Work in progress

Semaphore

Ute Bormann, Tel2

2023-10-13

Inhalt

1. Eigenschaften von Semaphoren
2. Implementierung von Semaphoren
3. Beurteilung von Semaphoren

Teil 1:
Eigenschaften von Semaphoren

Semaphore (Dijkstra, 1965)

- „Verfeinerung“ des Konzepts des blockierenden Wartens
- Je nach Variante zwei Erweiterungen:
  - Klassische Semaphore sind fair
    - Im Wartefall Einreihen in eine FIFO-Queue

      ⇒ Abarbeitung in Ankunftsreihenfolge

Semaphore (Dijkstra, 1965)

- „Verfeinerung“ des Konzepts des blockierenden Wartens
- Je nach Variante zwei Erweiterungen:
  - Klassische Semaphore sind fair
    - Im Wartefall Einreihen in eine FIFO-Queue

      ⇒ Abarbeitung in Ankunftsreihenfolge
  - Counting Semaphore
    - Semaphor enthält Zähler (vorinitialisiert mit n)
    - Blockiert, wenn bereits n Prozesse im „krit. Abschnitt“ und n+1. will rein

      ⇒ flexiblere Nutzung möglich
      - n=1: Schutz eines kritischen Abschnitts
      - n=0: Einseitige Synchronisation
      - n>1: Verwaltung von n gleichartigen Betriebsmitteln
    - Zähler wird bei Eintritt dekrementiert, bei Austritt inkrementiert
    - Semaphor blockiert bei Zählerstand = 0

- Realisierung u.U. als abstrakter Datentyp (Sema) bestehend aus:
  - Variable mit ganzzahligen Werten ≥ 0
    ⇒ aktueller Wert gibt an, wieviele noch rein dürfen
  - Warteschlange für blockierte Prozesse
    ⇒ stellt enqueue/dequeue-Operationen bereit
- Zwei Operationen;
  - P() //passeeren, auch down() genannt, ≙ lock()
    ⇒ Zähler dekrementieren, ggf. blockieren
  - V() //vrijgeven, auch up() genannt, ≙ unlock()
    ⇒ Zähler inkrementieren, ggf. aufwecken

Kritischer Abschnitt
(mehrseitige Synchronisation)

⇒ Lösung analog zu Lock-Verfahren

```
Sema s(1);

...
```

Prozess/Thread A	Prozess/Thread B
`...`	`...`
`s.P();`	`s.P();`
`// kritischer Abschnitt`	`// kritischer Abschnitt`
`s.V();`	`s.V();`

Leser-/Schreiber-Problem

⇒ Lösung analog zu Lock-Verfahren

```
Sema rw(1);

reader() {
   ...
   rw.P();
   // Lesen
   rw.V();
   ...
}

writer() {
   ...
   rw.P();
   // Schreiben
   rw.V();
   ...
}
```

Leser und Schreiber alternativ

Leser-/Schreiber-Problem

⇒ Lösung analog zu Lock-Verfahren

```
Sema rw(1);
int count = 0;
```

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

  // Lesen

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

  ...
}
```

```
writer() {
  ...
  rw.P();
  // Schreiben
  rw.V();
  ...
}
```

Leser und Schreiber alternativ

Mehr als einen Leser zulassen

Leser-/Schreiber-Problem

⇒ Lösung analog zu Lock-Verfahren

```
Sema rw(1);
int count = 0;
Sema s(1);

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

```
writer() {
   ...
   rw.P();
   // Schreiben
   rw.V();
   ...
}
```

Leser und Schreiber alternativ

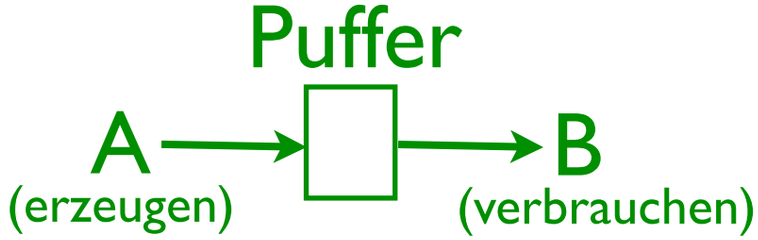
Mehr als einen Leser zulassen

Zähler schützen

(Achtung: Leser werden bei dieser Lösung bevorzugt)

Erzeuger/Verbraucher-Problem
(Einseitige Synchronisation)

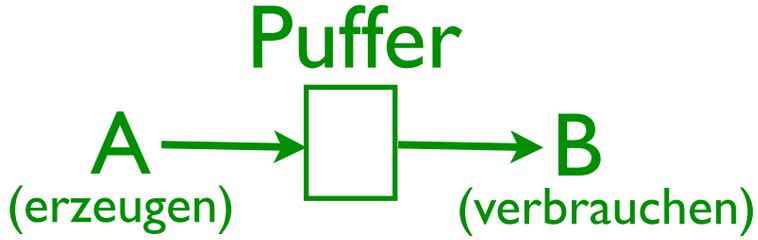
⇒ Lösung im Prinzip analog zu Ereignisvariablen




```
Prozess/Thread A   Condition c;   Prozess/Thread B
...                                ...
erzeugen();                        c.wait();
c.signal(); //nicht-verpuffend     verbrauchen();
...                                ...
```

Erzeuger/Verbraucher-Problem
(Einseitige Synchronisation)

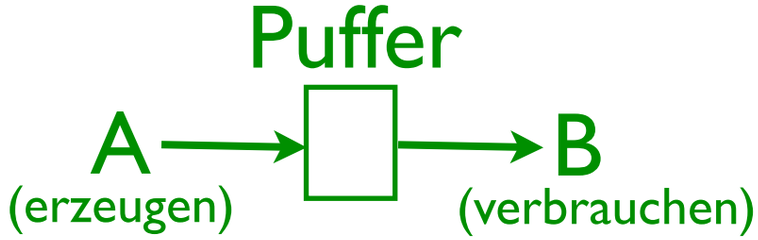
⇒ Lösung im Prinzip analog zu Ereignisvariablen




```
Prozess/Thread A   Sema s(0);   Prozess/Thread B
...                             ...
erzeugen();                     s.P();
s.V();                          verbrauchen();
...                             ...
```

Erzeuger/Verbraucher-Problem
(Einseitige Synchronisation)

⇒ Lösung im Prinzip analog zu Ereignisvariablen




```
Prozess/Thread A   Sema s(0);   Prozess/Thread B

...                              ...
erzeugen();                      s.P();
s.V();                           verbrauchen();
...                              ...
```

Auch mehrfache Erzeugung möglich, Semaphor verbucht Anzahl der erzeugten und noch nicht verbrauchten Elemente

