Nicht-blockierende Synchronisation in Betriebssystemen

09.02.2021

Fabian Bläse

Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Nicht-blockierende Synchronization

Nachteile blockierender Synchronisation

- Verklemmung
- Performancenachteil durch gegenseitigen Ausschluss
- Benötigt Unterstützung von Hardware / Betriebssystem

Nicht-blockierende Synchronisation

Lock-Free

Wait-Free

Nicht-blockierende Synchronisation

Lock-Free

- Keine Locks
- Verhindert Verklemmung
- Zugriff auf gemeinsame Speicherbereiche über CAS

Wait-Free

Nicht-blockierende Synchronisation

Lock-Free

- Keine Locks
- Verhindert Verklemmung
- Zugriff auf gemeinsame Speicherbereiche über CAS

Wait-Free

- Kein Warten/Wiederholen
- Zugriff hat definierte Maximallänge
- Verhindert Verhungern
- Wait-free ist (meist) auch Lock-free

Lock-free Algorithmus mit CAS

```
1  i := 0
2
3
4  function increment() {
5
6      do {
7          oldval := i
8          newval := i + 1
9      } while ( CAS(&i, oldval, newval) == false )
10
11 }
```

Native: Lock-Free Scheduler Queue

Lock-Free Scheduler Queue

Anforderungen

- Warteschlange ohne Kapazitätsgrenze
- Wiederverwendbare Listenelemente
- Lock-Free

Lock-Free Scheduler Queue

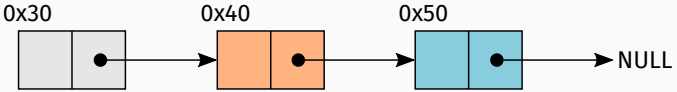
Anforderungen

- Warteschlange ohne Kapazitätsgrenze
- Wiederverwendbare Listenelemente
- Lock-Free

