# Locks vs. Ereignisvariablen

Ute Bormann, TI2

2023-10-13

# Inhalt

1. Anwendung von Locks
2. Ereignisvariablen
3. Aktives vs. blockierendes Warten

# Teil 1:
# Anwendung von Locks

# Bisher: Realisierung von Lock-Verfahren

- Unterbrechungsausschluss
- aktives Warten mit Schlossvariablen
  ⇒ verschiedene Algorithmen betrachtet
- Potentielle Probleme:
  - kein gegenseitiger Ausschluss
  - Verklemmungen
  - Verhungern
  - After-you-after-you
- Spezielle unteilbare Operationen (z.B. Test_and_set)

| | A | B |
|---|---|---|
| Eintrittsprotokoll | `m.lock();` | `m.lock();` |
| kritischer Abschnitt | `i++;` | `i--;` |
| Austrittsprotokoll | `m.unlock();` | `m.unlock();` |

- Beispielimplementierung

```
class Mutex {
  bool key;
public:
  Mutex() {key = false;}
  lock() {while(test_and_set(key));}
  unlock() {key = false;}
}

Mutex m;
```

# Anwendung auf Beispielsituationen

## Kritischer Abschnitt

- Entweder A oder B ⇒ mehrseitige Synchronisation

```
    A                Mutex m;              B
…                                      …
m.lock();                              m.lock();
… kritischer Abschnitt…                … kritischer Abschnitt …
m.unlock();                            m.unlock();
```

# Anwendung auf Beispielsituationen

## Kritischer Abschnitt

- Entweder A oder B ⇒ mehrseitige Synchronisation

```
    A                Mutex m;              B
…                                      …
m.lock();                              m.lock();
… kritischer Abschnitt…                … kritischer Abschnitt …
m.unlock();                            m.unlock();
```

## Leser-/Schreiber-Problem

- Leser nebenläufig zulassen
- Schreibender Zugriff muss exklusiv sein
- Während des Schreibens auch keine Leser zulassen
  ⇒ ebenfalls mit Locks umsetzbar (nächste Folie)

# Leser-/Schreiber-Problem

1. Schritt: Leser und Schreiber alternativ

```
Mutex rw;

reader() {
  ...
    rw.lock();
  // Lesen

    rw.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

# Leser-/Schreiber-Problem

1. Schritt: Leser und Schreiber alternativ
2. Schritt: Mehr als einen Leser zulassen

```
Mutex rw;
int count = 0;

reader() {
  ...

  count++;
  if (count==1)
    rw.lock();

  // Lesen

  count--;
  if (count==0)
    rw.unlock();

  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

# Leser-/Schreiber-Problem

1. Schritt: Leser und Schreiber alternativ
2. Schritt: Mehr als einen Leser zulassen

```
Mutex rw;
int count = 0;

reader() {
  ...
  count++;
  if (count==1)
    rw.lock();

  // Lesen

  count--;
  if (count==0)
    rw.unlock();

  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

# Leser-/Schreiber-Problem

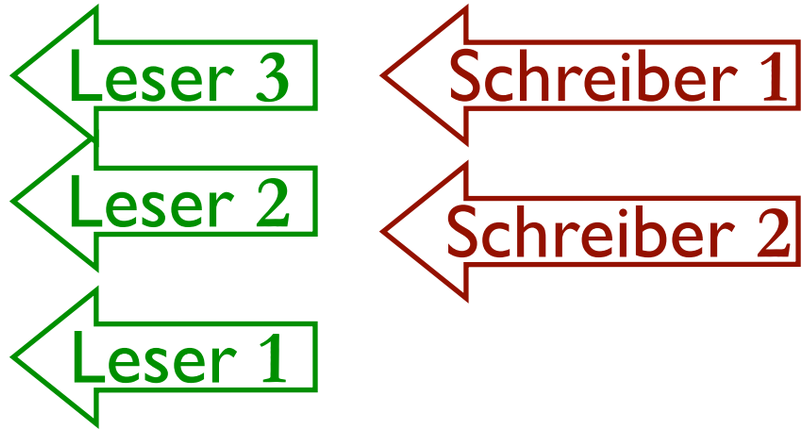
1. Schritt: Leser und Schreiber alternativ
2. Schritt: Mehr als einen Leser zulassen
3. Schritt: Zähler schützen

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem



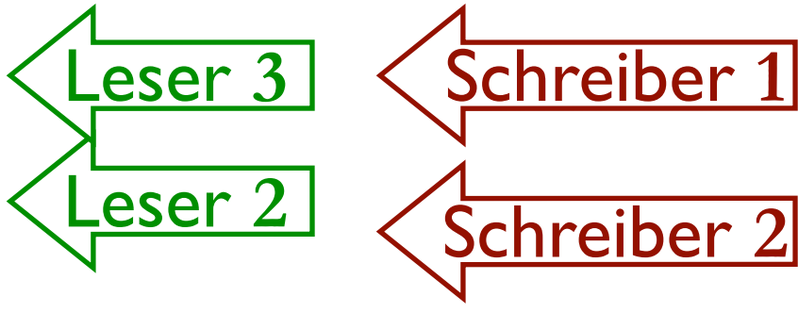

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Beispielabläufe durchspielen

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem



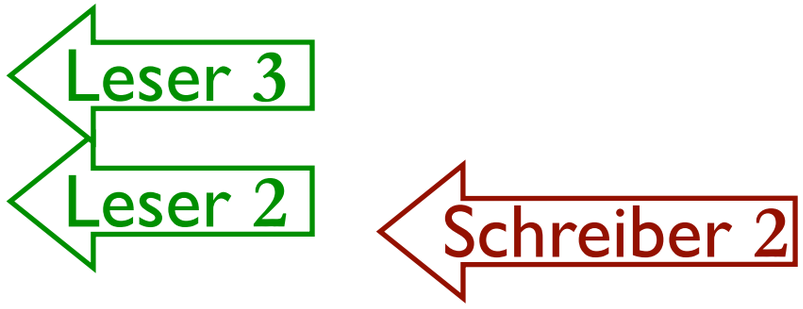

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

Leser 1

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem




```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```


```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```


(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

Leser 3

Schreiber 2

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

Leser 2

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Schreiber 1

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Schreiber 2

Leser 3

Schreiber 1

Leser 2

Leser 1

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Schreiber 2

Schreiber 1

Leser 3

Leser 1

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Schreiber 2

Schreiber 1

Leser 2

Leser 3

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Schreiber 2

Schreiber 1

Leser 3 2

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Schreiber 2

Schreiber 1

Leser 3

Leser 2

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Schreiber 2

Schreiber 1

Leser 3

Leser 2

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Leser 2

Leser 3

Leser 1

Schreiber 2

Schreiber 1

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Leser 2

Leser 3

Leser 1

Schreiber 2

Schreiber 1

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Leser-/Schreiber-Problem

```
Mutex rw;
int count = 0;
Mutex s;

reader() {
  ...
  s.lock();
  count++;
  if (count==1)
    rw.lock();
  s.unlock();
  // Lesen
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

```
writer() {
  ...
  rw.lock();
  // Schreiben
  rw.unlock();
  ...
}
```

Leser 2

Schreiber 2

Schreiber 1

Leser 1

(Achtung: Leser werden bei dieser Lösung bevorzugt)

# Fragen – Teil 1

- Warum kann man das Leser/Schreiber-Problem nicht mit einem einzigen Lock lösen?

