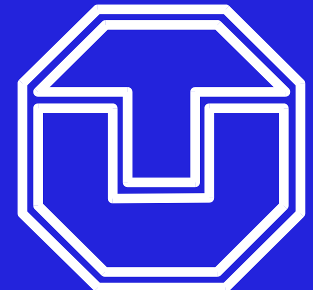
Threads

Betriebssysteme

Hermann Härtig
TU Dresden

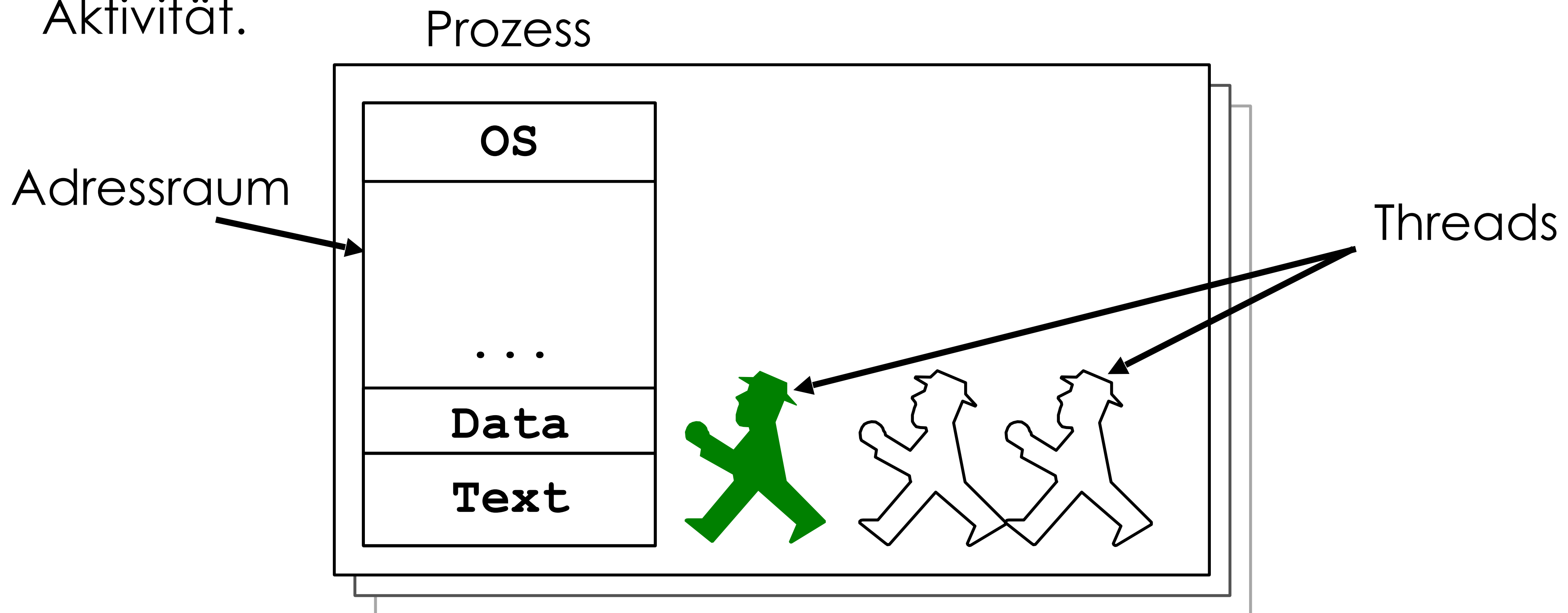


Definition: Thread

Eine selbständige

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

Aktivität.



Thread-Zustände

- **aktiv:** Threads wird gerade auf der CPU ausgeführt

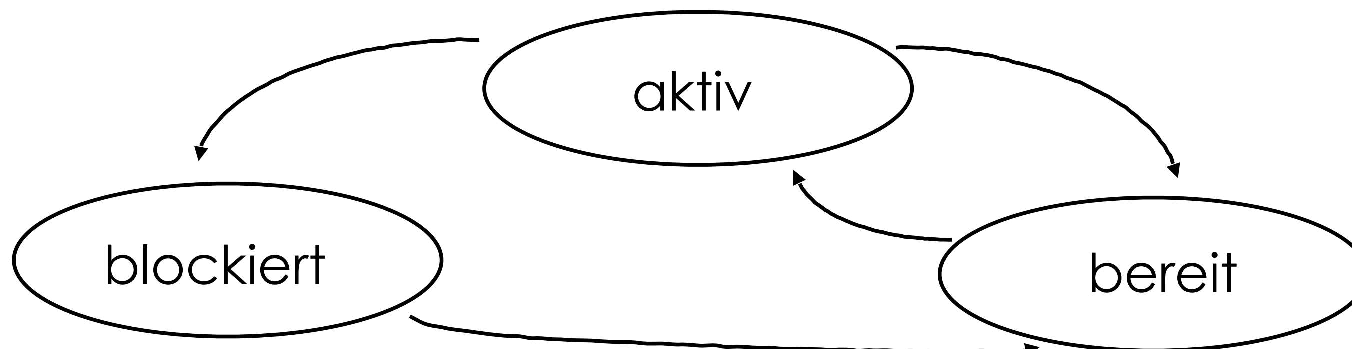
  Pro CPU ist zu jedem Zeitpunkt maximal ein Thread aktiv.

- **blockiert:** warten auf ein Ereignis (z. B. Botschaft, Freigabe eines Betriebsmitels) um weiterarbeiten zu können.

  Beim Blockieren wird CPU für andere Threads freigegeben.

- **bereit:** nicht blockiert, aber momentan nicht aktiv

  Die Bereit-Menge enthält alle Threads, die ummittelbar aktiv werden können.



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 zu jedem Zeitpunkt genau einer CPU zugeordnet
- nur die bereiten Threads erhalten CPU (werden aktiv)
- Fairness: 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) {}`** darf nicht dazu führen, dass andere Threads nie wieder „drankommen“

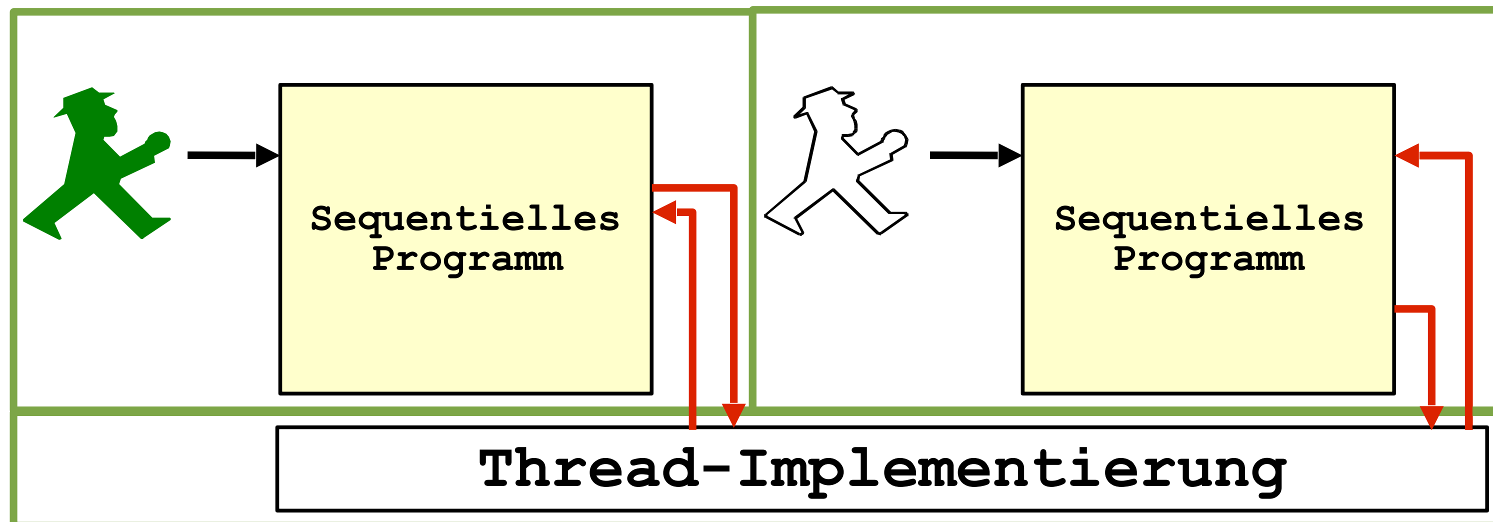
Kooperative vs. preemptive Umschaltung

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“



Beispiel

Langlaufender Thread

```
Thread1 {

  raytrace(image[0]);


  raytrace(image[1]);


  raytrace(image[2]);


  raytrace(image[3]);

}
```

Periodischer Thread

```
Thread2 {

  while (true) {

     receive(msg);
     // blockiert Thread2
     // bis zum Eintreffen
     // der Nachricht
     handle(msg);
  }
}
```

Kooperative Umschaltung

Langlaufender Thread

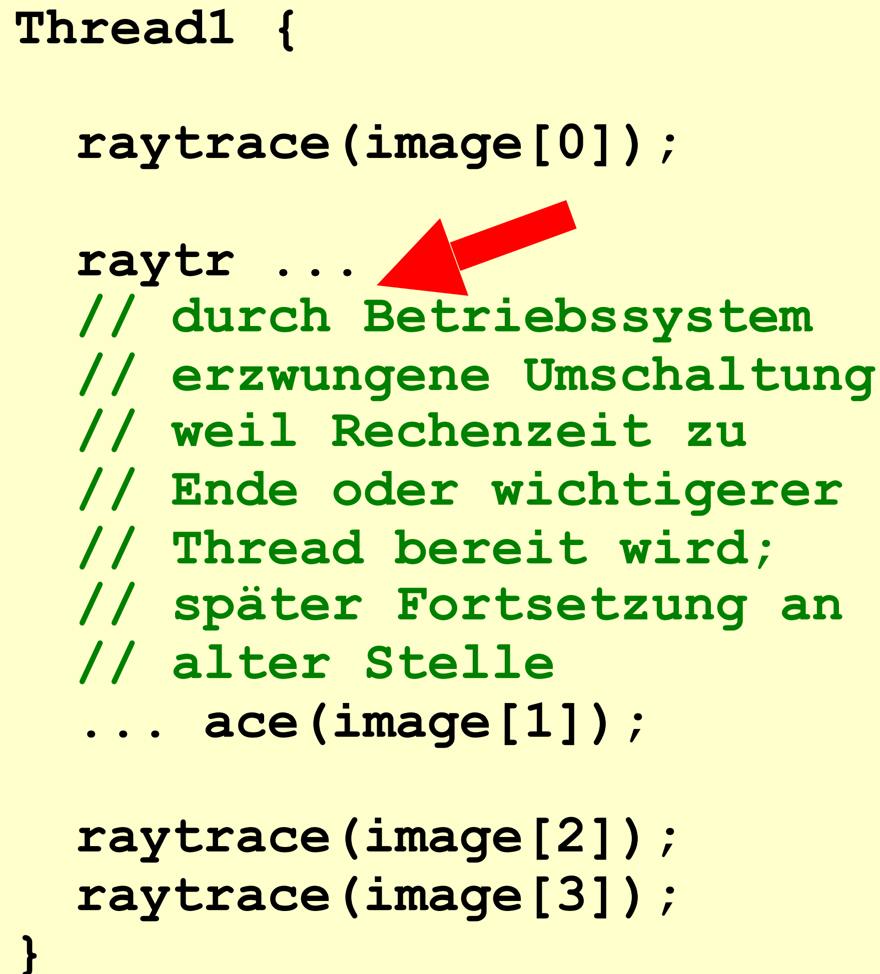
```
Thread1 {

  raytrace(image[0]);
  schedule();

  raytrace(image[1]);
  schedule();

  raytrace(image[2]);
  schedule();

  raytrace(image[3]);
  schedule();
}
```

