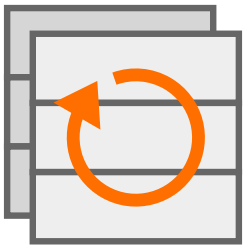


Operating
System
Group

TUHH
Technische Universität Hamburg

BSB – Übung 3: Pro-/Epilogmodell

Yannick Loeck

2022-05-11

Inhalt

TUHH
Technische Universität Hamburg

BSB - HÜ3

Interrupts lange sperren erzeugt Latenzen.

Worum geht es in der Übung heute:

- Aufteilen von Interrupt-Behandlung
- Prolog und Epilog

Probleme bei Harter Synchronisation

TUHH
Technische Universität Hamburg

Main

```
while (1) {
  download_more_ram();
  print_ram_contents();
}
```

Probleme bei Harter Synchronisation

Main

```
while (1) {
  download_more_ram();
  print_ram_contents();
}
```

ISR

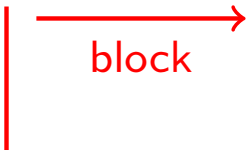
```
on_critical_temp() {
  throttle();
  print_info();
}
```

- Gemeinsame Daten durch Ausgabe

Probleme bei Harter Synchronisation

Main

```
while (1) {
  download_more_ram();
  cli();
  print_ram_contents();
  sti();
}
```



ISR

```
on_critical_temp() {
  throttle();
  print_info();
}
```

- Problem: Worst Case Execution Time von `print_ram_contents`

Bessere Lösung

Main

```
while (1) {
  download_more_ram();
  print_ram_contents();
}
```

ISR

```
prologue() {
  throttle();    // HW
  return true;
}

epilogue() {
  print_info(); // SW
}
```

- Auftrennung der ISR
- Schneller Hardwareteil (jederzeit)

Bessere Lösung

Technische Universität Hamburg

Main

```
while (1) {
  download_more_ram();
  enter();
  print_ram_contents();
  leave();
}
```

sync

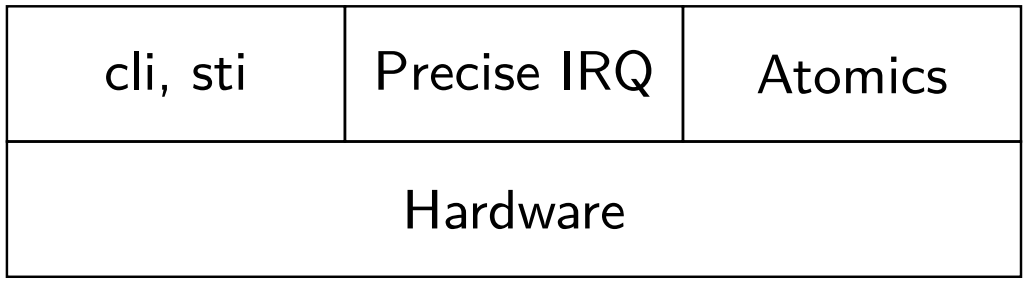
ISR

```
prologue() {
  throttle();    // HW
  return true;
}

epilogue() {
  print_info(); // SW
}
```

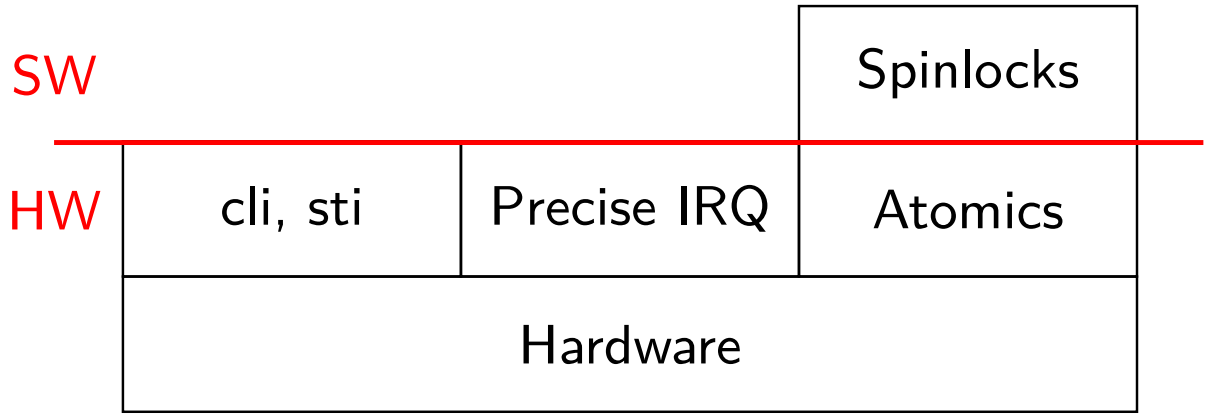
- Auftrennung der ISR
- Schneller Hardwareteil (jederzeit)
- Langsamer Softwareteil (synchron)

Harte Synchronisation `cli, sti` E0 und E1 auf *einer CPU*



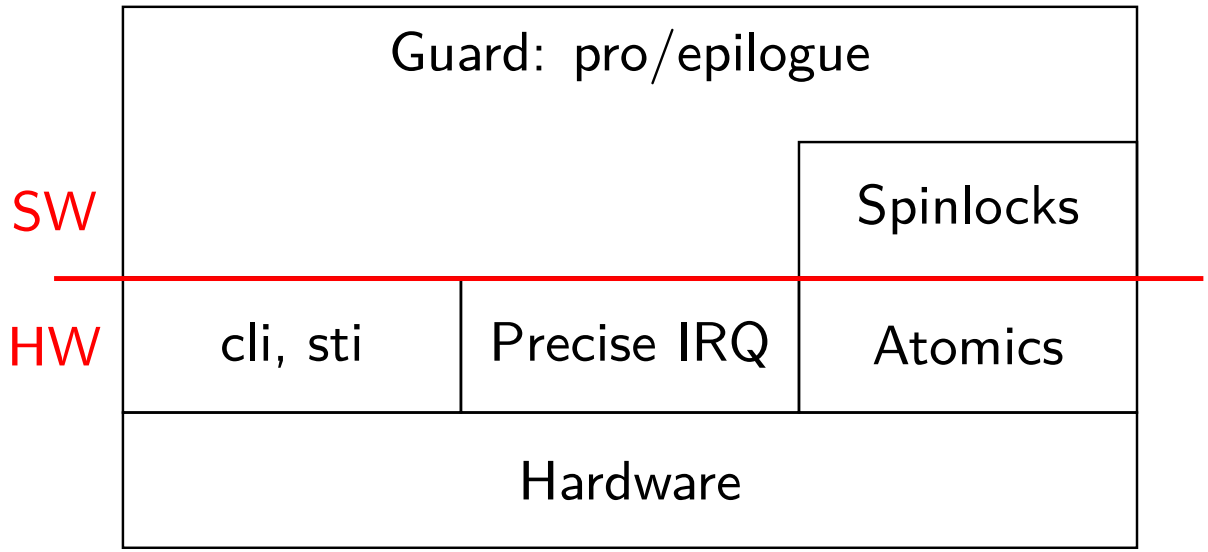
Synchronisationsprimitiven

Harte Synchronisation	`cli, sti`	E0 und E1 auf *einer CPU*
Spinlocks	`lock, unlock`	mehrere CPUs



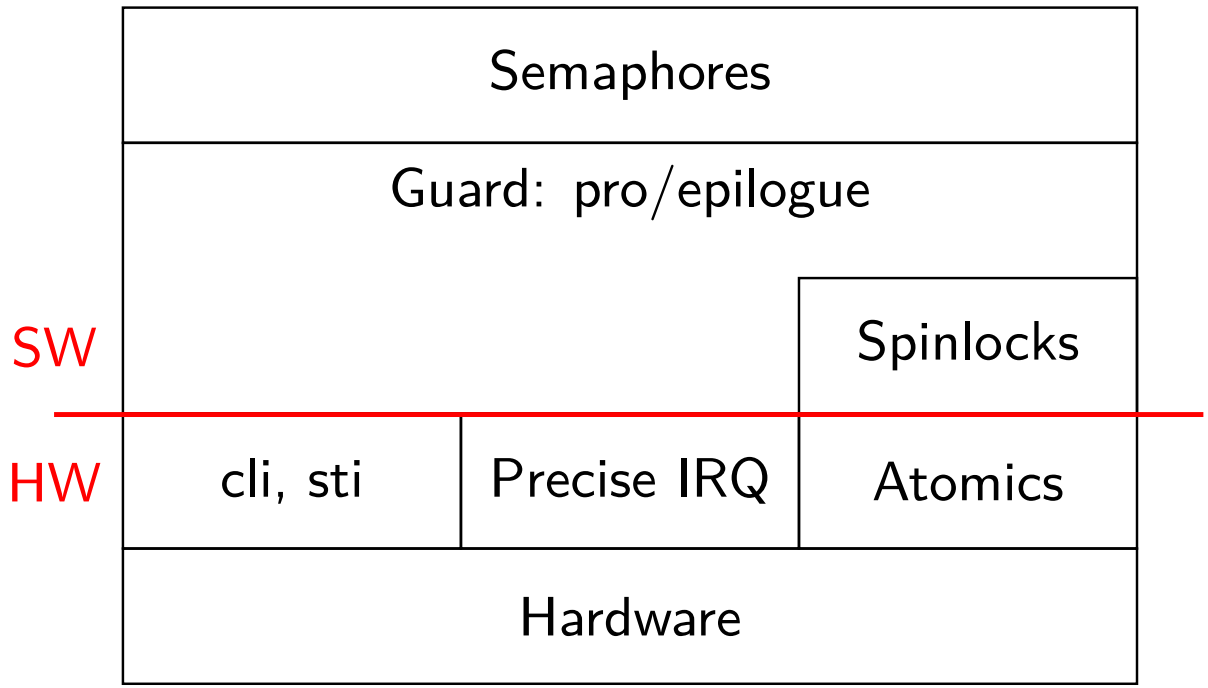
Synchronisationsprimitiven

Harte Synchronisation	`cli, sti`	E0 und E1 auf *einer CPU*
Spinlocks	`lock, unlock`	mehrere CPUs



Synchronisationsprimitiven

Harte Synchronisation	`cli, sti`	E0 und E1 auf *einer CPU*
Spinlocks	`lock, unlock`	mehrere CPUs



