Futexe
für partitionierte, sichere Betriebssysteme

Alexander Züpke

alexander.zuepke@hs-rm.de

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

SYSGO
EMBEDDING INNOVATIONS

Übersicht

- Futex Grundlagen
- Futexe für partitionierte Systeme
- Implementierungsdetails
  - Mutexe
  - Condition Variablen
  - ...
- Aktuelle Forschung
- Zusammenfassung

Futex Konzept

- Ziel: Mutexe für Linux Anwendungen
  - Performant
  - Wenig Overhead
- Idee
  - Platziere die Mutexe

    in den Userspace oder in ein SHM
- Futex → ***fast userspace mutex***

Franke, Russell, Kirkwood:

Fuss, futex and furwocks: Fast Userlevel Locking in Linux, 2002

Futex Konzept

- Ziel: Mutexe für Linux Anwendungen
  - Performant
  - Wenig Overhead
- Idee
  - Platziere die Mutexe

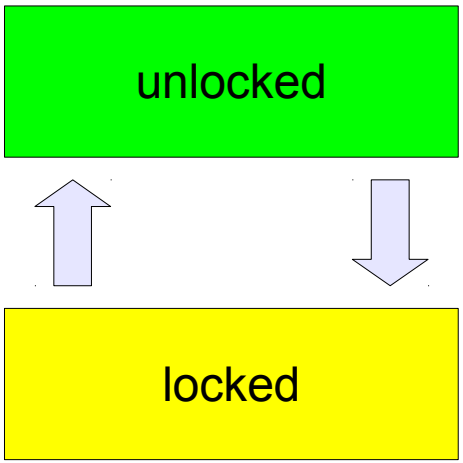
    in den Userspace oder in ein SHM
- Futex → ***fast userspace mutex***

  Franke, Russell, Kirkwood:

  Fuss, futex and furwocks: Fast Userlevel Locking in Linux, 2002

- Wie geht das?

Futex Konzept

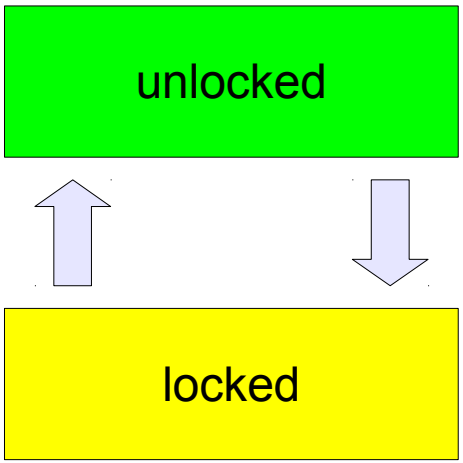


- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

    Zustände werden atomar geändert

- Wie geht das?

Futex Konzept



- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

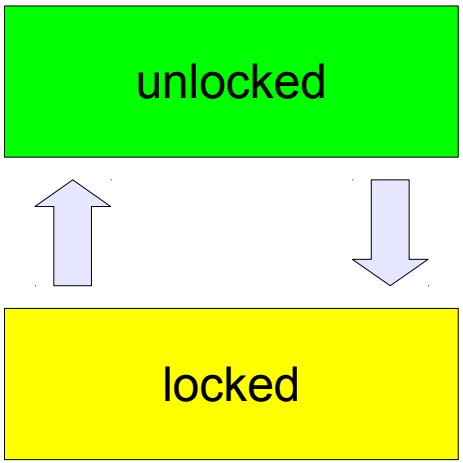
    Zustände werden atomar geändert

`mutex_lock()`
  Atomarer Übergang 0 → 1

`mutex_unlock()`
  Atomarer Übergang 1 → 0

- Wie geht das?

Futex Konzept



- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

    Zustände werden atomar geändert

`mutex_lock()`
  Atomarer Übergang 0 → 1

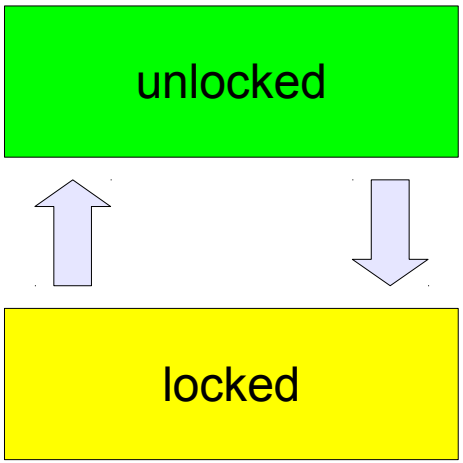
`mutex_unlock()`
  Atomarer Übergang 1 → 0

→ trivial bei nur einem Thread

→ einfach und performant

- Wie geht das?

Futex Konzept



- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

    Zustände werden atomar geändert

`mutex_lock()`
Atomarer Übergang 0 → 1

`mutex_unlock()`
Atomarer Übergang 1 → 0

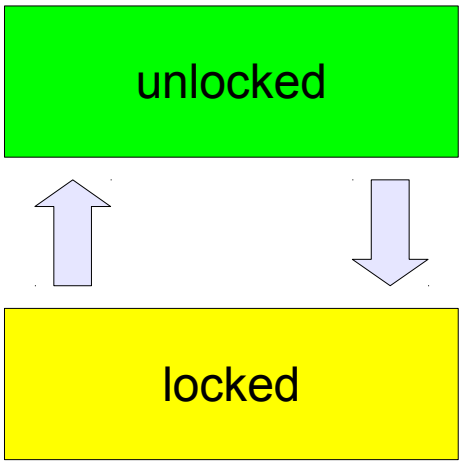
→ trivial bei nur einem Thread

→ einfach und performant

→ was passiert, wenn ein zweiter Thread dazukommt?

- Wie geht das?

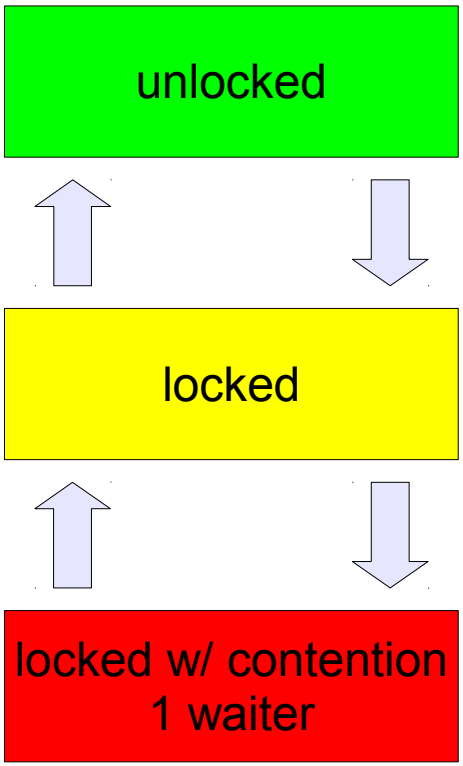
Futex Konzept



- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

    Zustände werden atomar geändert
  - Erster Wartender?

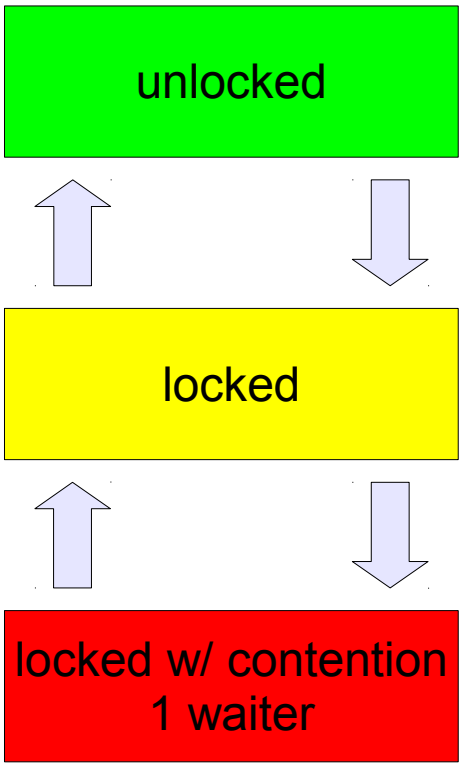
Futex Konzept



- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

    Zustände werden atomar geändert
  - Erster Wartender?

Futex Konzept



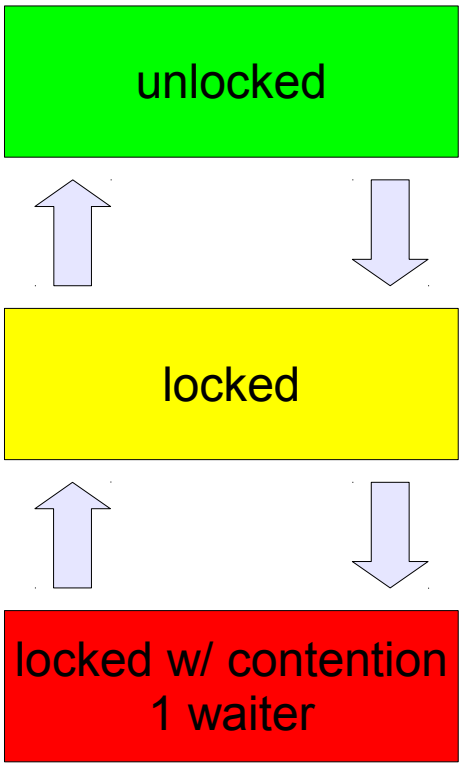
- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

    Zustände werden atomar geändert
  - Erster Wartender:

    → Atomar ”ich warte” anzeigen

    → Suspendieren im Betriebssystemkern

Futex Konzept



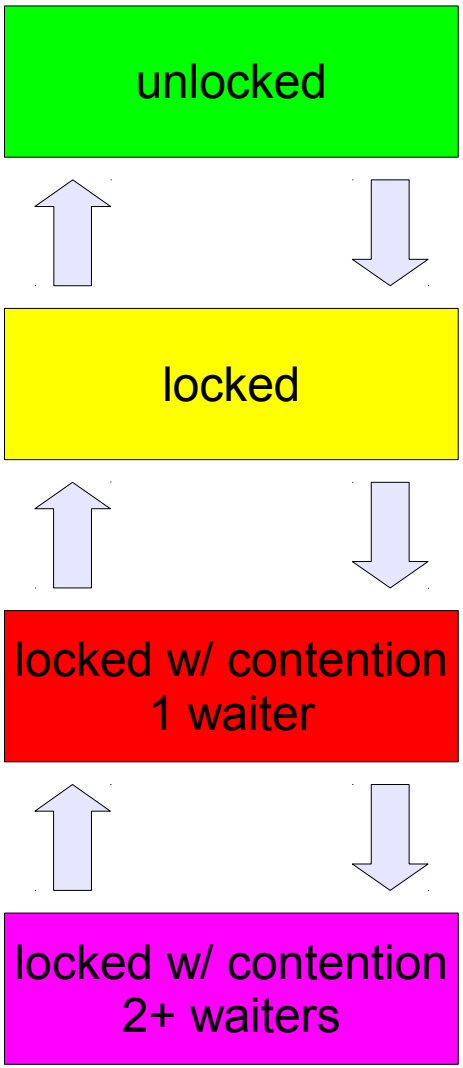
- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

    Zustände werden atomar geändert
  - Erster Wartender:

    → Atomar ”ich warte” anzeigen

    → Suspendieren im Betriebssystemkern
  - Weitere Wartende?

Futex Konzept



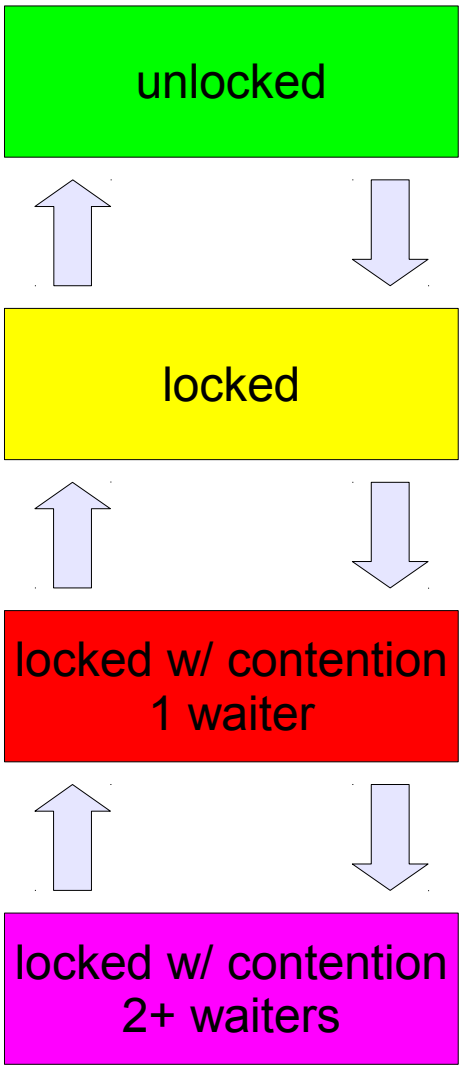
- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

    Zustände werden atomar geändert
  - Erster Wartender:

    → Atomar ”ich warte” anzeigen

    → Suspendieren im Betriebssystemkern
  - Weitere Wartende?

