Übung zu Betriebssysteme

Zeitscheibenscheduling

19. & 20. Dezember 2019

Andreas Ziegler, Bernhard Heinloth, Christian Eichler

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

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Motivation

Kooperative Ablaufplanung

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

Unterbrechende Ablaufplanung

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

}
```

Unterbrechende Ablaufplanung

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

ϟ Scheduler Interrupt

```
}
```

Unterbrechende Ablaufplanung

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

ϟ Scheduler Interrupt

```
}
```

Aufgabe: **Präemptives Scheduling mittels Timer.**

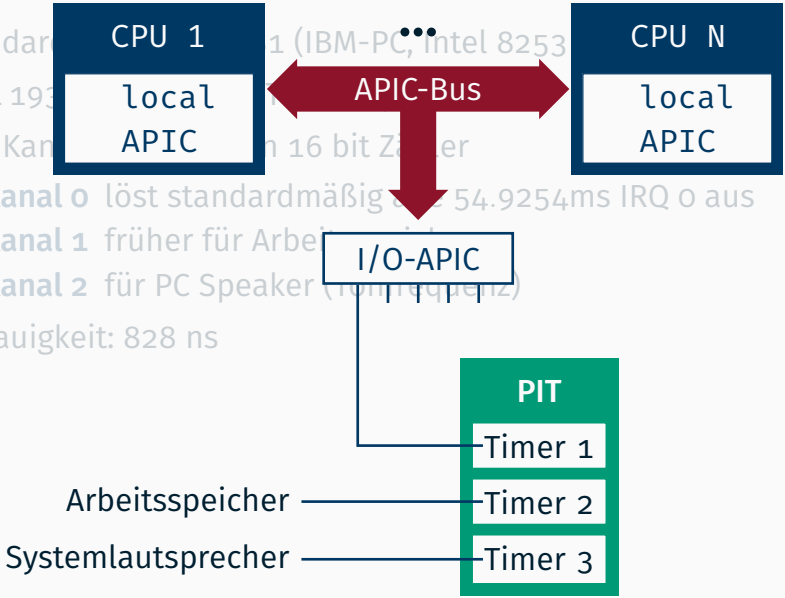
Zeitgeber

Programmable Interval Timer (PIT)

- Standardtimer seit 1981 (IBM-PC, Intel 8253/8254)
- ca. 1 193 182 Hz ($= \frac{1}{3}$ NTSC-Freq.)
- 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)
- Genauigkeit: 828 ns

Programmable Interval Timer (PIT)

- Standar… …1 (IBM-PC, Intel 8253
- ca. 1 193…
- drei Kan… …n 16 bit Zä…er
  - **Kanal 0** löst standardmäßig … 54.9254ms IRQ 0 aus
  - **Kanal 1** früher für Arbeit…
  - **Kanal 2** für PC Speaker (Ton…frequenz)
- Genauigkeit: 828 ns



Programmable Interval Timer (PIT)

- Standardtimer seit 1981 (IBM-PC, Intel 8253/8254)
- ca. 1 193 182 Hz ($= \frac{1}{3}$ NTSC-Freq.)
- 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)
- Genauigkeit: 828 ns
- 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.)
- 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)
- Genauigkeit: 828 ns
- 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.)
- 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)
- Genauigkeit: 828 ns
- via PIC bzw. I/O APIC → (relativ) langsam