Prolog vs. Epilog

TUHH
Technische Universität Hamburg

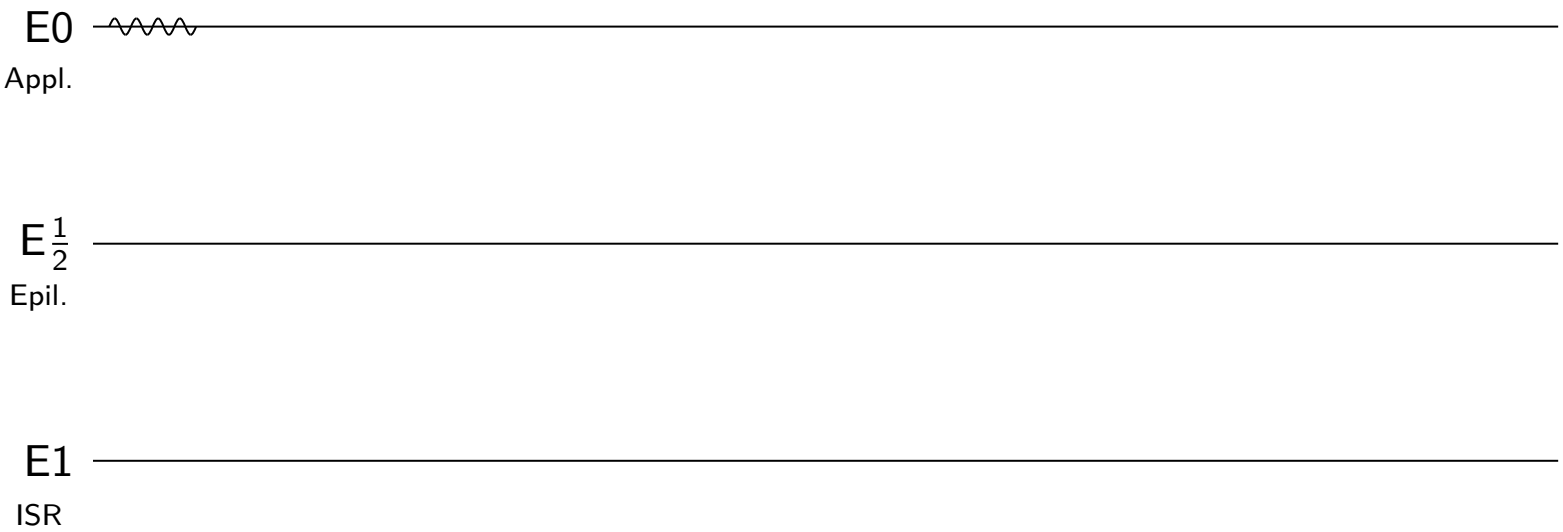
Prolog

- Von der HW aktiviert
- Interrupts gesperrt
- Latenzreduzierung
- Kann Epilog anfordern

Epilog

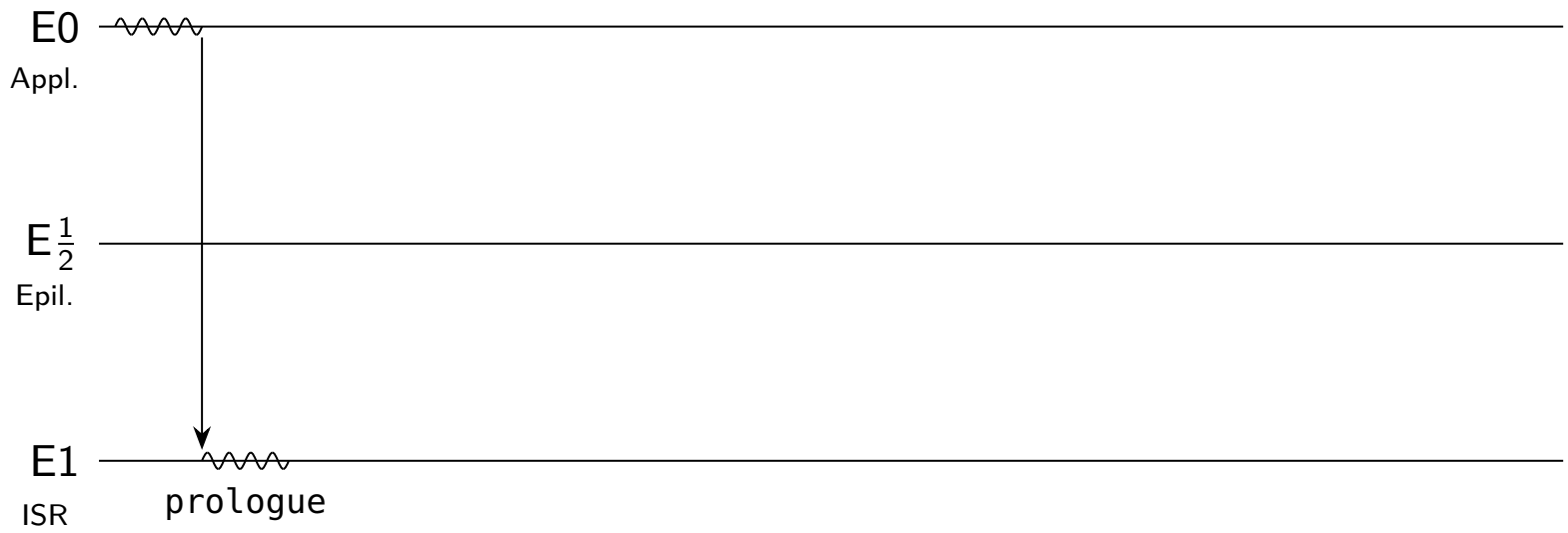
- Folge einer Prologausführung
- Interrupts aktiv
- Ausführung greedy
- Maximal ein Epilog gleichzeitig
- Epiloge auf selber CPU wie Prolog

Pro-/Epilogmodell



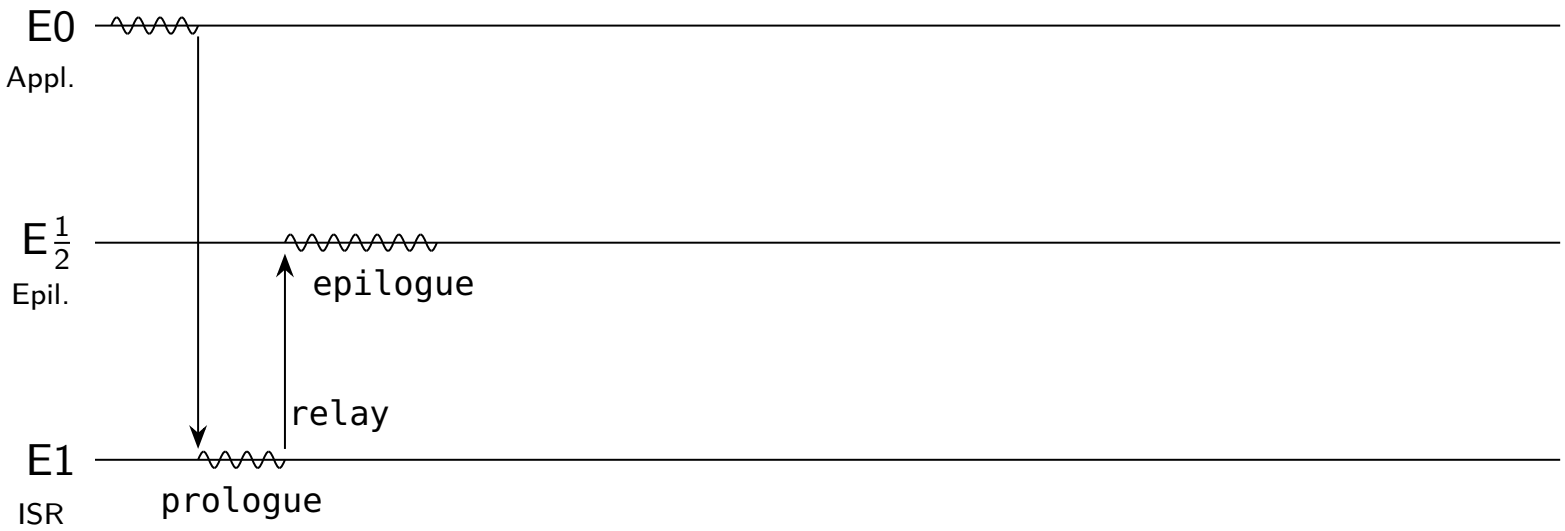
- Anwendungsfluss auf E0
- Prologe (kritischer Teil der ISR) auf E1, Interrupts deaktiviert
- Epiloge (*optionaler* Rest der ISR) auf E$\frac{1}{2}$

Pro-/Epilogmodell



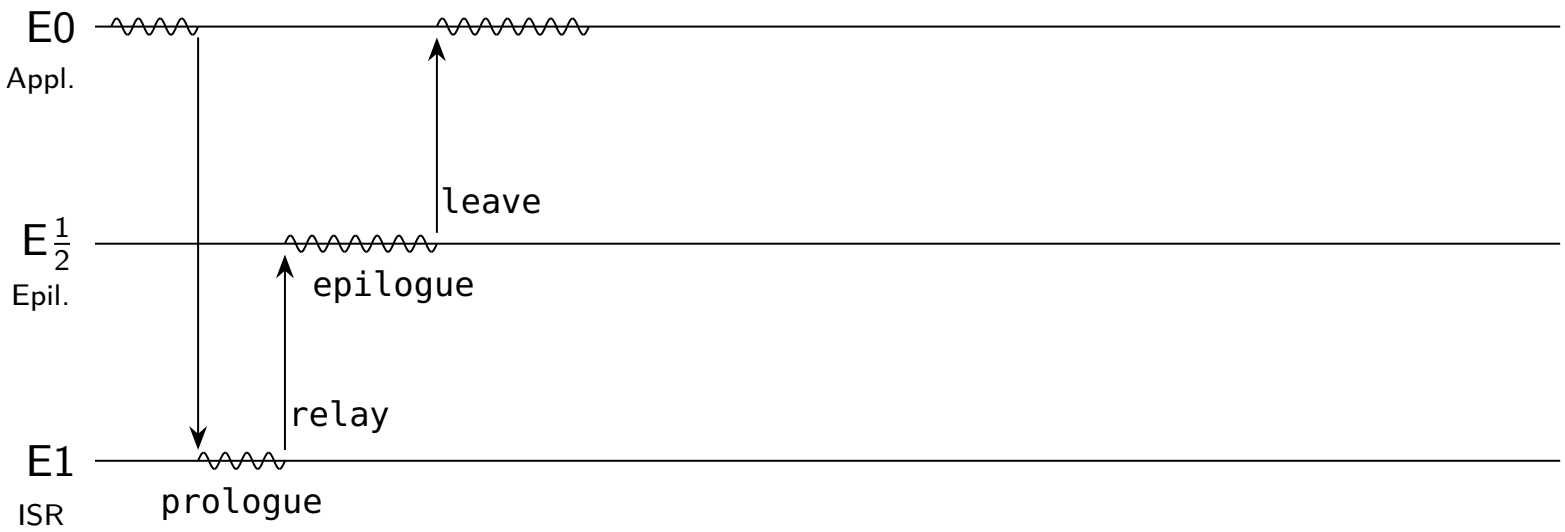
- Interrupt: Wechsel auf E1
- `interrupt_handler` startet Prolog

