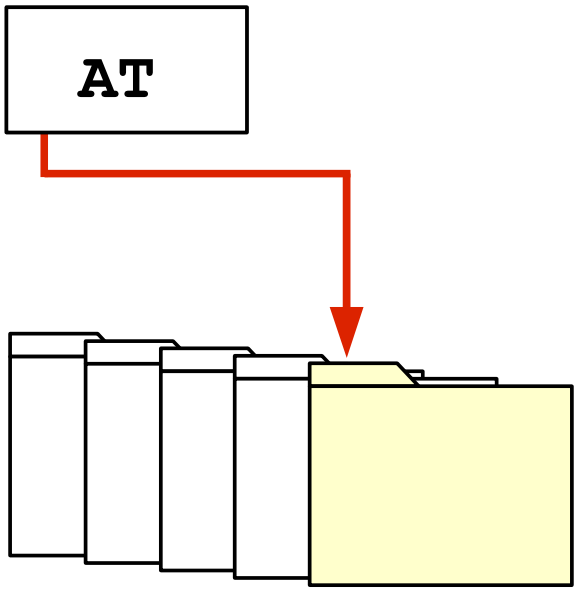


Thread4

- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

Thread-Erzeugung

```
switchto(Z,NT) {

   store_CPU_state();
   TTAB[AT].Zustand = Z;
   TTAB[AT].SP = SP;
   AT = NT;
   SP = TTAB[AT].SP;
   load_CPU_state();
}
```



Umschaltstelle

- alle nicht „aktiven“ Threads stehen im Kern an dieser Umschaltstelle
- neuer Thread:
  - finde freien TCB
  - initiale Register des Threads werden analog zu store_CPU_state() in TCB gespeichert
  - Ausführung des neuen Threads beginnt an dieser Umschaltstelle

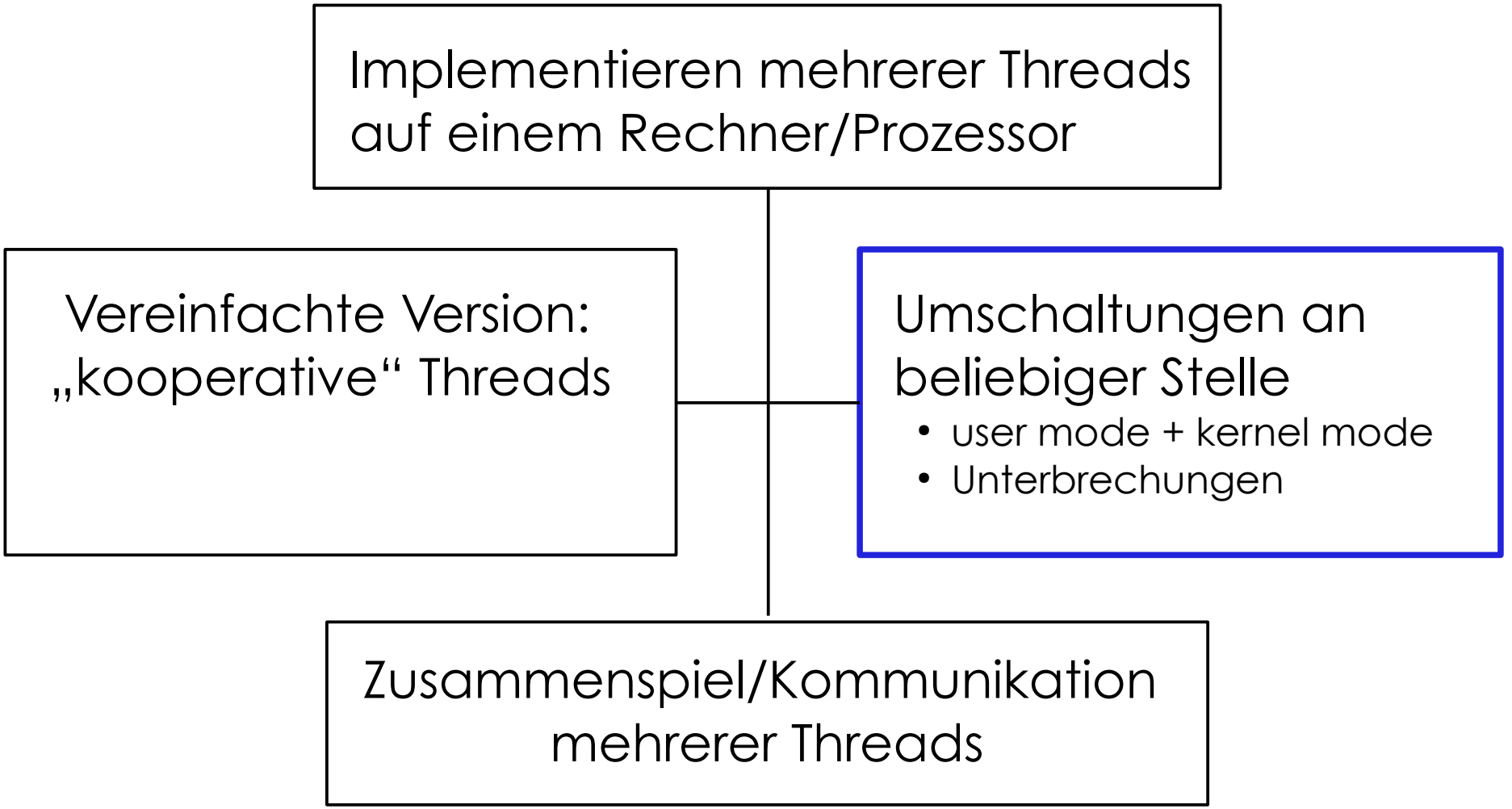
Ausblick: Scheduling

Auswahl des nächsten aktiven Threads aus der Menge der bereiten Threads

- zu bestimmten Punkten (Zeit, Ereignis)
- nach einem bestimmten Verfahren und einer Metrik für die Wichtigkeit jedes Threads (z. B. Priorität)

➔ Mehr dazu am Ende dieses Kapitels

Wegweiser


Implementieren mehrerer Threads auf einem Rechner/Prozessor
Vereinfachte Version: „kooperative“ Threads
Umschaltungen an beliebiger Stelle
• user mode + kernel mode
• Unterbrechungen
Zusammenspiel/Kommunikation mehrerer Threads

Wiederholung RA: Unterbrechungen

Unterbrechungen

- asynchron: Interrupts
- synchron: Exceptions

unterbrechen den Ablauf eines aktiven Threads an beliebiger Stelle

Auslöser von Interrupts

- E/A-Geräte – melden Erledigung asynchroner E/A-Aufträge
- Uhren (spezielle E/A-Geräte)

Auslöser von Exceptions

- Fehler bei Instruktionen (Division durch 0 etc.)
- Seitenfehler und Schutzfehler(ausgelöst durch MMU)
- explizites Auslösen – Systemaufrufe (Trap)

Ablauf von HW-Interrupts

- Gerätesteuerung (Controller) löst Unterbrechung aus
- CPU erkennt Anliegen von Interrupts zwischen Befehlen
- CPU schaltet auf Kern-Modus und Kern-Stack des aktiven Threads um und rettet dorthin:
  User-Stack-Pointer, User-PC, User-Flags, ...
- CPU lädt Kern-PC aus IDT (-> nächste Folie)

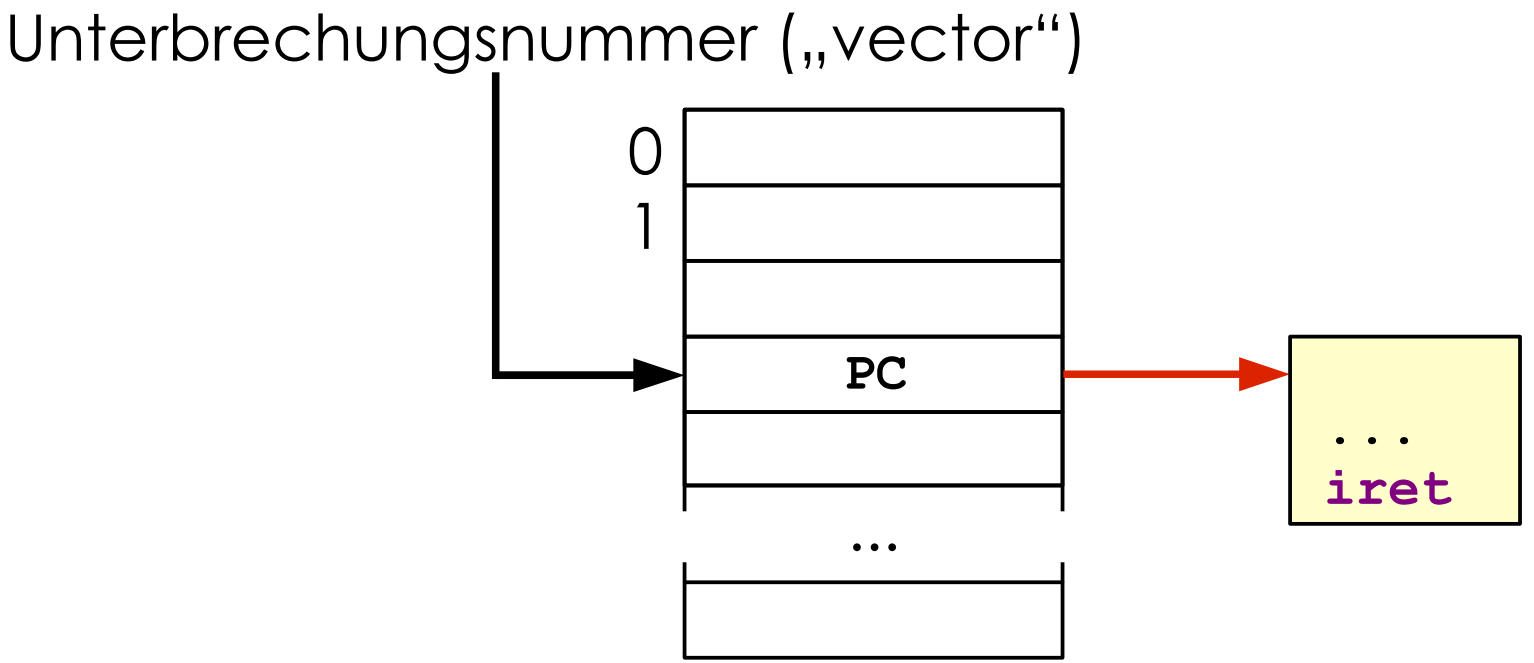
Fortsetzung per SW im BS-Kern

- IRET: restauriert Modus, User-Stack-Pointer, User-PC, User-Flags vom aktuellen Kern-Stack

Mehr zu Unterbrechungen (1)

Gesteuert durch eine Unterbrechungstabelle
(bei x86 „IDT“ - interrupt descriptor table)

- wird von der Unterbrechungshardware interpretiert
- ordnet jeder Unterbrechungsquelle eine Funktion zu



Mehr zu Unterbrechungen (2)

Erfordernisse für Unterbrechungen

- Unterbrechungsprioritäten
  - ➔ legen fest, welche Unterbrechung welche Unterbrechungsbehandlung unterbrechen darf
- Funktionen zum Sperren und Entsperren von Unterbrechungen

```
disable_interrupts_and-save-flags () {
  pushf //legt Inhalt von Flags auf dem Keller ab
  cli   //löscht Interrupt-Enabled-Flag in Flags
}

restore_flags () {
  popf  //restauriert altes Prozessorstatuswort (Flags)
}
```

Problem

Ein „unkooperativer“ Thread könnte immer noch die Umschaltung verhindern, indem er alle Unterbrechungen sperrt!

```
cli
while (true);
```

Lösung des Problems

- Unterscheidung zwischen Kern- und User-Modus
- Sperren von Unterbrechungen ist nur im Kern-Modus erlaubt
  + Annahme, dass BS sorgfältig programmiert wurde

RA: Privilegierungsstufen (CPU-Modi)

CPU-Modi

- Kern-Modus: alles ist erlaubt
- Nutzer-Modus: bestimmte Operationen werden unterbunden
  z. B. das Sperren von Unterbrechungen

Umschalten zwischen den Modi

- eine spezielle Instruktion löst eine Exception (Trap) aus
- ***ein*** fester Einsprungpunkt im Kern pro Interrupt/Exception-Vektor, dahinter Verteilung auf die verschiedenen Systemaufrufe
- Unterbrechung erzwingt diese Umschaltung
  - → manche CPU haben mehr als zwei Privilegstufen, z. B. IA32 hat 4 „Ringe“
- → zwei unterschiedlich privilegierte Modi sind notwendig

Preemptives (nicht kooperatives) Scheduling

Hardware-seitig

- Umschaltung in Kern
- rette PC, Flags, ... auf den Kern-Stack des unterbrochenen Threads

`iret`: Wiederherstellen von Modus, PC, Flags, ...

