9. Foliensatz Betriebssysteme

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences
(1971–2014: Fachhochschule Frankfurt am Main)
Fachbereich Informatik und Ingenieurwissenschaften
christianbaun@fb2.fra-uas.de

Lernziele dieses Foliensatzes

- Am Ende dieses Foliensatzes kennen/verstehen Sie...
  - was **kritische Abschnitte** und **Wettlaufsituationen** sind
  - was **Synchronisation** ist
    - wie **Signalisierung** die Ausführungsreihenfolge der Prozesse beeinflusst
    - wie mit **Blockieren** kritische Abschnitte gesichert werden
    - welche Probleme (**Verhungern** und **Deadlocks**) beim Blockieren entstehen können
    - wie **Deadlock-Erkennung mit Matrizen** funktioniert
  - verschiedene Möglichkeiten der **Kommunikation** zwischen Prozessen:
    - **Gemeinsamer Speicher** (Shared Memory)
    - **Nachrichtenwarteschlangen** (Message Queues)
    - **Pipes**
    - **Sockets**
  - verschiedene Möglichkeiten der **Kooperation** von Prozessen
    - wie **Semaphore** kritische Abschnitte sichern können
    - den Unterschied zwischen **Semaphor** und **Mutex**

Übungsblatt 9 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes

Interprozesskommunikation (IPC)

- Prozesse müssen nicht nur Lese- und Schreibzugriffe auf Daten ausführen, sondern auch:
  - sich gegenseitig aufrufen
  - aufeinander warten
  - sich abstimmen
  - kurz gesagt: Sie müssen miteinander **interagieren**
- Bei **Interprozesskommunikation** (IPC) ist zu klären:
  - Wie kann ein Prozess Informationen an andere weiterreichen?
  - Wie können mehrere Prozesse auf gemeinsame Ressourcen zugreifen?

Frage: Wie verhält es sich hier mit Threads?

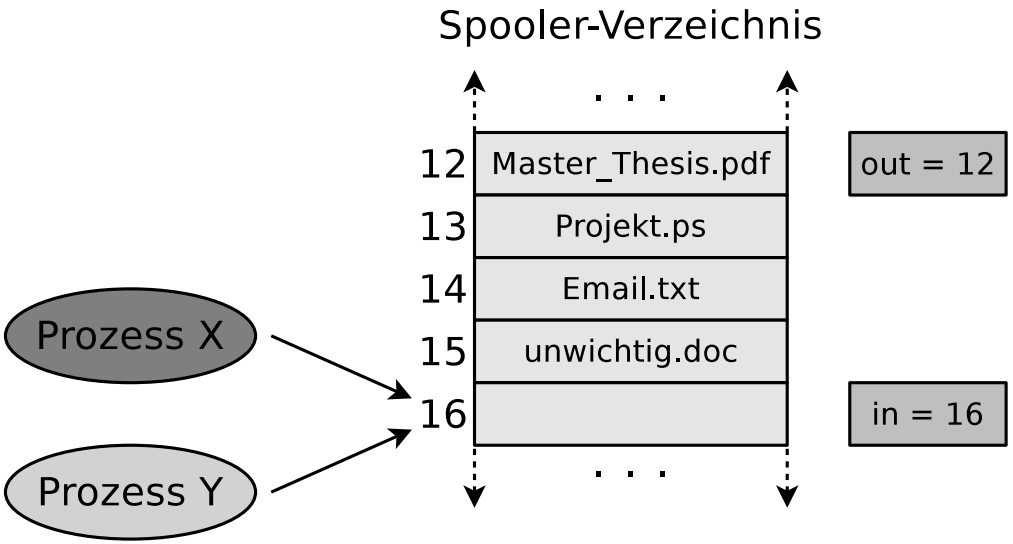
- Bei Threads gelten die gleichen Herausforderungen und Lösungen wie bei Interprozesskommunikation mit Prozessen
- Nur die Kommunikation zwischen den Threads eines Prozesses ist problemlos möglich, weil sie im gleichen Adressraum agieren

