Übung zu Betriebssysteme

Ereignisbearbeitung und Synchronisation

23. Januar 2025

Maximilian Ott, Dustin Nguyen, Phillip Raffeck & Bernhard Heinloth

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

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FAU Friedrich-Alexander-Universität
Technische Fakultät

Motivation

Problem

Wie sieht es mit gegenseitigem Ausschluss auf Fadenebende in StuBS aus?

Problem

Wie sieht es mit gegenseitigem Ausschluss auf Fadenebende in StuBS aus?

Wir haben doch bereits ein `spinlock` bzw. `ticketlock` implementiert...

Mutex mit aktivem Warten

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

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()
```

Mutex mit aktivem 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()`

active

ready list

App1 App2 App3

Mutex mit aktivem 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()`

active

App1

ready list

App2 App3

Mutex mit aktivem Warten

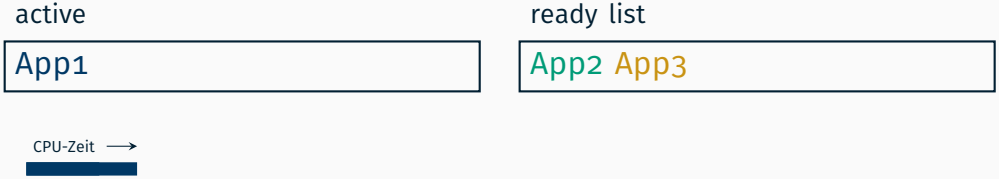
Beispiel: Drei Threads auf einer CPU

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

Mutex mit aktivem Warten

Beispiel: Drei Threads auf einer CPU

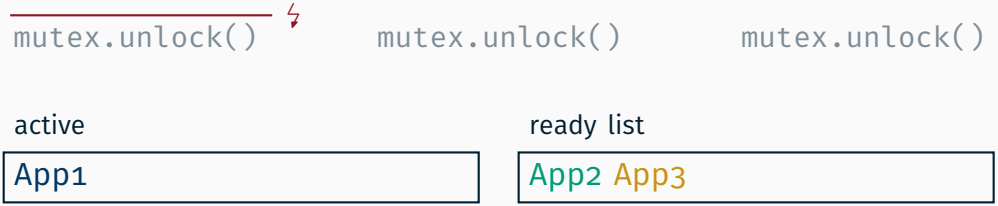
App1	App2	App3
`mutex.lock()`	`mutex.lock()`	`mutex.lock()`
`// code`	`// code`	`// code`
`mutex.unlock()`	`mutex.unlock()`	`mutex.unlock()`



Mutex mit aktivem 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()`



Mutex mit aktivem 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()`

active

ready list

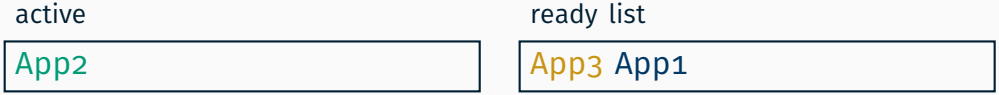
App2 App3 App1

CPU-Zeit ⟶

Mutex mit aktivem 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()`



Mutex mit aktivem 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()`

active

App2

ready list

App3 App1

Mutex mit aktivem 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()`

active

App2

ready list

App3 App1

Mutex mit aktivem 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()`