Pro-/Epilogmodell



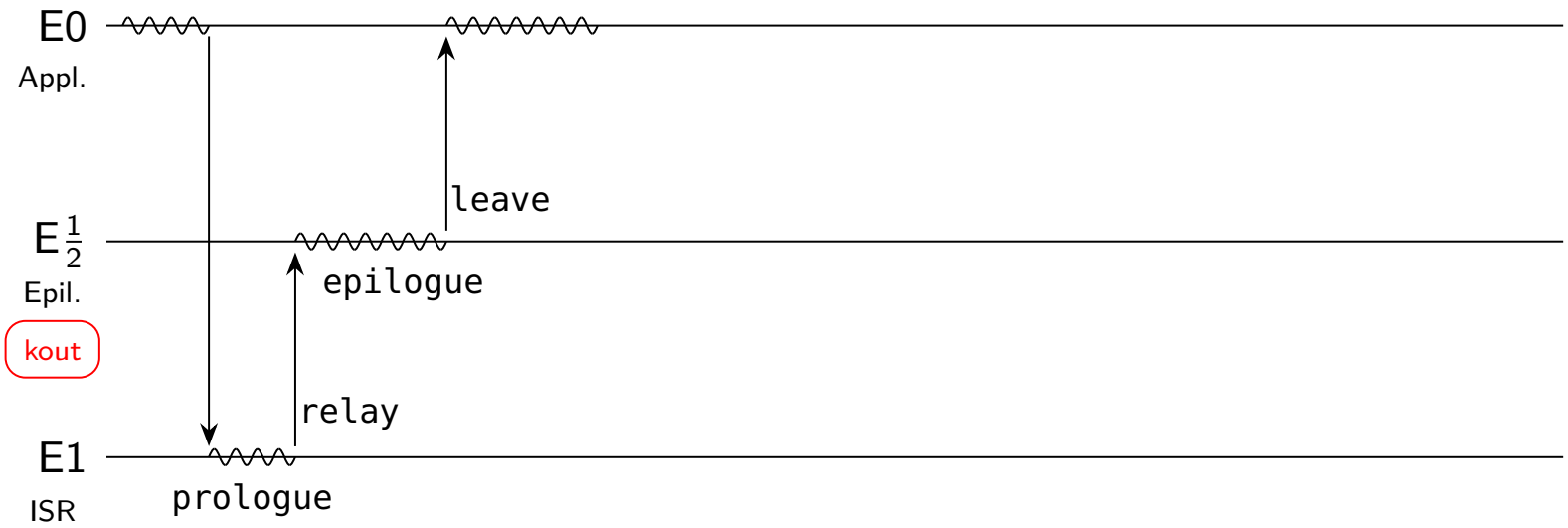
- Prolog gibt boolean zurück
- 0: Rückkehr zu E0
- 1: Epilog angefordert, `relay`

Pro-/Epilogmodell



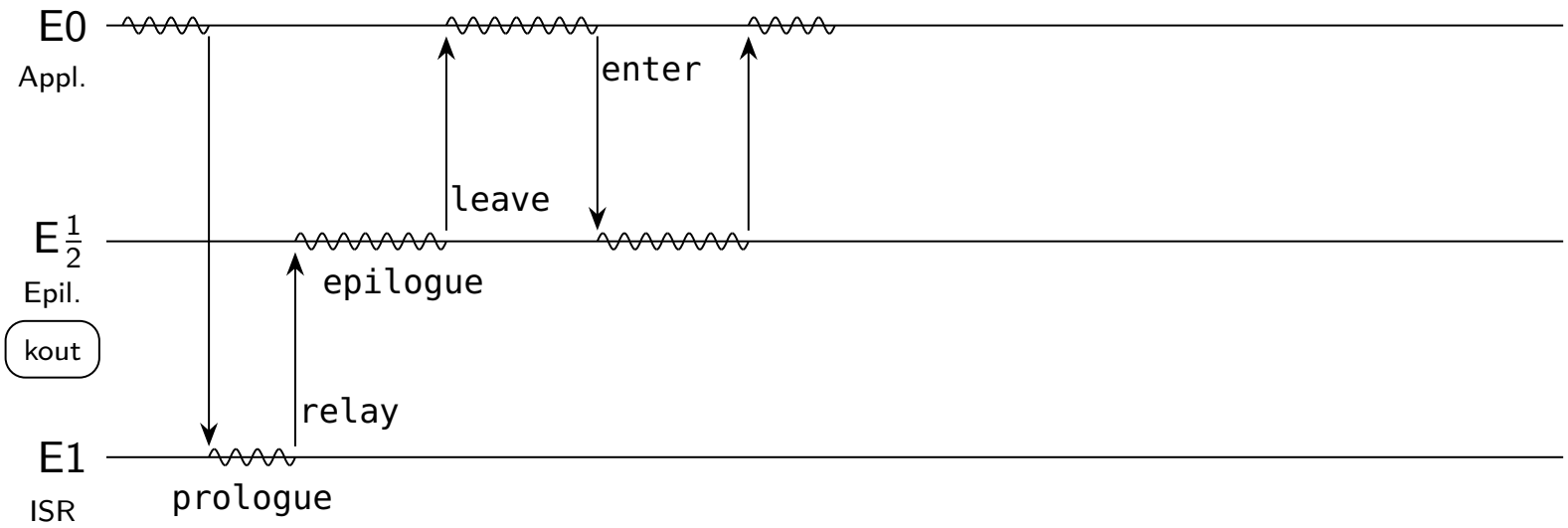
- Epilog kehrt mit `leave` zu E0 zurück

Pro-/Epilogmodell



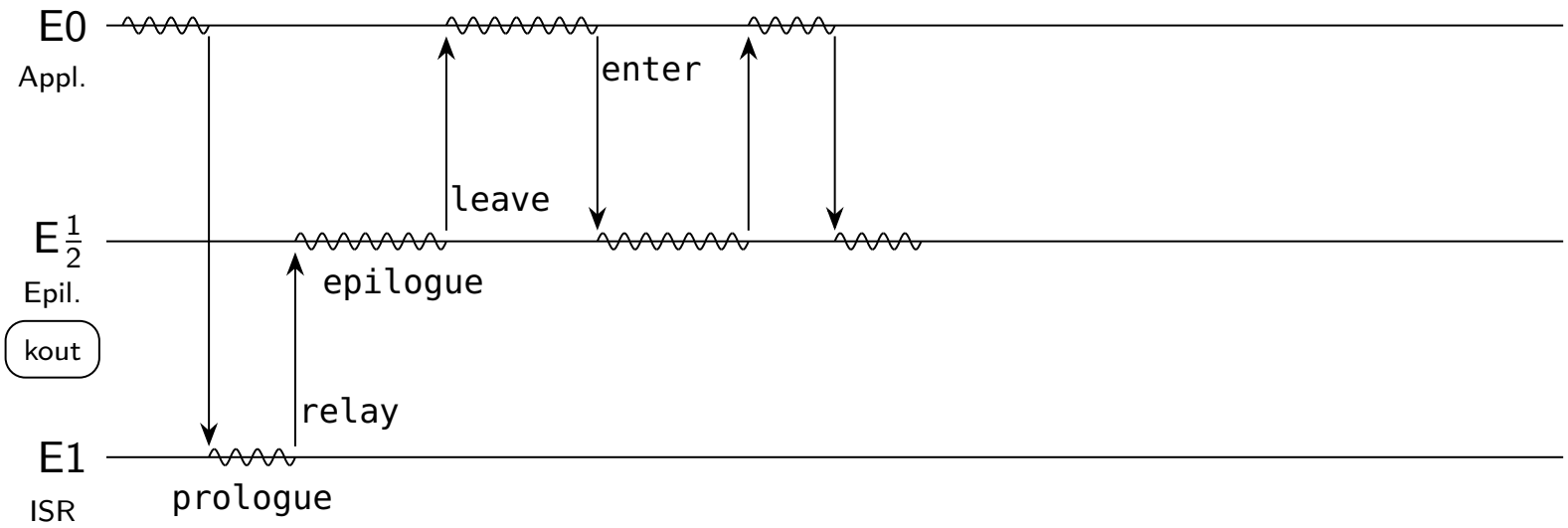
- Kritischen Abschnitt auf $E\frac{1}{2}$ verschieben
- `kout` nur noch auf Epilogebene
- Von E0 zu $E\frac{1}{2}$ mit `enter`

