Betriebssysteme I

WS 2019/2020

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 5. Dezember 2019

Betriebssysteme I

WS 2019/2020

3 Threadinteraktion

Klassen der Interaktion zwischen Threads (nach Stallings)

- Threads kennen sich gegenseitig nicht
  - nutzen aber gemeinsame Ressourcen (Geräte, Dateien, ...)
    - unbewußt (**Wettstreit**)
    - bewußt (**Kooperation durch Teilen**)
  - wichtig: **Synchronisation** (☞ **3.1**)
- Threads kennen sich (d.h. die Prozeß-/Threadkennungen)
  - **Kooperation durch Kommunikation** (☞ **3.2**)
- **Anmerkungen**:
  - Threads können ggf. in unterschiedlichen Prozessen liegen
  - in der Literatur hier i.a. keine klare Unterscheidung zwischen Threads und Prozessen

Betriebssysteme I

WS 2019/2020

3 Threadinteraktion

3.1 Synchronisation

Inhalt (1):

- Einführung und Motivation
- Wechselseitiger Ausschluß
- Wechselseitiger Ausschluß mit aktivem Warten
  - Lösungsversuche, korrekte Lösungen
- Semaphore

- Tanenbaum 2.3.1-2.3.6
- Stallings 5.1-5.4.1

Inhalt (2):

- Klassische Synchronisationsprobleme
  - Erzeuger/Verbraucher-Problem
  - Leser/Schreiber-Problem
- Monitore
- Synchronisation mit Java

- Tanenbaum 2.4.2, 2.4.3, 2.3.7
- Stallings 5.4.4, 5.5

3.1.1 Einführung und Motivation

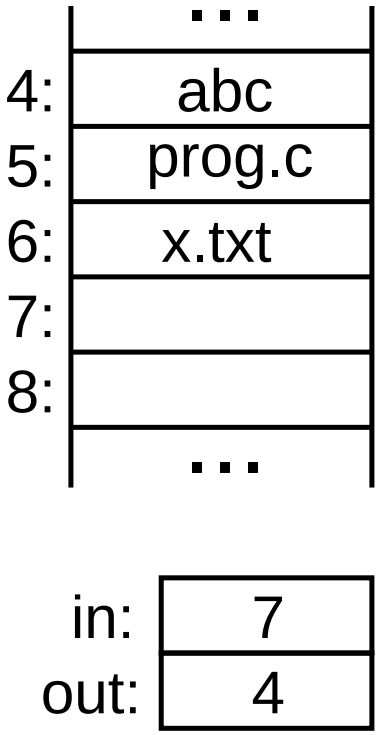
- Mehrprogrammbetrieb führt zu Nebenläufigkeit
  - Abarbeitung im Wechsel (praktisch) äquivalent zu echt paralleler Abarbeitung
- Mehrere Threads können gleichzeitig versuchen, auf gemeinsame Ressourcen zuzugreifen
  - Beispiel: Drucker
- Für korrekte Funktion in der Regel notwendig:
  - zu einem Zeitpunkt darf nur jeweils einem Thread der Zugriff erlaubt werden

Beispiel: Drucker-Spooler

- Threads tragen zu druckende Dateien in Spool-Bereich ein:
  - Spooler-Verzeichnis mit Einträgen 0, 1, 2, ... für Dateinamen
  - zwei gemeinsame Variable:
    - `out`: nächste zu druckende Datei
    - `in`: nächster freier Eintrag
  - in gemeinsamem Speicherbereich oder im Dateisystem
- Druck-Thread überprüft, ob Aufträge vorhanden sind und druckt die Dateien

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich



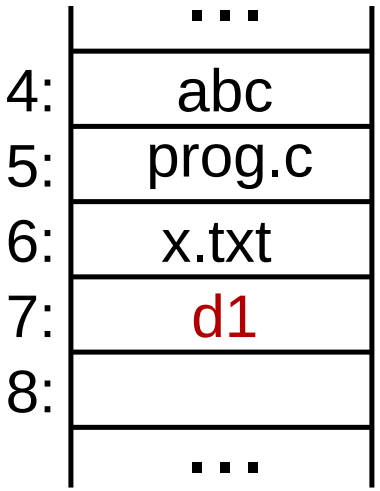
Thread A

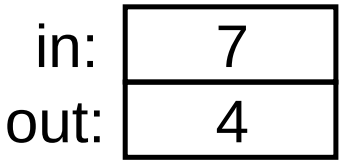
```
...
s[in]="d1";
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich





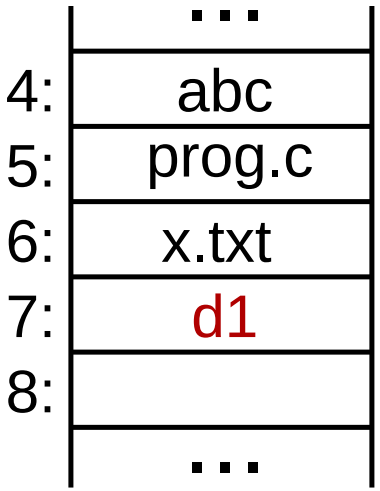
Thread A

```
...
s[in]="d1";
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich



in:	7
out:	4

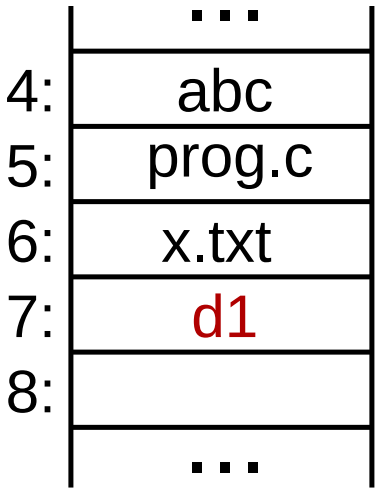
Thread A

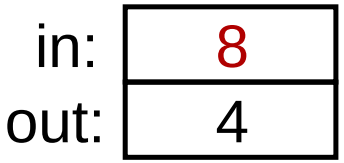
```
...
s[in]="d1";
in=in+1;
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich



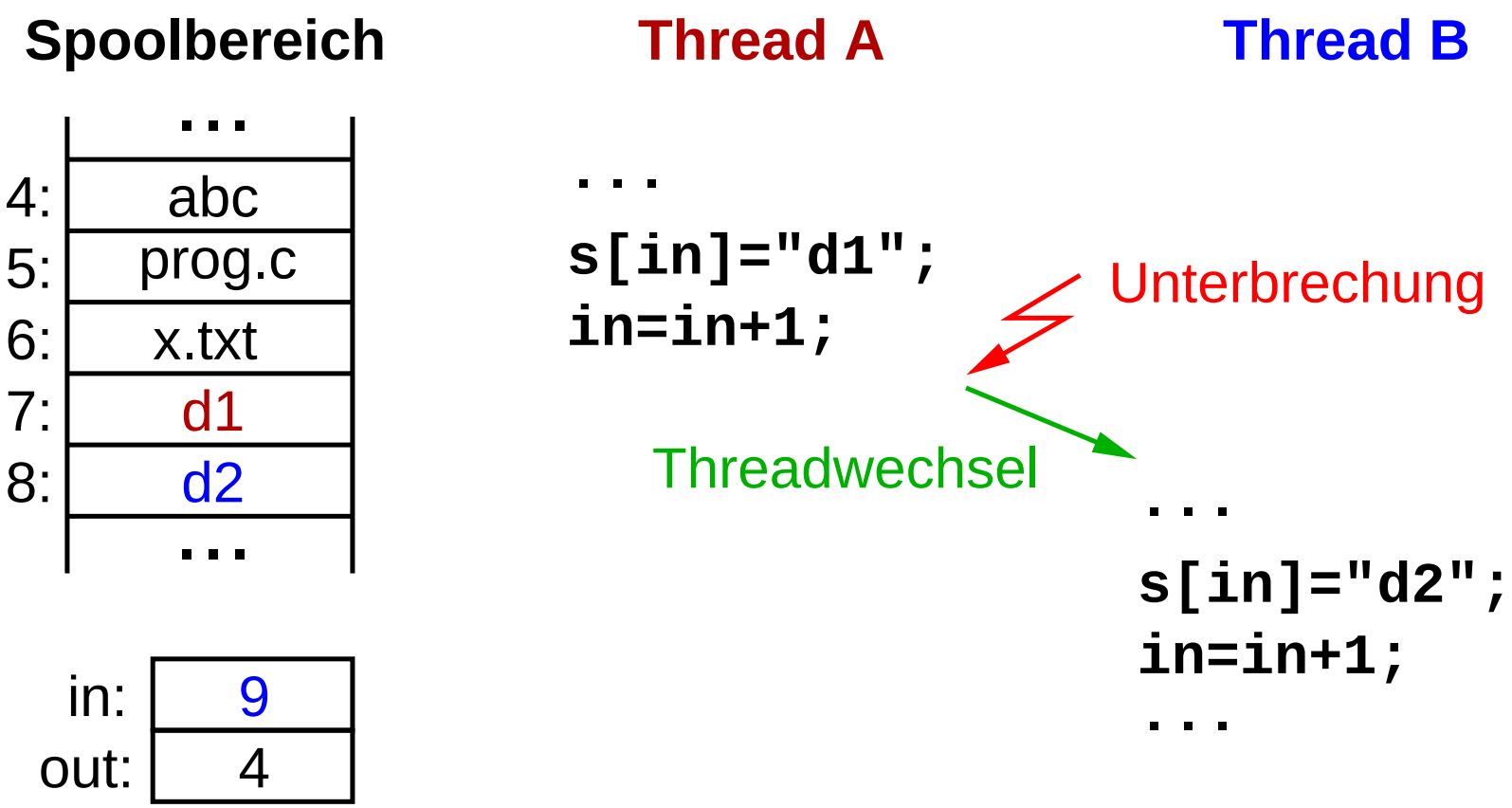


Thread A

