

Übung zu Betriebssystembau

Zeitscheibenscheduling

10. Dezember 2024

Alexander Krause

Arbeitsgruppe Systemsoftware
Technische Universität Dortmund

(Mit Material vom Lehrstuhl 4 der FAU)

```
1 int i = 0;
2 while (true){
3     {
4         Guarded _;
5         kout << i++ << endl;
6     }
7     scheduler.resume();
8 }
```



```
1 int i = 0;
2 while (true){
3     {
4         Guarded _;
5         kout << i++ << endl;
6     }
7
8 }
```



Unterbrechende Ablaufplanung

```
1 int i = 0;
2 while (true){
3     {
4         Guarded _;
5         kout << i++ << endl;
6     }
7
8 }
```

⚡ Scheduler Interrupt



```
1 int i = 0;
2 while (true){
3     {
4         Guarded _;
5         kout << i++ << endl;
6     }
7 _____ ⚡ Scheduler Interrupt
8 }
```

Aufgabe: Präemptives Scheduling mittels Timer.



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 ($\approx 1 \text{ MHz} / 833,333$)

→ Genauigkeit: 833,333

PIC 8259A

Interruptleitung

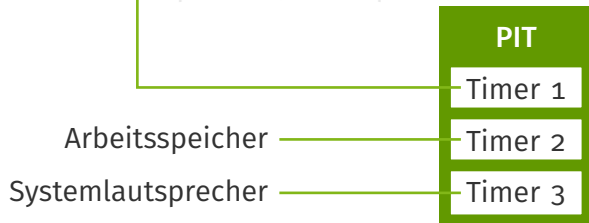
CPU

- drei Kanäle mit je einem 16 bit Zähler

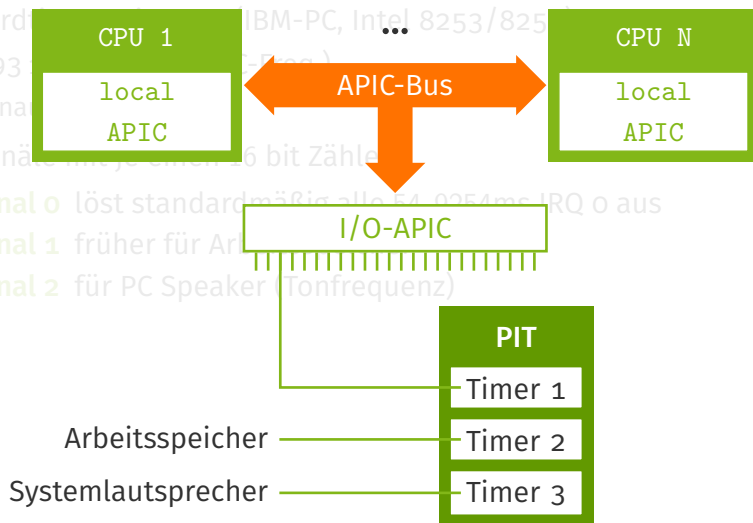
Kanal 0 löst standardmäßig alle 54,9254ms IRQ 0 aus

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 frühere BSB-Varianten, **geht aber besser.***
- `machine/pit.h`



- 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
- für Zeit & Datum
- Betrieb im ausgeschalteten Zustand mittels Batterie



- 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
 - kann keinen Interrupt auslösen
- `machine/tsc.h`



- 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



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



- ≥ 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

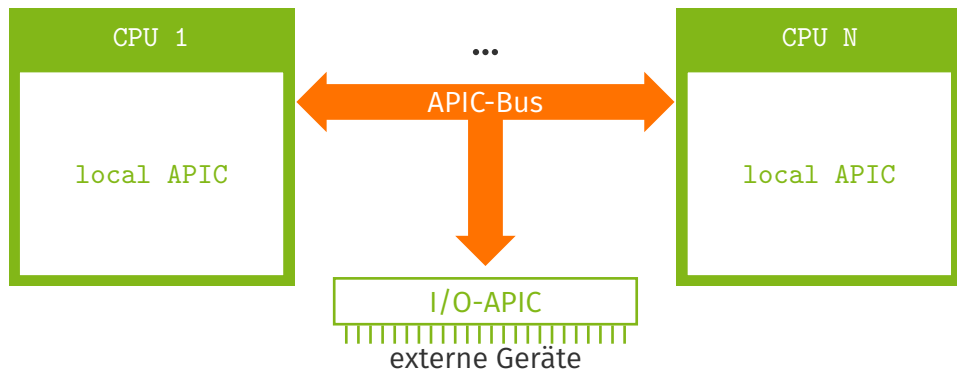


- ≥ 100 MHz
 - Genauigkeit: 10 ns oder besser
- 32 bit Zähler
- verwendet Busfrequenz
 - abhängig vom System
 - aber unabhängig von Stromsparmmodus
 - Interrupt geht nur an den entsprechenden Kern
 - 8 Möglichkeiten (bis $\frac{1}{128}$ Busfrequenz) durch Vorteiler

