Übung zu Betriebssystembau

Prolog/Epilog-Modell

14. November 2023

Peter Ulbrich & Alexander Lochmann
(Mit Material vom Lehrstuhl 4 der FAU)

Arbeitsgruppe Systemsoftware
Technische Universität Dortmund

technische universität
dortmund

arbeitsgruppe
systemsoftware

Motivation

Interrupts verändern (potenziell) den Zustand des Systems

Interrupts verändern (potenziell) den Zustand des Systems

```
01 main(){
02   while(1){
03     // ...
04     consume();
05     // ...
06   }
07 }
```

```
01 interrupt_handler(){
02
03
04   produce();
05
06
07 }
```

Ohne Synchronisation

E_0 (Anwendung)

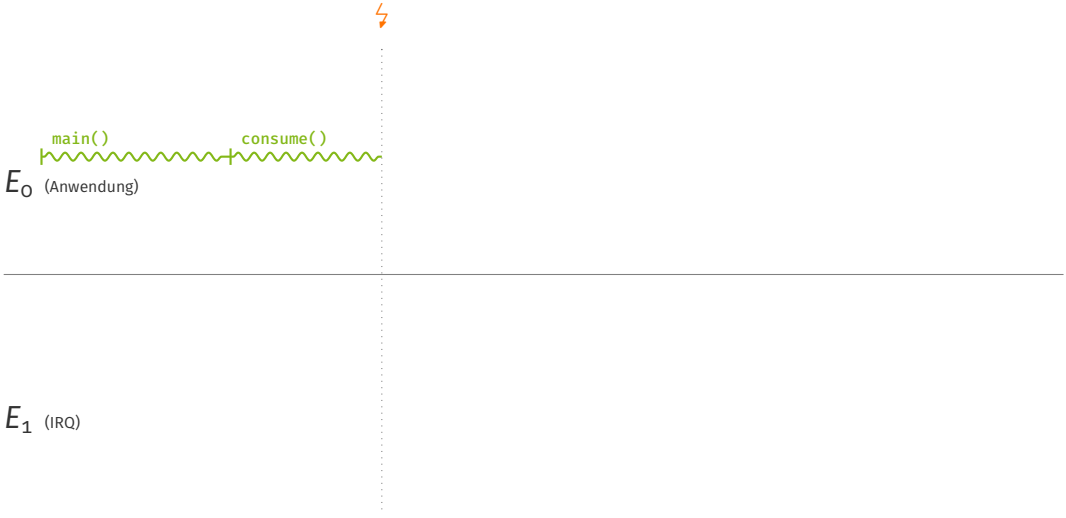
E_1 (IRQ)

Ohne Synchronisation

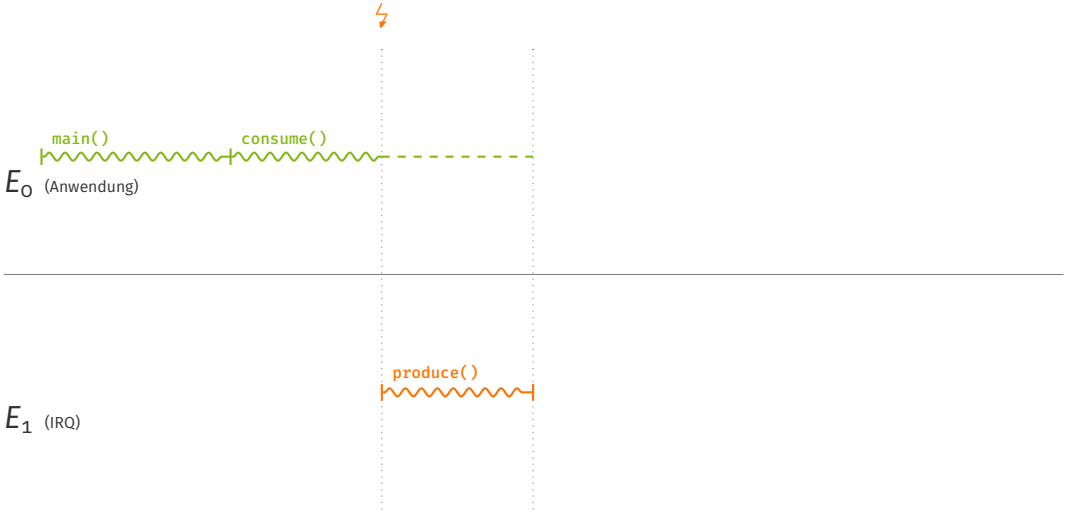
E_0 (Anwendung)

E_1 (IRQ)

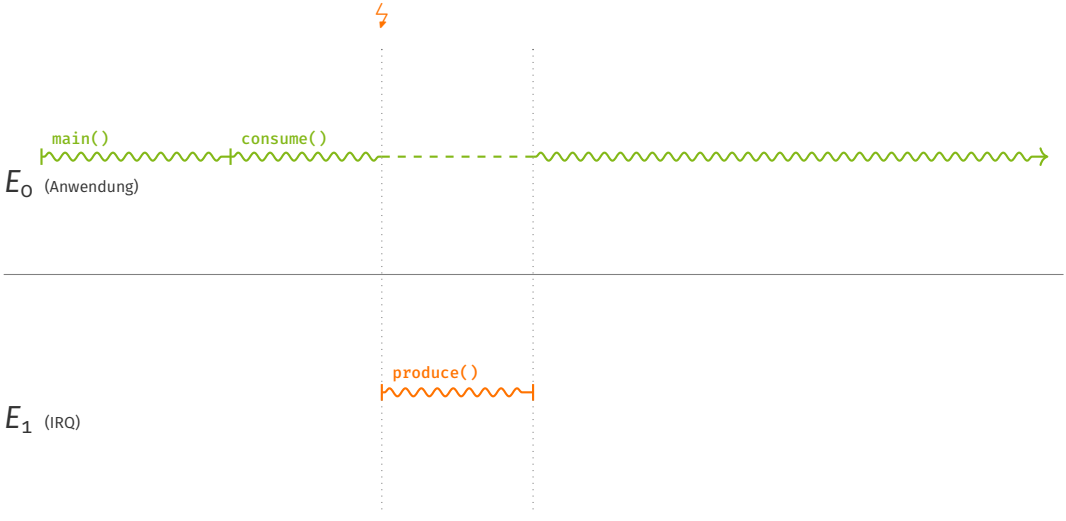
Ohne Synchronisation



Ohne Synchronisation



Ohne Synchronisation



Ohne Synchronisation

```
01 int buf[SIZE];
02 int pos = 0;
03
04 void produce(int data) {
05     if (pos < SIZE)
06         buf[pos++] = data;
07 }
08
09 int consume() {
10     return pos > 0 ? buf[--pos] : -1;
11 }
```

Ohne Synchronisation

Lost Update möglich!

Bewährtes Hausmittel: Mutex

```
01 void produce(int data) {
02     mutex.lock();
03     if (pos < SIZE)
04         buf[pos++] = data;
05     mutex.unlock();
06 }
07
08 int consume() {
09     mutex.lock();
10     int r = pos > 0 ? buf[--pos] : -1;
11     mutex.unlock();
12     return r;
13 }
```

Bewährtes Hausmittel: Mutex

Verklemmt sich!

Weiche Synchronisation

```
01 void produce(int data) {
02     if (pos < SIZE)
03         buf[pos++] = data;
04 }
05
06 int consume() {
07     int x, r = -1;
08     if (pos > 0)
09         do {
10             x = pos;
11             r = buf[x];
12         } while(!CAS(&pos, x, x-1));
13     return r;
14 }
```

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

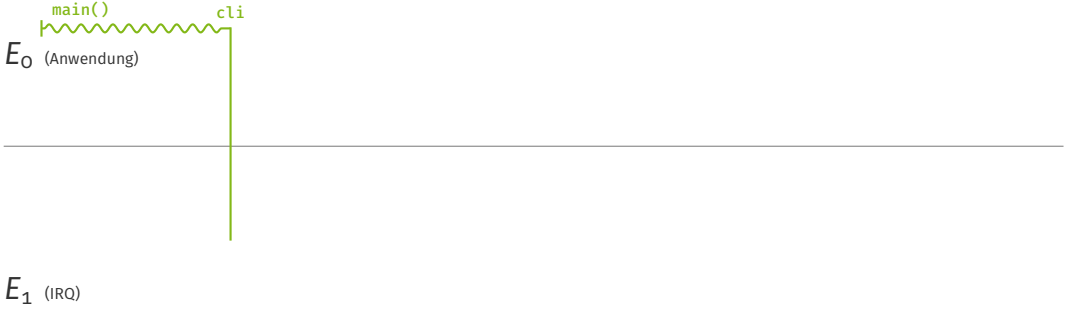
Betriebssystemarchitekt sollte Treiber- und Anwendungsentwicklern entgegenkommen → für Erfolg des Betriebsystems entscheidend

Harte Synchronisation

E_0 (Anwendung)

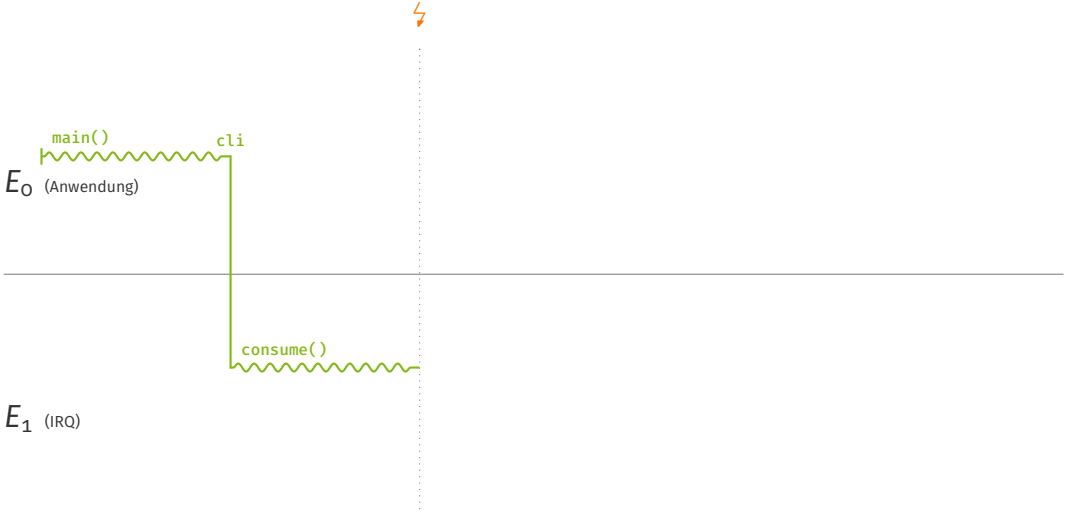
E_1 (IRQ)