Problem

- Wiederverwendung von Listenelementen macht ABA-Problem wahrscheinlich

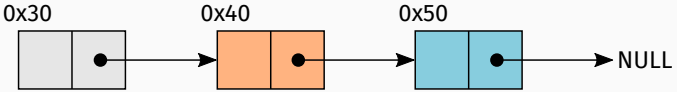
ABA Problem



Thread 1

Thread 2

ABA Problem

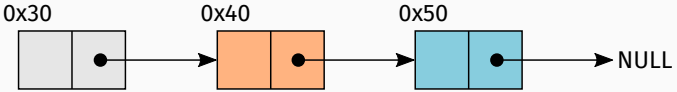


item: 0x40

Thread 1

Thread 2

ABA Problem



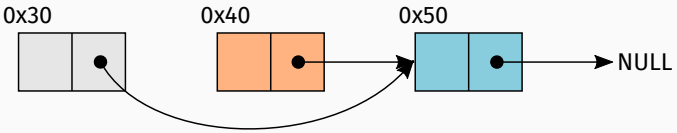
item: 0x40

next: 0x50

Thread 1

Thread 2

ABA Problem



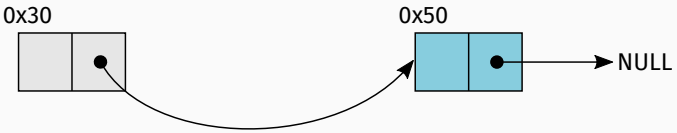
item: 0x40

next: 0x50

Thread 1

Thread 2

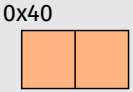
ABA Problem



item: 0x40

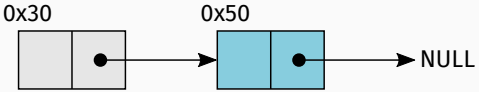
next: 0x50

Thread 1



Thread 2

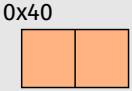
ABA Problem



item: 0x40

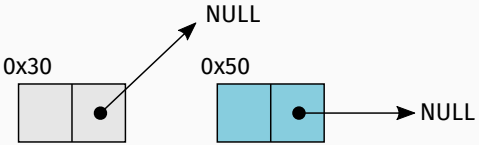
next: 0x50

Thread 1



Thread 2

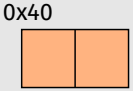
ABA Problem



item: 0x40

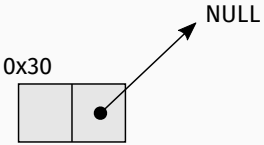
next: 0x50

Thread 1



Thread 2

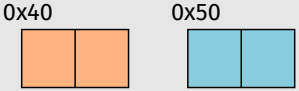
ABA Problem



item: 0x40

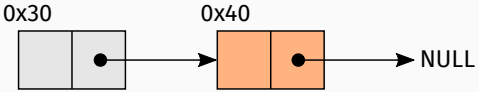
next: 0x50

Thread 1



Thread 2

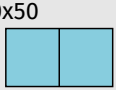
ABA Problem



item: 0x40

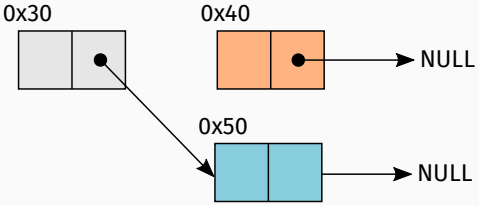
next: 0x50

Thread 1



Thread 2

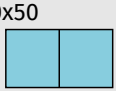
ABA Problem



item: 0x40

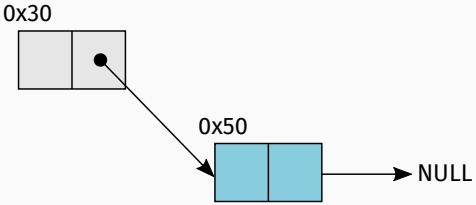
next: 0x50

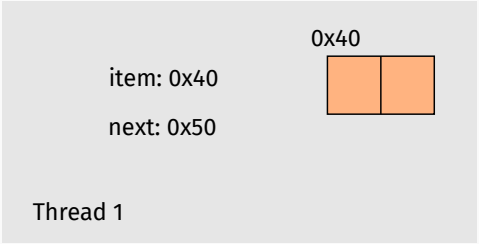
Thread 1

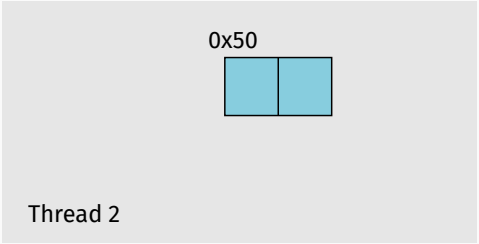


Thread 2

ABA Problem


0x30
0x50
NULL


0x40
item: 0x40
next: 0x50
Thread 1


0x50
Thread 2

Hazard Pointer

Idee

Wiederverwendung von Speicheradressen verhindern, die noch von anderen Threads verwendet werden.

Hazard Pointer

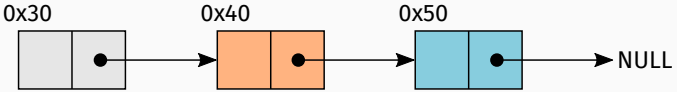
Idee

Wiederverwendung von Speicheradressen verhindern, die noch von anderen Threads verwendet werden.

Umsetzung

- Vermerke Pointer, die gerade vom Thread verwendet werden
- In Thread-lokaler Datenstruktur
- Prüfe vor Wiederverwendung die Hazard Pointer aller anderen Threads

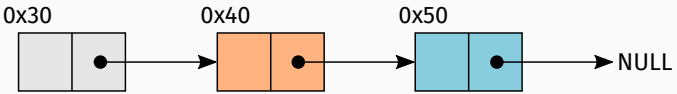
Hazard Pointer



Thread 1

Thread 2

Hazard Pointer



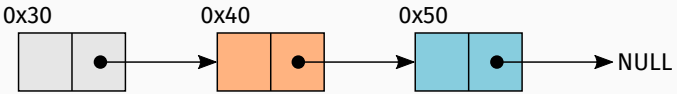
item: 0x40

Thread 1

HP: 0x40

Thread 2

Hazard Pointer



item: 0x40

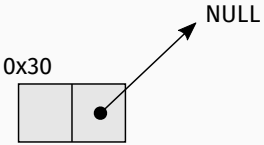
next: 0x50

Thread 1

HP: 0x40

Thread 2

Hazard Pointer

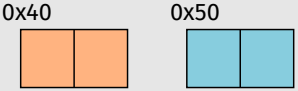


item: 0x40

next: 0x50

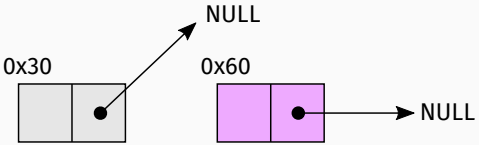
Thread 1

HP: 0x40



Thread 2

Hazard Pointer

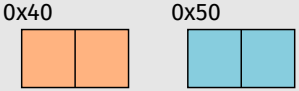


item: 0x40

next: 0x50

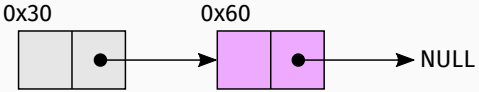
Thread 1

HP: 0x40



Thread 2

Hazard Pointer

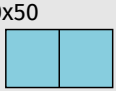


item: 0x40

next: 0x50

Thread 1

HP: 0x40



Thread 2

Hazard Pointer im Betriebssystem

Problem

Hazard Pointer pro Thread nötig. Zugriff auf Thread-lokale Datenstrukturen.

Hazard Pointer im Betriebssystem

Problem

Hazard Pointer pro Thread nötig. Zugriff auf Thread-lokale Datenstrukturen.

