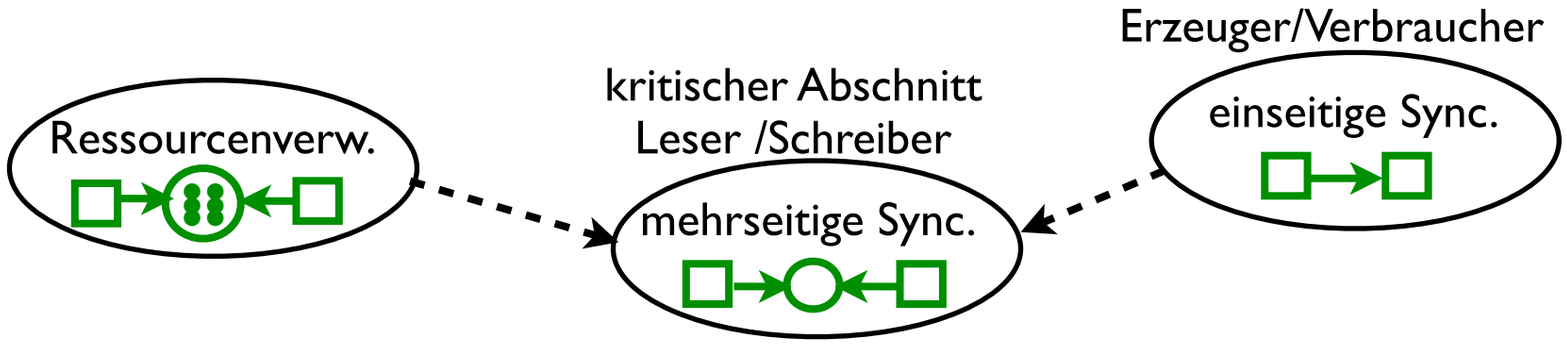
Work in progre

Monitore

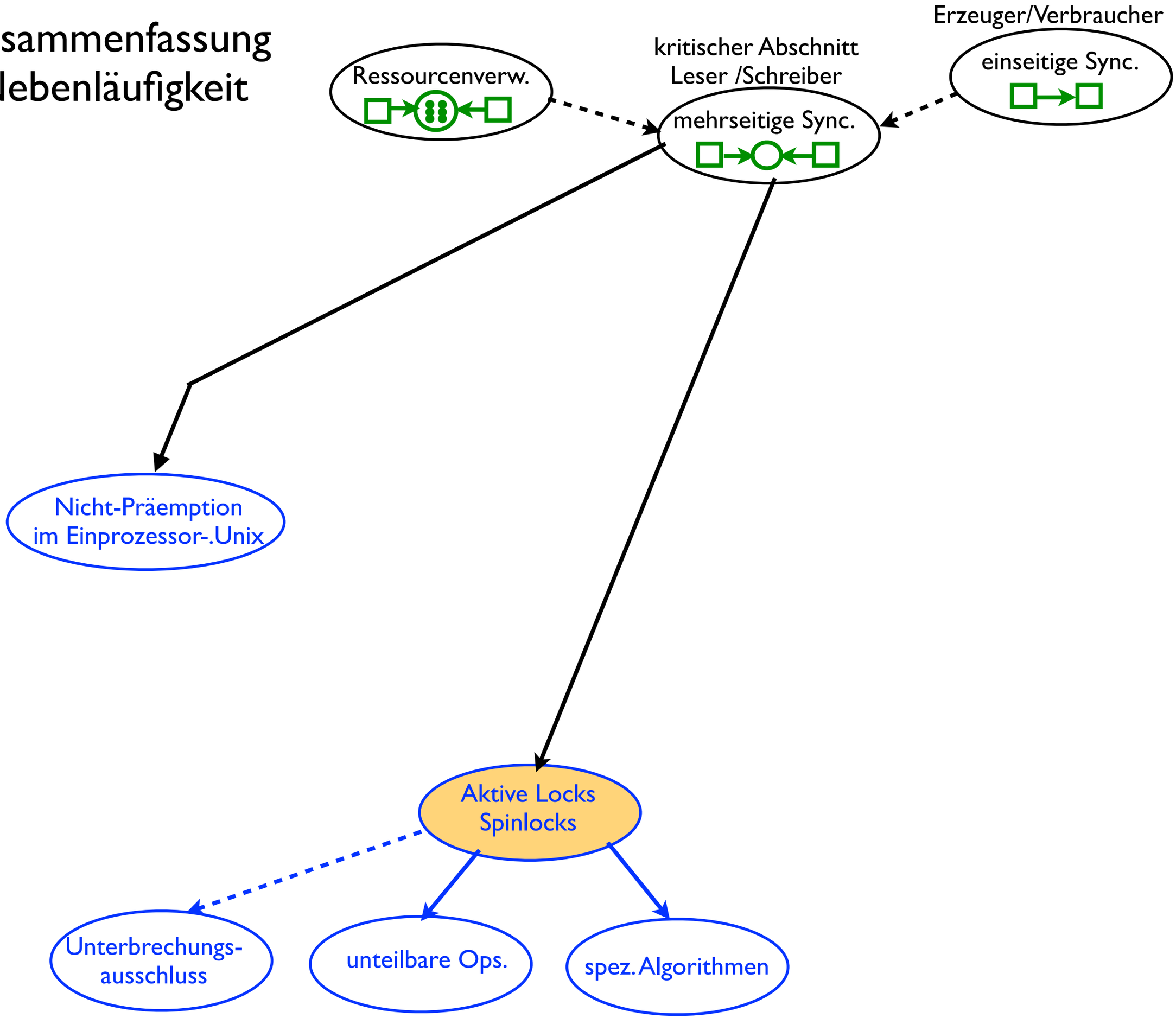
Ute Bormann, Tel2

2023-10-13

Zusammenfassung Nebenläufigkeit



Zusammenfassung Nebenläufigkeit

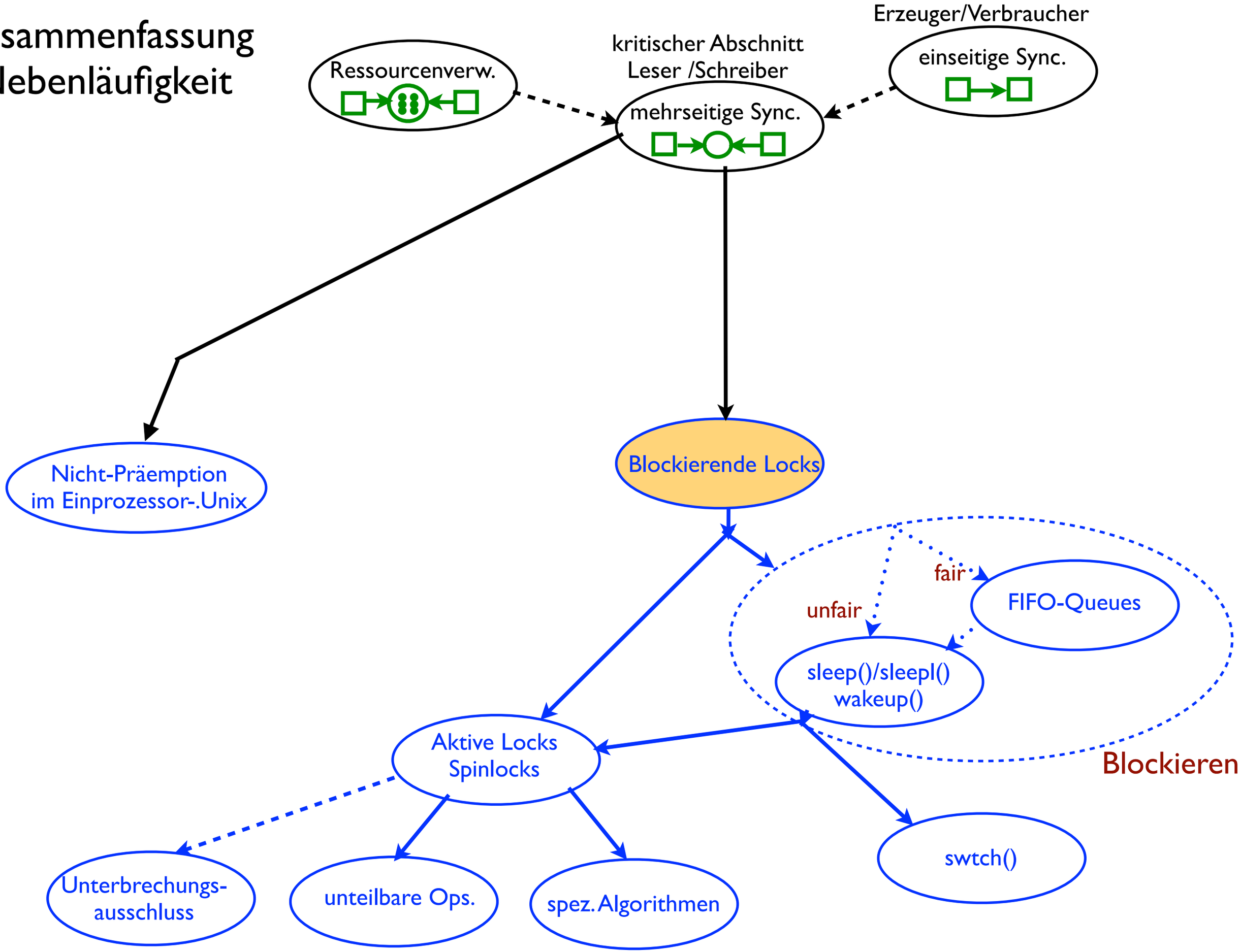


Realisierung von Spinlocks

- 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();
//kritischer Abschnitt
s.spin_unlock();
```

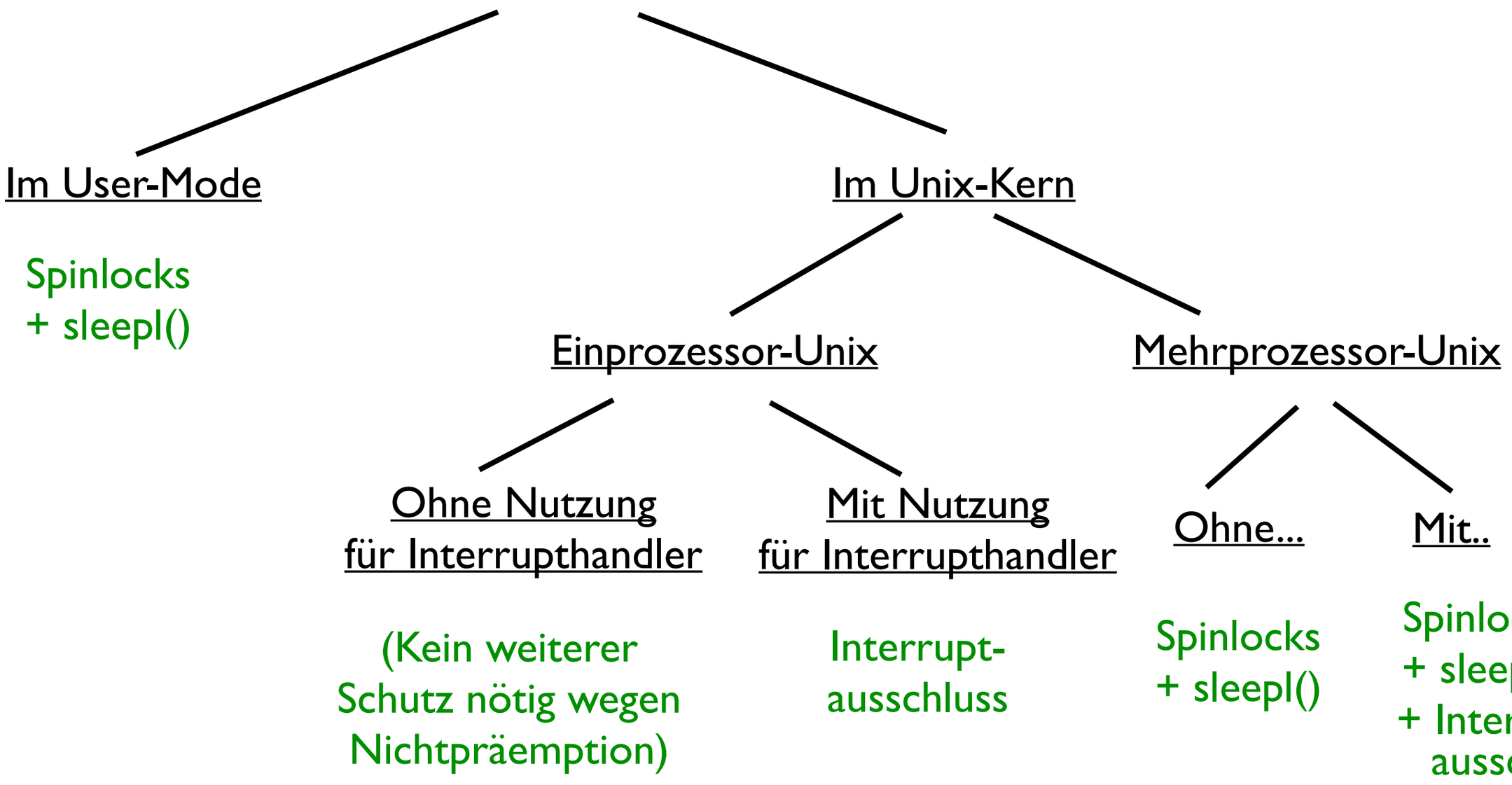
