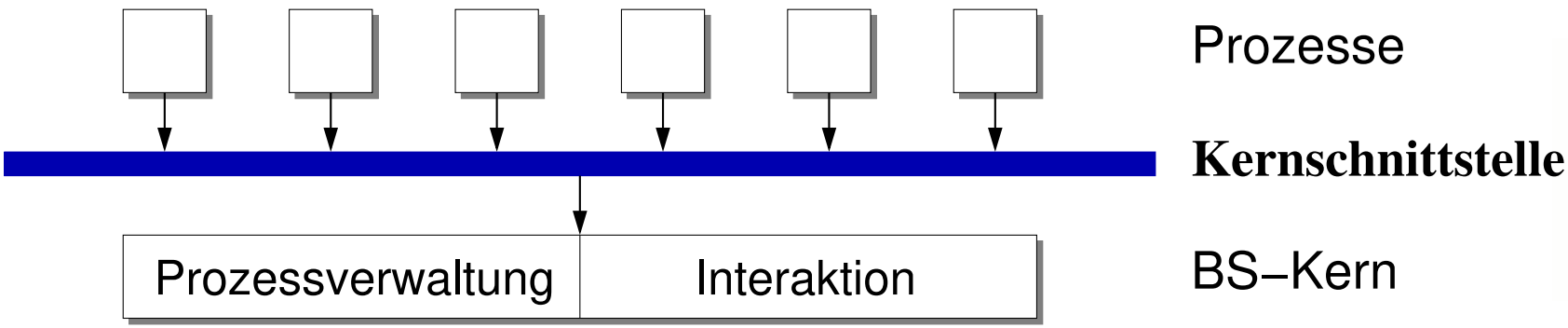
Kapitel 4
Prozessinteraktion

4.1 Interaktionsarten

Prozesse als Teile eines Betriebssystems, die zusammen auf eine gemeinsame Aufgabe hin arbeiten, müssen nicht nur Daten bearbeiten, sondern auch:

- sich aufrufen (bzw. beauftragen)
- aufeinander warten
- sich abstimmen
- . . . d. h. sie müssen **interagieren**.



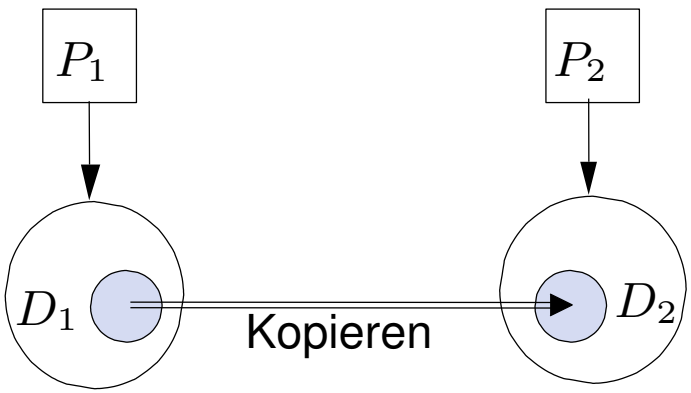
Die Prozessinteraktion bildet (neben der Prozessverwaltung) den zweiten wesentlichen Aufgabenbereich eines Betriebssystemkerns.

Begriffe: Kommunikation vs. Kooperation

Prozessinteraktion besitzt zwei Aspekte:

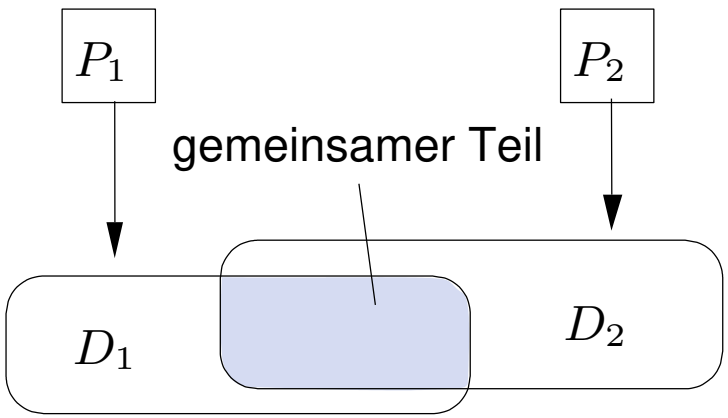
- Zeitlicher Aspekt: Synchronisation
- Funktionaler Aspekt: Informationsaustausch, zwei Formen:
  - Kommunikation
  - Kooperation

Kommunikation
(= expliziter Datentransport)



(gerichtete Beziehung)

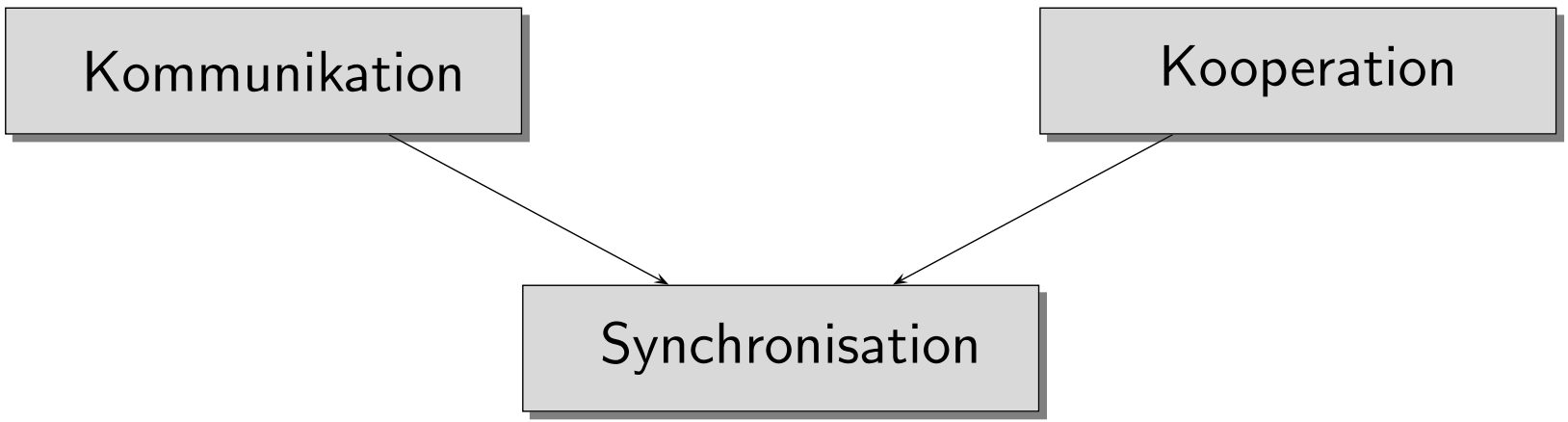
Kooperation
(= Zugriff auf gemeinsame Daten)



(symmetrische Beziehung)

Interaktionsformen: Zusammenhang

- Von den drei Formen der Interaktion ist Synchronisation die elementarste: Sowohl Kommunikation als auch Kooperation benötigen i.d.R. eine zeitliche Abstimmung zwischen den Interaktionspartnern



- Wir werden daher zunächst die *Synchronisation* behandeln

4.2 Synchronisation

Vorbemerkung (Verwendung des Kernausschlusses):

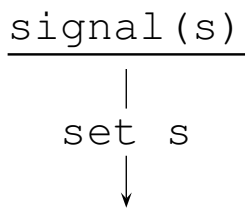
- Synchronisationsvorgänge kennen wir bereits aus der Diskussion des Kernausschlusses (Kap. 3).
- Wir müssen uns nicht darum kümmern, wie Interaktionsoperationen im Kern auf gemeinsame Daten zugreifen, weil sie als Kernoperationen unter gegenseitigem Ausschluss stehen.
- Im Folgenden geht es um die Synchronisation von Prozessen außerhalb des Kerns, die allerdings auf *unteilbare Kernoperationen* zurückgreifen können.

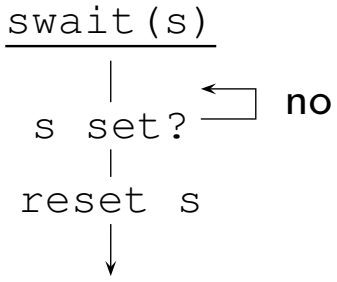
4.2.1 Signalisierung

- Ist eine spezielle Art der Synchronisation.
- Bei der Signalisierung soll eine *Reihenfolgebeziehung* hergestellt werden, z. B.: Ein Abschnitt A in einem Prozess P_1 soll **vor** einem Abschnitt B in einem Prozess P_2 ausgeführt werden.
- Dazu bietet der Kern die Operationen `signal` und `swait` an, die eine gemeinsame binäre Variable `s` benutzen – diese gibt an, ob P_1 den Abschnitt A beendet hat
- Zu Laufzeit wird P_2 evtl. auf das Signal von P_1 warten und erst dann den Abschnitt B ausführen

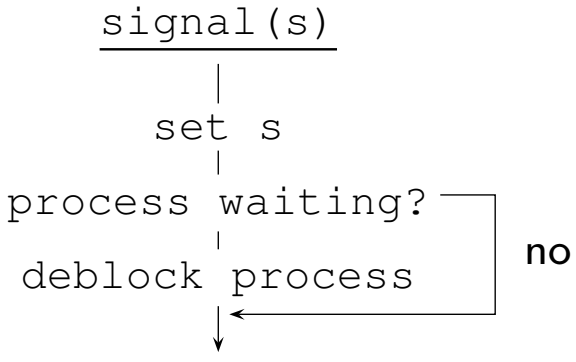
Signalisierung: Implementierung

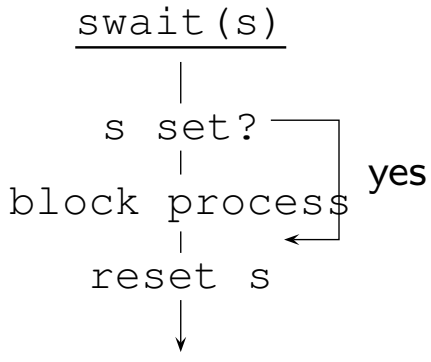
- In ihrer einfachsten Form können die Operationen folgendermaßen realisiert werden:





- Nachteil: Dies bedeutet aktives Warten (*busy waiting*) an der Signalisierungsvariablen `s`, d. h. der Prozessor bleibt belegt
- Ist die Wartezeit zu lange, sollte die CPU freigegeben werden (Signalisieren mit Wartezustand, nur ein Prozess wird aufgeweckt):





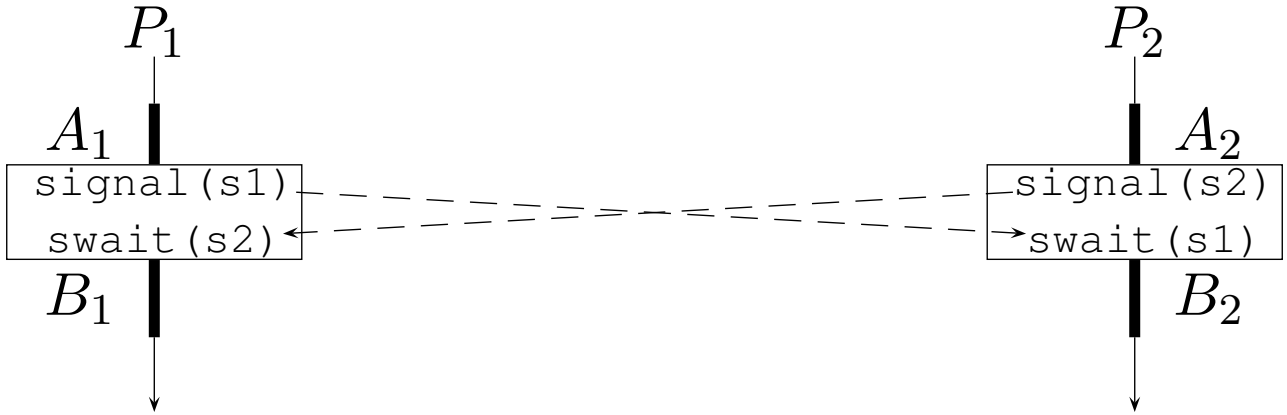
Beispielimplementierung der Signalisierung (Java)

```
class Signal {
  private boolean set = false;  // Eigentliche Signalvariable

  public synchronized void signal() {
    set = true;
    notify();  // if a process is waiting deblock it
  }

  public synchronized void swait() {
    if (set == false)
      wait();      // wait for signal, leave the method
    set = false;
  }
}
```

- Das Schlüsselwort `synchronized` bewirkt in Java den gegenseitigen Ausschluss aller damit gekennzeichneten Methoden eines Objekts
- Die Java-Methode `wait()` blockiert die Prozessausführung, bis der Prozess deblockiert wird (nicht zu verwechseln mit `swait`im Kern!)
- Die Java-Methode `notify()` deblockiert einen wartenden Prozess

Wechselseitige Synchronisierung

- Ein symmetrischer Einsatz der Operationen bewirkt, dass sowohl A_1 vor B_2 als auch A_2 vor B_1 ausgeführt werden.



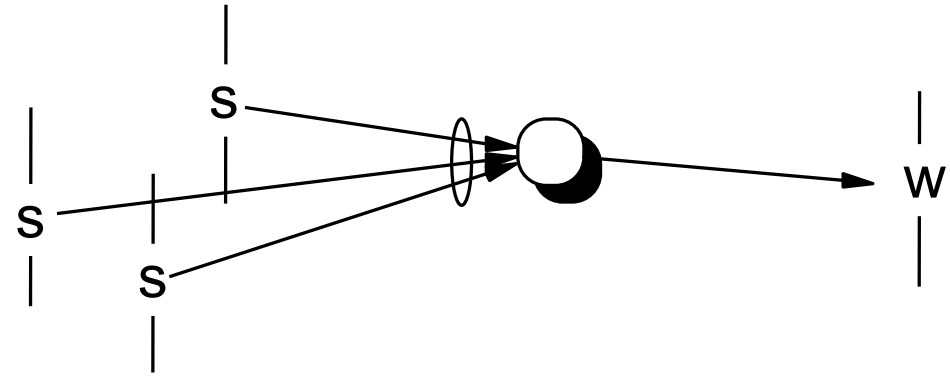
- Die Prozesse P_1 und P_2 **synchronisieren** sich an dieser Stelle (*handshake*, *rendezvous*)
- Wir können das Operationspaar `signal` und `swait` als eine Operation `sync` zusammenfassen:

Beispielimplementierung für Synchronisierung

```
class Synchronisation {
  private boolean set = false;

  public synchronized void sync() {
    if (set == false) { // i am first
      set = true;       // indicate my arrival and ...
      wait();           // wait for my partner
    } else {            // i am second
      set = false;      // reset signal for reuse and ...
      notify();         // deblock my waiting partner
    }
  }
}
```

Gruppensignalisierung und Stauräume

- An Signalisierung können mehr als zwei Prozesse beteiligt sein, beispielsweise:
  - **UND-Signalisieren**: Ein Prozess soll erst weiterlaufen, wenn mehrere Prozesse ein Signal gesetzt haben (UND-Verknüpfung auf Signalisierungsseite)

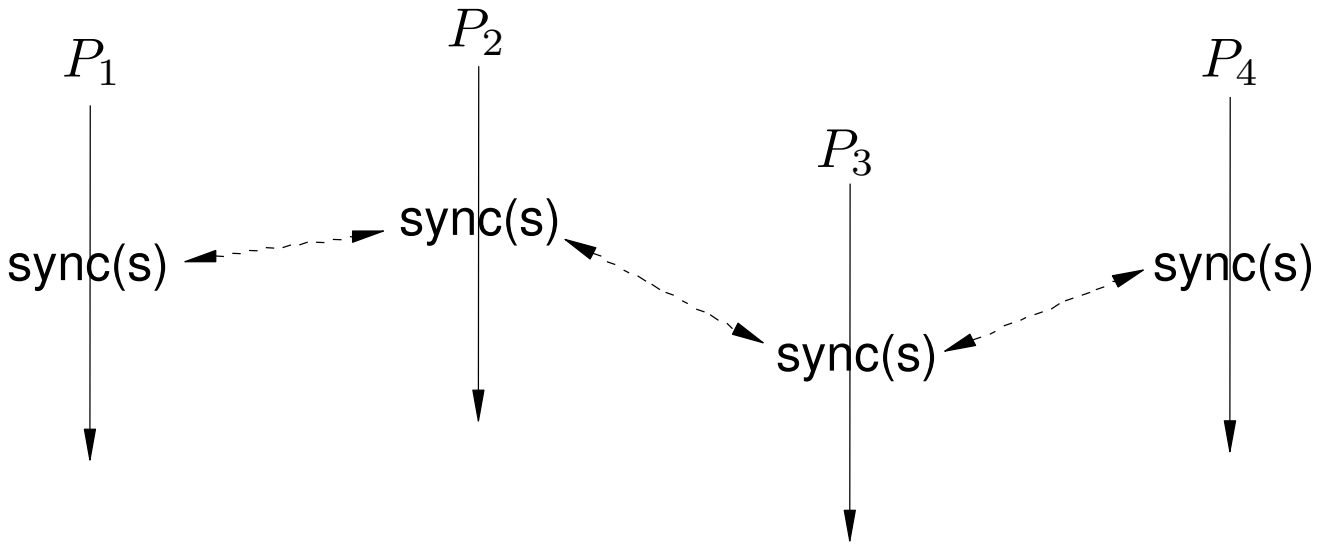


  - **ODER-Warten**: Mehrere Prozesse warten auf ein Signal, dann wird einer von ihnen deblockiert (ODER-Verknüpfung auf Warteseite)

  Sowie alle möglichen Kombinationen der beiden Arten.

- Da jetzt mehrere Signale bzw. mehrere wartende Prozesse anstehen können, muss dafür in der Signalimplementierung entsprechende Kapazität (z. B. mehrere Signalvariablen) vorgesehen werden.

Barrierensynchronisation für Gruppen



- Alle Prozesse synchronisieren sich an einer Stelle.
- Die Prozesse dürfen erst weiterlaufen, wenn alle anderen Prozesse die Synchronisationsstelle erreicht haben.