```
...
s[in]="d1";
in=in+1;
```

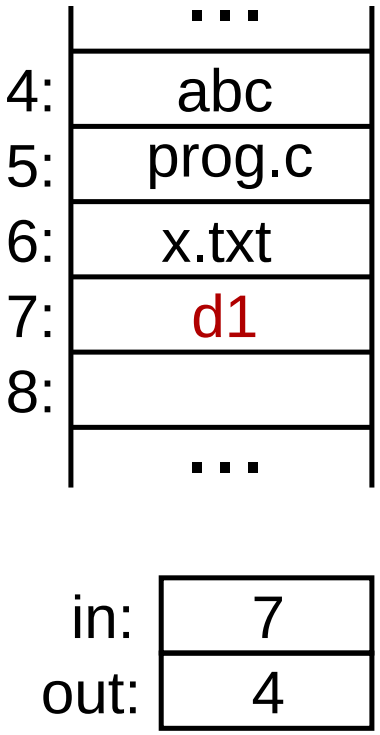
Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf



Beispiel: Drucker-Spooler, fehlerhafter Ablauf

Spoolbereich

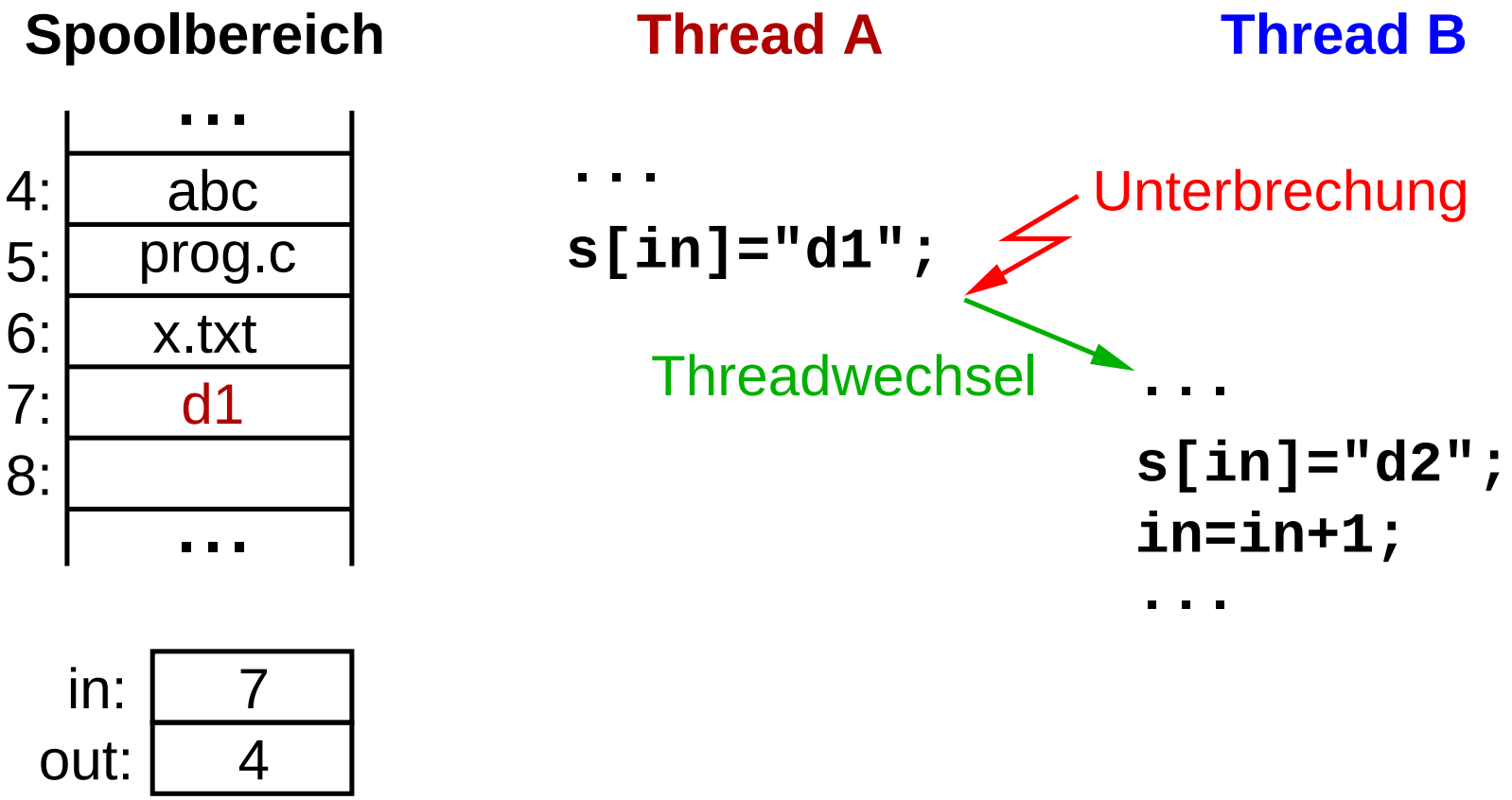


Thread A

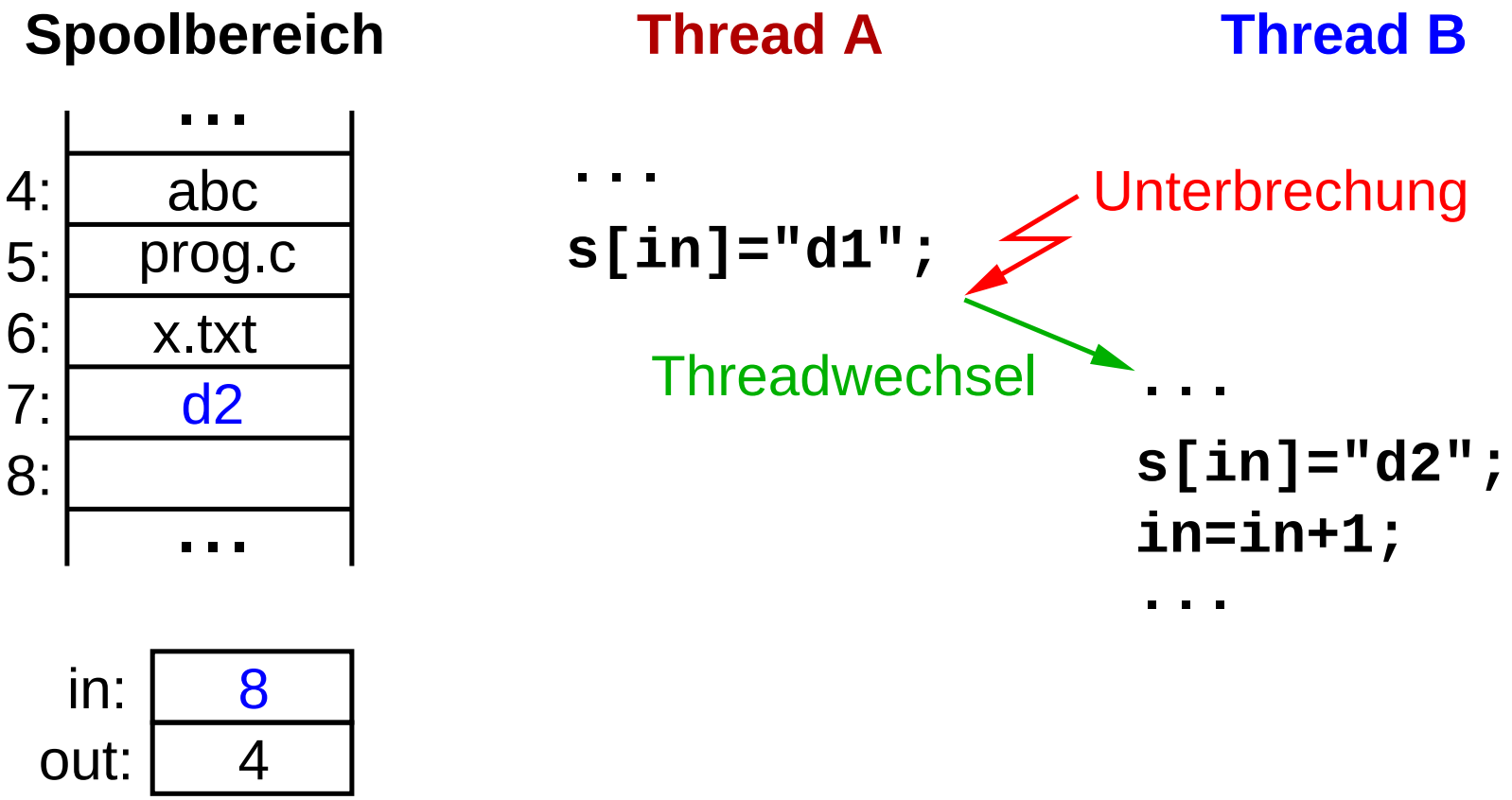
```
...
s[in]="d1";
```

Thread B

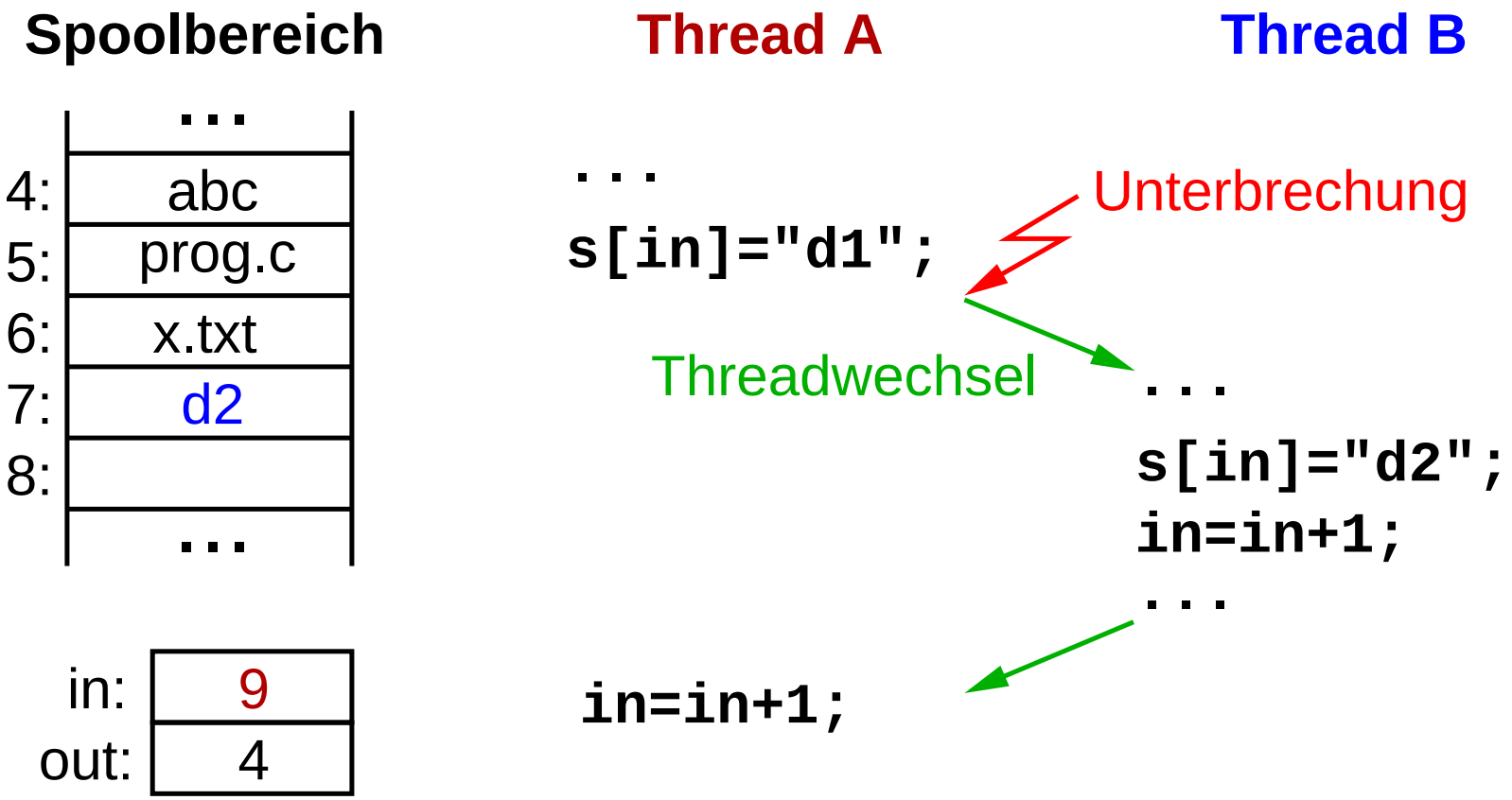
Beispiel: Drucker-Spooler, fehlerhafter Ablauf



Beispiel: Drucker-Spooler, fehlerhafter Ablauf



Beispiel: Drucker-Spooler, fehlerhafter Ablauf



➥ ***Race Condition***

Arten der Synchronisation

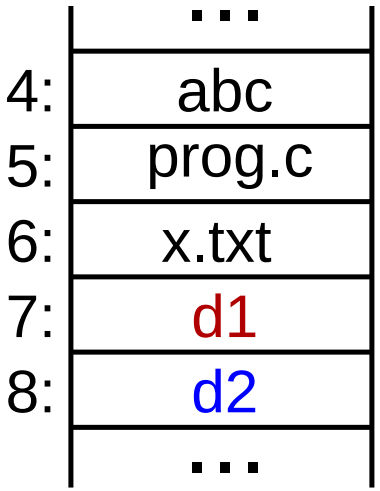
- **Sperrsynchronisation**
  - stellt sicher, daß Aktivitäten in verschiedenen Threads **nicht gleichzeitig** ausgeführt werden
  - d.h., die Aktivitäten werden nacheinander (in beliebiger Reihenfolge) ausgeführt
  - z.B. kein gleichzeitiges Drucken
- **Reihenfolgesynchronisation**
  - stellt sicher, daß Aktivitäten in verschiedenen Threads in einer **bestimmten Reihenfolge** ausgeführt werden
  - z.B. erst Datei erzeugen, dann lesen

3.1.2 Wechselseitiger Ausschluß

- **Kritischer Abschnitt**
  - Abschnitt eines Programms, der Zugriffe auf ein gemeinsam genutztes Objekt (**kritische Ressource**) enthält
- **Wechselseitiger Ausschluß** von Aktivitäten
  - zu jeder Zeit darf nur ein Thread die Aktivität ausführen
  - Sperrsynchronisation
- Gesucht: Methode zum wechselseitigen Ausschluß kritischer Abschnitte

Beispiel: Drucker-Spooler mit wechselseitigem Ausschluß

Spoolbereich



Thread A

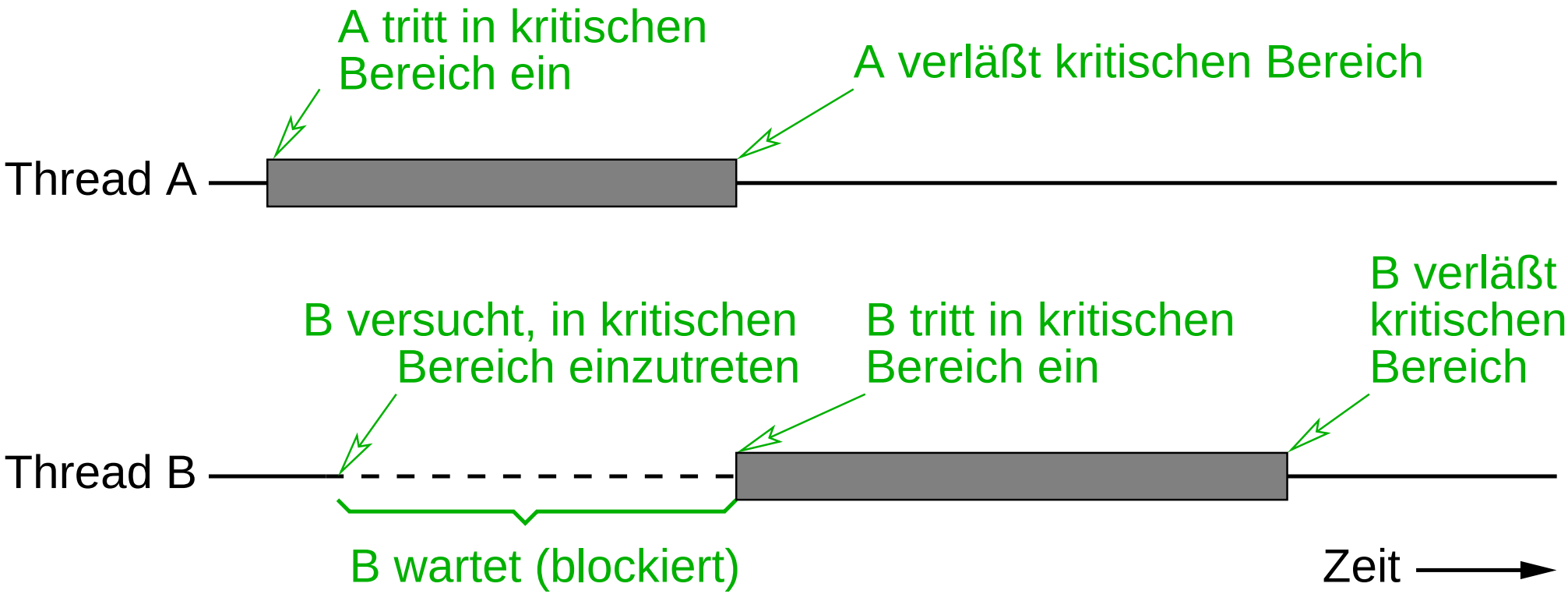
```
begin_region();
s[in]="d1";
in=in+1;
end_region();
```

Thread B

```
begin_region();
s[in]="d2";
in=in+1;
end_region();
```

➥ Frage: Implementierung von `begin_region()` / `end_region()`?

Idee des wechselseitigen Ausschlusses



Anforderungen an Lösung zum wechselseitigen Ausschluß:

1. Höchstens ein Thread darf im kritischen Abschnitt (k.A.) sein
2. Keine Annahmen über Geschwindigkeit / Anzahl der CPUs
3. Thread außerhalb des k.A. darf andere nicht behindern
4. Kein Thread sollte ewig warten müssen, bis er in k.A. eintreten darf
   - ➥ Voraussetzung: kein Thread bleibt ewig im k.A.
5. Sofortiger Zugang zum k.A., wenn kein anderer Thread im k.A. ist

Lösungsversuch 1: Sperren der Interrupts

- Abgesehen von freiwilliger Abgabe der CPU: Threadwechsel nur durch Interrupt
- Sperren der Interrupts in `begin_region()`, Freigabe in `end_region()`
- Probleme:
  - Ein-/Ausgabe ist blockiert
  - BS verliert Kontrolle über den Thread
  - Funktioniert nur bei Einprozessor-Rechnern
- Anwendung aber im BS selbst

Lösungsversuch 2: Sperrvariable

- Variable `belegt` zeigt an ob kritischer Abschnitt belegt
- Thread 0

```
while(belegt);          } begin_region()
belegt = true;          }
// kritischer Abschnitt
belegt = false          } end_region()
```

- Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:

Lösungsversuch 2: Sperrvariable

- Variable `belegt` zeigt an ob kritischer Abschnitt belegt
- Thread 0

```
while(belegt);          } begin_region()
belegt = true;          }
// kritischer Abschnitt
belegt = false          } end_region()
```

- Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:
  - Threads führen gleichzeitig `begin_region()` aus
  - lesen gleichzeitig `belegt`
  - finden `belegt` auf `false`

Lösungsversuch 2: Sperrvariable

- Variable `belegt` zeigt an ob kritischer Abschnitt belegt
- Thread 0

