Betriebssysteme (BS)

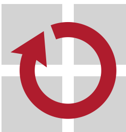
VL 11.2 – Fadensynchronisation – Mechanismen

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 20 – 25. Januar 2021



https://www4.cs.fau.de/Lehre/WS20/V_BS

Agenda

Einleitung
- Motivation
- Erstes Fazit

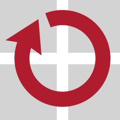
Prioritätsebenenmodell mit Fäden

Mechanismen
- Randbedingungen
- Mutex, Implementierungsvarianten
- Passives Warten
- Semaphore

Beispiel: Windows
- Warteobjekte
- Optimierungen für Mehrkernsysteme

Zusammenfassung

Referenzen



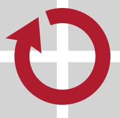
Fadensynchronisation: Annahmen

- Fäden können unvorhersehbar verdrängt werden
  - zu jedem beliebigen Zeitpunkt
  - von beliebigen anderen Fäden höherer, gleicher, oder niedrigerer Priorität (⟵ Fortschrittsgarantie)
- Annahmen typisch für Arbeitsplatzrechner ⇝ VL 9
  - *probabilistic, interactive, preemptive, online CPU scheduling*
  - andere Arten des Schedulings werden im Folgenden nicht betrachtet

Problematisch ist hier die Fortschrittsgarantie

Bei rein prioritätsgesteuertem Scheduling (Fäden innerhalb einer Prioritätsstufe werden sequentiell abgearbeitet) könnten wir das Ebenenmodell der Unterbrechungsbehandlung (⇝ VL 5) einfach auf Fadenprioritäten ausdehen und vergleichbaren Mechanismen (expliziter Ebenenwechsel, algorithmisch unter der Annahme von *run-to-completion*) synchronisieren.

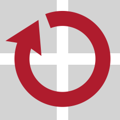
- typisch für ereignisgesteuerte Echtzeitssysteme ⇝ [EZS]
- in Windows/Linux: Bereich der Echtzeitprioritäten ⇝ VL 9
- bei mehreren Kernen bleibt das Problem der echten Parallelität!



Fadensynchronisation: Überblick

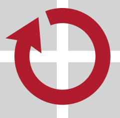
- **Ziel:** aus Anwendersicht
  Koordinierung und Interaktion
  - Koordinierung des exklusiven Zugriffs auf wiederverwendbare Betriebsmittel (gegenseitiger Ausschluss) $\leadsto$ Mutex
  - Interaktion / Koordinierung von konsumierbaren Betriebsmitteln (Synchronisation) $\leadsto$ Semaphore
- **Implementierungsansatz:** für den BS-Entwickler
  Steuerung der CPU-Zuteilung an Fäden
  - Fäden werden zeitweise von der Zuteilung ausgenommen
  - „Warten” als BS-Konzept