(Synchronisationsbarriere, Barrierensynchronisation, Gruppenrendezvous)

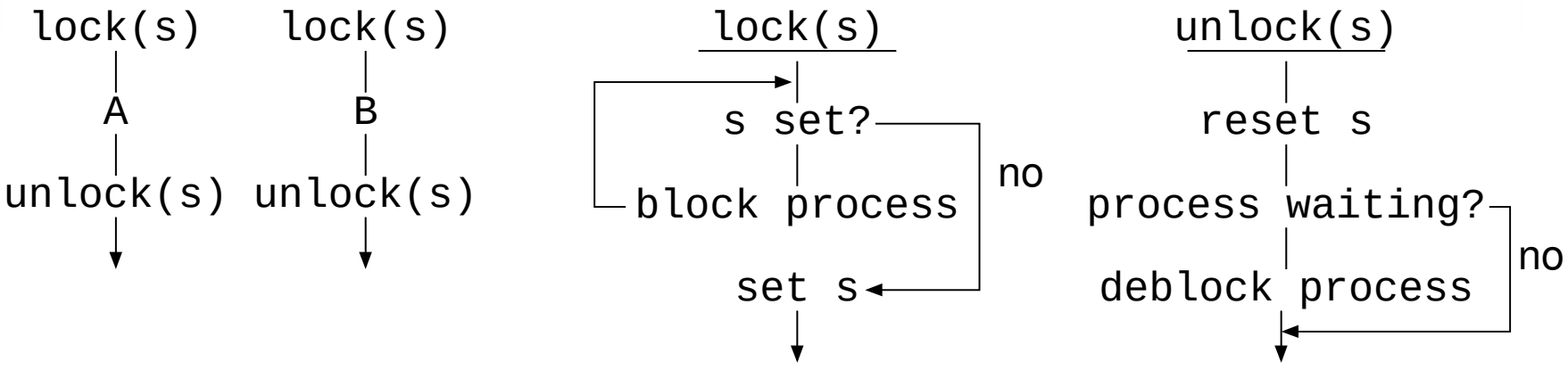
Beispielimplementierung für Barriere

```
class BarrierSync {
  private int number = p;        // number of processes
  private int count = 0;         // number of waiting processes

  public synchronized void sync() {
    count = count+1;
    if (count < number) {        // not everybody here so ...
      wait();                    // wait, leave the method
    } else {
      notifyAll();               // deblock all waiting processes
      count = 0;
    }
  }
}
```

- Das Schlüsselwort `synchronized` bewirkt in Java den gegenseitigen Ausschluss aller damit gekennzeichneten Methoden eines Objekts
- Die Java-Methode `wait()` blockiert die Prozessausführung, bis der Prozess deblockiert wird (nicht zu verwechseln mit `swait` im Kern!)
- Die Java-Methode `notifyAll()` deblockiert alle wartende Prozesse

4.2.2 Sperren

- Eine weitere Klasse von Signalisierungsoperationen sind *Sperren*, die verwendet werden, um *kritische Abschnitte (critical sections)*, z. B. A und B, zu sichern (gegenseitiger Ausschluß, engl. *mutual exclusion*): Es darf keine Überlappung in der Ausführung von A und B stattfinden, d.h. die Ausführungen von A und B *schließen sich gegenseitig aus*.



- Zweckmäßigerweise geben wir diesen Operationen die entsprechenden Namen: *Sperren (lock)* und *Entsperren (unlock)*
- **Anmerkung:** Anders als bei `swait(s)` weiter oben, wird `s` in `lock(s)` in einer Schleife abgefragt (und nicht in **`if .. else ...`**), weil zwischen dem Deblockieren des wartenden Prozesses und seinem Setzen der Sperre ein weiterer Prozess die Sperre setzen könnte.

Kritische Abschnitte: Beispiel

- Fehlerhafte Beispiel-Befehlsfolge bei gleichzeitiger Ausführung von zwei Zuweisungen: `c = c + 1` in P_1 und `c = c - 1` in P_2:

P_1	c	P_2
	5	
c lesen $\rightarrow$ 5	5	
c + 1 $\rightarrow$ 6	5	c lesen $\rightarrow$ 5
c schreiben	$\rightarrow$ 6	c − 1 $\rightarrow$ 4
	$\rightarrow$ 4	c schreiben

- Der Effekt von `c=c+1` geht verloren! Warum? Wie repariert man das? **Ursache:** die beiden Zuweisungen sind kritische Abschnitte, die nicht gleichzeitig ausgeführt werden dürfen

- Ist das Objekt im Kern realisiert, so wird die Sicherung des kritischen Abschnitts durch die Kernsperre implizit vorgenommen.
- Ist das Objekt außerhalb des Kerns realisiert, so muss der kritische Abschnitt explizit gesichert werden, z.B. mit `lock/unlock`
- Wir werden dieses Beispiel später nochmals betrachten

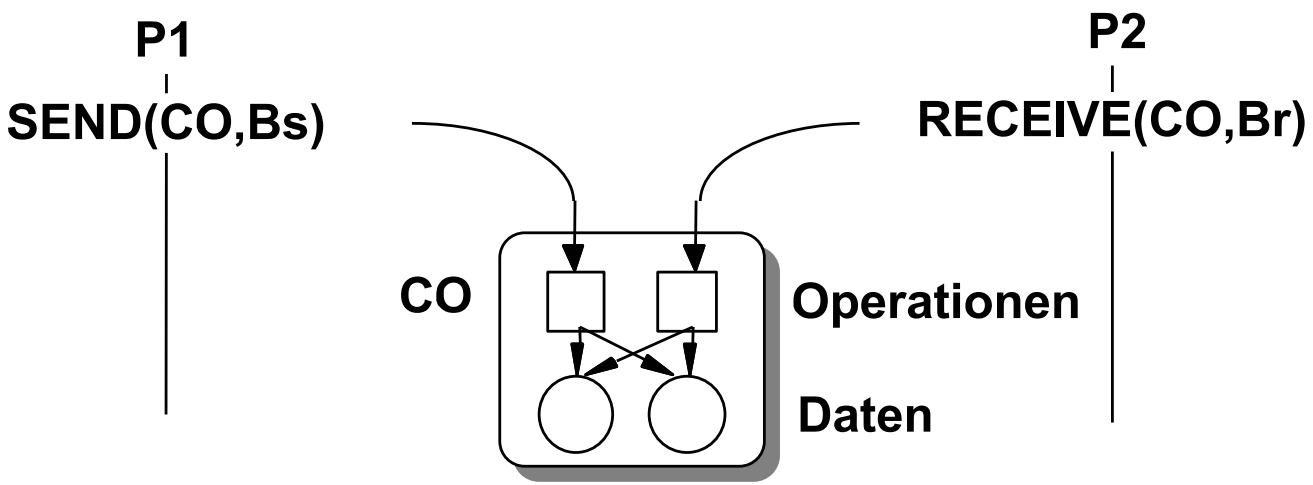
Implementierungsbeispiel Sperre

```
class Lock {
  private boolean locked = false;

  public synchronized void lock() {
    while(locked)
      wait();          // wait for free lock, leave the method
    locked = true;
  }

  public synchronized void unlock() {
    locked = false;   // release lock
    notify();         // wake up waiting process
  }
}
```

4.3 Kommunikation der Prozesse: Kanalkonzept

Kanal ist ein allgemeines Konzept der Kommunikation:

- Ein *Kanal* ist ein Datenobjekt, das die Operationen *Senden (send)* und *Empfangen (receive)* für Prozesse zur Verfügung stellt.
- Parameter von *send* und *receive*:
  - Name des Kanal(objekt)s (CO)
  - Adresse eines Behälters
    - Sender: Adresse der zu verschickenden Nachricht im Prozessadressraum (oder die Nachricht selbst), *buffer send* (Bs).
    - Empfänger: Adresse im Prozessadressraum, wohin die empfangene Nachricht geschrieben werden soll, *buffer receive* (Br).

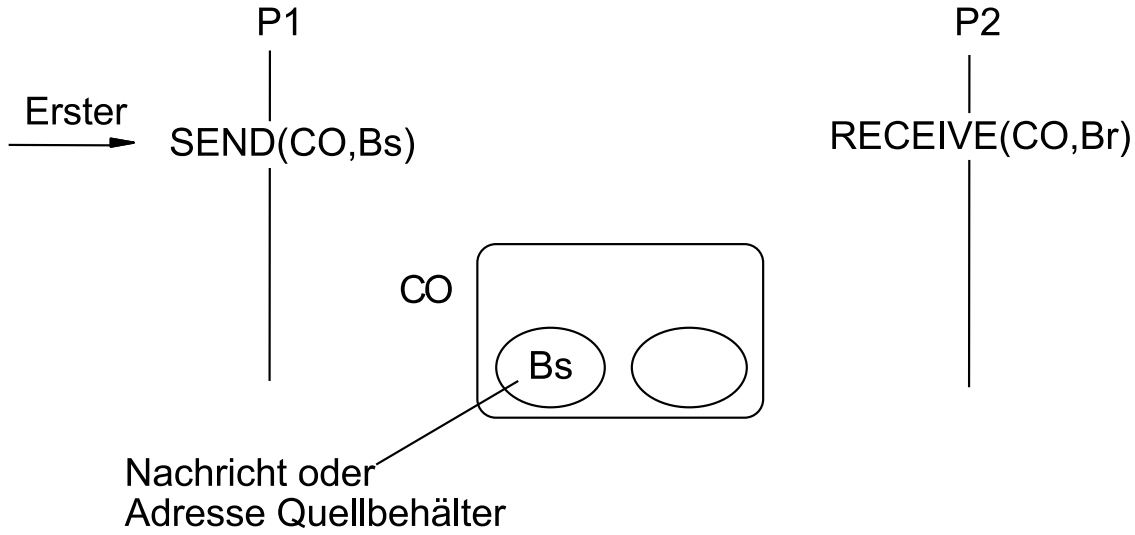


Asynchrones Send-Recv: Zwischenspeicherung

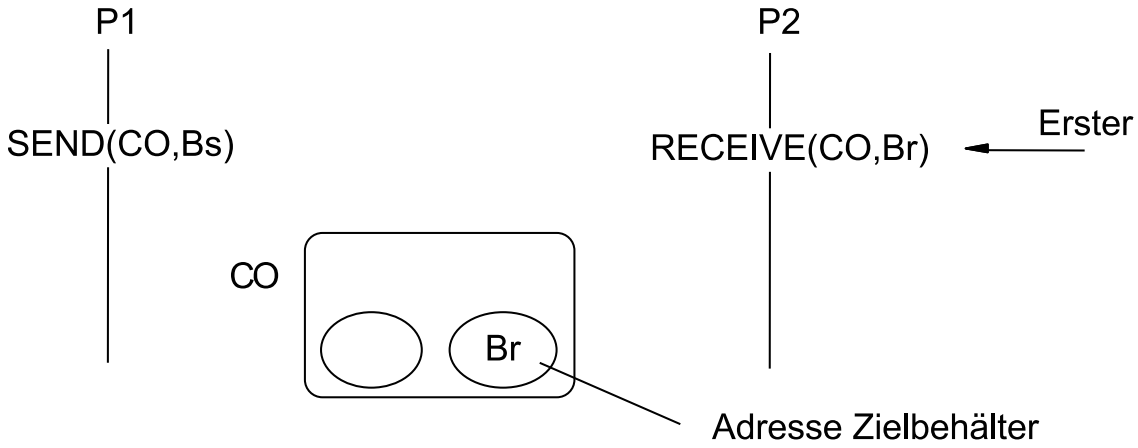
- Da Sender und Empfänger ihre Operationen zu beliebigen Zeitpunkten aufrufen können, sind zwei Fälle zu berücksichtigen:
  1. Erst Senden, dann Empfangen
  2. Erst Empfangen, dann Senden
- Wenn Prozesse in den Operationen nicht aufgehalten (blockiert) werden sollen (**asynchrone Kommunikation**), besteht die Notwendigkeit der Zwischenspeicherung im Kanal.
  - **Sender zuerst**: Die Nachricht bzw. ihre Adresse wird im Kanal abgelegt und bei einem nachfolgenden Empfangen abgeholt
  - **Empfänger zuerst**: Die Adresse des Zielpuffers wird abgelegt; bei einem nachfolgenden Senden wird die Nachricht dorthin kopiert
- Der Kanal muss eine Variable zur Aufnahme dieser Daten vorsehen.

