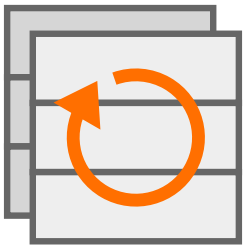


Operating System Group

TUHH

Technische Universität Hamburg

BSB – Übung 5: Zeitscheibenscheduling

Yannick Loeck

2022-06-15

Inhalt

TUHH
Technische Universität Hamburg

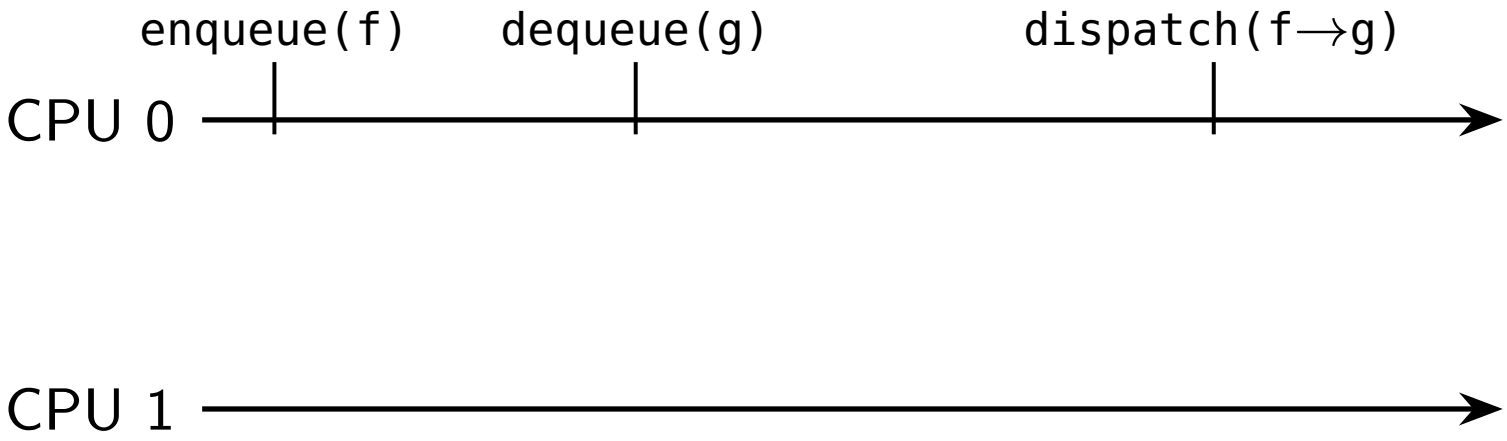
BSB - HÜ5

Scheduling ohne Kooperation der Threads.

Worum geht es in der Übung heute:

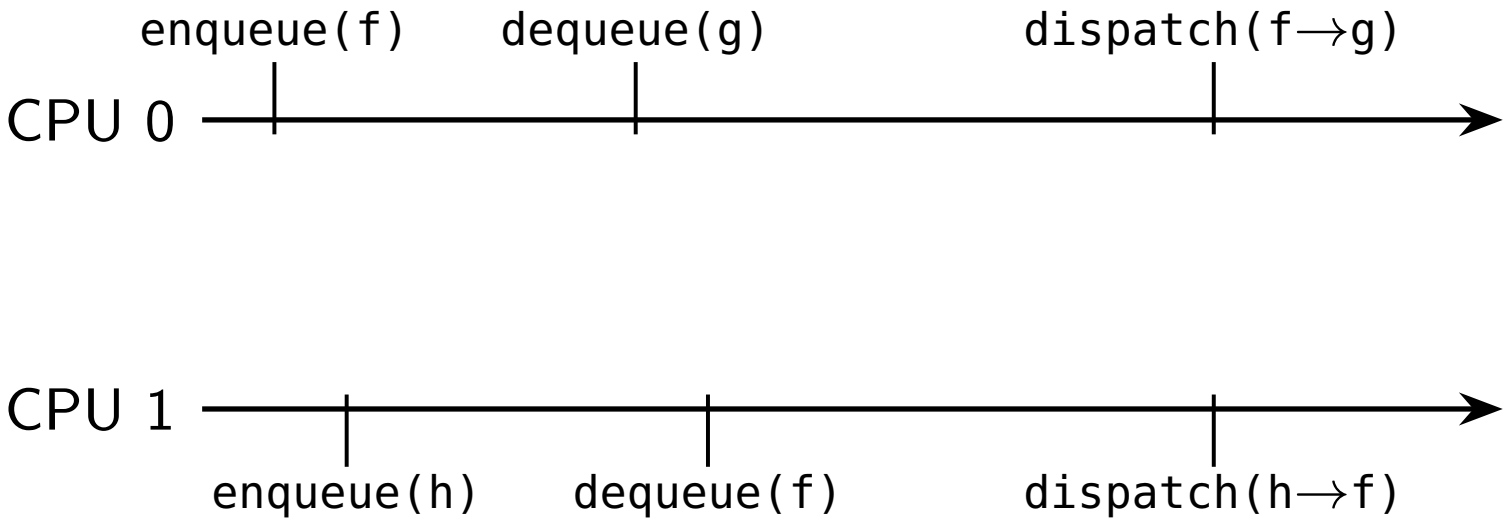
- Konsistenzprobleme beim Scheduling
- Timer-Konfiguration
- Beenden von Threads mit IPIs

Konsistenzprobleme bei `resume()`



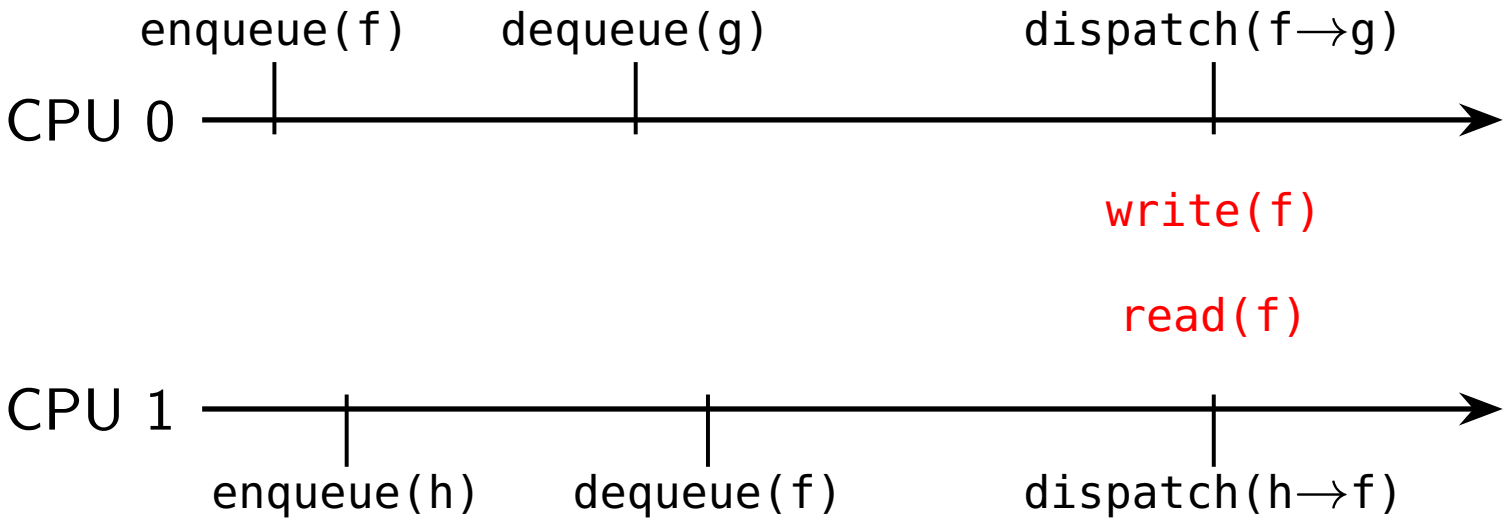
- Schützen der ready-Liste reicht nicht immer aus

