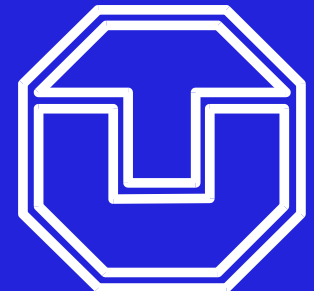


Threads

Betriebssysteme

Hermann Härtig
TU Dresden



Wegweiser

Einführung und Wiederholung

Implementieren mehrerer Threads
auf einem Rechner/Prozessor

Zusammenspiel/Kommunikation
mehrerer Threads

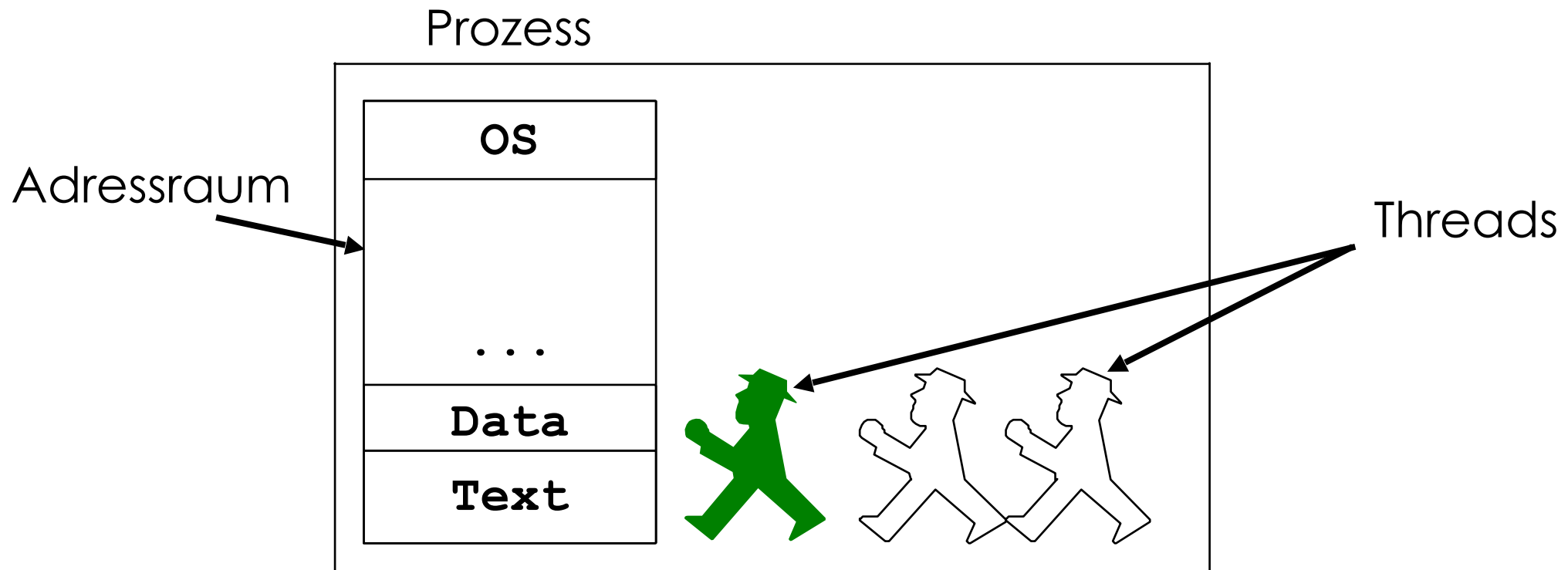
Scheduling

Definition: Thread

Eine selbständige

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

Aktivität.

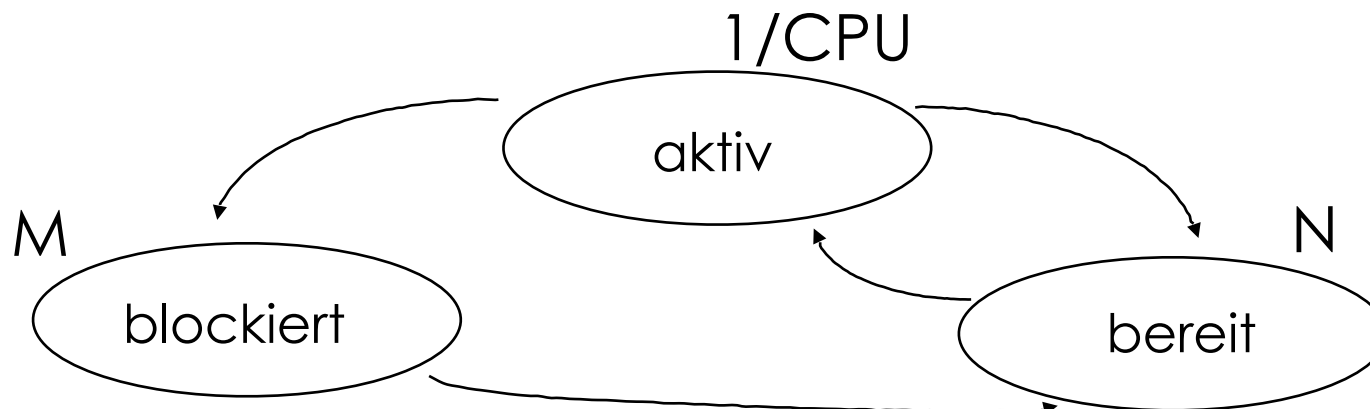


Notationen

- $P \parallel Q$
- `create (Prozedur, Stack) ;`
 `...`
 `join (thread) ;`
- `COBEGIN`
 `P (Params) ; Q () ;`
 `COEND`

Thread-Zustände

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



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*)
- „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“

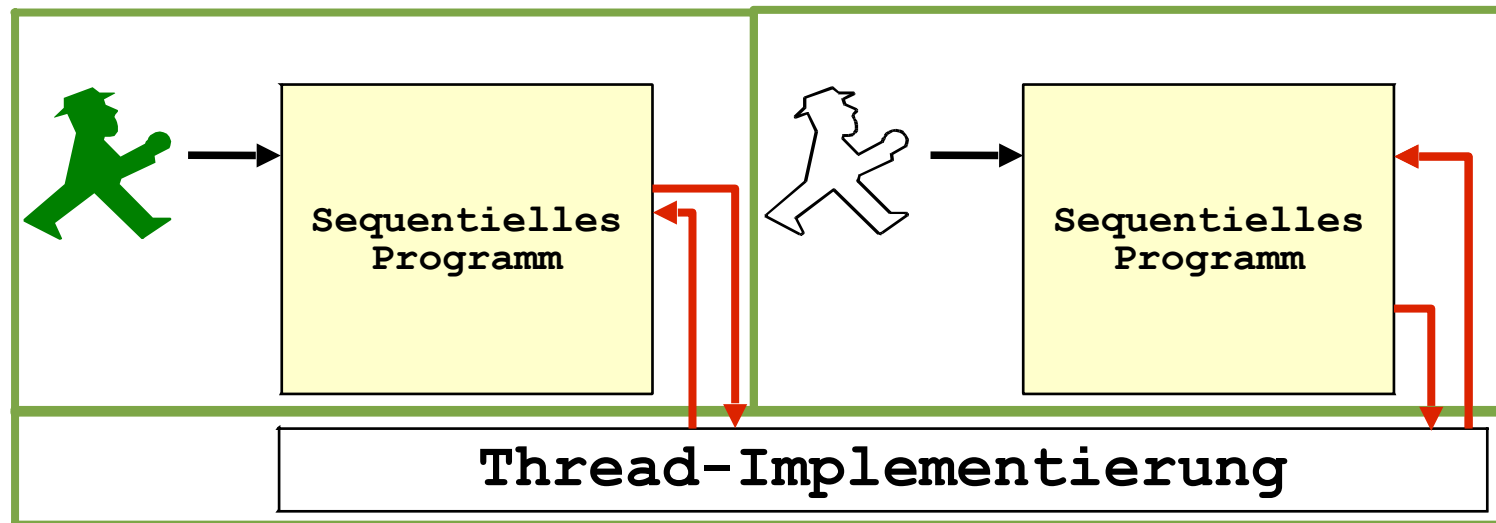
Kooperative vs. preemptive Umschaltung

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“



Beispiel

Langlaufender Thread

```
Thread1 {  
  
    Raytrace (Bildbereich[0]);  
  
    Raytrace (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) ;  
    }  
}
```

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

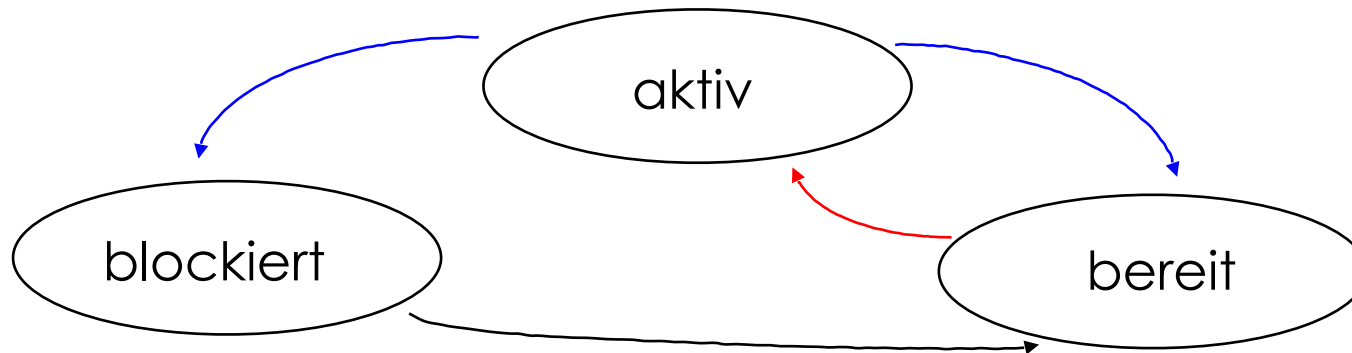
Ziel-Thread = schedule()

- 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 aufwändig, 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.