Kritische Abschnitte

- Laufen mehrere parallel ausgeführte Prozesse, unterscheidet man:
  - **Unkritische Abschnitte**: Die Prozesse greifen gar nicht oder nur lesend auf gemeinsame Daten zu
  - **Kritische Abschnitte**: Die Prozesse greifen lesend und schreibend auf gemeinsame Daten zu
    - Kritische Abschnitte dürfen nicht von mehreren Prozessen gleichzeitig durchlaufen werden
- Damit Prozesse auf gemeinsam genutzten Speicher ($\Longrightarrow$ Daten) zugreifen können, ist **wechselseitiger Ausschluss** (*Mutual Exclusion*) nötig

Kritische Abschnitte – Beispiel: Drucker-Spooler

Prozess X		Prozess Y
`next_free_slot = in;` (Ergebnis: **16**)		
	Prozess-wechsel	
		`next_free_slot = in;` (Ergebnis: **16**) `Speichere Daten in next_free_slot;` (Ergebnis: **16**) `in = next_free_slot + 1;` (Ergebnis: **17**)
	Prozess-wechsel	
`Speichere Daten in next_free_slot;` (Ergebnis: **16**) `in = next_free_slot + 1;` (Ergebnis: **17**)		



- Das Spooler-Verzeichnis ist konsistent
  - Aber der Eintrag von **Prozess Y** wurde von **Prozess X** überschrieben und ging verloren
- Eine solche Situation heißt **Race Condition**

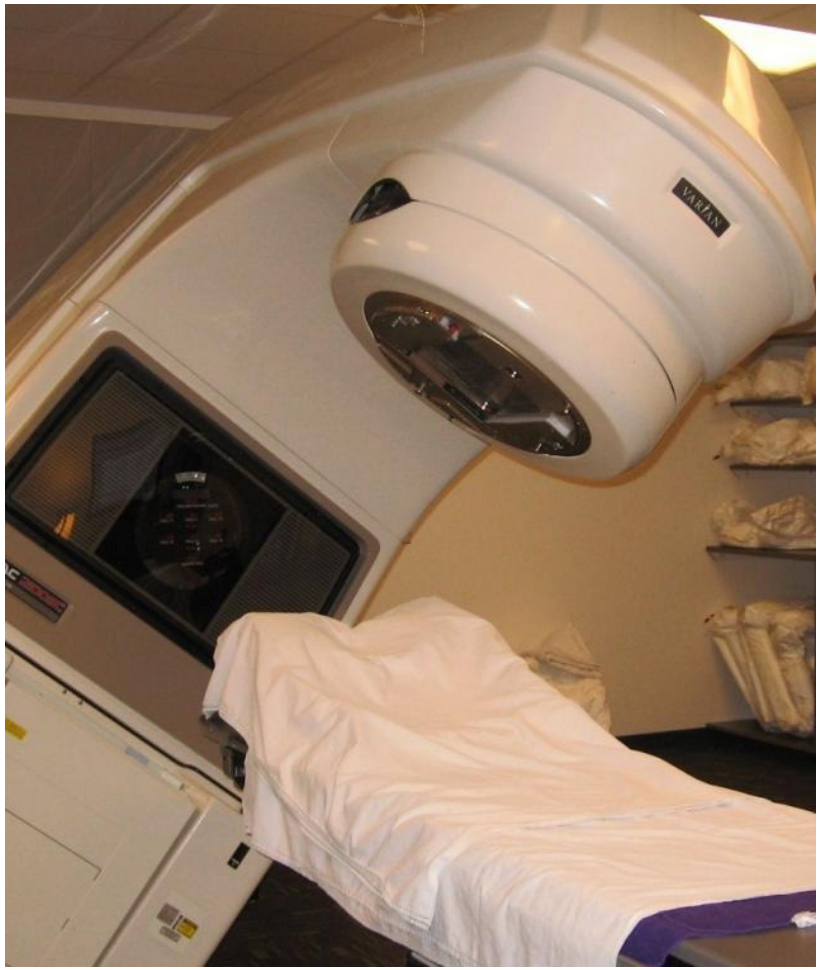
Race Condition (*Wettlaufsituation*)

