

# Übung zu Betriebssysteme

## Zeitscheibenscheduling

---

10. Januar 2024

Maximilian Ott, Dustin Nguyen, Phillip Raffeck & Bernhard Heinloth

Lehrstuhl für Informatik 4  
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme  
und Betriebssysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Motivation

---

# Kooperative Ablaufplanung

```
int i = 0;
while (true){
    Guarded section;
    kout << i++ << endl;
    scheduler.resume();
}
```

## Unterbrechende Ablaufplanung

```
int i = 0;
while (true){
    Guarded section;
    kout << i++ << endl;
}
```

# Unterbrechende Ablaufplanung

```
int i = 0;
while (true){
    Guarded section;
    kout << i++ << endl;
}

```

⚡ Scheduler Interrupt

# Unterbrechende Ablaufplanung

```
int i = 0;
while (true){
    Guarded section;
    kout << i++ << endl;
}

```

⚡ Scheduler Interrupt

Aufgabe: **Präemptives Scheduling mittels Timer.**

**Zeitgeber**

---

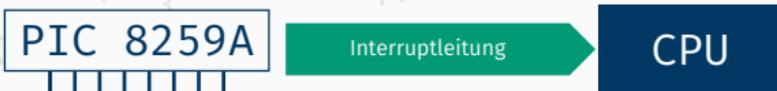
# Programmable Interval Timer (PIT)

- Standardtimer seit 1981 (IBM-PC, Intel 8253/8254)
- ca. 1 193 182 Hz ( $= \frac{1}{3}$  NTSC-Freq.)
  - Genauigkeit: 838 ns
- drei Kanäle mit je einen 16 bit Zähler
  - Kanal 0** löst standardmäßig alle 54.9254ms IRQ 0 aus
  - Kanal 1** früher für Arbeitsspeicher
  - Kanal 2** für PC Speaker (Tonfrequenz)

# Programmable Interval Timer (PIT)

- Standardtimer seit 1981 (IBM-PC, Intel 8253/8254)
- ca. 1 193 182 Hz (=  $\frac{1}{2}$  NTSC-Freq.)

→ Genauigkeit

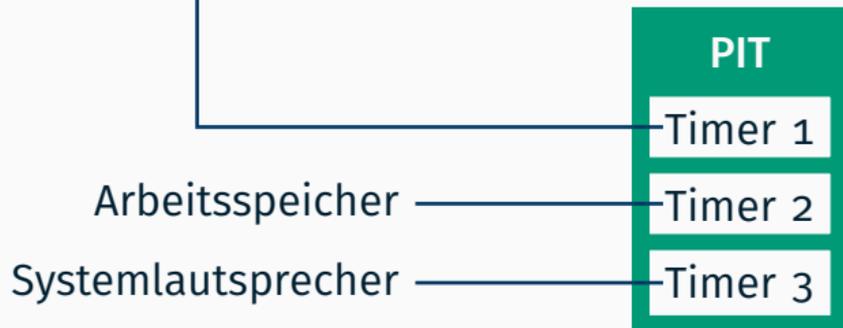


- drei Kanäle mit je einen 16 bit Zähler

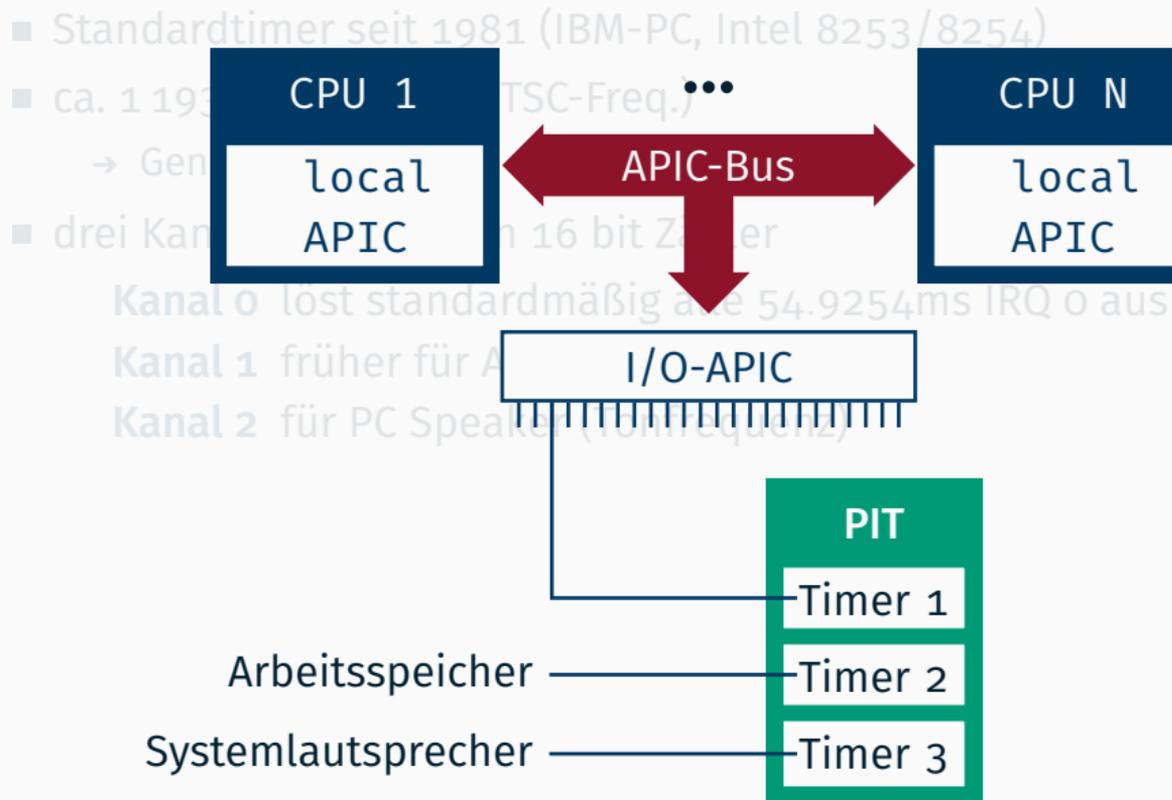
Kanal 0 löst standardmäßig alle 54.9254ms IRQ 0 aus

Kanal 1 früher für Arbeitsspeicher

Kanal 2 für PC Speaker (Tonfrequenz)



# Programmable Interval Timer (PIT)



# Programmable Interval Timer (PIT)

- Standardtimer seit 1981 (IBM-PC, Intel 8253/8254)
- ca. 1 193 182 Hz ( $= \frac{1}{3}$  NTSC-Freq.)
  - Genauigkeit: 838 ns
- drei Kanäle mit je einen 16 bit Zähler
  - Kanal 0** löst standardmäßig alle 54.9254ms IRQ 0 aus
  - Kanal 1** früher für Arbeitsspeicher
  - Kanal 2** für PC Speaker (Tonfrequenz)
- via PIC bzw. I/O APIC → (relativ) langsam

# Programmable Interval Timer (PIT)

- Standardtimer seit 1981 (IBM-PC, Intel 8253/8254)
- ca. 1 193 182 Hz ( $= \frac{1}{3}$  NTSC-Freq.)
  - Genauigkeit: 838 ns
- drei Kanäle mit je einen 16 bit Zähler
  - Kanal 0** löst standardmäßig alle 54.9254ms IRQ 0 aus
  - Kanal 1** früher für Arbeitsspeicher
  - Kanal 2** für PC Speaker (Tonfrequenz)
- via PIC bzw. I/O APIC → (relativ) langsam
- *ausreichend für BS*

# Programmable Interval Timer (PIT)

- Standardtimer seit 1981 (IBM-PC, Intel 8253/8254)
  - ca. 1 193 182 Hz ( $= \frac{1}{3}$  NTSC-Freq.)
    - Genauigkeit: 838 ns
  - drei Kanäle mit je einen 16 bit Zähler
    - Kanal 0** löst standardmäßig alle 54.9254ms IRQ 0 aus
    - Kanal 1** früher für Arbeitsspeicher
    - Kanal 2** für PC Speaker (Tonfrequenz)
  - via PIC bzw. I/O APIC → (relativ) langsam
  - *ausreichend für BS (bis WS13), geht aber besser.*
- machine/pit.h

# Real Time Clock (RTC)

- seit 1984 (IBM-PC/AT)
- 32 768 Hz (=  $2^{15}$  Hz, Verwendung in Uhren)
  - Standardmäßig Interrupts bei 1 024 Hz (fast 1 ms)
  - 12 weitere Möglichkeiten von 2 bis 8 192 Hz durch Vorteiler
  - IRQ 8 (Problem?)
- für Zeit & Datum
- Betrieb im ausgeschalteten Zustand mittels Batterie

# Time Stamp Counter (TSC)

- seit 1993 (Pentium)
- 64 bit, auslesbar über Assemblerinstruktion `rdtsc`
- Taktfrequenz wie CPU

# Time Stamp Counter (TSC)

- seit 1993 (Pentium)
- 64 bit, auslesbar über Assemblerinstruktion `rdtsc`
- Taktfrequenz wie CPU
  - ursprünglich Erhöhung mit jedem Clock-Signal
  - unterschiedliche Takte abhängig vom Stromsparmodes
  - bei neueren Versionen: konstante Rate entsprechend nominaler Geschwindigkeit

# Time Stamp Counter (TSC)

- seit 1993 (Pentium)
- 64 bit, auslesbar über Assemblerinstruktion `rdtsc`
- Taktfrequenz wie CPU
  - ursprünglich Erhöhung mit jedem Clock-Signal
  - unterschiedliche Takte abhängig vom Stromsparmmodus
  - bei neueren Versionen: konstante Rate entsprechend nominaler Geschwindigkeit
- kann keinen Interrupt auslösen

# Time Stamp Counter (TSC)

- seit 1993 (Pentium)
  - 64 bit, auslesbar über Assemblerinstruktion `rdtsc`
  - Taktfrequenz wie CPU
    - ursprünglich Erhöhung mit jedem Clock-Signal
    - unterschiedliche Takte abhängig vom Stromsparmmodus
    - bei neueren Versionen: konstante Rate entsprechend nominaler Geschwindigkeit
  - kann keinen Interrupt auslösen
- `machine/tsc.h`

# ACPI Power Management Timer

- seit es ACPI-Mainboards gibt (1996)
- 3 579 545 Hz (= NTSC-Freq.)
- ein 24 oder 32 bit Zähler
  - besser als alte (nicht konstante) TSC
  - Zugriff über I/O Port
- kann auch keinen Interrupt auslösen

# High Precision Event Timer (HPET)

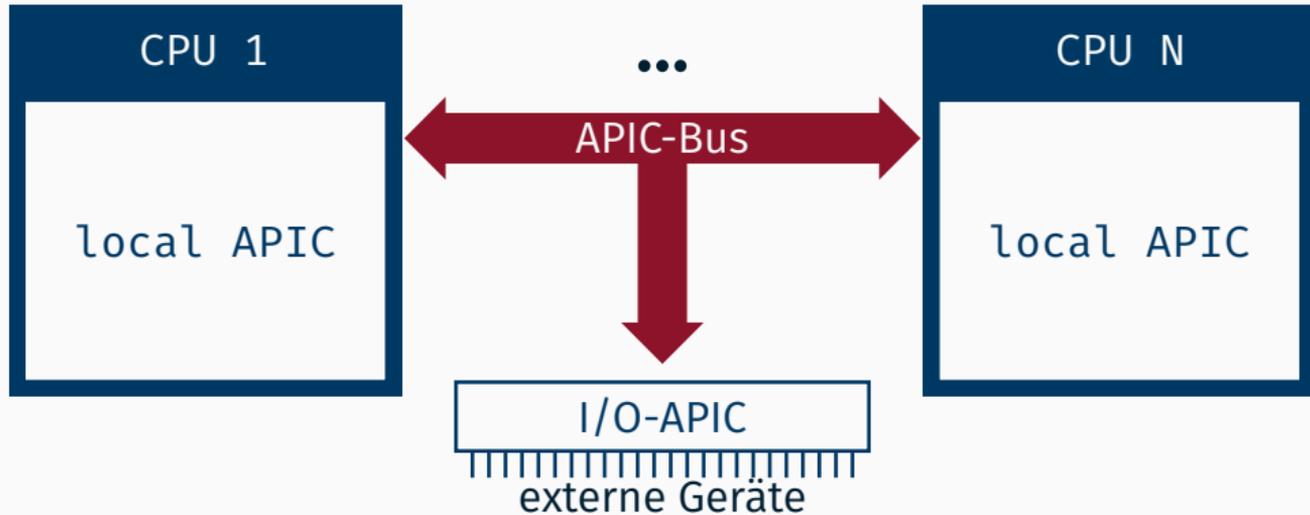
- von Intel und Microsoft 2005 als PIT- & RTC-Ersatz veröffentlicht
- $\geq 10$  MHz
  - Genauigkeit: 100 ns oder besser
- ein 64 bit Zähler
  - min. drei 32 oder 64 bit breite Vergleichseinrichtungen
  - konfigurierbarer Interrupt bei Gleichheit

- $\geq 100$  MHz
  - Genauigkeit: 10 ns oder besser
- 32 bit Zähler
- verwendet Busfrequenz
  - abhängig vom System
  - aber unabhängig von Stromsparmmodus
  - Interrupt geht nur an den entsprechenden Kern
  - 8 Möglichkeiten (bis  $\frac{1}{128}$  Busfrequenz) durch Vorteiler

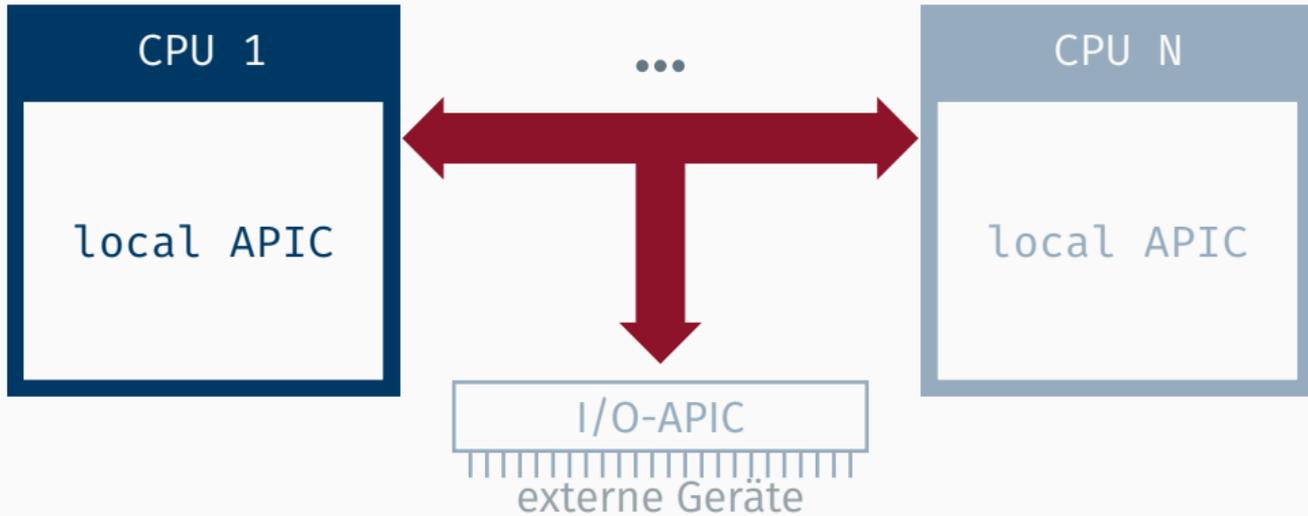
- $\geq 100$  MHz
  - Genauigkeit: 10 ns oder besser
- 32 bit Zähler
- verwendet Busfrequenz
  - abhängig vom System
  - aber unabhängig von Stromsparmmodus
  - Interrupt geht nur an den entsprechenden Kern
  - 8 Möglichkeiten (bis  $\frac{1}{128}$  Busfrequenz) durch Vorteiler