Futex Konzept



- Futexe sind Variablen im Userspace
- Futexe kodieren Zustände!
  - Beispiel Mutex:

    Zustände werden atomar geändert
  - Erster Wartender:

    → Atomar ”ich warte” anzeigen

    → Suspendieren im Betriebssystemkern
  - Weitere Wartende:

    → Suspendieren im Betriebssystemkern

    → bilden eine Warteschlange!

Futexe in Linux

- Futex := 32-bit Integer Variable im Userspace
- Atomare CAS oder LL/SC Operationen
- Die Glibc bietet:
  - Mutexe und Condition Variablen
  - Semaphoren, Reader-Writer Locks, Barriers, …
- Der Linux-Kernel bietet Systemcalls für:
  - Suspendiere aktuellen Thread
  - Wecke eine Anzahl von Wartenden
- Erster Prototyp in Linux Kernel Version 2.5.7

Futexe in Linux

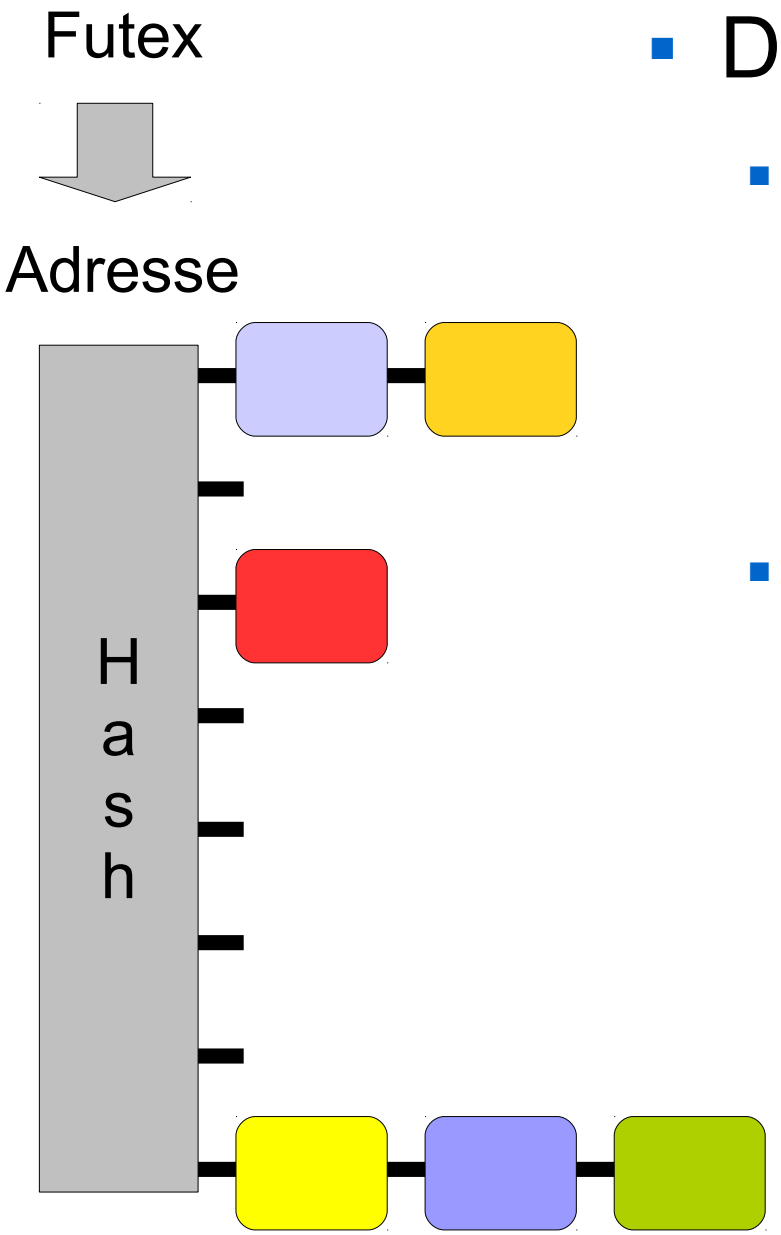
- Futexe in Linux
  - … benötigen Systemcalls nur im Konfliktfall
  - … bieten eine beliebige Anzahl von Futexen
  - … benötigen keine BS-Ressourcen,
    bis der erste Thread wartet
  - … bieten auch Mutexe mit Prioritätsvererbung

Futexe in Linux

- Futexe in Linux
  - … benötigen Systemcalls nur im Konfliktfall
  - … bieten eine beliebige Anzahl von Futexen
  - … benötigen keine BS-Ressourcen,
    bis der erste Thread wartet
  - … bieten auch Mutexe mit Prioritätsvererbung

Wie sieht es im Kernel aus?

Futexe in Linux



- Datenstrukturen im Linux-Kernel
  - Futex-Hash (global)
    - Array fester Größe
    - Liste mit Futex-Objekten
    - Lock pro Hashbucket
  - Futex-Objekt (eins pro Futex)
    - Key (Futex Adresse)
    - Warteschlange
    - Pointer auf Lock
    - Listenpointer

Futexe in Linux

- Futexe in Linux
  - … benötigen Systemcalls nur im Konfliktfall
  - … bieten eine beliebige Anzahl von Futexen
  - … benötigen keine BS-Ressourcen,
    bis der erste Thread wartet
  - … bieten auch Mutexe mit Prioritätsvererbung
- Fazit: Futexe sind toll

Futexe in Linux

- Futexe in Linux
  - … benötigen Systemcalls nur im Konfliktfall
  - … bieten eine beliebige Anzahl von Futexen
  - … benötigen keine BS-Ressourcen,
    bis der erste Thread wartet
  - … bieten auch Mutexe mit Prioritätsvererbung
- Fazit: Futexe sind toll

  … für Un*x-artige Kernel!

Futexe in Linux

- Futexe in Linux
  - … benötigen Systemcalls nur im Konfliktfall
  - … bieten eine beliebige Anzahl von Futexen
  - … benötigen keine BS-Ressourcen,
    bis der erste Thread wartet
  - … bieten auch Mutexe mit Prioritätsvererbung
- **Aber: was ist mit ...**
  - … sicherheitskritische Anwendungen?
  - … partitionierten Systemen?
  - … Mikrokernen?

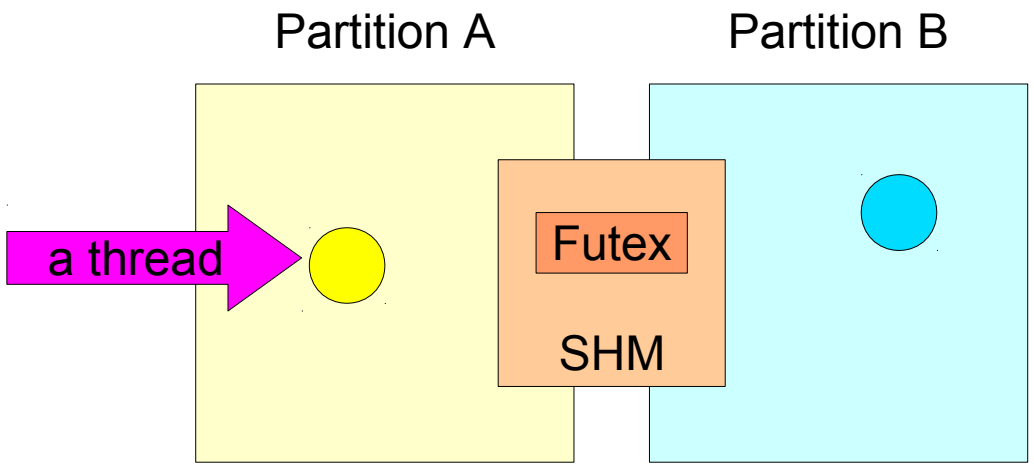
Motivation

- Definition ”Partitionierte Systeme”
  - ***space and time partitioning*** (z.b. ARINC 653)
  - Isolierte (Gruppen von) Prozessen
  - Alle Betriebsmittel sind partitioniert

Motivation

- Definition ”Partitionierte Systeme”
  - ***space and time partitioning*** (z.b. ARINC 653)
  - Isolierte (Gruppen von) Prozessen
  - Alle Betriebsmittel sind partitioniert

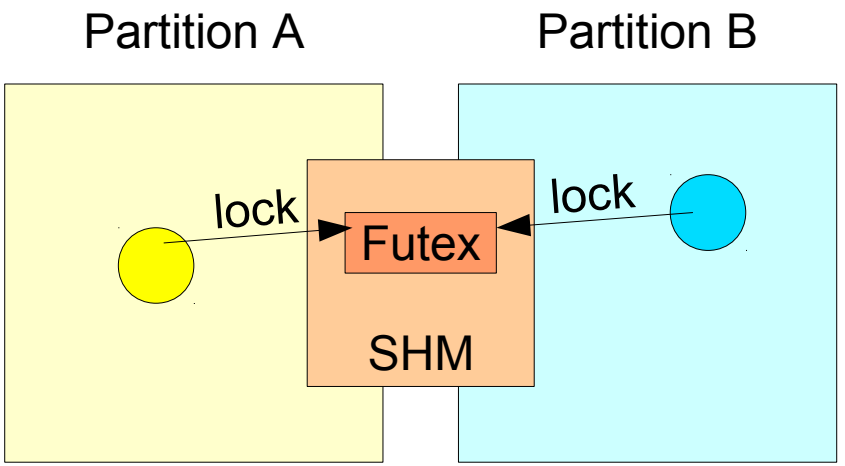
Beispiel



Motivation

- Definition ”Partitionierte Systeme”
  - ***space and time partitioning*** (z.b. ARINC 653)
  - Isolierte (Gruppen von) Prozessen
  - Alle Betriebsmittel sind partitioniert

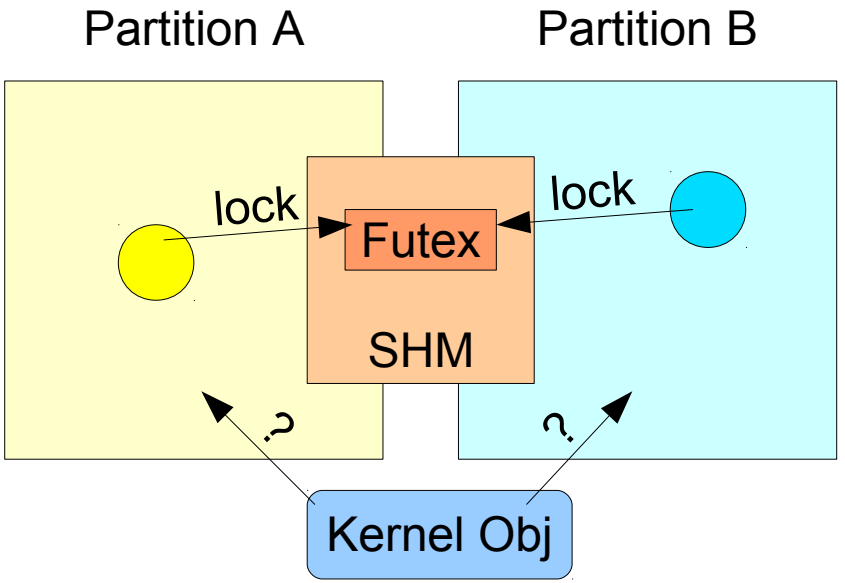
Beispiel



Motivation

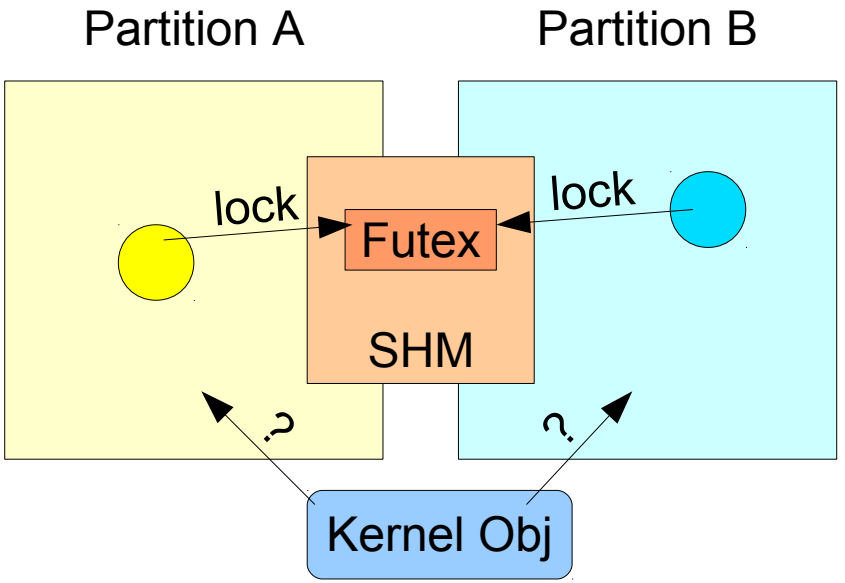
- Definition ”Partitionierte Systeme”
  - ***space and time partitioning*** (z.b. ARINC 653)
  - Isolierte (Gruppen von) Prozessen
  - Alle Betriebsmittel sind partitioniert

Beispiel



Motivation

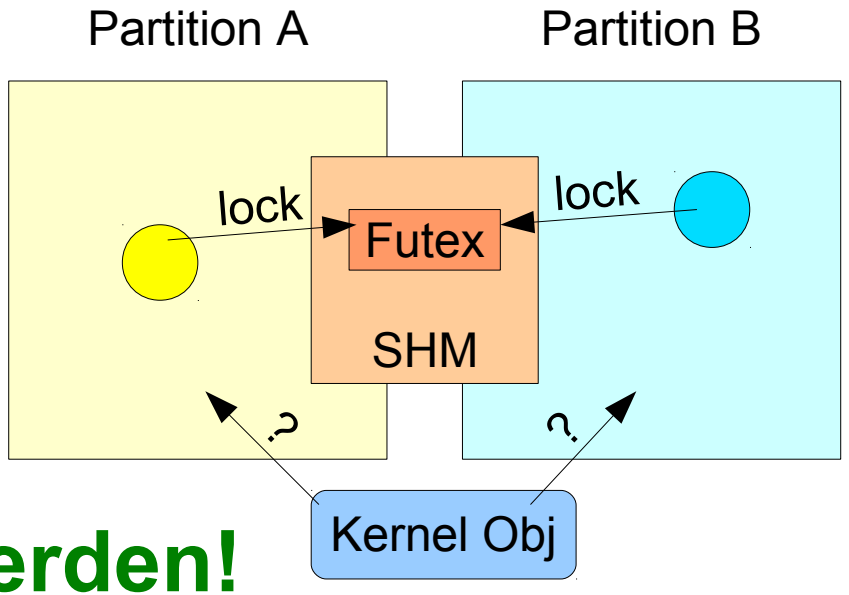
- Definition ”Partitionierte Systeme”
  - ***space and time partitioning*** (z.b. ARINC 653)
  - Isolierte (Gruppen von) Prozessen
  - Alle Betriebsmittel sind partitioniert
- Problem
  - Welcher Partition gehört die Warteschlange?
  - ... vorab allokieren?