- *ausreichend für BS*

→ `machine/pit.h`

Programmable Interval Timer (PIT)

- Standardtimer seit 1981 (IBM-PC, Intel 8253/8254)
- ca. 1 193 182 Hz ($= \frac{1}{3}$ NTSC-Freq.)
- 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)
- Genauigkeit: 828 ns
- 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
  - ursprünglich Erhöhung mit jedem Clock-Signal
  - unterschiedliche Takte abhängig vom Stromsparmodus
  - 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 Stromsparmodus
  - 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 Stromsparmodus
  - 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
- ein 64 bit Zähler
  - min. drei 32 oder 64 bit breite Vergleichseinrichtungen
  - konfigurierbarer Interrupt bei Gleichheit
- Genauigkeit: 100 ns oder besser

LAPIC Timer

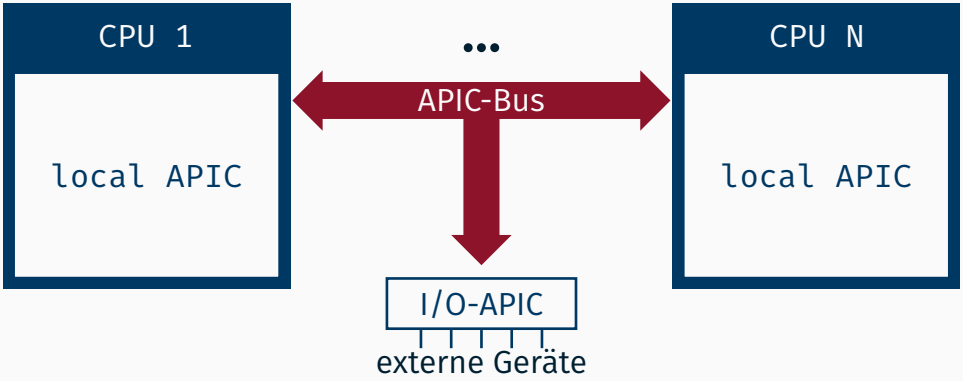
- $\geq$ 100 MHz
- 32 bit Zähler
- Taktfrequenz wie CPU Busfrequenz
  - abhängig vom System
  - aber unabhängig von Stromsparmodus
  - Interrupt geht nur an den entsprechenden Kern
  - 8 Möglichkeiten (bis $\frac{1}{128}$ Busfrequenz) durch Vorteiler
- Genauigkeit: 10 ns oder besser

LAPIC Timer

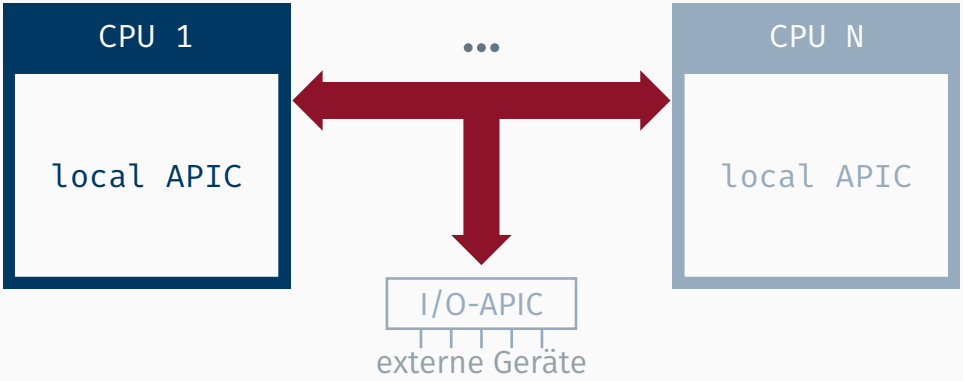
- $\geq$ 100 MHz
- 32 bit Zähler
- Taktfrequenz wie CPU Busfrequenz
  - abhängig vom System
  - aber unabhängig von Stromsparmodus
  - Interrupt geht nur an den entsprechenden Kern
  - 8 Möglichkeiten (bis $\frac{1}{128}$ Busfrequenz) durch Vorteiler
- Genauigkeit: 10 ns oder besser

Perfekt für unsere Bedürfnisse

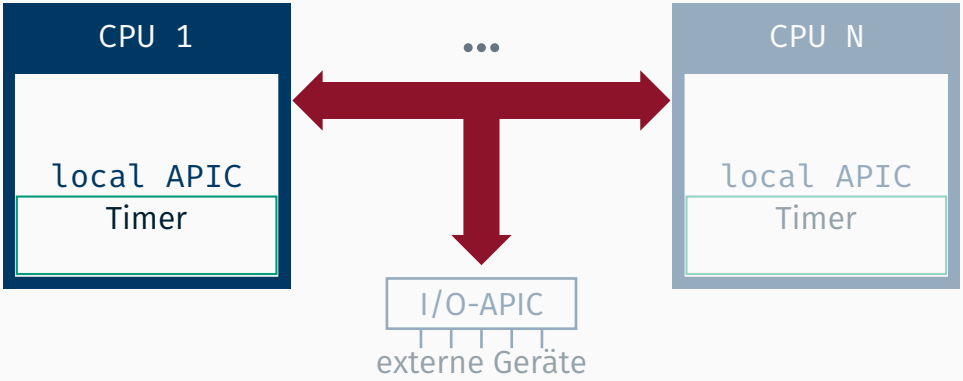
Funktionsweise des LAPIC Timers



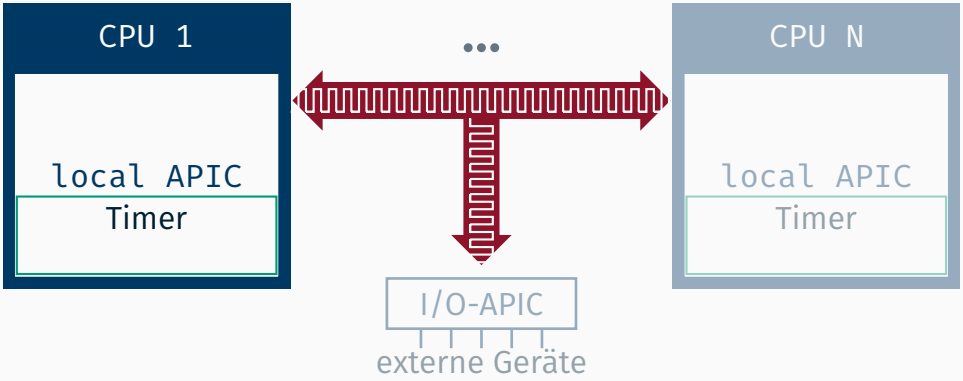
Funktionsweise des LAPIC Timers



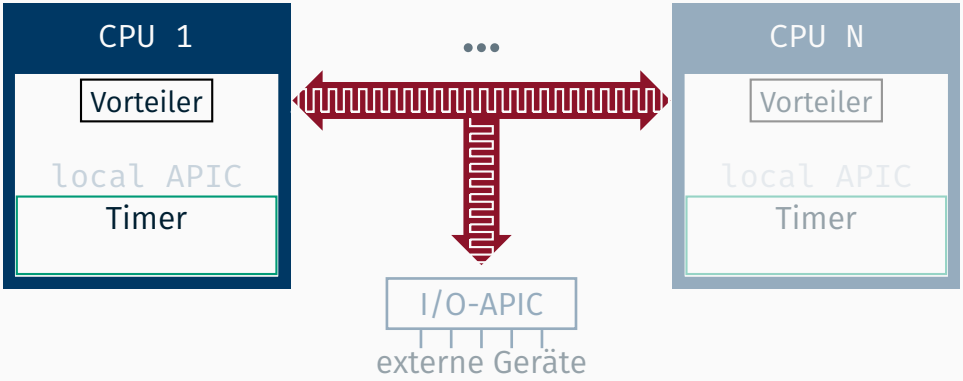
Funktionsweise des LAPIC Timers



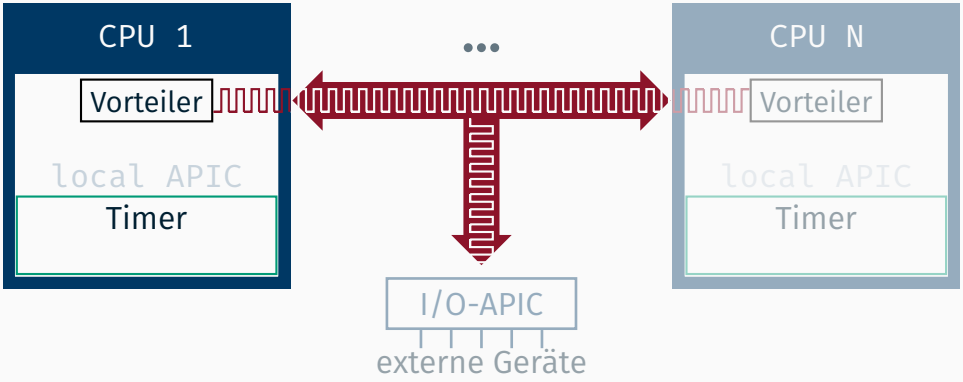