# Teil 2:
# Ereignisvariablen

# Anwendung auf Beispielsituationen

## Kritischer Abschnitt

- Entweder A oder B ⇒ mehrseitige Synchronisation

```
    A                Mutex m;              B
…                                      …
m.lock();                              m.lock();
… kritischer Abschnitt…                … kritischer Abschnitt …
m.unlock();                            m.unlock();
```

## Leser-/Schreiber-Problem

- Leser nebenläufig zulassen
- Schreibender Zugriff muss exklusiv sein
- Während des Schreibens auch keine Leser zulassen

  ⇒ ebenfalls mit Locks umsetzbar

## → Erzeuger-/Verbraucher-Problem

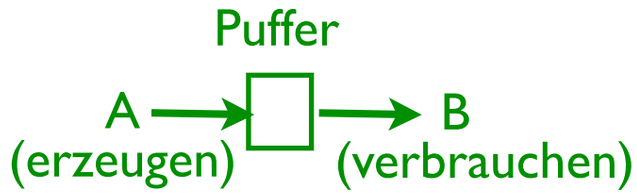
- Erst erzeugen, dann verbrauchen ⇒ Einseitige Synchronisation

  ⇒ neues Konzept: Ereignisvariablen (nächste Folie)

# Erzeuger-/Verbraucher-Problem

- Neues Konzept: Ereignisvariablen (Condition Variables)

  ⇒ Realisierung einseitiger Synchronisation

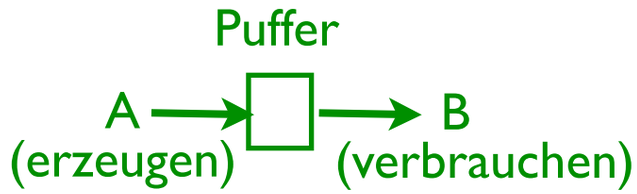



```
class Condition {
...
public:
  Condition();
  wait();
  signal();
}

Condition c;
```

# Erzeuger-/Verbraucher-Problem

- Neues Konzept: Ereignisvariablen (Condition Variables)

  ⇒ Realisierung einseitiger Synchronisation




```
class Condition {
...
public:
  Condition();
  wait();
  signal();
}

Condition c;
```

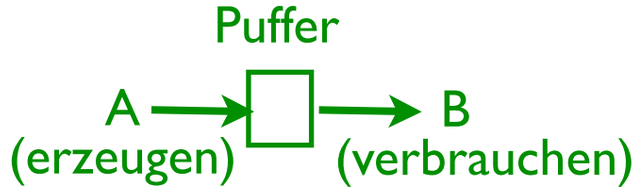
- Nutzung:

```
   A     Condition c;     B

                          c.wait();
erzeugen();               verbrauchen();
c.signal();
```

# Erzeuger-/Verbraucher-Problem

- Neues Konzept: Ereignisvariablen (Condition Variables)

  ⇒ Realisierung einseitiger Synchronisation

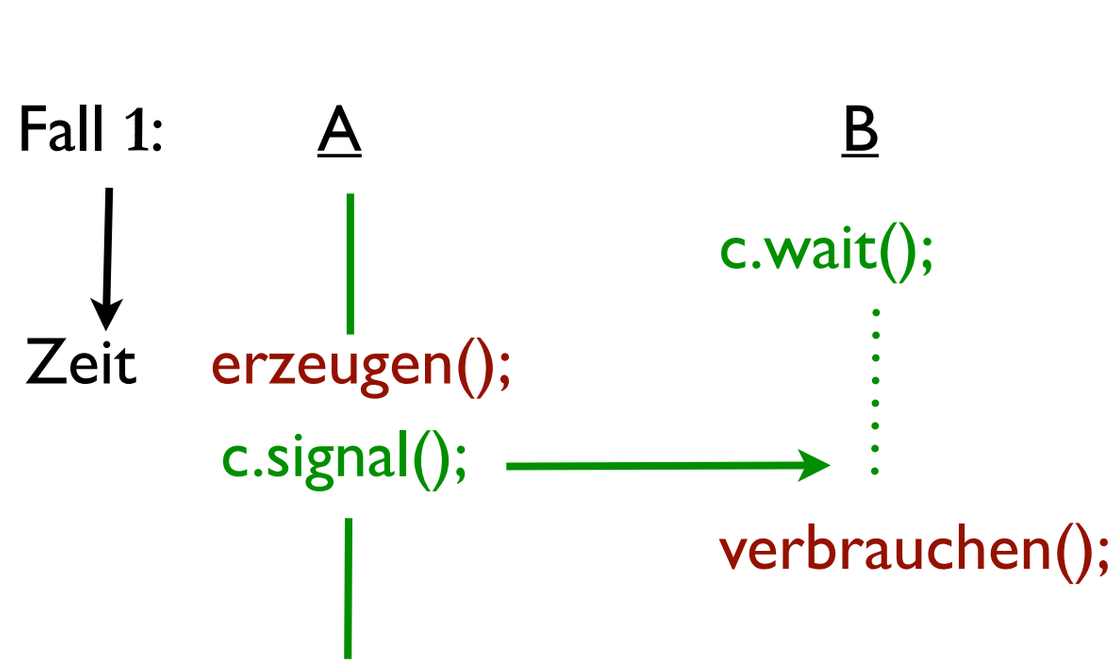



```
class Condition {
...
public:
  Condition();
  wait();
  signal();
}
```

```
Condition c;
```

- Nutzung:

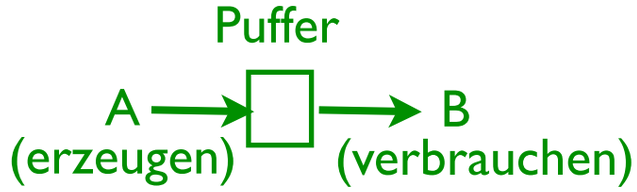
```
   A     Condition c;     B

                          c.wait();
erzeugen();               verbrauchen();
c.signal();
```



# Erzeuger-/Verbraucher-Problem

- Neues Konzept: Ereignisvariablen (Condition Variables)

  ⇒ Realisierung einseitiger Synchronisation

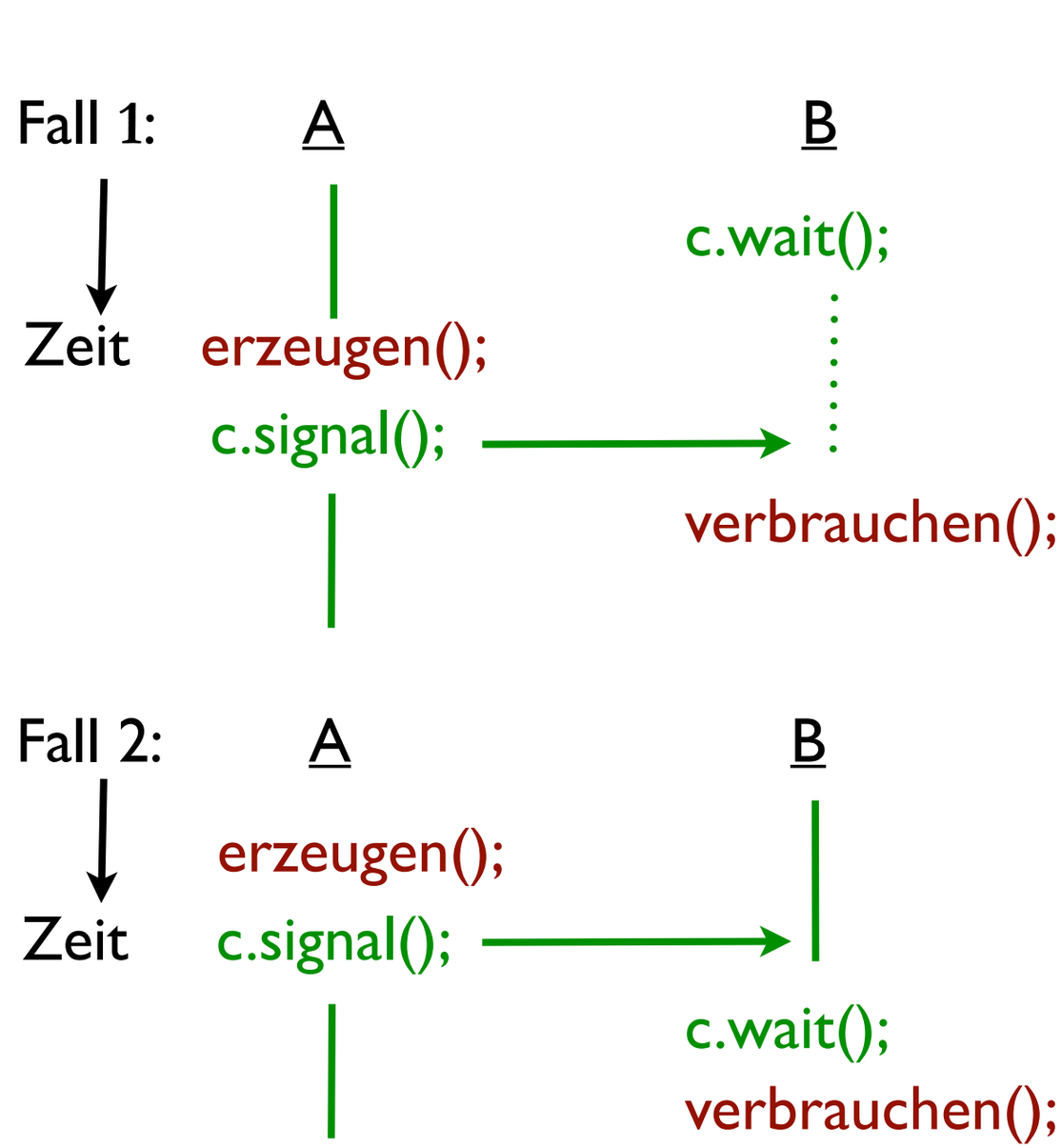



```
class Condition {
...
public:
  Condition();
  wait();
  signal();
}
```

```
Condition c;
```

