

# Übung zu Betriebssysteme

## Interruptsynchronisation

---

21. & 22. November 2019

Andreas Ziegler, Bernhard Heinloth, Christian Eichler

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



Lehrstuhl für Verteilte Systeme  
und Betriebssysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
TECHNISCHE FAKULTÄT

# Motivation

---

**Interrupts verändern (potenziell) den Zustand des Systems**

**Interrupts verändern (potenziell) den Zustand des Systems**

```
main(){
    while(1){
        // ...
        consume();
        // ...
    }
}

IRQ_handler(){
    produce();
}
```

# Ohne Synchronisation

$E_0$  (Anwendung)

  |  
  | main()  
  |~~~~~

---

$E_1$  (IRQ)

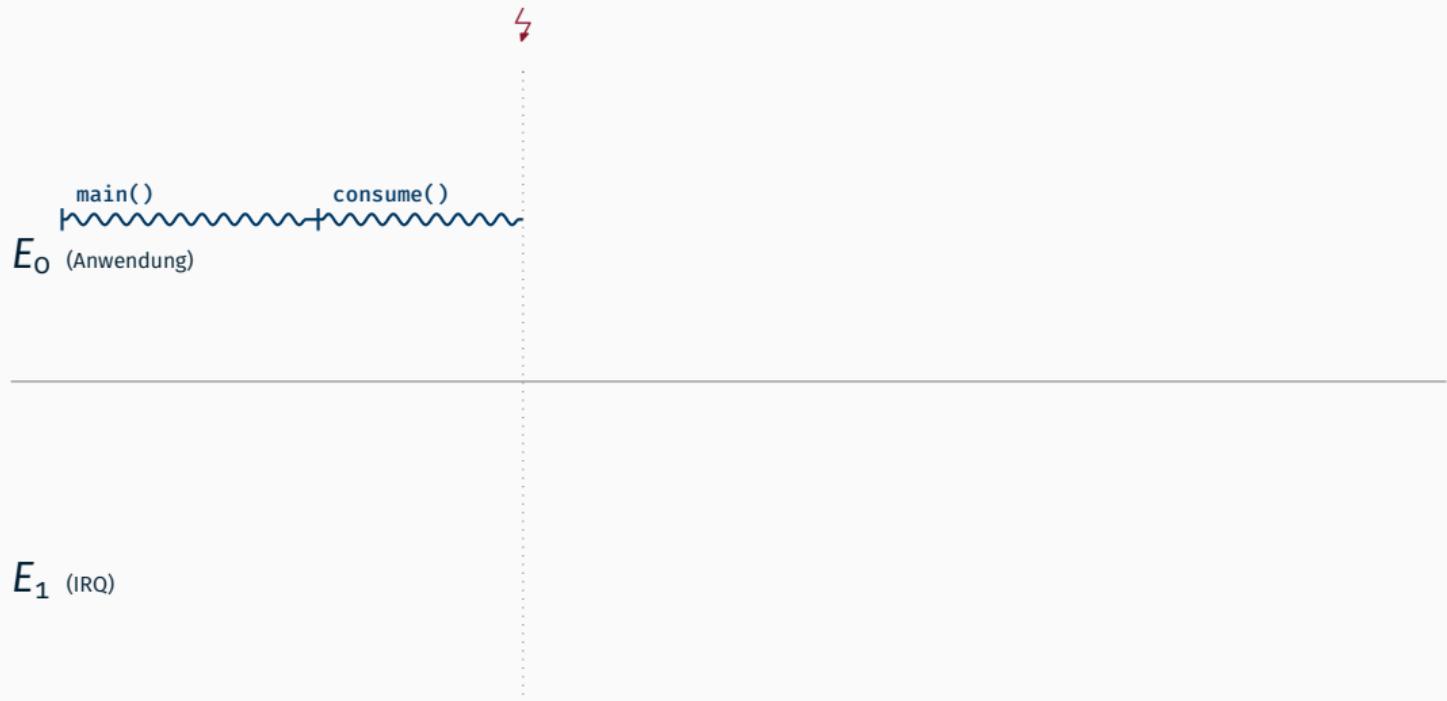
# Ohne Synchronisation



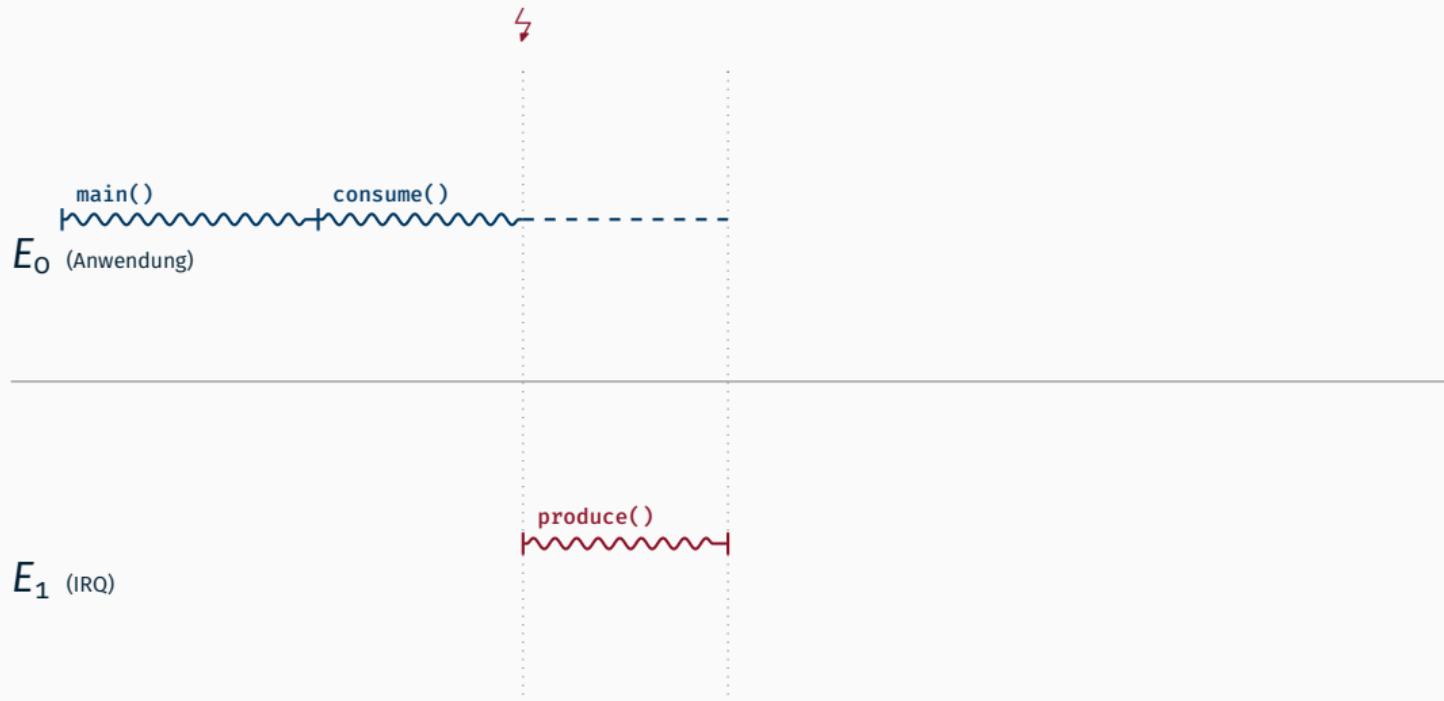
---

$E_1$  (IRQ)

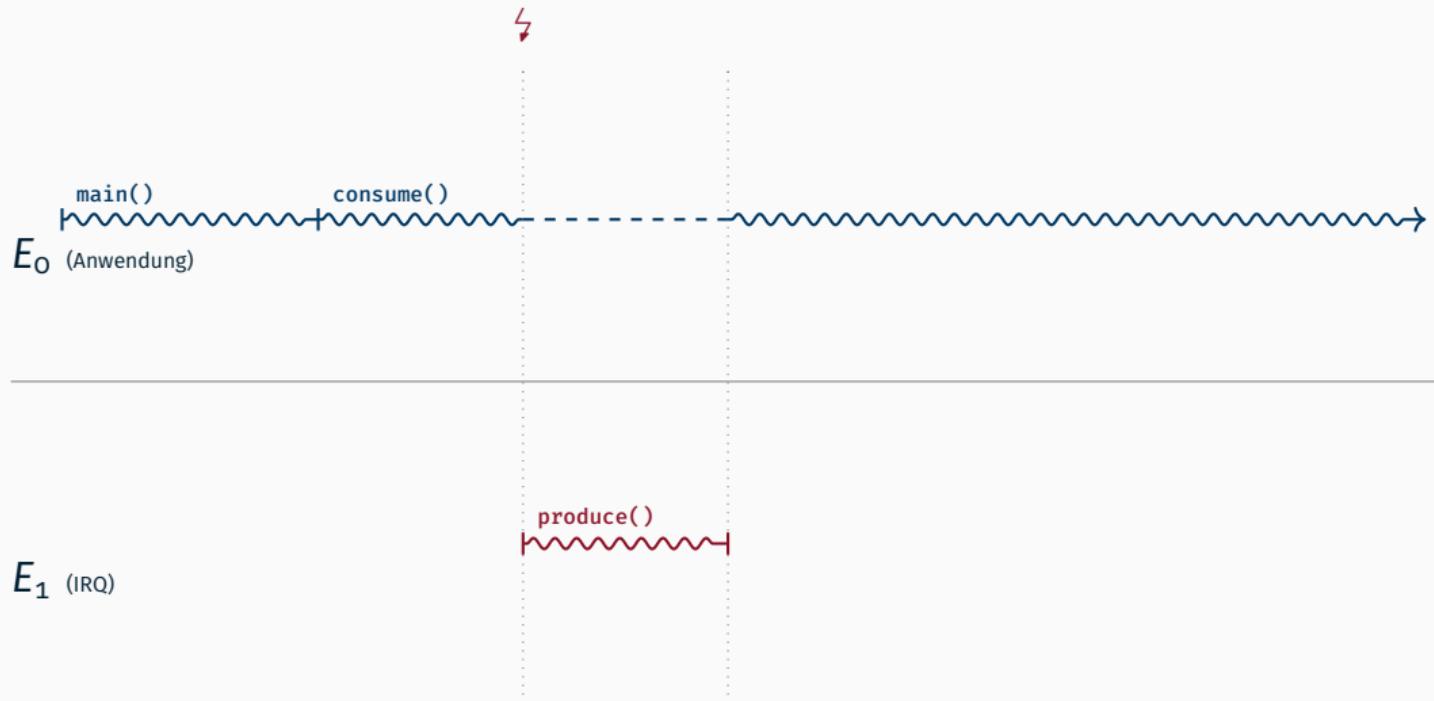
# Ohne Synchronisation



# Ohne Synchronisation



# Ohne Synchronisation



## Ohne Synchronisation

```
int buf[SIZE];
int pos = 0;

void produce(int data) {
    if (pos < SIZE)
        buf[pos++] = data;
}

int consume() {
    return pos > 0 ? buf[--pos] : -1;
}
```

# Ohne Synchronisation

**Lost Update möglich!**

## Bewährtes Hausmittel: Mutex

```
void produce(int data) {  
    mutex.lock();  
    if (pos < SIZE)  
        buf[pos++] = data;  
    mutex.unlock();  
}  
  
int consume() {  
    mutex.lock();  
    int r = pos > 0 ? buf[--pos] : -1;  
    mutex.unlock();  
    return r;  
}
```

# Bewährtes Hausmittel: Mutex

**Verklemmt sich!**

# Weiche Synchronisation

```
void produce(int data) {  
    if (pos < SIZE)  
        buf[pos++] = data;  
}  
  
int consume() {  
    int x, r = -1;  
    if (pos > 0)  
        do {  
            x = pos;  
            r = buf[x];  
        } while(!CAS(&pos, x, x-1));  
    return r;  
}
```

# Weiche Synchronisation

Funktioniert!

# Weiche Synchronisation

## Optimistischer Ansatz

- + keine Interruptsperre
- + kann sehr effizient sein
- kein generischer Ansatz
- kann sehr kompliziert werden
- fehleranfällig

# Weiche Synchronisation

## Optimistischer Ansatz

- + keine Interruptsperre
- + kann sehr effizient sein
- kein generischer Ansatz
- kann sehr kompliziert werden
- fehleranfällig

Betriebssystemarchitekt sollte Treiber- und Anwendungsentwicklern entgegenkommen  
→ für Erfolg des Betriebssystems entscheidend

# Harte Synchronisation

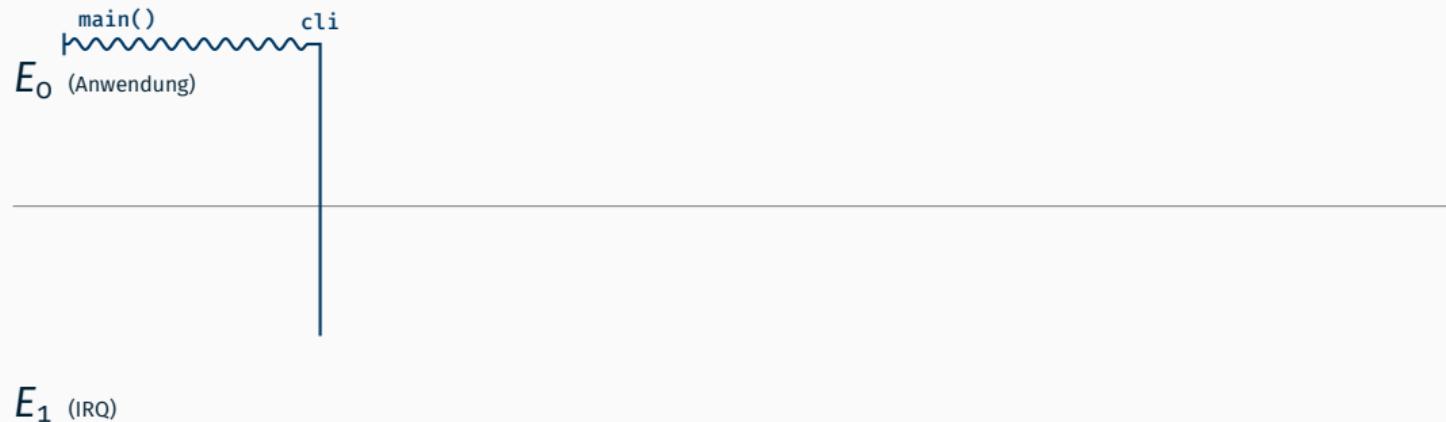
$E_0$  (Anwendung)

  |  
  +---+  
    main()  
    +---+  
        |  
        +---+  
          main()  
          +---+

---

$E_1$  (IRQ)