active

App2

ready list

App3 App1

Mutex mit aktivem 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()`

active

ready list

App3 App1 App2

Mutex mit aktivem 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()`

active

App3

ready list

App1 App2

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

Mutex mit aktivem 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()`

active

App3

ready list

App1 App2

Mutex mit aktivem 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()`

active

App3

ready list

App1 App2

Mutex mit aktivem 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()`

active

ready list: App1 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

Mutex mit aktivem 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()`

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

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

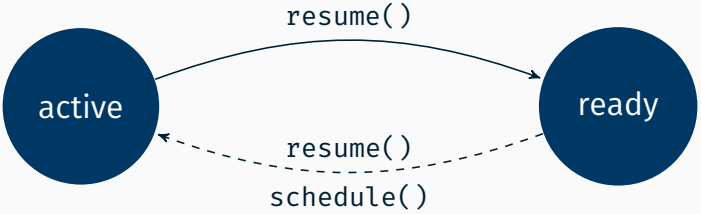
- Analog zur Interruptsperre mit `cli`
- **Ziel:** Kein (präemptives) Scheduling
- Realisierbar durch
  - Multitasking (temporär) deaktivieren
  - Erweiterung des Schedulers
  - Wechsel auf Epilogebene
- **Vorteile:**
  - konsistent
  - (relativ) einfach zu implementieren
- **Nachteile:**
  - Breitbandwirkung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

Idee: Passives Warten

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

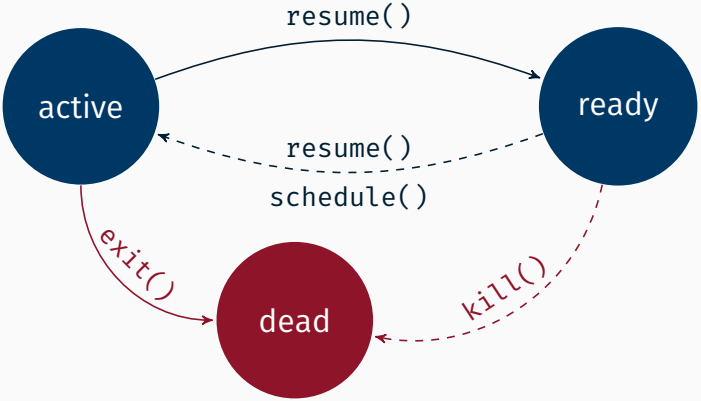
Idee: Passives Warten

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

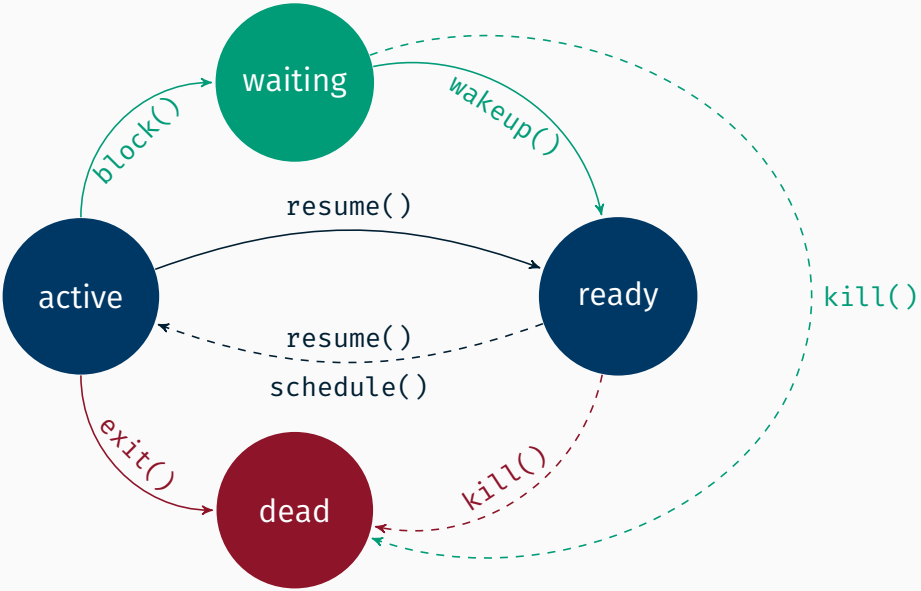


Idee: Passives Warten

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 room**s
(Liste mit wartenden Threads)

Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

App1

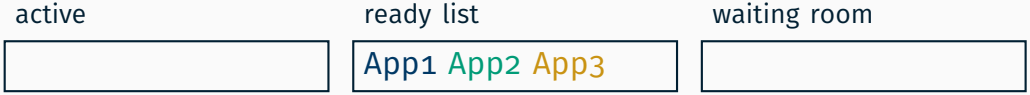
```
mutex.lock()
// code
mutex.unlock()
// mehr code
```

App2

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

App3