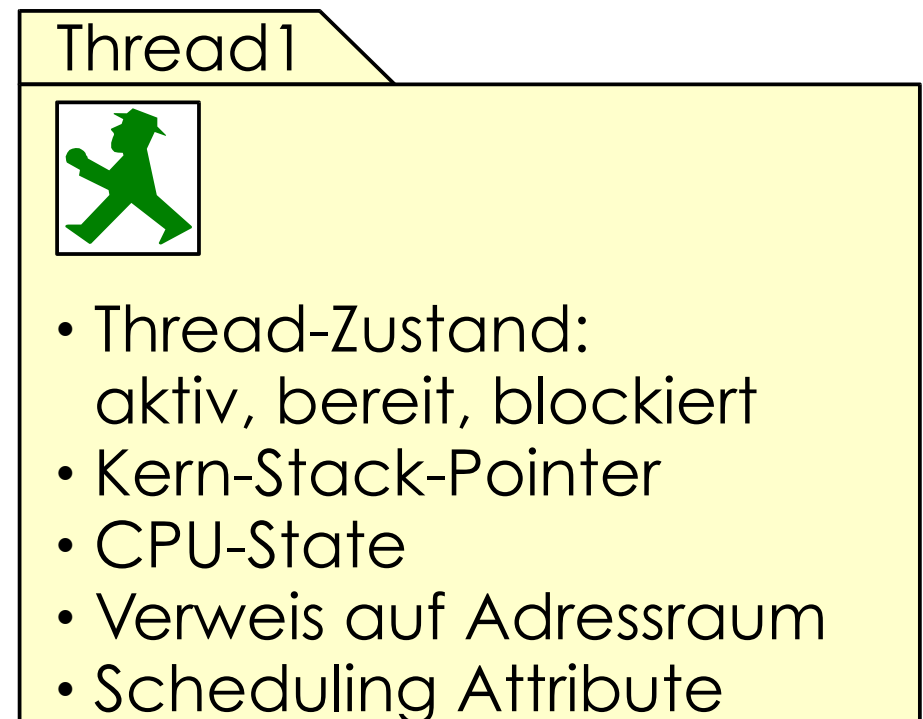
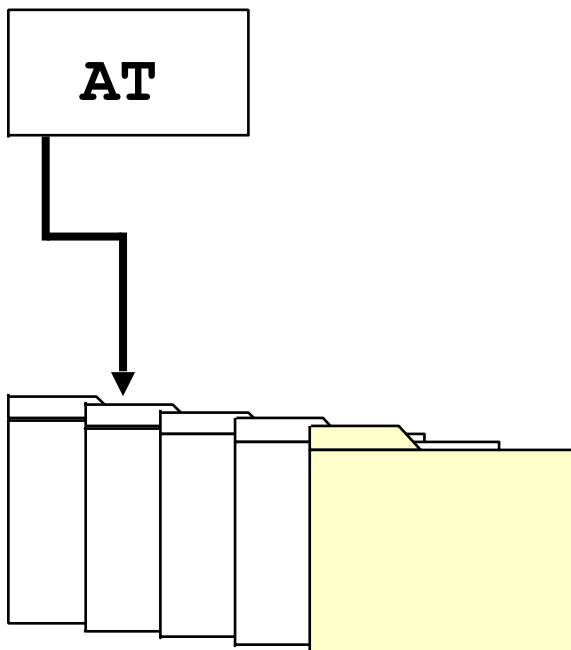
Betriebsmittel eines Threads

- 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) – im Speicher**
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



Ein paar Bezeichner

Im Folgenden:

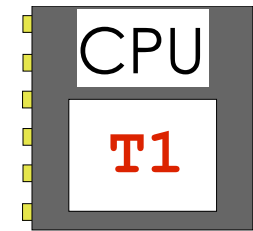
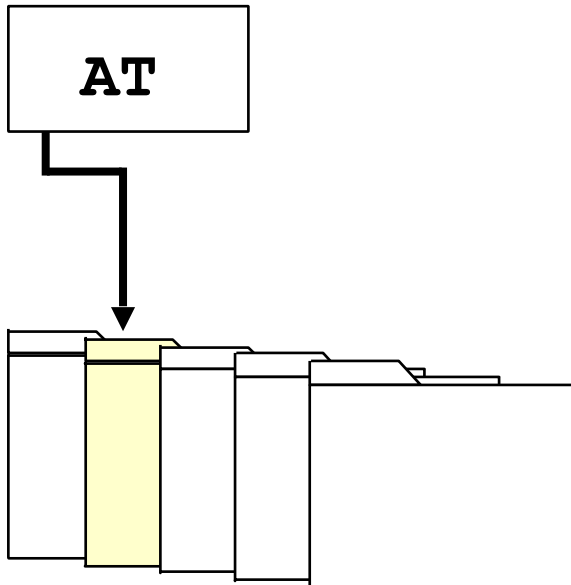
- **TCBTAB[A]** Eintrag in der TCB-Tabelle für Thread A
- **AT** aktiver Thread
- **ZT** Ziel-Thread
- **TCBTAB[AT]** „aktueller TCB“

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

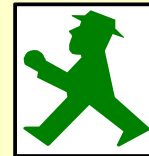
- **SP** Stack Pointer – Zeiger an die aktuelle Position im Stack
- **PC** Program Counter –
Zeiger auf den nächsten auszuführenden Befehl

Thread-Umschaltung

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



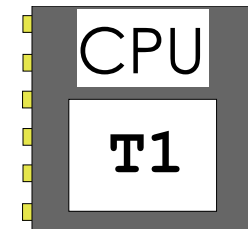
Thread1



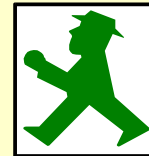
- Thread-Zustand
- Kern-Stack-Pointer
- **CPU-State**
- Verweis auf Adressraum
- Scheduling Attribute

Thread-Umschaltung

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

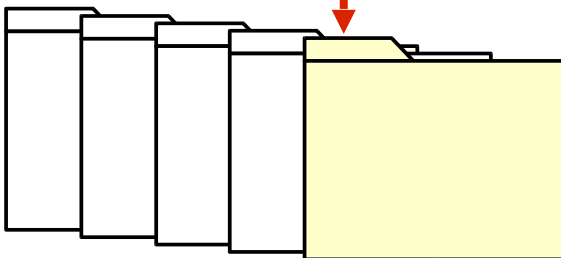


Thread4



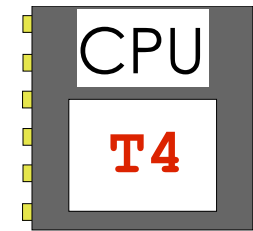
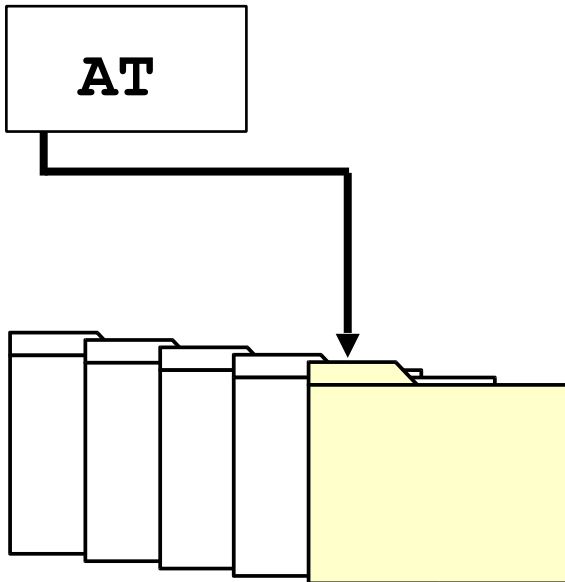
- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

AT

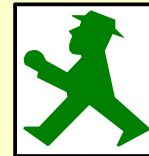


Thread-Umschaltung

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



Thread4



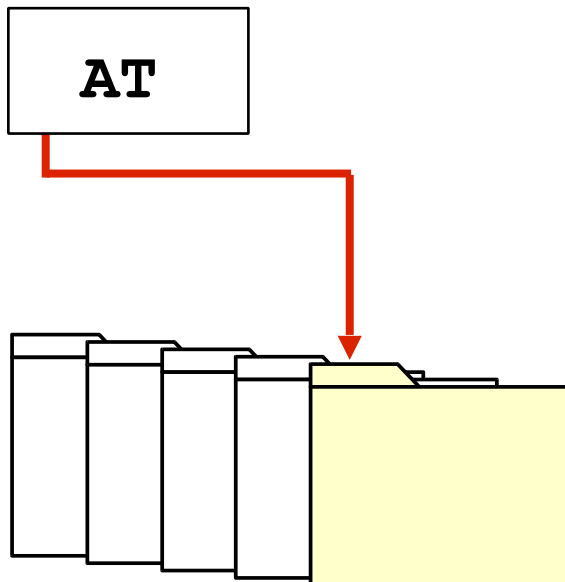
- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

