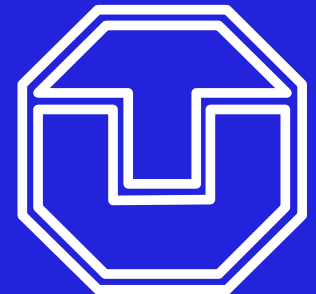


# Threads

Betriebssysteme

Hermann Härtig  
TU Dresden



# Wegweiser

Einführung und Wiederholung

Implementieren mehrerer Threads  
auf einem Rechner/Prozessor

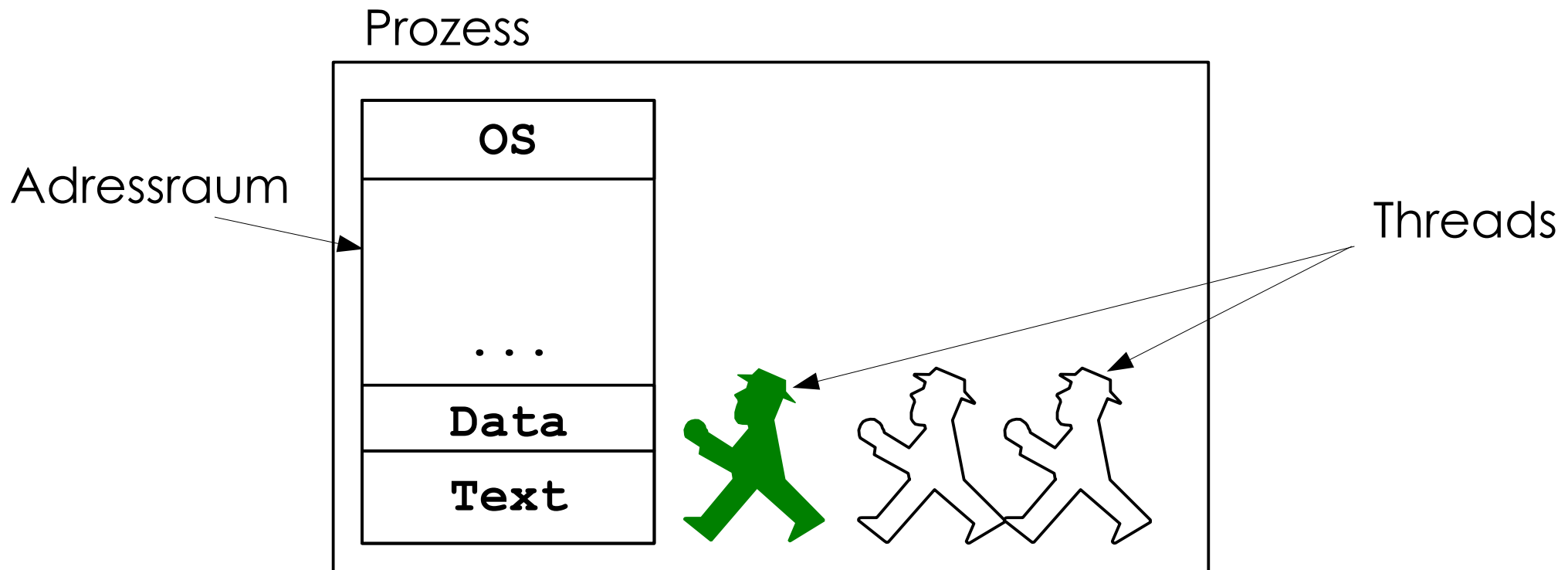
Zusammenspiel/Kommunikation  
mehrerer Threads