**Perfekt für unsere Bedürfnisse**

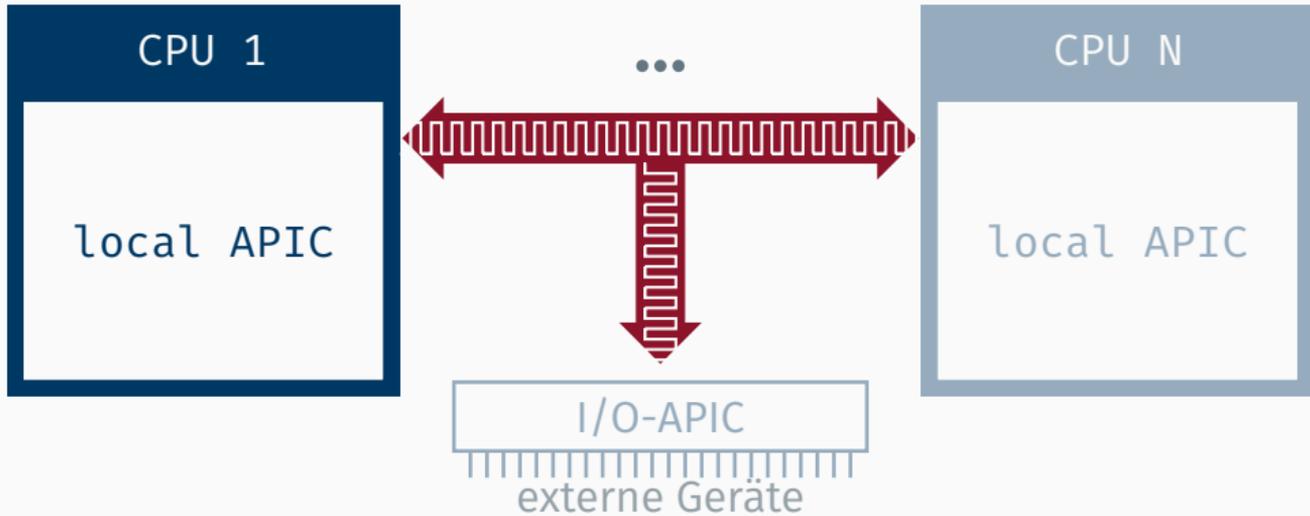
# Funktionsweise des LAPIC Timers



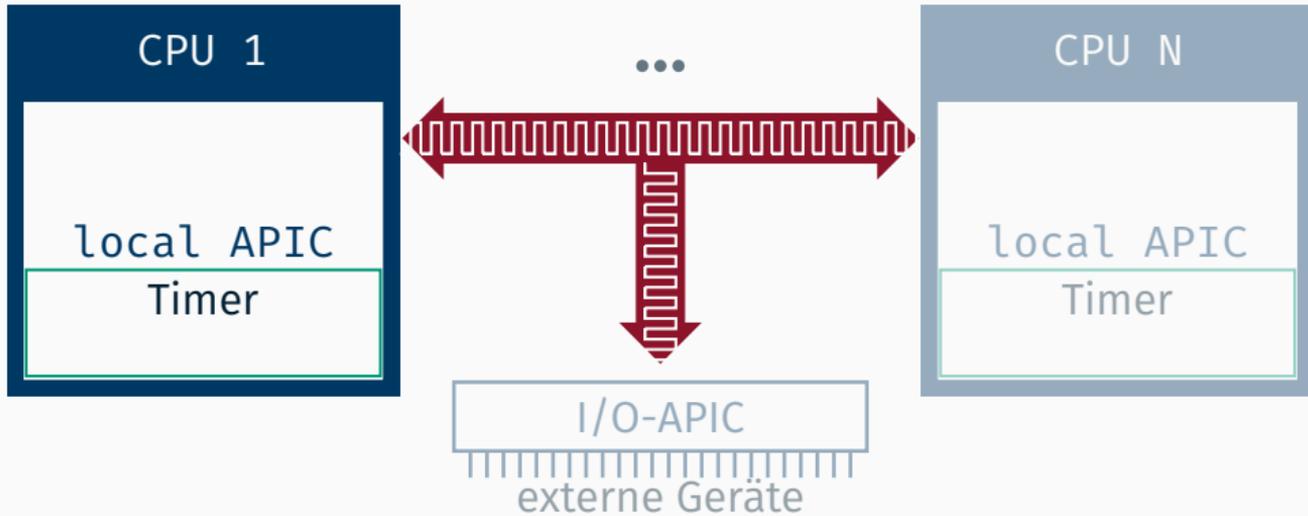
# Funktionsweise des LAPIC Timers



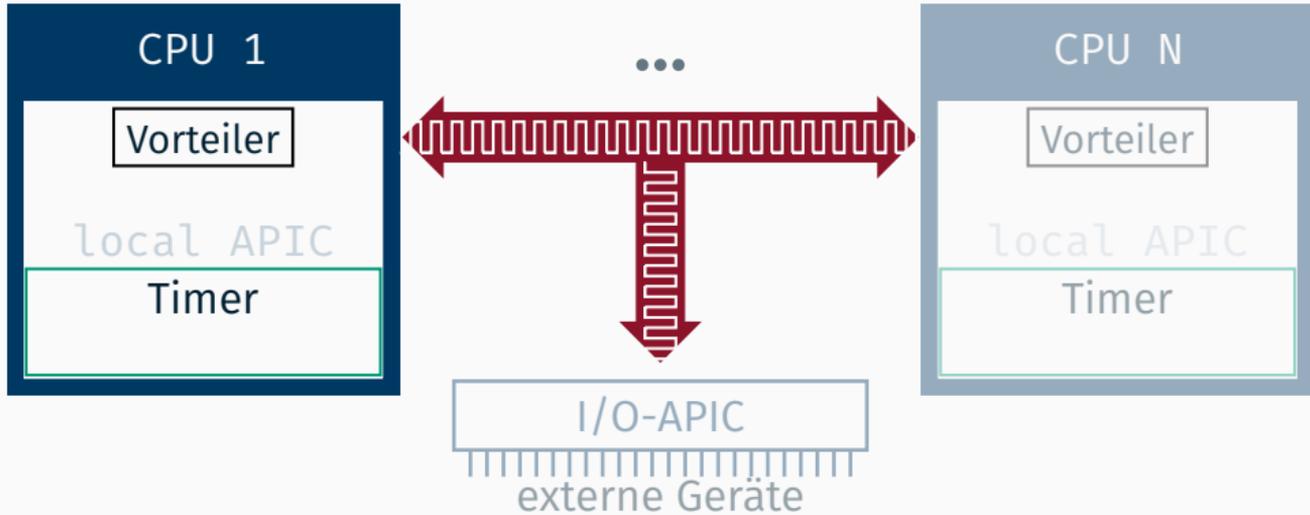
# Funktionsweise des LAPIC Timers



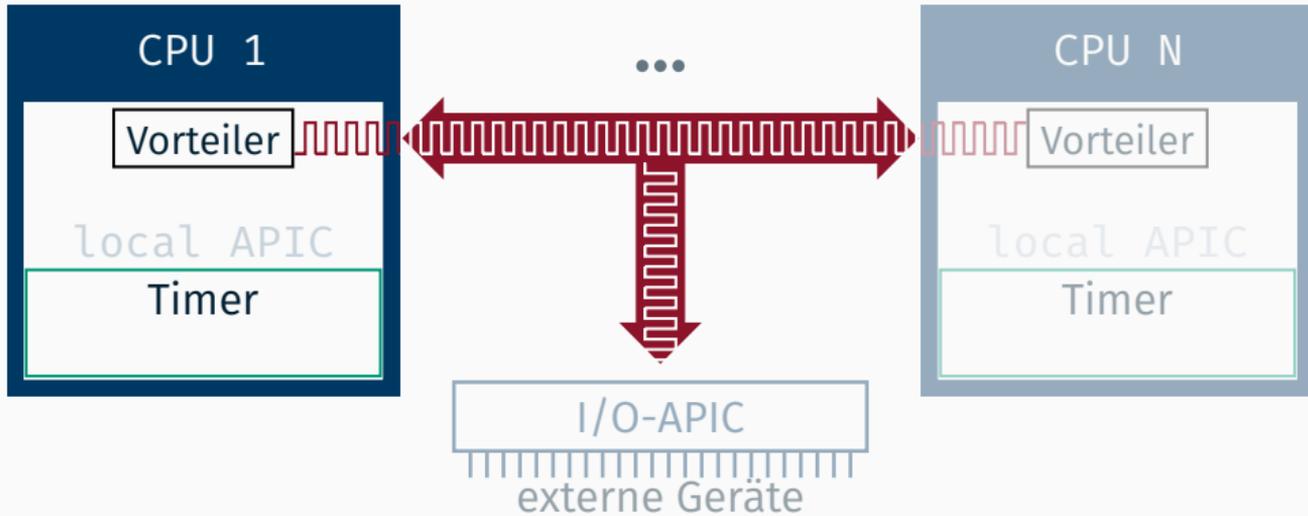
# Funktionsweise des LAPIC Timers



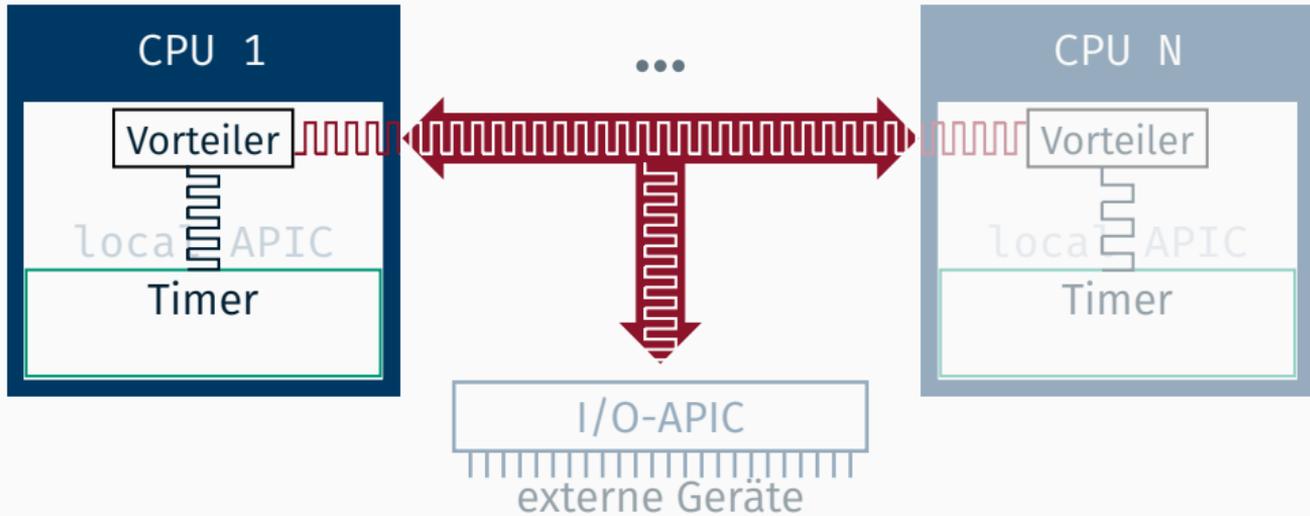
# Funktionsweise des LAPIC Timers



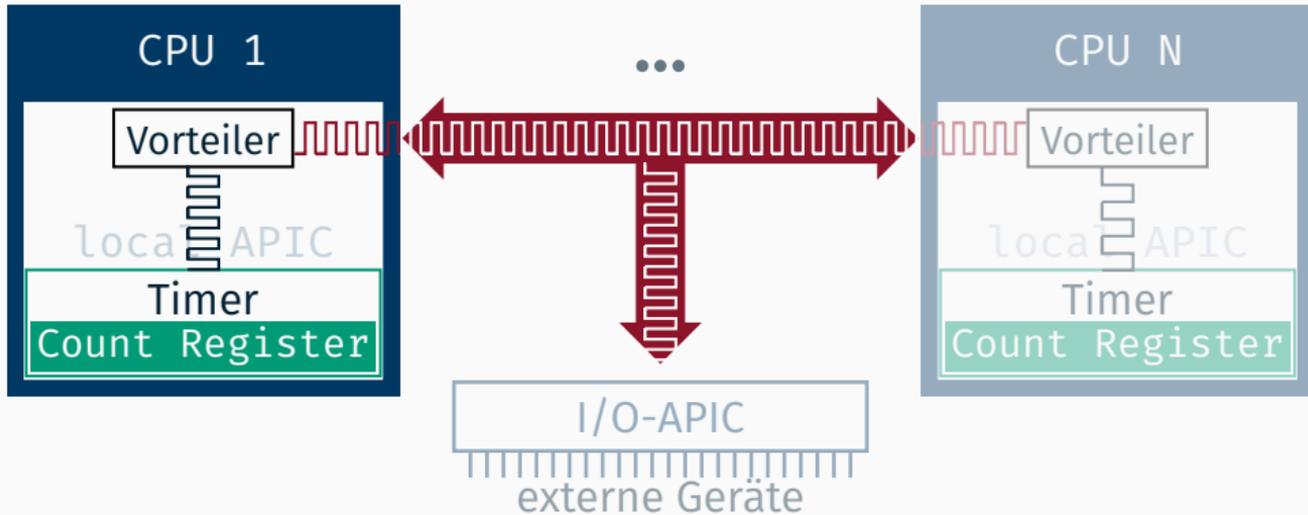
# Funktionsweise des LAPIC Timers



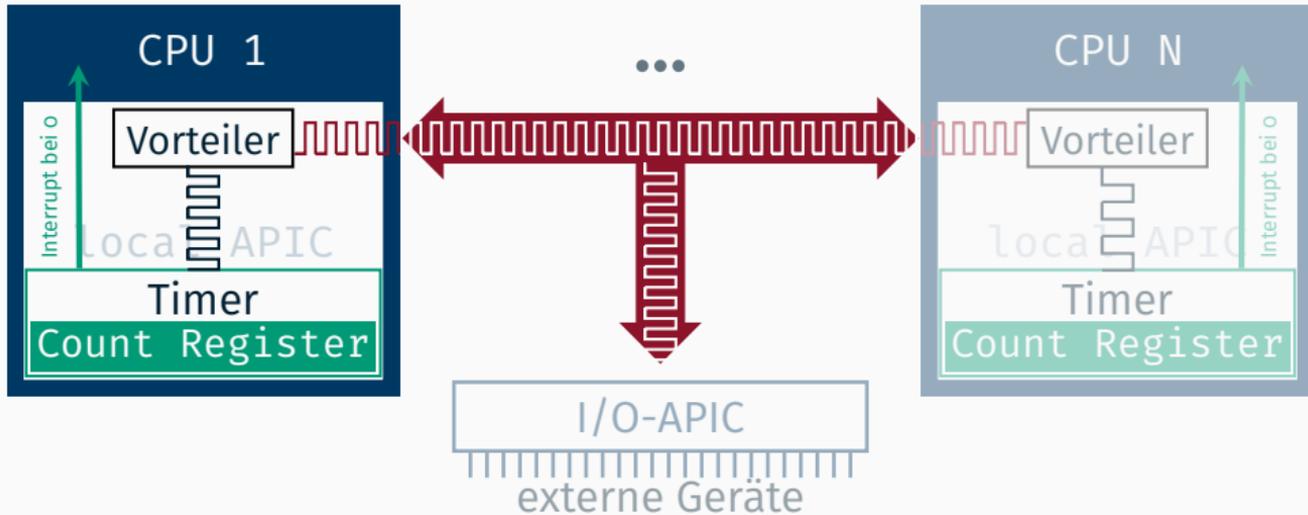
# Funktionsweise des LAPIC Timers



# Funktionsweise des LAPIC Timers



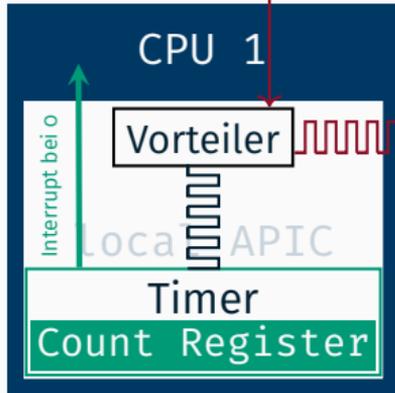
# Funktionsweise des LAPIC Timers



# Funktionsweise des LAPIC Timers

Divide Configuration Register

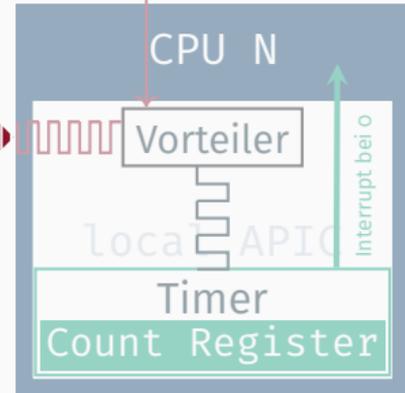
0xf<sub>ee</sub>003e0



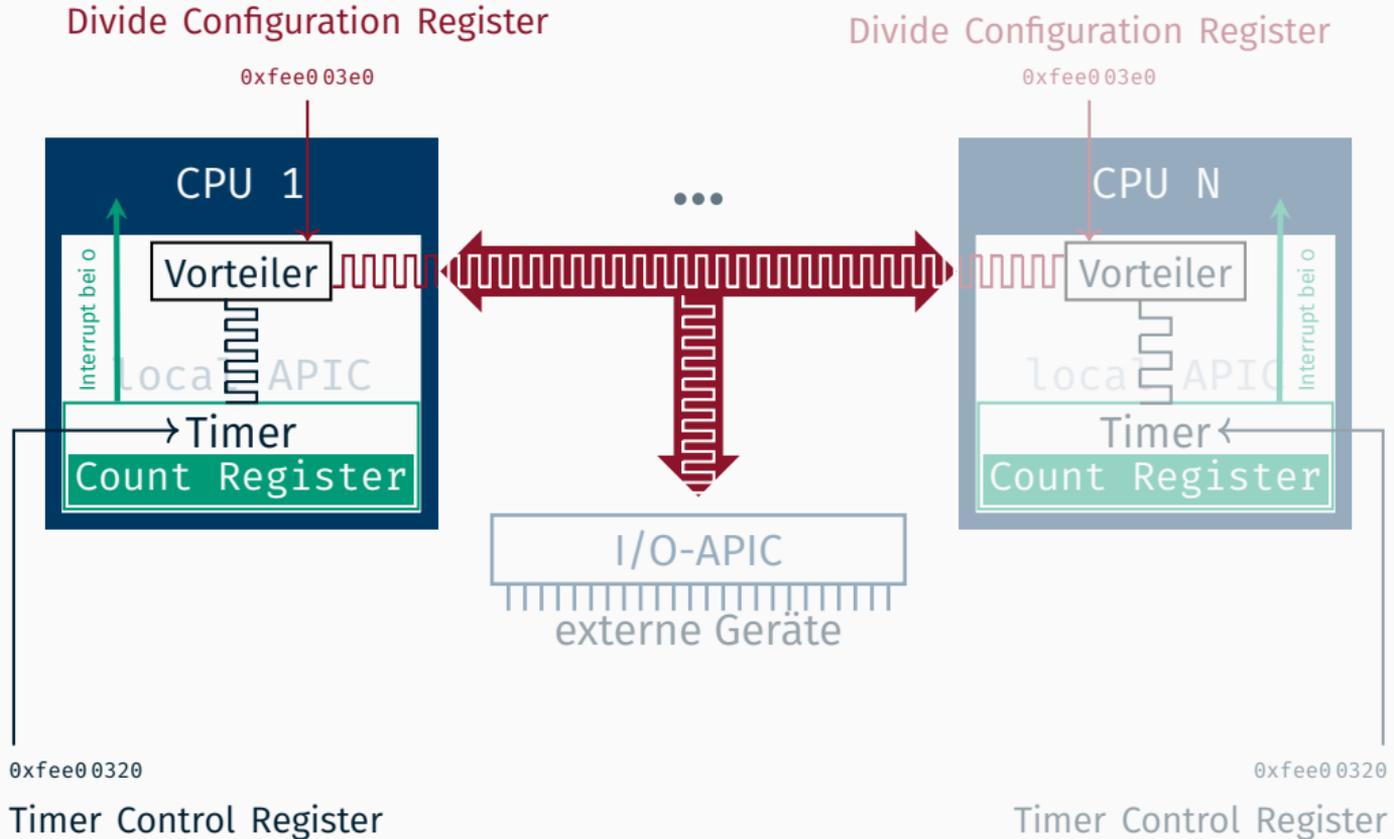
...

Divide Configuration Register

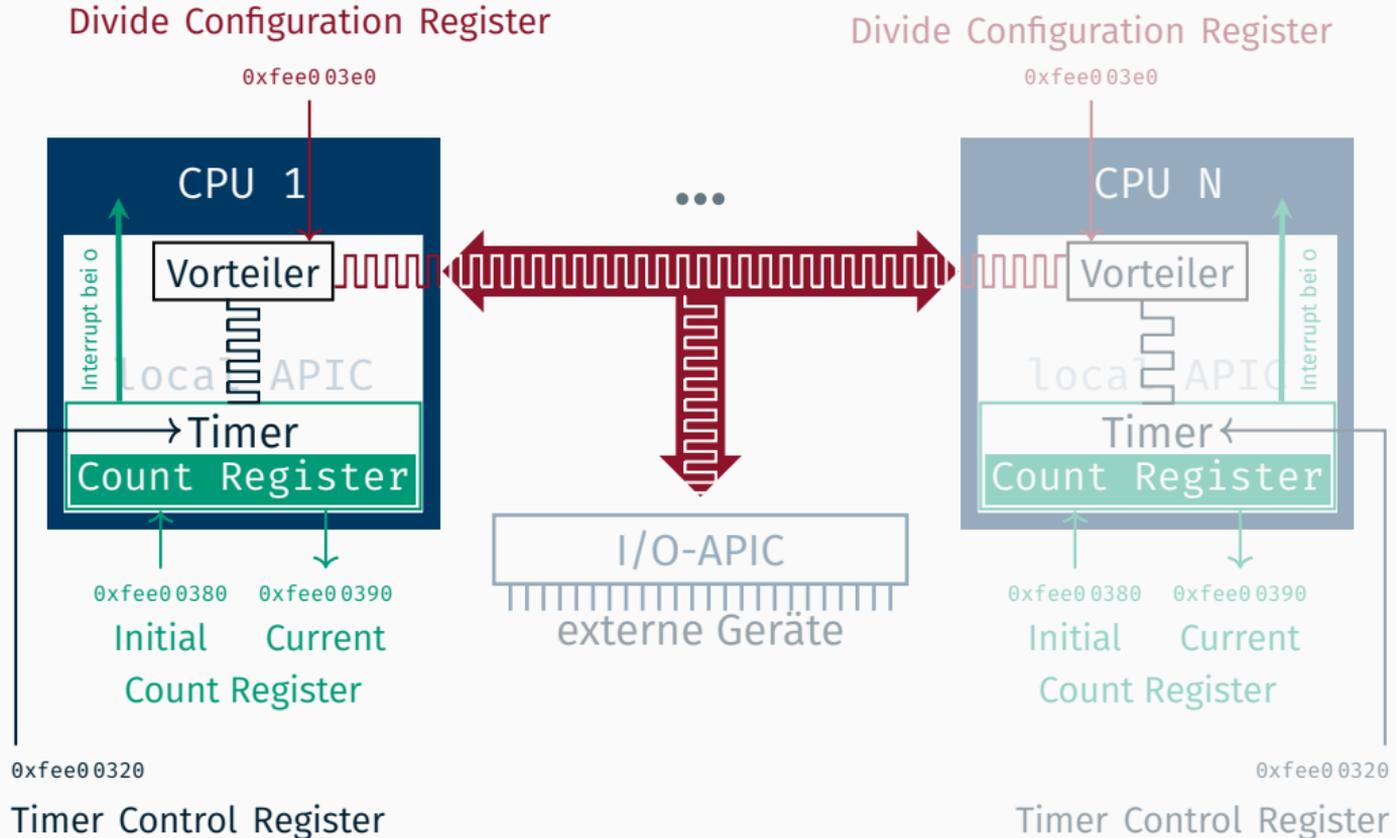
0xf<sub>ee</sub>003e0



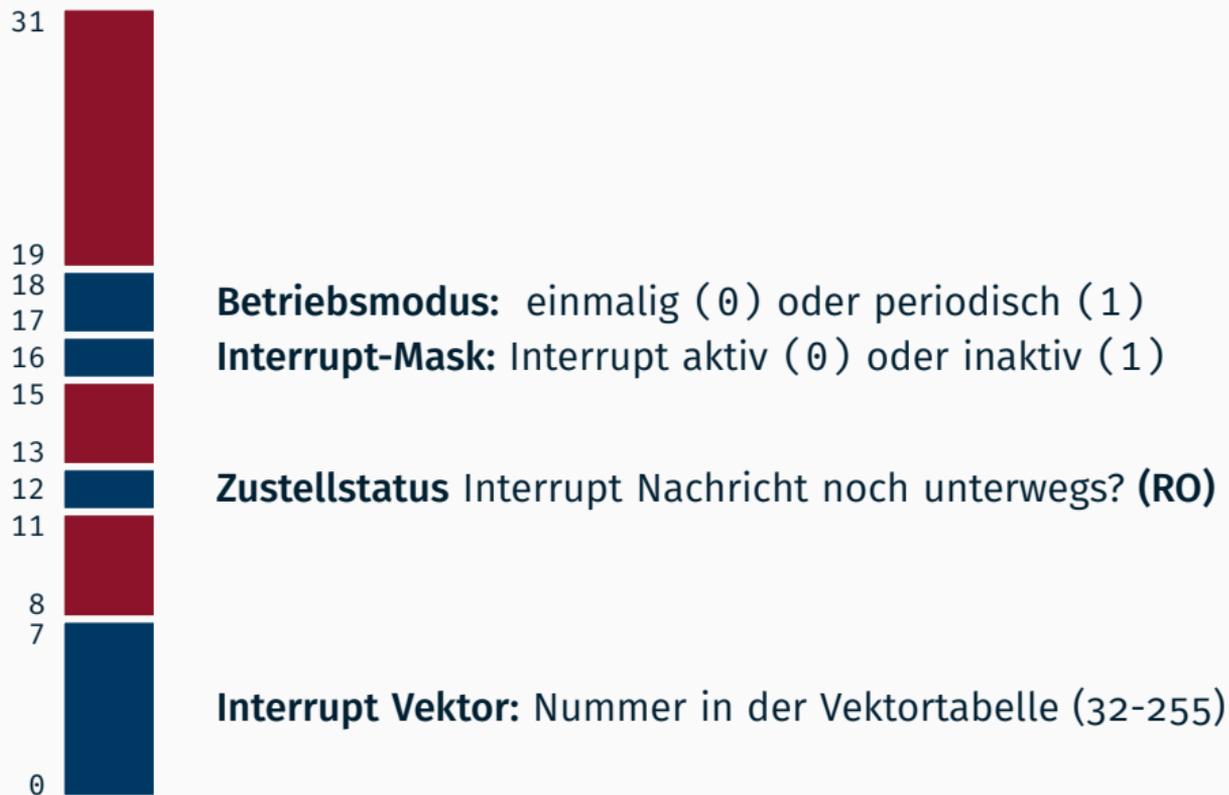
# Funktionsweise des LAPIC Timers



# Funktionsweise des LAPIC Timers



# Aufbau des Timer Control Register Eintrags



# Zusammenfassung LAPIC Timer

- jede CPU hat einen eigenen 32bit Timer
- Änderung am INITIAL COUNT REGISTER startet den Timer
- zu Beginn wird der initiale Startzählwert aus dem INITIAL COUNT REGISTER in das CURRENT COUNT REGISTER kopiert
- welches im  $\frac{\text{Bustakt}}{\text{Vorteiler}}$  dekrementiert wird
- bei 0 wird – sofern aktiviert – ein Interrupt ausgelöst
- je nach Betriebsmodus wird gestoppt oder wieder neu begonnen

# Umsetzung

---

**windup** stellt das Unterbrechungsintervall ein  
(z.B. alle 1000 Mikrosekunden)

**windup** stellt das Unterbrechungsintervall ein  
(z.B. alle 1000 Mikrosekunden)

**activate** setzt den Timer und aktiviert Interrupts

**windup** stellt das Unterbrechungsintervall ein  
(z.B. alle 1000 Mikrosekunden)

**activate** setzt den Timer und aktiviert Interrupts

**prologue** fordert Epilog an  
(und kann zu Testzwecken eine Ausgabe tätigen)

**windup** stellt das Unterbrechungsintervall ein  
(z.B. alle 1000 Mikrosekunden)

**activate** setzt den Timer und aktiviert Interrupts

**prologue** fordert Epilog an  
(und kann zu Testzwecken eine Ausgabe tätigen)

**epilogue** wechselt die Anwendung mittels `scheduler.resume()`

# Frequenz des LAPIC-Timers?

## 10.5.4 APIC Timer

The local APIC unit contains a 32-bit programmable timer that is available to software to time events or operations. This timer is set up by programming four registers: the divide configuration register (see Figure 10-10), the initial-count and current-count registers (see Figure 10-11), and the LVT timer register (see Figure 10-8).