- (springt zurück in User)

im Kern

```
interrupt_handler()
{
 if (io_pending) {
    send_message (io_thread);
 // thread ändert Zustand von
 // „aktiv“ nach „bereit“
    switch_to (io_thread);
 // Ausführung wird hier
 // fortgesetzt, wenn auf diesen
 // Thread zurückgeschaltet wird
 }
 schedule;
 iret    // back to user mode
}
```

Wegweiser

Zusammenspiel/Kommunikation mehrerer Threads

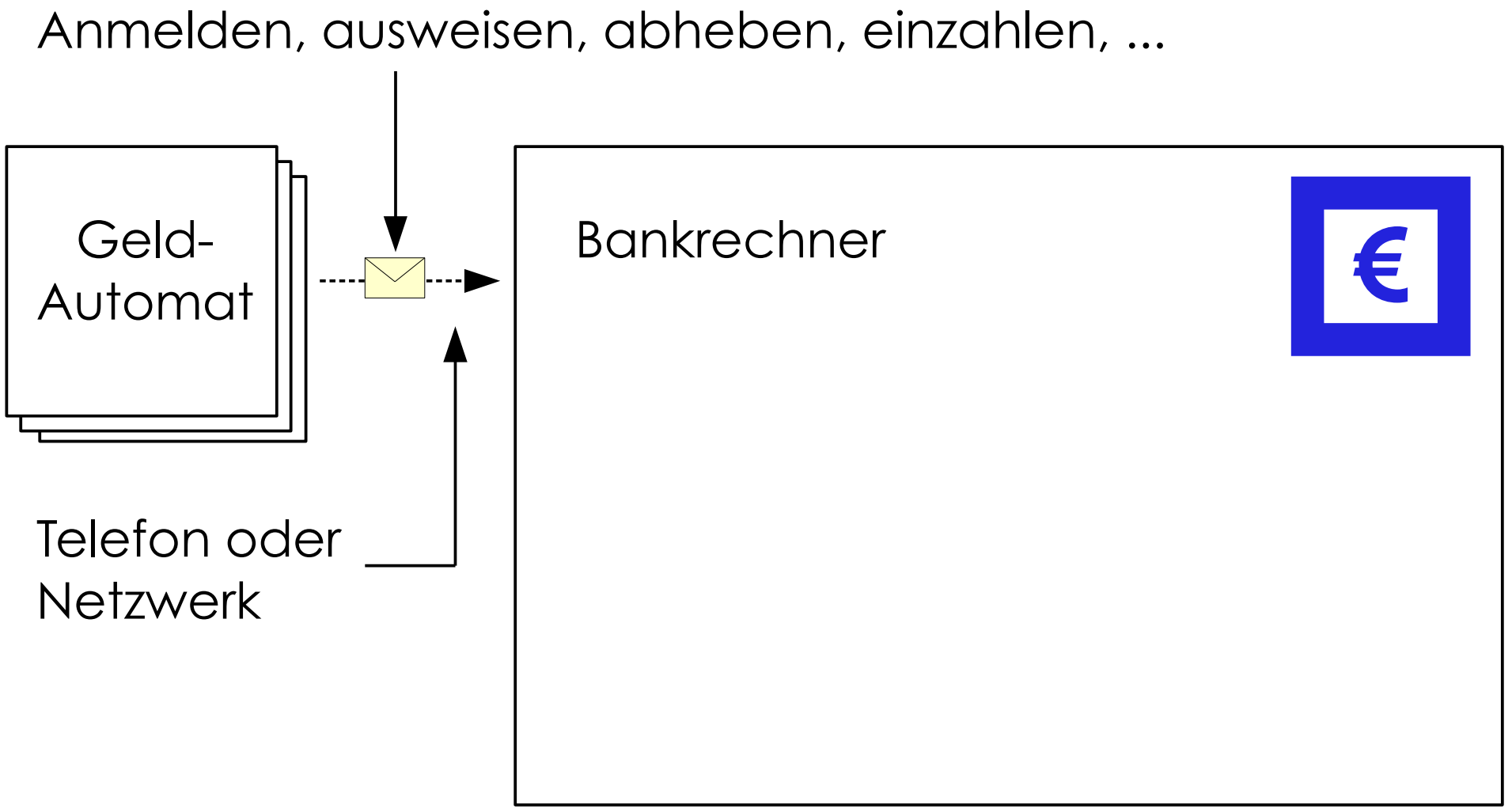
Nutzung gemeinsamen Speichers
- race conditions
- ➔ Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluß und dessen Durchsetzung
- mit/ohne HW-Unterstützung
- im Ein-/Mehrprozessor-Fall
- mit/ohne busy waiting

Grundsatz

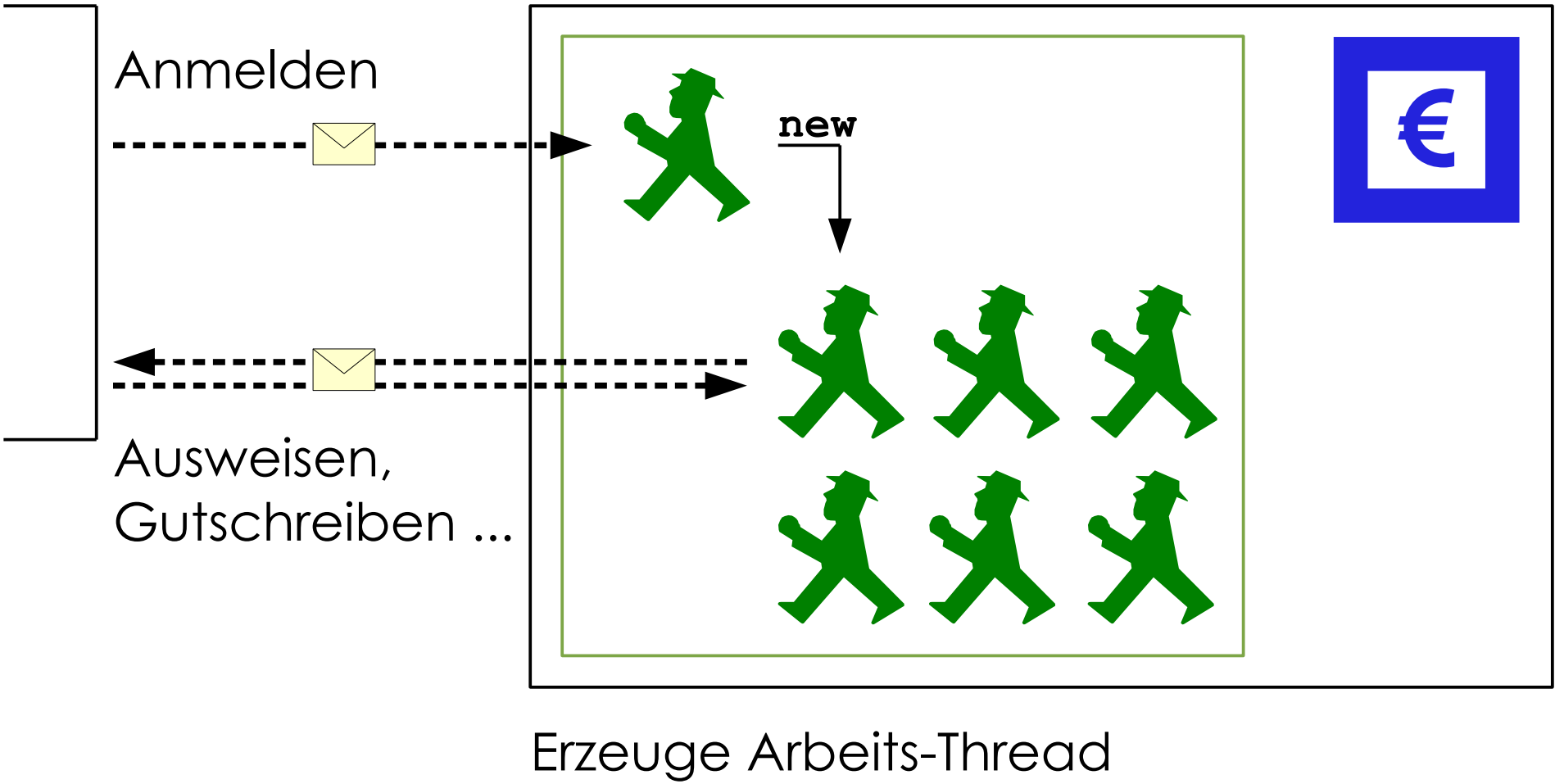
Beim Einsatz paralleler Threads dürfen keine Annahmen über die relative Ablaufgeschwindigkeit von Threads gemacht werden.

Beispiel Geldautomat

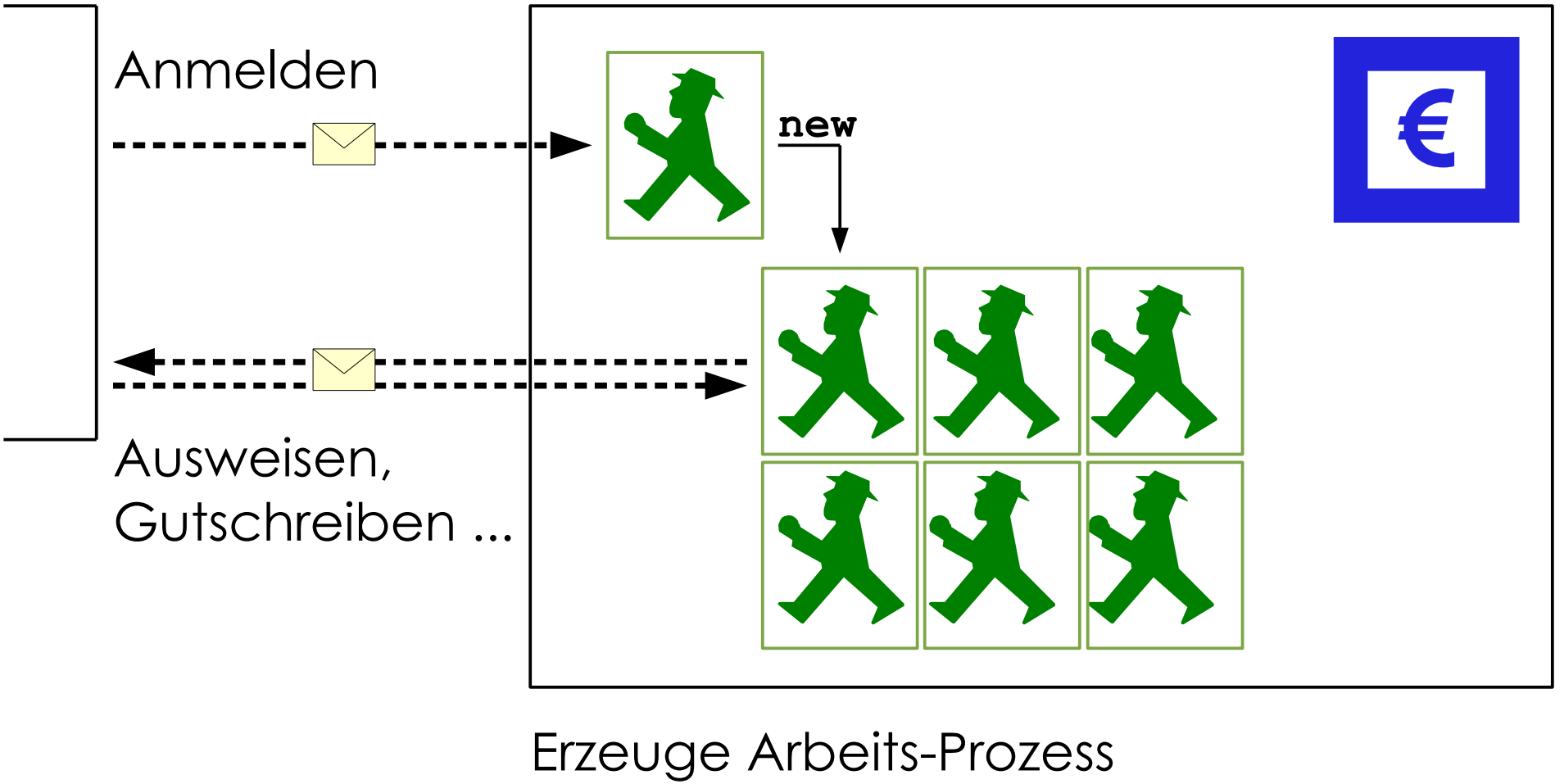


(grob nach Nichols, Buttlar, Farell)