Zusammenfassung Nebenläufigkeit



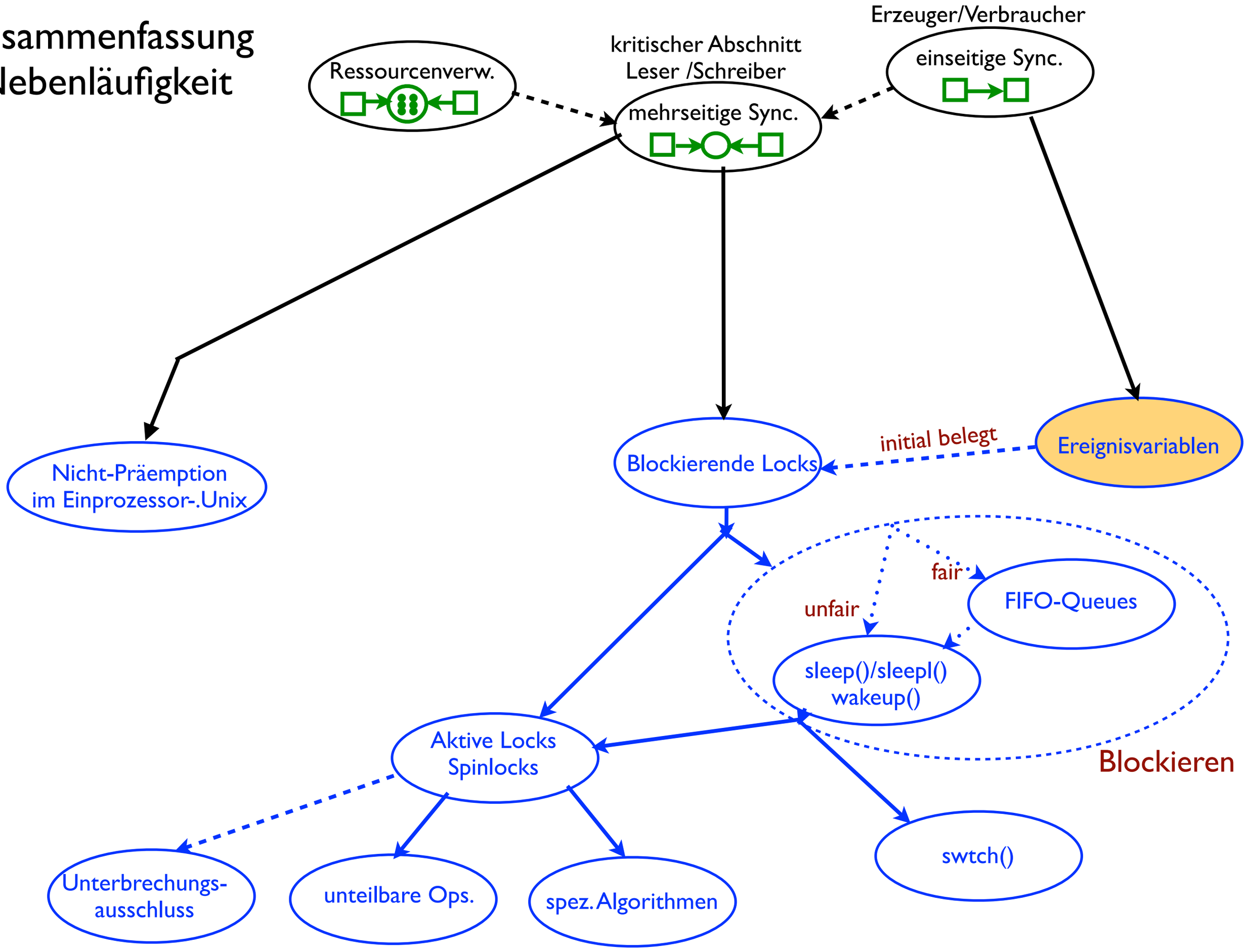
Blockierendes Warten in User Mode

```
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();

}
```

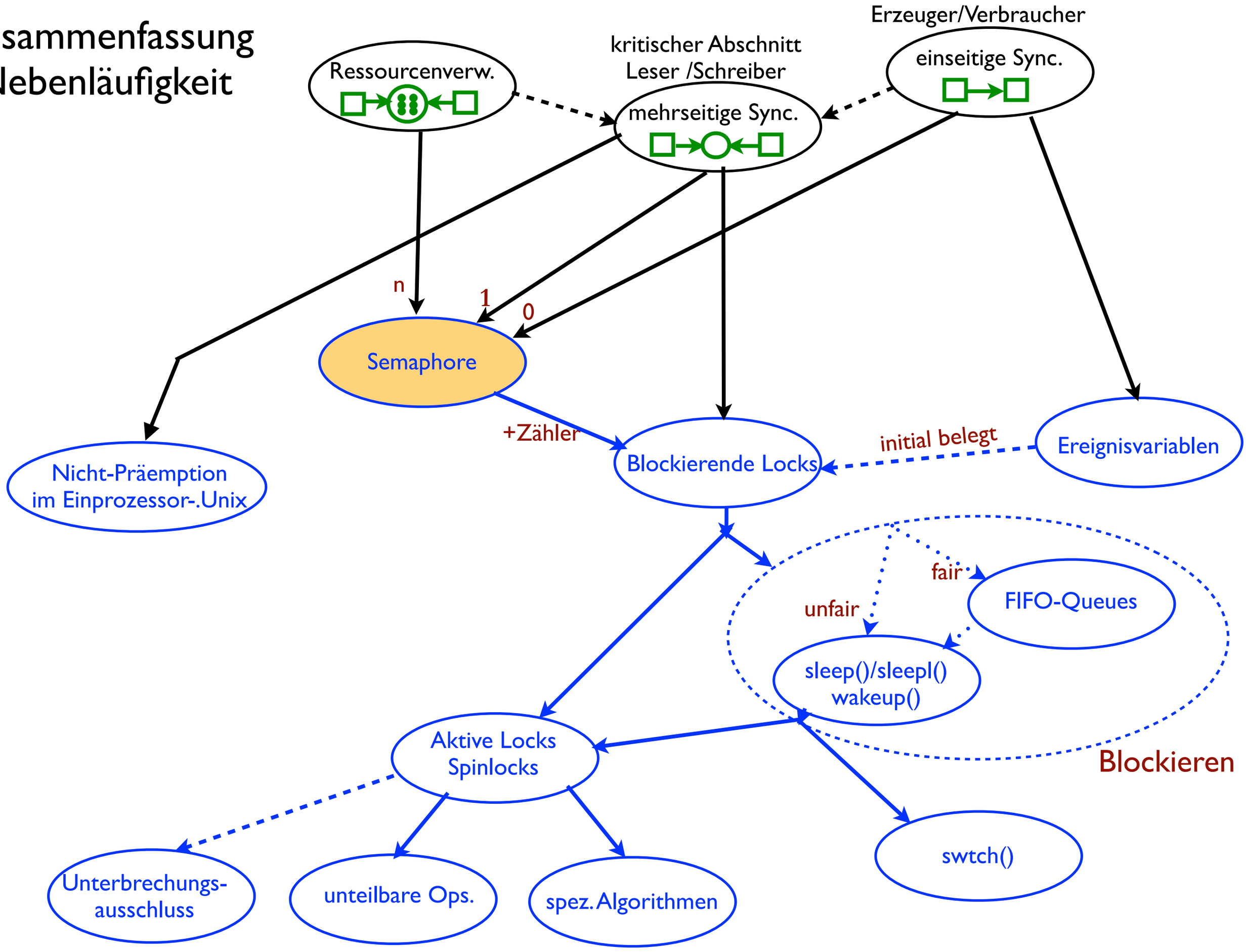
Zusammenfassung: Realisierung von block_lock()



Zusammenfassung Nebenläufigkeit



Zusammenfassung Nebenläufigkeit



Realisierung von Semaphoren im User Mode

- Abstrakte Implementierung:

```
class Sema {
...
}

void Sema:P() {
  s.spin_lock();
  while (counter==0)
    sleepl(this,s);
  --counter;
  s.spin_unlock();

}
```

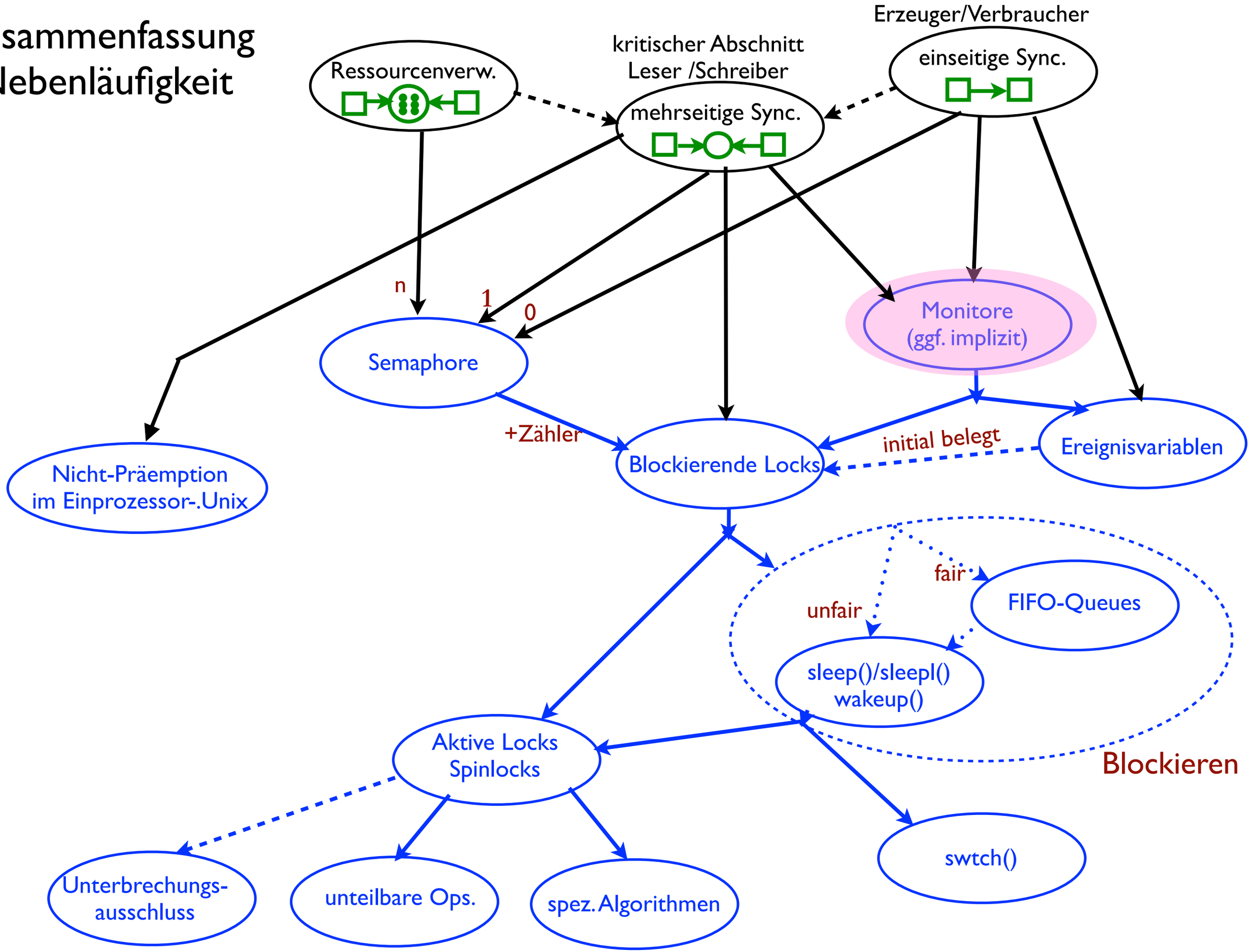
```
void Sema::V() {
  s.spin_lock();
  ++counter;
  s.spin_unlock();