If CPUID.06H:EAX.ARAT[bit 2] = 1, the processor's APIC timer runs at a constant rate regardless of P-state transitions and it continues to run at the same rate in deep C-states.

If CPUID.06H:EAX.ARAT[bit 2] = 0 or if CPUID 06H is not supported, the APIC timer may temporarily stop while the processor is in deep C-states or during transitions caused by Enhanced Intel SpeedStep® Technology.

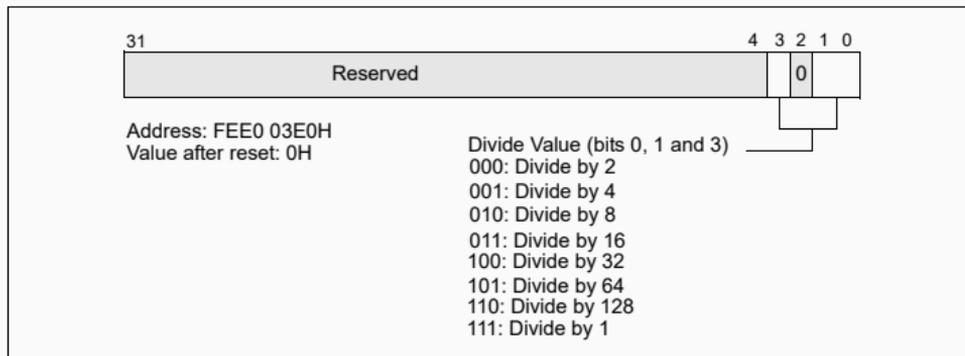


Figure 10-10. Divide Configuration Register

The APIC timer frequency will be the processor's bus clock or core crystal clock frequency (when TSC/core crystal clock ratio is enumerated in CPUID leaf 0x15) divided by the value specified in the divide configuration register.

# Frequenz des LAPIC-Timers?

## 10.5.4 APIC Timer

The local APIC unit contains a 32-bit programmable timer that is available to software to time events or operations. This timer is set up by programming four registers: the divide configuration register (see Figure 10-10), the initial-count and current-count registers (see Figure 10-11), and the LVT timer register (see Figure 10-8).

If CPUID.06H:EAX.ARAT[bit 2] = 1, the processor's APIC timer runs at a constant rate regardless of P-state transitions and it continues to run at the same rate in deep C-states.

If CPUID.06H:EAX.ARAT[bit 2] = 0 or if CPUID 06H is not supported, the APIC timer may temporarily stop while the processor is in deep C-states or during transitions caused by Enhanced Intel SpeedStep® Technology.

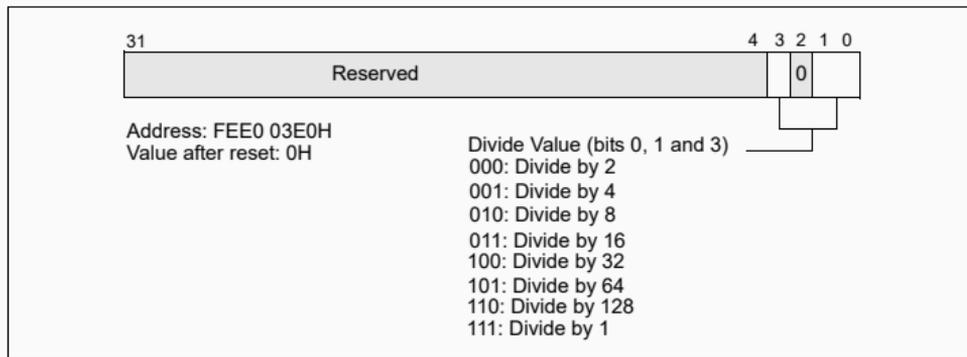


Figure 10-10. Divide Configuration Register

The APIC timer frequency will be the processor's bus clock or core crystal clock frequency (when TSC/core crystal clock ratio is enumerated in CPUID leaf 0x15) divided by the value specified in the divide configuration register.

# LAPIC-Timer einstellen

## 1. LAPIC-Timer kalibrieren

# LAPIC-Timer einstellen

## 1. LAPIC-Timer kalibrieren

- unter Verwendung des PIT

(mittels `PIT::set` Wartezeit einstellen und in `PIT::waitForTimeout` warten)

# LAPIC-Timer einstellen

## 1. LAPIC-Timer kalibrieren

- unter Verwendung des PIT  
(mittels `PIT::set` Wartezeit einstellen und in `PIT::waitForTimeout` warten)
- in der Funktion `LAPIC::Timer::ticks`  
(Funktion gibt die Anzahl der Ticks in einer **Millisekunde** zurück)

# LAPIC-Timer einstellen

## 1. LAPIC-Timer kalibrieren

- unter Verwendung des PIT  
(mittels `PIT::set` Wartezeit einstellen und in `PIT::waitForTimeout` warten)
- in der Funktion `LAPIC::Timer::ticks`  
(Funktion gibt die Anzahl der Ticks in einer **Millisekunde** zurück)
- benötigt zur Konfiguration die ebenfalls noch zu implementierende Funktion `LAPIC::Timer::set`

# LAPIC-Timer einstellen

## 1. LAPIC-Timer kalibrieren

- unter Verwendung des PIT  
(mittels `PIT::set` Wartezeit einstellen und in `PIT::waitForTimeout` warten)
- in der Funktion `LAPIC::Timer::ticks`  
(Funktion gibt die Anzahl der Ticks in einer **Millisekunde** zurück)
- benötigt zur Konfiguration die ebenfalls noch zu implementierende Funktion `LAPIC::Timer::set`
- Hilfsstrukturen in `lapic_timer.cc` & `lapic_registers.h`

## 2. Initialen Wert und Vorteiler korrekt setzen

## 2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden

## 2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden
- Startwertzähler  $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Vorteiler} \cdot 1000}$  berechnen, dabei gilt  $\text{Vorteiler} = 2^x$  mit  $x \in \{0, \dots, 7\}$

## 2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden
- Startwertzähler  $initial = \frac{n \cdot \text{LAPIC::Timer::ticks()}}{\text{Vorteiler} \cdot 1000}$  berechnen, dabei gilt  $\text{Vorteiler} = 2^x$  mit  $x \in \{0, \dots, 7\}$
- Möglichst kleiner Vorteiler, aber kein Überlauf (32 bit!)

## 2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden
- Startwertzähler  $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Vorteiler} \cdot 1000}$  berechnen, dabei gilt  $\text{Vorteiler} = 2^x$  mit  $x \in \{0, \dots, 7\}$
- Möglichst kleiner Vorteiler, aber kein Überlauf (32 bit!)
- **Beispiel:** `watch.windup(5000000);`  
 $n \quad 5\,000\,000 \mu\text{s} = 5 \text{ s}$   
`LAPIC::Timer::ticks`  $1\,000\,000 \text{ ms}^{-1}$

## 2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden
- Startwertzähler  $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Vorteiler} \cdot 1000}$  berechnen, dabei gilt  $\text{Vorteiler} = 2^x$  mit  $x \in \{0, \dots, 7\}$
- Möglichst kleiner Vorteiler, aber kein Überlauf (32 bit!)
- **Beispiel:** `watch.windup(5000000);`

$$n \quad 5\,000\,000 \mu\text{s} = 5 \text{ s}$$

$$\text{LAPIC}::\text{Timer}::\text{ticks} \quad 1\,000\,000 \text{ ms}^{-1}$$

$$\text{Vorteiler} \quad 1 = 2^0$$

## 2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden
- Startwertzähler  $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Vorteiler} \cdot 1000}$  berechnen, dabei gilt  $\text{Vorteiler} = 2^x$  mit  $x \in \{0, \dots, 7\}$
- Möglichst kleiner Vorteiler, aber kein Überlauf (32 bit!)
- **Beispiel:** `watch.windup(5000000);`

	$n$	5 000 000 $\mu\text{s} = 5 \text{ s}$
LAPIC::Timer::ticks		1 000 000 $\text{ms}^{-1}$
Vorteiler		1 = $2^0$
initial		5 000 000 000

## 2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden
- Startwertzähler  $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Vorteiler} \cdot 1000}$  berechnen, dabei gilt  $\text{Vorteiler} = 2^x$  mit  $x \in \{0, \dots, 7\}$
- Möglichst kleiner Vorteiler, aber kein Überlauf (32 bit!)
- **Beispiel:** `watch.windup(5000000);`

	$n$	5 000 000 $\mu\text{s} = 5 \text{ s}$
LAPIC::Timer::ticks		1 000 000 $\text{ms}^{-1}$
Vorteiler		1 = $2^0$
initial		5 000 000 000 ⚡

## 2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden
- Startwertzähler  $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Vorteiler} \cdot 1000}$  berechnen, dabei gilt  $\text{Vorteiler} = 2^x$  mit  $x \in \{0, \dots, 7\}$
- Möglichst kleiner Vorteiler, aber kein Überlauf (32 bit!)
- **Beispiel:** `watch.windup(5000000);`

$$n \quad 5\,000\,000 \mu\text{s} = 5 \text{ s}$$

$$\text{LAPIC}::\text{Timer}::\text{ticks} \quad 1\,000\,000 \text{ ms}^{-1}$$

$$\text{Vorteiler} \quad 2 = 2^1$$

*initial*

## 2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden
- Startwertzähler  $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Vorteiler} \cdot 1000}$  berechnen, dabei gilt  $\text{Vorteiler} = 2^x$  mit  $x \in \{0, \dots, 7\}$
- Möglichst kleiner Vorteiler, aber kein Überlauf (32 bit!)
- **Beispiel:** `watch.windup(5000000);`

	$n$	5 000 000 $\mu\text{s} = 5 \text{ s}$
LAPIC::Timer::ticks		1 000 000 $\text{ms}^{-1}$
Vorteiler		2 = $2^1$
initial		2 500 000 000

## 2. Initialen Wert und Verteiler korrekt setzen

- Interrupt soll alle  $n$  Mikrosekunden ausgelöst werden
- Startwertzähler  $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Verteiler} \cdot 1000}$  berechnen, dabei gilt  $\text{Verteiler} = 2^x$  mit  $x \in \{0, \dots, 7\}$
- Möglichst kleiner Verteiler, aber kein Überlauf (32 bit!)
- **Beispiel:** `watch.windup(5000000);`

	$n$	5 000 000 $\mu\text{s} = 5 \text{ s}$
LAPIC::Timer::ticks		1 000 000 $\text{ms}^{-1}$
Verteiler		2 = $2^1$
initial		2 500 000 000 ✓

# Ablaufbeispiel (Standardfall)

$E_0$  (Anwendung)

app1



---

$E_{\frac{1}{2}}$  (Epilog)

---

$E_1$  (IRQ/Prolog)

# Ablaufbeispiel (Standardfall)

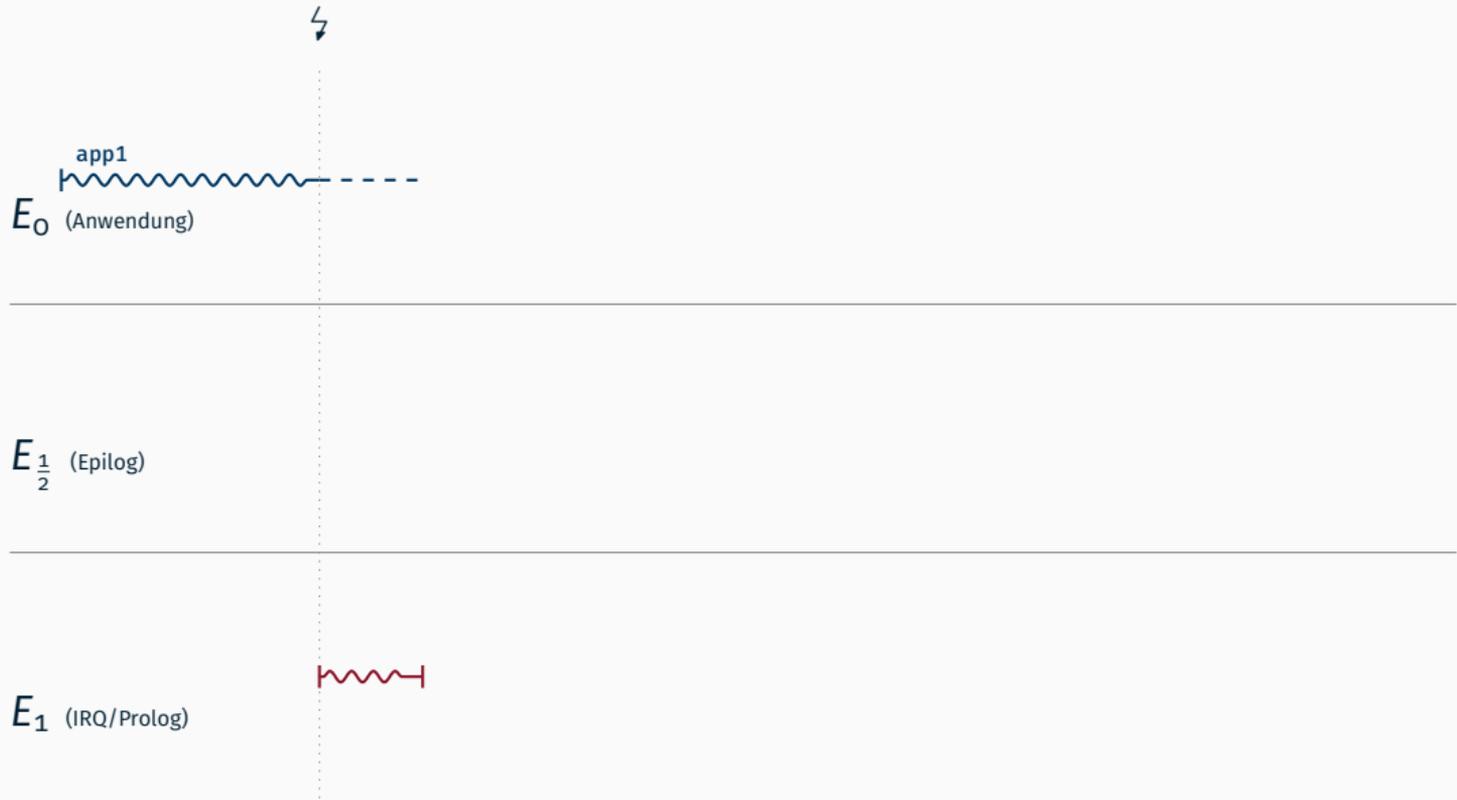


app1  
 $E_0$  (Anwendung)

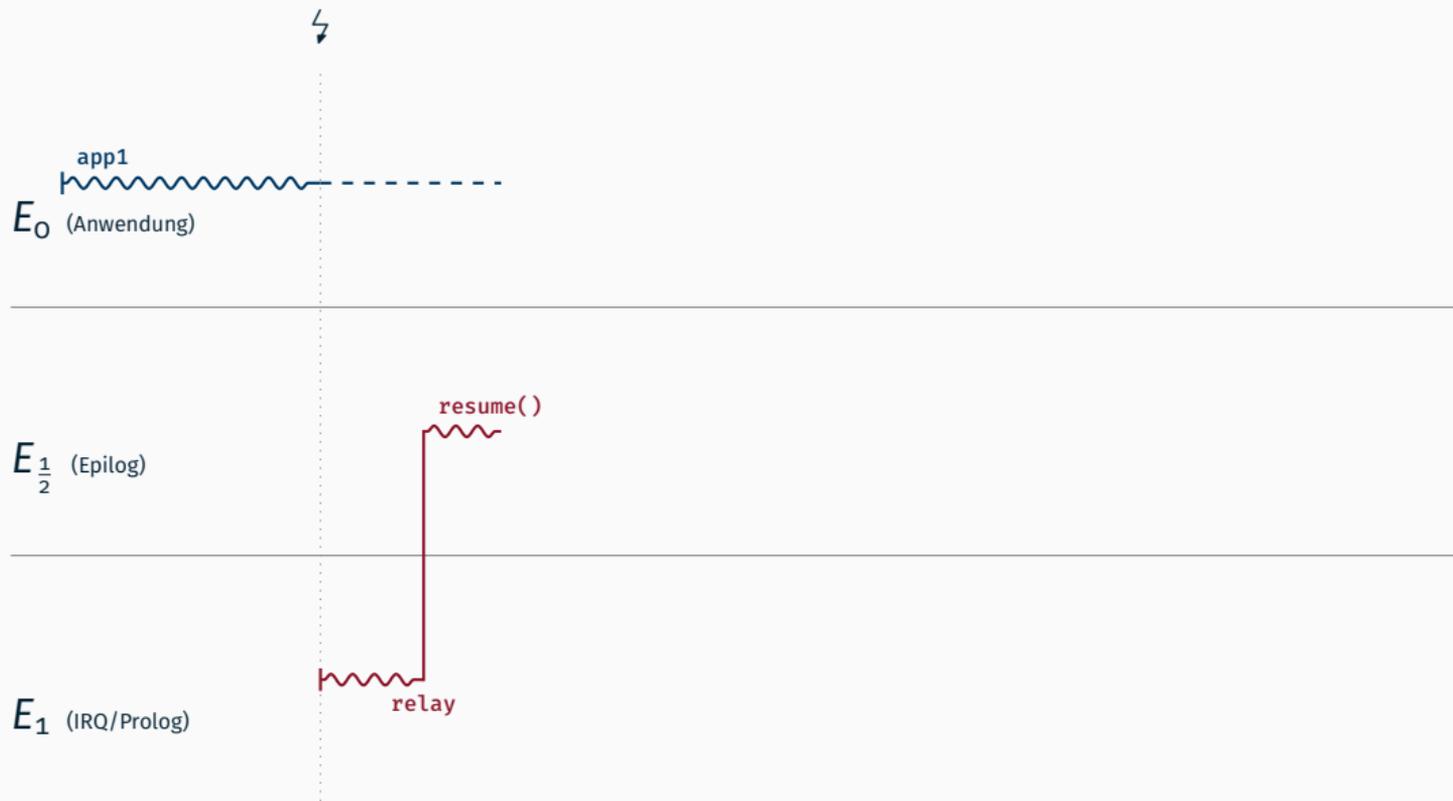
$E_{\frac{1}{2}}$  (Epilog)

$E_1$  (IRQ/Prolog)

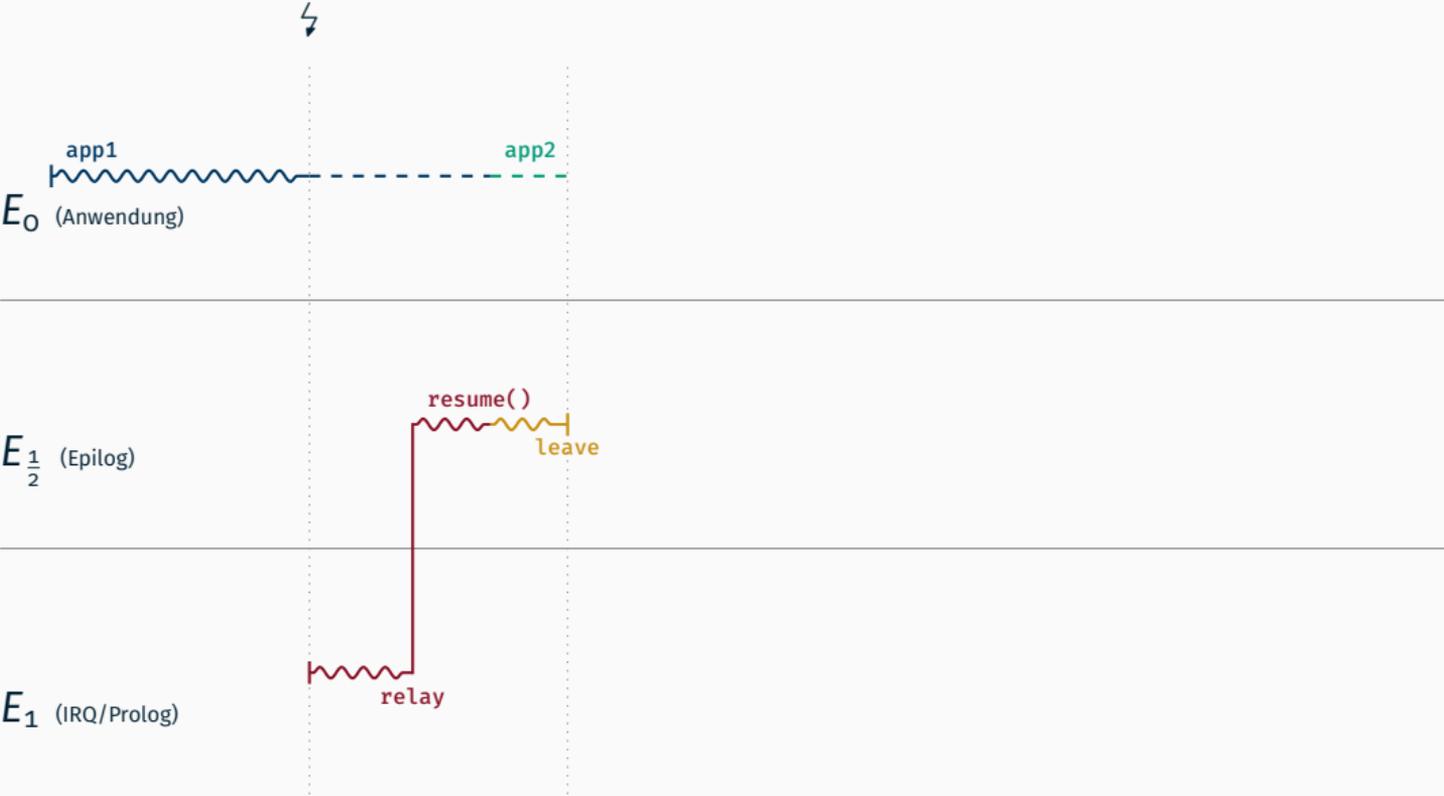
# Ablaufbeispiel (Standardfall)