Funktionsweise des LAPIC Timers



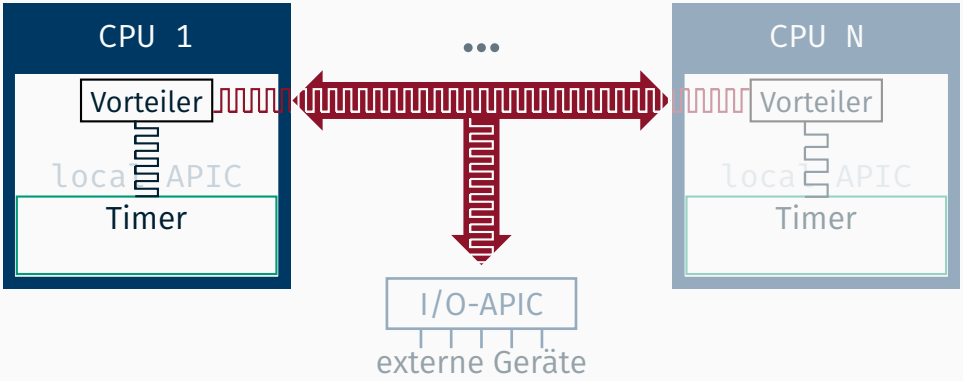
Funktionsweise des LAPIC Timers



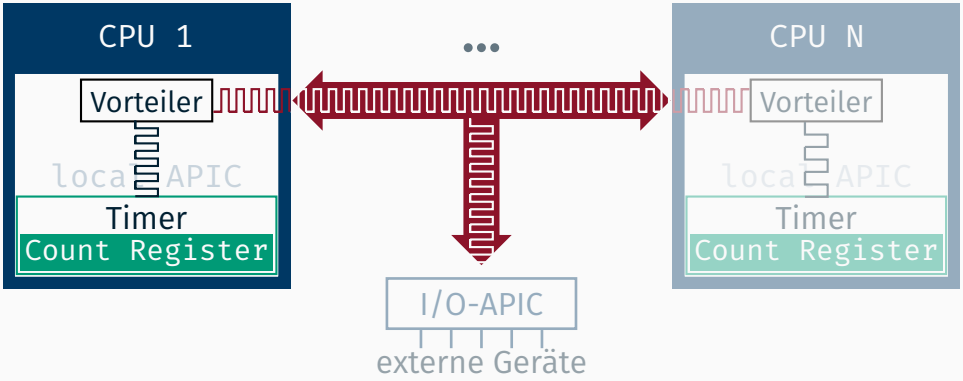
Funktionsweise des LAPIC Timers



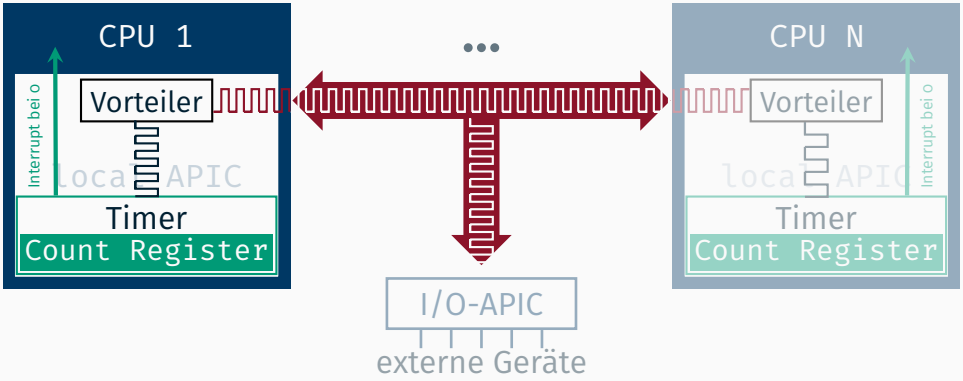
Funktionsweise des LAPIC Timers



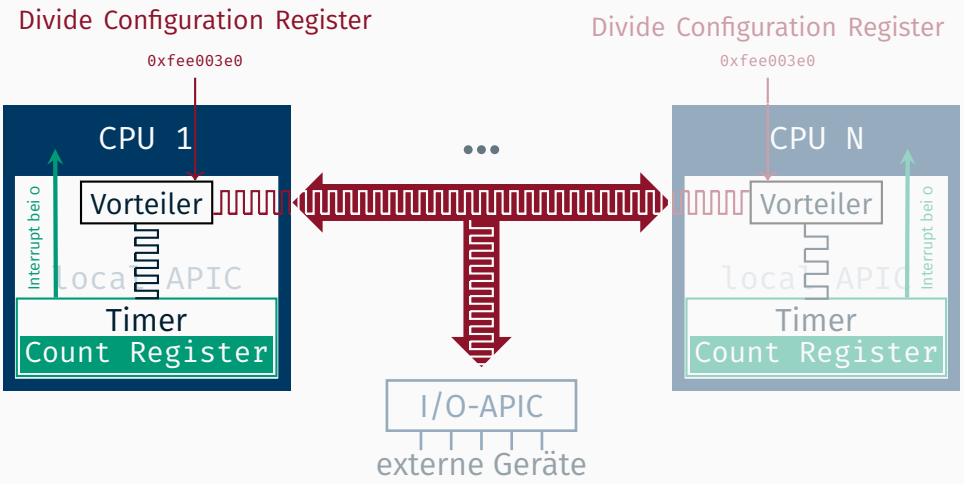
Funktionsweise des LAPIC Timers



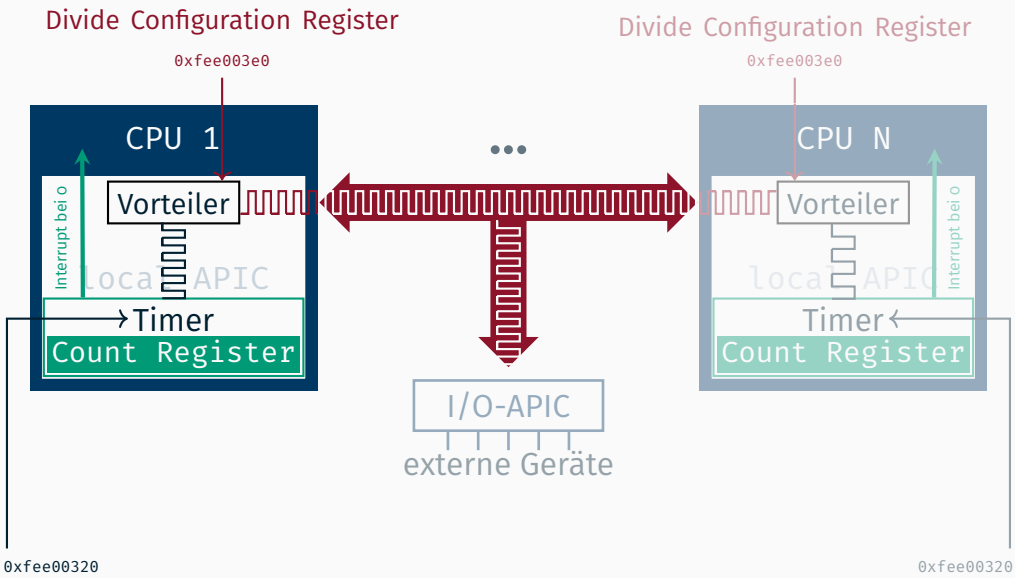
Funktionsweise des LAPIC Timers



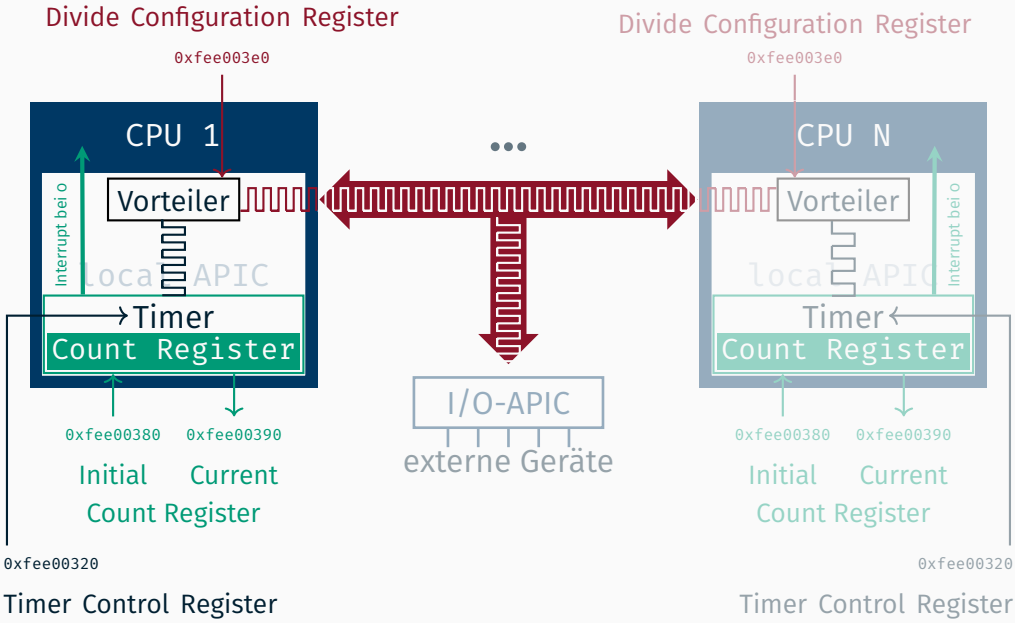
Funktionsweise des LAPIC Timers



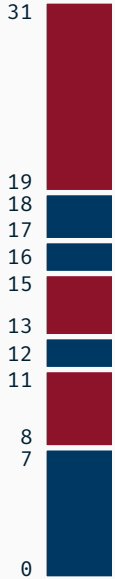
Funktionsweise des LAPIC Timers



Funktionsweise des LAPIC Timers



Aufbau des Timer Control Register Eintrags



Betriebsmodus: einmalig (`0`) oder periodisch (`1`)
Interrupt-Mask: Interrupt aktiv (`0`) oder inaktiv (`1`)

Zustellstatus Interrupt Nachricht noch unterwegs? **(RO)**

Interrupt Vektor: Nummer in der Vektortabelle (32-255)

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 $\frac{Bustakt}{Vorteiler}$ dekrementiert wird
- bei 0 wird – sofern aktiviert – ein Interrupt ausgelöst
- je nach Betriebsmodus wird gestoppt oder wieder neu begonnen

Umsetzung

Watch

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()`

Watch

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()`

- Scheduling läuft nun auf Epilogebene
  (d.h. automatische Synchronisation der Ready-Liste)

Watch

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()`

- Scheduling läuft nun auf Epilogebene
  (d.h. automatische Synchronisation der Ready-Liste)
- somit `Guarded_Scheduler` für Anwendungsebene notwendig

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 **Milli**sekunde 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 **Milli**sekunde 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 **Milli**sekunde zurück)