# Harte Synchronisation



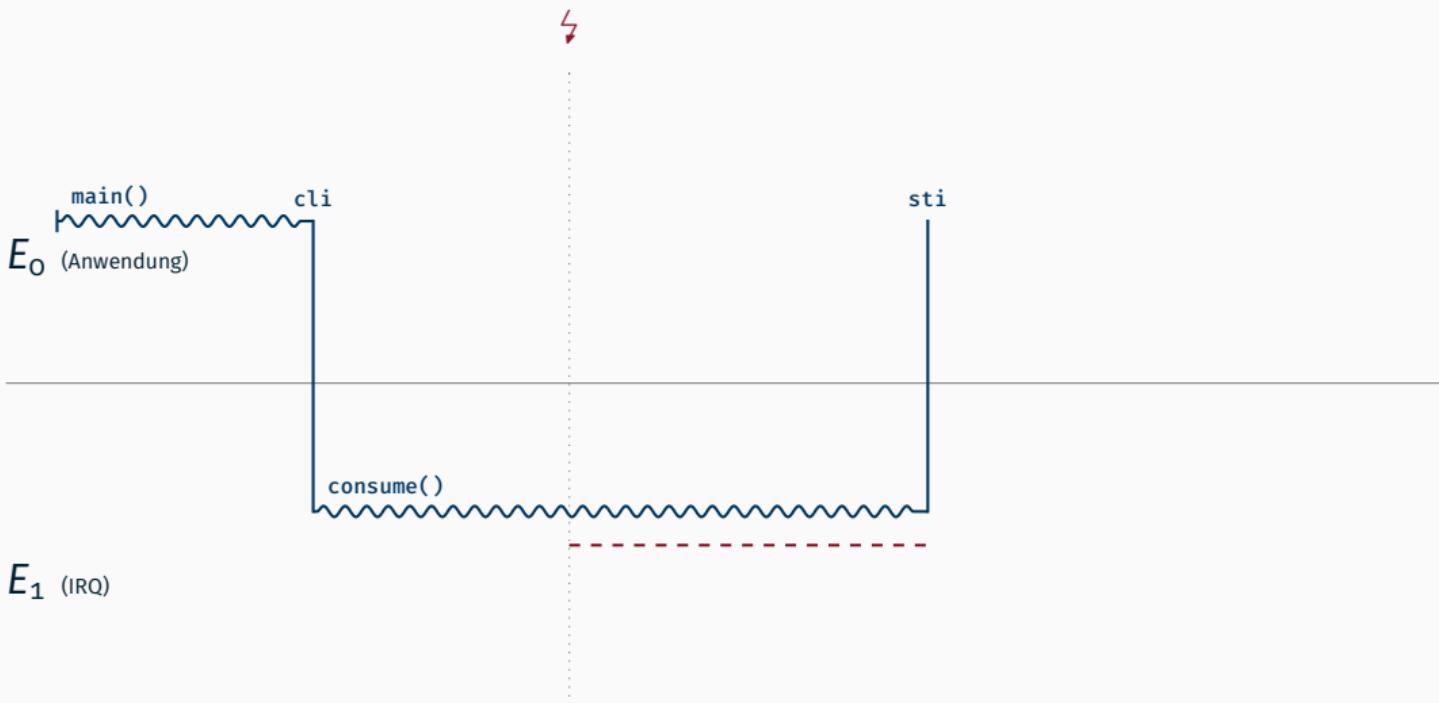
# Harte Synchronisation



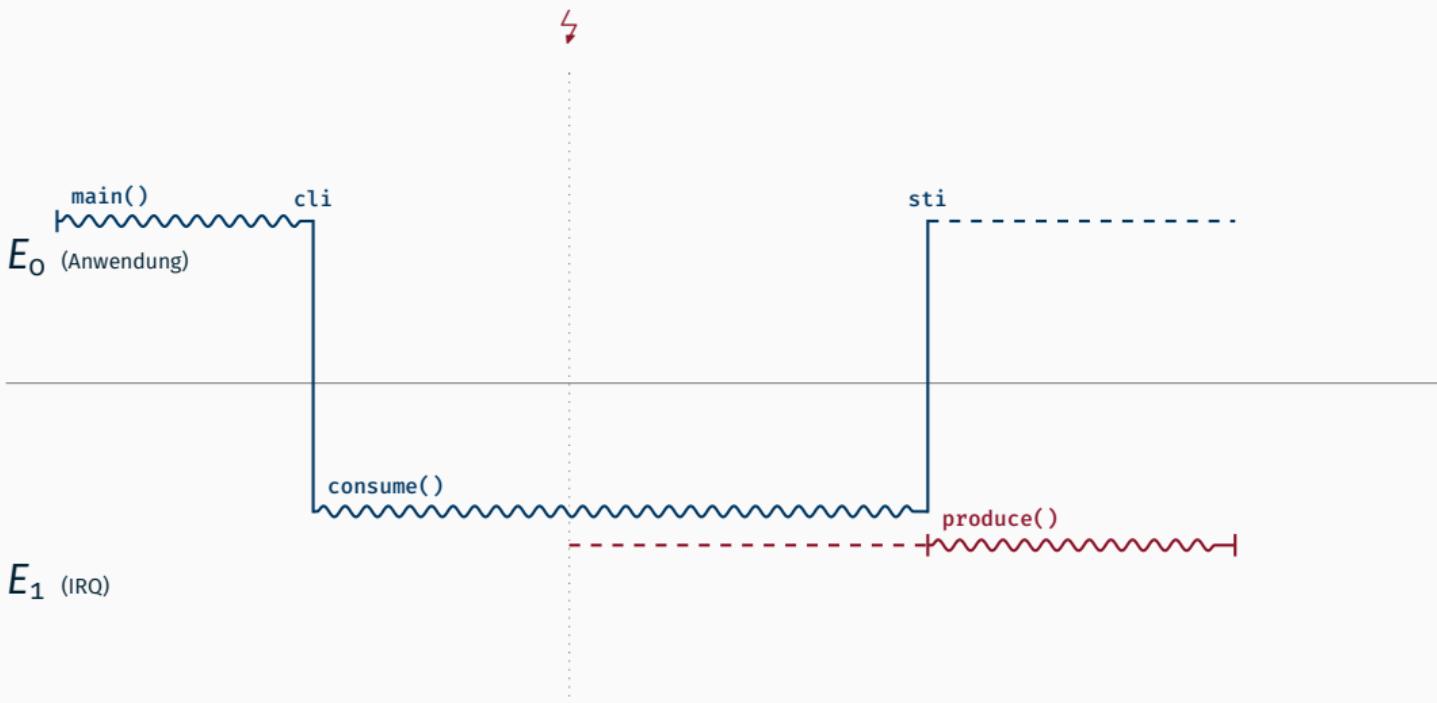
# Harte Synchronisation



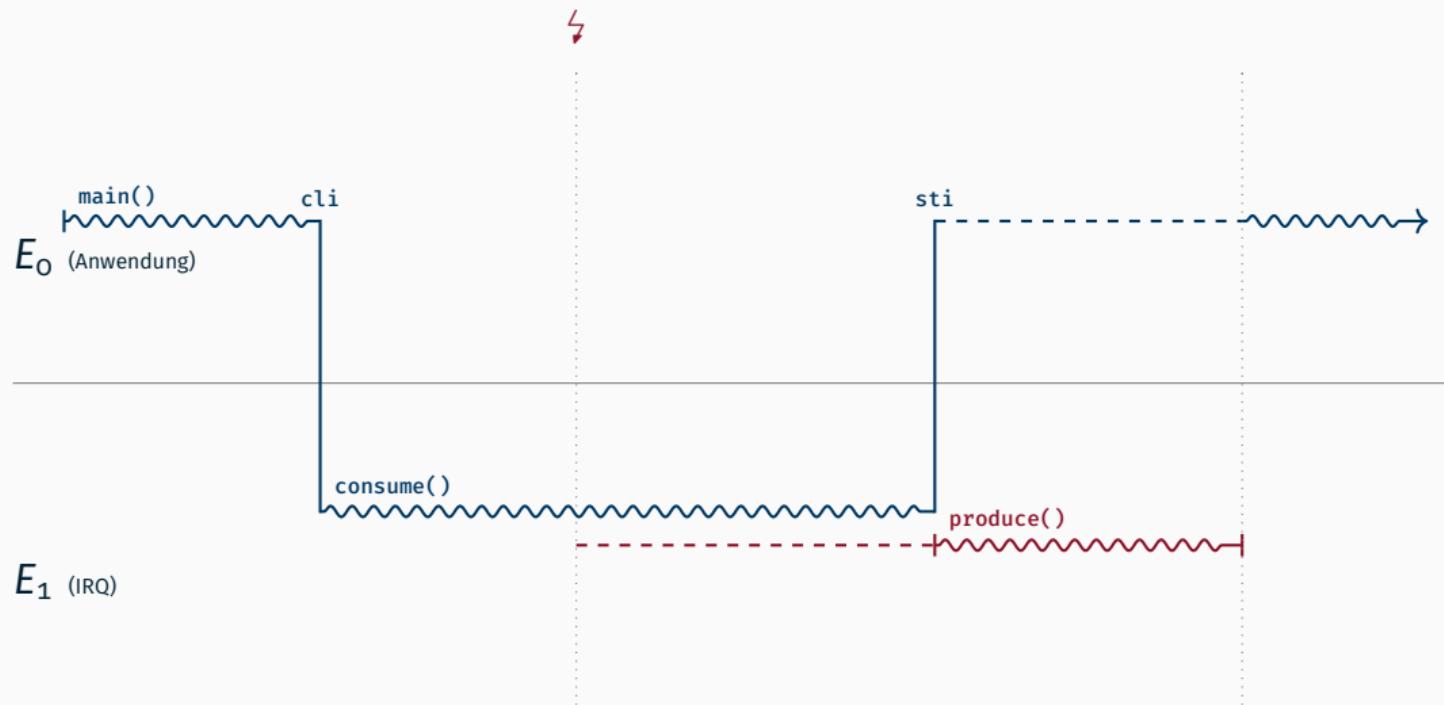
# Harte Synchronisation



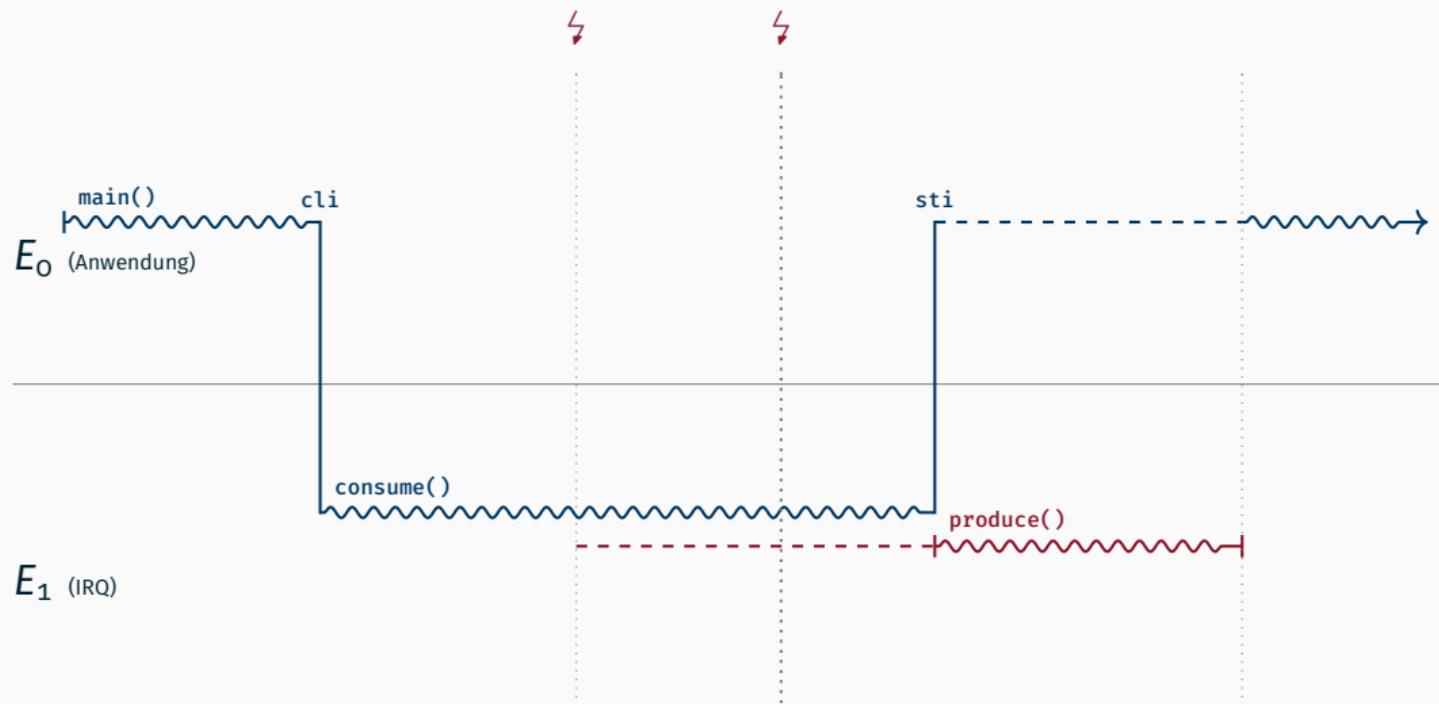
# Harte Synchronisation



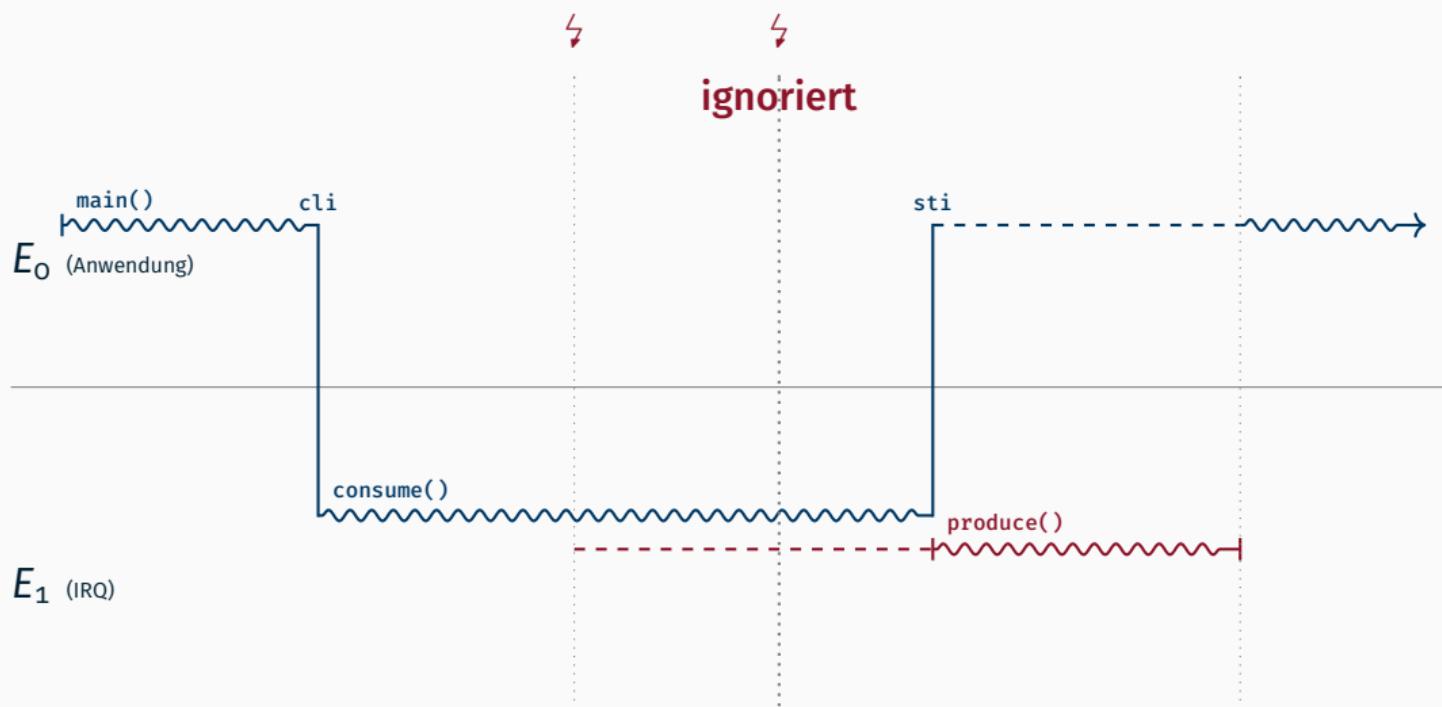
# Harte Synchronisation



# Harte Synchronisation



# Harte Synchronisation



# Harte Synchronisation

## Pessimistischer Ansatz

- + einfach
- + funktioniert immer
- Verzögerung von IRQs
  - hohe Latenz, ggf. Verlust von Interrupts
- blockiert pauschal alle IRQs

# Prolog/Epilog-Modell

---

# Idee

## Aufspalten der IRQ-Behandlung in

**Prolog** erledigt das Nötigste auf  $E_1$

**Epilog** läuft auf neuer Ebene  $\frac{1}{2}$  und übernimmt Synchronisation  
somit Ebene 1 / IRQs wieder frei

# Idee

## Aufspalten der IRQ-Behandlung in

**Prolog** erledigt das Nötigste auf  $E_1$

**Epilog** läuft auf neuer Ebene  $\frac{1}{2}$  und übernimmt Synchronisation  
somit Ebene 1 / IRQs wieder frei

## Operationen

- höhere Ebene betreten: **cli**
- höhere Ebene verlassen: **sti**
- niedrigere Ebene unterbrechen: **IRQ-Leitung**

bei **harter Synchronisation**

# Idee

## Aufspalten der IRQ-Behandlung in

**Prolog** erledigt das Nötigste auf  $E_1$

**Epilog** läuft auf neuer Ebene  $\frac{1}{2}$  und übernimmt Synchronisation  
somit Ebene 1 / IRQs wieder frei

## Operationen

- höhere Ebene betreten: **cli**, **enter**
- höhere Ebene verlassen: **sti**, **leave**
- niedrigere Ebene unterbrechen: **IRQ-Leitung**, **relay**

bei **harter Synchronisation** und **Prolog/Epilog-Modell**

# Prolog/Epilog-Modell

$E_0$  (Anwendung)  
  |  
  +---+  
    main()  
    +---+

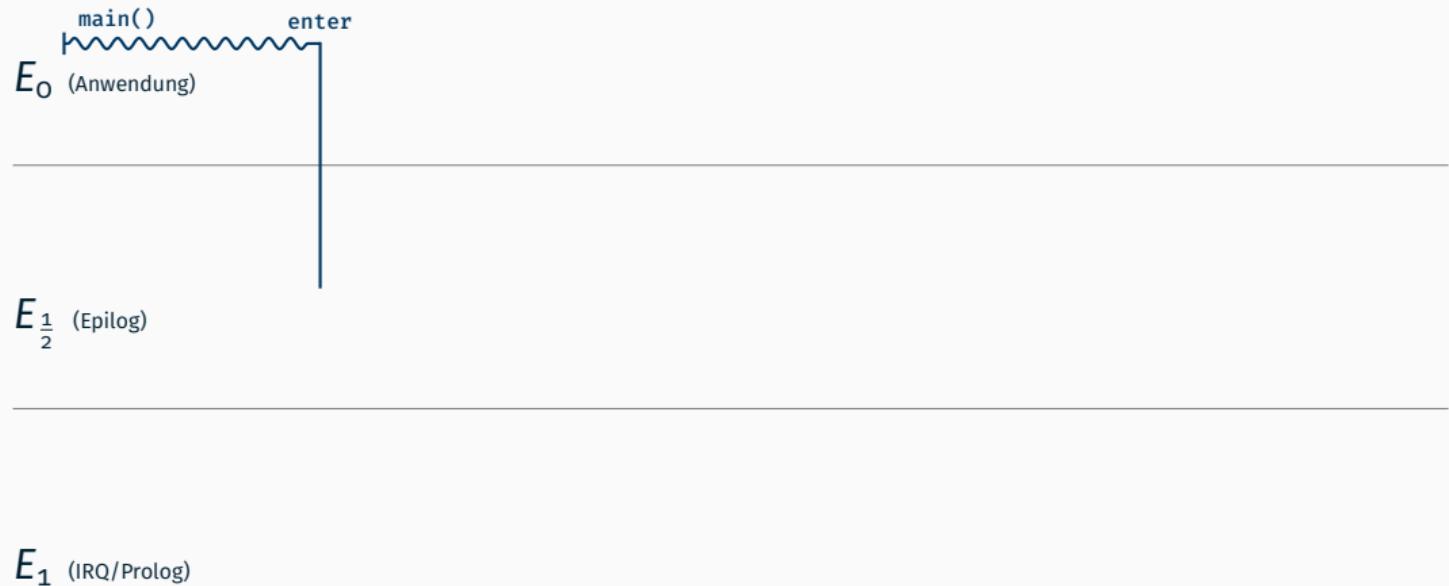
---

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

---

$E_1$  (IRQ/Prolog)