Periodischer Thread

```
Thread2 {

  while (true) {

     receive(msg);
     // blockiert Thread2
     // bis zum Eintreffen
     // der Nachricht
     handle(msg);
  }
}
```

Preemptive Umschaltung

Langlaufender Thread



Periodischer Thread

```
Thread2 {

  while (true) {

     receive(msg);
     // blockiert Thread2
     // bis zum Eintreffen
     // der Nachricht
     handle(msg);
  }
}
```

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

Umschaltmechanismen

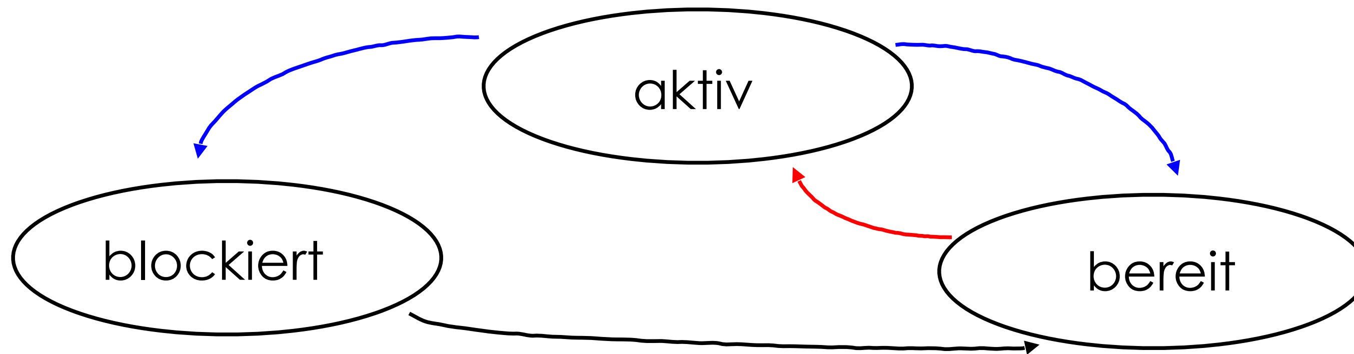
`schedule()`

- 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*

Zustand eines Threads

- Kern-Stack
- CPU-Zustand: CPU-Register, FPU-Register, MMX, SSE, AVX, …

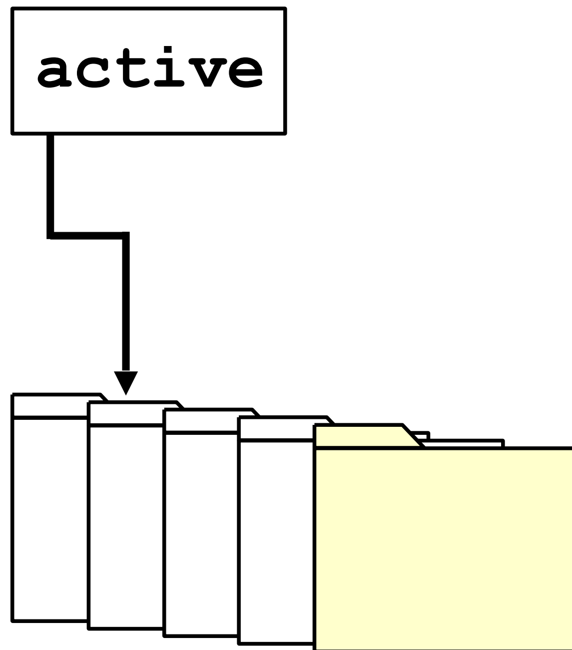
  ein Nutzer eines (nicht kooperativen) Thread-Systems muss sich darauf verlassen können, dass nicht ein anderer Thread einen Teil seiner Register zerstört hat
- Thread-Zustand: aktiv, bereit, oder blockiert
- Verweis auf Adressraum
- Scheduling-Attribute: z.B. Priorität, verbrauchte Zeit

Thread Control Block (TCB)

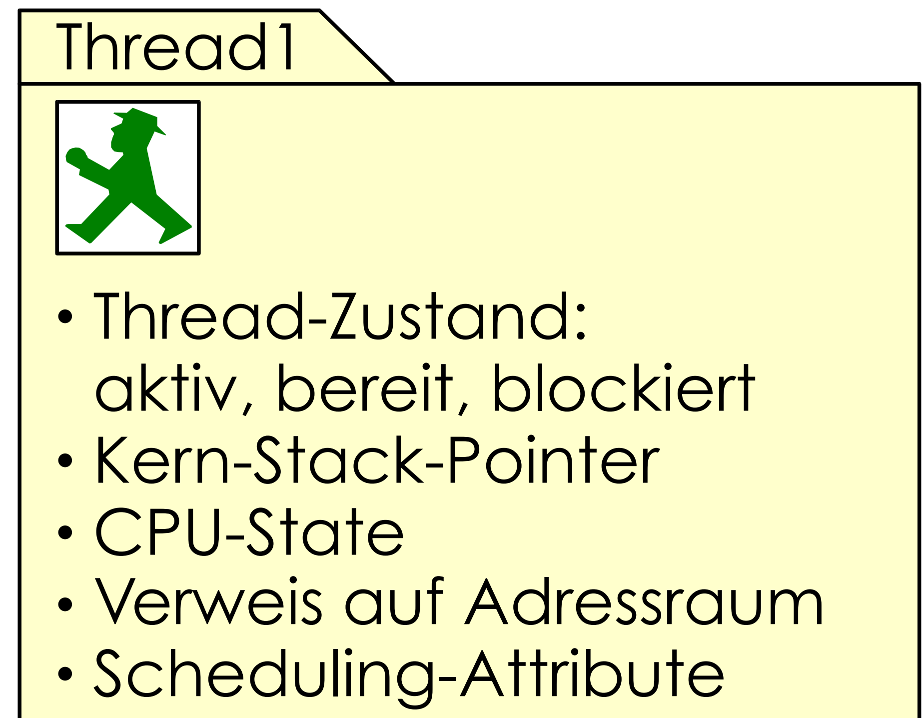
- zentrale Struktur des Kerns zur Verwaltung eines Threads

Thread Control Blocks und TCB-Liste

- TCB-Liste (**`tcblist`**)
- aktiver Thread (**`active`**)
  globale Variable der Thread-Implementierung



Thread Control Block



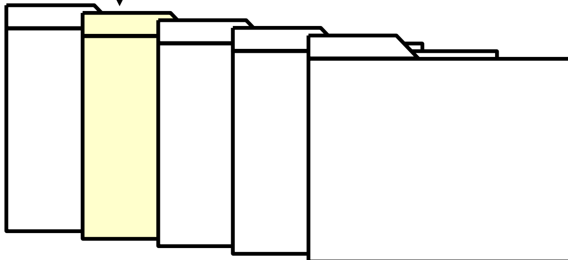
Thread Control Blocks und TCB-Liste

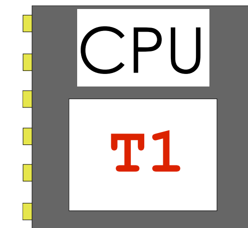
`tcblist[x]`	Eintrag in der TCB-Tabelle für Thread x
`active`	aktiver Thread
`target`	Ziel-Thread
`tcblist[active]`	aktueller TCB
`store_cpu_state`	speichert Register in aktuellen TCB
`load_cpu_state`	restauriert Register aus aktuellem TCB
`SP`	Stack Pointer: Zeiger an die aktuelle Position im Stack

Thread-Umschaltung

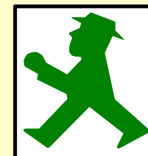
```
switch_to(target)
{
  store_cpu_state();
  tcblist[active].SP = SP;
  active = target;
  SP = tcblist[active].SP;
  load_cpu_state();
}
```

active





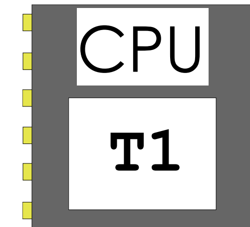
Thread1



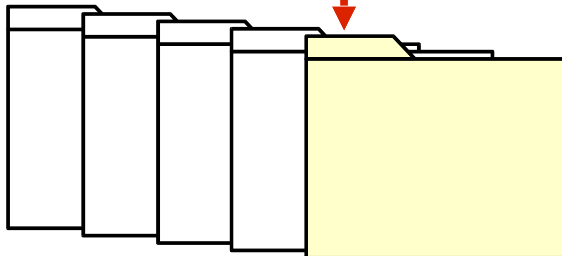
- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling-Attribute

Thread-Umschaltung

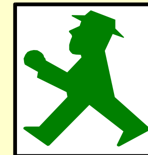
```
switch_to(target)
{
  store_cpu_state();
  tcblist[active].SP = SP;
  active = target;
  SP = tcblist[active].SP;
  load_cpu_state();
}
```



`active`



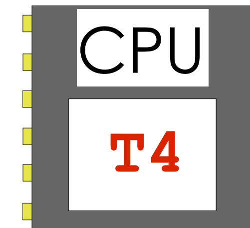
Thread4



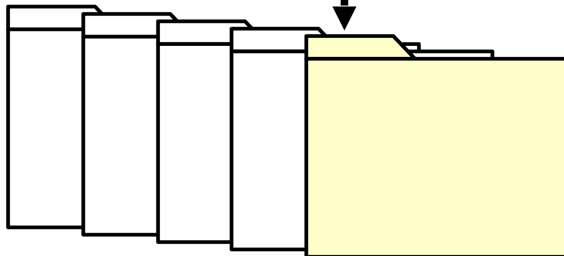
- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling-Attribute

Thread-Umschaltung

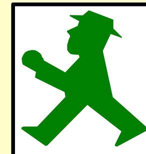
```
switch_to(target)
{
  store_cpu_state();
  tcblist[active].SP = SP;
  active = target;
  SP = tcblist[active].SP;
  load_cpu_state();
}
```



active



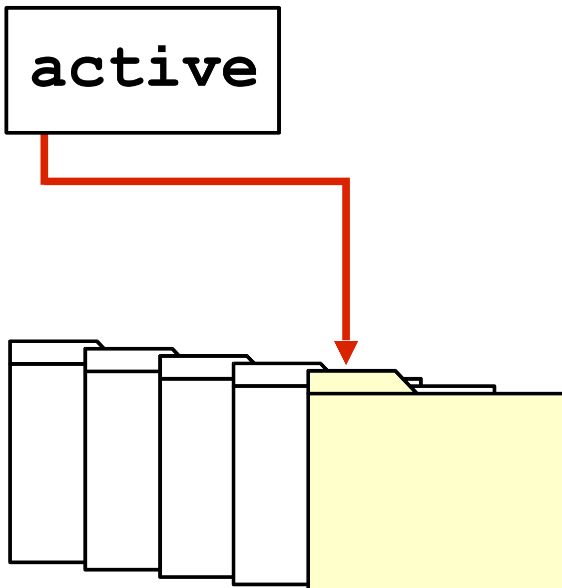
Thread4



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

Thread-Erzeugung

```
switch_to(target)
{
  store_cpu_state();
  tcblist[active].SP = SP;
  active = target;
  SP = tcblist[active].SP;
  load_cpu_state();
}
```



Umschaltstelle

- alle nicht aktiven Threads stehen im Kern an dieser Umschaltstelle

Neuer Thread:

- finde freien TCB
- initialisiere Register des Threads analog zu store_cpu_state() im TCB
- Ausführung des neuen Threads beginnt an dieser Umschaltstelle

Bewertung der kooperativen Umschaltung

- Nicht kooperierende Threads können System lahm legen
- Sehr aufwändig, Umschaltstellen im Vorhinein festzulegen
- heute sehr geringe Bedeutung, praktisch keine mehr in Betriebssystemen
- Lediglich zur Implementierung von Threads im User-Mode: „User-Level Thread Packages"