Konsistenzprobleme bei `resume()`



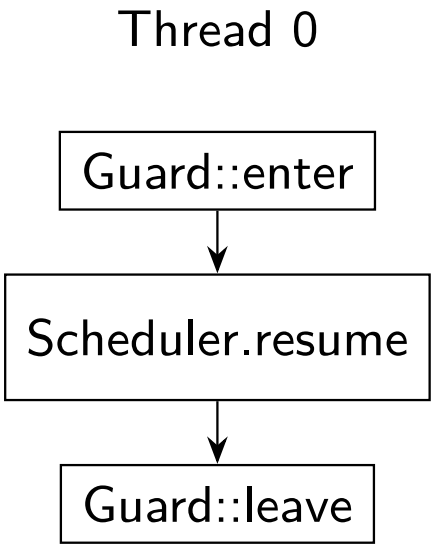
- Schützen der ready-Liste reicht nicht immer aus
- Mehrere CPUs in `resume()`: gemeinsame Daten
- Können gleichzeitig Kontext lesen und schreiben
- Einfache Lösung: ganzes `resume()` `Guard`-en

Konsistenzprobleme bei `resume()`



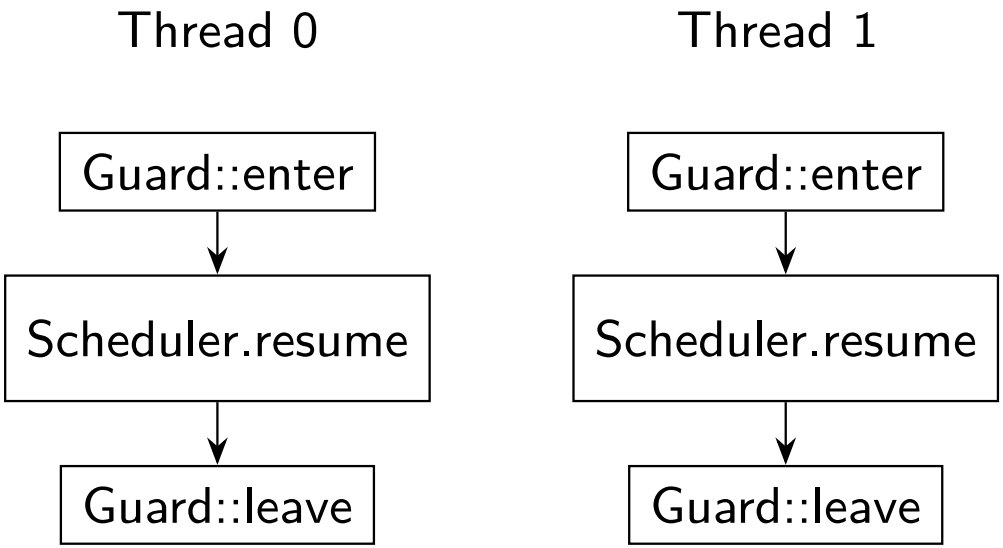
- Schützen der ready-Liste reicht nicht immer aus
- Mehrere CPUs in `resume()`: gemeinsame Daten
- Können gleichzeitig Kontext lesen und schreiben
- Einfache Lösung: ganzes `resume()` `Guard`-en