Pro-/Epilogmodell



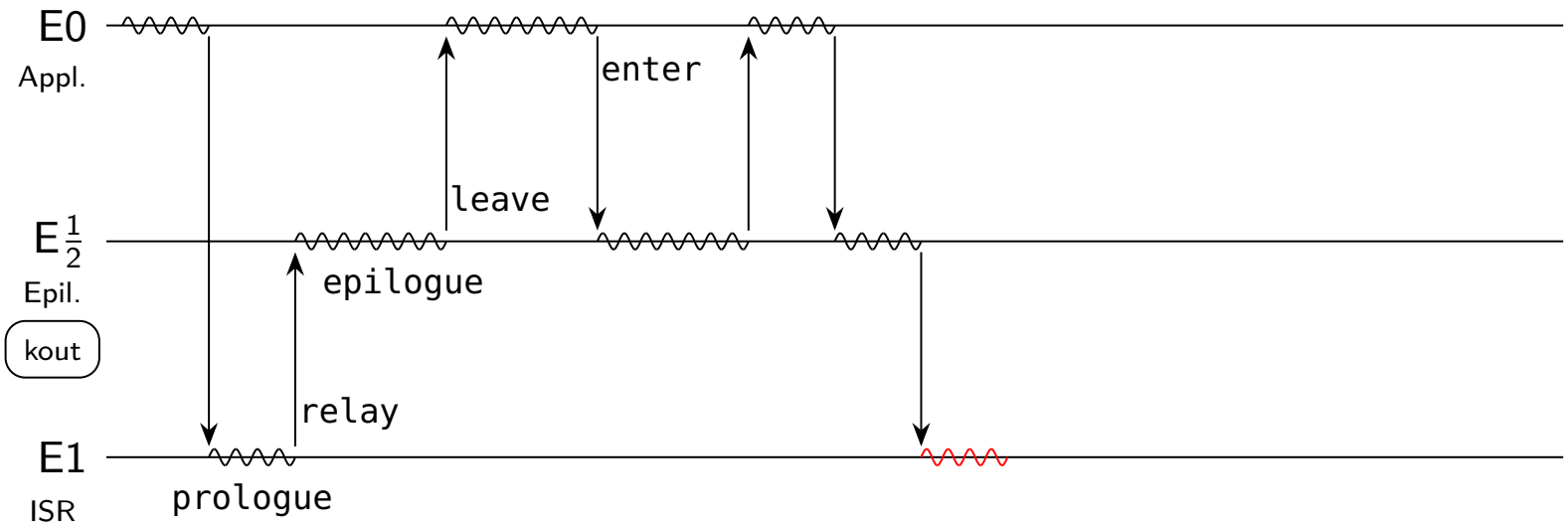
- Rückkehr wieder mit `leave`

Pro-/Epilogmodell



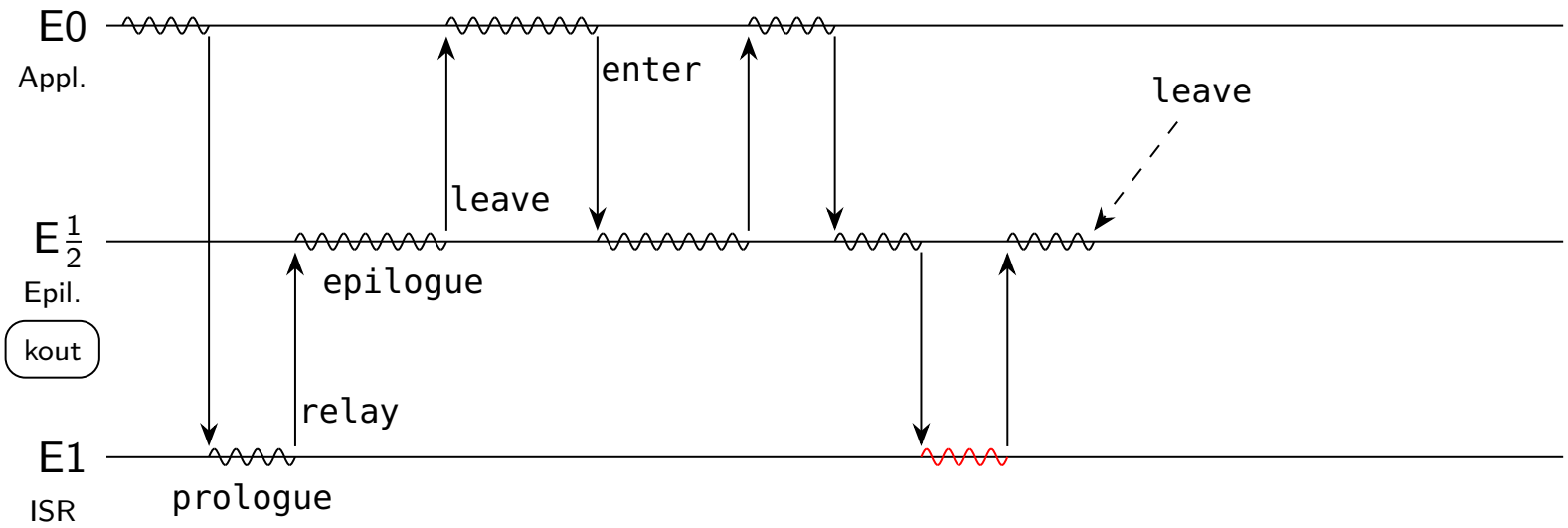
- Auf $E\frac{1}{2}$ kann Interrupt auftreten
- Laufender kritischer Abschnitt unterbrochen

Pro-/Epilogmodell



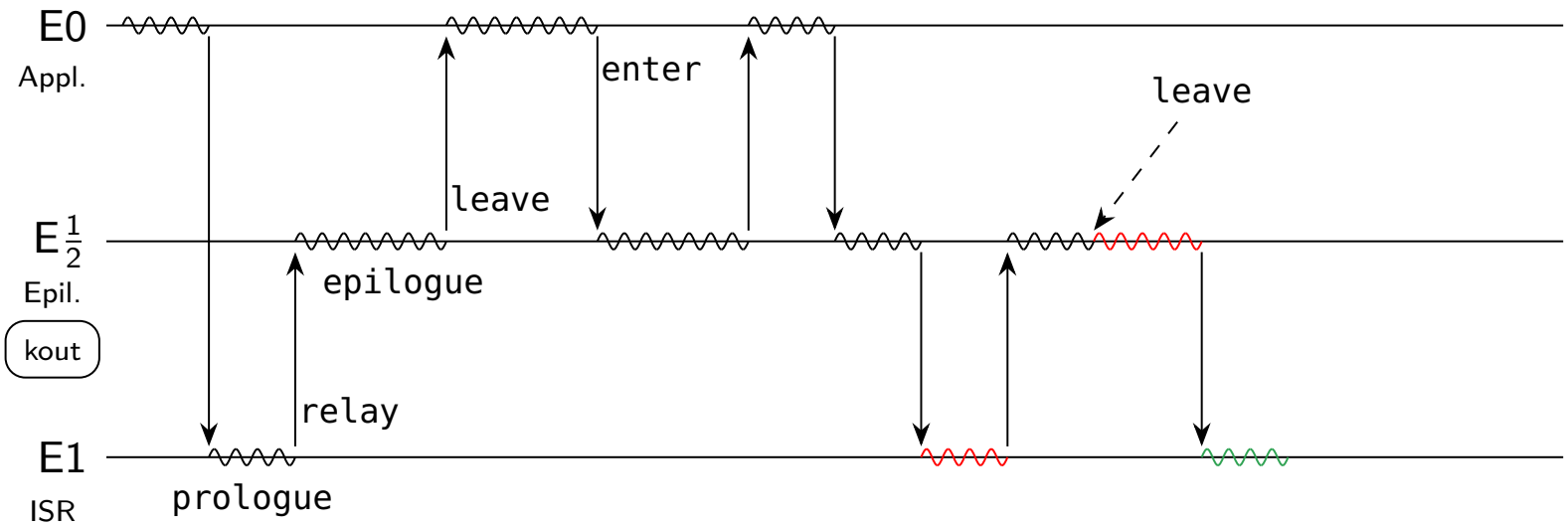
- Ausführung des neuen Prologs

Pro-/Epilogmodell



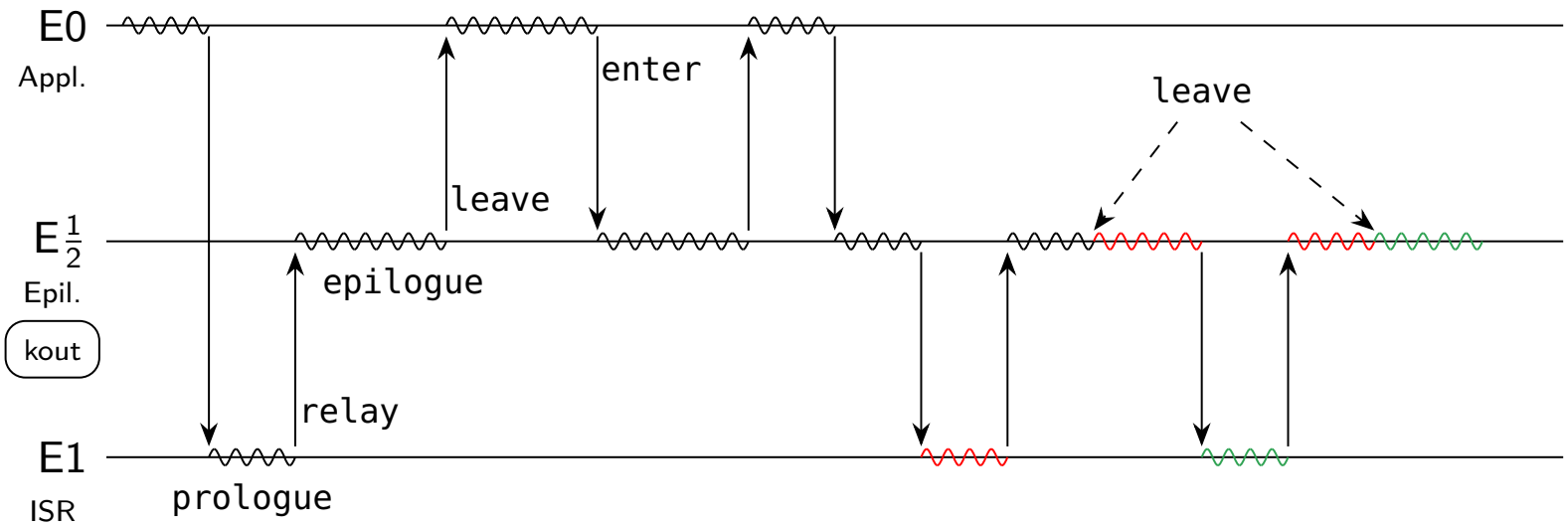
- Unterbrochene Ausführung wird fortgesetzt
- run-to-completion, greedy
- Danach `leave`
- Prüft ob noch Epiloge da sind und führt sie aus

Pro-/Epilogmodell



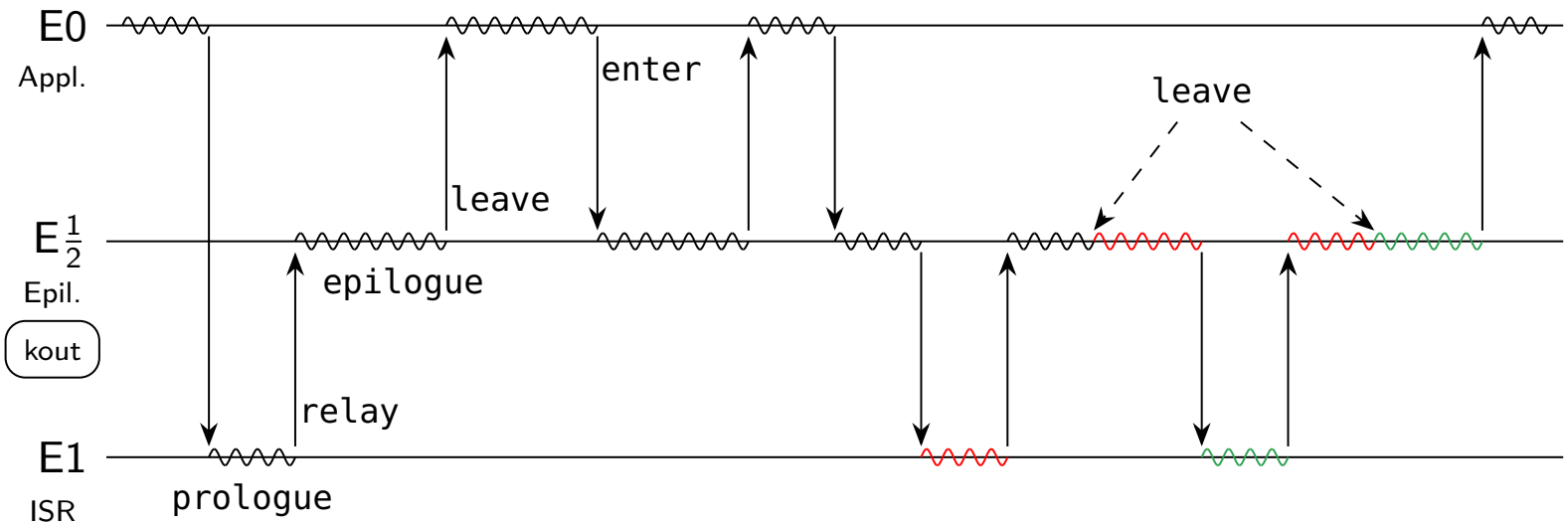
- Unterbrochene Ausführung wird fortgesetzt
- run-to-completion, greedy
- Danach `leave`
- Prüft ob noch Epiloge da sind und führt sie aus

