

# Übung zu Betriebssysteme

## Ereignisbearbeitung und Synchronisation

---

18. & 21. Januar 2018

Andreas Ziegler  
Bernhard Heinloth

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



Lehrstuhl für Verteilte Systeme  
und Betriebssysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
TECHNISCHE FAKULTÄT

**Bitte evaluiert die Übung!**

Eine TAN für Tafel- und Rechnerübung zusammen,  
uns interessieren vor allem die Zusatzfragen

# Motivation

---

# Problem

**Wie sieht es mit gegenseitigem Ausschluss auf Fadenebende in OOSTUBS/MPSTUBS aus?**

## Problem

**Wie sieht es mit gegenseitigem Ausschluss auf Fadenebende in OOSTUBS/MPSTUBS aus?**

Wir haben doch bereits ein spinlock implementiert...

# Mutex mit aktivem Warten

```
mutex.lock()  
// code  
mutex.unlock()
```

# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

**App1**

```
mutex.lock()  
// code
```

**App2**

```
mutex.lock()  
// code
```

**App3**

```
mutex.lock()  
// code
```

# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active



ready list



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active

App1

ready list

App2 App3

# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

mutex.unlock()

App2

mutex.lock()

// code

mutex.unlock()

App3

mutex.lock()

// code

mutex.unlock()

active

App1

ready list

App2 App3

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

mutex.unlock()

App2

mutex.lock()

// code

mutex.unlock()

App3

mutex.lock()

// code

mutex.unlock()

active

App1

ready list

App2 App3

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

-----  
mutex.unlock()

App2

mutex.lock()

// code

mutex.unlock()

App3

mutex.lock()

// code

mutex.unlock()

active

App1

ready list

App2 App3

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

mutex.unlock()

App2

mutex.lock()

// code

mutex.unlock()

App3

mutex.lock()

// code

mutex.unlock()

active



ready list



CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

mutex.unlock()

App2

mutex.lock()

// code

mutex.unlock()

App3

mutex.lock()

// code

mutex.unlock()

active

App2

ready list

App3 App1

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active

App2

ready list

App3 App1

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active

App2

ready list

App3 App1

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

mutex.unlock()

App2

mutex.lock()

// code

mutex.unlock()

App3

mutex.lock()

// code

mutex.unlock()

active

App2

ready list

App3 App1

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

mutex.unlock()

App2

mutex.lock()

// code

mutex.unlock()

App3

mutex.lock()

// code

mutex.unlock()