Harte Synchronisation

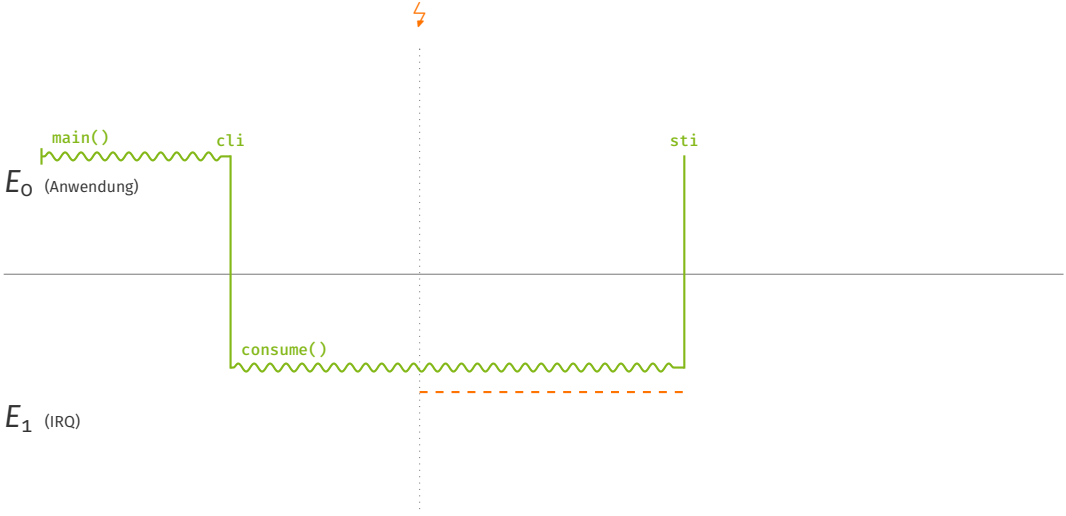


Harte Synchronisation

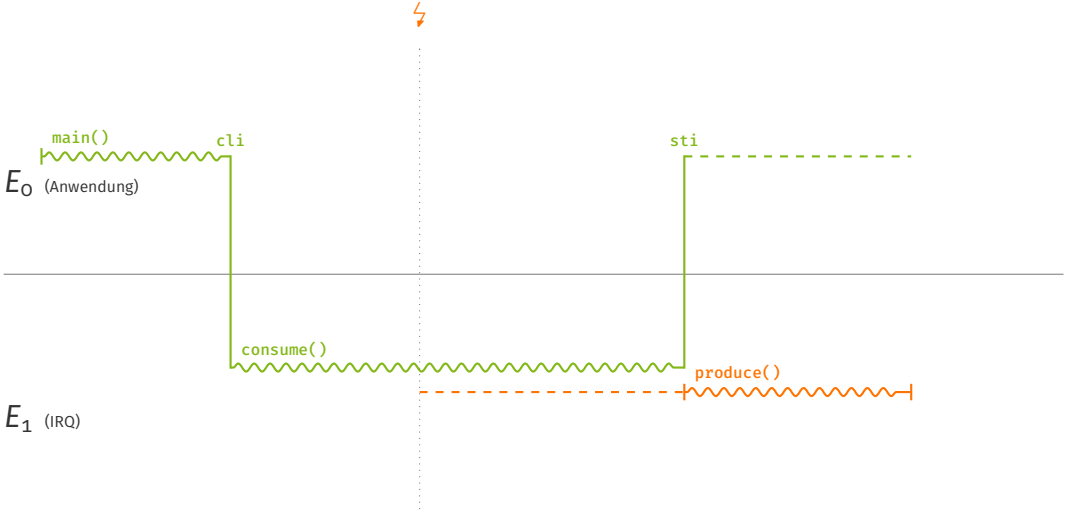
Harte Synchronisation



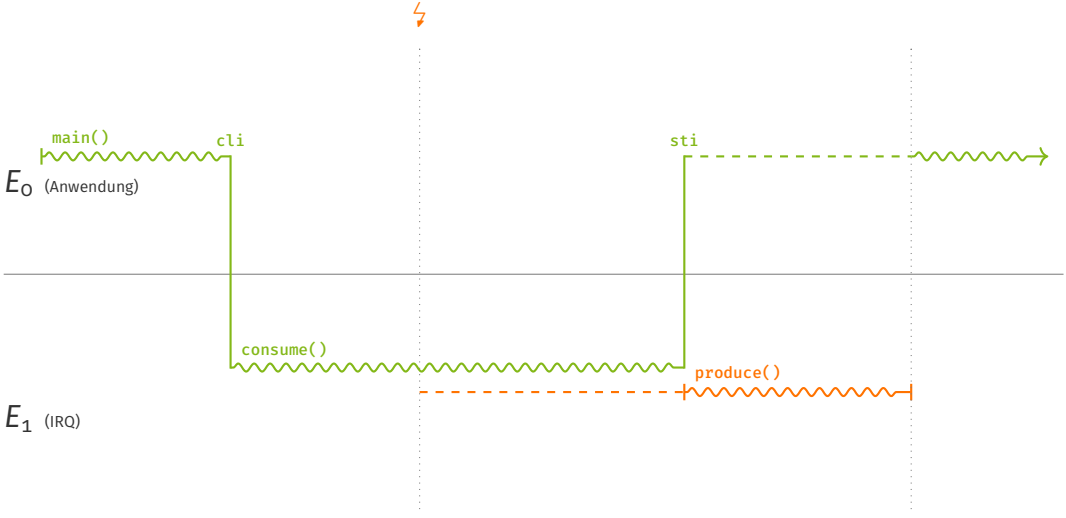
Harte Synchronisation



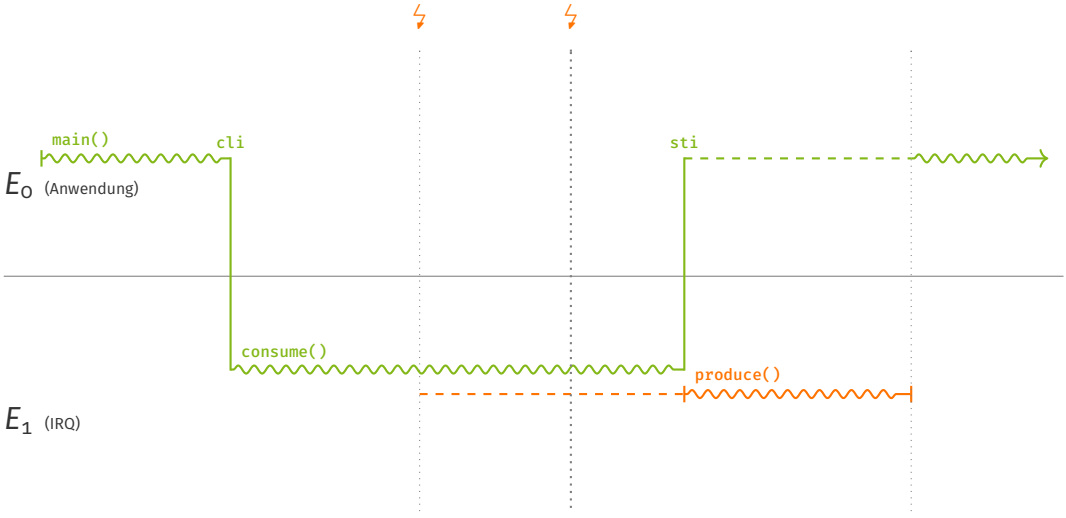
Harte Synchronisation



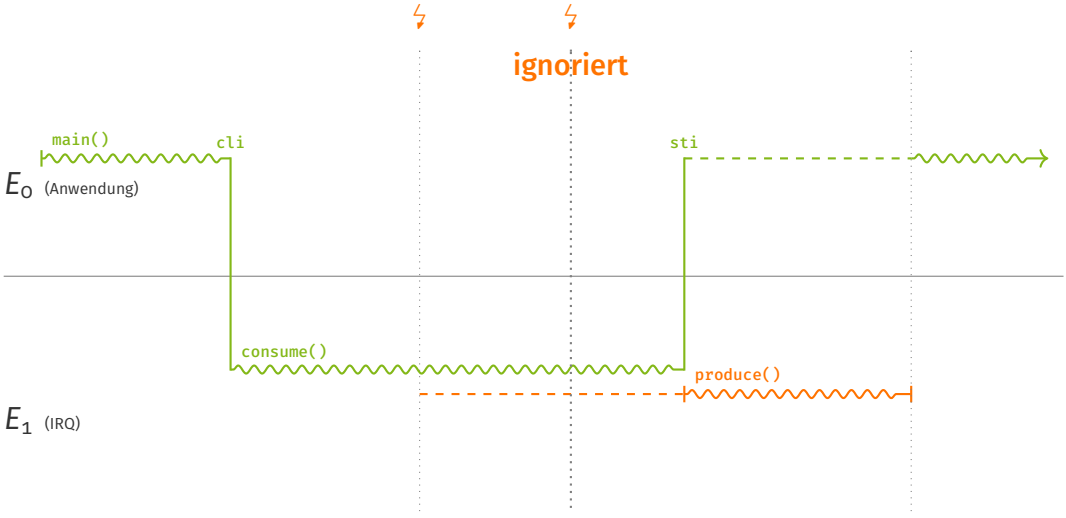
Harte Synchronisation



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

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**

bei **harter Synchronisation** und **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

Operationen

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

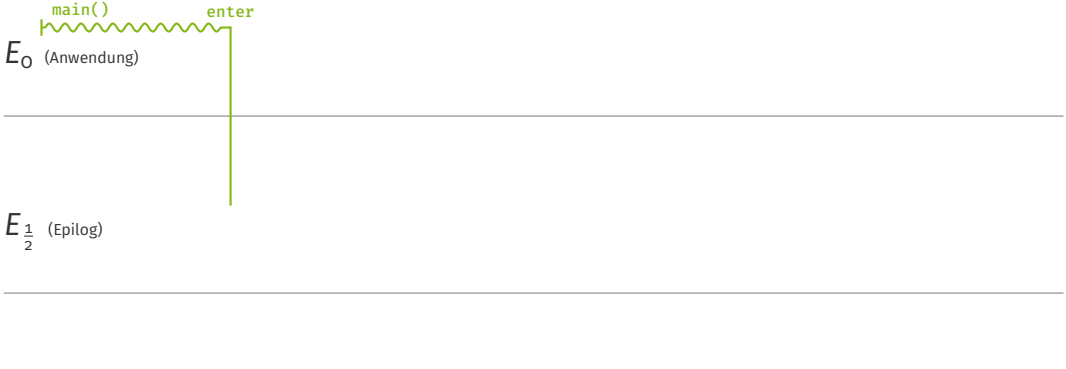
bei **harter Synchronisation** und **Prolog/Epilog-Modell**