Konsistenzprobleme bei `resume()`



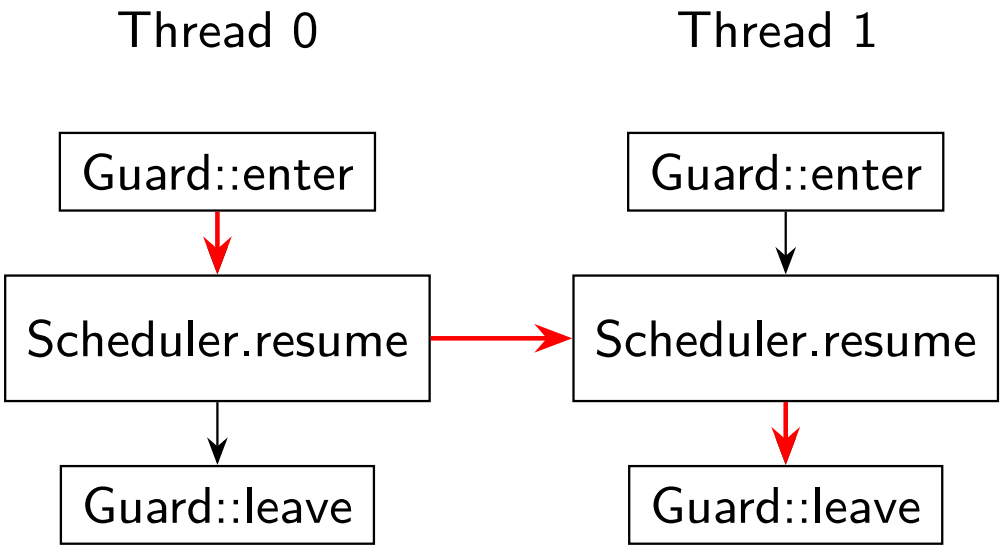
- Scheinbar: enter() → resume() → leave()
- Mitten drin ist aber ein Kontrollflusswechsel

Konsistenzprobleme bei `resume()`



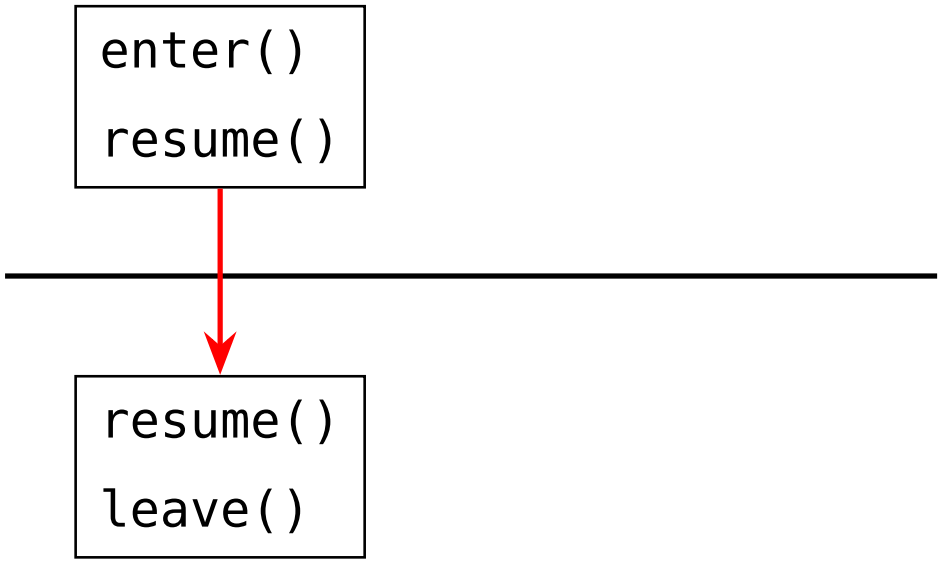
- Scheinbar: enter() → resume() → leave()
- Mitten drin ist aber ein Kontrollflusswechsel
- Wo landen wir auf der anderen Seite?