```
while(belegt);          } begin_region()
belegt = true;          }
// kritischer Abschnitt
belegt = false          } end_region()
```

- Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:
  - Threads führen gleichzeitig `begin_region()` aus
  - lesen gleichzeitig `belegt`
  - finden `belegt` auf `false`
  - setzen `belegt=true`

Lösungsversuch 2: Sperrvariable

- Variable `belegt` zeigt an ob kritischer Abschnitt belegt
- Thread 0

```
while(belegt);          } begin_region()
belegt = true;          }
// kritischer Abschnitt
belegt = false          } end_region()
```

- Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:
  - Threads führen gleichzeitig `begin_region()` aus
  - lesen gleichzeitig `belegt`
  - finden `belegt` auf `false`
  - setzen `belegt=true` und betreten kritischen Abschnitt!!!

Lösungsversuch 3: Strikter Wechsel

- Variable `turn` gibt an, wer an der Reihe ist
- Thread 0

```
while (turn != 0);
// kritischer Abschnitt
turn = 1;
```

Thread 1

```
while (turn != 1);
// kritischer Abschnitt
turn = 0
```

- Problem:
  - Threads **müssen** abwechselnd in kritischen Abschnitt
  - verletzt Anforderungen 3, 4, 5

Lösungsversuch 4: Erst belegen, dann prüfen

- Variable `interested[i]` zeigt an, ob Thread `i` in den kritischen Abschnitt will
- Thread 0

```
interested[0] = true;
while (interested[1]);
// kritischer Abschnitt
interested[0] = false;
```

Thread 1

```
interested[1] = true
while (interested[0]);
// kritischer Abschnitt
interested[1] = false
```

- Problem:
  - Verklemmung, falls Threads `interested` gleichzeitig auf `true` setzen

Eine richtige Lösung: Peterson-Algorithmus

- Thread 0

```
interested[0] = true;
turn = 1;
while ((turn != 0) &&
       interested[1]);
// kritischer Abschnitt
interested[0] = false;
```

Thread 1

```
interested[1] = true
turn = 0;
while ((turn != 1) &&
       interested[0]);
// kritischer Abschnitt
interested[1] = false
```

- Verklemmung wird durch `turn` verhindert
- Jeder Thread bekommt die Chance, den kritischen Bereich zu betreten
  - keine **Verhungerung**

Zur Korrektheit des Peterson-Algorithmus

- **Wechselseitiger Ausschluß**:
  - Widerspruchsannahme: T0 und T1 beide im k.A.
  - damit: `interested[0]=true` und `interested[1]=true`
  - falls `turn=1`:
    - da T0 im k.A.: in der `while`-Schleife muß `turn==0` oder `interested[1]==false` gewesen sein
    - falls `turn==0` war: Widerspruch! (wer hat `turn=1` gesetzt?)
    - falls `interested[1]==false` war:
      T1 hat `interested[1]=true` noch nicht ausgeführt, hätte also später `turn==0` gesetzt und blockiert, Widerspruch!
  - falls `turn=0`:
    - symmetrisch!

Betriebssysteme I

WS 2019/2020

14.11.2019

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 5. Dezember 2019

Zur Korrektheit des Peterson-Algorithmus

- **Verklemmungs- und Verhungerungsfreiheit**:
  - Annahme: T0 dauernd in `while`-Schleife blockiert
  - Damit: immer `turn=1` und `interested[1]=true`
  - Mögliche Fälle für T1:
    - T1 will nicht in k.A.: `interested[1]` wäre `false`!
    - T1 wartet in Schleife: geht nicht wegen `turn==1` !
    - T1 ist immer im k.A.: nicht erlaubt!
    - T1 kommt immer wieder in k.A.: geht nicht, da T1 `turn=0` setzt, damit kann aber T0 in k.A.!
  - In allen Fällen ergibt sich ein Widerspruch

Lösungen mit Hardware-Unterstützung

- Problem bei den Lösungsversuchen:
  - Abfragen und Ändern einer Variable sind zwei Schritte
- Lösung: **atomare** *Read-Modify-Write* Operation der CPU
  - z.B. Maschinenbefehl *Test-and-Set*

```
bool TestAndSet(bool &var) { // var: Referenzparameter
    bool tmp = var; var = true; return tmp;
}
```

  - ununterbrechbar, auch in Multiprozessorsystemen unteilbar
- Lösung mit *Test-and-Set*:

```
while(TestAndSet(belegt));
// kritischer Abschnitt
belegt = false;
```

Aktives Warten (*Busy Waiting*)

- In bisherigen Lösungen: Warteschleife (***Spinlock***)
- Probleme:
  - Thread belegt CPU während des Wartens
  - Bei Einprozessorsystem und Threads mit Prioritäten sind Verklemmungen möglich:
    - Thread H hat höhere Priorität wie L, ist aber blockiert
    - L rechnet, wird in kritischem Abschnitt unterbrochen; H wird rechenbereit
    - H will in kritischen Abschnitt, wartet auf L; L kommt nicht zum Zug, solange H rechenbereit ist ...
- Notwendig bei Multiprozessorsystemen
  - für kurze kritische Abschnitte im BS-Kern

3.1.4 Semaphore

- Eingeführt durch Edsger Wybe Dijkstra (1965)
- Allgemeines Synchronisationskonstrukt
  - nicht nur wechselseitiger Ausschluß, auch Reihenfolge-synchonisation
- Semaphor ist i.W. eine ganzzahlige Variable
  - Wert kann auf nichtnegative Zahl initialisiert werden
  - zwei **atomare** Operationen:
    - `P()` (auch `wait`, `down` oder `acquire`)
      - verringert Wert um 1
      - falls Wert < 0: Thread blockieren
    - `V()` (auch `signal`, `up` oder `release`)
      - erhöht Wert um 1
      - falls Wert ≤ 0: einen blockierten Thread wecken

Semaphor-Operationen

```
struct Semaphor {
  int count;              // Semaphor−Zähler
  ThreadQueue queue;      // Warteschlange für blockierte Threads
}
```

```
void P(Semaphor &s) {
  s.count−−;
  if (s.count < 0) {
    Thread in s.queue
      ablegen;
    Thread blockieren;
  }
}
```

```
void V(Semaphor &s) {
  s.count++;
  if (s.count <= 0) {
    Einen Thread T aus s.queue
      entfernen;
    T auf bereit setzen;
  }
}
```

➥ Hinweis: Tanenbaum definiert Semaphore etwas anders (Zähler zählt höchstens bis 0 herunter)

Interpretation des Semaphor-Zählers

- Zähler ≥ 0: Anzahl freier Ressourcen
- Zähler < 0: Anzahl wartender Threads

Wechselseitiger Ausschluß mit Semaphoren

- Thread 0

```
P(Mutex);
// kritischer Abschnitt
V(Mutex);
```

Thread 1

```
P(Mutex);
// kritischer Abschnitt
V(Mutex);
```

- Semaphor `Mutex` wird mit 1 vorbelegt
- Semaphor, das an positiven Werten nur 0 oder 1 haben kann, heißt **binäres Semaphor**

Reihenfolgesynchronisation mit Semaphoren

- Beispiel: Thread 1 darf Datei erst lesen, nachdem Thread 0 sie erzeugt hat
- Thread 0

```
// Datei erzeugen
V(Sema);
```

Thread 1

```
P(Sema);
// Datei lesen
```

- Semaphor `Sema` wird mit 0 vorbelegt
  - damit: Thread 1 wird blockiert, bis Thread 0 die `V()`-Operation ausgeführt hat
- Merkregel:
  - `P()`-Operation an der Stelle, wo gewartet werden muß
  - `V()`-Operation signalisiert, daß Wartebedingung erfüllt ist
  - vgl. die alternativen Namen `wait()` / `signal()` für `P()` / `V()`

Realisierung von Semaphoren

- Eng verbunden mit Thread-Implementierung
- Bei Kernel-Threads:
  - Implementierung im BS-Kern
  - Operationen sind Systemaufrufe
  - atomare Ausführung durch Interrupt-Sperre und Spinlocks gesichert

Das Erzeuger/Verbraucher-Problem

- Situation:
  - Zwei Thread-Typen: Erzeuger, Verbraucher
  - Kommunikation über einen gemeinsamen, beschränkten Puffer der Länge `N`
    - Operationen `insert_item()`, `remove_item()`
  - Erzeuger legen Elemente in Puffer, Verbraucher entfernen sie
- Synchronisation:
  - Sperrsynchronisation: wechselseitiger Ausschluß
  - Reihenfolgesynchronisation:
    - kein Entfernen aus leerem Puffer: Verbraucher muß warten
    - kein Einfügen in vollen Puffer: Erzeuger muß warten

Lösung des Erzeuger/Verbraucher-Problems

Erzeuger

```
while (true) {
   item = Produce();

   insert_item(item);

}
```

Verbraucher

```
while (true) {

   item = remove_item();

   Consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
```

für wechselseitigen Ausschluß

Erzeuger

```
while (true) {
   item = Produce();

   P(mutex);
   insert_item(item);
   V(mutex);

}
```

Verbraucher

```
while (true) {

   P(mutex);
   item = remove_item();
   V(mutex);

   Consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
Semaphor full = 0;
```

für wechselseitigen Ausschluß

verhindert Entfernen aus leerem Puffer

Erzeuger

```
while (true) {
   item = Produce();

   P(mutex);
   insert_item(item);
   V(mutex);
   V(full);
}
```

Verbraucher

```
while (true) {
   P(full);
   P(mutex);
   item = remove_item();
   V(mutex);

   Consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
Semaphor full = 0;
```

für wechselseitigen Ausschluß

verhindert Entfernen aus leerem Puffer

Erzeuger

```
while (true) {
   item = Produce();

   P(mutex);
   insert_item(item);
   V(mutex);
   V(full);
}
```

Verbraucher

```
while (true) {
   P(full);
   P(mutex);
   item = remove_item();
   V(mutex);

   Consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
Semaphor full = 0;
Semaphor empty = N;
```

für wechselseitigen Ausschluß

verhindert Entfernen aus leerem Puffer

verhindert Einfügen in vollen Puffer

Erzeuger

```
while (true) {
   item = Produce();
   P(empty);
   P(mutex);
   insert_item(item);
   V(mutex);
   V(full);
}
```

Verbraucher

```
while (true) {
   P(full);
   P(mutex);
   item = remove_item();
   V(mutex);
   V(empty);
   Consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
Semaphor full = 0;
Semaphor empty = N;
```

für wechselseitigen Ausschluß

verhindert Entfernen aus leerem Puffer

verhindert Einfügen in vollen Puffer

Erzeuger

```
while (true) {
   item = Produce();
   P(empty);
   P(mutex);
   insert_item(item);
   V(mutex);
   V(full);
}
```

Verbraucher

```
while (true) {
   P(full);
   P(mutex);
   item = remove_item();
   V(mutex);
   V(empty);
   Consume(item);
}
```

Das Leser/Schreiber-Problem

- Gemeinsamer Datenbereich mehrerer Threads
- Zwei Klassen von Threads (bzw. Zugriffen)
  - Leser (*Reader*)
    - dürfen gleichzeitig mit anderen Lesern zugreifen
  - Schreiber (*Writer*)
    - stehen unter wechselseitigem Ausschluß, auch mit Lesern
      - verhindert Lesen von inkonsistenten Daten
- Typisches Problem in Datenbank-Systemen

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   readDataBase();

   UseData();
}
```

Semaphore und gemeinsame Variable

Schreiber

```
while(true) {
   CreateData();

   writeDataBase();

}
```

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   readDataBase();

   UseData();
}
```

Semaphore und gemeinsame Variable

```
Semaphor db=1; // Schützt Datenbank
```

Schreiber

```
while(true) {
   CreateData();
   P(db);
   writeDataBase();
   V(db);
}
```

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   P(db);

   readDataBase();

   V(db);

   UseData();
}
```

Semaphore und gemeinsame Variable

```
Semaphor db=1; // Schützt Datenbank
```

Schreiber

```
while(true) {
   CreateData();
   P(db);
   writeDataBase();
   V(db);
}
```

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   rc++;
   if (rc == 1)
     P(db);

   readDataBase();

   rc--;
   if (rc == 0)
     V(db);

   UseData();
}
```

Semaphore und gemeinsame Variable

```
int rc=0;       // Anzahl Leser
Semaphor db=1;  // Schützt Datenbank
```

Schreiber

```
while(true) {
   CreateData();
   P(db);
   writeDataBase();
   V(db);
}
```

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {
   P(mutex);
   rc++;
   if (rc == 1)
     P(db);
   V(mutex);
   readDataBase();
   P(mutex);
   rc--;
   if (rc == 0)
     V(db);
   V(mutex);
   UseData();
}
```

Semaphore und gemeinsame Variable

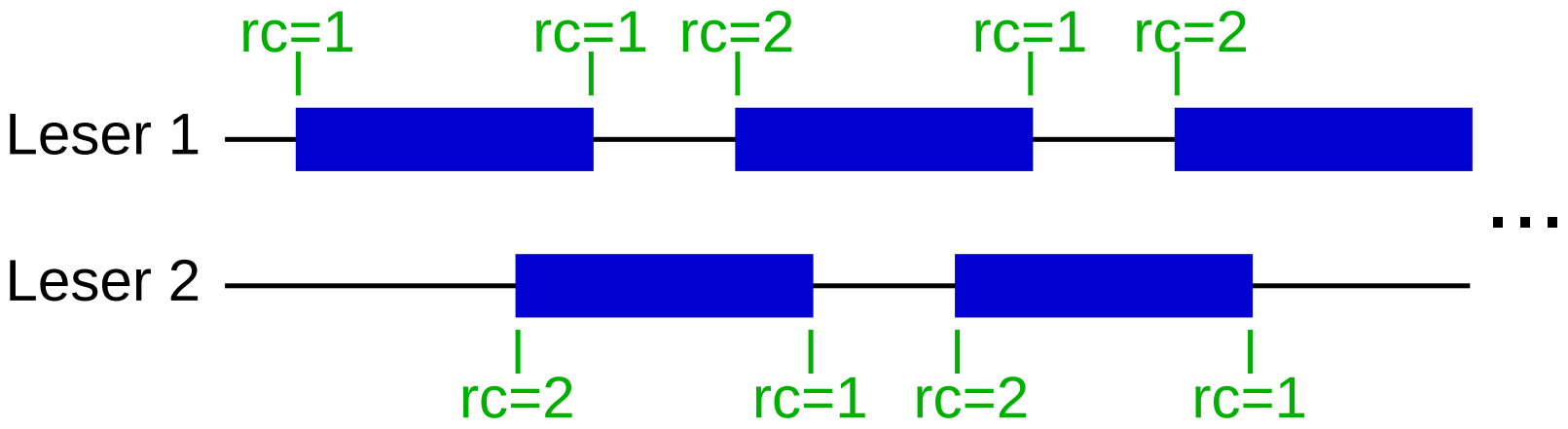
```
int rc=0;        // Anzahl Leser
Semaphor db=1;   // Schützt Datenbank
Semaphor mutex=1;
```