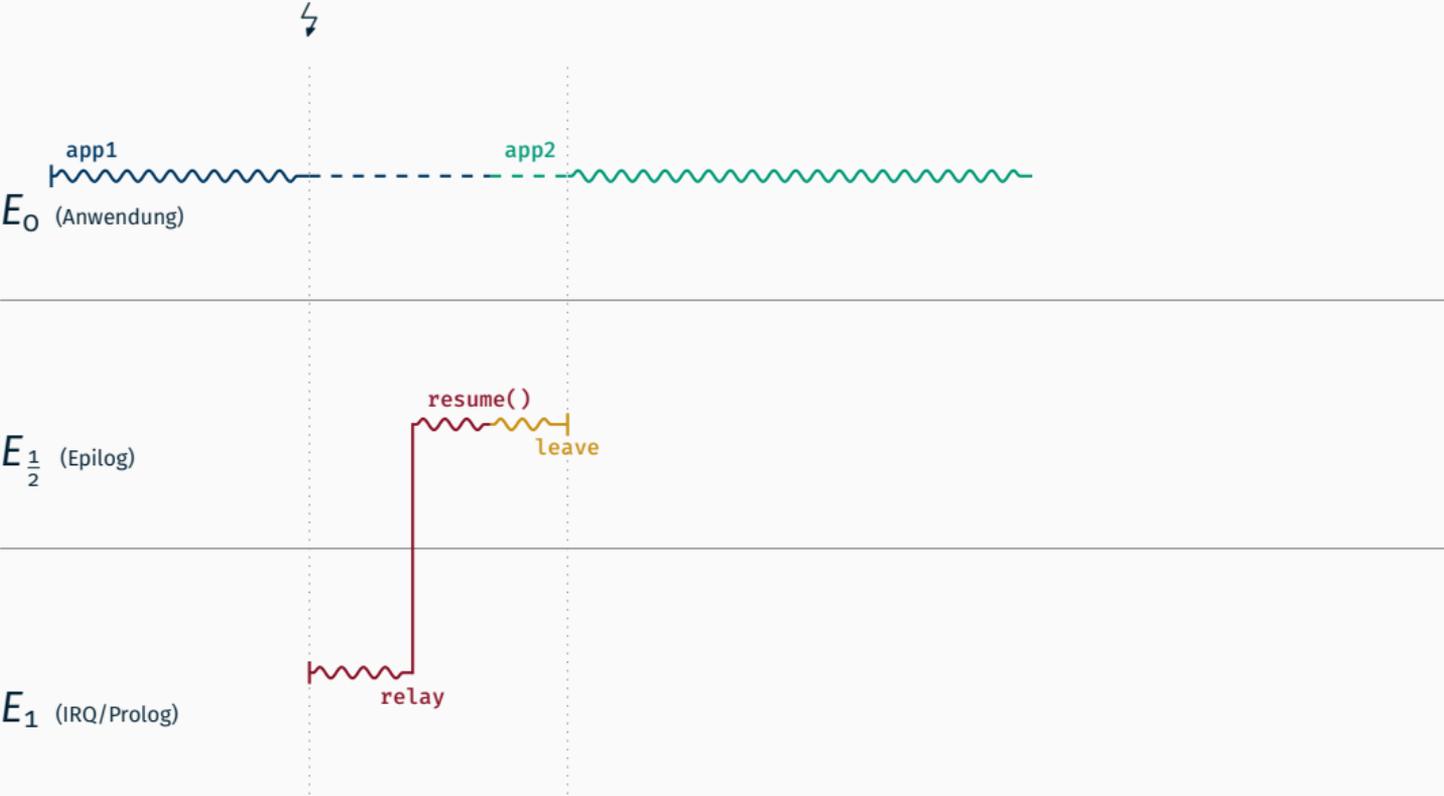
# Ablaufbeispiel (Standardfall)



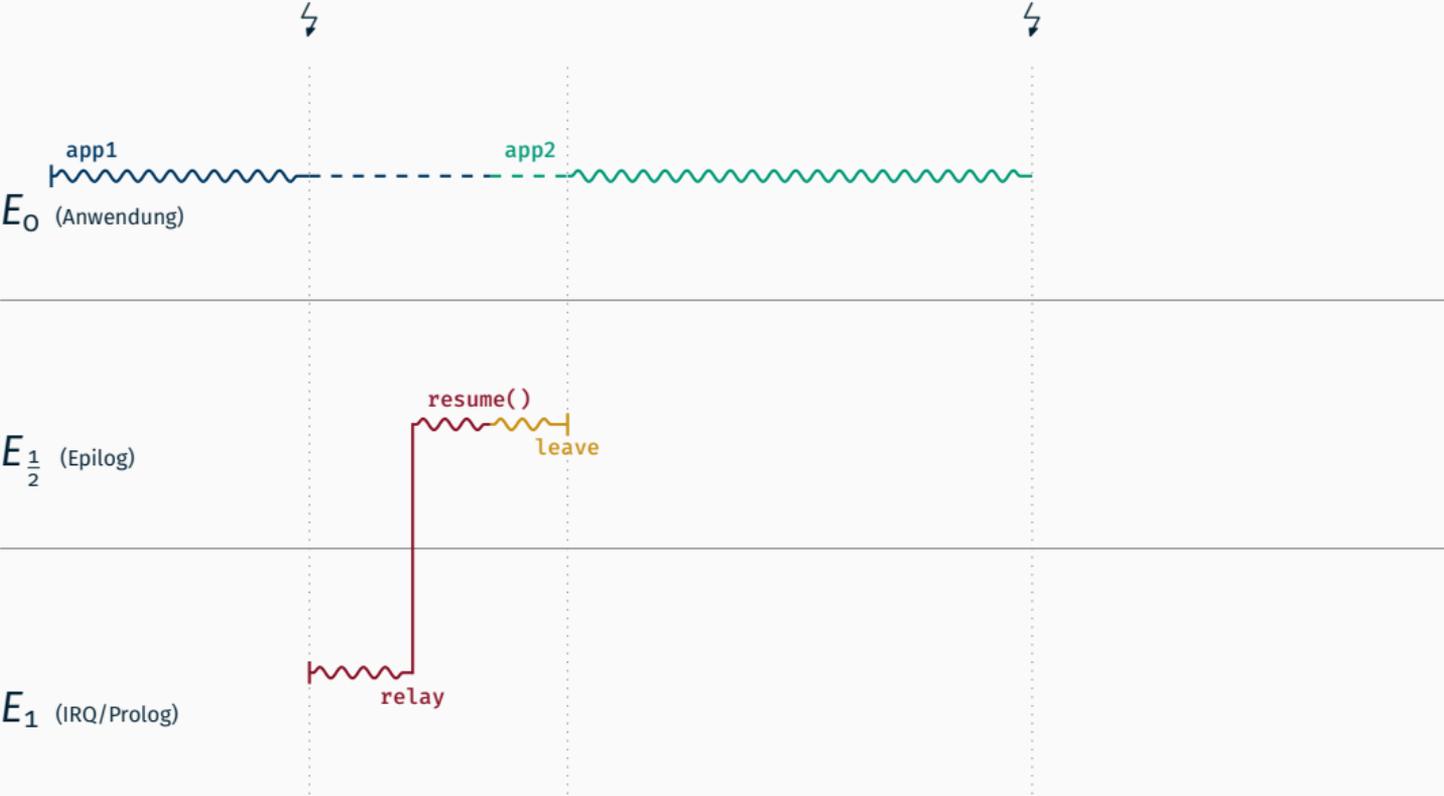
# Ablaufbeispiel (Standardfall)



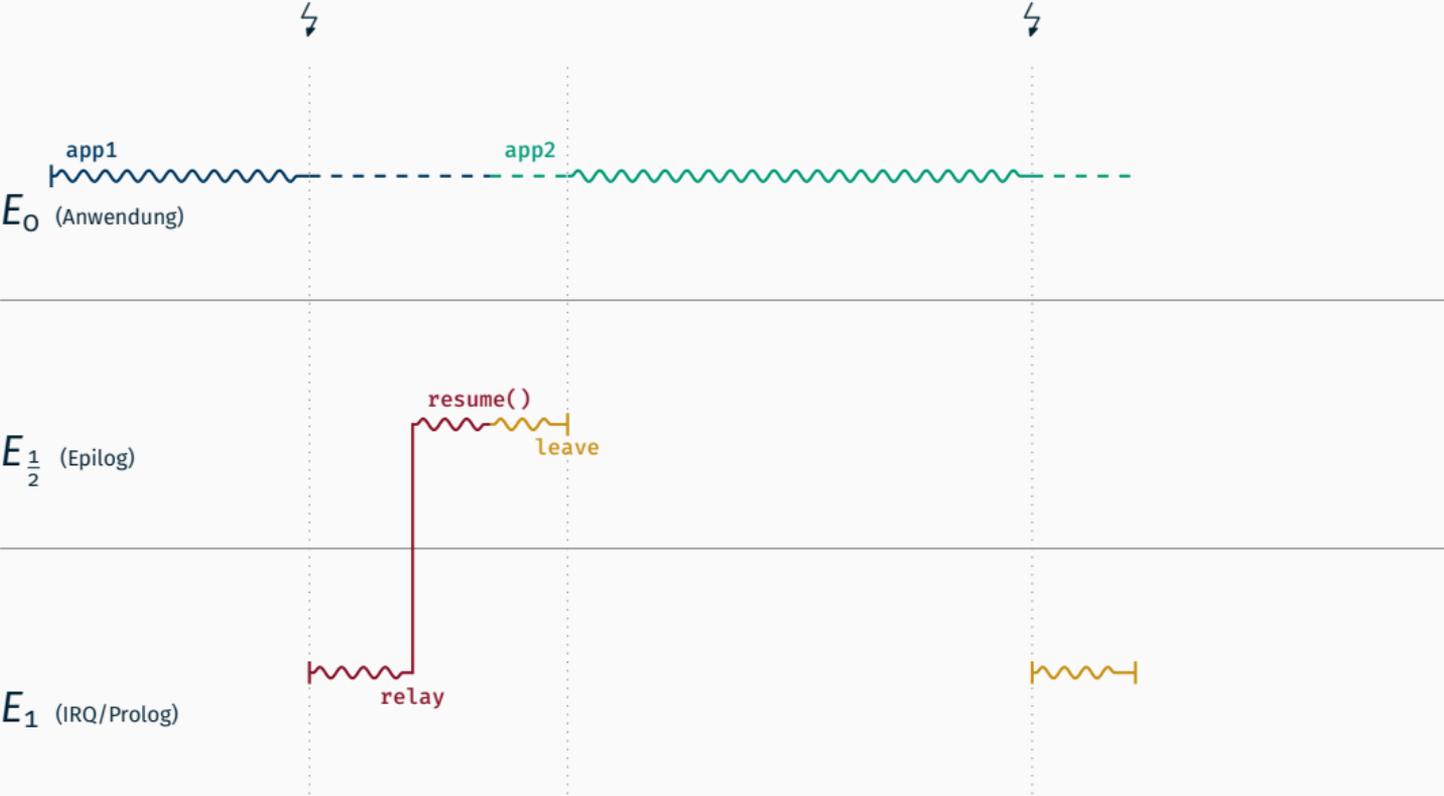
# Ablaufbeispiel (Standardfall)



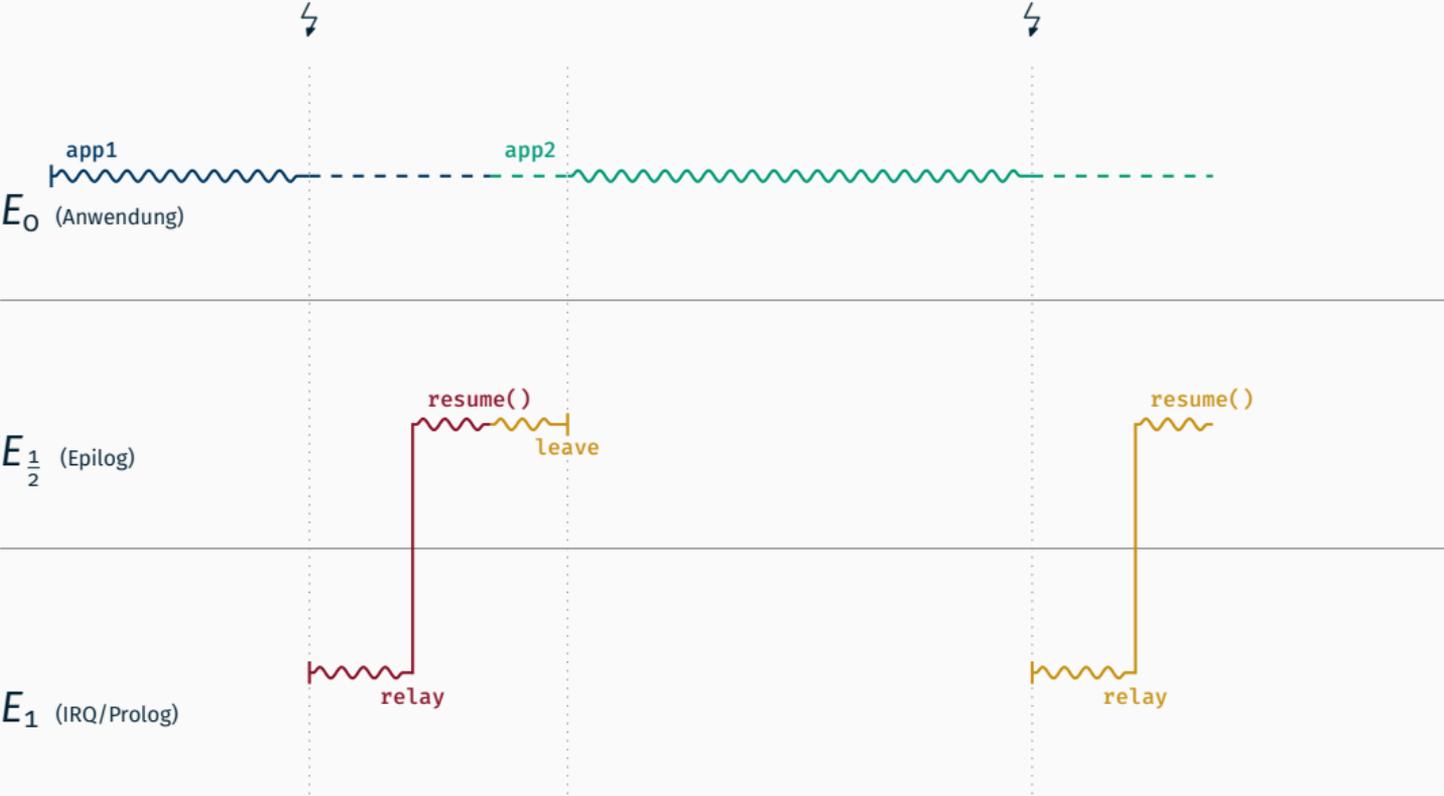
# Ablaufbeispiel (Standardfall)



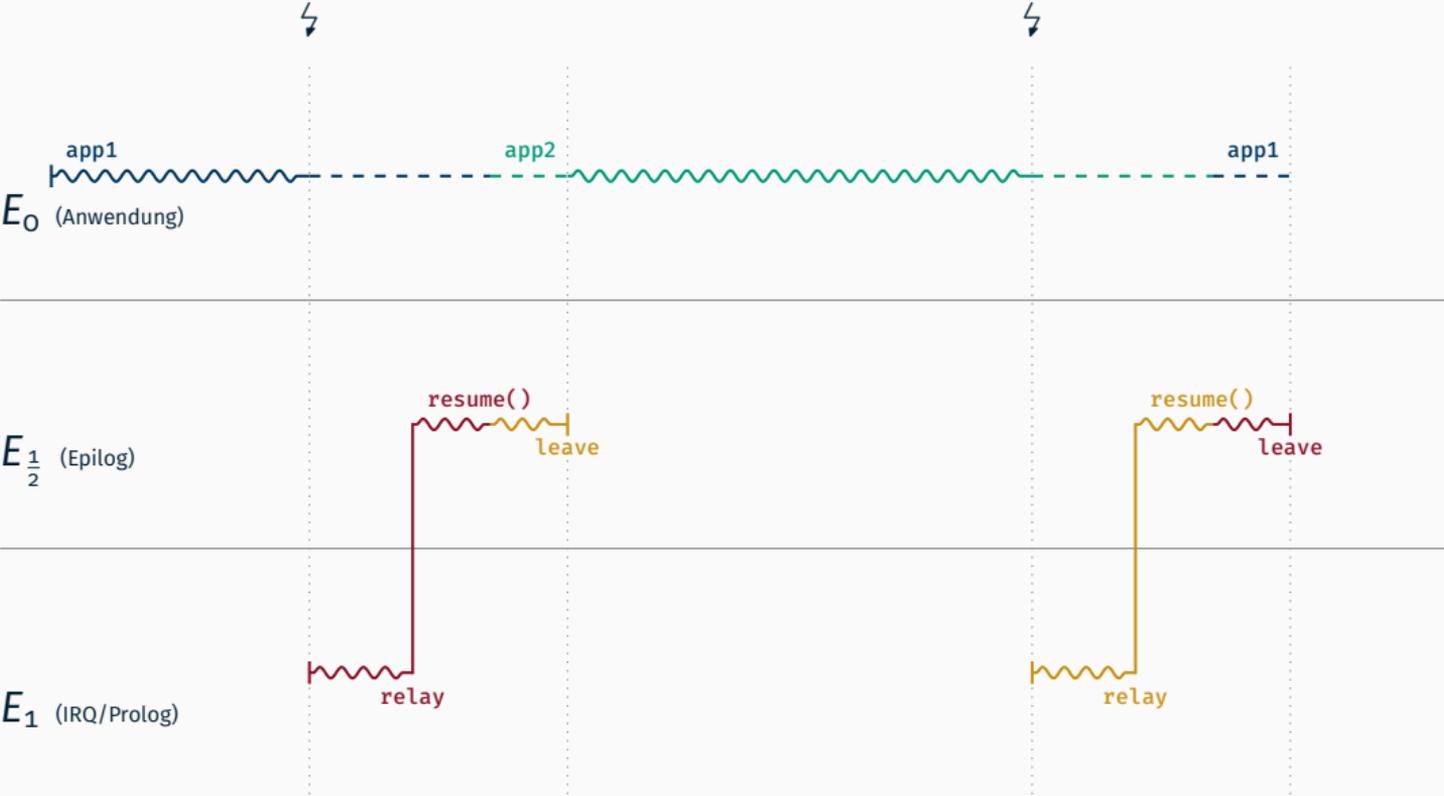
# Ablaufbeispiel (Standardfall)