- Semaphore flexibler als Ereignisvariablen:
  - Jedes V() erhöht den Zähler
  - Jedes P() erniedrigt den Zähler

    ⇒ Zähler simuliert Füllstand des Puffers

  - Zähler = 0 ⇒ in P() warten

    ⇒ nicht mehr verbrauchen als erzeugt worden ist

- Jedoch keine Obergrenze des Zählers

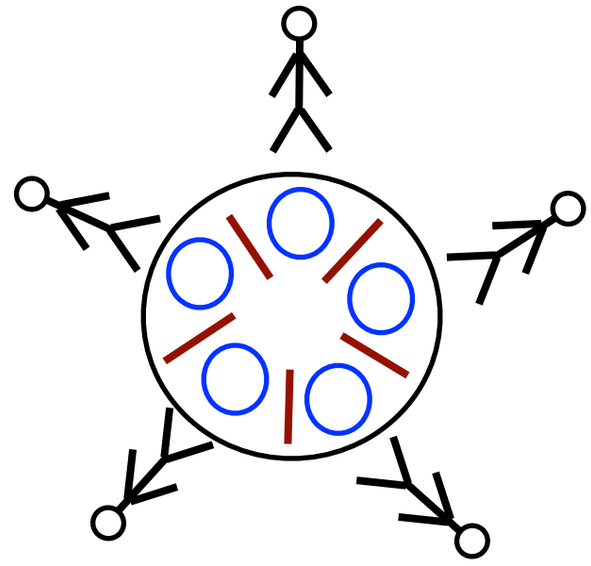
  ⇒ beliebig viele V() ausführbar

- Also beschränkte Puffergröße nicht direkt abfangbar

  ⇒ Lösung: Zweites Semaphor zählt verbliebenen Platz

Speisende Philosophen

- Wdh.: Fünf Philosophen sitzen am Tisch mit je einer Schüssel Reis und je einem Stäbchen dazwischen



- Problem:
  Man benötigt zwei Stäbchen zum Essen, aber es gibt nur insgesamt 5

  ⇒ nur zwei Philosophen können gleichzeitig essen

- Jedes Stäbchen ist kritischer Abschnitt

  ⇒ z.B. geschützt durch je ein Semaphor

- Gefahr der Verklemmung, wenn alle gleichzeitig linkes Stäbchen nehmen

  ⇒ Mögliche Lösung: Einen Philosophen temporär vom Wettbewerb ausschließen (z.B. nur 4 Stühle)

Mögliche Semaphor-Lösung

(1. Schritt)

```
struct Sema {
       ...
       Sema (int count=1)
          //Default-Initialisierung
    }
```

```
Sema stick[5];   //Array von 5 Semaphoren, mit 1 initialisiert

philosoph(int i) {
  while (1) {
     ... denken ...

     stick[i].P();
     stick[(i+1)%5].P();
     ... speisen ...
     stick[i].V();
     stick[(i+1)%5].V();

  };
}
```

Mögliche Semaphor-Lösung

(1. Schritt)

```
struct Sema {
       ...
       Sema (int count=1)
          //Default-Initialisierung
    }
```

```
Sema stick[5];   //Array von 5 Semaphoren, mit 1 initialisiert

philosoph(int i) {
  while (1) {
     ... denken ...

     stick[i].P();
     stick[(i+1)%5].P();
     ... speisen ...
     stick[i].V();
     stick[(i+1)%5].V();

  };
}
```

⇒ Gefahr der Verklemmung

Mögliche Semaphor-Lösung

(2. Schritt)

```
struct Sema {
        ...
        Sema (int count=1)
            //Default-Initialisierung
    }
```

```
Sema stick[5];    //Array von 5 Semaphoren, mit 1 initialisiert
Sema chair(4);    //mit 4 initialisiertes Semaphor

philosoph(int i) {
  while (1) {
     ... denken ...
     chair.P();
     stick[i].P();
     stick[(i+1)%5].P();
     ... speisen ...
     stick[i].V();
     stick[(i+1)%5].V();
     chair.V();
  };
}
```

Kleine Aufgabe

Kann man mit der folgenden Semaphornutzung einen kritischen Abschnitt schützen?

- Falls ja, ist diese Lösung zufriedenstellend?
- Falls nein, warum nicht?

```
...
sema s1(1);
sema s2(0);
...
```

Thread A

```
...
s1.P();
... kritischer Abschnitt...
s2.V();
...
```

Thread B

```
...
s2.P();
... kritischer Abschnitt...
s1.V();
...
```

Fragen – Teil 1

- Welche zusätzlichen Eigenschaften zeichnen *Semaphore* gegenüber blockierenden Locks aus?
- Wie wird eine einseitige bzw. eine mehrseitige Synchronisation durch Semaphore ausgedrückt?
- Wie können Semaphore zur Lösung des Problems der *speisenden Philosophen* eingesetzt werden?

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
  - Mo 30.1. – Fr 3.2.2023: Projektwahl (über Web-Formular)

Teil 2:
Implementierung von Semaphoren

Konzeptionelle Implementierung von fairen Semaphoren

```
class Sema {
  int counter;
  Queue q;
public:
  Sema (int count);
  void P();
  void V();
}
```

```
class Queue {
  ...
public:
  Queue();
  void enqueue();
  void dequeue();
  bool empty();
}
```

```
Sema::Sema(int count) {
  counter = count;
}
```

```
void Sema::P() {


  if (counter==0)
    q.enqueue();
  else
    --counter;


}
```

```
void Sema::V() {


  if (!q.empty())
    q.dequeue();
  else
    ++counter;


}
```

Konzeptionelle Implementierung von fairen Semaphoren

Zum Vergleich:

```
void Block::block_lock() {

  while (test_and_set(key))
    sleep(this);

}
```

```
void Block::block_unlock() {

    key = false;
    wakeup(this);

}
```

```
void Sema::P() {


  if (counter==0)
    q.enqueue();
  else
    --counter;


}
```

```
void Sema::V() {


  if (!q.empty())
    q.dequeue();
  else
    ++counter;


}
```

Konzeptionelle Implementierung von fairen Semaphoren

Zum Vergleich:

```
void Block::block_lock() {

  while (test_and_set(key))
    sleep(this);

}
```

```
void Block::block_unlock() {

    key = false;
    wakeup(this);

}
```

```
void Sema::P() {

  if (counter==0)
    q.enqueue();
  else
    --counter;

}
```

```
void Sema::V() {

  if (!q.empty())
    q.dequeue();
  else
    ++counter;

}
```