Prolog/Epilog-Modell

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

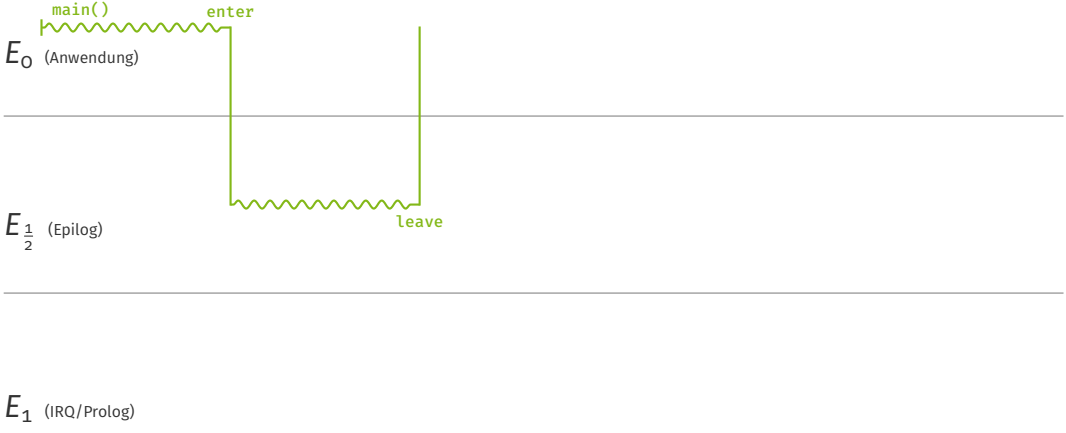
E_1 (IRQ/Prolog)

Prolog/Epilog-Modell

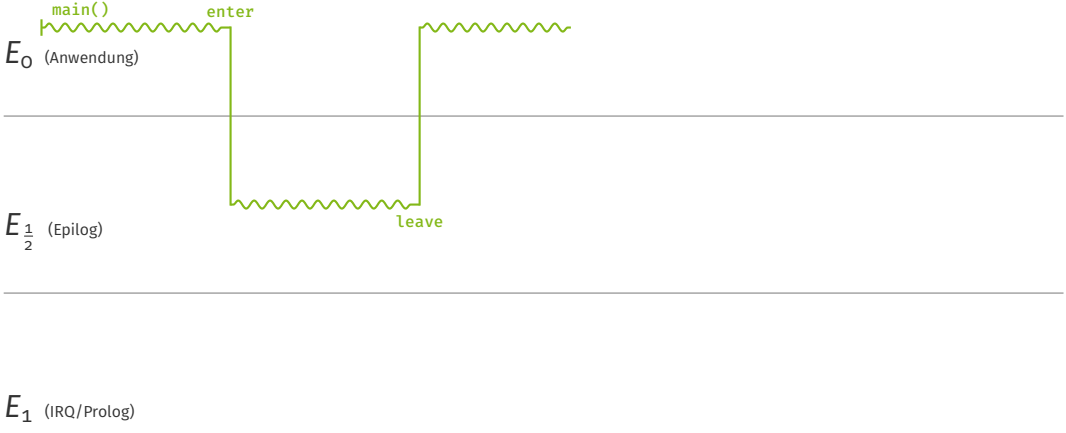


Prolog/Epilog-Modell

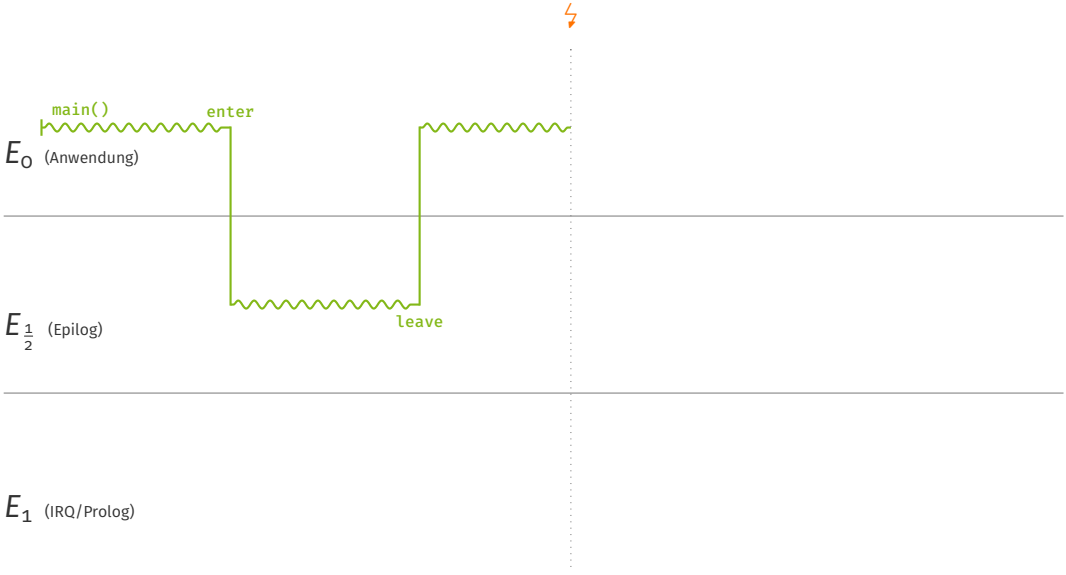
Prolog/Epilog-Modell



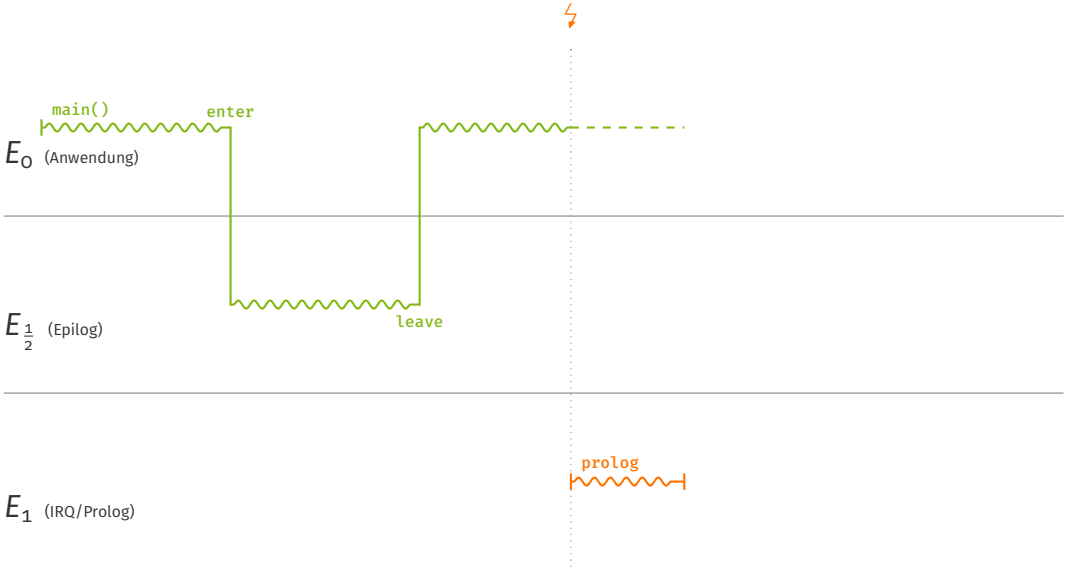
Prolog/Epilog-Modell



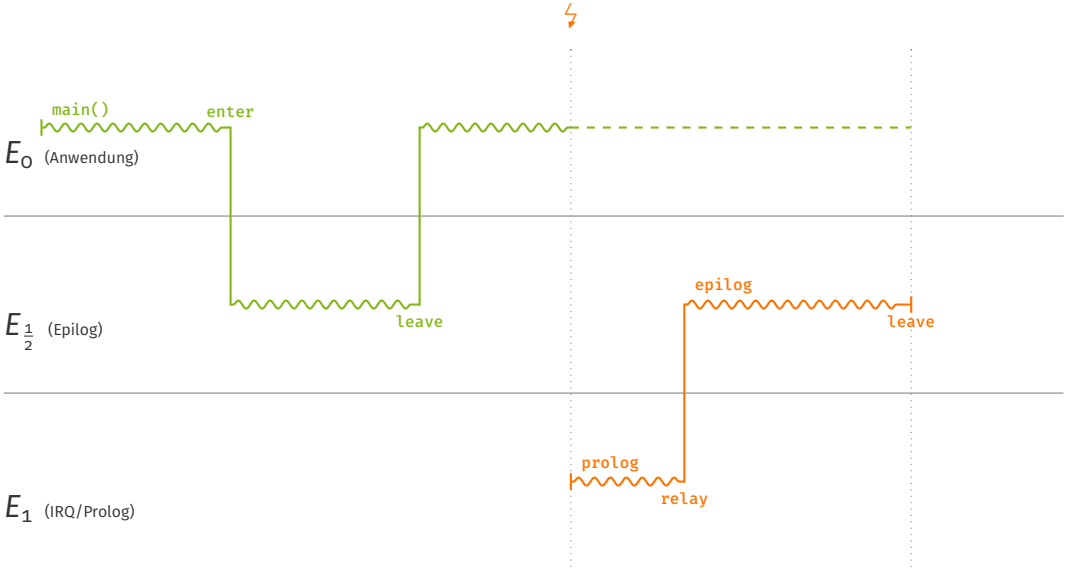
Prolog/Epilog-Modell



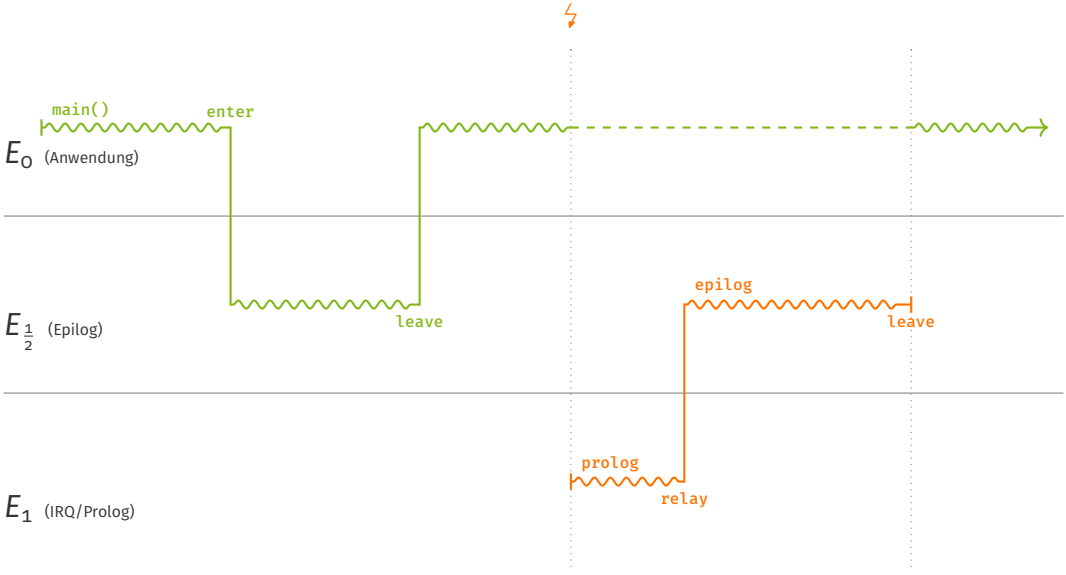
Prolog/Epilog-Modell



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

```
01 main(){
02   while(1){
03     enter();
04     consume();
05     leave();
06   }
07 }
```