Scheduling

# Definition: Thread

Eine selbständige

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

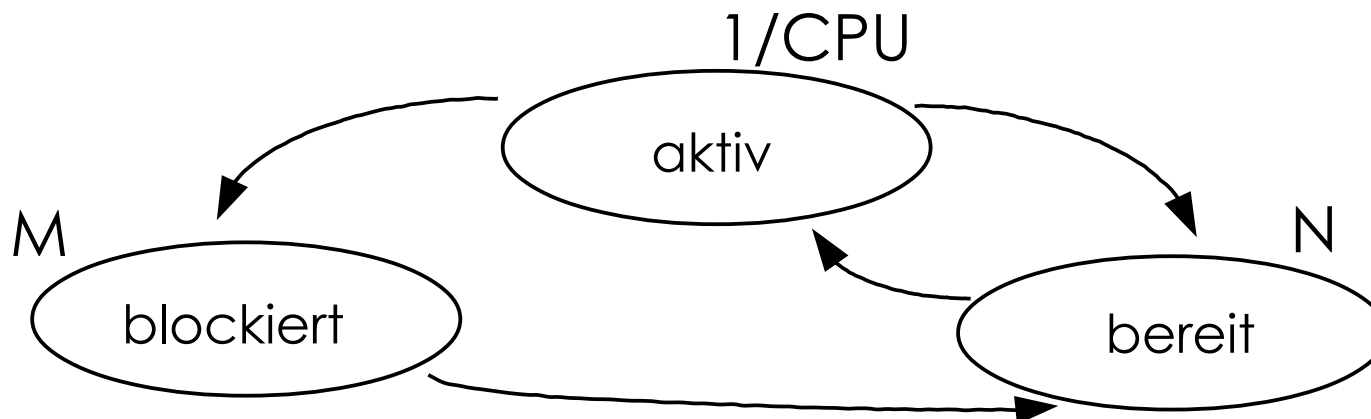


# Notationen

- $P \parallel Q$
- `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“

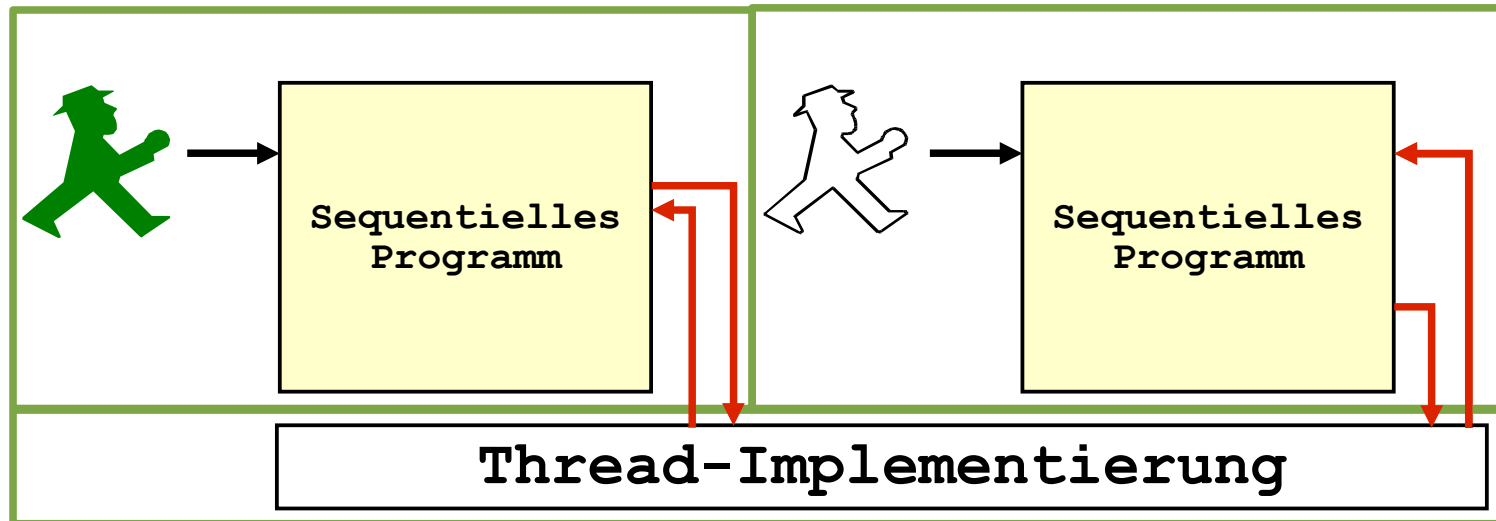
# Kooperative vs. preemptive Umschaltung