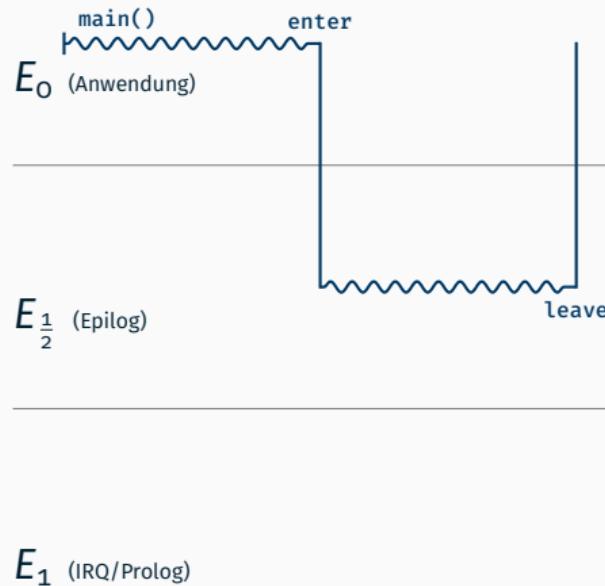
# Prolog/Epilog-Modell



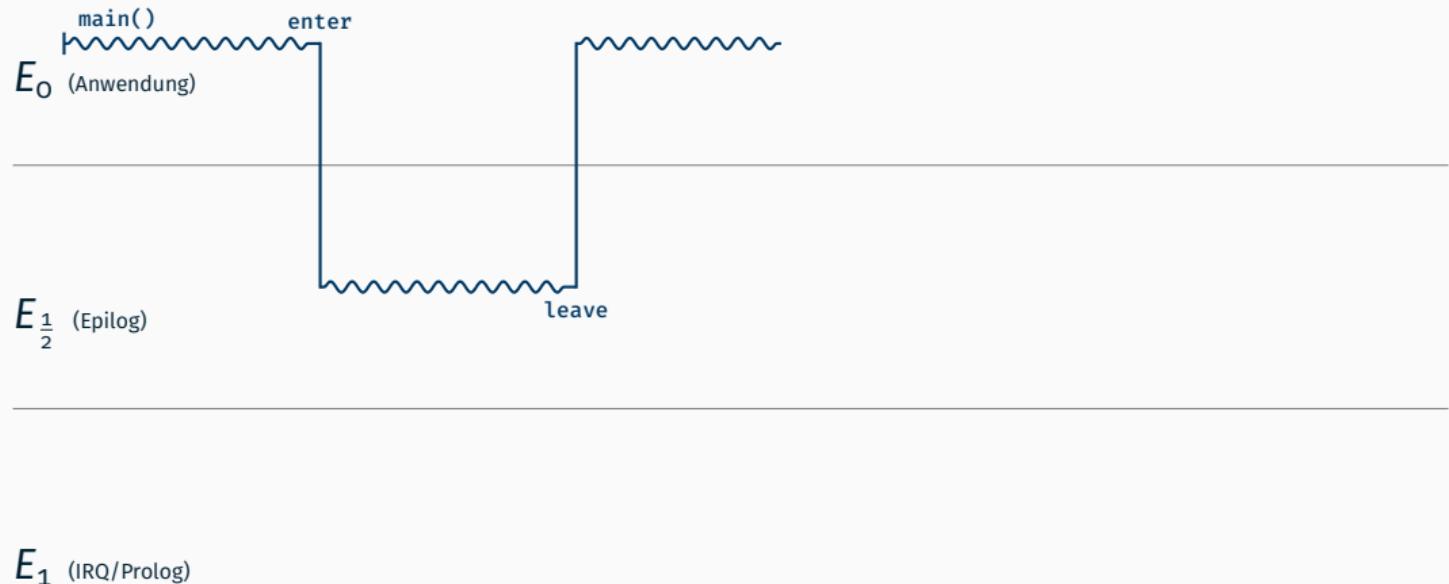
# Prolog/Epilog-Modell



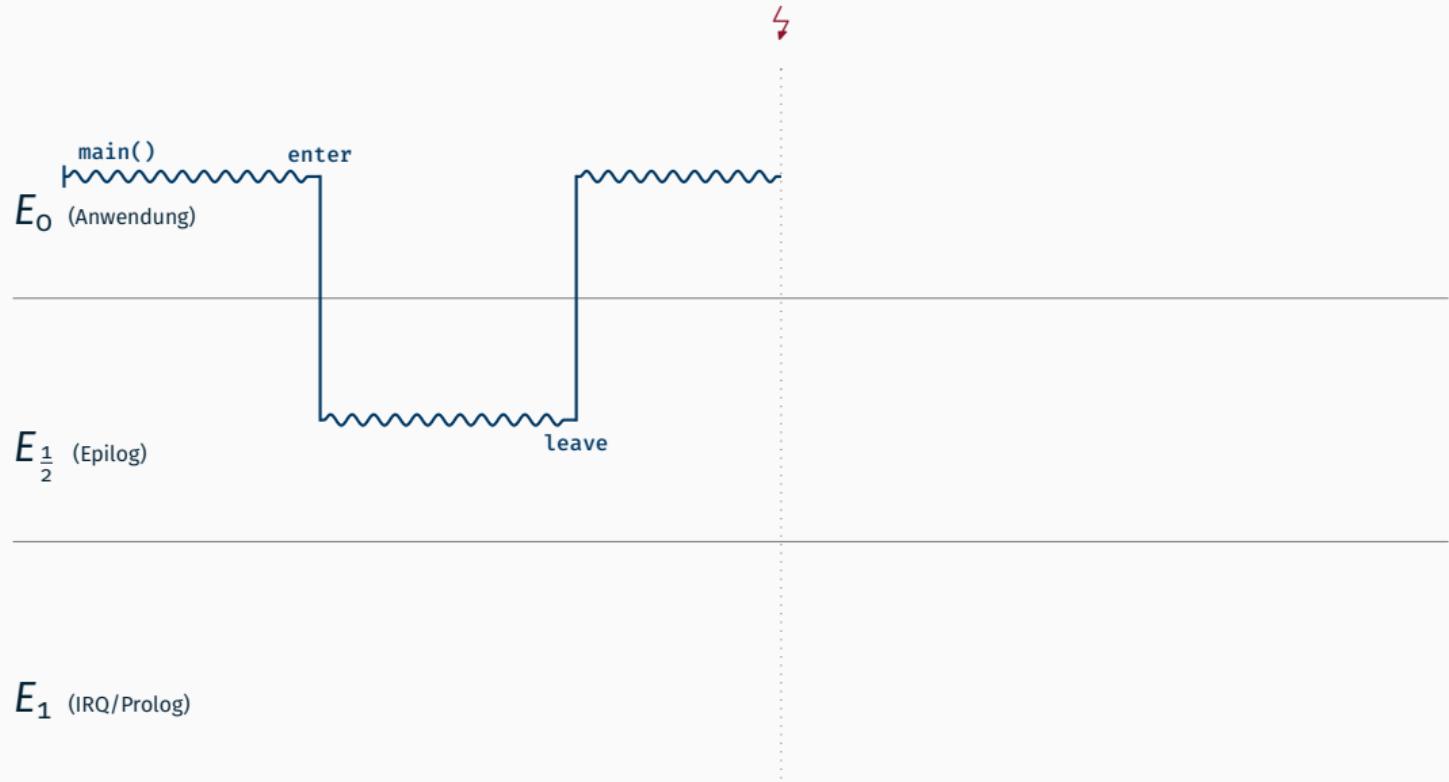
# Prolog/Epilog-Modell



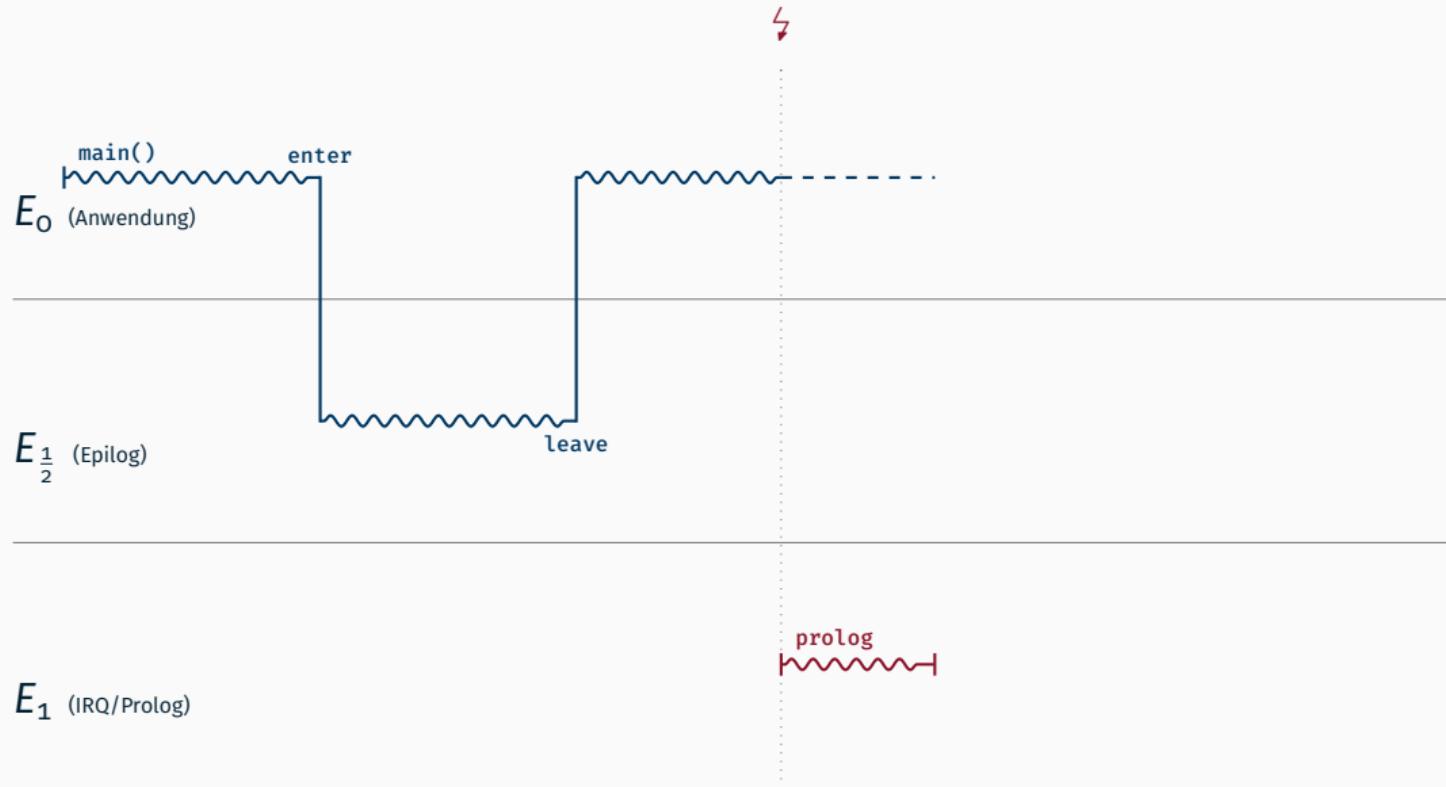
# Prolog/Epilog-Modell



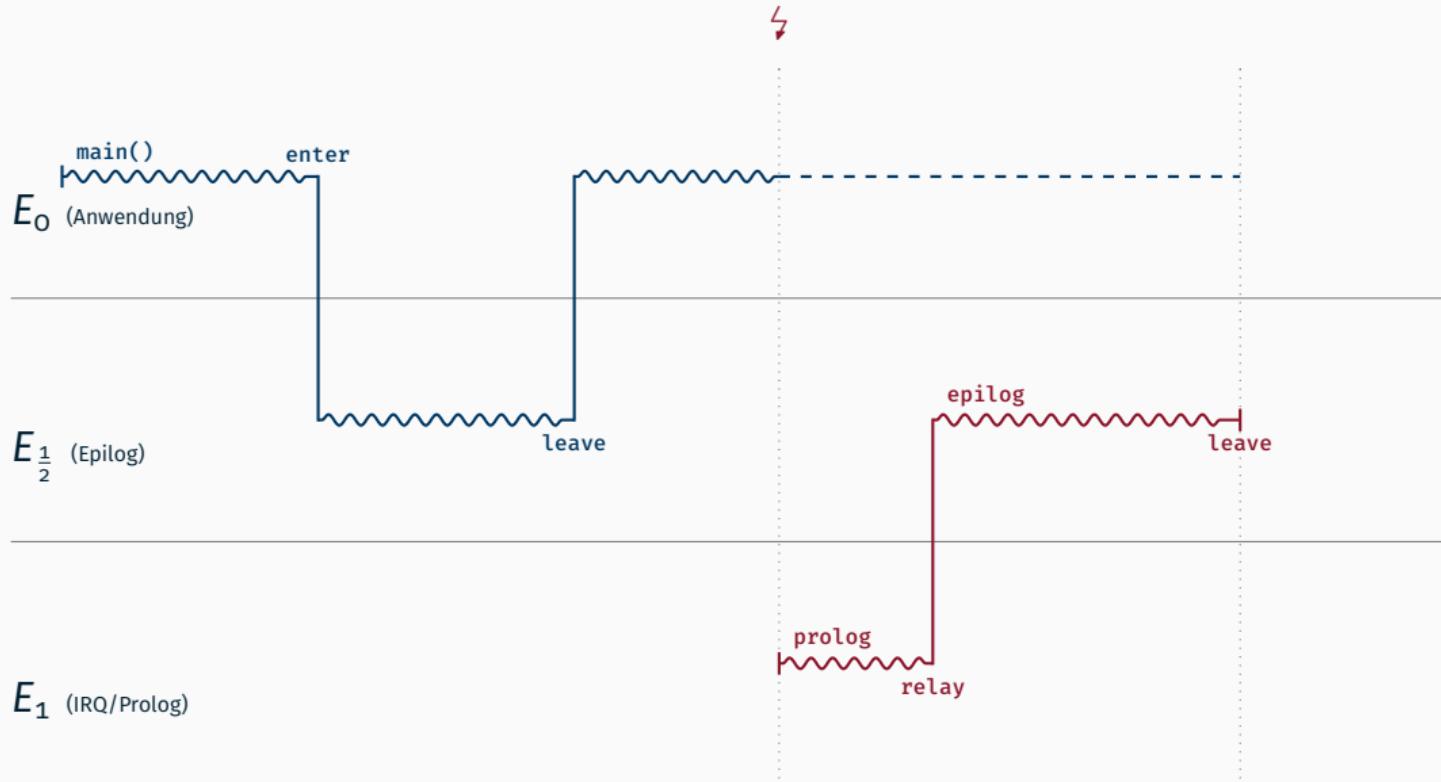
# Prolog/Epilog-Modell



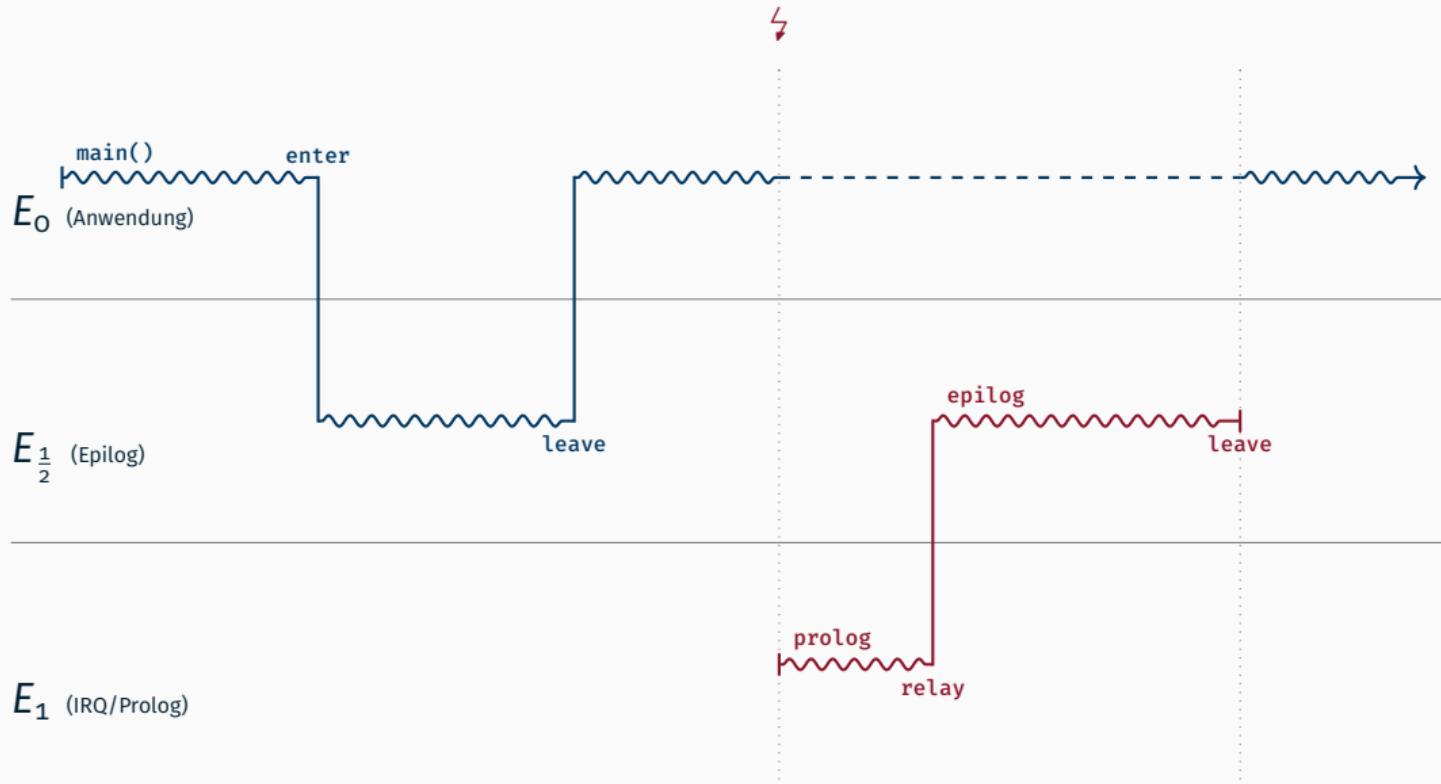
# Prolog/Epilog-Modell



# Prolog/Epilog-Modell



# Prolog/Epilog-Modell



# Prolog/Epilog-Modell

## Kombinierter Ansatz

- + einfaches Programmiermodell  
(für Anwendungsentwickler)
- + geringer Interruptverlust
- Ebene  $\frac{1}{2}$  ist zusätzlicher Overhead
- etwas mehr Arbeit für den Betriebssystemarchitekten