Konzeptionelle Implementierung von fairen Semaphoren

Zum Vergleich:

```
void Block::block_lock() {

  while (test_and_set(key))
    sleep(this);

}
```

```
void Block::block_unlock() {

    key = false;
    wakeup(this);

}
```

```
void Sema::P() {


  if (counter==0)
    q.enqueue();
  else
    --counter;


}
```

```
void Sema::V() {


  if (!q.empty())
    q.dequeue();
  else
    ++counter;


}
```

Konzeptionelle Implementierung von fairen Semaphoren

Zum Vergleich:

```
void Block::block_lock() {

  while (test_and_set(key))
    sleep(this);

}
```

```
void Block::block_unlock() {

    key = false;
    wakeup(this);

}
```

```
void Sema::P() {

  if (counter==0)
    q.enqueue();
  else
    --counter;

}
```

```
void Sema::V() {

  if (!q.empty())
    q.dequeue();
  else
    ++counter;

}
```

Konzeptionelle Implementierung von fairen Semaphoren

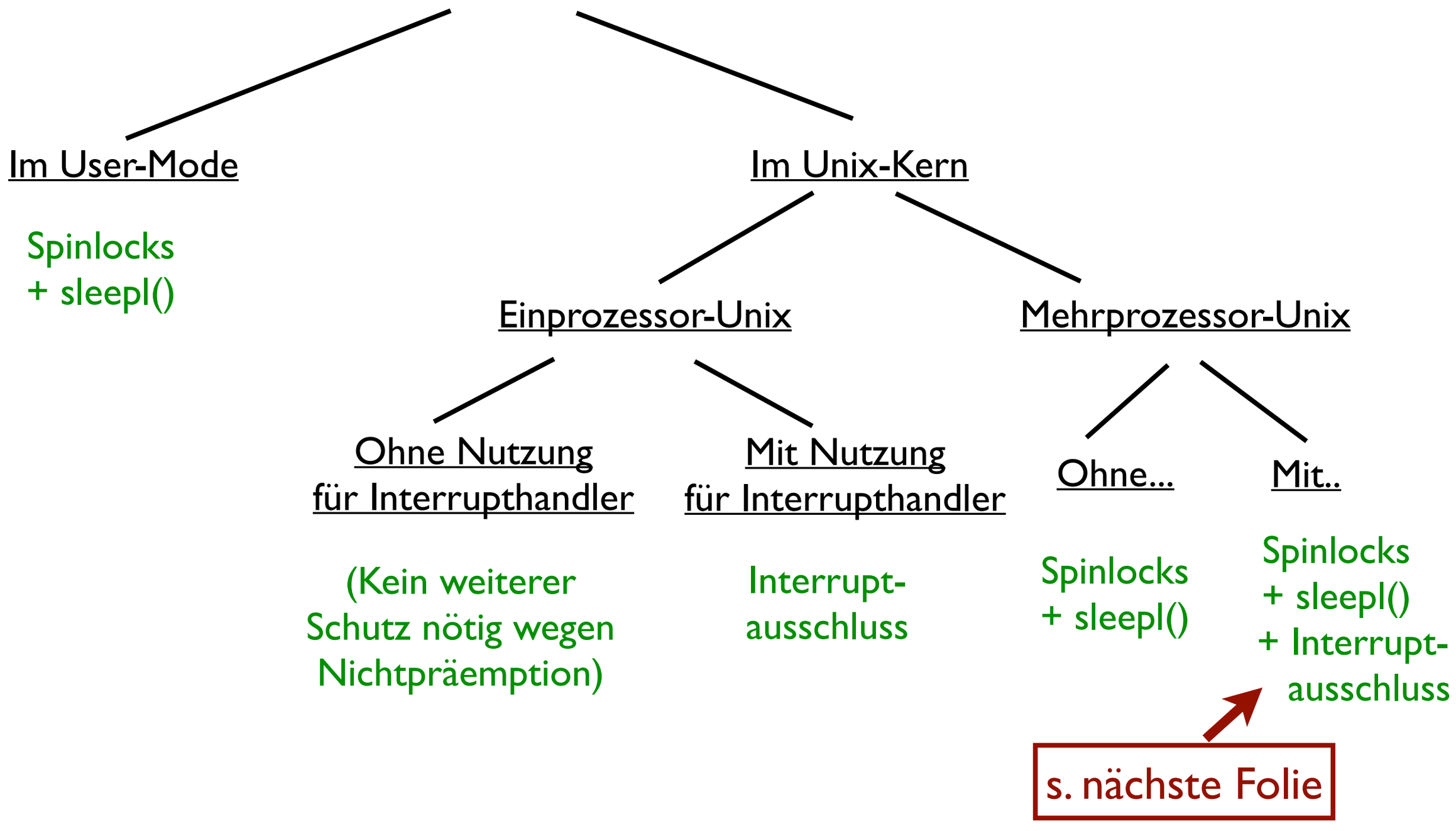
```
class Sema {
  int counter;
  Queue q;
public:
  Sema (int count);
  void P();
  void V();
}

Sema::Sema(int count) {
  counter = count;
}

void Sema::P() {


  if (counter==0)
    q.enqueue();
  else
    --counter;


}
```

```
class Queue {
  ...
public:
  Queue();
  void enqueue();
  void dequeue();
  bool empty();
}
```

```
void Sema::V() {


  if (!q.empty())
    q.dequeue();
  else
    ++counter;


}
```

⇒ Potentielle Nebenläufigkeitsprobleme:
Lost-Wakeup, counter-Verwaltung, ...

Realisierung von Semaphoren: Nebenläufigkeitsprobleme vermeiden



gleiche Varianten wie bei block-lock

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();
}
```

Konzeptionelle Implementierung von fairen Semaphoren

```
class Sema {
  int counter;
  Queue q;   Spin s;
public:
  Sema (int count);
  void P();
  void V();
}

Sema::Sema(int count) {
  counter = count;
}

void Sema::P() {
  disable_interrupts();
  s.spin_lock();
  if (counter==0)
    q.enqueue();
  else
    --counter;
  s.spin_unlock();
  enable_interrupts();
}
```

```
class Queue {
  ...
public:
  Queue();
  void enqueue();  // ⇒ sleepl()
  void dequeue();
  bool empty();
}
```

```
void Sema::V() {
  disable_interrupts();
  s.spin_lock();
  if (!q.empty())
    q.dequeue();
  else
    ++counter;
  s.spin_unlock();
  enable_interrupts();
}
```

Fragen – Teil 2

- In welches Problem wird eine allzu „einfache“ Semaphor-Implementierung laufen?

Teil 3:
Beurteilung von Semaphoren

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

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

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

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

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