- Nutzung:

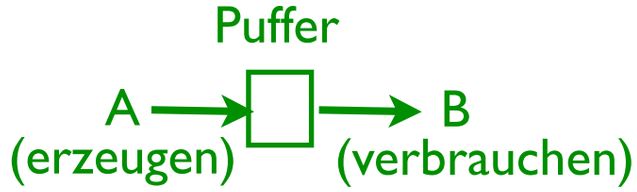
| A | Condition c; | B |
|---|---|---|
| | | `c.wait();` |
| `erzeugen();` | | `verbrauchen();` |
| `c.signal();` | | |



# Erzeuger-/Verbraucher-Problem

- Neues Konzept: Ereignisvariablen (Condition Variables)

  ⇒ Realisierung einseitiger Synchronisation

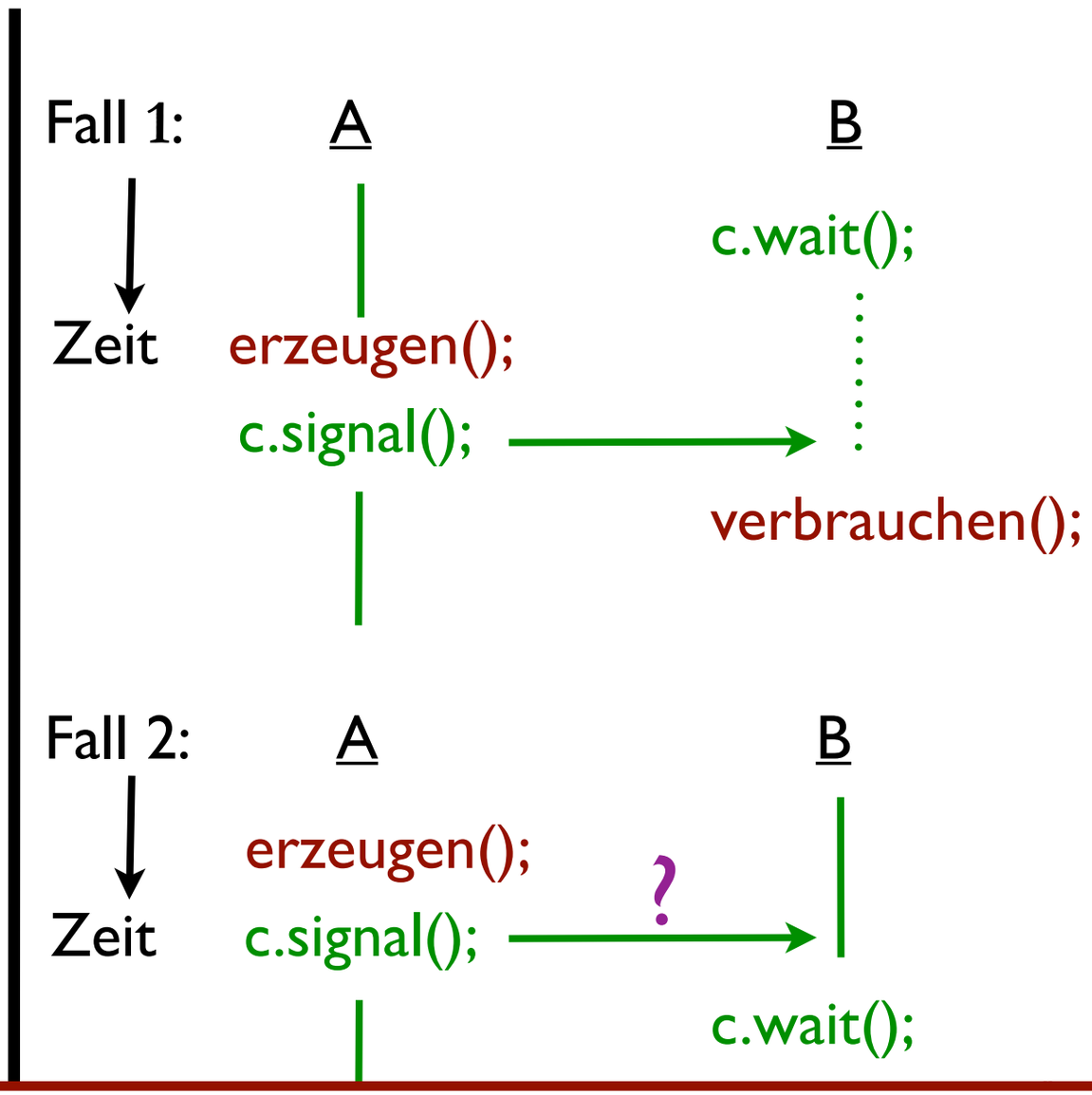



```
class Condition {
...
public:
  Condition();
  wait();
  signal();
}
```

```
Condition c;
```

- Nutzung:

| A | Condition c; | B |
|---|---|---|
| | | `c.wait();` |
| `erzeugen();` | | `verbrauchen();` |
| `c.signal();` | | |

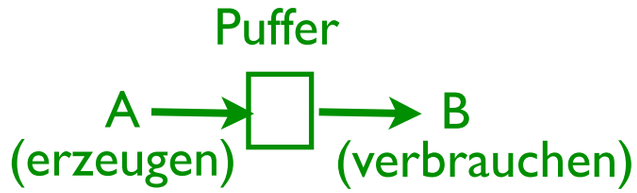



- Gibt auch `signal()`-Varianten, die im Nicht-Wartezustand verpuffen!
- Achtung: `signal()` in dieser Form nicht sammelbar!

# Erzeuger-/Verbraucher-Problem

- Neues Konzept: Ereignisvariablen (Condition Variables)

→ Auch Realisierung über Locking-Mechanismus möglich




```
class Condition {
...
public:
  Condition();
  wait();
  signal();
}

Condition c;
```

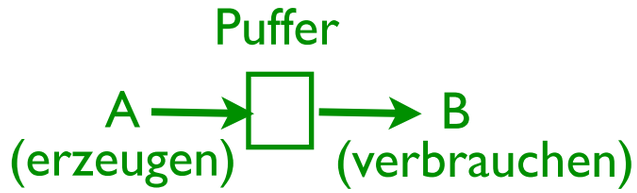
- Nutzung:

```
   A        Condition c;    B

                          c.wait();
erzeugen();               verbrauchen();
c.signal();
```

# Erzeuger-/Verbraucher-Problem

- Neues Konzept: Ereignisvariablen (Condition Variables)

→ Auch Realisierung über Locking-Mechanismus möglich




```
class Condition {
  …
public:
  Condition();
  wait();
  signal();
}

Condition c;
```

```
class Mutex {
  bool key;
public:
  Mutex() {key=false;}
  lock();
  unlock();
}

Mutex e;
```

- Nutzung:

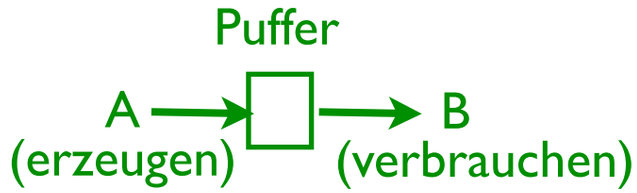
```
A     Condition c;     B

                       c.wait();
erzeugen();            verbrauchen();
c.signal();
```

- Achtung: `signal()` in dieser Form nicht sammelbar!
- Gibt auch `signal()`-Varianten, die im Nicht-Wartezustand verpuffen!