Zeitverhältnisse bei SEND/RECV

- Sender zuerst:



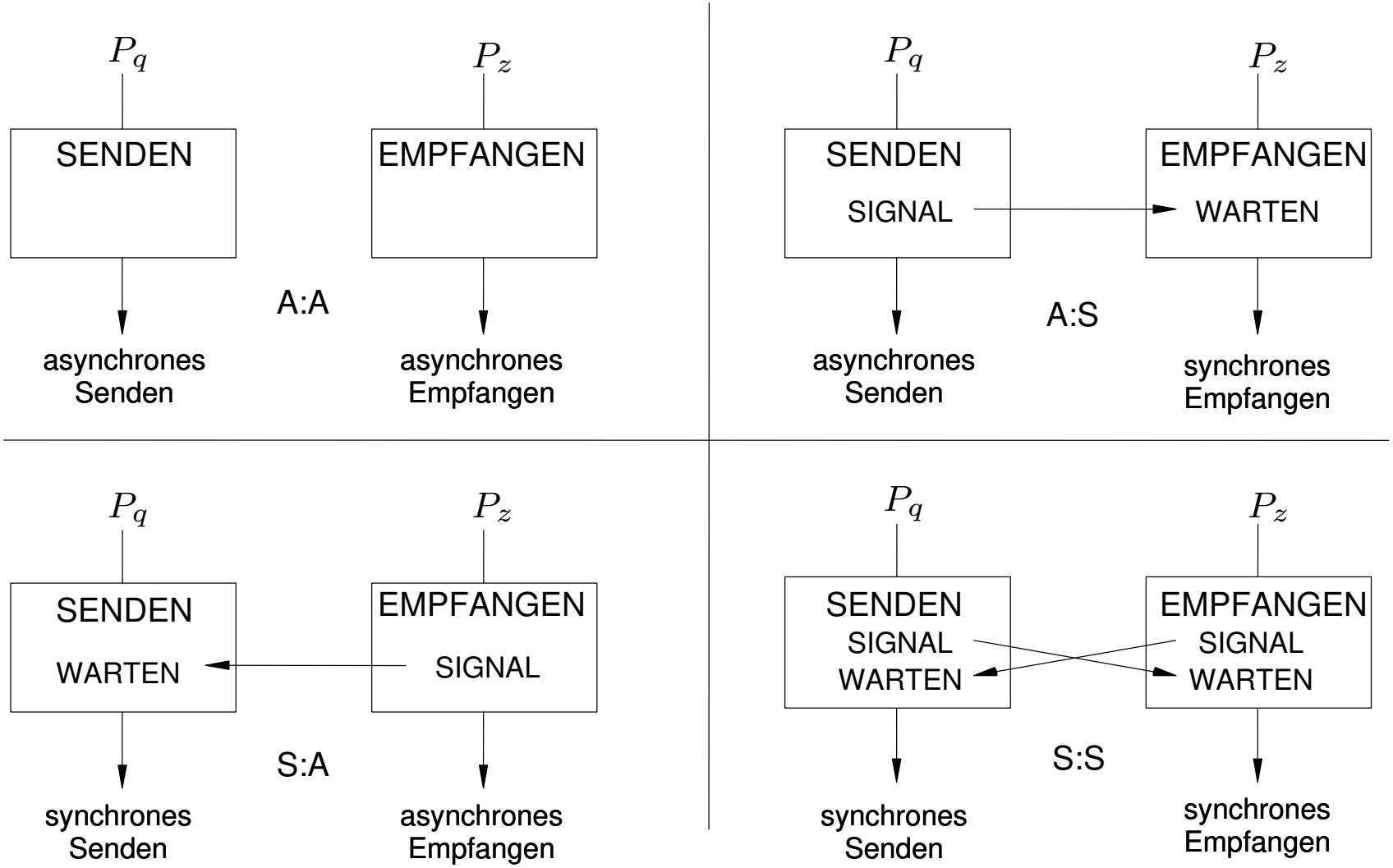
- Empfänger zuerst:



4.3.1 Synchrone/Asynchrone Kommunikation

- Bisher wurde *keine zeitl. Abstimmung* zw. Sender und Empfänger gefordert, man nennt dieses Vorgehen **asynchron** (asynchrones Senden/Empfangen):
  - Beide rufen die jeweilige Operation auf, legen ggf. Daten im Kanal ab, verlassen die Prozedur und arbeiten weiter, ohne auf den Partner zu warten
- Manchmal ist der Empfänger auf den Empfang der Nachricht angewiesen, d. h. er kann erst nach ihrem Erhalt weiter arbeiten, man spricht dann von einem **synchronen Empfangen**:
  - Der Prozess wird in der Empfangsoperation so lange aufgehalten (blockiert), bis das Senden erfolgt
  - So **synchronisiert** er sich mit dem Sender (wartet auf ihn)
- Alternativ ist **synchrones Senden** möglich: der Sender wird solange blockiert (weil er evtl. die gesendeten Daten überschreiben muß), bis die dazugehörige Empfangsoperation aufgerufen wird.
- Durch Kombination ergeben sich die vier folgenden Varianten

Koordinationsvarianten der Kommunikation

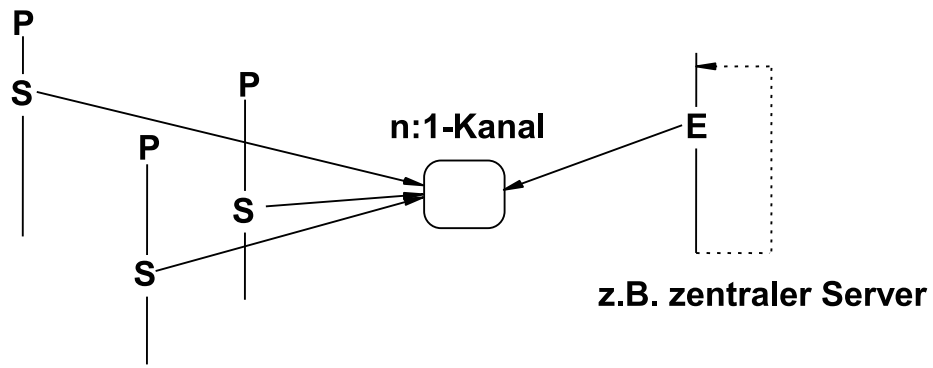


"Rendezvous", "Handshake"

4.3.2 Kapazität

Bisher wurde im Kanal genau eine Nachricht bzw. ein „Empfangswunsch“ gespeichert.

- Wünschenswert: Fähigkeit, mehrere Nachrichten zu „puffern“
- Beispiel: Mehrere Prozesse senden an einen zentralen „Server-Prozess“



Orthogonal dazu:

- Fähigkeit, mehrere Empfangswünsche zu „puffern“
- Beispiel: Server besteht aus mehreren replizierten Prozessen, die über einen gemeinsamen Kanal adressiert werden.

Datenstrukturen und ihre Kapazität

- Zur „Nachrichtenpufferung“ werden spezielle Datenstrukturen verwendet (i. d. R. Warteschlangen).
- Für einen n:n/S:S-Kanal (beliebig viele Sender, beliebig viele Empfänger, synchrones Senden / Empfangen) bedeutet dies z. B.:
  - Warteschlange für wartende Senderprozesse
  - Warteschlange für gespeicherte Nachrichten
  - Warteschlange für wartende Empfängerprozesse
  - Warteschlange für gespeicherte Behälteradressen
- Die Kanalkapazität beeinflusst die Effizienz und Semantik der Operationen
  - **Unbegrenzte Kapazität**: erfordert dynamische Speicherverwaltung (anfordern/freigeben) in den Kommunikationsoperationen
  - **Begrenzte Kapazität**: erfordert Mechanismen bei „Überlauf“
- Mögliche Überlaufmechanismen:
  - Überschreiben
  - Abweisung der Operation
  - Blockierung des Aufrufers, bis Kapazität frei wird

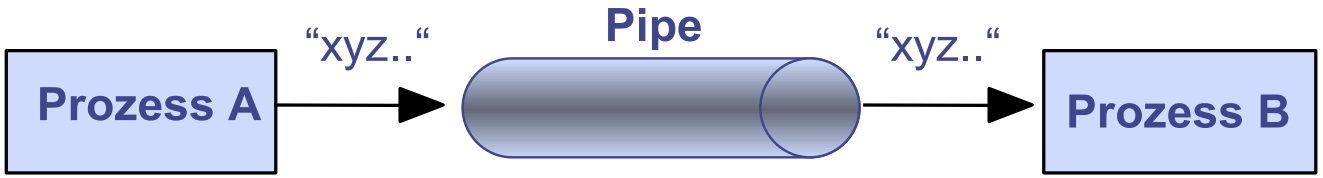
4.3.3 Zuordnung von Kanälen: Ports

- Kanal ist ein eigenständiges Kommunikationsobjekt und kann unabhängig von konkreten Sendern und Empfängern existieren.
- Es ist jedoch gelegentlich sinnvoll, ein Kommunikationsobjekt fest einem Prozess zuzuordnen.
- Dies kann sendeseitig oder empfangsseitig erfolgen:
  - Besitzt ein Prozess einen Kanal, in dem er alle seine ausgehenden Nachrichten ablegt, so spricht man von einem **Ausgangsport**.
  - Besitzt ein Prozess einen Kanal, in dem er alle seine eingehenden Nachrichten ablegt, so spricht man von einem **Eingangsport**.