Motivation

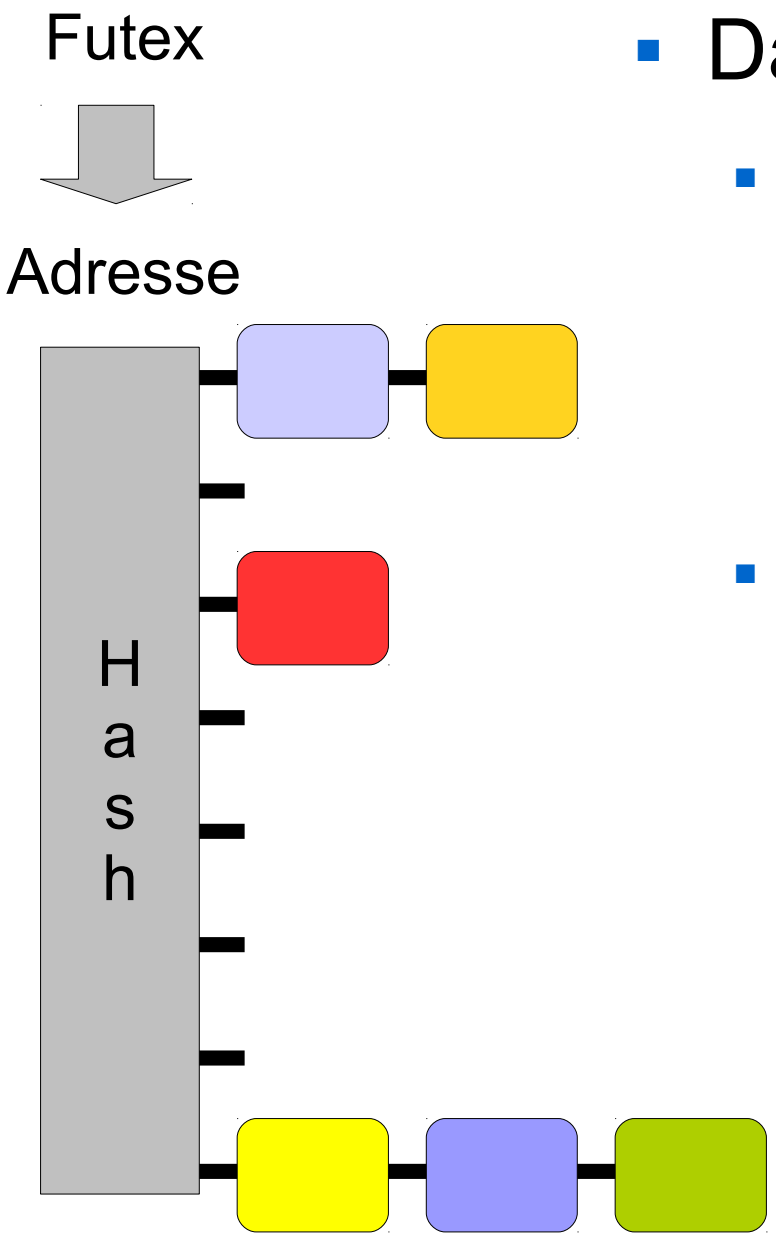
- Definition ”Partitionierte Systeme”
  - ***space and time partitioning*** (z.b. ARINC 653)
  - Isolierte (Gruppen von) Prozessen
  - Alle Betriebsmittel sind partitioniert
- Problem
  - Welcher Partition gehört die Warteschlange?
  - ... vorab allokieren?

    → zu pessimistisch



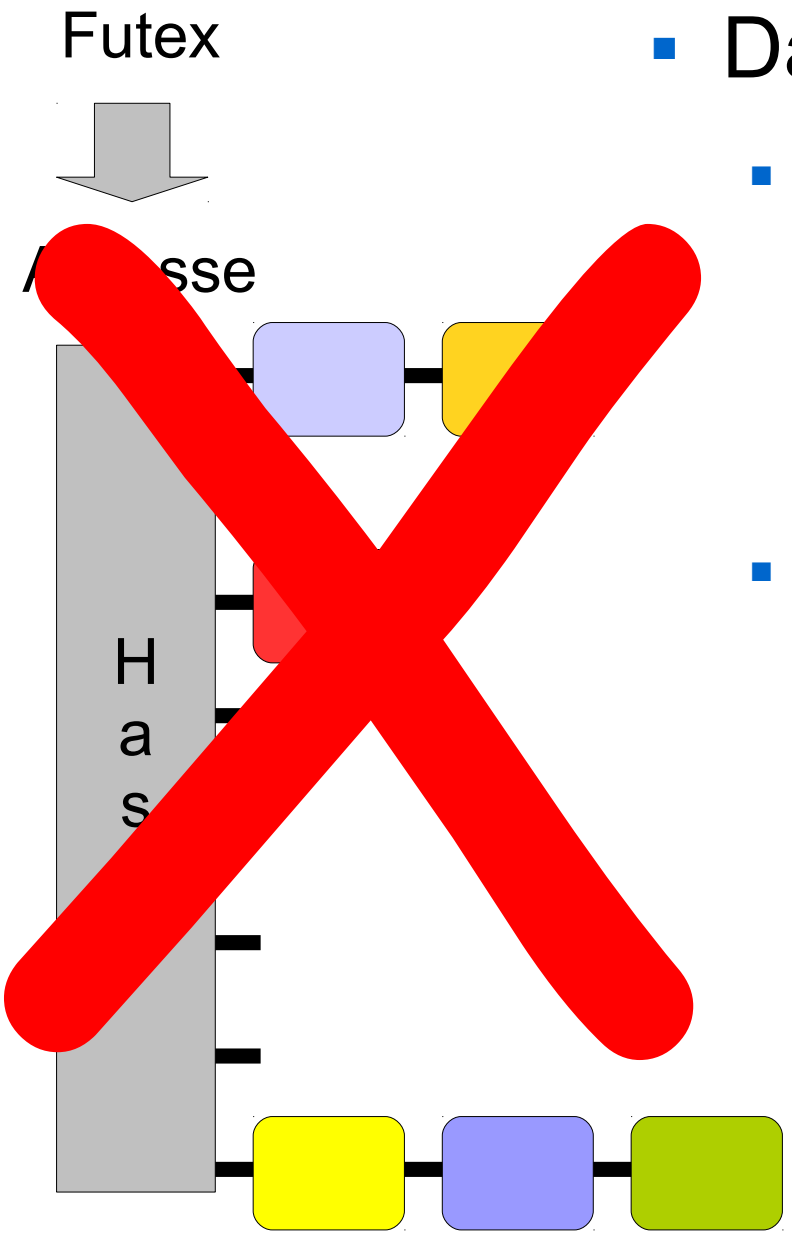
- Idee: **Kernel Objekt loswerden!**

Motivation



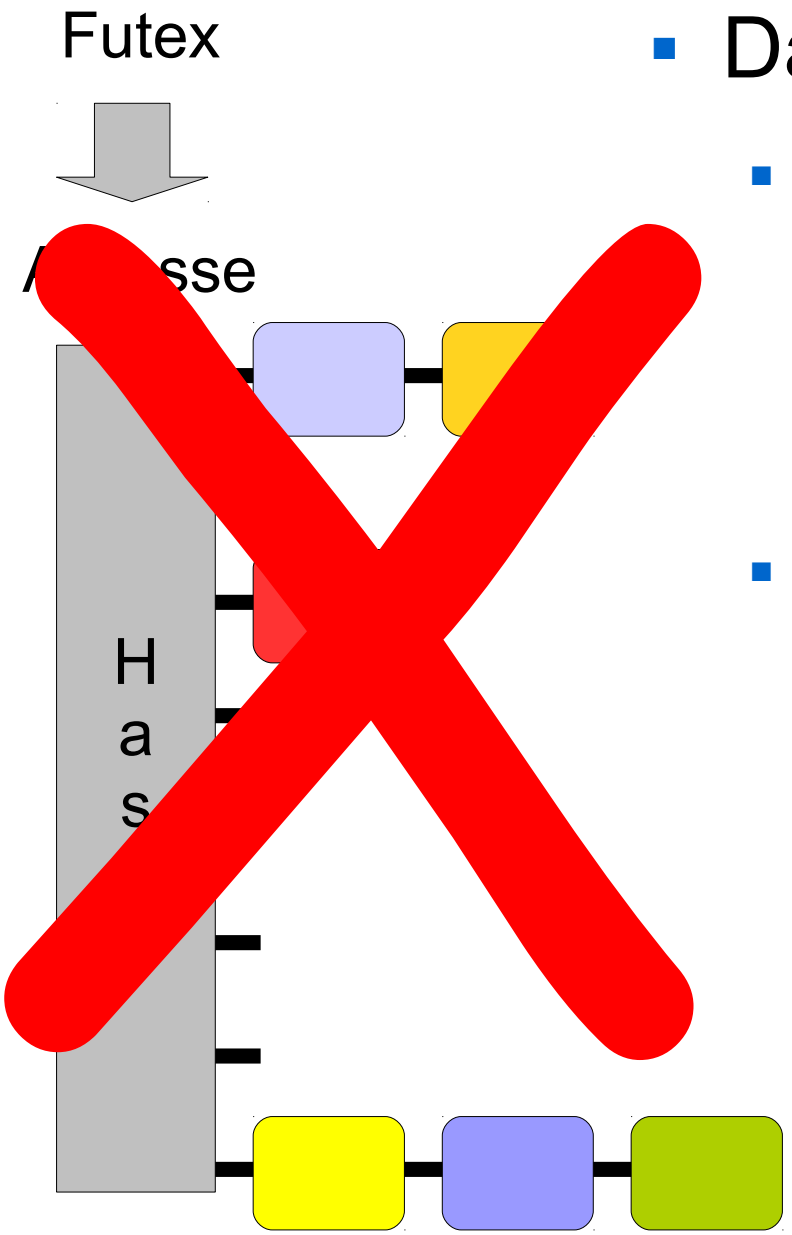
- Datenstrukturen im Linux-Kernel
  - Futex-Hash (global)
    - Array fester Größe
    - Liste mit Futex-Objekten
    - Lock pro Hashbucket
  - Futex-Objekt (eins pro Futex)
    - Key (Futex Adresse)
    - Warteschlange
    - Pointer auf Lock
    - Listenpointer

Motivation



- Datenstrukturen im Linux-Kernel
  - Futex-Hash (global)
    - ~~Array fester Größe~~
    - ~~Liste mit Futex-Objekten~~
    - ~~Lock pro Hashbucket~~
  - Futex-Objekt (eins pro Futex)
    - Key (Futex Adresse)
    - Warteschlange
    - ~~Pointer auf Lock~~
    - ~~Listenpointer~~

Motivation



- Datenstrukturen im Linux-Kernel
  - Futex-Hash (global)
    - ~~Array fester Größe~~
    - ~~Liste mit Futex-Objekten~~
    - ~~Lock pro Hashbucket~~
  - Futex-Objekt (eins pro Futex)
    - Key (Futex Adresse)
    - Warteschlange
    - ~~Pointer auf Lock~~
    - ~~Listenpointer~~

in den TCB damit

Requirements

- Algorithmen mit ”guter” WCET Laufzeit
  - Abhängig von der Anzahl der wartenden Threads
  - **Aber**: Anzahl möglicherweise vorab nicht bekannt

    → schwierig über Partitionsgrenzen hinweg

Requirements

- Algorithmen mit ”guter” WCET Laufzeit
  - Abhängig von der Anzahl der wartenden Threads
  - **Aber**: Anzahl möglicherweise vorab nicht bekannt

    → schwierig über Partitionsgrenzen hinweg
- Warteschlangen
  - Doppelt-verkettete Listen sind O(1)