  wakeup(this);
  }
```

Werbeblock

PROBE: „**Pro**jektvorstellung mit allen **Be**teiligten“

- Für alle, die in 2023 von Fach Informatik angebotenes Bachelorprojekt belegen wollen
- Derzeitige Planung:
    - Ab 16.01.2023, Kurzvorstellungen des Projektangebots: Kurzbeschreibungen/Folien in Stud.IP-Veranstaltung „Vorstellung Bachelorprojekte 2023“
    - Mo 16.01.2023, 16:15-17:45, Überblick über das Projektstudium: BBB-Meeting (+ Audio-Folien) in Stud.IP-Veranstaltung „Vorstellung Bachelorprojekte 2023“
    - Mo 23.1. – Fr 27.1.2023: Schnuppertermine der Projekte in Präsenz (je 1h) Konkrete Termine werden noch bekanntgegeben. Voraussichtlich:
        - Mo 23.1. 16-19
        - Di 24.1. 08-10
        - Mi 25.1. 08-10 un 12-14
        - Do 26.1. 16-19
        - Fr 27.1. 14-17
    - Mo 30.1. – Fr 3.2.2023: Projektwahl (über Web-Formular)

Zusammenfassung Nebenläufigkeit



Inhalt

1. Eigenschaften von Monitoren
2. Umsetzung von Monitoren

Teil 1:
Eigenschaften von Monitoren

19 Monitore, Ute Bormann

Beurteilung von Semaphoren

- Einfache Programmierabstraktion
- Flexibel einsetzbar
- Aber auch Probleme:

a) Wie lock()/unlock() kein Schutz vor falscher Nutzung

- Aufruf von P()/V() kann vergessen werden
- P()/V() nicht an Blockstrukturen gebunden
- P()/V() nicht immer regelmäßig geschachtelt

  ⇒ Durch ungeschickte Schachtelung Verklemmungen möglich

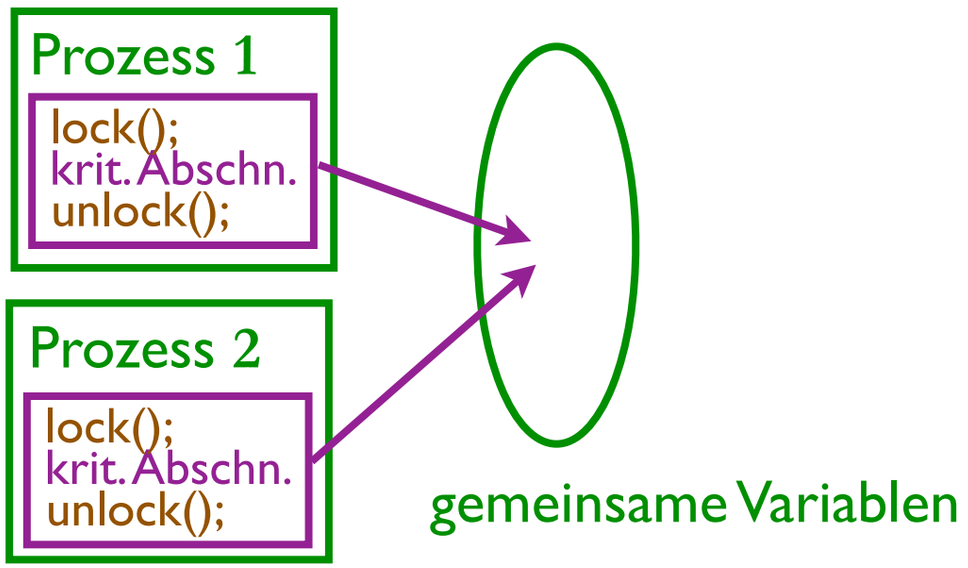
...

Monitore (Brinch Hansen 1973 / Hoare 1974)

- Schutz von kritischen Abschnitten bisher durch explizite Eintritts-/ Austrittsprotokolle realisiert
  - lock()/unlock() (Schlossvariable)
  - P()/V() (Semaphore)

⇒ über das Programm verstreut

⇒ kann leicht vergessen werden

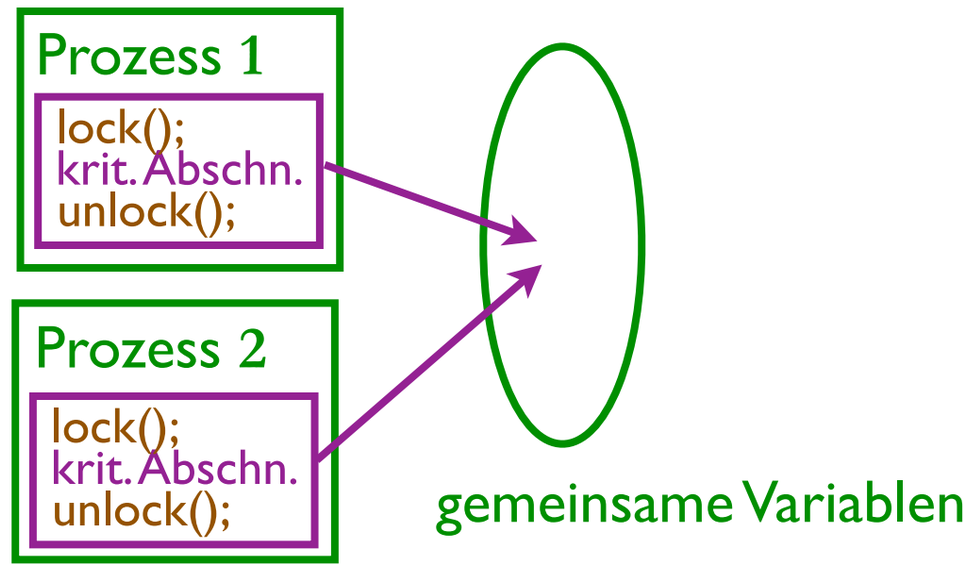


Monitore (Brinch Hansen 1973 / Hoare 1974)

- Schutz von kritischen Abschnitten bisher durch explizite Eintritts-/ Austrittsprotokolle realisiert
  - lock()/unlock() (Schlossvariable)
  - P()/V() (Semaphore)

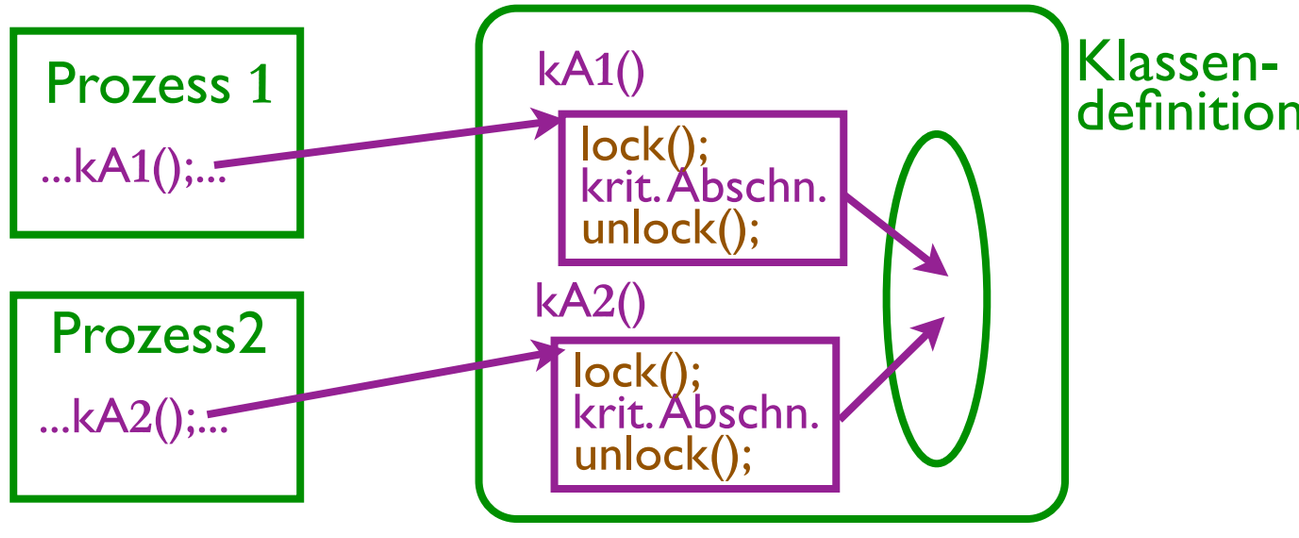
⇒ über das Programm verstreut

⇒ kann leicht vergessen werden



Stattdessen:

1. Verwendung von abstrakten Datentypen/Klassen

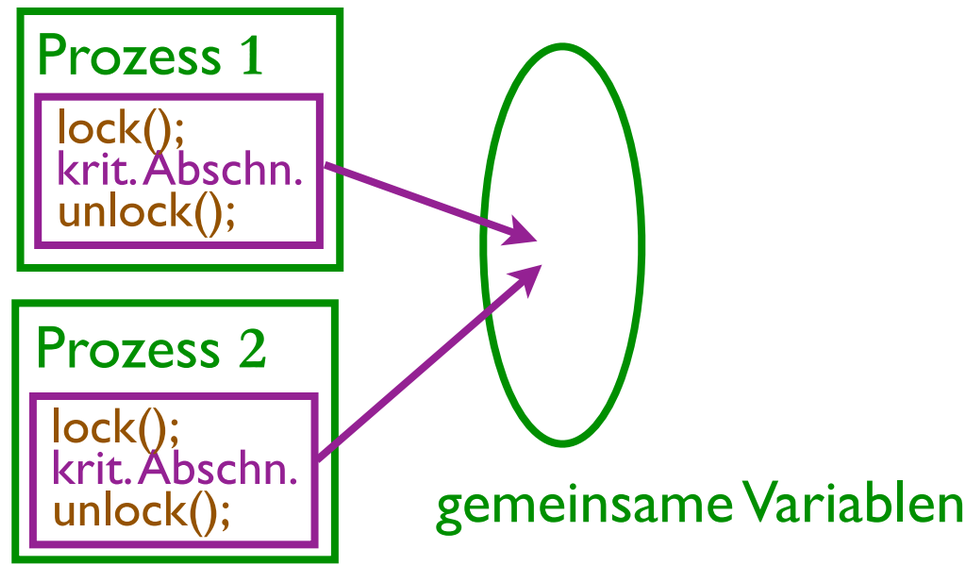


Monitore (Brinch Hansen 1973 / Hoare 1974)

- Schutz von kritischen Abschnitten bisher durch explizite Eintritts-/ Austrittsprotokolle realisiert
  - lock()/unlock() (Schlossvariable)
  - P()/V() (Semaphore)

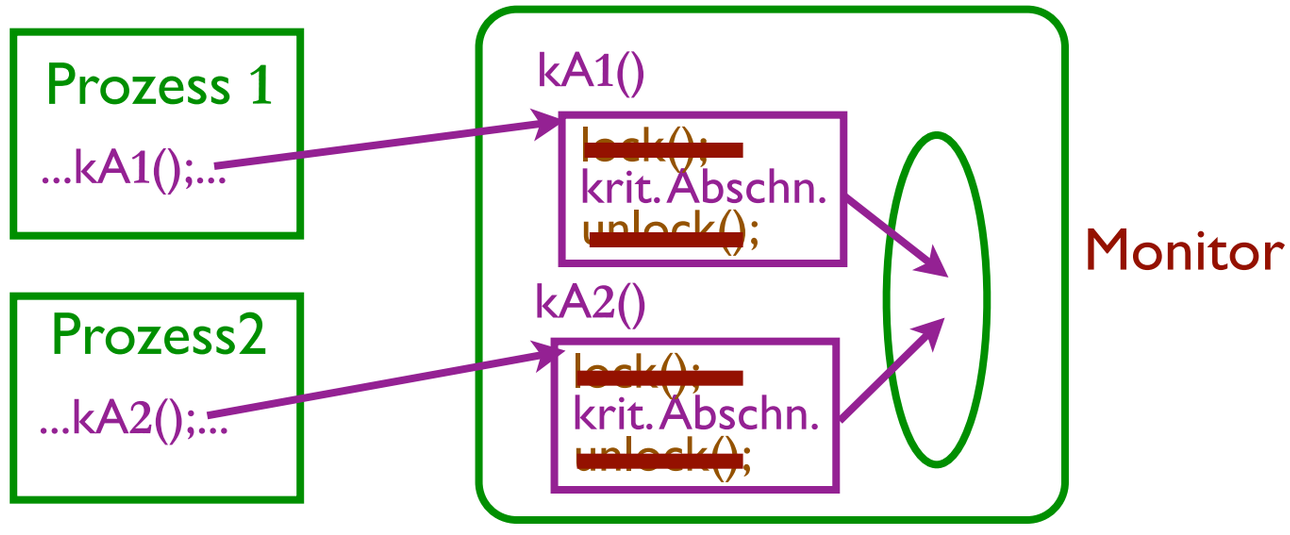
⇒ über das Programm verstreut

⇒ kann leicht vergessen werden



Stattdessen:

2. Verwendung von Monitoren
   - spezielle abstrakte Datentypen/Klassen
   - implizit gegen Nebenläufigkeit geschützt



Folge: Nebenläufigkeitsschutz ist im Monitor versteckt

⇒ Muss nicht mitprogrammiert werden

Achtung: Klassische Monitore gibt es in C++ nicht
⇒ Freistilsyntax!

Beispiel: Definition eines Monitors

```
monitor Zaehler {
  int z;
public:
  Zaehler (int i);
  void inkrement();
  void dekrement();
  int wert();
};
```

Folge: Nebenläufigkeitsschutz ist im Monitor versteckt

⇒ Muss nicht mitprogrammiert werden

Beispiel: Definition eines Monitors

```
monitor Zaehler {
  int z;
public:
  Zaehler (int i);
  void inkrement();
  void dekrement();
  int wert();
};
```

```
...  //Konstruktor

void Zaehler::inkrement() {

  ++z;

}

void Zaehler::dekrement() {

  --z;

}

int Zaehler::wert() {
  return z;
}
```

Folge: Nebenläufigkeitsschutz ist im Monitor versteckt

⇒ Muss nicht mitprogrammiert werden

Beispiel: Definition eines Monitors

```
monitor Zaehler {
  int z;
public:
  Zaehler (int i);
  void inkrement();
  void dekrement();
  int wert();
};
```

```
...  //Konstruktor

void Zaehler::inkrement() {

→ ++z;

}

  void Zaehler::dekrement() {

→ --z;

  }

  int Zaehler::wert() {
? → return z;
  }
```

Verwendung:

```
Zaehler i;
...
i.inkrement();
...
```

Folge: Nebenläufigkeitsschutz ist im Monitor versteckt

⇒ Muss nicht mitprogrammiert werden

Beispiel: Definition eines Monitors

```
class monitor Zaehler {
  int z;
public:
  Zaehler (int i);
  void inkrement();
  void dekrement();
  int wert();
};
```

Verwendung:

```
Zaehler i;
...
i.inkrement();
...
```

```
...  //Konstruktor

void Zaehler::inkrement() {
  lock(…);
  ++z;
  unlock(…);
}

void Zaehler::dekrement() {
  lock(…);
  --z;
  unlock(…);
}

int Zaehler::wert() {
  lock(…);
  int i;
  i = z;
  unlock(…);
  return i;
}
```

⇒ Umsetzung des Monitors mit C++-Klassen macht Gebrauch von entsprechenden Schutzmechanismen (z.B. Locks oder Semaphoren)

- Monitore sehen darüber hinaus i.d.R. Ereignisvariable (Condition Variables) vor

  ⇒ Beschreibung weiterer Synchronisationsbeziehungen

  (insbes. einseitige Synchronisation)

- Zwei vordefinierte Operationen darauf:

  - signal(): Anzeigen eines Ereignisses (hier: verpuffend)
    ⇒ wenn niemand wartet, kein Effekt

  - wait(): Warten auf ein Ereignis (hier: unbedingtes Warten)
    ⇒ nur aufrufen, wenn Ereignis noch nicht eingetreten, also gewartet werden muss
    ⇒ zusätzliche Abfrage im Vorfeld erforderlich

- Beispiel: Erzeuger-Verbraucher-Problem

  ⇒ z.B. Integer-Mailbox (hier: ohne Überlaufschutz)

Integer-Mailbox (ohne Überlaufschutz)

```
monitor Mailbox {          //kein C++
  int wert;


public:
  Mailbox();
  void rein(int i);        //erzeugen
  int raus();              //verbrauchen
};
```

Integer-Mailbox (ohne Überlaufschutz)

```
monitor Mailbox {          //kein C++
  int wert;
  bool voll;
  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
```

```
class Condition {

public:
  Condition();
  void wait();
  void signal(); //verpuffend
};
```

Integer-Mailbox (ohne Überlaufschutz)

```
monitor Mailbox {          //kein C++
  int wert;
  bool voll;
  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ... voll = false;
```

```
class Condition {

public:
  Condition();
  void wait();
  void signal(); //verpuffend
};
```

```
void Mailbox::rein(int i){
  wert = i;              //ohne Überlaufschutz
  voll = true;
  gefuellt.signal();
}

int Mailbox::raus() {
  if (!voll)
    gefuellt.wait();
  voll = false;
  return(wert);
}
```

Integer-Mailbox (ohne Überlaufschutz)

```
monitor Mailbox {          //kein C++
  int wert;
  bool voll;
  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ... voll = false;
```

```
class Condition {

public:
  Condition();
  void wait();
  void signal(); //verpuffend
};
```

```
void Mailbox::rein(int i){
  wert = i;          //ohne Überlaufschutz
  voll = true;
  gefuellt.signal();
}

int Mailbox::raus() {
  if (!voll)
    gefuellt.wait(); //→ enqueue() → sleepl()...
  voll = false;
  return(wert);
}
```

Achtung: Auch hier gegenseitiger Ausschluss implizit (z.B. intern über lock()/unlock() realisiert)

⇒ kein „lost wakeup“

⇒ aber Nutzung von sleepl() erforderlich

Integer-Mailbox (ohne Überlaufschutz)

```
monitor Mailbox {          //kein C++
  int wert;
  bool voll;
  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ... voll = false;

void Mailbox::rein(int i){
  wert = i;          //ohne Überlaufschutz
  voll = true;
  gefuellt.signal();
}

int Mailbox::raus() {
  if (!voll)
    gefuellt.wait();
  voll = false;
  return(wert);
}
```

```
class Condition {

public:
  Condition();
  void wait();
  void signal(); //verpuffend
};
```

Zum Vergleich:

```
Sema s(0);
void Mailbox::rein(int i){
  wert = i;
  s.V();
}

int Mailbox::raus() {
  s.P();
  return(wert);
}
```

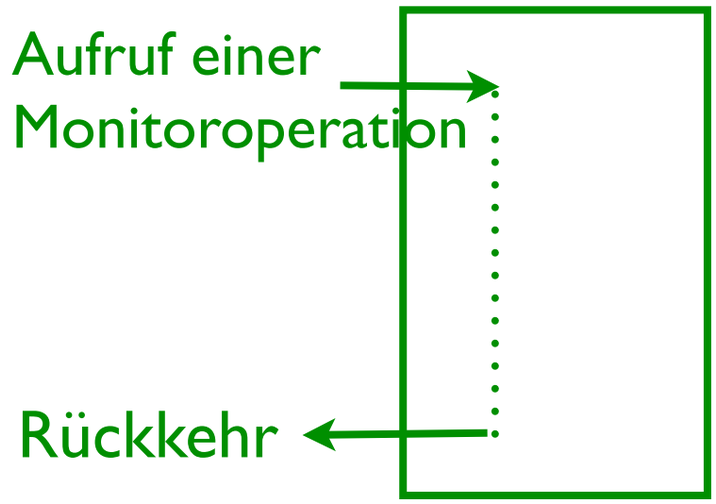
- signal() verpuffend vs. V() nicht verpuffend
  (Zustand im Semaphorzähler vermerkt)
- Bei Semaphor auch mehrere Puffer denkbar
  (aber hier nicht vorgesehen)
- Bei Semaphor: Lost Wakeup i.d.R.
  durch zusätzlichen Lock vermeiden

Original-Monitorkonzept:
Etliche Randbedingungen der Monitorrealisierung

⇒ damals auch Fairness wichtiges Entwurfsziel

- Gegenseitiger Ausschluss aller Operationen

„Eingänge“ und „Ausgänge“ eines Monitors

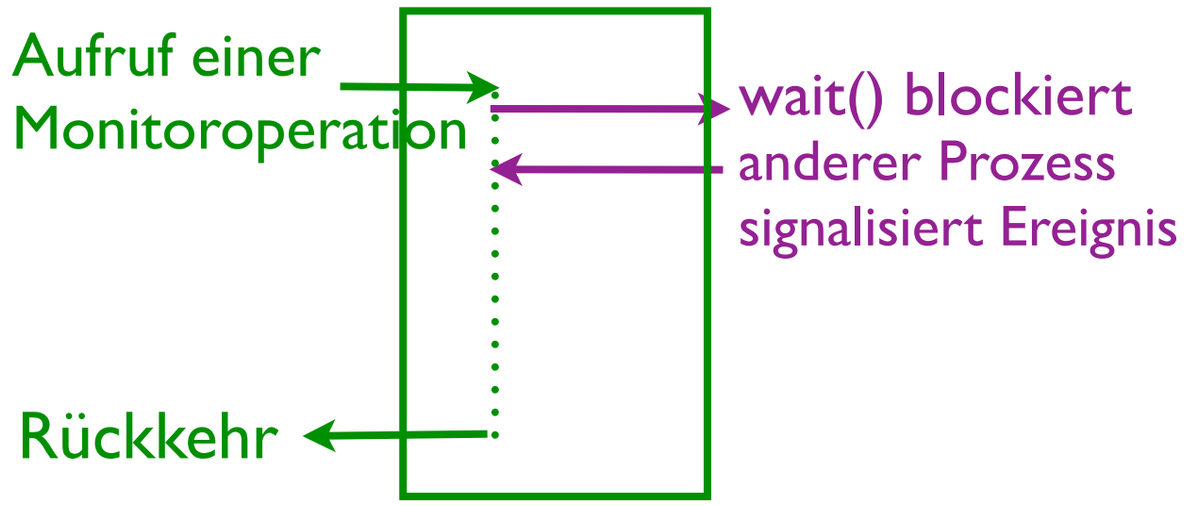


Original-Monitorkonzept: Etliche Randbedingungen der Monitorrealisierung

⇒ damals auch Fairness wichtiges Entwurfsziel

- Gegenseitiger Ausschluss aller Operationen
  ⇒ bei wait() muss Monitor verlassen werden
  (⇒ sleepl())
- Zu einem Zeitpunkt kann ein Prozess nur auf ein Ereignis warten
- Signale werden nicht gespeichert
  ⇒ wenn keiner wartet, geht es verloren

„Eingänge“ und „Ausgänge“ eines Monitors

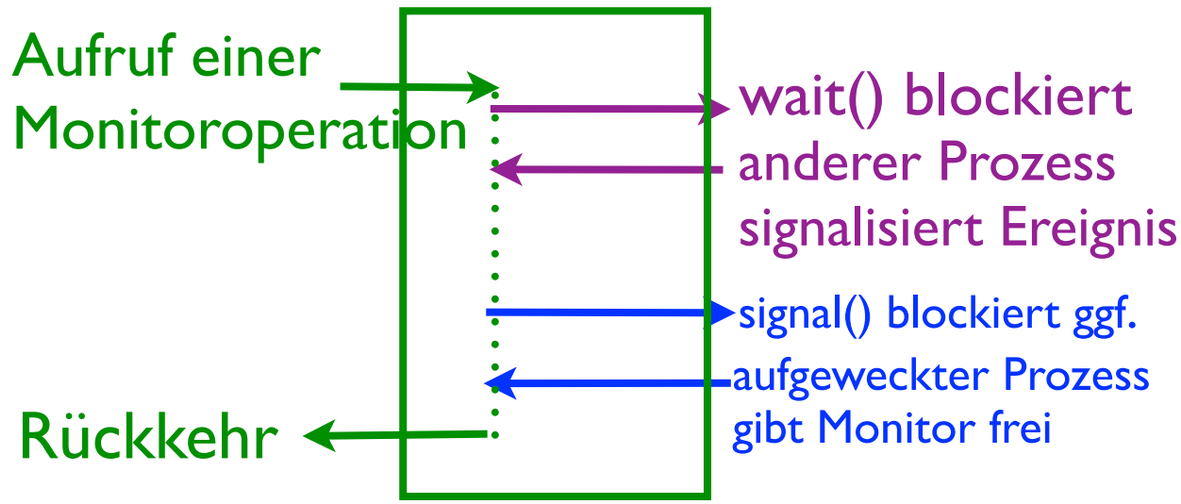


Original-Monitorkonzept: Etliche Randbedingungen der Monitorrealisierung

⇒ damals auch Fairness wichtiges Entwurfsziel

- Gegenseitiger Ausschluss aller Operationen
  ⇒ bei wait() muss Monitor verlassen werden
  (⇒ sleepl())
- Zu einem Zeitpunkt kann ein Prozess nur auf ein Ereignis warten
- Signale werden nicht gespeichert
  ⇒ wenn keiner wartet, geht es verloren
- Nach Signal wird Monitor an darauf wartenden Prozess übergeben (falls einer wartet)
  ⇒ besonders fair
- Wenn der wieder raus ist, kommt signalisierender Prozess wieder rein
  ⇒ besonders fair
- Faire Warteschlangen an den „Eingängen“

„Eingänge“ und „Ausgänge“ eines Monitors



Beschreibung des Leser-/Schreiber-Problems mit Monitoren

- „Entweder beliebig viele Leser oder ein Schreiber“

  ⇒ Leseoperation nicht direkt im Monitor realisierbar (sonst keine Nebenläufigkeit)

- Stattdessen Beginn/Ende des Lesens und Schreiben als Ganzes als Operationen vorsehen

- Im folgenden Umsetzung mit fairen Original-Monitoren

Zur Erinnerung: Lösung mit Locks

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 im k.A.
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

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

(Achtung: Leser werden bei dieser Lösung bevorzugt)

Zur Erinnerung: Lösung mit Locks

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 im k.A.
  s.lock();
  count--;
  if (count==0)
    rw.unlock();
  s.unlock();
  ...
}
```