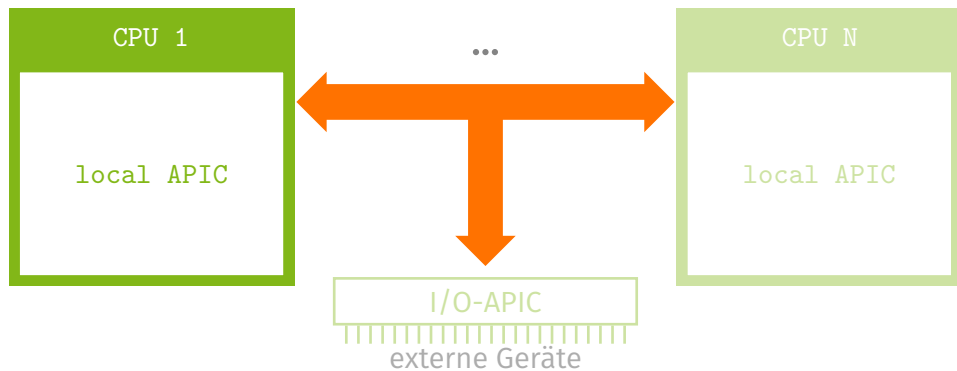
Perfekt für unsere Bedürfnisse



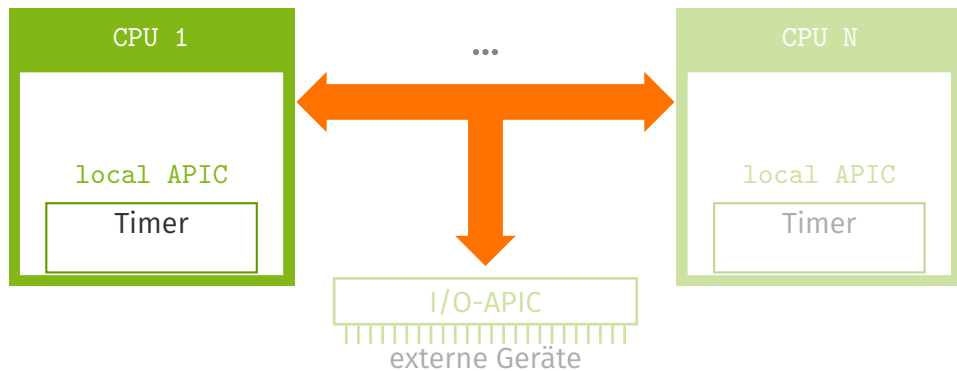
Funktionsweise des LAPIC Timers



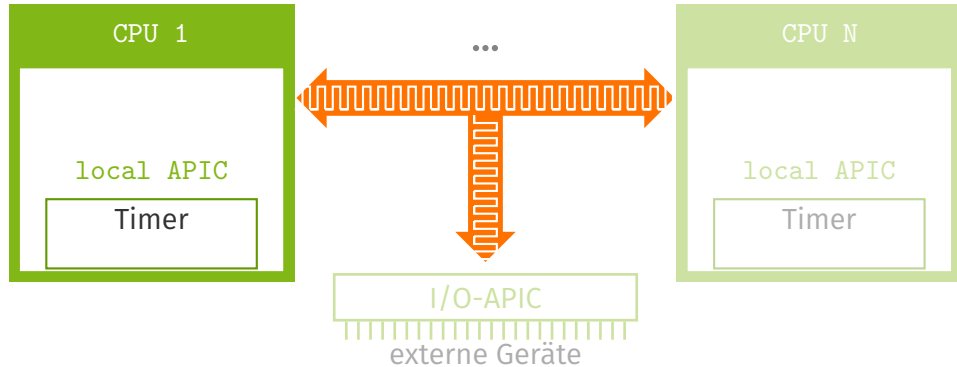
Funktionsweise des LAPIC Timers



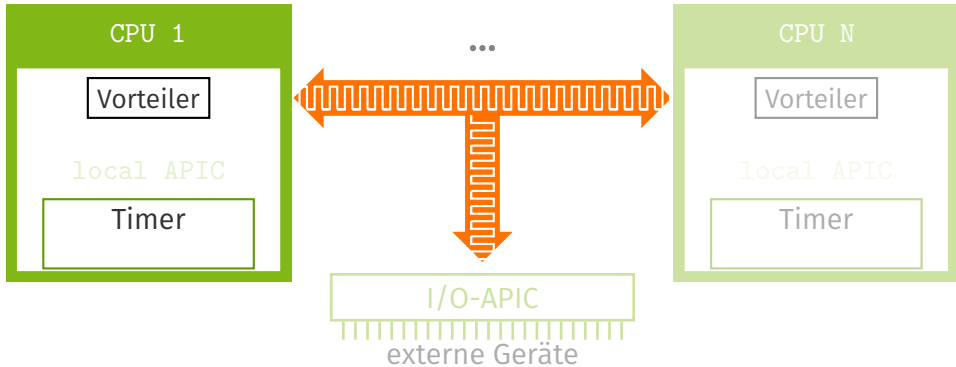
Funktionsweise des LAPIC Timers



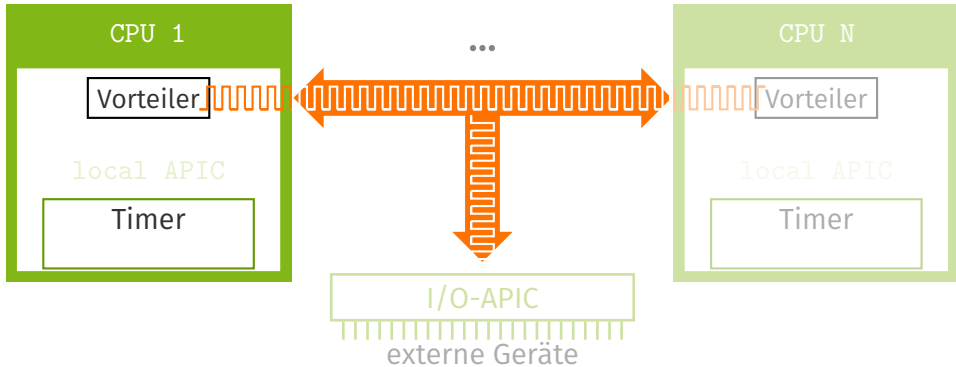
Funktionsweise des LAPIC Timers



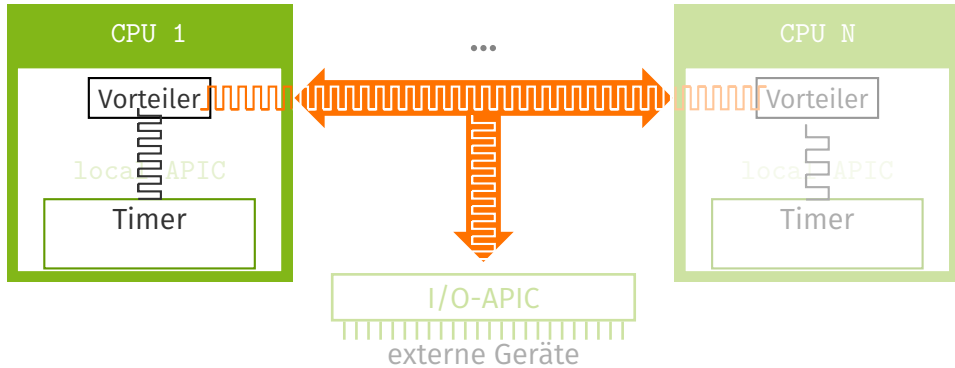
Funktionsweise des LAPIC Timers



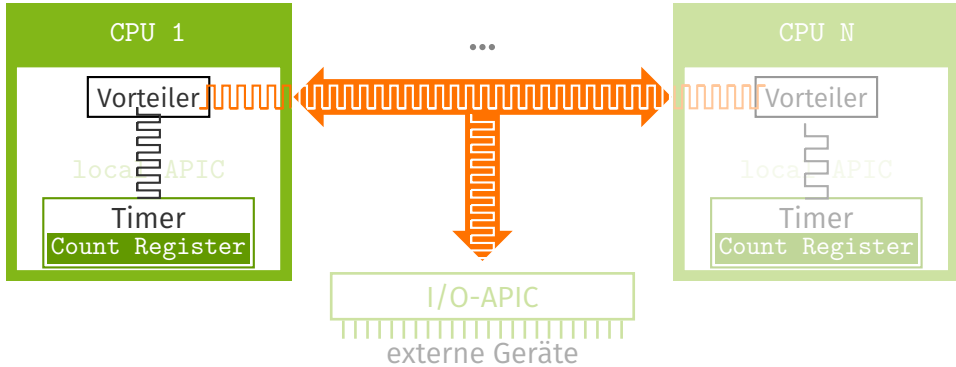
Funktionsweise des LAPIC Timers



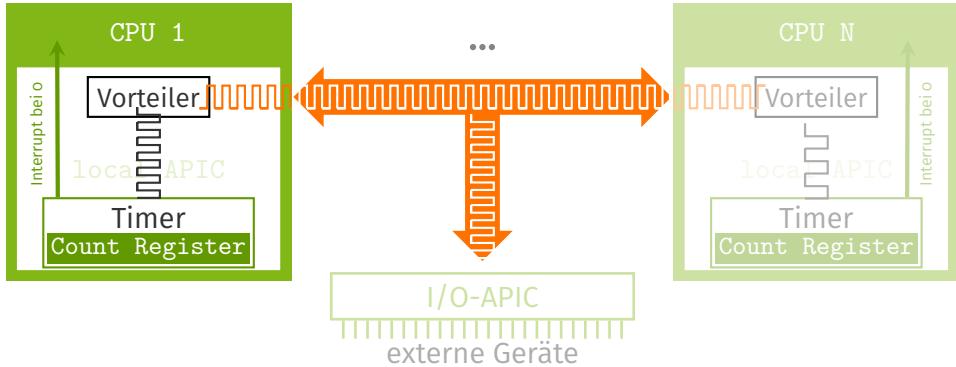
Funktionsweise des LAPIC Timers



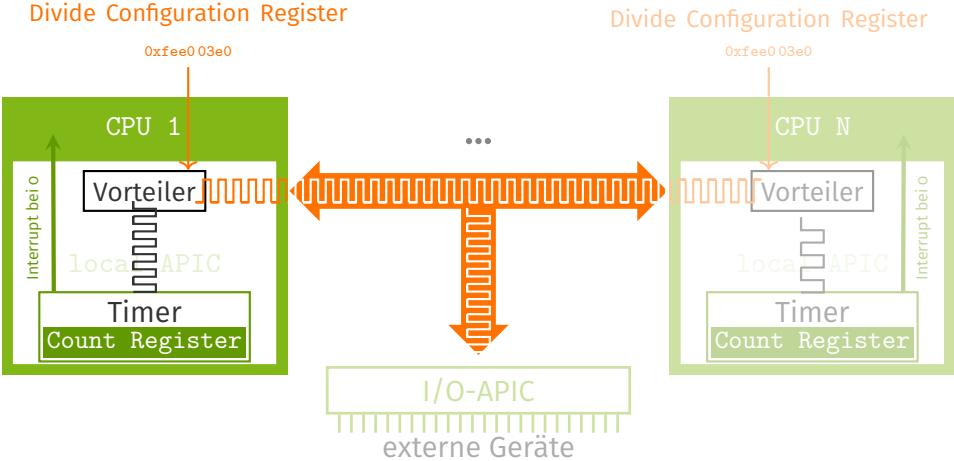
Funktionsweise des LAPIC Timers



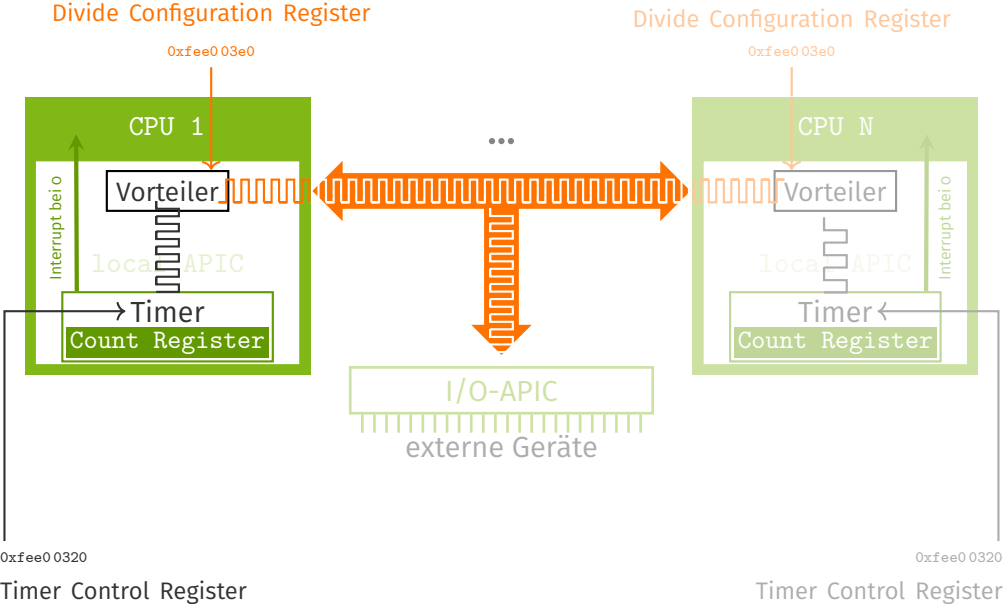
Funktionsweise des LAPIC Timers



Funktionsweise des LAPIC Timers



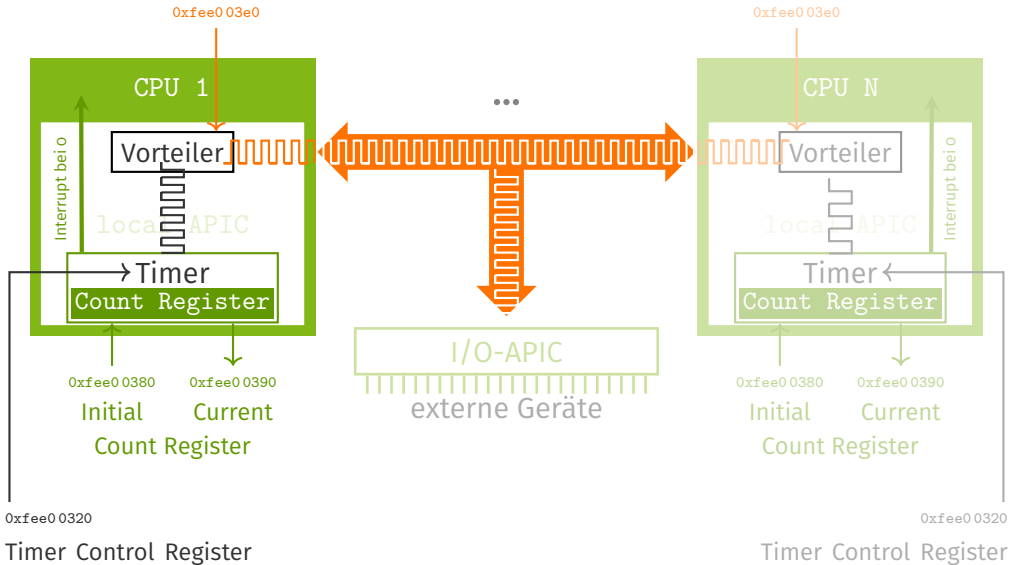
Funktionsweise des LAPIC Timers



Funktionsweise des LAPIC Timers

Divide Configuration Register

Divide Configuration Register



0xfee00320

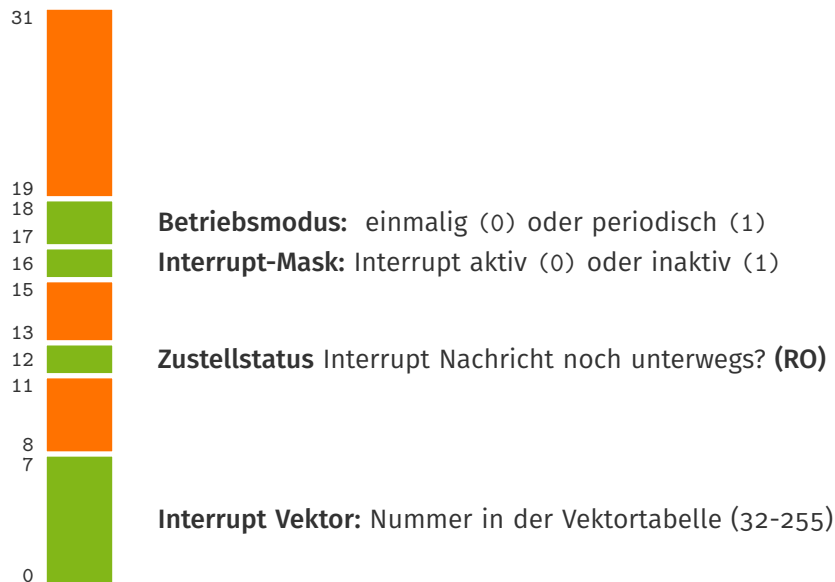
0xfee00320

Timer Control Register

Timer Control Register



Aufbau des Timer Control Register Eintrags



- 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



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()`



1. LAPIC-Timer kalibrieren



1. LAPIC-Timer kalibrieren

- in der Funktion `LAPIC::Timer::ticks`
(Funktion gibt die Anzahl der Ticks in einer **Millisekunde** zurück)



1. LAPIC-Timer kalibrieren

- in der Funktion `LAPIC::Timer::ticks`
(Funktion gibt die Anzahl der Ticks in einer **Millisekunde** zurück)
- benötigt zur Konfiguration die Funktion `LAPIC::Timer::set`, Aufruf mit
 - maximalem Wert für den LAPIC-Zähler,
 - als `ONE_SHOT` und
 - ohne Unterbrechungen



1. LAPIC-Timer kalibrieren

- in der Funktion `LAPIC::Timer::ticks`
(Funktion gibt die Anzahl der Ticks in einer **Millisekunde** zurück)
- benötigt zur Konfiguration die Funktion `LAPIC::Timer::set`, Aufruf mit
 - maximalem Wert für den LAPIC-Zähler,
 - als `ONE_SHOT` und