active



ready list



CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active

App3

ready list

App1 App2

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active

App3

ready list

App1 App2

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active

App3

ready list

App1 App2

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock() ↳
```

```
// code
```

```
mutex.unlock()
```

active

App3

ready list

App1 App2

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active



ready list



CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active

App1

ready list

App2 App3

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active

App1

ready list

App2 App3

CPU-Zeit →



# Mutex mit aktivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

active



ready list



CPU-Zeit →



**Verschwendung von CPU-Zeit**

# Mutex mit harter Synchronisation

## Mutex mit harter Synchronisation

- Analog zur Interruptsperre mit cli

## Mutex mit harter Synchronisation

- Analog zur Interruptsperre mit cli
- **Ziel:** Kein (präemptives) Scheduling

## Mutex mit harter Synchronisation

- Analog zur Interruptsperre mit cli
- **Ziel:** Kein (präemptives) Scheduling
- Realisierbar durch

## Mutex mit harter Synchronisation

- Analog zur Interruptsperre mit cli
- **Ziel:** Kein (präemptives) Scheduling
- Realisierbar durch
  - Multitasking (temporär) deaktivieren

## Mutex mit harter Synchronisation

- Analog zur Interruptsperre mit cli
- **Ziel:** Kein (präemptives) Scheduling
- Realisierbar durch
  - Multitasking (temporär) deaktivieren
  - Erweiterung des Schedulers

# Mutex mit harter Synchronisation

- Analog zur Interruptsperre mit cli
- **Ziel:** Kein (präemptives) Scheduling
- Realisierbar durch
  - Multitasking (temporär) deaktivieren
  - Erweiterung des Schedulers
  - Wechsel auf Epilogebene

# Mutex mit harter Synchronisation

- Analog zur Interruptsperre mit cli
- **Ziel:** Kein (präemptives) Scheduling
- Realisierbar durch
  - Multitasking (temporär) deaktivieren
  - Erweiterung des Schedulers
  - Wechsel auf Epilogebene
- **Vorteile:**
  - konsistent
  - (relativ) einfach zu implementieren

# Mutex mit harter Synchronisation

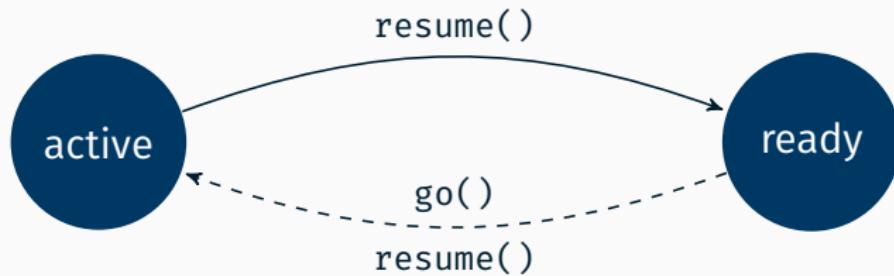
- Analog zur Interruptsperre mit cli
- **Ziel:** Kein (präemptives) Scheduling
- Realisierbar durch
  - Multitasking (temporär) deaktivieren
  - Erweiterung des Schedulers
  - Wechsel auf Epilogebene
- **Vorteile:**
  - konsistent
  - (relativ) einfach zu implementieren
- **Nachteile:**
  - Breitbandwirkung
  - Prioritätsverletzung
  - Prophylaktisch

## Idee: Passives Warten

**Ansatz:** Fäden, die den kritischen Abschnitt nicht betreten können, werden blockiert  
(d.h. von der CPU-Zuteilung ausgeschlossen)

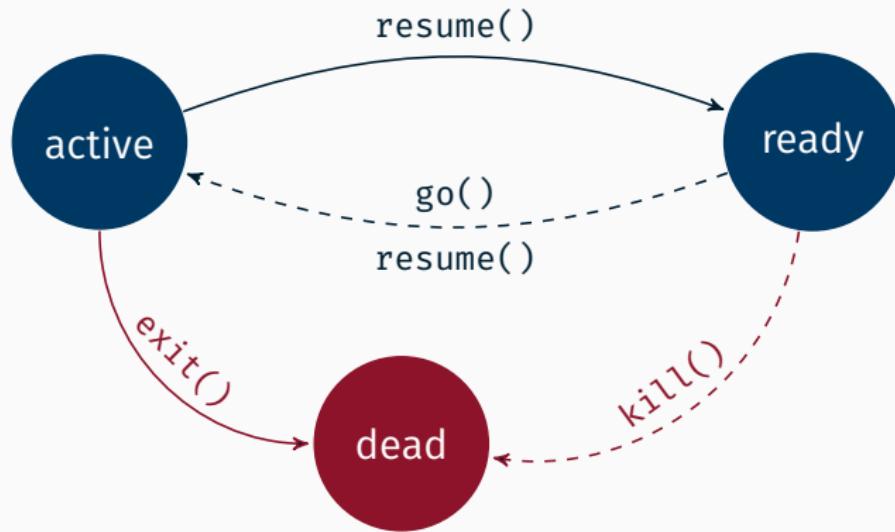
# Idee: Passives Warten

**Ansatz:** Fäden, die den kritischen Abschnitt nicht betreten können, werden blockiert  
(d.h. von der CPU-Zuteilung ausgeschlossen)

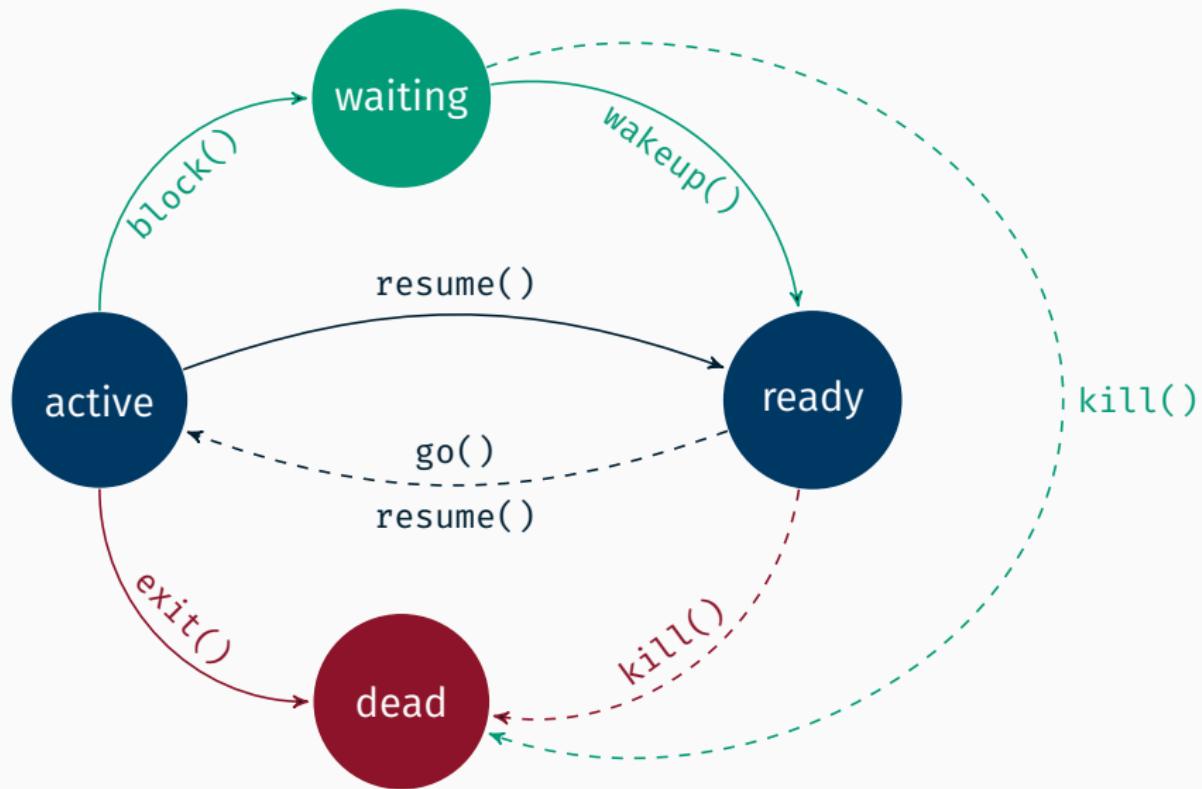


# Idee: Passives Warten

**Ansatz:** Fäden, die den kritischen Abschnitt nicht betreten können, werden blockiert  
(d.h. von der CPU-Zuteilung ausgeschlossen)



# Idee: Passives Warten



# Idee: Passives Warten



Einführung eines **waiting rooms**  
(Liste mit wartenden Prozessen)

# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