- **Unbeabsichtigte Wettlaufsituation** zweier Prozesse, die den Wert der gleichen Speicherstelle ändern wollen
  - Das Ergebnis eines Prozesses hängt von der Reihenfolge oder dem zeitlichen Ablauf anderer Ereignisse ab
  - Häufiger Grund für schwer auffindbare Programmfehler
- Problem: Das Auftreten und die Symptome hängen von unterschiedlichen Ereignissen ab
  - Bei jedem Testdurchlauf können die Symptome unterschiedlich sein oder verschwinden
- Vermeidung ist u.a durch das Konzept der **Semaphore** ($\Longrightarrow$ Folie 65) möglich

Therac-25: Race Condition mit tragischem Ausgang (1/2)

- Therac-25 ist ein Elektronen-Linearbeschleuniger zur Strahlentherapie von Krebstumoren
- Verursachte Mitte der 80er Jahre in den USA tödliche Unfälle durch mangelhafte Programmierung und Qualitätssicherung
  - Einige Patienten erhielten eine bis zu hundertfach erhöhte Strahlendosis

An Investigation of the Therac-25 Accidents. Nancy Leveson, Clark S. Turner. IEEE Computer, Vol. 26, No. 7, July 1993, S.18-41
`http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html`



Bildquelle: Google Bildersuche.
Häufig gezeigtes Bild in diesem Kontext.
(Autor und Lizenz: unbekannt)

Therac-25: Race Condition mit tragischem Ausgang (2/2)

- Eine Race Condition („Texas-Bug“) führte zu fehlerhaften Einstellungen des Geräts und damit zu erhöhter Strahlendosis
  - Der Kontroll-Prozess synchronisierte nicht korrekt mit dem Prozess der Eingabeaufforderung
  - Der Fehler trat nur während einer schnellen Eingabekorrektur (Zeitfenster: 8 Sekunden) durch den Benutzer auf
  - Bei Tests trat der Fehler nicht auf, weil es Erfahrung (Routine) erforderte, um das Gerät so schnell zu bedienen

The Worst Computer Bugs in History: Race conditions in Therac-25:
`https://www.bugsnag.com/blog/bug-day-race-condition-therac-25`

„Once the data entry phase was marked complete, the magnet setting phase began. However, if a specific sequence of edits was applied in the Data Entry phase during the 8 second magnet setting phase, the setting was not applied to the machine hardware, due to the value of the completion variable. The UI would then display the wrong mode to the user, who would confirm the potentially lethal treatment.“

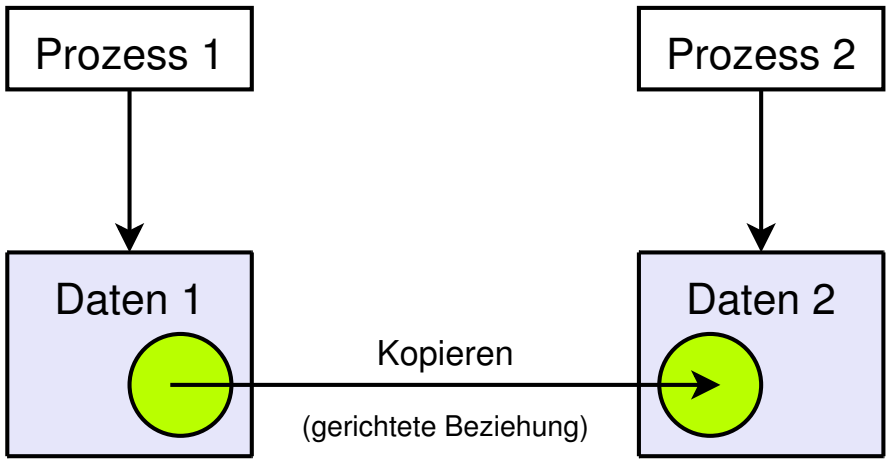
Weitere interessante Quellen

`https://www-dssz.informatik.tu-cottbus.de/information/slides_studis/ss2009/mehner_RisikoComputer_zs09.pdf`
Killer Bug. Therac-25: Quick-and-Dirty: `https://www.viva64.com/en/b/0438/`
Killed by a machine: The Therac-25: `https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/`