Schreiber

```
while(true) {
   CreateData();
   P(db);
   writeDataBase();
   V(db);
}
```

Eigenschaft der skizzierten Lösung

- Die Synchronisation ist unfair: Leser haben Priorität vor Schreibern
  - Schreiber kann verhungern



- Mögliche Lösung:
  - neue Leser blockieren, wenn ein Schreiber wartet

Betriebssysteme I

WS 2019/2020

21.11.2019

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 5. Dezember 2019

3.1.6 Monitore

Motivation

- Programmierung mit Semaphoren ist schwierig
  - Reihenfolge der P/V-Operationen: Verklemmungsgefahr
  - Synchronisation über gesamtes Programm verteilt

Monitor (Hoare, 1974; Brinch Hansen 1975)

- Modul mit Daten, Prozeduren und Initialisierungscode
  - Zugriff auf die Daten nur über Monitor-Prozeduren
  - (entspricht in etwa einer Klasse)
- Alle Prozeduren stehen unter wechselseitigem Ausschluß
  - nur jeweils ein Thread kann Monitor benutzen
- Programmiersprachkonstrukt: Realisierung durch Übersetzer

Bedingungsvariable (Zustandsvariable, *condition variables*)

- Zur Reihenfolgesynchronisation zwischen Monitor-Prozeduren
- Darstellung anwendungsspezifischer Bedingungen
  - z.B. voller Puffer im Erzeuger/Verbraucher-Problem
- Zwei Operationen:
  - `wait()`: Blockieren des aufrufenden Threads
  - `signal()`: Aufwecken blockierter Threads
- Bedingungsvariable verwaltet Warteschlange blockierter Threads
- Bedingungsvariable hat kein „Gedächtnis“:
  - `signal()` weckt nur einen Thread, der `wait()` bereits aufgerufen hat

Funktion von `wait()`:

- Aufrufender Thread wird blockiert
  - nach Ende der Blockierung kehrt `wait()` zurück
- Aufrufender Thread wird in die Warteschlange der Bedingungsvariable eingetragen
- Monitor steht bis zum Ende der Blockierung anderen Threads zur Verfügung

Funktion von `signal()`:

- Falls Warteschlange der Bedingungsvariable nicht leer:
  - mindestens einen Thread wecken:
    aus Warteschlange entfernen und Blockierung aufheben

Varianten für `signal()`:

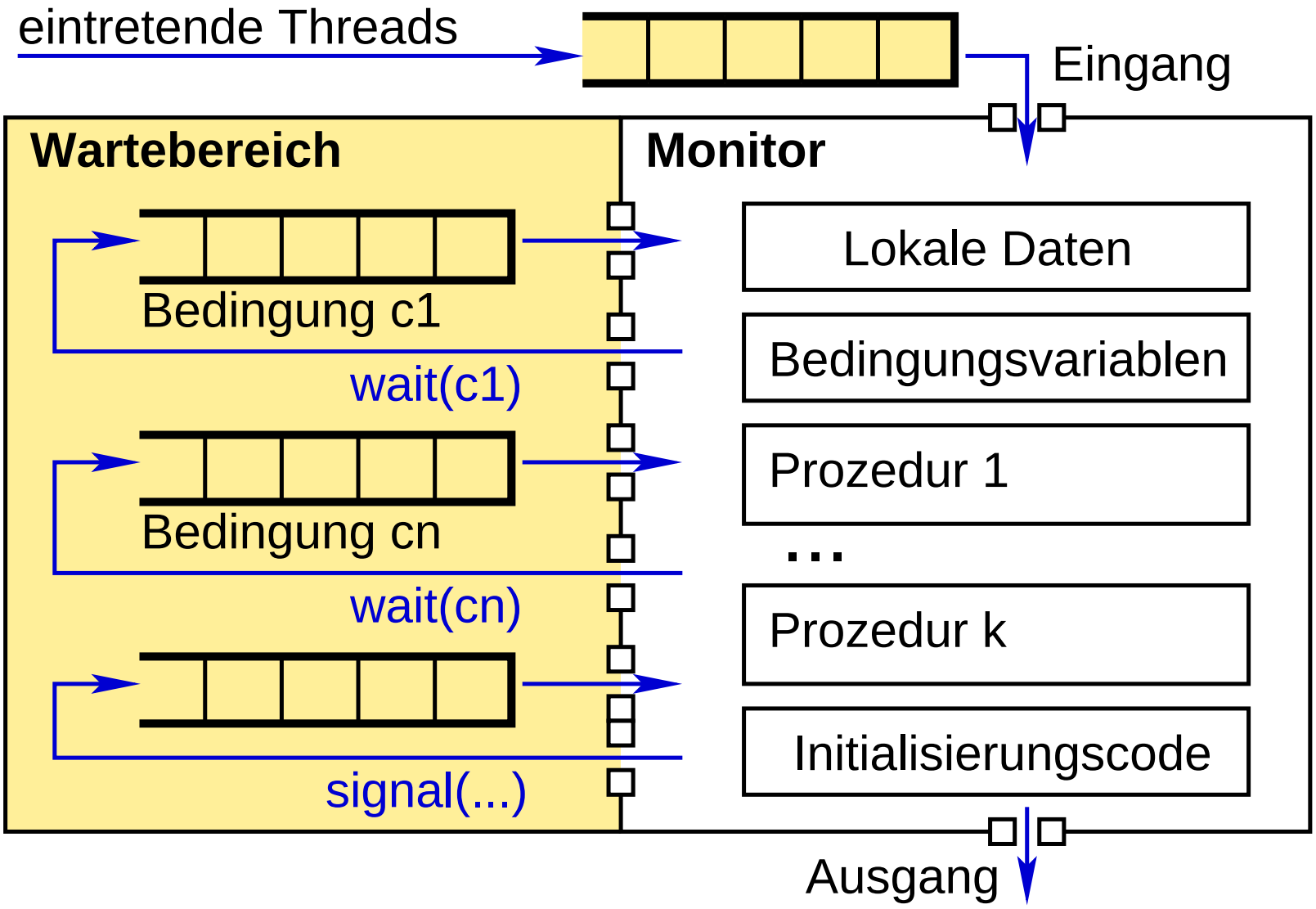
1. Ein Thread wird geweckt (meist der am längsten wartende)
   - **a)** signalisierender Thread bleibt im Besitz des Monitors
   - **b)** geweckter Thread erhält den Monitor sofort
     - **i.** signalisierender Thread muß sich erneut bewerben (Hoare)
     - **ii.** `signal()` muß letzte Anweisung in Monitorprozedur sein (Brinch Hansen)
2. Alle Threads werden geweckt
   - ➥ signalisierender Thread bleibt im Besitz des Monitors

➥ Bei 1a) und 2) ist nach Rückkehr aus `wait()` **nicht** sicher, daß die Bedingung (noch) erfüllt ist!

Typische Verwendung von `wait()` und `signal()`

- Testen einer Bedingung
  - bei Variante 1b):
    - `if (!Bedingung) wait(condVar);`
  - bei Varianten 1a) und 2):
    - `while (!Bedingung) wait(condVar);`
- Signalisieren der Bedingung
  - [`if (Bedingung)`] `signal(condVar);`

Aufbau eines Monitors nach Hoare



Semaphor-Realisierung m. Monitor (Pascal-artig, Brinch Hansen)

```
monitor Semaphor
  condition nonbusy;
  integer count;

  procedure P
  begin
    count := count - 1;
    if count < 0 then
      wait(nonbusy);
  end;

  procedure V
  begin
    count := count + 1;
    if count <= 0 then
      signal(nonbusy);
  end;

  count = 1;
end monitor;
```

➥ Umgekehrt können auch Monitore (insbes. Bedingungsvariable) mit Semaphoren nachgebildet werden

Erzeuger/Verbraucher m. Monitor (Pascal-artig, Brinch Hansen)

```
monitor ErzeugerVerbraucher
  condition nonfull, nonempty;
  integer count;

  procedure Insert(item: integer)
  begin
    if count = N then
      wait(nonfull);
    insert_item(item);
    count := count + 1;
    signal(nonempty);
  end;

  function Remove: integer
  begin
    if count = 0 then
      wait(nonempty);
    Remove := remove_item();
    count := count - 1;
    signal(nonfull);
  end;

  count = 0;
end monitor;
```

Motivation für Broadcast-Signalisierung (Variante 2)

- Aufwecken aller Threads sinnvoll, wenn unterschiedliche Wartebedingungen vorliegen
- Beispiel: Erzeuger/Verbraucher-Problem mit variablem Bedarf an Puffereinträgen

```
procedure Insert(item: ..., size: integer)
begin
  while count + size > N do
    wait(nonfull);
  ...
```

- Nachteil: viele Threads konkurrieren um Wiedereintritt in den Monitor

Basiskonzept: Java bietet Monitor-ähnliche Konstrukte an

- Klassenkonzept statt Modulkonzept
- Synchronisierte Methoden
  - müssen explizit als `synchronized` deklariert werden
  - stehen (pro Objekt!) unter wechselseitigem Ausschluß
- Keine expliziten Bedingungsvariablen, stattdessen genau eine implizite Bedingungsvariable pro Objekt
- Basisklasse `Object` definiert Methoden `wait()`, `notify()` und `notifyAll()`
  - diese Methoden werden von allen Klassen geerbt
  - `notify()`: Signalisierungsvariante 1a)
  - `notifyAll()`: Signalisierungsvariante 2)

Beispiel: Erzeuger/Verbraucher-Problem

```
public class ErzVerb {
  ...
  public synchronized void Insert(int item) {
    while (count == buffer.getSize()) { // Puffer voll?
      try {
        wait();                         //   ja: warten ...
      }
      catch (InterruptedException e) {}
    }
    buffer.insertItem(item);            // Item eintragen
    count++;
    notifyAll();                        // alle wecken
  }
```

Beispiel: Erzeuger/Verbraucher-Problem ...

```
public synchronized int Remove() {
  int result;
  while (count == 0) {                    // Puffer leer?
    try {
      wait();                             //   ja: warten ...
    }
    catch (InterruptedException e) {}
  }
  result = buffer.removeItem();           // Item entfernen
  count--;
  notifyAll();                            // alle wecken
  return result;
}
```

Anmerkungen zum Beispiel

- Vollständiger Code ist im WWW verfügbar
  - über die Vorlesungsseite
- `notify()` statt `notifyAll()` ist **nicht** korrekt!
  - funktioniert nur bei genau einem Erzeuger und genau einem Verbraucher
    - da nur eine Bedingungsvariable für zwei verschiedene Bedingungen benutzt wird, kann es sein, daß der falsche Thread aufgeweckt wird
  - Übungsaufgabe:
    - mit Programm aus WWW ausprobieren!
    - mit Simulator (Java-Applet) nachvollziehen!

Weiterführende Synchronisationskonzepte

- Ab JDK 1.5: Paket `java.util.concurrent`
- Bietet u.a.:
  - Semaphore: Klasse `Semaphore`
  - *Mutual Exclusion Locks* (Mutex): Schnittstelle `Lock`
    - Verhalten wie binäres Semaphor
    - Zustände: gesperrt, frei
  - Bedingungsvariable: Schnittstelle `Condition`
    - fest an ein `Lock` gebunden
    - erlaubt zusammen mit `Lock` Nachbildung des vollständigen Monitor-Konzepts
    - `Lock` wird für wechselseitigen Ausschluß der Monitor-Prozeduren genutzt

Klasse `Semaphore`

- Konstruktor: `Semaphore(int wert)`
  - erzeugt Semaphor mit angegebenem Initialwert
- Wichtigste Methoden:
  - `void acquire()`
    - entspricht P-Operation
  - `void release()`
    - entspricht V-Operation

Schnittstelle `Lock`

- Wichtigste Methoden:
  - `void lock()`
    - sperren (entspricht P bei binärem Semaphor)
  - `void unlock()`
    - freigeben (entspricht V bei binärem Semaphor)
  - `Condition newCondition()`
    - erzeugt neue Bedingungsvariable, die an dieses `Lock`-Objekt gebunden ist
      - beim Warten an der Bedingungsvariable wird dieses `Lock` freigegeben
- Implementierungsklasse: `ReentrantLock`
  - neu erzeugte Sperre ist zunächst frei

Schnittstelle `Condition`

- Wichtigste Methoden:
  - `void await()`
    - *wait*-Operation: warten auf Signalisierung
    - Thread wird blockiert, zur `Condition` gehöriges `Lock` wird freigegeben
    - nach Signalisierung: `await` kehrt erst zurück, wenn `Lock` wieder erfolgreich gesperrt ist
  - `void signal()`
    - Signalisierung eines wartenden Threads (Variante 1a)
  - `void signalAll()`
    - Signalisierung aller wartenden Threads (Variante 2)

Anmerkungen

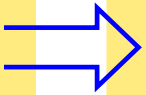
- Mit diesen Objekten lassen sich Monitore nachbilden:

Monitor

```
monitor Bsp

   condition cond;
   procedure foo
   begin
      if ... then
         wait (cond);
      ...
      signal (cond);

   end;
end monitor;
```



Java-Code

```
public class Bsp {
   Lock mutex;      // = new ReentrantLock();
   Condition cond; // = mutex.newCondition();
   public void foo() {
      mutex.lock();
      while(...)
         cond.await();
      ...
      cond.signal();
      mutex.unlock();
   }
}
```

Im Konstruktor

- Ähnliche Konzepte wie `Lock` und `Condition` auch in der POSIX Thread-Schnittstelle (☞ **2.6**, Folie 117)

Anmerkungen ...

- Herstellen der signalisierten Bedingung und Signalisierung muß bei gesperrtem Mutex erfolgen!
- Sonst: Gefahr des *lost wakeup*

Thread 1

```
condition = true;
cond.signal();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

Anmerkungen ...

- Herstellen der signalisierten Bedingung und Signalisierung muß bei gesperrtem Mutex erfolgen!
- Sonst: Gefahr des *lost wakeup*

Thread 1

```
condition = true;
cond.signal();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

Anmerkungen ...