## 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

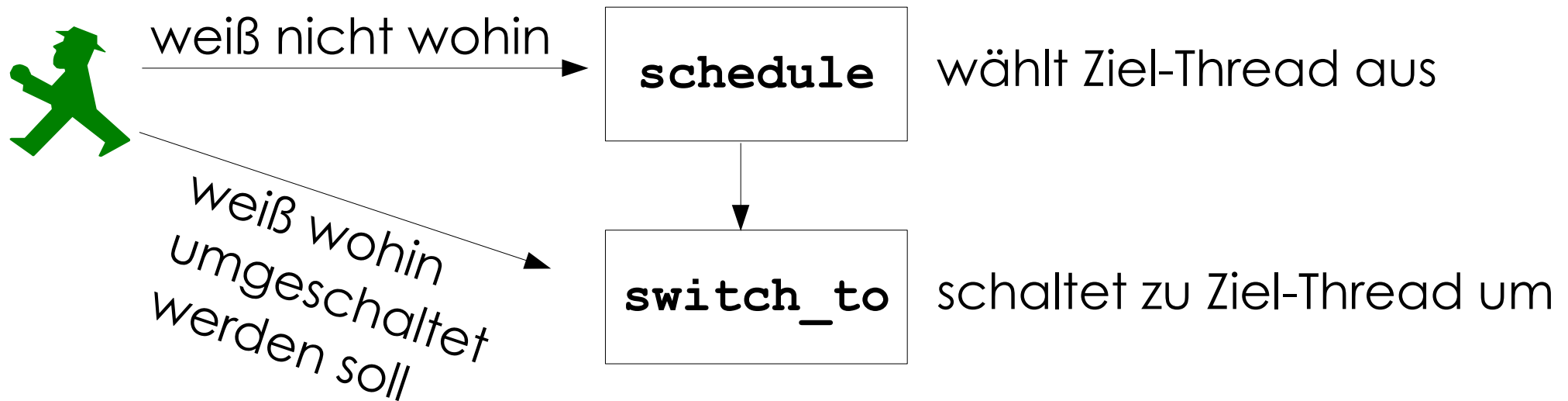
## **Ziel-Thread = schedule ()**

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

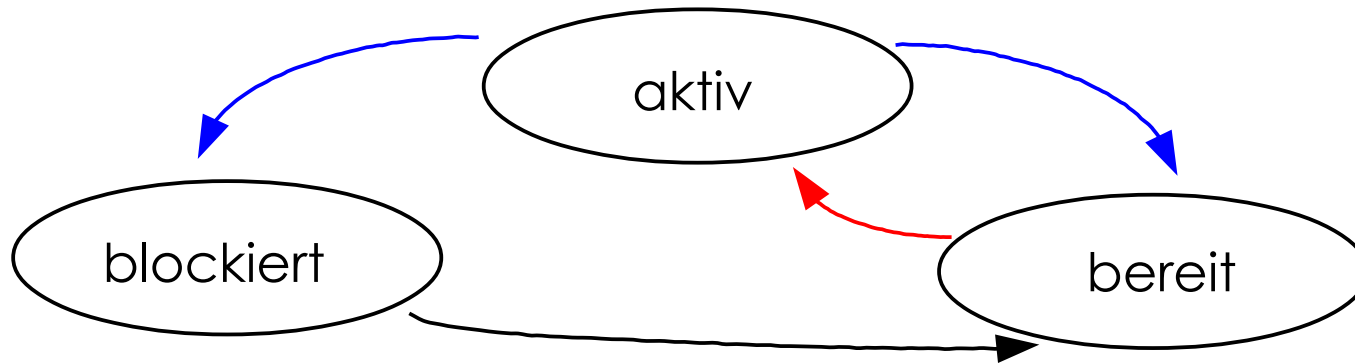
## **switch\_to (Ziel-Thread)**

- 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

# Umschaltmechanismen



# 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.

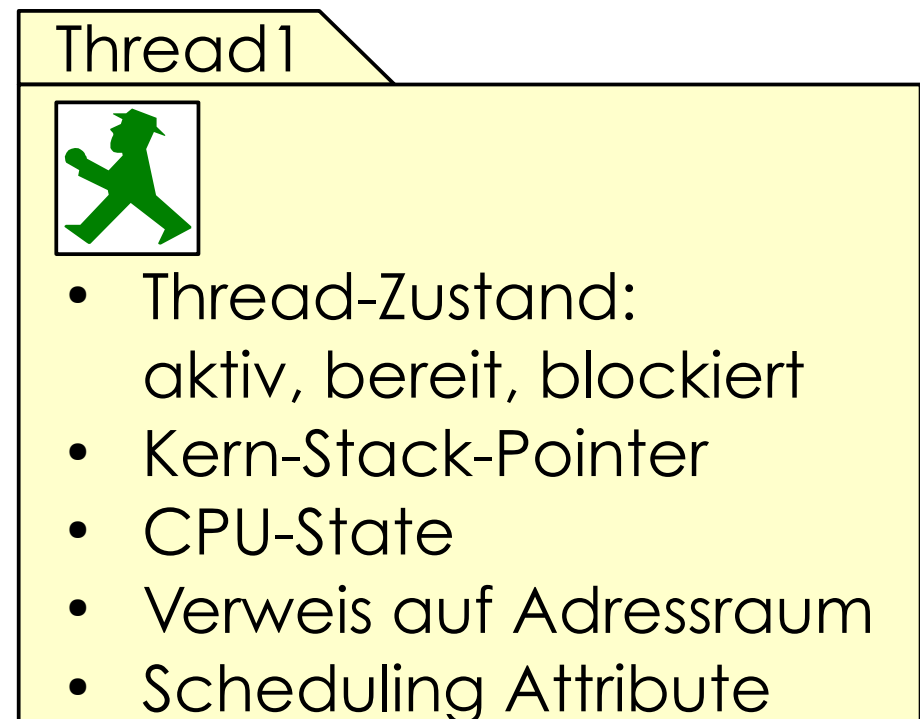
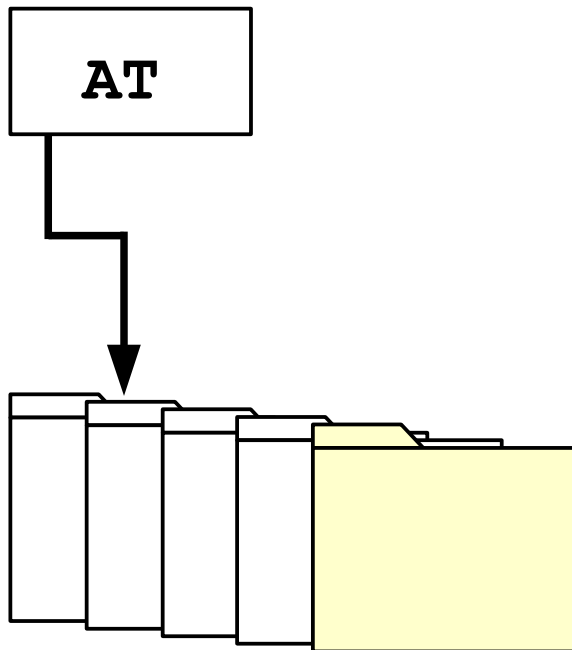
# Betriebsmittel eines Threads