```
void *drucken(char *param) {
  for (;;) {
    pthread_mutex_lock(said_mutex);
    if (said < 10) {
      said++;
      pthread_mutex_unlock(said_mutex);
      cout << param << endl;
    } else {
    → pthread_mutex_unlock(said_mutex);
      break;
    }
  }
}
```

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

```
void *drucken(char *param) {
  for (;;) {
    s.P();
    if (said < 10) {
      said++;
      s.V();
      cout << param << endl;
    } else {
      s.V();
      break;
    }
  }
}
```

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

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

```
philosoph(int i) {
  while (1) {
    ... denken ...
    stick[i].P();
    stick[(i+1)%5].P();
    ... speisen ...
    stick[i].V();
    stick[(i+1)%5].V();
  };
}
```

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

b) Fairness von Semaphoren nicht immer adäquat

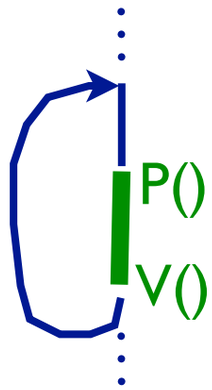
- Gefahr von Konvois

Ad b) Potentielles Problem mit fairen Semaphoren

- Einreihen in Warteschlange erzeugt feste Reihenfolge der Abarbeitung

  ⇒ kann zu Konvois führen

- Beispielprogramm:

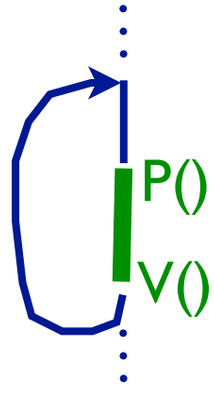


Ad b) Potentielles Problem mit fairen Semaphoren

- Einreihen in Warteschlange erzeugt feste Reihenfolge der Abarbeitung

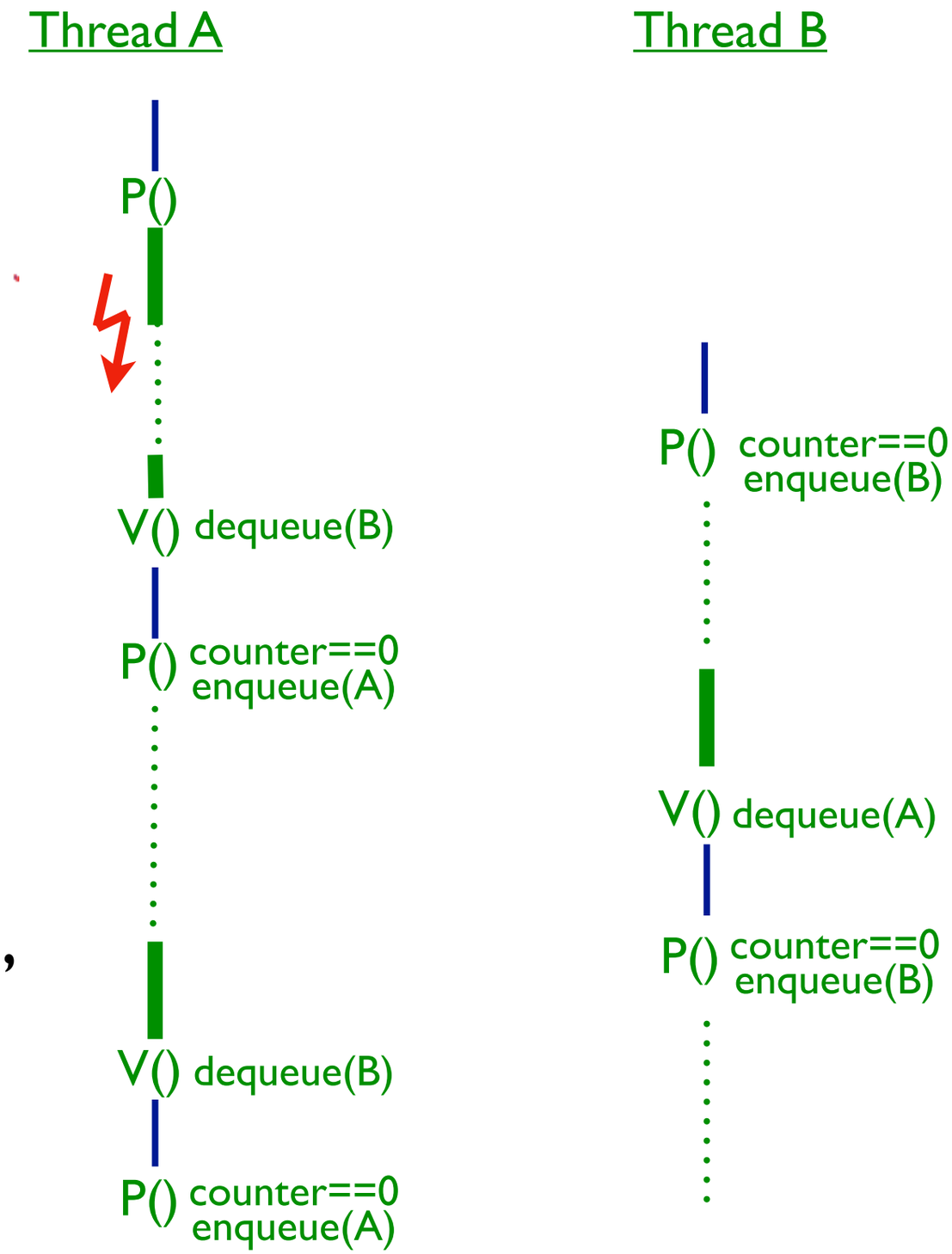
  ⇒ kann zu Konvois führen

- Beispielprogramm:



- Daraus u.U. resultierender Beispielablauf:

- U.U. unnötig viele Threadwechsel, auch wenn Zeitscheibe noch nicht aufgebraucht

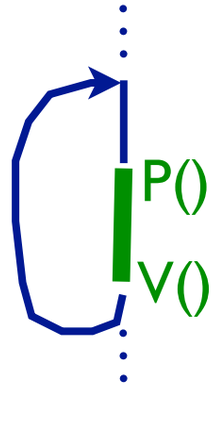


Ad b) Potentielles Problem mit fairen Semaphoren

- Einreihen in Warteschlange erzeugt feste Reihenfolge der Abarbeitung

  ⇒ kann zu Konvois führen

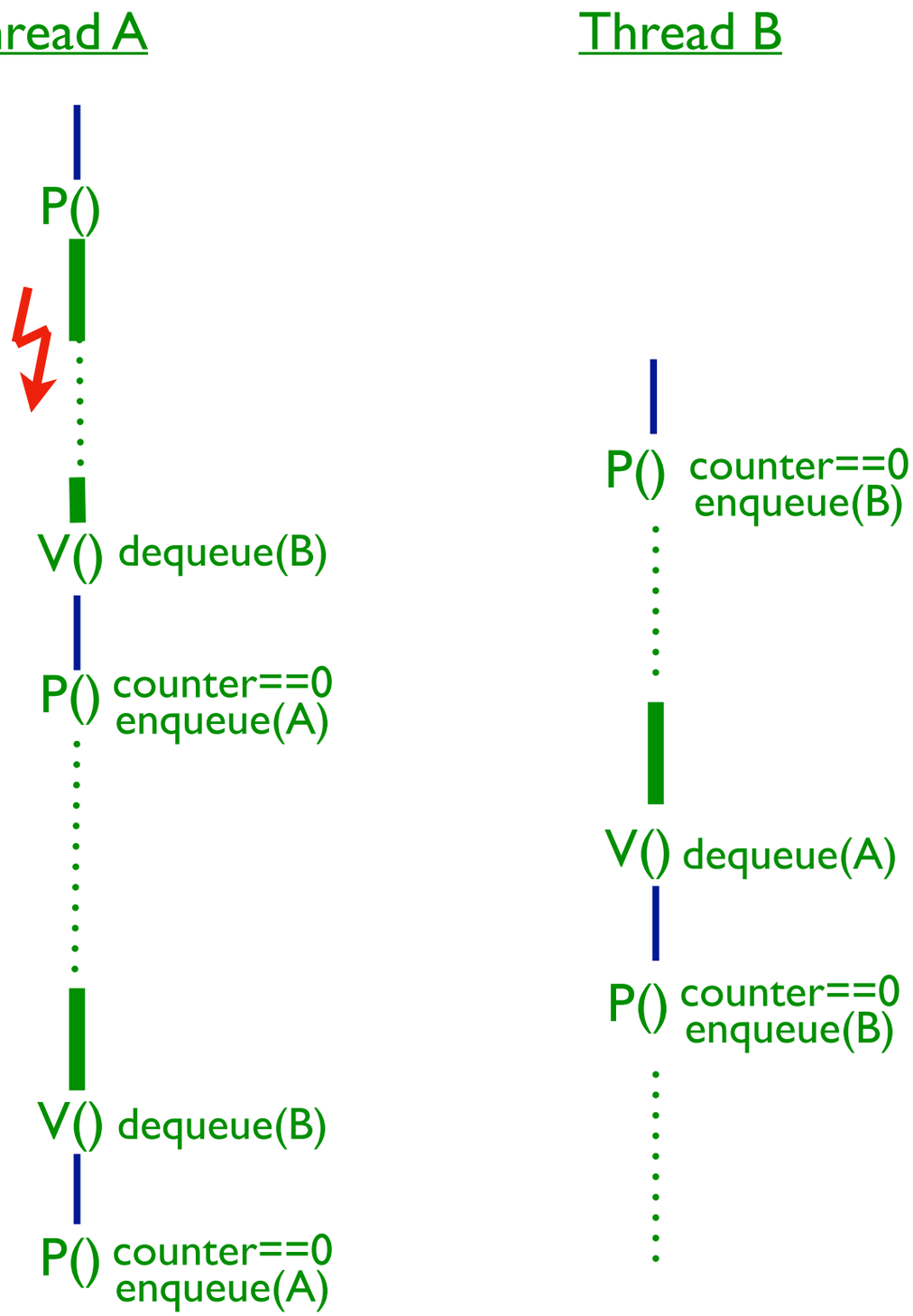
- Beispielprogramm:



- Daraus u.U. resultierender Beispielablauf:

- U.U. unnötig viele Threadwechsel, auch wenn Zeitscheibe noch nicht aufgebraucht

  ⇒ unfaire Semaphore in Unix



Realisierung von Semaphoren im Unix-Kern

- Auf der Grundlage von sleep()/wakeup()

  ⇒ unfair, aber Fairness u.U. ohnehin hinderlich

- Abstrakte Implementierung:

```
class Sema {
...                    //keine Queue vorsehen
}

void Sema::P() {                  void Sema::V() {
  while (counter==0)                ++counter;
    sleep(this);                    wakeup(this);
  --counter;                      }
}
```

Realisierung von Semaphoren im Unix-Kern

Zum Vergleich: Faire Semaphore

```
void Sema::P() {
  if (counter==0)
    q.enqueue();
  else
    --counter;
}
```

```
void Sema::V() {
  if (!q.empty())
    q.dequeue();
  else
    ++counter;
}
```

```
void Sema::P() {
  while (counter==0)
    sleep(this);
  --counter;
}
```

```
void Sema::V() {
  ++counter;
  wakeup(this);
}
```

Realisierung von Semaphoren im Unix-Kern

- Allerdings Problem der „donnernden Herden“: Alle auf diesem Semaphor wartenden Prozesse werden aufgeweckt

  ⇒ nur einer kommt sofort durch, Rest (ggf.) wieder schlafenlegen

  ⇒ also erneute Abfrage von counter nötig

```
void Sema::P() {
  while (counter==0)
    sleep(this);
  --counter;
}
```

```
void Sema::V() {
  ++counter;
  wakeup(this);
}
```

Realisierung von Semaphoren im Unix-Kern

Zum Vergleich: Implementierung des block_lock():

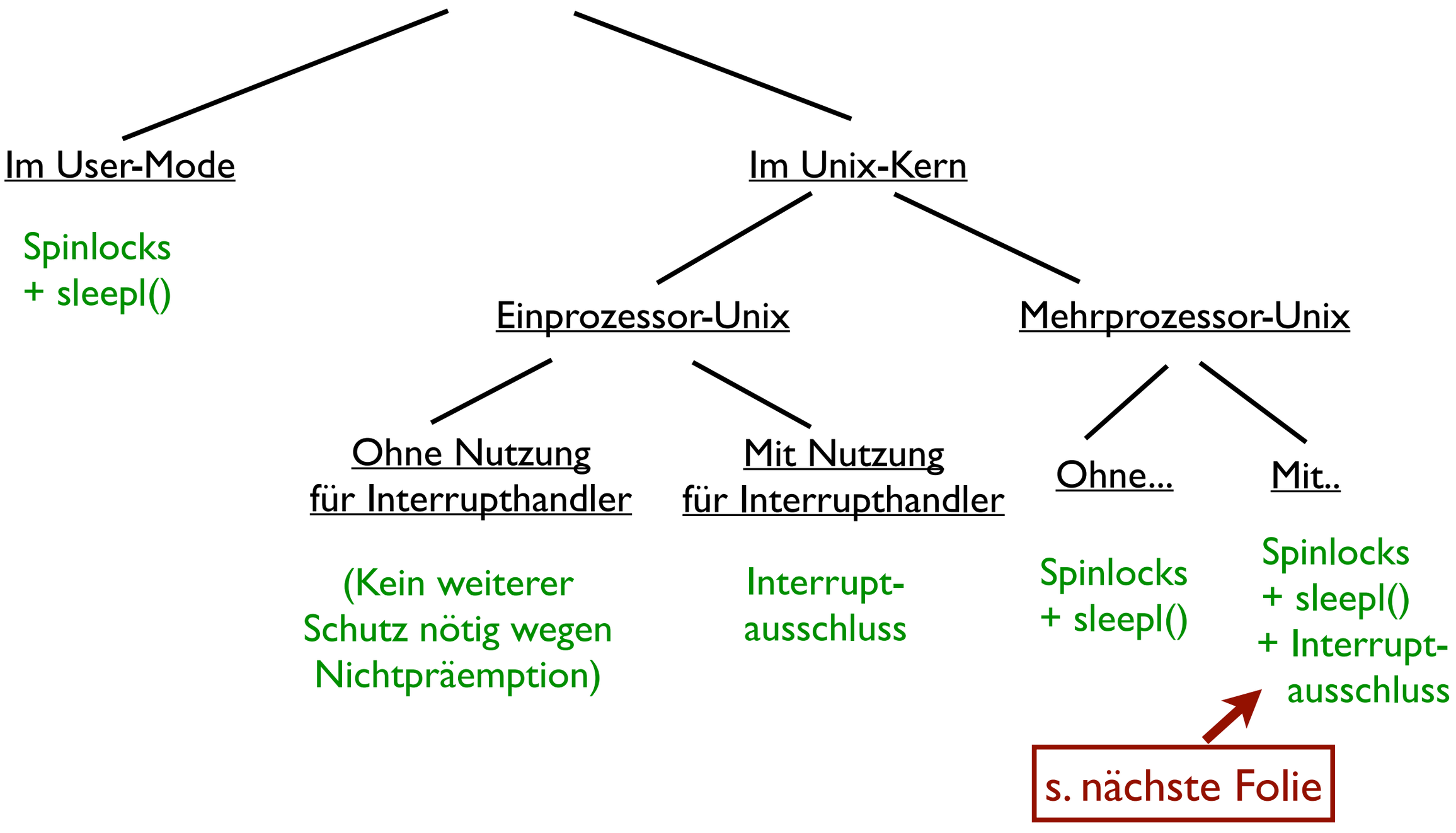
```
void Block::block_lock() {

  while (test_and_set(key))
    sleep(this);

}
```

```
void Block::block_unlock() {

    key = false;
    wakeup(this);

}
```

```
void Sema::P() {
  while (counter==0)
    sleep(this);
  --counter;
}
```

```
void Sema::V() {
  ++counter;
  wakeup(this);
}
```

Zusammenfassung: Realisierung von Semaphoren



⇒ dieselben Varianten auch bei unfairen Semaphoren denkbar

Realisierung von Semaphoren im Unix-Kern

- Abstrakte Implementierung:

```
class Sema {
...                    //keine Queue vorsehen
}

void Sema:P() {                    void Sema::V() {
  disable_interrupts();              disable_interrupts();
  s.spin_lock();                     s.spin_lock();