active	ready list	waiting room
<div style="border: 1px solid black; width: 250px; height: 40px;"></div>	<div style="border: 1px solid black; width: 250px; height: 40px; background-color: #ffffcc;"><b>App1 App2 App3</b></div>	<div style="border: 1px solid black; width: 250px; height: 40px;"></div>

# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

**App1**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App2**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App3**

mutex.lock()

// code

mutex.unlock()

// mehr code

active

ready list

waiting room

App1

App2 App3

# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

**App1**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App2**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App3**

mutex.lock()

// code

mutex.unlock()

// mehr code

active

ready list

waiting room

App1

App2 App3

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

**App1**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App2**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App3**

mutex.lock()

// code

mutex.unlock()

// mehr code

active

ready list

waiting room

App1

App2 App3

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

**App1**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App2**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App3**

mutex.lock()

// code

mutex.unlock()

// mehr code

active

App1

ready list

App2 App3

waiting room

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

mutex.unlock()

// mehr code

App2

mutex.lock()

// code

mutex.unlock()

// mehr code

App3

mutex.lock()

// code

mutex.unlock()

// mehr code

active

ready list

waiting room

App2 App3 App1

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

mutex.unlock()

// mehr code

App2

mutex.lock()

// code

mutex.unlock()

// mehr code

App3

mutex.lock()

// code

mutex.unlock()

// mehr code

active

ready list

waiting room

App2

App3 App1

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

mutex.lock()

// code

mutex.unlock()

// mehr code

App2

mutex.lock()

// code

mutex.unlock()

// mehr code

App3

mutex.lock()

// code

mutex.unlock()

// mehr code

active

ready list

waiting room

App2

App3 App1

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

active



ready list



waiting room



CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

active

App3

ready list

App1

waiting room

App2

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

active

App3

ready list

App1

waiting room

App2

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

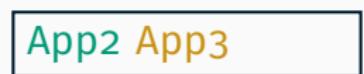
active



ready list



waiting room



CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

**App1**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App2**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App3**

mutex.lock()

// code

mutex.unlock()

// mehr code

active

ready list

waiting room

App1

App2 App3

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

**App1**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App2**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App3**

mutex.lock()

// code

mutex.unlock()

// mehr code

active

ready list

waiting room

App1

App2 App3

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