   - benötigt zur Konfiguration die ebenfalls noch zu implementierende Funktion `LAPIC::Timer::set`
   - Datei `lapic_registers.h` bietet passende Structs und Zugriffsfunktionen

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle *n* Mikrosekunden ausgelöst werden

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

- Interrupt soll alle n Mikrosekunden ausgelöst werden
- Startwertzähler $\mathit{initial} = \frac{n \cdot \texttt{LAPIC::timer\_ticks()}}{\mathit{Vorteiler} \cdot 1000}$ berechnen, dabei gilt $\mathit{Vorteiler} = 2^x$ mit $x \in \{0, \ldots, 7\}$

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

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

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

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

n	5 000 000 µs = 5 s
`LAPIC::Timer::ticks`	1 000 000

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

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

n	5 000 000 µs = 5 s
`LAPIC::Timer::ticks`	1 000 000
Vorteiler	$1 = 2^0$

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

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

n	5 000 000 µs = 5 s
`LAPIC::Timer::ticks`	1 000 000
Vorteiler	$1 = 2^0$
initial	5 000 000 000

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

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

n	5 000 000 µs = 5 s
`LAPIC::Timer::ticks`	1 000 000
Vorteiler	$1 = 2^0$
initial	5 000 000 000 ↯

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

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

n	5 000 000 µs = 5 s
`LAPIC::Timer::ticks`	1 000 000
Vorteiler	$2 = 2^1$
initial	

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

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