Im Folgenden befassen wir uns mit der Perspektive der BS-Entwicklerin

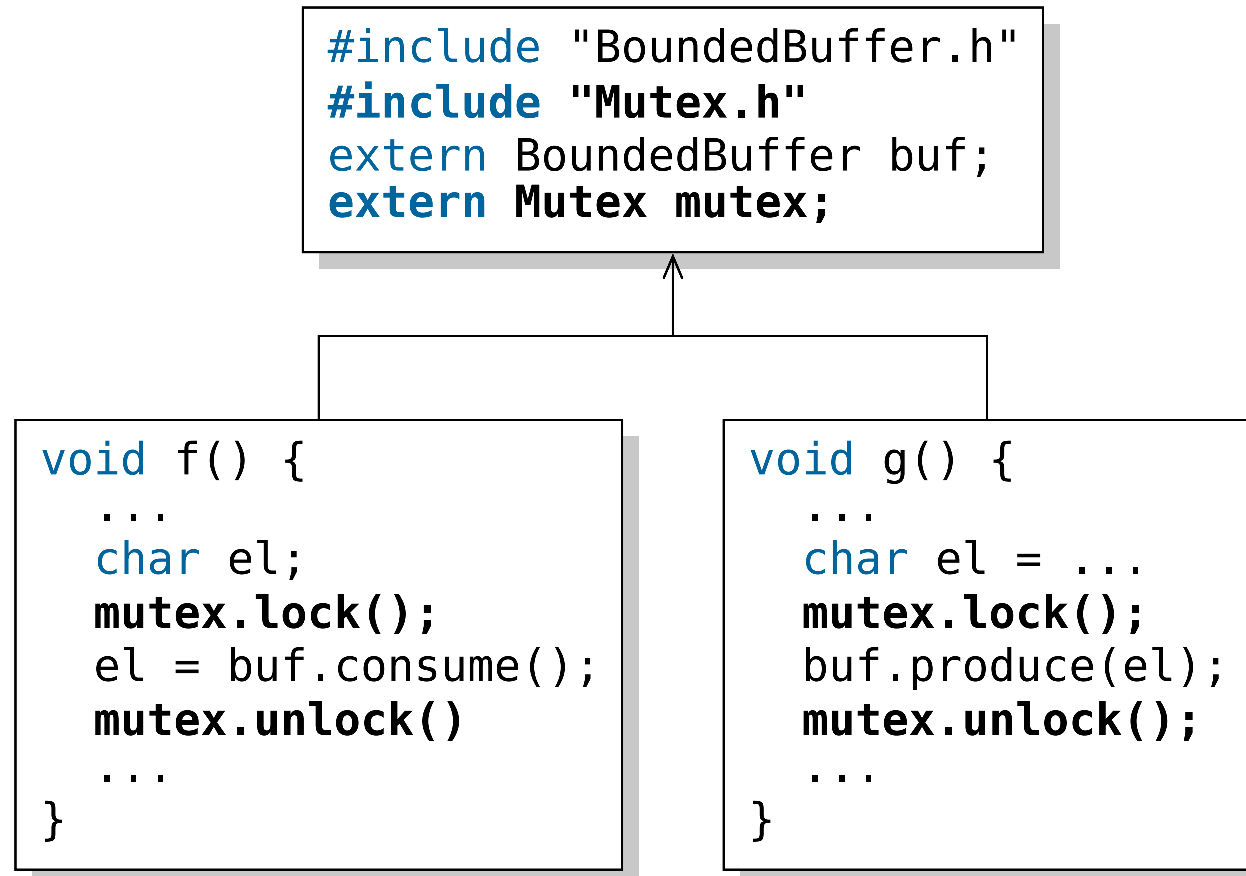


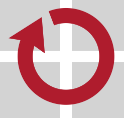
Mutex

- **Mutex** $\mapsto$ Kurzform von **mu**tual **ex**clusion
  - Ursprung: Bezeichnername eines zweiwertigen Semaphor, eingesetzt für gegenseitigen Ausschluss [2]
  - allgemein: Algorithmus für die Sicherstellung von gegenseitigem Ausschluss in einem kritischen Gebiet
  - hier: Systemabstaktion `class Mutex`
- Schnittstelle
  - `void Mutex::lock()`
    - Betreten und Sperren des kritischen Gebiets
    - Faden kann blockieren
  - `void Mutex::unlock()`
    - Verlassen und Freigeben des kritischen Gebiets
- Korrektheitsbedingung
  - Es befindet sich maximal ein Faden im kritischen Gebiet
    - Für ausgeführte Operationen gilt: $\sum_{\texttt{lock()}} - \sum_{\texttt{unlock()}} \leq 1$



Mutex: Verwendung



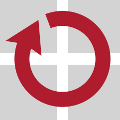


Mutex mit aktivem Warten (↦ *spin lock*)

- Implementierung rein auf der Benutzerebene
  - markiere Belegung in boolscher Variable (0 ↦ frei, 1 ↦ belegt)
  - warte in `lock()` aktiv, bis Variable 0 wird

```
// __sync_lock_test_and_set ist ein gcc builtin fuer
// (CPU-spezifisches) test-and-set (ab gcc 4.1)
class SpinningMutex {
  volatile int locked;
public:
  SpinningMutex() : locked (0) {}
  void lock() {
    while (__sync_lock_test_and_set(
            &locked, 1) == 1)
      ;
  }
  void unlock() {
    locked = 0;
  }
};
```

```
// g++-4.2 -O3
// -fomit-frame-pointer
lock:
    mov 0x4(%esp),%edx
l1: mov $0x1,%eax
    xchg %eax,(%edx)
    sub $0x1,%eax
    je l1
    repz ret
unlock:
    mov 0x4(%esp),%eax
    movl $0x0,(%eax)
    ret
```



Mutex mit aktivem Warten: Bewertung

Vorteile

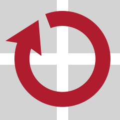
- Konsistenz ist sichergestellt, Korrektheitsbedingung wird erfüllt
  - unter der Vorraussetzung von Fortschrittsgarantie für alle Fäden
- Synchronisation erfolgt ohne Beteiligung des Betriebssystems
  - keine Systemaufrufe erforderlich

Nachteile

- aktives Warten verschwendet viel CPU-Zeit
  - mindestens bis die Zeitscheibe abgelaufen ist
  - bei Zeitscheiben von 10–800 msec ganz erheblich!
  - Faden wird eventuell vom Scheduler „bestraft” ($\rightsquigarrow$ VL 9)

Fazit

Aktives Warten ist – wenn überhaupt – nur auf **Multiprozessormaschinen** eine Alternative.