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

Unterbrechungen

Unterbrechungen

- unterbrechen den Ablauf eines aktiven Threads an beliebiger Stelle
- asynchron: *Interrupts*, synchron: *Exceptions*

Auslöser von Interrupts

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

Auslöser von Exceptions

- Fehler bei Instruktionen (Division durch 0 etc.)
- Seitenfehler und Schutzfehler (ausgelöst durch MMU)
- explizites Auslösen: Systemaufruf („Software-Interrupt“, Trap)

Ablauf von Hardware-Interrupts

- Gerätesteuerung (Controller) löst Unterbrechung aus
- CPU erkennt Auftreten von Interrupts zwischen Befehlen
- CPU schaltet zum Kern-Modus und Kern-Stack des aktiven Threads um und sichert dorthin:

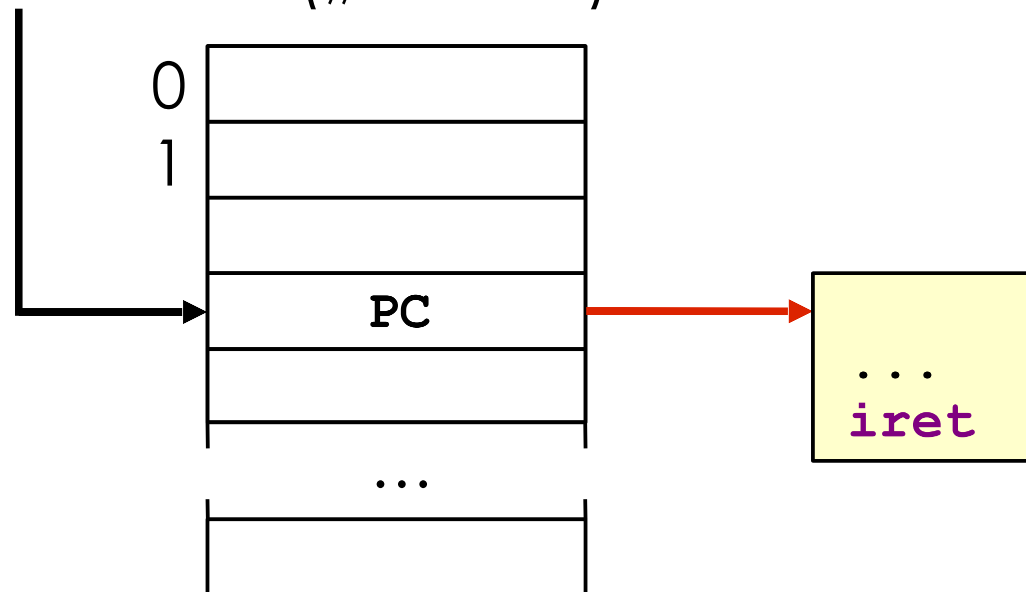
  User-Stack-Pointer, User-PC, User-Flags, …
- CPU lädt Kern-PC aus IDT (siehe nächste Folie)
- Fortsetzung der Ausführung im BS-Kern
- Instruktion **`iret`**: restauriert Modus, User-Stack-Pointer, User-PC, User-Flags vom aktuellen Kern-Stack

Ablauf von Hardware-Interrupts

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

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

Unterbrechungsnummer („vector“)



Preemptives (nicht kooperatives) Scheduling

Hardware-seitig

- Interrupt von **Timer**-Gerät
- Umschaltung in den Kern
- CPU-Zustand sichern

im Kern

```
interrupt_handler() {
  if (io_pending) {
    // ein Thread wartet auf IO
    wakeup(io_thread);
    // wartender Thread von
    // „blockiert“ nach „bereit“
    switch_to(io_thread);
    // direkter Wechsel zum
    // wartenden Thread
 // Ausführung wird hier
 // fortgesetzt, wenn auf diesen
 // Thread zurückgeschaltet wird
  }
  schedule();
  iret   // back to user mode
}
```

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

- springt zurück in User

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 später in der Vorlesung

Mehr zu Unterbrechungen

Unterbrechungsprioritäten

legen fest, welche Unterbrechung welche Unterbrechungsbehandlung unterbrechen darf

Sperren und Entsperren von Unterbrechungen

- wird benötigt, damit Datenstrukturen konsistent verändert werden können, z.B. Liste der bereiten Threads
- Steuerung: Flag in Prozessor-Steuer-Register („flags“)
- spezielle Instruktionen
  `cli` „clear interrupt flag“ disallow interrupts
  `sti` „set interrupt flag“ allow interrupts
- auch per load/store Instruktionen: push_flags/pop_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: Kern sorgfältig konstruiert

Zusammenfassung CPU-Modi

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

Umschalten zwischen den Modi

- spezielle Instruktionen lösen eine Exception (Trap) aus
- fester Einsprungpunkt im Kern pro Interrupt/Exception-Vektor, dahinter Verteilung auf die verschiedenen Fälle (welcher Gerätetreiber, welcher Systemaufruf)
- Unterbrechung erzwingt diese Umschaltung automatisch
- Notwendig: zwei unterschiedlich privilegierte Modi
- manche CPUs haben mehr Modi, z.B. x86 hat 4 „Ringe“

Wegweiser

Nutzung gemeinsamen Speichers

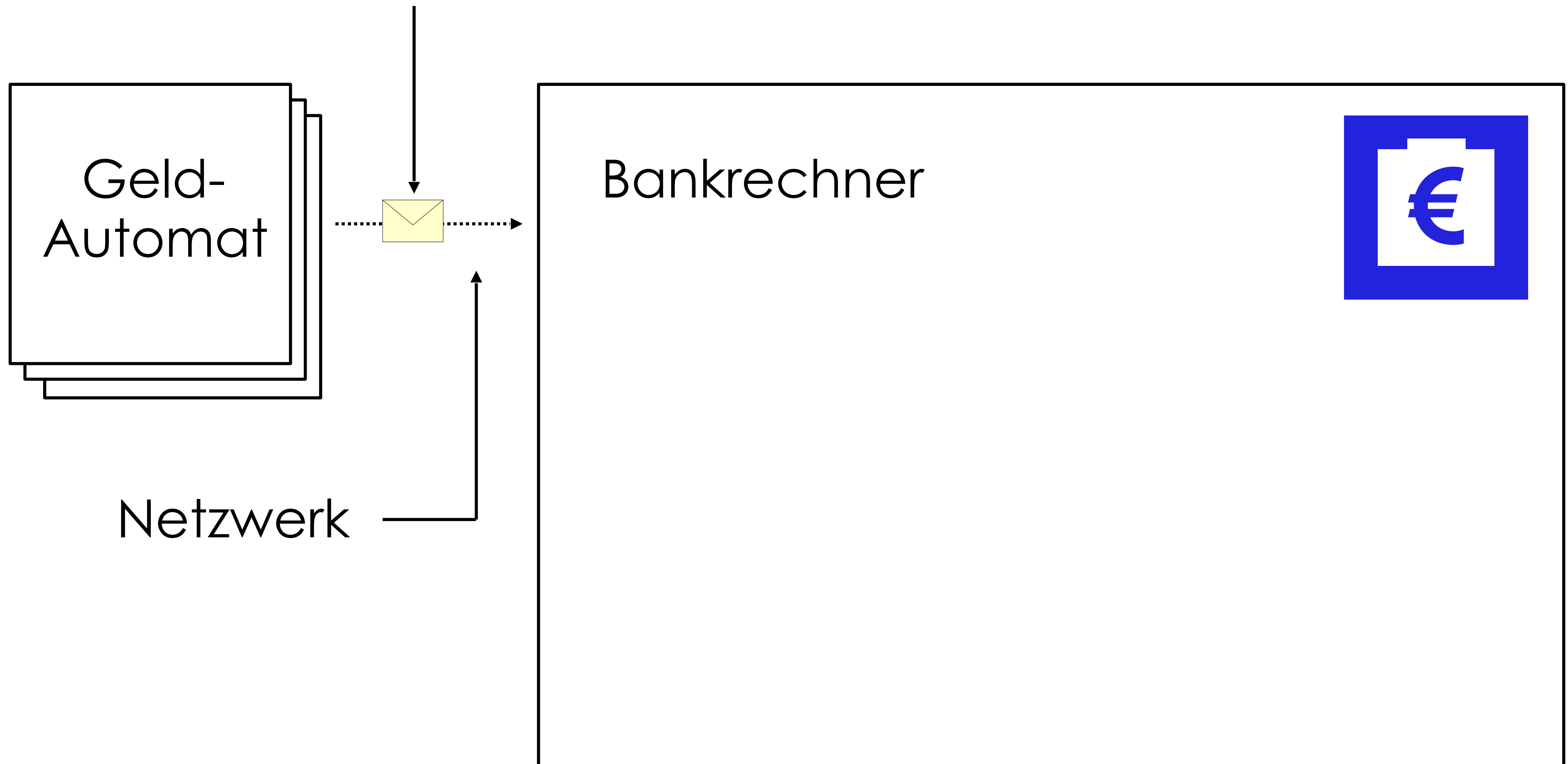
- Wettlaufsituation
- Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluss und dessen Implementierung

- mit/ohne HW-Unterstützung
- im Ein-/Mehrprozessor-Fall
- mit/ohne busy waiting

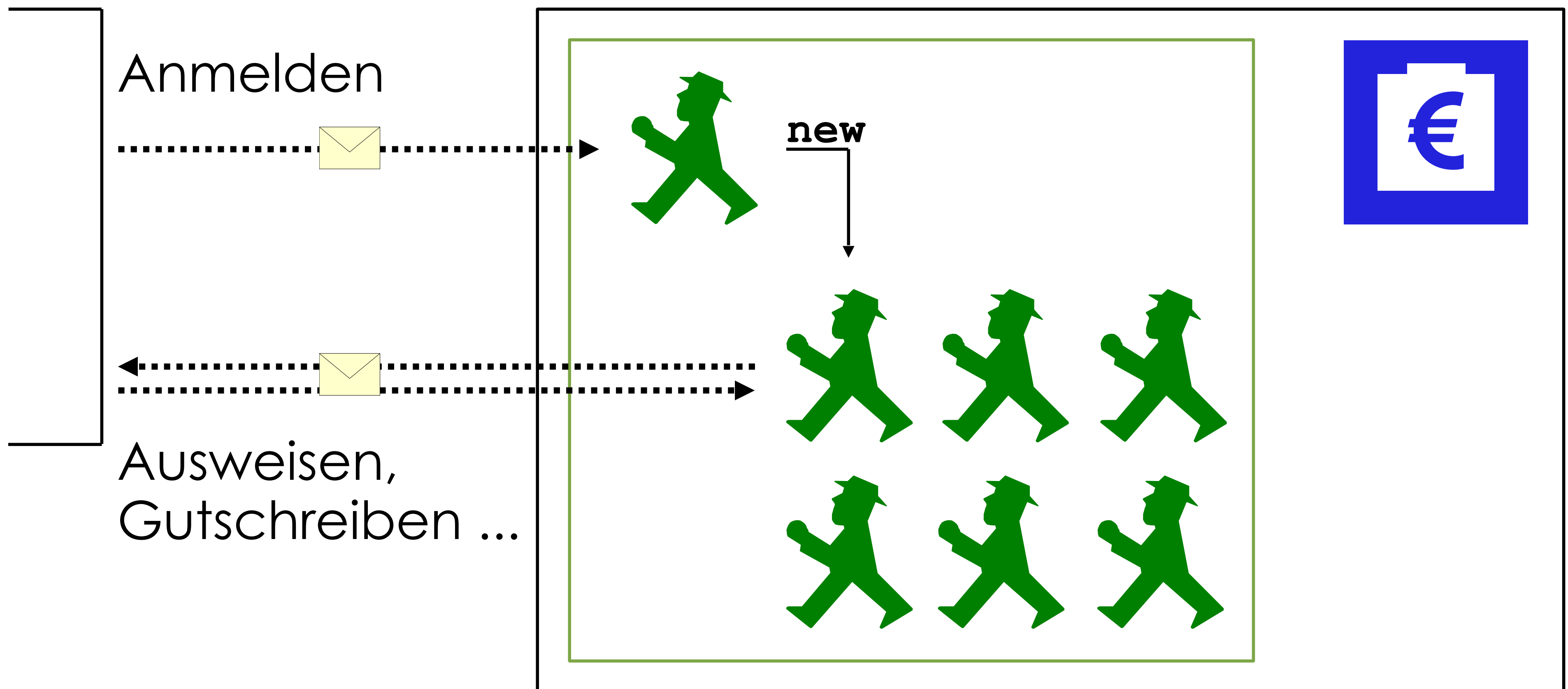
Konzepte für die parallele Programmierung