n	5 000 000 µs = 5 s
`LAPIC::Timer::ticks`	1 000 000
Vorteiler	$2 = 2^1$
initial	2 500 000 000

LAPIC-Timer einstellen

2. Initialen Wert und Vorteiler korrekt setzen

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

n	5 000 000 µs = 5 s
`LAPIC::Timer::ticks`	1 000 000
Vorteiler	$2 = 2^1$
initial	2 500 000 000 ✓

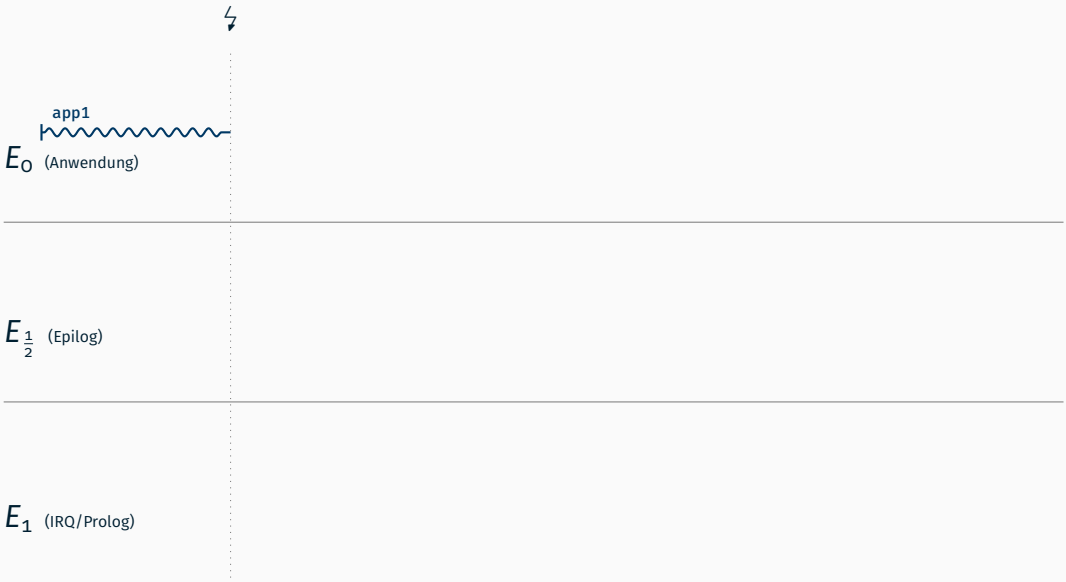
Ablaufbeispiel (Standardfall)

E_0 (Anwendung)

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

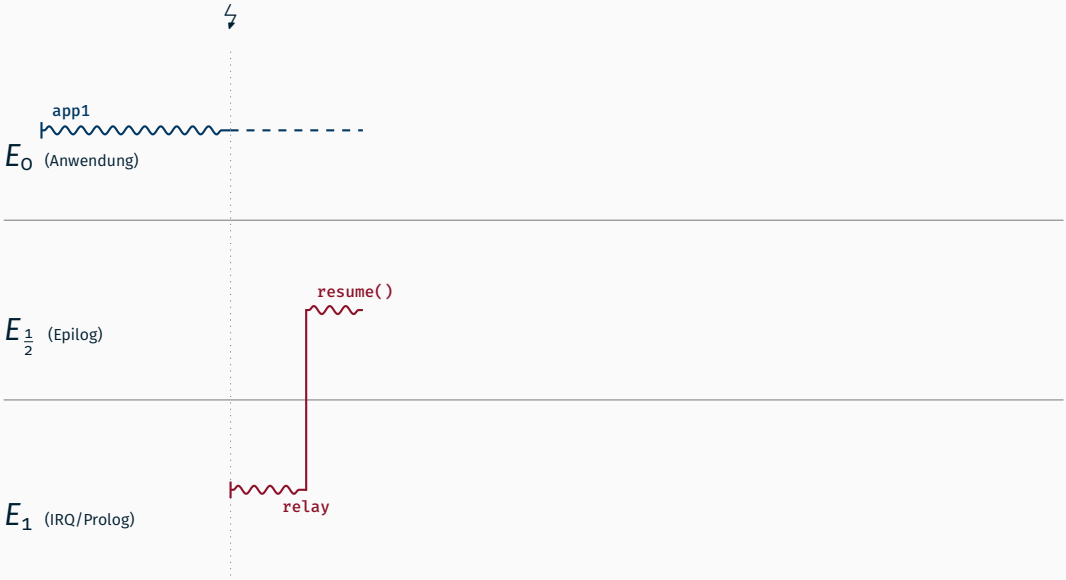
E_1 (IRQ/Prolog)

Ablaufbeispiel (Standardfall)

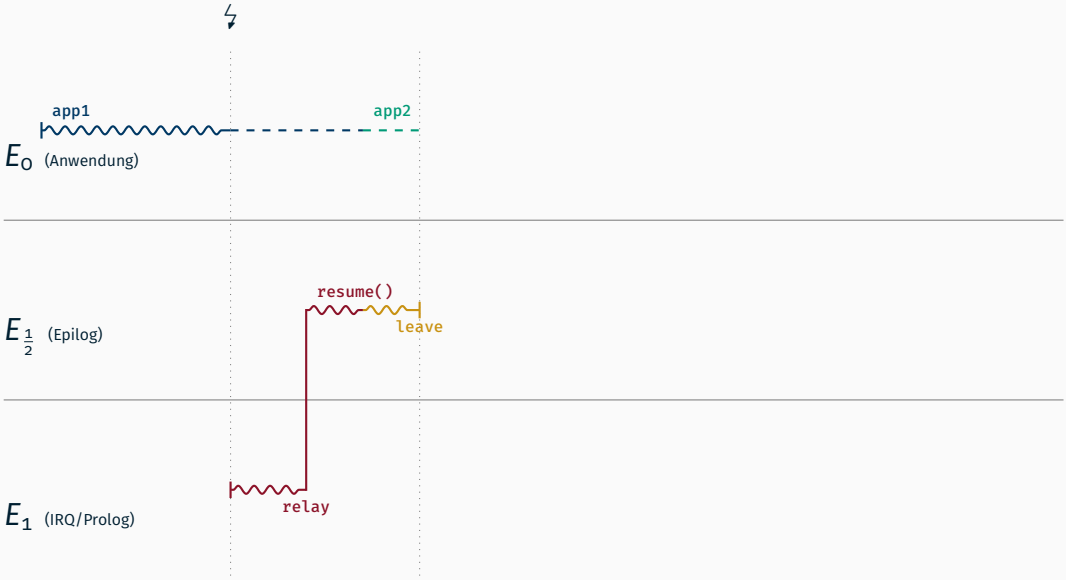


Ablaufbeispiel (Standardfall)

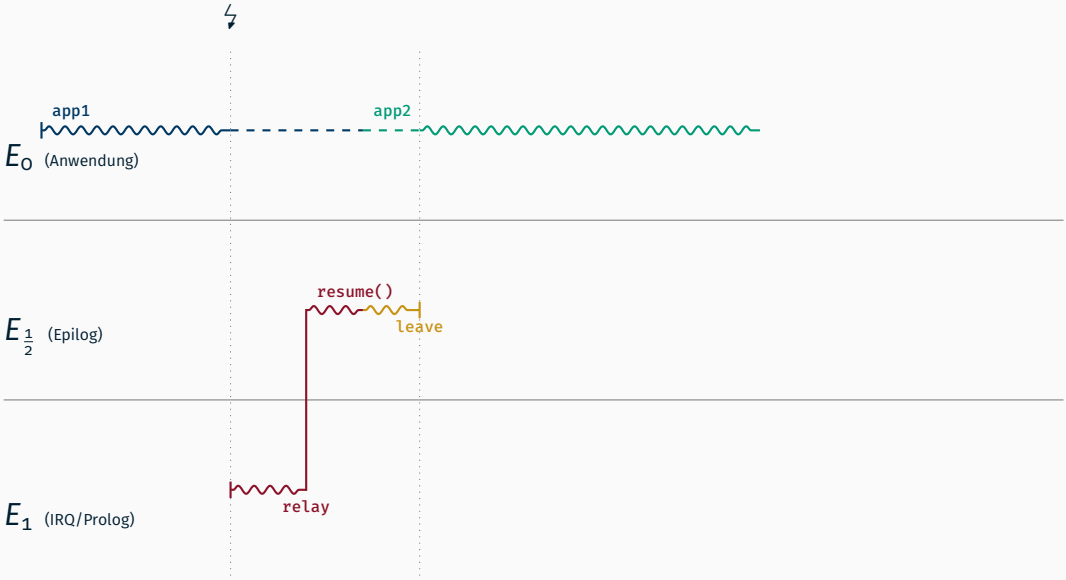
Ablaufbeispiel (Standardfall)



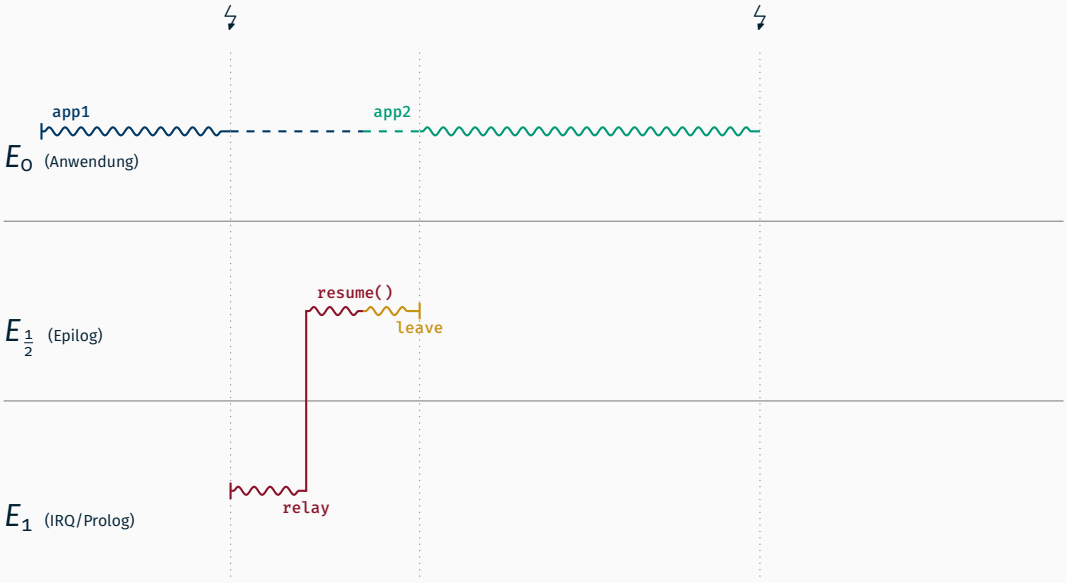
Ablaufbeispiel (Standardfall)



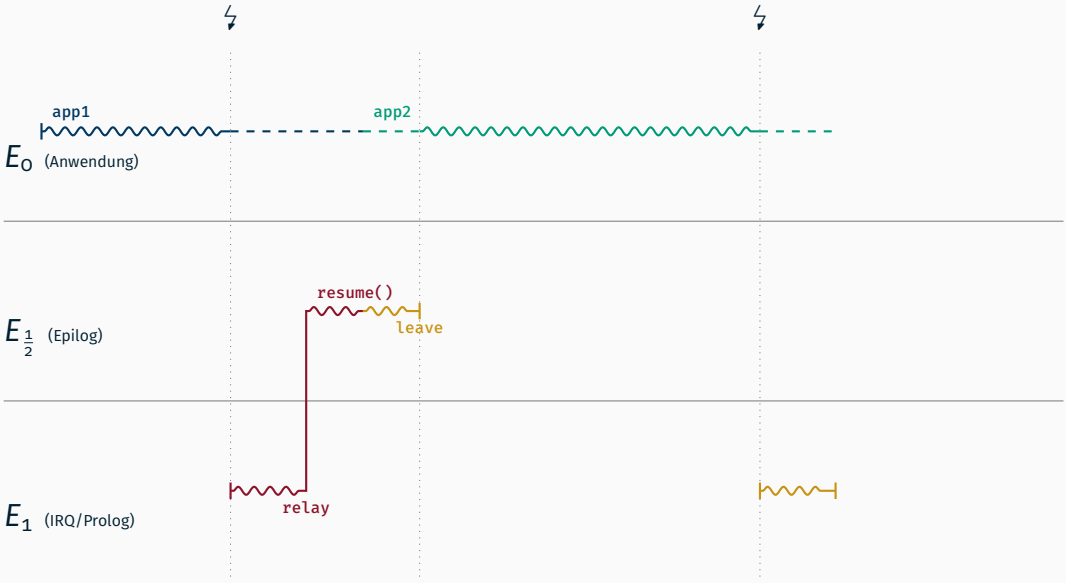
Ablaufbeispiel (Standardfall)



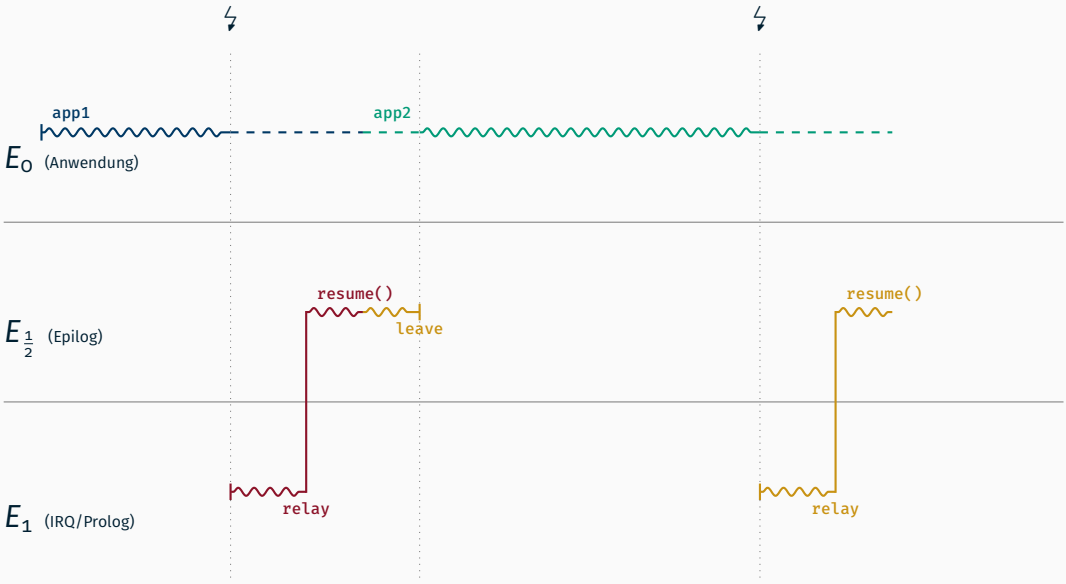
Ablaufbeispiel (Standardfall)



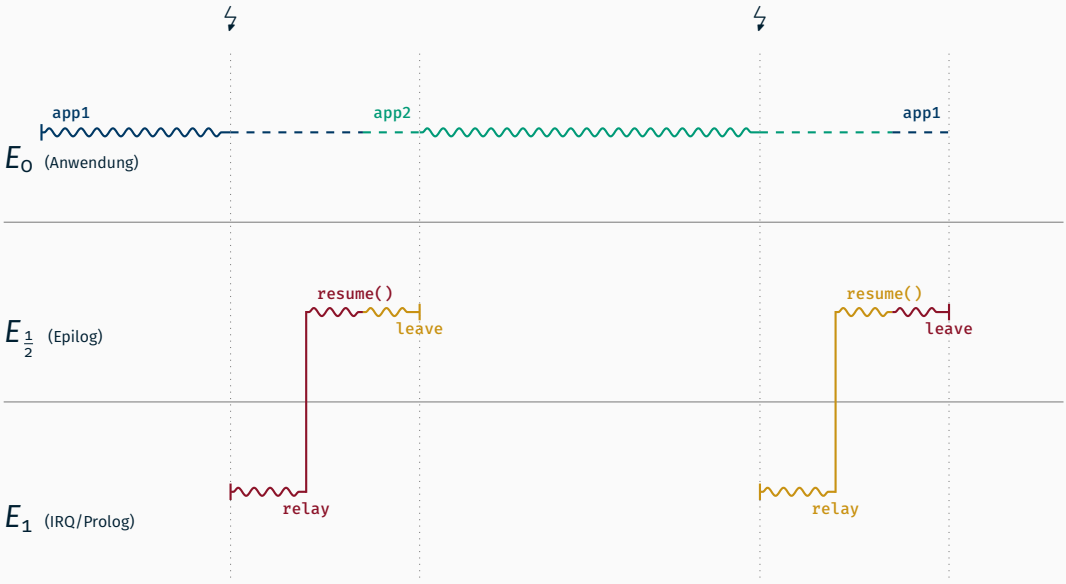
Ablaufbeispiel (Standardfall)



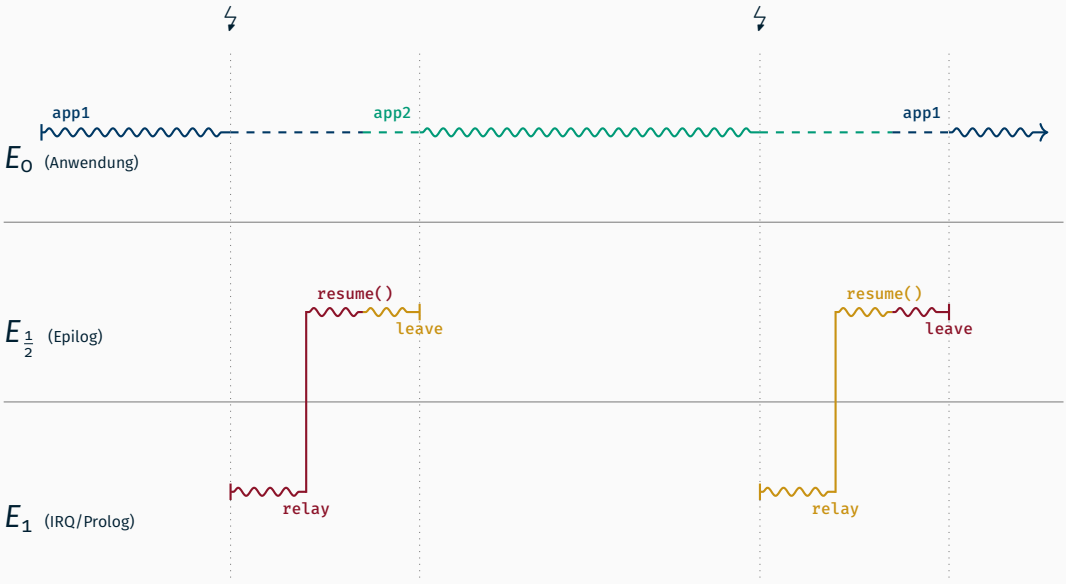
Ablaufbeispiel (Standardfall)



Ablaufbeispiel (Standardfall)



Ablaufbeispiel (Standardfall)



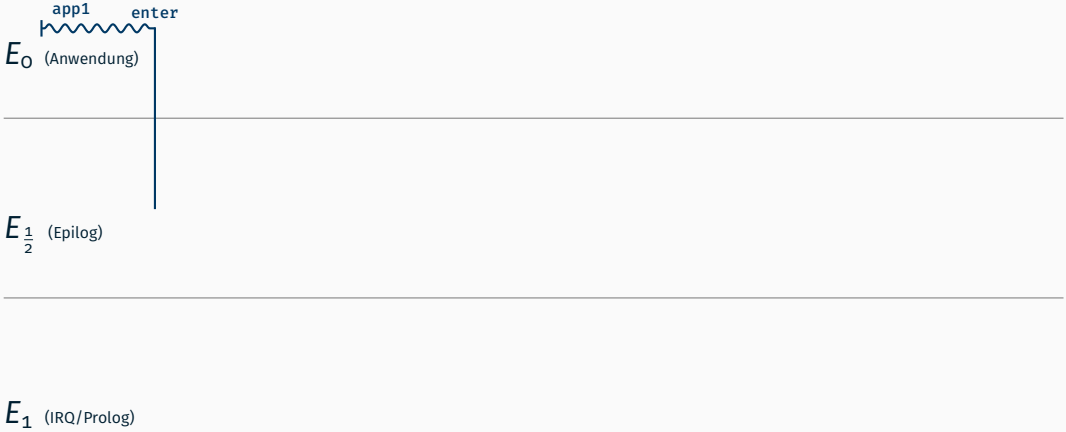
Ablaufbeispiel bei Faden in Systemebene

E_0 (Anwendung)

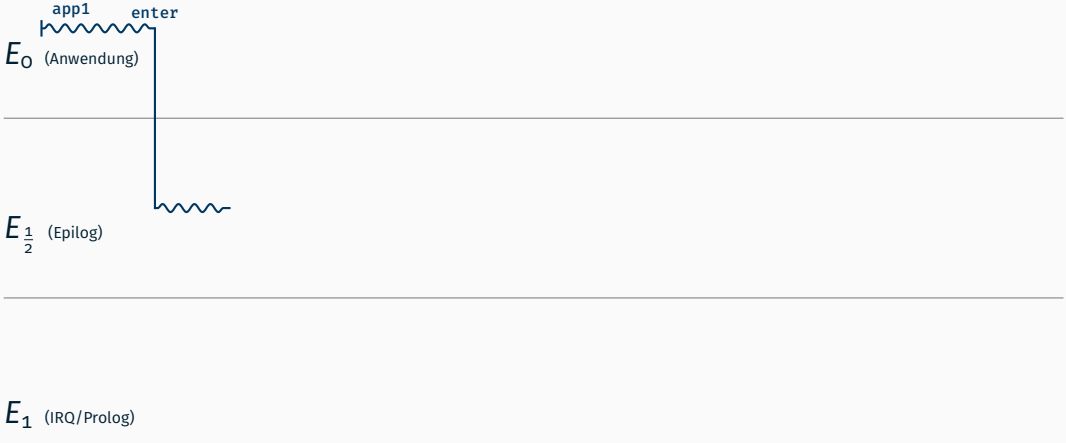
$E_{\frac{1}{2}}$ (Epilog)

E_1 (IRQ/Prolog)

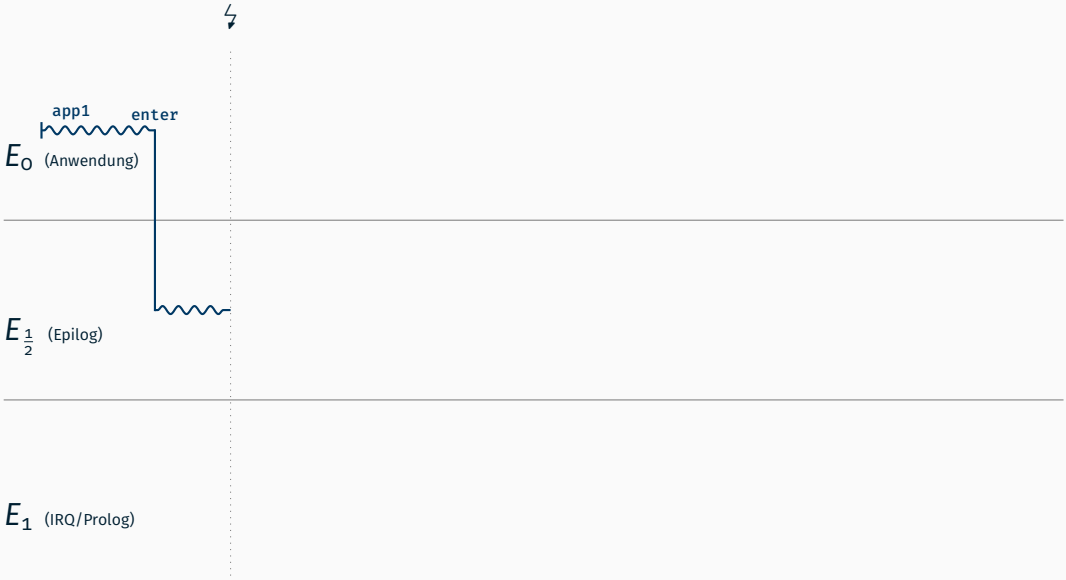
Ablaufbeispiel bei Faden in Systemebene



Ablaufbeispiel bei Faden in Systemebene

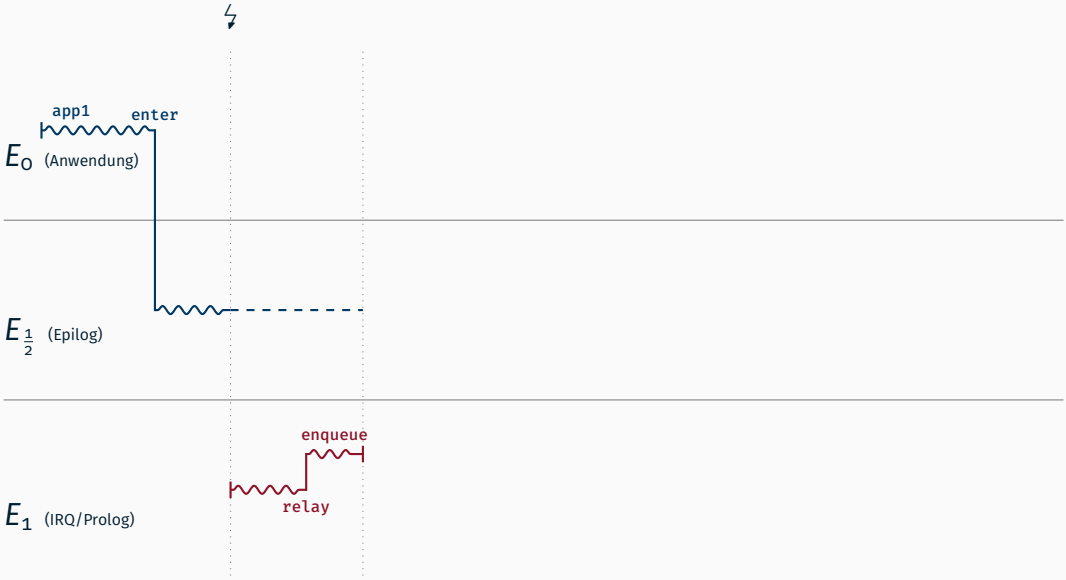


Ablaufbeispiel bei Faden in Systemebene

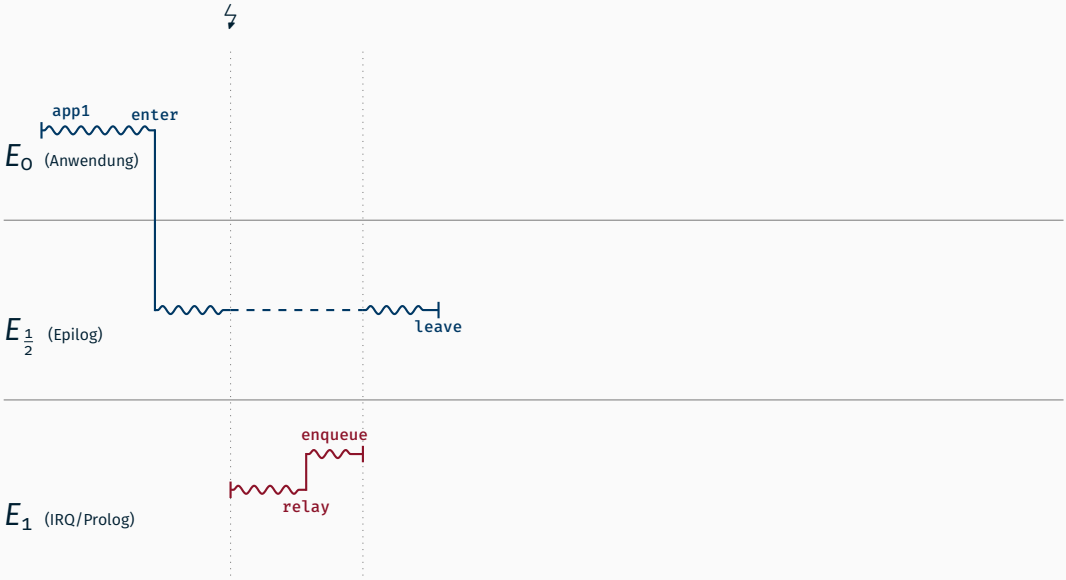


Ablaufbeispiel bei Faden in Systemebene

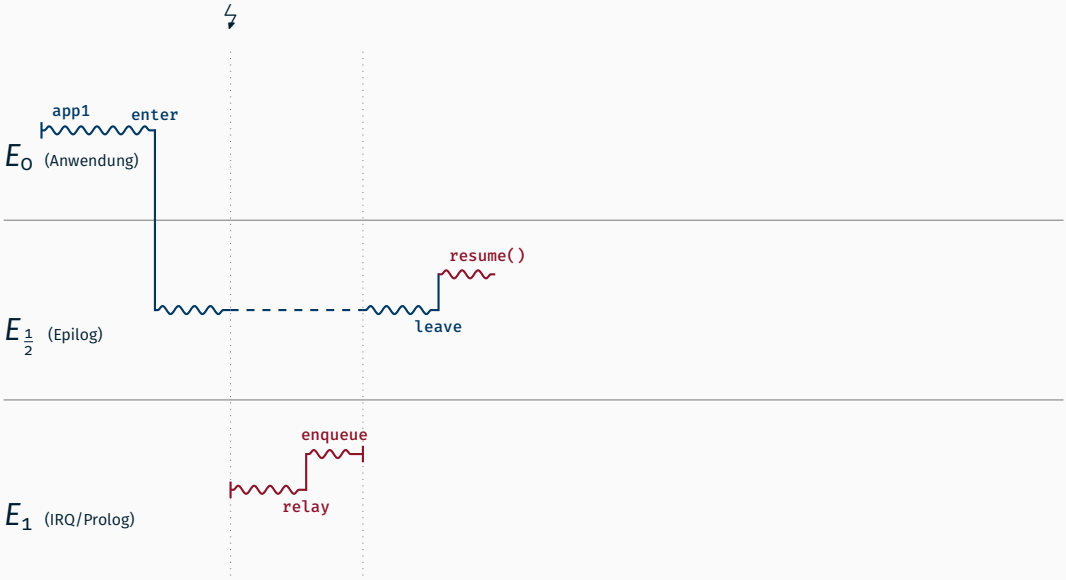
Ablaufbeispiel bei Faden in Systemebene



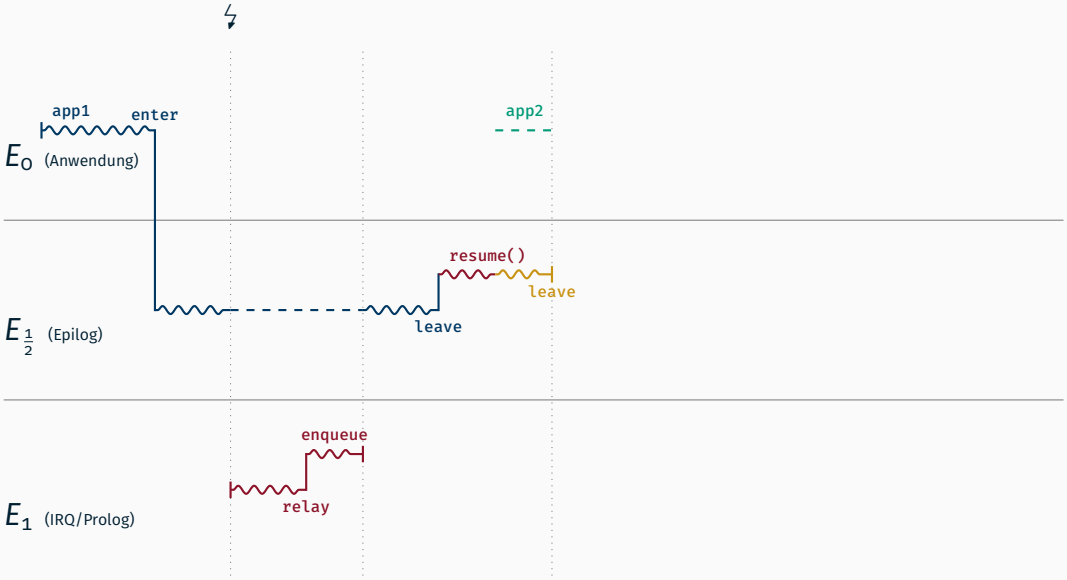