    … ausgenommen sortiertes Einfügen
  - Sortierte Warteschlangen mit Laufzeit O(log n)

    sind oft akzeptabel
  - O(n) ist nur akzeptabel wenn n begrenzt ist

Requirements

- Algorithmen mit ”guter” WCET Laufzeit
  - Abhängig von der Anzahl der wartenden Threads
  - **Aber**: Anzahl möglicherweise vorab nicht bekannt

    → schwierig über Partitionsgrenzen hinweg
- Warteschlangen
  - Doppelt-verkettete Listen sind O(1)

    … ausgenommen sortiertes Einfügen
  - Sortierte Warteschlangen mit Laufzeit O(log n)

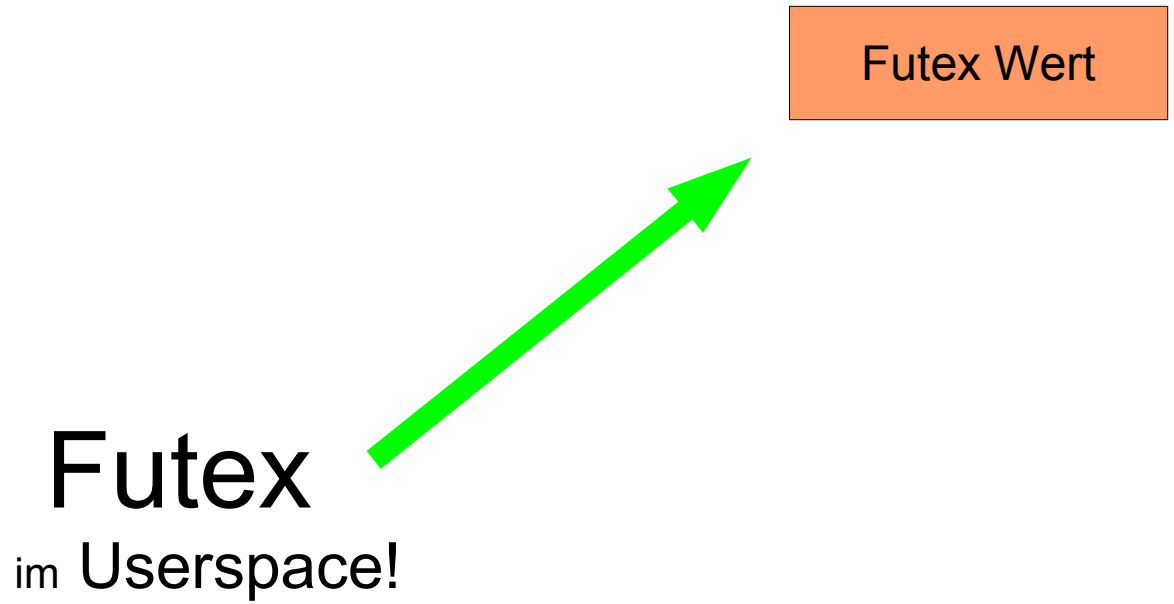
    sind oft akzeptabel
  - O(n) ist nur akzeptabel wenn n begrenzt ist

    → doppelt-verkettete Listen mit FIFO-Reihenfolge

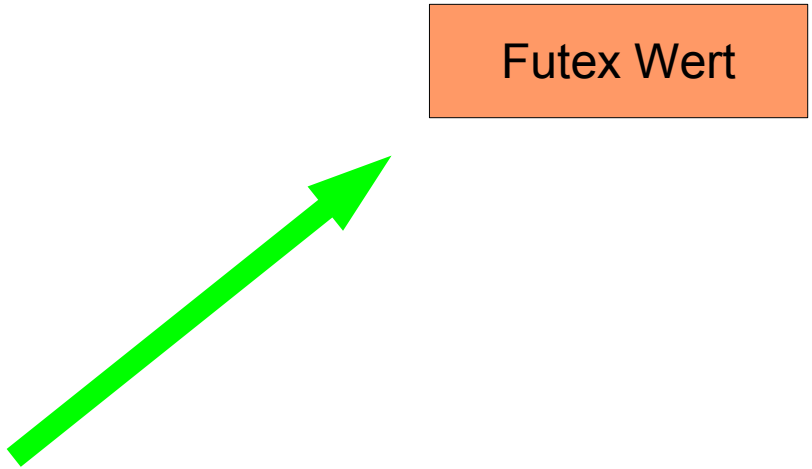
Requirements

Futex

Requirements



Requirements



Futex
im Userspace!

Identifizierbar anhand seiner Adresse

Requirements

- Richtige Warteschlange finden

Futex Wert

Requirements

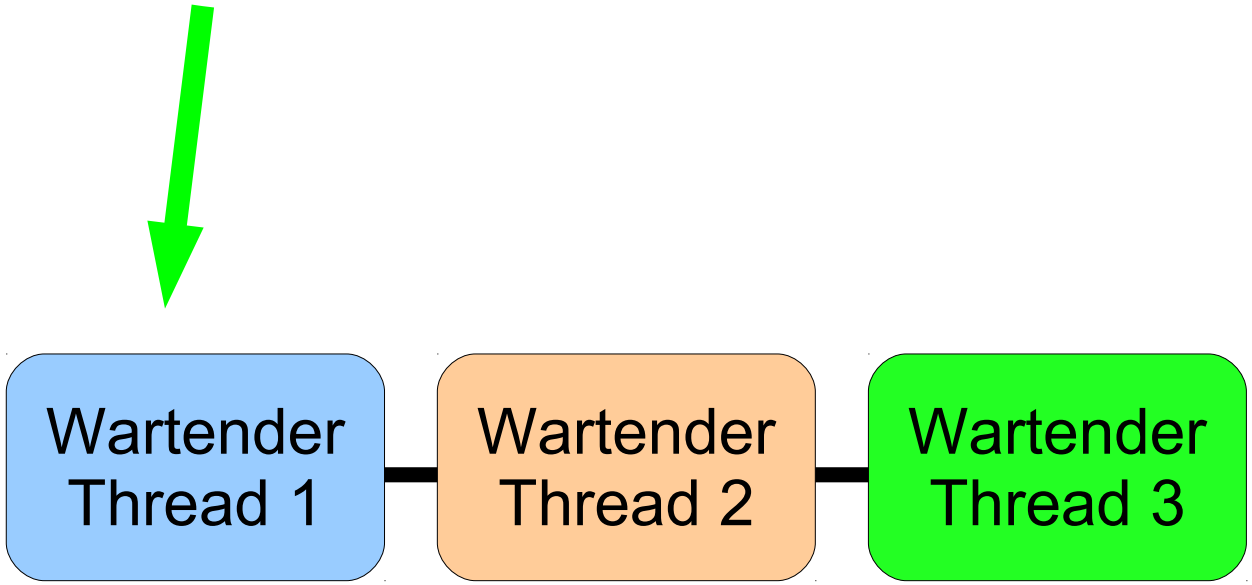
- Richtige Warteschlange finden

Futex Wert

Requirements

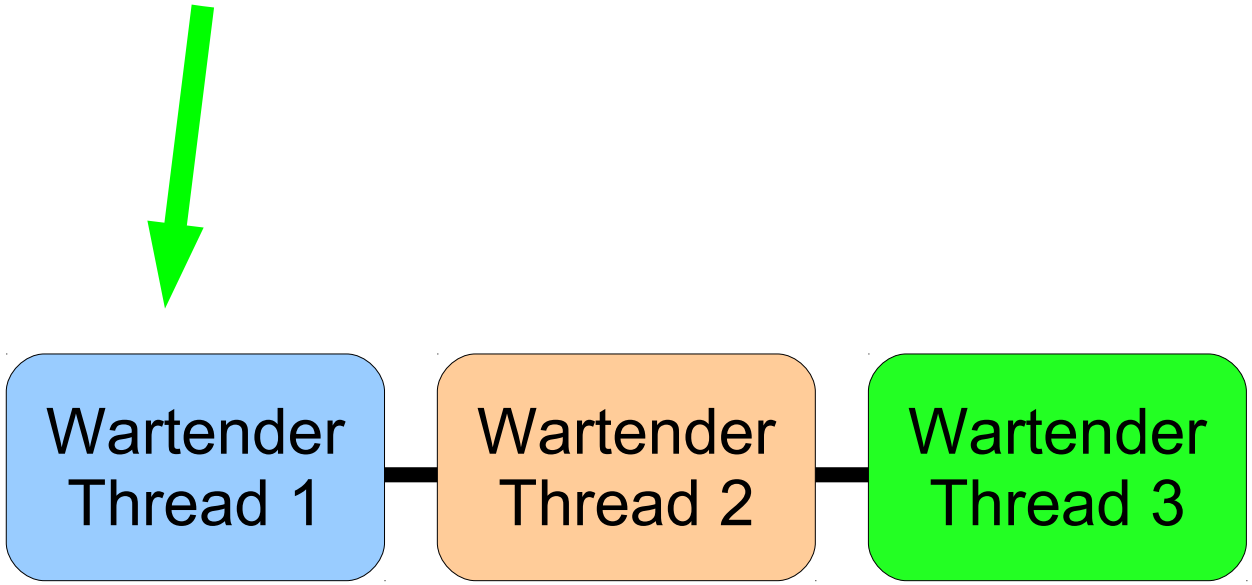
- Richtige Warteschlange finden
  - Thread ID des ersten Wartenden

Futex Wert

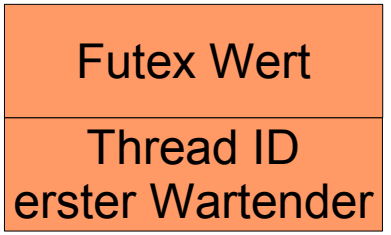


Requirements

- Richtige Warteschlange finden
  - Thread ID des ersten Wartenden
  - **Thread ID im Userspace ablegen**

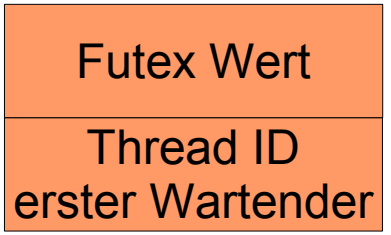


Requirements



- Richtige Warteschlange finden
  - Thread ID des ersten Wartenden
  - **Thread ID im Userspace ablegen**
- Warteschlange mit linearem Speicherverbrauch
  - Sortierung nach Prioritäten wäre nett …
  - Zunächst FIFO → doppelt-verkettete Liste

Requirements



- Richtige Warteschlange finden
  - Thread ID des ersten Wartenden
  - **Thread ID im Userspace ablegen**
- Warteschlange mit linearem Speicherverbrauch
  - Sortierung nach Prioritäten wäre nett …
  - Zunächst FIFO → doppelt-verkettete Liste

- Internes Locking der Warteschlange
  - Vereinfachte Annahme: Global Kernel Lock ...

- Mutex-Implementierung ?
- Condition Variablen ?
- Locking der Warteschlange ?
- Robustness ?
- Sortiert nach Priorität ?

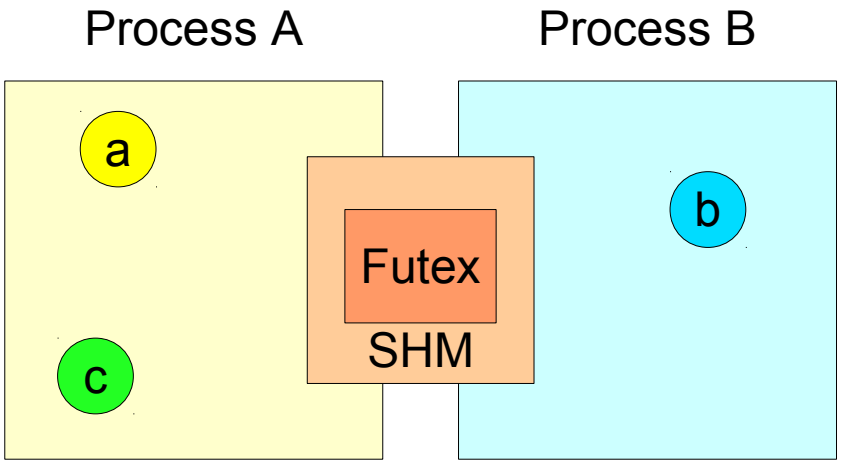
- **Mutex-Implementierung ?**
- Condition Variablen ?
- Locking der Warteschlange ?
- Robustness ?
- Sortiert nach Priorität ?