- Herstellen der signalisierten Bedingung und Signalisierung muß bei gesperrtem Mutex erfolgen!
- Sonst: Gefahr des *lost wakeup*

Thread 1

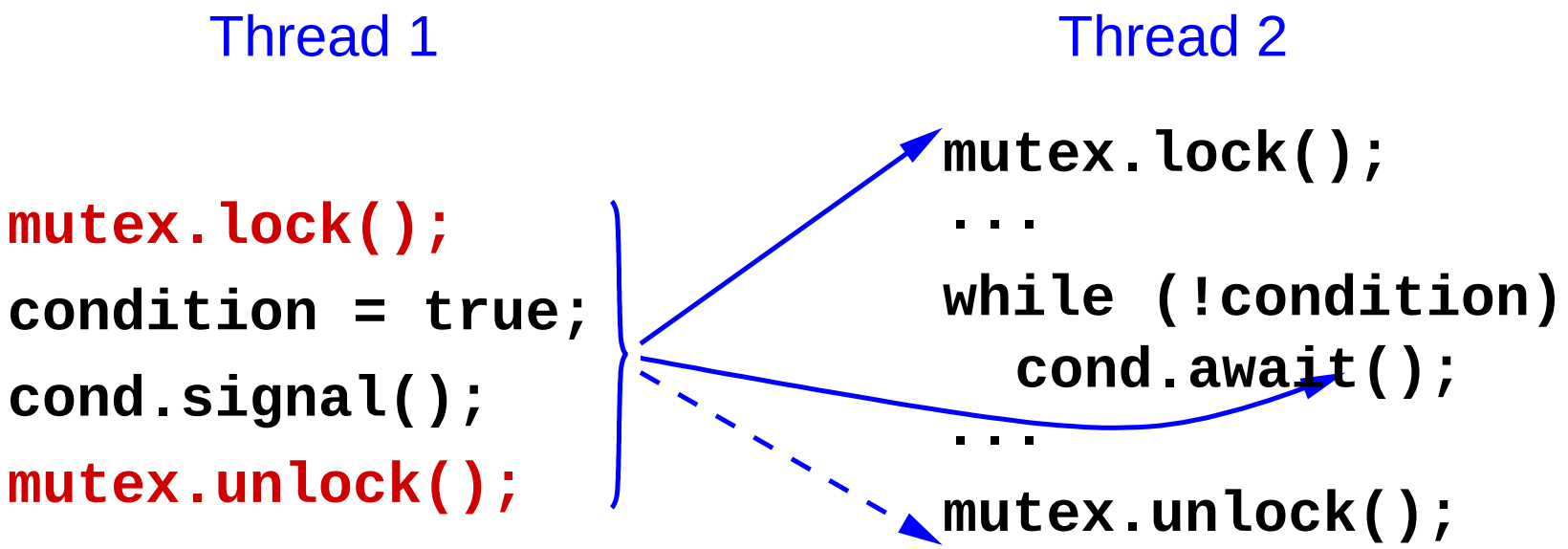
```
mutex.lock();
condition = true;
cond.signal();
mutex.unlock();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

Anmerkungen ...

- Herstellen der signalisierten Bedingung und Signalisierung muß bei gesperrtem Mutex erfolgen!
- Sonst: Gefahr des *lost wakeup*



Synchronisationspaket `BSsync` für die Übungen

- Für die Übungen verwenden wir eine vereinfachte Version der `java.util.concurent`-Klassen
  - weniger Methoden (nur die wichtigsten, siehe vorherige Folien)
  - Optionen zur besseren Fehlersuche
  - `Lock` direkt als Klasse implementiert
    - d.h. `new Lock()` statt `new ReentrantLock()`
- JAR-Archiv `BSsync.jar` und API-Dokumentation im WWW verfügbar
  - über die Vorlesungsseite

Unterstützung der Fehlersuche in `BSsync`

- Konstruktor `Semaphore(int wert, String name)`
  Konstruktor `Lock(String name)`
  Methode `newCondition(String name)` von `Lock`
  - Erzeugung eines Objekts mit gegebenem Namen
- Attribut `public static boolean verbose`
  in den Klassen `Semaphore` und `Lock`
  - schaltet Protokoll aller Operationen auf Semaphoren bzw. Locks und Bedingungsvariablen ein
  - Protokoll benutzt obige Namen
- Erlaubt Verfolgung des Programmablaufs
  - z.B. bei Verklemmungen

Betriebssysteme I

WS 2019/2020

28.11.2019

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 5. Dezember 2019

Beispiel: Erzeuger/Verbraucher-Problem

```
public class ErzVerb {
  private Lock mutex;             // Wechsels. Ausschluß
  private Condition nonfull;      // Warten bei vollem Puffer
  private Condition nonempty;     // Warten bei leerem Puffer
  private int count;              // Zählt belegte Pufferplätze

  Buffer buffer;

  public ErzVerb(int size) {
    buffer = new Buffer(size);
    mutex = new Lock("mutex");                          // Lock erzeugen
    nonfull = mutex.newCondition("nonfull");    // Bedingungsvar.
    nonempty = mutex.newCondition("nonempty");// erzeugen
    count = 0;
  }
```

Beispiel: Erzeuger/Verbraucher-Problem ...

```
public void Insert(int item) {
  mutex.lock();                          // Mutex sperren
  while (count == buffer.getSize())      // Puffer voll?
    nonfull.await();                     //   ja: warten...
  buffer.insertItem(item);               // Item eintragen
  count++;
  nonempty.signal();                     // ggf. Thread wecken
  mutex.unlock();                        // Mutex freigeben
}
```

Beispiel: Erzeuger/Verbraucher-Problem ...

```
public int Remove() {
  int result;
  mutex.lock();                          // Mutex sperren
  while (count == 0)                     // Puffer leer?
    nonempty.await();                    //   ja: warten...
  result = buffer.removeItem();          // Item entfernen
  count--;
  nonfull.signal();                      // ggf. Thread wecken
  mutex.unlock();                        // Mutex freigeben
  return result;
}
```

Beispiel: Erzeuger/Verbraucher-Problem ...

```
  public static void main(String argv[]) {
    ErzVerb ev = new ErzVerb(5);
    Lock.verbose = true;                    // Protokoll anschalten
    Producer prod = new Producer(ev);
    Consumer cons = new Consumer(ev);
    prod.start();
    cons.start();
  }
}
```

➥ Vollständiger Code im WWW (Vorlesungsseite)!

Betriebssysteme I

WS 2019/2020

3 Threadinteraktion

3.2 Kommunikation

Inhalt:

- Einführung
- Elementare Kommunikationsmodelle
- Adressierung
- Varianten und Erweiterungen

- Tanenbaum 2.3.8, 8.2.3, 8.2.4
- Stallings 5.6, 6.7, 13.3
- Nehmer/Sturm 7

Methoden zur Kommunikation

- Speicherbasierte Kommunikation
  - über gemeinsamen Speicher (s. Erzeuger/Verbraucher-Problem)
  - i.d.R. zwischen Threads desselben Prozesses
  - gemeinsamer Speicher auch zwischen Prozessen möglich
  - Synchronisation muß explizit programmiert werden
- Nachrichtenbasierte Kommunikation
  - Senden / Empfangen von Nachrichten (über das BS)
  - i.d.R. zwischen Threads verschiedener Prozesse
  - auch über Rechnergrenzen hinweg möglich
  - implizite Synchronisation

Nachrichtenbasierte Kommunikation

- Nachrichtenaustausch durch zwei Primitive:
  - `send(Ziel,Nachricht)` – Versenden einer Nachricht
  - `receive(Quelle,Nachricht)` – Empfang einer Nachricht
    - oft: spezieller Parameterwert für beliebige Quelle
      evtl. Quelle auch als Rückgabewert
- Implizite Synchronisation:
  - Empfang einer Nachricht erst **nach** dem Senden möglich
  - `receive` blockiert, bis Nachricht vorhanden ist
    - manchmal zusätzlich auch nichtblockierende `receive`-Operationen; ermöglicht *Polling*

Beispiel: Erzeuger/Verbraucher-Kommunikation mit Nachrichten

- Typisch: BS puffert Nachrichten bis zum Empfang
  - Puffergröße wird vom BS bestimmt (meist konfigurierbar)
  - falls Puffer voll ist: Sender wird in `send()`-Operation blockiert (Flußkontrolle, ☞ Rechnernetze I)

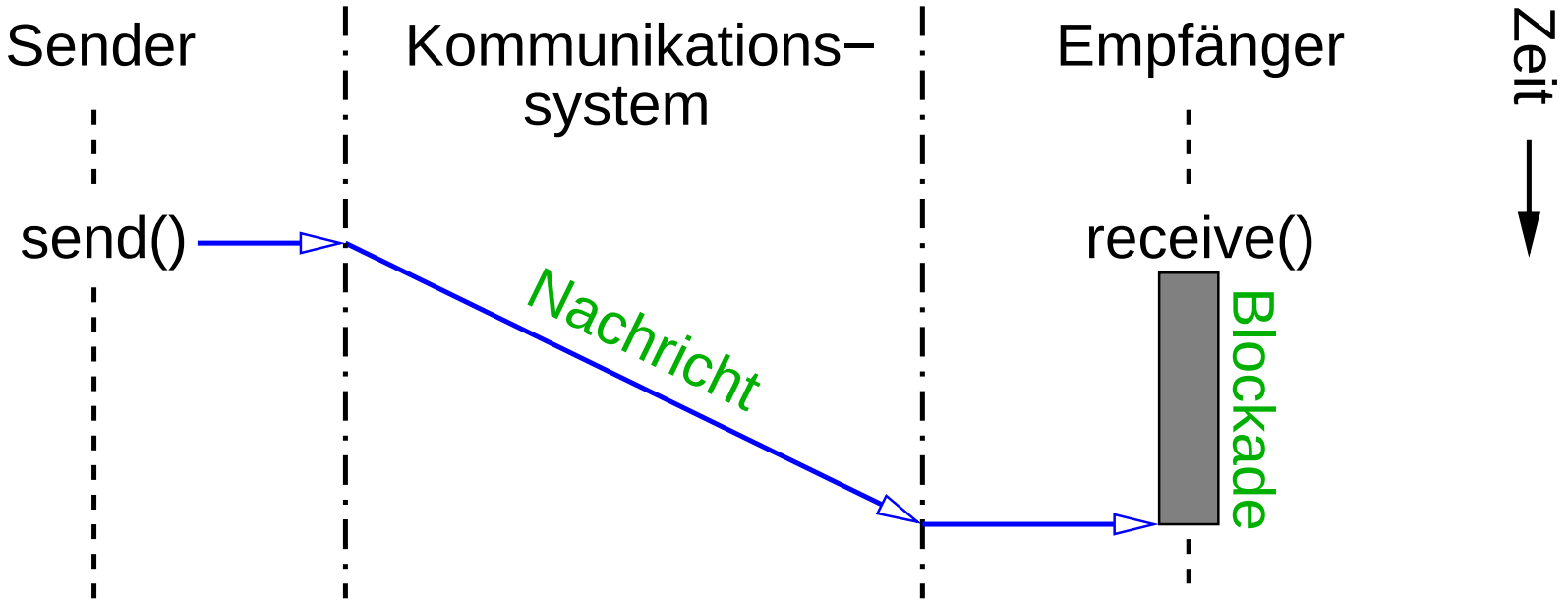
```
void producer() {
  int item;
  Message m;
  while (true) {
    item = produce_item();
    build_message(&m, item);
    send(consumer, m);
  }
}
```