  while (counter==0)                 ++counter;
    sleepl(this,s);                  s.spin_unlock();
  --counter;                         enable_interrupts();
  s.spin_unlock();                   wakeup(this);
  enable_interrupts();               }
}
```

- Schutz von unerwünschter Nebenläufigkeit ggf. auch hier durch Unterbrechungsausschluss sowie, falls erforderlich, Spinlock und Verwendung von sleepl()

Original-Unix-Implementierung sieht syntaktisch etwas anders aus:

- Kein C++ ⇒ keine Klasse Sema

  ⇒ Semaphor wird über Zählervariable identifiziert

- Kern-interne Routinen zum Unterbrechungsausschluss verwendet (splhigh(), splx())
- Kein Spinlock/sleepl() nötig, da Einprozessor-Implementierung

```
P(int *sp) {
  x = splhigh();
  while (*sp==0)
    sleep(sp,...);
  --*sp;
  splx(x);
}
```

```
V(int *sp) {
  x = splhigh();
  ++*sp;
  splx(x);
  wakeup(sp);
}
```

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

b) Fairness von Semaphoren nicht immer adäquat
   - Gefahr von Konvois

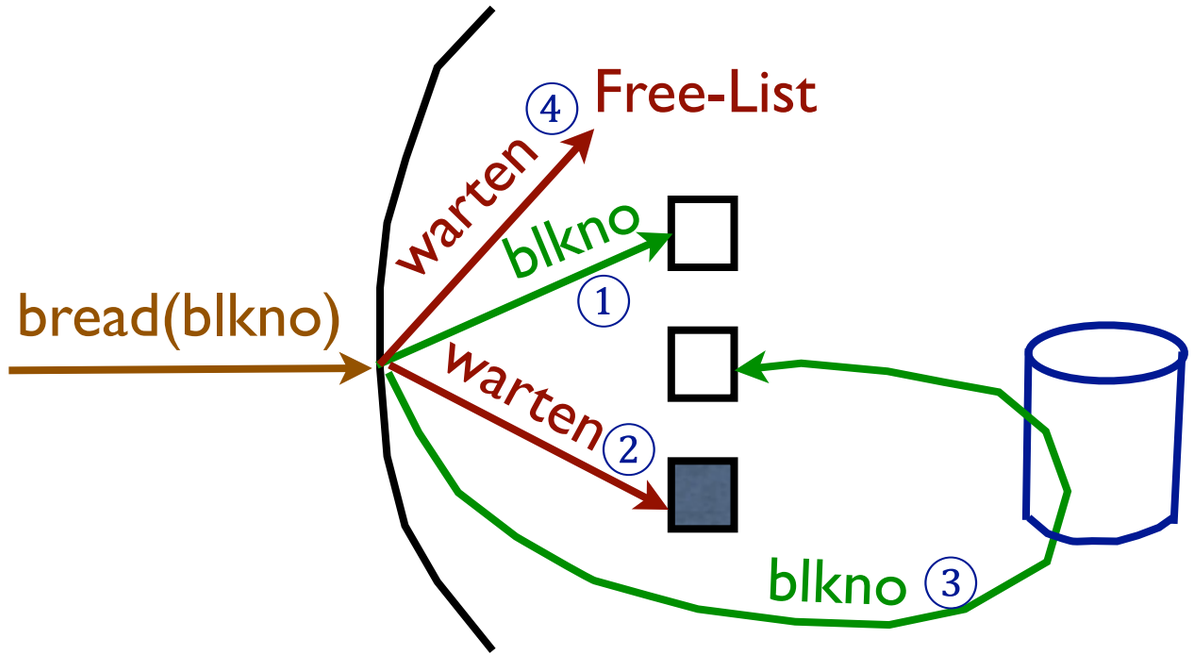
→ c) U.U. unerwünschte Kombination von drei Semantiken
   - Prüfen auf Verfügbarkeit einer Ressource
   - Ggf. Blockieren auf Verfügbarkeit
   - Reservieren der Ressource

     ⇒ nicht immer als Kombination sinnvoll

Ad c) Beispiel: Verwaltung eines Buffer-Caches (fiktives Beispiel)

- Wdh. Buffer-Cache: Plattenblöcke im Hauptspeicher puffern
  ⇒ u.U. Vermeiden von Plattenzugriffen

- Vier Situationen unterscheiden:
  1. Gewünschter Block im Buffer-Cache und verfügbar (in Freelist)
  2. Gewünschter Block im Buffer-Cache und gesperrt (keine Nebenläuf.)
     ⇒ warten
  3. Gewünschter Block nicht im Buffer-Cache, aber Platz zum Einlesen
  4. Gewünschter Block nicht im Buffer-Cache, kein Platz zum Einlesen (klassischer Buffer-Cache) ⇒ warten



Grober Algorithmus in Einprozessorsystem (vereinfacht)