Thread-Struktur



Oder mit Prozessen



Wegweiser

Zusammenspiel/Kommunikation mehrerer Threads

Nutzung gemeinsamen Speichers
- race conditions
- ➔ Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluß und dessen Durchsetzung
- mit/ohne HW-Unterstützung
- im Ein-/Mehrprozessor-Fall
- mit/ohne busy waiting

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

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

bool abbuchen(int Betrag) {

 if (Betrag <= Kontostand){

  Kontostand -= Betrag;

  return true;

 } else return false;
}
```

Maschinensprache

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

Variante 3 – Die kundenfreundliche Bank

T1 : abbuchen(1);

```
load    R, Kontostand
```

```
sub     R, Betrag
store   R, Kontostand
```

T2 : abbuchen(20);

```
load    R, Kontostand
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

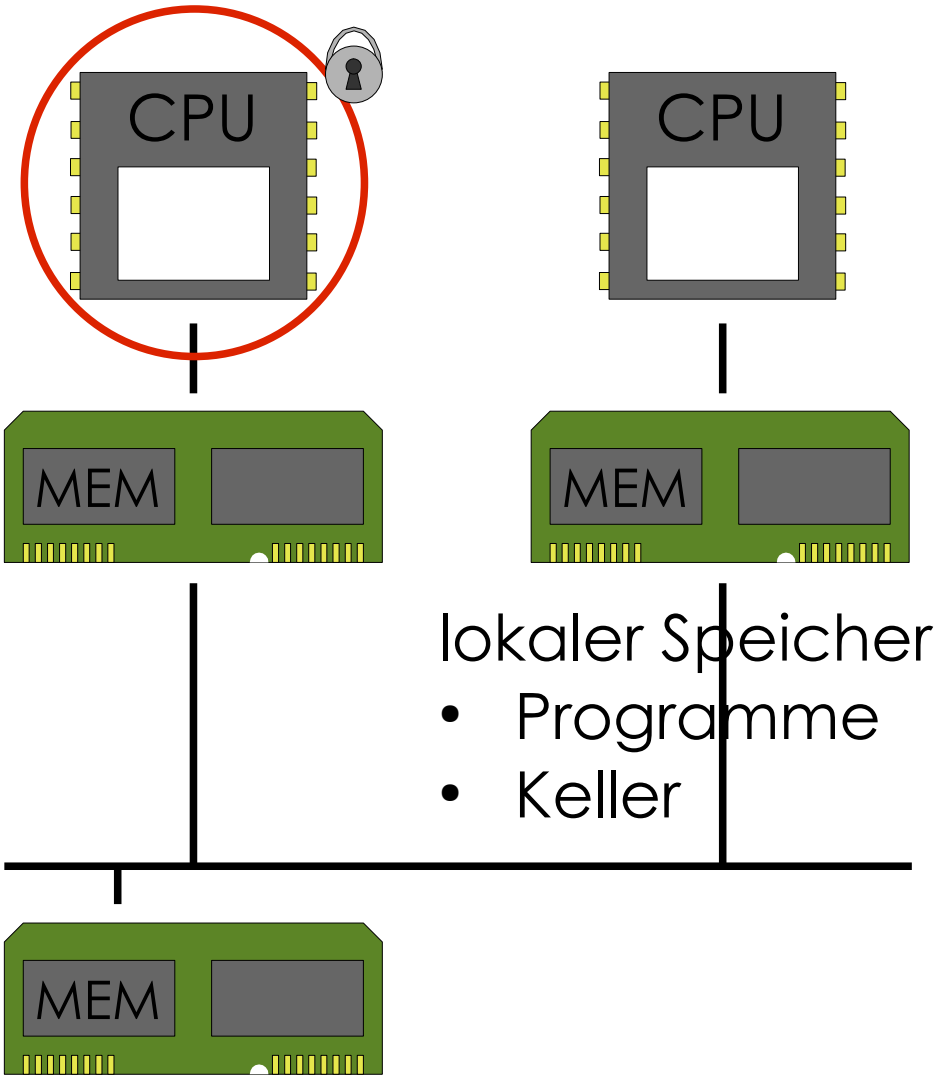
- mit HW-Unterstützung

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!

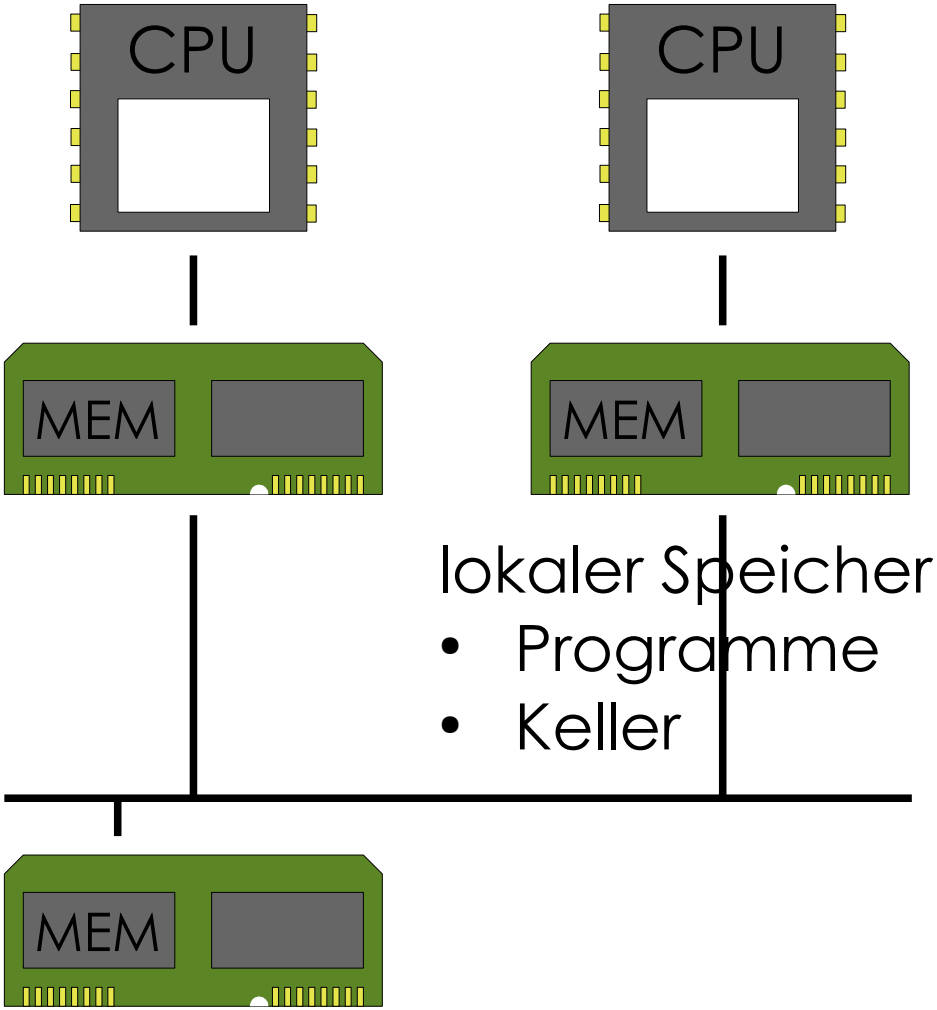


lokaler Speicher
- Programme
- Keller

gemeinsamer Speicher
- **Kontostand**

Lösungsversuch 2 – KA mit einer Instruktion

```
T1: ...

   ld     R, Betrag

   sub    Kontostand, R
```



lokaler Speicher
- Programme
- Keller

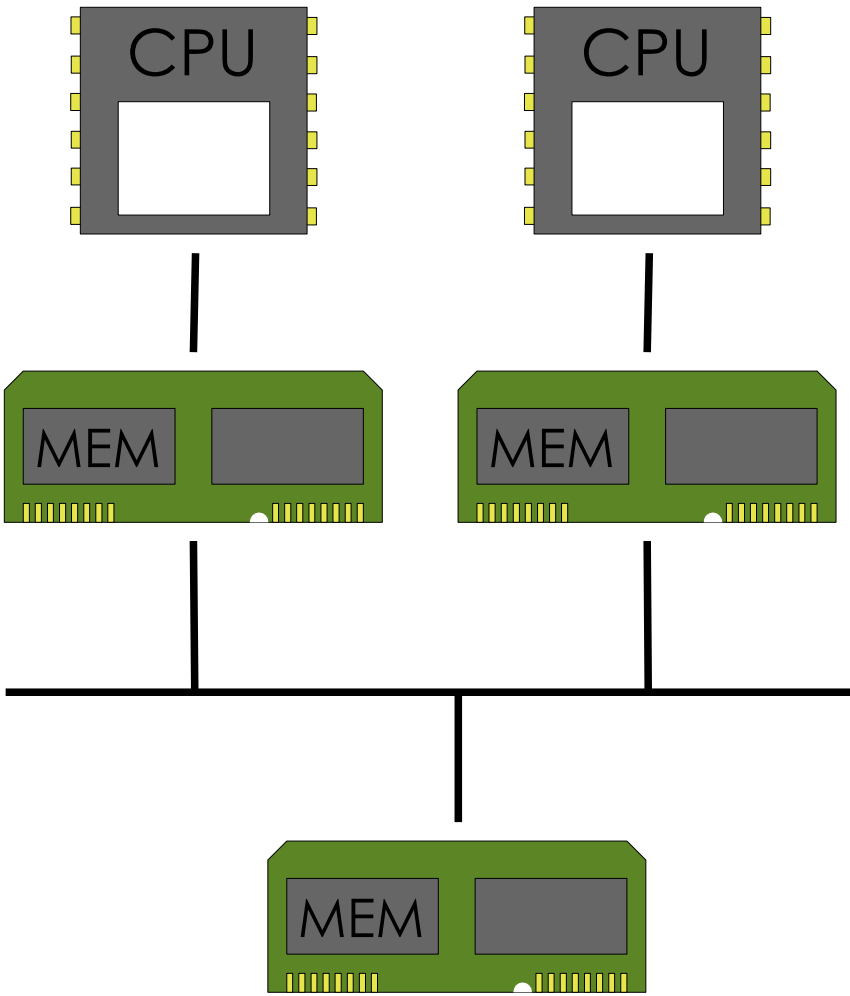
sub ist nicht unterbrechbar

aber: funktioniert im MP-Fall auch nicht!