Lösung: Kooperatives Scheduling

- Kontextwechsel nur an zur Compilezeit bekannten Codestellen
- Ausschluss von Kontextwechsel während Listenoperation

$\rightarrow$ Nur ein Hazard Pointer pro Prozessor

Implizit kooperatives Scheduling

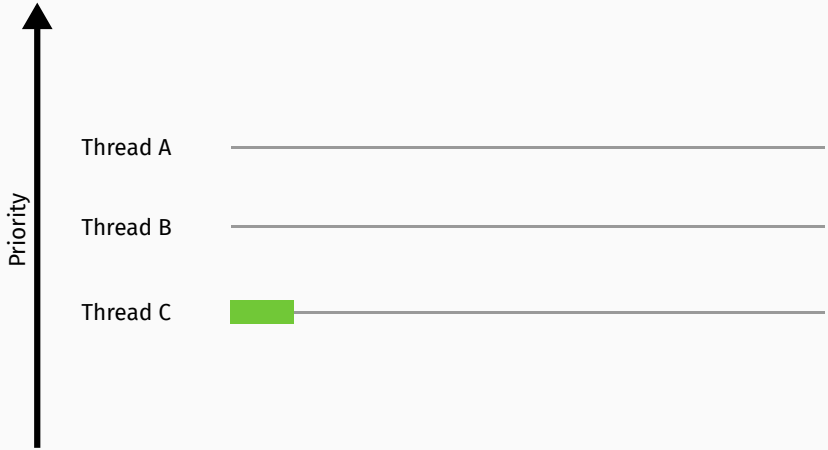
```
1    sub   [rcx + 88], 10
2    jge   skip
3    call  Switch
4  skip:
```

- Automatisches Einfügen der Scheduler-Aufrufe
- Software-Instruktionszähler, um Häufigkeit der Kontextwechsel zu bestimmen
- Transparent gegenüber dem Nutzer

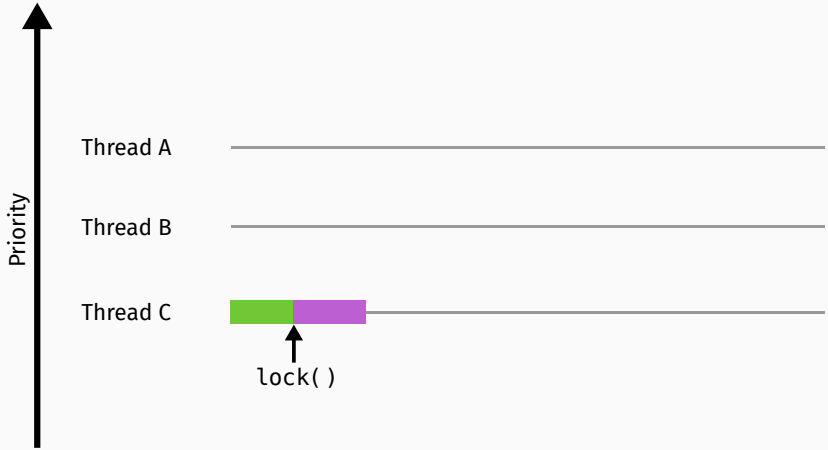
entnommen aus Negele et. al (”On the Design and Implementation of an Efficient Lock-Free Scheduler”)