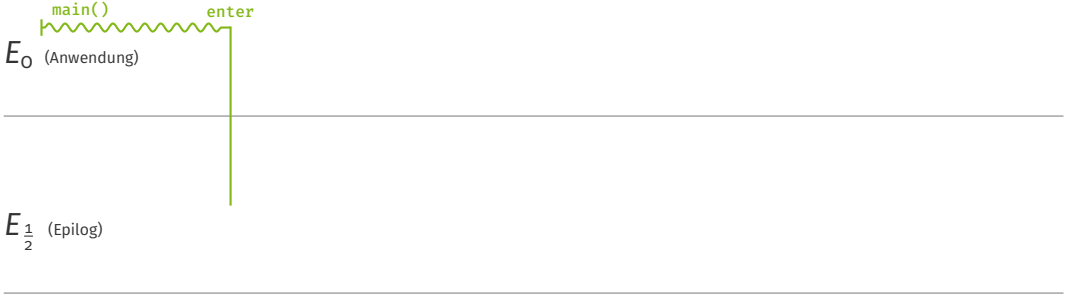
```
01 epilog(){
02
03   // ...
04   produce();
05   // ...
06
07 }
```

Umsetzung

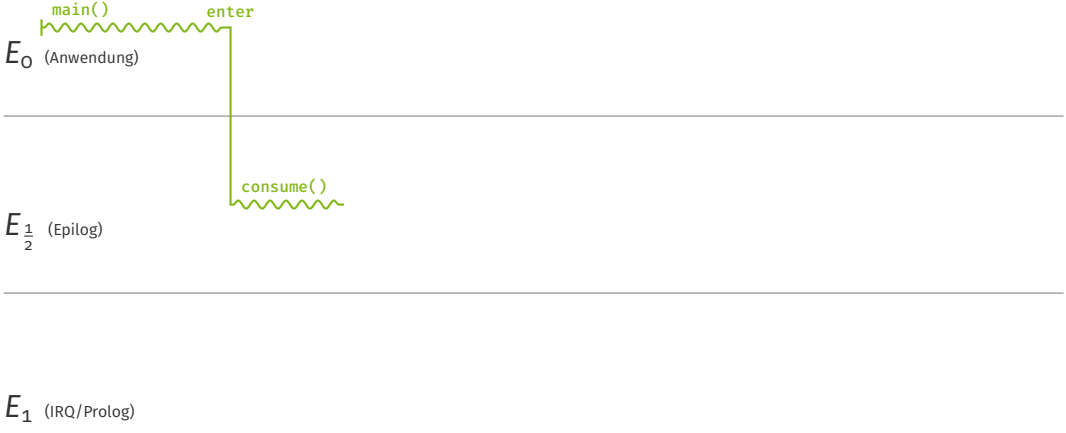
$E_{\frac{1}{2}}$ (Epilog)

E_1 (IRQ/Prolog)

Umsetzung



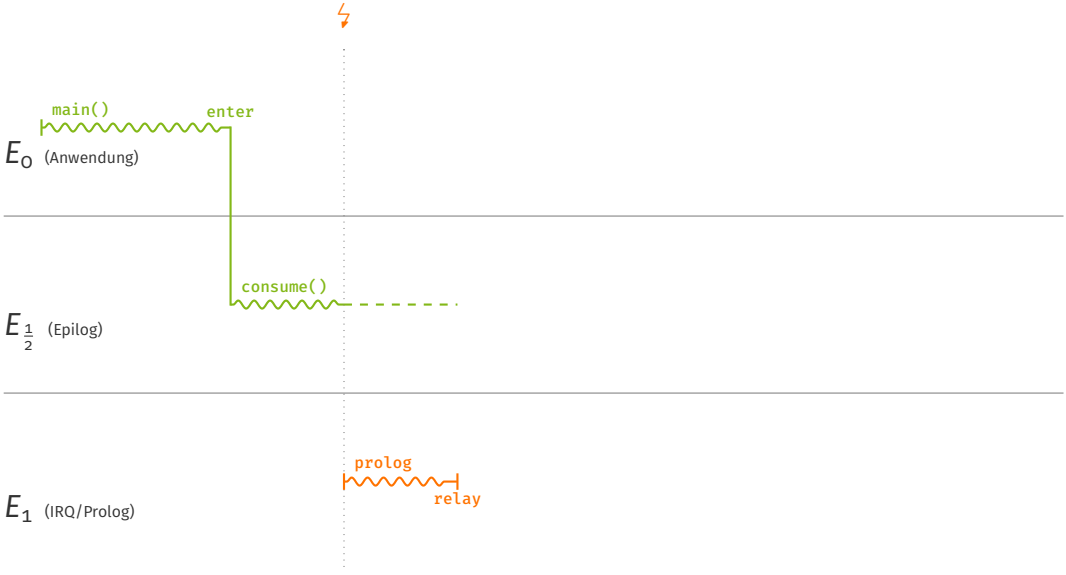
Umsetzung


main()
enter
E_0 (Anwendung)
consume()
$E_{\frac{1}{2}}$ (Epilog)
E_1 (IRQ/Prolog)

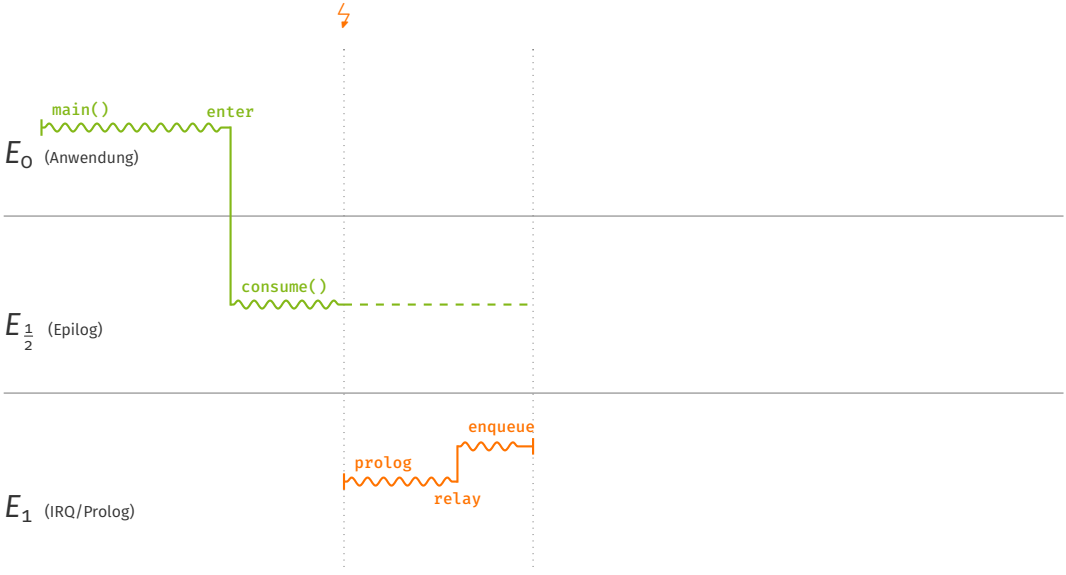
Umsetzung


main()
enter
E_0 (Anwendung)
consume()
$E_{\frac{1}{2}}$ (Epilog)
E_1 (IRQ/Prolog)

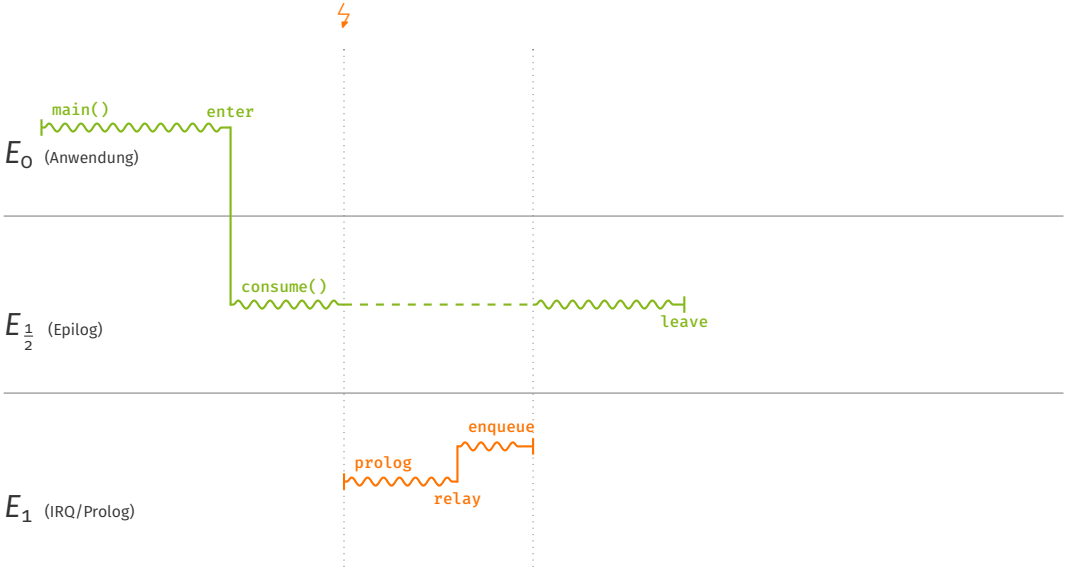
Umsetzung


main()
enter
E_0 (Anwendung)
consume()
$E_{\frac{1}{2}}$ (Epilog)
prolog
relay
E_1 (IRQ/Prolog)