Mutex Protokoll

- Beispiel
  - 2 Prozesse
  - 3 Threads
  - Futex im Shared Memory
  - Mutex Protokol

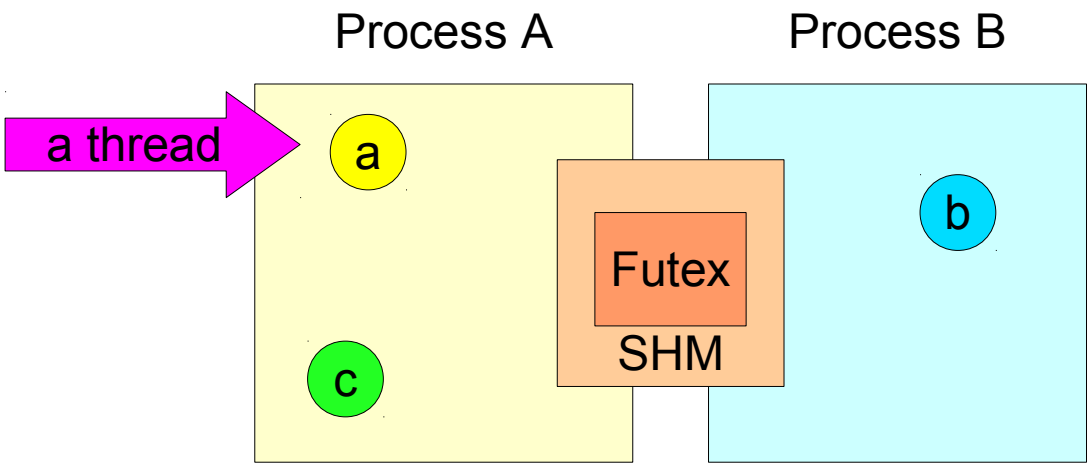
Mutex Protokoll

- Beispiel
  - 2 Prozesse
  - 3 Threads
  - Futex im Shared Memory
  - Mutex Protokol



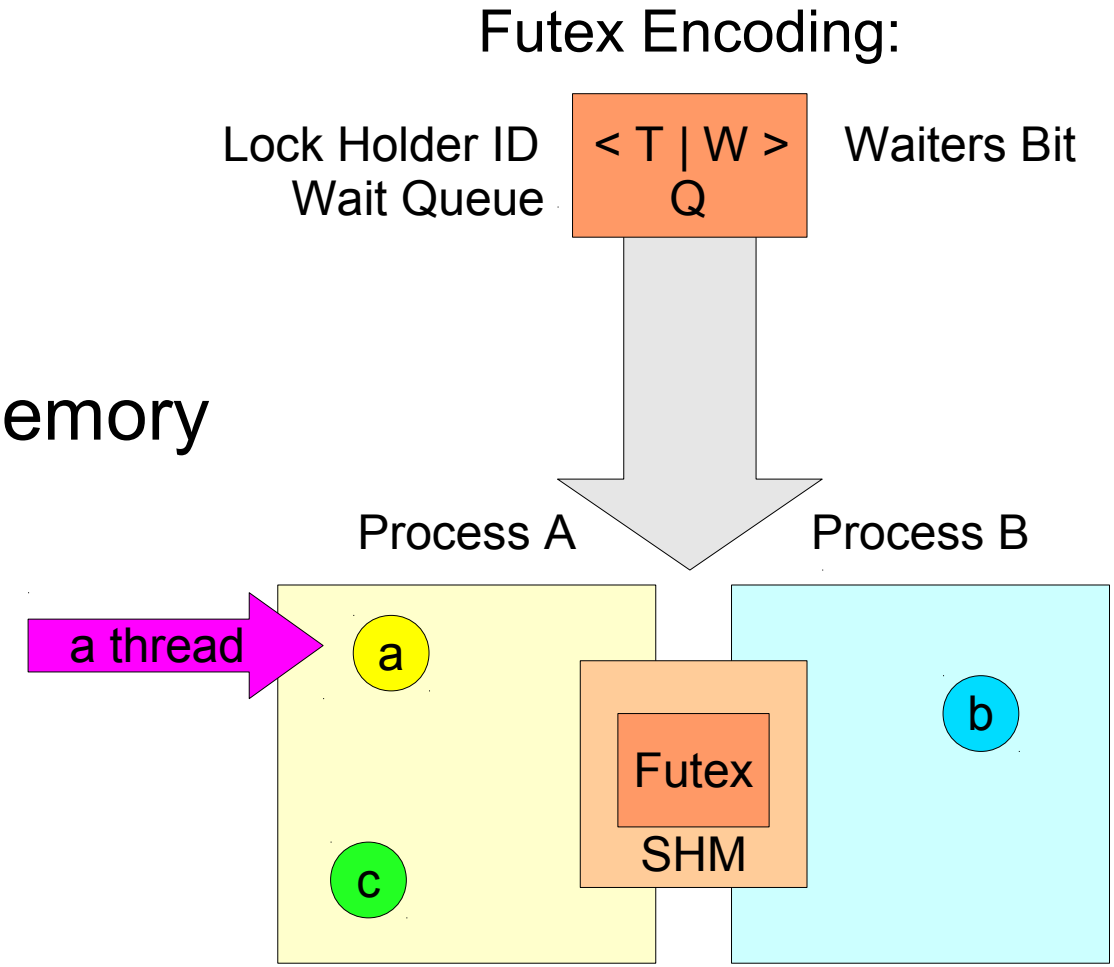
Mutex Protokoll

- Beispiel
  - 2 Prozesse
  - 3 Threads
  - Futex im Shared Memory
  - Mutex Protokol



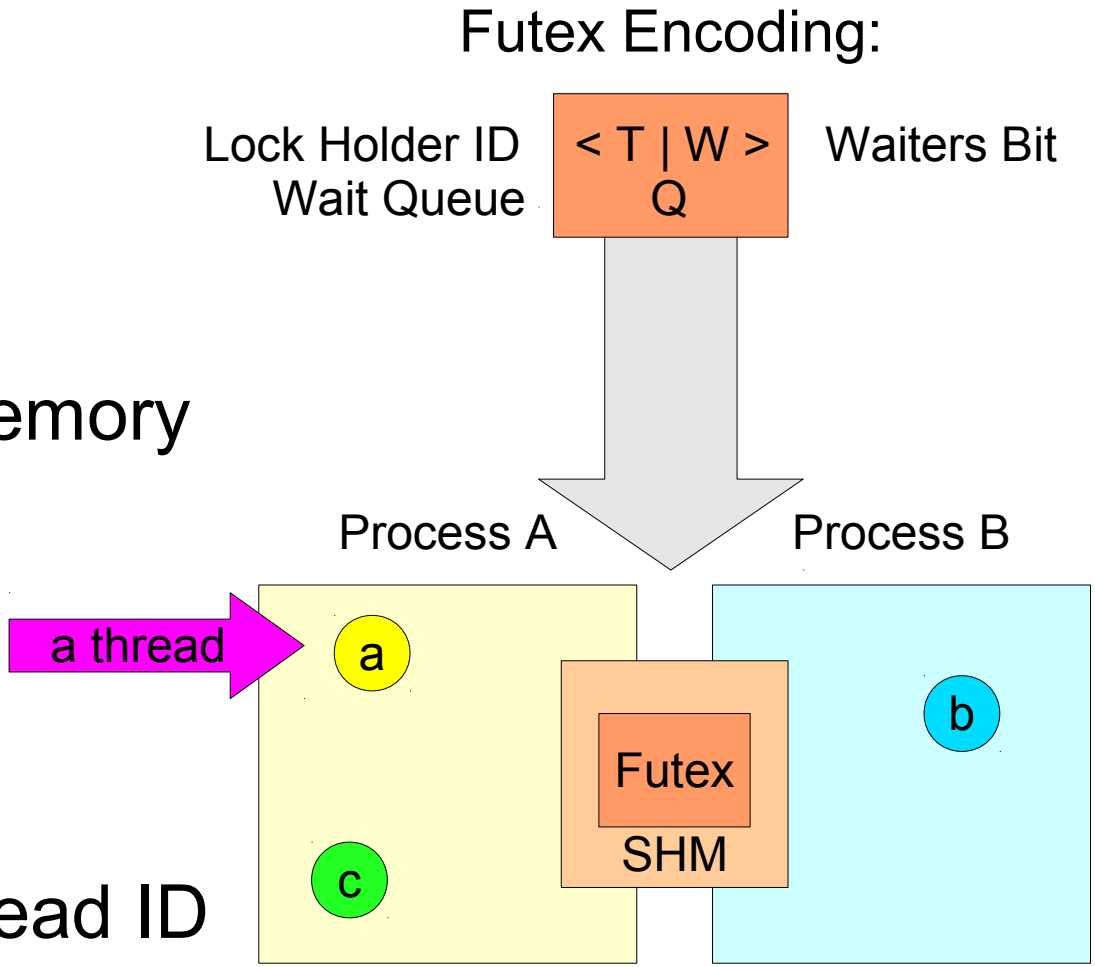
Mutex Protokoll

- Beispiel
  - 2 Prozesse
  - 3 Threads
  - Futex im Shared Memory
  - Mutex Protokol



Mutex Protokoll

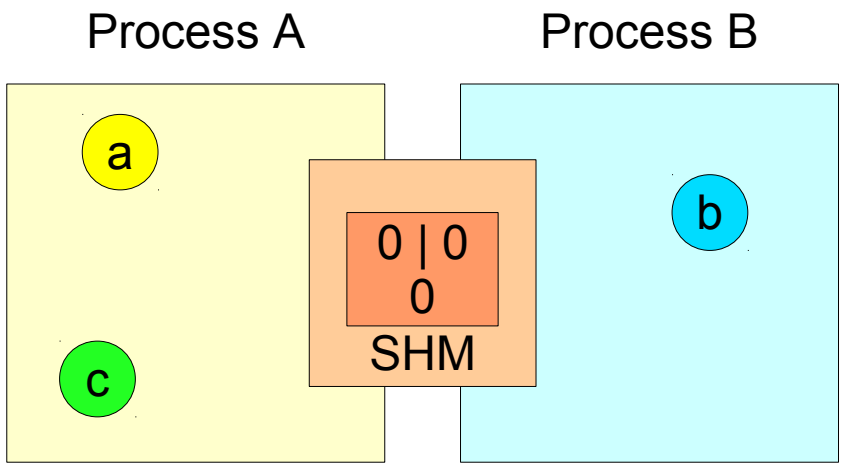
- Beispiel
  - 2 Prozesse
  - 3 Threads
  - Futex im Shared Memory
  - Mutex Protokol



- Symbole
  - T: lock holder's thread ID
  - W: bit indicating non-empty wait queue
  - Q: thread ID of first waiting thread

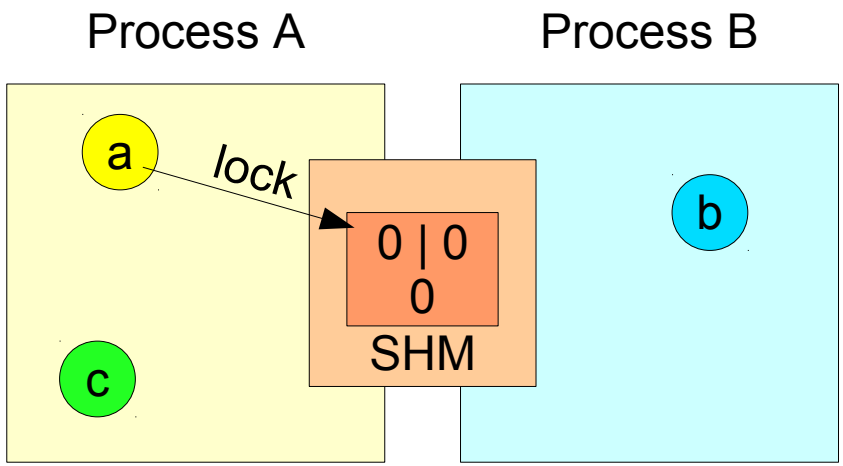
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



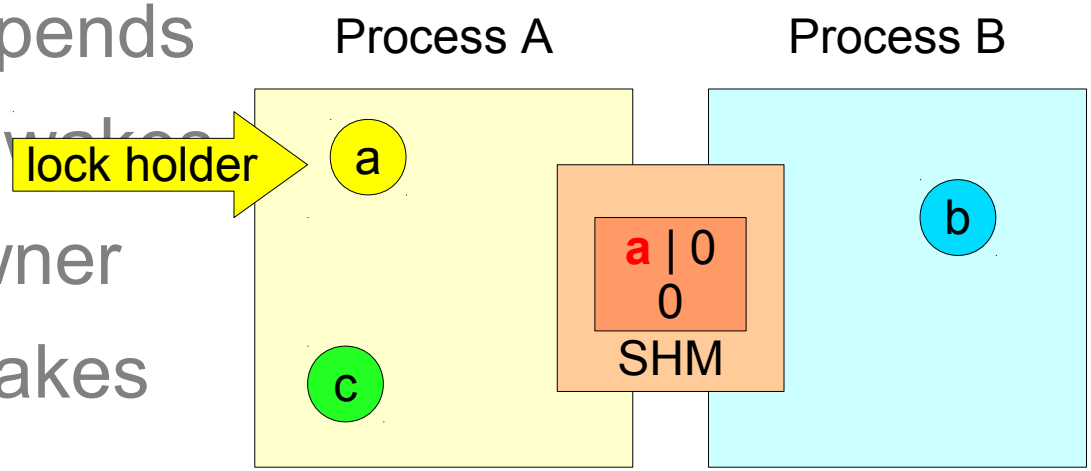
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