```
bread() {
  again:
1   if (find buffer in cache) {
      if (buffer BUSY) {
        sleep(&buffer);
        goto again;
      }
      set buffer BUSY;
      freecnt--;
      take buffer off freelist;
    } else {
      if (freecnt==0) {
        sleep(&freecnt);
        goto again;
      }
      freecnt--;
      set first buffer on freelist BUSY;
      take first buffer off freelist;
      reassign buffer to different blocknr;
      read info from disk;
    }
  return(buffer);
}
```

Grober Algorithmus in Einprozessorsystem (vereinfacht)

```
bread() {
  again:
    if (find buffer in cache) {
      if (buffer BUSY) {
        sleep(&buffer);
        goto again;
      }
      set buffer BUSY;
      freecnt--;
      take buffer off freelist;
    } else {
      if (freecnt==0) {
        sleep(&freecnt);
        goto again;
      }
      freecnt--;
      set first buffer on freelist BUSY;
      take first buffer off freelist;
      reassign buffer to different blocknr;
      read info from disk;
    }
  return(buffer);
}
```

2

Grober Algorithmus in Einprozessorsystem (vereinfacht)

```
bread() {
  again:
    if (find buffer in cache) {
      if (buffer BUSY) {
        sleep(&buffer);
        goto again;
      }
      set buffer BUSY;
      freecnt--;
      take buffer off freelist;
    } else {
      if (freecnt==0) {
        sleep(&freecnt);
        goto again;
      }
      freecnt--;
③     set first buffer on freelist BUSY;
      take first buffer off freelist;
      reassign buffer to different blocknr;
      read info from disk;
    }
  return(buffer);
}
```

Grober Algorithmus in Einprozessorsystem (vereinfacht)

```
bread() {
  again:
    if (find buffer in cache) {
      if (buffer BUSY) {
        sleep(&buffer);
        goto again;
      }
      set buffer BUSY;
      freecnt--;
      take buffer off freelist;
    } else {
      if (freecnt==0) {
④       sleep(&freecnt);
        goto again;
      }
      freecnt--;
      set first buffer on freelist BUSY;
      take first buffer off freelist;
      reassign buffer to different blocknr;
      read info from disk;
    }
  return(buffer);
}
```

Grober Algorithmus in Einprozessorsystem (vereinfacht)

```
bread() {
  again:
    if (find buffer in cache) {
      if (buffer BUSY) {
        sleep(&buffer);
        goto again;
      }
      set buffer BUSY;
      freecnt--;
      take buffer off freelist;
    } else {
      if (freecnt==0) {
        sleep(&freecnt);
        goto again;
      }
      freecnt--;
      set first buffer on freelist BUSY;
      take first buffer off freelist;
      reassign buffer to different blocknr;
      read info from disk;
    }
  return(buffer);
}
```

Realisierung von bread() in Multiprozessorumgebung (fiktiv):

a) Zwei Arten von kritischen Abschnitten:
   - Zugriff auf gewünschten Puffer (inkl. Ressourcenverwaltung des Puffers)
   - Zugriff auf „Freelist“

b) Ressourcenverwaltung (beschränkte Anzahl von freien Puffern)

⇒ müssen in Multiprozessorumgebung geschützt werden

Realisierung von bread() in Multiprozessorumgebung (fiktiv):

a) Zwei Arten von kritischen Abschnitten:
   - Zugriff auf gewünschten Puffer (inkl. Ressourcenverwaltung des Puffers)
   - Zugriff auf „Freelist“

b) Ressourcenverwaltung (beschränkte Anzahl von freien Puffern)

⇒ müssen in Multiprozessorumgebung geschützt werden

1. Versuch:

- Einfach Semaphore verwenden:
  - Nebenläufigkeitsschutz für Puffer ⇒ sema buffer(1); //n Semaphore
  - Nebenläufigkeitsschutz für Freelist ⇒ sema freesema(1); //1 Semaphor
  - Ressourcenverwaltung ⇒ sema freelist(n); //1 Semaphor

Grober Algorithmus (vereinfacht)

```
bread() {
  again:
    if (find buffer in cache) {
      if (buffer BUSY) {
        sleep(&buffer);
        goto again;
      }
      set buffer BUSY;
      freecnt--;
      take buffer off freelist;
    } else {
      if (freecnt==0) {
        sleep(&freecnt);
        goto again;
      }
      freecnt--;
      set first buffer on freelist BUSY;
      take first buffer off freelist;
      reassign buffer to different blocknr;
      read info from disk;
    }
  return(buffer);
}
```

```
bread() {
  again:
    if (find buffer in cache) {
      P(buffer);

      P(freelist);
      P(freesema);
      take buffer off freelist;
      V(freesema);
    } else {
      P(freelist);

      P(first buffer on freelist);
      P(freesema);
      take first buffer off freelist;
      V(freesema);
      reassign buffer to different blocknumber;
      read info from disk;
    }
  return(buffer);
}
```

Grober Algorithmus (vereinfacht)

```
bread() {
  again:
    if (find buffer in cache) {
      if (buffer BUSY) {
        sleep(&buffer);
        goto again;
      }
      set buffer BUSY;
      freecnt--;
      take buffer off freelist;
    } else {
      if (freecnt==0) {
        sleep(&freecnt);
        goto again;
      }
      freecnt--;
      set first buffer on freelist BUSY;
      take first buffer off freelist;
      reassign buffer to different blocknr;
      read info from disk;
    }
  return(buffer);
}
```