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



Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

App1

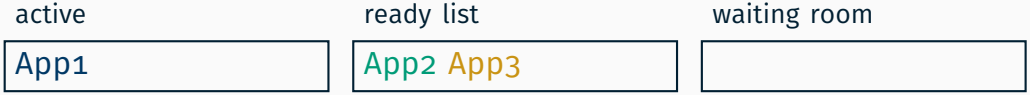
```
mutex.lock()
// code
mutex.unlock()
// mehr code
```

App2

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

App3

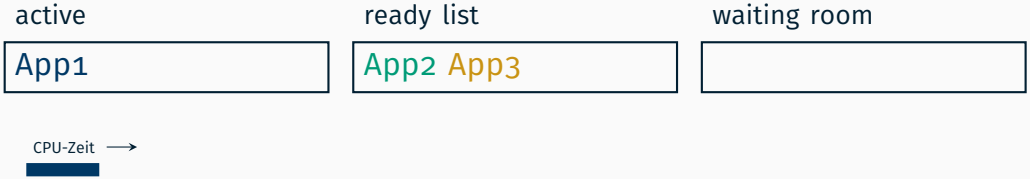
```
mutex.lock()
// code
mutex.unlock()
// mehr code
```



Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

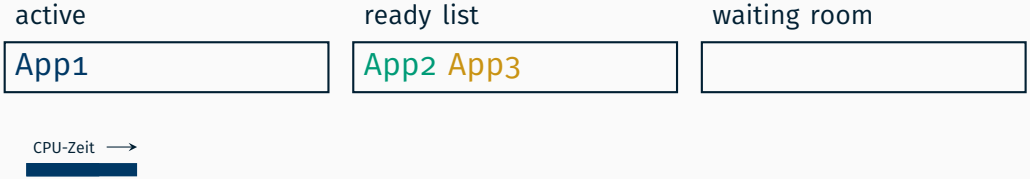
App1	App2	App3
`mutex.lock()`	`mutex.lock()`	`mutex.lock()`
`// code`	`// code`	`// code`
`mutex.unlock()`	`mutex.unlock()`	`mutex.unlock()`
`// mehr code`	`// mehr code`	`// mehr code`



Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

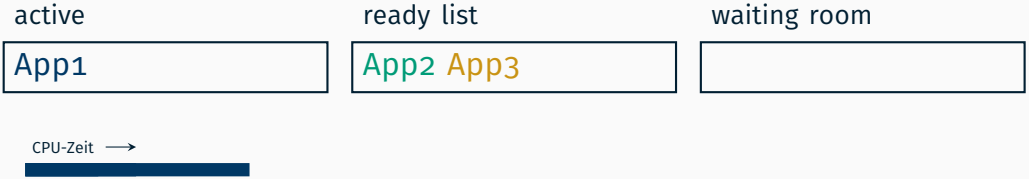
App1	App2	App3
`mutex.lock()`	`mutex.lock()`	`mutex.lock()`
`// code`	`// code`	`// code`
`mutex.unlock()`	`mutex.unlock()`	`mutex.unlock()`
`// mehr code`	`// mehr code`	`// mehr code`



Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

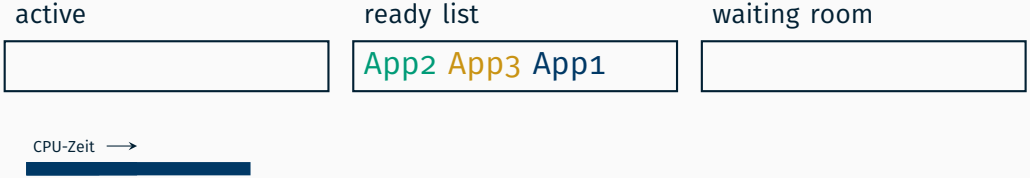
App1	App2	App3
`mutex.lock()`	`mutex.lock()`	`mutex.lock()`
`// code`	`// code`	`// code`
`mutex.unlock()`	`mutex.unlock()`	`mutex.unlock()`
`// mehr code`	`// mehr code`	`// mehr code`



Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

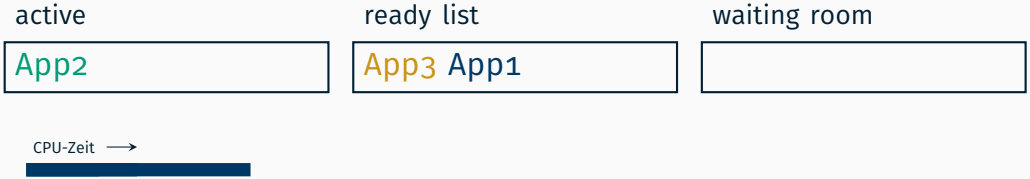
App1	App2	App3
`mutex.lock()`	`mutex.lock()`	`mutex.lock()`
`// code`	`// code`	`// code`
`mutex.unlock()`	`mutex.unlock()`	`mutex.unlock()`
`// mehr code`	`// mehr code`	`// mehr code`



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`



Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

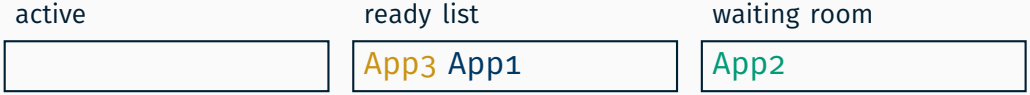
App1	App2	App3
`mutex.lock()`	`mutex.lock()`	`mutex.lock()`
`// code`	`// code`	`// code`
`mutex.unlock()`	`mutex.unlock()`	`mutex.unlock()`
`// mehr code`	`// mehr code`	`// mehr code`

active	ready list	waiting room
App2	App3 App1	

Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