Konsistenzprobleme bei `resume()`

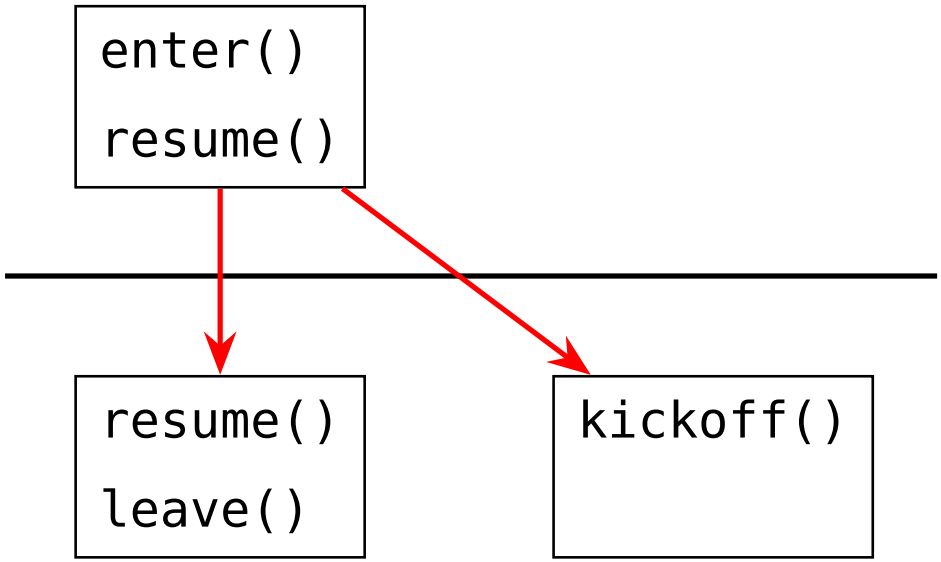


- Scheinbar: enter() → resume() → leave()
- Mitten drin ist aber ein Kontrollflusswechsel
- Wo landen wir auf der anderen Seite?