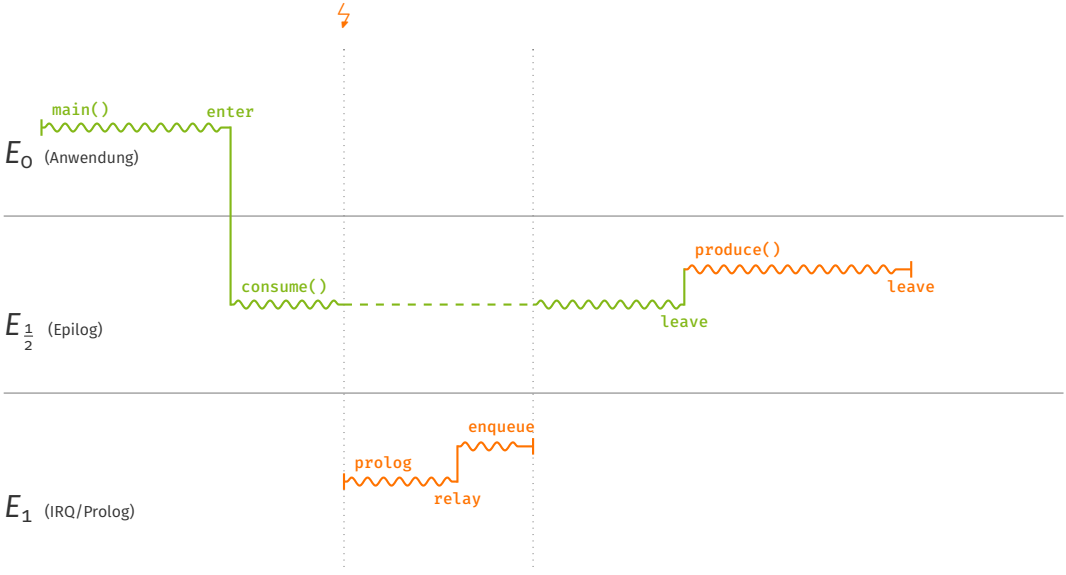
Umsetzung


main()
enter
E_0 (Anwendung)
consume()
$E_{\frac{1}{2}}$ (Epilog)
enqueue
prolog
relay
E_1 (IRQ/Prolog)

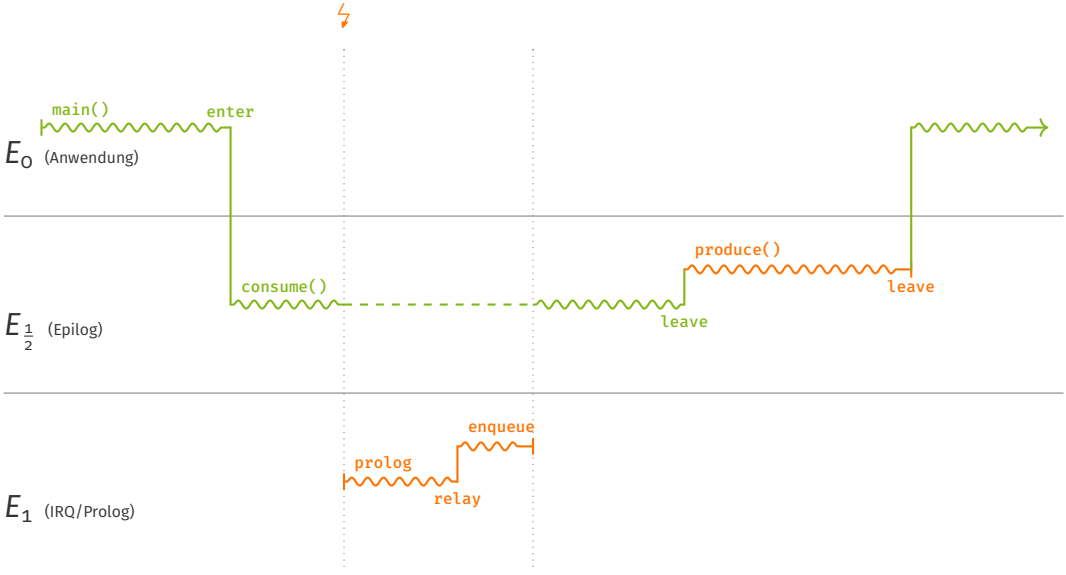
Umsetzung


main()
enter
E_0 (Anwendung)
consume()
leave
$E_{\frac{1}{2}}$ (Epilog)
enqueue
prolog
relay
E_1 (IRQ/Prolog)

Umsetzung


main()
enter
E_0 (Anwendung)
consume()
produce()
leave
leave
$E_{\frac{1}{2}}$ (Epilog)
enqueue
prolog
relay
E_1 (IRQ/Prolog)

Umsetzung


main()
enter
E_0 (Anwendung)
consume()
produce()
leave
$E_{\frac{1}{2}}$ (Epilog)
leave
enqueue
prolog
relay
E_1 (IRQ/Prolog)

Implementierung

Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

Was wird gebraucht?

Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

Was wird gebraucht?

- `Guard` mit `enter()`, `leave()` und `relay()` für Prioritätsebenen
- Epilogwarteschlange `GateQueue` zum Einreihen der Epiloge

Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

Was wird gebraucht?

- `Guard` mit `enter()`, `leave()` und `relay()` für Prioritätsebenen
- Epilogwarteschlange `GateQueue` zum Einreihen der Epiloge

Was muss angepasst werden?

Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

Was wird gebraucht?

- `Guard` mit `enter()`, `leave()` und `relay()` für Prioritätsebenen
- Epilogwarteschlange `GateQueue` zum Einreihen der Epiloge

Was muss angepasst werden?

- Unterbrechungsbehandlung (`interrupt_handler`) und `Gate` (von `trigger()` zu `prolog()` und `epilog()`)
- alle Treiber (`Keyboard`)
- die Anwendung (`Application`)

Wechsel von harter Synchronisation zu Prolog/Epilog-Modell

Was wird gebraucht?

- `Guard` mit `enter()`, `leave()` und `relay()` für Prioritätsebenen
- Epilogwarteschlange `GateQueue` zum Einreihen der Epiloge

Was muss angepasst werden?

- Unterbrechungsbehandlung (`interrupt_handler`) und `Gate` (von `trigger()` zu `prolog()` und `epilog()`)
- alle Treiber (`Keyboard`)
- die Anwendung (`Application`)
- *alles was hart synchronisiert*

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)
- Eine `Gate`-Instanz darf **nicht mehrfach 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)
- Eine `Gate`-Instanz darf **nicht mehrfach in einer** Epilogwarteschlangen vorkommen
- Eine `Gate`-Instanz kann aber **gleichzeitig in unterschiedlichen** 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)
- Eine `Gate`-Instanz darf **nicht mehrfach in einer** Epilogwarteschlangen vorkommen
- Eine `Gate`-Instanz kann aber **gleichzeitig in unterschiedlichen** 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)
- Eine `Gate`-Instanz darf **nicht mehrfach in einer** Epilogwarteschlangen vorkommen
- Eine `Gate`-Instanz kann aber **gleichzeitig in unterschiedlichen** 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)
- Eine `Gate`-Instanz darf **nicht mehrfach in einer** Epilogwarteschlangen vorkommen
- Eine `Gate`-Instanz kann aber **gleichzeitig in unterschiedlichen** Epilogwarteschlangen vorkommen
- Zu jedem Zeitpunkt darf **maximal ein Kern** Epiloge ausführen
  ➔ Verwendung eines **big kernel lock** (BKL) via Spinlock

Prolog/Epilog-Modell auf Mehrkernprozessoren

- Jeder Kern hat eine **eigene** Epilogwarteschlange
  (damit die Epiloge auf dem selben Kern wie deren zugehörige Prologe ausgeführt werden)
- Eine `Gate`-Instanz darf **nicht mehrfach in einer** Epilogwarteschlangen vorkommen
- Eine `Gate`-Instanz kann aber **gleichzeitig in unterschiedlichen** Epilogwarteschlangen vorkommen
- Zu jedem Zeitpunkt darf **maximal ein Kern** Epiloge ausführen
  ➔ Verwendung eines **big kernel lock** (BKL) via Spinlock
- Korrekte Sperrreihenfolge ist extrem wichtig!