- Eingangsports sind n:1-Kanäle, Ausgangsports 1:n-Kanäle.
- Ports sind in modernen Betriebssystemen die am meisten verbreiteten Kommunikationsobjekte.

Kommunikationskonzepte in Unix und Win NT

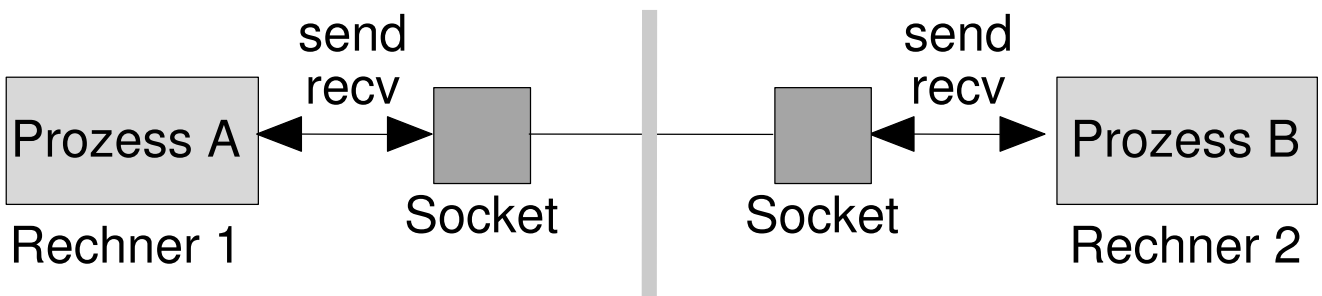
- Pipe
  - Spezieller 1:1 Kanal für kontinuierlichen, gerichteten Zeichenstrom



  - Die Pipe hat eine begrenzte Kapazität.
  - Lokaler Mechanismus zwischen genau zwei Prozessen
  - Ist die Pipe voll, so wird der sendende (schreibende) Prozess blockiert.
  - Ist die Pipe leer, so wird der empfangende (lesende) Prozess blockiert.
  - Eine Pipe ist somit ein “Bounded-Buffer” zwischen zwei Prozessen.
  - Beispiel unter Unix:
    `cat datei.txt | grep Name`
    `cat` gibt eine Datei aus, `grep` durchsucht die Eingabe nach `Name`,
    die Pipe `|` verbindet die Ausgabe von `cat` mit der Eingabe von `grep`

Sockets (Unix, Windows NT)

- Sockets sind Endpunkte einer Duplex-Verbindung für nichtlokale Kommunikation, z. B. zwischen Computern (verteilte Systeme)
- Ein Socket kann von mehreren Prozessen benutzt werden
- Verschiedene Socket-Typen werden angeboten, z. B.:
  - stream socket – verbindungsorientiert, d. h. Verbindung zu anderem Socket muss zunächst aufgebaut werden; das System stellt sicher, dass Daten nicht verloren gehen und in Sendereihenfolge ankommen.
  - datagram socket – paketorientiert, d. h. Daten werden in Pakete aufgeteilt und einzeln versandt. Pakete können einander überholen oder verloren gehen.
- Einsatz ist sowohl blockierend (synchron) als auch nichtblockierend (asynchron) möglich
- Zur Kommunikation werden Ports verwendet



Ablauf einer Verbindung über Sockets

Client:

- Socket erstellen
- erstellten Socket mit der Server-Adresse verbinden, von welcher Daten angefordert werden sollen
- Senden und Empfangen von Daten
- Verbindung trennen
- Socket schließen

Server:

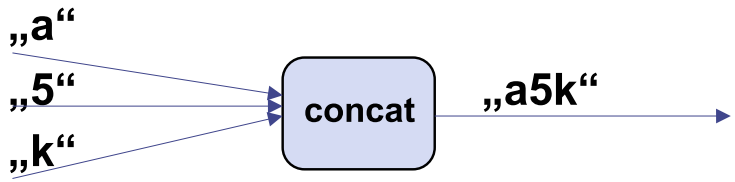
- Socket erstellen
- Binden des Sockets an einen Port, über welche Anfragen akzeptiert werden
- Anfrage akzeptieren, dafür ein neues Socket-Paar für diesen Client erstellen
- Bearbeiten der Client-Anfrage auf dem neuen Client-Socket
- Client-Socket wieder schließen

4.3.4 Gruppenkommunikation

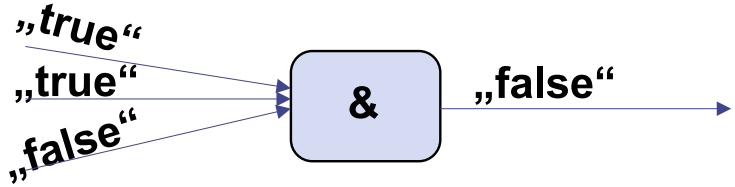
- Es gibt Situationen, wo ein Prozess identische Nachrichten an viele (*multicast*) oder an alle Prozesse (*broadcast*) schickt.
- Symmetrisch: viele Prozesse senden an einen Empfänger (Teil)nachrichten, die als Zusammenfassung die eigentliche Nachricht bilden (*combine*).
- Man spricht von Gruppenkommunikation vs. Einzelkommunikation
- Dadurch ergeben sich folgende Varianten:
  - 1:1-Kanal (wie bisher, „one-to-one-communication“)
  - 1:n-Kanal (broadcast, multicast, „one-to-many-comm.“)
  - n:1-Kanal (combine, „many-to-one-communication“)
  - n:n-Kanal (all-to-all, „many-to-many-communication“)

Combine: Art der Zusammenfassung

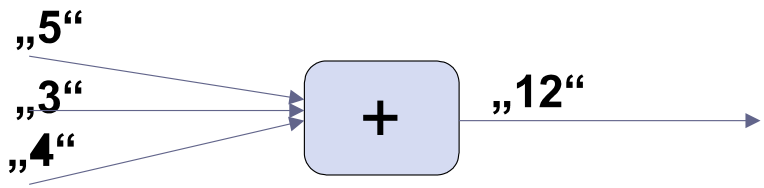
- Während die Vervielfältigung semantisch eindeutig ist (es werden Kopien der Nachricht versandt), muss die Art der Kombination festgelegt werden (i. d. R. durch einen Operationsparameter).
- Tatsächlich gibt es die verschiedensten Variationen, z.B.:
  - Konkatenation:



  - Logische Verknüpfung:



  - Arithmetische Addition:



Kooperation: potentielles Problem - Teil 1

Kooperation: mehrere Prozesse greifen auf dieselben Daten zu.
Um Fehler und Inkonsistenzen zu vermeiden, müssen die Zugriffe koordiniert sein.

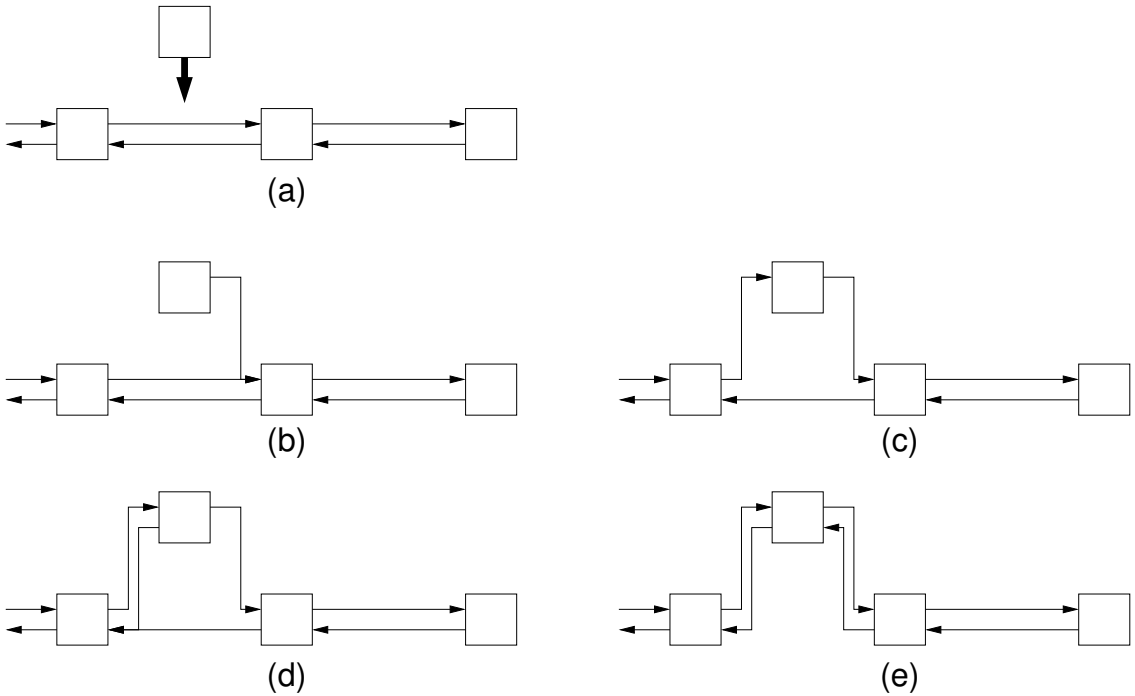
Vorgestellte Synchronisations- und Sperrmechanismen

- `signal(s)`, `swait(s)`: busy wait, Signalvariable, `class Signal`
- `sync(s)`: rendevous, gesteuert mit einer Signalvariable, class Synchronisation, Gruppensynchronisation, `class BarrierSync`
- `lock(s)`, `unlock(s)`: Sperren (um krit. Abschnitt), `class Lock`

Die werden wir nutzen, sie aber auch erweitern.

