

Übung zu Betriebssysteme

Aufgabe 5: Zeitscheibenscheduling

Wintersemester 2021/22

Bernhard Heinloth

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



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

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

Unterbrechende Ablaufplanung

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

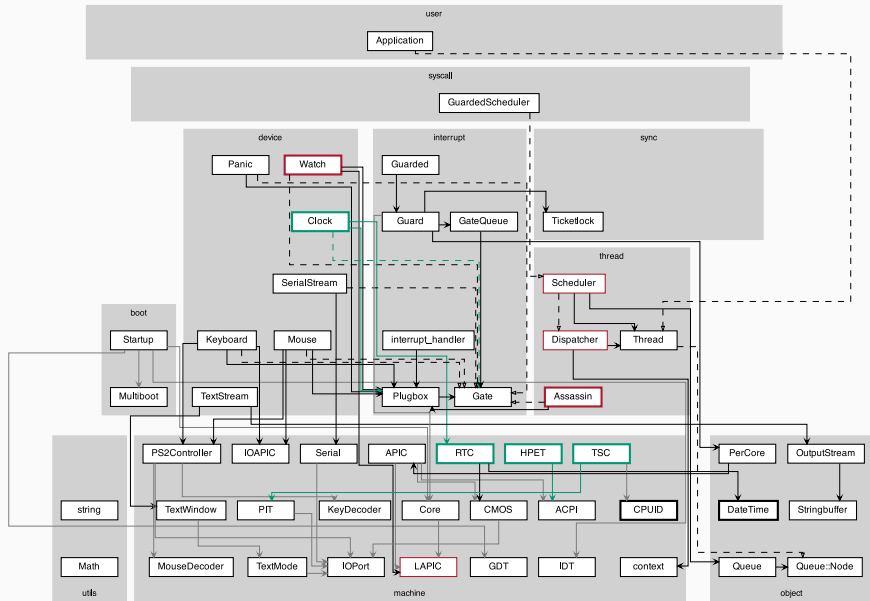
⚡ Scheduler Interrupt

Lernziele

- Präemptiven Schedulings durch Timer-Interrupts
- Schutz kritischer Abschnitte im Betriebssystem

Umsetzung

- Konfiguration des LAPIC Timers
- Umbau der Anwendungen auf präemptives Scheduling
- Interprozessorbenachrichtigung in MPStuBS
- *Optional:* TSC, RTC und/oder HPET



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()`

1. LAPIC-Timer kalibrieren

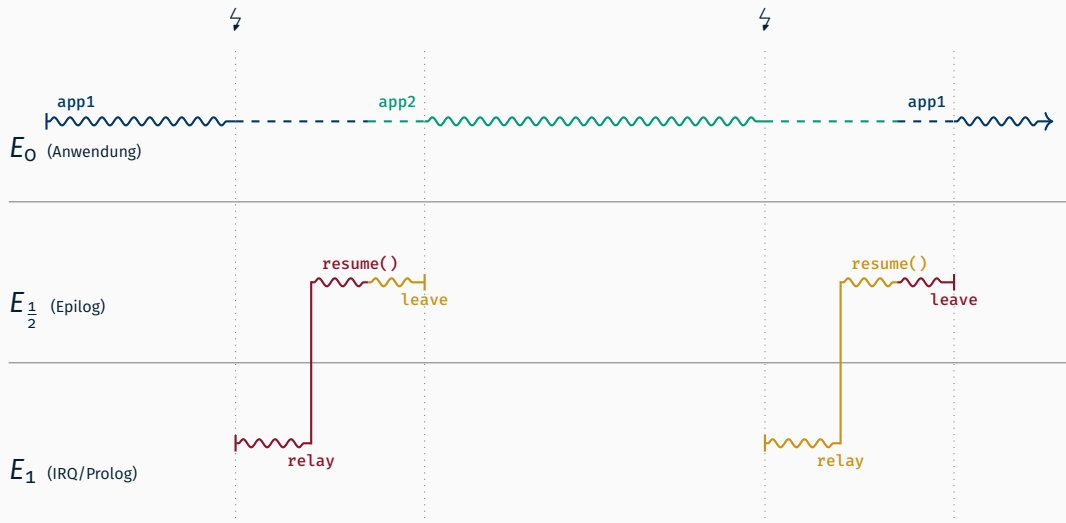
- unter Verwendung des PIT
(mittels `PIT::set` Wartezeit einstellen und in `PIT::waitForTimeout` warten)
- in der Funktion `LAPIC::Timer::ticks`
(Funktion gibt die Anzahl der Ticks in einer **Millisekunde** zurück)
- benötigt zur Konfiguration die ebenfalls noch zu implementierende Funktion `LAPIC::Timer::set`
- Hilfsstrukturen in `lapic_timer.cc` & `lapic_registers.h`