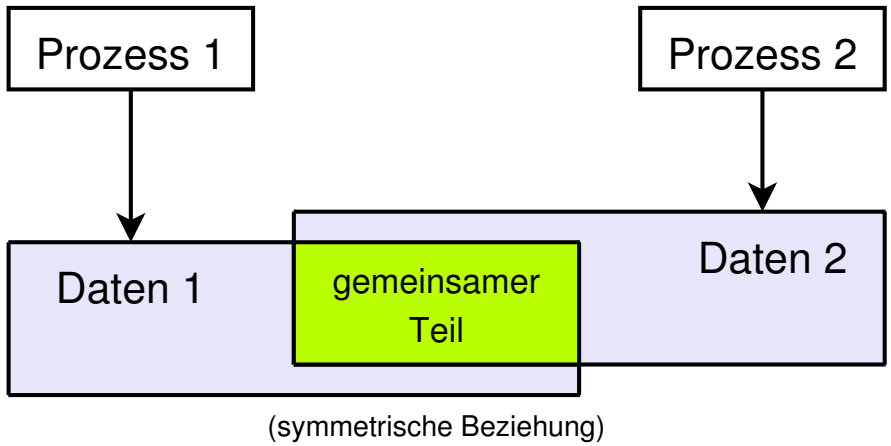
Kommunikation vs. Kooperation

- Die Prozessinteraktion besitzt 2 Aspekte:
  - Funktionaler Aspekt: **Kommunikation** und **Kooperation**
  - Zeitlicher Aspekt: **Synchronisation**

Kommunikation
(= expliziter Datentransport)

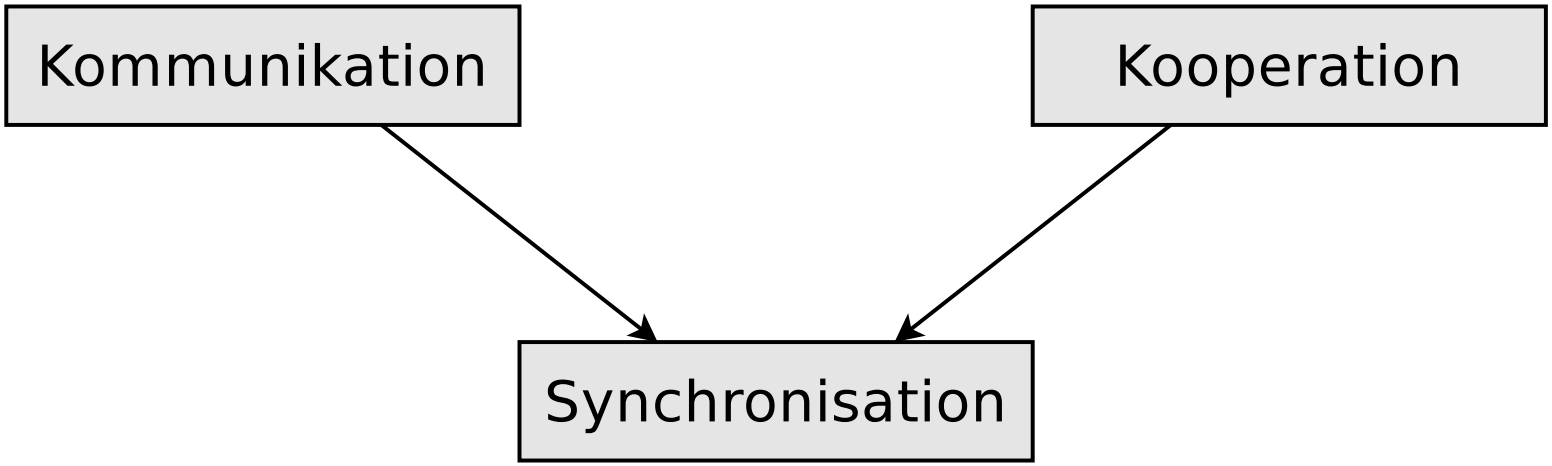


Kooperation
(= Zugriff auf gemeinsame Daten)