- 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



# 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

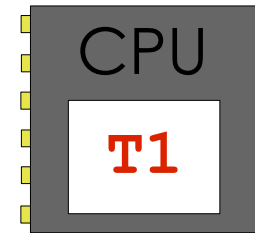
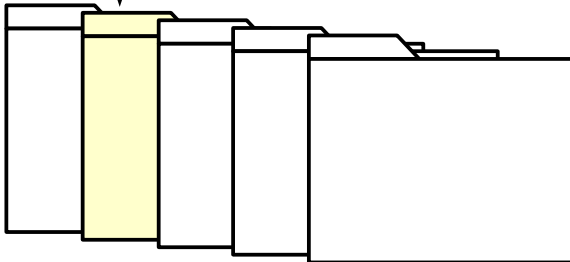
Im Folgenden:

- **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();
}
```

AT



Thread1

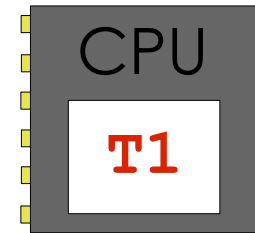
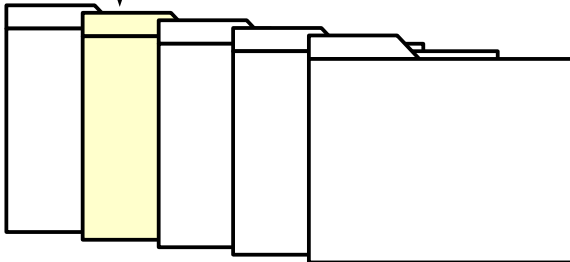


- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

# Thread-Umschaltung

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

AT



Thread1

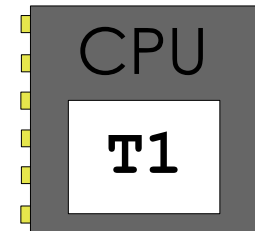