Thread-Erzeugung

```
switch_to(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 später in der Vorlesung

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

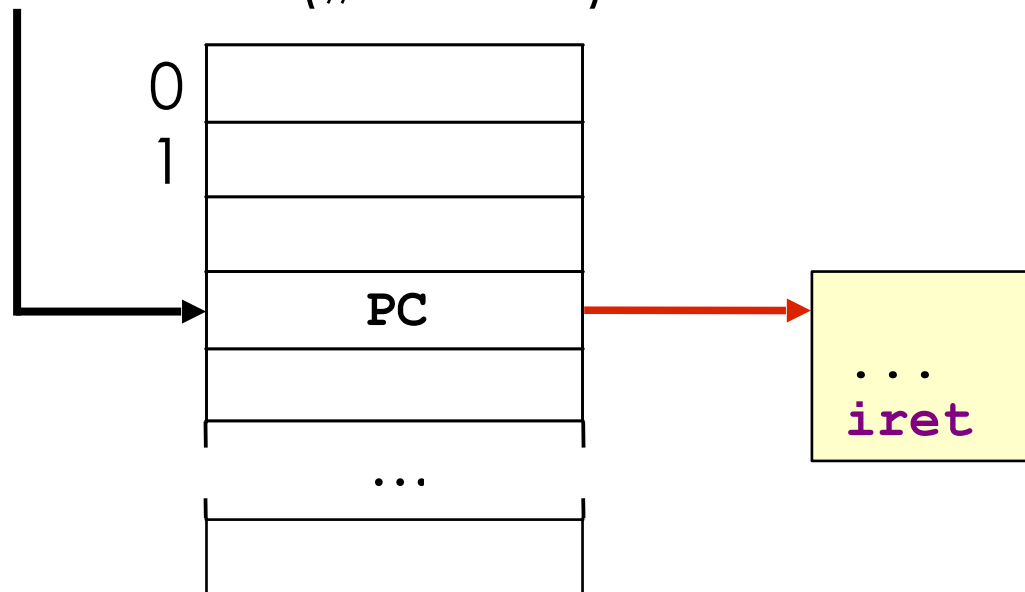
- Gerätesteuerung (Controller) löst Unterbrechung aus
- CPU erkennt Anliegen von Interrupts zwischen Befehlen
- CPU schaltet auf Kern-Modus und Kern-Stack des aktiven Threads um und rettet dorthin:
User-Stack-Pointer, User-PC, User-Flags, ...
- CPU lädt Kern-PC aus IDT (-> nächste Folie)
- ➡ Fortsetzung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

Unterbrechungsnummer („vector“)



Mehr zu Unterbrechungen (2)

Erfordernisse für Unterbrechungen

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

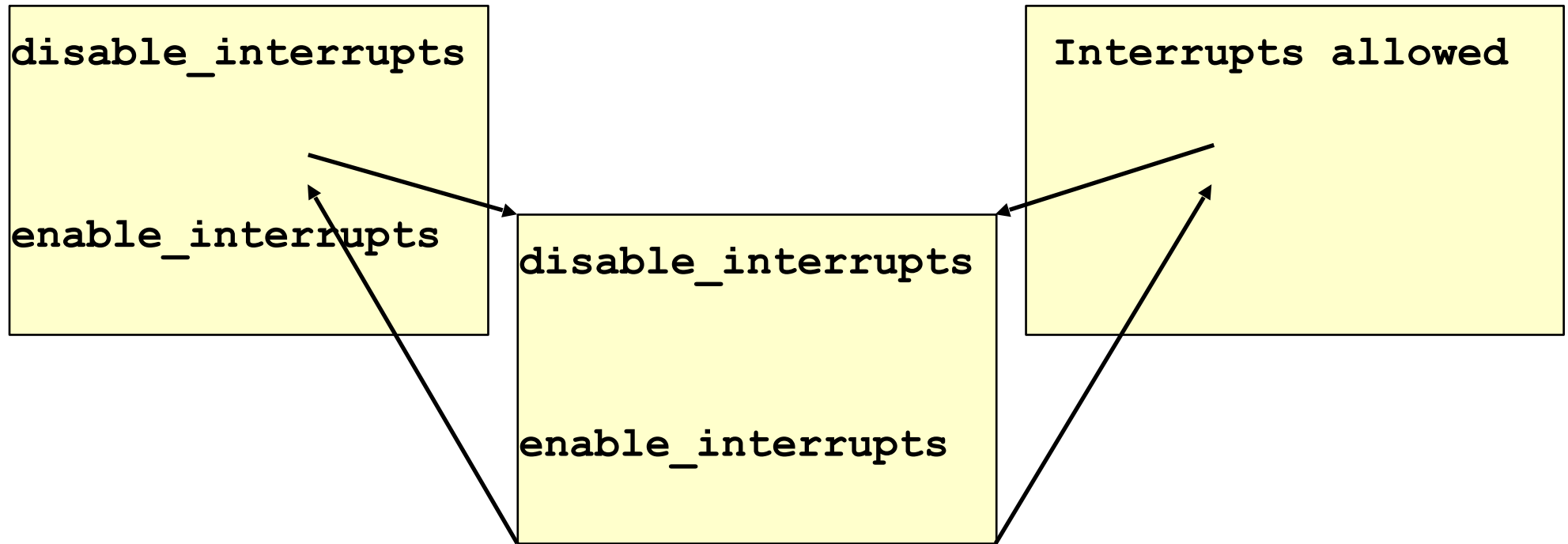
Steuerung: Flag in Prozessor-Steuer-Register (x86: „flags“)

special instructions:

cli	„clear interrupt“	disallow interrupts
sti	„set interrupt“	allow interrupts

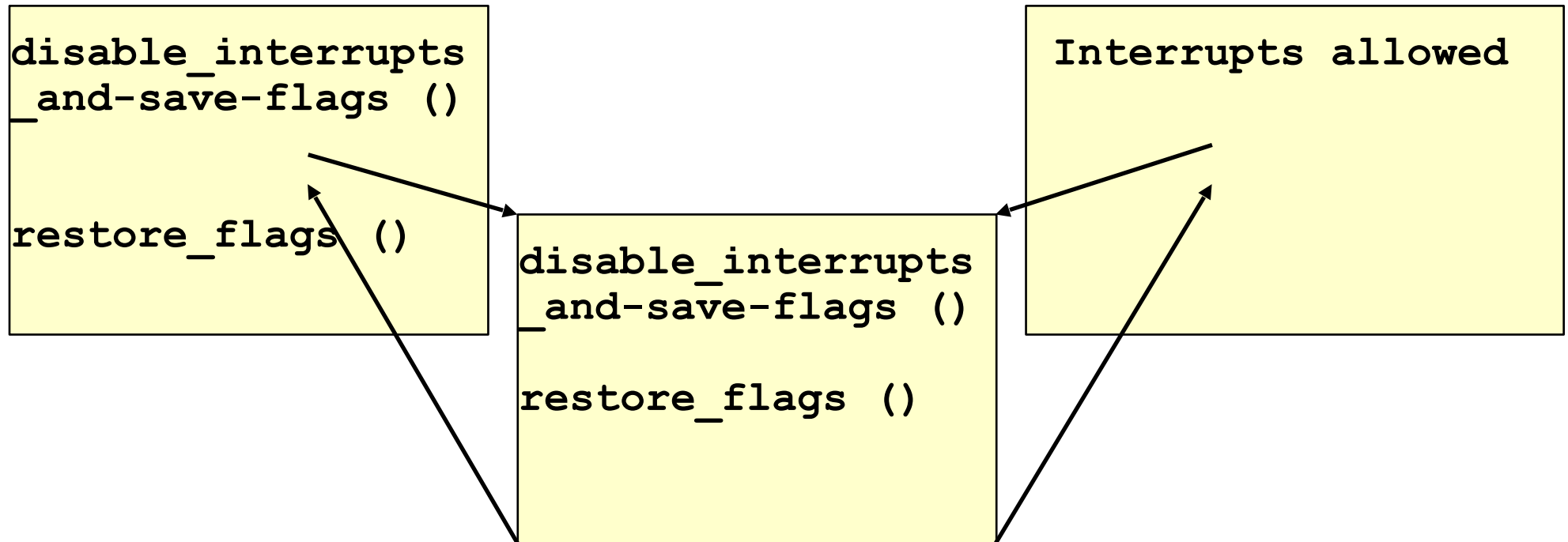
auch per load/store Instruktionen: push_flags/pop_flags

Sperren von Unterbrechungen (2)



```
disable_interrupts () {  
    cli    //löscht Interrupt-Enabled-Flag in Flags  
}  
enable_interrupts () {  
    sti    //setzt Interrupt-Enabled-Flag in Flags  
}
```

Sperren von Unterbrechungen (3)