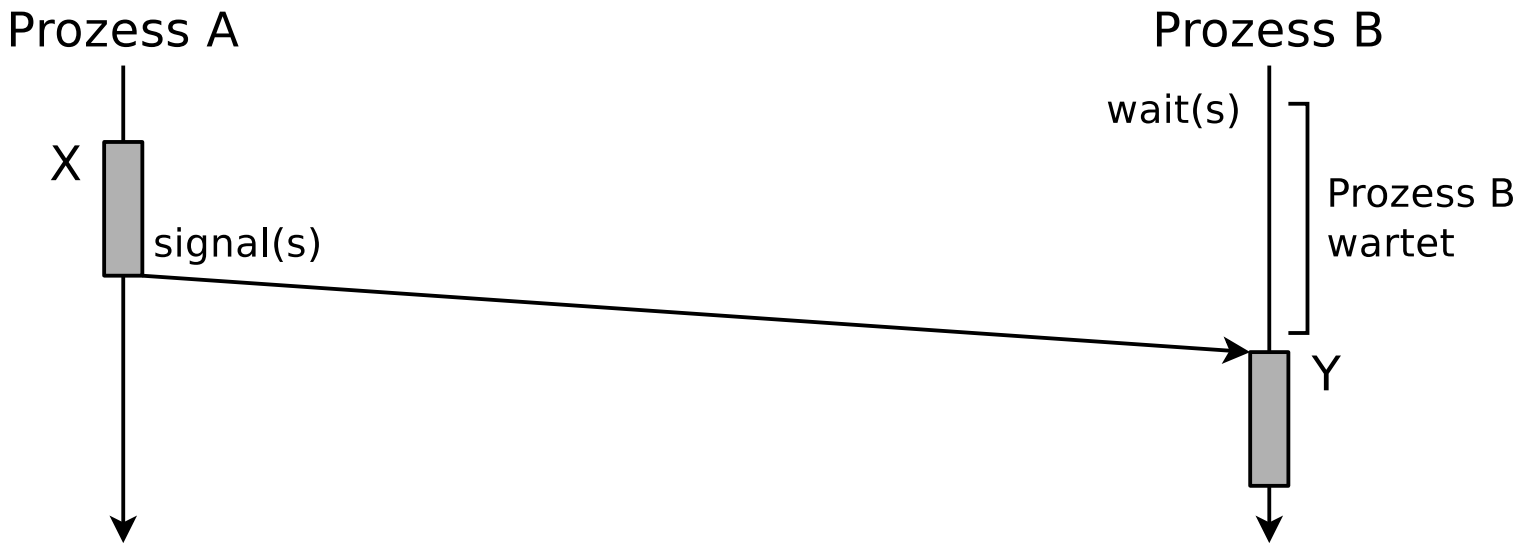
Interaktionsformen

- Kommunikation und Kooperation basieren auf Synchronisation
  - Synchronisation ist die elementarste Form der Interaktion
    - Grund: Kommunikation und Kooperation benötigen eine zeitliche Abstimmung zwischen den Intaraktionspartnern, um korrekte Ergebnisse zu erhalten
  - Darum behandeln wir zuerst die **Synchronisation**