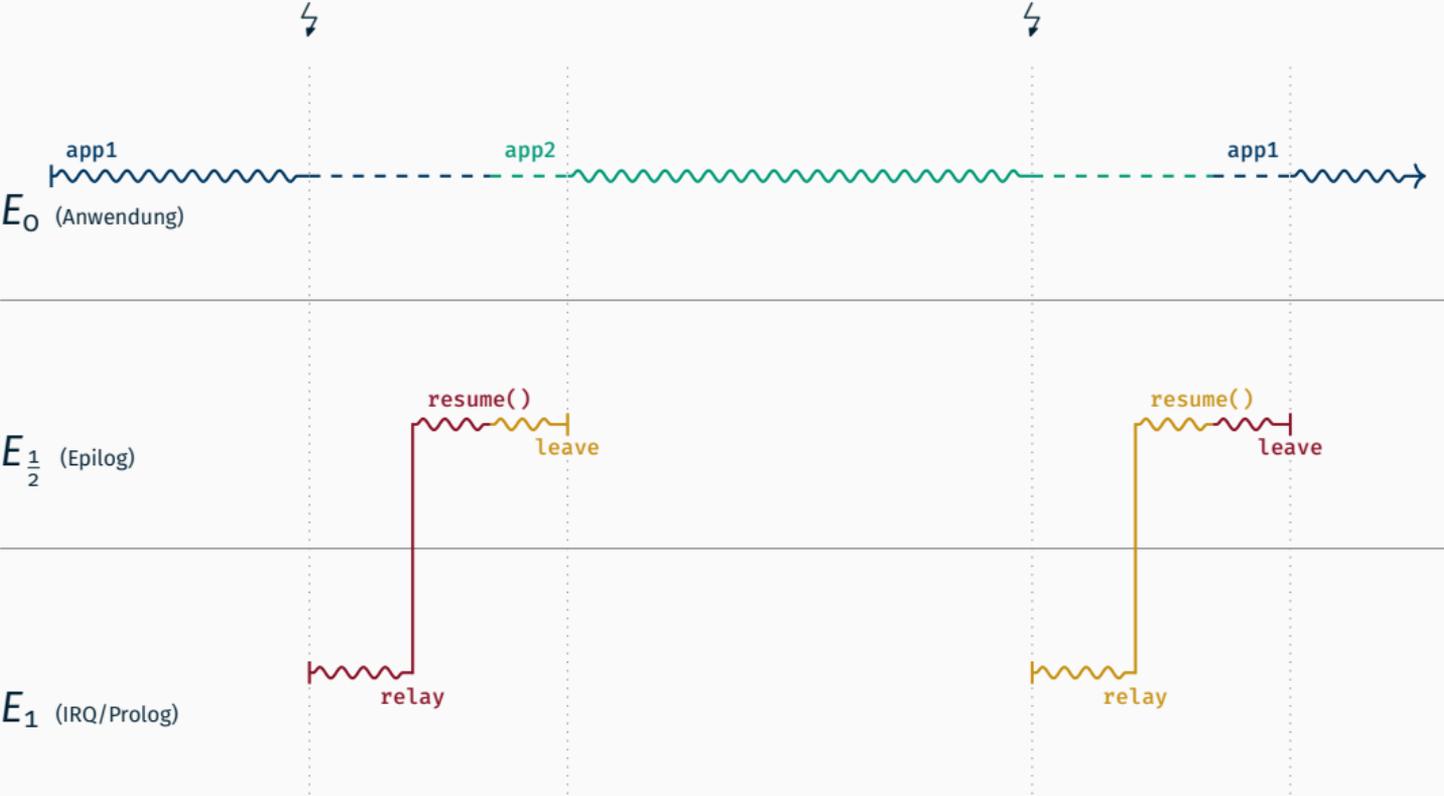
# Ablaufbeispiel (Standardfall)



# Ablaufbeispiel (Standardfall)



# Ablaufbeispiel (Standardfall)



# Ablaufbeispiel bei Faden in Systemebene

$E_0$  (Anwendung) <sup>app1</sup>  


---

$E_{\frac{1}{2}}$  (Epilog)

---

$E_1$  (IRQ/Prolog)

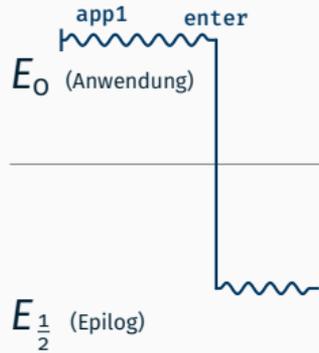
# Ablaufbeispiel bei Faden in Systemebene

app1 enter  
 $E_0$  (Anwendung)

$E_{\frac{1}{2}}$  (Epilog)

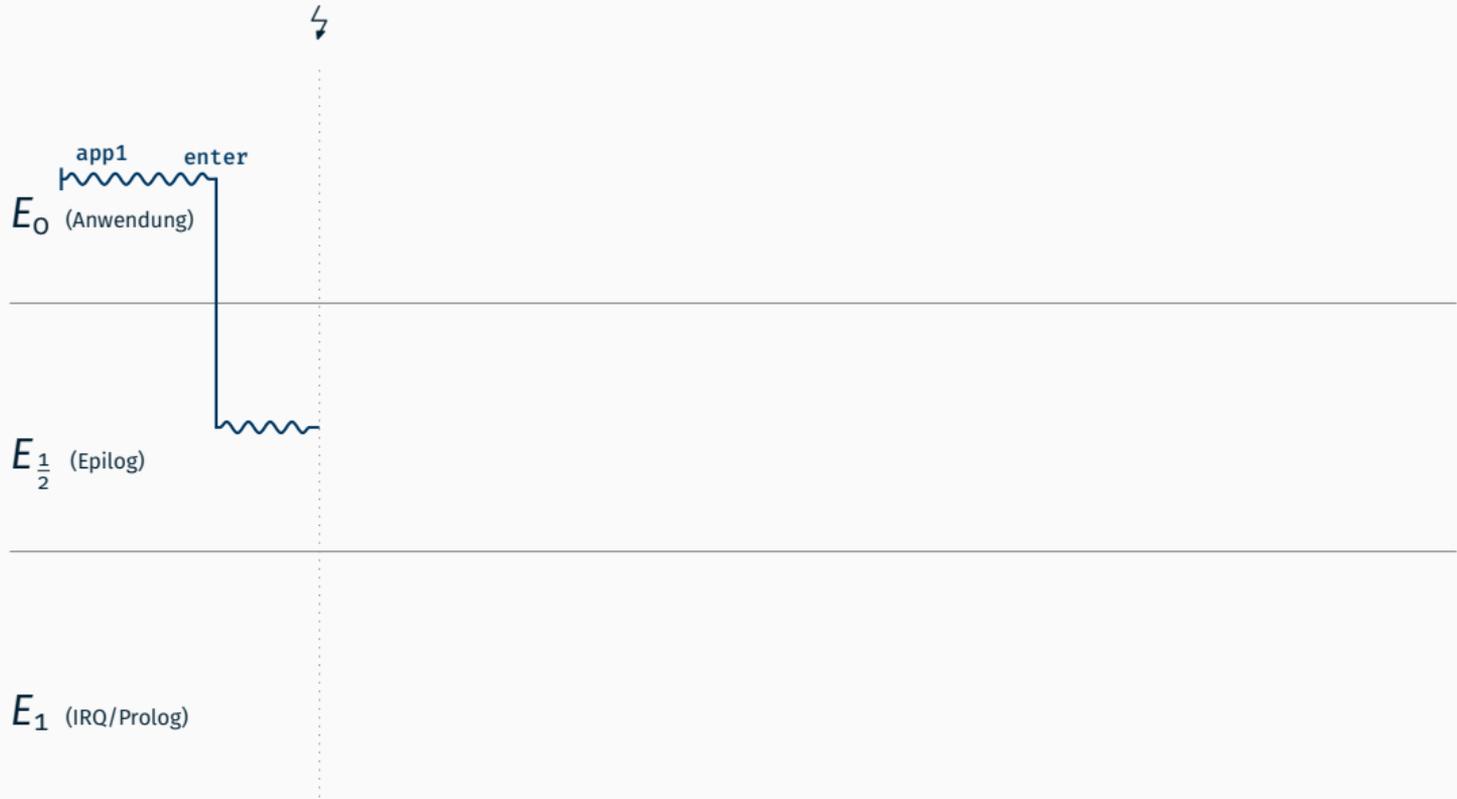
$E_1$  (IRQ/Prolog)

# Ablaufbeispiel bei Faden in Systemebene



$E_1$  (IRQ/Prolog)

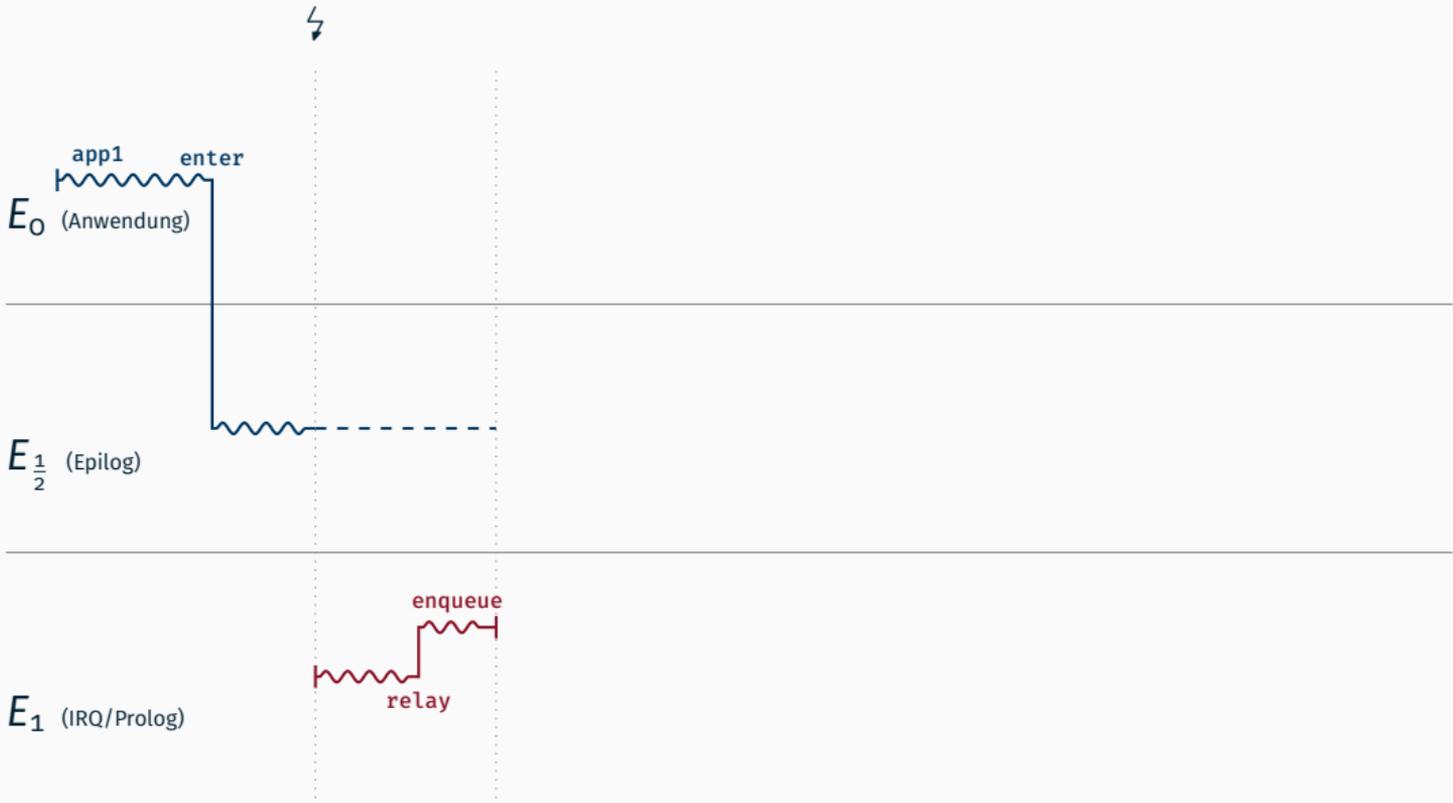
# Ablaufbeispiel bei Faden in Systemebene



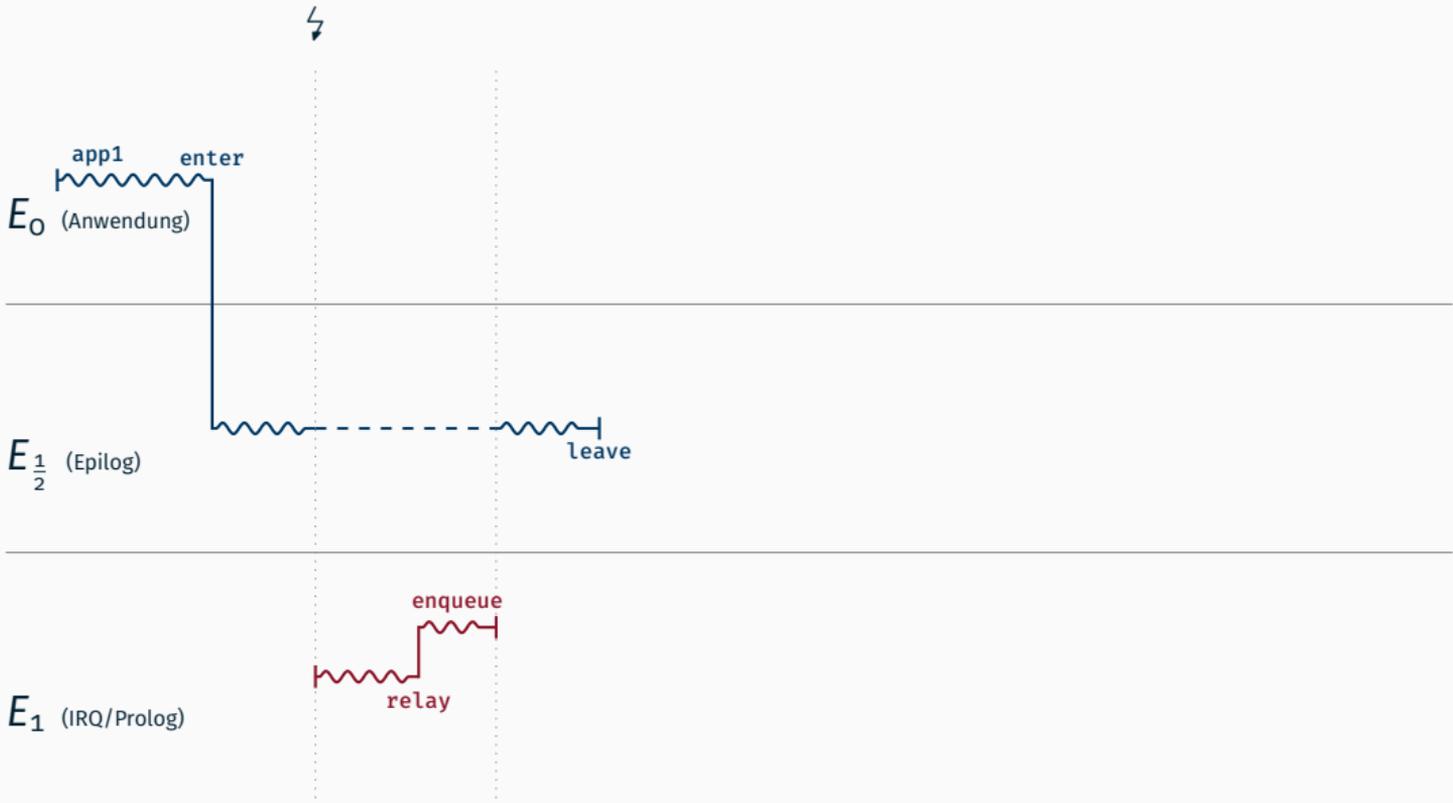
# Ablaufbeispiel bei Faden in Systemebene



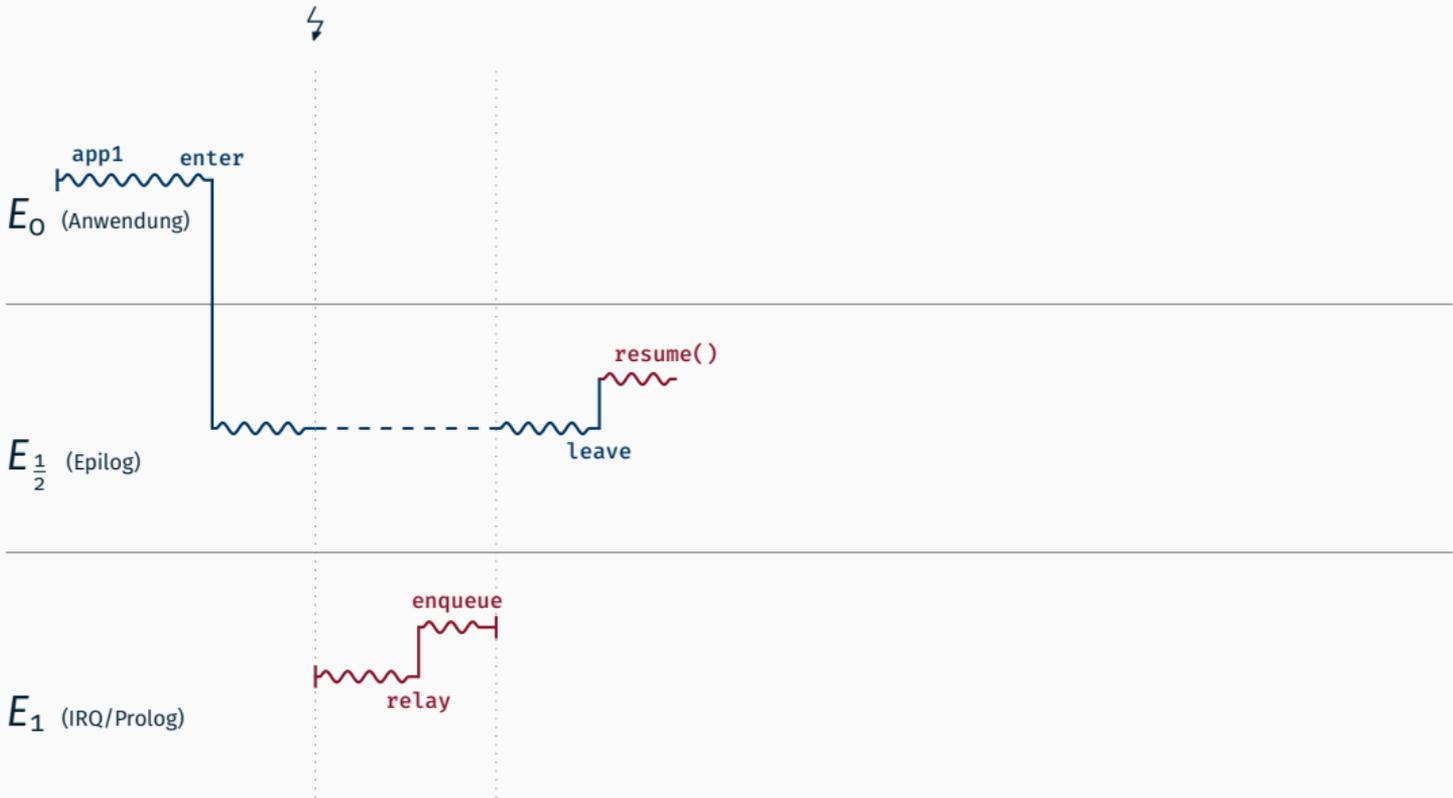
# Ablaufbeispiel bei Faden in Systemebene



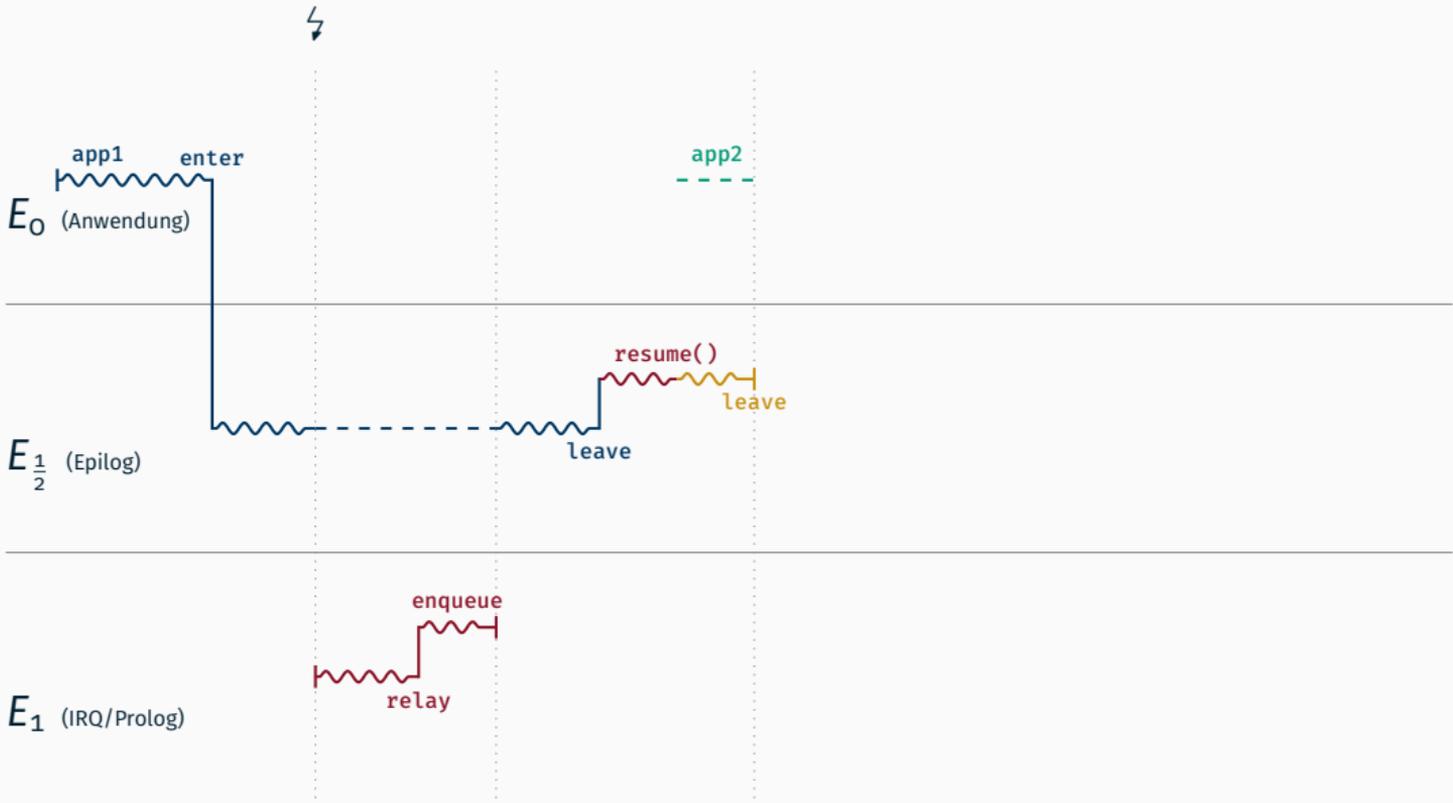
# Ablaufbeispiel bei Faden in Systemebene