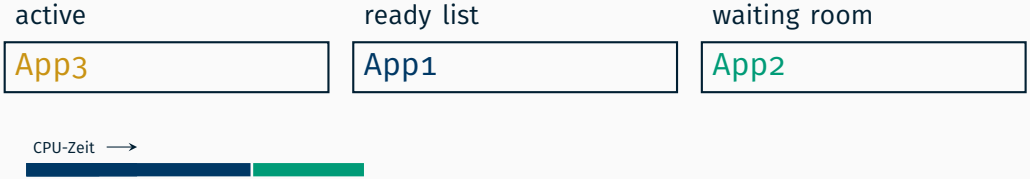
App1	App2	App3
`mutex.lock()`	`mutex.lock()`	`mutex.lock()`
`// code`	`// code`	`// code`
`mutex.unlock()`	`mutex.unlock()`	`mutex.unlock()`
`// mehr code`	`// mehr code`	`// mehr code`



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`



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
App3	App1	App2

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
	App1	App2 App3

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
App1		App2 App3

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
App1		App2 App3

Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

App1

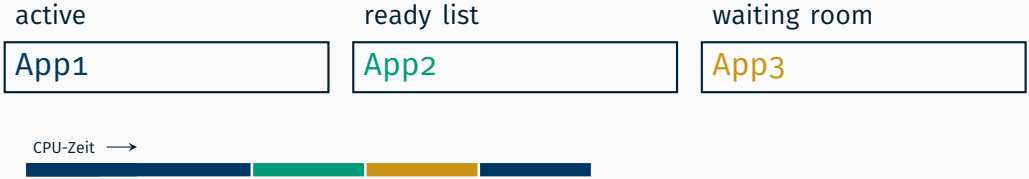
```
mutex.lock()
// code
mutex.unlock()
// mehr code
```

App2

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

App3

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



Mutex mit passivem Warten

Beispiel: Drei Threads auf einer CPU

App1

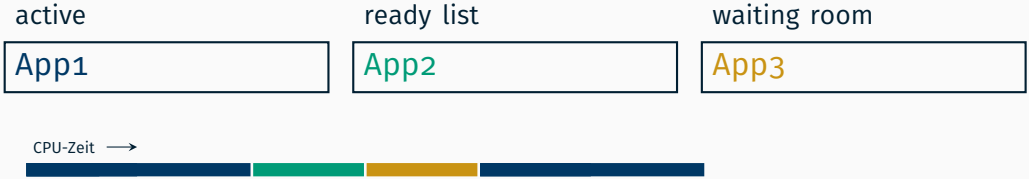
```
mutex.lock()
// code
mutex.unlock()
// mehr code
```

App2

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

App3

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



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`

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
	App2 App1	App3

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
App2	App1	App3

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
App2	App1	App3

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
App2	App1	App3

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
App2	App1 App3	

Semaphore

Wortherkunft

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*[†] (passieren)

- $c > 0$ dekrementieren
- $c = 0$ warten

`V()` von *Verhoog* (erhöhen) bzw. *Vrijgave*[†] (freigeben)

- nächsten wartenden Faden aufwecken oder
- Zähler c erhöhen

[†] nach Edsger W. Dijkstra, Over de sequentialiteit van procesbeschrijvingen, ca. 1962