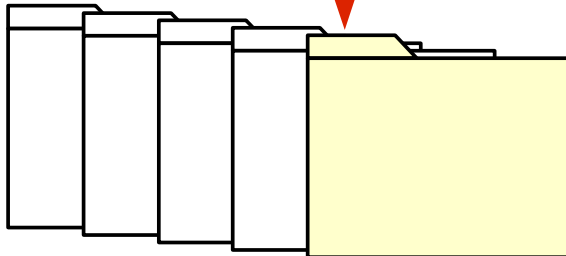


- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

# Thread-Umschaltung

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

AT



Thread4

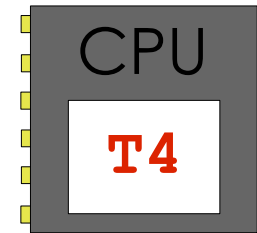
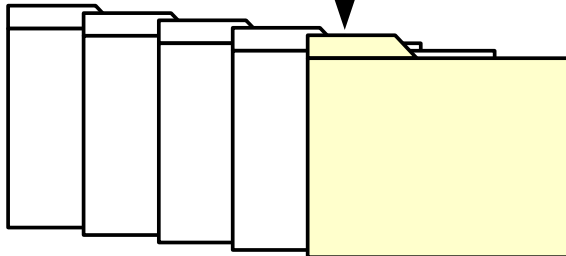


- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

# Thread-Umschaltung

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

AT



Thread4

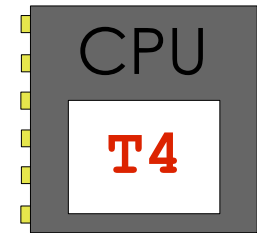
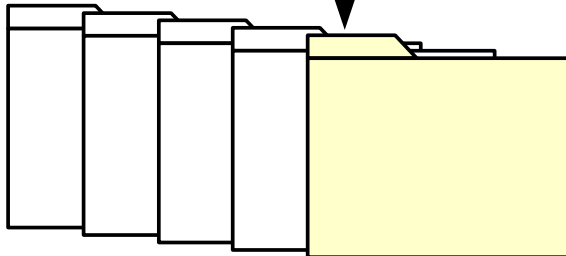


- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

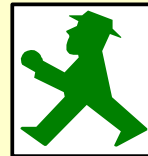