Beispiel Geldautomat



(grob nach Nichols, Buttlar, Farell)

Thread-Struktur



Erzeuge Arbeits-Thread

Gemeinsamer Speicher

Beispiel: Geld abheben

- Threads im selben Prozess benutzen denselben Adressraum
- Kommunikation erfolgt über gemeinsamen Speicher
- zwischen Prozessen kann gemeinsamer Speicher eingerichtet werden

```
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;

T1: abbuchen(1);
T2: abbuchen(20);
```

```
T1:
     ...
```

```
T2:
     ...
```

Beispiel für einen möglichen Ablauf

```
Kontostand = 20;

T1: abbuchen(1);
T2: abbuchen(20);
```

```
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`**

Variante 2

```
Kontostand = 20;

T1: abbuchen(1);
T2: abbuchen(20);
```

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

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

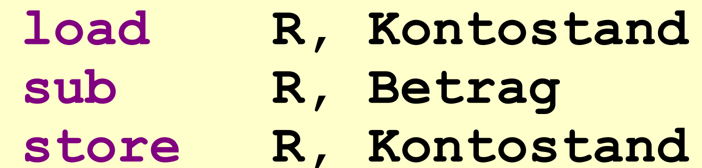
```
   Kontostand -= Betrag;
   return true;
} else return false;
```

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);

```
load     R, Kontostand
```

T2 : abbuchen(20);

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

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

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

Gemeinsamer Speicher

Wettlaufsituation
Race Condition

- Geschwindigkeit der Threads beeinflusst Korrektheit des Ergebnisses
- Ergebnisse wären bei sequenzieller Abarbeitung unmöglich

Kritischer Abschnitt
Critical Section

- der Programmteil, in dem auf gemeinsamem Speicher gearbeitet wird

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

bool abbuchen(int Betrag) {

  if (Betrag <= Kontostand) {

    Kontostand -= Betrag;
    return true;

  } else return false;
}
```

Grundsatz

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

Wegweiser

Nutzung gemeinsamen Speichers
- Wettlaufsituation
- Kritischer Abschnitt

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

Konzepte für die parallele Programmierung

Markierung des kritischen Abschnitts

```
Globale Daten



  enter_section();

  Arbeite mit globalen Daten

  leave_section();
```

Markierung des kritischen Abschnitts

```
int Kontostand;

bool abbuchen(int Betrag) {

  enter_section();

  if (Betrag <= Kontostand) {

    Kontostand -= Betrag;

    leave_section();
    return true;

  } else {

    leave_section();
    return false;
  }
}
```

Schlechte Lösung

```
bool blocked = false;

enter_section() {
  while (blocked == true) {}
  blocked = true;
}

leave_section() {
  blocked = false;
}
```

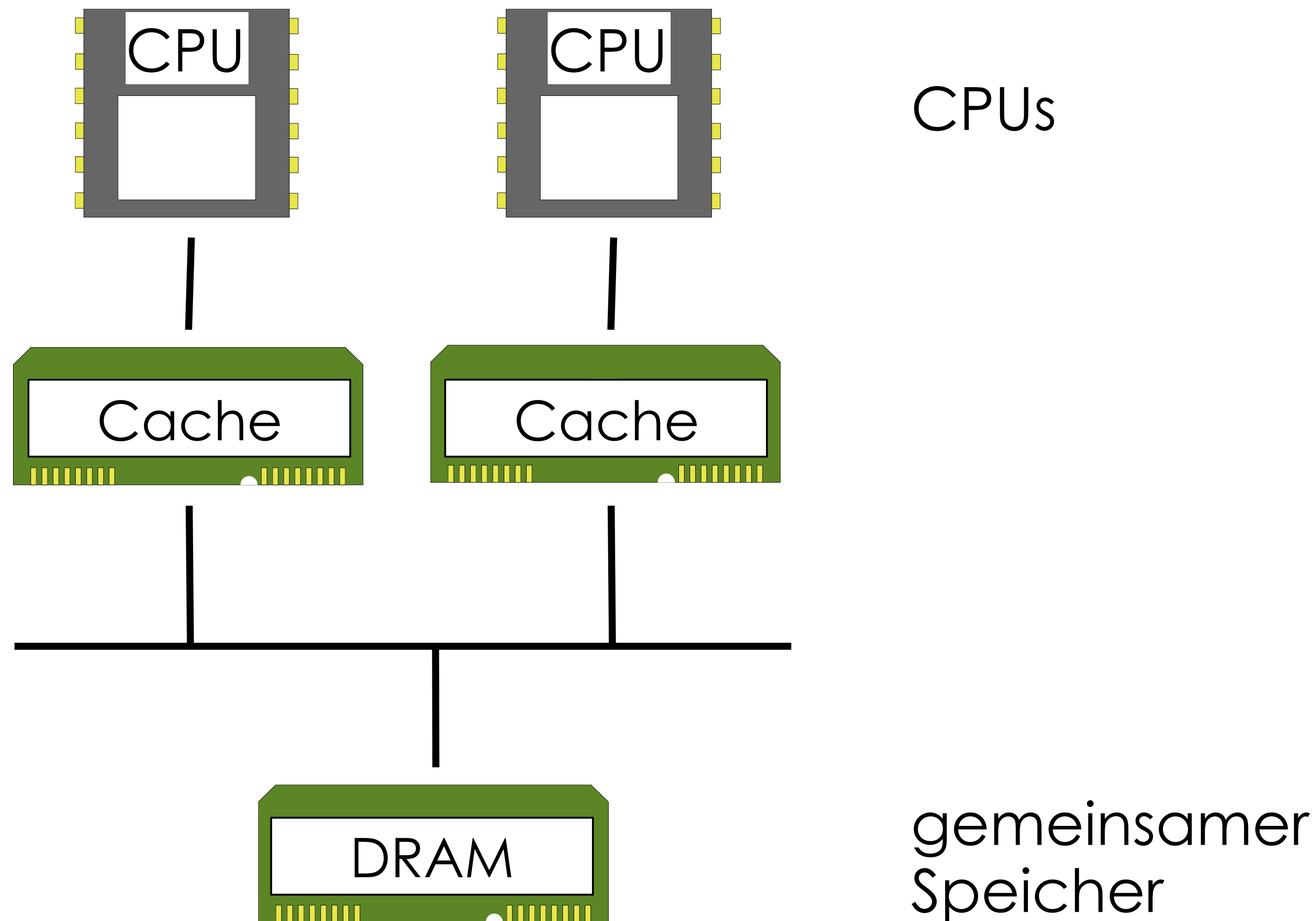
Lösung nach Peterson

```
CPU0:

enter_section(0){

  interested[0] = true;
  // Interesse bekunden
  turn = 1;

  while (interested[1] &&
    turn == 1) {}
}


leave_section(0){
  interested[0] = false;
}
```

```
CPU1:

enter_section(1){

  interested[1] = true;
  // Interesse bekunden
  turn = 0;

  while (interested[0] &&
    turn == 0) {}
}


leave_section(1){
  interested[1] = false;
}
```

Peterson-Algorithmus funktioniert **nicht** auf Prozessoren mit „weak memory consistency“ (→ Verteilte Betriebssysteme)

Modell einer Dual-Core-Maschine



CPUs

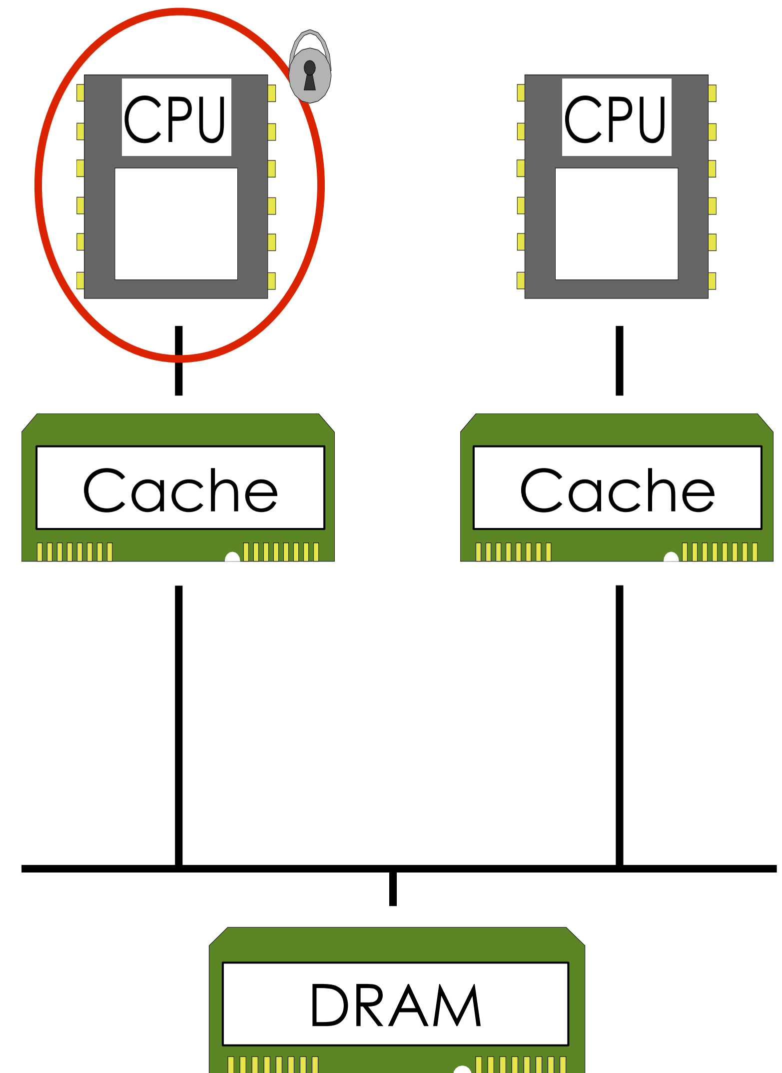
gemeinsamer Speicher

Lösungsversuch 1: Unterbrechungssperre

```
cli    // enter_section

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

sti    // leave_section
```

Die Unterbrechungssperre verhindert, dass Thread im kritischen Abschnitt preemptiert wird.

Nicht ausreichend bei mehreren Cores!