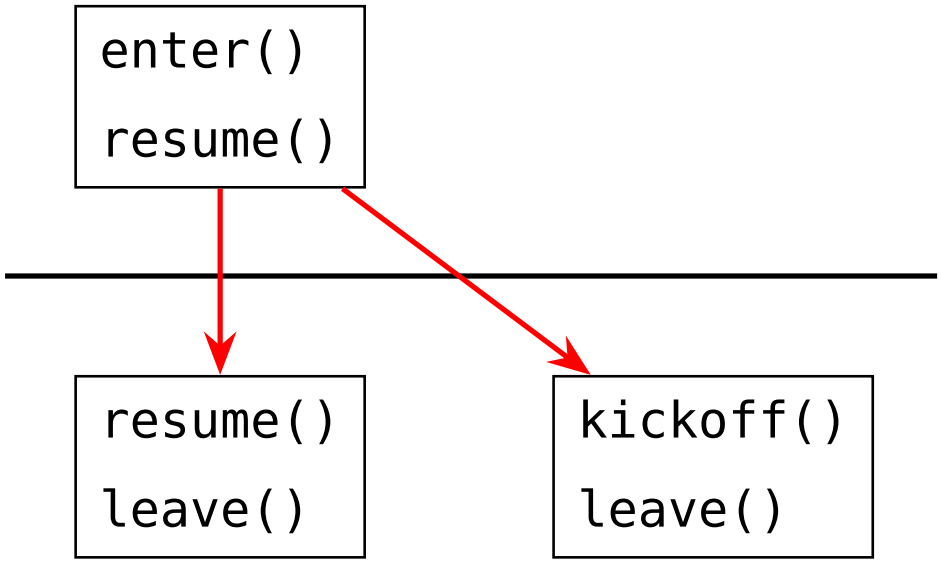
Konsistenzprobleme bei `resume()`



Konsistenzprobleme bei `resume()`

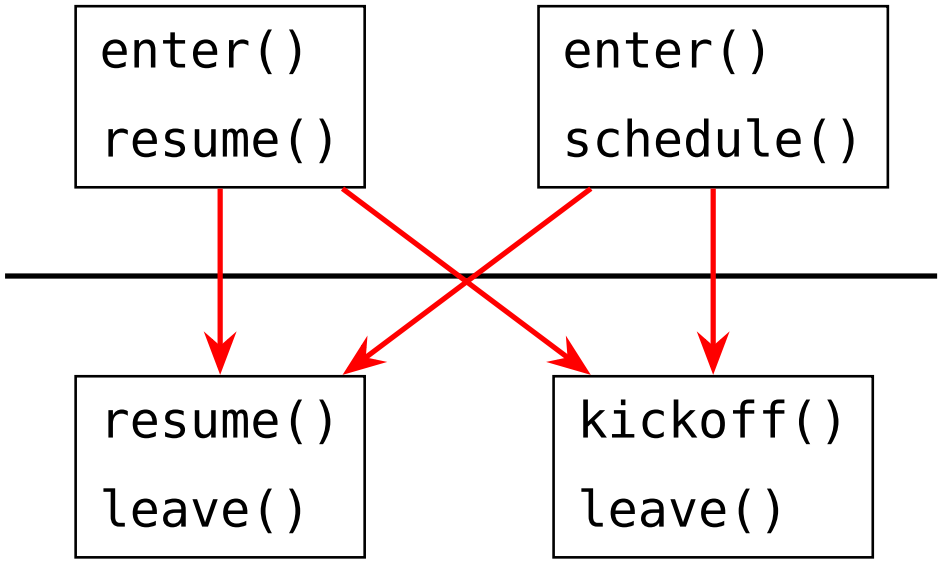


Konsistenzprobleme bei `resume()`



- Alle Kombinationen können auftreten
- Wichtig: `enter()` und `leave()` immer paarweise!

Konsistenzprobleme bei `resume()`



- Alle Kombinationen können auftreten
- **Wichtig:** `enter()` und `leave()` immer paarweise!

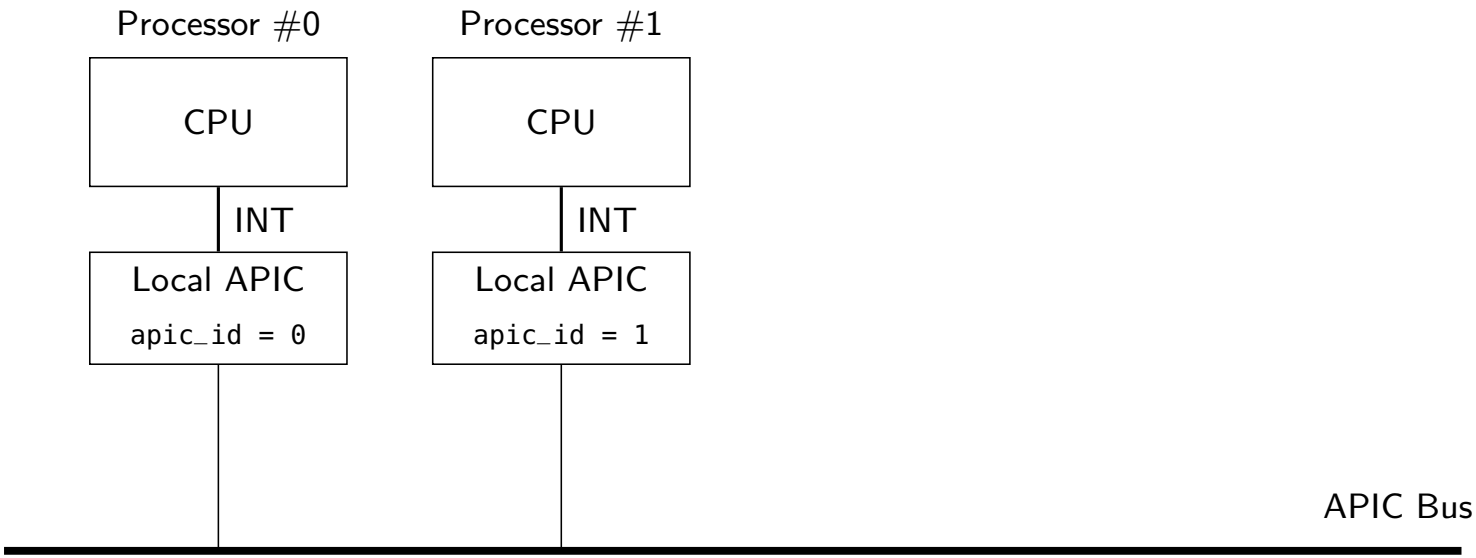
Scheduling

TUHH
Technische Universität Hamburg

- Threads rufen Scheduler selbst auf
- *Was, wenn nicht?*
- System soll periodisch reschedulen $\Rightarrow$ Timer
- CPU wird dem Thread weggenommen: Preemptive Scheduling