# Erzeuger-/Verbraucher-Problem

- Neues Konzept: Ereignisvariablen (Condition Variables)

→ Auch Realisierung über Locking-Mechanismus möglich




```
class Condition {          class Mutex {
  …                          bool key;
public:                    public:
  Condition();               Mutex() {key=false;}
  wait();                    lock();
  signal();                  unlock();
}                          }

Condition c;               Mutex c;
```

- Nutzung:

```
   A          Mutex c;        B

                              c.lock();
erzeugen();                   verbrauchen();
c.unlock();
```

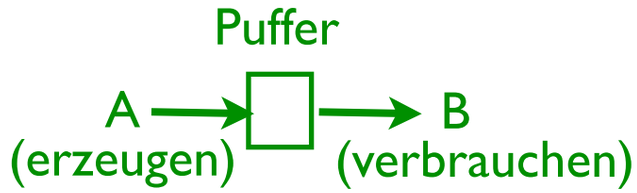
?

- Achtung: `signal()` in dieser Form nicht sammelbar!
- Gibt auch `signal()`-Varianten, die im Nicht-Wartezustand verpuffen!

# Erzeuger-/Verbraucher-Problem

- Neues Konzept: Ereignisvariablen (Condition Variables)

→ Auch Realisierung über Locking-Mechanismus möglich




```
class Condition {
  …
public:
  Condition();
  wait();
  signal();
}

Condition c;
```

```
class Mutex {
  bool key;
public:
  Mutex(bool init) {key=init;}
  lock();
  unlock();
}

Mutex c(true);
```

- Nutzung:

```
   A        Mutex c(true);        B

                          c.lock();
erzeugen();               verbrauchen();
c.unlock();
```

# Kleine Aufgabe

Kann man die Implementierung von Ereignisvariablen direkt auf sleep()/wakeup() abbilden?

**Statt:**

```
A                    Condition c;          B

                                           c.wait();
erzeugen();                                verbrauchen();
c.signal();
```

**Nun also:**

```
A                    ....                  B

                                           sleep(wchan);
erzeugen();                                verbrauchen();
wakeup(wchan);
```

# Anwendung auf Beispielsituationen

## Kritischer Abschnitt

- Entweder A oder B ⇒ mehrseitige Synchronisation

```
     A                Mutex m;              B

…                                      …
m.lock();                              m.lock();
… kritischer Abschnitt…                … kritischer Abschnitt …
m.unlock();                            m.unlock();
```

## Leser-/Schreiber-Problem

- Leser nebenläufig zulassen
- Schreibender Zugriff muss exklusiv sein
- Während des Schreibens auch keine Leser zulassen

  ⇒ ebenfalls mit Locks umsetzbar

## Erzeuger-/Verbraucher-Problem

- Erst erzeugen, dann verbrauchen ⇒ Einseitige Synchronisation

  ⇒ neues Konzept: Ereignisvariablen

## → Speisende Philosophen

- Stäbchen als kritische Abschnitte modellierbar
- Vermeidung der potentiellen Verklemmung erfordert zusätzliche Maßnahmen (⇒ später)

# Fragen – Teil 2

- Wie kann man eine einseitige Synchronisation mit Hilfe von `wait()` und `signal()` vornehmen? Wie kann man diese Primitiven in etwa auf `lock()` und `unlock()` abbilden?

# Teil 3:
# Aktives vs. blockierendes Warten

# Aktives vs. blockierendes Warten

- Bisherige Locking-Algorithmen verwenden aktives Warten (busy waiting)
- Verharren in Schleife, bis kritischer Abschnitt frei: Spinlocks
- Nachteil: Verbrauchen unnötig Prozessorkapazität durch permanente Abfragen (bis Zeitscheibe aufgebraucht)
- Bei Einprozessorsystemen: Sowieso Prozesswechsel erforderlich, damit anderer Prozess aus kritischem Abschnitt austreten kann
- Bei kurzen kritischen Abschnitten in Multiprozessorsystemen i.d.R. o.k.

# Aktives vs. blockierendes Warten

- Bisherige Locking-Algorithmen verwenden aktives Warten (busy waiting)
- Verharren in Schleife, bis kritischer Abschnitt frei: Spinlocks
- Nachteil: Verbrauchen unnötig Prozessorkapazität durch permanente Abfragen (bis Zeitscheibe aufgebraucht)
- Bei Einprozessorsystemen: Sowieso Prozesswechsel erforderlich, damit anderer Prozess aus kritischem Abschnitt austreten kann
- Bei kurzen kritischen Abschnitten in Multiprozessorsystemen i.d.R. o.k.

Alternative: Blockierendes Warten

- Prozess legt sich schlafen, wenn kritischer Abschnitt nicht frei ⇒ sleep()
- Bei Verlassen des kritischen Abschnitts darauf wartende Prozesse aufwecken ⇒ wakeup()
- (In unserer Implementierungsumgebung so realisiert)

- Im folgenden Verwendung folgender Terminologie:
  - lock()/unlock() (bzw. mutex_lock()/mutex_unlock())

    ⇒ für Locking-Algorithmen allgemein

  - spin_lock()/spin_unlock()

    ⇒ für aktives Warten

  - block_lock()/block_unlock()

    ⇒ für blockierendes Warten

- Außerdem: objektorientierte Implementierung verwenden

# Realisierung von Spinlocks

- entspricht der bisherigen Klasse Mutex

```
class Spin
  bool key;
public:
  Spin() {key = false;}
  spin_lock() {while(test_and_set(key));}
  spin_unlock() {key = false;}
}

Spin s;
...

s.spin_lock();
...
s.spin_unlock();
```

# Realisierung von blockierendem Warten

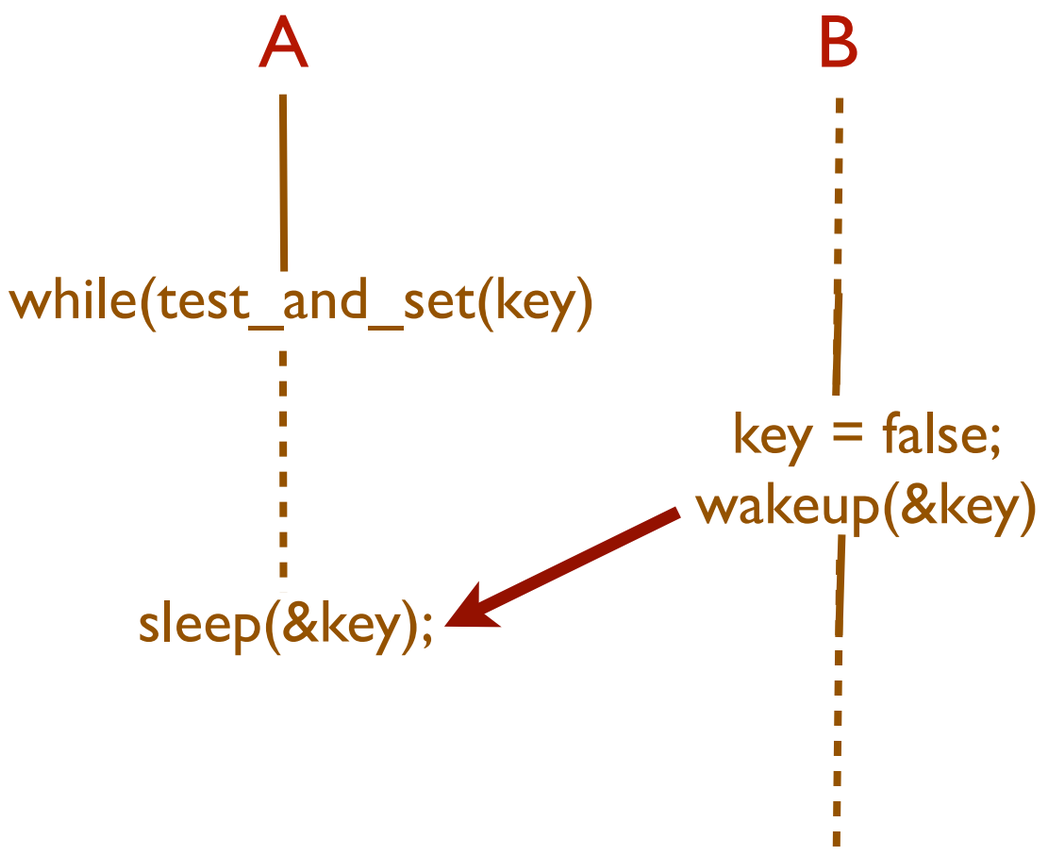
1. Versuch

```
class Block {
  bool key;
public:
  Block() {key = false;}

  void block_lock() {
    while(test_and_set(key))
      sleep(&key);
  }

  void block_unlock() {
    key = false;
    wakeup(&key);
  }
}
```