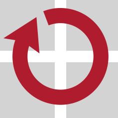


Mutex mit harter Synchronisation

- Implementierung mit „harter Fadensynchronisation"
  - deaktiviere Verdrängbarkeit vor Betreten des kritischen Gebiets
    - neue Systemoperation: `forbid()`
  - reaktiviere Verdrängbarkeit nach Verlassen des kritischen Gebiets
    - neue Systemoperation: `permit()`

```
class HardMutex {
public:
  void lock() {
    forbid();     // schalte Multitasking ab
  }
  void unlock() {
    permit();     // schalte Multitasking wieder an
  }
};
```

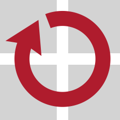
In der Welt der Echtzeitsysteme steht dieses Verfahren hinter dem *non-preemptive critical section (NPCS) protocol* [6, EZS].



Mutex mit harter Synchronisation: Implementierung

- Implementierung durch den Scheduler, z. B. über
  - spezielle nicht verdrängbare Prioritätsklasse
    - OSEK OS / AUTOSAR OS: Ressource `RES_SCHED` [7]
  - eigene Prioritätsebene $E_{1/4}$ für den Scheduler
    - war faktisch so in AmigaOS realisiert
  - `resume()` schaltet einfach zum Aufrufer zurück
- oder ganz einfach durch Betreten der Epilogebene
  - Fadenumschaltung ist üblicherweise auf der Epilogebene angesiedelt
    - so lange ein Faden auf der Epilogebene ist kann er nicht verdrängt werden
    - Vorraussetzung: Kontrollflüsse der Epilogebene werden sequentialisiert
  - $\leadsto$ **Sequentialisierung auch mit Epilogen!**

```
void forbid() {
  enter();
}
void permit() {
  leave();
}
```

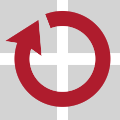


Mutex mit harter Synchronisation: Bewertung

- **Vorteile**
  - Konsistenz ist sichergestellt, Korrektheitsbedingung wird erfüllt
  - einfach zu implementieren
- **Nachteile**
  - Breitbandwirkung
    - alle Fäden (und ggfs. sogar Epiloge!) werden pauschal verzögert
  - Prioritätsverletzung
    - „unbeteiligte” Kontrollflüsse mit höherer Priorität werden verzögert
  - prophylaktisches Verfahren
    - Nachteile werden in Kauf genommen, auch wenn die Wahrscheinlichkeit einer tatsächlichen Kollision sehr klein ist.

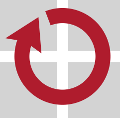
Fazit

Fadensynchronisation auf Epilogebene hat viele Nachteile. Sie ist nur auf Einprozessorsystemen für kurze, selten betretene kritische Gebiete geeignet – oder wenn sowieso mit Epilogen synchronisiert werden muss.