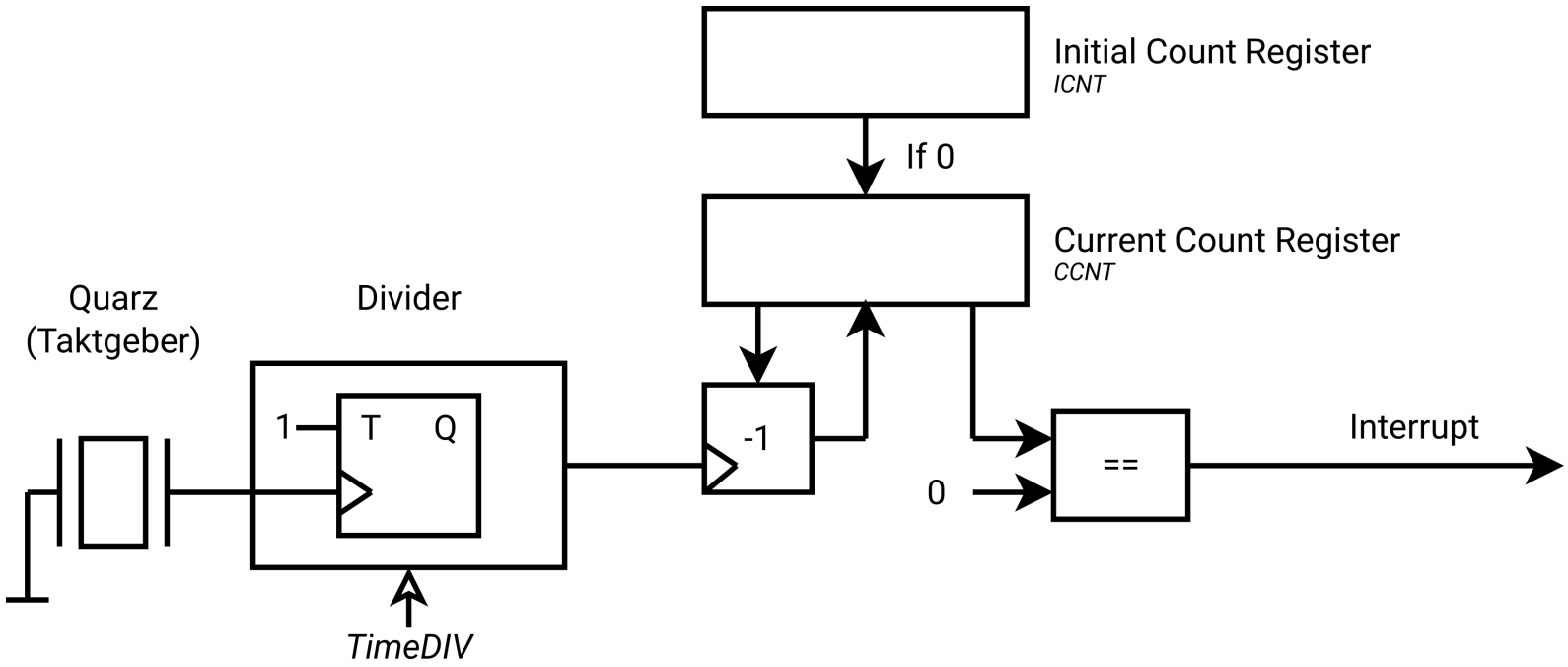
Timer

APIC-Timer



- Timer in LAPIC integriert
- Interrupts füllen nicht den Bus
- Alle *n* ticks ein Interrupt an die CPU
- Timer-Intervall konfigurieren (in StuBS in `windup(us)`)
- Timer an jeder CPU aktivieren (`activate()`)

APIC-Timer - Aufbau



- Zählt in 32-bit Register (CCNT) runter
- Bei 0: Interrupt, reset auf Wert aus ICNT
- Basis-Clock von außen: APIC Bus Clock
- Überlauf verhindern: Takt wird über Timerdiv geteilt

APIC-Timer in StuBS

Konfiguration über 4 memory-mapped Register:

- `TIMER_CONTROL`: Betriebsmodus, Interrupt-Vektor
- `TIMER_INITIAL_COUNTER`: Initialwert schreiben, Timer stoppt bei 0
- `TIMER_CURRENT_COUNTER`: Aktuellen Wert auslesen
- `TIMER_DIVIDE_CONFIGURATION`: Taktteilung

Achtung

Reservierte bits im Kontrollregister!