Begründung: folgende Folien

gemeinsamer Speicher
- **Kontostand**

RA: Mehrprozessormaschine



CPUs

lokaler Speicher
- Programme
- Keller

Bus mit Arbiter

gemeinsamer Speicher

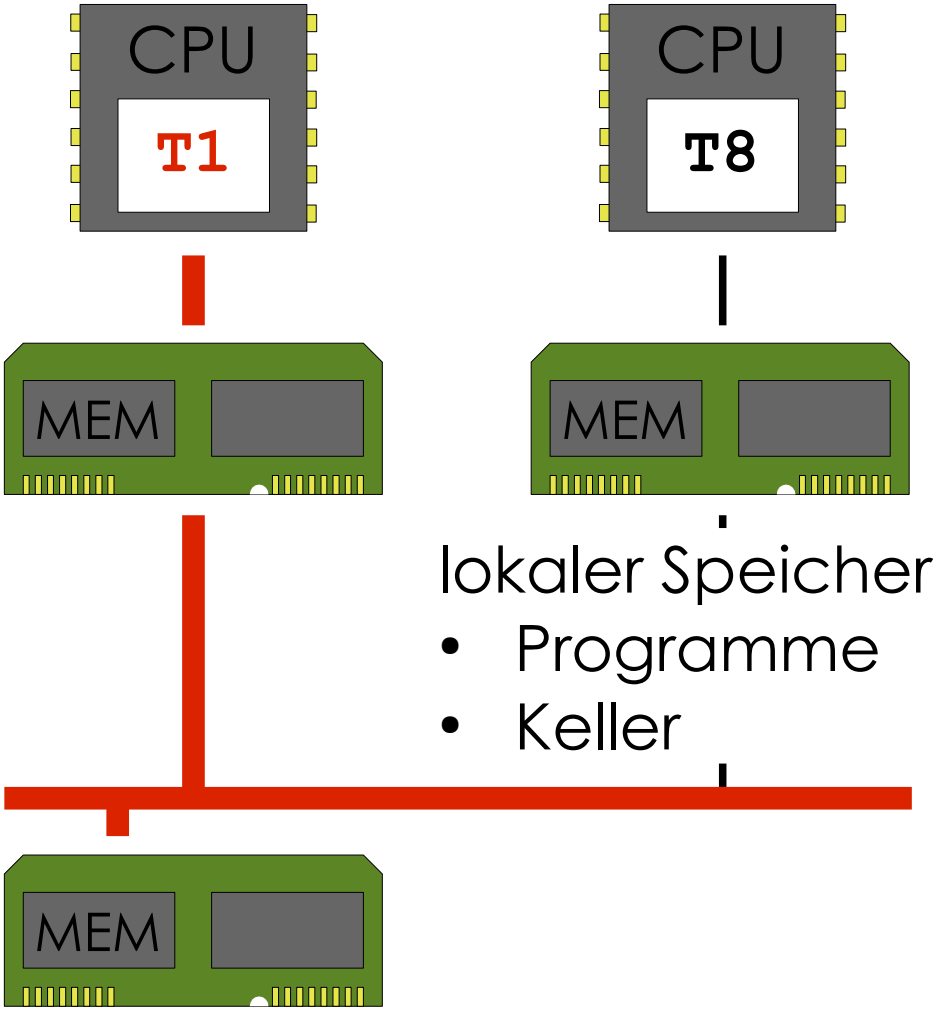
Lösungsversuch 2 – KA mit einer Instruktion

```
T1: ...

   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 !

→ ...



lokaler Speicher
- Programme
- Keller

gemeinsamer Speicher
- **`Kontostand`**

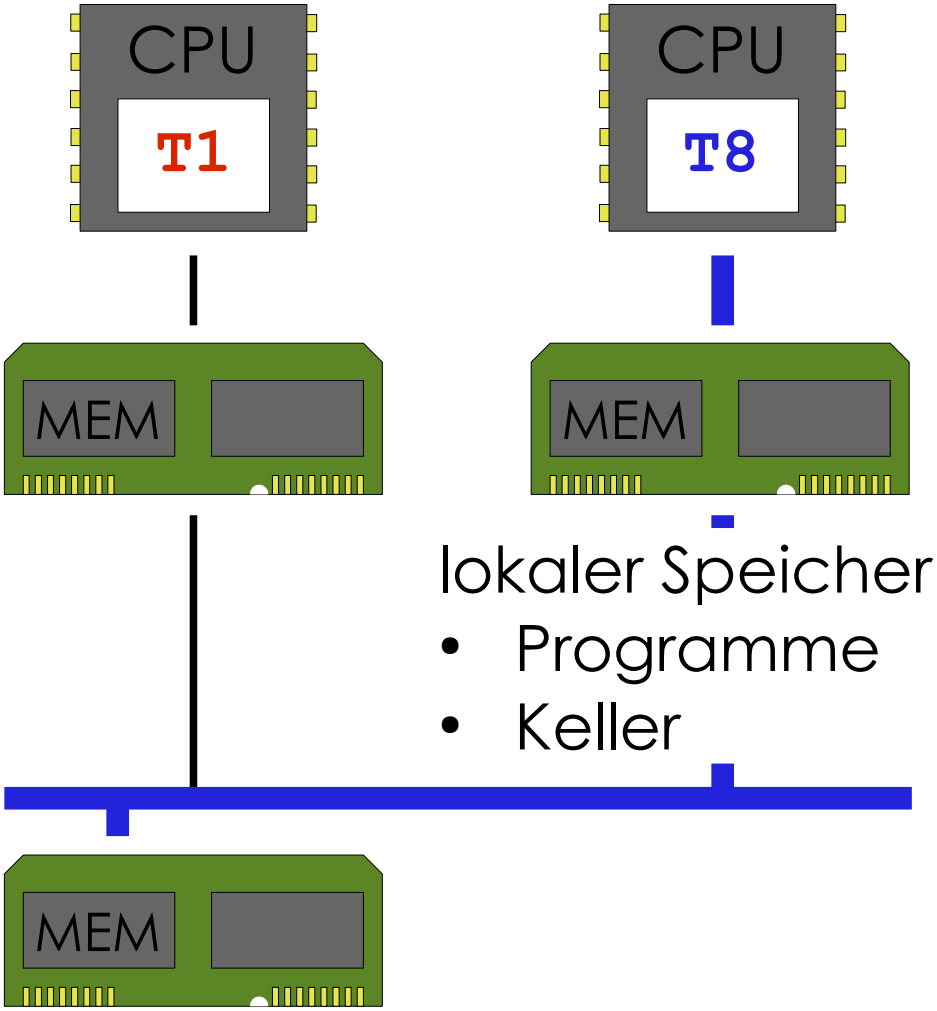
Lösungsversuch 2 – KA mit einer Instruktion

```
T1: ...

   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 !



lokaler Speicher
- Programme
- Keller

gemeinsamer Speicher
- **`Kontostand`**

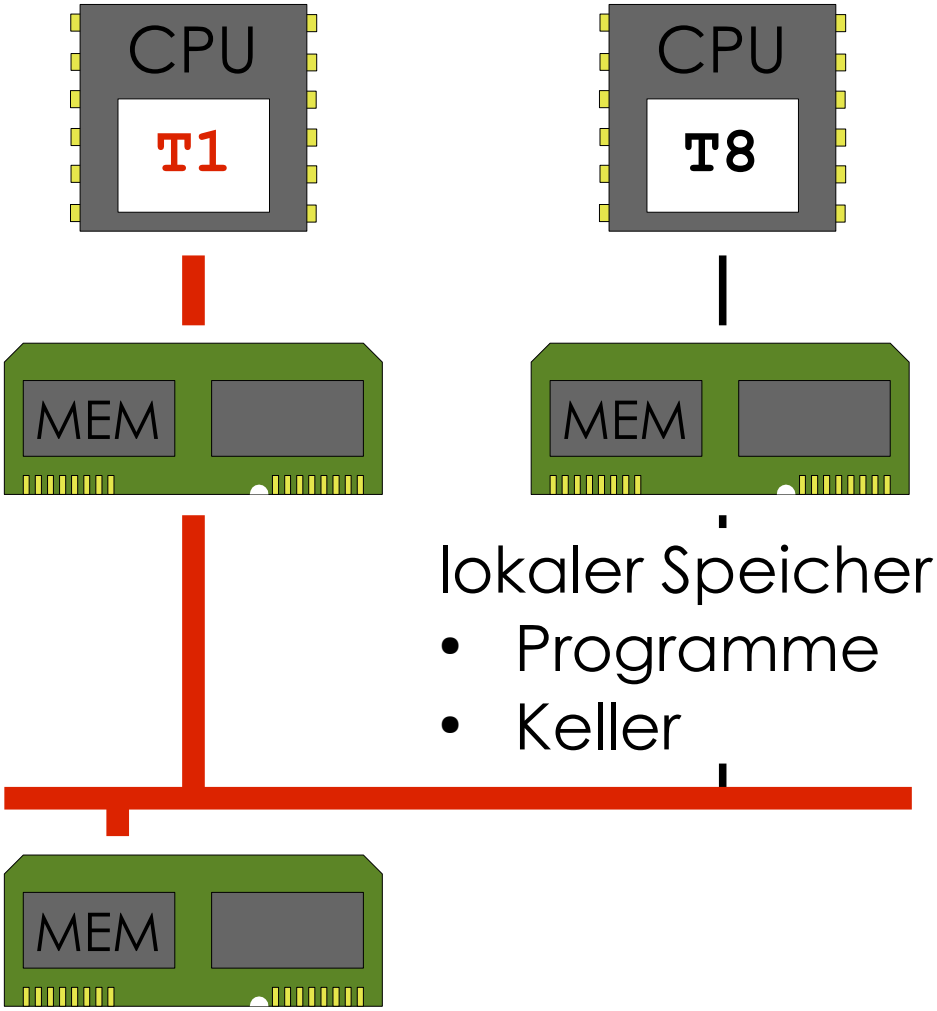
Lösungsversuch 2 – KA mit einer Instruktion

```
T1: ...

   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 !



lokaler Speicher
- Programme
- Keller

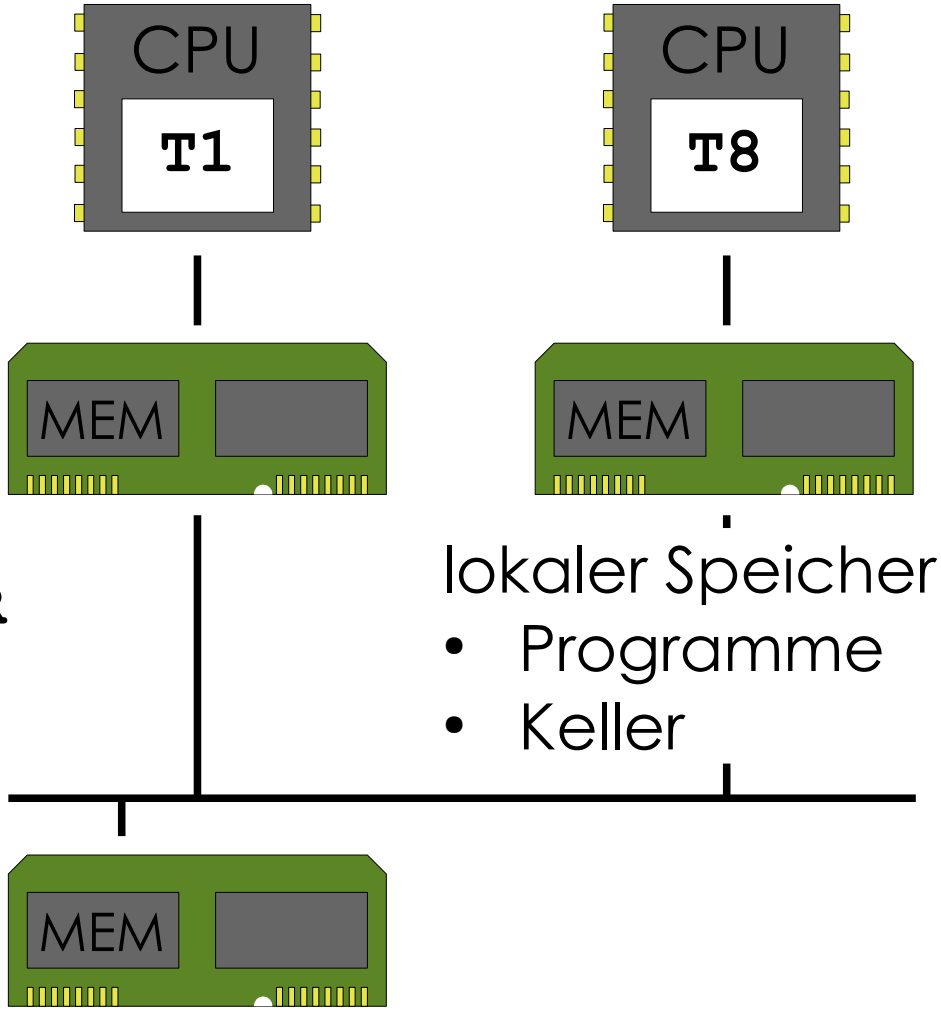
gemeinsamer Speicher
- **`Kontostand`**

Lösungsversuch 3 – atomare Instruktion