```
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: BS-Kern sorgfältig konstruiert

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“
- Notwendig: mindestens zwei unterschiedlich privilegierte Modi

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

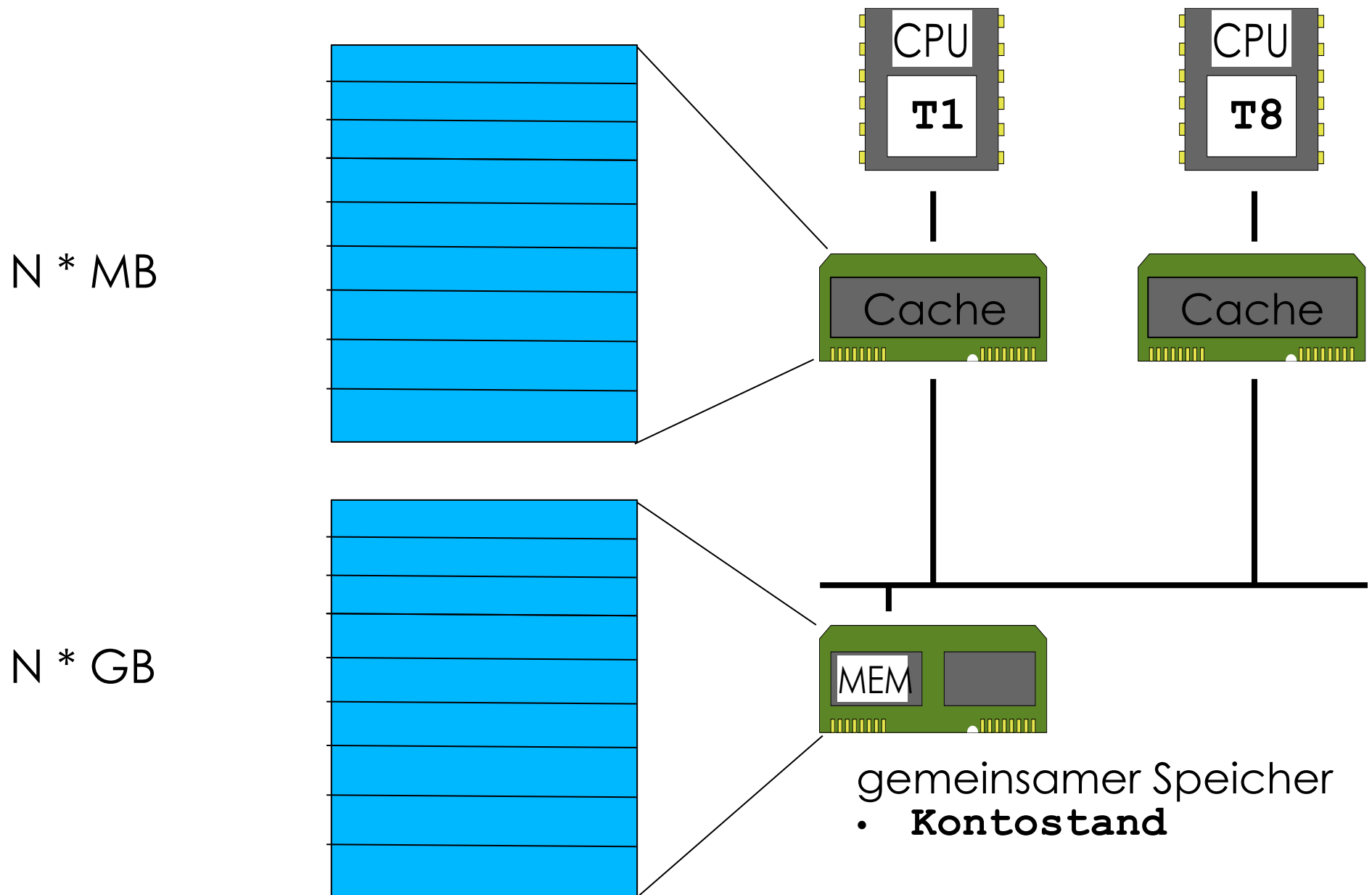
Nutzung gemeinsamen Speichers

- race conditions
- Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluss
und dessen Durchsetzung