Pro-/Epilogmodell



- Unterbrochene Ausführung wird fortgesetzt
- run-to-completion, greedy
- Danach `leave`
- Prüft ob noch Epiloge da sind und führt sie aus

Pro-/Epilogmodell



- Wenn alle Epiloge fertig sind: Rückkehr zu E0

Die `Guard`-Klasse

Wir brauchen Funktionen:

- `relay`: Von Prolog zu Epilog
- `enter`: Von E0 zu E^1_2
- `leave`: Verbleibende Epiloge ausführen, zurück zu E0

Und Datenstrukturen:

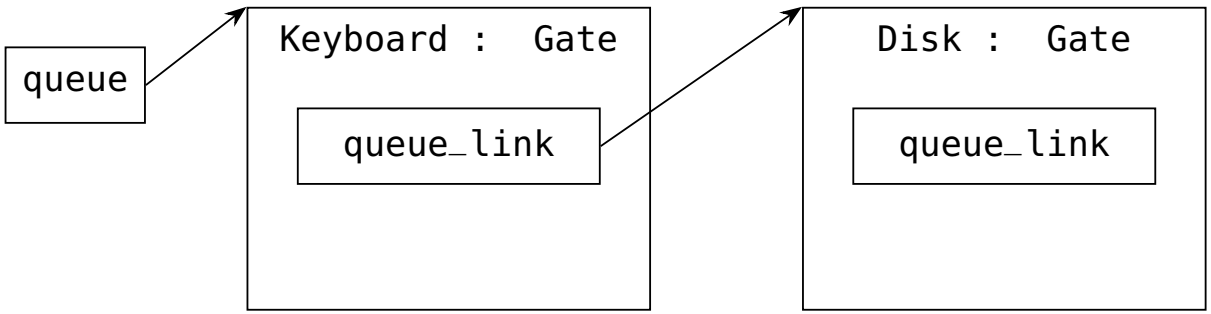
- Möglichkeit Epiloge zu speichern:
  Verkettete Liste für Gate-Objekte (pro CPU)
- Unterscheiden ob wir auf E0 oder E^1_2 unterbrochen wurden:
  1 bit E^1_2-Anzeige (pro CPU)
- (Spin-/Ticket-)Lock, da nur eine CPU gleichzeitig auf E^1_2 sein darf

Verkettete Liste

TUHH
Technische Universität Hamburg

```
Queue<Gate> queue;
queue.enqueue(&Keyboard);
```

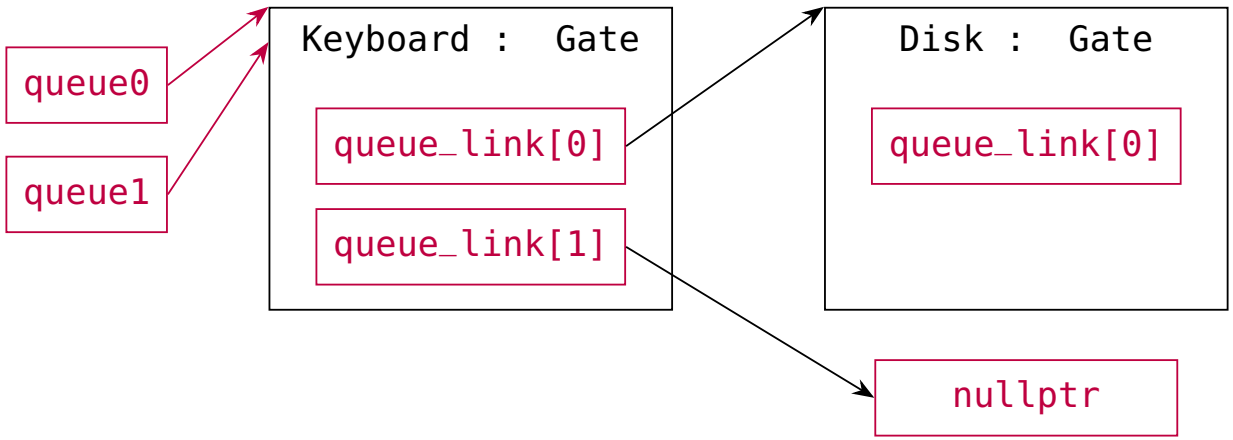
- Templated einfach verkettete Liste
- `next`-Pointer in den Gate-Objekten (kein zusätzliches `malloc`)
- "Magisches Element": `Gate* queue_link`
- Jedes Element kann also nur ein mal in einer Liste sein



Verkettete Liste

```
Queue<Gate> queue0(0), queue1(1); // MPStuBS
queues[Core::getID()].enqueue(&Keyboard);
```

- Templated einfach verkettete Liste
- `next`-Pointer in den Gate-Objekten (kein zusätzliches `malloc`)
- "Magisches Element": `Gate* queue_link`
- Jedes Element kann also nur ein mal in einer Liste sein

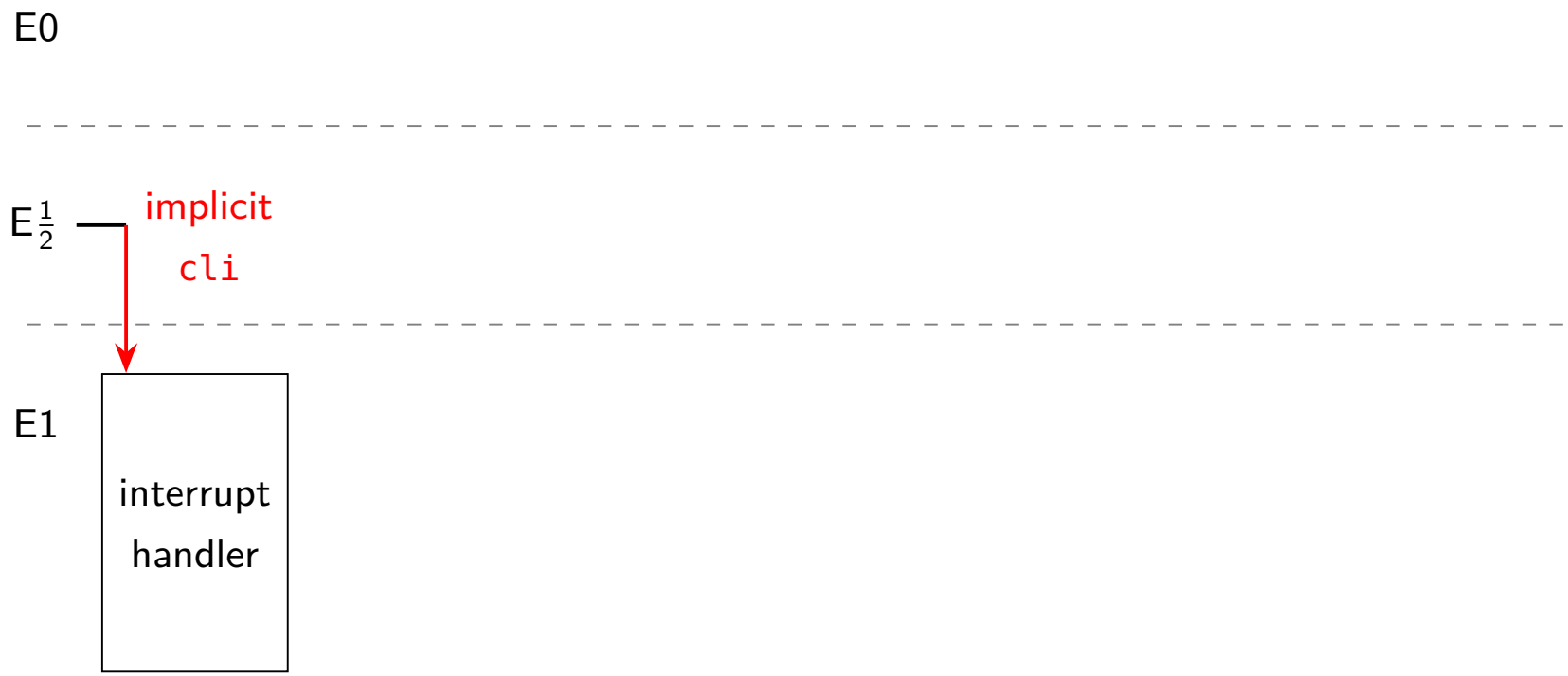


Funktionen von Guard

Ablauf von Prolog und Epilog in StuBS

- Auf E0 und $E\frac{1}{2}$ sind Interrupts aktiv
- Hier: Interrupt auf $E\frac{1}{2}$