```
Semaphore mutex(1);

void func() {

  mutex.p();  // lock

  // critical section

  mutex.v();  // unlock

}
```

```
Semaphore mutex(1);

void func() {

  mutex.p();  // lock

  // critical section

  mutex.v();  // unlock

}
```

```
Semaphore empty(size);
Semaphore full(0);

void producer(){
  empty.p();
  // produce
  full.v();
}

void consumer(){
  full.p();
  // consume
  empty.v();
}
```

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)
- analog zu Wartezimmer den Thread aus Ready-Liste des Schedulers nehmen

Schlafen legen

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

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- analog zu Wartezimmer den Thread aus Ready-Liste des Schedulers nehmen

Schlafen legen

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

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- analog zu Wartezimmer den Thread aus Ready-Liste des Schedulers nehmen

(alternative Darstellung)

Schlafen legen

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

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- analog zu Wartezimmer den Thread aus Ready-Liste des Schedulers nehmen

- mit jedem Tick die Wartezeit dekrementieren

Schlafen legen

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

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- analog zu Wartezimmer den Thread aus Ready-Liste des Schedulers nehmen

- mit jedem Tick die Wartezeit dekrementieren •

Schlafen legen

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

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- analog zu Wartezimmer den Thread aus Ready-Liste des Schedulers nehmen

- mit jedem Tick die Wartezeit dekrementieren ●●

Schlafen legen

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

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- analog zu Wartezimmer den Thread aus Ready-Liste des Schedulers nehmen

- mit jedem Tick die Wartezeit dekrementieren •••

Schlafen legen

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

- ähnlich der Funktion `sleep(3)`
- jedoch mit Wartezeit in Millisekunden (statt Sekunden)
- analog zu Wartezimmer den Thread aus Ready-Liste des Schedulers nehmen

- mit jedem Tick die Wartezeit dekrementieren
- nach Ablauf der Wartezeit Thread wieder im Scheduler einreihen

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 $1000\times$ pro Sekunde in der Epilogebene)

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 $1000\times$ pro Sekunde in der Epilogebene)

Das muss besser gehen!

Alternative Variante

- Einführung einer absoluten Zeit

Alternative Variante

- Einführung einer absoluten Zeit

Beispiel: absolute Zeit `T = 1334ms`

Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert

Beispiel: absolute Zeit `T = 1334ms`

Alternative Variante

- Einführung einer absoluten Zeit
  - wird mit jedem Tick inkrementiert

Beispiel: absolute Zeit `T = 1335ms`

Alternative Variante

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

Beispiel: absolute Zeit `T = 1336ms`

Alternative Variante

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

Beispiel: absolute Zeit `T = 1337ms`

Alternative Variante

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

Beispiel: absolute Zeit `T = 1337ms`

Alternative Variante

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

Beispiel: absolute Zeit `T = 1337ms`

1. Thread **foo**: `666ms + 1337ms`

Alternative Variante

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

Beispiel: absolute Zeit `T = 1337ms`

1. Thread **foo**: `666ms + 1337ms`

Alternative Variante

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

Beispiel: absolute Zeit `T = 1337ms`

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 = 1337ms`

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 = 1337ms`