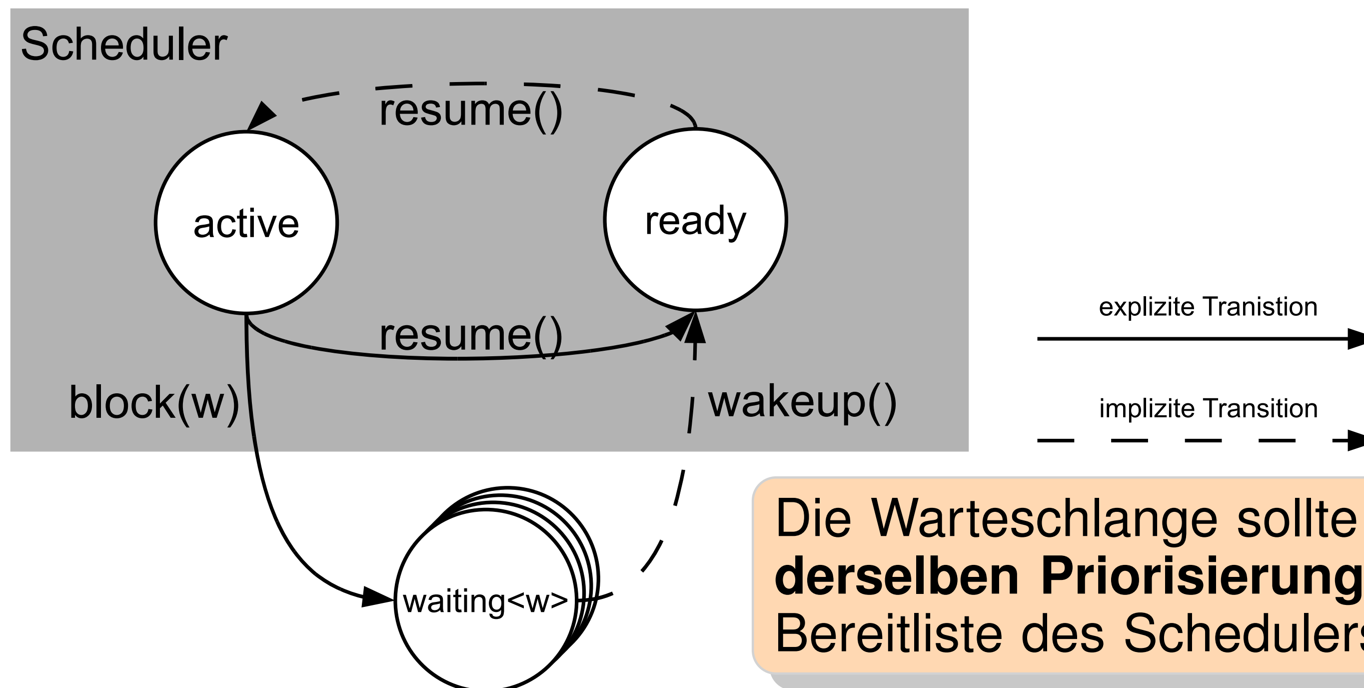
Passives Warten: Motivation

- Bisherige Mutex-Implementierungen sind nicht ideal
  - Mutex mit aktivem Warten
    $\leadsto$ Verschwendung von CPU-Zeit
  - Mutex mit harter Synchronisation
    $\leadsto$ grobgranular, proritätsverletzend
- **Besserer Ansatz:** Faden so lange von der CPU-Zuteilung ausschließen, wie der Mutex belegt ist.
- Erfordert neues BS-Konzept: **passives Warten**
  - Fäden können auf ein Ereignis „passiv warten”
    - passiv warten $\mapsto$ von CPU-Zuteilung ausgeschlossen sein
    - (Neuer) Fadenzustand: *wartend* (auf Ereignis)
  - Eintreffen des Ereignisses bewirkt Verlassen des Wartezustands
    - Faden wird in CPU-Zuteilung eingeschlossen
    - Anschließender Fadenzustand: *bereit*

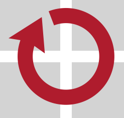


Passives Warten: Implementierung

- Erforderliche Abstraktionen:
  - Operationen: `block(), wakeup()`
    - Betreten bzw. Verlassen des Wartezustands
  - Warteobjekt: `Waitingroom`
    - repräsentiert das Ereignis auf das gewartet wird
    - enthält üblicherweise eine Warteschlange der wartenden Fäden

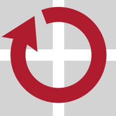


Die Warteschlange sollte sinnvollerweise mit **derselben Priorisierungsstrategie** wie die Bereitliste des Schedulers verwaltet werden!



Mutex mit passivem Warten: Implementierung

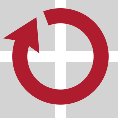
```
class WaitingMutex : public Waitingroom {
  int locked;
public:
  WaitingMutex() : locked (0) {}
  void lock() {
    while (__sync_lock_test_and_set(&locked, 1) == 1)
          scheduler.block(*this);
  }
  void unlock() {
    locked = 0;
    // Maximal einen wartenden Thread holen und aufwecken
    Customer* t = dequeue();
    if (t) scheduler.wakeup(*t);
  }
};
```



Mutex mit passivem Warten: Implementierung

```
class WaitingMutex : public Waitingroom {
  int locked;
public:
  WaitingMutex() : locked (0) {}
  void lock() {
    while (__sync_lock_test_and_set(&locked, 1) == 1)
          scheduler.block(*this);
  }
  void unlock() {
    locked = 0;
    // Maximal einen wartenden Thread holen und aufwecken
    Customer* t = dequeue();
    if (t) scheduler.wakeup(*t);
  }
};
```

Bei dieser Lösung gibt es noch ein Problem. . .