 - ohne Unterbrechungen
- unter Verwendung des PIT
 - Wartezeit von mehreren MS einstellen: `PIT::set`
 - mittels `PIT::waitForTimeout` warten
 - Start- und Endwert des LAPIC-Zählerregisters merken



1. LAPIC-Timer kalibrieren

- in der Funktion `LAPIC::Timer::ticks`
(Funktion gibt die Anzahl der Ticks in einer **Millisekunde** zurück)
- benötigt zur Konfiguration die Funktion `LAPIC::Timer::set`, Aufruf mit
 - maximalem Wert für den LAPIC-Zähler,
 - als `ONE_SHOT` und
 - ohne Unterbrechungen
- unter Verwendung des PIT
 - Wartezeit von mehreren MS einstellen: `PIT::set`
 - mittels `PIT::waitForTimeout` warten
 - Start- und Endwert des LAPIC-Zählerregisters merken
- 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 $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Vorteiler} \cdot 1000}$ berechnen, dabei gilt $\text{Vorteiler} = 2^x$ mit $x \in \{0, \dots, 7\}$
- Möglichst kleiner Vorteiler, aber kein Überlauf (32 bit!)



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);`
 $n \quad 5\,000\,000 \mu\text{s} = 5 \text{ s}$
`LAPIC::Timer::ticks` $1\,000\,000 \text{ ms}^{-1}$
 $\text{Vorteiler} \quad 1 = 2^0$



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 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 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);`
 $n \quad 5\,000\,000 \mu\text{s} = 5 \text{ s}$
`LAPIC::Timer::ticks` $1\,000\,000 \text{ ms}^{-1}$
 $\text{Vorteiler} \quad 2 = 2^1$
 $initial$



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



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 ms^{-1}
 - Vorteiler 2 = 2^1
 - $initial$ 2 500 000 000 ✓



Ablaufbeispiel (Standardfall)

app1

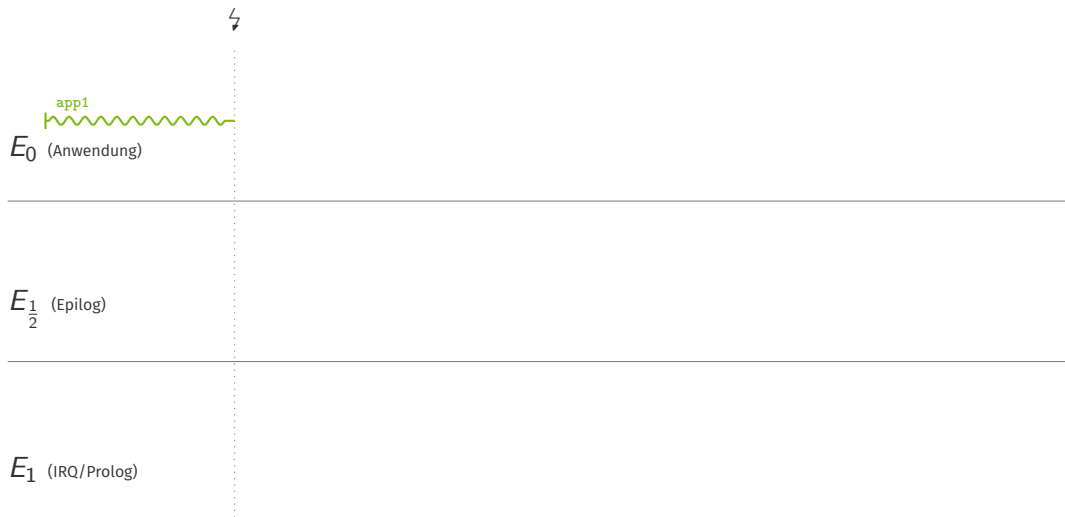
 E_0 (Anwendung)

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

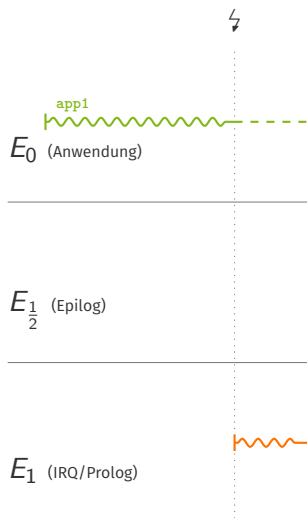
E_1 (IRQ/Prolog)



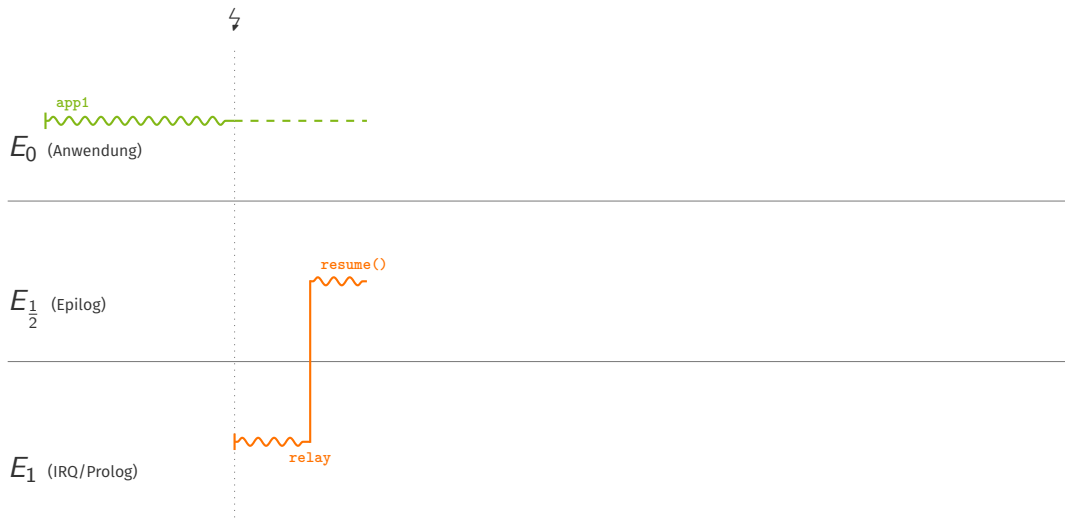
Ablaufbeispiel (Standardfall)



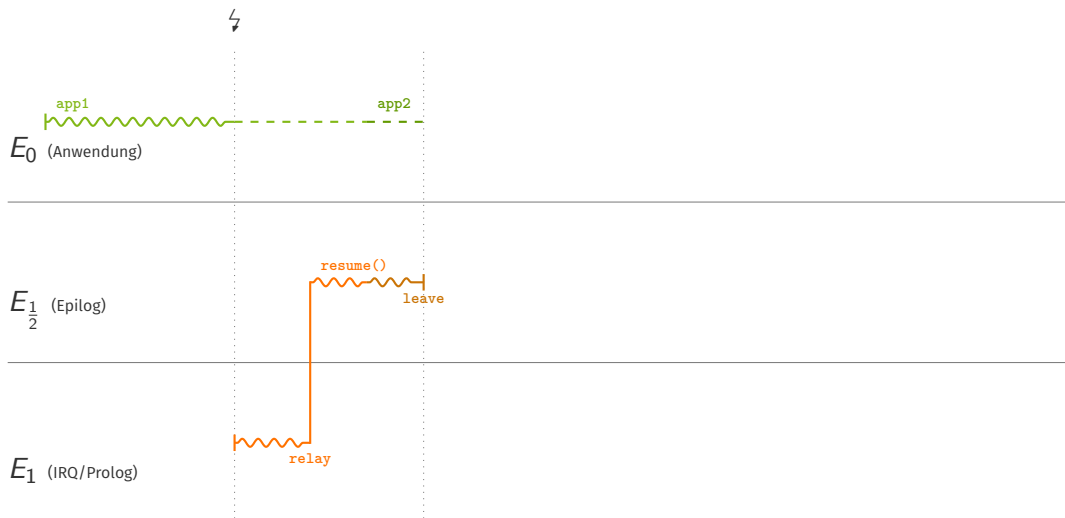
Ablaufbeispiel (Standardfall)