```
bread() {
  again:
    if (find buffer in cache) {
      P(buffer);

      P(freelist);
      P(freesema);
      take buffer off freelist;
      V(freesema);
    } else {
      P(freelist);

      P(first buffer on freelist);
      P(freesema);
      take first buffer off freelist;
      V(freesema);
      reassign buffer to different blocknumber;
      read info from disk;
    }
  return(buffer);
}
```

Realisierung von bread() in Multiprozessorumgebung:

a) Zwei Arten von kritischen Abschnitten:
   - Zugriff auf gewünschten Puffer (inkl. Ressourcenverwaltung des Puffers)
   - Zugriff auf „Freelist“

b) Ressourcenverwaltung (beschränkte Anzahl von freien Puffern)

⇒ müssen in Multiprozessorumgebung geschützt werden

1. Versuch:

- Einfach Semaphore verwenden:
  - Nebenläufigkeitsschutz für Puffer ⇒ sema buffer(1); //n Semaphore
  - Nebenläufigkeitsschutz für Freelist ⇒ sema freesema(1); //1 Semaphor
  - Ressourcenverwaltung ⇒ sema freelist(n); //1 Semaphor

⇒ Trifft Semantik nicht genau

⇒ Während des Wartens kann:

- Inhalt des gewünschten Puffers falsch werden
- Ein anderer Puffer den richtigen Inhalt bekommen

2. Versuch:

- Zusätzliche Abfragen einbauen

  ⇒ sehr umständlich

Grober Algorithmus (vereinfacht)

```
bread() {
  again:
    if (find buffer in cache) {
      if (buffer BUSY) {
        sleep(&buffer);
        goto again;
      }
      set buffer BUSY;
      freecnt--;
      take buffer off freelist;
    } else {
      if (freecnt==0) {
        sleep(&freecnt);
        goto again;
      }
      freecnt--;
      set first buffer on freelist BUSY;
      take first buffer off freelist;
      reassign buffer to different blocknr;
      read info from disk;
    }
  return(buffer);
}
```

```
bread() {
  again:
    if (find buffer in cache) {
      P(buffer);
      if (wrong buffer) {
        V(buffer);
        goto again;
      }
      P(freelist);
      P(freesema);
      take buffer off freelist;
      V(freesema);
    } else {
      P(freelist);
      if (find buffer in cache) {
        V(freelist);
        goto again;
      }
      P(first buffer on freelist);
      P(freesema);
      take first buffer off freelist;
      V(freesema);
      reassign buffer to different blocknumber;
      read info from disk;
    }
  return(buffer);
}
```

2. Versuch:

- Zusätzliche Abfragen einbauen

  ⇒ sehr umständlich

→ ⇒ Semaphore sind in realen Umgebungen u.U. zu mächtig

  (bei fairen Semaphoren sogar Verklemmung möglich, da unterschiedlich geschachtelt)

Alternative: Auf Semaphore verzichten

- Ressourcenverwaltung wieder selbst realisieren

  ⇒ ursprünglichen Algorithmus aufgreifen

- Nebenläufigkeitsschutz auch mit Locks und Unterbrechungsausschluss realisierbar

  ⇒ geeignete kritische Abschnitte festlegen (ggf. Nutzung von sleepI())

Semaphore in der Unix-Multithreading-Umgebung

- Flexibel einsetzbar: zum Schutz von kritischen Abschnitten, einseitiger Synchronisation und „Ressourcen-Verwaltung“

- In Sema.hh abstrakte Nutzungsschnittstelle bereitgestellt (C++-Klasse):

  In main():

```
Sema s(count,1);  //s initialisiert auf count;
                  //2. Parameter: Art der Semaphore
...               //Threads erzeugen
```

In Prozedur eines Threads:

```
s.P();
... kritischer Abschnitt ...
s.V();
```

⇒ gleiche Syntax wie in obigen Beispielen

- Achtung:
  Originalschnittstelle des Pthread-Multithreadings hat deutlich andere Syntax
  (keine Klassenstruktur, andere Namen):

```
sema_init (&s, 1, count);
                    //Params: Name, Art, initialer Zählerwert
sema_wait(&s);                  // = s.P();
sema_post(&s);                  // = s.V();
```

Semaphore in C++20

- Mittlerweile:
  Semaphor-Unterstützung durch gängige C++-Compiler
- Initialisierung:

```
#include <semaphore>
std::counting_semaphore platz(10);
std::counting_semaphore produkt(0);
```

- Und die Aufrufe von P und V
  (in C++ wie in Java als „acquire()“ und „release()“):

```
platz.acquire();        // entspricht platz.P()
 // erzeugen, sobald Platz vorhanden
produkt.release();      // entspricht produkt.V()
 // nun kann Produkt an anderer Stelle verbraucht werden
```

Beispiel: Erzeuger/Verbraucher

```
#include <deque>
#include <iostream>
#include <semaphore>
#include <thread>

static std::counting_semaphore platz{10}, produkt{0};
using Lager = std::deque<unsigned long>;
static Lager lager;
static std::mutex m;

void erzeuger(void) {
  Lager::value_type cnt = 0;
  while (true) {
    platz.acquire();
    {
      const std::lock_guard<std::mutex> lock(m);
      lager.push_back(cnt++);
    }
    produkt.release();
  }
}

void verbraucher(void) {
  while (true) {
    produkt.acquire();
    {
      const std::lock_guard<std::mutex> lock(m);
      std::cout << lager.front() << std::endl;
      lager.pop_front();
    }
    platz.release();
  }
}

int main() {
  std::thread e{erzeuger}, v{verbraucher};

  e.join();
  v.join();
```

Fragen – Teil 3

- Welche Probleme gibt es mit „fairen Semaphoren“?
  Was sind „Konvois“, und was sind „donnernde Herden“?

Zusammenfassung

- Eigenschaften von Semaphoren
- Lösung der Beispielszenarien mit Semaphoren
- Implementierung von Semaphoren
- Beurteilung von Semaphoren

Semaphore – Fragen

1. Welche zusätzlichen Eigenschaften zeichnen *Semaphore* gegenüber blockierenden Locks aus?
2. Wie wird eine einseitige bzw. eine mehrseitige Synchronisation durch Semaphore ausgedrückt?
3. Wie können Semaphore zur Lösung des Problems der *speisenden Philosophen* eingesetzt werden?
4. In welches Problem wird eine allzu „einfache“ Semaphor-Implementierung laufen?
5. Welche Probleme gibt es mit „fairen Semaphoren“?
   Was sind „Konvois“, und was sind „donnernde Herden“?