# Thread-Umschaltung

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

AT



Thread4



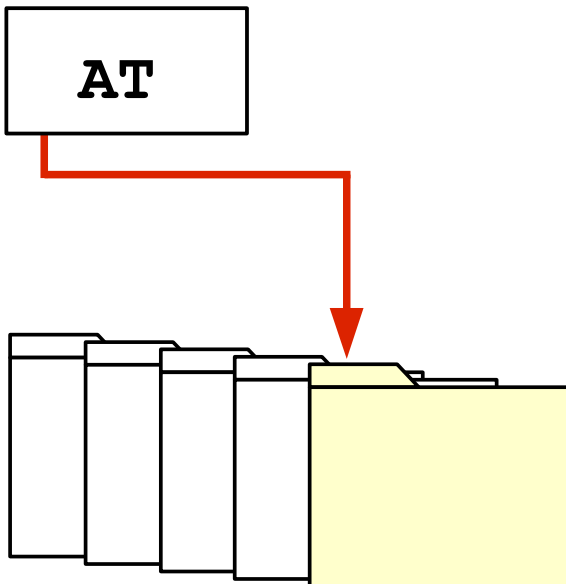
- Thread-Zustand
- Kern-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling Attribute

# 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

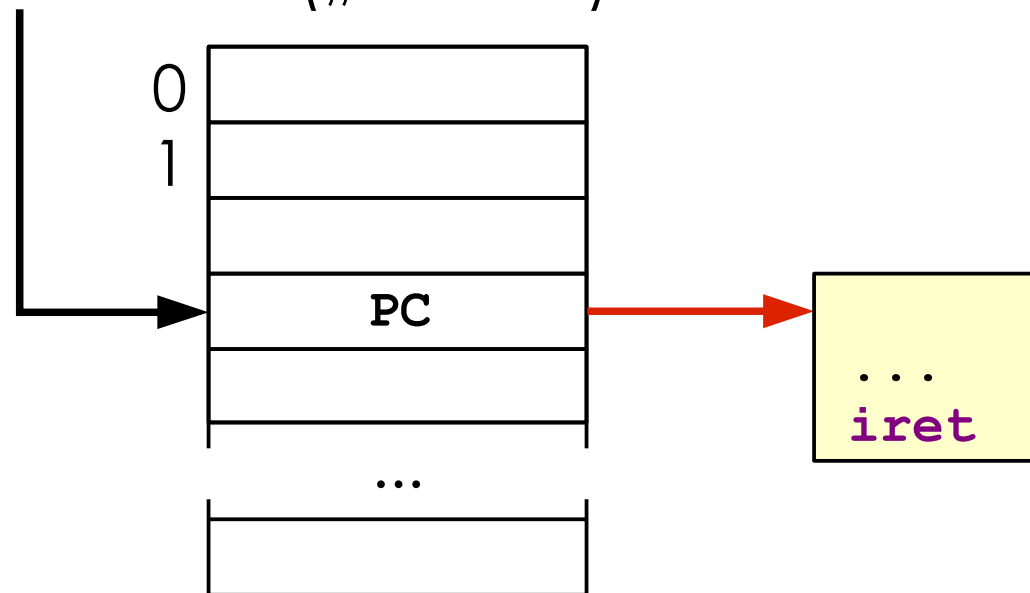
- Gerätesteuerung (Controller) löst Unterbrechung aus
- CPU erkennt Anliegen von Interrupts zwischen Befehlen
- CPU schaltet auf Kern-Modus und Kern-Stack des aktiven Threads um und rettet dorthin:  
User-Stack-Pointer, User-PC, User-Flags, ...