```
void consumer() {
  int item;
  Message m;
  while(true) {
    receive(producer, &m);
    item = extract_item(m);
    consume_item(item);
  }
}
```

Klassifikation (nach Nehmer/Sturm)

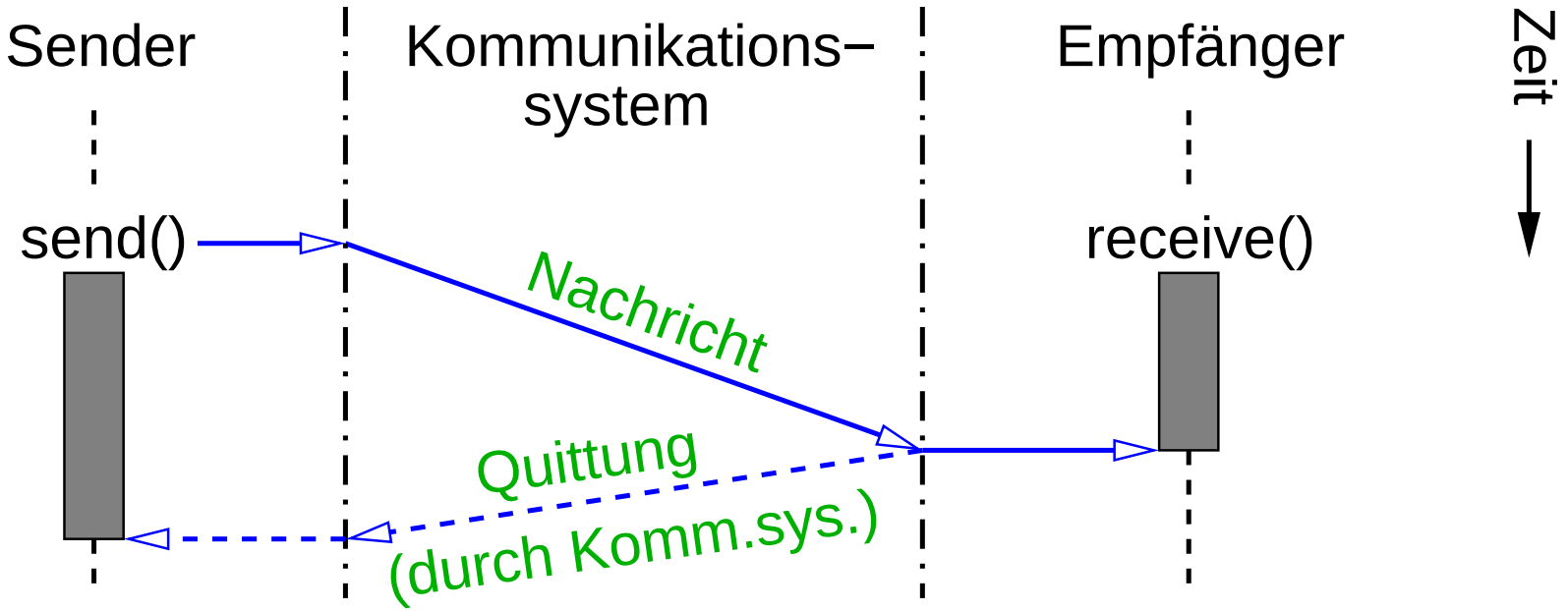
- Zeitliche Kopplung der Kommunikationspartner:
  - synchrone vs. asynchrone Kommunikation
  - auch: blockierende vs. nicht-blockierende Kommunikation
  - wird der Sender blockiert, bis der Empfänger die Nachricht empfangen hat?
- Muster des Informationsflusses:
  - Meldung vs. Auftrag
  - Einweg-Nachricht oder Auftragserteilung mit Ergebnis?

Asynchrone Meldung



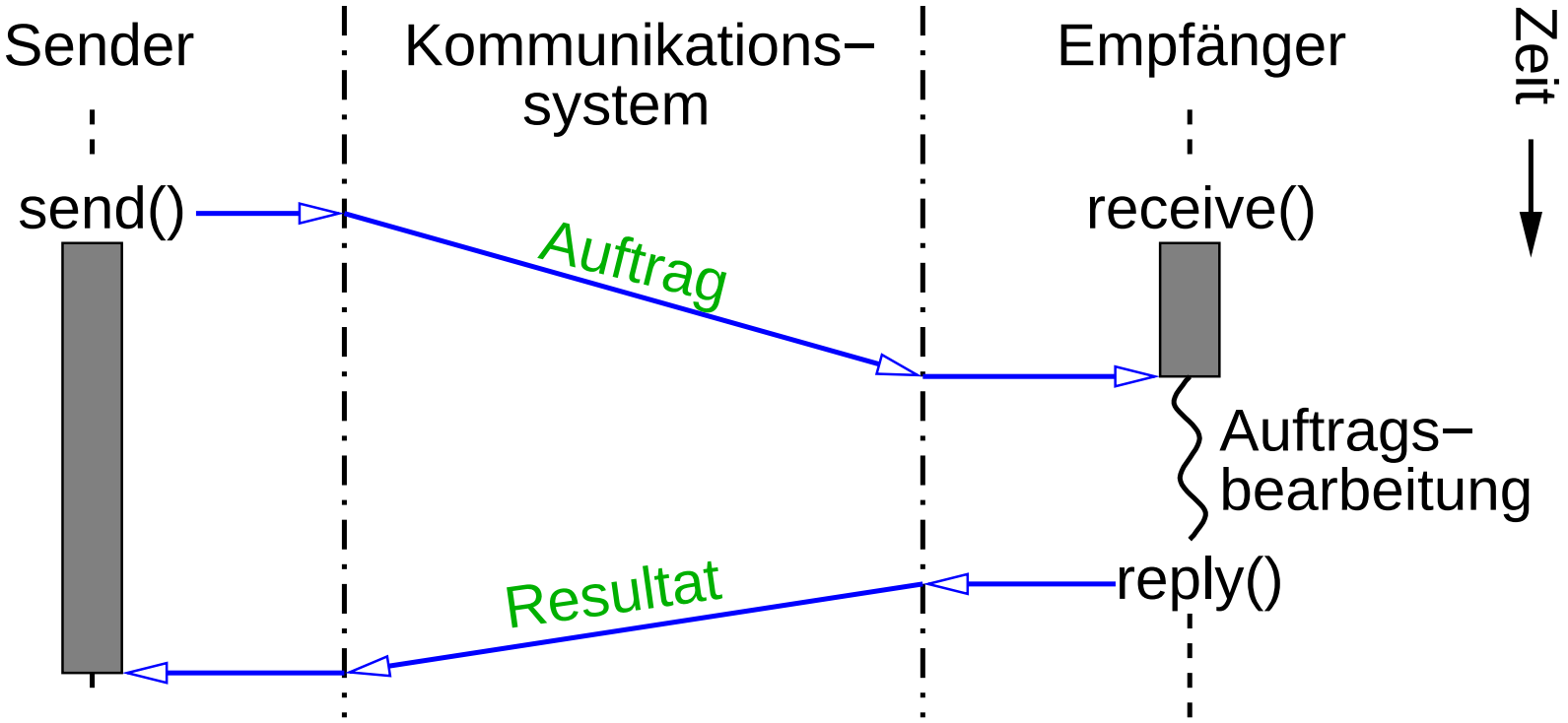
- Sender wird nicht mit Empfänger synchronisiert
- Kommunikationssystem (BS) muß Nachricht ggf. puffern, falls Empfänger (noch) nicht auf Nachricht wartet

Synchrone Meldung



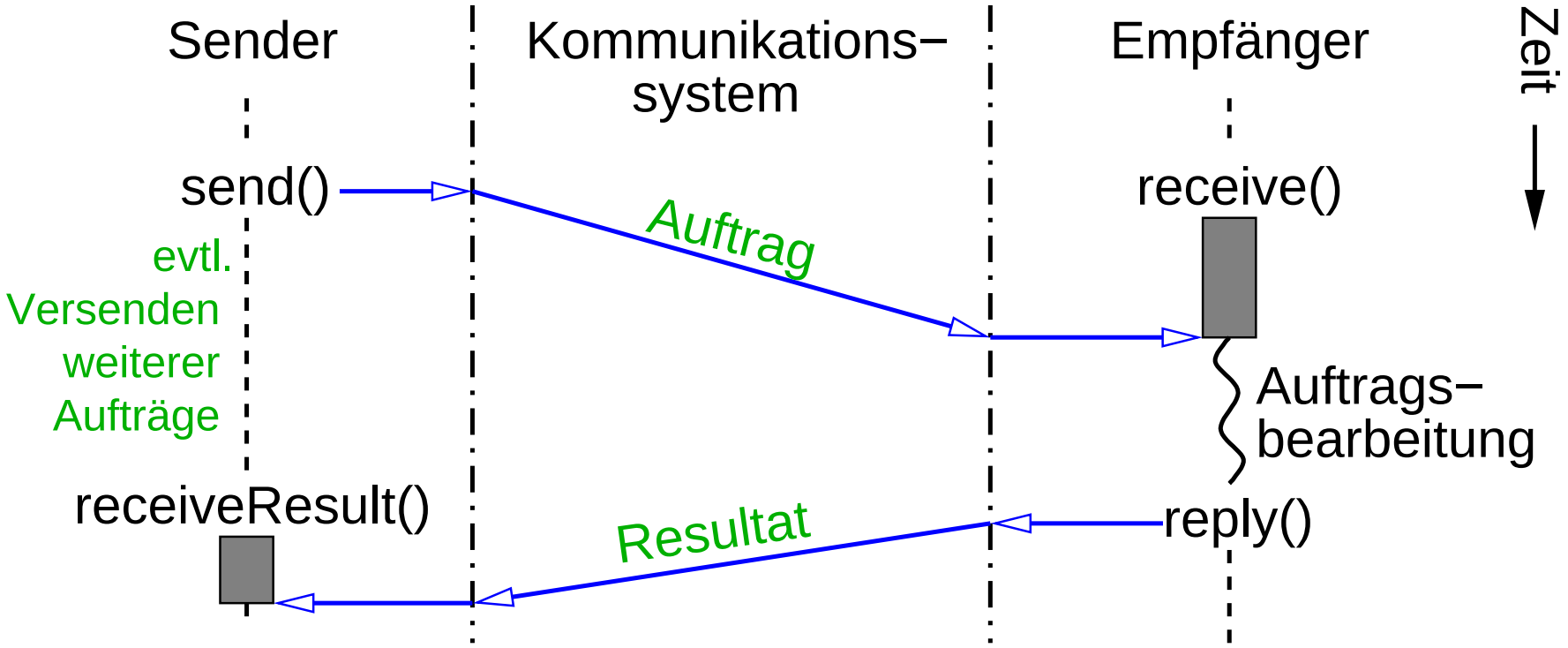
- Sender wird blockiert, bis Nachricht empfangen ist
- **Rendezvous**-Technik
- Keine Pufferung erforderlich

Synchroner Auftrag



- Empfänger sendet Resultat zurück
- Sender wird blockiert, bis Resultat vorliegt

Asynchroner Auftrag



- Sender kann mehrere Aufträge gleichzeitig erteilen
  - Parallelverarbeitung möglich

Beispiel: Ressourcen-Pool

- Server verwaltet exklusive Ressource
  - stellt Dienste `Acquire`, `Release` und `Use` zur Verfügung
  - Kommunikationsmodell: synchrone Aufträge

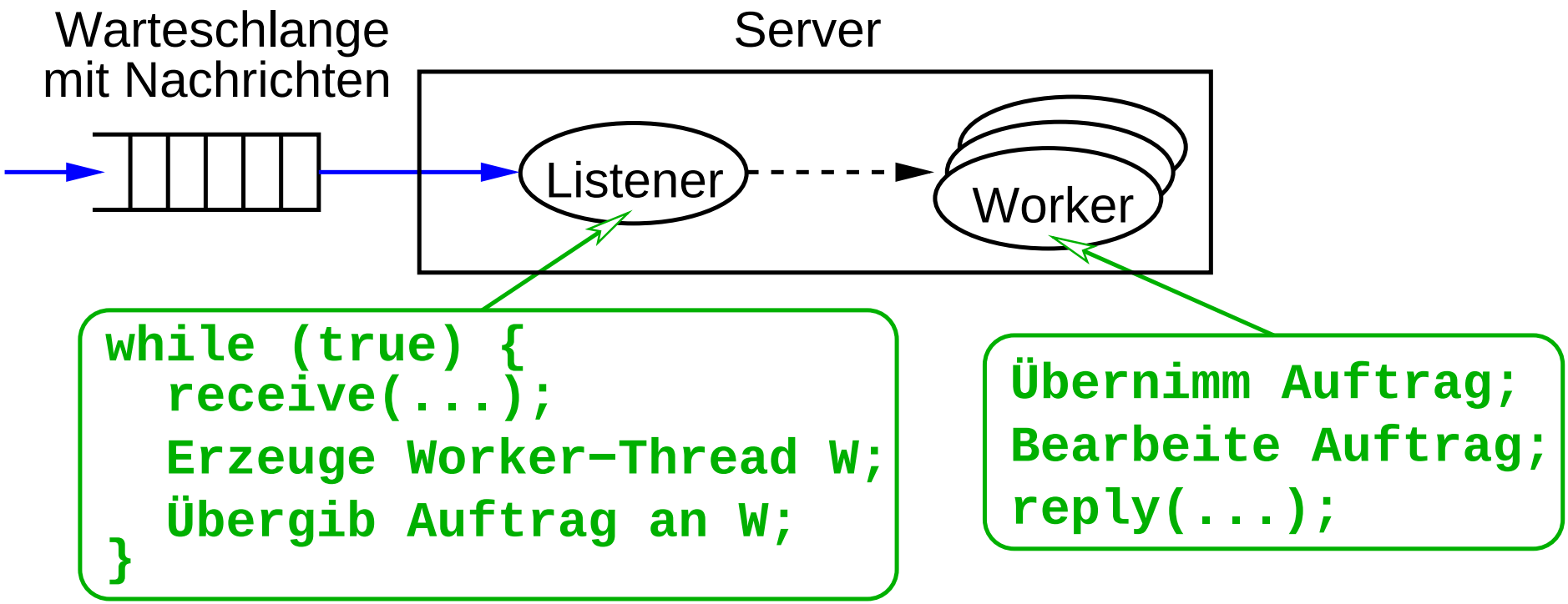
- Clients senden Aufträge nach folgendem Muster:
  - ```
    send(Server, ("Acquire", Parameter));
      send(Server, ("Use", Parameter));
      send(Server, ("Use", Parameter));
      ...
    send(Server, ("Release", Parameter));
    ```
    ```

Beispiel: Ressourcen-Pool, einfacher sequentieller Server

```
while (true) {
  receive(...);
  switch (Dienst) {
  case "Acquire":
    if (Ressource frei) Resultat = RessourceId;
    else                Resultat = "Ressource belegt";
    break;
  case "Use": ...; break;
  case "Release": Ressource freigeben; break;
  }
  reply(Resultat);
}
```

➥ Bei belegten Resourcen: aktives Warten nötig

Beispiel: Ressourcen-Pool, *multi-threaded* Server



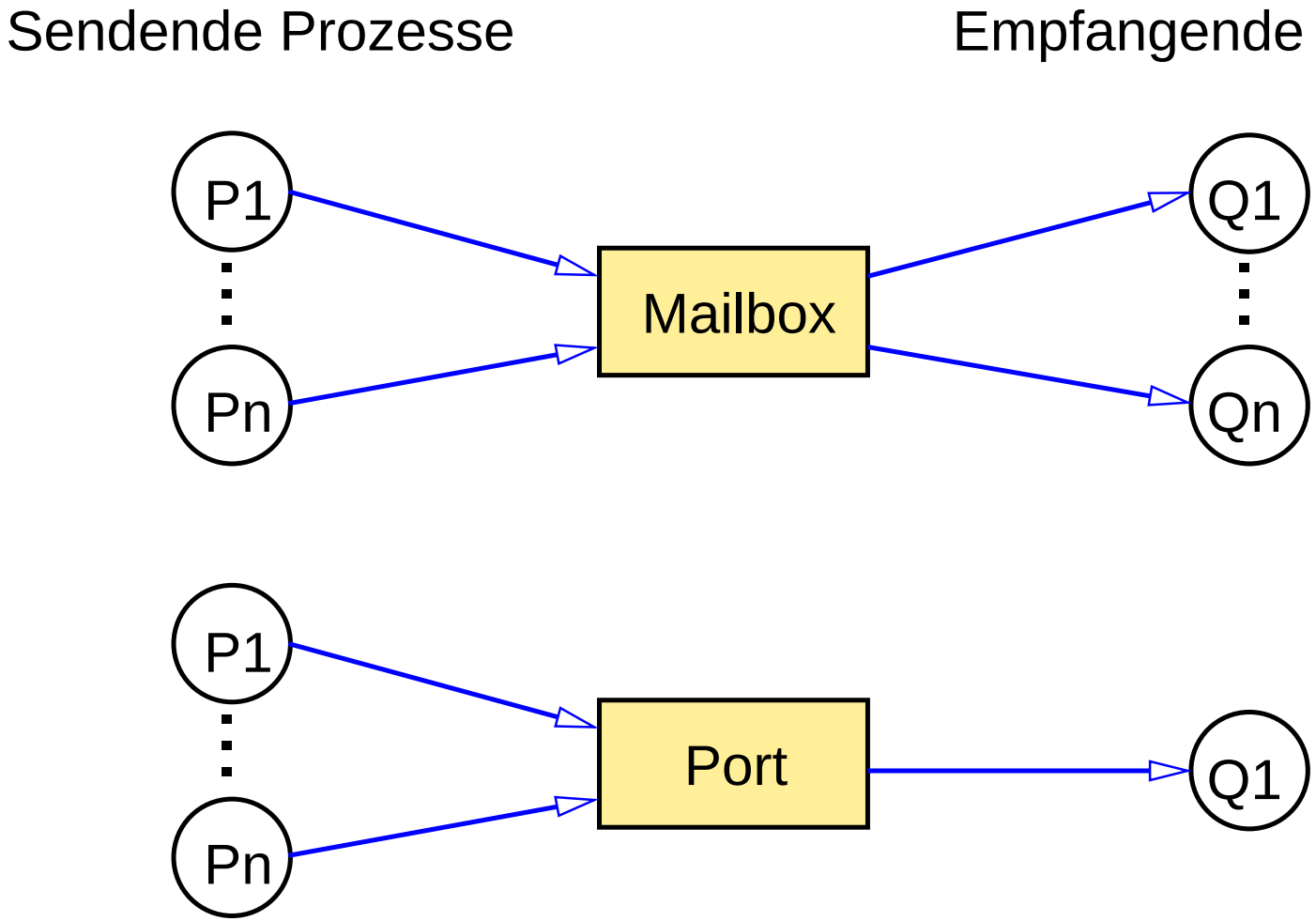
- Listener nimmt Aufträge entgegen,
  erzeugt für jeden Auftrag einen Worker zur Bearbeitung
  - Worker kann blockieren, wenn Ressource belegt

Wie werden Sender bzw. Empfänger festgelegt?