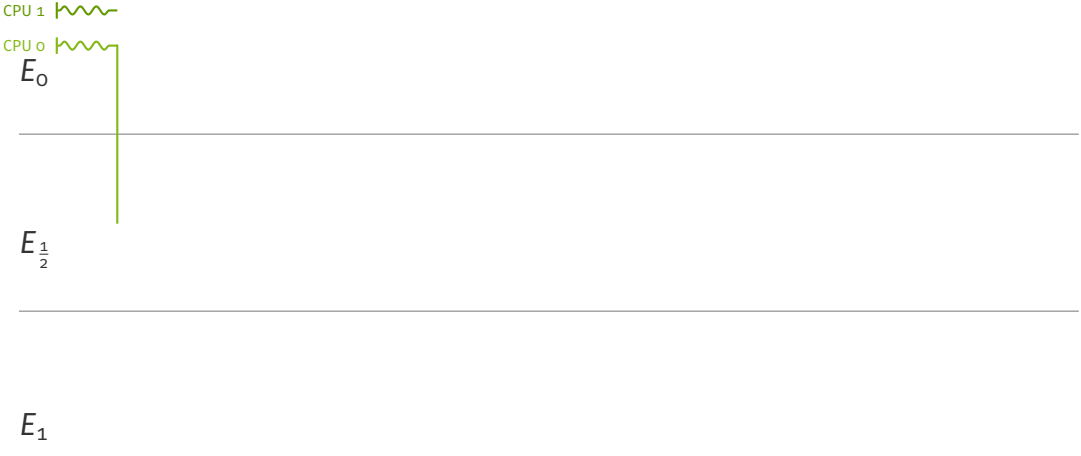
Beispiel für Mehrkernprozessoren

E_0

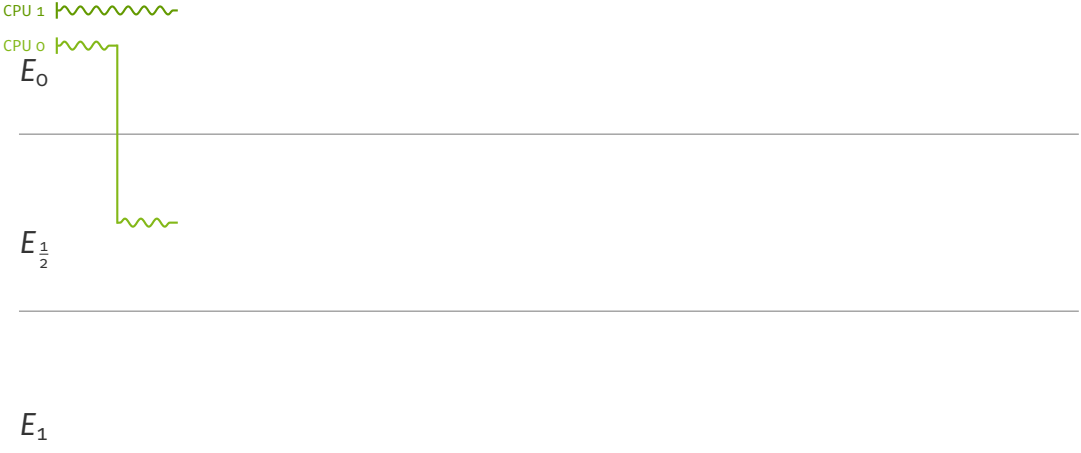
$E_{\frac{1}{2}}$

E_1

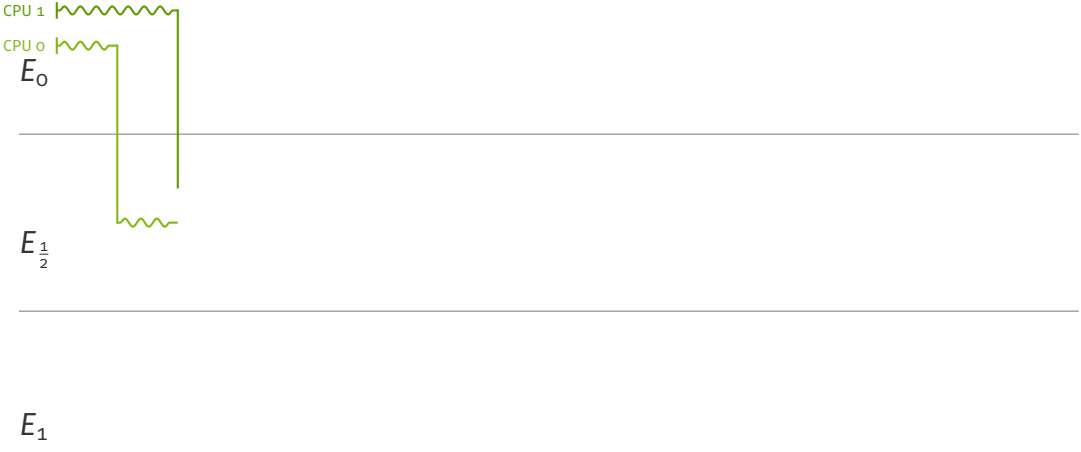
Beispiel für Mehrkernprozessoren



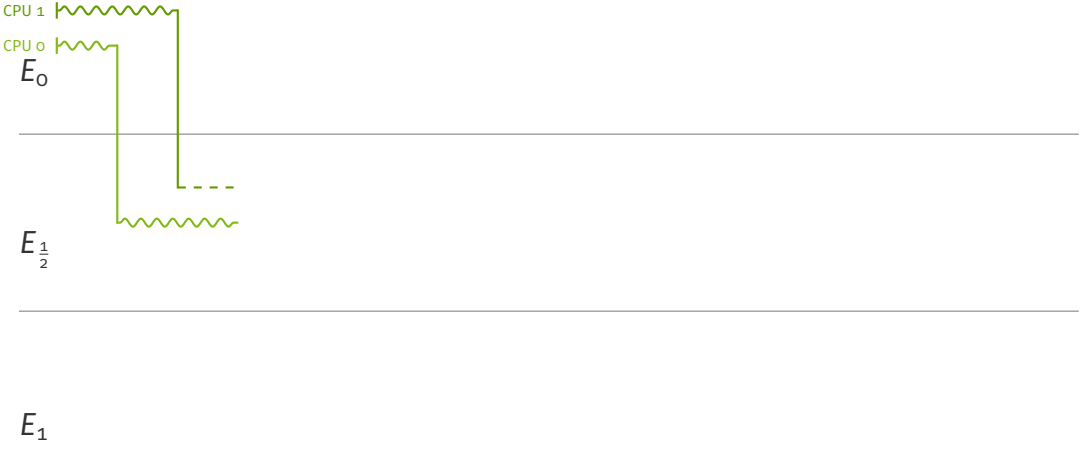
Beispiel für Mehrkernprozessoren



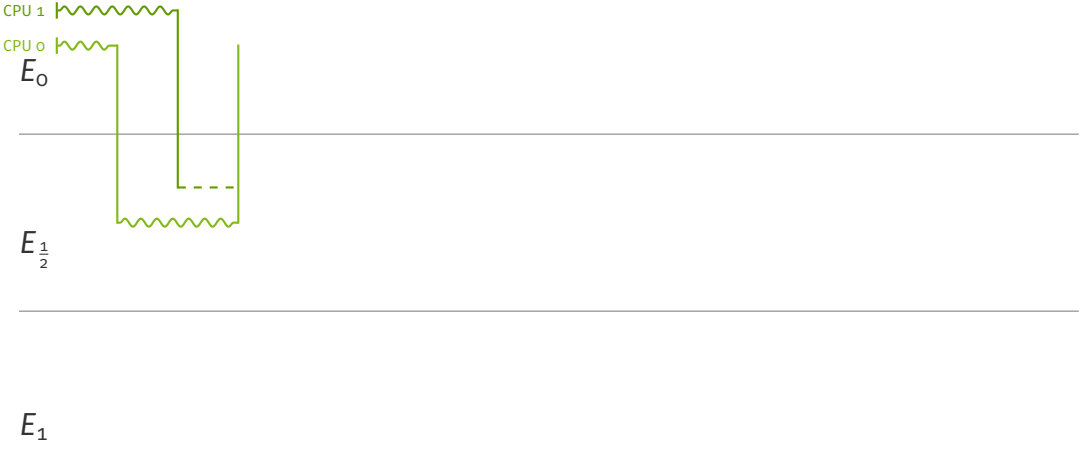
Beispiel für Mehrkernprozessoren



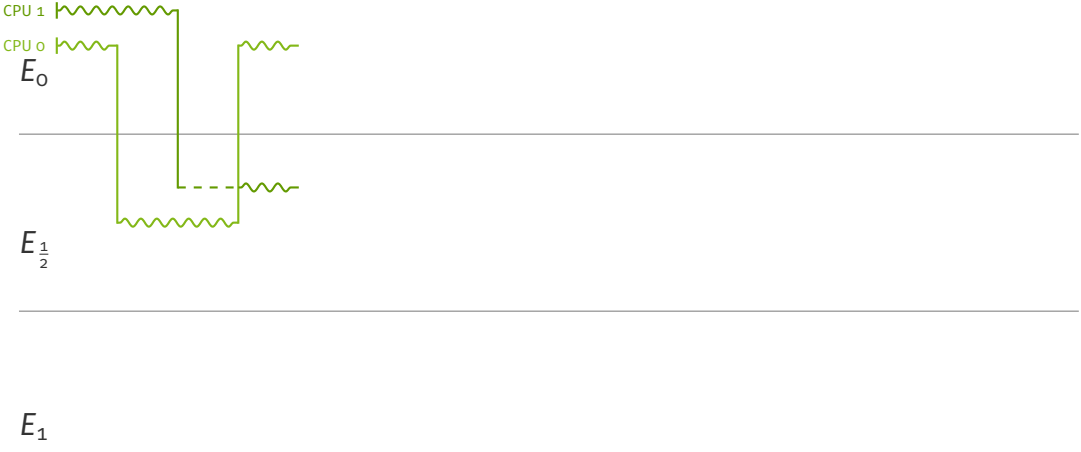
Beispiel für Mehrkernprozessoren



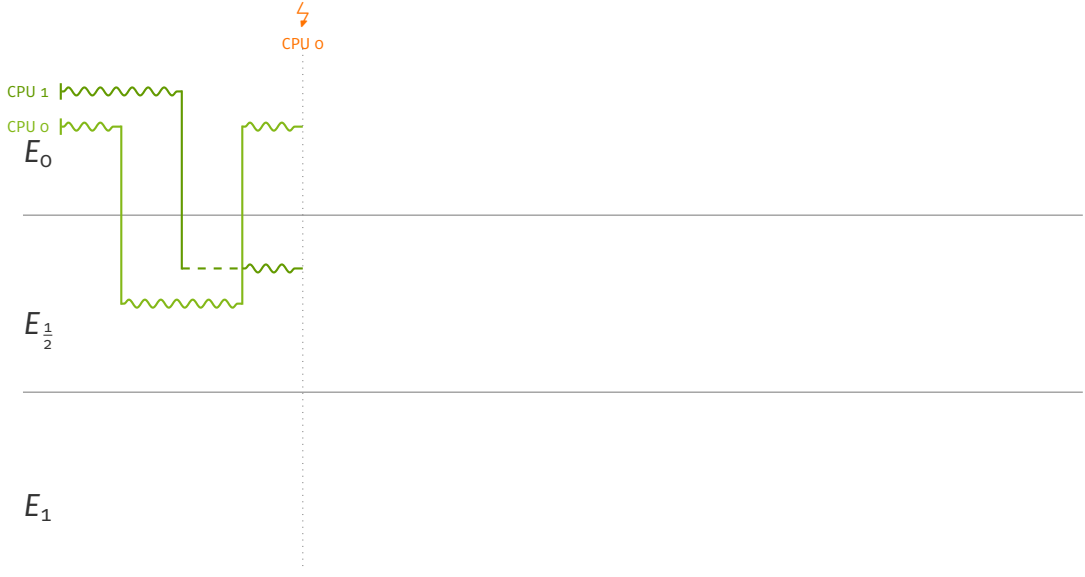