# Realisierung von blockierendem Warten

```
class Block {
  bool key;
public:
  Block() {key = false;}

  void block_lock() {
    while(test_and_set(key))
      sleep(&key);
  }

  void block_unlock() {
    key = false;
    wakeup(&key);
  }
}
```

1. Versuch




- Achtung: block_lock()/block-unlock() sind selbst kritische Abschnitte:
  - Verwendung von unteilbarer Operation (test_and_set)
  - Zugriff auf Sleep-Queue (wakeup() ist verpuffend)

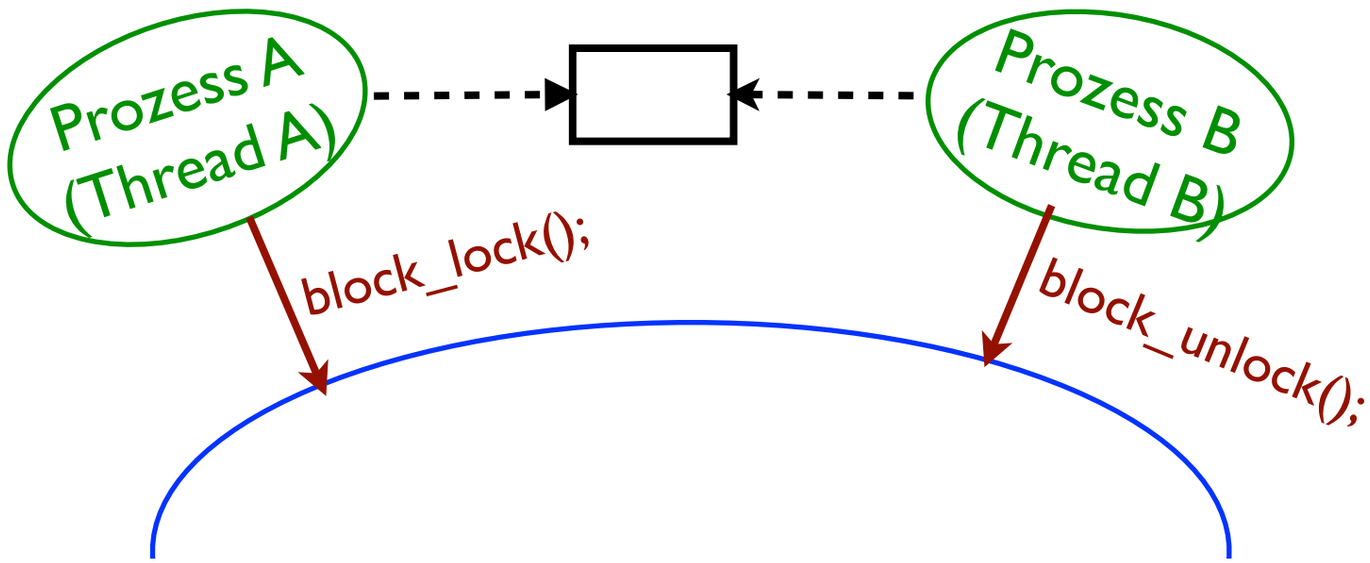
    ⇒ „Lost-Wakeup-Problem“

# Lösungen des Lost-Wakeup-Problems (in Unix)

a) Bei Einprozessorsystem:

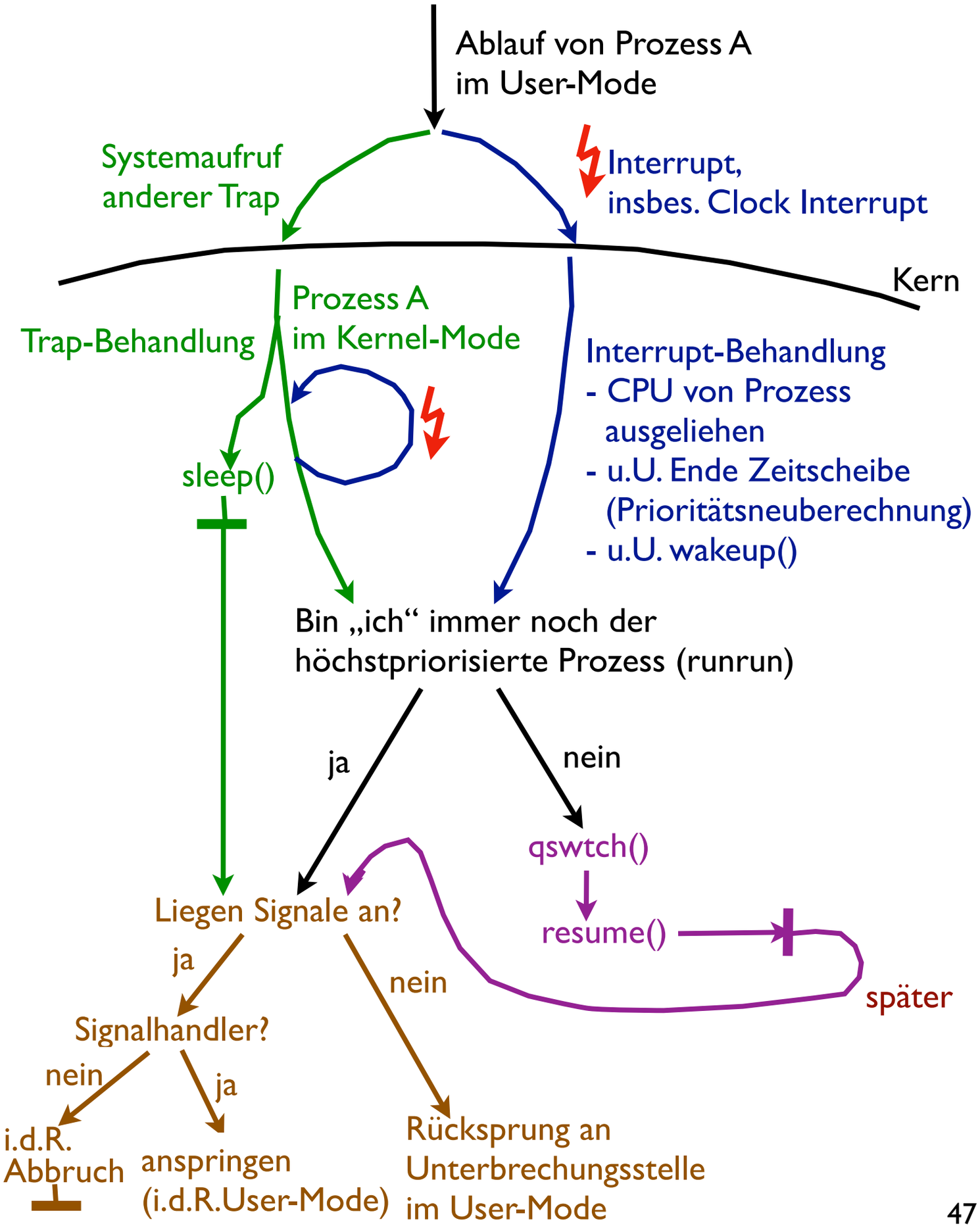
- Realisierung von blockierendem Lock im Unix-Kern

  ⇒ Nicht-Präemption im Kern verhindert unerwünschten Prozesswechsel



# Einordnung einer Prozessumschaltung im Prozessablauf

(vereinfacht)



- Allerdings auch im Kern Nebenläufigkeit möglich:

  Kernel-Mode A  Interrupt-Handler

  [Interrupt-Handler  Interrupt-Handler]

  ⇒ können auch auf gemeinsame Datenstrukturen zugreifen

  ⇒ gleiches Lockingverfahren auch dafür nutzbar?

- Allerdings auch im Kern Nebenläufigkeit möglich:

  Kernel-Mode A ⇝ Interrupt-Handler

  [Interrupt-Handler ⇝ Interrupt-Handler]

  ⇒ können auch auf gemeinsame Datenstrukturen zugreifen

  ⇒ gleiches Lockingverfahren auch dafür nutzbar?