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

(Achtung: Leser werden bei dieser Lösung bevorzugt)

```
monitor Leser-Schreiber {     //keine Monitore in C++
  int leser(0);               //C++: Initialisierung im Konstruktor
  Condition schreibbar;
```

```
void lesen_ein() {

  leser++;

}
```

```
void lesen_aus() {

  leser--;
  if (leser == 0)
    schreibbar.signal();  //⇒ dequeue();

}
```

```
void schreiben() {

  if (leser != 0)
    schreibbar.wait();    //⇒ enqueue();
  ... //schreiben
  schreibbar.signal();

}
```

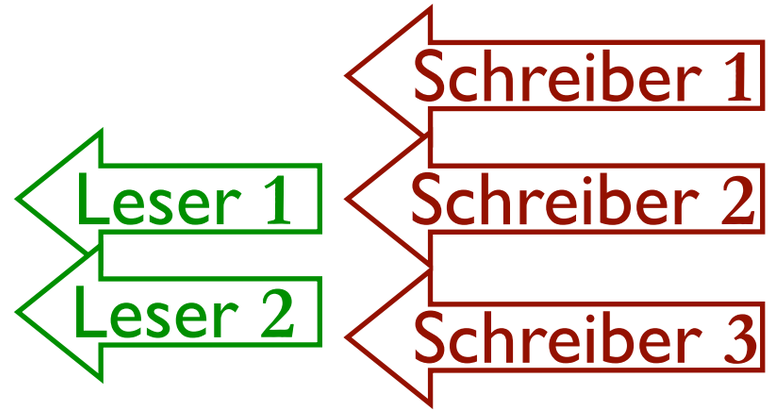
```
lesen_ein();

... //lesen

lesen_aus();
```

```
monitor Leser-Schreiber {     //keine Monitore in C++
  int leser(0);               //C++: Initialisierung im Konstruktor
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();   //⇒ dequeue();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();     //⇒ enqueue();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```

```
lesen_ein();

... //lesen

lesen_aus();
```

Für Beispielabläufe implizite Locks sichtbar machen

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();  //⇒ dequeue();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();    //⇒ enqueue();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```




```
lesen_ein();
... //lesen
lesen_aus();
```

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();  //⇒ dequeue();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();    //⇒ enqueue();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```

Leser 2

Leser 1

Schreiber 3

```
lesen_ein();

... //lesen

lesen_aus();
```

Schreiber 2

Schreiber 1

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();  //⇒ dequeue();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();    //⇒ enqueue();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```

Leser 1

Schreiber 1

Leser 2

Schreiber 3

```
lesen_ein();
... //lesen
lesen_aus();
```

Schreiber 2

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();  //⇒ dequeue();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();    //⇒ enqueue();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```

Leser 2

Leser 1

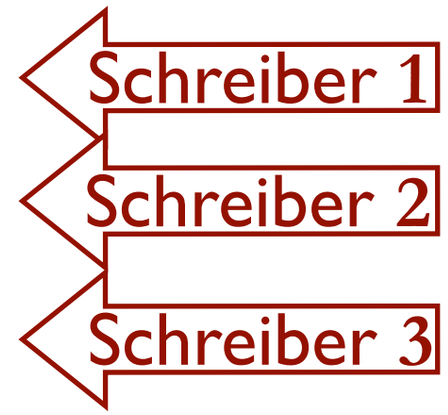
```
lesen_ein();
... //lesen
lesen_aus();
```

Schreiber 1

Schreiber 2

Schreiber 3

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();  //⇒ dequeue();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();    //⇒ enqueue();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```




```
lesen_ein();
... //lesen
lesen_aus();
```

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();  //⇒ dequeue();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```

Schreiber 1

Schreiber 2

```
lesen_ein();
... //lesen
lesen_aus();
```

Leser 2

Schreiber 3

Auch hier keine Fairness:
Leser haben Priorität

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();  //⇒ dequeue();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```

Schreiber 1

```
lesen_ein();
... //lesen
lesen_aus();
```

Leser 2

Schreiber 2

Schreiber 3

Auch hier keine Fairness:
Leser haben Priorität

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```

Schreiber 1

Leser 2

```
lesen_ein();
... //lesen
lesen_aus();
```

Leser 1

Schreiber 2

Schreiber 3

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```

Schreiber 1

Leser 2

```
lesen_ein();
... //lesen
lesen_aus();
```

Leser 1

Schreiber 2

Schreiber 3

```
monitor Leser-Schreiber {
  int leser(0);
  Condition schreibbar;

void lesen_ein() {
  lock(…);
  leser++;
  unlock(…);
}

void lesen_aus() {
  lock(…);
  leser--;
  if (leser == 0)
    schreibbar.signal();   //⇒ dequeue();
  unlock(…);
}

void schreiben() {
  lock(…);
  if (leser != 0)
    schreibbar.wait();     //⇒ enqueue();
  ... //schreiben
  schreibbar.signal();
  unlock(…);
}
```

Schreiber 1

Leser 2

```
lesen_ein();

... //lesen

lesen_aus();
```

Leser 1

Schreiber 2

Schreiber 3

Kleine Aufgabe

Kann man das Speisende-Philosophen-Problem lösen, indem man das Aufnehmen und das Niederlegen der beiden Stäbchen jeweils in einer Monitor-Operation (`essenswunsch()` bzw. `essen_fertig()`) kapselt, in der die betreffenden Stäbchen (Sticks) dem Philosophen i zugewiesen werden? Begründung.

(`stick[i]==-1` → Stäbchen liegt auf Tisch)

```
monitor Speisende_Philosophen {
 ...
 void essenswunsch(int i) {
   stick[i] = i;
   stick[(i+1) % 5] = i;
 }
 void essen_fertig(int i) {
   stick[i] = -1;
   stick[(i+1) % 5] = -1;
 }
 ...
}
```

Philosoph i:

```
for (;;) {
... denken ...
essenswunsch(i);
... essen ...
essen_fertig(i);
}
```

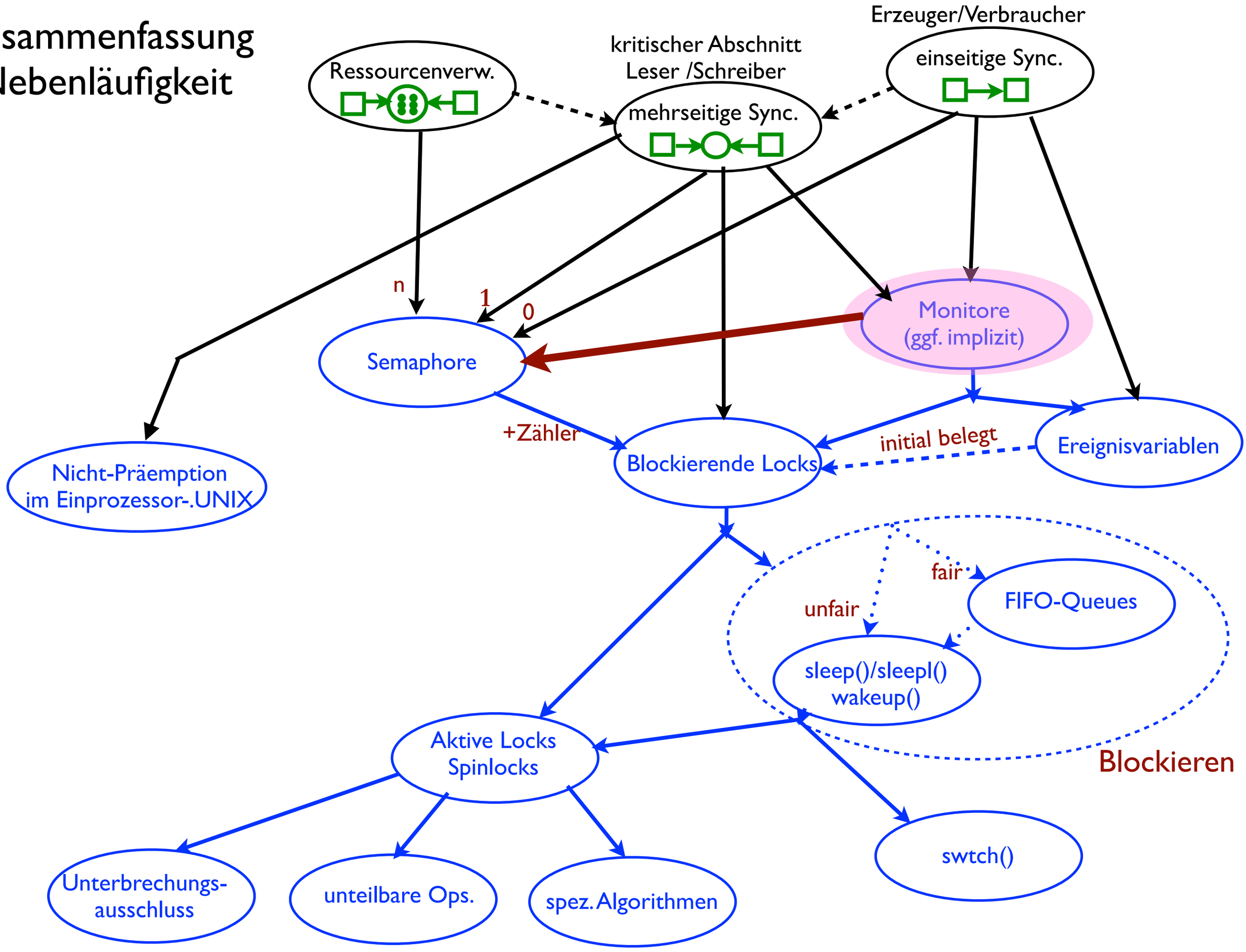
Fragen – Teil 1

- Was ist ein *Monitor* (als Synchronisationsverfahren)?
- Unter welchen Bedingungen wird ein Monitor betreten bzw. wieder verlassen?

Teil 2:
Umsetzung von Monitoren

19 Monitore, Ute Bormann

Zusammenfassung Nebenläufigkeit



Konzeptionelle Implementierung von Monitoren durch Semaphore

- Monitorsemantik muss irgendwie realisiert werden

  ⇒ vom Compiler bzw. „von Hand“

- Kann teilweise durch Semaphore ausgedrückt werden:
  - gegenseitiger Ausschluss aller Operationen (Funktionen/Methoden)

    ⇒ jede Operation mit P()/V() umschließen (mit 1 initialisiertes Semaphor)

  - einseitige Synchronisation durch wait()/signal()

    ⇒ pro Ereignisvariable je ein mit 0 initialisiertes Semaphor

Konzeptionelle Implementierung von Monitoren durch Semaphore

- Monitorsemantik muss irgendwie realisiert werden

  ⇒ vom Compiler bzw. „von Hand“

- Kann teilweise durch Semaphore ausgedrückt werden:
  - gegenseitiger Ausschluss aller Operationen (Funktionen/Methoden)

    ⇒ jede Operation mit P()/V() umschließen (mit 1 initialisiertes Semaphor)

  - einseitige Synchronisation durch wait()/signal()

    ⇒ pro Ereignisvariable je ein mit 0 initialisiertes Semaphor

- Semaphorsemantik allein reicht nicht aus:
  - signal() ohne wait() wird ignoriert ⇒ bei Semaphoren dazu zusätzliche Abfrage nötig

Konzeptionelle Implementierung von Monitoren durch Semaphore

- Monitorsemantik muss irgendwie realisiert werden

  ⇒ vom Compiler bzw. „von Hand“

- Kann teilweise durch Semaphore ausgedrückt werden:
  - gegenseitiger Ausschluss aller Operationen (Funktionen/Methoden)

    ⇒ jede Operation mit P()/V() umschließen (mit 1 initialisiertes Semaphor)

  - einseitige Synchronisation durch wait()/signal()

    ⇒ pro Ereignisvariable je ein mit 0 initialisiertes Semaphor

- Semaphorsemantik allein reicht nicht aus:
  - signal() ohne wait() wird ignoriert ⇒ bei Semaphoren dazu zusätzliche Abfrage nötig
  - signal() führt zum Blockieren, falls anderer darauf wartet (Fairness)
  - signal()-Warteschlange wird gegenüber Neueintritt in Monitor bevorzugt (Fairness)

  ⇒ zusätzliche Maßnahmen erforderlich

Allerdings: Fairness ohnehin problematisch

Konzeptionelle Implementierung von Monitoren durch Semaphore

- Monitorsemantik muss irgendwie realisiert werden

  ⇒ vom Compiler bzw. „von Hand“

- Kann teilweise durch Semaphore ausgedrückt werden:
  - gegenseitiger Ausschluss aller Operationen (Funktionen/Methoden)

    ⇒ jede Operation mit P()/V() umschließen (mit 1 initialisiertes Semaphor)

  - einseitige Synchronisation durch wait()/signal()

    ⇒ pro Ereignisvariable je ein mit 0 initialisiertes Semaphor

- Semaphorsemantik allein reicht nicht aus:
  - signal() ohne wait() wird ignoriert ⇒ bei Semaphoren dazu zusätzliche Abfrage nötig
  - ~~signal() führt zum Blockieren, falls anderer darauf wartet (Fairness)~~
  - ~~signal()-Warteschlange wird gegenüber Neueintritt in Monitor bevorzugt (Fairness)~~

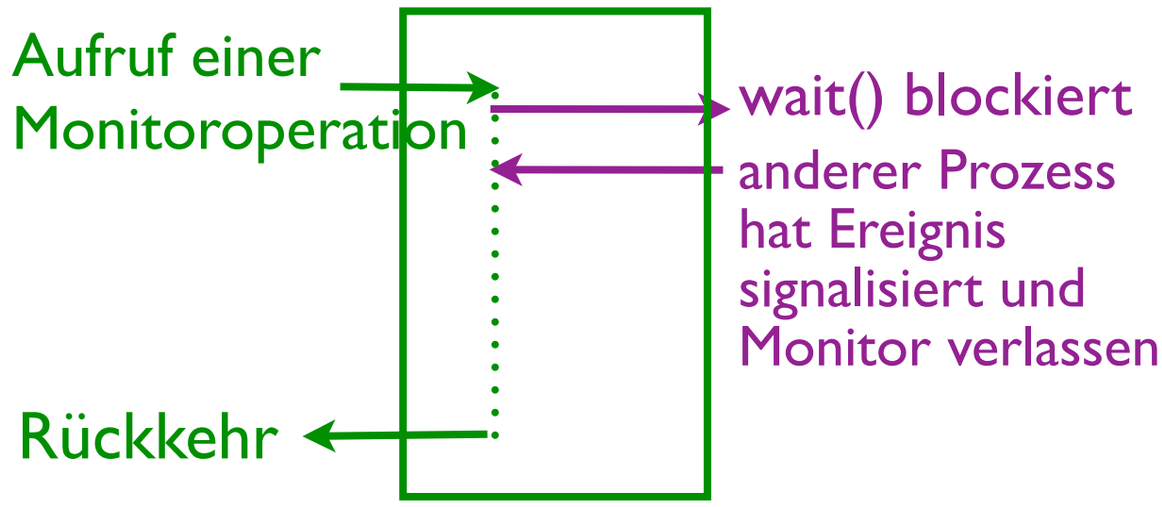
  ~~⇒ zusätzliche Maßnahmen erforderlich~~

⇒ i.d.R. nur unfaire Form von Monitoren umsetzen
(unter Nutzung von sleepI()-Semantik, um Lock im Wartezustand freizugeben)

Vereinfachung: Unfaire Monitore

- Gegenseitiger Ausschluss aller Operationen
  ⇒ bei wait() muss Monitor verlassen werden
  (⇒ sleepl())
- Zu einem Zeitpunkt kann ein Prozess nur auf ein Ereignis warten
- Signale werden nicht gespeichert
  ⇒ wenn keiner wartet, geht es verloren
- ~~Nach Signal wird Monitor an darauf wartenden Prozess übergeben (falls einer wartet)~~
  ~~⇒ besonders fair~~
- ~~Wenn der wieder raus ist, kommt signalisierender Prozess wieder rein~~
  ~~⇒ besonders fair~~
- ~~Faire Warteschlangen an den „Eingängen“~~

„Eingänge“ und „Ausgänge“ eines Monitors



Unfaire Monitore in der Unix-Multithreading-Umgebung

- Keine Monitore vorgesehen
- Verschiedene Alternativen möglich, um unfaire Monitore zu simulieren
  - Modellierung mit Semaphoren (s. konzeptionelle Implementierung)
  - Modellierung mit Mutex_lock()/Mutex_unlock() und Ereignisvariablen

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

```
class Mutex {
  mutex_t mutex;
public:
  Mutex(...)  {...}
  void lock() {...}
  void unlock() {...}
  ...
}
```

- Mit wait() kann man auf Ereignis warten ⇒ sleep()

Unfaire Monitore in der Unix-Multithreading-Umgebung

- Keine Monitore vorgesehen
- Verschiedene Alternativen möglich, um unfaire Monitore zu simulieren
  - Modellierung mit Semaphoren (s. konzeptionelle Implementierung)
  - Modellierung mit Mutex_lock()/Mutex_unlock() und Ereignisvariablen

```
class Condition {
  cond_t cond;
public:
  Condition(...) {...}
  void wait(Mutex &m) {...}
  void signal() {...}
  ...
 }
```

```
class Mutex {
  mutex_t mutex;
  friend class Condition;
public:
  Mutex(...)  {...}
  void lock() {...}
  void unlock() {...}
  ...
}
```

- Mit wait() kann man auf Ereignis warten ⇒ sleep()
- Kritischer Abschnitt muss vorher verlassen werden
  ⇒ auch hier sleepl()-Semantik erforderlich
  ⇒ Parameter des wait(): Mutex

Achtung:
Durch einfache Abbildung auf sleep()/sleepl()/wakeup() keine originale Monitor-Semantik:

- Keine (fairen) FIFO-Warteschlangen
- Bei signal() keine Prozessorabgabe

Originalsyntax in der Pthread-Umgebung:

```
pthread_cond_t c;                pthread_mutex_t s;
pthread_cond_wait(&c, &s);       pthread_mutex_lock(&s);
pthread_cond_signal(&c);         pthread_mutex_unlock(&s);
```

Wdh.: Monitorimplementierung für Integer-Mailbox (ohne Überlaufschutz)

```
monitor Mailbox {          //kein C++
  int wert;
  bool voll;

  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ... voll = false;

void Mailbox::rein(int i){
  wert = i;              //ohne Überlaufschutz
  voll = true;
  gefuellt.signal();