Fiasco: Wait-Free Locking with Helping

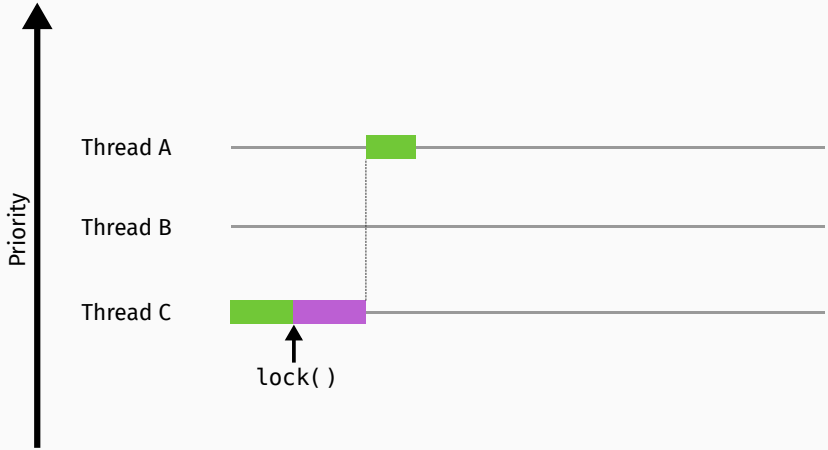
Prioritätsumkehr



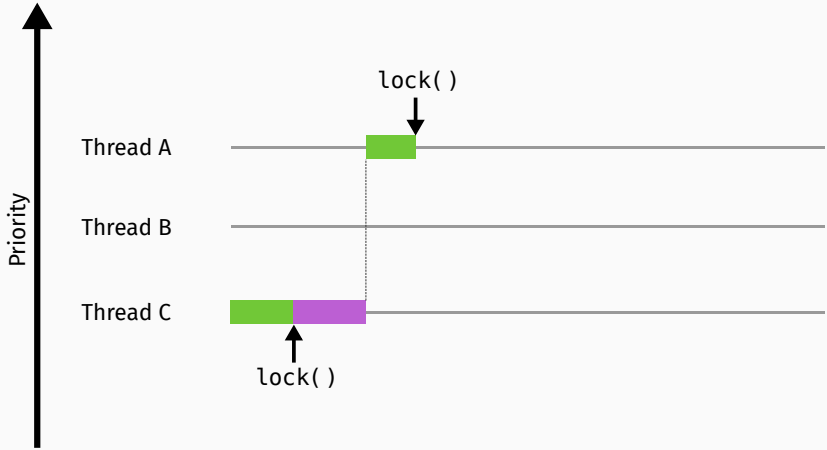
Prioritätsumkehr



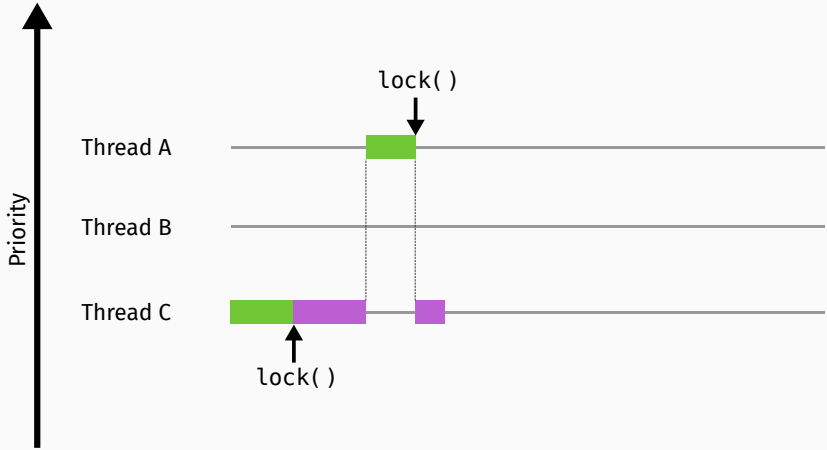
Prioritätsumkehr



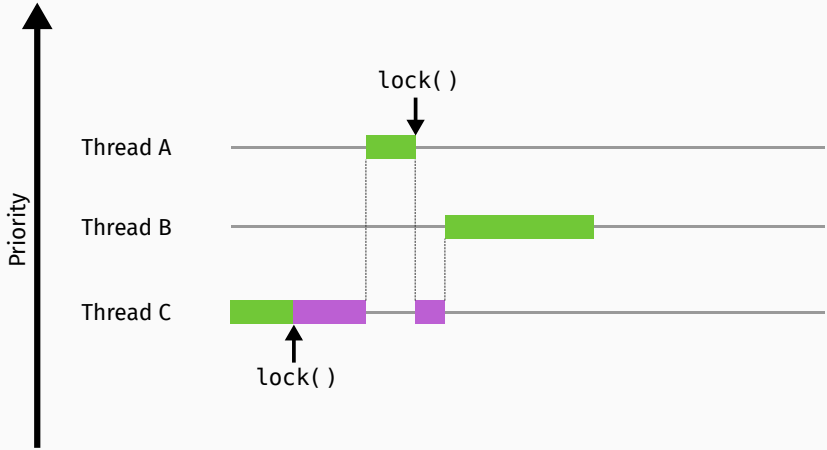
Prioritätsumkehr



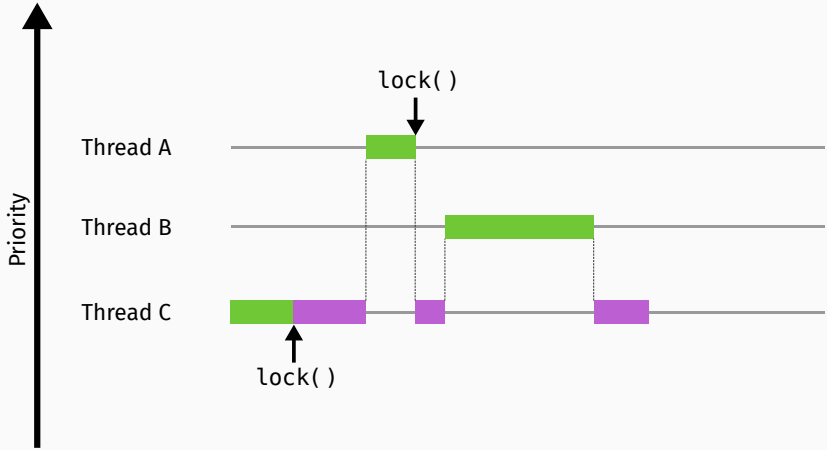
Prioritätsumkehr



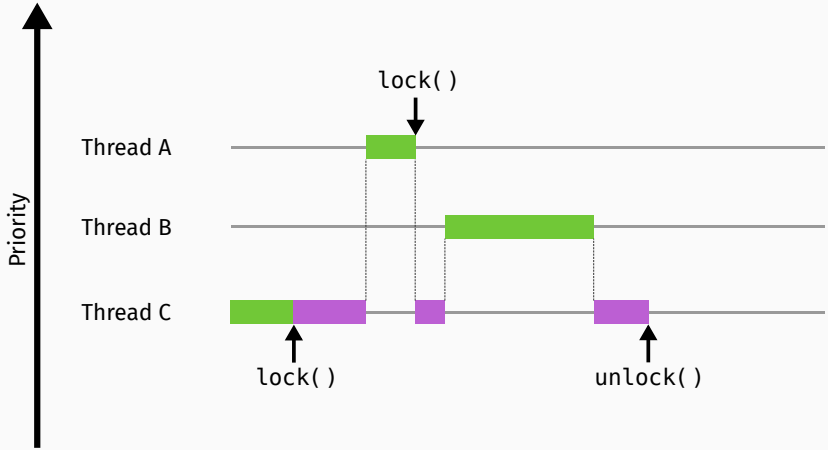
Prioritätsumkehr