`Kontostand` im gemeinsamen Speicher

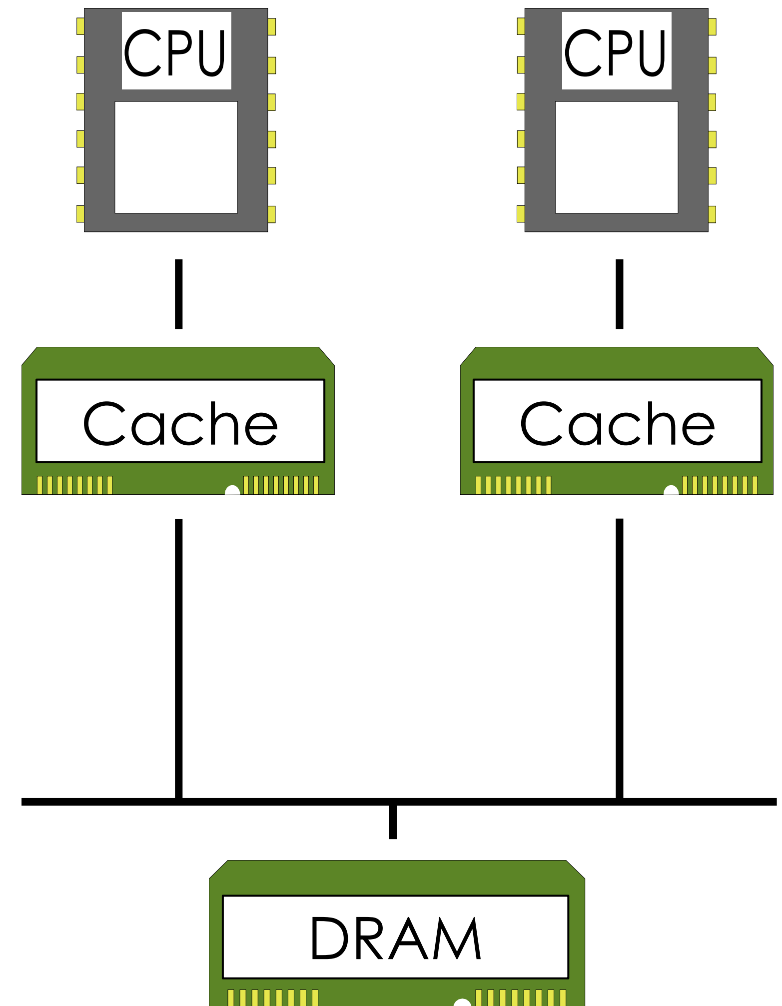
Lösungsversuch 2: KA mit einer Instruktion

```
ld      R, Betrag
sub     Kontostand, R
```

sub ist nicht unterbrechbar

Funktioniert im Multicore-Fall auch nicht.

Begründung: folgende Folien

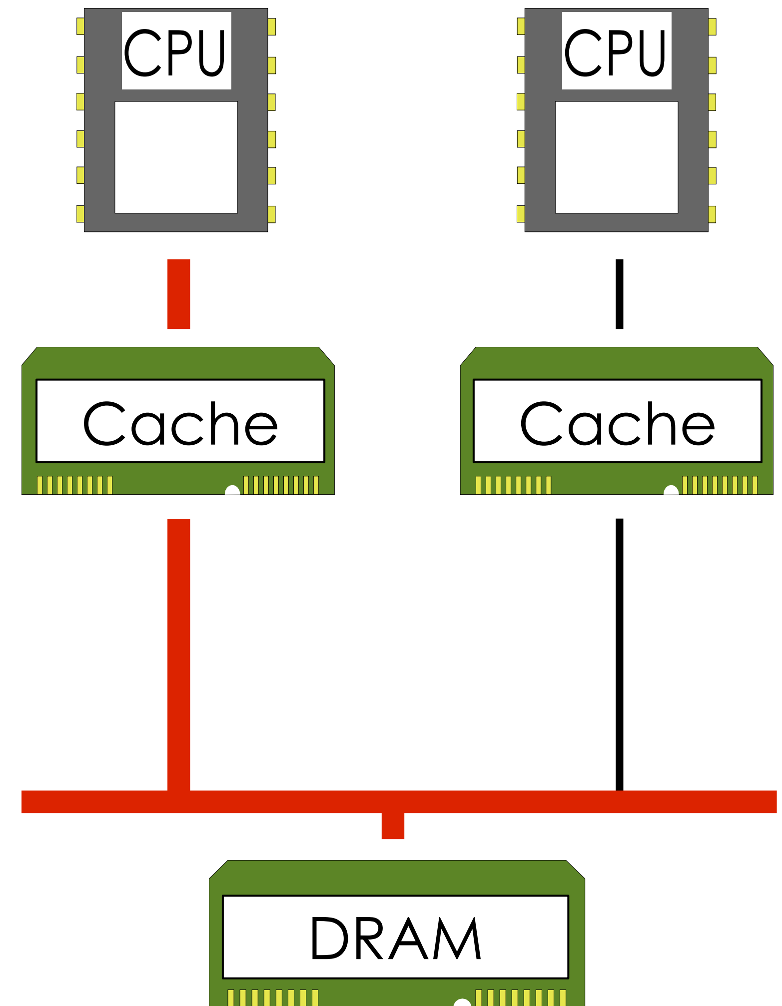


Kontostand im gemeinsamen Speicher

Lösungsversuch 2: KA mit einer Instruktion

```
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

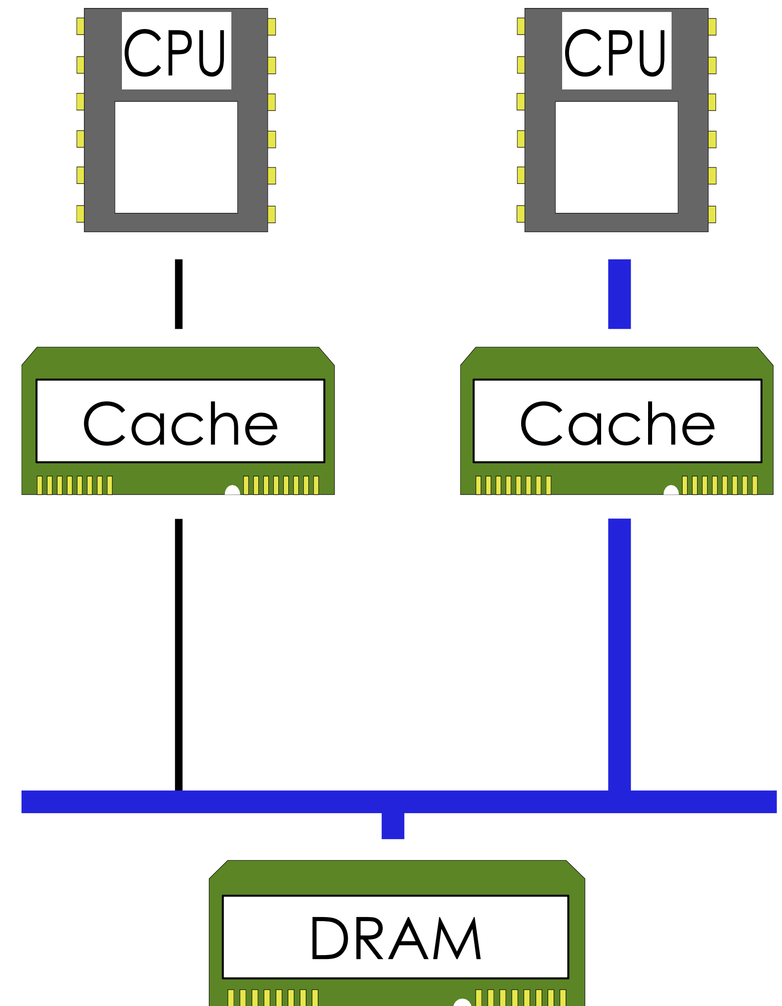


Lösungsversuch 2: KA mit einer Instruktion

```
ld        R, Betrag

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

- zweiter Threads auf anderer CPU führt **sub** mit demselben Startwert aus
- Ergebnis wie einmalige Subtraktion

Lösung funktioniert daher ebenfalls nicht!



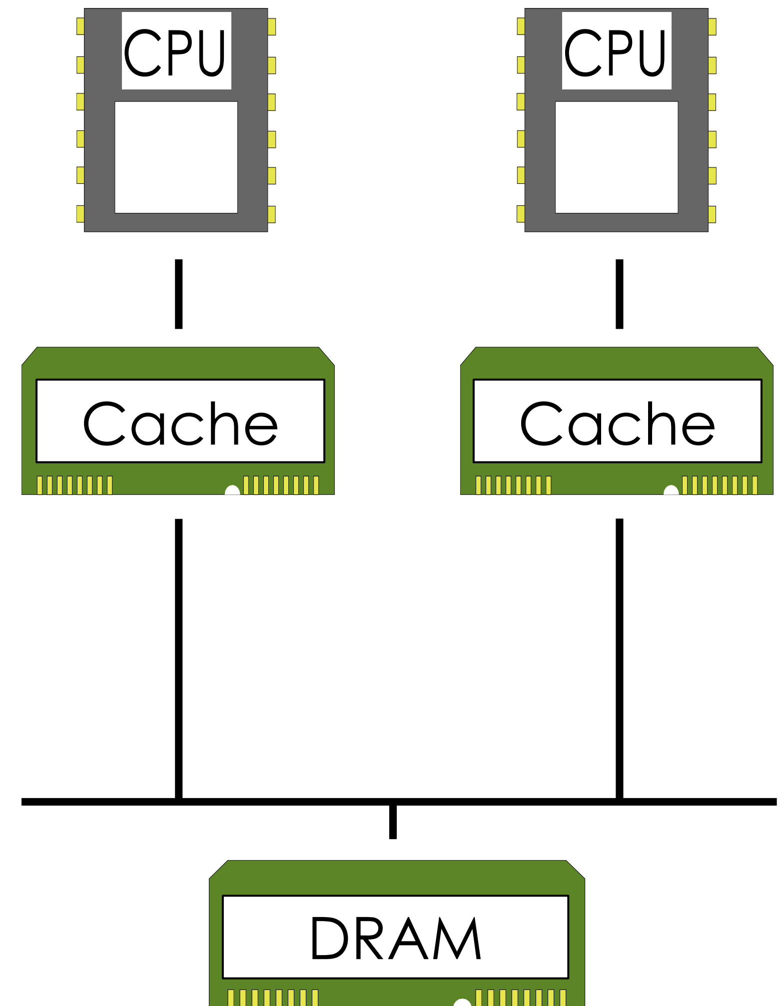
Kontostand im gemeinsamen Speicher

Lösungsversuch 3: atomare Instruktion

```
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

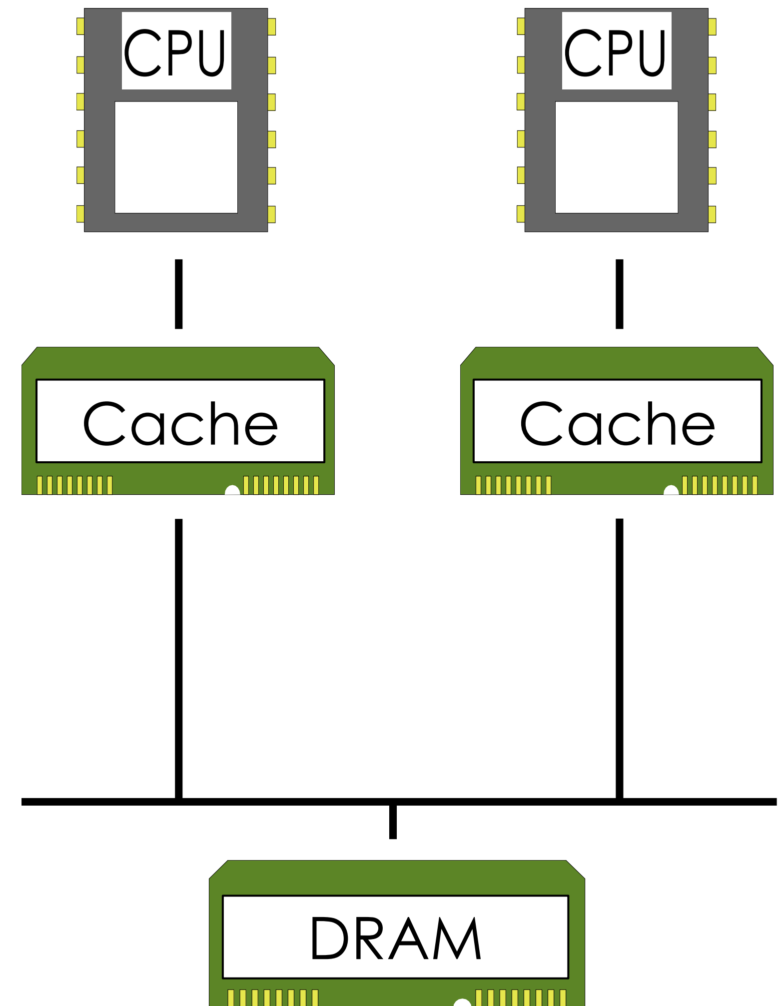
Diese Lösung funktioniert.



`Kontostand` im gemeinsamen Speicher

Implementierung von atomaren Instruktionen

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



`Kontostand` im gemeinsamen Speicher

Kritischer Abschnitt: Anforderungen

Wechselseitiger Ausschluss / Sicherheit

Keine zwei Threads dürfen sich zur selben Zeit im selben kritischen Abschnitt befinden.

Lebendigkeit

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

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() {
  cli      //disable irqs
}


unlock() {
  sti      //restore
}
```