```
T1: ...

 ld             R, Betrag

 atomic_sub     Kontostand, R
```

- Die Instruktion **`atomic_sub Kontostand, R`** ist nicht unterbrechbar.
- Der Bus bleibt zwischen den Bus-Zyklen für andere CPUs gesperrt.

→ funktioniert.



Kritischer Abschnitt (1)

```
int Kontostand;

bool abbuchen(int Betrag) {

  entersection();

  if (Betrag <= Kontostand) {

     Kontostand -= Betrag;

     leavesection();
     return true;

  } else {

        leavesection();
        return false;
  }
}
```

Kritischer Abschnitt (2)

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

```
entersection:
  pushf
  cli       //disable irqs
  ret


leavesection:
  popf      //restore
  ret
```

Implementierung mittels Sperrvariable

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

void entersection() {

  while (Lock != 0);
  //busy waiting

  Lock = 1;
}

void leavesection() {

  Lock = 0;
}
```

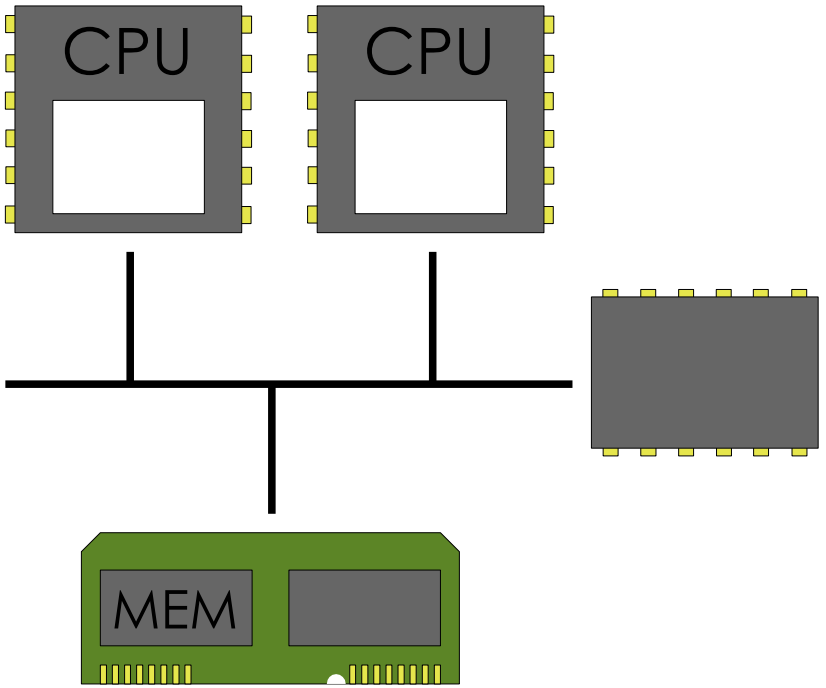
Implementierung mittels Sperrvariable

- **funktioniert nicht!**
- warum?
- ➔ Frage: welche unteilbare Einheit stellt die HW zur Verfügung?

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

void entersection() {

  while (Lock != 0);
  //busy waiting

  Lock = 1;
}

void leavesection() {

  Lock = 0;
}
```

RA: unteilbare Operationen der HW

Mehrprozessormaschine



CPUs

Bus mit Arbiter

gemeinsamer Speicher

Der Arbiter sequentialisiert die Buszugriffe:

- 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

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

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


leavesection:
  mov      lock, #0
  ret
```

- Entersection gleich mit Lock variable erläutern, d.h. Es gibt belibig viele kritische Abschnitte in einem System

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)

```
pid_t owner;
lock_t lock = 0;

void entersection() {

  while(test_and_set(lock)){

   Kern.Switch_to (owner);

  }

  owner = AT;
  //aktueller Thread
}

void leavesection() {

  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 entersection() {

  while(test_and_set(lock)){

   Kern.Switch_to (owner);

  }

  owner = AT;
  //aktueller Thread
}

void leavesection() {

  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 entersection(
    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

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

Wegweiser

Zusammenspiel/Kommunikation mehrerer Threads

Nutzung gemeinsamen Speichers
- race conditions
- ➔ Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluß und dessen Durchsetzung
- ohne HW-Unterstützung

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){

}
```

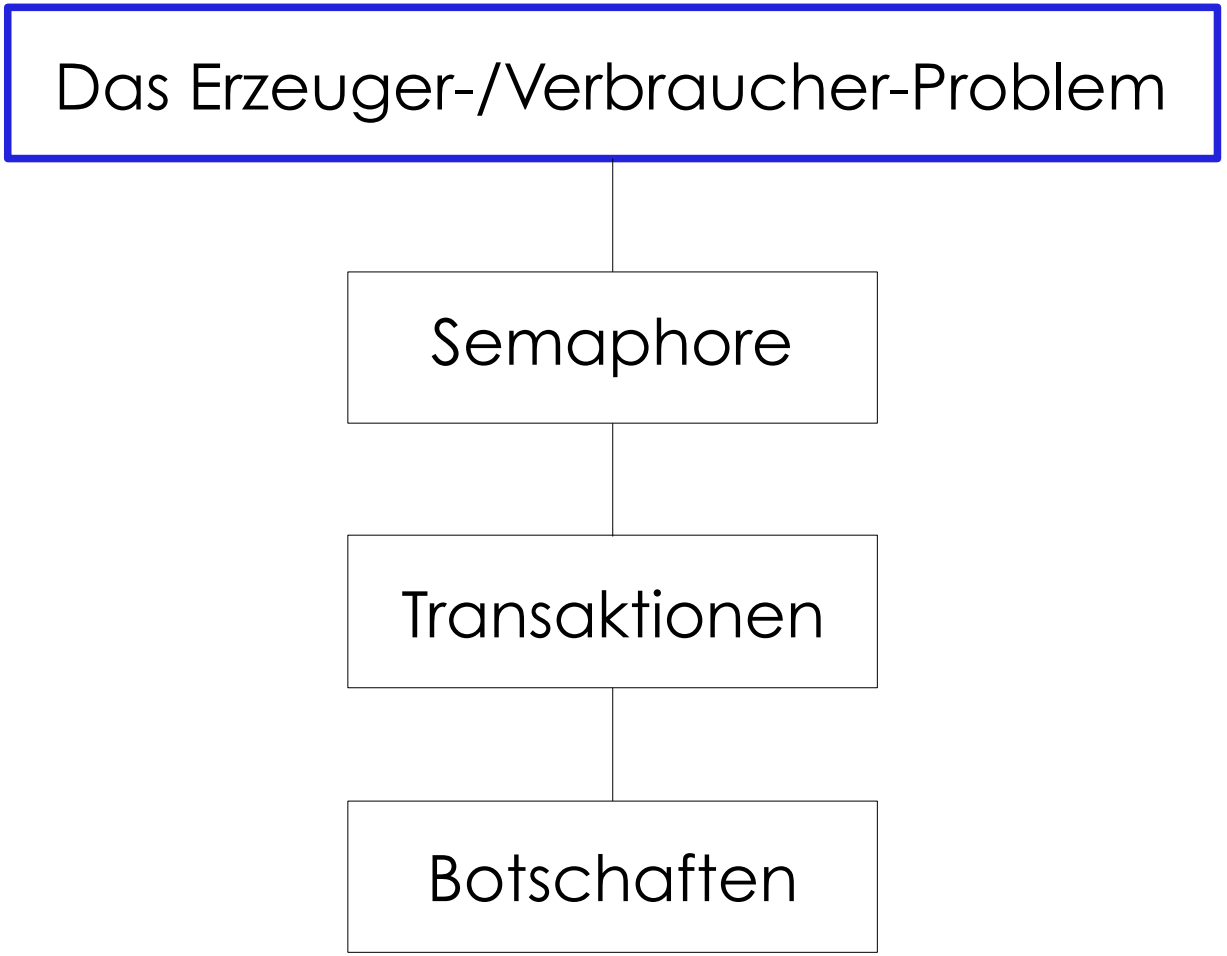
Lösung nach Peterson

```
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

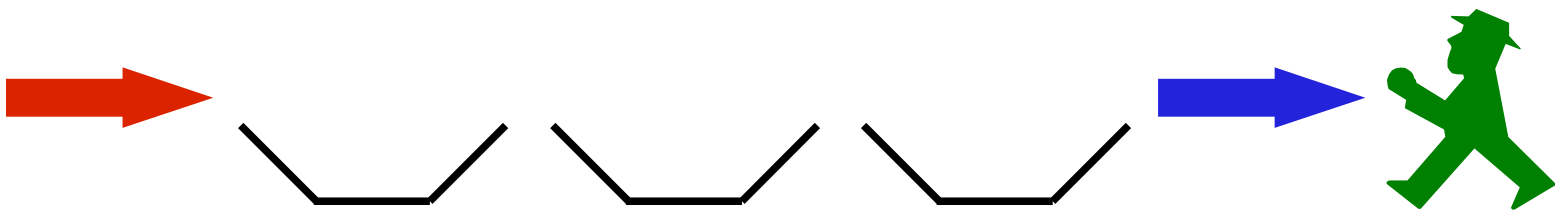
Erzeuger-/Verbraucher-Probleme

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

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


    enter_item(item);


  }
}
```

Verbraucher

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


    item = remove_item();


    consume_item(item);
  }
}
```

Ein Implementierungsversuch

Erzeuger

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

  while(true) {

     item = produce_item();




     enter_item(item);
     count++;



  }
}
```

Verbraucher

```
void consume() {

  while(true) {




     item = remove_item();
     count--;



     consume_item(item);
  }
}
```

Ein Implementierungsversuch

Erzeuger

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

  while(true) {

     item = produce_item();




     enter_item(item);
     count++;

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

Verbraucher

```
void consume() {

  while(true) {

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

     item = remove_item();
     count--;




     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);
  }
}
```