# Prolog/Epilog-Modell

## Kombinierter Ansatz

- + einfaches Programmiermodell  
(für Anwendungsentwickler)
  - + geringer Interruptverlust
  - Ebene  $\frac{1}{2}$  ist zusätzlicher Overhead
  - etwas mehr Arbeit für den Betriebssystemarchitekten
- **guter Kompromiss**

# Umsetzung

```
main(){
    while(1){
        enter();           // ...
        consume();
        leave();          // ...
    }
}

epilog(){
    produce();
}
```

# Umsetzung

$E_0$  (Anwendung)  
  |  
  | main()  
  |~~~~~

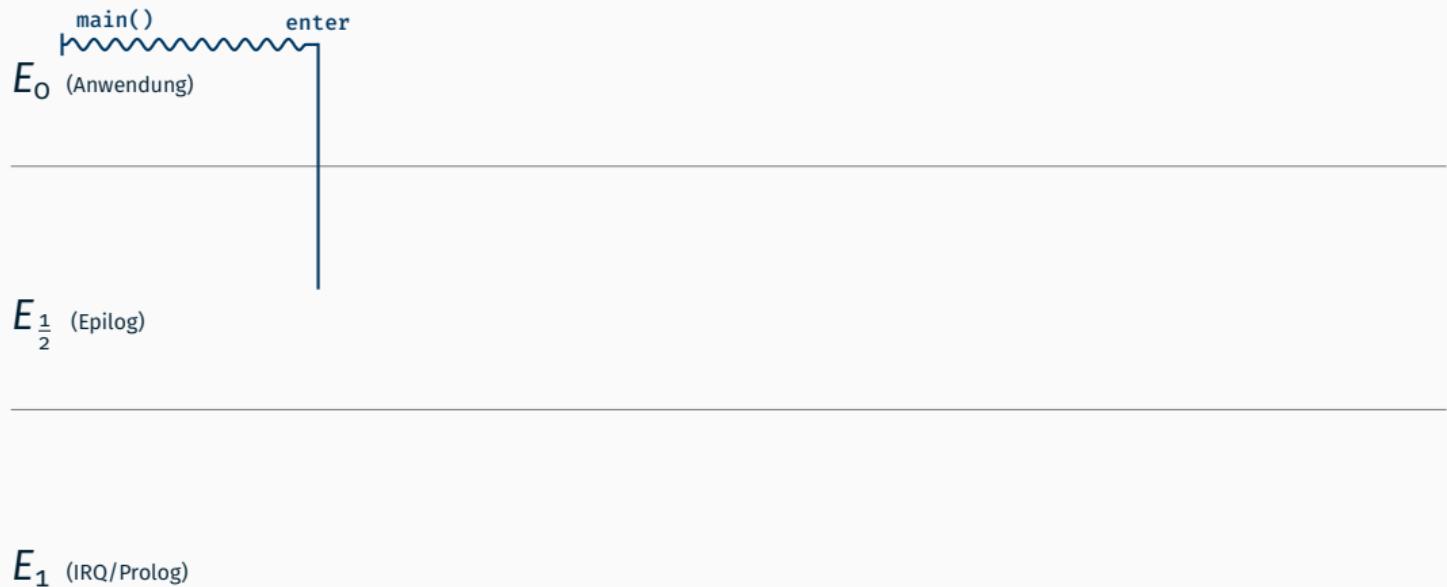
---

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

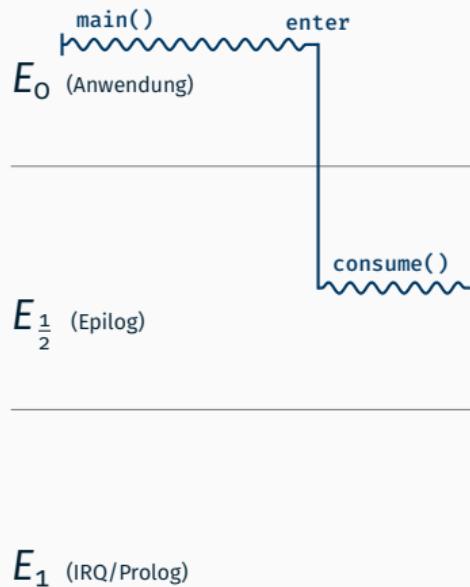
---

$E_1$  (IRQ/Prolog)