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 = 1337ms`

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 = 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 ($\mathcal{O}(n)$)

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 ($\mathcal{O}(n)$)

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 = 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
- Bei 32bit Überlauf möglich (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
- Bei 32bit Überlauf möglich (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

Beispiel:

1. Thread **foo**: `666ms`

Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - Es wird nur die Zeitdifferenz zum Vorgänger gespeichert

Beispiel:

1. Thread **foo**: `666ms`

Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - Es wird nur die Zeitdifferenz zum Vorgänger gespeichert
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$

Beispiel:

1. Thread **foo**: `666ms`

Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - Es wird nur die Zeitdifferenz zum Vorgänger gespeichert
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$

Beispiel:

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

Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - Es wird nur die Zeitdifferenz zum Vorgänger gespeichert
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$

Beispiel:

1. Thread **foo**: `666ms`

Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - Es wird nur die Zeitdifferenz zum Vorgänger gespeichert
  - 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
  - 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$
- 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$
- Neue Threads nach Schlafdauer einordnen ($\mathcal{O}(n)$)

Beispiel:

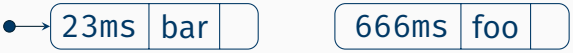
1. Thread **foo**: `666ms`
2. Thread **bar**: $\texttt{23ms} - t_0 = \texttt{23ms}$

Effiziente Variante

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

Beispiel:

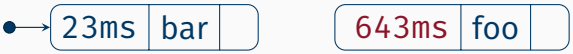
1. Thread **foo**: `666ms`
2. Thread **bar**: `23ms`

Effiziente Variante

- Verwendung der relativen Delta-Zeit
  - Es wird nur die Zeitdifferenz zum Vorgänger gespeichert
  - 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**: 23ms



Effiziente Variante

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

  $\Rightarrow$ 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ($\mathcal{O}(n)$)
  - Nachfolgendes Element muss angepasst werden ($\mathcal{O}(1)$)

  $\Rightarrow$ 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ($\mathcal{O}(n)$)
  - Nachfolgendes Element muss angepasst werden ($\mathcal{O}(1)$)

  $\Rightarrow$ 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = 0\text{ms}$
- Neue Threads nach Schlafdauer einordnen ($\mathcal{O}(n)$)
  - Nachfolgendes Element muss angepasst werden ($\mathcal{O}(1)$)

  $\Rightarrow$ 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$
- Neue Threads nach Schlafdauer einordnen ($\mathcal{O}(n)$)
  - Nachfolgendes Element muss angepasst werden ($\mathcal{O}(1)$)

  $\Rightarrow$ 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$
- Neue Threads nach Schlafdauer einordnen ($\mathcal{O}(n)$)
  - Nachfolgendes Element muss angepasst werden ($\mathcal{O}(1)$)

  $\Rightarrow$ 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$
- Neue Threads nach Schlafdauer einordnen ($\mathcal{O}(n)$)
  - Nachfolgendes Element muss angepasst werden ($\mathcal{O}(1)$)

  $\Rightarrow$ 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$
- Neue Threads nach Schlafdauer einordnen ($\mathcal{O}(n)$)
  - Nachfolgendes Element muss angepasst werden ($\mathcal{O}(1)$)

  $\Rightarrow$ 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
  - Vorgänger des ersten Elements hat Zeit $t_0 = \texttt{0ms}$