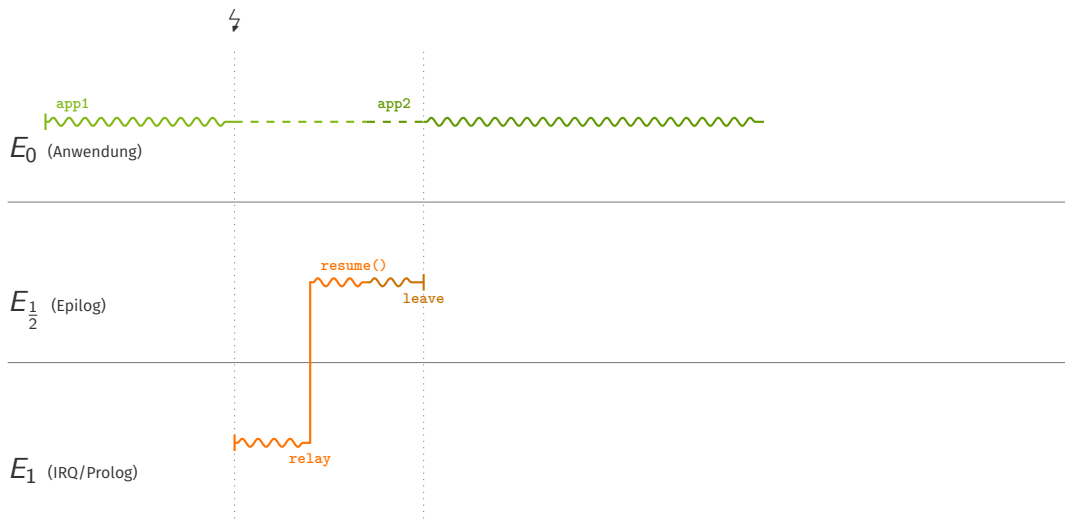
Ablaufbeispiel (Standardfall)



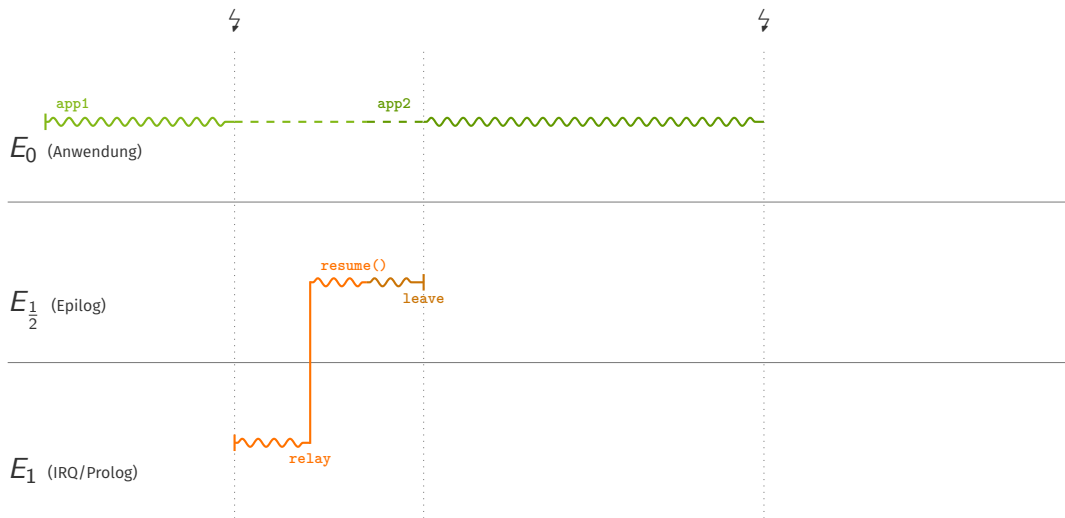
Ablaufbeispiel (Standardfall)



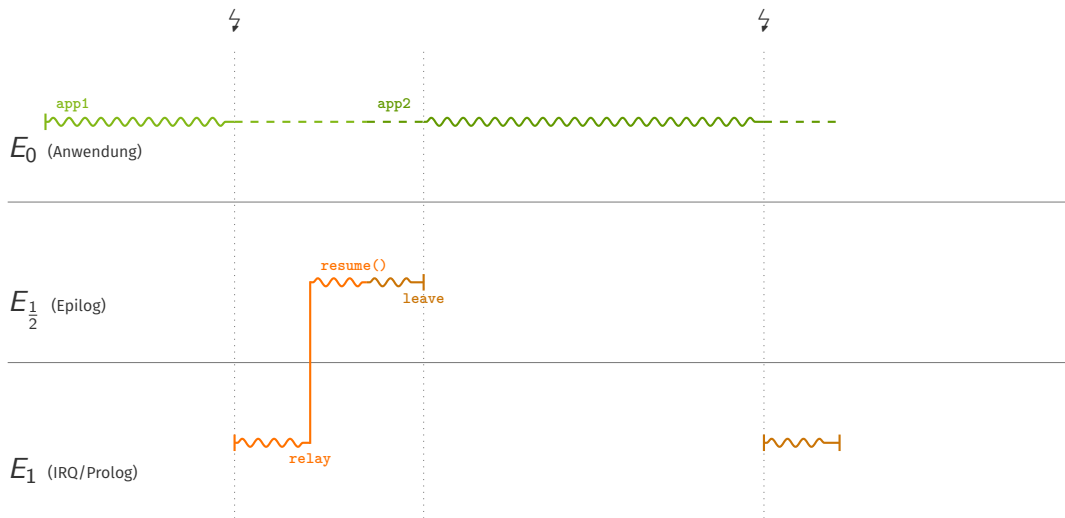
Ablaufbeispiel (Standardfall)



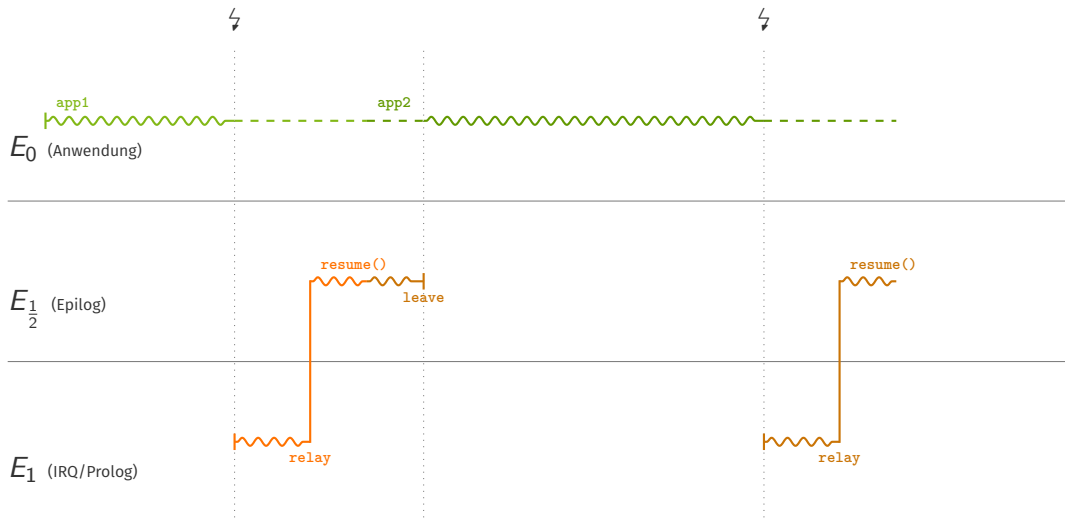
Ablaufbeispiel (Standardfall)



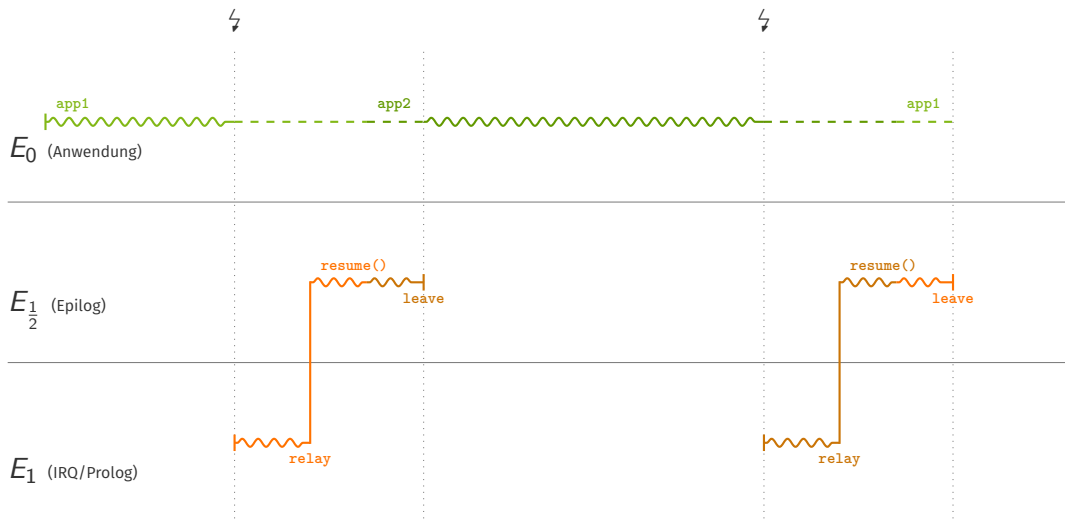
Ablaufbeispiel (Standardfall)