Problem 1: Timerdiv einstellen

TUHH
Technische Universität Hamburg

Wir können:

- Taktteiler einstellen
- Timer-Resetwert einstellen
- Timer neu starten

- Überlauf der Register verhindern
- Höherer Timerdiv: Präzisionsverlust

- Ziel: Maximale Präzision für entsprechendes Timer-Intervall

Problem 2: Was ist überhaupt Zeit?

TUHH
Technische Universität Hamburg

- Wir arbeiten mit der APIC Bus Clock
  - Die kennen wir nicht!
  - Kann sogar bei jedem Boot anders sein

Problem 2: Was ist überhaupt Zeit?

TUHH
Technische Universität Hamburg

- Wir arbeiten mit der APIC Bus Clock
  - Die kennen wir nicht!
  - Kann sogar bei jedem Boot anders sein

- Uhr deren Takt wir kennen: **P**rogrammable **I**nterval **T**imer
- Messung zur Umrechnung von Echtzeit zu APIC-Timer-Zeit
  1. PIT feste Zeitspanne laufen lassen, aktiv auf Ablauf warten
  2. APIC-Timer währenddessen runterzählen lassen
  3. Differenz durch Zeitspanne: Takt

Assassin

Beenden von Threads

TUHH
Technische Universität Hamburg

- Bisher: Thread ruft selbst `resume()` auf, das prüft `kill_flag`
- Jetzt: `resume()` und Überprüfung bei Timer-Interrupt
- Wir wollen Thread sofort beenden, nicht maximal ΔT warten
- Lösung: Interrupt an andere CPU schicken
- IRQs jetzt nicht mehr nur von externen Geräten

Inter Processor Interrupt

TUHH
Technische Universität Hamburg

- LAPICs kommunizieren über Bus auch untereinander
- In StuBS: `LAPIC::IPI::send(apic_id, vector)`
- Implementieren im `Assassin`

Es bleiben noch 2 Fragen:

1. Warten bis der Thread tatsächlich “tot” ist?

2. Wozu jetzt noch das `kill_flag`?

Inter Processor Interrupt

TUHH
Technische Universität Hamburg

- LAPICs kommunizieren über Bus auch untereinander
- In StuBS: `LAPIC::IPI::send(apic_id, vector)`
- Implementieren im `Assassin`

Es bleiben noch 2 Fragen:

1. Warten bis der Thread tatsächlich "tot" ist?
   Nein, da Scheduler kill und IPI-Epilog beide auf $E_{\frac{1}{2}}$ sind!
2. Wozu jetzt noch das `kill_flag`?

Inter Processor Interrupt

TUHH
Technische Universität Hamburg

- LAPICs kommunizieren über Bus auch untereinander
- In StuBS: `LAPIC::IPI::send(apic_id, vector)`
- Implementieren im `Assassin`

Es bleiben noch 2 Fragen:

1. Warten bis der Thread tatsächlich "tot" ist?
   Nein, da Scheduler kill und IPI-Epilog beide auf $E_{\frac{1}{2}}$ sind!
2. Wozu jetzt noch das `kill_flag`?
   Wenn Thread vor Ende von IPI auf andere CPU migriert wird!

Aufgabe 5

TUHH
Technische Universität Hamburg

- APIC-Timer konfigurieren
- Preemptive Scheduling einführen, Scheduler `Guard`-en
- MPStuBS: Andere Threads über IPI beenden

Hinweise:

- Timer-Interrupt kann jederzeit Threads auf eine andere CPU migrieren: `Guard` prüfen
- Bei Timer-Berechnung auf Präzision und Integer Overflows achten