- Interrupt-Handler sind zeitkritisch und haben keinen eigenen Prozesskontext

  ⇒ können sich nicht schlafenlegen

  ⇒ können zur mehrseitigen Synchronisation kein blockierendes Warten verwenden

  (stattdessen Unterbrechungsausschluss)

- Aber Teilnahme an einseitiger Synchronisation möglich:
  - Prozess wartet auf Ereignis (wait() ≙ lock())
  - Interrupt-Handler signalisiert es (signal() ≙ unlock())

```
Interrupt-Handler                    Prozess A (Kern)
      ...                                  ...
//erzeugen                          block_lock();  //≙ wait()
block_unlock(); //≙ signal()   //verbrauchen
```

⇒ dafür blockierendes Warten (mit BELEGT-Initialisierung) nutzbar

# Realisierung von blockierendem Warten

```
class Block {
  bool key;
public:
  Block(bool init) {key = init;} //Bei einseitiger Sync: init==true

  void block_lock() {                    // Prozess

    while(test_and_set(key))
      sleep(&key);
  }

  void block_unlock() {                  // ggf. Interrupt-Handler
    key = false;
    wakeup(&key);
  }
}
```

- Weiterhin Lost-Wakeup möglich

# Realisierung von blockierendem Warten

2. Versuch

```
class Block {
  bool key;
public:
  Block(bool init) {key = init;} //Bei einseitiger Sync: init==true

  void block_lock() {                    // Prozess
    disable_interrupts();
    while(test_and_set(key))
      sleep(&key);
    enable_interrupts();
  }

  void block_unlock() {                  // ggf. Interrupt-Handler
    key = false;
    wakeup(&key);
  }
}
```

- Vermeidung des Lost-Wakeup durch Interrupt-Ausschluss
- (Kein Problem bei unlock(): Kann ruhig von Interrupt-Handler unterbrochen werden)

```
void block_lock() {                    // Prozess
    disable_interrupts();
    while(test_and_set(key))
      sleep(&key);
    enable_interrupts();
  }
```

Geht das denn gut?

- Prozess legt sich im Zustand ausgeschalteter Interrupts schlafen

  ⇒ Wer sollte ihn wieder aufwecken?

```
void block_lock() {                    // Prozess
    disable_interrupts();
    while(test_and_set(key))
      sleep(&key);
    enable_interrupts();
  }
```

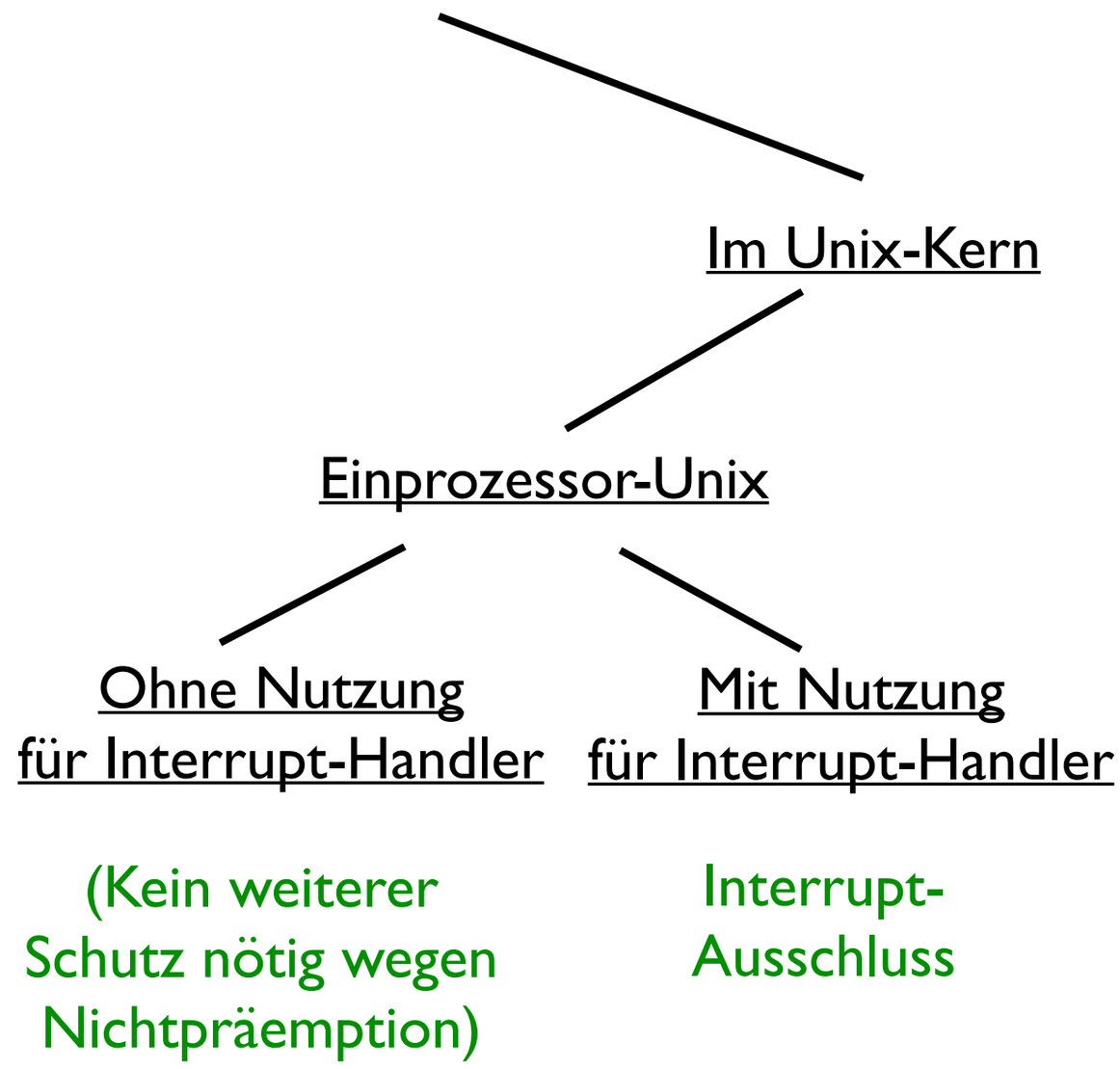
Geht das denn gut?

- Prozess legt sich im Zustand ausgeschalteter Interrupts schlafen

  ⇒ Wer sollte ihn wieder aufwecken?

- In Unix hat jeder Prozess einen eigenen Interrupt-Kontext
  - nach Prozesswechsel sind/werden die Interrupts vermutlich wieder aktiviert
  - sofern nicht auch bei jenem gerade ausgeschaltet...

# Zusammenfassung: Realisierung von block_lock()



## b) Bei Mehrprozessorsystemen:

- Nicht-Präemption bezieht sich nur auf eine CPU

  ⇒ Prozesse auf verschiedenen CPUs können gleichzeitig (auch im Kern) laufen

# Realisierung von blockierendem Warten

1. Versuch

```
class Block {
  bool key;
public:
  Block(bool init) {key = init;}

  void block_lock() {                    // Prozess 1 auf CPU 1

    while(test_and_set(key))
      sleep(&key);
  }

  void block_unlock() {                  // Prozess 2 auf CPU 2
    key = false;
    wakeup(&key);
  }
}
```

- Weiterhin Lost-Wakeup möglich

# Realisierung von blockierendem Warten

1. Versuch

```
class Block {
  bool key;
public:
  Block(bool init) {key = init;}

  void block_lock() {                    // Prozess 1 auf CPU 1
    disable_interrupts();
    while(test_and_set(key))
      sleep(&key);
    enable_interrupts();
  }

  void block_unlock() {                  // Prozess 2 auf CPU 2
    key = false;
    wakeup(&key);
  }
}
```

- Weiterhin Lost-Wakeup möglich

## b) Bei Mehrprozessorsystemen:

- Nicht-Präemption bezieht sich nur auf eine CPU

  ⇒ Prozesse auf verschiedenen CPUs können gleichzeitig (auch im Kern) laufen

- Vermeidung des Lost-Wakeup-Problems bei blockierenden Locks durch Schutz des in block_lock() enthaltenen kritischen Abschnitts mit eigenem aktiven Lock (Spinlock)