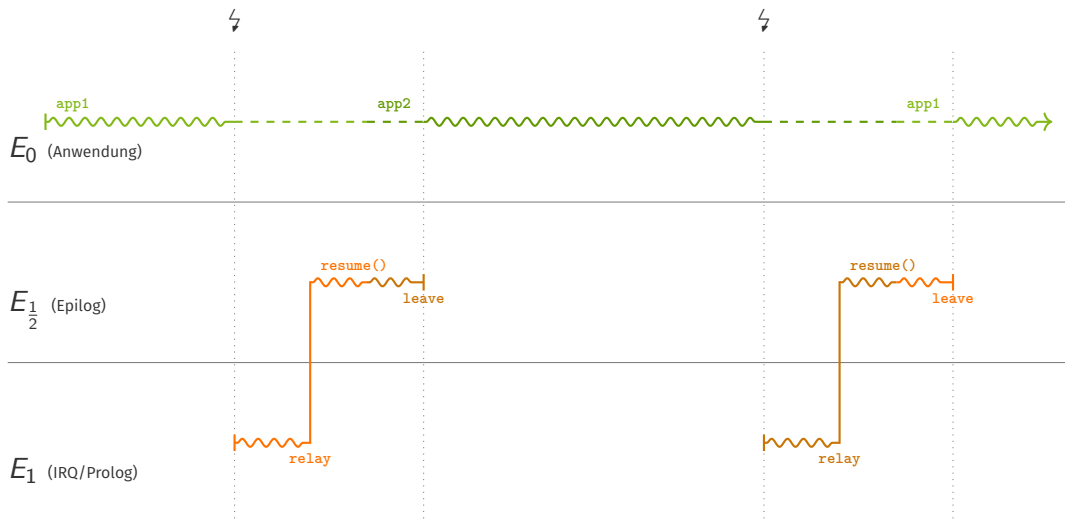
Ablaufbeispiel (Standardfall)



Ablaufbeispiel (Standardfall)



Ablaufbeispiel (Standardfall)



Ablaufbeispiel bei Faden in Systemebene

app1

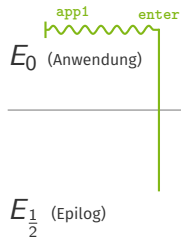
 E_0 (Anwendung)

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

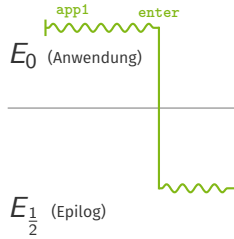
E_1 (IRQ/Prolog)



Ablaufbeispiel bei Faden in Systemebene



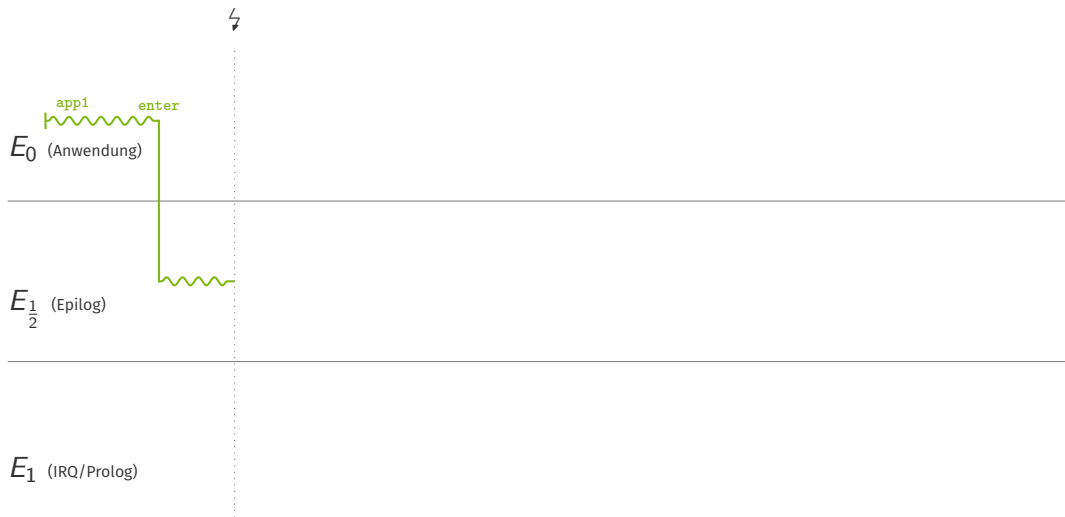
Ablaufbeispiel bei Faden in Systemebene



E_1 (IRQ/Prolog)