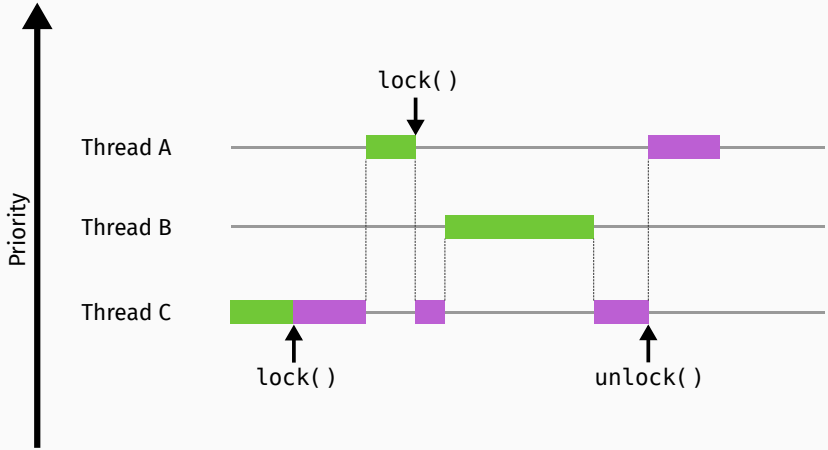
Prioritätsumkehr



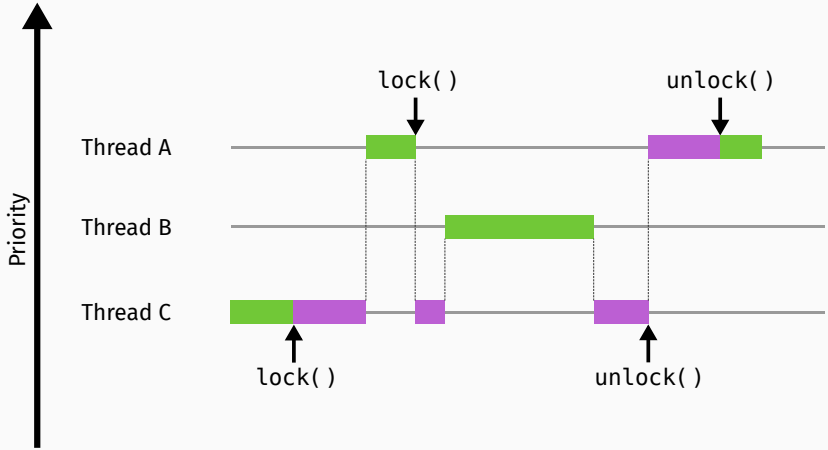
Prioritätsumkehr



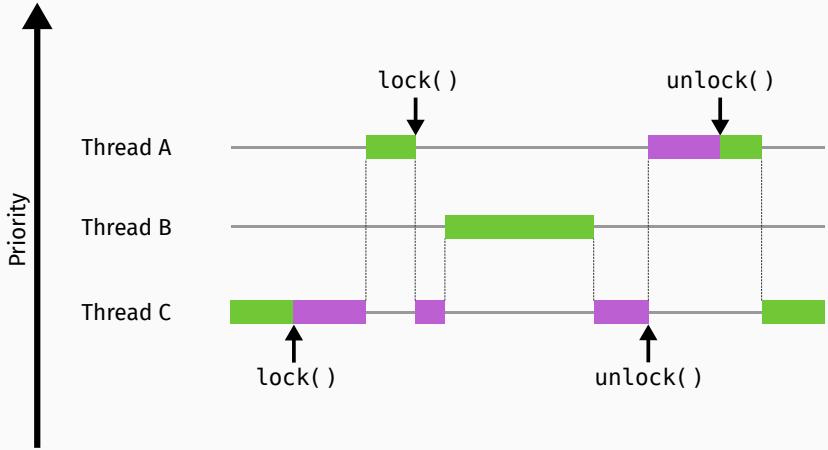
Prioritätsumkehr



Prioritätsumkehr



Prioritätsumkehr



Wait-Free Locking with Helping

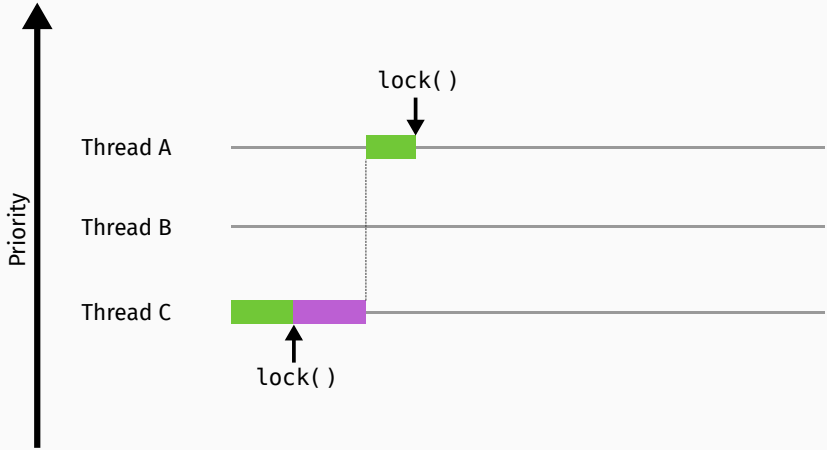
Anforderungen

- Verhindern von Prioritätsumkehr
- Verwendung von Locks
- Priorität vererben, statt blockieren