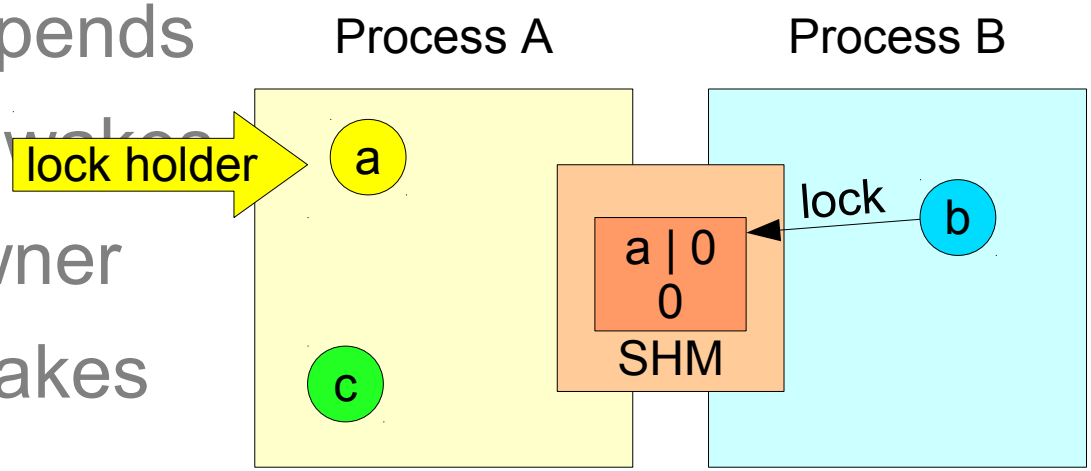
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



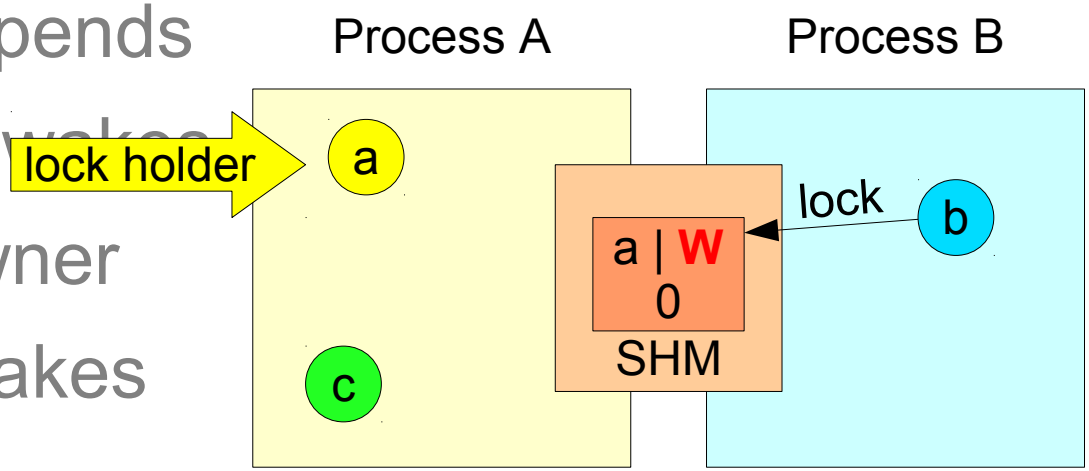
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



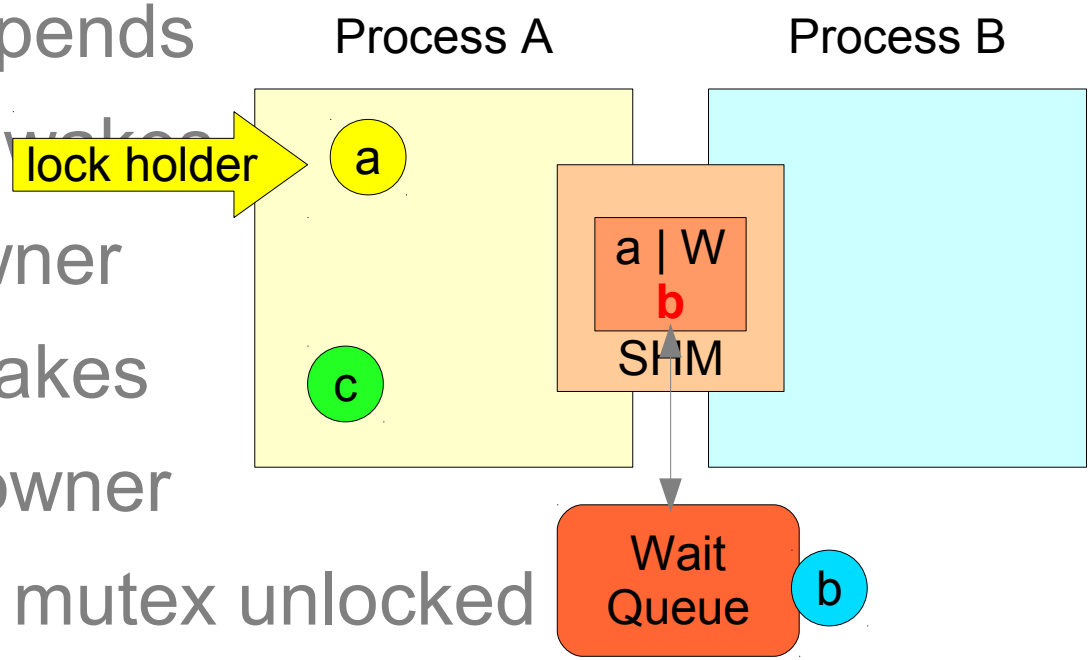
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



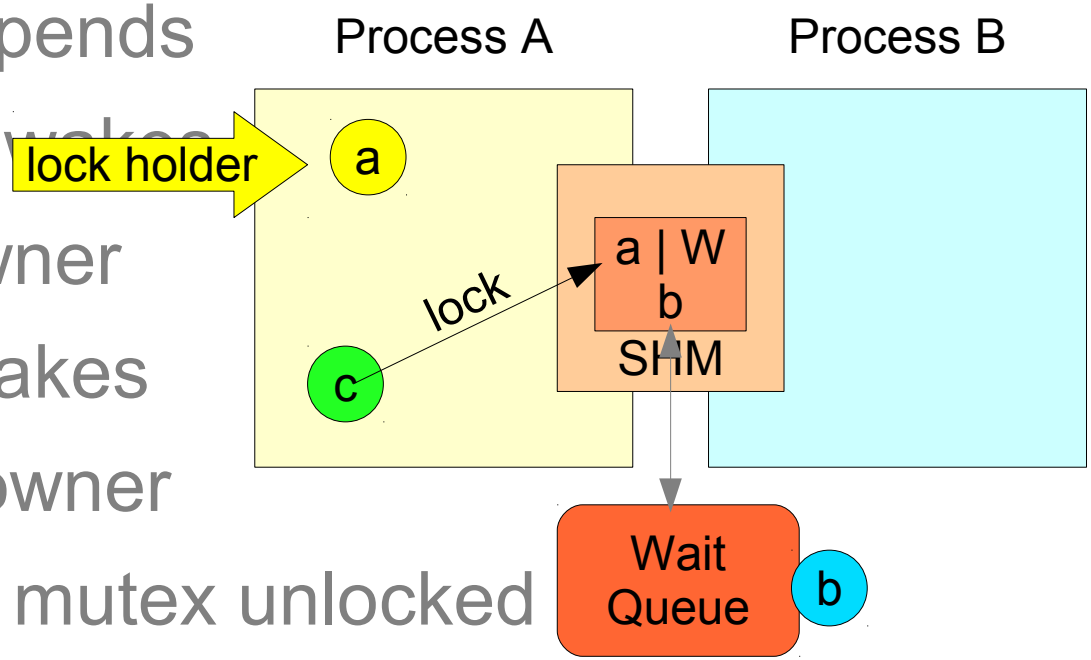
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



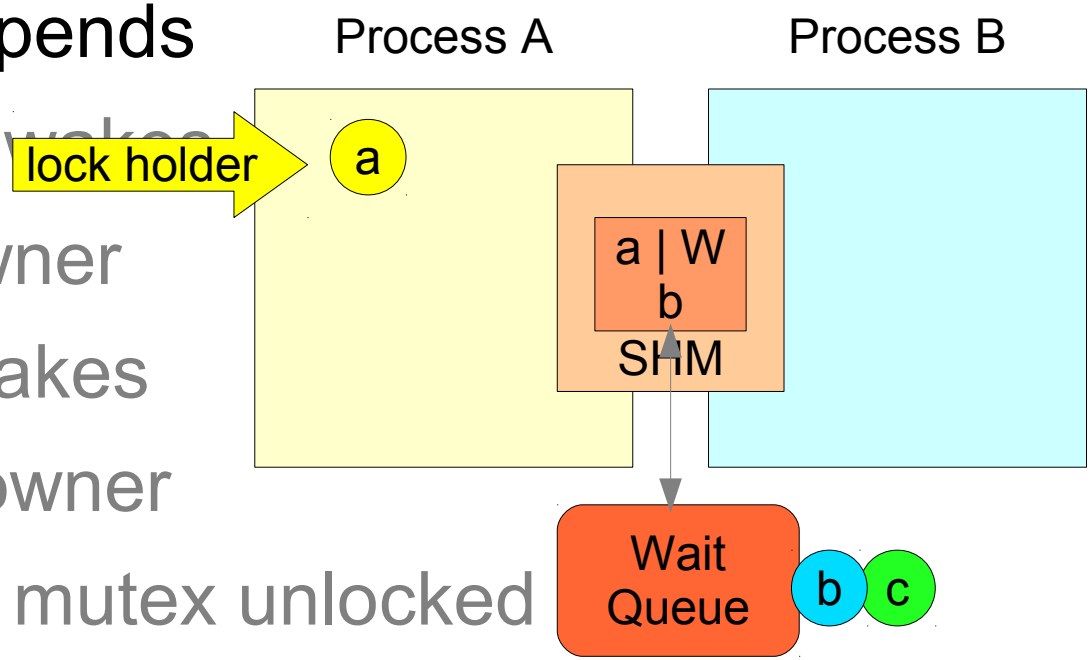
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



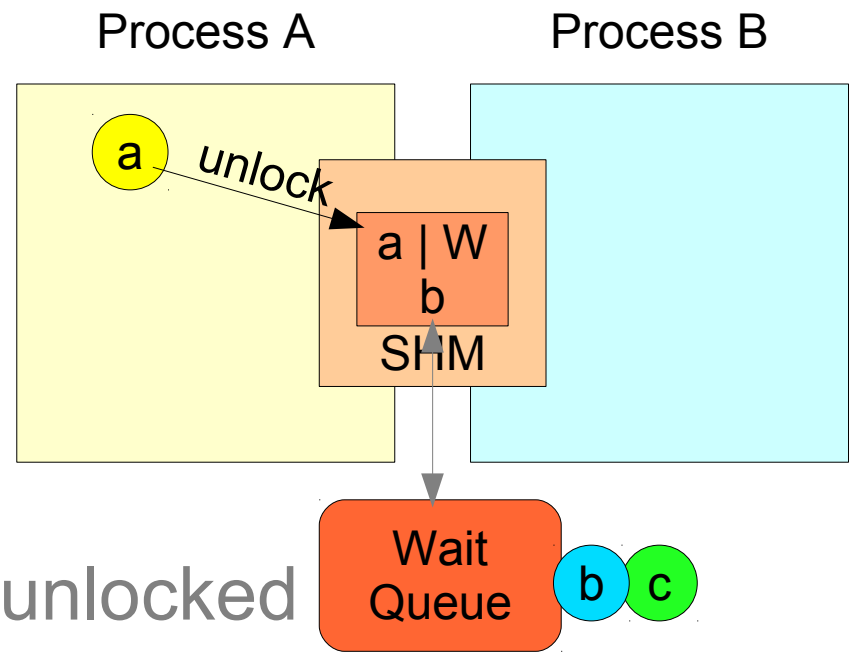
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



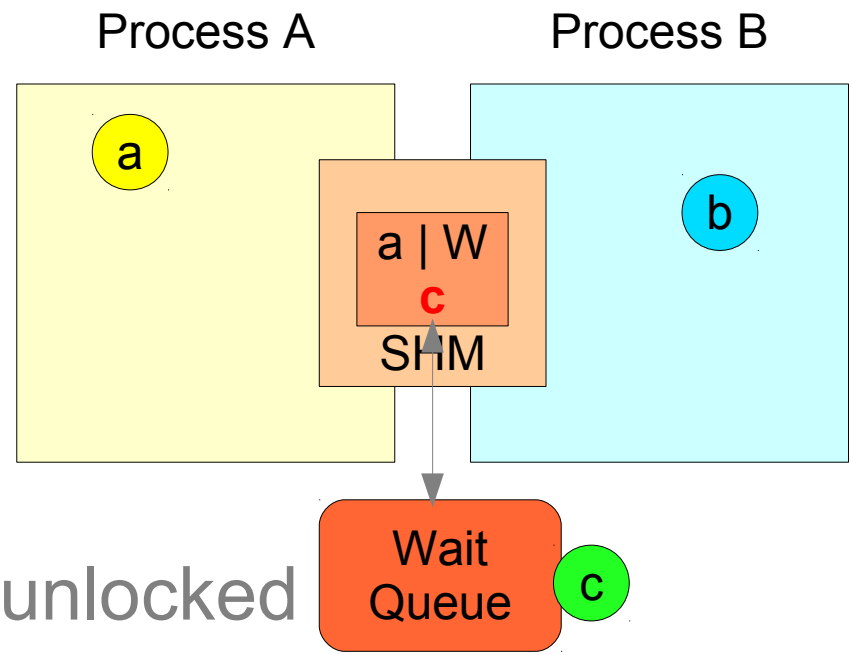
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