- CPU lädt Kern-PC aus IDT (-> nächste Folie)
- ➔ Fortsetzung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

Unterbrechungsnummer („vector“)



# Mehr zu Unterbrechungen (2)

## Erfordernisse für Unterbrechungen

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

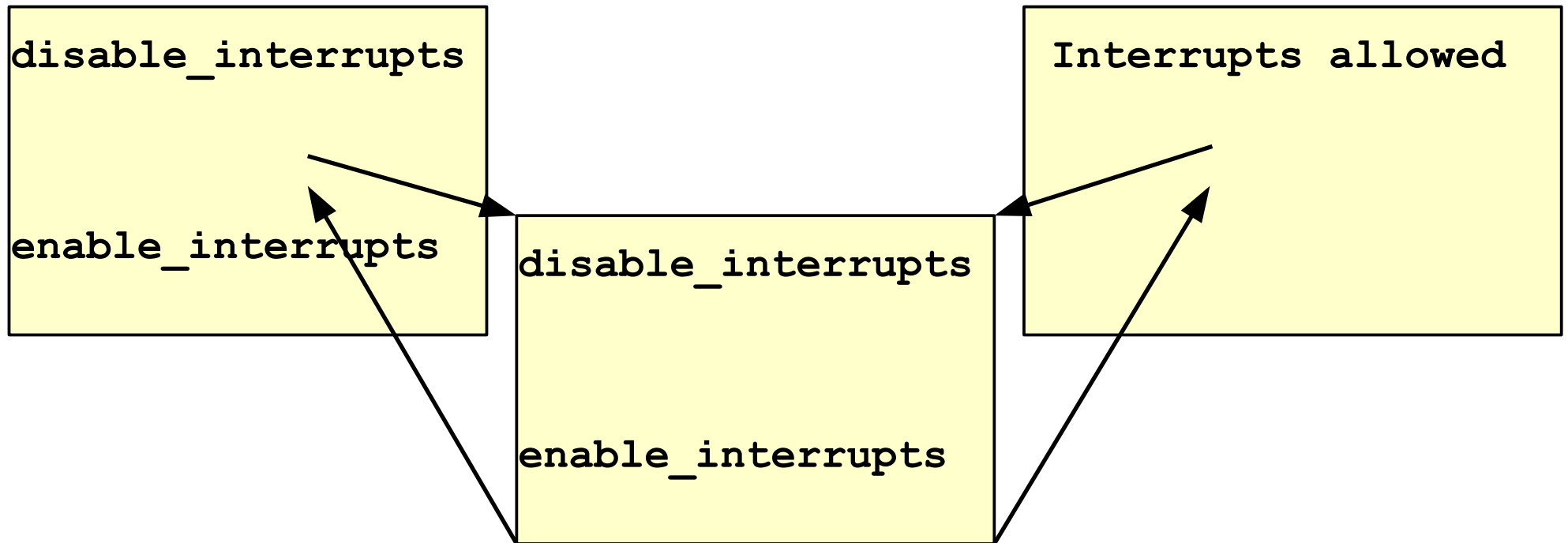
Steuerung: Flag in Prozessor-Steuer-Register (X86: „flags“)

special instructions:

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

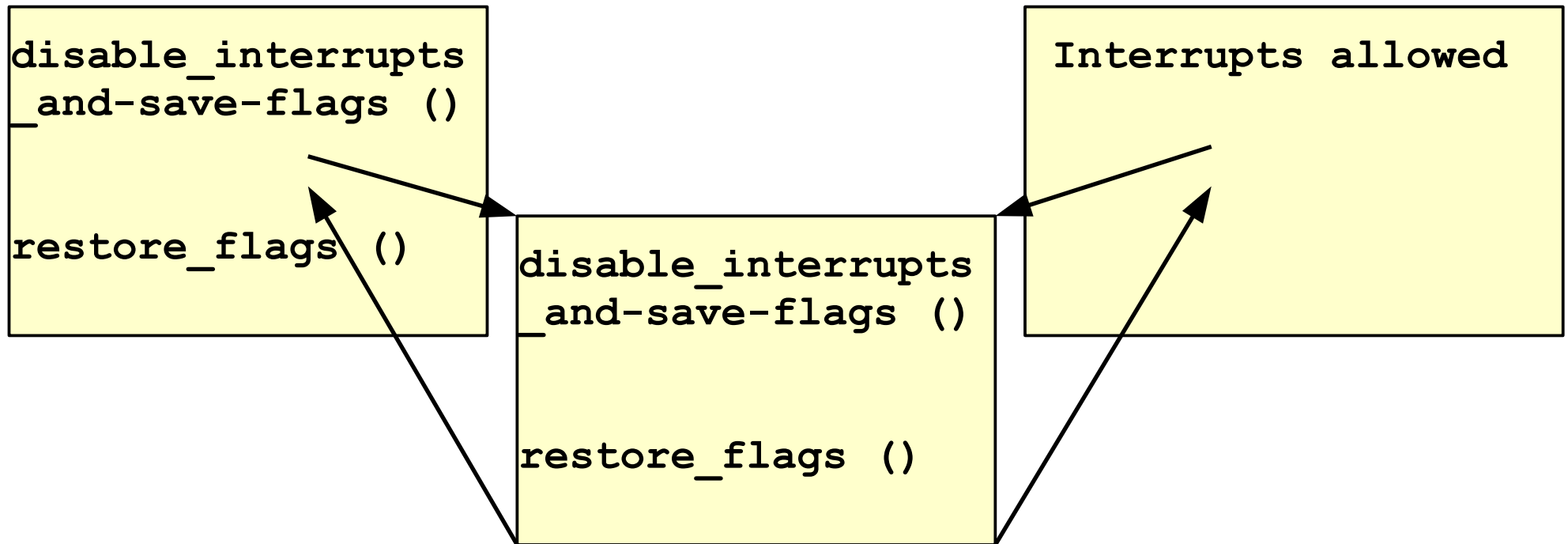
auch per load/store Instruktionen: push\_flags/pop\_flags