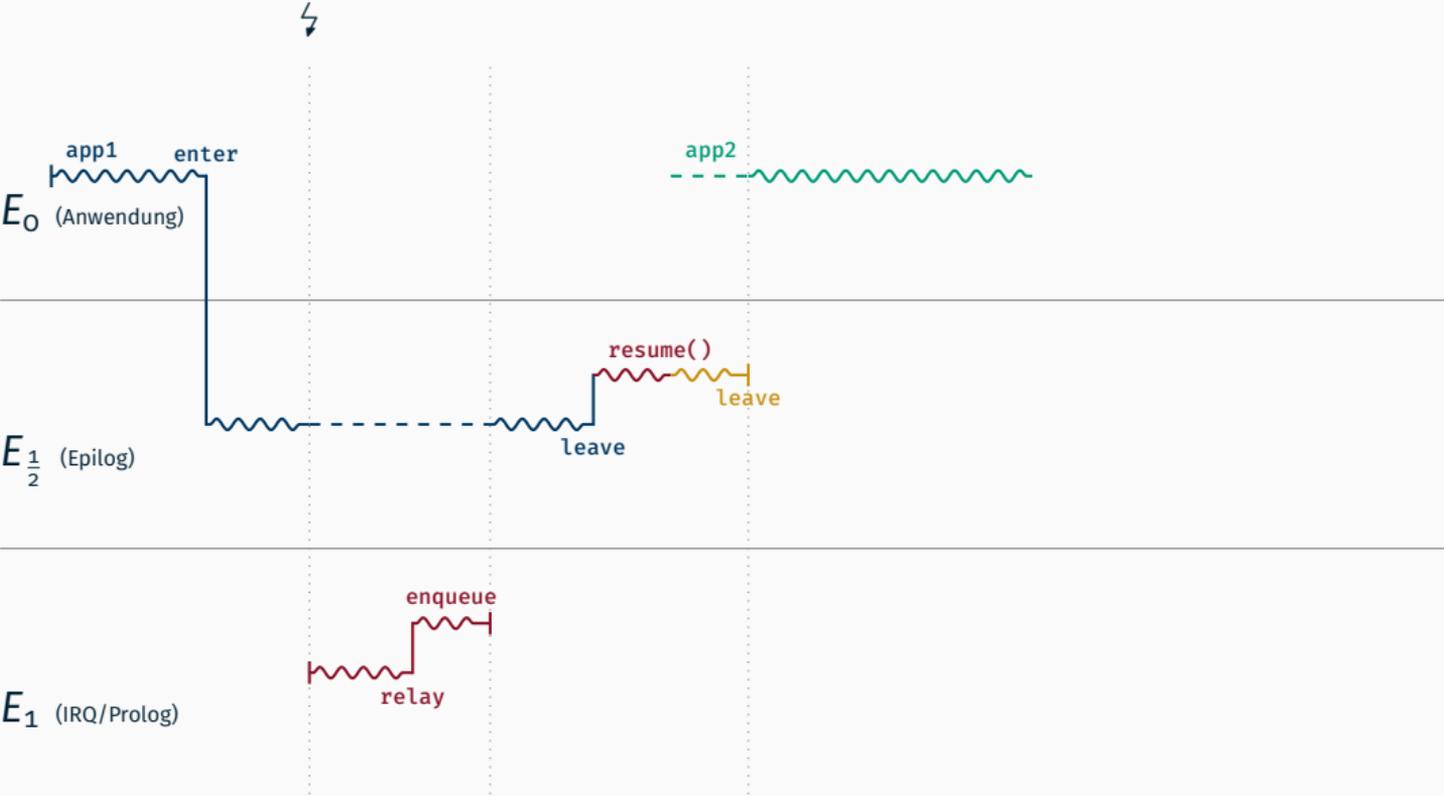
# Ablaufbeispiel bei Faden in Systemebene



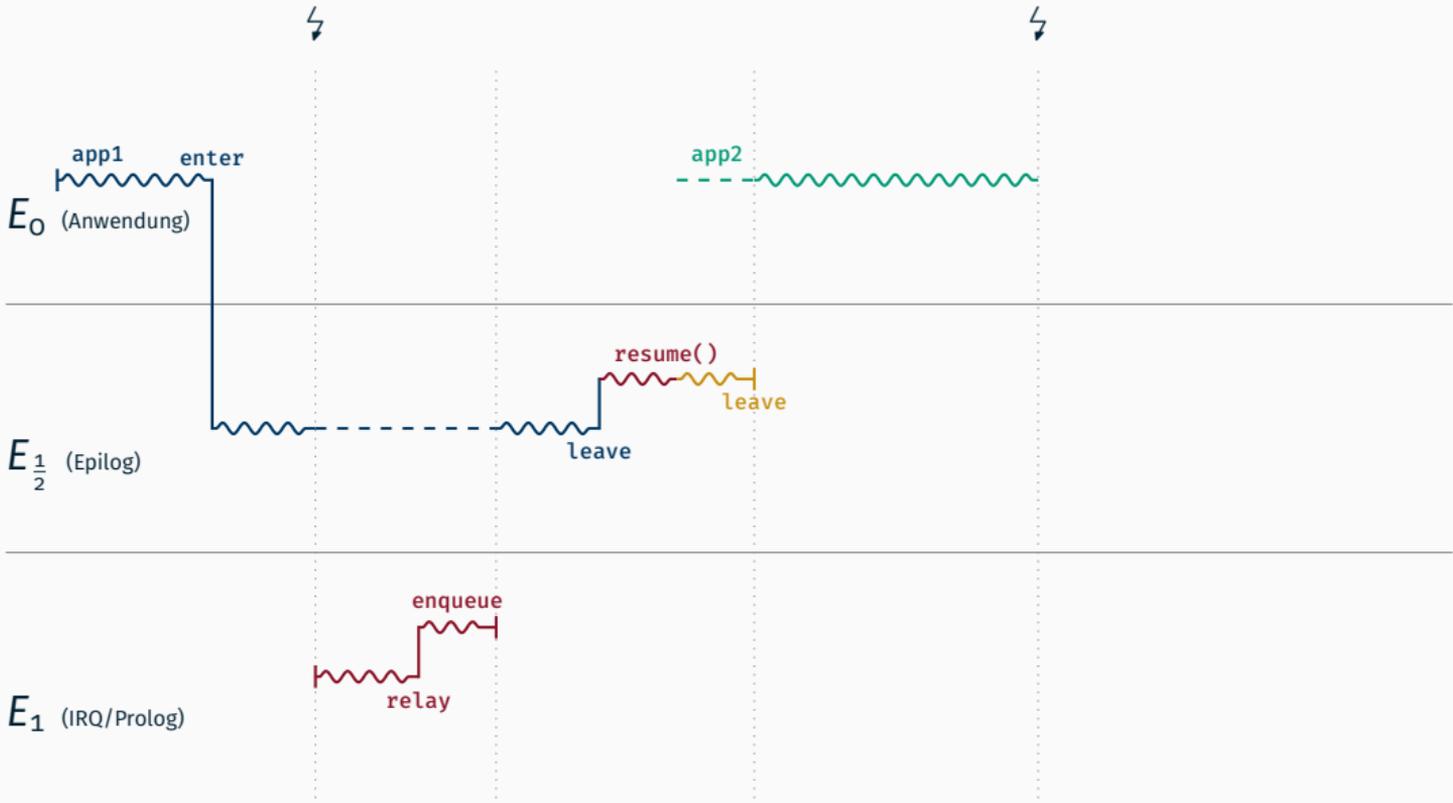
# Ablaufbeispiel bei Faden in Systemebene



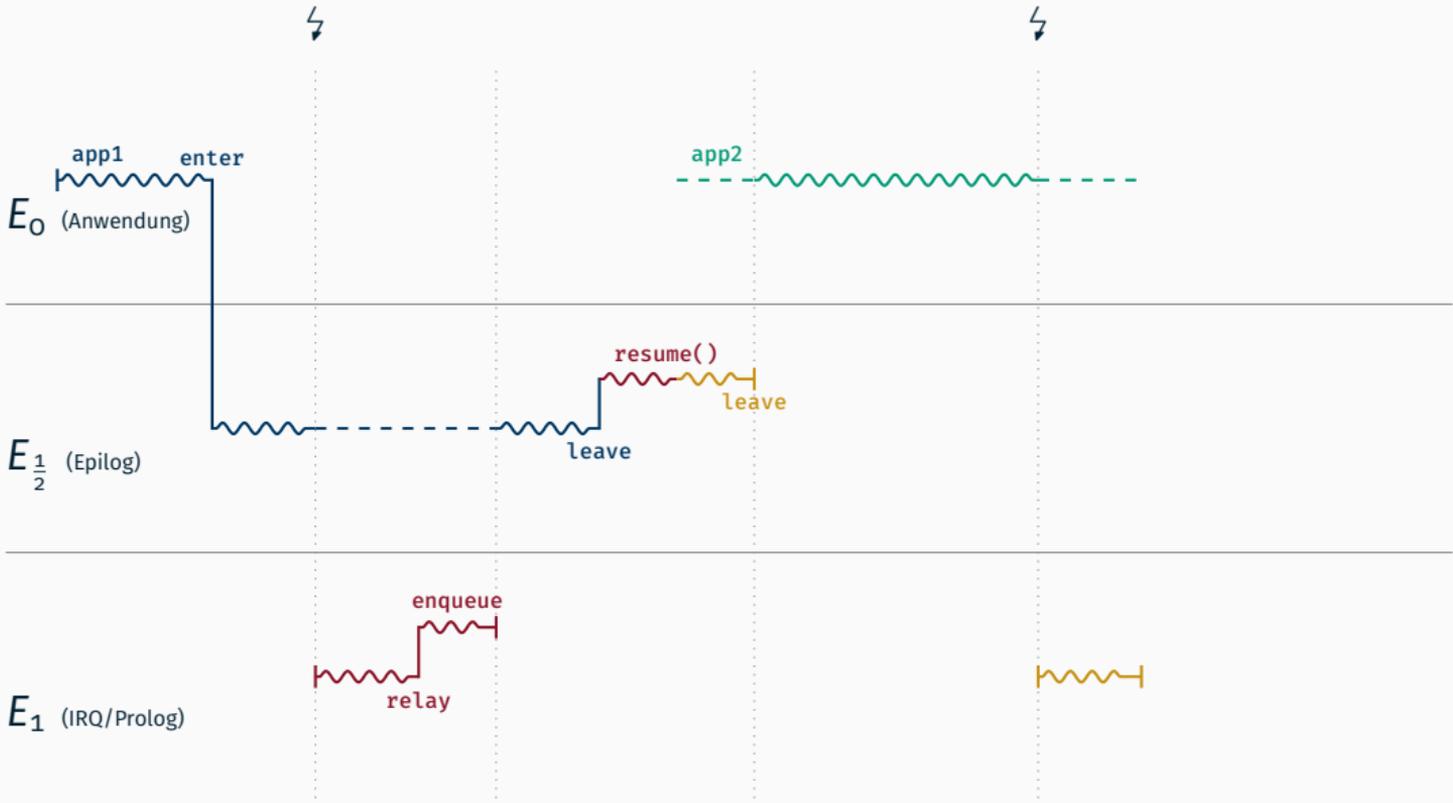
# Ablaufbeispiel bei Faden in Systemebene



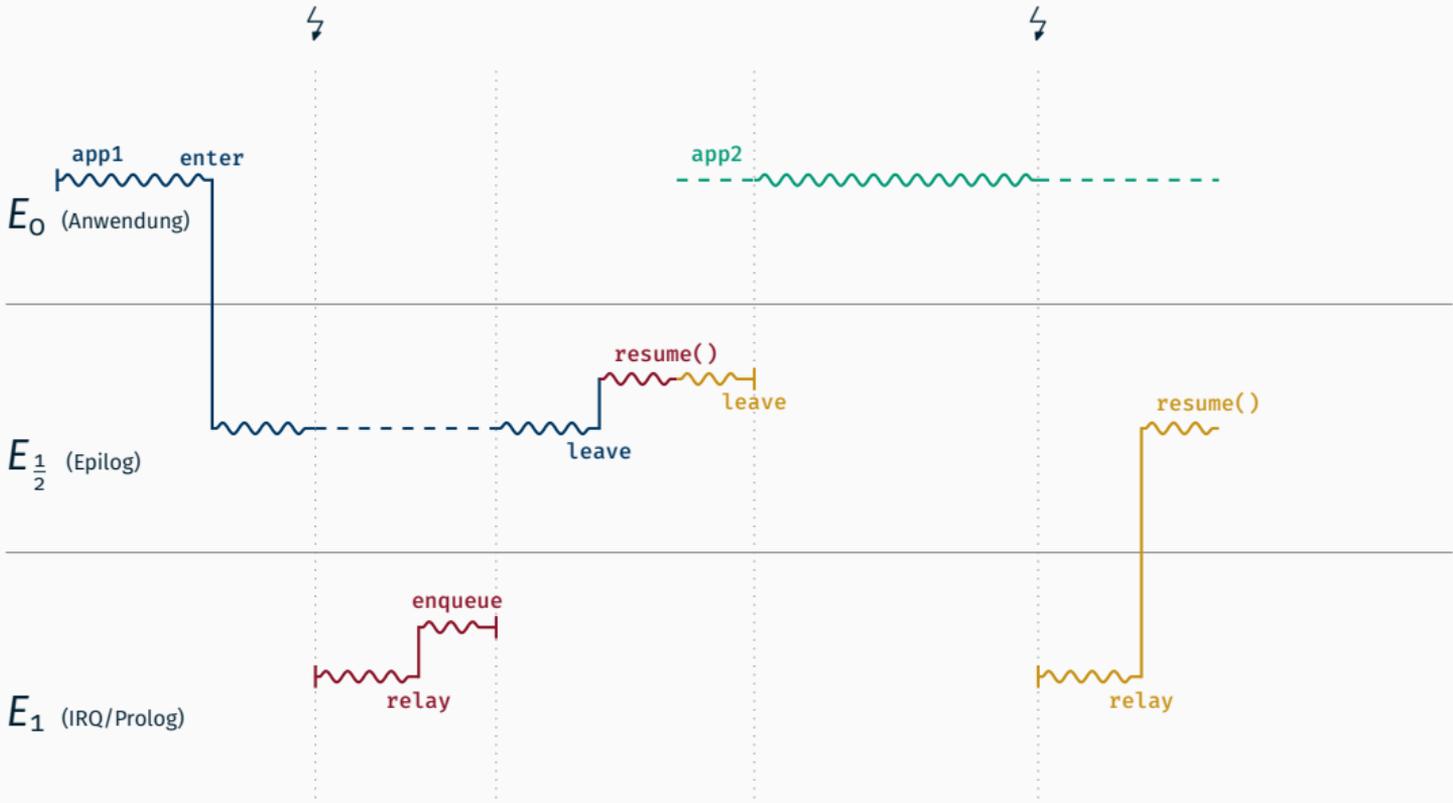
# Ablaufbeispiel bei Faden in Systemebene



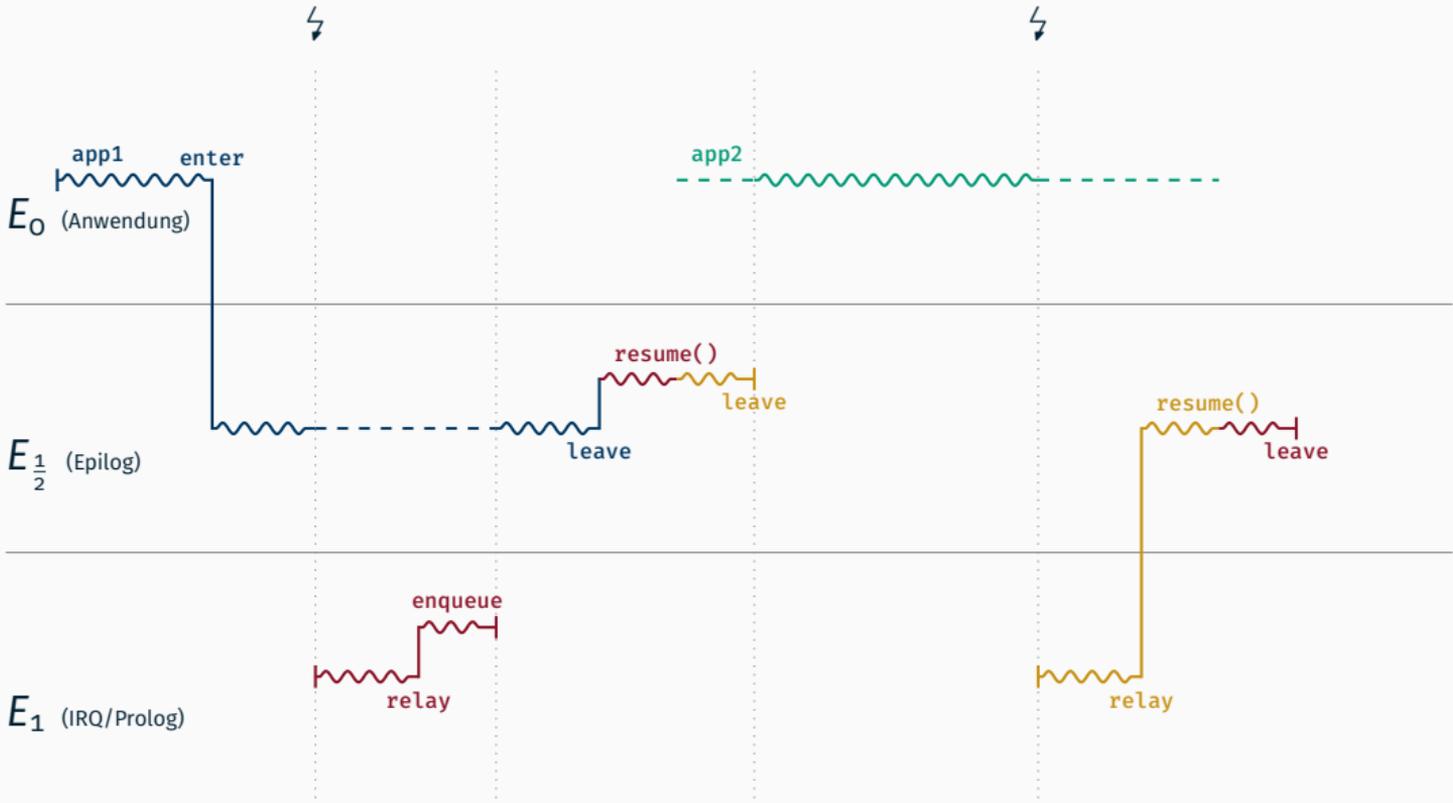
# Ablaufbeispiel bei Faden in Systemebene



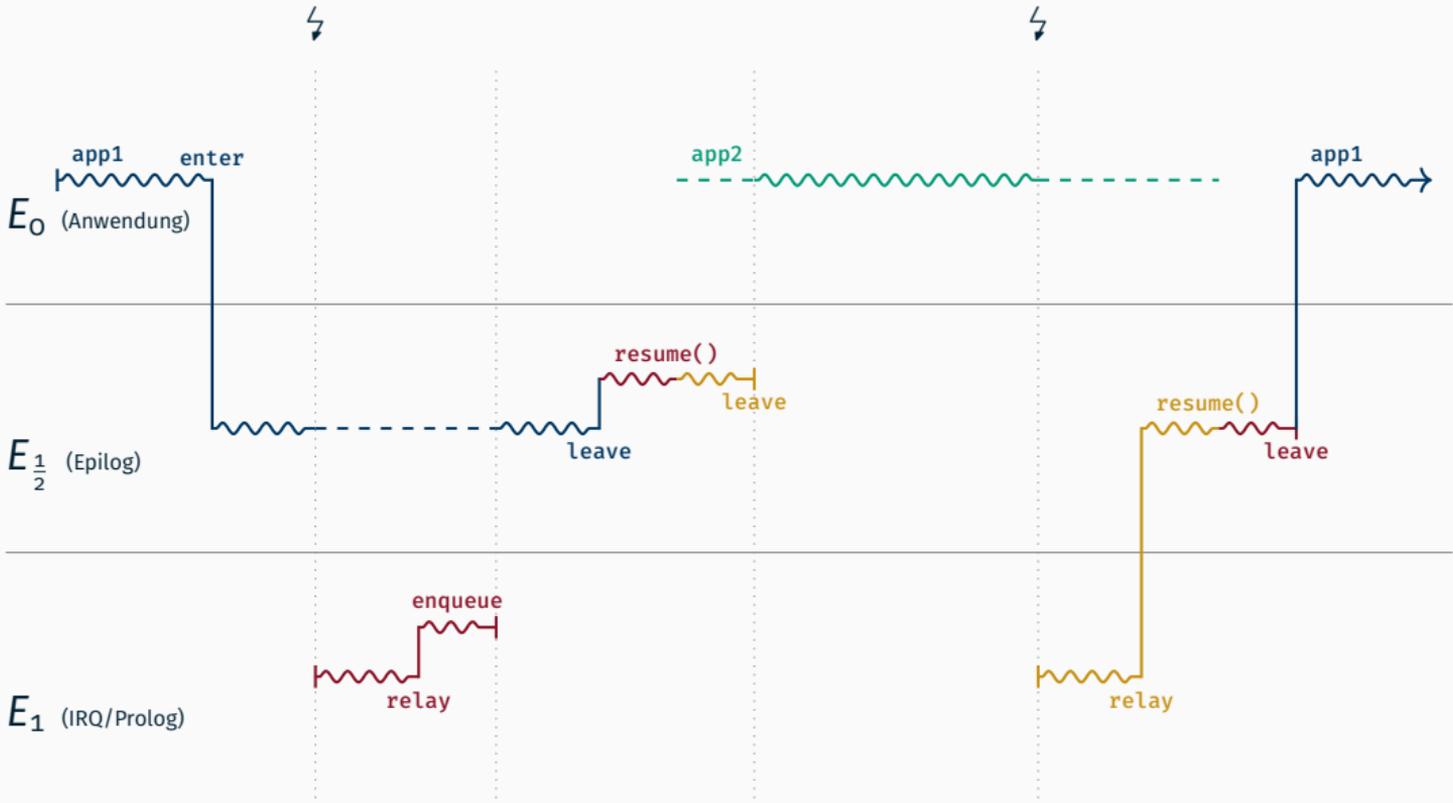
# Ablaufbeispiel bei Faden in Systemebene