Implementierung mittels Sperrvariable

- Prüfen und Ändern der Sperrvariable nicht atomar
- nicht sicher: mehrere Threads können kritischen Abschnitt betreten

Lösung funktioniert nicht!

```
bool lock = false;
// lock == false: frei
// lock == true: gesperrt

void lock(bool *lock) {

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

  *lock = true;
}

void unlock(bool *lock) {

  *lock = false;
}
```

Implementierung mit HW Unterstützung

Vorteile

- funktioniert für mehrere CPUs

Nachteile

- Aktives Warten: **busy waiting**
- hohe Busbelastung
- ein Thread kann verhungern
- nicht für Threads auf demselben Prozessor

Konsequenz

- geeignet für kurze KA bei kleiner CPU-Anzahl

```
lock:
  mov   R, #1
  xchg  R, lock
  // R = lock; lock = 1;

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

  ret

unlock:
  mov   lock, #0
  ret
```

Implementierung ohne Busy Waiting

- Busy Waiting belegt unnötig die CPU
- zum Blockieren ist ein (teurer) Kernaufruf nötig
- also: Kernaufruf nur bei gesperrtem kritischem Abschnitt (selten)
- **hier aber: Race Condition** **`owner`** möglicherweise noch nicht gesetzt

```
pid_t owner;
lock_t lock = 0;

void lock(lock_t *lock) {

  while(xchg(*lock, 1)) {

    switch_to(owner);

  }

  owner = current;
  // 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

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

  // Semantik atomar !!

  if (*lock == old) {

     *lock = new;
     return old;

  } else {

     return *lock;
  }
}
```

```
void lock(lock_t *lock) {

  int owner;

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

      switch_to(owner);

  }
}

void unlock(lock_t *lock) {

  *lock = 0;
}
```

Alternative: Atomare Operation CAS

current == 1	lock	current == 2
	0

	2	`cas(lock, 0, 2)`
		`owner == 0`
		`// in KA`

`cas(lock, 0, 1)`
`owner == 2`
 	
`switch_to(2)`
`// nicht in KA: loop`	

		`// KA fertig` `*lock = 0`

`cas(lock, 0, 1)`
`owner == 0`
`// in KA`
	0
	1

Feingranulare Locks

```
struct Konto { int Kontostand; lock_t lock; };

bool abbuchen(int Betrag, struct Konto * konto) {

  lock(konto->lock);

  if (Betrag <= konto->Kontostand) {

     konto->Kontostand -= Betrag;

     unlock(konto->lock);
     return true;

  } else {

     unlock(konto->lock);
     return false;
  }
}
```

Feingranulare Locks

```
struct Konto { int Kontostand; lock_t lock; };

bool überweisen(…, struct Konto *k1, struct Konto *k2) {

  lock(k1->lock);
  lock(k2->lock);

        ...

  unlock(k1->lock);
  unlock(k2->lock);

}
```

→ Transaktionen

Wegweiser

Nutzung gemeinsamen Speichers

- Wettlaufsituation
- Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluss und dessen Implementierung

- mit/ohne HW-Unterstützung
- im Ein-/Mehrprozessor-Fall
- mit/ohne busy waiting

Konzepte für die parallele Programmierung

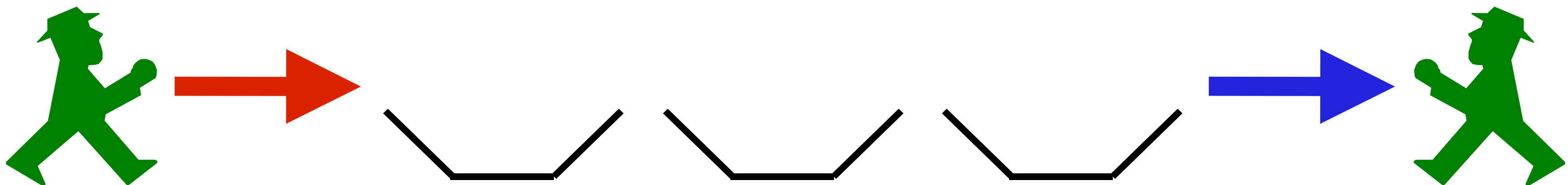
Erzeuger-Verbraucher-Problem

Beispiele

- Datenfluss-Verarbeitung: z.B. Video-Player
- Warten auf Geräte: z.B. Netzwerkpakete

Reduziert auf das Wesentliche

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



Ein Implementierungsversuch

Erzeuger

```
void produce() {

  while(true) {

     while (count == N) {}
     item = produce_item();
     insert_item(item);
  }
}
```

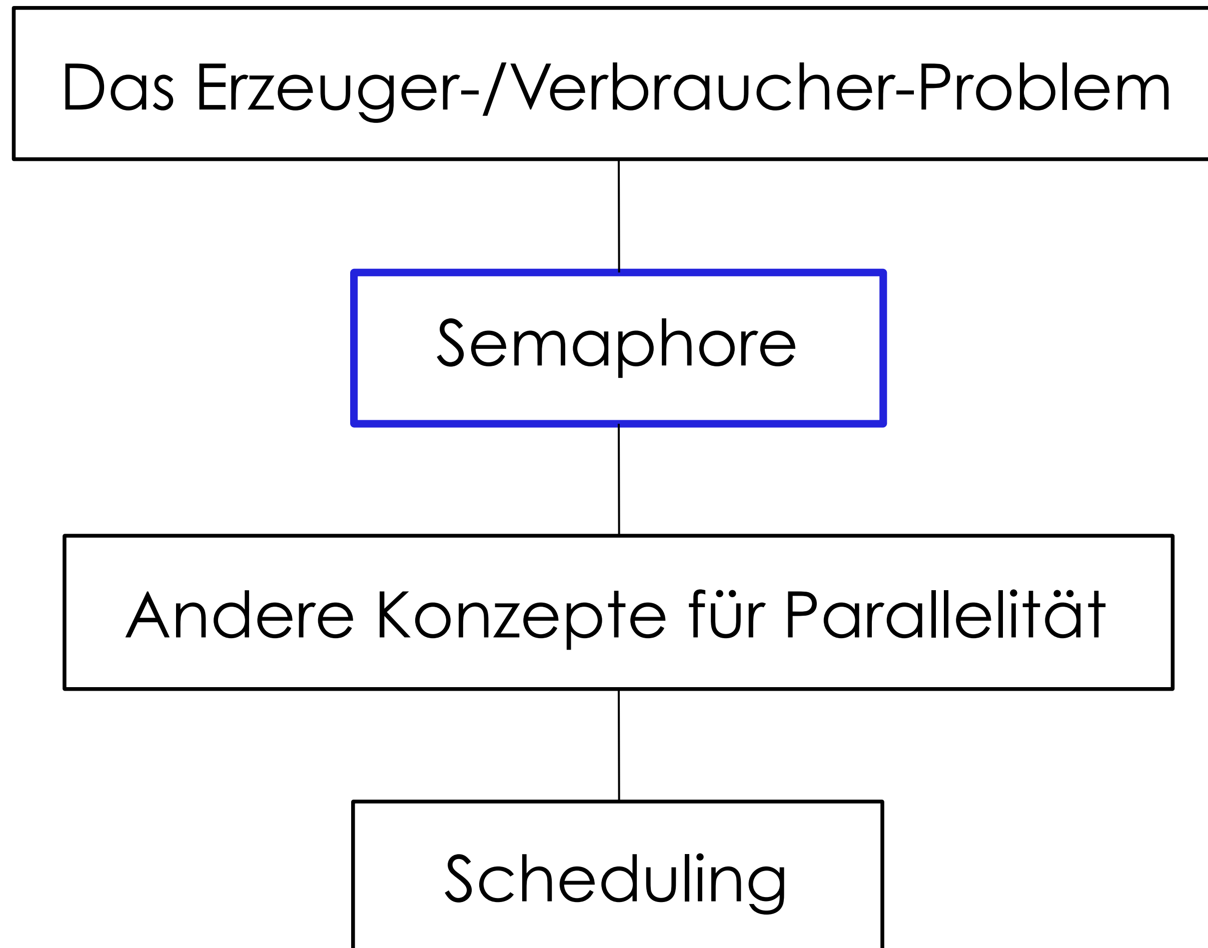
Verbraucher

```
void consume() {

  while(true) {

     while (count == 0) {}
     item = remove_item();
     consume_item(item);
  }
}
```

- Verbraucher soll warten, wenn keine Elemente im Puffer
- Erzeuger soll warten, wenn Puffer voll
- Unterschied zu wechselseitigem Ausschluss: Warten ist nicht mehr selten und kurz

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

Wegweiser



Semaphore

```
class Semaphore {

  Semaphore(int howMany);
  //Konstruktor

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

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

}
```

Kritischer Abschnitt mit Semaphoren

```
Semaphore mutex(1);
```

```
mutex.down();

do_critical_stuff();
//Kritischer Abschnitt

mutex.up();
```

```
mutex.down();

do_critical_stuff();
//Kritischer Abschnitt

mutex.up();
```

Nur **ein Thread** kann kritischen Abschnitt betreten.

Beschränkte Zahl

```
Semaphore 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

```
Semaphore mutex(3);
```

```
mutex.down();

insert_item(item)
```

```
remove_item(item)


mutex.up();
```

Maximal **drei Einträge** können im Puffer sein.

Eigenschaften von Semaphoren

- Semaphore kombinieren zwei interne Bausteine:

Zähler	für Sicherheit / Korrektheit
Warteschlange	für Lebendigkeit / Fairness

- Operationen **`down`** und **`up`** manipulieren den Zähler
- können in verschiedenen Threads verwendet werden
- kein Busy Waiting, sondern Blockieren in der Warteschlange mithilfe von Systemaufrufen:

`sleep(queue)`	blockiert Aufrufer und trägt in WS ein
`wakeup(queue)`	weckt den ältesten Thread in WS auf

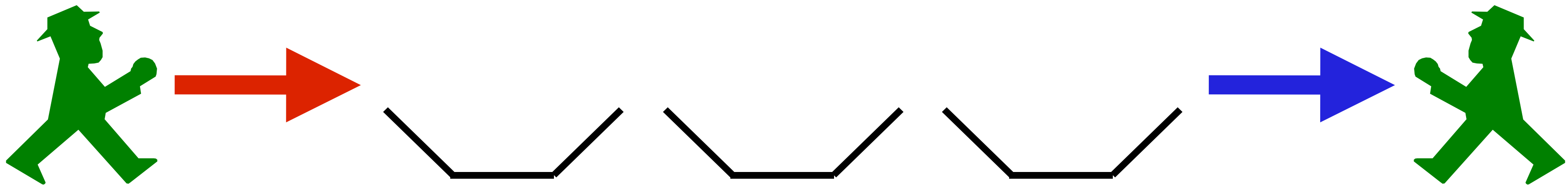
Semaphor-Implementierung

```
class Semaphore {
  int count;
  Queue queue;

 public:

  Semaphore(int howMany);

  void down();
  void up();
}

Semaphore::Semaphore(
    int howMany) {

  count = howMany;
}
```

```
Semaphore::down() {

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

  count--;
}

Semaphore::up() {

  count++;

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

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

Intuition (aber unvollständig)

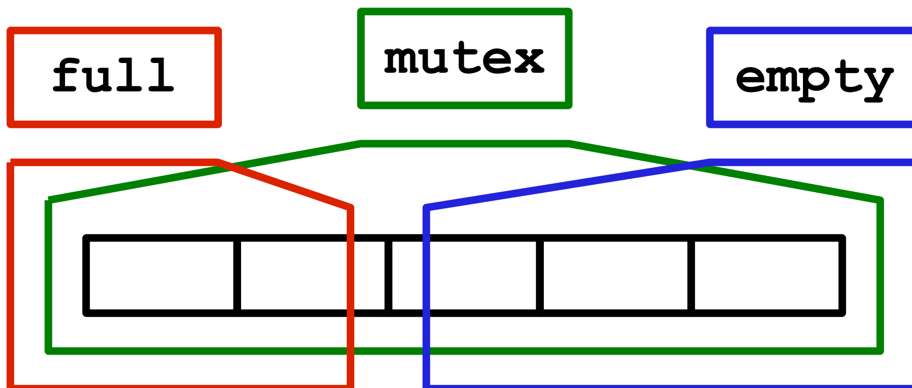
Erzeuger-Thread **3-elementiger** Puffer Verbraucher-Thread



```
Semaphore ev(3);

ev.down();

insert_item();
```

```
remove_item();


ev.up();
```

EV-Problem mit Semaphoren




```
const int size = 100
//Anzahl der Puffereinträge

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

Semaphore empty(size);
//Anzahl der freien
//Puffereinträge

Semaphore full(0);
//Anzahl der belegten
//Puffereinträge
```

EV-Problem mit Semaphoren

Erzeuger

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



    insert_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();

    insert_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();

    insert_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();

    insert_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();

    insert_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);

  }
}
```

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 Bedingungsvariablen

Erzeuger

```
void produce() {




  insert_item();
  count++;





}
```

Verbraucher

```
void consume() {




  remove_item();
  count--;





}
```

EV-Problem mit Bedingungsvariablen

Erzeuger

```
void produce() {

  lock(mutex);



  insert_item();
  count++;



  unlock(mutex);

}
```

Verbraucher

```
void consume() {

  lock(mutex);



  remove_item();
  count--;



  unlock(mutex);

}
```

EV-Problem mit Bedingungsvariablen

Erzeuger

```
void produce() {

  lock(mutex);

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

  insert_item();
  count++;



  unlock(mutex);

}
```

Verbraucher

```
void consume() {

  lock(mutex);



  remove_item();
  count--;

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

  unlock(mutex);

}
```

EV-Problem mit Bedingungsvariablen

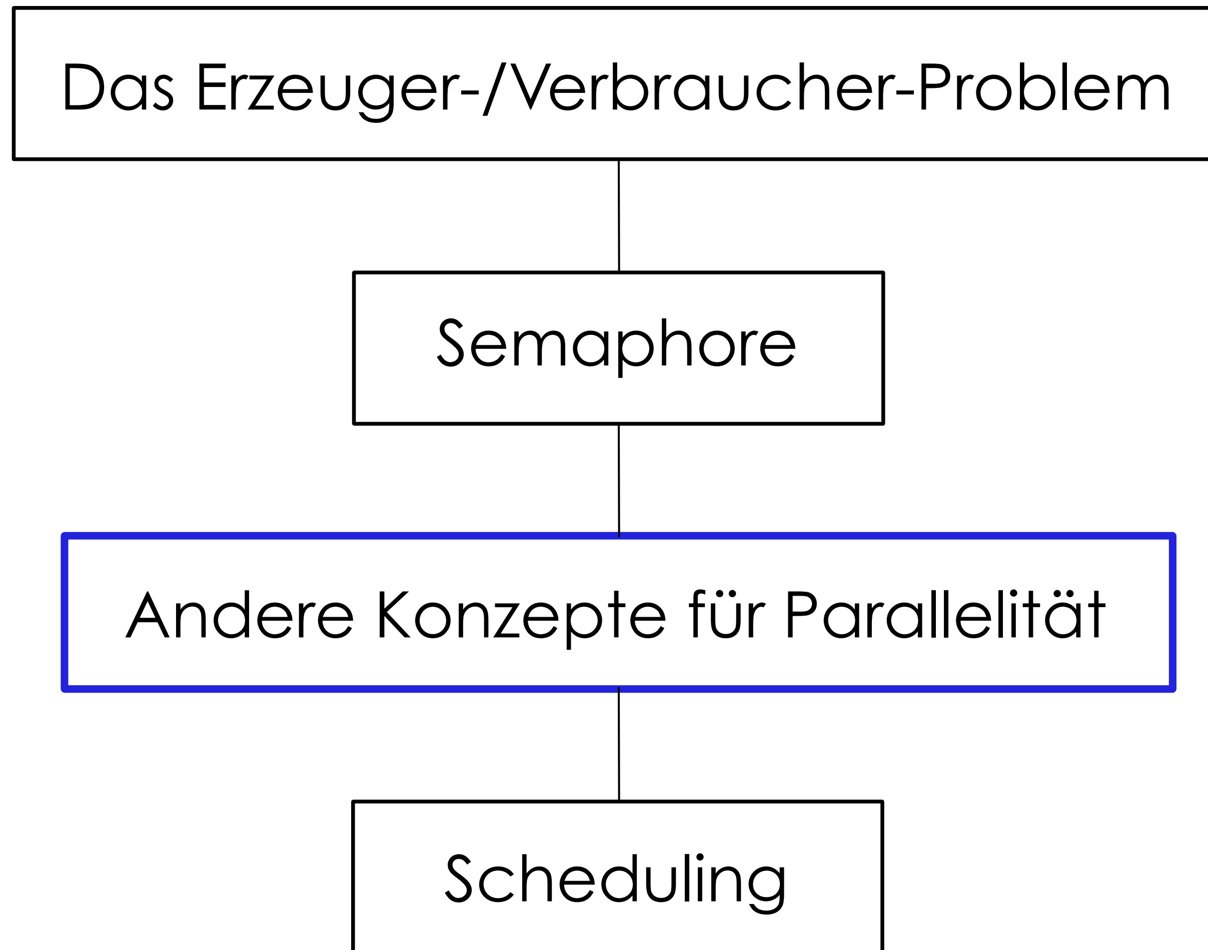
Erzeuger

```
void produce() {
  lock(mutex);

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

  insert_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);

}
```

Wegweiser



Leser/Schreiber-Problem

Lesen und Schreiben in gemeinsamen Datenstrukturen

- Mehrere Prozesse dürfen gleichzeitig lesen
- Nur einer schreiben

Subtiler Unterschied zum Erzeuger-Verbraucher-System

- Kein Puffer
- Beliebige Parallelität bei den Lesern erlaubt

Andere Sprachkonstrukte

Monitore, Aktoren, Coroutinen, …

- Datentypen, die von mehreren Threads genutzt werden können
- Operationen können kritische Abschnitte enthalten, welche unter Wechselseitigem Ausschluss ablaufen
- Implementierung mit Locks, Semaphoren, Bedingungsvariablen in der Sprachumgebung oder im Compiler

Bei mehreren beteiligten Objekten

Problem

- Jedes Konto mit Semaphoren / Bedingungsvariablen implementiert
- Konto2 nicht zugänglich/ verfügbar
- Abbruch nach 1. Teiloperation

Abhilfe

Alle an einer komplexen Operation beteiligten Objekte vorher sperren

```
void ueberweisen(
     int betrag,
     Konto konto1,
     Konto 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,
     Konto konto1,
     Konto konto2) {

 lock(konto1);
 konto1.abbuchen(betrag);

 lock(konto2);
 if (!konto2.
   gutschrift(betrag)) {

   konto1.gutschrift(betrag);
 }

 unlock(konto1);
 unlock(konto2);

}
```

Transaktionen

Motivation

- Synchronisation komplexer Operationen mit mehreren beteiligten Objekten
- Rückgängigmachen von Teiloperationen gescheiterter komplexer Operationen
- Optimistische Ausführung: Reagieren falls etwas nicht klappt, anstatt (wie bei Semaphoren) proaktiv zu sperren

Voraussetzungen

- Transaktionsmanager:
  mehr dazu in der Datenbanken-Vorlesung
- alle beteiligten Objekte verfügen über bestimmte Operationen

Konto-Beispiel mit Transaktionen

```
void ueberweisung(int betrag,
                  Konto konto1, Konto konto2) {

  begin_transaction();

  use(konto1);
  konto1.abbuchen(betrag);