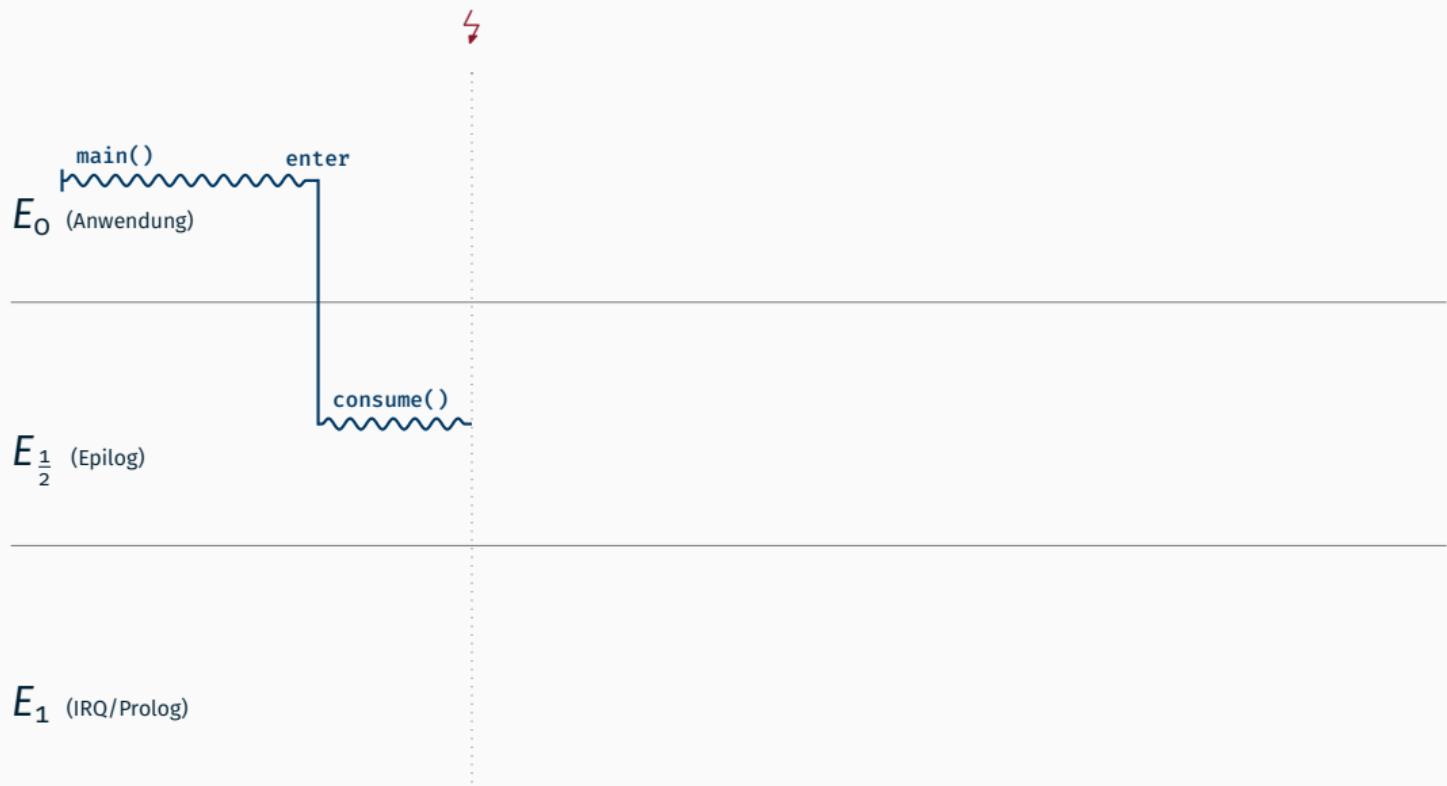
# Umsetzung



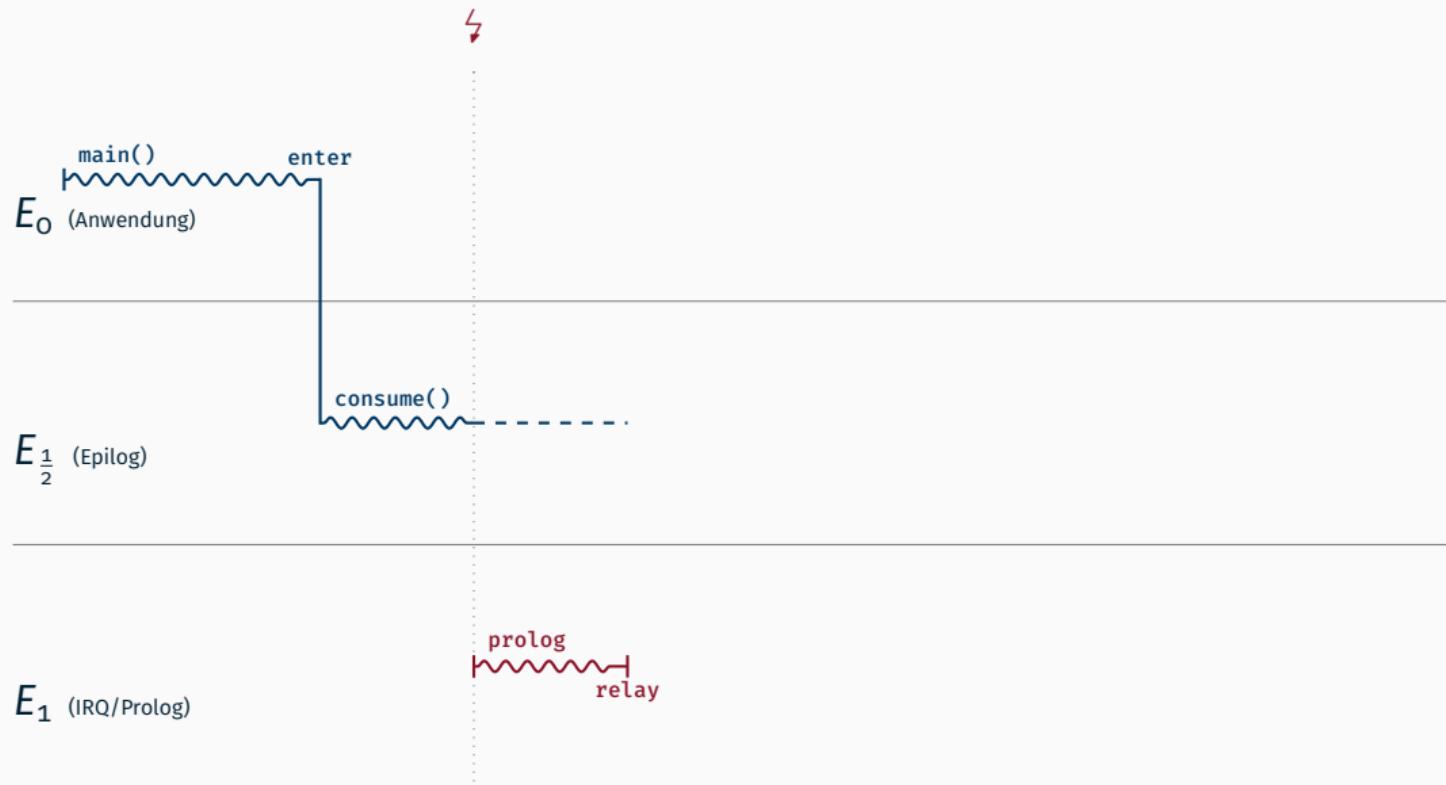
# Umsetzung



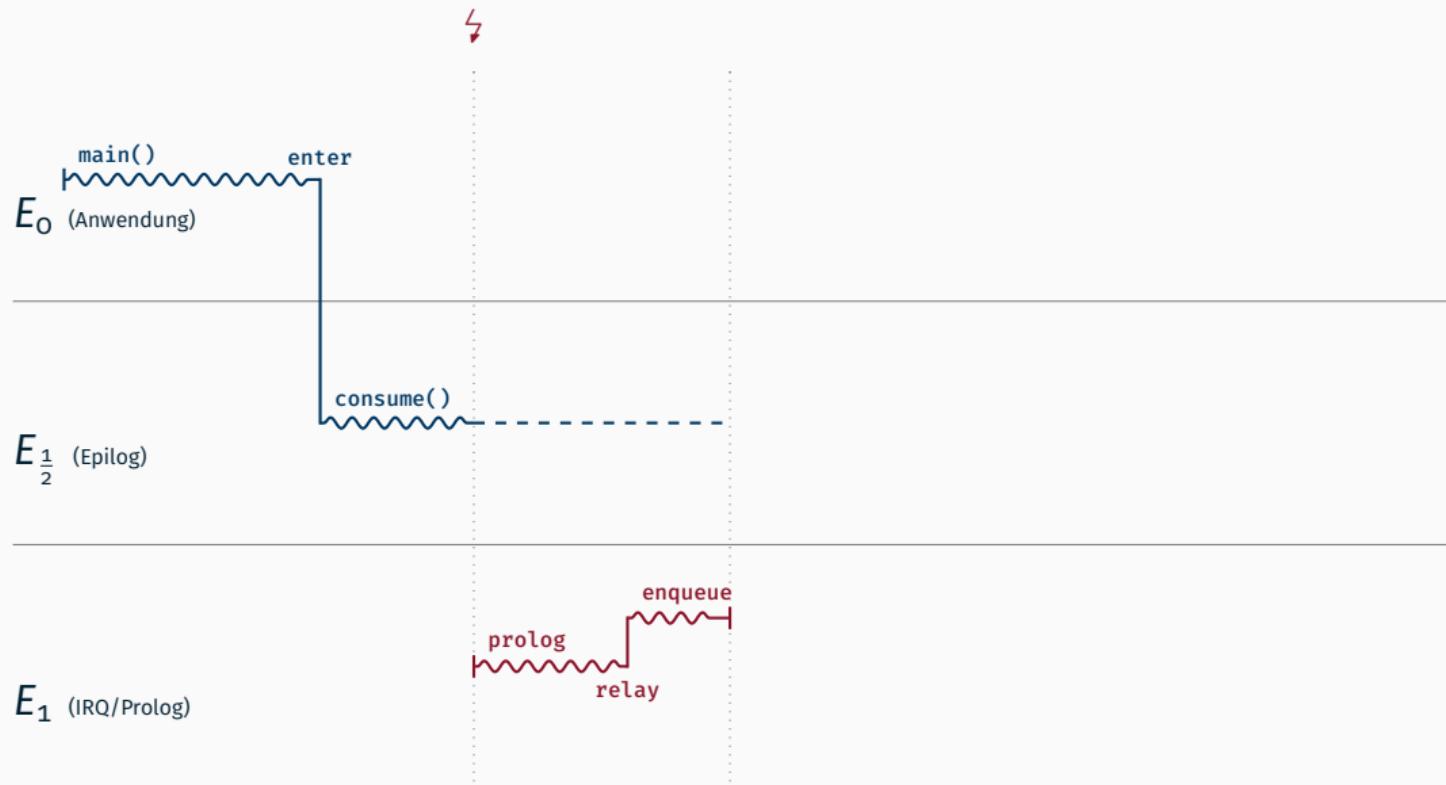
# Umsetzung



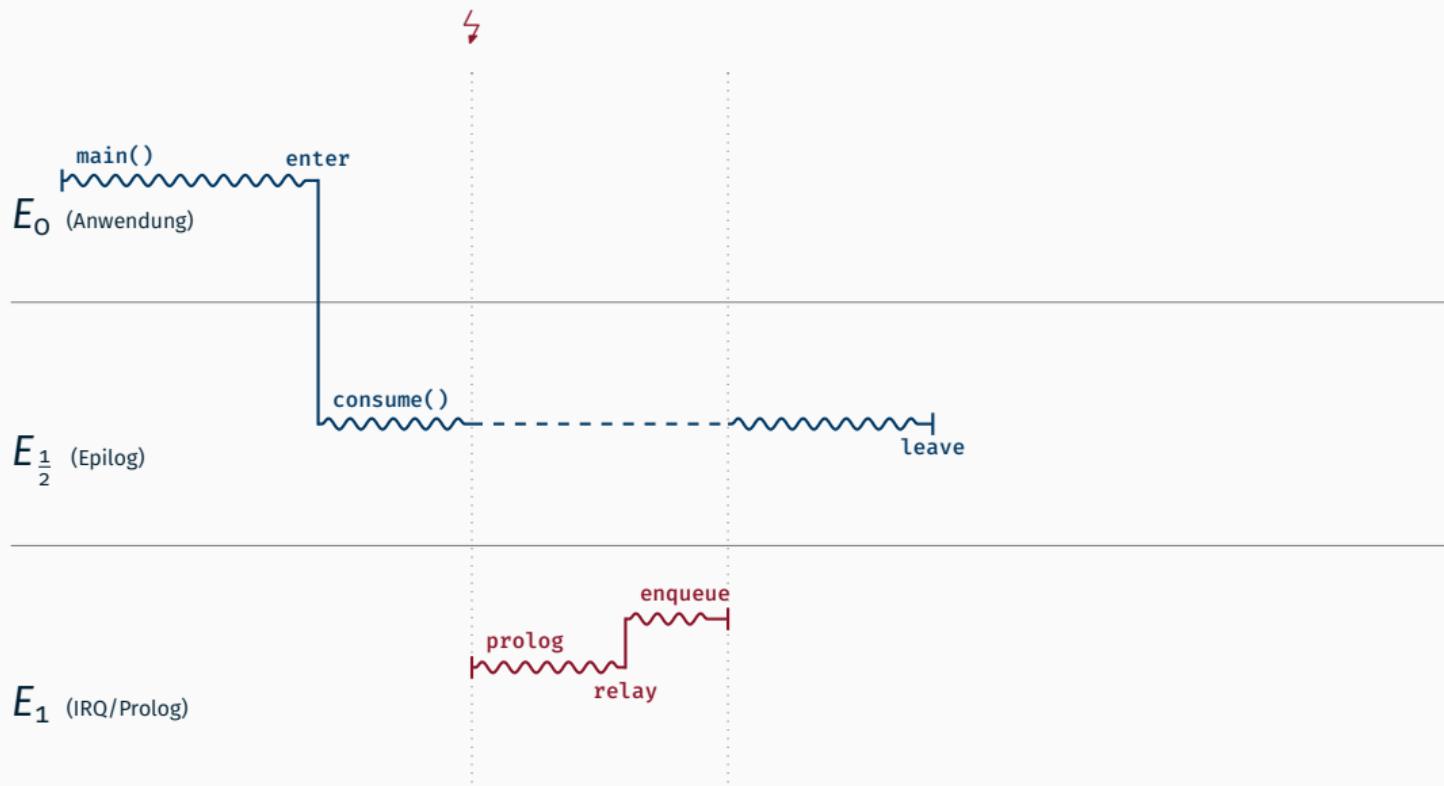
# Umsetzung



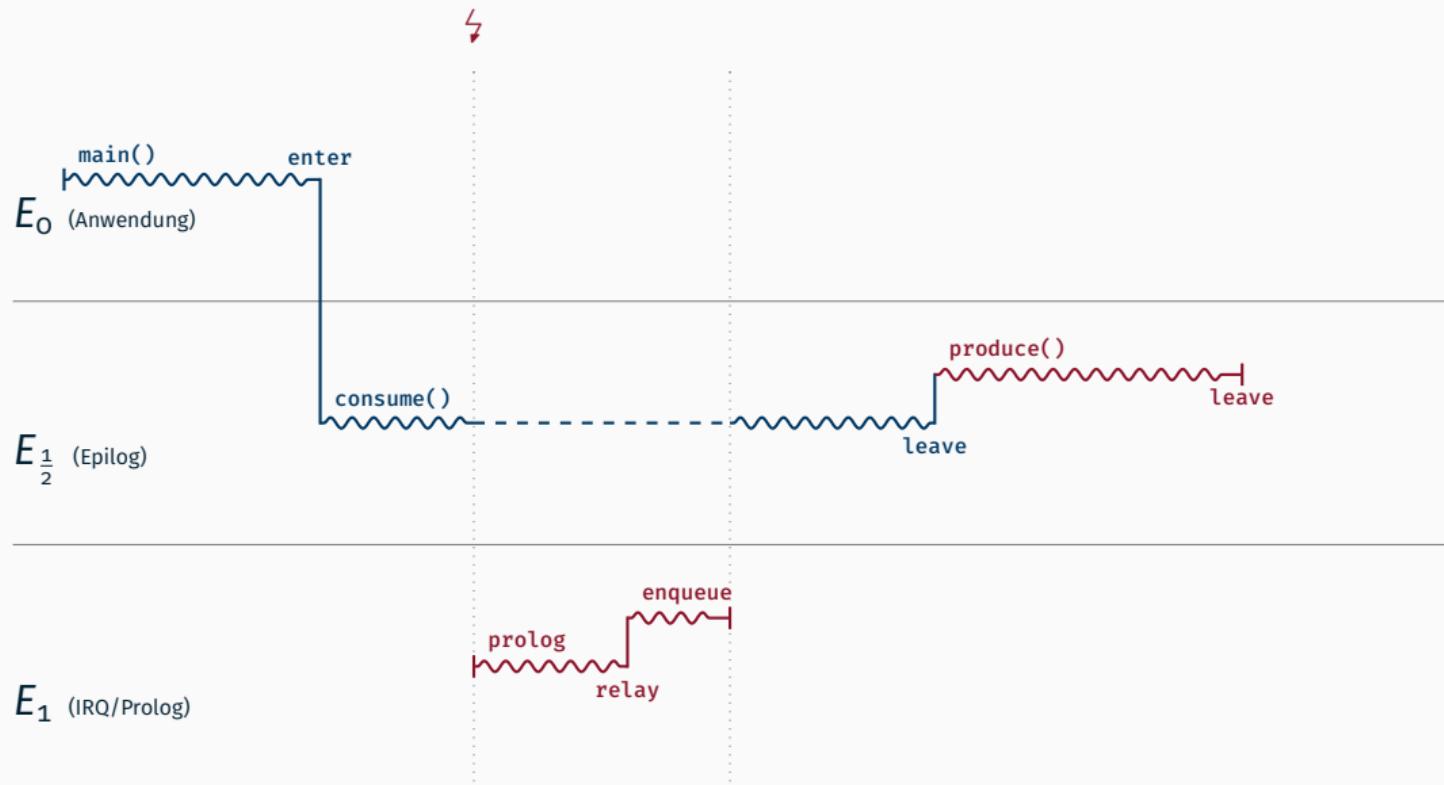
# Umsetzung



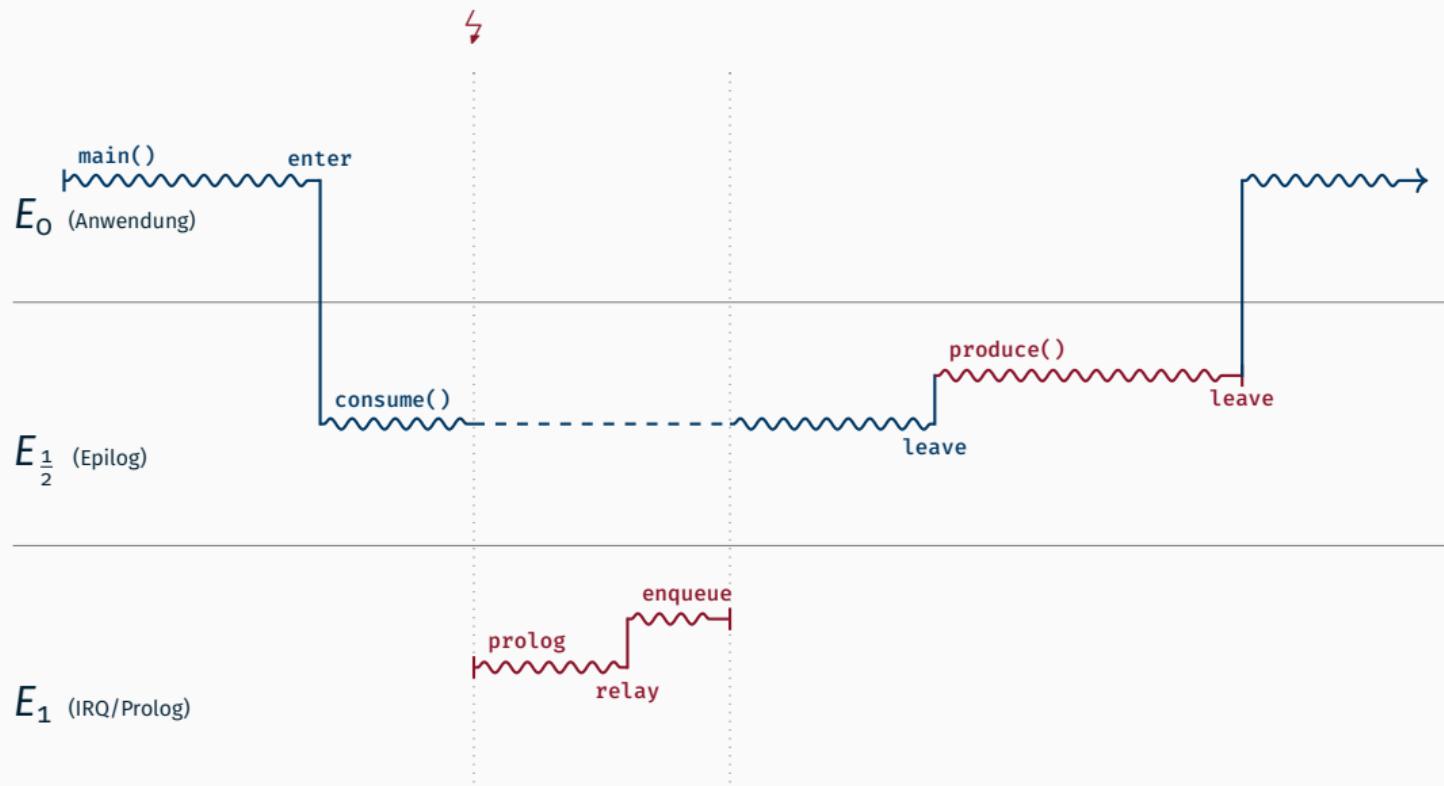
# Umsetzung



# Umsetzung



# Umsetzung



# Implementierung

---

# Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

**Was wird gebraucht?**

# Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

Was wird gebraucht?

List/Queue zum Einreihen der Epiloge

Guard mit `enter()`, `leave()` und `relay()` für Prioritätsebenen

# Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

**Was wird gebraucht?**

**List/Queue** zum Einreihen der Epiloge

**Guard** mit `enter()`, `leave()` und `relay()` für Prioritätsebenen

**Was muss angepasst werden?**

# Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

**Was wird gebraucht?**

**List/Queue** zum Einreihen der Epiloge

**Guard** mit `enter()`, `leave()` und `relay()` für Prioritätsebenen

**Was muss angepasst werden?**

- Keyboard
- Application
- Guardian

# Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

**Was wird gebraucht?**

**List/Queue** zum Einreihen der Epiloge

**Guard** mit `enter()`, `leave()` und `relay()` für Prioritätsebenen

**Was muss angepasst werden?**

- Keyboard
- Application
- Guardian
- *alles was hart synchronisiert*

# Aufgabenstellung

- Warteschlange gegeben

# Aufgabenstellung

- Warteschlange gegeben, Implementierung ist allerdings nicht unterbrechungstransparent
  - benötigt Schutz vor Unterbrechungen im **Guard** oder
  - durch Modifikation der **Queue** selbst

# Aufgabenstellung

- Warteschlange gegeben, Implementierung ist allerdings nicht unterbrechungstransparent
  - benötigt Schutz vor Unterbrechungen im **Guard** oder
  - durch Modifikation der **Queue** selbst
- **Gate** muss geändert werden  
(**trigger()** zu **prolog()** und **epilog()**)

# Aufgabenstellung

- Warteschlange gegeben, Implementierung ist allerdings nicht unterbrechungstransparent
  - benötigt Schutz vor Unterbrechungen im **Guard** oder
  - durch Modifikation der **Queue** selbst
- **Gate** muss geändert werden  
(**trigger()** zu **prolog()** und **epilog()**)
- jede **Gate**-Instanz darf nur einmal in der Epilogwarteschlange vorkommen

## **Besonderheiten in MPStuBS**

---

# Prolog/Epilog-Modell auf Mehrkernprozessoren

- Jeder Kern hat eine eigene Epilogwarteschlange  
(damit die Epiloge auf dem selben Kern wie deren zugehörige Prologe ausgeführt werden)

# Prolog/Epilog-Modell auf Mehrkernprozessoren

- Jeder Kern hat eine eigene Epilogwarteschlange  
(damit die Epiloge auf dem selben Kern wie deren zugehörige Prologe ausgeführt werden)
- jede **Gate**-Instanz darf maximal in einer Epilogwarteschlangen vorkommen

# Prolog/Epilog-Modell auf Mehrkernprozessoren

- Jeder Kern hat eine eigene Epilogwarteschlange  
(damit die Epiloge auf dem selben Kern wie deren zugehörige Prologe ausgeführt werden)
- jede **Gate**-Instanz darf maximal in einer Epilogwarteschlangen vorkommen
- Zu jedem Zeitpunkt darf maximal ein Kern Epiloge ausführen

# Prolog/Epilog-Modell auf Mehrkernprozessoren

- Jeder Kern hat eine eigene Epilogwarteschlange  
(damit die Epiloge auf dem selben Kern wie deren zugehörige Prologe ausgeführt werden)
- jede **Gate**-Instanz darf maximal in einer Epilogwarteschlangen vorkommen
- Zu jedem Zeitpunkt darf maximal ein Kern Epiloge ausführen  
→ Verwendung eines **big kernel lock** (BKL)

# Prolog/Epilog-Modell auf Mehrkernprozessoren

- Jeder Kern hat eine eigene Epilogwarteschlange  
(damit die Epiloge auf dem selben Kern wie deren zugehörige Prologe ausgeführt werden)
- jede **Gate**-Instanz darf maximal in einer Epilogwarteschlangen vorkommen
- Zu jedem Zeitpunkt darf maximal ein Kern Epiloge ausführen  
→ Verwendung eines **big kernel lock** (BKL)
- Korrekte Sperrreihenfolge ist extrem wichtig!