}

int Mailbox::raus() {


  if (!voll)
    gefuellt.wait();

  voll = false;

  return(wert);
}
```

Wdh.: Monitorimplementierung für Integer-Mailbox (ohne Überlaufschutz)

```
class Mailbox {      //simuliert mit C++-Klassen
  int wert;
  bool voll;
  Mutex schutz;
  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ... voll = false;

void Mailbox::rein(int i){
  schutz.lock();
  wert = i;            //ohne Überlaufschutz
  voll = true;
  gefuellt.signal();
  schutz.unlock();
}

int Mailbox::raus() {
  schutz.lock();

  while (!voll)
    gefuellt.wait(schutz); //→ sleepl()...

  voll = false;
  schutz.unlock();
  return(wert);
}
```

```
monitor Mailbox {
  int wert;
  bool voll;

  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ...

void Mailbox::rein(int i){

  wert = i;
  voll = true;
  gefuellt.signal();

}

int Mailbox::raus() {

  if (!voll)
    gefuellt.wait();

  voll = false;

  return(wert);
}
```

Wdh.: Monitorimplementierung für Integer-Mailbox (ohne Überlaufschutz)

```
class Mailbox {      //simuliert mit C++-Klassen
  int wert;
  bool voll;
  Mutex schutz;
  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ... voll = false;

void Mailbox::rein(int i){
  schutz.lock();
  wert = i;            //ohne Überlaufschutz
  voll = true;
  gefuellt.signal();
  schutz.unlock();
}

int Mailbox::raus() {
  schutz.lock();

  while (!voll)
    gefuellt.wait(schutz); //→ sleepl()...

  voll = false;
  schutz.unlock();
  return(wert);
}
```

```
monitor Mailbox {
  int wert;
  bool voll;

  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ...

void Mailbox::rein(int i){

  wert = i;
  voll = true;
  gefuellt.signal();

}

int Mailbox::raus() {


  if (!voll)
    gefuellt.wait();

  voll = false;

  return(wert);
}
```

Wdh.: Monitorimplementierung für Integer-Mailbox (ohne Überlaufschutz)

```
class Mailbox {      //simuliert mit C++-Klassen
  int wert;
  bool voll;
  Mutex schutz;
  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ... voll = false;

void Mailbox::rein(int i){
  schutz.lock()
  wert = i;           //ohne Überlaufschutz
  voll = true;
  gefuellt.signal();
  schutz.unlock();
}

int Mailbox::raus() {
  schutz.lock();
  int i;
  while (!voll)
    gefuellt.wait(schutz); //→ sleepl()...
  i = wert;
  voll = false;
  schutz.unlock();
  return(i);
}
```

```
monitor Mailbox {
  int wert;
  bool voll;

  Condition gefuellt;
public:
  Mailbox();
  void rein(int i);
  int raus();
};
...  //Konstruktor: ...

void Mailbox::rein(int i){

  wert = i;
  voll = true;
  gefuellt.signal();

}

int Mailbox::raus() {


  if (!voll)
    gefuellt.wait();

  voll = false;

  return(wert);
}
```

Threads in Java (Beispiel)

```
class MyClass implements Runnable {
  public MyClass(int val1, float val2) {
    .. Konstruktor ..
  }
  public void run() {
    .. do something
  }
}

public static void main(String[] args) {
  MyClass obj = new MyClass(12, 25.35);
  Thread thr = new Thread(obj);
  thr.start();
}
```

Gegenseitiger Ausschluß in Java

- Sprachmittel für gegenseitigen Ausschluß (Schlüsselwort **synchronized**).
- Pro Krit. Abschnitt ein Objekt als Locking-Variable

```
IrgendeineKlasse myObj = new IrgendeineKlasse();
synchronized (myObj) {
  .. kritischer Abschnitt ..
}
```

ist äquivalent zu

```
Mutex myObj;
myObj.lock();
.. kritischer Abschnitt ..
myObj.unlock();
```

* Schutz vor nebenläufigen Zugriffen, die ebenfalls synchronized erfolgen
* nicht-synchronized Zugriffe können weiterhin nebenläufig erfolgen.
* Java-Terminologie: Thread wird „Eigentümer des Objektes“

Signale/Ereignisvariablen in Java

- Jedes Java-Objekt kann auf Signale warten (wait)
- Jedes Java-Objekt kann wartende Threads aufwecken (notify)
- Thread muß dafür Eigentümer des Objektes werden

```
MyClass myObj;
synchronized(myObj) {
  ... erzeugen() ...
  myObj.notify();
}
```

```
MyClass myObj;
synchronized(myObj) {
  myObj.wait();
  ... verbrauchen() ...
  }
```

(Dem Beispiel fehlt noch das Abfangen der Exceptions in wait())

- **wait()** Eigentümerschaft auf Objekt für Dauer des Wartens aufgeben. Nach Aufwecken wiedererlangen ⇒ analog zu sleepl()
- **notify()** weckt irgendeinen Thread auf (keine Fairneß)
- **notifyAll()** weckt alle Threads auf ⇒ analog zu wakeup()

Monitore in Java

Schlüsselwort `synchronized` vor Methoden (statt Programmabschnitten).
⇒ Schutz der Instanz der Klasse bei allen synchronized Methoden-Aufrufen vor nebenläufigem Zugriff

```
public synchronized void einzahlen(int betrag) {
   .. ändere Konto ..
}
```

Schutz erfolgt nur gegenüber

- anderen Methoden der Klasse, die ebenfalls synchronized sind
- kritischen Abschnitten, die sich über das Objekt synchronisieren

Als Condition-Variable kann das Objekt selbst (this) genutzt werden, solange nur eine solche Variable benötigt wird.

Beispiel: Monitorimplementierung für Integer-Mailbox in Java (ohne Überlaufschutz)

```
public class Mailbox {
    private int wert;
    private boolean voll;
    public Mailbox() {voll = false;}

    //-----------------------------------------------
    public synchronized boolean rein(int i) {

        wert = i;
        voll = true;
        this.notify();
        return true;
    }

    //-----------------------------------------------
    public synchronized int raus() {
        if (!voll)
            this.wait();
        voll = false;
        return wert;
    }
}
```

Beispiel: Monitorimplementierung für Integer-Mailbox in Java (ohne Überlaufschutz)

```
public class Mailbox {
    private int wert;
    private boolean voll;
    public Mailbox() {voll = false;}

    //-----------------------------------------------
    public synchronized boolean rein(int i) {

        wert = i;
        voll = true;
        this.notify();
        return true;
    }

    //-----------------------------------------------
    public synchronized int raus() {
        if (!voll)
            this.wait();
        voll = false;
        return wert;
    }
}
```

```
monitor Mailbox {  //kein C++
  int wert;
  bool voll;
  Condition gefuellt;  //…
  // Konstruktor …
```

```
void Mailbox::rein(int i) {

  wert = i;
  voll = true;
  gefuellt.signal();
}
```

```
int Mailbox::raus() {
  if (!voll)
    gefuellt.wait();
  voll = false;
  return(wert);
}
```

Beispiel: Monitorimplementierung für Integer-Mailbox in Java (ohne Überlaufschutz)

Umgesetzt mit C++-Klassen

```
public class Mailbox {
    private int wert;
    private boolean voll;                                    ...
    public Mailbox() {voll = false;}

    //-----------------------------------------------
    public synchronized boolean rein(int i) {

        wert = i;
        voll = true;
        this.notify();
        return true;
    }

    //-----------------------------------------------
    public synchronized int raus() {
        if (!voll)
            this.wait();
        voll = false;
        return wert;
    }
}
```

```
void Mailbox::rein(int i) {
  schutz.lock();
  wert = i;
  voll = true;
  gefuellt.signal();
  schutz.unlock();
}
```

```
int Mailbox::raus() {
  schutz.lock();
  int i;
  while (!voll)
    gefuellt.wait(schutz);
  i = wert;
  voll = false;
  schutz.unlock();
  return(i);
}
```

Nebenläufigkeitsschutz in C++

- Mittlerweile auch in C++-Bibliothek Ergänzungen für Nebenläufigkeitsschutz vorgenommen:
  - Locks (std::mutex)
  - Ereignisvariablen (std::condition_variable)
- Damit auch Monitore umsetzbar
- Nutzt intern Pthread-Funktionalität

Integer-Mailbox in C++

Bisher:

```
class Mailbox {
  int wert;
  bool voll;
  Mutex schutz;
  Condition gefuellt;

public:
  Mailbox();
  void rein(int i);
  int raus();
}

 ...  //Konstruktor: … voll = false; …
```

```
void Mailbox::rein(int i) {
  schutz.lock();
  wert = i;       //ohne Überlaufschutz
  voll = true;
  gefuellt.signal();
  schutz.unlock();
}

int Mailbox::raus() {
  schutz.lock();
  int i;
  while (!voll)
    gefuellt.wait(schutz); //→ sleepl()...
  i = wert;
  voll = false;
  schutz.unlock();
  return(i);
}
```

Nun:

```
class Mailbox {
 private:
   int wert;
   bool voll;
   std::mutex schutz;
   std::condition_variable gefuellt;

 public:
   Mailbox() : voll(false) {}
   void rein(int i);
   int raus();
}
```

```
void Mailbox::rein(int i) {
   std::unique_lock<std::mutex> lock(schutz);
   wert = i;                  // ohne Überlaufschutz
   voll = true;
   gefuellt.notify_one();
}

int Mailbox::raus() {
   std::unique_lock<std::mutex> lock(schutz);
   while (!voll)
     gefuellt.wait(lock);
   voll = false;
   return wert;
}
```

Fragen – Teil 2

- Mit Hilfe welcher Synchronisationsmechanismen können Monitore modelliert werden?

Zusammenfassung

- Monitorkonzept
  - Impliziter gegenseitiger Ausschluss
  - Ereignisvariablen
- Einsatz von Monitoren
- Realisierung von Monitoren
- Nebenläufigkeitsschutz in Java und C++

Monitore – Fragen

1. Was ist ein *Monitor* (als Synchronisationsverfahren)?
2. Unter welchen Bedingungen wird ein Monitor betreten bzw. wieder verlassen?
3. Mit Hilfe welcher Synchronisationsmechanismen können Monitore modelliert werden?