# Blockierendes Warten in Mehrprozessorsystem

```
class Block {
  bool key;
  Spin s;
public:
  Block();
  void block_lock();
  void block_unlock();
};
...

void Block::block_lock() {

  s.spin_lock();
  while (test_and_set(key))
    sleep  (this );
  s.spin_unlock();

}

void Block::block_unlock() {

  s.spin_lock();
  key = false;
  wakeup(this);
  s.spin_unlock();

}
```

# Blockierendes Warten in Mehrprozessorsystem

```
class Block {
  bool key;
  Spin s;
public:
  Block();
  void block_lock();
  void block_unlock();
};
...

void Block::block_lock() {

  s.spin_lock();
  while (test_and_set(key))
    sleep  (this );
  s.spin_unlock();

}

void Block::block_unlock() {

  s.spin_lock();
  key = false;
  wakeup(this);
  s.spin_unlock();

}
```

Achtung: Prozess darf sich nicht mit Spinlock schlafenlegen

⇒ Verklemmung

- A hat Spin, will in k.A.
- B will k.A. verlassen, braucht Spin

# Blockierendes Warten in Mehrprozessorsystem

```
class Block {
  bool key;
  Spin s;
public:
  Block();
  void block_lock();
  void block_unlock();
};
...

void Block::block_lock() {

  s.spin_lock();
  while (test_and_set(key))
    sleep  (this );
  s.spin_unlock();

}

void Block::block_unlock() {

  s.spin_lock();
  key = false;
  wakeup(this);
  s.spin_unlock();

}
```

Achtung: Prozess darf sich nicht mit Spinlock schlafenlegen

⇒ Verklemmung

- A hat Spin, will in k.A.
- B will k.A. verlassen, braucht Spin

⇒ Verwendung einer sleep-Variante:
sleepl()

# Blockierendes Warten in Mehrprozessorsystem

```
class Block {
  bool key;
  Spin s;
public:
  Block();
  void block_lock();
  void block_unlock();
};
...

void Block::block_lock() {

  s.spin_lock();
  while (test_and_set(key))
    sleep  (this );
  s.spin_unlock();

}

void Block::block_unlock() {

  s.spin_lock();
  key = false;
  wakeup(this);
  s.spin_unlock();

}
```

## Verwaltung der Sleep-Queue

```
struct SleepQ {
  …

};

SleepQ slq;  //eigentlich viele

sleep  (void *chan          ) {

  … Einreihen in die SleepQ …


  swtch();

}

wakeup(void *chan) {

  … Entnehmen aus der SleepQ …

}
```

# Blockierendes Warten in Mehrprozessorsystem

```
class Block {
  bool key;
  Spin s;
public:
  Block();
  void block_lock();
  void block_unlock();
};
...

void Block::block_lock() {

  s.spin_lock();
  while (test_and_set(key))
    sleepl (this,s);
  s.spin_unlock();

}

void Block::block_unlock() {

  s.spin_lock();
  key = false;
  wakeup(this);
  s.spin_unlock();

}
```

## Verwaltung der Sleep-Queue

```
struct SleepQ {
  …

};

SleepQ slq;  //eigentlich viele

  sleepl (void *chan, Spin &sp) {

  … Einreihen in die SleepQ …

    sp.spin_unlock();
    swtch();
    sp.spin_lock();
  }

  wakeup(void *chan) {

  … Entnehmen aus der SleepQ …

  }
```

# Blockierendes Warten in Mehrprozessorsystem

```
class Block {
  bool key;
  Spin s;
public:
  Block();
  void block_lock();
  void block_unlock();
};
...

void Block::block_lock() {

  s.spin_lock();
  while (test_and_set(key))
    sleepl (this,s);
  s.spin_unlock();

}

void Block::block_unlock() {

  s.spin_lock();
  key = false;
  wakeup(this);
  s.spin_unlock();

}
```

## Verwaltung der Sleep-Queue

```
struct SleepQ {
  …
  Spin s2;
};

SleepQ slq;  //eigentlich viele

sleepl (void *chan, Spin &sp) {
  slq.s2.spin_lock();
  … Einreihen in die SleepQ …
  slq.s2.spin_unlock();
  sp.spin_unlock();
  swtch();
  sp.spin_lock();
}

wakeup(void *chan) {
  slq.s2.spin_lock();
  … Entnehmen aus der SleepQ …
  slq.s2.spin_unlock();
}
```

# Blockierendes Warten in Mehrprozessorsystem

```
class Block {
  bool key;
  Spin s;
public:
  Block();
  void block_lock();
  void block_unlock();
};
...

void Block::block_lock() {

  s.spin_lock();
  while (test_and_set(key))
    sleepl (this,s);
  s.spin_unlock();

}

void Block::block_unlock() {

  s.spin_lock();
  key = false;
  wakeup(this);
  s.spin_unlock();

}
```

## Verwaltung der Sleep-Queue

```
struct SleepQ {
  …
  Spin s2;
};

SleepQ slq;  //eigentlich viele

sleepl (void *chan, Spin &sp) {
  slq.s2.spin_lock();
  … Einreihen in die SleepQ …
  slq.s2.spin_unlock();
  sp.spin_unlock();
  swtch();
  sp.spin_lock();
}

wakeup(void *chan) {
  slq.s2.spin_lock();
  … Entnehmen aus der SleepQ …
  slq.s2.spin_unlock();
}
```

# Blockierendes Warten in Mehrprozessorsystem

```
class Block {
  bool key;
  Spin s;
public:
  Block();
  void block_lock();
  void block_unlock();
};
...
```

```
void Block::block_lock() {
  disable_interrupts();
  s.spin_lock();
  while (test_and_set(key))
    sleepl (this,s);
  s.spin_unlock();
  enable_interrupts();
}

void Block::block_unlock() {
  disable_interrupts();
  s.spin_lock();
  key = false;
  wakeup(this);
  s.spin_unlock();
  enable_interrupts();
}
```