# Beispiel für Mehrkernprozessoren

CPU 1 |~~~~~

CPU 0 |~~~~~

$E_0$

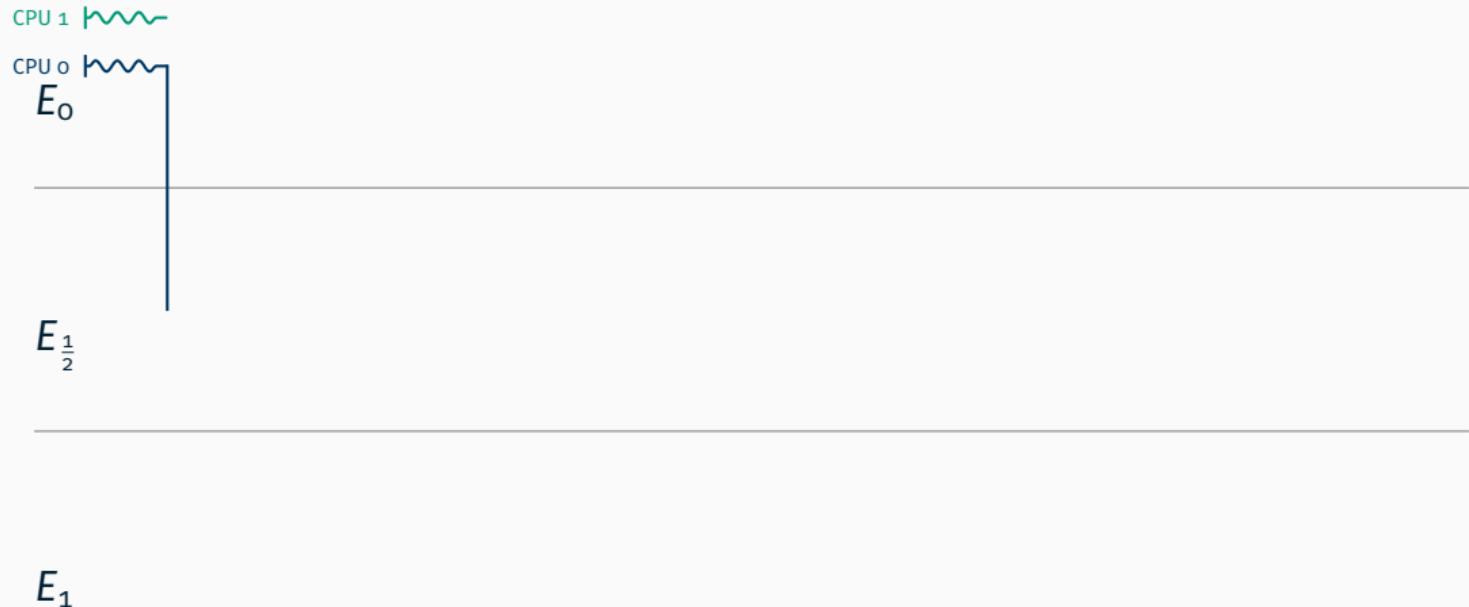
---

$E_{\frac{1}{2}}$

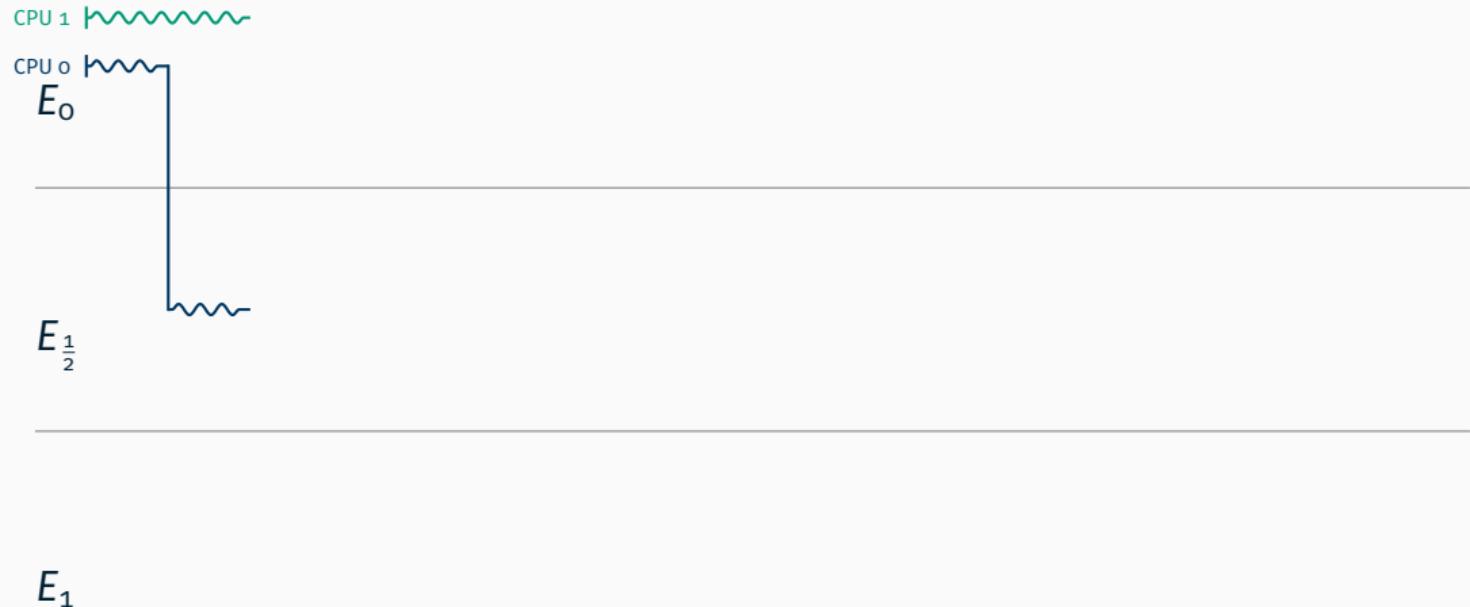
---

$E_1$

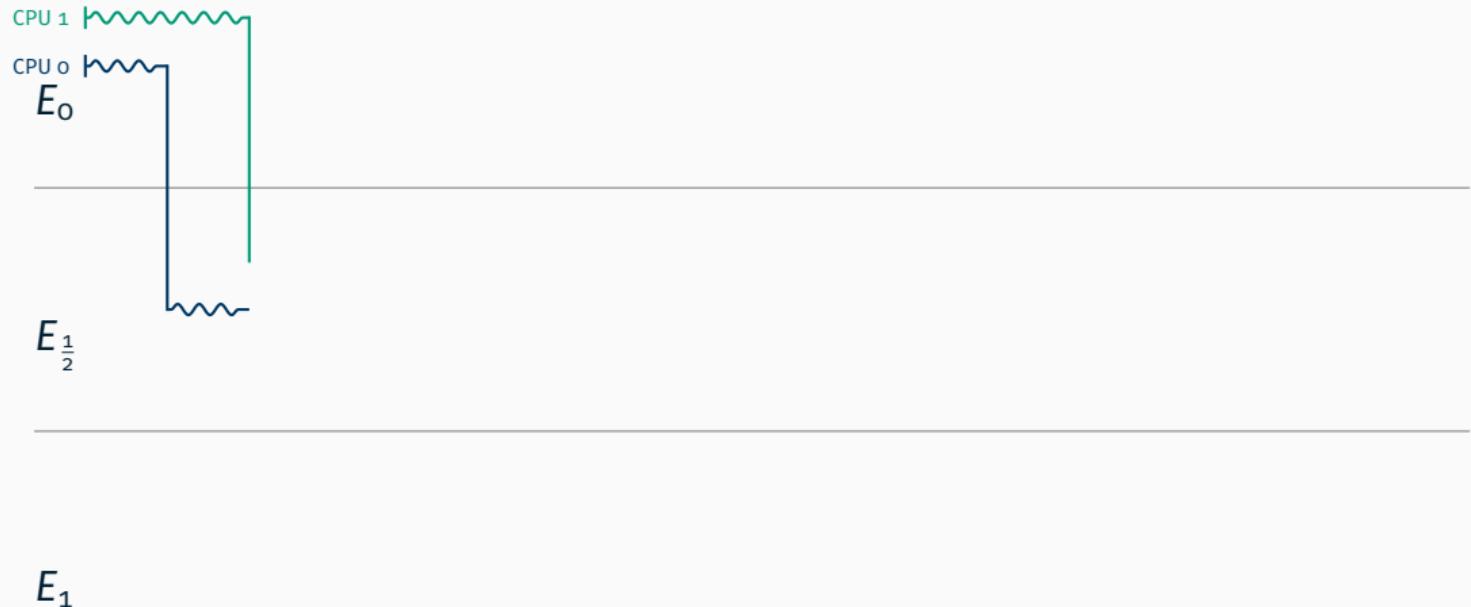
# Beispiel für Mehrkernprozessoren



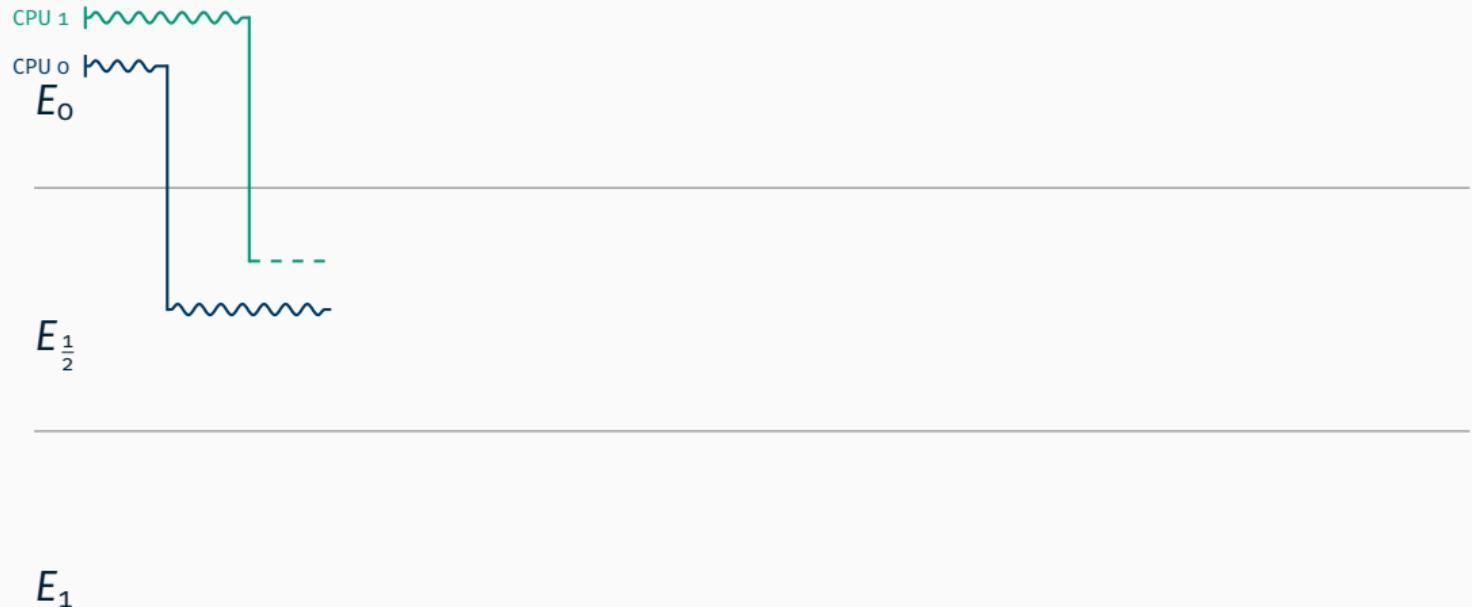
# Beispiel für Mehrkernprozessoren



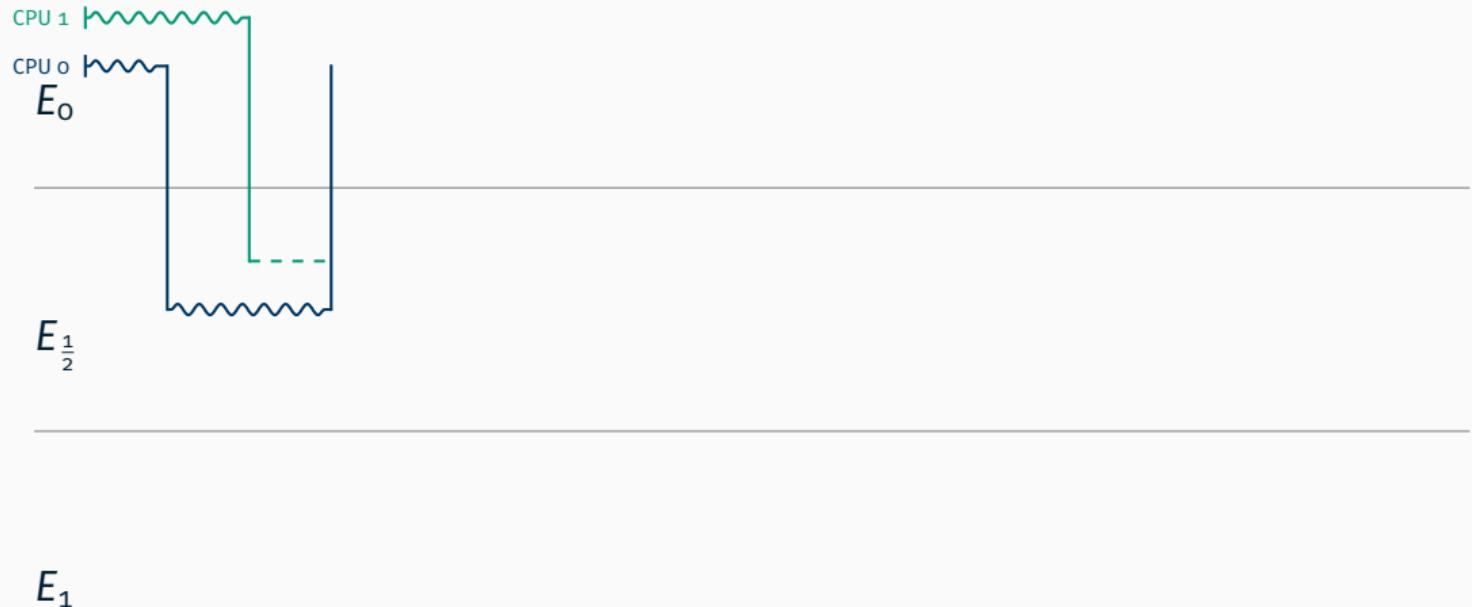
# Beispiel für Mehrkernprozessoren



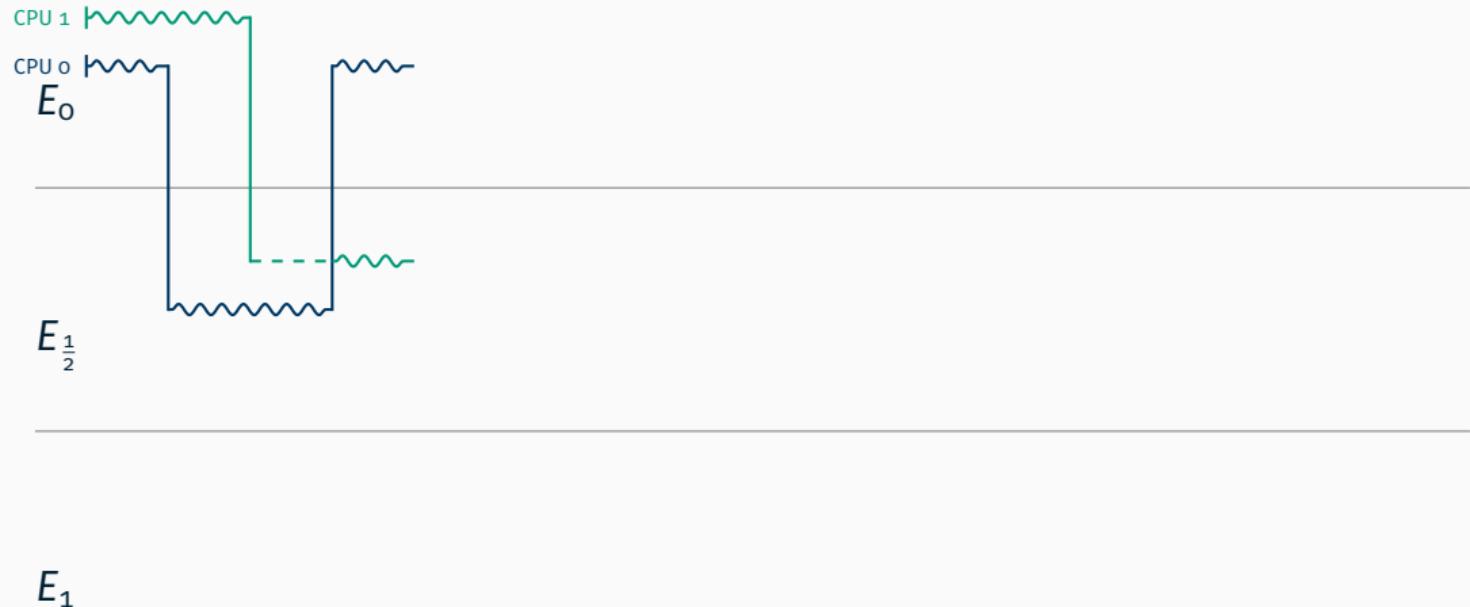