Ablauf von Prolog und Epilog in StuBS

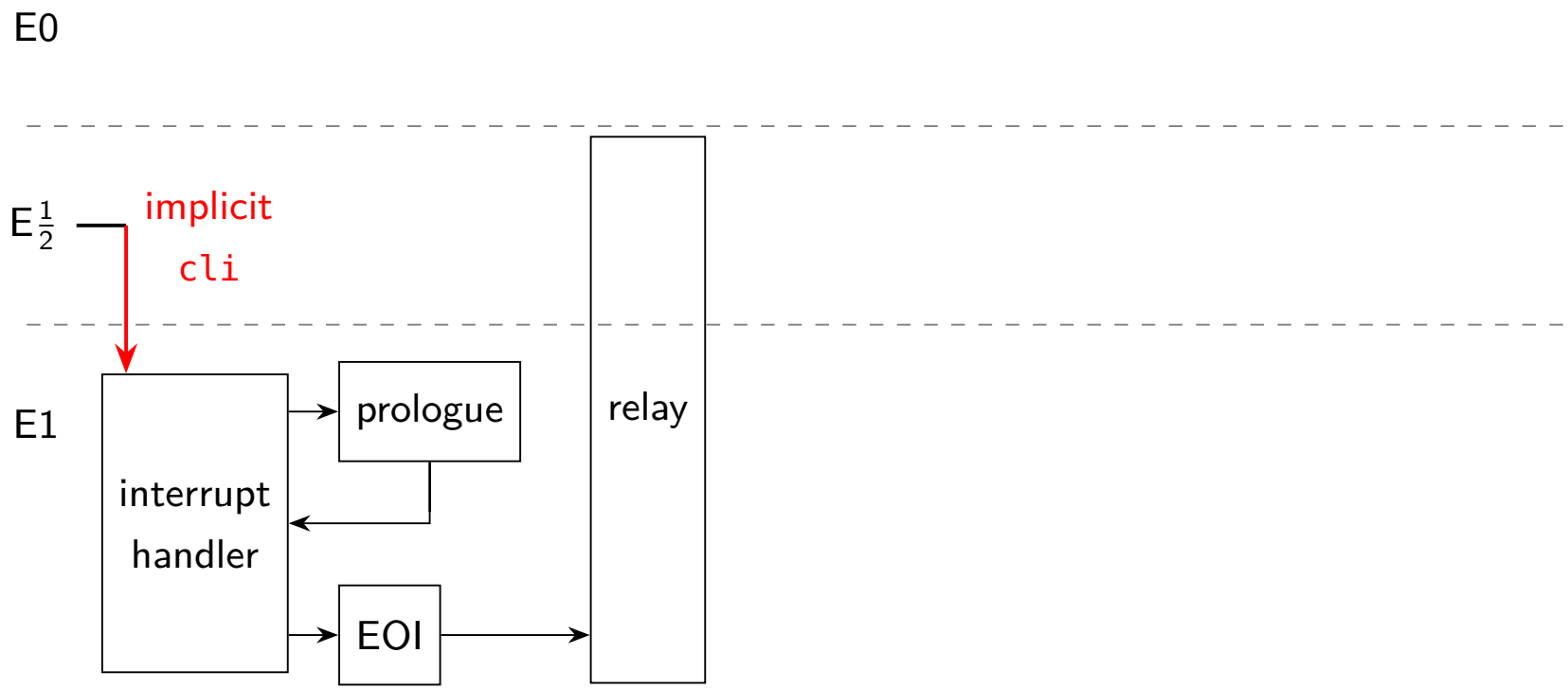


- Keine Interrupts auf E1

Ablauf von Prolog und Epilog in StuBS

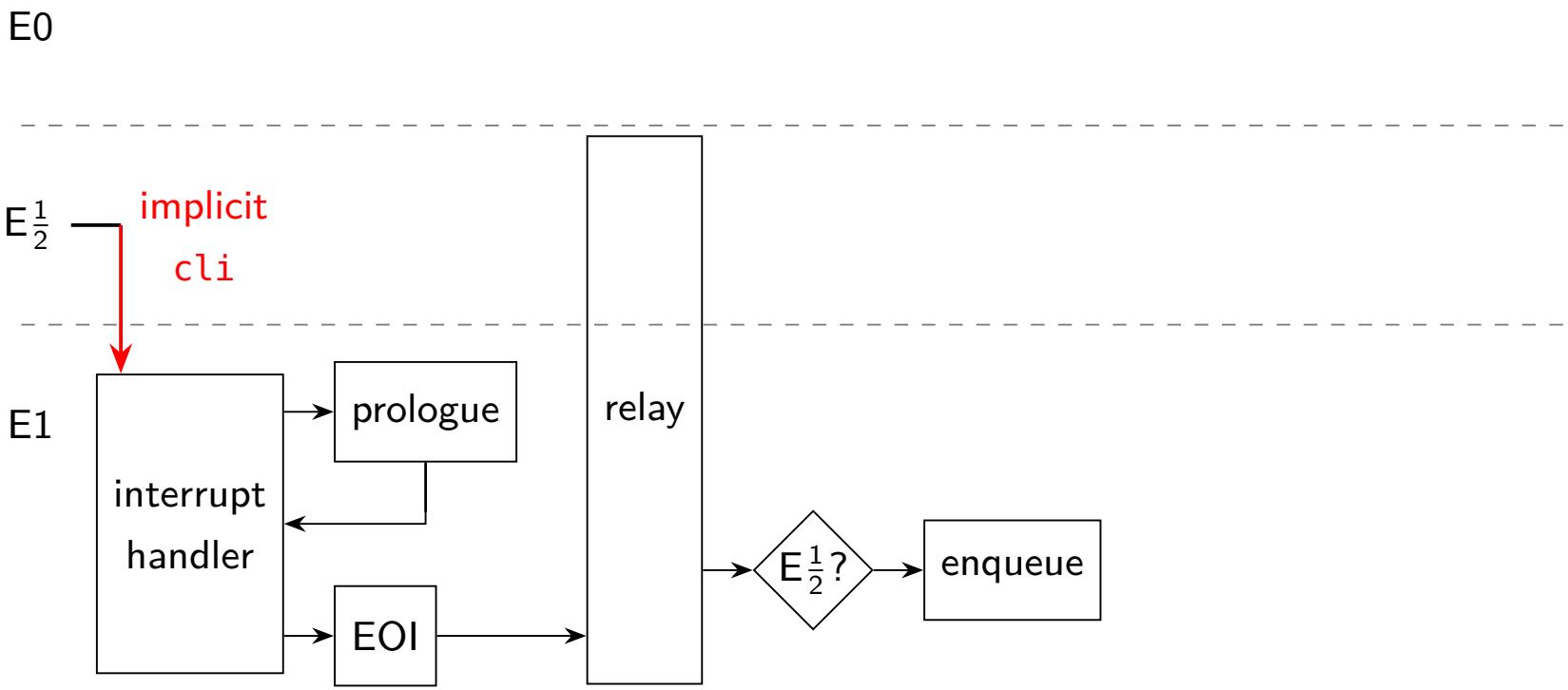
- Handler ruft Prolog auf
- Annahme hier: Epilog angefordert

Ablauf von Prolog und Epilog in StuBS



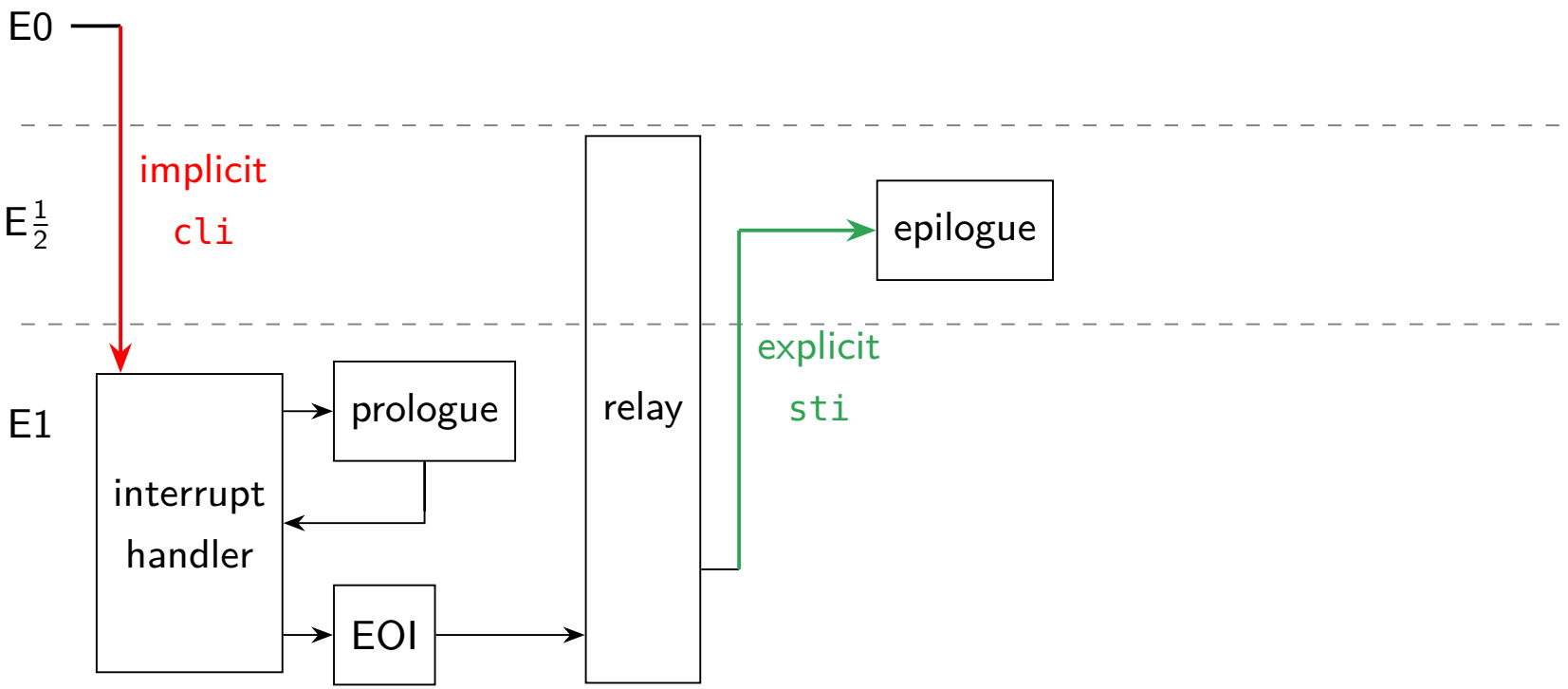
- Kritischer Teil von Interrupt mit Prolog beendet: EOI
- `relay` handelt abhängig von unterbrochener Ebene