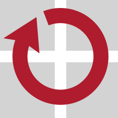


Mutex mit passivem Warten: Implementierung

```
class WaitingMutex : public Waitingroom {
  int volatile locked;
public:
  WaitingMutex() : locked (0) {}
  void lock() {
    mutex.lock();
    while (locked == 1)
        scheduler.block(*this);
    locked = 1;
    mutex.unlock();
  }
  void unlock() {
    mutex.lock();
    locked = 0;
    // Maximal einen wartenden Thread holen und aufwecken
    Customer* t = dequeue();
    if (t) scheduler.wakeup(*t);
    mutex.unlock();
  }
};
```

`lock()` und `unlock()` bilden ein eigenes kritisches Gebiet

Kann man dieses kritische Gebiet mit einem `Mutex` schützen?

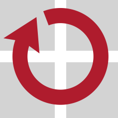


Mutex mit passivem Warten: Implementierung

```
class WaitingMutex : public Waitingroom {
  int volatile locked;
public:
  WaitingMutex() : locked (0) {}
  void lock() {
    enter();
    while (locked == 1)
        scheduler.block(*this);
    locked = 1;
    leave();
  }
  void unlock() {
    enter();
    locked = 0;
    // Maximal einen wartenden Thread holen und aufwecken
    Customer* t = dequeue();
    if (t) scheduler.wakeup(*t);
    leave();
  }
};
```

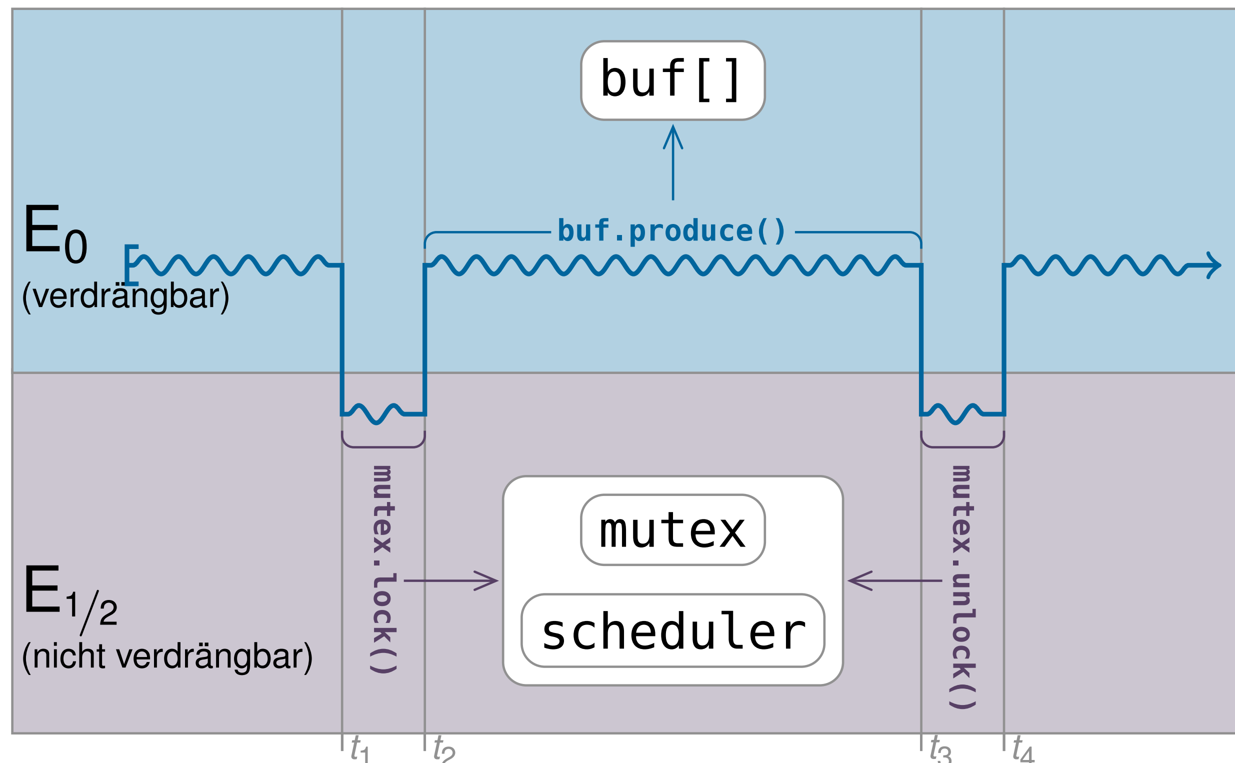
Mit einem `HardMutex` ginge es!