```
// mehr code
```

App2

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

```
// mehr code
```

App3

```
mutex.lock()
```

```
// code
```

```
mutex.unlock()
```

```
// mehr code
```

active

ready list

waiting room

App1

App2

App3

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

**App1**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App2**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App3**

mutex.lock()

// code

mutex.unlock()

// mehr code

active

ready list

waiting room

App1

App2

App3

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

**App1**

mutex.lock()

// code

mutex.unlock()

// mehr code



**App2**

mutex.lock()

// code

mutex.unlock()

// mehr code

**App3**

mutex.lock()

// code

mutex.unlock()

// mehr code

active

App1

ready list

App2

waiting room

App3

CPU-Zeit →



# Mutex mit passivem Warten

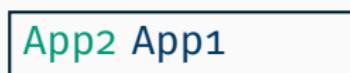
**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

active



ready list



waiting room



CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

active

App2

ready list

App1

waiting room

App3

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

active

App2

ready list

App1

waiting room

App3

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

active

App2

ready list

App1

waiting room

App3

CPU-Zeit →



# Mutex mit passivem Warten

**Beispiel:** Drei Threads auf einer CPU

App1	App2	App3
mutex.lock()	mutex.lock()	mutex.lock()
// code	// code	// code
mutex.unlock()	mutex.unlock()	mutex.unlock()
// mehr code	// mehr code	// mehr code

active

App2

ready list

App1 App3

waiting room

CPU-Zeit →



# Semaphore

---

## sem·a·phore

1. any apparatus for signaling, as by an arrangement of lights, flags, and mechanical arms on railroads
2. a system for signaling by the use of two flags, one held in each hand: the letters of the alphabet are represented by the various positions of the arms
3. any system of signaling by semaphore

*nach V. E. Neufeld, Webster's New World Dictionary, Simon & Schuster Inc., third college edition, 1988*

# Operationen

**init()** Zähler  $c$  mit positivem Wert initialisieren

**P()** von *Prolaag* (versuchen zu verringern)  
bzw. *Passeering*<sup>†</sup> (passieren)

$c > 0$  dekrementieren

$c = 0$  warten

**V()** von *Verhoog* (erhöhen) bzw. *Vrijgave*<sup>†</sup> (freigeben)

- nächsten wartenden Faden aufwecken oder
- Zähler  $c$  erhöhen

<sup>†</sup> nach Edsger W. Dijkstra, Over de sequentialiteit van procesbeschrijvingen, ca. 1962

## Zeitgesteuertes Warten

---

## Schlafen legen

```
App::action(){
    foo();
    sleep(13);
    bar();
}
```

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion sleep(3)

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- Verwendung des Wartezimmerkonzepts, um Thread aus Scheduler zu nehmen

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- Verwendung des Wartezimmerkonzepts, um Thread aus Scheduler zu nehmen  
`waiting room (13ms)`

App

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- Verwendung des Wartezimmerkonzepts, um Thread aus Scheduler zu nehmen



*(alternative Darstellung)*

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- Verwendung des Wartezimmerkonzepts, um Thread aus Scheduler zu nehmen



- mit jedem Tick wird die Wartezeit dekrementiert

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- Verwendung des Wartezimmerkonzepts, um Thread aus Scheduler zu nehmen



- mit jedem Tick wird die Wartezeit dekrementiert •

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- Verwendung des Wartezimmerkonzepts, um Thread aus Scheduler zu nehmen



- mit jedem Tick wird die Wartezeit dekrementiert 

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- Verwendung des Wartezimmerkonzepts, um Thread aus Scheduler zu nehmen



- mit jedem Tick wird die Wartezeit dekrementiert 

## Schlafen legen