## 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, dass BS-Kern sorgfältig konstruiert ist

# 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

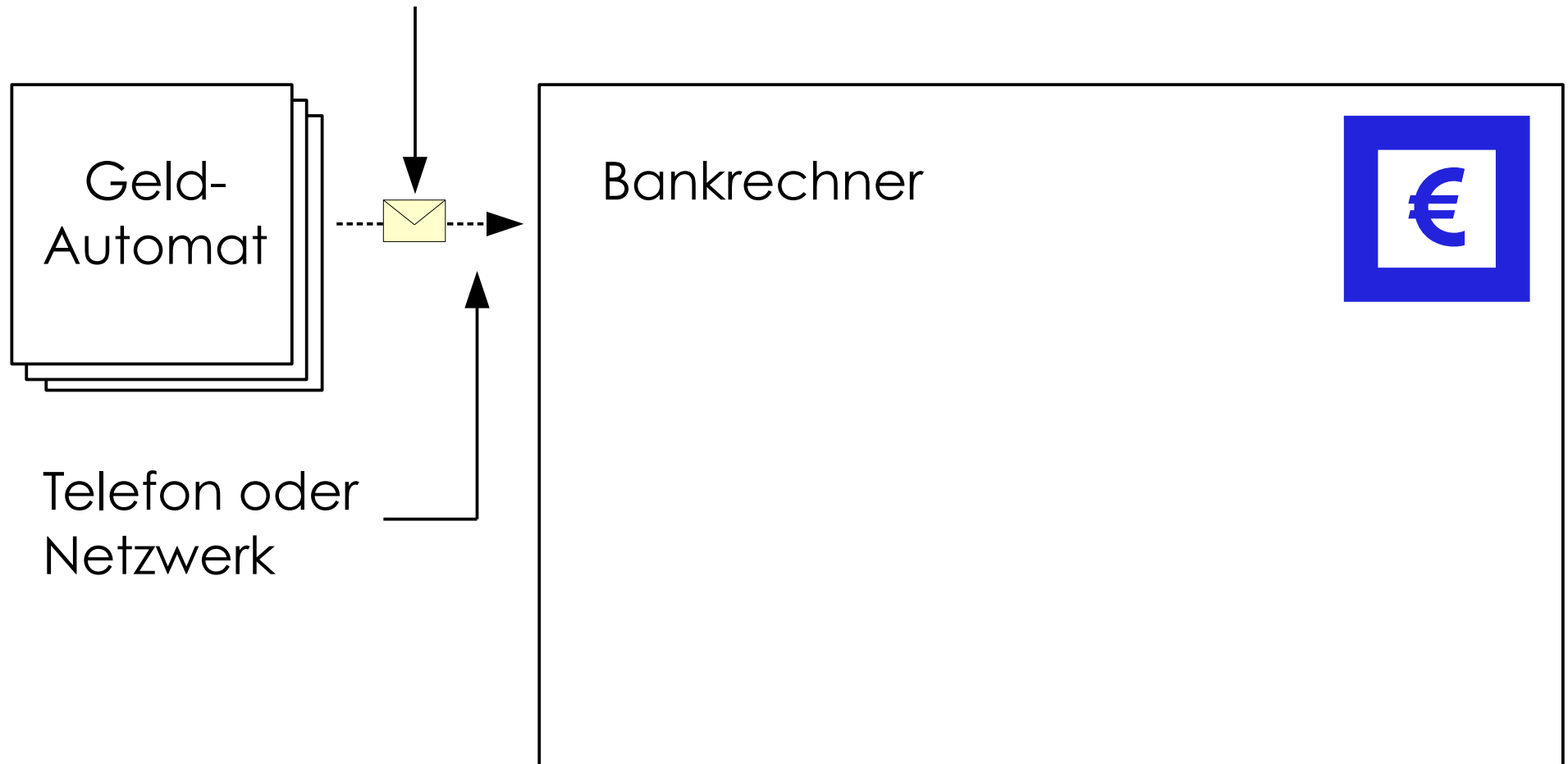
- mit/ohne HW-Unterstützung
- im Ein-/Mehrprozessor-Fall
- mit/ohne busy waiting

# Grundsatz

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

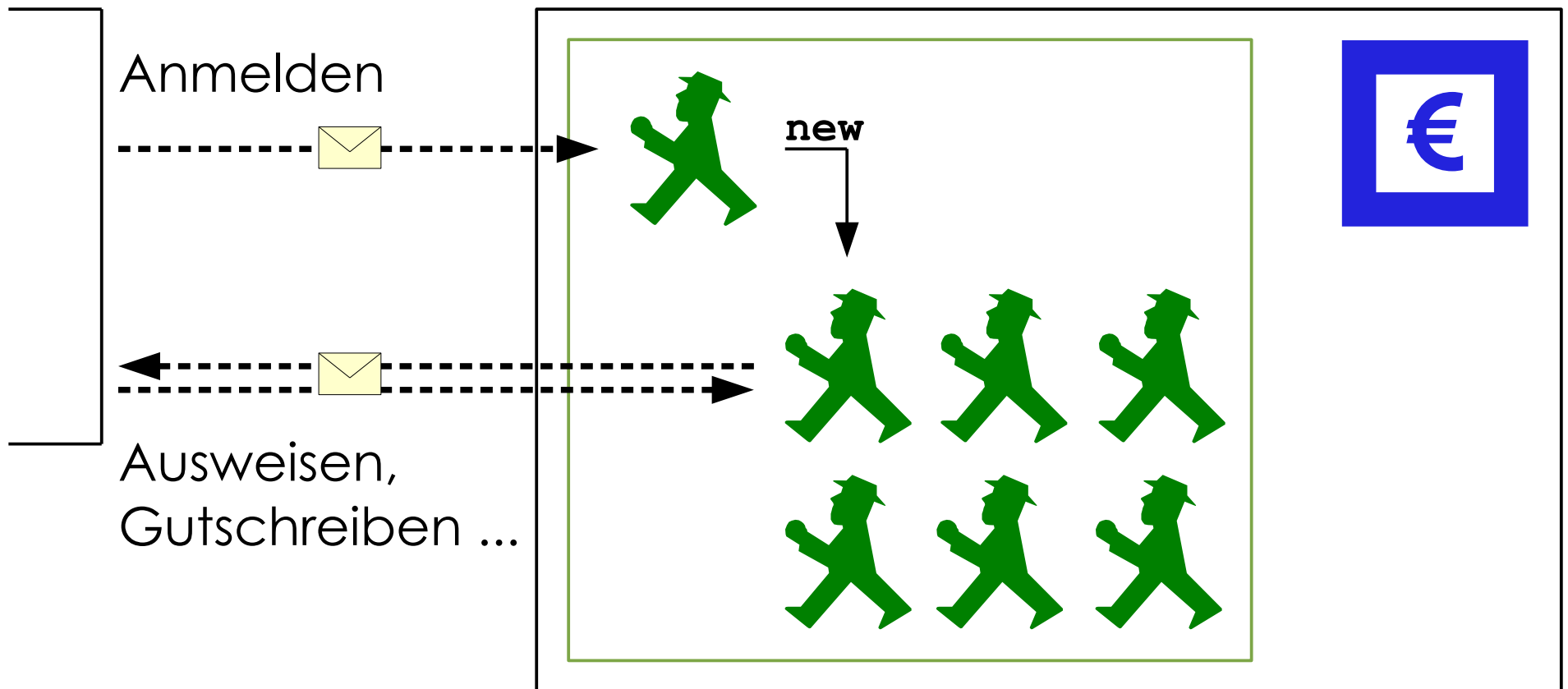
# Beispiel Geldautomat

Anmelden, ausweisen, abheben, einzahlen, ...



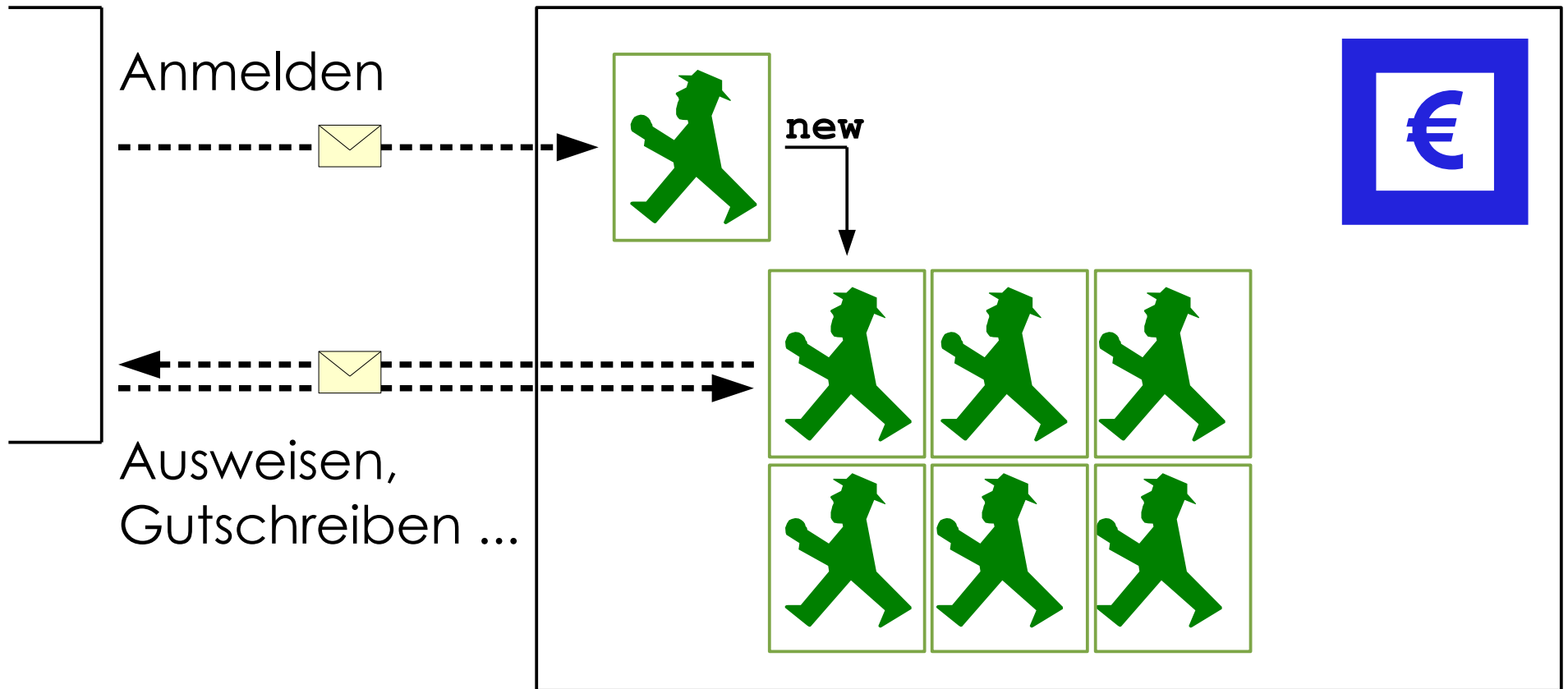
(grob nach Nichols, Buttlar, Farell)

# 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