- Direkte Adressierung
  - Kennung des jeweiligen Prozesses
- Indirekte Adressierung
  - Nachrichten werden an Warteschlangen-Objekt (**Mailbox**) gesendet und von dort gelesen
  - Vorteil: höhere Flexibilität
    - Mailbox kann von mehreren Prozessen gelesen werden
      - **Port**: Mailbox mit nur einem möglichen Empfänger-Prozess
    - Empfänger kann mehrere Mailboxen / Ports besitzen

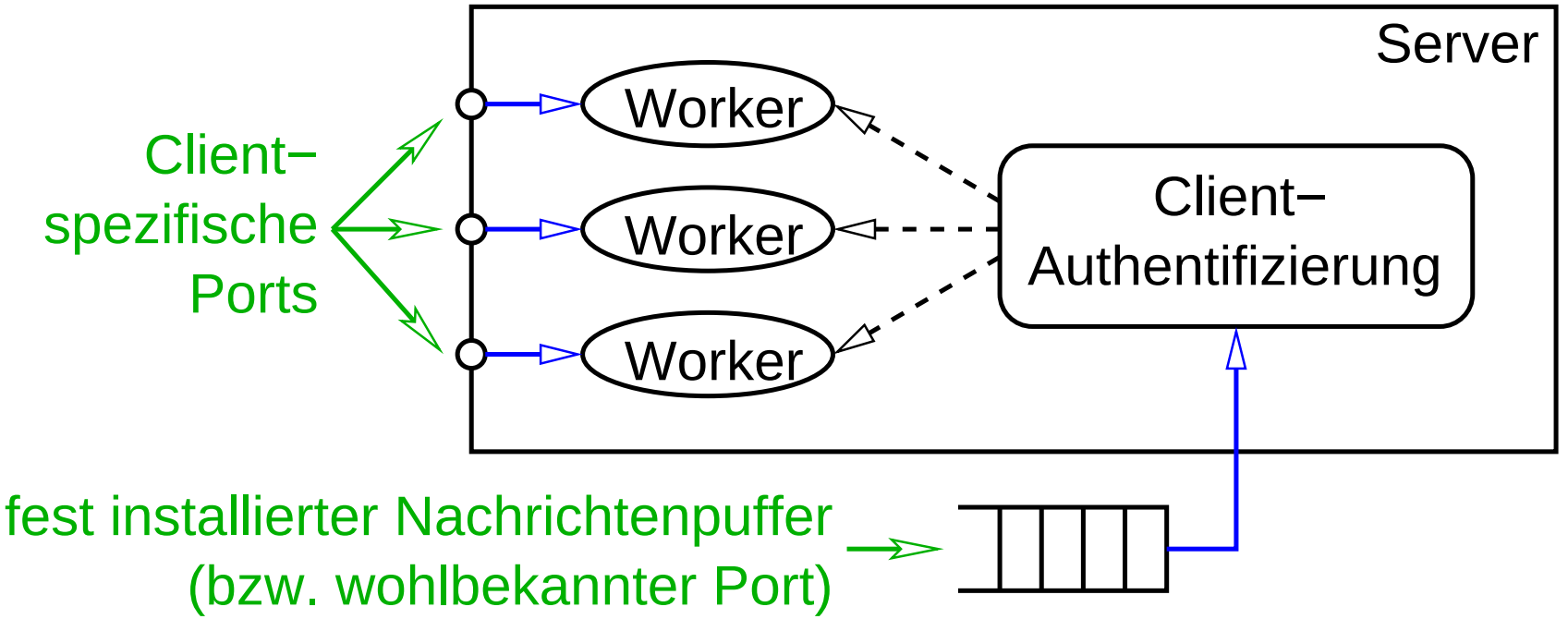
(Anm.: Für Sender und Empfänger werden nur Prozesse betrachtet)

Mailbox und Port



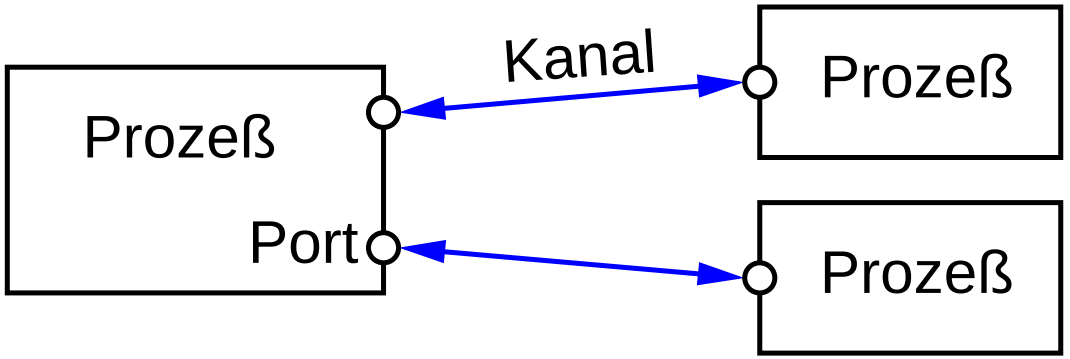
Anwendung von Ports: z.B. selektiver Nachrichtenempfang

- Server kann nach Anmeldung eines Clients für jeden Client einen eigenen Port erzeugen
  - jeder Port kann durch einen eigenen Worker-Prozeß (oder Thread) bedient werden



Kanäle

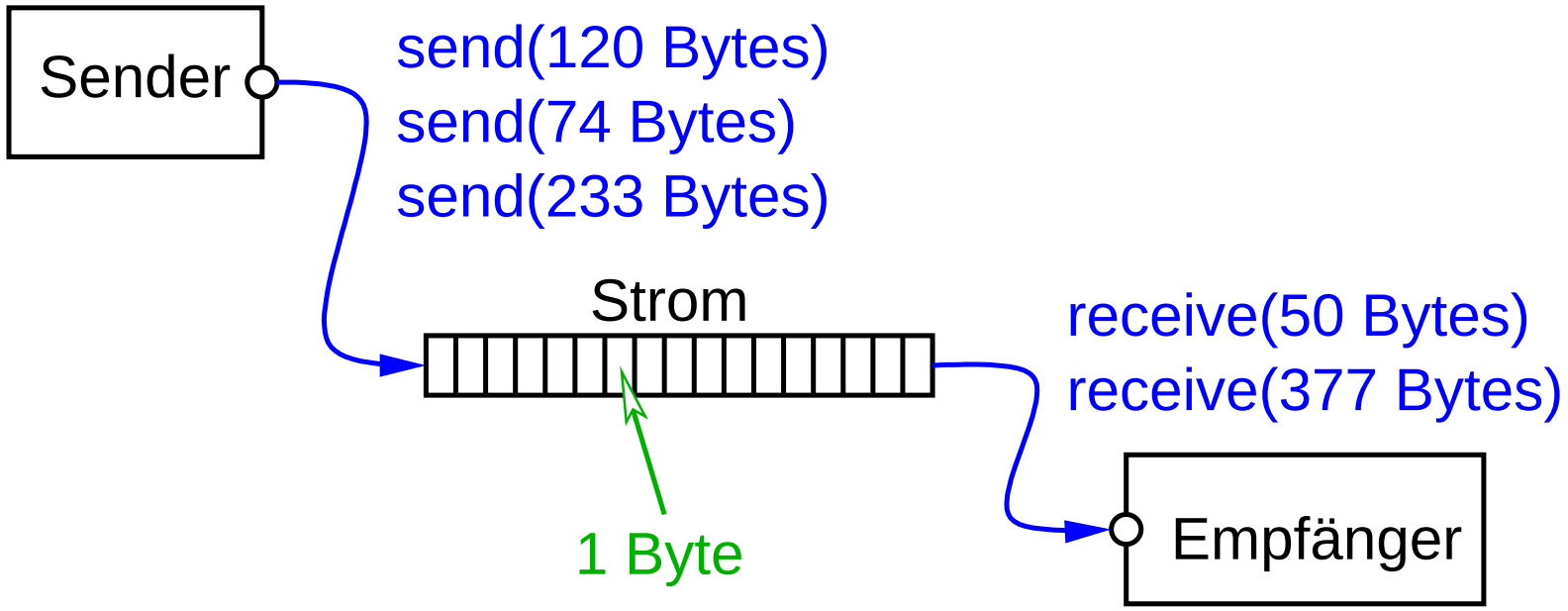
- Bisher: verbindungslose Kommunikation
  - wer Adresse eines Ports kennt, kann senden
- Kanal: logische Verbindung zw. zwei Kommunikationspartnern



  - expliziter Auf- und Abbau einer Verbindung (zw. zwei Ports)
  - meist bidirektional: Ports für Senden und Empfangen
  - garantierte Nachrichtenreihenfolge (FIFO)
  - Beispiel: TCP/IP-Verbindung

Ströme (*Streams*)

- Kanal zur Übertragung von Sequenzen von Zeichen (Bytes)
- Keine Nachrichtengrenzen oder Längenbeschränkungen
- Beispiele: TCP/IP-Verbindung, UNIX *Pipes*, Java *Streams*



Ströme in POSIX: *Pipes*

- *Pipe*: Unidirektionaler Strom
- Schreiben von Daten in die Pipe:
  - `write(int pipe_desc, char *msg, int msg_len)`
    - `pipe_desc`: Dateideskriptor
  - bei vollem Puffer wird Schreiber blockiert
- Lesen aus der Pipe:
  - `int read(int pipe_desc, char *buff, int max_len)`
    - `max_len`: Größe des Empfangspuffers `buff`
    - Rückgabewert: Länge der tatsächlich gelesenen Daten
  - bei leerem Puffer wird Leser blockiert

Ströme in POSIX: Erzeugen einer *Pipe*

- Unbenannte (*unnamed*) *Pipe*:
  - ist zunächst nur im erzeugenden Prozeß bekannt
  - Dateideskriptoren werden an Kindprozesse vererbt
- Beispielcode:

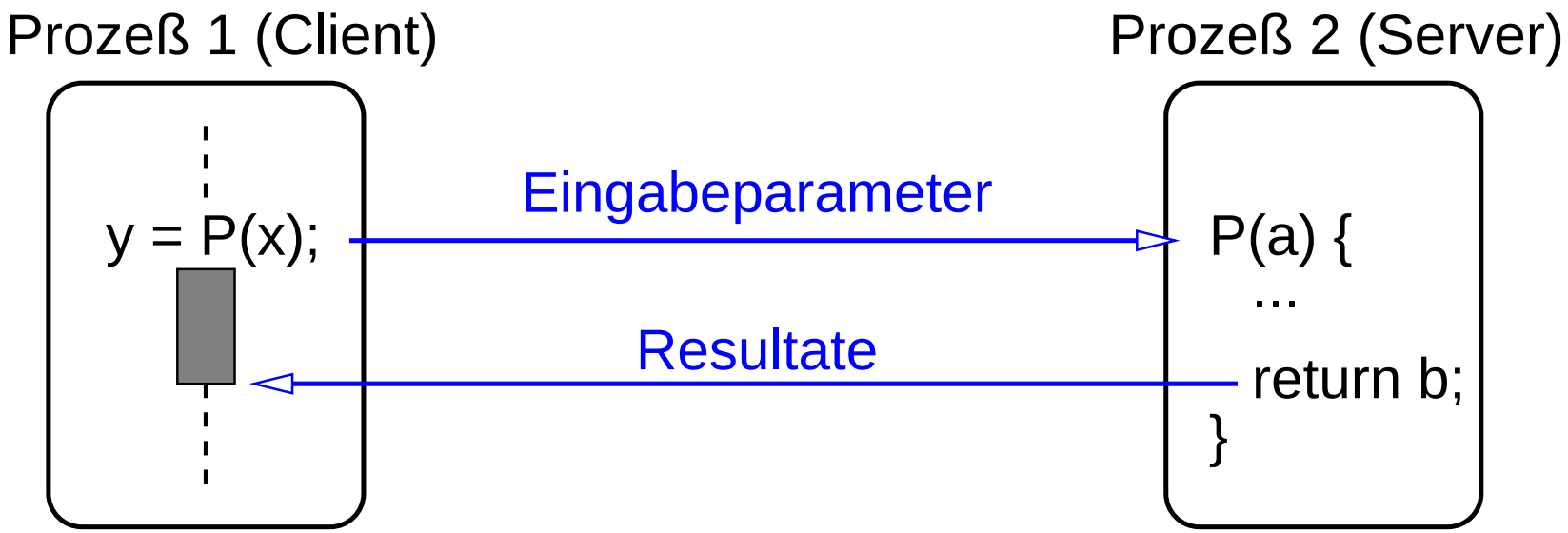
```
int pipe_ends[2];    // Dateideskriptoren der Pipe-Enden
pipe(pipe_ends);     // Erzeugen der Pipe
if (fork() != 0) {
   // Vaterprozeß
   write(pipe_ends[1], data, ...);
} else {
   // Kindprozeß (erbt Dateideskriptoren)
   read(pipe_ends[0], data, ...);
}
```

Ströme in POSIX: Erzeugen einer *Pipe* ...