Ablaufbeispiel bei Faden in Systemebene



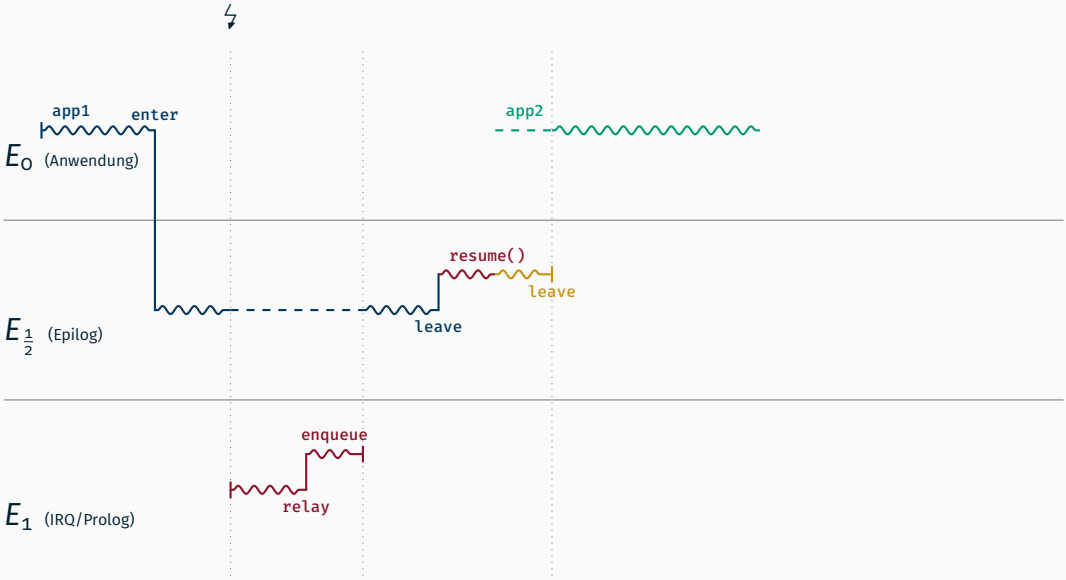
Ablaufbeispiel bei Faden in Systemebene



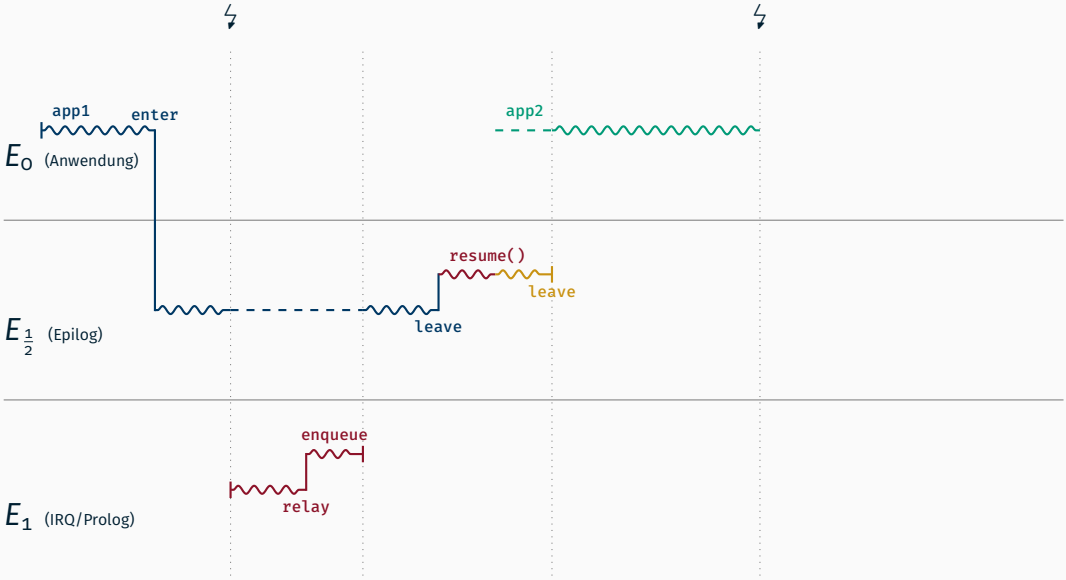
Ablaufbeispiel bei Faden in Systemebene



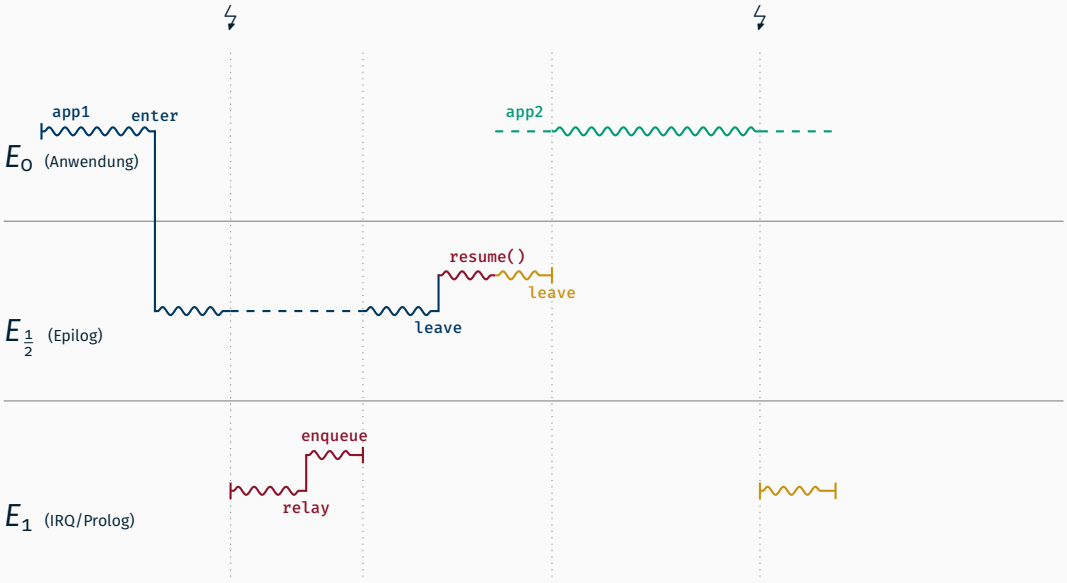
Ablaufbeispiel bei Faden in Systemebene



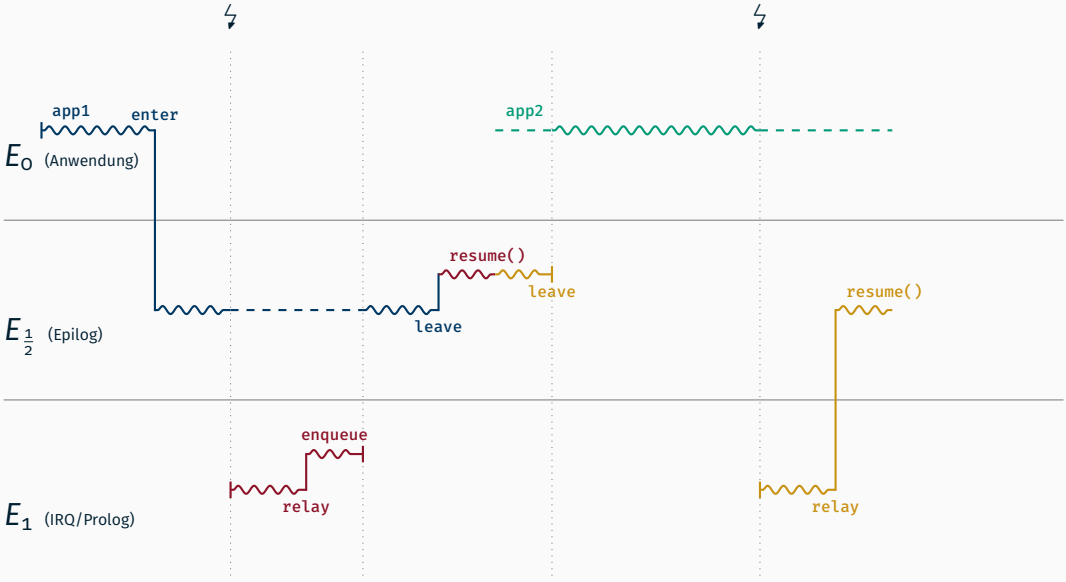
Ablaufbeispiel bei Faden in Systemebene



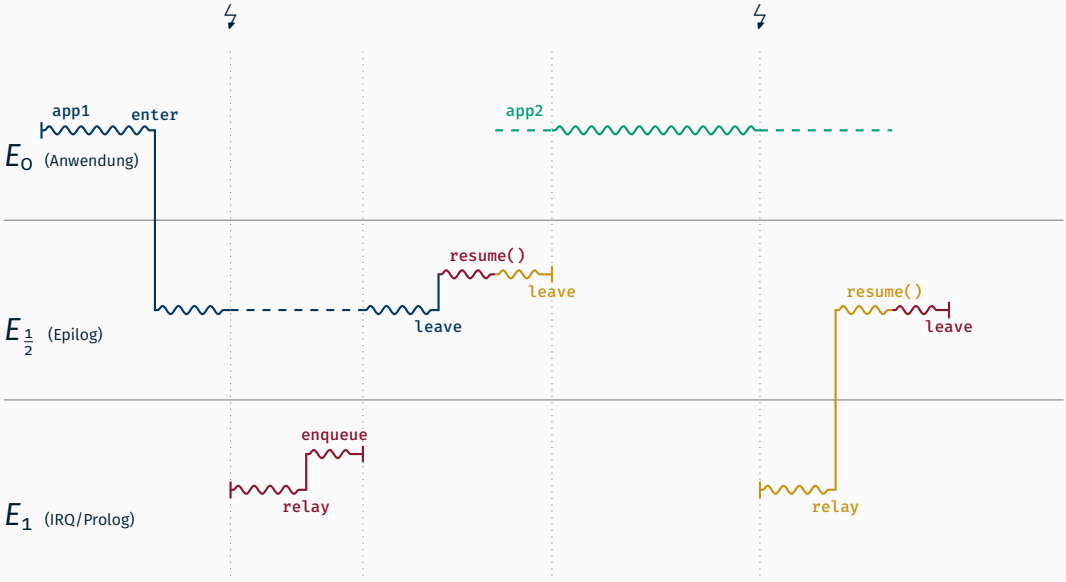
Ablaufbeispiel bei Faden in Systemebene



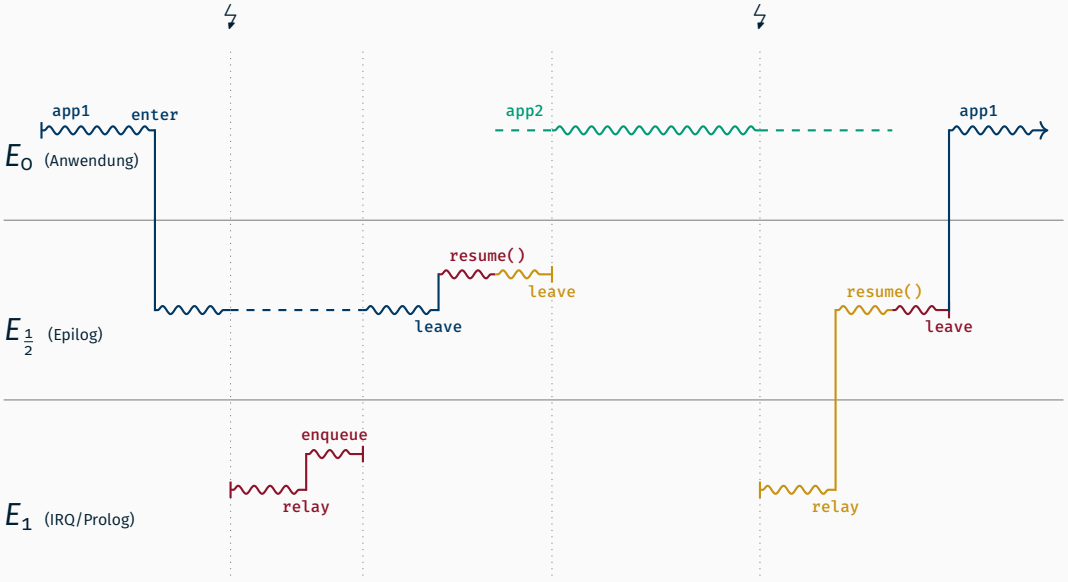
Ablaufbeispiel bei Faden in Systemebene



Ablaufbeispiel bei Faden in Systemebene



Ablaufbeispiel bei Faden in Systemebene



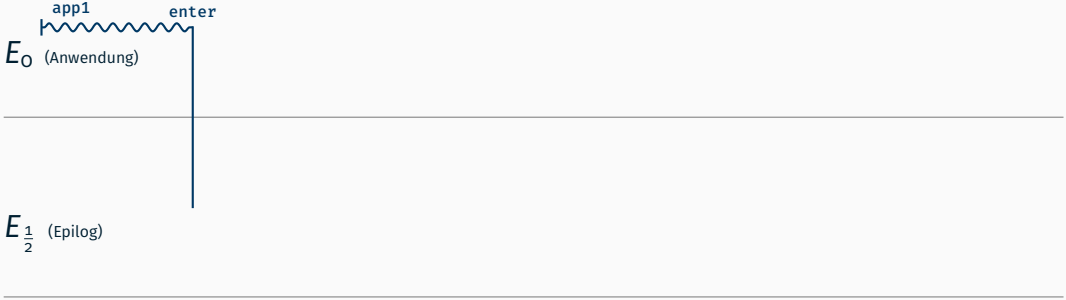
Ablaufbeispiel mit neuem Thread

E_0 (Anwendung)

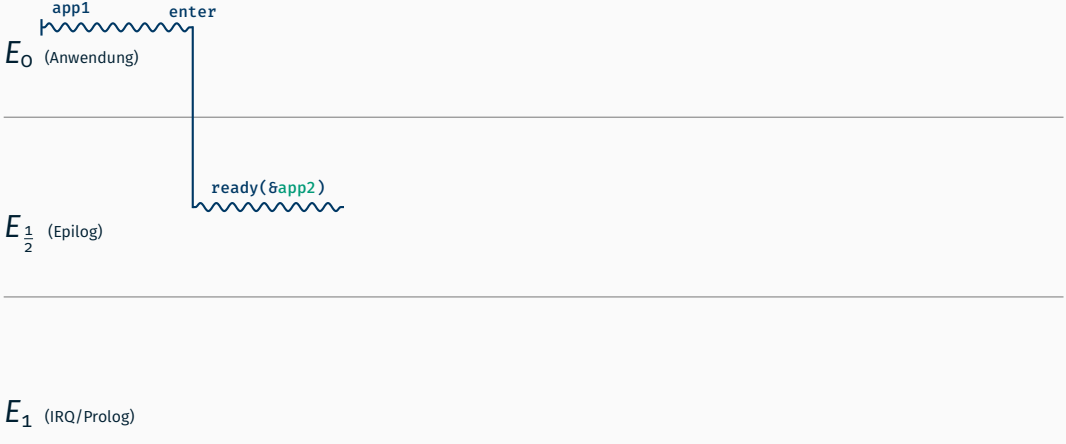
$E_{\frac{1}{2}}$ (Epilog)

E_1 (IRQ/Prolog)

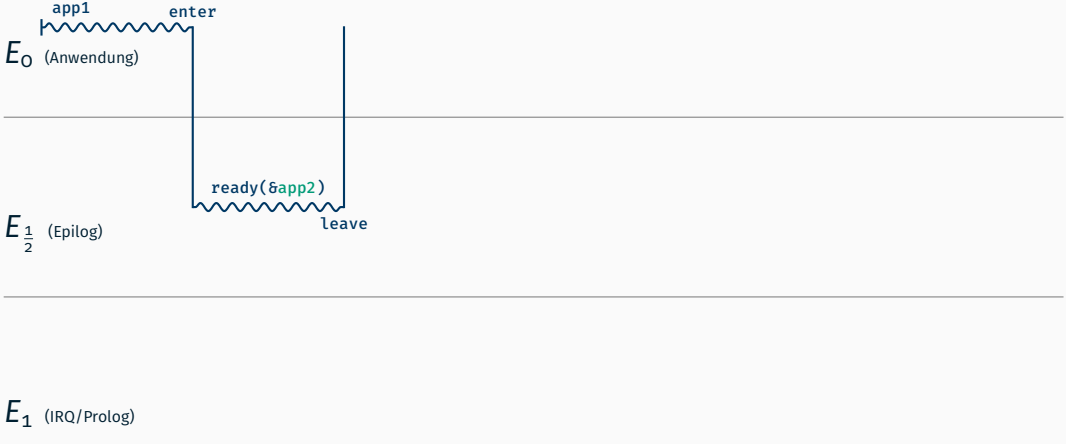
Ablaufbeispiel mit neuem Thread



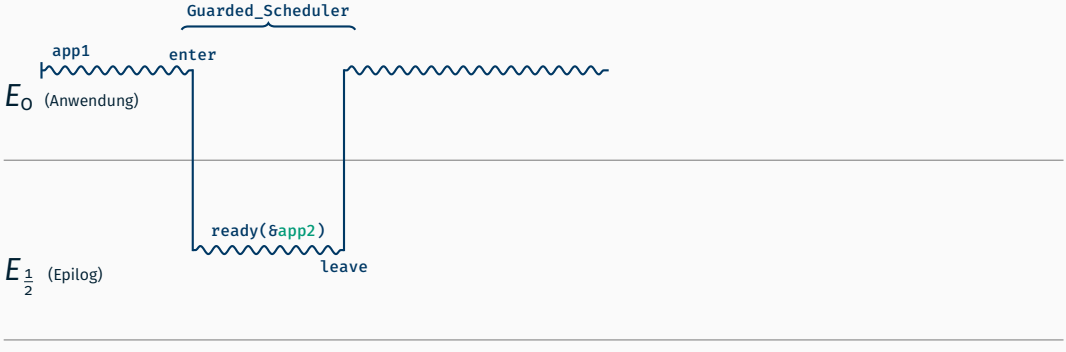
Ablaufbeispiel mit neuem Thread



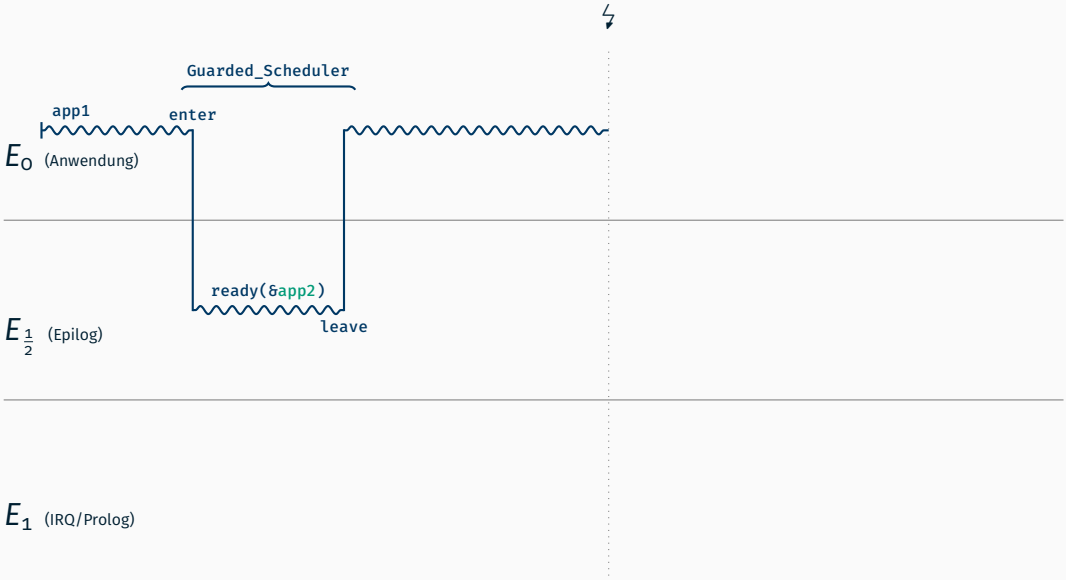
Ablaufbeispiel mit neuem Thread



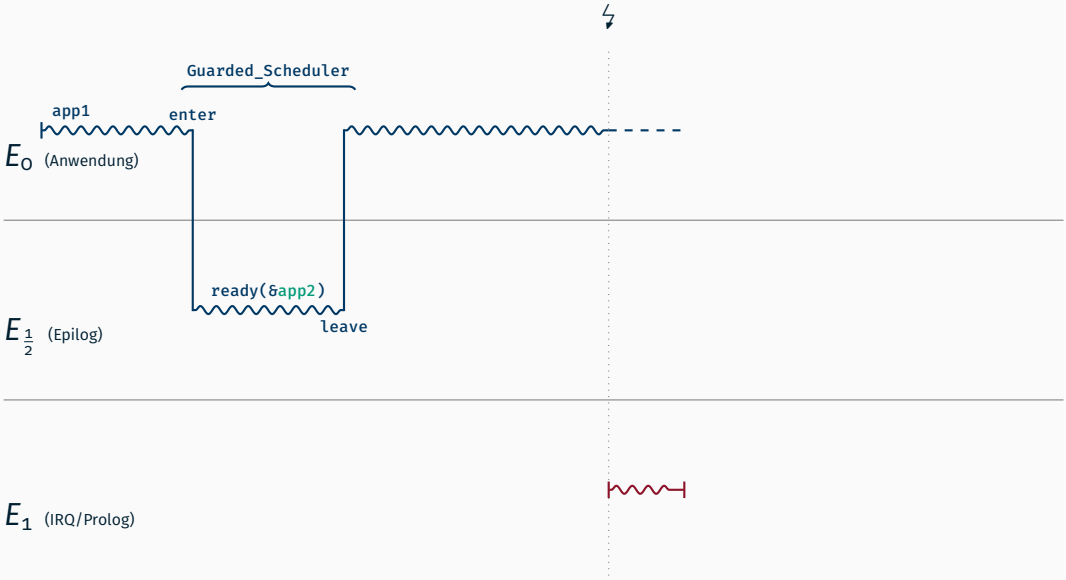
Ablaufbeispiel mit neuem Thread



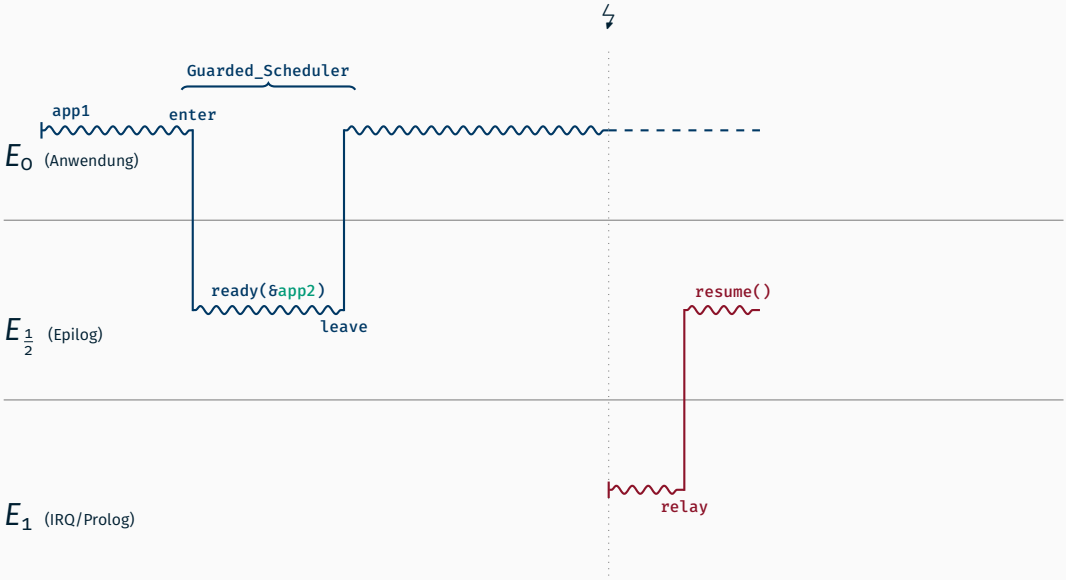
Ablaufbeispiel mit neuem Thread



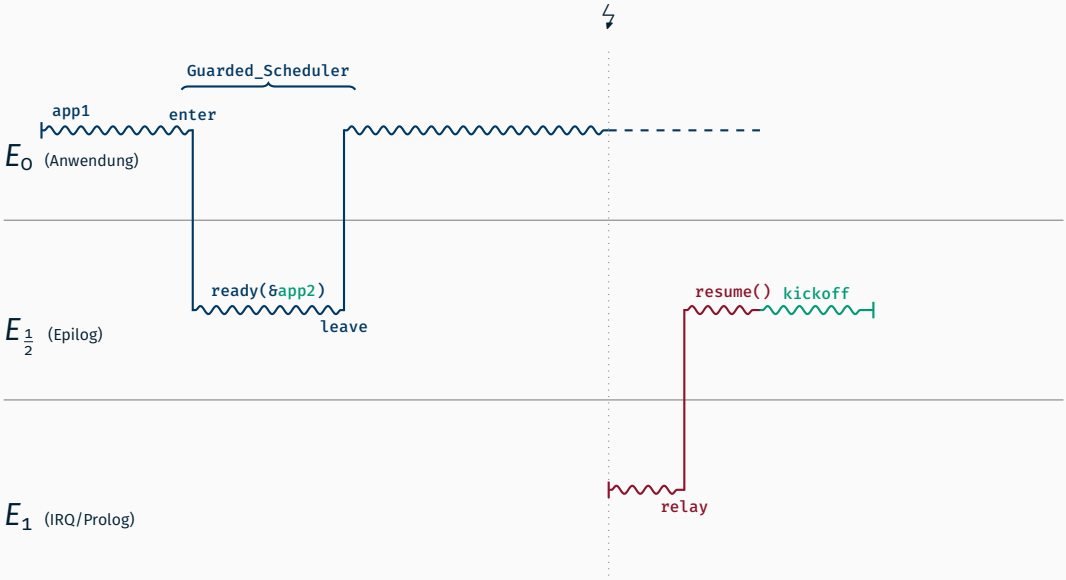
Ablaufbeispiel mit neuem Thread



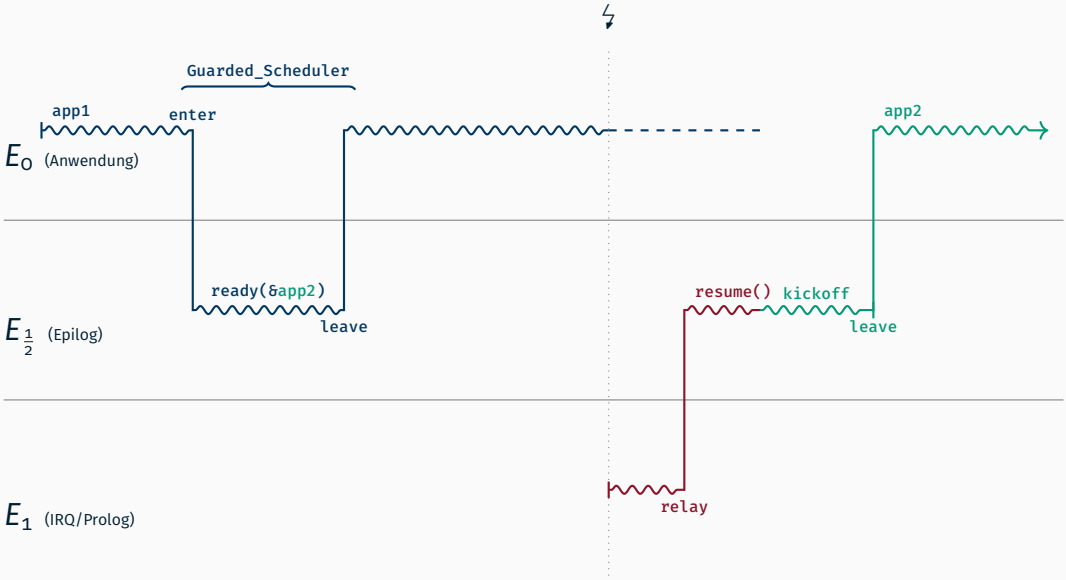
Ablaufbeispiel mit neuem Thread



Ablaufbeispiel mit neuem Thread



Ablaufbeispiel mit neuem Thread



Besonderheiten

Anwendungsfaden beenden

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

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
(Was muss man tun, damit der Thread sich selbst killen kann?)

Anwendungsfaden beenden

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

OOStuBS keine Änderung: aus der Ready-Liste entfernen