# Beispiel für Mehrkernprozessoren



# Beispiel für Mehrkernprozessoren



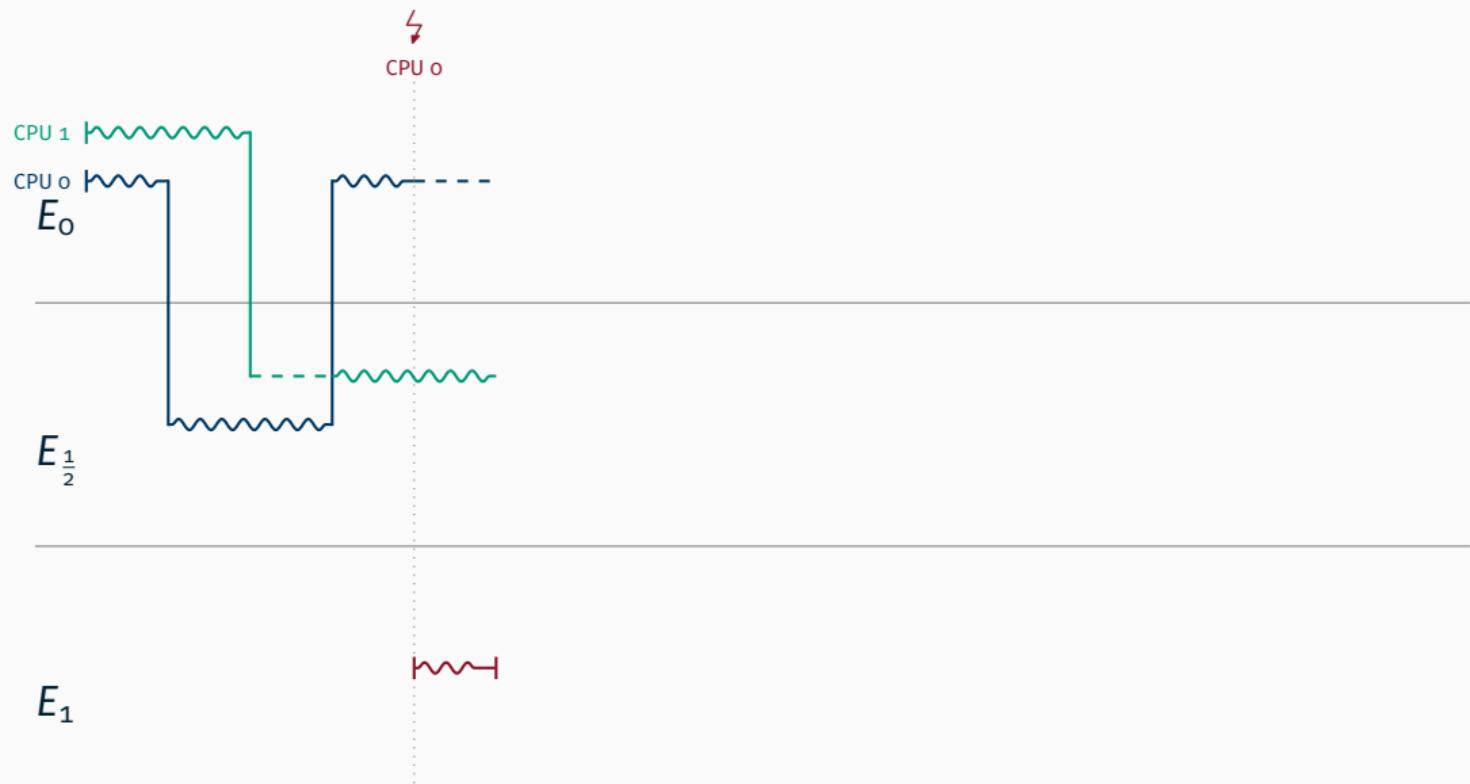
# Beispiel für Mehrkernprozessoren



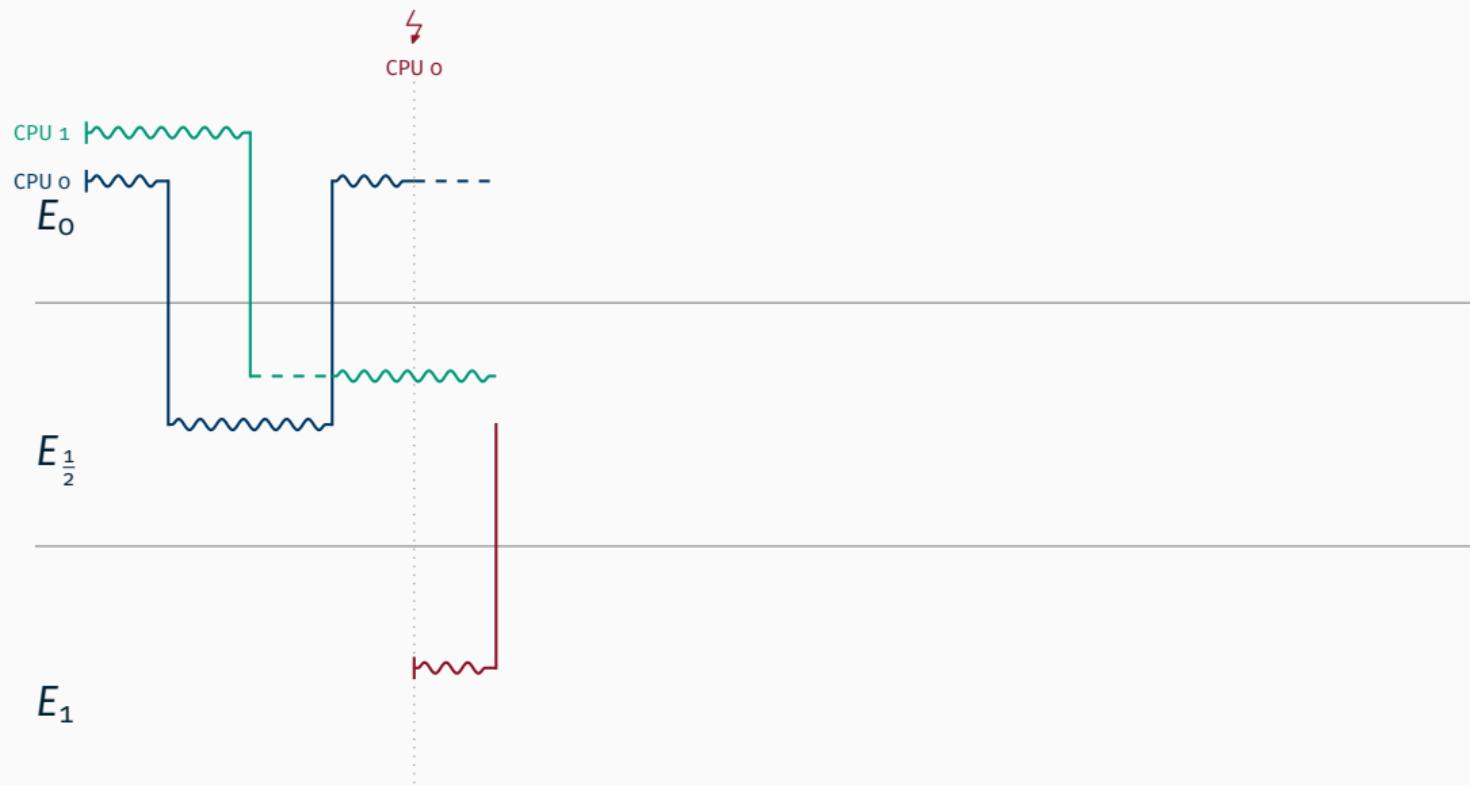
# Beispiel für Mehrkernprozessoren



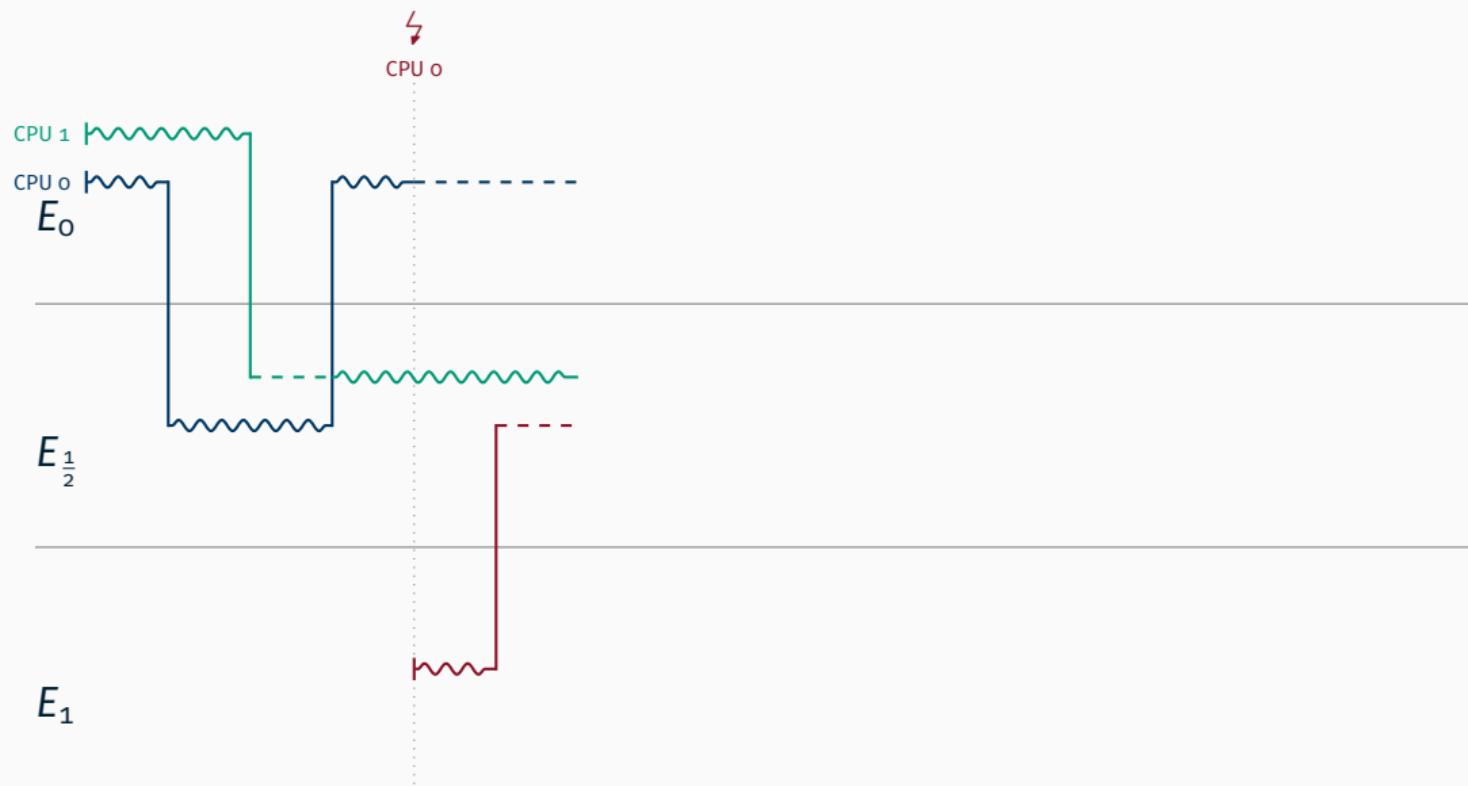
# Beispiel für Mehrkernprozessoren



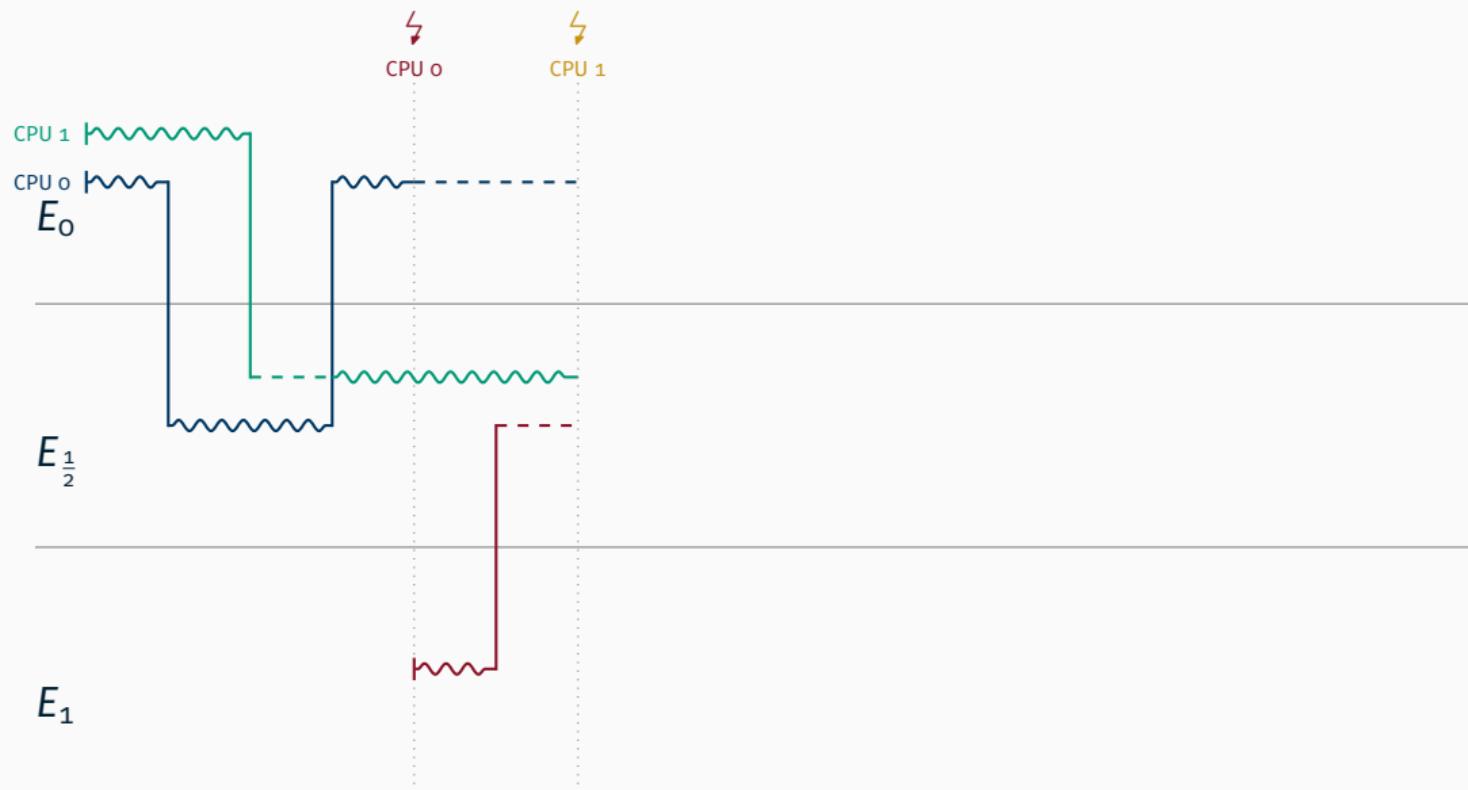
# Beispiel für Mehrkernprozessoren



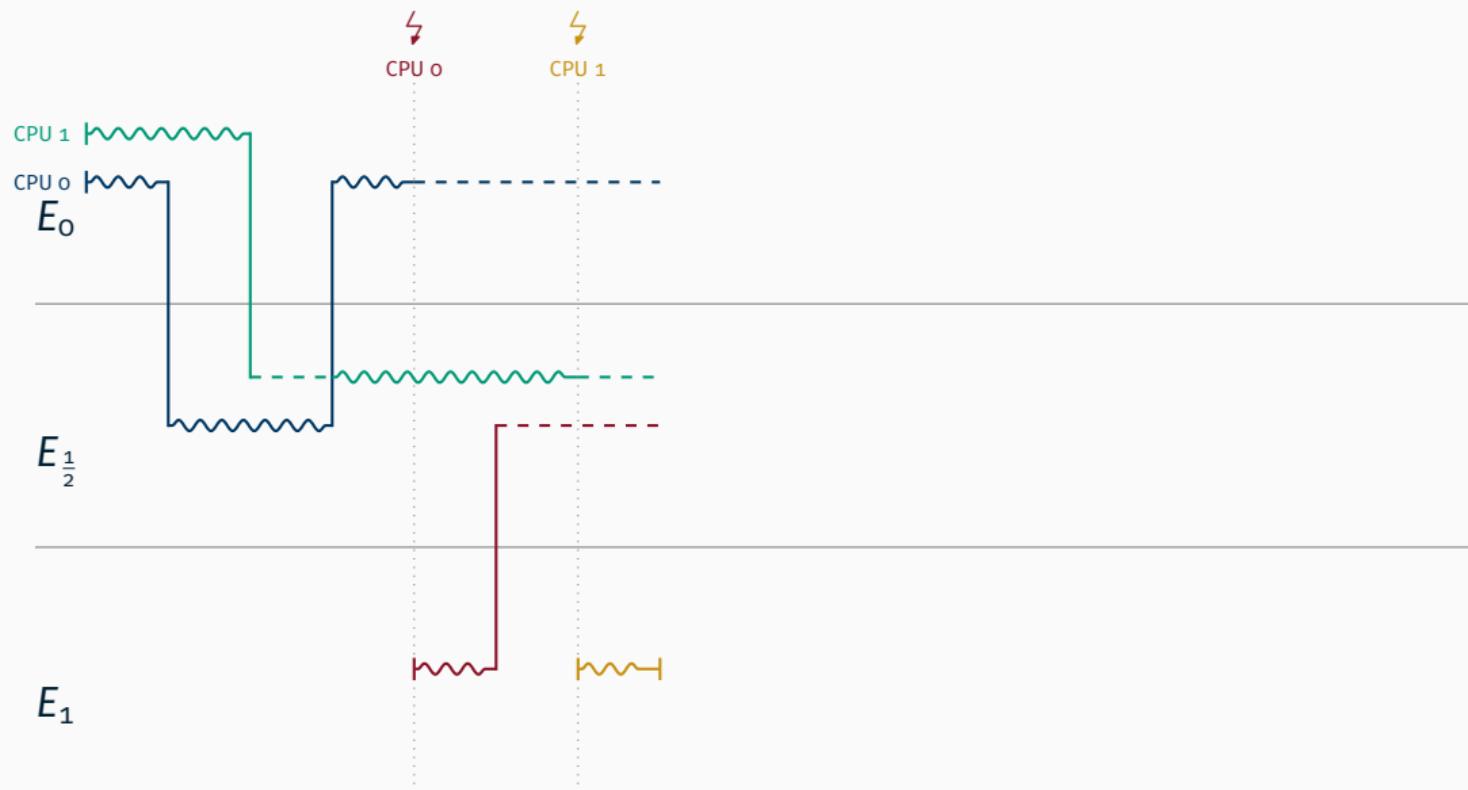
# Beispiel für Mehrkernprozessoren



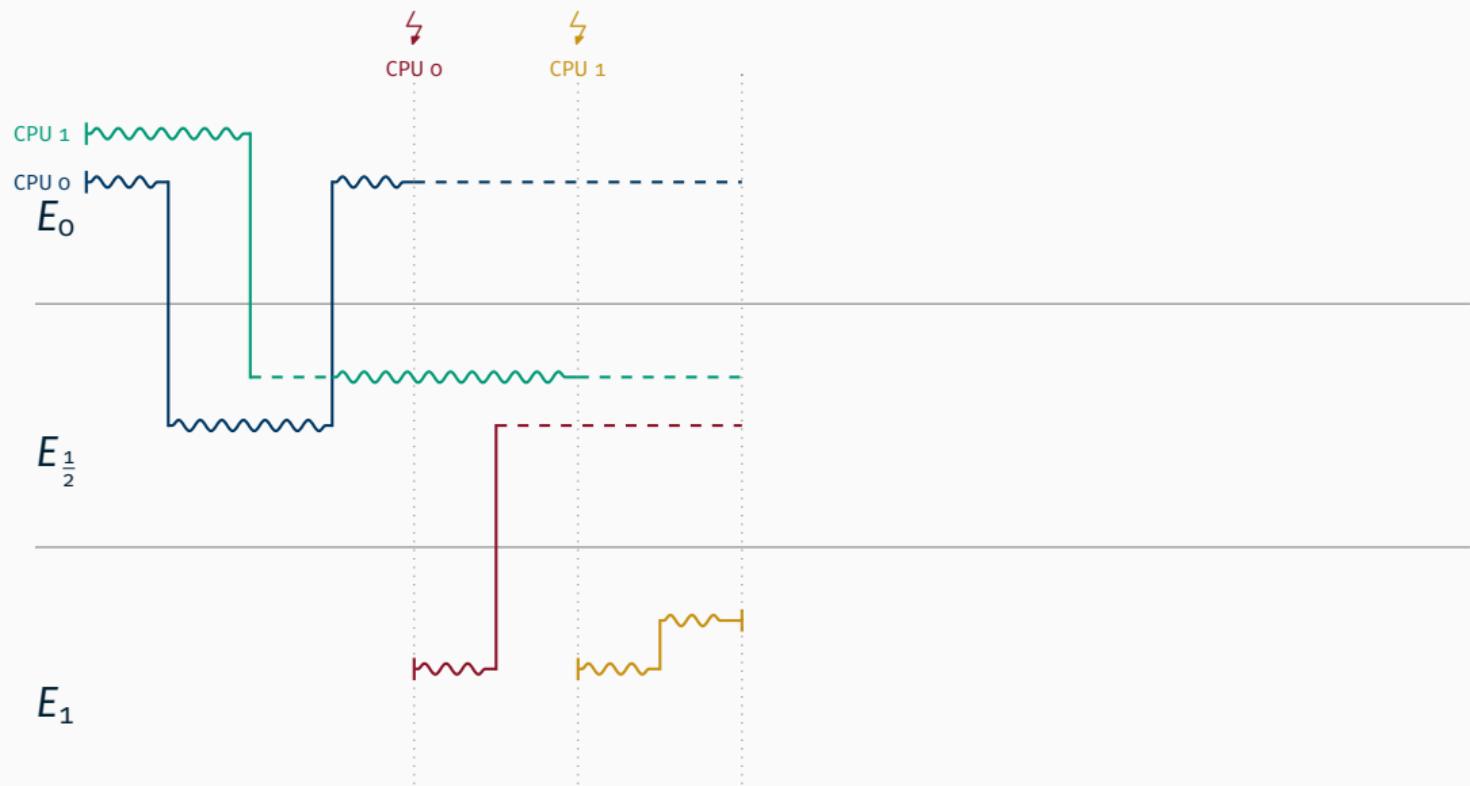
# Beispiel für Mehrkernprozessoren



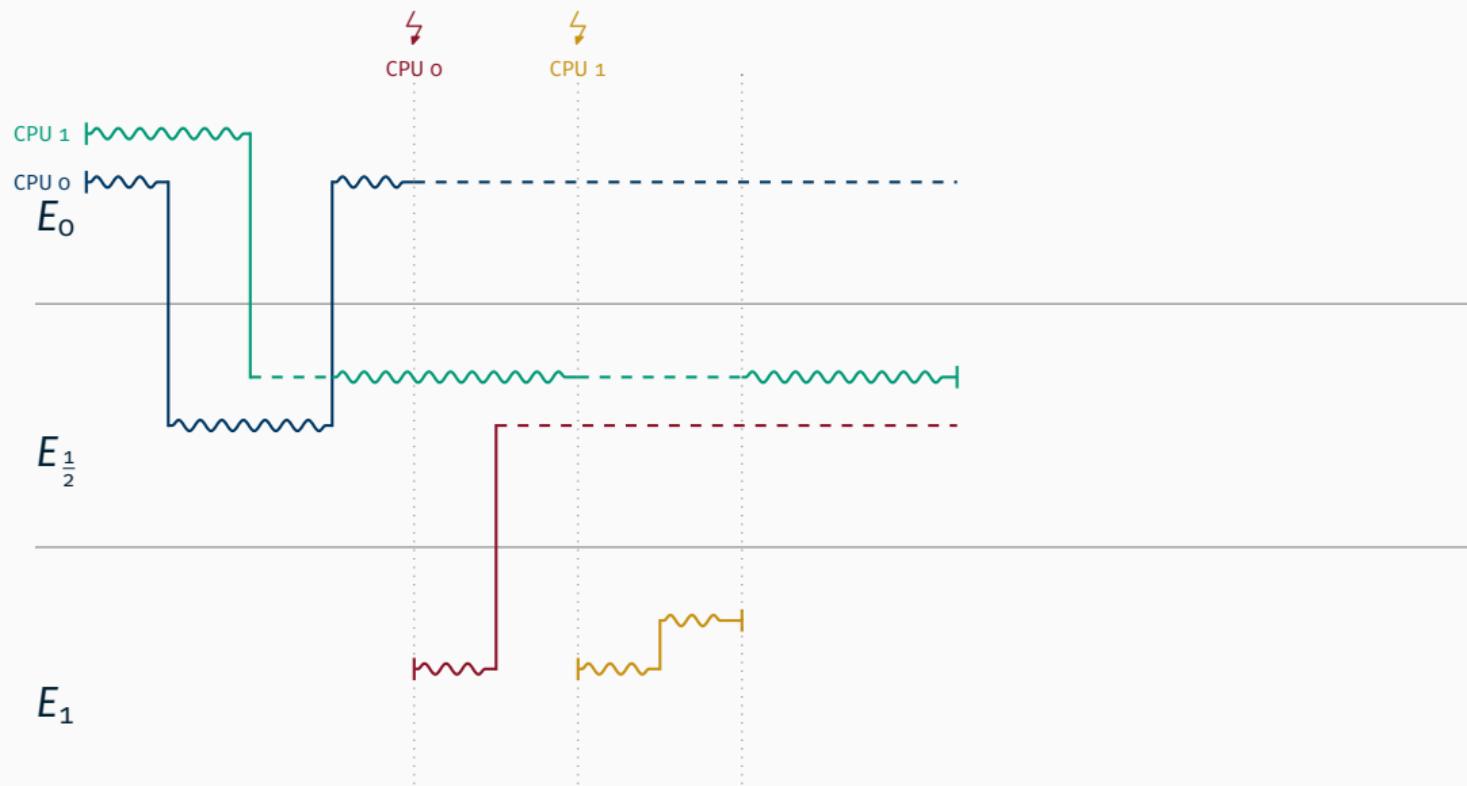