Faktisch schützt man `lock()` und `unlock()` somit, wie hier dargestellt, auf **Epilogebene**.



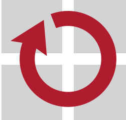
Mutex mit passivem Warten: Fazit

- Mutex-Zustand liegt nun im Kern auf der Epilogebene
  - genauer: auf derselben Ebene wie der Scheduler Zustand
- Das ist ein allgemein verwendbares Prinzip
  - Implementierung der Synchronisationsmechanismen für E_0-Kontrollflüsse wird auf $E_{1/2}$ synchronisiert.



Noch besser wäre natürlich **weiche Synchronisation**.

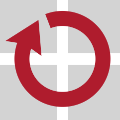
Dazu mehr in [CS]!



Semaphore

- Semaphore ist das klassische Synchronisationsobjekt
  - Edgar W. Dijkstra, 1963 [3]
  - In vielen BS: Grundlage für alle Warte-/Synchronisationsobjekte
  - Für uns: Semaphore $\mapsto$ Warteobjekt + Zähler
- Zwei Standardoperationen (mit jeweils diversen Namen [2–4])
  - `prolaag()`, **`P()`**, `wait()`, `down()`, `acquire()`, `pend()`
    - wenn zähler $>$ 0 vermindere Zähler
    - wenn zähler $\leq$ 0 warte bis Zähler $>$ 0 und probiere es noch einmal
  - `verhoog()`, **`V()`**, `signal()`, `up()`, `release()`, `post()`
    - erhöhe Zähler
    - wenn Zähler $=$ 1 wecke gegebenenfalls wartenden Faden
- Es gibt vielfältigste Varianten

Implementierung der Standardvariante erfolgt in der Übung!

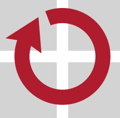


Semaphore: Verwendung

- Semantik der Semaphore eignet sich besonders für die Implementierung von Erzeuger/Verbraucher-Szenarien
  - Also für den geordneten Zugriff auf konsumierbare Betriebsmittel
    - Zeichen von der Tastatur
    - Signale, die auf Fadenebene weiterverarbeitet werden sollen
    - ...
  - Interner Zähler repräsentiert die Anzahl der Ressourcen
    - Erzeuger ruft `V()` auf für jedes erzeugte Element.
    - Verbraucher ruft `P()` auf, um ein Element zu konsumieren
      $\leadsto$ wartet gegebenenfalls.

Beachte!

- `P()` kann auf Fadenebene blockieren, `V()` blockiert jedoch nie!
- Als Erzeuger kommt daher auch ein Kontrollfluss auf Epilogebene oder Unterbrechungsebene in Frage. (Entsprechende Synchronisation des internen Semaphorzustands vorrausgesetzt.)



Semaphore vs. Mutex: Einordnung

- Mutex wird „klassisch“ als binärer Semaphor bezeichnet [2]
  - Mutex $\mapsto$ Semaphore mit initialem Zählerwert 1
  - `lock()` $\mapsto$ `P()`, `unlock()` $\mapsto$ `V()`
- Die Semantik ist (heute) jedoch i. a. deutlich strenger:
  - Ein belegter Mutex hat (implizit oder explizit) einen Besitzer.
    - Nur dieser Besitzer darf `unlock()` aufrufen.
    - Muteximplementierungen in z. B. Linux oder Windows überprüfen dies.
  - Ein Mutex kann (üblicherweise) auch rekursiv belegt werden
    - Interner Zähler: *Derselbe* Faden kann mehrfach `lock()` aufrufen; nach der entsprechenden Anzahl von `unlock()`-Aufrufen ist der Mutex frei
    - Eine Semaphore kann hingegen von *jedem* Faden verändert werden.

Semaphore als Basis aller Dinge?

In vielen BS ist Semaphore die Grundabstraktion für Fadensynchronisation. Sie wird deshalb in der Literatur oft als (notwendige) Implementierungbasis für Mutex, Bedingungsvariable, Leser-Schreiber-Sperre etc. angesehen.