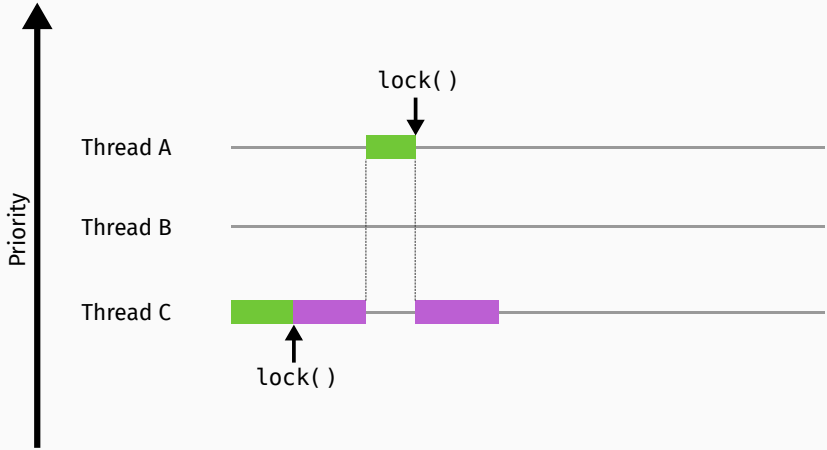
Eigenschaften

+ Einfach zu implementieren
- Verklemmung weiterhin möglich

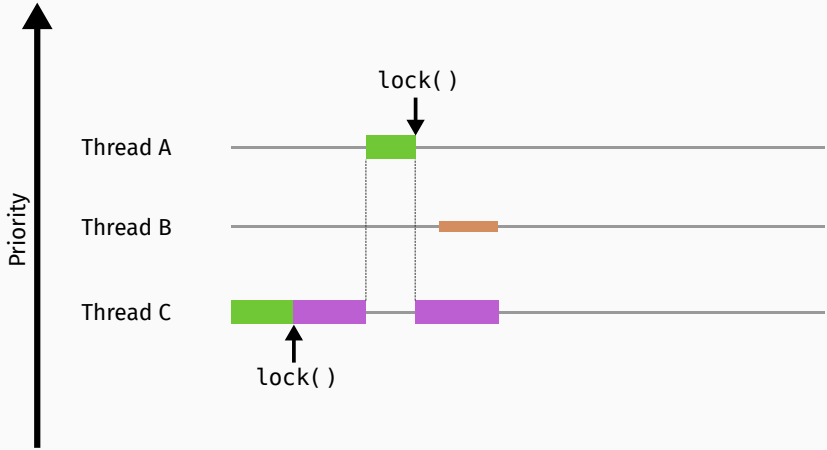
Wait-Free Locking with Helping



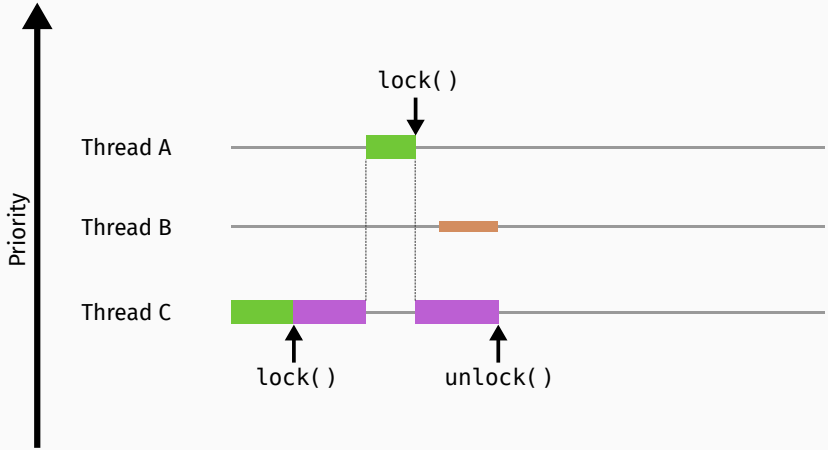
Wait-Free Locking with Helping



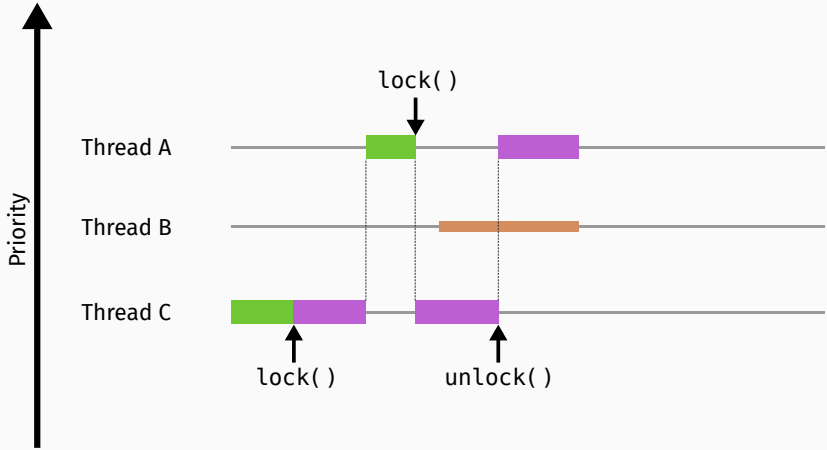
Wait-Free Locking with Helping



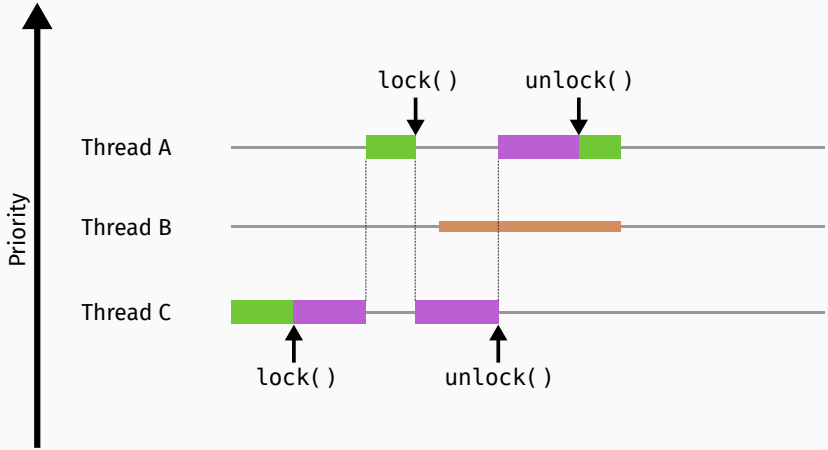
Wait-Free Locking with Helping



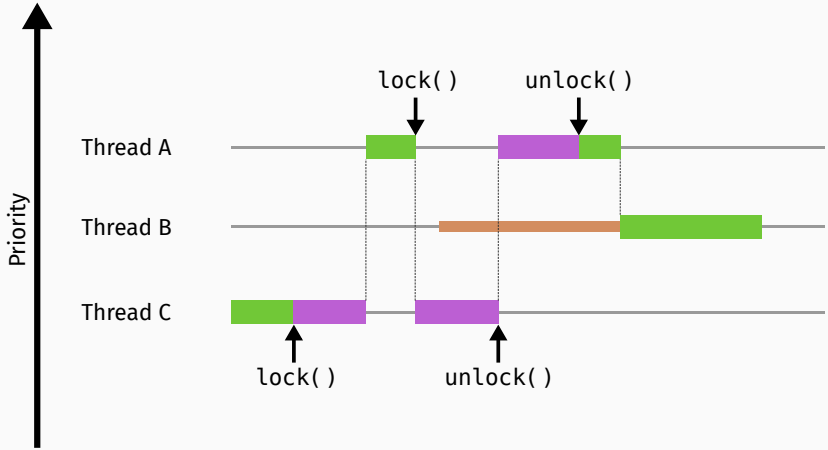
Wait-Free Locking with Helping



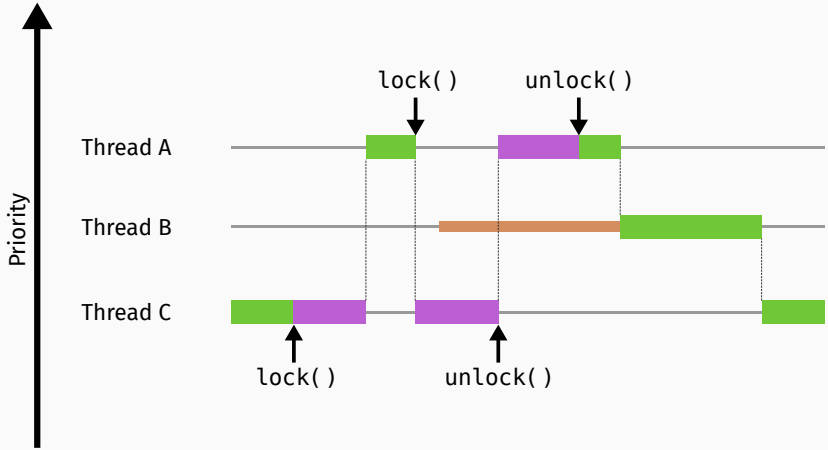
Wait-Free Locking with Helping



Wait-Free Locking with Helping

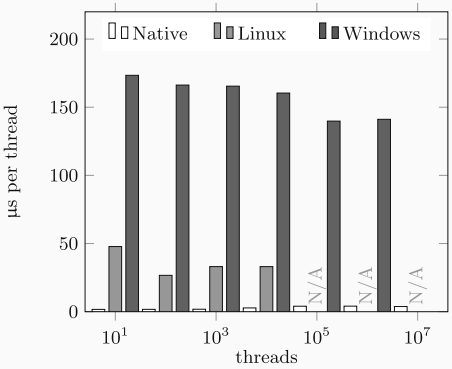


Wait-Free Locking with Helping



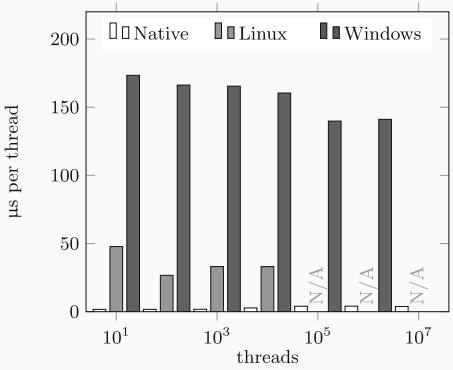
Evaluation

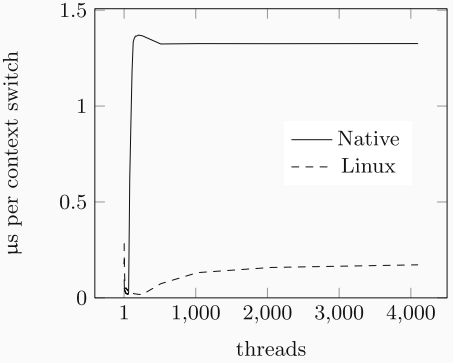
Performance: Native



entnommen aus Negele et. al ("On the Design and Implementation of an Efficient Lock-Free Scheduler")

Performance: Native