# Ablaufbeispiel bei Faden in Systemebene



# Ablaufbeispiel bei Faden in Systemebene



# Ablaufbeispiel mit neuem Thread

$E_0$  (Anwendung) <sup>app1</sup>  


---

$E_{\frac{1}{2}}$  (Epilog)

---

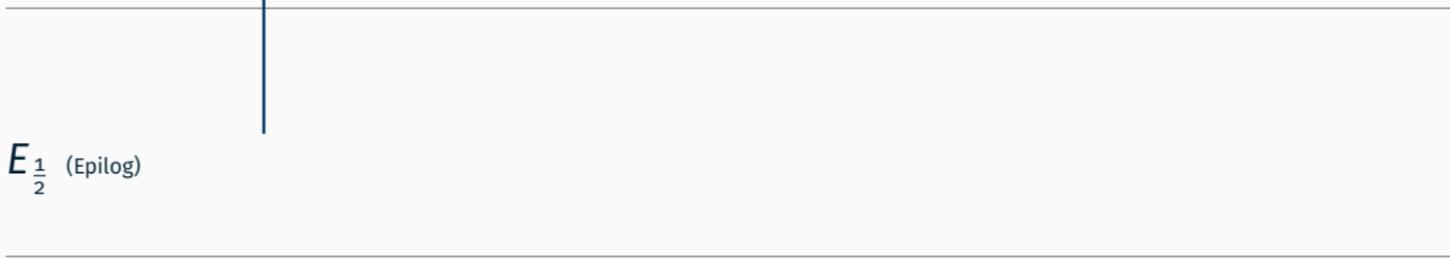
$E_1$  (IRQ/Prolog)

# Ablaufbeispiel mit neuem Thread

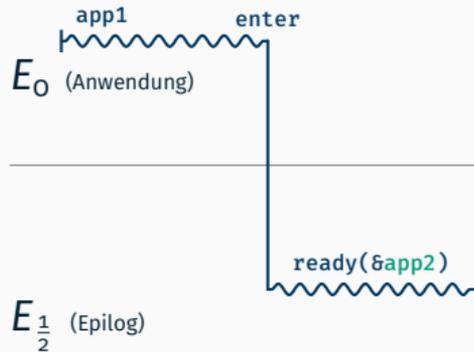
app1  
enter  
 $E_0$  (Anwendung)

$E_{\frac{1}{2}}$  (Epilog)

$E_1$  (IRQ/Prolog)

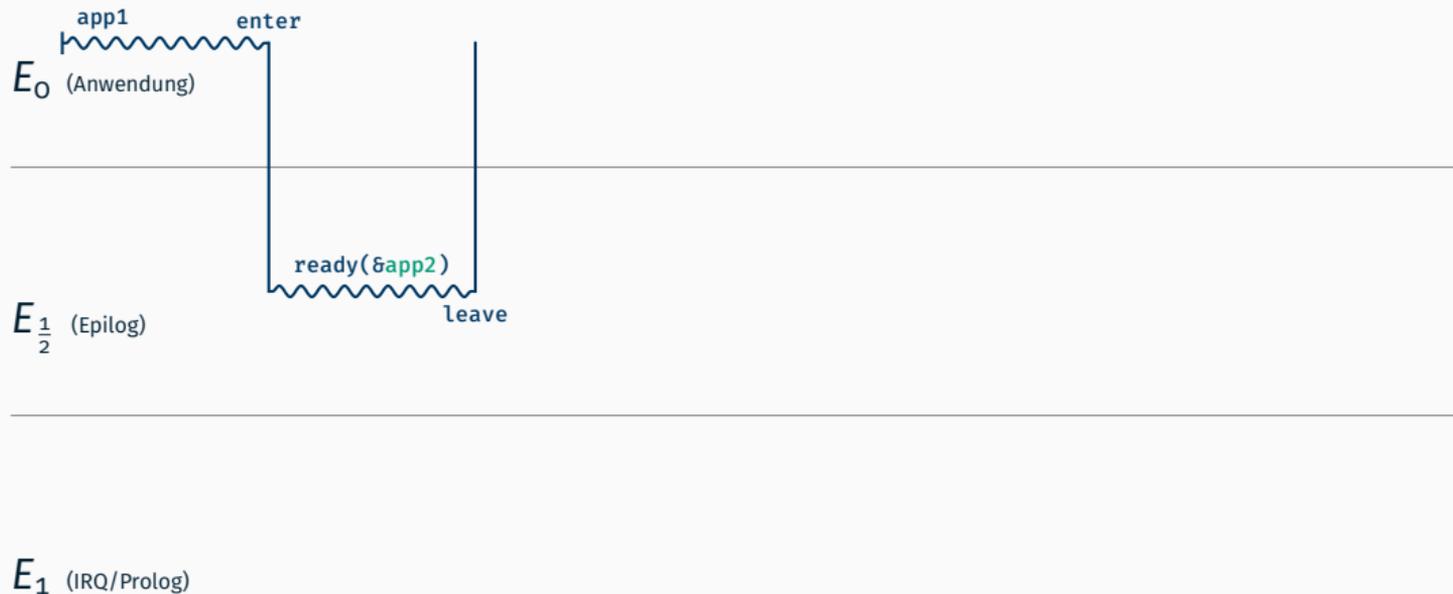


# Ablaufbeispiel mit neuem Thread

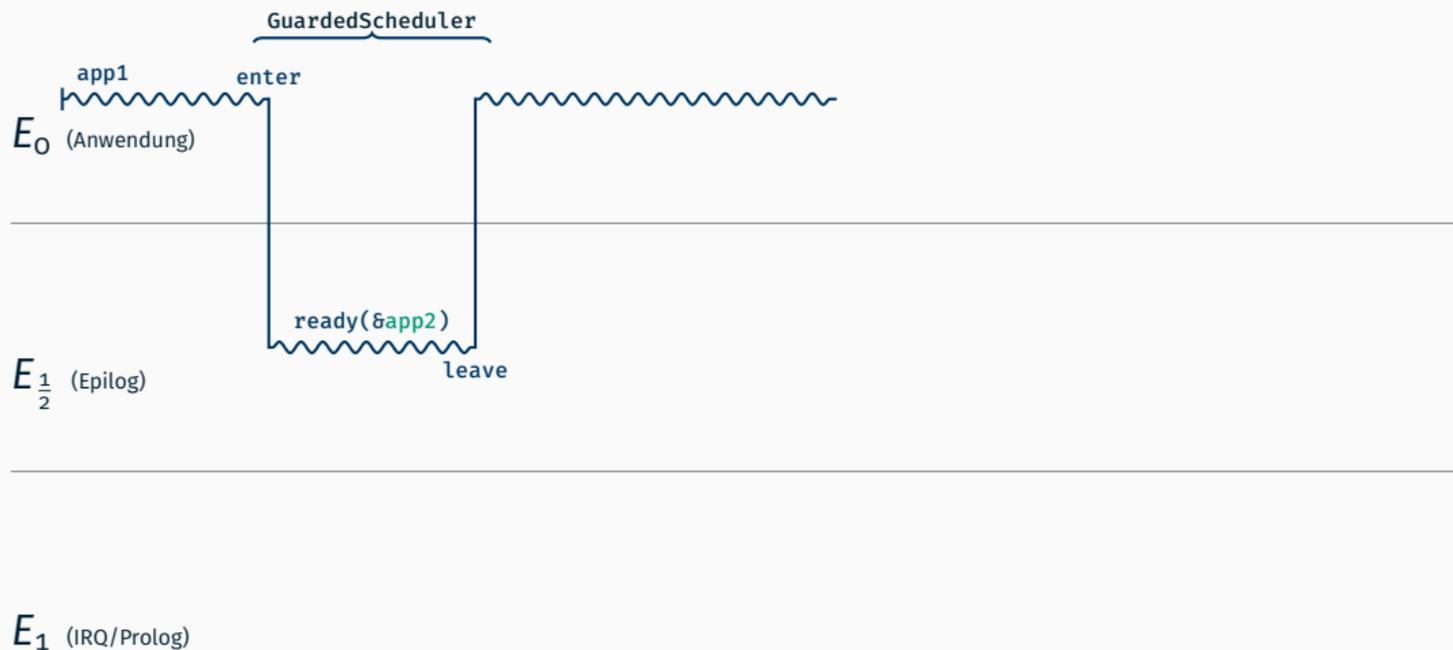


$E_1$  (IRQ/Prolog)