Ablauf von Prolog und Epilog in StuBS



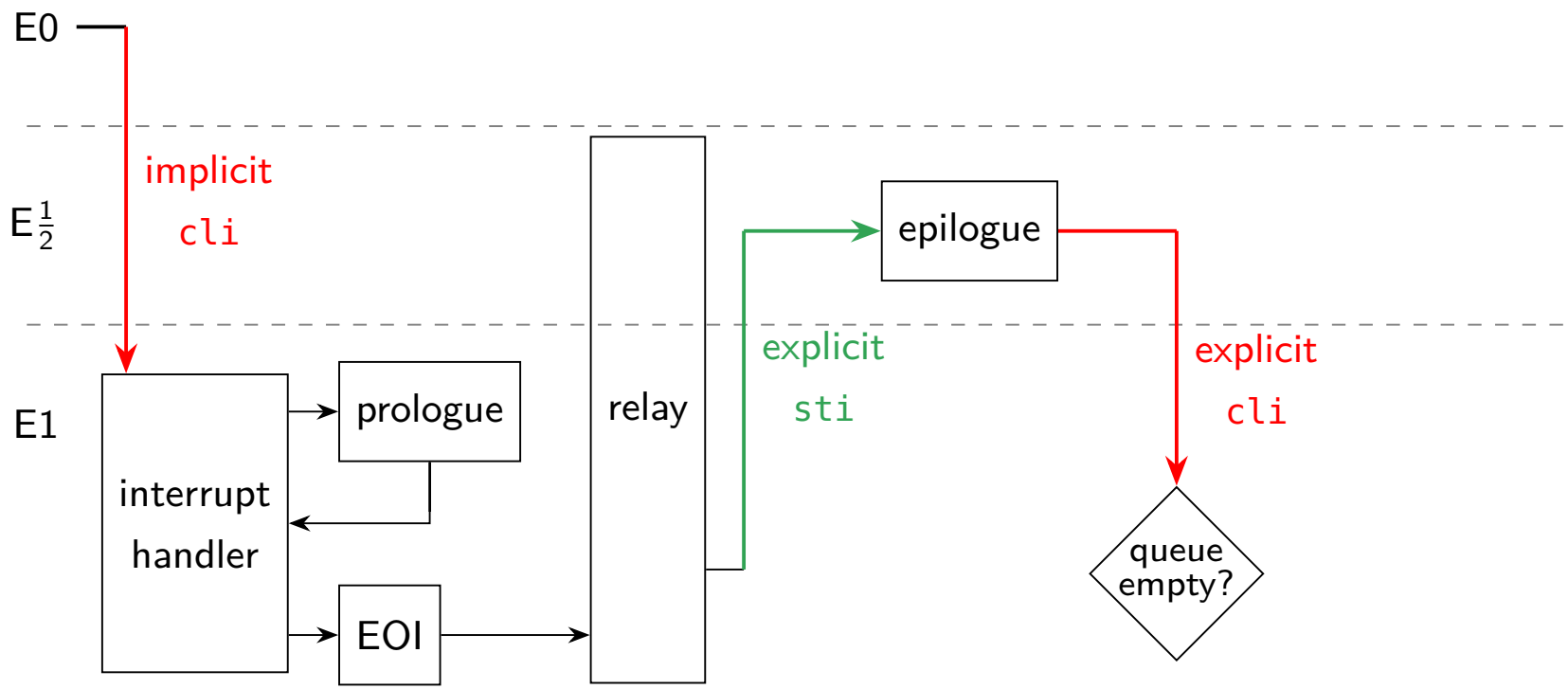
- Wurde $E\frac{1}{2}$ unterbrochen: Epilog in queue einhängen
- Alter Epilog soll erst fortgesetzt werden

Ablauf von Prolog und Epilog in StuBS



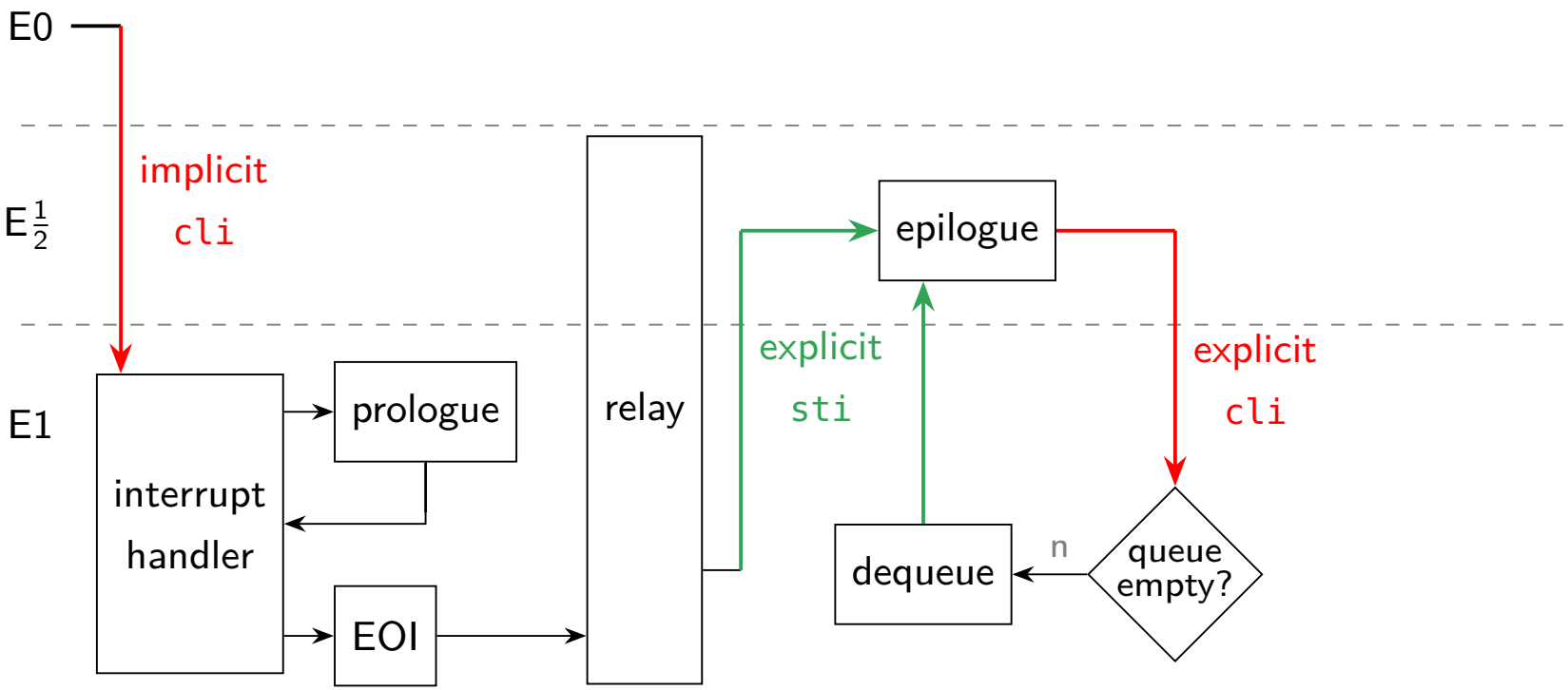
- Wurde E0 unterbrochen: Wechsel auf $E\frac{1}{2}$
- Ausführung des Epilogs

Ablauf von Prolog und Epilog in StuBS



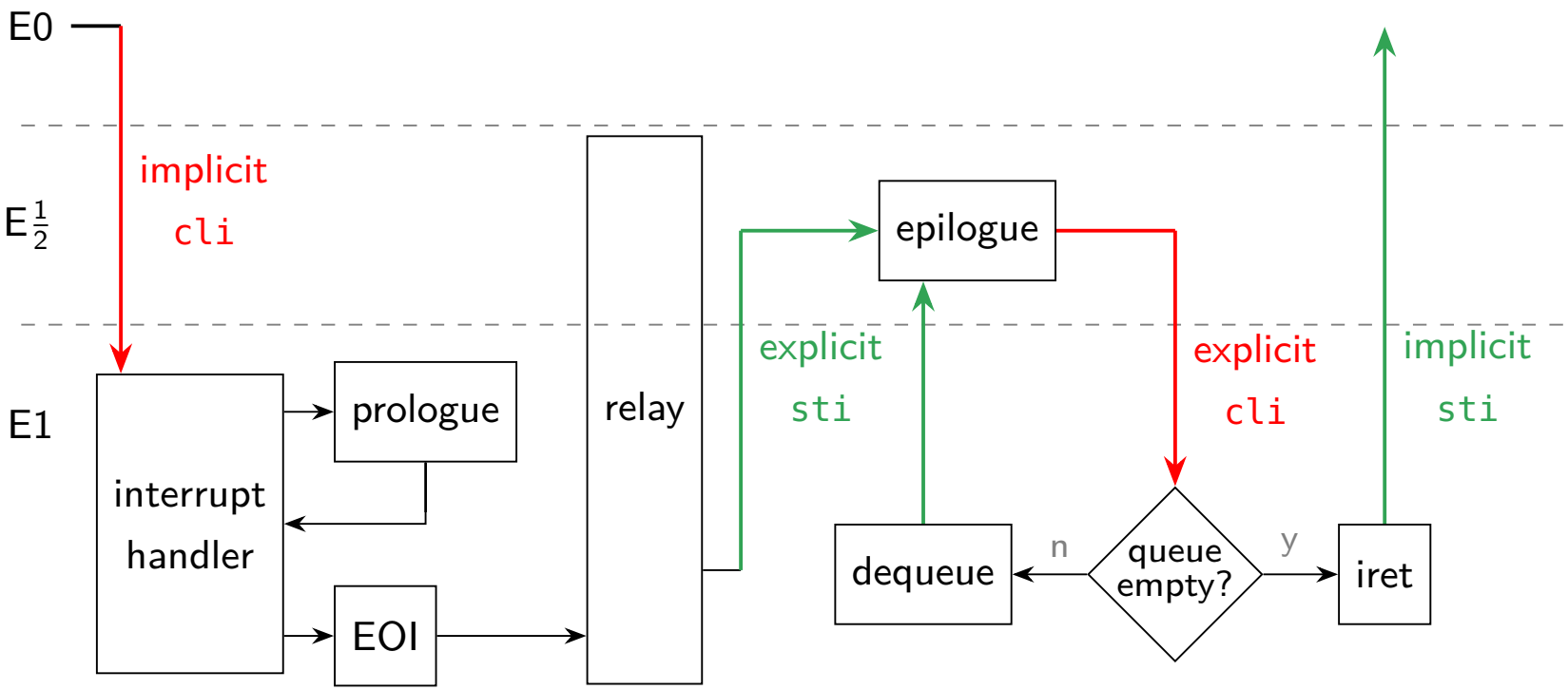
- $E\frac{1}{2}$ kann unterbrochen werden
- Zwischendurch können Interrupts kommen
- Queue prüfen