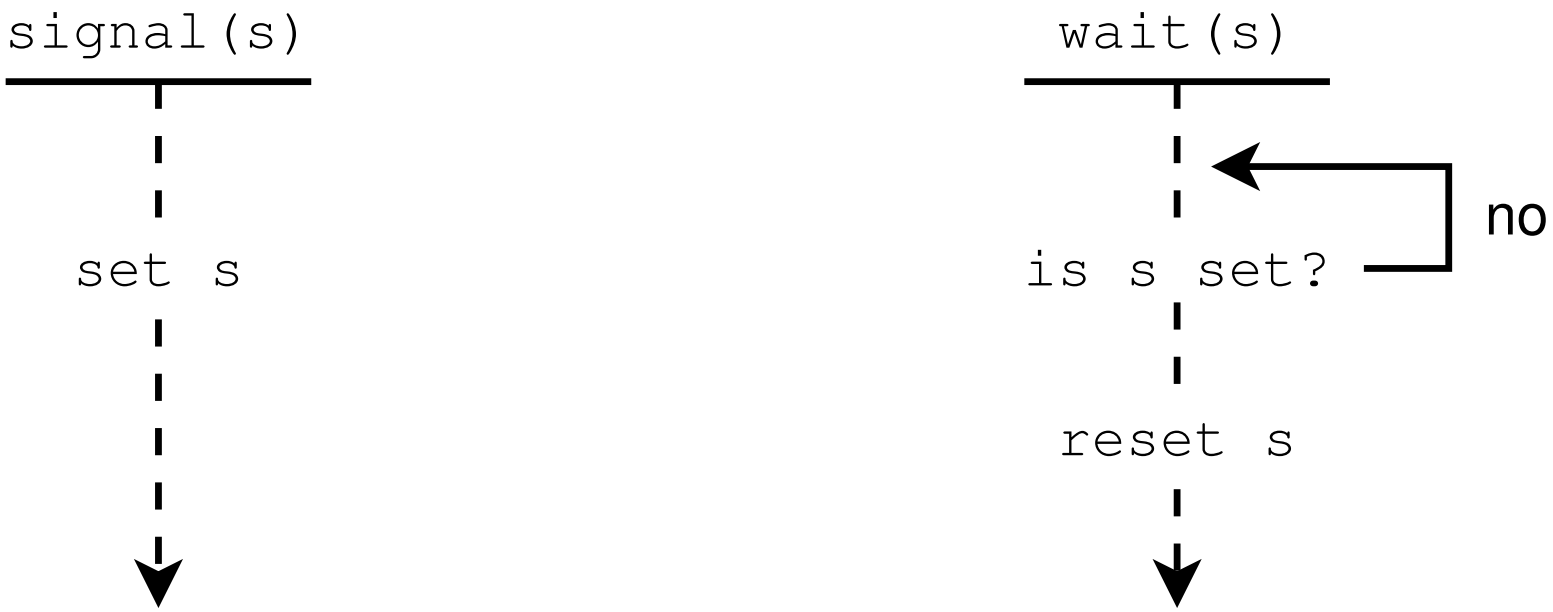


Signalisierung

- Eine Möglichkeit um Prozesse zu synchronisieren
- Mit Signalisierung wird eine **Ausführungsreihenfolge** festgelegt
- Beispiel: Abschnitt **X** von Prozess P_A soll **vor** Abschnitt **Y** von Prozess P_B ausgeführt werden
  - Die Operation `signal` signalisiert, wenn Prozess P_A den Abschnitt **X** abgearbeitet hat
  - Prozess P_B muss eventuell auf das Signal von Prozess P_A warten



Einfachste Form der Signalisierung (aktives Warten)