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

Atomare Instruktionen

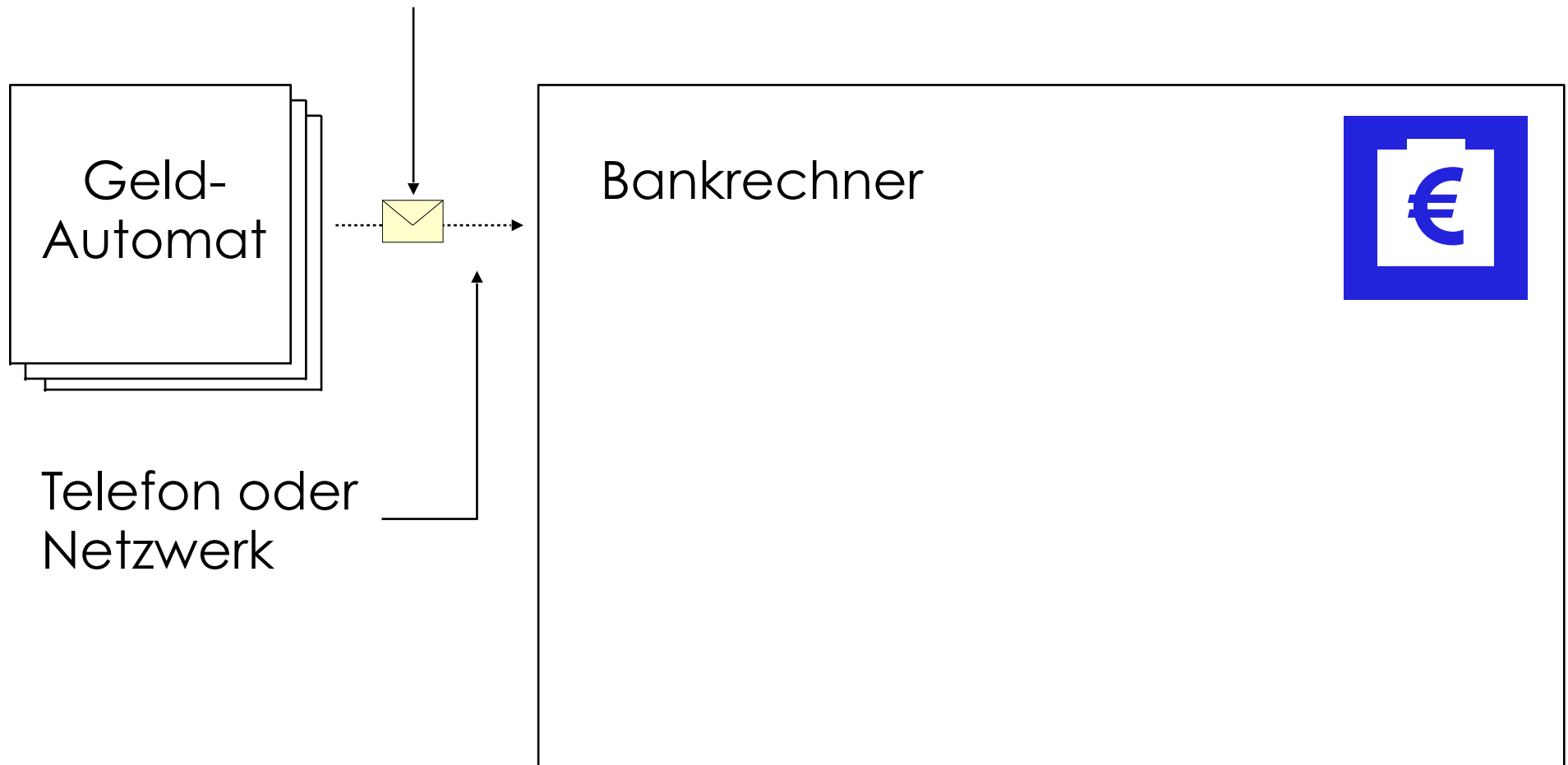


Grundsatz

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

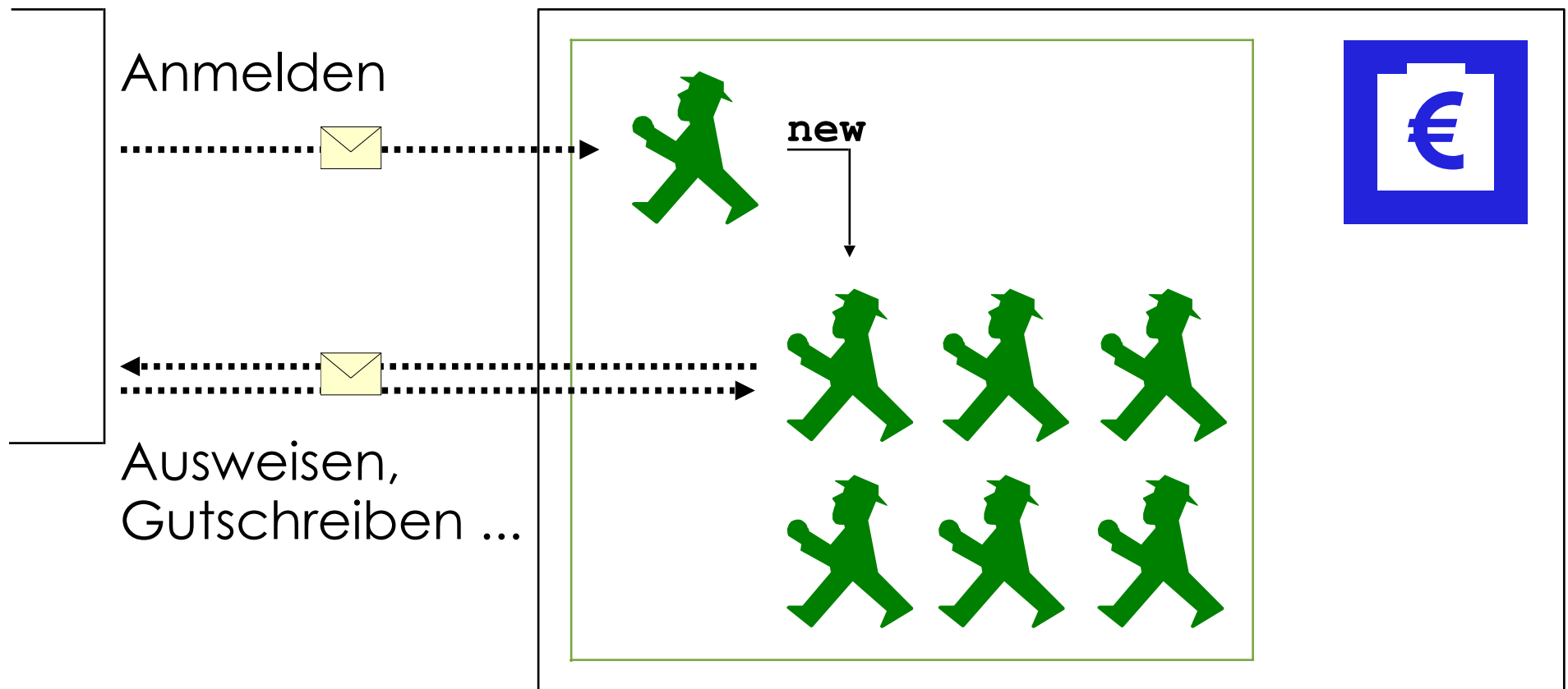
Beispiel Geldautomat

Anmelden, ausweisen, abheben, einzahlen, ...



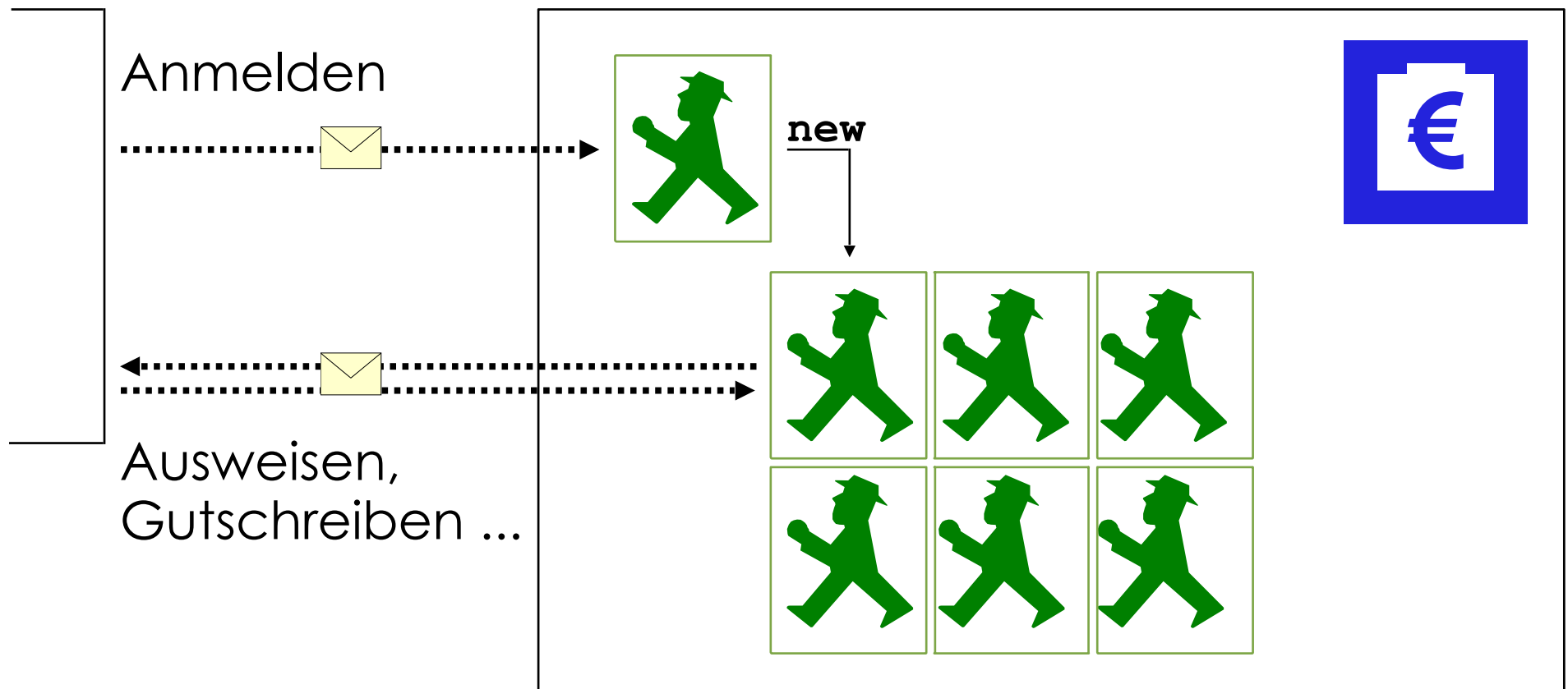
(grob nach Nichols, Buttlar, Farell)

Thread-Struktur



Erzeuge Arbeits-Thread

Oder mit Prozessen



Erzeuge Arbeits-Prozess

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 T 1	Erfolg T 2
1	19	True	False
2			
3			
4			
5			

Variante 2

```
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 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
```

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**

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

Ein globaler Kritischer Abschnitt (1)

Globale Daten

```
lock();
```

```
    Arbeite mit globalen Daten
```

```
unlock();
```

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

Ein globaler Kritischer Abschnitt (2)

```
int Kontostand;

bool abbuchen(int Betrag) {

    lock();

    if (Betrag <= Kontostand) {

        Kontostand -= Betrag;

        unlock();
        return true;

    } else {

        unlock();
        return false;

    }
}
```

Kritischer Abschnitt (3)

```
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;

    }
}
```

Kritischer Abschnitt (4)

```
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

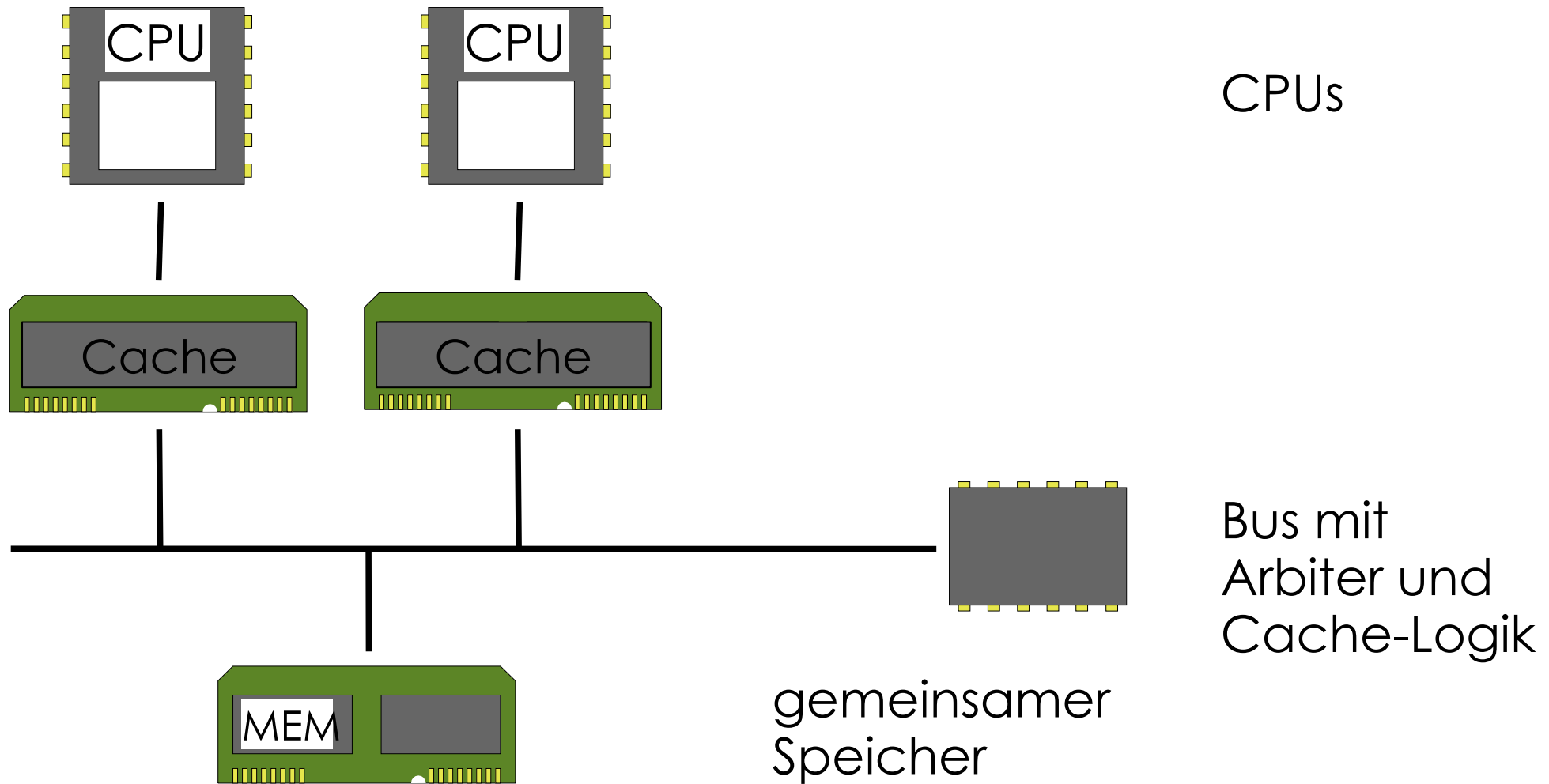
Lösungsversuch 1 – mit Unterbrechungssperre