entnommen aus Negele et. al ("On the Design and Implementation of an Efficient Lock-Free Scheduler")

Performance: Fiasco

System	Max. Verzögerung
Fiasco µ-Kernel / L⁴Linux	65 µs
L4/x86 / L⁴Linux	541 µs
RTLinux	58 µs

entnommen aus Hohmuth und Härtig (”Pragmatic Nonblocking Synchronization for Real-Time Systems”)

Portabilität und Code-Komplexität

Component	Lines	Ratio
Interrupt Handling	299	75%
Memory Management	357	20%
Modules	82	17%
Multiprocessing	215	37%
Runtime Support	174	12%
Scheduler	547	37%
Total	1674	27%

entnommen aus Negele (”Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System”)

Portabilität und Code-Komplexität

Component	Lines	Ratio
Interrupt Handling	299	75%
Memory Management	357	20%
Modules	82	17%
Multiprocessing	215	37%
Runtime Support	174	12%
Scheduler	547	37%
Total	1674	27%

entnommen aus Negele (”Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System”)

Anwendbarkeit

Lock-free Scheduler Queue

Wait-free locking with helping

Anwendbarkeit

Lock-free Scheduler Queue

+ Lock-free
+ Hardwareunabhängig
- Nur kooperatives Scheduling

Wait-free locking with helping

Anwendbarkeit

Lock-free Scheduler Queue

- \+ Lock-free
- \+ Hardwareunabhängig
- \- Nur kooperatives Scheduling

Wait-free locking with helping

- \+ Löst Prioritätsumkehr
- \+ Keine Anpassung nötig
- \- Verklemmung weiterhin möglich

Zusammenfassung: Nicht-blockierende Betriebssysteme

- Vorteile nicht-blockierender Synchronisation
  1. Keine Verklemmung
  2. Keine Prioritätsumkehr
  3. Hardwareunabhängig
- Eigenschaften und Einschränkungen nicht-blockierender Datenstrukturen
  1. Lock-free, Wait-free
  2. ABA-Problem
- Anwendung nicht-blockierender Datenstrukturen im Betriebssystem
  1. **Native**: Lock-free unbounded scheduler queue
  2. **Fiasco**: Wait-free locking with helping

Fragen?