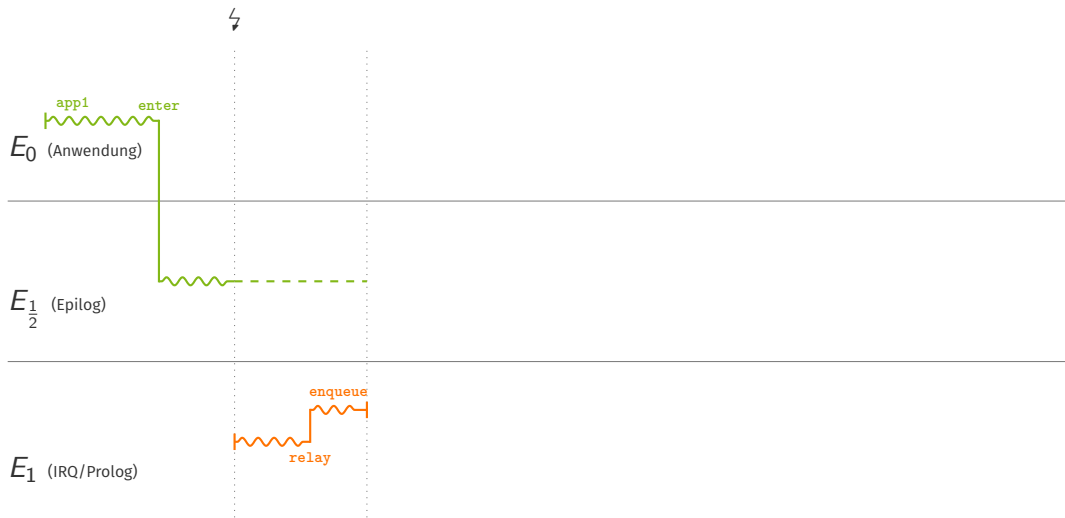
Ablaufbeispiel bei Faden in Systemebene



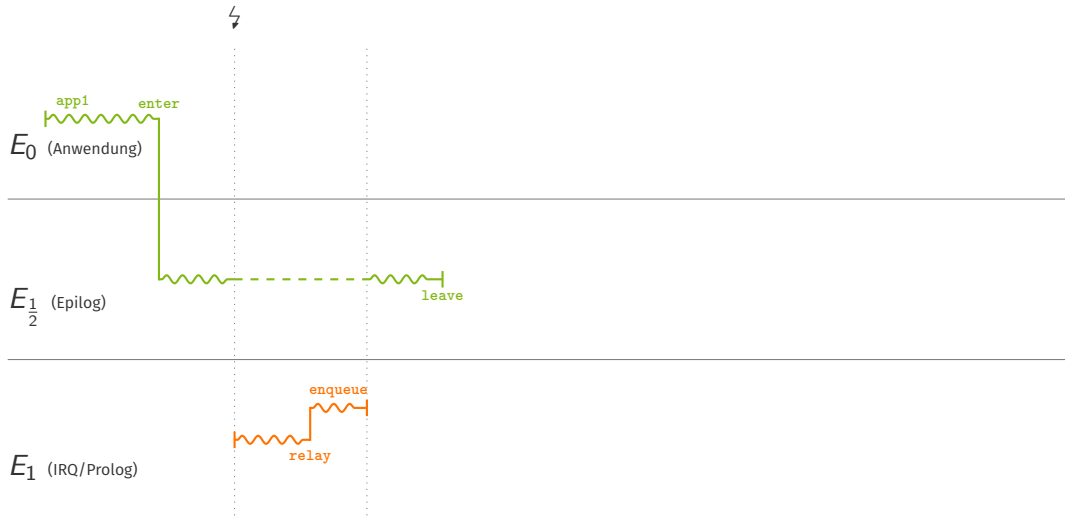
Ablaufbeispiel bei Faden in Systemebene



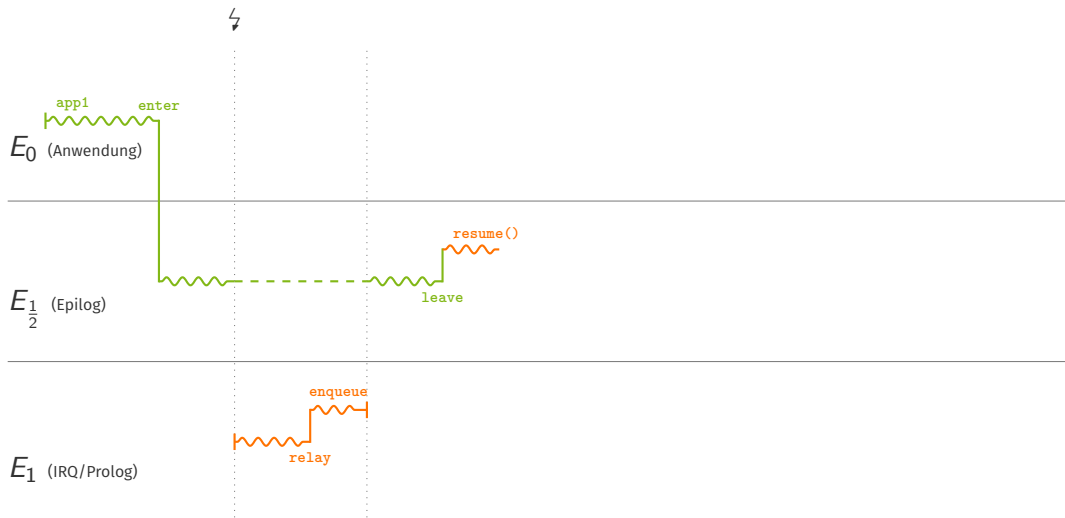
Ablaufbeispiel bei Faden in Systemebene



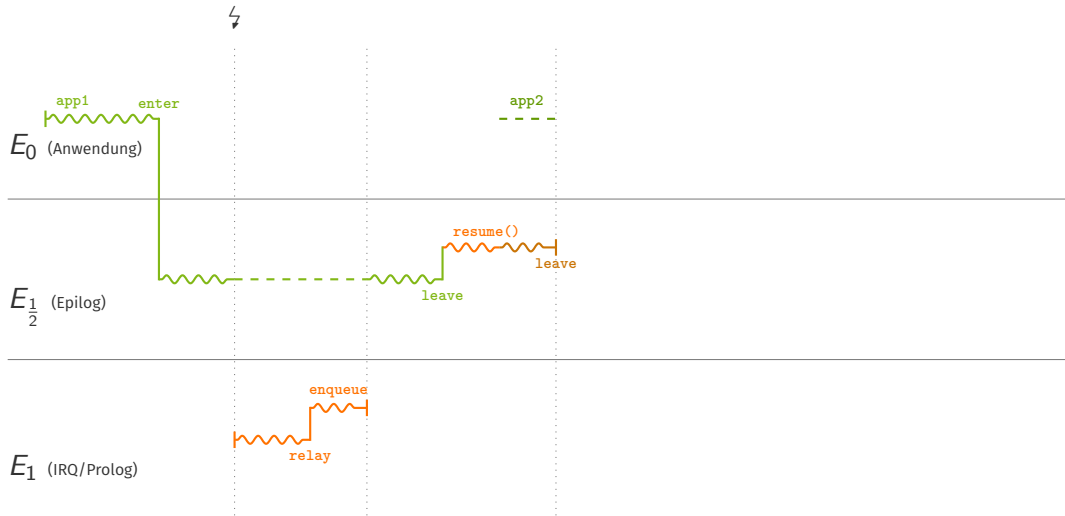
Ablaufbeispiel bei Faden in Systemebene



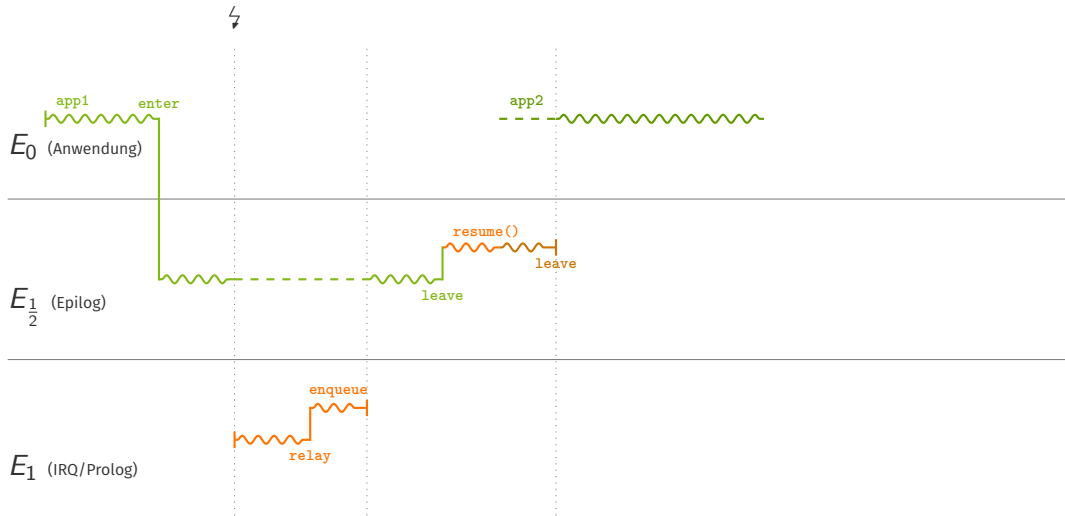
Ablaufbeispiel bei Faden in Systemebene



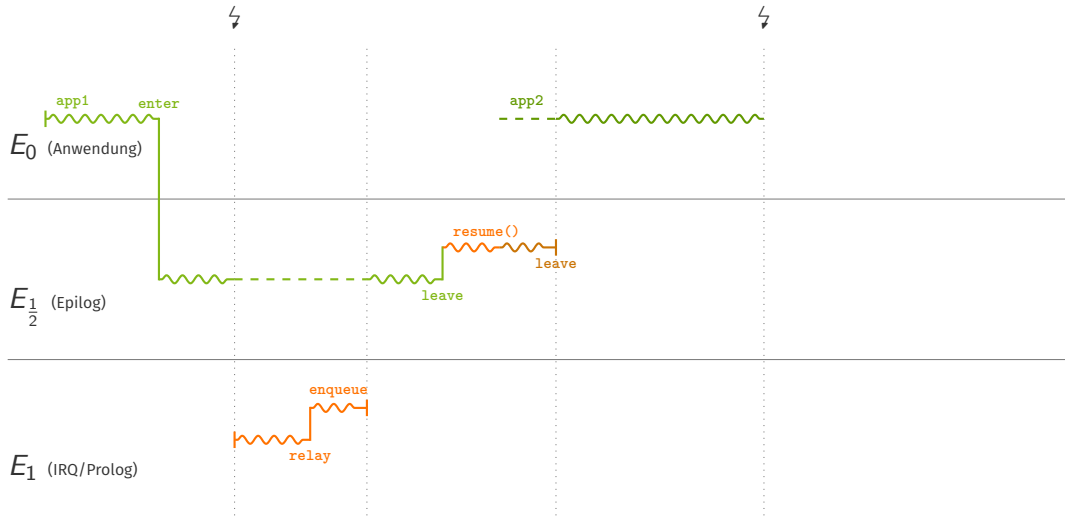
Ablaufbeispiel bei Faden in Systemebene



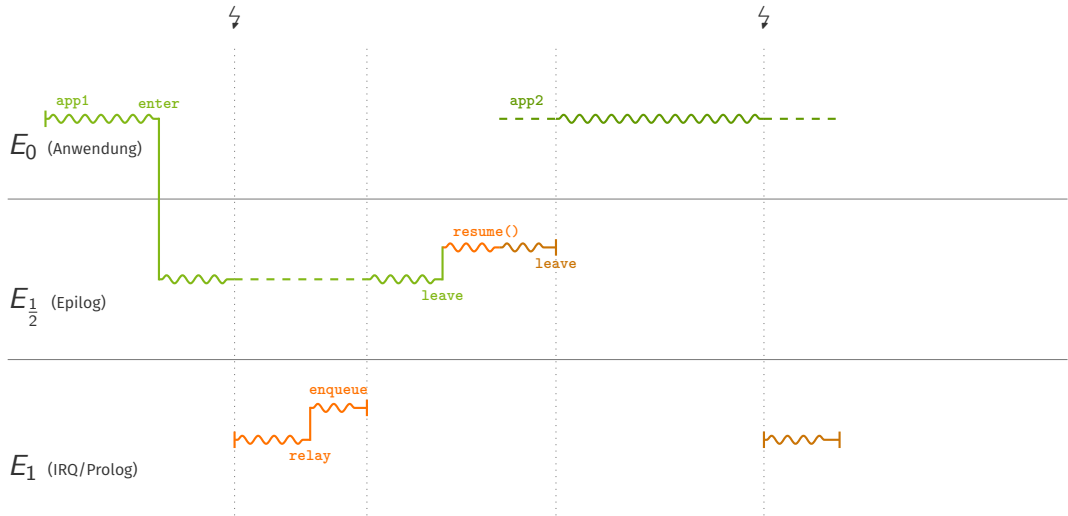
Ablaufbeispiel bei Faden in Systemebene