```
T1: ...  
  pushf  
  cli  
  
  ld    R, Kontostand  
  sub   R, Betrag  
  sto   R, Kontostand  
  
  popf
```

← Globales Lock

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

Einfaches Modell einer Dual-CPU Maschine

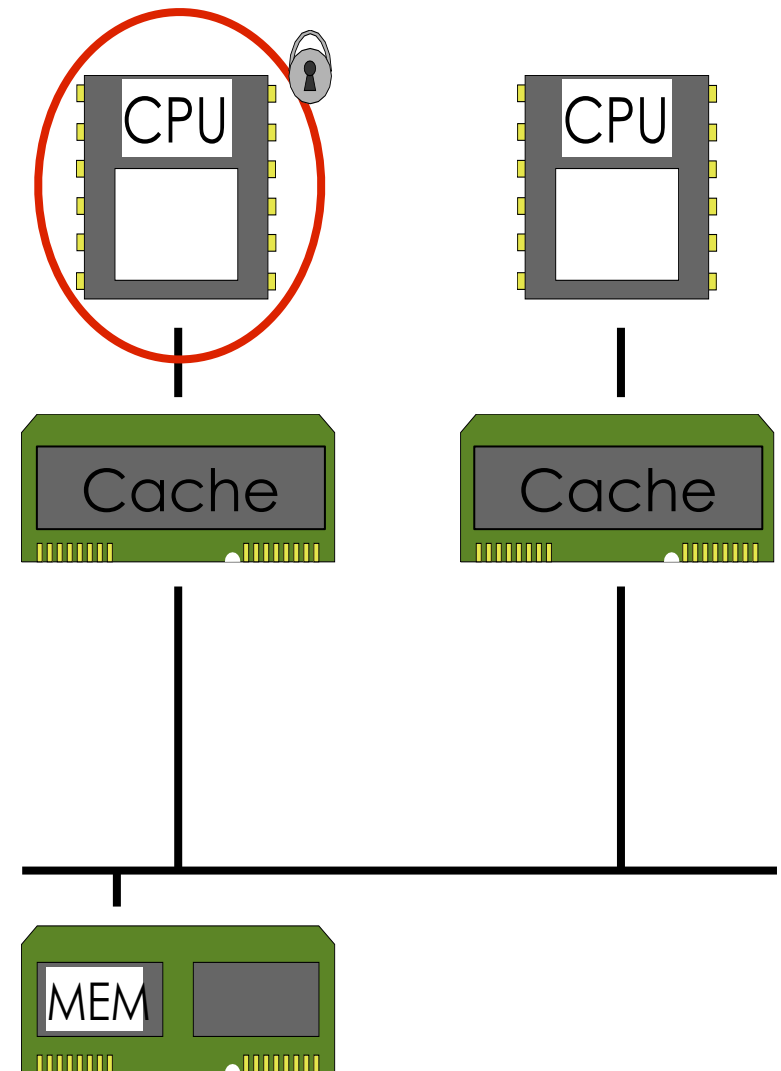


Lösungsversuch 1 – mit Unterbrechungssperre