Kooperation: potentielles Problem - Teil 2

- **Beispiel**: Listenoperation „Einfügen“, aufgelöst in Einzelschritte



- In den Situationen c) und d) ist die Listenstruktur **inkonsistent**, d. h. ein zu diesen Zeitpunkten parallel zugreifender Prozess sähe eine **fehlerhafte** Datenstruktur!

4.3.5 Sperren bei Kooperation

- Kooperation von Prozessen auf gemeinsamen Daten fällt mit dem Problem des kritischen Abschnitts bzw. des gegenseitigen Ausschlusses zusammen (s. Kernsperre)
- Zur Erinnerung: Ein kritischer Abschnitt ist eine Operationsfolge, bei der ein nebenläufiger Zugriff auf gemeinsame Daten oder verzahnte Ausführung zu Fehlern führen kann
- Zur Sicherung kritischer Abschnitte können wir die weiter oben eingeführten Sperroperationen mit einer Sperre `s` einsetzen:

Kooperation – Beispiel

- Man stelle sich zwei Prozesse vor, die eine Zählervariable inkrementieren bzw. dekrementieren. Man kann daraus ein Kooperationsobjekt machen, auf das beide zugreifen.

```
class SharedCounter {
  private int c = 0;

  public void increment() {
    c = c+1;
  }

  public void decrement() {
    c = c−1;
  }
}
```

Kooperation – Problem (Wdh. vom früheren Beispiel)

- Fehlerhafte Beispiel-Befehlsfolge bei gleichzeitiger Ausführung von `increment` (in P_1) und `decrement` (in P_2):

P_1	c	P_2
	5	
c lesen $\rightarrow 5$	5	
$5 + 1 \rightarrow 6$	5	c lesen $\rightarrow 5$
c schreiben	$\rightarrow 6$	$5 - 1 \rightarrow 4$
	$\rightarrow 4$	c schreiben

- Der Effekt von `increment` geht verloren!
  Ursache: sowohl `increment` als auch `decrement` sind kritische Abschnitte, die nicht gleichzeitig ausgeführt werden dürfen

- Ist das Objekt im Kern realisiert, so wird die Sicherung des kritischen Abschnitts durch die Kernsperre implizit vorgenommen.
- Ist das Objekt außerhalb des Kerns realisiert, so muss der kritische Abschnitt explizit gesichert werden!

Kritische Abschnitte außerhalb des Kerns

Methoden `increment` und `decrement` sind kritische Abschnitte,
die wir mithilfe von Sperren (Locks) sichern, so dass im obigen Beispiel kein Fehler mehr auftritt:

```
class SharedCounter {
  private int c = 0;
  private Lock l = new Lock();

  public void increment() {
    l.lock();
    c = c+1;
    l.unlock();
  }

  public void decrement() {
    l.lock();
    c = c−1;
    l.unlock();
  }
}
```

Zur Implementierung von `lock`/`unlock` siehe Folie 16.

4.3.6 Semaphor

- Zur Sicherung kritischer Abschnitte wird in Betriebssystemen, außer einer Sperre (Lock), auch ein sog. **Semaphor** verwendet.
- Semaphore sind *Zählsperren*: sie können einer bestimmten Anzahl von Prozessen das Betreten des kritischen Abschnitts erlauben (im Gegensatz zu Sperren, die stets nur einen Prozess zulassen).
- Eingeführt ca. 1965 von E. W. Dijkstra, ist ein Semaphor ursprünglich ein Zähler (Ganzzahlvariable) `S`, mit Operationen `P(S)` und `V(S)`
  - Ursprung – Einspurige Eisenbahn:
    `P` (holl.) = Passieren, `V` (holl.) = Freigeben.
  - `P()` und `V()` sind unteilbare (atomare) Operationen!
  - `P()` (entspricht `lock`) dekrementiert den Zähler;
    wird der Zähler dadurch negativ, so wird der Prozess blockiert
    `V()` (entspricht `unlock`) inkrementiert den Zähler;
    ist der Zähler danach nichtpositiv (d.h. es gibt blockierte Prozesse), so wird einer der blockierten Prozesse deblockiert.
- In der Literatur sind Semaphore oft nicht-negativ, dafür werden blockierte Prozesse in einer Warteschlange verwaltet

Beispielimplementierung Semaphor

```
class Semaphore {
  private int c = 1; // capacity counter, 1 for mutual exclusion
                     // c=1: free, c<=0: occupied   ;
                     // if c<0 : -c is the number of
                     // waiting processes

  public Semaphore(int capacity) { // Konstruktor
    c = capacity;
  }

  public synchronized void P() {
    c = c-1;
    if (c<0)
      wait(); //enqueue process
  }

  public synchronized void V() {
    c = c+1;
    if (c<=0)
      notify(); //deblock one process
  }
}
```

Semaphore: Beispiel

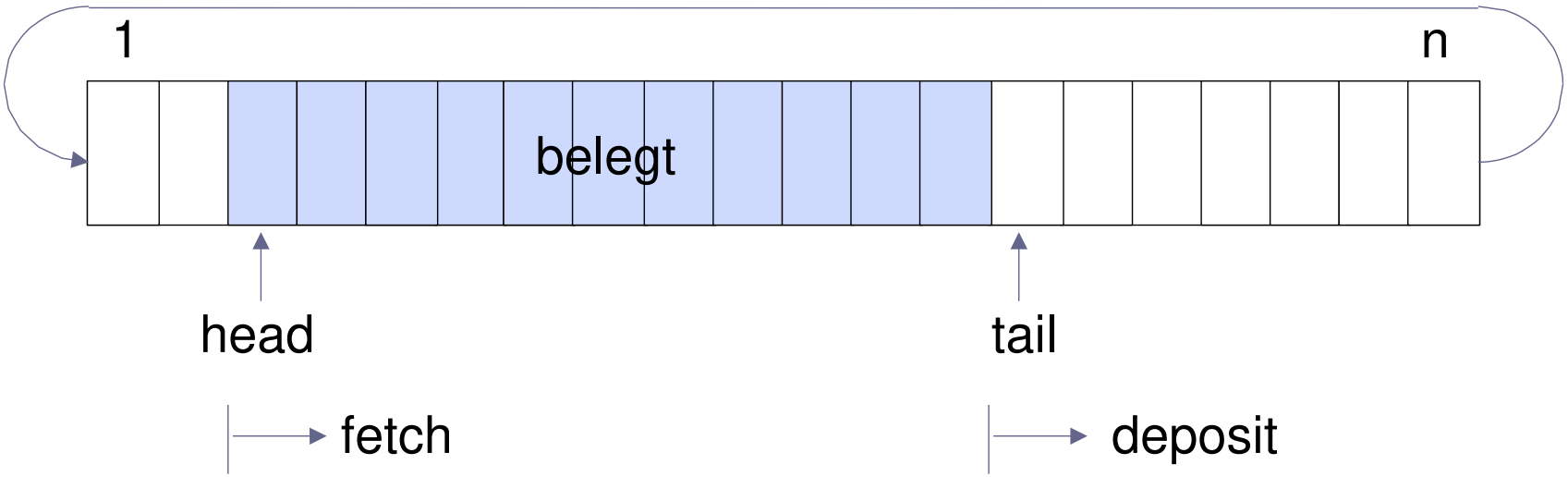
- Im einfachsten Fall können Semaphore wie Sperren verwendet werden (`P` entspricht `lock`, `V` entspricht `unlock`)

```
class SharedCounter {
  private int c = 0;
  private Semaphore S = new Semaphore(1);
                // Semaphorzähler mit Initialwert 1
  public void increment() {
    S.P();
    c = c+1;
    S.V();
  }

  public void decrement() {
    S.P();
    c = c−1;
    S.V();
  }
}
```

- Mit Semaphoren können aber auch kompliziertere Interaktionen realisiert werden – siehe folgende Folien

Semaphor – Anwendung: Bounded Buffer

- Mehrere Prozesse benutzen einen gemeinsamen Puffer:
  - Prozesse können Daten dort ablegen: `deposit(data)`
  - Prozesse können Daten dort abholen: `fetch(data)`
- Neben der Sicherstellung des gegenseitigen Ausschlusses müssen offensichtlich noch weitere Bedingungen berücksichtigt werden:
  - `deposit` darf nur aufgerufen werden, wenn noch Platz im Puffer vorhanden ist.
  - `fetch` darf nur aufgerufen werden, wenn Puffer nicht leer ist.