- Neue Threads nach Schlafdauer einordnen ($\mathcal{O}(n)$)
  - Nachfolgendes Element muss angepasst werden ($\mathcal{O}(1)$)

  $\Rightarrow$ 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
  - *Optional:* Demonstration mittels PC Speaker

Aufgaben

- Implementierung von Semaphoren mit passivem Warten
  - Verwendung in der neuen `getKey()` Funktion
- Zeitgesteuertes Schlafen der Threads
  - *Optional:* Demonstration mittels PC Speaker
- Leerlauf des Prozessor (falls keine Threads vorhanden)

Aufgaben

- Implementierung von Semaphoren mit passivem Warten
  - Verwendung in der neuen `getKey()` Funktion
- Zeitgesteuertes Schlafen der Threads
  - *Optional:* Demonstration mittels PC Speaker
- Leerlauf des Prozessor (falls keine Threads vorhanden)
  - *Optional:* Energiesparender „Tickless Kernel“

Aufgaben

- Implementierung von Semaphoren mit passivem Warten
  - Verwendung in der neuen `getKey()` Funktion
- Zeitgesteuertes Schlafen der Threads
  - *Optional:* Demonstration mittels PC Speaker
- Leerlauf des Prozessor (falls keine Threads vorhanden)
  - *Optional:* Energiesparender „Tickless Kernel“
- Kapselung in Systemaufrufschnittstellen (`syscall`)
  welche sich um den Wechsel in die Epilogebene kümmern

(viel zu) viele Wartezimmer

- jede Semaphore ist gleichzeitig ein Wartezimmer

(viel zu) viele Wartezimmer

- jede Semaphore ist gleichzeitig ein Wartezimmer
- und jeder Wecker (`bell`) ebenfalls

(viel zu) viele Wartezimmer

- jede Semaphore ist gleichzeitig ein Wartezimmer
- und jeder Wecker (`bell`) ebenfalls
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt

(viel zu) viele Wartezimmer

- jede Semaphore ist gleichzeitig ein Wartezimmer
- und jeder Wecker (`bell`) ebenfalls
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt
  - → einfache Implementierung, da wir beim Wecker die neuen Ablaufplanmethoden für die Semaphoren verwenden können.
    Wirkt aber auf den ersten Blick halt komisch.

(viel zu) viele Wartezimmer

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

(viel zu) viele Wartezimmer

- jede Semaphore ist gleichzeitig ein Wartezimmer
- und jeder Wecker (`bell`) ebenfalls
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt
  - → einfache Implementierung, da wir beim Wecker die neuen Ablaufplanmethoden für die Semaphoren verwenden können.
    Wirkt aber auf den ersten Blick halt komisch.
- Umsetzung durch Ableitung von `waitingroom`
- Wecker werden zur Laufzeit als temporäre Objekte erzeugt

(viel zu) viele Wartezimmer

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

(viel zu) viele Wartezimmer

- jede Semaphore ist gleichzeitig ein Wartezimmer
- und jeder Wecker (`bell`) ebenfalls
  - auch wenn bei unseren Weckern nur je ein Faden darin weilt
  - → einfache Implementierung, da wir beim Wecker die neuen Ablaufplanmethoden für die Semaphoren verwenden können.
    Wirkt aber auf den ersten Blick halt komisch.
- Umsetzung durch Ableitung von `waitingroom`
- Wecker werden zur Laufzeit als temporäre Objekte erzeugt (d.h. sie liegen auf dem Stapelspeicher)
- Alle aktiven Wecker werden selbst wieder in einer verketteten Liste (vom `bellringer`) verwaltet

Richtig ticken

Der `bellringer` prüft regelmäßig die Wecker

Richtig ticken

Der `bellringer` prüft regelmäßig die Wecker

- unter Verwendung des LAPIC Timers

Richtig ticken

Der `bellringer` prüft regelmäßig die Wecker

- unter Verwendung des LAPIC Timers
- welcher mit `windup(1000)` auf Millisekundentakt gestellt wird

Richtig ticken

Der `bellringer` prüft regelmäßig die Wecker

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

Leerlauf

Problem: zu wenig Threads bereit

Leerlauf

Problem: zu wenig Threads bereit

Lösung: je ein `IdleThread` pro CPU

Leerlauf

Problem: zu wenig Threads bereit

Lösung: je ein `IdleThread` pro CPU