```
T1: ...  
  pushf  
  cli  
  
  ld    R, Kontostand  
  sub   R, Betrag  
  sto   R, Kontostand  
  
  popf
```

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

→ **genügt im MP-Fall nicht!**



gemeinsamer Speicher
• **Kontostand**

Lösungsversuch 2 – KA mit einer Instruktion

Ti: ...

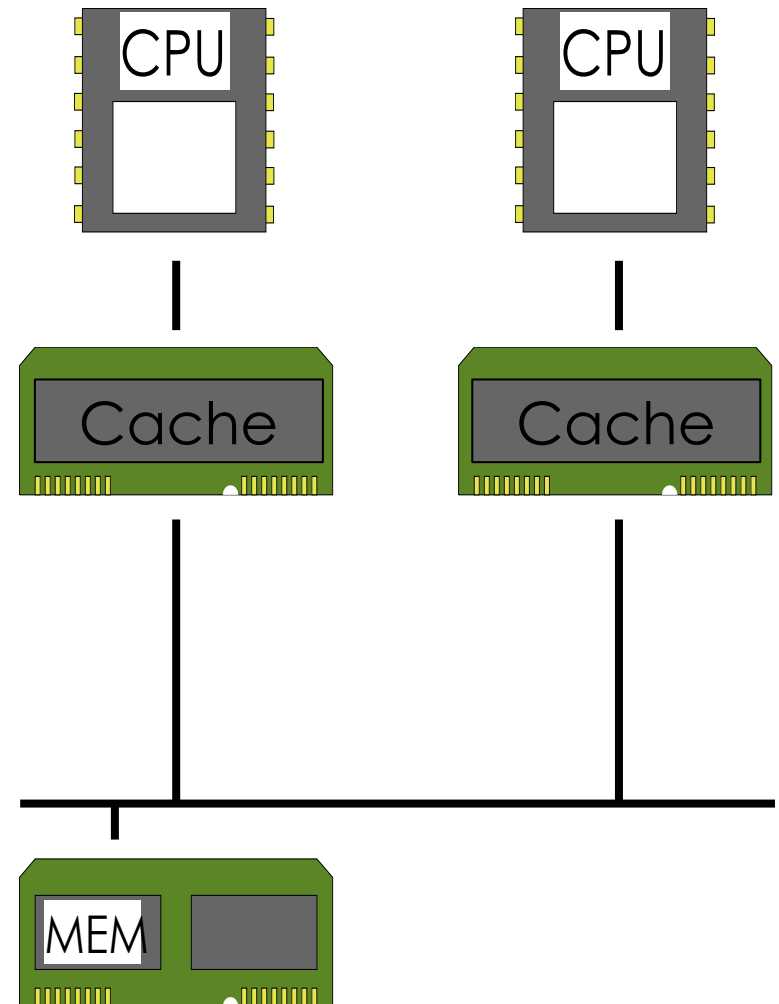
ld R, Betrag

sub Kontostand, R

sub ist nicht unterbrechbar

aber: funktioniert im
MP-Fall auch nicht!

Begründung:
folgende Folien



gemeinsamer Speicher
• **Kontostand**

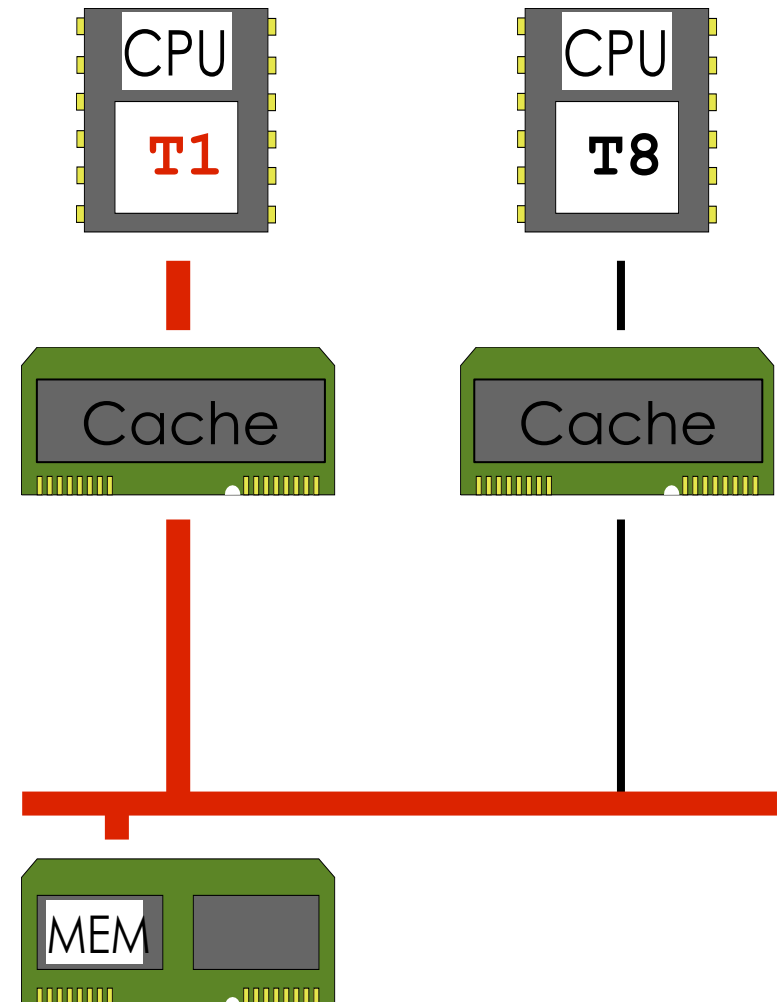
Lösungsversuch 2 – KA mit einer Instruktion

Ti: ...

ld R, Betrag

➔ **μload** TempR, Kontostand
μsub TempR, R
μstore TempR, Kontostand

- Die Instruktion **sub Kontostand, R** hat zwei Buszyklen
- Einzelne Buszyklen sind die atomare Einheit



gemeinsamer Speicher
• **Kontostand**

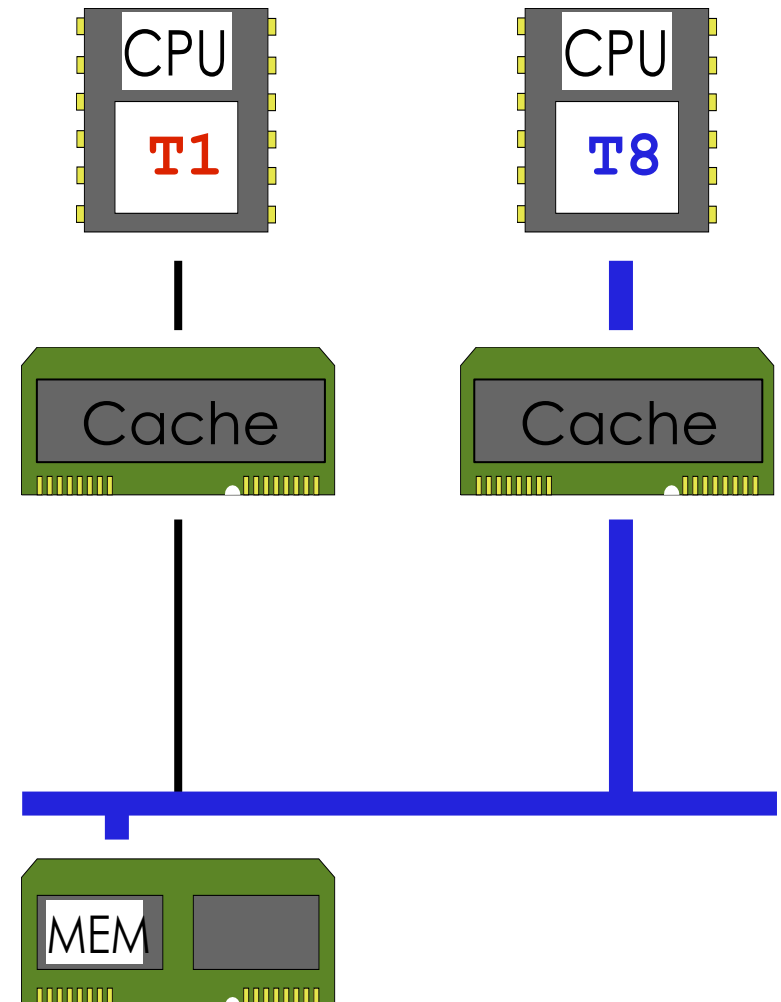
Lösungsversuch 2 – KA mit einer Instruktion

Ti: ...

ld R, Betrag

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

- Die Instruktion **sub Kontostand, R** hat zwei Buszyklen
 - Einzelne Buszyklen sind die atomare Einheit
- funktioniert so nicht !



gemeinsamer Speicher
• **Kontostand**

Lösungsversuch 2 – KA mit einer Instruktion

Ti: ...

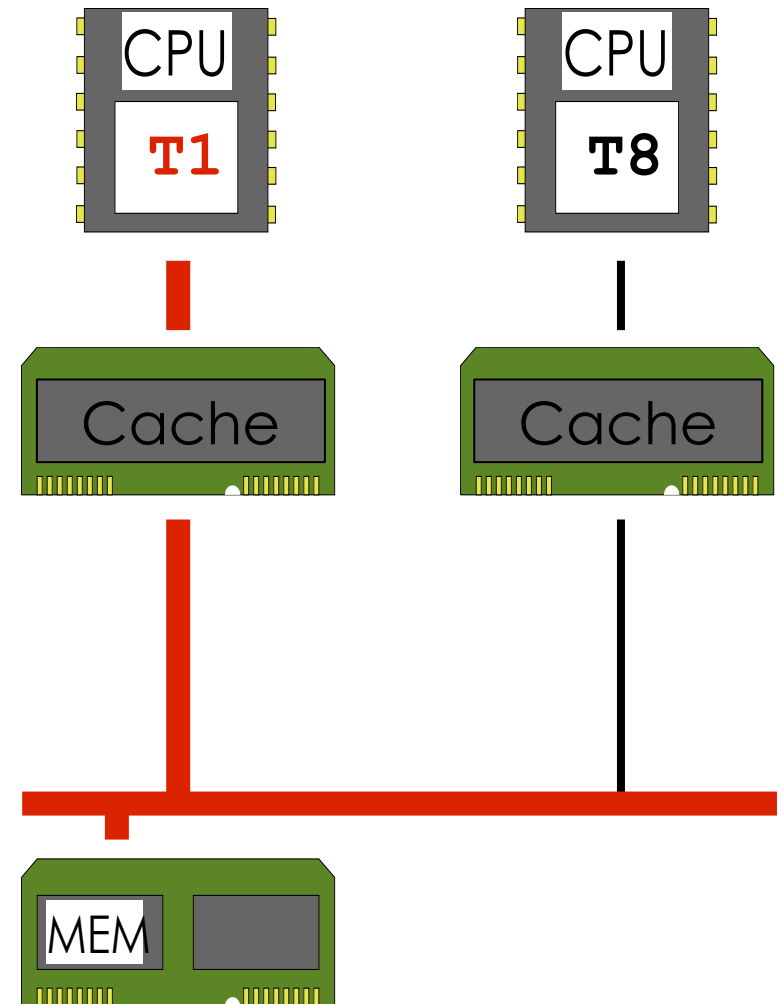
ld R, Betrag

μload TempR, Kontostand

μsub TempR, R

➔ **μstore** TempR, Kontostand

- Die Instruktion **sub Kontostand, R** hat zwei Buszyklen
 - Einzelne Buszyklen sind die atomare Einheit
- ➔ funktioniert so nicht !



gemeinsamer Speicher
• **Kontostand**

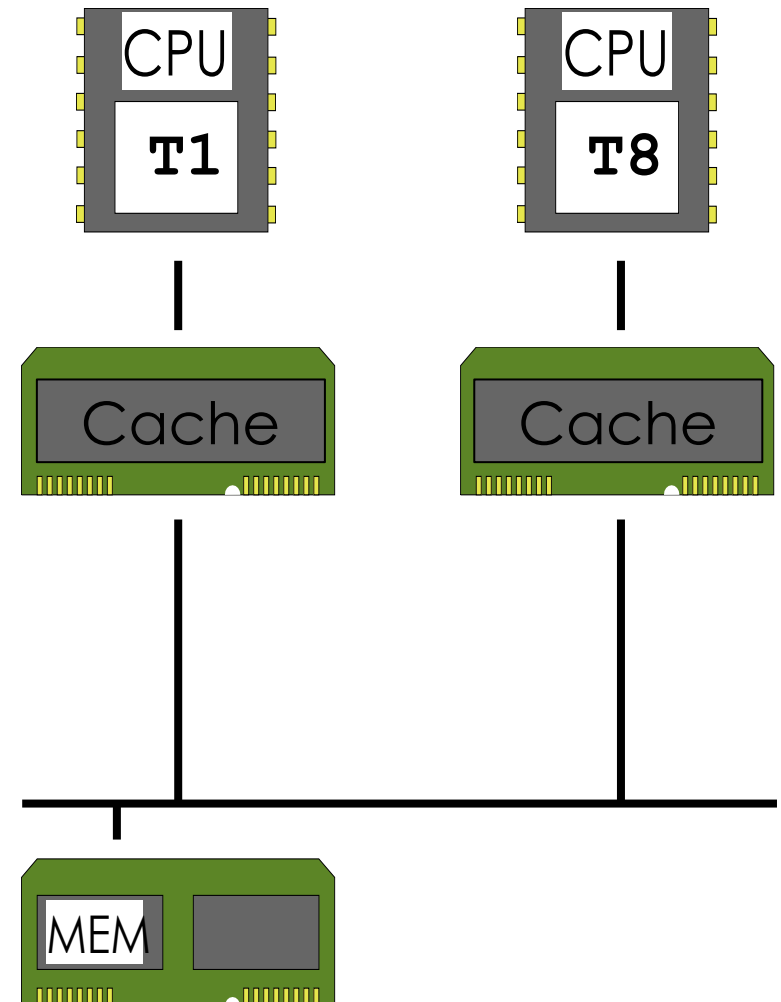
Lösungsversuch 3 – atomare Instruktion

Ti: ...

ld R, Betrag

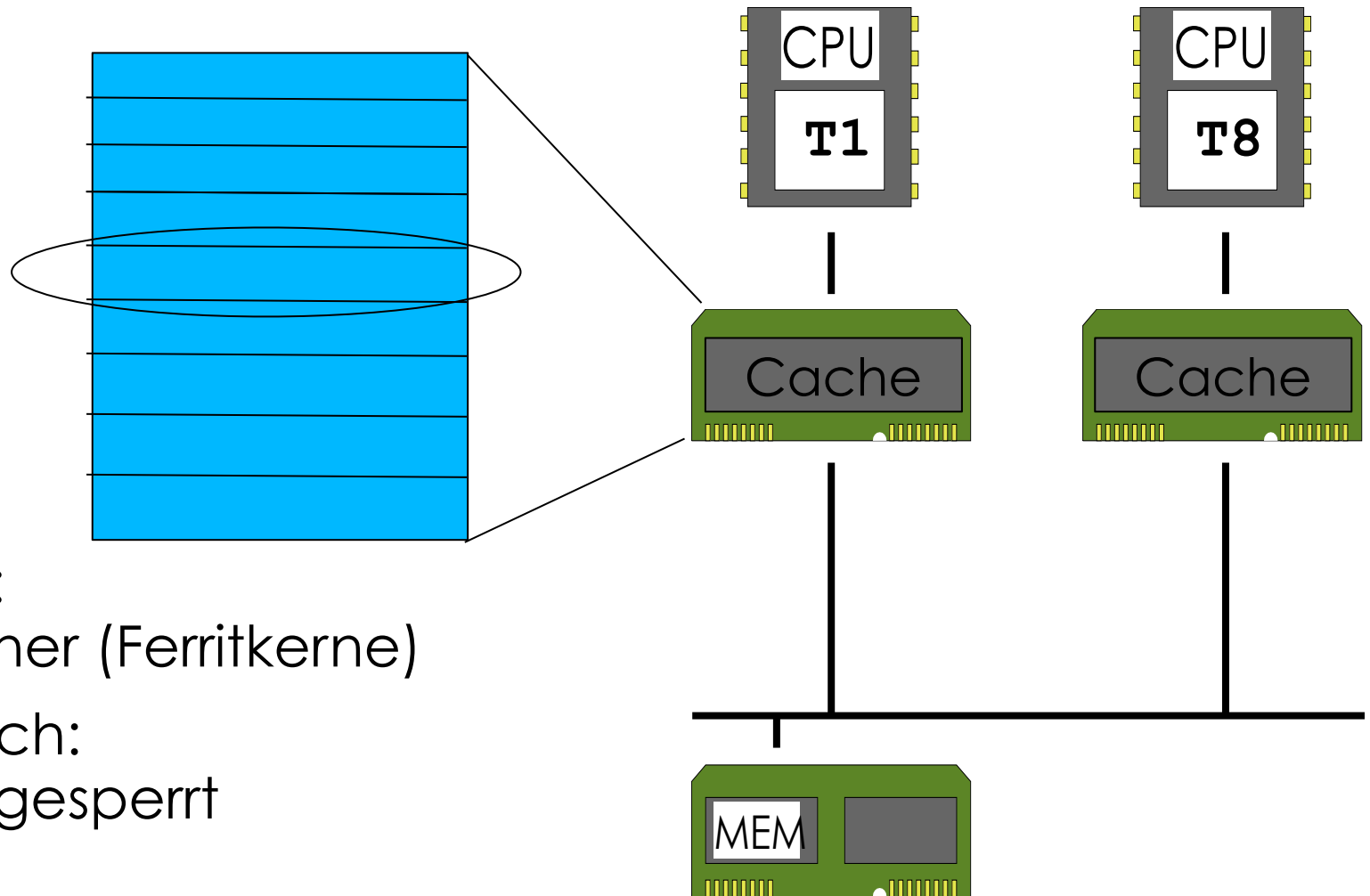
atomic_sub Kontostand, R

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



gemeinsamer Speicher
• **Kontostand**

Atomare Instruktionen



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

gemeinsamer Speicher
• **Kontostand**

Kritischer Abschnitt: Anforderungen

Bedingungen/Anforderungen

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

→ Wechselseitiger Ausschluss

Sicherheit

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

Lebendigkeit

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

Im Folgenden verschiedene Lösungsansätze ...

Implementierung mittels Unterbrechungssperre

Vorteile

- einfach und effizient

Nachteile

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

Konsequenz

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