```
App::action(){  
    foo();  
    sleep(13);  
    bar();  
}
```

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- Verwendung des Wartezimmerkonzepts, um Thread aus Scheduler zu nehmen



- mit jedem Tick wird die Wartezeit dekrementiert
- nach Ablauf der Wartezeit wird der Thread wieder im Scheduler eingereiht

# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms

# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms

Verwaltung mittels verketteter Liste



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms

Verwaltung mittels verketteter Liste



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms

Verwaltung mittels verketteter Liste hat den Nachteil, dass bei jedem Tick die gesamte Liste durchlaufen werden muss



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms

Verwaltung mittels verketteter Liste hat den Nachteil, dass bei jedem Tick die gesamte Liste durchlaufen werden muss •



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms

Verwaltung mittels verketteter Liste hat den Nachteil, dass bei jedem Tick die gesamte Liste durchlaufen werden muss ••



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms

Verwaltung mittels verketteter Liste hat den Nachteil, dass bei jedem Tick die gesamte Liste durchlaufen werden muss **...**



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms

Verwaltung mittels verketteter Liste hat den Nachteil, dass bei jedem Tick die gesamte Liste durchlaufen werden muss ( $\mathcal{O}(n)$ , und das 1000× pro Sekunde in der Epiloge Ebene)



# Naive Datenstruktur

**Beispiel:** Drei Anwendungen legen sich nacheinander schlafen

1. Thread **foo** für 666ms
2. Thread **bar** für 23ms
3. Thread **baz** für 42ms

Verwaltung mittels verketteter Liste hat den Nachteil, dass bei jedem Tick die gesamte Liste durchlaufen werden muss ( $\mathcal{O}(n)$ , und das 1000× pro Sekunde in der Epiloge Ebene)



*Das muss besser gehen!*

## Alternative Variante

- Einführung einer absoluten Zeit

## Alternative Variante

- Einführung einer absoluten Zeit

**Beispiel:** absolute Zeit  $T = 1334\text{ms}$

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert

**Beispiel:** absolute Zeit  $T = 1334\text{ms}$

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert •

**Beispiel:** absolute Zeit  $T = 1335\text{ms}$

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert ••

**Beispiel:** absolute Zeit  $T = 1336\text{ms}$

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert •••

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

1. Thread **foo**:  $666\text{ms} + 1337\text{ms}$

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

1. Thread **foo**:  $666\text{ms} + 1337\text{ms}$



## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

1. Thread **foo**: 666ms + 1337ms
2. Thread **bar**: 23ms + 1337ms



## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

1. Thread **foo**: 666ms + 1337ms
2. Thread **bar**: 23ms + 1337ms



## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

1. Thread **foo**: 666ms + 1337ms
2. Thread **bar**: 23ms + 1337ms



## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

1. Thread **foo**: 666ms + 1337ms
2. Thread **bar**: 23ms + 1337ms



## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

1. Thread **foo**: 666ms + 1337ms
2. Thread **bar**: 23ms + 1337ms
3. Thread **baz**: 42ms + 1337ms



# Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

1. Thread **foo**: 666ms + 1337ms
2. Thread **bar**: 23ms + 1337ms
3. Thread **baz**: 42ms + 1337ms



## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )

**Beispiel:** absolute Zeit  $T = 1337\text{ms}$

1. Thread **foo**: 666ms + 1337ms
2. Thread **bar**: 23ms + 1337ms
3. Thread **baz**: 42ms + 1337ms



## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )
- Wenn die aktuelle Zeit T dem ersten Element entspricht:  
Thread wieder dem Scheduler übergeben ( $\mathcal{O}(1)$ )

**Beispiel:** absolute Zeit T = 1337ms

1. Thread **foo**: 666ms + 1337ms
2. Thread **bar**: 23ms + 1337ms
3. Thread **baz**: 42ms + 1337ms



## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )
- Wenn die aktuelle Zeit T dem ersten Element entspricht: Thread wieder dem Scheduler übergeben ( $\mathcal{O}(1)$ )

**Beispiel:** absolute Zeit T = 1360ms

1. Thread **foo**: 666ms + 1337ms
2. Thread **bar**: 23ms + 1337ms
3. Thread **baz**: 42ms + 1337ms



## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )
- Wenn die aktuelle Zeit T dem ersten Element entspricht:  
Thread wieder dem Scheduler übergeben ( $\mathcal{O}(1)$ )

## Nachteile

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )
- Wenn die aktuelle Zeit T dem ersten Element entspricht:  
Thread wieder dem Scheduler übergeben ( $\mathcal{O}(1)$ )