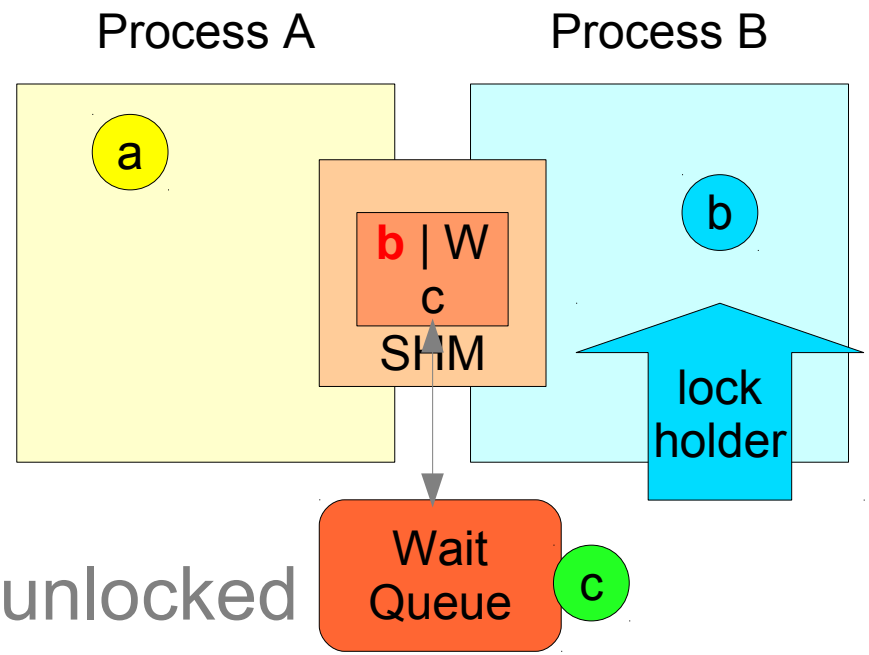
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



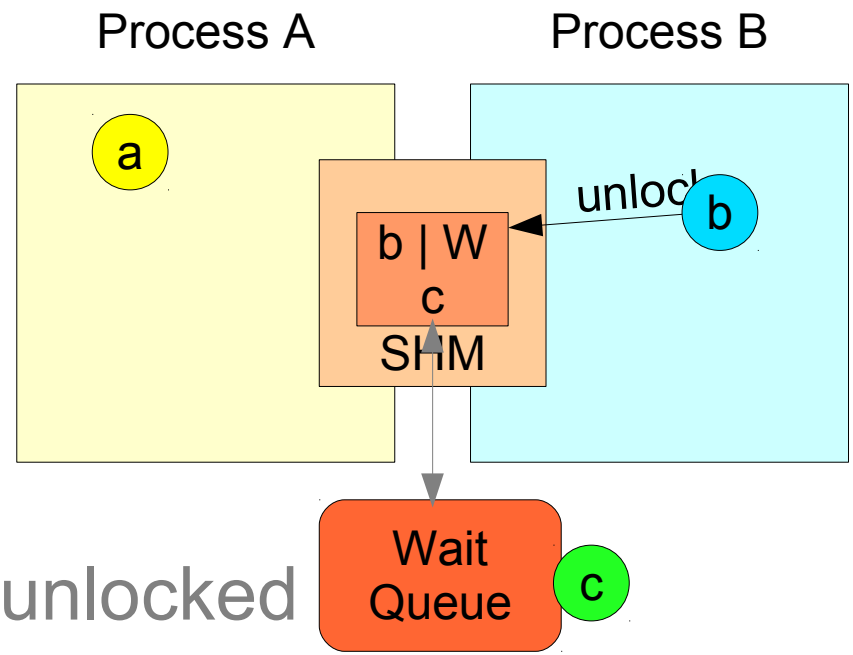
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



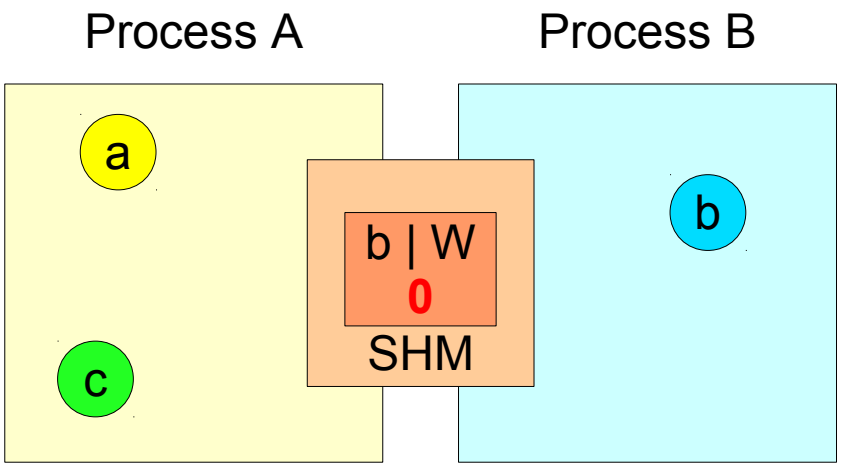
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



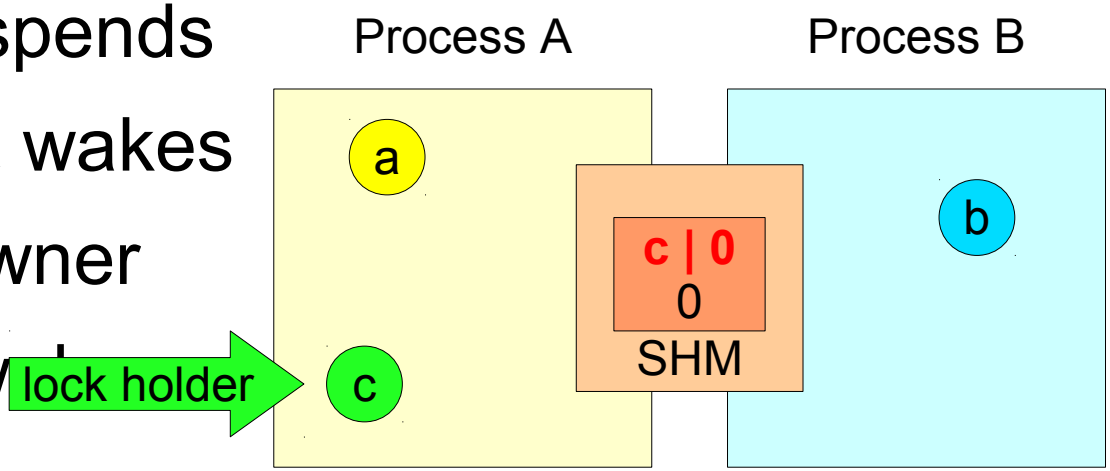
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



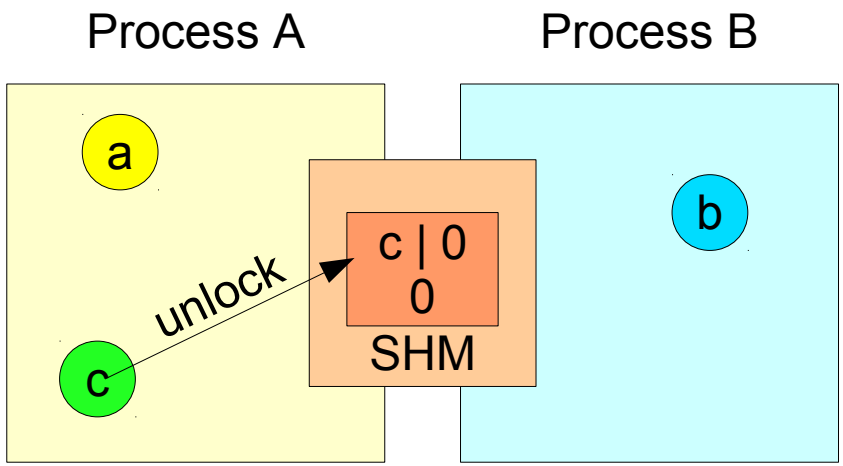
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & w
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



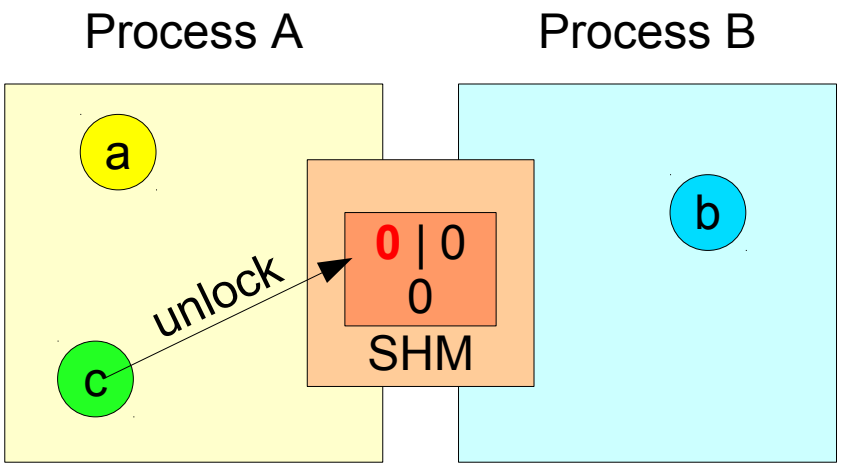
Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



Mutex Protokoll

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



- Mutex-Implementierung → OK
- **Condition Variablen ?**
- Locking der Warteschlange ?
- Robustness ?
- Sortiert nach Priorität ?

Condition Variablen

- Condition Variablen haben ein Support-Mutex
- CVs nutzen auch Futexe
  - Futex-Wert: atomarer Zähler → Wartezyklus
  - Warteschlange: wartende Threads

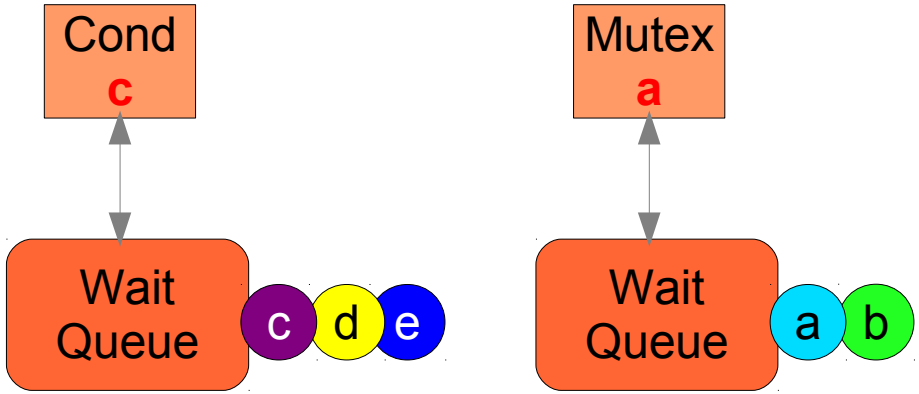
Condition Variablen

- Condition Variablen haben ein Support-Mutex
- CVs nutzen auch Futexe
  - Futex-Wert: atomarer Zähler → Wartezyklus
  - Warteschlange: wartende Threads

- `cond_wait()`
  - Gibt Mutex frei (Aufrufer von `cond_wait()` hält Mutex)
  - Systemcall, um Aufrufer zu suspendieren
  - Bei Rückkehr: Aufrufer hält das Mutex wieder