Beispiel für Mehrkernprozessoren



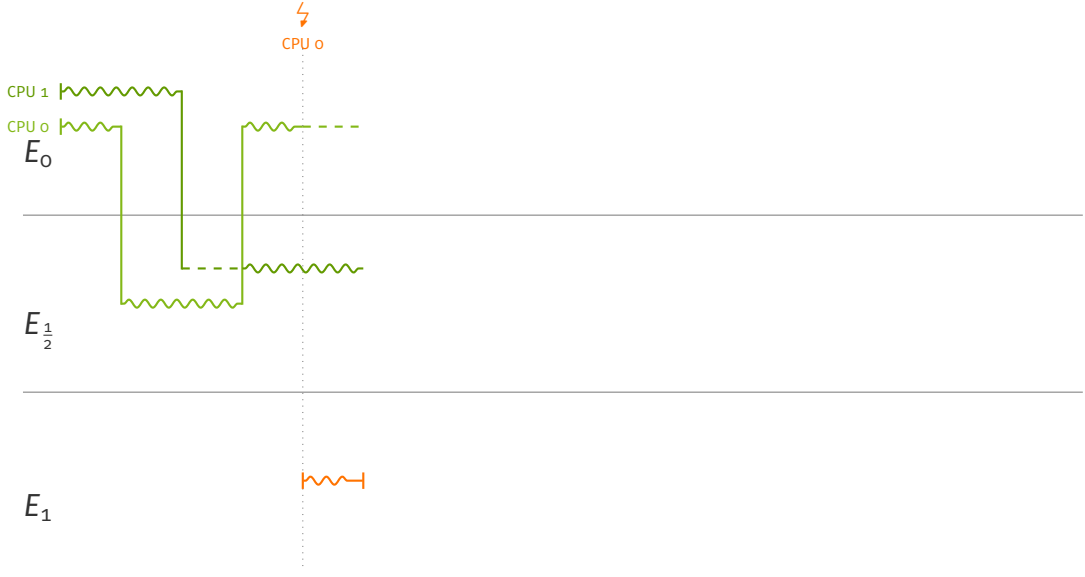
Beispiel für Mehrkernprozessoren



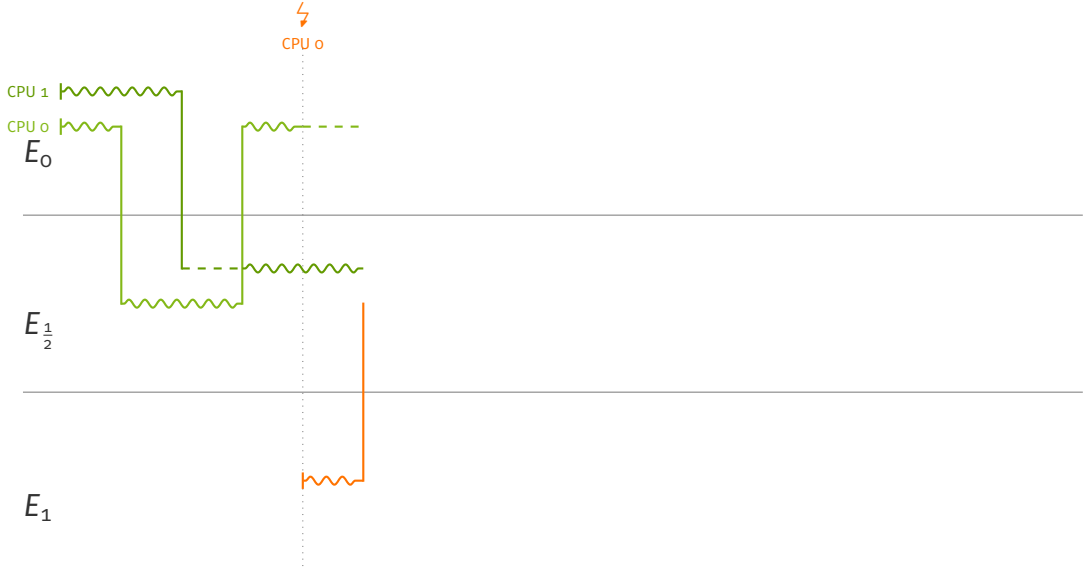
Beispiel für Mehrkernprozessoren



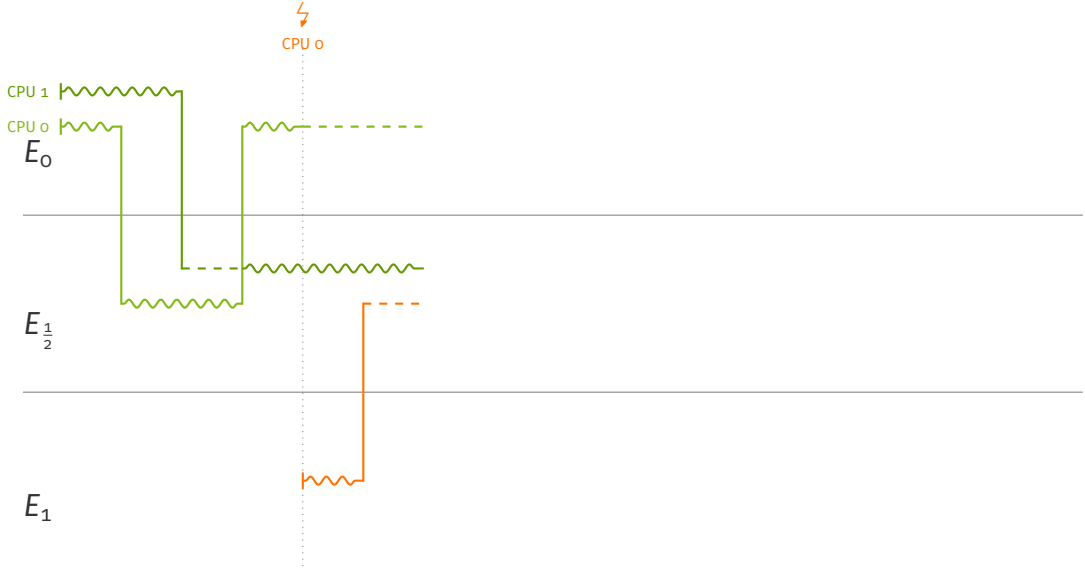
Beispiel für Mehrkernprozessoren



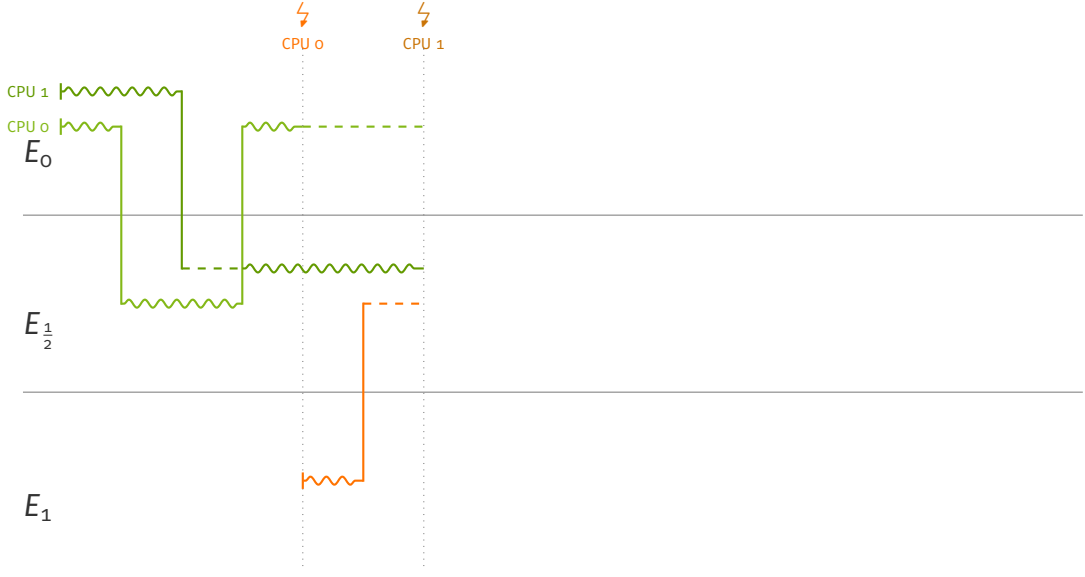
Beispiel für Mehrkernprozessoren



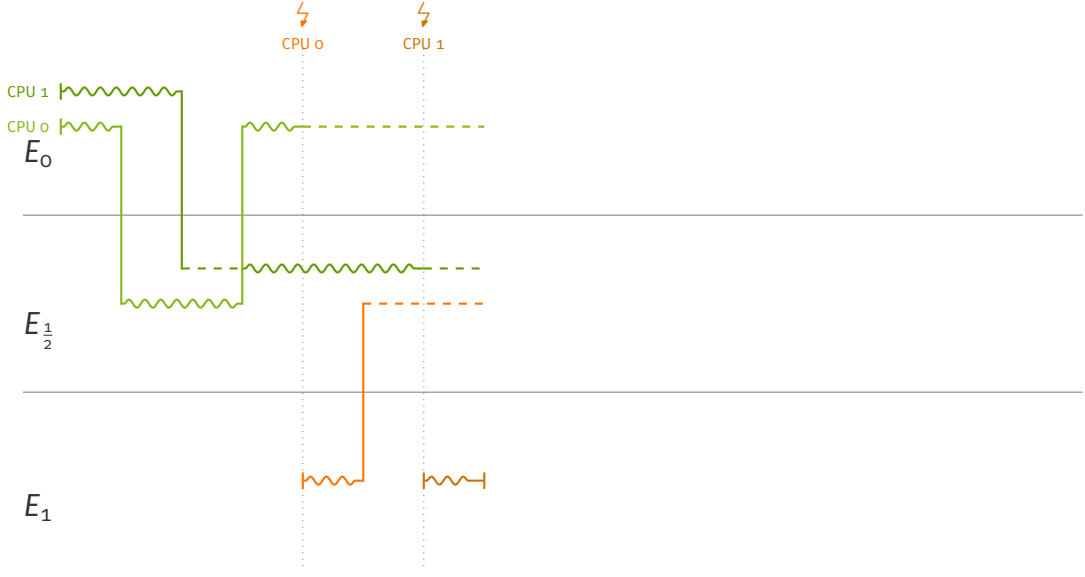
Beispiel für Mehrkernprozessoren



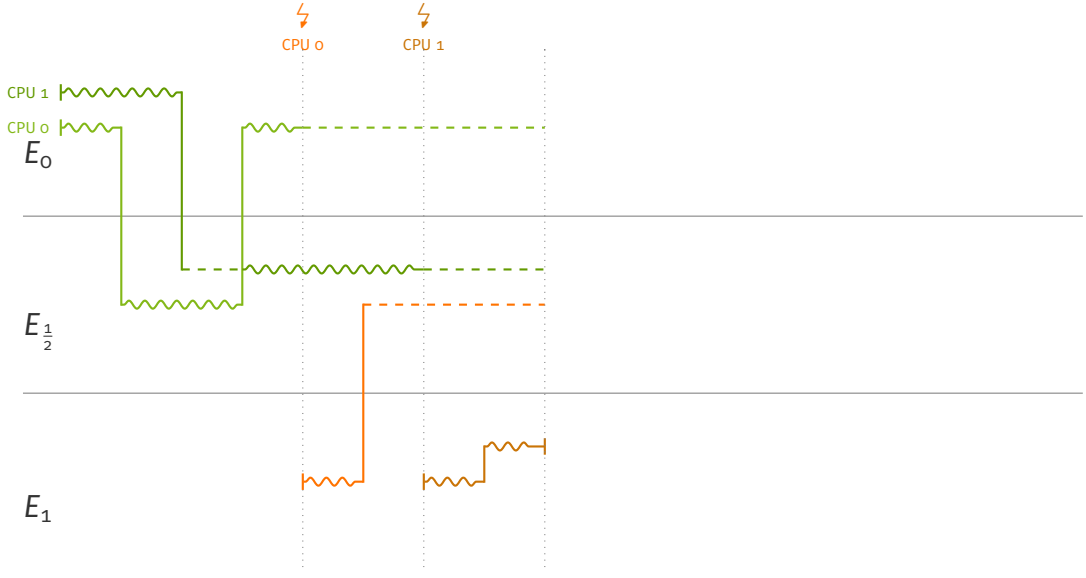
Beispiel für Mehrkernprozessoren