  use(konto2);
  if (!konto2.gutschreiben(betrag)) {

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

  commit_transaction();
  // alle Locks werden freigegeben
}
```

Transaktionen: „ACID“

Atomar

Komplexe Operationen werden ganz oder gar nicht ausgeführt.

Konsistent (Consistent)

Es werden konsistente Zustände in konsistente Zustände überführt, Zwischenzustände dürfen inkonsistent sein.

Isoliert

Transaktionen dürfen parallel durchgeführt werden genau dann, wenn sichergestellt ist, dass das Ergebnis einer möglichen sequentiellen Ausführung der Transaktionen entspricht.

Dauerhaft

Nach dem Commit einer Transaktion ist deren Wirkung verfügbar.

Botschaften

Senden

- synchron, d. h. terminiert erst, wenn das korrespondierende **`receive/wait`** durchgeführt wurde
- Oder asynchron, d. h. terminiert sofort („fire and forget“)

Warten

- akzeptiert eine Botschaft von irgendeinem Thread
- liefert die ID des Sender-Threads

Empfangen

- akzeptiert eine Botschaft von einem bestimmten Sender-Thread

Botschaften

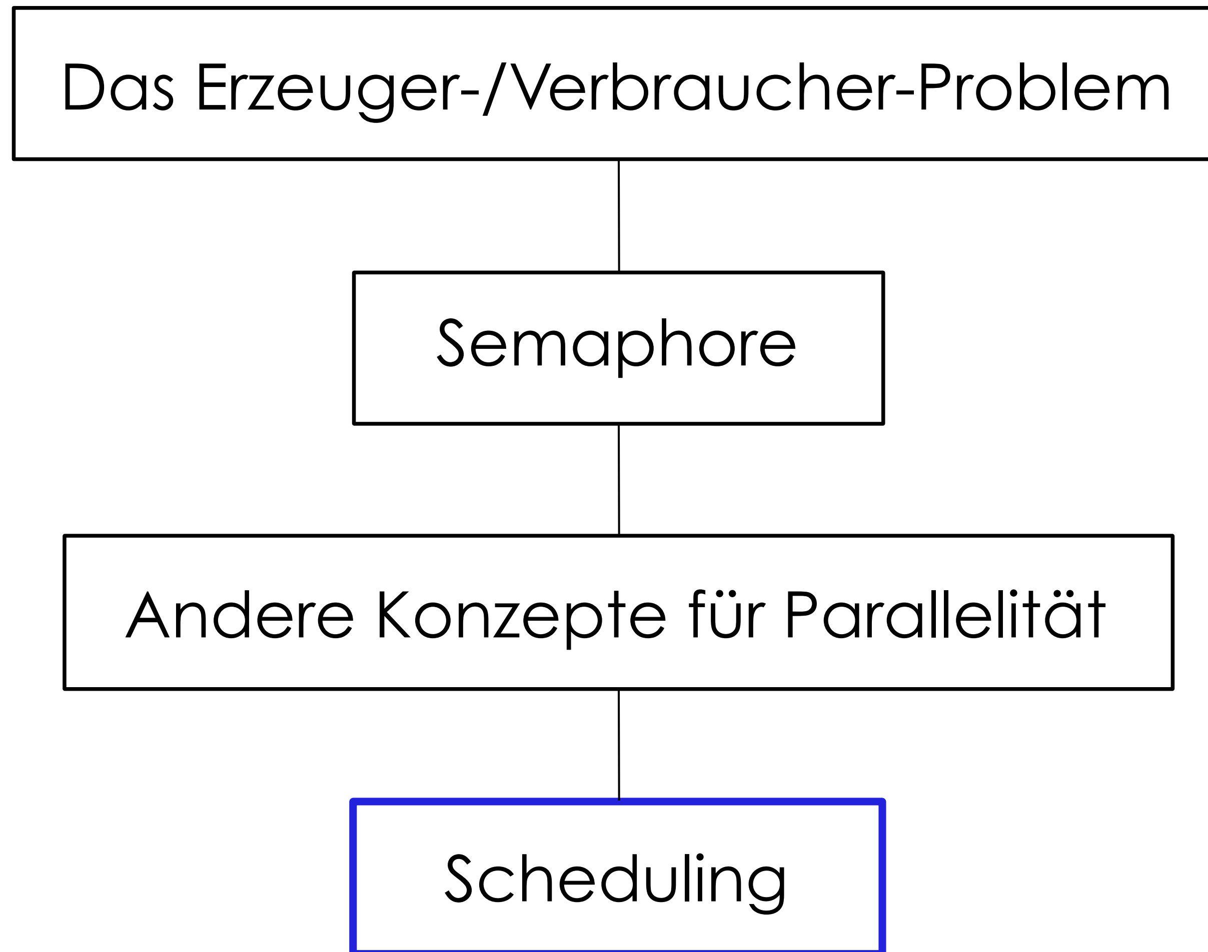
Botschaften-Puffer

- Nicht nötig bei synchronen Botschaften (einfacher)
- Asynchron: Botschaften-System hält eigene Puffer bereit oder Sender darf Botschaft nicht überschreiben, bis Empfänger sie abholt

Botschaften-Systeme

- Mikrokerne
- Message-Passing Interface (MPI) für Hochleistungsrechnen
- Kein gemeinsamer Speicher für Kommunikation nötig

Wegweiser



Scheduling

Aufgabe

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

Scheduling

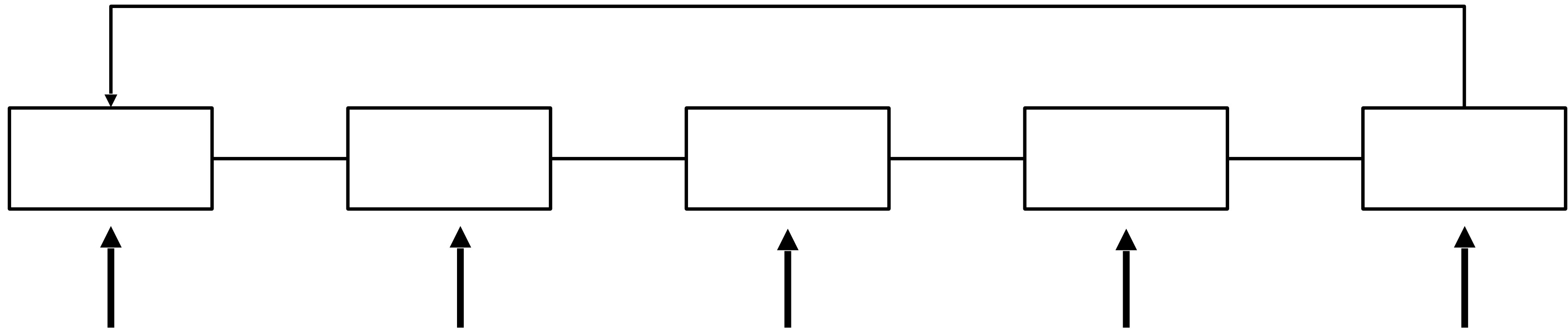
Kriterien

- Fairness: fairer Anteil an CPU für den Thread
- Effizienz: möglichst vollständige Auslastung der CPU
- Antwortzeiten: interaktive Anwendungen
- Wartezeiten: Hintergrundjobs
- Durchsatz: hohe Zahl von Aufgaben pro Zeiteinheit
- Zusagen: ein Prozess muss spätestens dann fertig sein
- Energie: Verbrauch minimieren

Probleme

- Kriterien sind widersprüchlich
- Anwender-Threads oft mit unvorhersehbarem Verhalten

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
- häufig: Kombination von Round Robin und Prioritäten

Fragestellungen

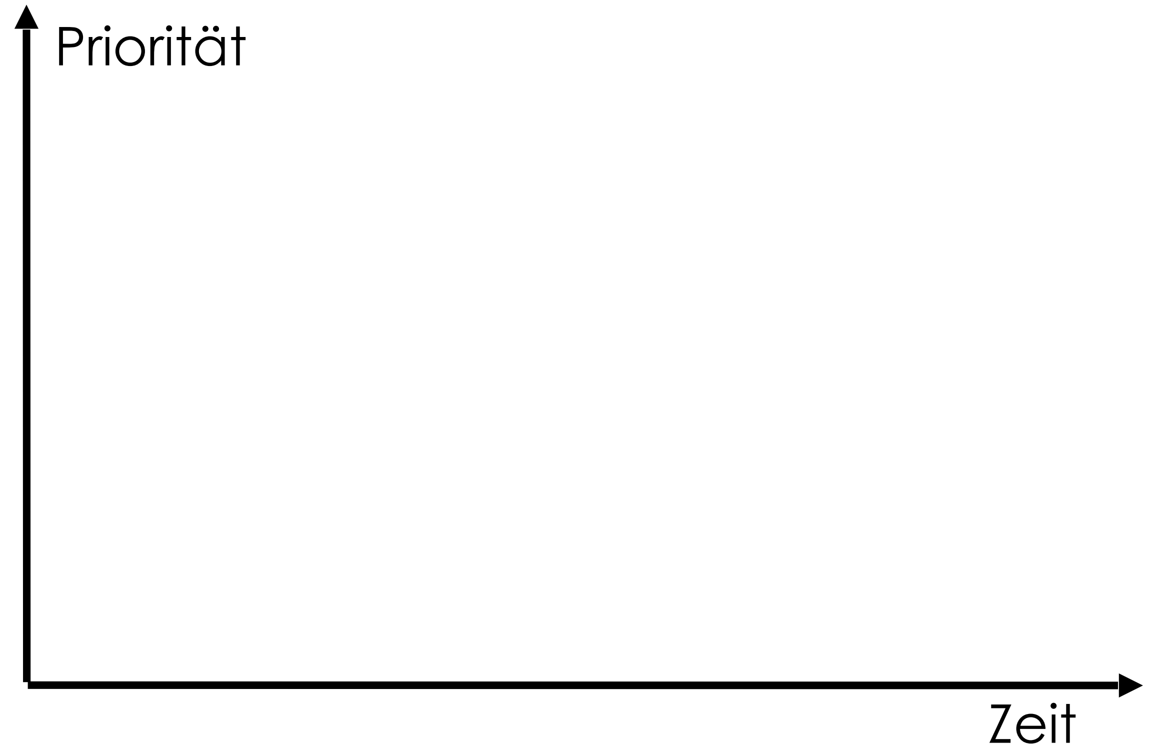
- feste Prioritäten pro Thread oder dynamisch wechselnd?
- Thread mit höherer Priorität als der rechnende wird bereit: Umschaltung sofort oder „sanfte" Umverteilung?
- Zuweisung von Prioritäten: vom Nutzer („nice") oder nach Heuristik (z.B. nach Anteil von Blockierzeit)?

„Prioritätsumkehr“ (Priority Inversion)

Beispielszenario

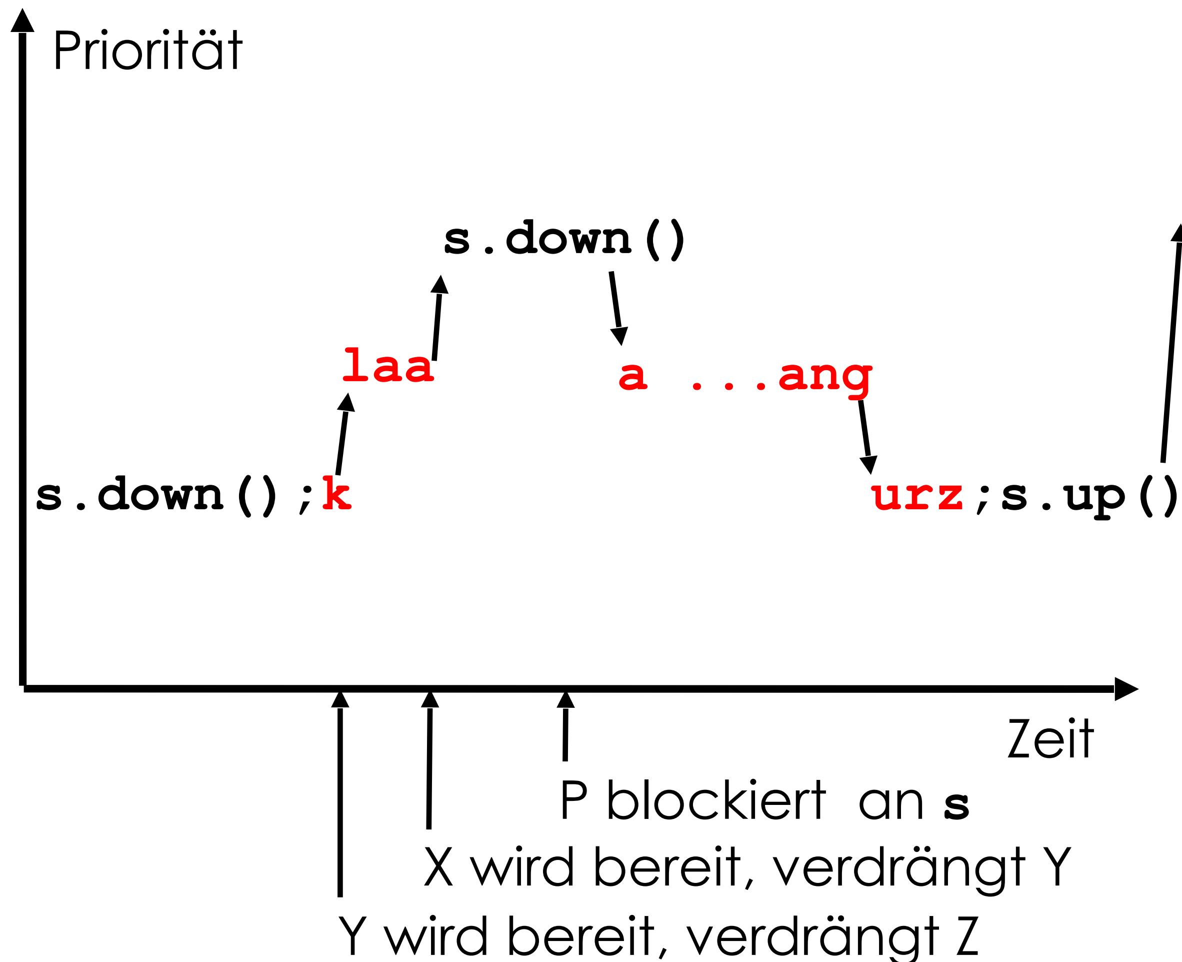
- drei Threads X, Y, Z
- X höchste, Z niedrigste Priorität
- werden bereit in der Reihenfolge: Z, Y, X

```
//X:
  {s.down();kurz;s.up()};

//Y:
  {laaaaaaaaaaaaaaaang};

//Z:
  {s.down();kurz;s.up()};
```



„Prioritätsumkehr“ (Priority Inversion)

```
//X:
   {s.down();kurz;s.up()};

//Y:
   {laaaaaaaaaaaaaaaang};

//Z:
   {s.down();kurz;s.up()};
```



- 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