Ablauf von Prolog und Epilog in StuBS



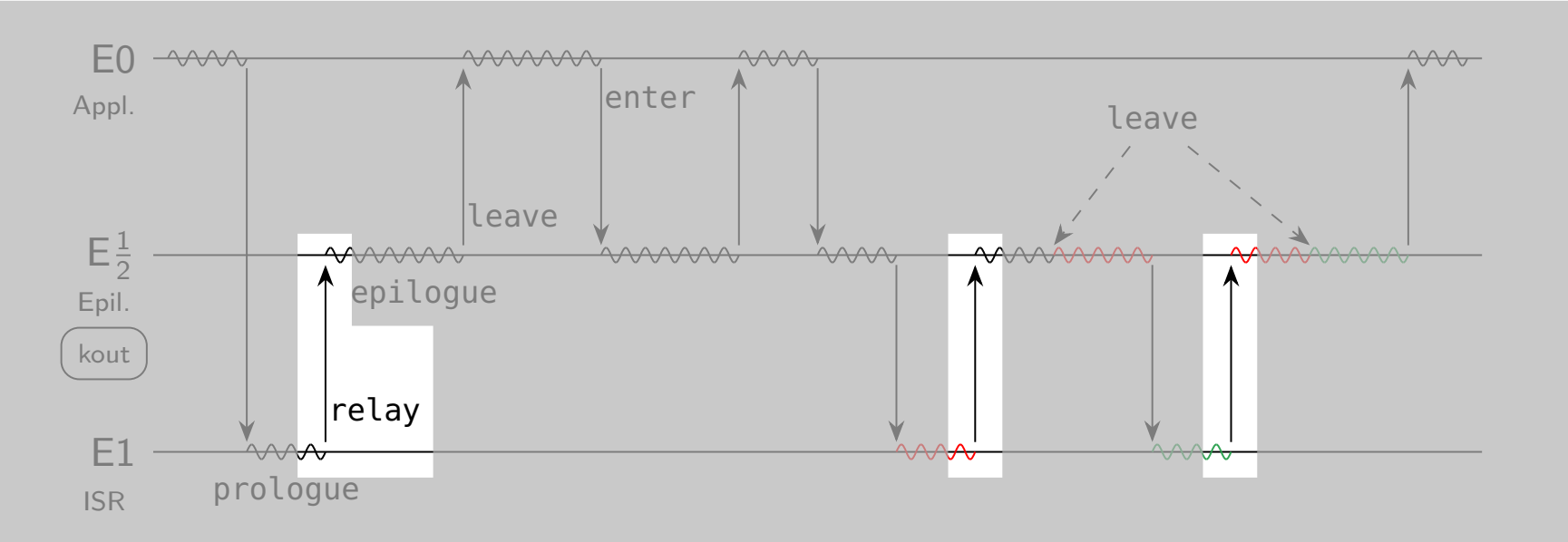
- Epiloge aus queue in Schleife ausführen

Ablauf von Prolog und Epilog in StuBS



- Queue leer: Wechsel zurück auf E0

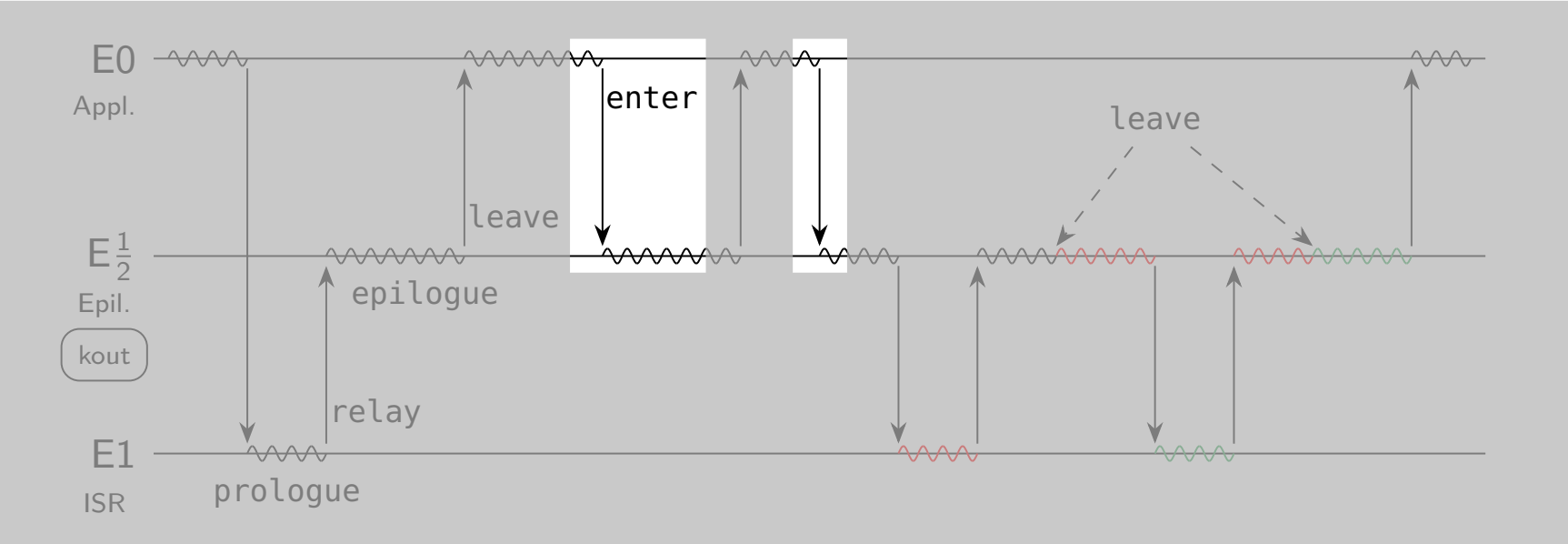
Guard::relay(Gate* g)



Ebene 1 hat Arbeit für Ebene $\frac{1}{2}$

- Je nach unterbrochener Ebene
- E0: Ebene $\frac{1}{2}$ betreten, Epilog ausführen
- E$\frac{1}{2}$: Epilog in queue einhängen

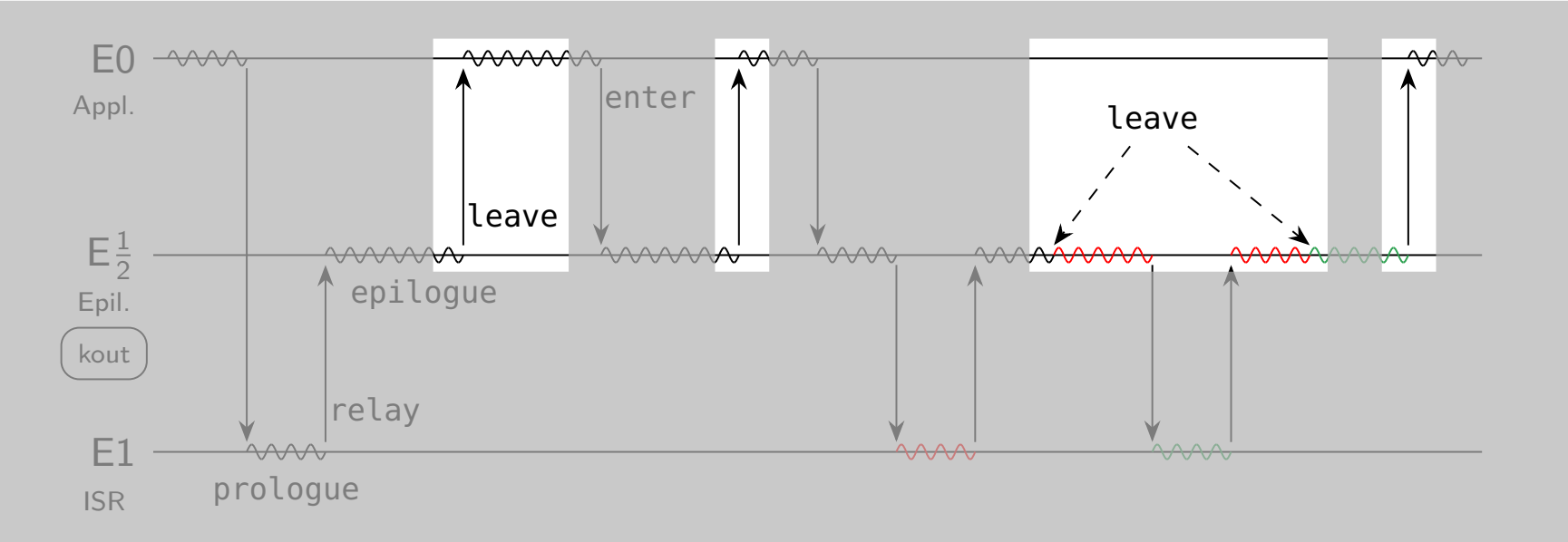
Guard::enter()



Wechsel von Ebene 0 auf Ebene $\frac{1}{2}$

- $E\frac{1}{2}$-Anzeige setzen (auf der aktuellen CPU)
- Lock nehmen

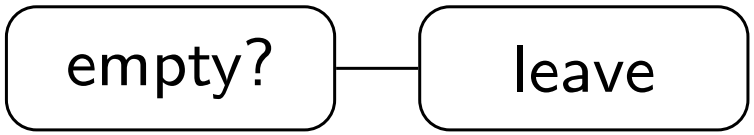
Guard::leave()



Wechsel von Ebene $\frac{1}{2}$ auf Ebene 0

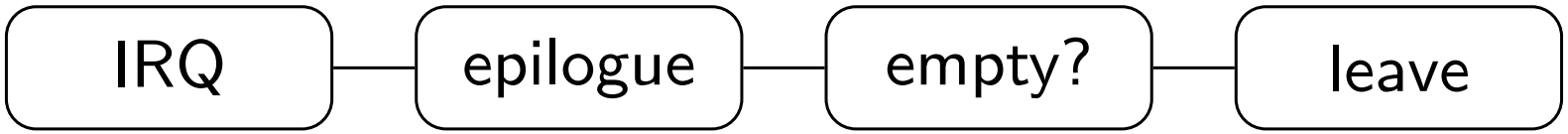
- Epiloge aus der Liste ausführen bis diese leer ist
- E$\frac{1}{2}$-Anzeige löschen (auf der aktuellen CPU)
- Lock freigeben

Problem bei leave



- Wir prüfen Bedingung und handeln *danach* abhängig vom Wert

Problem bei leave



- Wir prüfen Bedingung und handeln *danach* abhängig vom Wert
- Bedingung kann sich dazwischen ändern
- Dann könnte neuer Epilog verloren gehen

Lost-Wakeup-Problem

Kann bei Interrupts immer auftreten.
Check und leave muss atomar sein (z.B. Harte Synchronisation)

Problem bei leave

- Wir prüfen Bedingung und handeln *danach* abhängig vom Wert
- Bedingung kann sich dazwischen ändern
- Dann könnte neuer Epilog verloren gehen

Lost-Wakeup-Problem

Kann bei Interrupts immer auftreten.
Check und leave muss atomar sein (z.B. Harte Synchronisation)

Zusatz: Guarded-Objekt

TUHH
Technische Universität Hamburg

- `leave` am Ende nicht vergessen
- `Guarded` kann das erleichtern
- Ruft `enter` und `leave` in Konstruktor und Destruktor auf

```
{
   enter();
   // Code
   leave()
}
```

Zusatz: `Guarded`-Objekt

TUHH
Technische Universität Hamburg

- `leave` am Ende nicht vergessen
- `Guarded` kann das erleichtern
- Ruft `enter` und `leave` in Konstruktor und Destruktor auf
- Resource Acquisition is Initialization

```
{
  Guarded g;
  // Code
} // Impliziter Destruktor-Aufruf
```

Aufgabe 3

TUHH
Technische Universität Hamburg

- Gate-Objekte: `prologue` und `epilogue` statt `trigger`
- Interrupt-Handler anpassen
- Guard und Guarded implementieren
- Test-Anwendung von A2 nur noch mit `Guarded` schützen

Achtung

Wenig Code, dafür sehr fehleranfällig.