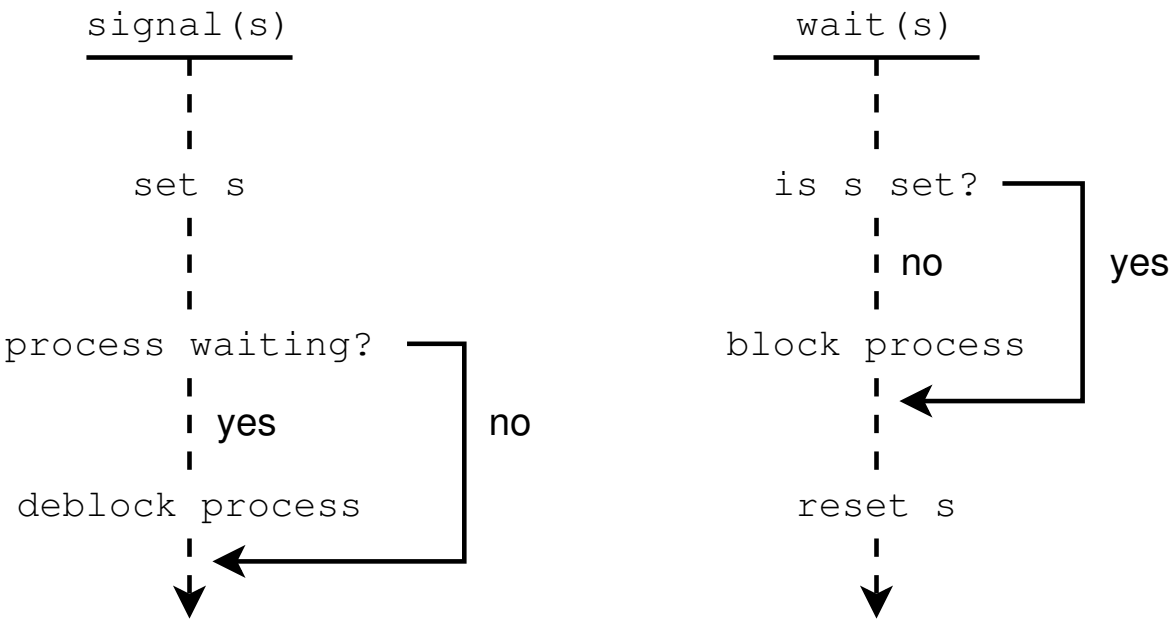


- Die Abbildung zeigt **aktives Warten** an der Signalvariable `s`
  - Die Signalvariable kann sich zum Beispiel in einer lokalen Datei befinden
  - Nachteil: Rechenzeit der CPU wird verschwendet, weil die `wait`-Operation den Prozessor in regelmäßigen Abständen belegt
- Diese Technik heißt auch **Warteschleife** oder **Spinlock**

Das aktive Warten heißt in der Literatur auch *Busy Waiting* oder *Polling*

Signalisieren und Warten

- Besseres Konzept: Prozess P_B blockieren, bis Prozess P_A den Abschnitt **X** abgearbeitet hat
  - Vorteil: Vergeudet keine Rechenzeit des Prozessors
  - Nachteil: Es kann nur ein Prozess warten
  - Diese Technik heißt in der Literatur auch **passives Warten**