## Nachteile

- absolute Zeit ist ein neuer Zustand

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )
- Wenn die aktuelle Zeit T dem ersten Element entspricht:  
Thread wieder dem Scheduler übergeben ( $\mathcal{O}(1)$ )

## Nachteile

- absolute Zeit ist ein neuer Zustand
- 32bit-Überlauf (nach 49.7 Tagen)

## Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert
- Berechnung der Endzeit beim Einfügen neuer Threads
- Einordnen in einer Vorrangwarteschlange ( $\mathcal{O}(n)$ )
- Wenn die aktuelle Zeit T dem ersten Element entspricht:  
Thread wieder dem Scheduler übergeben ( $\mathcal{O}(1)$ )

## Nachteile

- absolute Zeit ist ein neuer Zustand
- 32bit-Überlauf (nach 49.7 Tagen)

*Geht das nicht effizienter (ohne solche Probleme)?*

## Effiziente Variante

---

- Verwendung der relativen Delta-Zeit

## Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert  
(negative Zeitdifferenzen sind nicht erlaubt!)

# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert  
(negative Zeitdifferenzen sind nicht erlaubt!)

## Beispiel:

1. Thread **foo**: 666ms

# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)

## Beispiel:

1. Thread **foo**: 666ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$

## Beispiel:

1. Thread **foo**: 666ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$

## Beispiel:

1. Thread **foo**:  $666\text{ms} - t_0 = 666\text{ms}$



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$

## Beispiel:

1. Thread **foo**: 666ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$

## Beispiel:

1. Thread **foo**: 666ms
2. Thread **bar**: 23ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit

- es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
- Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$

## Beispiel:

1. Thread **foo**: 666ms
2. Thread **bar**: 23ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )

## Beispiel:

1. Thread **foo**: 666ms
2. Thread **bar**: 23ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )

## Beispiel:

1. Thread **foo**: 666ms
2. Thread **bar**: 23ms –  $t_0 = 23\text{ms}$



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )

## Beispiel:

1. Thread **foo**: 666ms
2. Thread **bar**: 23ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**: 666ms
2. Thread **bar**: 23ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**:  $666\text{ms} - t_{\text{bar}} = 643\text{ms}$
2. Thread **bar**:  $23\text{ms}$



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**: 643ms
2. Thread **bar**: 23ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**: 643ms
2. Thread **bar**: 23ms
3. Thread **baz**: 42ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**: 643ms
2. Thread **bar**: 23ms
3. Thread **baz**: 42ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**: 643ms
2. Thread **bar**: 23ms
3. Thread **baz**:  $42\text{ms} - t_0 = 42\text{ms}$



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**: 643ms
2. Thread **bar**: 23ms
3. Thread **baz**:  $42\text{ms} - t_0 = 42\text{ms} > t_{\text{bar}}$



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**: 643ms
2. Thread **bar**: 23ms
3. Thread **baz**:  $42\text{ms} - t_0 - t_{\text{bar}} = 19\text{ms}$



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**: 643ms
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )  
⇒ keine Vorrangwarteschlange!

## Beispiel:

1. Thread **foo**: 643ms
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )  
⇒ keine Vorrangwarteschlange!

## Beispiel:

1. Thread **foo**:  $643\text{ms} - t_{\text{baz}} = 624\text{ms}$
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )  
⇒ keine Vorrangwarteschlange!

## Beispiel:

1. Thread **foo**: 624ms
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )  
⇒ keine Vorrangwarteschlange!
- Erstes Element wird mit jedem Tick dekrementiert

## Beispiel:

1. Thread **foo**: 624ms
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )  
⇒ keine Vorrangwarteschlange!
- Erstes Element wird mit jedem Tick dekrementiert •

## Beispiel:

1. Thread **foo**: 624ms
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )  
⇒ keine Vorrangwarteschlange!
- Erstes Element wird mit jedem Tick dekrementiert 

## Beispiel:

1. Thread **foo**: 624ms
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )  
⇒ keine Vorrangwarteschlange!
- Erstes Element wird mit jedem Tick dekrementiert **...**

## Beispiel:

1. Thread **foo**: 624ms
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )  
⇒ keine Vorrangwarteschlange!
- Erstes Element wird mit jedem Tick dekrementiert und bei  $0$  dem Scheduler übergeben

## Beispiel:

1. Thread **foo**: 624ms
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



# Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - es wird nur die Zeitdifferenz zum Vorgänger gespeichert (negative Zeitdifferenzen sind nicht erlaubt!)
  - Vorgänger des ersten Elements hat Zeit  $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ( $\mathcal{O}(n)$ )
  - Nachfolgendes Element muss angepasst werden ( $\mathcal{O}(1)$ )  
⇒ keine Vorrangwarteschlange!
- Erstes Element wird mit jedem Tick dekrementiert und bei  $0$  dem Scheduler übergeben ( $\mathcal{O}(1)$ )

## Beispiel:

1. Thread **foo**: 624ms
2. Thread **bar**: 23ms
3. Thread **baz**: 19ms



## Umsetzung

---

# Aufgaben

- Implementierung von Semaphoren mit passivem Warten

# Aufgaben

- Implementierung von Semaphoren mit passivem Warten
  - Verwendung in der neuen getkey( ) Funktion

# Aufgaben

- Implementierung von Semaphoren mit passivem Warten
  - Verwendung in der neuen getkey( ) Funktion
- Zeitgesteuertes Schlafen der Threads

# Aufgaben

- Implementierung von Semaphoren mit passivem Warten
  - Verwendung in der neuen getkey( ) Funktion
- Zeitgesteuertes Schlafen der Threads
- Leerlauf des Prozessor (falls keine Threads vorhanden)

# Aufgaben

- Implementierung von Semaphoren mit passivem Warten
  - Verwendung in der neuen getkey( ) Funktion
- Zeitgesteuertes Schlafen der Threads
- Leerlauf des Prozessor (falls keine Threads vorhanden)
- Kapselung in Systemaufrufschnittstellen (syscall)  
welche sich um den Wechsel in die Epilogegebene kümmern

## (viel zu) viele Wartezimmer

- Jede Semaphore und jeder Wecker (bell) sind gleichzeitig Wartezimmer

## (viel zu) viele Wartezimmer

- Jede Semaphore und jeder Wecker (bell) sind gleichzeitig Wartezimmer
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt

## (viel zu) viele Wartezimmer

- Jede Semaphore und jeder Wecker (bell) sind gleichzeitig Wartezimmer
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt
  - Grund dafür ist eine einfachere Implementierung, da wir beim Wecker die neuen Ablaufplanmethoden für die Semaphoren verwenden können.  
Wirkt aber dafür auf den ersten Blick komisch.

## (viel zu) viele Wartezimmer

- Jede Semaphore und jeder Wecker (bell) sind gleichzeitig Wartezimmer
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt
  - Grund dafür ist eine einfachere Implementierung, da wir beim Wecker die neuen Ablaufplanmethoden für die Semaphoren verwenden können.  
Wirkt aber dafür auf den ersten Blick komisch.
- Umsetzung durch Ableitung von `waitingroom`

## (viel zu) viele Wartezimmer

- Jede Semaphore und jeder Wecker (bell) sind gleichzeitig Wartezimmer
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt
  - Grund dafür ist eine einfachere Implementierung, da wir beim Wecker die neuen Ablaufplanmethoden für die Semaphoren verwenden können.  
Wirkt aber dafür auf den ersten Blick komisch.
- Umsetzung durch Ableitung von waitingroom
- Die Wecker (inklusive Wartezimmer) werden zur Laufzeit als temporäre Objekte erzeugt

## (viel zu) viele Wartezimmer

- Jede Semaphore und jeder Wecker (bell) sind gleichzeitig Wartezimmer
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt
  - Grund dafür ist eine einfachere Implementierung, da wir beim Wecker die neuen Ablaufplanmethoden für die Semaphoren verwenden können.  
Wirkt aber dafür auf den ersten Blick komisch.
- Umsetzung durch Ableitung von waitingroom
- Die Wecker (inklusive Wartezimmer) werden zur Laufzeit als temporäre Objekte erzeugt
  - d.h. sie liegen auf dem Stapelspeicher (Stack)

## (viel zu) viele Wartezimmer

- Jede Semaphore und jeder Wecker (bell) sind gleichzeitig Wartezimmer
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt
  - Grund dafür ist eine einfachere Implementierung, da wir beim Wecker die neuen Ablaufplanmethoden für die Semaphoren verwenden können.  
Wirkt aber dafür auf den ersten Blick komisch.
- Umsetzung durch Ableitung von `waitingroom`
- Die Wecker (inklusive Wartezimmer) werden zur Laufzeit als temporäre Objekte erzeugt
  - d.h. sie liegen auf dem Stapelspeicher (Stack)
- Die Wecker werden wieder in einer Liste (vom *Glöckner* / *bellringer*) verwaltet

Der bellringer prüft regelmäßig die Wecker

Der bellringer prüft regelmäßig die Wecker

- unter Verwendung des LAPIC Timers

Der bellringer prüft regelmäßig die Wecker

- unter Verwendung des LAPIC Timers
- welcher mit `windup(1000)` auf Millisekudentakt gestellt werden sollte

Der bellringer prüft regelmäßig die Wecker

- unter Verwendung des LAPIC Timers
- welcher mit `windup(1000)` auf Millisekudentakt gestellt werden sollte
- es reicht, wenn eine CPU das übernimmt

**Problem:** zu wenig Threads bereit

**Problem:** zu wenig Threads bereit

**Lösung:** je ein IdleThread pro CPU

**Problem:** zu wenig Threads bereit

**Lösung:** je ein IdleThread pro CPU