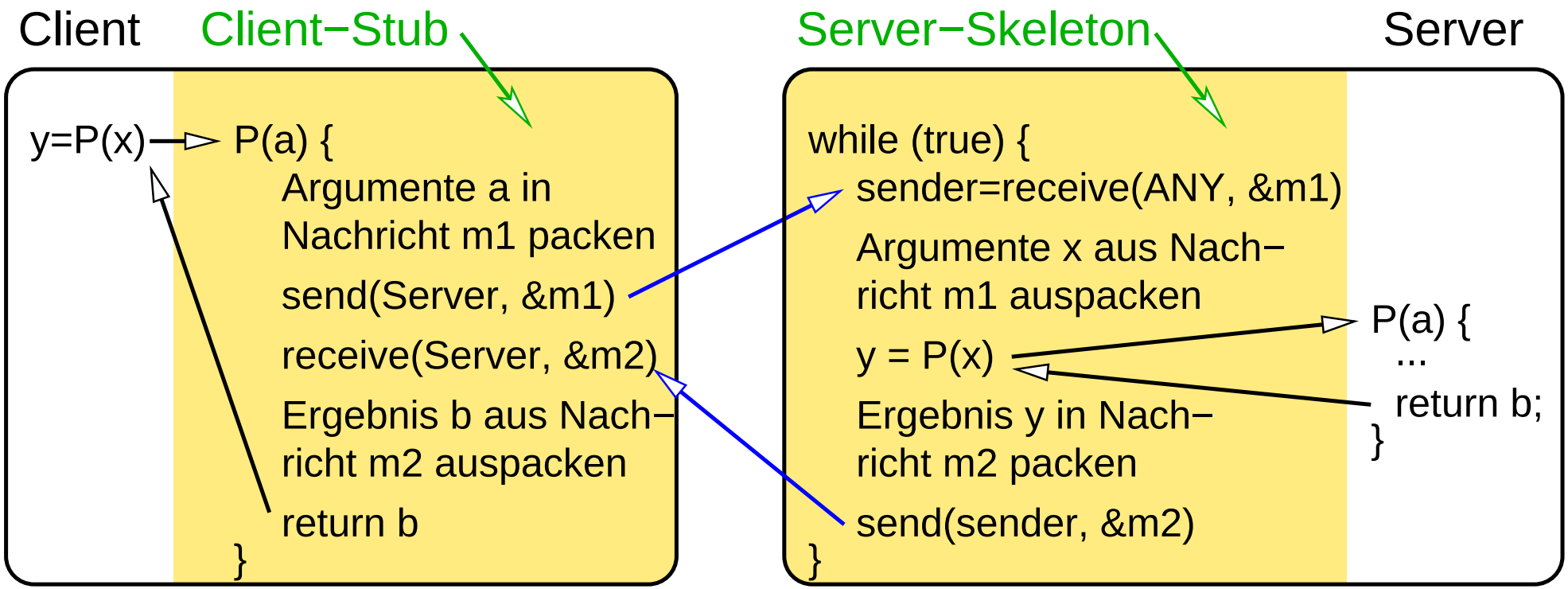
- Benannte (*named*) *Pipe*:
  - ist als spezielle „Datei“ im Dateisystem sichtbar
  - Zugriff erfolgt exakt wie auf normale Datei:
    - Systemaufrufe `open`, `close` zum Öffnen und Schließen
    - Zugriff über `read` und `write`
- Erzeugung:
  - Systemaufruf `mkfifo(char *name, mode_t mode)`
  - in Linux auch Shell-Kommando: `mkfifo <name>`
- Löschen durch normale Dateisystem-Operationen

Remote Procedure Call (RPC)

- Idee: Vereinfachung der Realisierung von synchronen Aufträgen
  - RPC: Aufruf einer Prozedur (Methode) in einem anderen Prozeß



Realisierung eines RPC



- Client-Stub und Server-Skeleton werden i.d.R. aus Schnittstellen-beschreibung generiert: **RPC-Compiler**

RPC: Diskussion

- Client muß sich zunächst an den richtigen Server binden
  - Namensdienste verwalten Adressen der Server
- Danach: RPC syntaktisch exakt wie lokaler Prozeduraufruf
- Semantische Probleme:
  - *Call-by-Reference* Parameterübergabe sehr problematisch
  - Kein Zugriff auf globale Variablen möglich
  - Nur streng getypte Schnittstellen verwendbar
    - Datentyp muß zum Ein- und Auspacken (***Marshaling***) genau bekannt sein
  - Behandlung von Fehlern in der Kommunikation?
- Beispiel: Java RMI

Signale

- Erlauben sofortige Reaktion des Empfängers auf eine Nachricht (asynchrones Empfangen)
- Asynchrone Unterbrechung des Empfänger-Prozesses
  - „Software-Interrupt“: BS-Abstraktion für Interrupts
- Operationen (abstrakt):
  - `Receive(Signalnummer, HandlerAdresse)`
  - `Signal(Empfänger, Signalnummer, Parameter)`
- Ideal: Erzeugung eines neuen Threads beim Eintreffen des Signals, der Signal-Handler abarbeitet
- Historisch: Signal wird durch unterbrochenen Thread behandelt

Betriebssysteme I

WS 2019/2020

05.12.2019

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

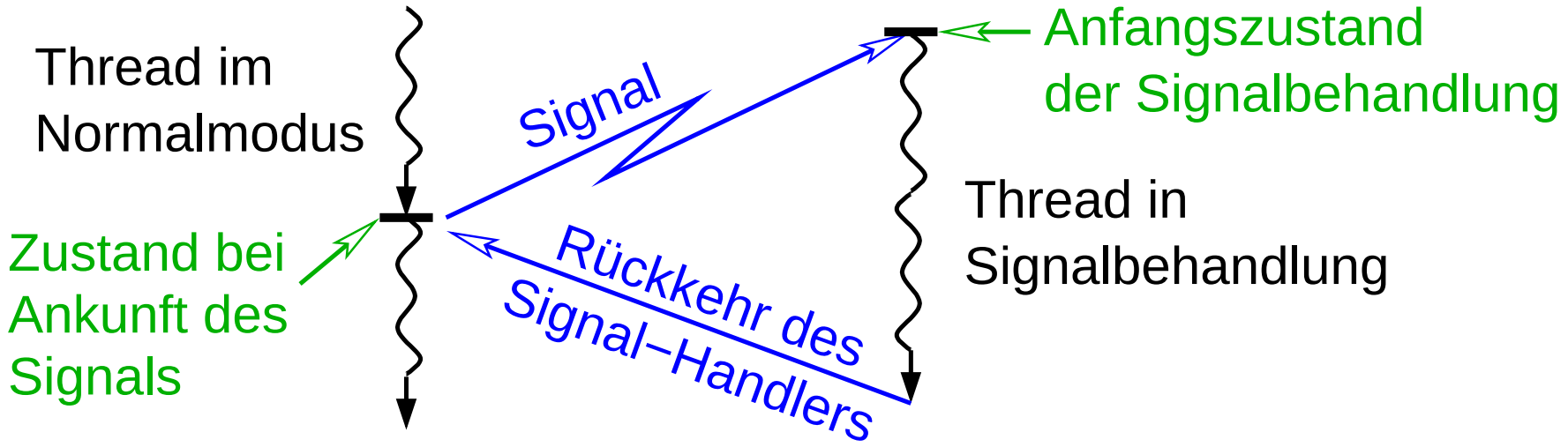
Stand: 5. Dezember 2019

Zur Klausur

- Termine:
  - Montag, **17.02.2020**, 12:00 Uhr s.t., PB-C 101
  - nächster Termin nach dem SoSe 2020
- Dauer: 60 Minuten, ohne Hilfsmittel

- **Anmeldefrist: 22.01.2020!**

Behandlung eines Signals im unterbrochenen Thread

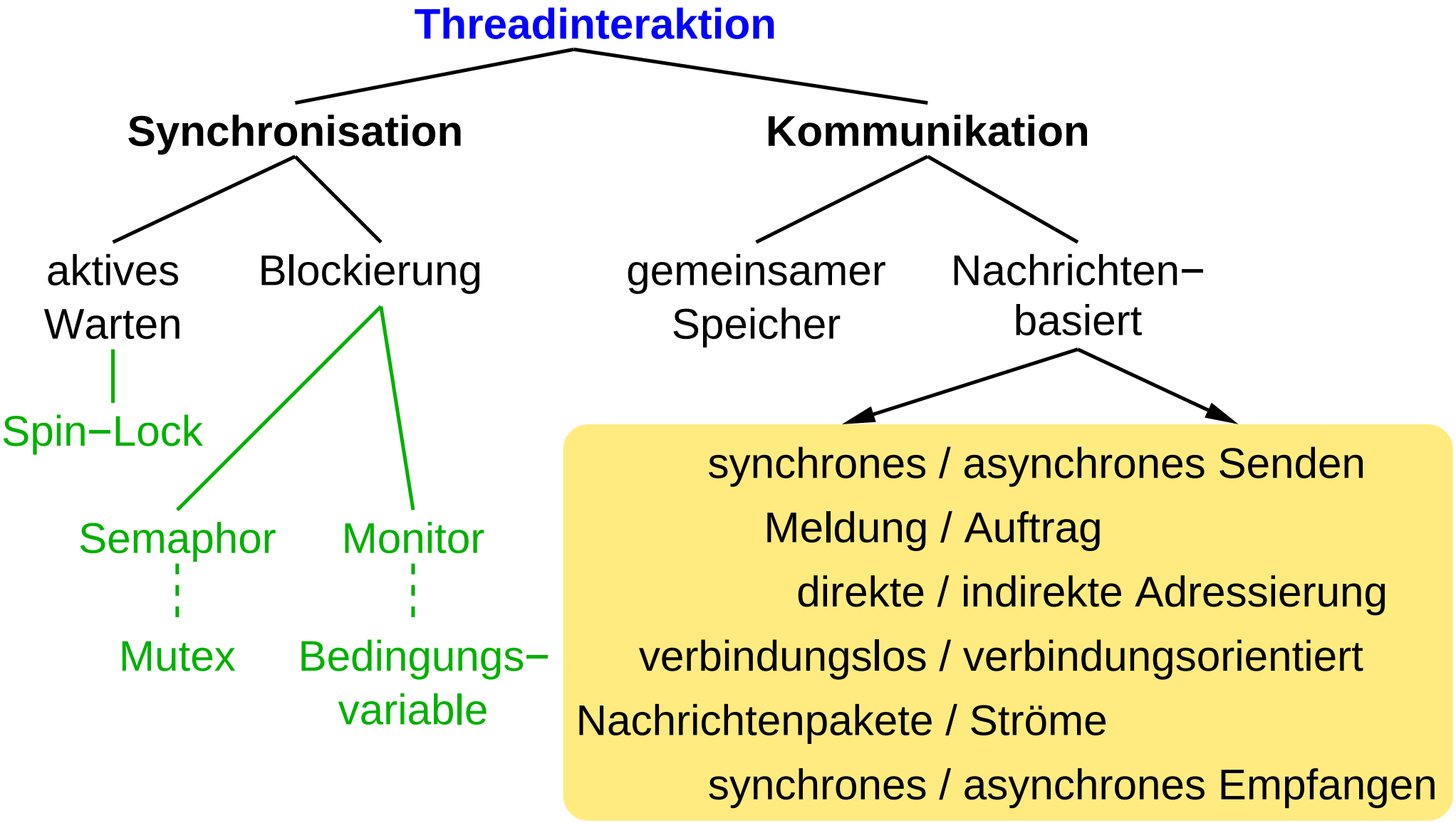


- BS sichert Threadzustand im Thread-Kontrollblock und stellt Zustand für Signalbehandlung her
- Kein wechselseitiger Ausschluß (z.B. Semaphore) möglich
  - ggf. müssen Signale gesperrt werden

Signale in POSIX

- Senden eines Signals an einen Prozeß (nicht Thread!):
  - `kill(pid_t pid, int signo)`
- Für jeden Signal-Typ ist eine Default-Aktion definiert, z.B.:
  - SIGINT: Terminieren des Prozesses (`^C`)
  - SIGKILL: Terminieren des Prozesses (nicht änderbar!)
  - SIGCHLD: Ignorieren (Kind-Prozeß wurde beendet)
  - SIGSTOP: Stoppen (Suspendieren) des Prozesses (`^Z`)
  - SIGCONT: Weiterführen des Prozesses (falls gestoppt)
- Prozeß kann Default-Aktionen ändern und eigene Signal-Handler definieren
- Handler wird von beliebigem Thread des Prozesses ausgeführt

3.3 Zusammenfassung / Wiederholung



3.3 Zusammenfassung / Wiederholung ...

- Threadinteraktion
  - Synchronisation (Sperr- und Reihenfolgesynchronisation)
  - Kommunikation
- Synchronisation
  - wechselseitiger Ausschluß
    - nur jeweils ein Thread darf im kritischen Abschnitt sein
    - kritischer Abschnitt: Zugriff auf gemeinsame Ressourcen
  - Lösungsansätze:
    - Sperren der Interrupts (nur im BS, Einprozessorsysteme)
    - Sperrvariable: Peterson-Algorithmus
    - mit Hardware-Unterstützung: *Read-Modify-Write*
    - Nachteil: Aktives Warten (Effizienz, Verklemmungsgefahr)

3.3 Zusammenfassung / Wiederholung ...

- Semaphor
  - besteht aus Zähler und Threadwarteschlange
    - `P()`: herunterzählen, ggf. blockieren
    - `V()`: hochzählen, ggf. blockierten Thread wecken
    - Atomare Operationen (im BS realisiert)
  - wechselseitiger Ausschluß:

Thread 0

```
P(Mutex);
// kritischer Abschnitt
V(Mutex);
```

Thread 1

```
P(Mutex);
// kritischer Abschnitt
V(Mutex);
```

  - auch für Reihenfolgesynchronisation nutzbar
    - Beispiel: Erzeuger/Verbraucher-Problem

3.3 Zusammenfassung / Wiederholung ...

- Monitor
  - Modul mit Daten, Prozeduren, Initialisierung
  - Datenkapselung
  - Prozeduren stehen unter wechselseitigem Ausschluß
  - Bedingungsvariable zur Synchronisation
    - `wait()` und `signal()`
  - Varianten bei der Signalisierung:
    - einen / alle wartenden Threads wecken?
    - erhält geweckter Thread sofot den Monitor?
      - falls nicht: Bedingung nach Rückkehr aus `wait()` erneut prüfen!

3.3 Zusammenfassung / Wiederholung ...

- Synchronisation in Java:
  - Klassen mit `synchronized` Methoden
    - wechselseitiger Ausschluß der Methoden (pro Objekt)
  - `wait()`, `notify()`, `notifyAll()`
    - genau eine (implizite) Bedingungsvariable pro Objekt
  - JDK 1.5: Semaphore, *Locks* und Bedingungsvariablen
    - *Locks* und Bedingungsvariable erlauben die genaue Nachbildung des Monitorkonzepts
      - *Locks* für wechselseitigen Ausschluß der Methoden
      - Bedingungsvariablen sind fest an *Lock* gebunden
      - mehrere Bedingungsvariablen pro Objekt möglich

3.3 Zusammenfassung / Wiederholung ...

- Nachrichtenbasierte Kommunikation
  - Primitive `send` und `receive`
  - synchrones / asynchrones Senden, Meldung / Auftrag
  - Mailboxen und Ports: indirekte Adressierung
  - Kanäle: logische Verbindung zwischen zwei Ports
    - Spezialfall: Ströme zur Übertragung von Sequenzen von Zeichen (z.B. POSIX Pipes)
  - RPC:
    - synchroner Auftrag, syntaktisch wie Prozeduraufruf
    - generierte Client- und Server-Stubs für *Marshaling*
  - Signale
    - asynchrone Benachrichtigung eines Prozesses
    - Unterbrechung eines Threads $\Rightarrow$ führt Signal-Hander aus