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

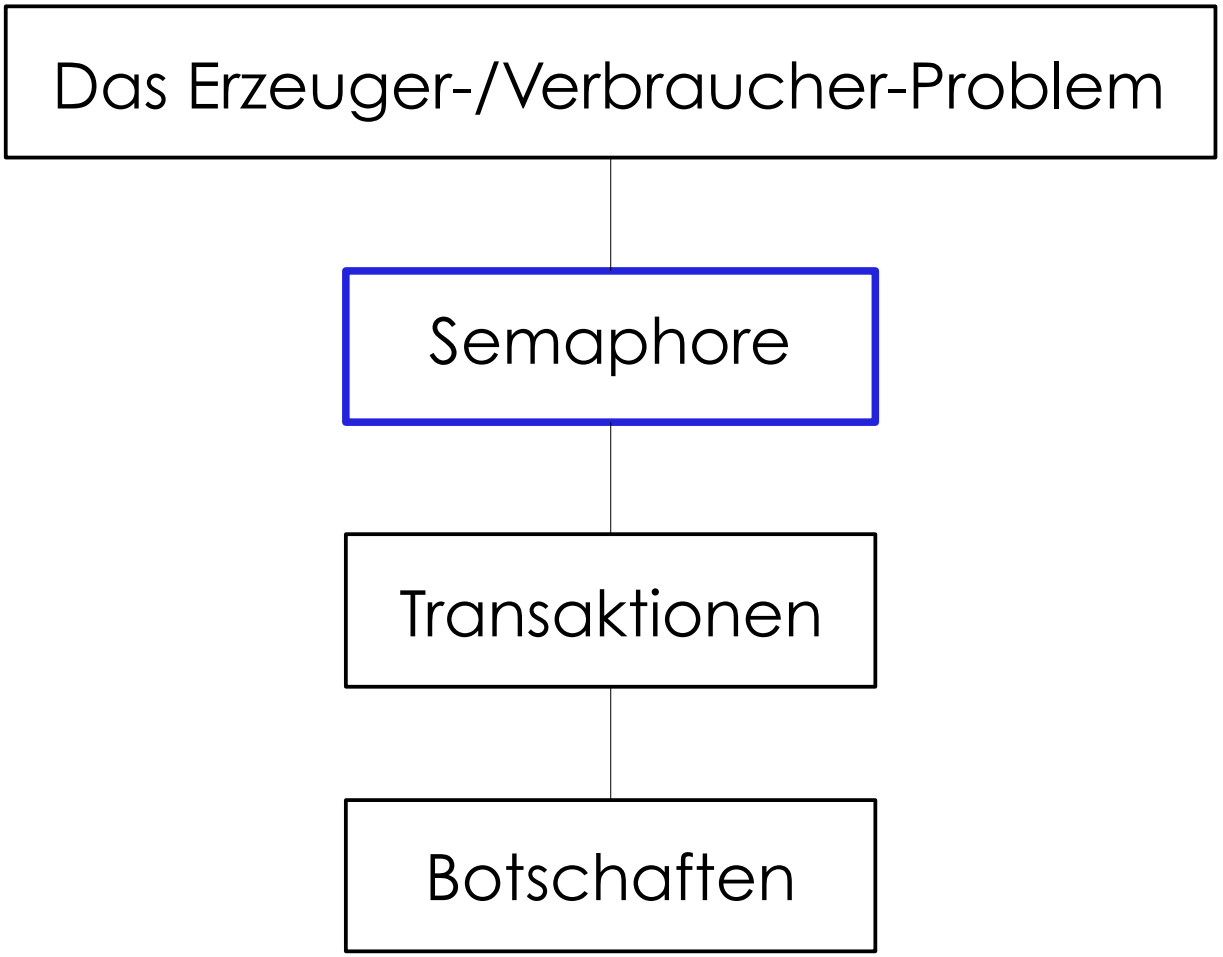
```
int count = 0;

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 P();
   //passeren = betreten

   void V();
   //verlaten = verlassen

}
```

Kritischer Abschnitt mit Semaphoren

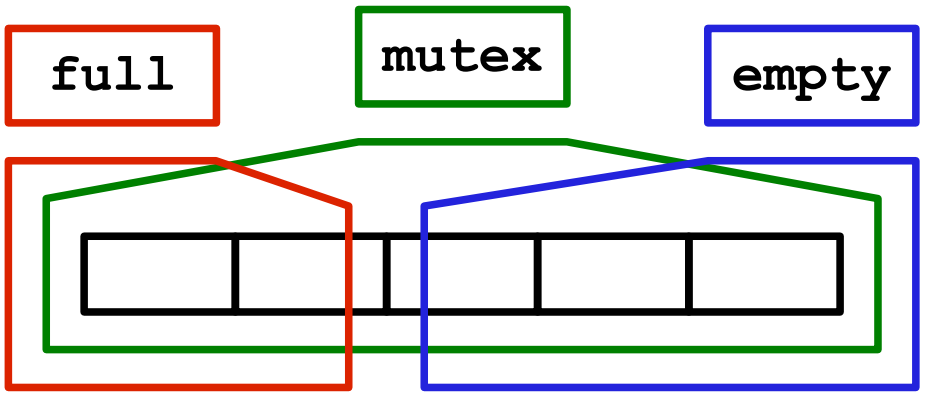
Wechselseitiger Ausschluss

```
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

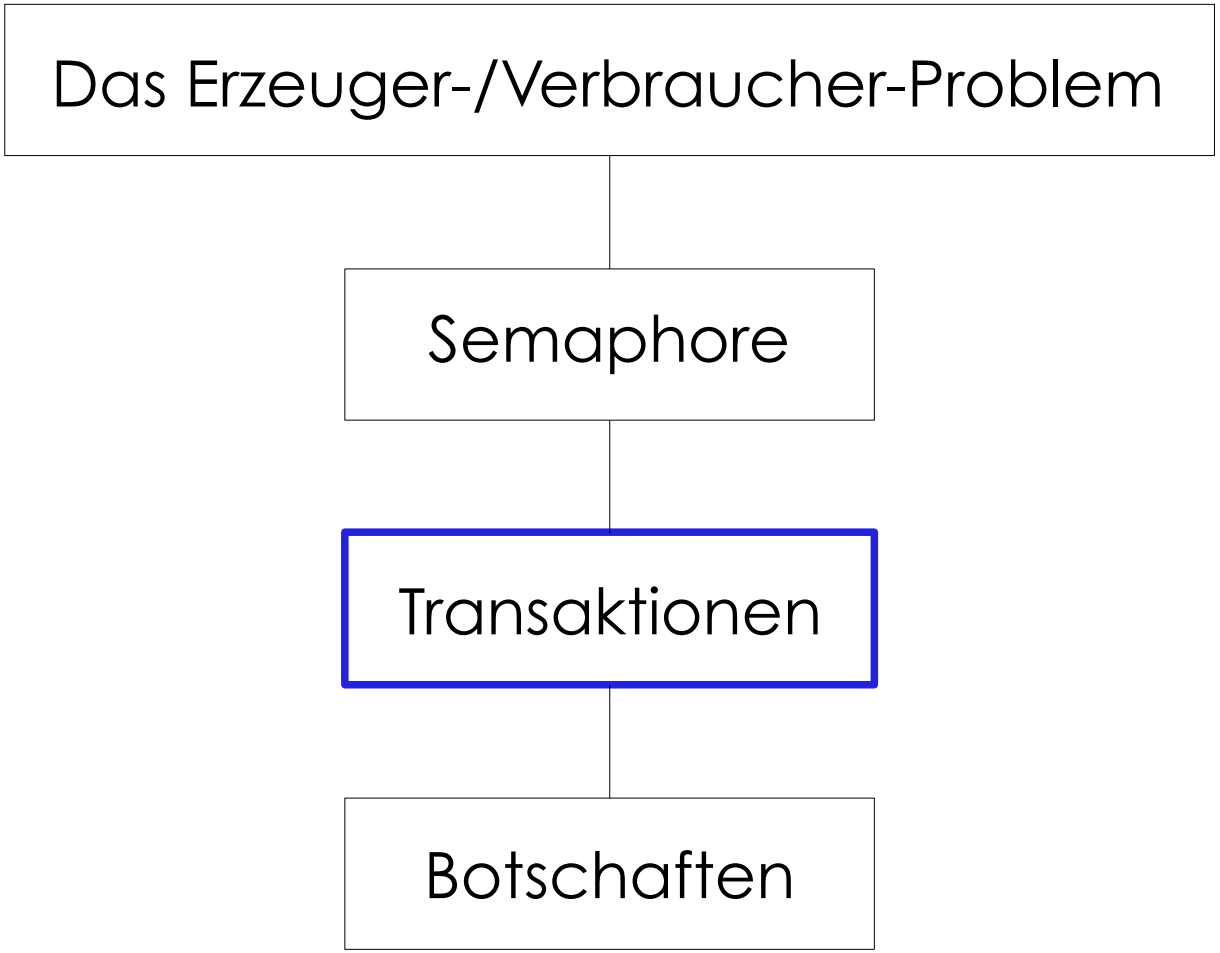
- beteiligte Objekte vor dem Zugriff sperren
- erst nach Abschluss der Gesamtoperation entsperren

```
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


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

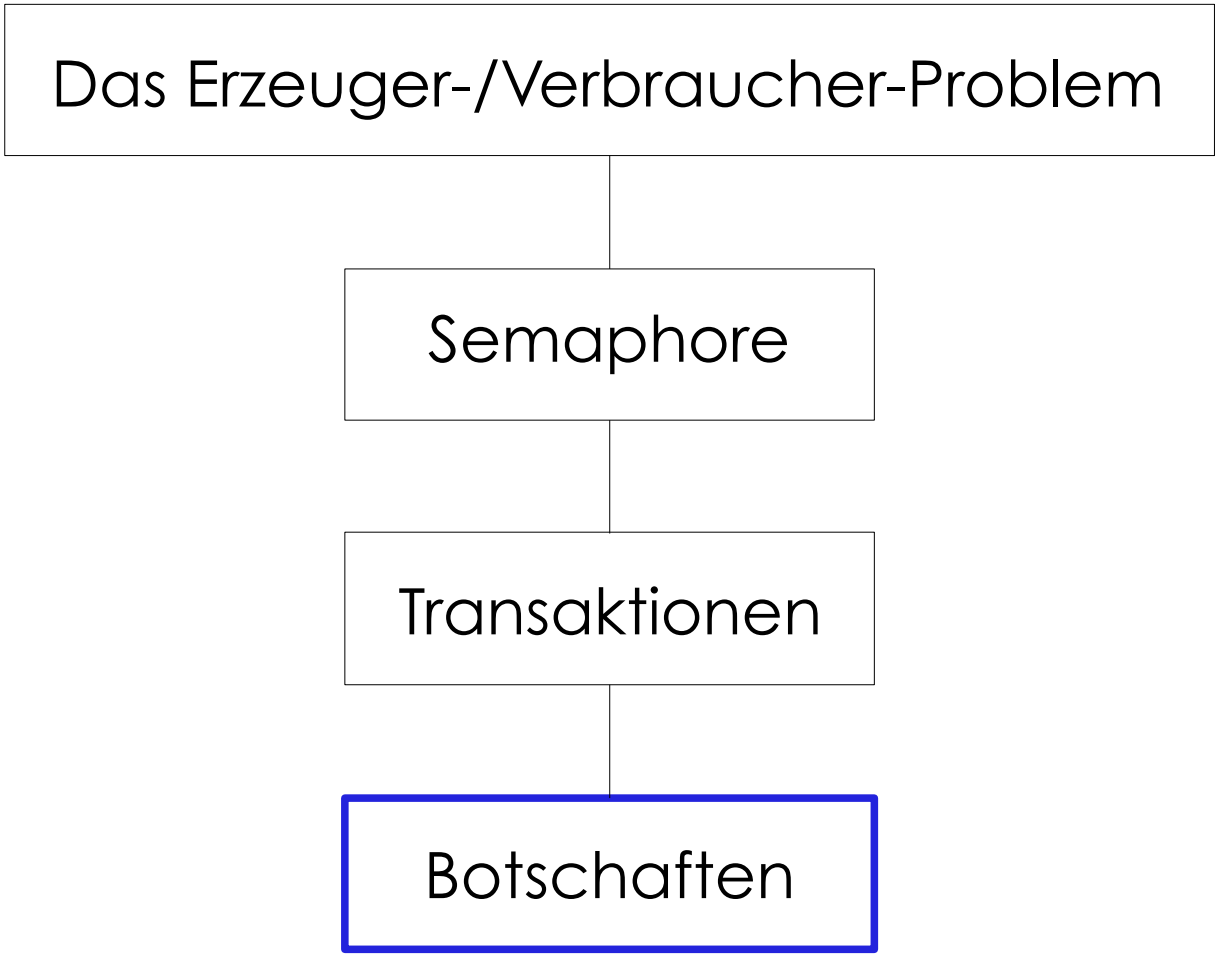
Nachteile von Semaphoren etc.