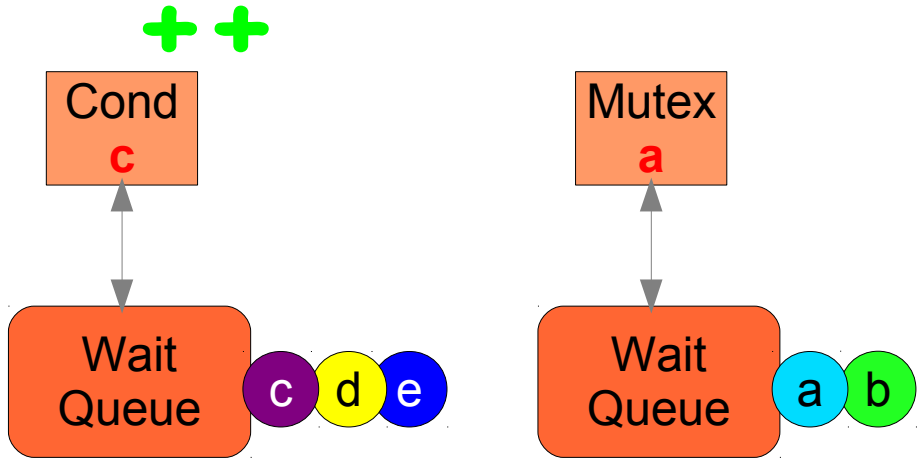
Condition Variablen

- `cond_signal()`



Condition Variablen

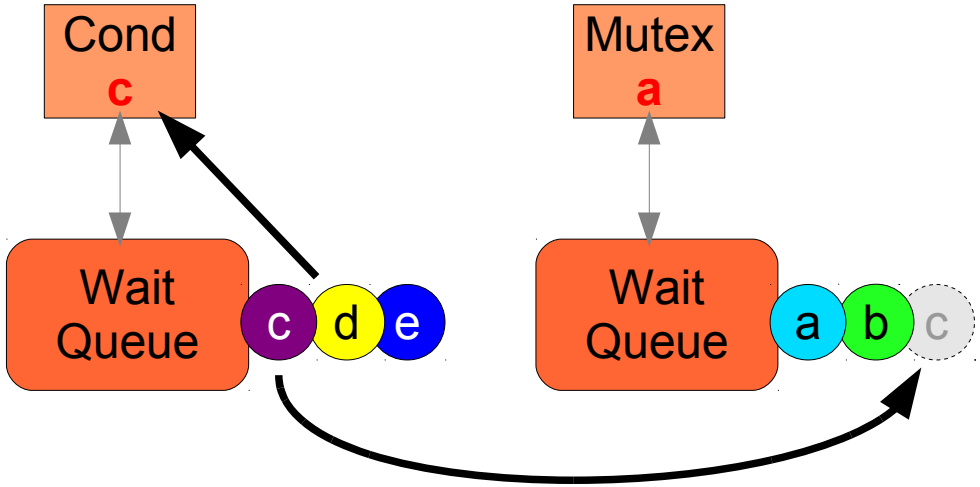
- `cond_signal()`



  - CV-Futex atomar erhöhen

Condition Variablen

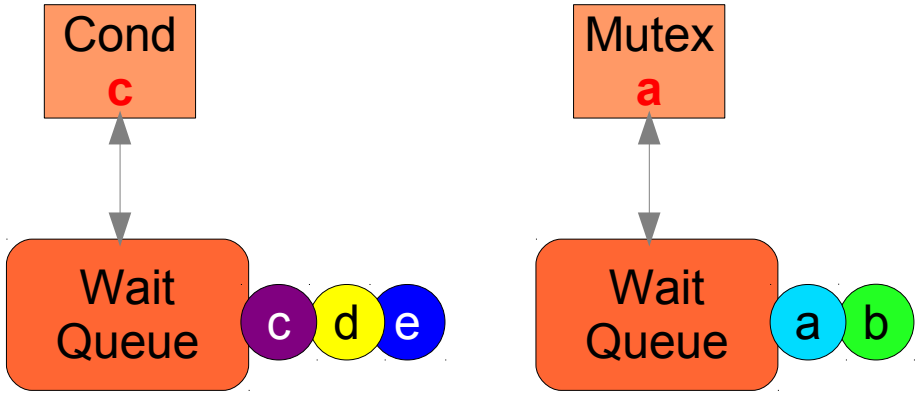
- `cond_signal()`



  - CV-Futex atomar erhöhen
  - Systemcall, um den ersten Wartenden von der Condition Variable Warteschlange auf die Mutex Warteschlange zu schieben
  - O(1) dank doppelt-verketteter Listen

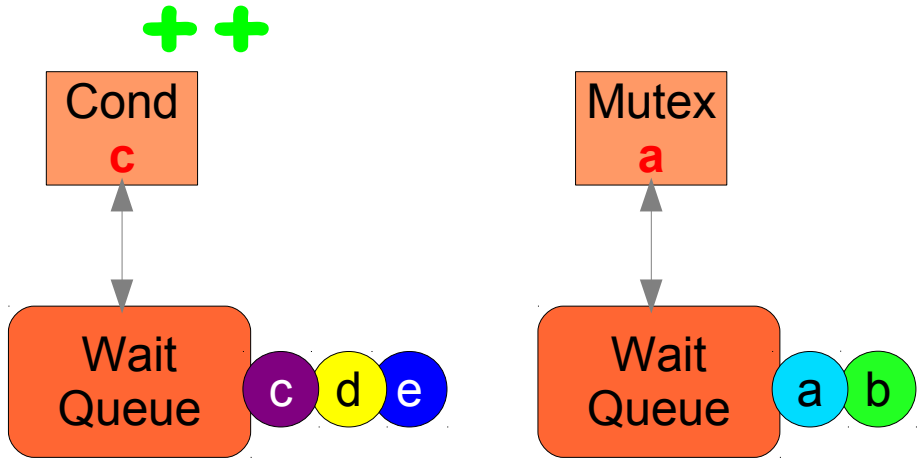
Condition Variablen

- `cond_broadcast()`



Condition Variablen

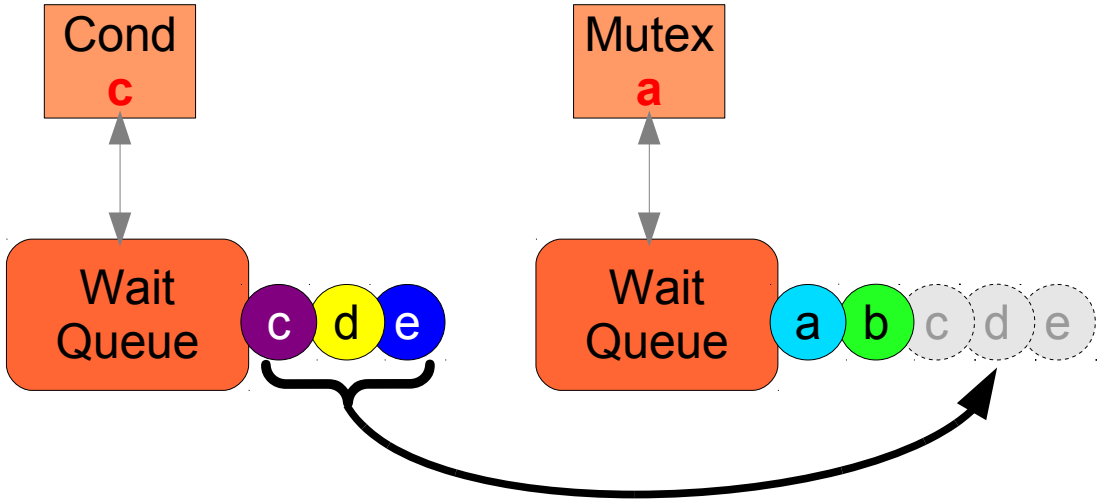
- `cond_broadcast()`



  - CV-Futex atomar erhöhen

Condition Variablen

- `cond_broadcast()`



  - CV-Futex atomar erhöhen
  - Systemcall, um alle Wartenden von der Condition Variable Warteschlange auf die Mutex Warteschlange zu schieben
  - O(1) dank doppelt-verketteter Listen

- Mutex-Implementierung → OK
- Condition Variablen → OK
- **Locking der Warteschlange ?**
- Robustness ?
- Sortiert nach Priorität ?

Internes Locking

- Warteschlangen werden vom Kern verwaltet

  → benötigt internes Locking

Internes Locking

- Warteschlangen werden vom Kern verwaltet

  → benötigt internes Locking

  mehre Ansätze möglich, abhängig vom …

- Futex ”Scope”:
  - Ein Adressraum
  - Eine Partition
  - Mehrere Partitionen

- Existierende Locks können benutzt werden!

Internes Locking

- Gehashte Adresse für's Locking
  - Futex Adresse → hash() → Lock in Array
    - ein Adressraum → virtuelle Adresse
    - mehrere Adressräume → physische Adresse
  - Locks sind vorallokiert

Internes Locking

- Gehashte Adresse für's Locking
  - Futex Adresse → hash() → Lock in Array
    - ein Adressraum → virtuelle Adresse
    - mehrere Adressräume → physische Adresse
  - Locks sind vorallokiert
- Beide Verfahren kombinieren:
  - Scope-Ansatz garantiert Partitioning
  - Hashing für Skalierbarkeit

Internes Locking

- Gehashte Adresse für's Locking
  - Futex Adresse → hash() → Lock in Array
    - ein Adressraum → virtuelle Adresse
    - mehrere Adressräume → physische Adresse
  - Locks sind vorallokiert
- Beide Verfahren kombinieren:
  - Scope-Ansatz garantiert Partitioning
  - Hashing für Skalierbarkeit
- Privilegien prüfen!

- Mutex-Implementierung → OK
- Condition Variablen → OK
- Locking der Warteschlange → OK
- **Robustness ?**
- Sortiert nach Priorität ?

Robustness

- Was passiert, wenn der User die Thread IDs manipuliert?
  - null oder ungültige ID
    - keine Wartenden gefunden
    - Liste bleibt intakt
    - Kern kann Wartende immer sauber ausqueuen

Robustness

- Was passiert, wenn der User die Thread IDs manipuliert?
  - null oder ungültige ID
    - keine Wartenden gefunden
    - Liste bleibt intakt
    - Kern kann Wartende immer sauber ausqueuen
  - Thread ID eines Wartenden eines anderen Futexes
    - prüfe Futex Adresse → niemand wird aufgeweckt

Robustness

- Was passiert, wenn der User die Thread IDs manipuliert?
  - null oder ungültige ID
    - keine Wartenden gefunden
    - Liste bleibt intakt
    - Kern kann Wartende immer sauber ausqueuen
  - Thread ID eines Wartenden eines anderen Futexes
    - prüfe Futex Adresse → niemand wird aufgeweckt
- Gleicher Fehler, als ob ein Thread niemals ein Mutex freigibt ...

Robustness

- Was passiert, wenn der User die Thread IDs manipuliert?
  - null oder ungültige ID
    - keine Wartenden gefunden
    - Liste bleibt intakt
    - Kern kann Wartende immer sauber ausqueuen
  - Thread ID eines Wartenden eines anderen Futexes
    - prüfe Futex Adresse → niemand wird aufgeweckt
- Gleicher Fehler, als ob ein Thread niemals ein Mutex freigibt ...

→ **Futex-Benutzer müssen einander trauen**

- Mutex-Implementierung → OK
- Condition Variablen → OK
- Locking der Warteschlange → OK
- Robustness → OK
- **Sortiert nach Priorität ?**

Sortiere Warteschlange

- Sortierte Warteschlangen
  - nach Priorität
  - … oder nach anderen Kriterien: Deadline, …

Sortiere Warteschlange

- Sortierte Warteschlangen
  - nach Priorität
  - … oder nach anderen Kriterien: Deadline, …

- Gesuchte Algorithmen
  - mit linearem Speicherverbrauch
  - mit **WCET** O(log n)
    - … und nicht amortisiert O(log n) …

→ Heaps und Binäre Suchbäume

Sortiere Warteschlange

- Sortierte Warteschlangen ...

  → nicht so einfach, wie ich zuerst dachte

Sortiere Warteschlange

- Sortierte Warteschlangen ...

  → nicht so einfach, wie ich zuerst dachte

- Weil:
  - Robustness: finde Wurzel in O(log n)
  - FIFO-Reihenfolge bei gleicher Priorität

    → schliesst Heaps aus!
  - Aber: Binäre Suchbäume unterstützen kein meld()

    → wake_all() in O(n) und präemptiv
  - schlecht für die Caches
    - z.B. durch 4K Ausrichtung der Threads im Kernel

- Mutex-Implementierung → OK
- Condition Variablen → OK
- Locking der Warteschlange → OK
- Robustness → OK
- Sortiert nach Priorität → TODO

  Gegenstand meiner aktuellen Forschung

- Mutex-Implementierung → OK
- Condition Variablen → OK
- Locking der Warteschlange → OK
- Robustness → OK

EN 50128
SIL 4
zertifiziert

Oktober 2013
TÜV Süd

- Sortiert nach Priorität → TODO

Gegenstand meiner aktuellen Forschung

Zusammenfassung

Zusammenfassung

- Mutexe und Condition Variablen mit FIFO-Reihenfolge
- Keine Allokationen zur Laufzeit im Kern nötig!
- Doppelt-verkettete Listen: alles in O(1) Zeit

Zusammenfassung

- Mutexe und Condition Variablen
  mit FIFO-Reihenfolge
- Keine Allokationen zur Laufzeit im Kern nötig!
- Doppelt-verkettete Listen: alles in O(1) Zeit

- Zusätzlich: ”wecke bliebige Anzahl Wartender”
  (und nicht nur Queues migrieren),
  → gleiche Flexibilität wie Linux

  → benötigt leider O(n) Zeit

Zusammenfassung

- Mutexe und Condition Variablen mit FIFO-Reihenfolge
- Keine Allokationen zur Laufzeit im Kern nötig!
- Doppelt-verkettete Listen: alles in O(1) Zeit

- Zusätzlich: ”wecke bliebige Anzahl Wartender”
  (und nicht nur Queues migrieren),
  → gleiche Flexibilität wie Linux

  → benötigt leider O(n) Zeit

  muss dann präemptiv implementiert werden!

Zusammenfassung

- Weitere Features für POSIX / PThreads:
  - Rekursive Mutexe
  - Error Checking Mutexe
  - Condition Variablen

Zusammenfassung

- Weitere Features für POSIX / PThreads:
  - Rekursive Mutexe
  - Error Checking Mutexe
  - Condition Variablen
  - RWLocks, Barrieren, pthread_once()
  - POSIX Semaphoren (sem_*(), nicht System V IPC)
  - Warten mit relativem oder absolutem Timeout
  - ”private” und ”pshared” Futexe

Zusammenfassung

- Weitere Features für POSIX / PThreads:
  - Rekursive Mutexe
  - Error Checking Mutexe
  - Condition Variablen
  - RWLocks, Barrieren, pthread_once()
  - POSIX Semaphoren (sem_*(), nicht System V IPC)
  - Warten mit relativem oder absolutem Timeout
  - ”private” und ”pshared” Futexe

TODO:
- `PTHREAD_PRIO_INHERIT`
- `PTHREAD_PRIO_PROTECT`

Vielen Dank!

Fragen?