bzw. Kill-Flag setzen und bei `resume` prüfen
(Was muss man tun, damit der Thread sich selbst killen kann?)

MPStuBS der Anwendungsfaden kann auch 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
(Was muss man tun, damit der Thread sich selbst killen kann?)

MPStuBS der Anwendungsfaden kann auch 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
(Was muss man tun, damit der Thread sich selbst killen kann?)

MPStuBS der Anwendungsfaden kann auch 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
(Was muss man tun, damit der Thread sich selbst killen kann?)

MPStuBS der Anwendungsfaden kann auch 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
(Was muss man tun, damit der Thread sich selbst killen kann?)

MPStuBS der Anwendungsfaden kann auch 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

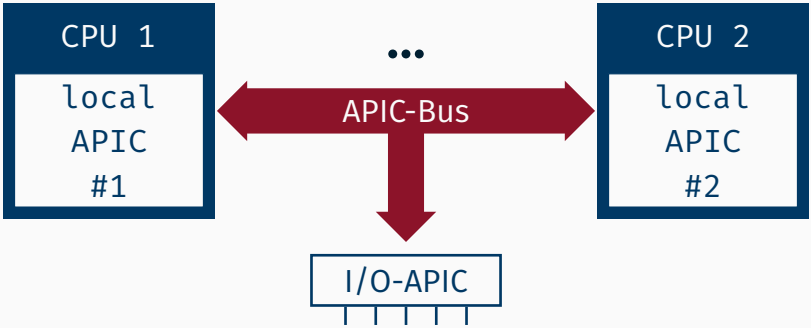
Präemptives Beenden mittels `Scheduler::kill(Thread&)`

OOStuBS keine Änderung: aus der Ready-Liste entfernen
bzw. Kill-Flag setzen und bei `resume` prüfen
(Was muss man tun, damit der Thread sich selbst killen kann?)

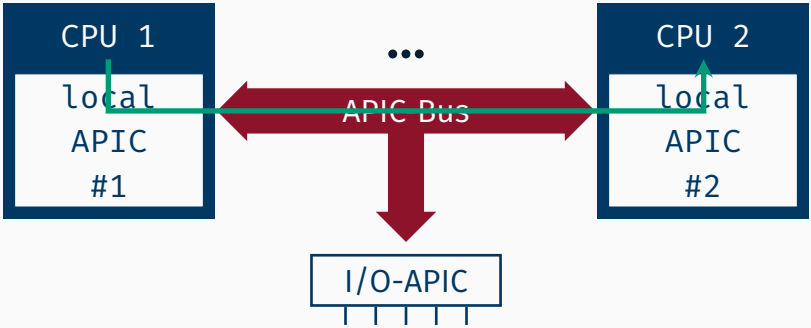
MPStuBS der Anwendungsfaden kann auch gerade auf einer anderen CPU laufen

- 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

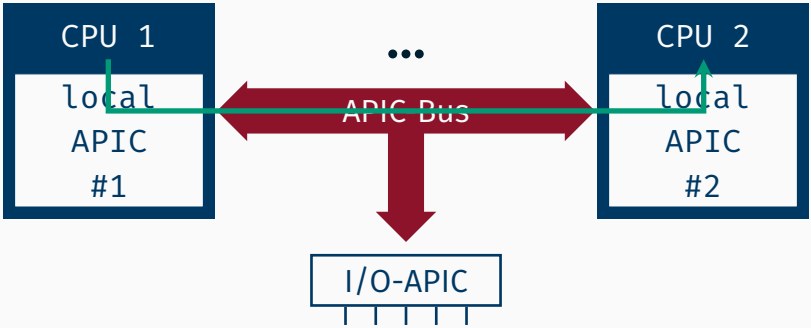
Inter Processor Interrupt



Inter Processor Interrupt

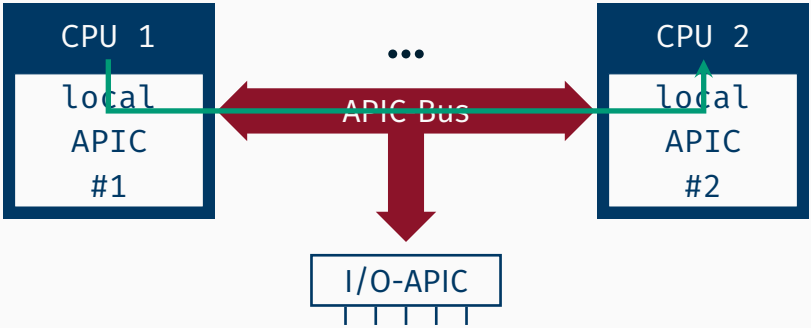


Inter Processor Interrupt




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

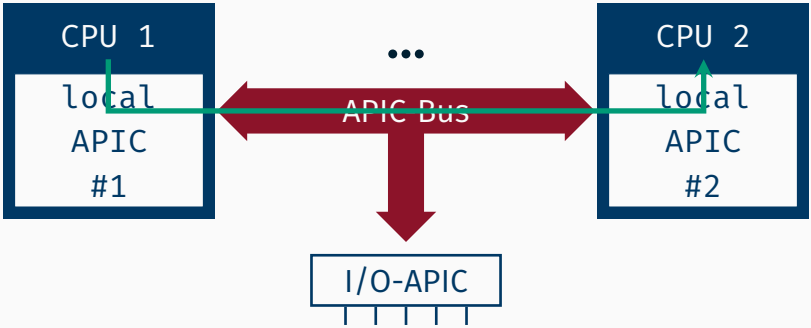
Inter Processor Interrupt




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

Interrupt

Inter Processor Interrupt




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

Empfänger

Interrupt

Der Teufel steckt im Detail

Was kann hier schon schief gehen?