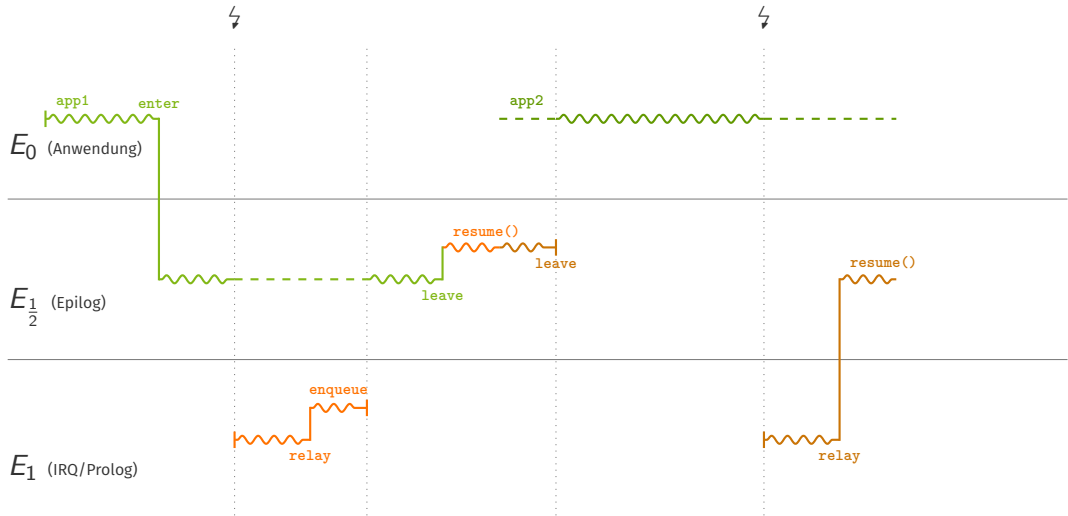
Ablaufbeispiel bei Faden in Systemebene



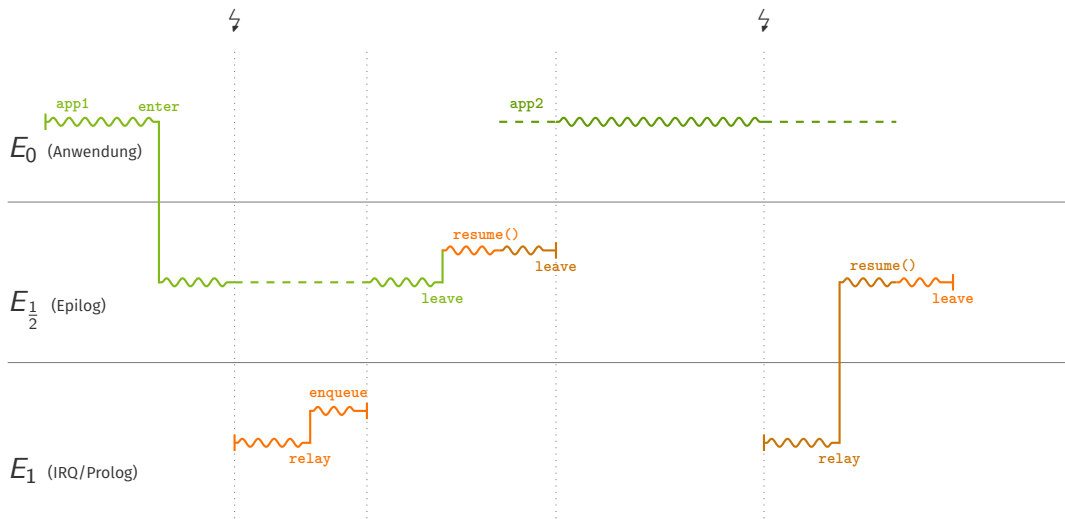
Ablaufbeispiel bei Faden in Systemebene



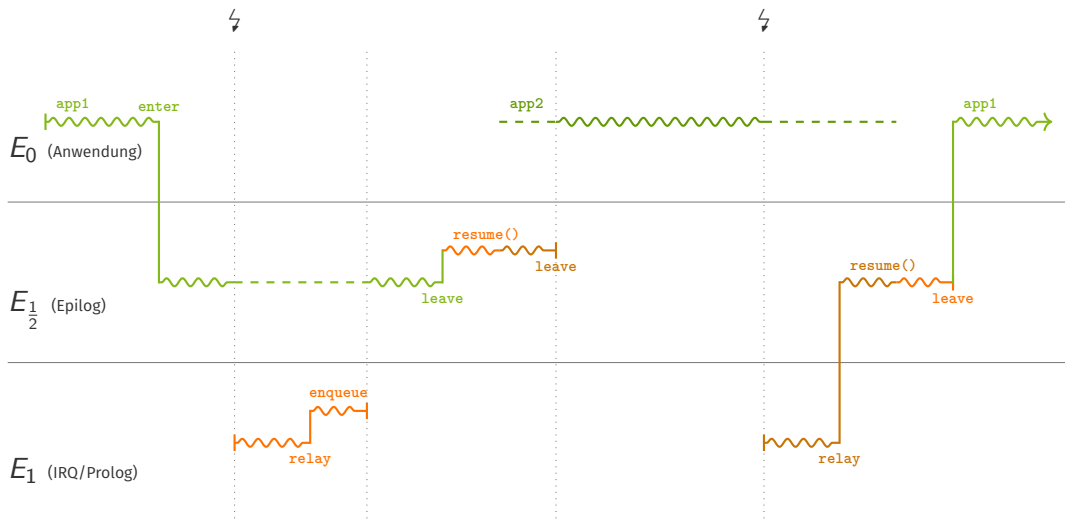
Ablaufbeispiel bei Faden in Systemebene



Ablaufbeispiel bei Faden in Systemebene



Ablaufbeispiel bei Faden in Systemebene



Ablaufbeispiel mit neuem Thread

app1

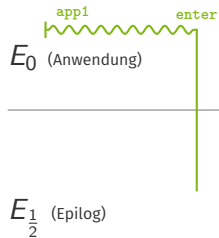
 E_0 (Anwendung)

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

E_1 (IRQ/Prolog)



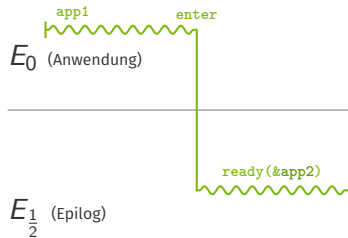
Ablaufbeispiel mit neuem Thread



E_1 (IRQ/Prolog)