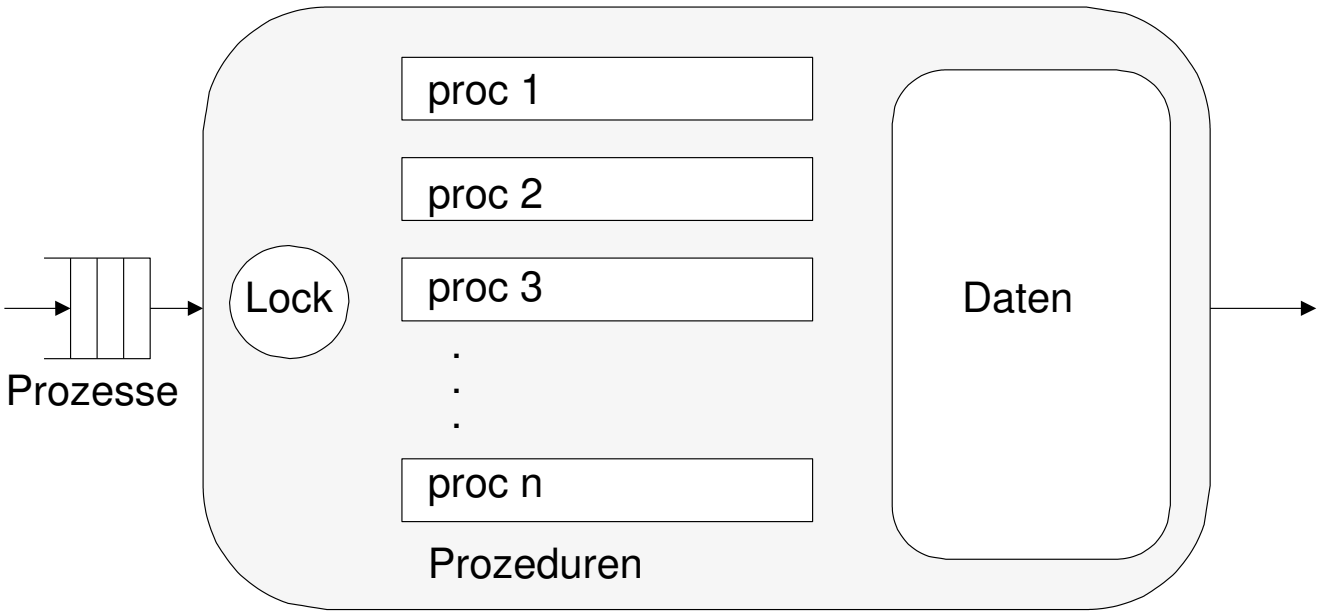


Bounded Buffer: Implementierung mit Semaphoren

```
class BoundedBuffer {
  Object buffer[n];
                          // Insgesamt drei Semaphore
  Semaphore full = new Semaphore(0);  // ... fuer volle Plaetze
  Semaphore empty = new Semaphore(n); // ... fuer leere Plaetze
  Semaphore mutex = new Semaphore(1); // ... fuer ggs. Ausschluss

  void deposit(Object data) {
    empty.P(); //wenn kein Platz, auf fetch warten
    mutex.P(); //ggs. Ausschluss sicherstellen
    buffer[tail] = data;
    tail = (tail+1)mod n; count = count+1;
    mutex.V(); // anderen Prozessen Zugriff erlauben
    full.V(); //wenn Prozesse in fetch warten, ersten deblockieren
  }

  Object fetch() {
    full.P(); // wenn leer, auf deposit warten
    mutex.P(); //ggs. Ausschluss sicherstellen
    result = buffer[head];
    head = (head+1) mod n; count = count−1;
    mutex.V(); // anderen Prozessen Zugriff erlauben
    empty.V(); // wenn Prozesse in deposit warten, ersten deblockieren
  }
}
```

4.3.7 Monitor: Motivation und Definition

- Expliziter Umgang mit Sperren und Semaphoren ist fehleranfällig. Wünschenswert wäre ein automatisches Sperren/Freigeben.
- Ein Objekt, das den gegenseitigen Ausschluss von Prozessen sicherstellt, ohne dass der Programmierer explizit Sperroperationen einfügt, heißt Monitor.
- Ein Monitor ist ein Objekt bestehend aus Prozeduren (im Bild: proc 1 – proc n) und Datenstrukturen, das zu jedem Zeitpunkt nur von einem Prozess benutzt werden darf.



Monitor-Beispiel: Zähler

- Das Monitorkonzept kann idealerweise von der Programmiersprache bereitgestellt werden und führt das Setzen und Freigeben von Sperren automatisch durch
- Das obige Beispiel einer Kooperation auf einer Zählervariable kann als Monitor folgendermaßen in Pdeusocode formuliert werden:

```
monitor sharedCounter {
  int c = 0;
  void increment() { c = c+1; }
  void decrement() { c = c−1; }
}
```

- Dies kann in Java „von Hand“ mit **`synchronized`** + **`private`** nachprogrammiert werden:

```
class SharedCounter {
  private int c = 0;
  public synchronized void increment() { c = c+1; }
  public synchronized void decrement() { c = c−1; }
}
```

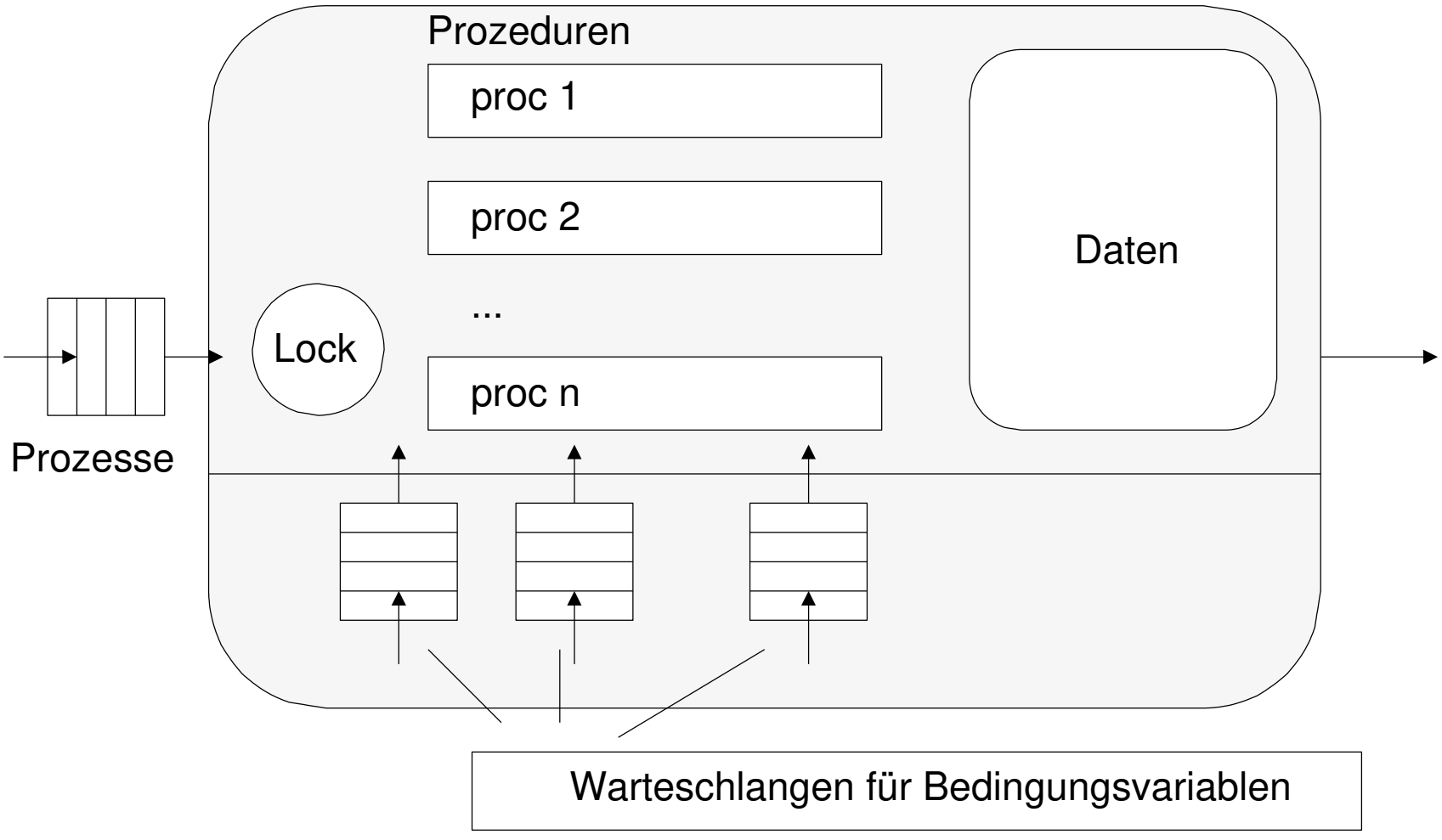
Bounded Buffer als Monitor

```
monitor BoundedBuffer {
  Object buffer[n];
  int head = 1;
  int tail = 1;
  int count = 0; // Aktuelle Anzahl Elemente im Puffer
  condition not_full = true; // Erste Bedingungsvariable
  condition not_empty = false; // Zweite Bedingungsvariable

  void deposit(Object data) {
    while (count ==  n)
      cwait(not_full); // Monitor wird freigegeben!
    buffer[tail] = data;
    tail = (tail+1) mod n;
    count = count+1;
    csignal(not_empty);
  }

  Object fetch() {
    while (count == 0)
      cwait(not_empty); // Monitor wird freigegeben!
    Object result = buffer[head];
    head = (head+1) mod n;
    count = count−1;
    csignal(not_full);
    return result;
  }
```

Bedingte Synchronisation

- Monitor für den Bounded Buffer zeigt ein potentielles Problem auf
- Während ein Prozess auf eine Bedingung (z. B. $count > 0$ bei `fetch`) wartet, *muss* der Monitor für andere Prozesse freigegeben werden, sonst können sich Prozesse gegenseitig blockieren: z.B. kein Prozess kann `deposit` ausführen und dadurch wartet `fetch` unendlich
- Als Lösung gibt es für Monitore auf Sprachebene das folgende Konzept der Bedingten Synchronisation, mit zwei Operationen:
  - `cwait(c)` Prozess gibt Monitor frei und wartet auf das nachfolgende `csignal(c)`, d. h. das Eintreten der Bedingung c. Nach dem Erhalten des Signals, setzt er im Monitor fort.
  - `csignal(c)` Ein wartender Prozess wird geweckt und belegt den Monitor. Gibt es keinen wartenden Prozess, hat die Prozedur keinen Effekt.
- Die wartenden Prozesse werden (wie auch bei der Signalisierung oder den Semaphoren) in einer Warteschlange verwaltet.
- Die Bedingungen werden durch logische Bedingungsvariablen implementiert

Monitor mit Bedingungsvariablen



Bedingte Synchronisation (Forts.)