Bei Nutzung durch Interrupt-Handler:
– darf nicht in Spin hängenbleiben

⇒ zusätzlich Interrupt-Ausschluss erforderlich

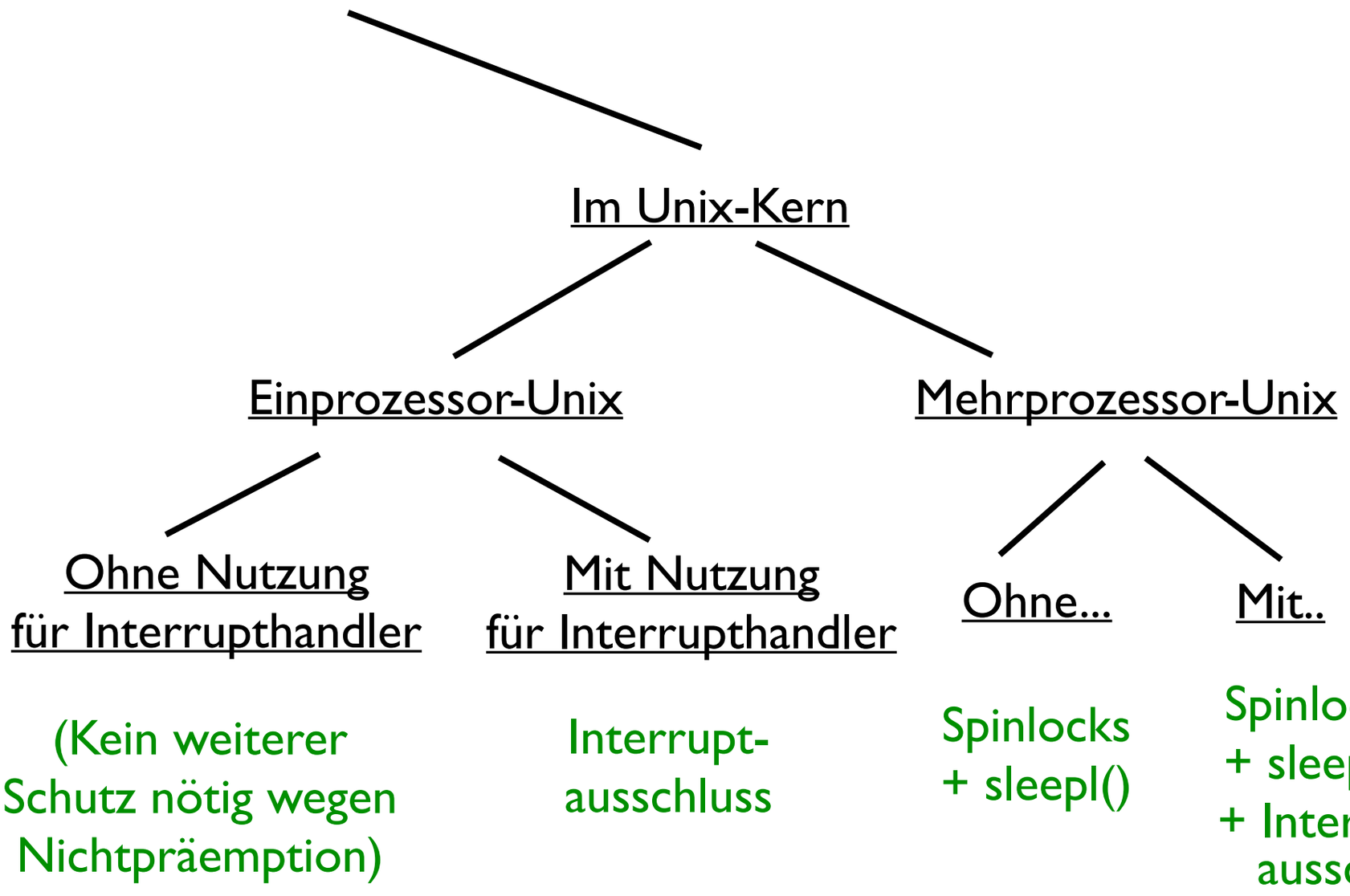
# Blockierendes Warten in Mehrprozessorsystem

```
class Block {
  bool key;
  Spin s;
public:
  Block();
  void block_lock();
  void block_unlock();
};
...

void Block::block_lock() {
  disable_interrupts();
  s.spin_lock();
  while (test_and_set(key))
    sleepl (this,s);
  s.spin_unlock();
  enable_interrupts();
}

void Block::block_unlock() {
  disable_interrupts();
  s.spin_lock();
  key = false;
  wakeup(this);
  s.spin_unlock();
  enable_interrupts();
}
```

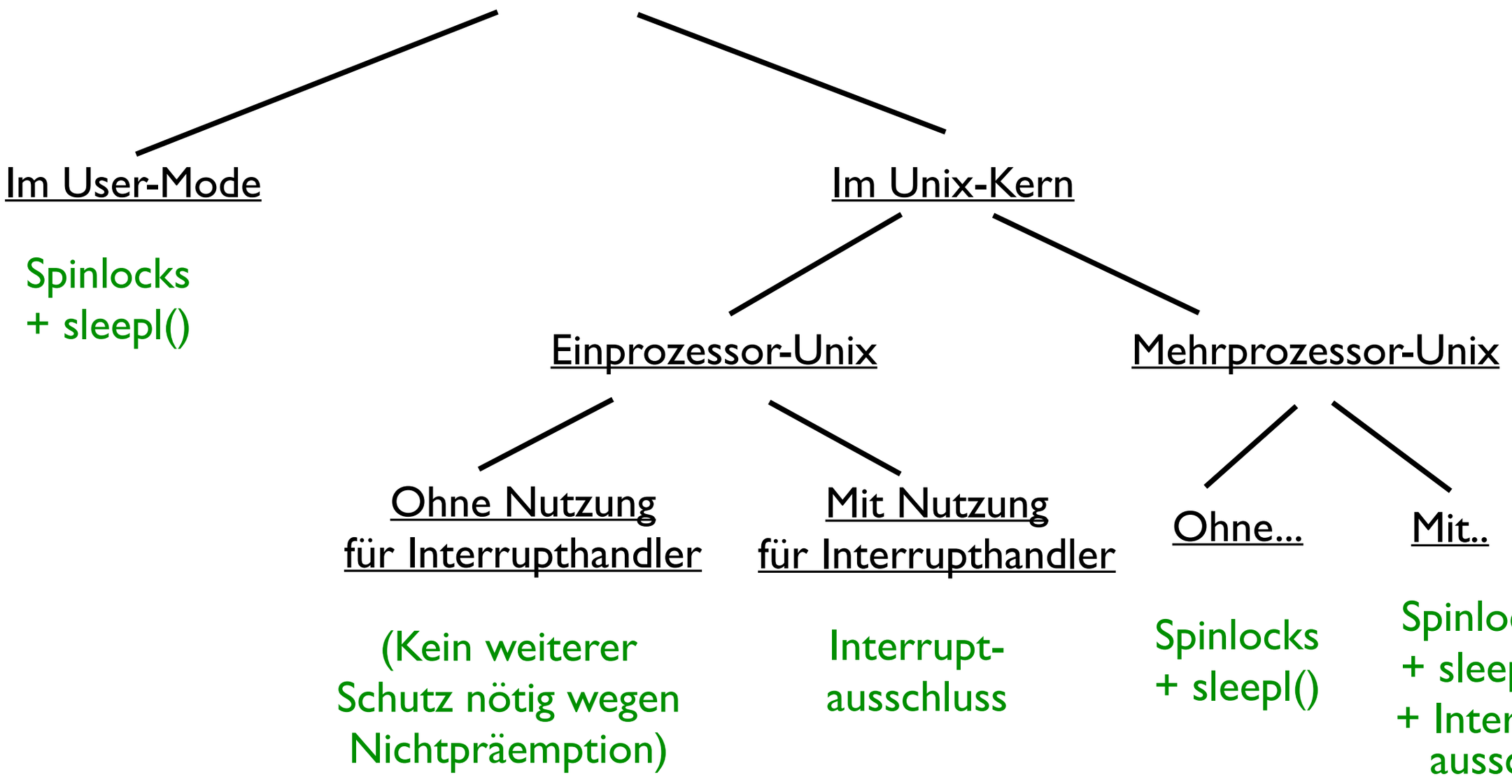
## Verwaltung der Sleep-Queue

```
struct SleepQ {
  …
  Spin s2;
};

SleepQ slq;  //eigentlich viele

sleepl (void *chan, Spin &sp) {
  slq.s2.spin_lock();
  sp.spin_unlock();
… Einreihen in die SleepQ …
  slq.s2.spin_unlock();
    swtch();
  sp.spin_lock();
}

wakeup(void *chan) {
  slq.s2.spin_lock();
  … Entnehmen aus der SleepQ …
  slq.s2.spin_unlock();
}
```

# Zusammenfassung: Realisierung von block_lock()



# Zusammenfassung: Realisierung von block_lock()



# Fragen – Teil 3

- Grenze die Begriffe *aktives* und *blockierendes* Warten gegeneinander ab.
- In einer Unix-Multiprozessorumgebung können mehrere Prozesse nebenläufig `sleep()`/`wakeup()` aufrufen. Warum ist dies ein kritischer Abschnitt? Warum kann man ihn nicht einfach dadurch schützen, dass man den Aufruf von `sleep()`/`wakeup()` von einem *Spinlock* umgibt? Was wird man stattdessen tun?

# Zusammenfassung

- Lösung Leser-Schreiber-Problem mit Lock-Verfahren
- Ereignisvariablen vs. Locks
- Aktives Warten vs. blockierendes Warten
- Realisierung von blockierenden Locks
- Blockierende Locks im Mehrprozessorsystem

# Locks vs. Ereignisvariablen – Fragen

1. Warum kann man das Leser/Schreiber-Problem nicht mit einem einzigen Lock lösen?
2. Wie kann man eine einseitige Synchronisation mit Hilfe von `wait()` und `signal()` vornehmen? Wie kann man diese Primitiven in etwa auf `lock()` und `unlock()` abbilden?
3. Grenze die Begriffe *aktives* und *blockierendes* Warten gegeneinander ab.
4. In einer Unix-Multiprozessorumgebung können mehrere Prozesse nebenläufig `sleep()`/`wakeup()` aufrufen. Warum ist dies ein kritischer Abschnitt? Warum kann man ihn nicht einfach dadurch schützen, dass man den Aufruf von `sleep()`/`wakeup()` von einem *Spinlock* umgibt? Was wird man stattdessen tun?