- ➔ 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. Cray T3D

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“

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);

     enter_item(item);

     send(item);
  }
}
```

Verbraucher

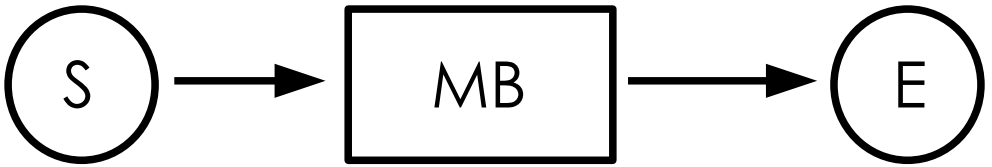
```
void consumer() {

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

 while (true) {

   receive(item);

   item = remove_item();

   send(empty);

   consume_item(item);

 }
}
```

Botschaften

Varianten des Grundkonzepts

- Adressat:

  Prozess, Thread oder Briefkasten (mail box)



- Pufferung/Synchronität:

  mit/ohne

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

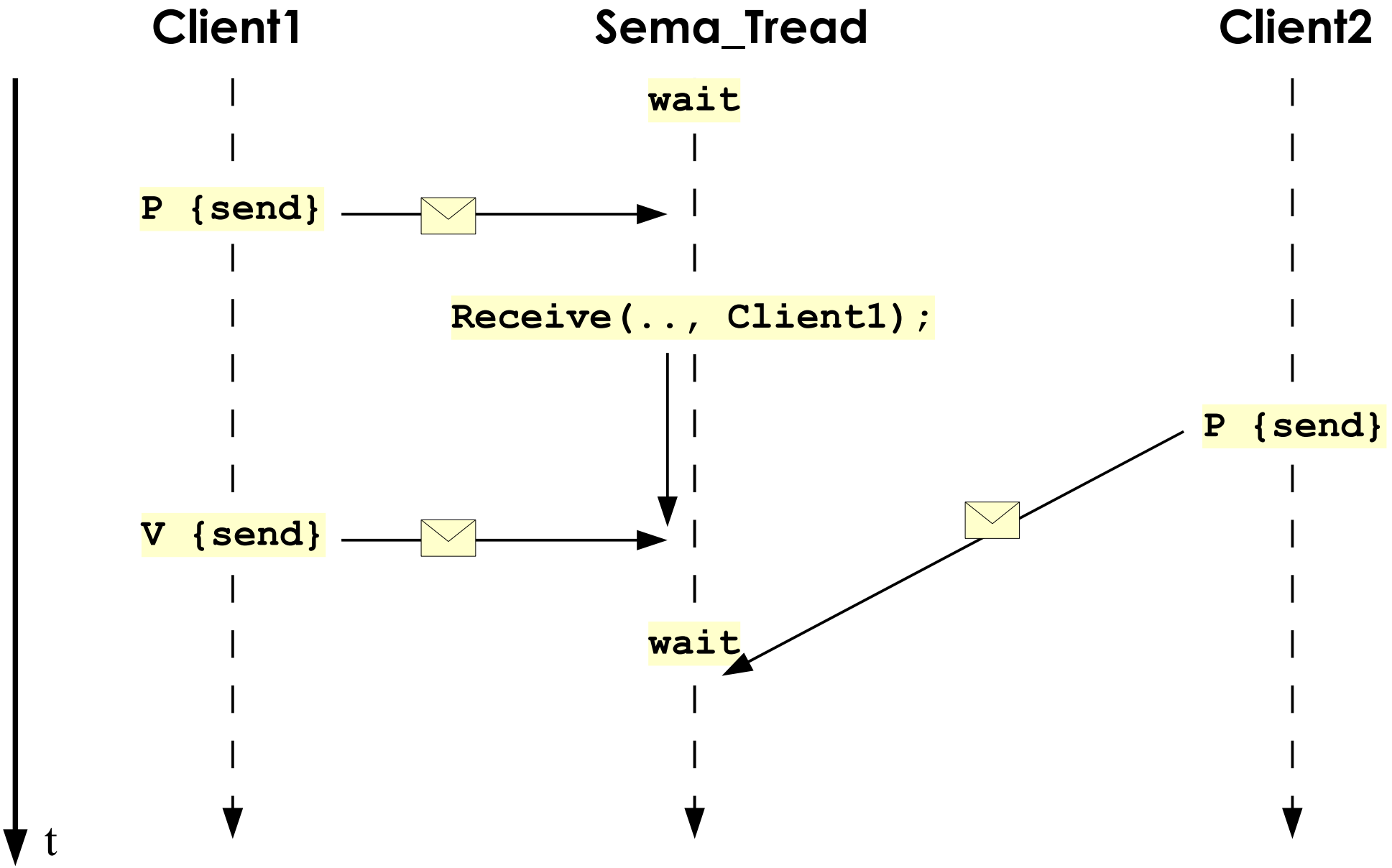
Semaphore-Thread

```
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

Wegweiser

Zusammenspiel (Kommunikation) mehrerer Threads

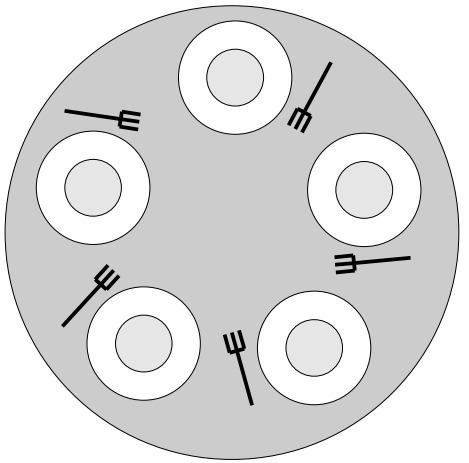
Einige typische Probleme

- Erzeuger / Verbraucher
- 5 Philosophen
- Leser/Schreiber

Das Zuschneiden von Threadsystemen
→ Verschiedene Schedulingverfahren

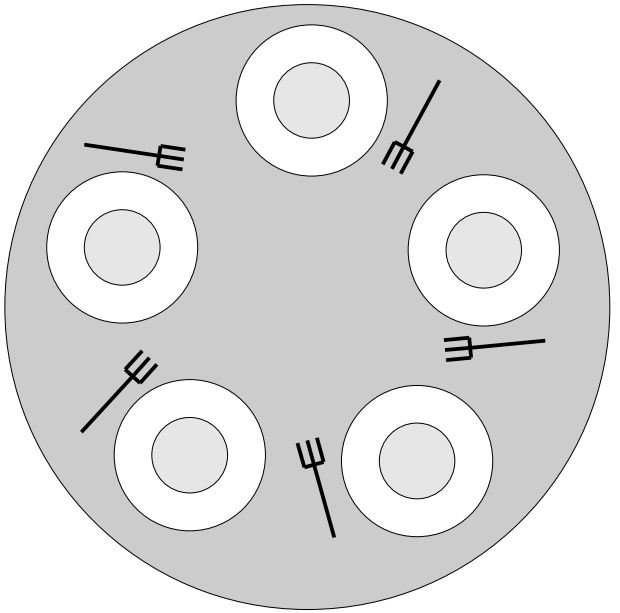
Dining Philosophers

- 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



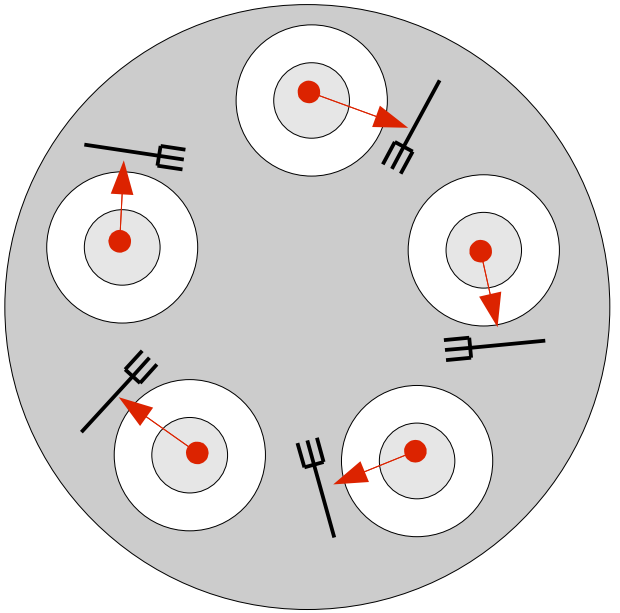
Die offensichtliche ( ) Lösung

```
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.P();

  //read

  DB.V();
}
```

```
void Writer() {

  DB.P();

  //write

  DB.V();

}
```

Leser/Schreiber mit Semaphoren

```
void Reader() {


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


  //read


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

}
```

```
void Writer() {




  DB.P();

  //write

  DB.V();


}
```

Leser/Schreiber mit Semaphoren

```
void Reader() {

  Mutex.P();
  Rcount++;
  if (Rcount == 1) DB.P();
  Mutex.V();

  //read

  Mutex.P();
  Rcount--;
  if (Rcount == 0) DB.V();
  Mutex.V();
}
```

```
void Writer() {




  DB.P();

  //write

  DB.V();


}
```