```
void IdleThread::action() {
    while (true) {
        if (!Scheduler::isEmpty())
            GuardedScheduler::resume();
    }
}
```

Leerlauf

Problem: zu wenig Threads bereit

Lösung: je ein `IdleThread` pro CPU

```
void IdleThread::action() {
    while (true) {
        if (!Scheduler::isEmpty())
            GuardedScheduler::resume();
    }
}
```

CPU fungiert effektiv als Heizkörper

Leerlauf

Problem: zu wenig Threads bereit

Lösung: je ein `IdleThread` pro CPU

```
void IdleThread::action() {
    while (true) {
        if (!Scheduler::isEmpty())
            GuardedScheduler::resume();
    }
}
```

CPU fungiert effektiv als Heizkörper, besser wäre jedoch ein Schlafzustand

Leerlauf

`Core::idle()` hält CPU bis zum nächsten Interrupt an

Leerlauf

`Core::idle()` hält CPU bis zum nächsten Interrupt an

```
void IdleThread::action() {
    while (true) {
        if (Scheduler::isEmpty())
            Core::idle();
        else
            GuardedScheduler::resume();
    }
}
```

Leerlauf

`Core::idle()` hält CPU bis zum nächsten Interrupt an

```
void IdleThread::action() {
    while (true) {
        if (Scheduler::isEmpty())
            Core::idle();
        else
            GuardedScheduler::resume();
    }
}
```

Thread ready

Leerlauf

`Core::idle()` hält CPU bis zum nächsten Interrupt an

```
void IdleThread::action() {
    while (true) {
        if (Scheduler::isEmpty())
            Core::idle();
        else
            GuardedScheduler::resume();
    }
}
```

Thread ready

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

Leerlauf

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

```
void IdleThread::action() {
    while (true) {
        if (Scheduler::isEmpty())
            Core::idle();
        else
            GuardedScheduler::resume();
    }
}
```

Thread ready

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

Leerlauf

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

```
void IdleThread::action() {
    while (true) {
        Core::Interrupt::disable();
        if (Scheduler::isEmpty())
            Core::idle();
        else {
            Core::Interrupt::enable();
            GuardedScheduler::resume();
        }
    }
}
```

Leerlauf mittels `Core::idle()`

```
namespace Core {
    inline void idle() {
        asm volatile("sti\n\t"
                     "hlt\n\t"
                               ::: "memory");
    }
}
```

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();
    Core::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();
    Core::idle();
    Watch::unblock();
}
```

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

Leerlauf in MPStuBS

Problem: Sobald ein Thread bereit ist, soll eine CPU im Leerlauf sofort mit der Abarbeitung beginnen

Leerlauf in MPStuBS

Problem: Sobald ein Thread bereit ist, soll eine CPU im Leerlauf sofort mit der Abarbeitung beginnen

Lösung: Aufwecken der CPU mittels IPI

Weitere Fragen?

Fast geschafft - dies ist die letzte *Pflicht*aufgabe!
Und bitte denkt an die Evaluation für Vorlesung & Übung