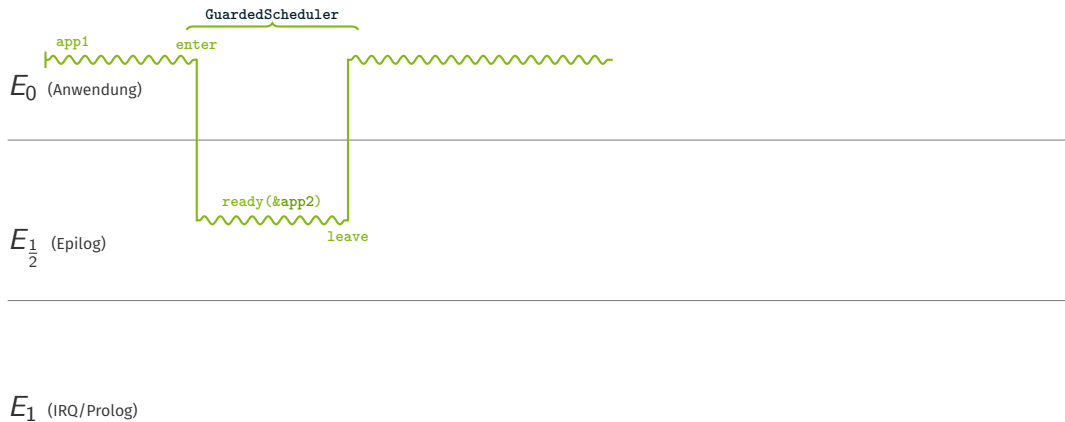
Ablaufbeispiel mit neuem Thread



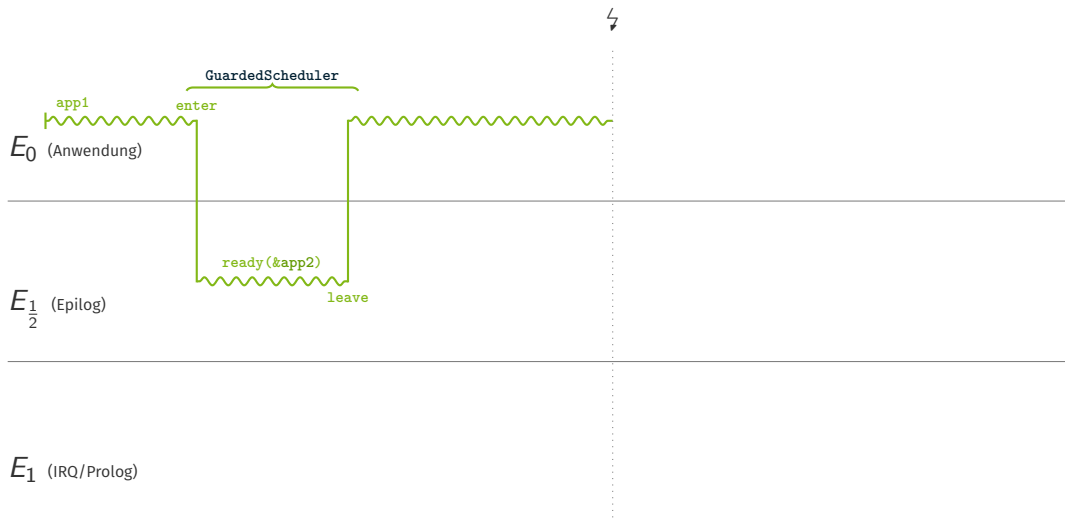
Ablaufbeispiel mit neuem Thread



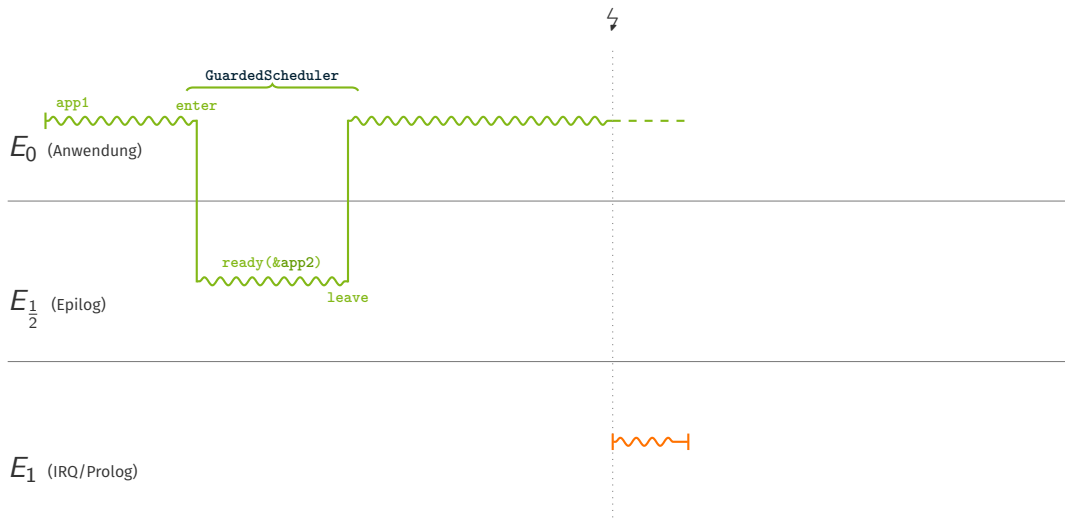
Ablaufbeispiel mit neuem Thread



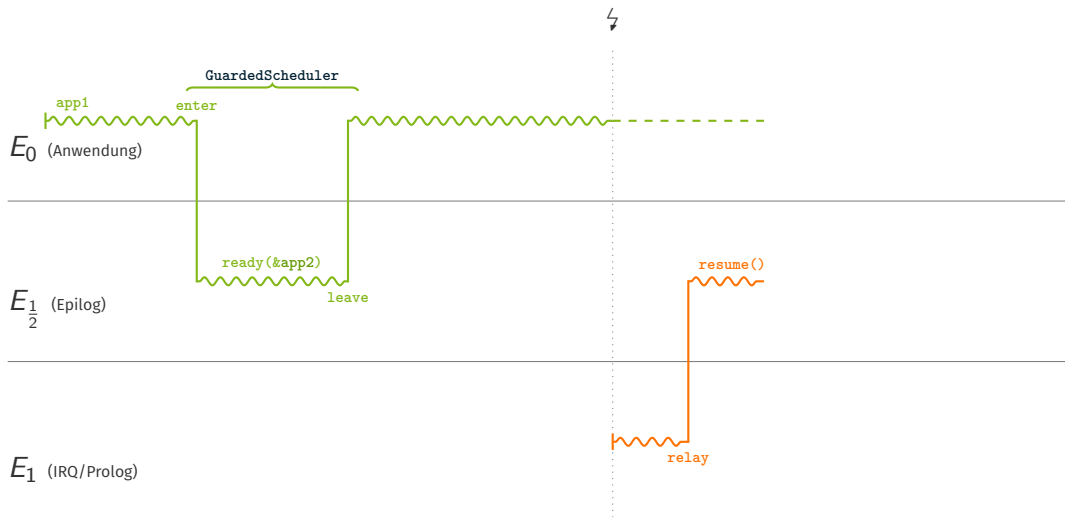
Ablaufbeispiel mit neuem Thread



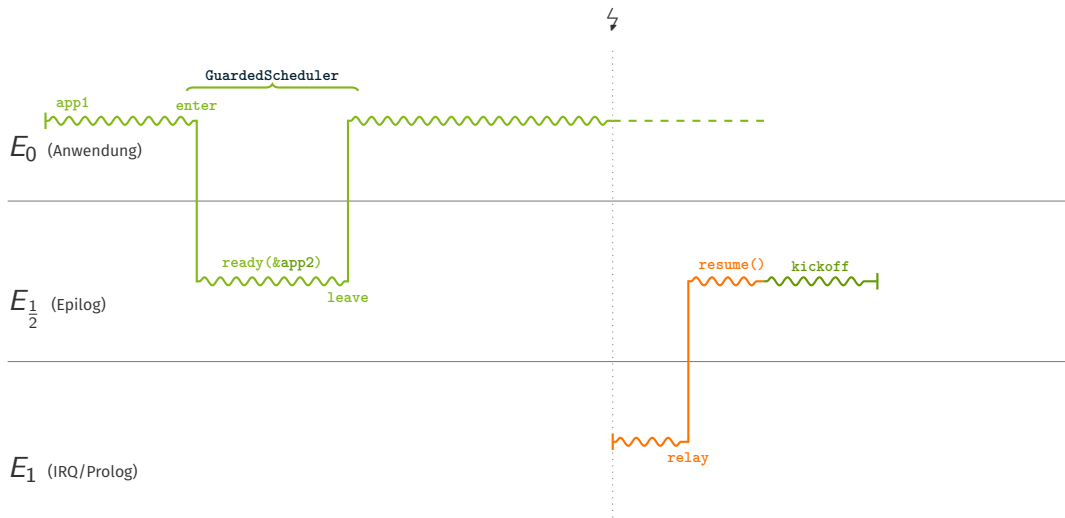
Ablaufbeispiel mit neuem Thread



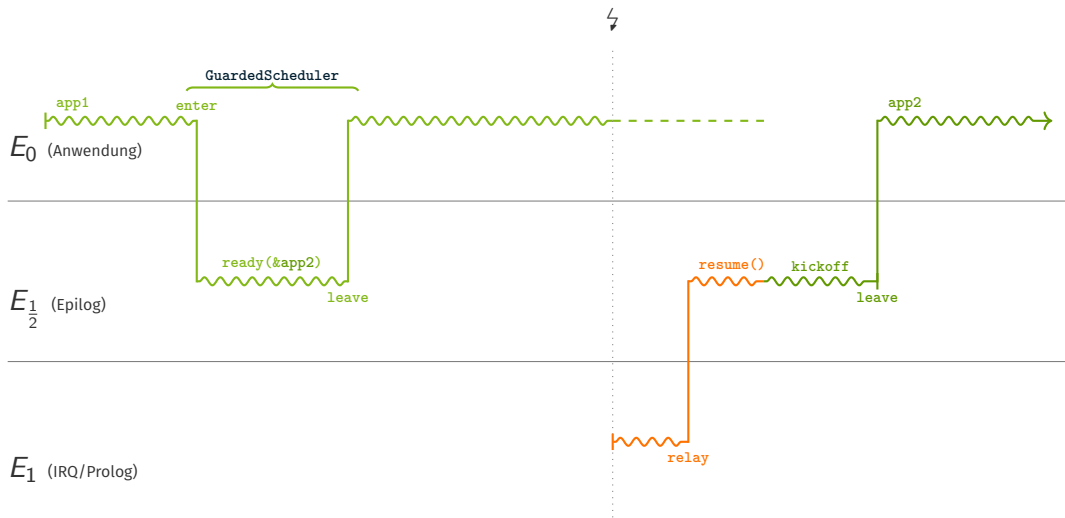
Ablaufbeispiel mit neuem Thread



Ablaufbeispiel mit neuem Thread



Ablaufbeispiel mit neuem Thread



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



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

OOSTuBS *keine Änderung*



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

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



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



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)



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



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



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)



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

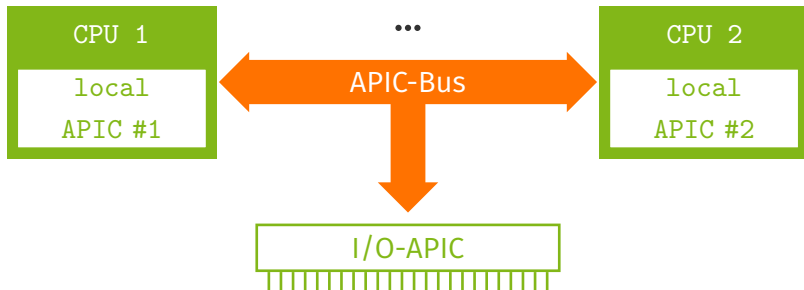
OOSTuBS *keine Änderung:* aus der Ready-Liste entfernen
bzw. Kill-Flag setzen und bei `resume` prüfen

MPStuBS der Anwendungsfaden kann gerade auf einer anderen CPU laufen

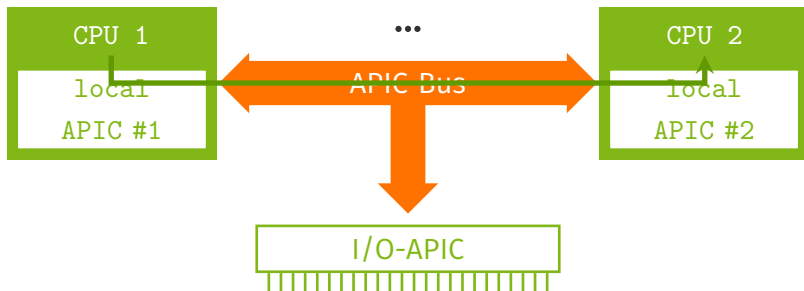
- Kill-Flag setzen (wie gehabt)
- falls er nicht in der Ready-Liste ist, läuft er wohl gerade auf einer anderen CPU
- diese andere CPU muss benachrichtigt werden
→ INTER PROCESSOR INTERRUPT (IPI)
- die angesprochene CPU muss dann das Kill-Flag des aktuellen Prozesses prüfen



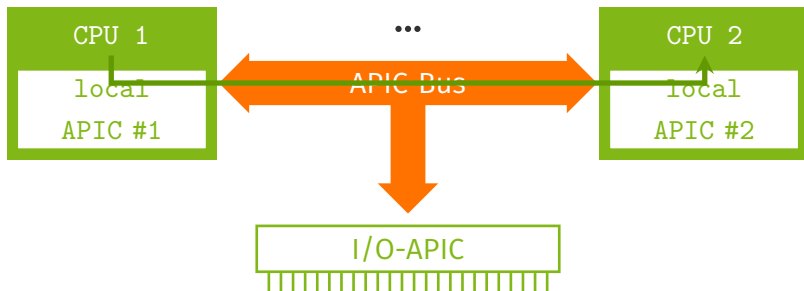
Inter Processor Interrupt



Inter Processor Interrupt



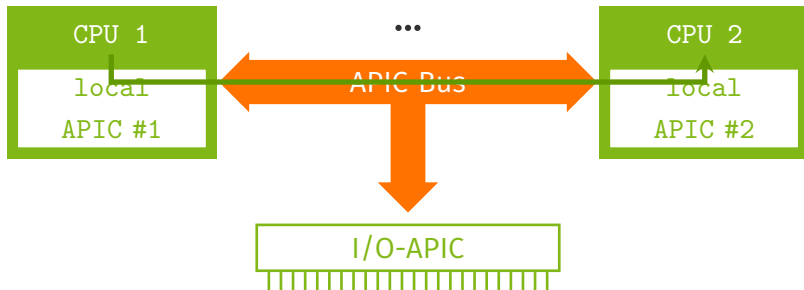
Inter Processor Interrupt



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



Inter Processor Interrupt

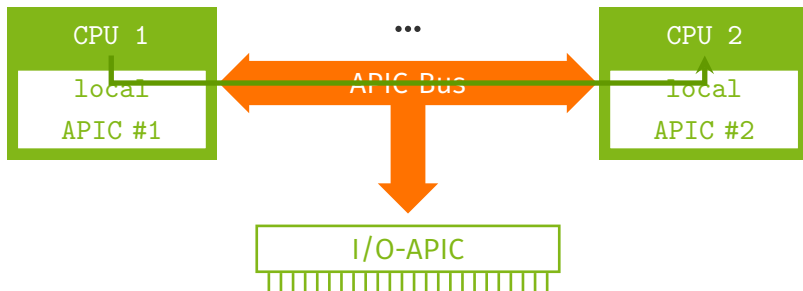


```
LAPIC::IPI::send(destination, vector);
```

└──────────┘
Interrupt



Inter Processor Interrupt

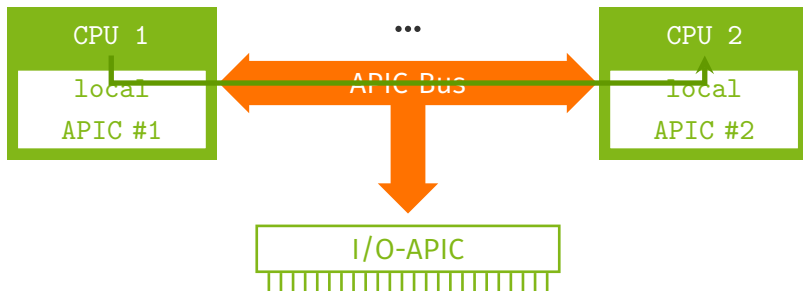


```
LAPIC::IPI::send(destination, vector);
```

Empfänger Interrupt



Inter Processor Interrupt



```
destination = APIC::getLAPICID(cpu);  
LAPIC::IPI::send(destination, vector);
```

Empfänger Interrupt



Was kann hier schon schief gehen?

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



Was kann hier schon schief gehen?

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

Viel - diese Zeile wird nicht atomar ausgeführt:

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



Was kann hier schon schief gehen?

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

Viel - diese Zeile wird nicht atomar ausgeführt:

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

```
2 call <Core::getID()> ⚡ Scheduler Interrupt
```

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



Was kann hier schon schief gehen?

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

Viel - diese Zeile wird nicht atomar ausgeführt:

```
1 ; Array lock an Adresse 0x2000  
2 call <Core::getID()> ⚡ Scheduler Interrupt  
3 mov [rax+0x2000], 0x1
```

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



- Kann nun eine fehlerhafte Anwendung unser Betriebssystem blockieren?



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



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



Gibt es noch Fragen?

**Abgabe der Aufgabe
bis Mittwoch, den 08. Januar**