```
void IdleThread::action() {  
    while (true) {  
        if(!scheduler.isEmpty())  
            Guarded_Scheduler::resume();  
    }  
}
```

# Leerlauf

**Problem:** zu wenig Threads bereit

**Lösung:** je ein IdleThread pro CPU

```
void IdleThread::action() {  
    while (true) {  
        if (!scheduler.isEmpty())  
            Guarded_Scheduler::resume();  
    }  
}
```

CPU fungiert effektiv als Heizkörper

**Problem:** zu wenig Threads bereit

**Lösung:** je ein IdleThread pro CPU

```
void IdleThread::action() {  
    while (true) {  
        if (!scheduler.isEmpty())  
            Guarded_Scheduler::resume();  
    }  
}
```

CPU fungiert effektiv als Heizkörper, besser wäre jedoch ein  
Schlafzustand

CPU::idle() hält Prozessor bis zum nächsten Interrupt an

## Leerlauf

CPU::idle() hält Prozessor bis zum nächsten Interrupt an

```
void IdleThread::action() {
    while (true) {
        if(scheduler.isEmpty())
            CPU::idle();
        else
            Guarded_Scheduler::resume();
    }
}
```

## Leerlauf

CPU::idle() hält Prozessor bis zum nächsten Interrupt an

```
void IdleThread::action() {
    while (true) {
        if(scheduler.isEmpty())
            CPU::idle(); Thread ready
        else
            Guarded_Scheduler::resume();
    }
}
```

## Leerlauf

CPU::idle() hält Prozessor bis zum nächsten Interrupt an

```
void IdleThread::action() {
    while (true) {
        if(scheduler.isEmpty())
            CPU::idle(); Thread ready
        else
            Guarded_Scheduler::resume();
    }
}
```

Durch Aufwachen eines wartenden Threads (oder  
Neueinplanung bei MPSTUBS) kann ein **Lost-Wakeup**  
passieren!

## Leerlauf

CPU::idle() hält Prozessor bis zum nächsten Interrupt an  
(mittels atomaren sti und hlt)

```
void IdleThread::action() {
    while (true) {
        if(scheduler.isEmpty())
            CPU::idle(); Thread ready
        else
            Guarded_Scheduler::resume();
    }
}
```

Durch Aufwachen eines wartenden Threads (oder  
Neueinplanung bei MPSTUBS) kann ein **Lost-Wakeup**  
passieren!

## Leerlauf

CPU::idle() hält Prozessor bis zum nächsten Interrupt an  
(mittels atomaren sti und hlt)

```
void IdleThread::action() {
    while (true) {
        CPU::disable_int();
        if(scheduler.isEmpty())
            CPU::idle();
        else {
            CPU::enable_int();
            Guarded_Scheduler::resume();
        }
    }
}
```

## *Optional: Nicht mehr ticken*

Im Leerlauf wird der Kern jedoch immer noch durch Unterbrechungen von unserer `watch` aufgeweckt.

## *Optional: Nicht mehr ticken*

Im Leerlauf wird der Kern jedoch immer noch durch Unterbrechungen von unserer watch aufgeweckt.  
⇒ **erhöhter Energieverbrauch**

## *Optional: Nicht mehr ticken*

Im Leerlauf wird der Kern jedoch immer noch durch Unterbrechungen von unserer watch aufgeweckt.

⇒ **erhöhter Energieverbrauch**

- Im Leerlauf kann der LAPIC Timer deaktiviert werden (Interrupt maskieren)

## *Optional: Nicht mehr ticken*

Im Leerlauf wird der Kern jedoch immer noch durch Unterbrechungen von unserer `watch` aufgeweckt.

⇒ **erhöhter Energieverbrauch**

- Im Leerlauf kann der LAPIC Timer deaktiviert werden (Interrupt maskieren)
- Erweiterung der `watch` im `IdleThread` anwenden:

```
if(scheduler.isEmpty()){
    watch.block();
    CPU::idle();
    watch.unblock();
}
```

## *Optional: Nicht mehr ticken*

Im Leerlauf wird der Kern jedoch immer noch durch Unterbrechungen von unserer `watch` aufgeweckt.

⇒ **erhöhter Energieverbrauch**

- Im Leerlauf kann der LAPIC Timer deaktiviert werden (Interrupt maskieren)
- Erweiterung der `watch` im `IdleThread` anwenden:

```
if(scheduler.isEmpty()){
    watch.block();
    CPU::idle();
    watch.unblock();
}
```

- Sonderbehandlung bei der für den `bellringer` verantwortlichen CPU berücksichtigen!

## Weitere Fragen?

---

Fast geschafft - dies ist die letzte *Pflichtaufgabe!*