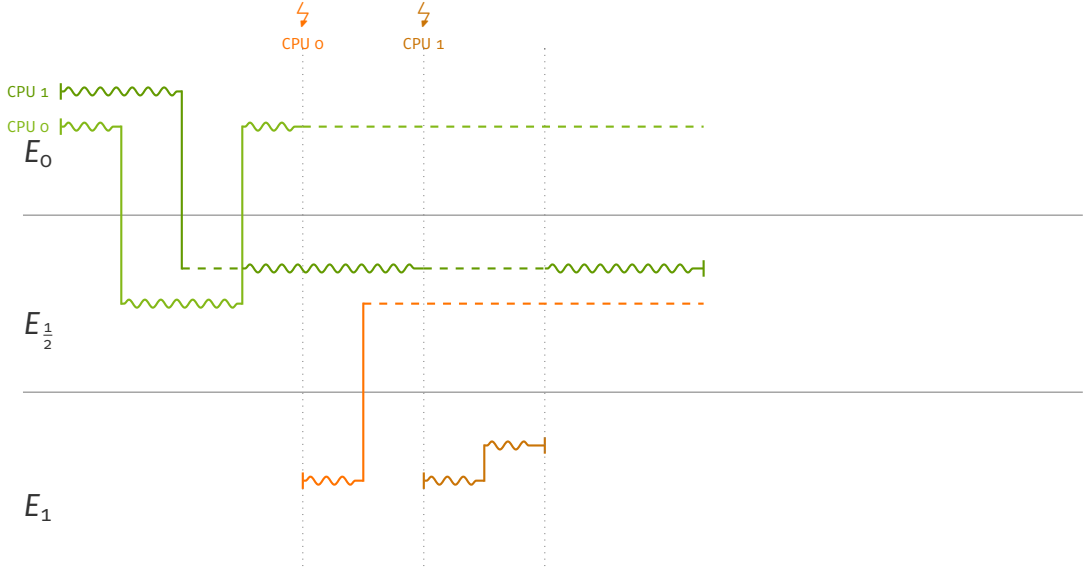
Beispiel für Mehrkernprozessoren



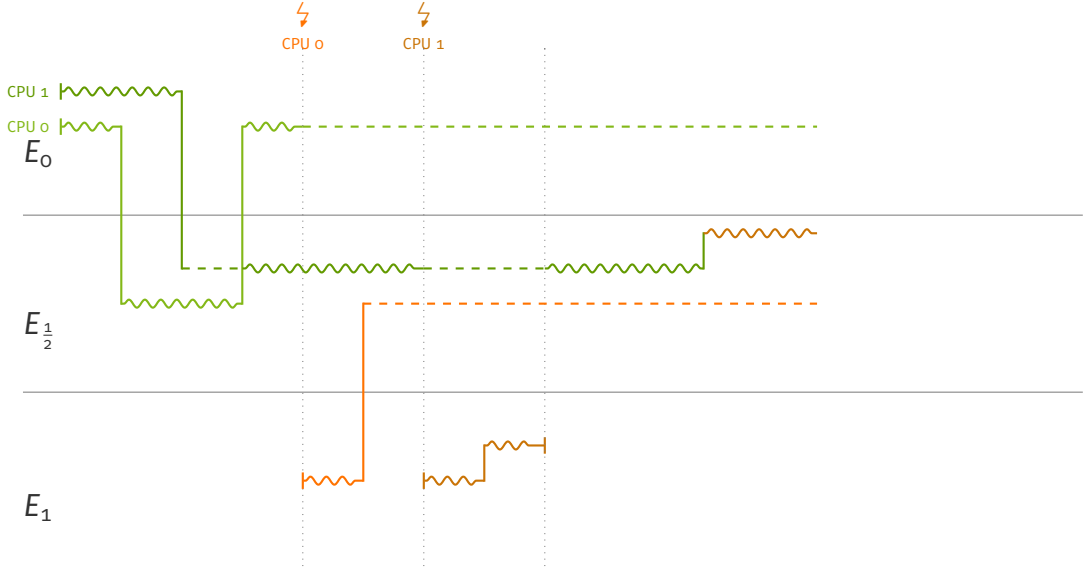
Beispiel für Mehrkernprozessoren



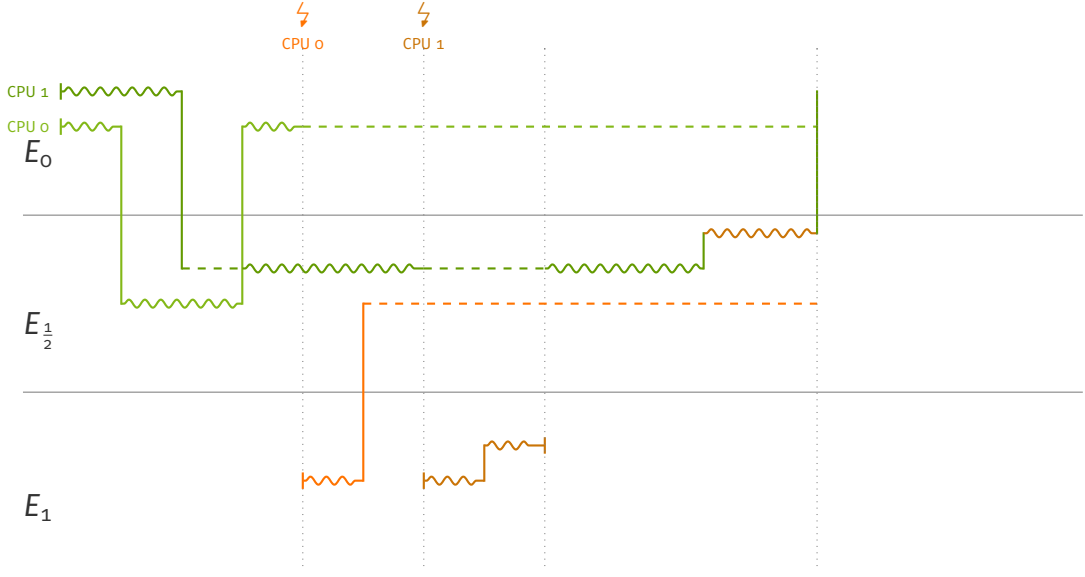
Beispiel für Mehrkernprozessoren



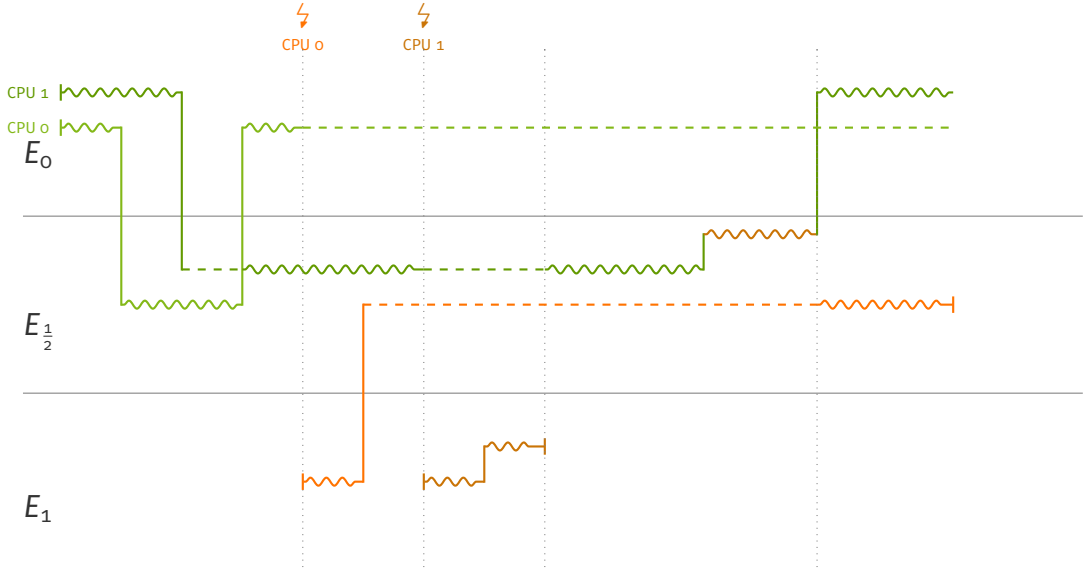
Beispiel für Mehrkernprozessoren



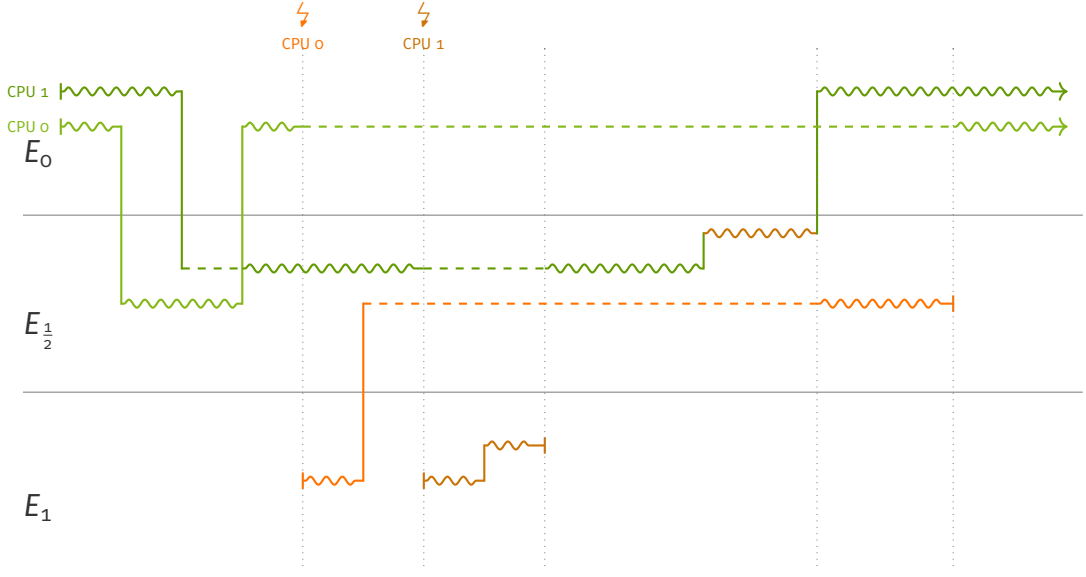
Beispiel für Mehrkernprozessoren



Beispiel für Mehrkernprozessoren



Beispiel für Mehrkernprozessoren



Fragen?

Abgabe der Aufgabe bis Mittwoch, den 29. November