# Beispiel für Mehrkernprozessoren



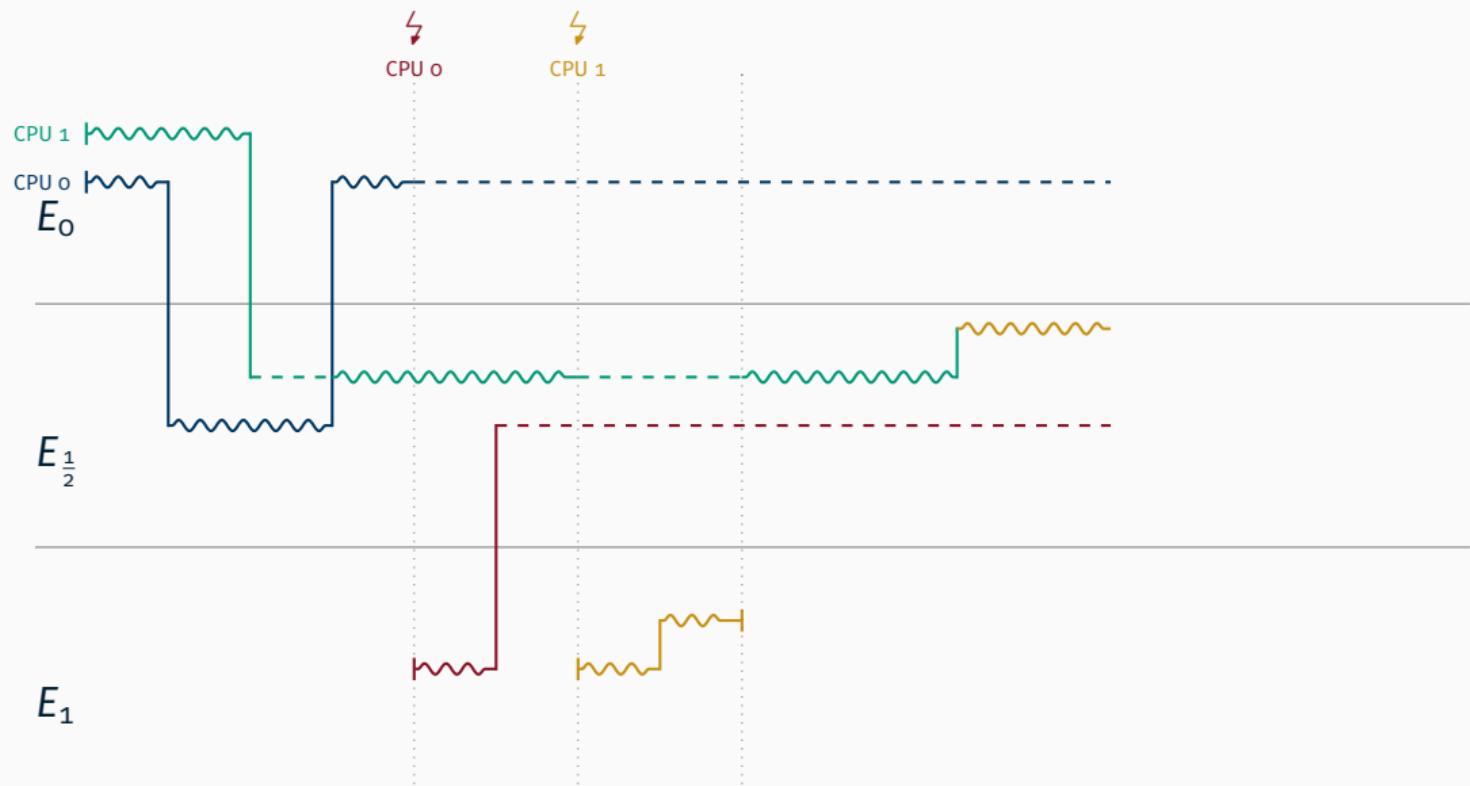
# Beispiel für Mehrkernprozessoren



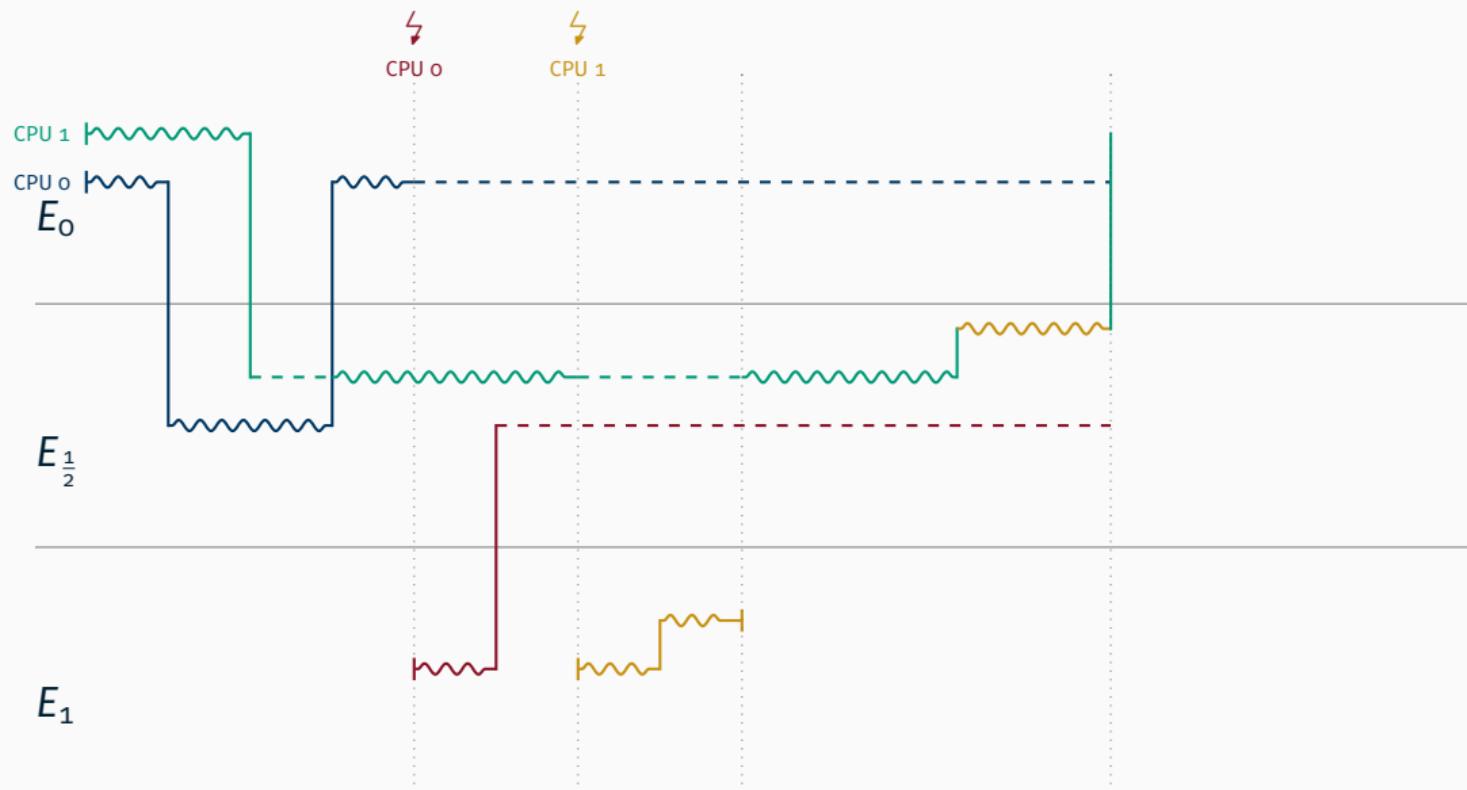
# Beispiel für Mehrkernprozessoren



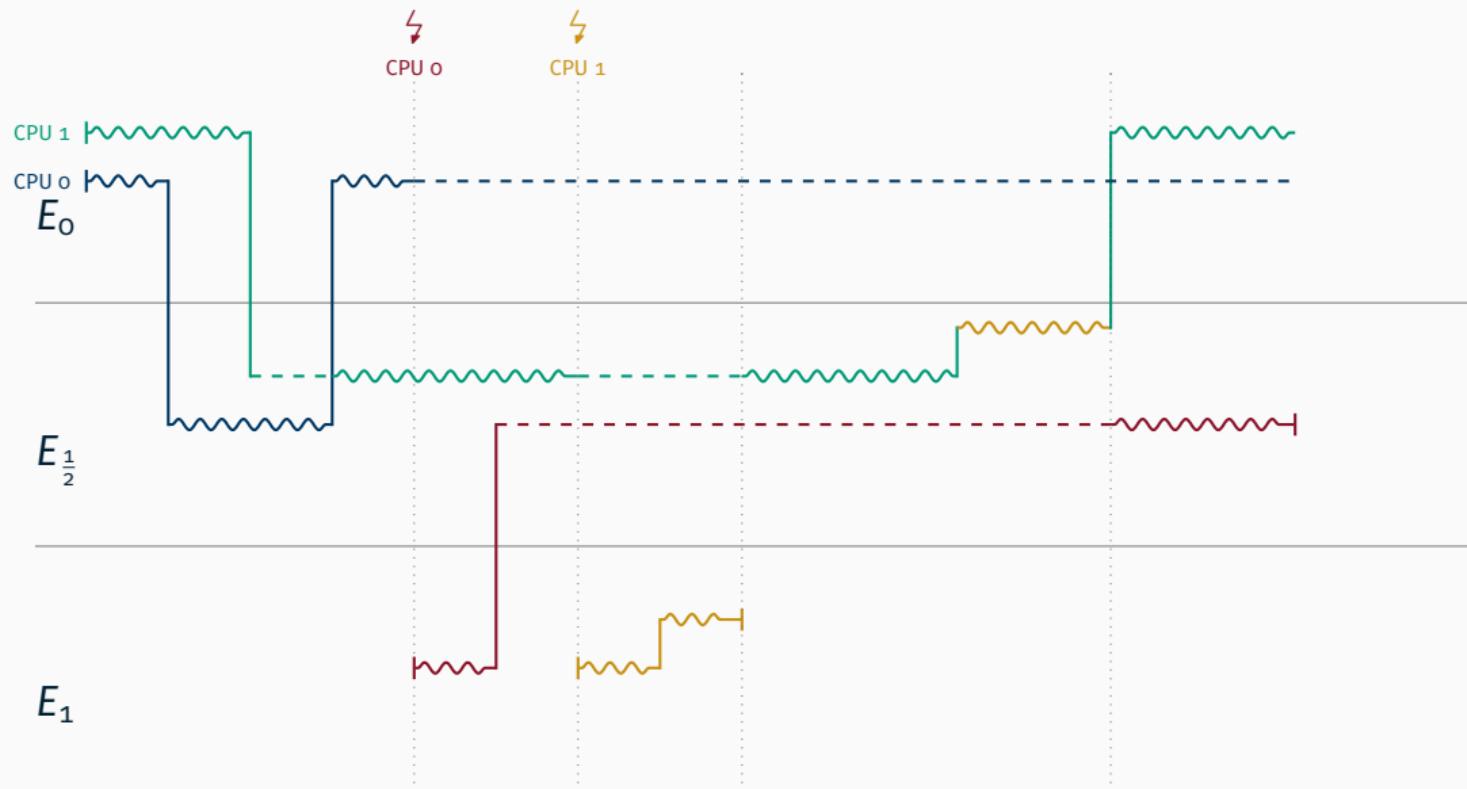
# Beispiel für Mehrkernprozessoren



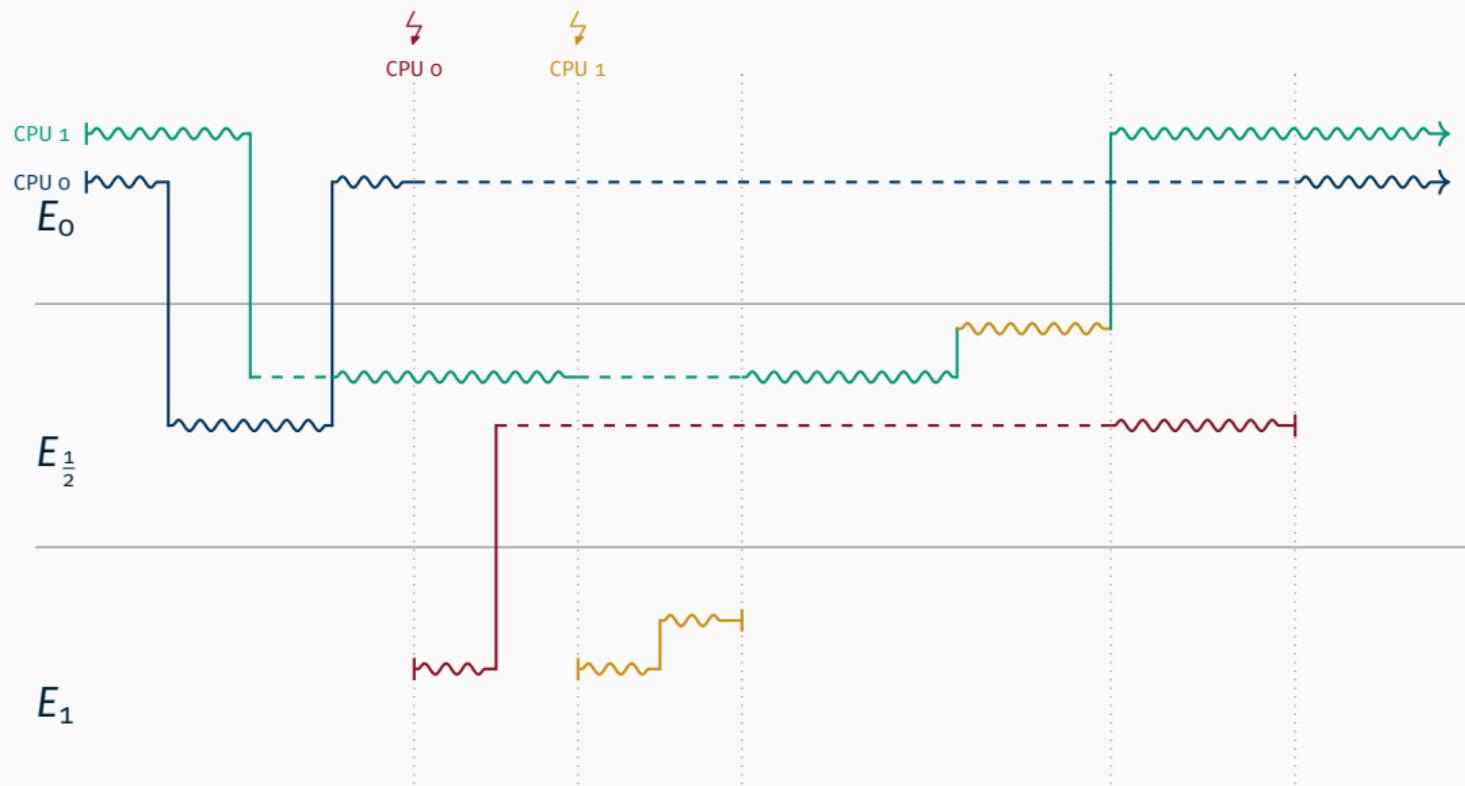
# Beispiel für Mehrkernprozessoren



# Beispiel für Mehrkernprozessoren



# Beispiel für Mehrkernprozessoren



# Fragen?

---

Nächste Woche (28. & 29. November)  
Abgabe von Aufgabe 2 im WinCIP