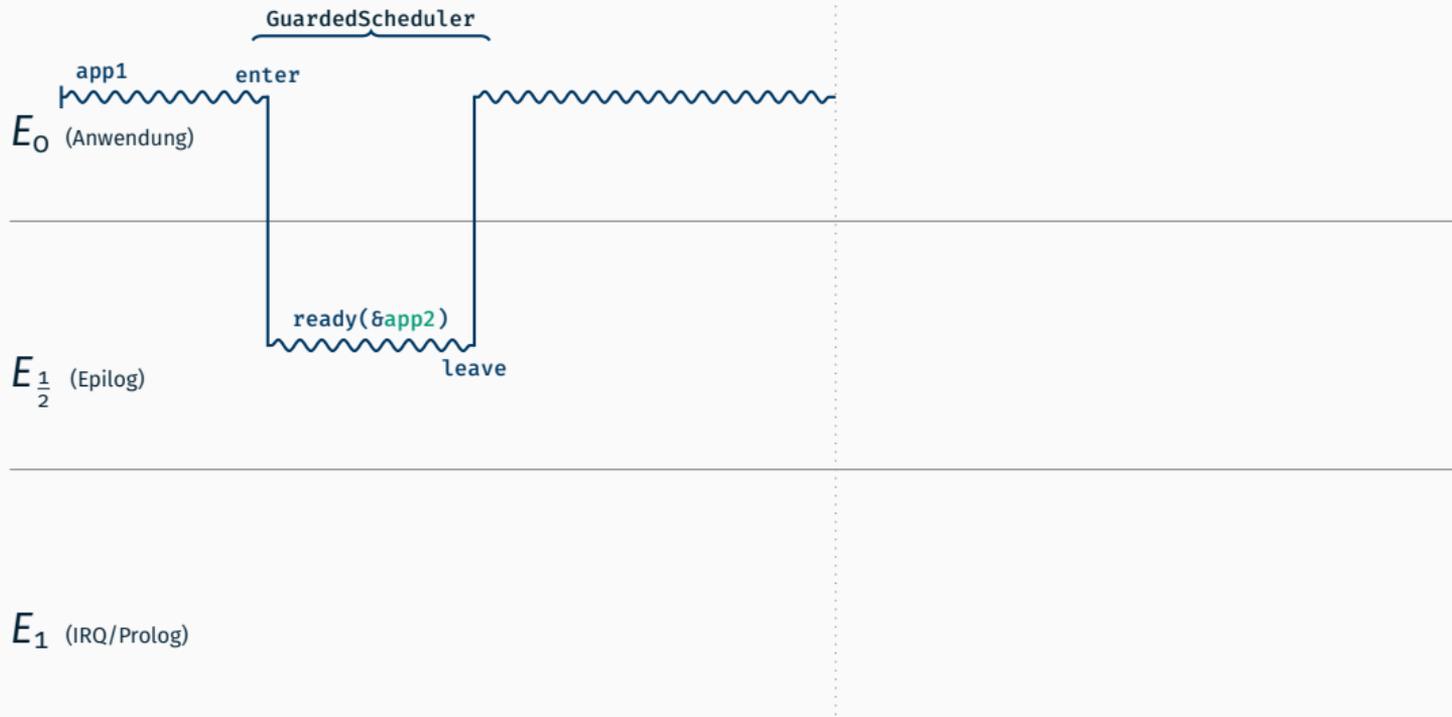
# Ablaufbeispiel mit neuem Thread



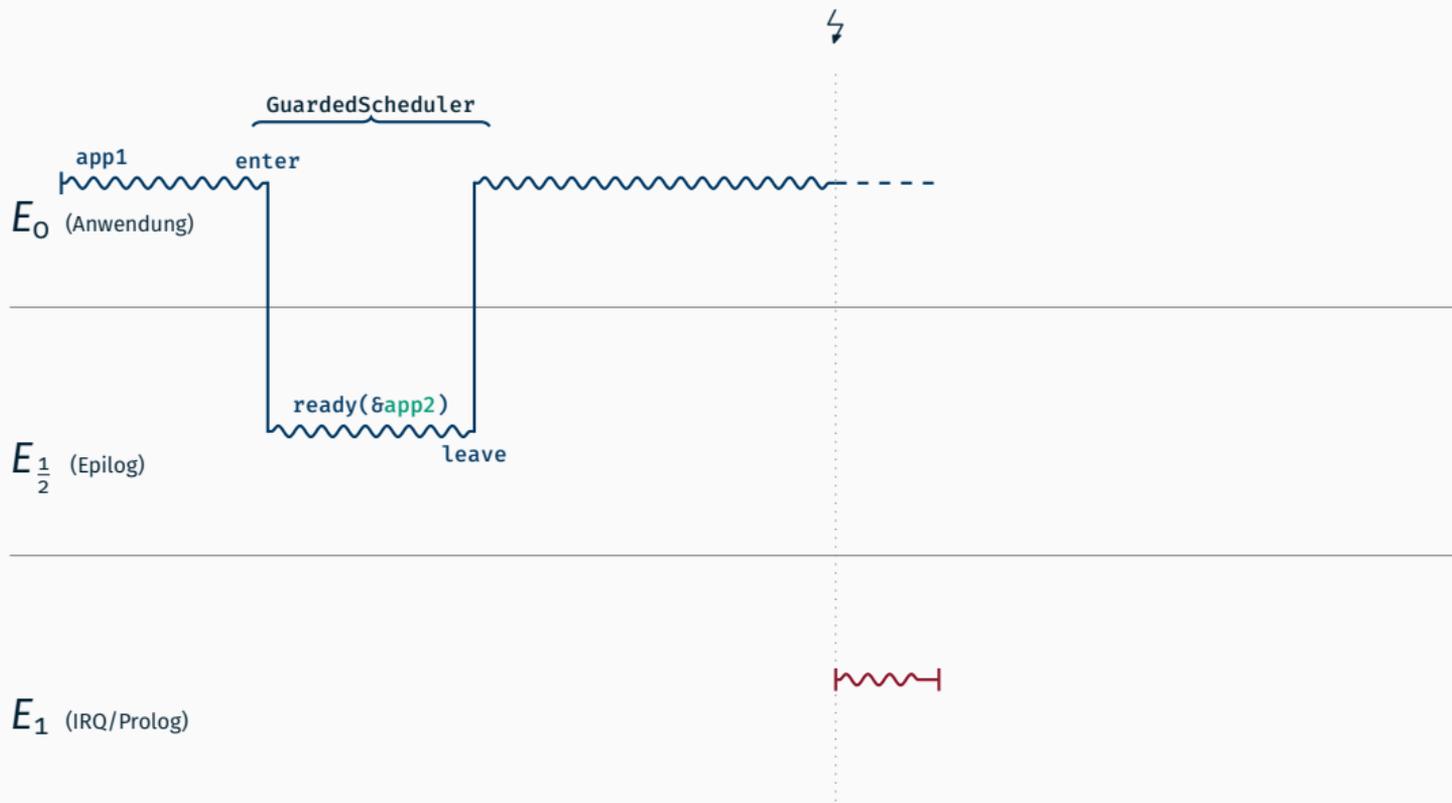
# Ablaufbeispiel mit neuem Thread



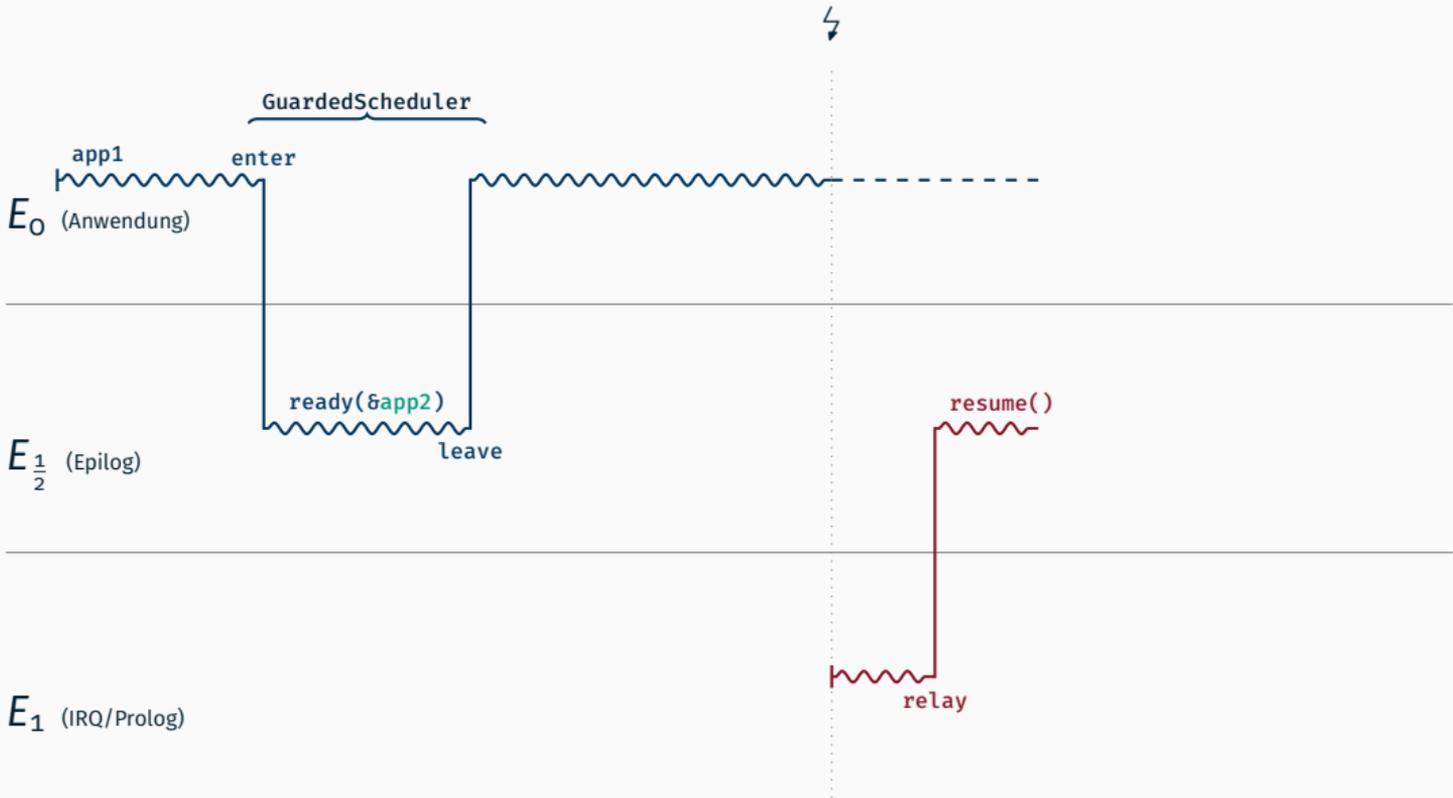
# Ablaufbeispiel mit neuem Thread



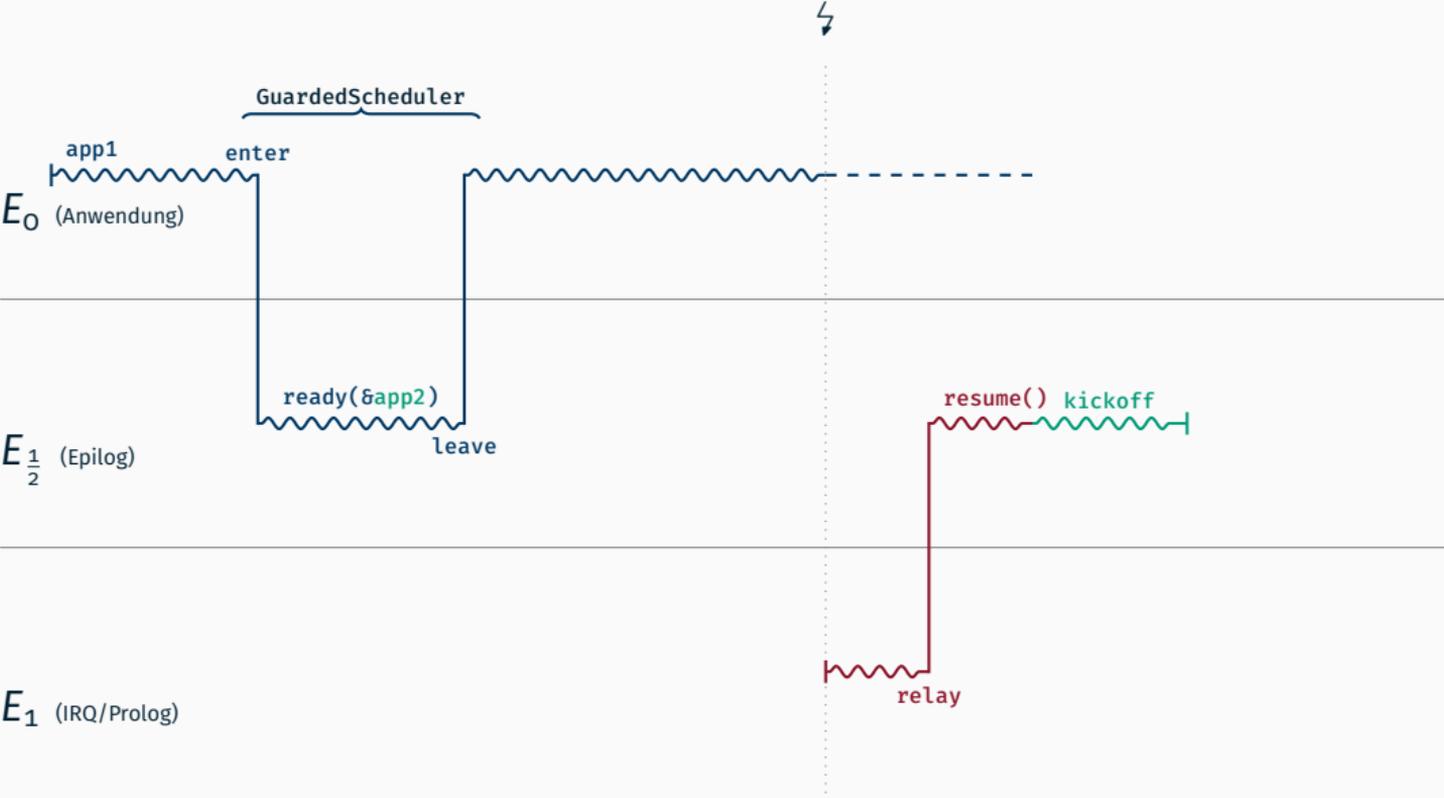
# Ablaufbeispiel mit neuem Thread



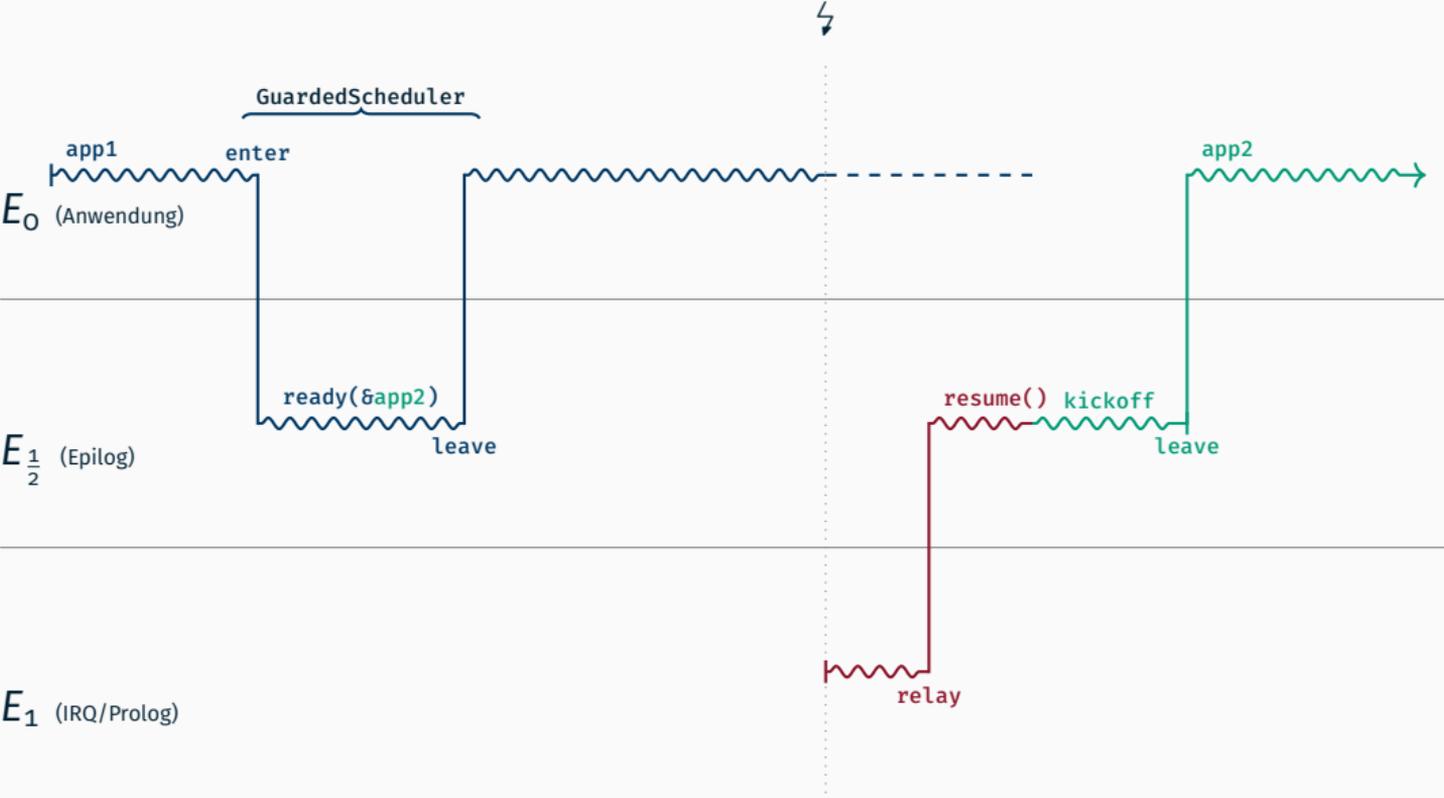
# Ablaufbeispiel mit neuem Thread



# Ablaufbeispiel mit neuem Thread



# Ablaufbeispiel mit neuem Thread



# Besonderheiten

---

## Anwendungsfaden beenden

Präemptives Beenden mittels `Scheduler::kill(Thread&)`

## Anwendungsfaden beenden

Präemptives Beenden mittels `Scheduler::kill(Thread&)`

**OOSTuBS** *keine Änderung*

## Anwendungsfaden beenden

Präemptives Beenden mittels `Scheduler::kill(Thread&)`

**OOSTuBS** *keine Änderung*: aus der Ready-Liste entfernen  
bzw. Kill-Flag setzen und bei resume prüfen

## Anwendungsfaden beenden

Präemptives Beenden mittels `Scheduler::kill(Thread&)`

**OOSTuBS** *keine Änderung*: aus der Ready-Liste entfernen  
bzw. Kill-Flag setzen und bei resume prüfen

**MPStuBS** der Anwendungsfaden kann gerade auf einer anderen CPU  
laufen

# Anwendungsfaden beenden

Präemptives Beenden mittels `Scheduler::kill(Thread&)`

**OOSTuBS** *keine Änderung*: aus der Ready-Liste entfernen  
bzw. Kill-Flag setzen und bei `resume` prüfen

**MPStuBS** der Anwendungsfaden kann gerade auf einer anderen CPU  
laufen

- Kill-Flag setzen (wie gehabt)

# Anwendungsfaden beenden

Präemptives Beenden mittels `Scheduler::kill(Thread&)`

**OOSTuBS** *keine Änderung*: aus der Ready-Liste entfernen  
bzw. Kill-Flag setzen und bei `resume` prüfen

**MPStuBS** der Anwendungsfaden kann gerade auf einer anderen CPU  
laufen

- Kill-Flag setzen (wie gehabt)
- falls er nicht in der Ready-Liste ist, läuft er wohl gerade auf einer anderen CPU

# Anwendungsfaden beenden

Präemptives Beenden mittels `Scheduler::kill(Thread&)`

**OOSTuBS** *keine Änderung*: aus der Ready-Liste entfernen  
bzw. Kill-Flag setzen und bei `resume` prüfen

**MPStuBS** der Anwendungsfaden kann gerade auf einer anderen CPU  
laufen

- Kill-Flag setzen (wie gehabt)
- falls er nicht in der Ready-Liste ist, läuft er wohl gerade auf einer anderen CPU
- diese andere CPU muss benachrichtigt werden

# Anwendungsfaden beenden

Präemptives Beenden mittels `Scheduler::kill(Thread&)`

**OOSTuBS** *keine Änderung*: aus der Ready-Liste entfernen  
bzw. Kill-Flag setzen und bei resume prüfen

**MPStuBS** der Anwendungsfaden kann gerade auf einer anderen CPU  
laufen

- Kill-Flag setzen (wie gehabt)
- falls er nicht in der Ready-Liste ist, läuft er wohl gerade auf einer anderen CPU
- diese andere CPU muss benachrichtigt werden  
→ INTER PROCESSOR INTERRUPT (IPI)

# Anwendungsfaden beenden

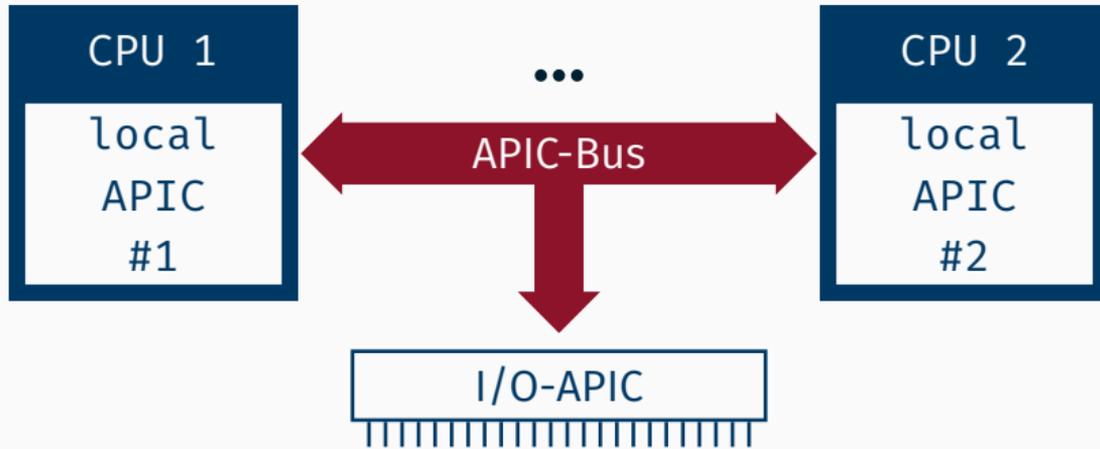
Präemptives Beenden mittels `Scheduler::kill(Thread&)`

**OOSTuBS** *keine Änderung*: aus der Ready-Liste entfernen  
bzw. Kill-Flag setzen und bei resume prüfen

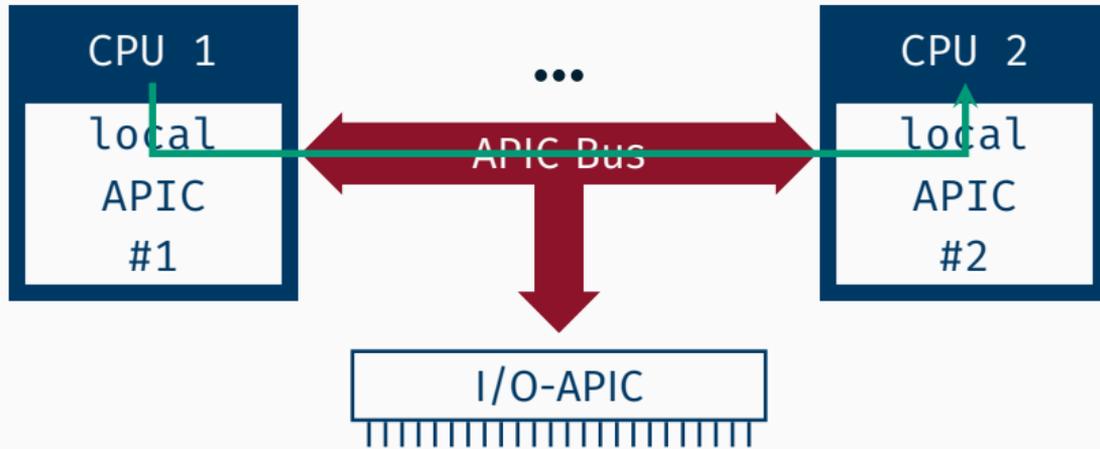
**MPStuBS** der Anwendungsfaden kann gerade auf einer anderen CPU  
laufen

- Kill-Flag setzen (wie gehabt)
- falls er nicht in der Ready-Liste ist, läuft er wohl gerade auf einer anderen CPU
- diese andere CPU muss benachrichtigt werden  
→ INTER PROCESSOR INTERRUPT (IPI)
- die angesprochene CPU muss dann das Kill-Flag des aktuellen Prozesses prüfen

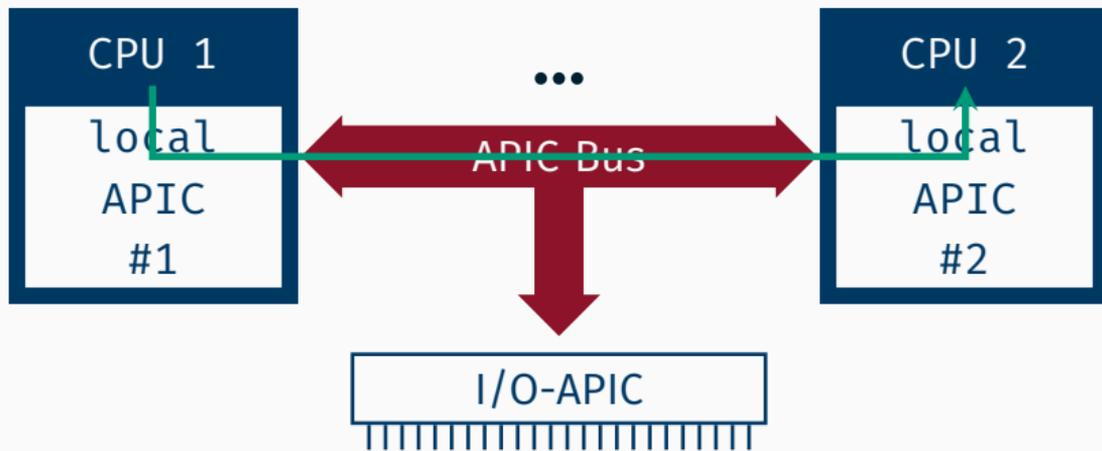
# Inter Processor Interrupt



# Inter Processor Interrupt

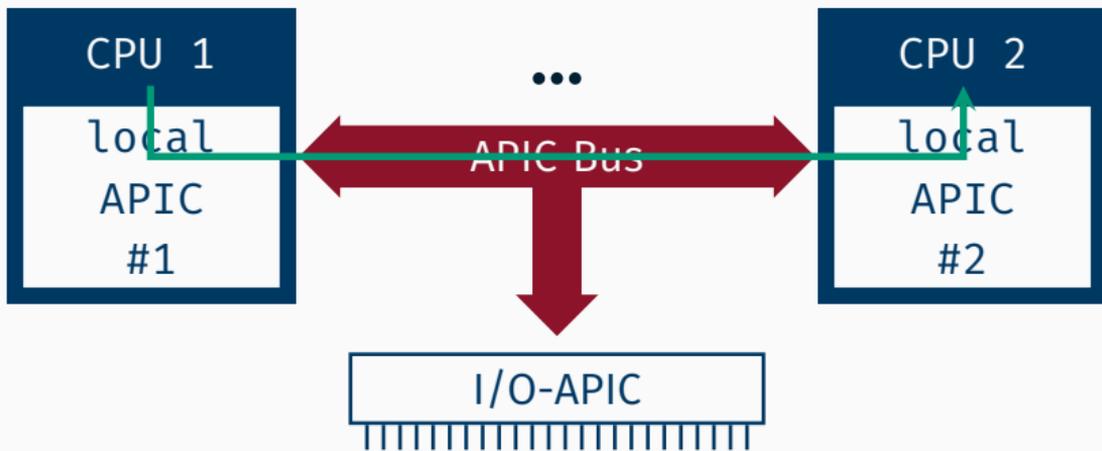


# Inter Processor Interrupt



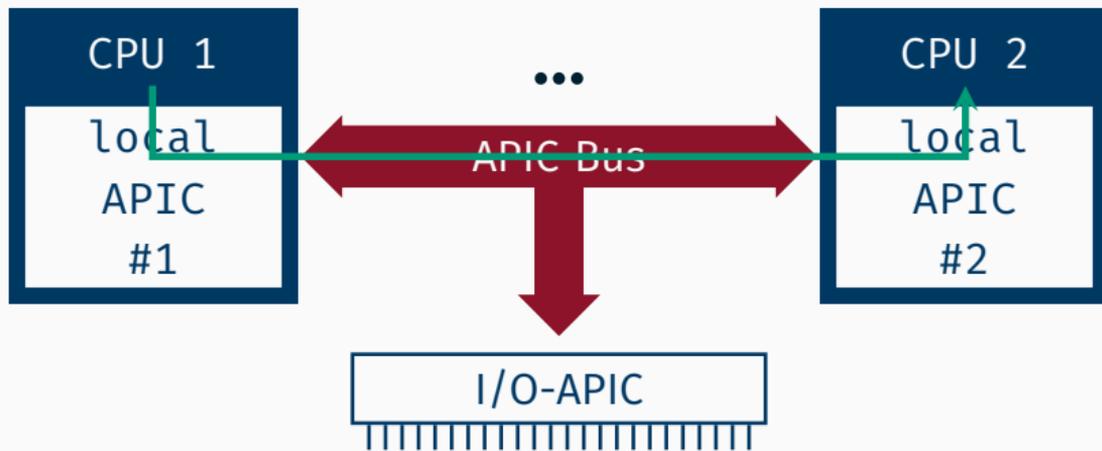
```
LAPIC::IPI::send(destination, vector);
```

# Inter Processor Interrupt



```
LAPIC::IPI::send(destination, vector);  
                        Interrupt
```

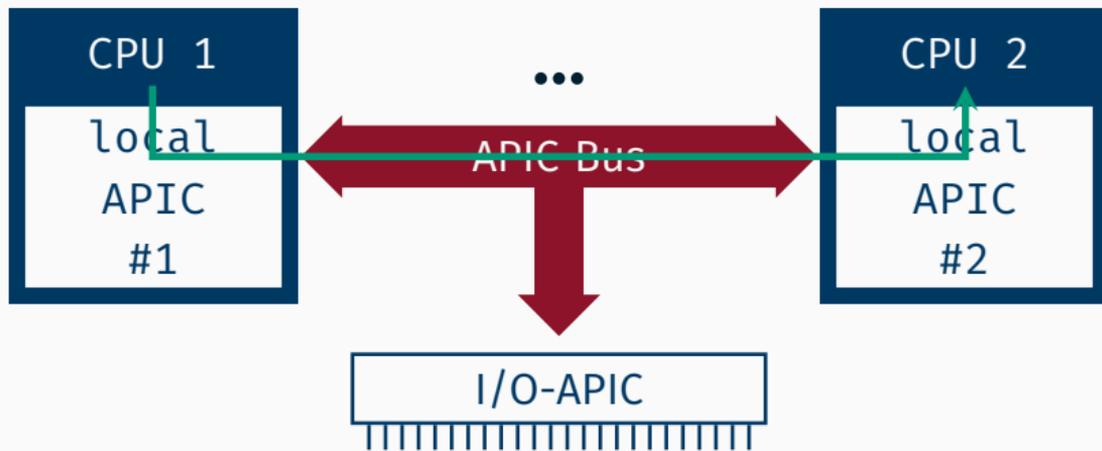
# Inter Processor Interrupt



```
LAPIC::IPI::send(destination, vector);
```

Empfänger      Interrupt

# Inter Processor Interrupt



```
destination = APIC::getLAPICID(cpu);
```

```
LAPIC::IPI::send(destination, vector);
```

Empfänger

Interrupt

# Der Teufel steckt im Detail

Was kann hier schon schief gehen?

```
lock[Core::getID()] = true;
```

# Der Teufel steckt im Detail

Was kann hier schon schief gehen?

```
lock[Core::getID()] = true;
```

**Viel - diese Zeile wird nicht atomar ausgeführt:**

```
; Array lock an Adresse 0x2000  
call <Core::getID(>  
mov [rax+0x2000], 0x1
```

# Der Teufel steckt im Detail

Was kann hier schon schief gehen?

```
lock[Core::getID()] = true;
```

**Viel - diese Zeile wird nicht atomar ausgeführt:**

```
; Array lock an Adresse 0x2000
```

```
call <Core::getID(>
```

```
mov [rax+0x2000], 0x1
```

⚡ Scheduler Interrupt

# Der Teufel steckt im Detail

Was kann hier schon schief gehen?

```
lock[Core::getID()] = true;
```

**Viel - diese Zeile wird nicht atomar ausgeführt:**

```
; Array lock an Adresse 0x2000
```

```
call <Core::getID(>
```

```
mov [rax+0x2000], 0x1
```

⚡ Scheduler Interrupt

Was passiert nun, wenn der Anwendungsfaden anschließend auf einer anderen CPU eingeplant wird?

# Fragen über Fragen

- Kann nun eine fehlerhafte Anwendung unser Betriebssystem blockieren?

# Fragen über Fragen

- Kann nun eine fehlerhafte Anwendung unser Betriebssystem blockieren?
- Ist unser implementiertes Schedulingverfahren *fair*?

# Fragen über Fragen

- Kann nun eine fehlerhafte Anwendung unser Betriebssystem blockieren?
- Ist unser implementiertes Schedulingverfahren *fair*?
- Unter welchen Umständen wäre es effizienter, den Timer abzuschalten (Stichwort *tickless*)?

# Fragen über Fragen

- Kann nun eine fehlerhafte Anwendung unser Betriebssystem blockieren?
- Ist unser implementiertes Schedulingverfahren *fair*?
- Unter welchen Umständen wäre es effizienter, den Timer abzuschalten (Stichwort *tickless*)?
- Sollen wir beim IPI in `Scheduler::kill` auf Antwort warten?

# Fragen?

---

Bitte evaluieren!