```
lock[System::getCPUID()] = true;
```

Der Teufel steckt im Detail

Was kann hier schon schief gehen?

```
lock[System::getCPUID()] = true;
```

Viel - diese Zeile wird nicht atomar ausgeführt:

```
; Array lock an Adresse 0x2000
call <System::getCPUID>
mov [rax+0x2000], 0x1
```

Der Teufel steckt im Detail

Was kann hier schon schief gehen?

```
lock[System::getCPUID()] = true;
```

Viel - diese Zeile wird nicht atomar ausgeführt:

```
; Array lock an Adresse 0x2000
call <System::getCPUID>
mov [rax+0x2000], 0x1
```

↯ Scheduler Interrupt

Der Teufel steckt im Detail

Was kann hier schon schief gehen?

```
lock[System::getCPUID()] = true;
```

Viel - diese Zeile wird nicht atomar ausgeführt:

```
; Array lock an Adresse 0x2000
call <System::getCPUID>
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*?

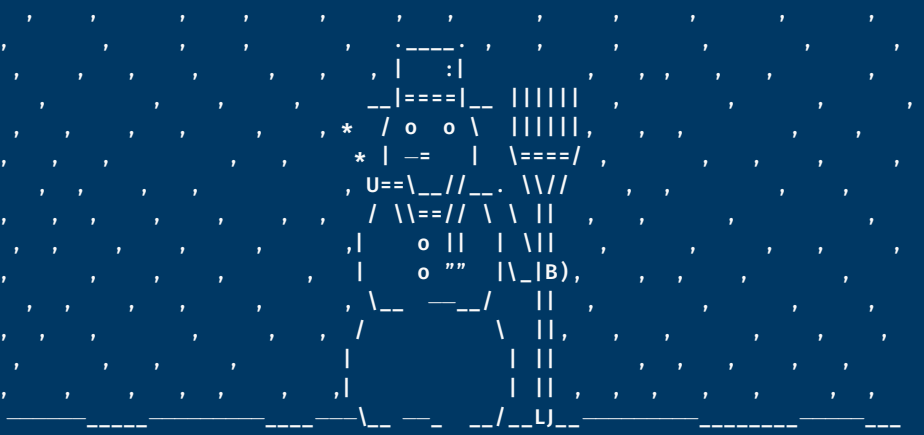
Fragen über Fragen

- 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?

weitere Fragen?



Wir wünschen euch ein frohes Fest

und einen guten Rutsch ins neue Jahr