- (Damit kann man wieder Fehler machen) Aber: Beachte den Unterschied zu den Signalisierungsoperationen `signal` / `swait` (Folien 4-6, 4-7):
  - `cwait` gibt den Monitor frei und blockiert den Prozess in einer Warteschlange. Beides geschieht atomar!
    `swait` hingegen gibt das Objekt nicht frei
  - Wenn ein durch `cwait` blockierter Prozess deblockiert wird, muss er erst den Monitor belegen bevor die Ausführung fortgesetzt wird
  - `csignal` hat keinen Effekt, wenn kein Prozess auf die Bedingung wartet: Ein späteres `cwait` blockiert den aufrufenden Prozess auf jeden Fall.
    `signal` setzt hingegen die Signal-Variable auf `set`: Ein späteres `swait` blockiert deshalb nicht!

Die Java-Methoden `wait` und `notify` entsprechen weitgehend `cwait` und `csignal`, allerdings gibt es in Java für jeden Monitor nur *eine* Bedingungsvariable, auf die sich `notify` und `wait` implizit beziehen!

Anzahlbegrenzte Kooperation

- Ein Kooperationsabschnitt (auch kritischer Abschnitt genannt) war bisher dadurch gekennzeichnet, dass sich zu einem Zeitpunkt genau ein Prozess „darin aufhält“.
- Dieses Prinzip kann man auf andere Kapazitäten als 1 erweitern.
- Man kann sowohl für die Zahl der „durchgelassenen“ als auch für die Zahl der wartenden Prozesse Obergrenzen vorsehen
- Gründe, die Anzahl der Prozesse in einem bestimmten Bereich zu begrenzen, außer den möglichen Konflikten, sind:
  - Platzmangel
  - Leistungsabfall

Mehrsortenkooperation, z. B. Leser/Schreiber

Beispiel: Leser-Schreiber-Kooperation (*Reader-Writer-Problem*)

- Nicht alle Prozesse greifen schreibend auf die gemeinsamen Daten zu. Einige lesen nur und dürfen gleichzeitig zueinander arbeiten
- In dem Kooperationsabschnitt dürfen sich daher
  - entweder ein Schreiber
  - oder beliebig viele Leser aufhalten
- Eine Lösung soll vermeiden, dass ein wartender Schreiber wegen kontinuierlich ankommender Leser potentiell für immer blockiert bleibt

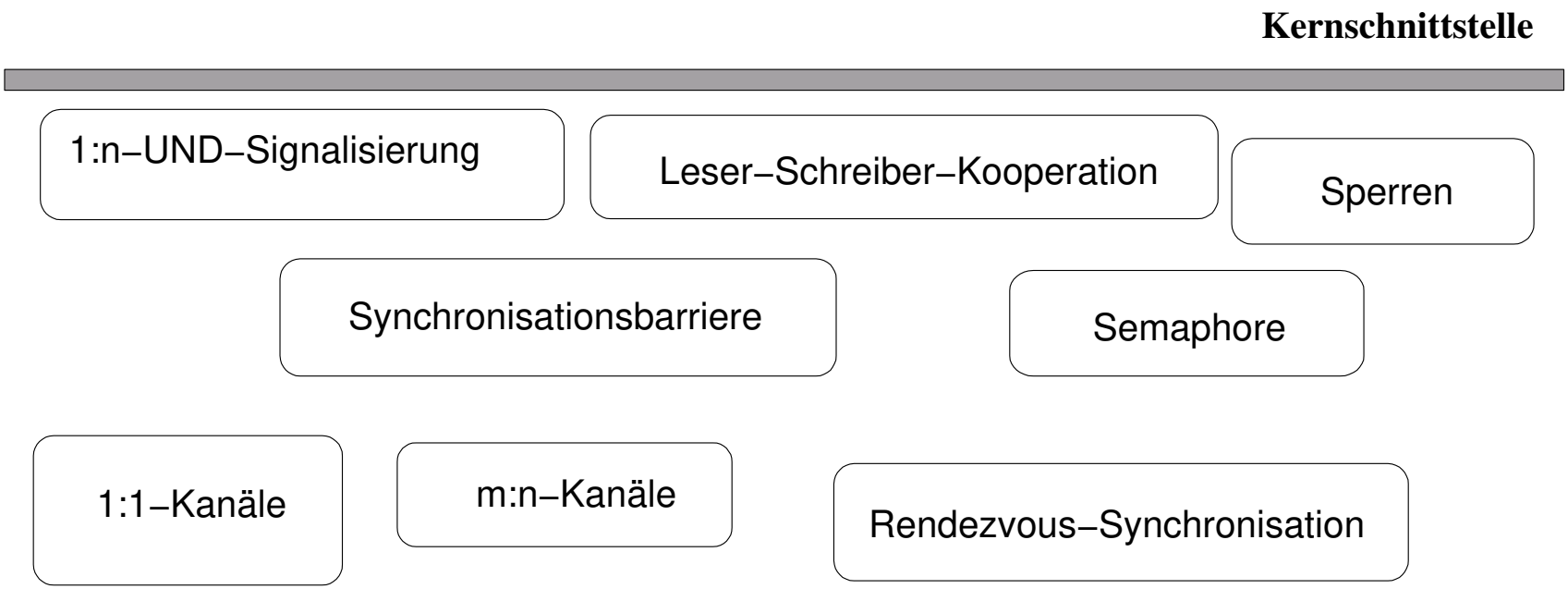
Verallgemeinerung:

- Es gebe k Sorten von Prozessen. Im Kooperationsabschnitt dürfen sich
  - c_1 Prozesse der Sorte 1 und/oder
  - c_2 Prozesse der Sorte 2 und/oder
  - ...
  - c_k Prozesse der Sorte k

  aufhalten
- Das Leser-Schreiber-Problem ist dann ein Spezialfall mit $k = 2, c_1 = 1$ und $c_2 = \infty$. Mehr Details hierzu in der Übung.

4.3.8 Interaktionsmechanismen: Überblick

- Die Sammlung von Interaktionsmechanismen ist als Vorrat zu verstehen, aus dem je nach Einsatzgebiet des Betriebssystems eine Teilmenge zur Verfügung gestellt werden kann.
- In einem Betriebssystem müssen nicht alle möglichen Varianten angeboten werden, aber man sollte den Programmierern schon eine ausreichende Wahlmöglichkeit lassen.



Vorgestellte Synchronisations- und Sperrmechanismen

- `signal(s), swait(s)`: busy wait, Signalvariables, `class Signal`
- `sync(s)`: rendevous, gesteuert mit Signalvariable, `class Synchronisation`, Gruppensynchronisation, `class BarrierSync`
- `lock(s), unlock(s)`: Sperren (um krit, Abschnitt), `class Lock`
- `Mutex`: gegenseitiger Ausschluss (mit div. Methoden zu realisieren).
- `Semaphore, P(), V()`: Zählsperre, `class Semaphore`
- `Monitor`: sperrt Prozeduren, in Java z.B. mit synchronized Methoden realisierbar.
- `cwait(c), csignal(c)`: bedingtes Warten und freigeben.

4.4 Beispiele aus einem konkreten BS

Kooperation und Koordination in Windows NT

Windows NT kennt vier verschiedene Synchronisationsobjekte: *semaphore*, *event*, *mutex*, *critical section*.

- Semaphore:
  - Wird mit einer positiven Zahl initialisiert und im Sinne einer anzahlbegrenzenden Kooperation (*Capacity Lock*) zur Betriebsmittelverwaltung verwendet.
  - Mit `CreateSemaphore()` wird das Objekt erzeugt und kann nach `OpenSemaphore()` benutzt werden.
  - Mit einer Warteoperation wie `WaitForSingleObject()` (entspricht P-Operation) wird der Zählerwert dekrementiert und mit `ReleaseSemaphore()` (entspricht V-Operation) inkrementiert.
  - Wenn der Zähler den Wert 0 erreicht hat, wirkt die Warteoperation blockierend.
  - Semaphore können auch adressraumübergreifend eingesetzt werden, (d. h. zwischen Threads unterschiedlicher Prozesse).

Koordination/Kooperation in Windows NT (Forts.)

- Mutex:
  - Ein Mutex dient dem gegenseitigen Ausschluss.
  - Nach `CreateMutex()` und `OpenMutex()` kann mit einer Warteoperation (z. B. `WaitForSingleObject()`) der kritische Abschnitt betreten werden.
  - Bei Verlassen des kritischen Abschnitts wird die Sperre mit `ReleaseMutex()` wieder freigegeben.
  - Ein Mutex kann von beliebigen Threads im System benutzt werden (prozessübergreifend)
- Critical Section:
  - Ein Critical-Section-Objekt ist eine vereinfachte/effizientere Variante des Mutex, speziell für Prozesse im selben Adressraum (d. h. Threads im selben Prozess).
  - Mit `InitializeCriticalSection()` angelegt, wird der kritische Bereich über `EnterCriticalSection()` betreten und mit `LeaveCriticalSection()` wieder verlassen.

Ausfall Vorlesungen

Achtung!

Die Vorlesungen am **17.10, 21.10, 24.10, 28.10** fallen aus.

Die nächste Vorlesung findet am **31.10** statt.