```
lock() :  
    pushf  
    cli          //disable irqs  
  
unlock() :  
    popf         //restore
```

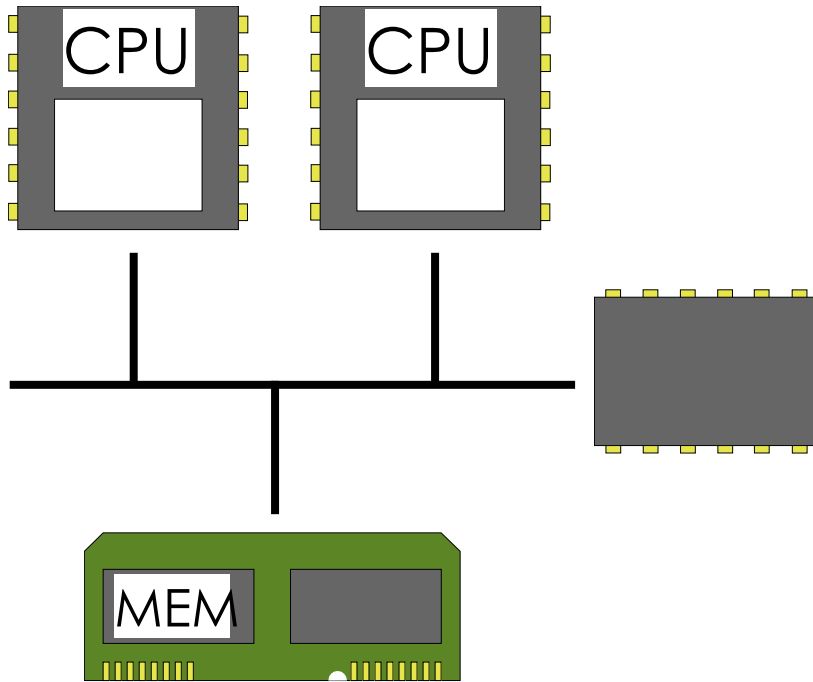
Implementierung mittels Sperrvariable

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

```
int Lock = 0;  
//Lock == 0: frei  
//Lock == 1: gesperrt  
  
void lock(int *lock) {  
    while (*lock != 0);  
    //busy waiting  
    *Lock = 1;  
}  
  
void unlock(int *lock) {  
    *lock = 0;  
}
```

RA: unteilbare Operationen der HW

Mehrprozessormaschine



CPU's

Bus/Cache
Logik

gemeinsamer
Speicher

Bus/Cache - Logik
implementiert Atomarität
für eine Adresse:

- Lesen
- Schreiben
- Lesen und Schreiben

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

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


Implementierung mit HW Unterstützung

Vorteile

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

Nachteile

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

Konsequenz

- geeignet für kurze KA bei kleiner CPU-Anzahl

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

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

unlock:
    mov        lock, #0
    ret
```

Implementierung für eine CPU im User-Mode

- Unterbrechungssperren:
sind nicht sicher
ein Kern-Aufruf (cli) pro
„lock“ ist zu teuer
- busy waiting belegt die
CPU
- also
Kernaufufruf nur bei
gesperrtem KA
(selten)
- jedoch race condition:
owner u. U. noch nicht
richtig gesetzt

```
pid_t owner;  
lock_t lock = 0;  
  
void lock(lock_T *lock) {  
    while(test_and_set(*lock)) {  
        ➡ Kern.Switch_to (owner);  
    }  
    ➡ owner = AT;  
    //aktueller Thread  
}  
  
void unlock(lock_T *lock) {  
    *lock = 0;  
}
```

Alternative: Atomic 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: Atomic 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: Atomic 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, myself)) {  
  
        Kern.Switch_to (owner);  
  
    }  
}
```

```
void unlock(  
    lock_t *lock) {  
  
    *lock = 0;  
}
```

Alternative: Atomare Operation CAS

myself == 1

lock

myself == 2

```
owner = cas(lock, 0, 1)
result == 2

switch_to(2)
nicht im KA (loop case)

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

0

2

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

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

0

1

KA: kritischer Abschnitt

Wegweiser

Zusammenspiel/Kommunikation
mehrerer Threads

Nutzung gemeinsamen Speichers

- race conditions
- Kritischer Abschnitt

Die Lösung: Wechselseitiger Ausschluss
und dessen Durchsetzung

- ohne HW-Unterstützung möglich?

Zwei 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

CPU0:

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

CPU1:

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

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