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

Scheduling

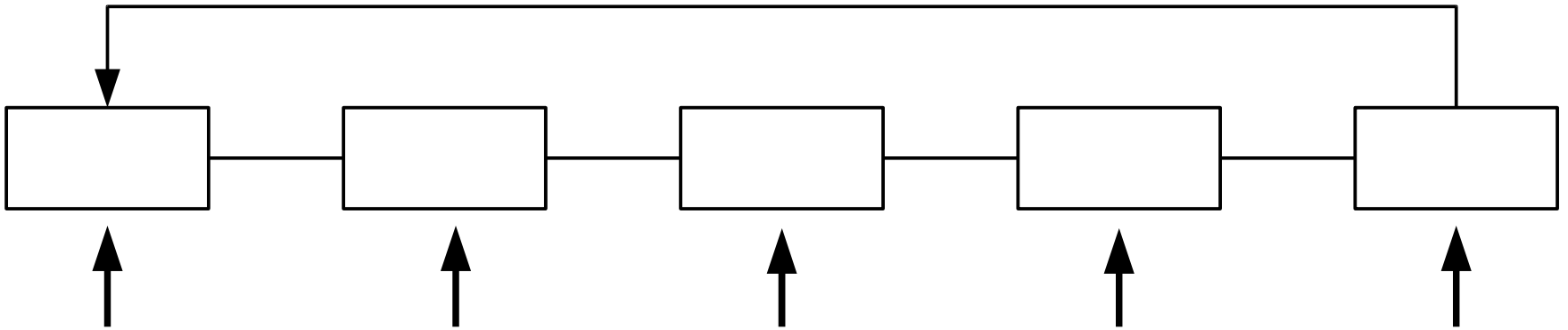
Kriterien

- 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

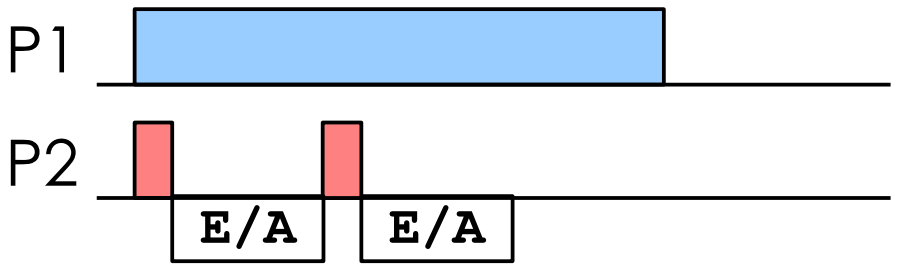
Beispiel für Heuristik

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

➔ bessere Auslastung von E/A-Geräten

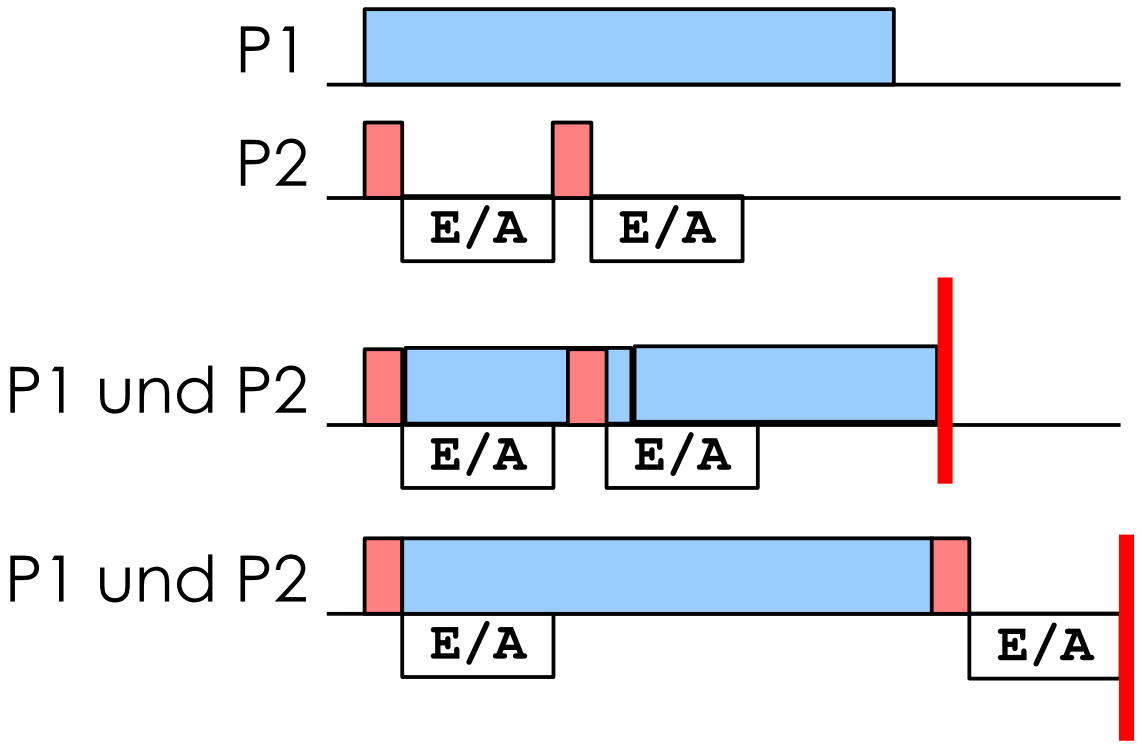
Beispiel

- P1 || P2
- P1 lang CPU
- P2 kurz CPU, langsame EA



Beispiel

- P1 || P2
- P1 lang CPU
- P2 kurz CPU, langsame EA



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

Ratenmonotones Scheduling

- Aufgabenstellung
  - ➔ Deadlines mittels statischer Prioritäten

- RMS-Technik

  kürzeste Periode → höchste Priorität

- nicht immer vollständige Auslastung der CPU möglich

- ➔ Es gibt kein besseres Verfahren, Threads statisch Prioritäten zu geben – optimal bzgl. Einplanbarkeit.

Zeitschranken-Scheduling

EDF - Earliest Deadline First

- Für jeden Thread ist dessen Deadline dem Scheduler bekannt.
- Der Scheduler wählt immer den Thread mit nächster Deadline.
- Im Einprozessorfall volle Auslastung der CPU möglich.

➔ Das Verfahren ist optimal bzgl. Einplanbarkeit bei dynamischen Prioritäten.

Zwei-Ebenen-Scheduling

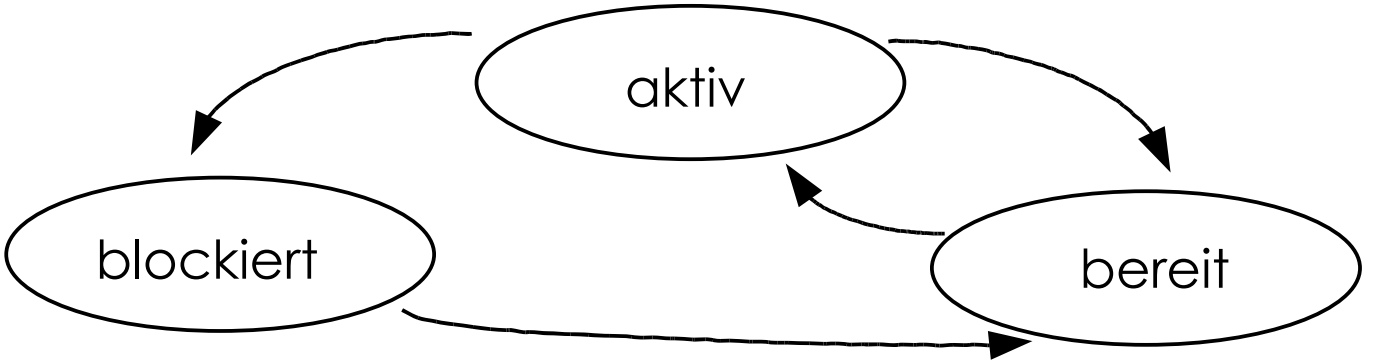
Ausgangspunkt

- 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.

Zwei-Ebenen-Scheduling

Ausgangspunkt

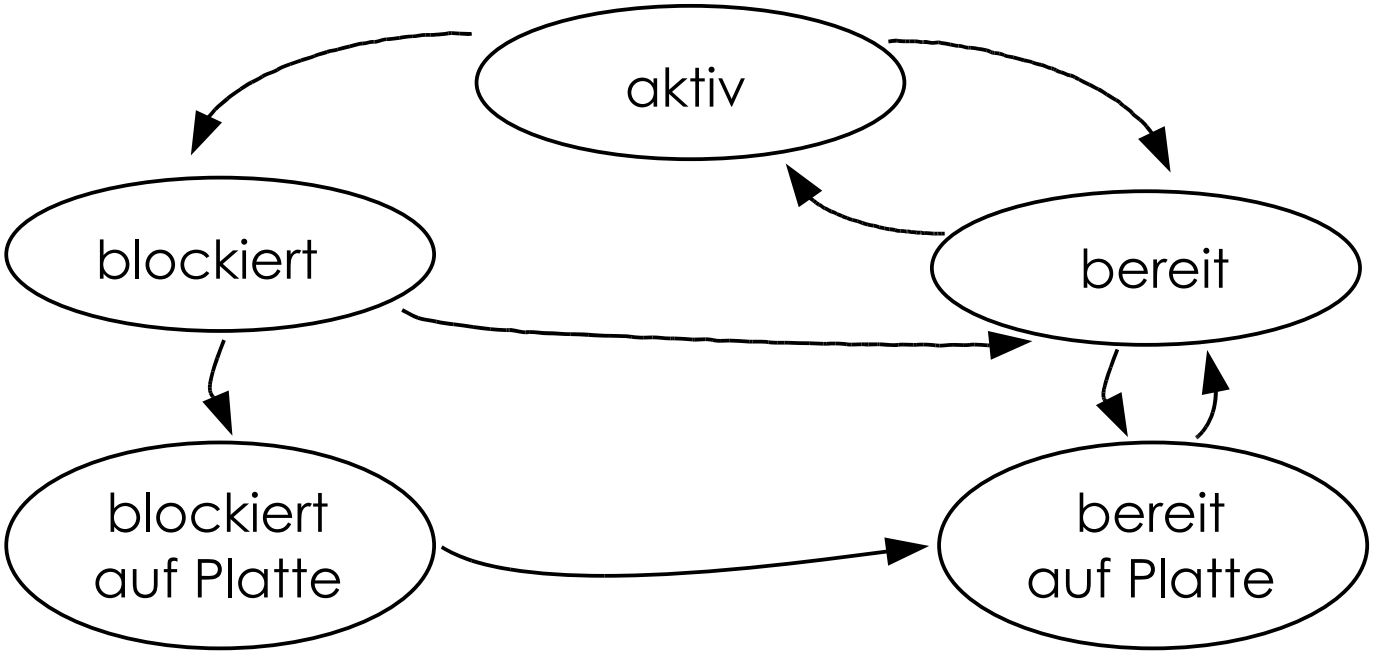
- Prozesse benötigen nicht nur CPU, sondern auch Speicher, manchmal müssen Prozesse ganz aus Hauptspeicher verdrängt werden.
- Nicht nur die Zuteilung von CPU zu Thread, sondern auch die Verdrängung auf Platte wird Gegenstand des Schedulings.



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)

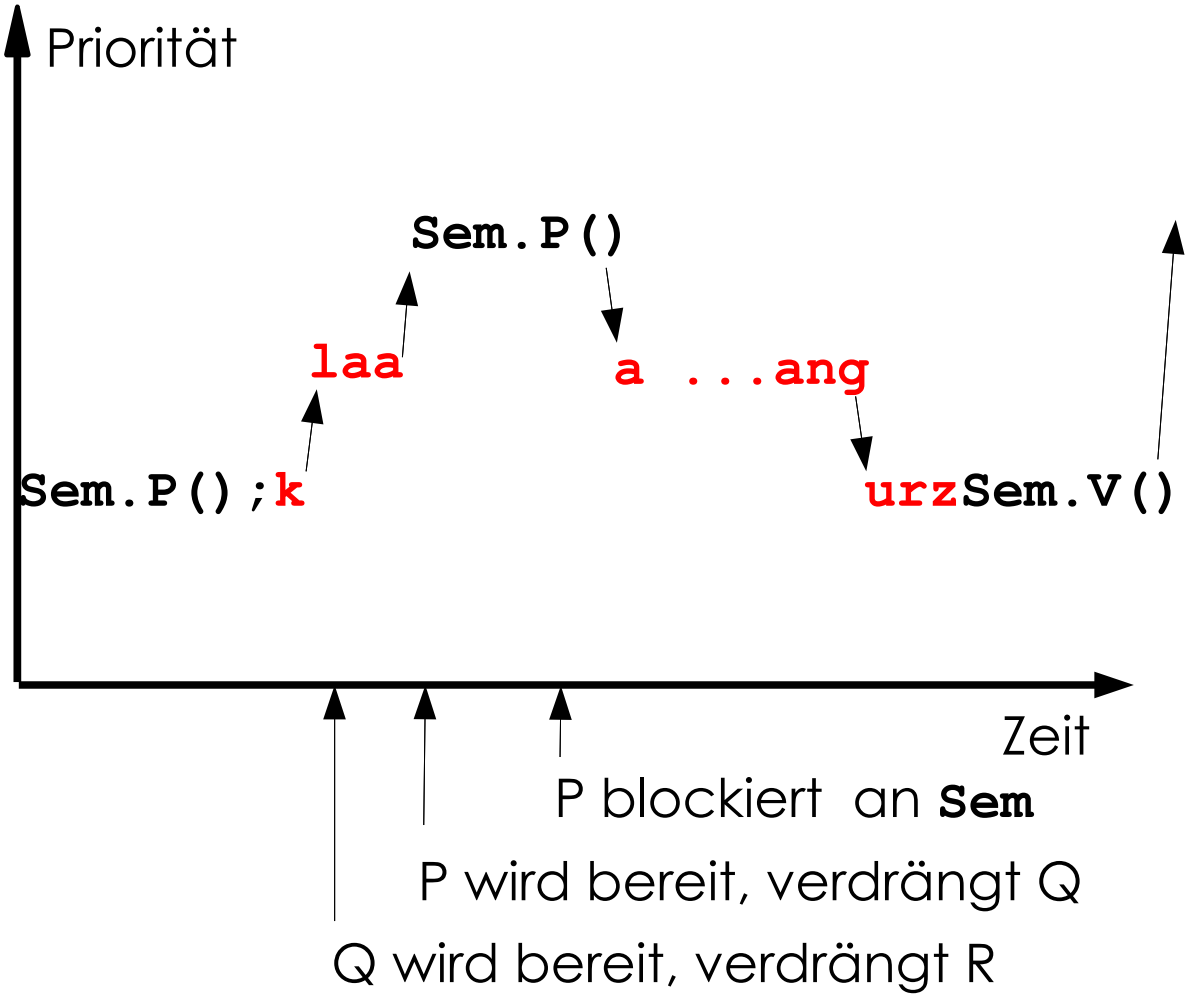
Beispielszenario:

- drei Threads P, Q, R
- P höchste, R niedrigste Priorität
- werden bereit in der Reihenfolge: R, Q, P

```
cobegin
  //P:
  {Sem.P();kurz;Sem.V()};
  //Q:
  {laaaaaaaaaaaaaaaang};
  //R:
  {Sem.p();kurz;Sem.V()};
coend
```

„Prioritätsumkehr“ (Priority Inversion)

```
cobegin
  //P:
  {Sem.P();kurz;Sem.V()};
  //Q:
  {laaaa ... aaaaaaaang};
  //R:
  {Sem.P();kurz;Sem.V()};
coend
```



P hat höchste Priorität, kann aber nicht laufen,
weil es an von R gehaltenem Semaphor blockiert und
Q läuft und damit R 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.

Ausblick

- Interaktion mit Speicher und Adressierung
- Kommunikation von Prozessen in einem verteilten System