2. Initialen Wert und Vorteiler korrekt setzen

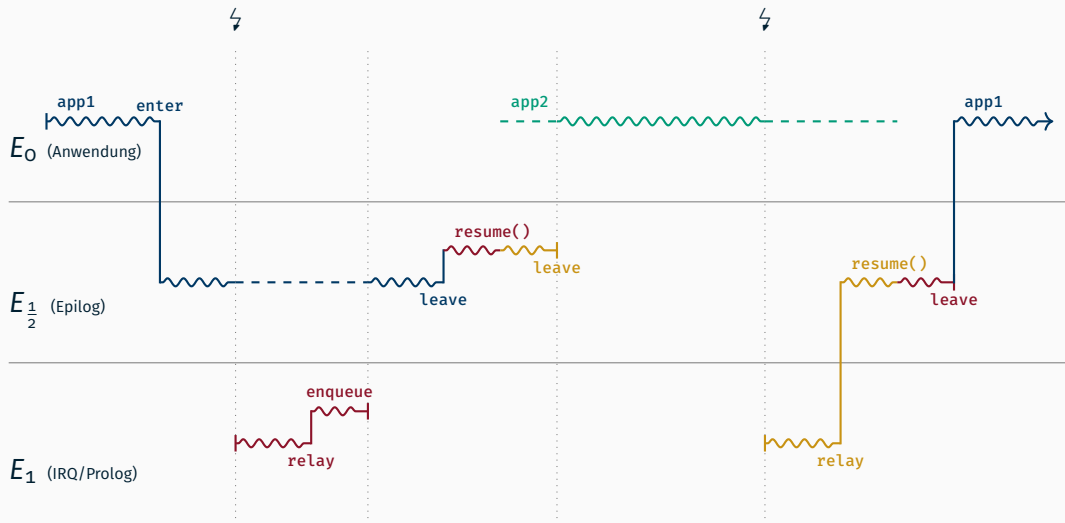
- Interrupt soll alle n Mikrosekunden ausgelöst werden
- Startwertzähler $initial = \frac{n \cdot \text{LAPIC}::\text{Timer}::\text{ticks}()}{\text{Vorteiler} \cdot 1000}$ berechnen, dabei gilt $\text{Vorteiler} = 2^x$ mit $x \in \{0, \dots, 7\}$
- Möglichst kleiner Vorteiler, aber kein Überlauf (32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

n	$5\,000\,000\ \mu\text{s} = 5\ \text{s}$
<code>LAPIC::Timer::ticks</code>	$1\,000\,000\ \text{ms}^{-1}$
Vorteiler	$2 = 2^1$
$initial$	$2\,500\,000\,000$ ✓

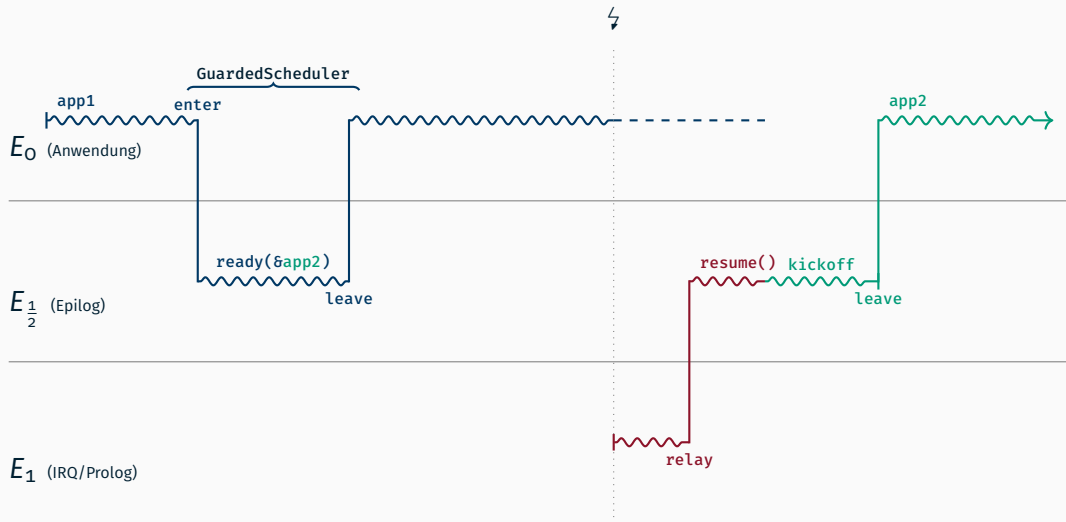
Ablaufbeispiel (Standardfall)



Ablaufbeispiel bei Faden in Systemebene



Ablaufbeispiel mit neuem Thread



Besonderheiten

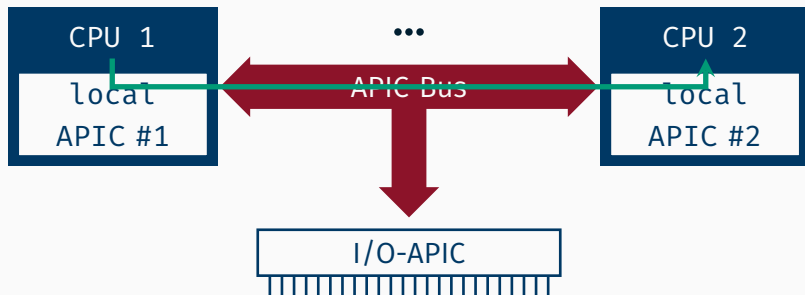
Präemptives Beenden mittels `Scheduler::kill(Thread&)`

OOSTuBS *keine Änderung:* aus der Ready-Liste entfernen
bzw. Kill-Flag setzen und bei resume prüfen

MPStuBS der Anwendungsfaden kann 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



```
destination = APIC::getLAPICID(cpu);  
  
LAPIC::IPI::send(  
    destination, vector  
    Empfänger    Interrupt
```

Der Teufel steckt im Detail

Was kann hier schon schief gehen?

```
lock[Core::getID()] = true;
```

Viel - diese Zeile wird nicht atomar ausgeführt:

```
; Array lock an Adresse 0x2000
```

```
call <Core::getID(>
```

```
mov [rax+0x2000], 0x1
```

⚡ Scheduler Interrupt

Was passiert nun, wenn der Anwendungsfaden anschließend auf einer anderen CPU eingeplant wird?

- 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?