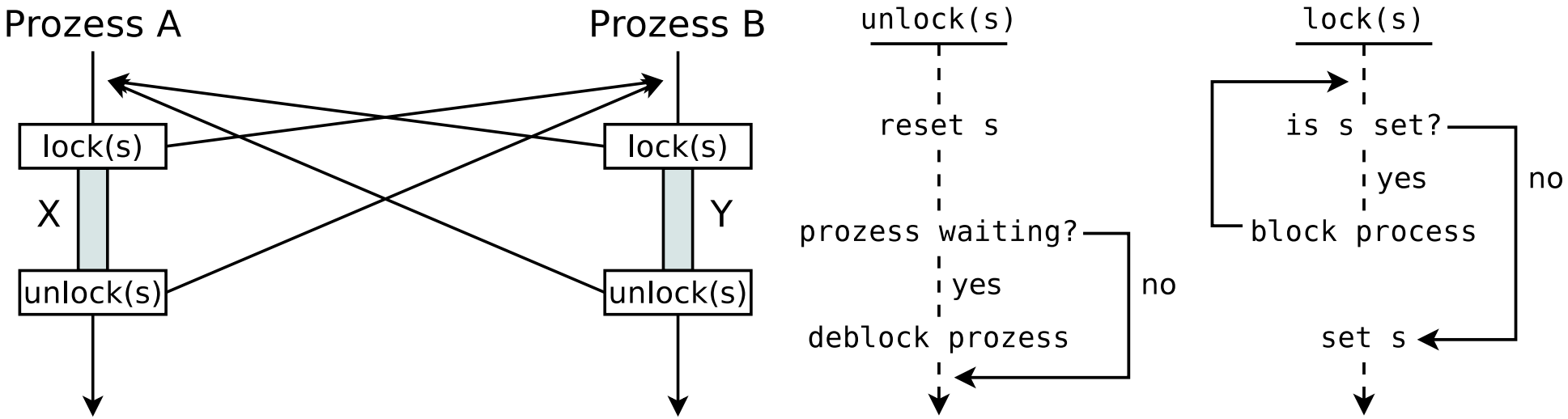


Eine Möglichkeit, um unter Linux eine Ausführungsreihenfolge mit passivem Warten festzulegen, ist die Funktion `sigsuspend`. Damit blockiert sich ein Prozess so lange selbst, bis ein anderer Prozess ihm mit der Funktion `kill` (oder dem gleichnamigen Systemaufruf) ein passendes Signal (meist `SIGUSR1` oder `SIGUSR2`) sendet und somit signalisiert, dass er weiterarbeiten soll.

Alternative Systemaufrufe und Funktionsaufrufe, mit denen sich ein Prozess selbst so lange blockieren kann, bis er durch einen Systemaufruf wieder geweckt wird, sind `pause` und `sleep`.

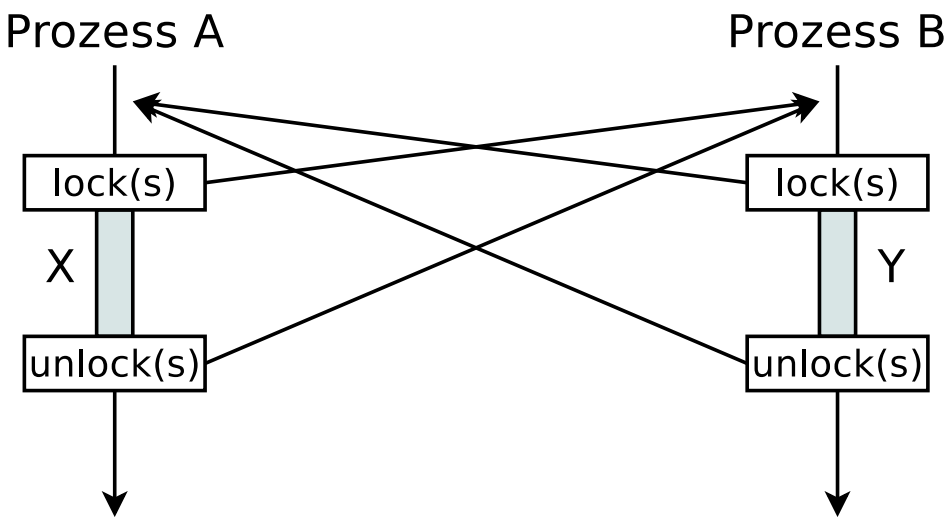
Schutz kritischer Abschnitte durch Sperren / Blockieren

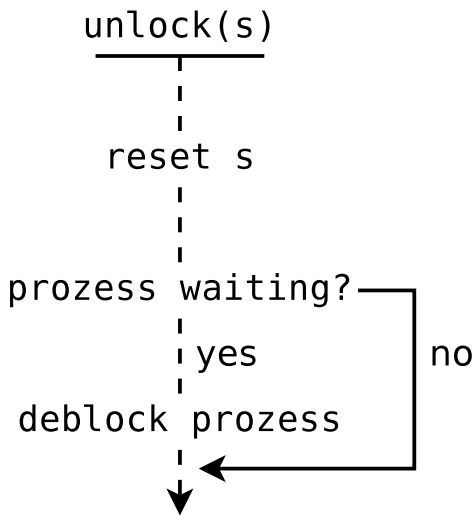
- Beim Signalisieren wird immer eine Ausführungsreihenfolge festlegt
  - Soll aber einfach nur sichergestellt werden, dass es **keine Überlappung** in der Ausführung der kritischen Abschnitte gibt, können die beiden Operationen `lock` und `unlock` eingesetzt werden

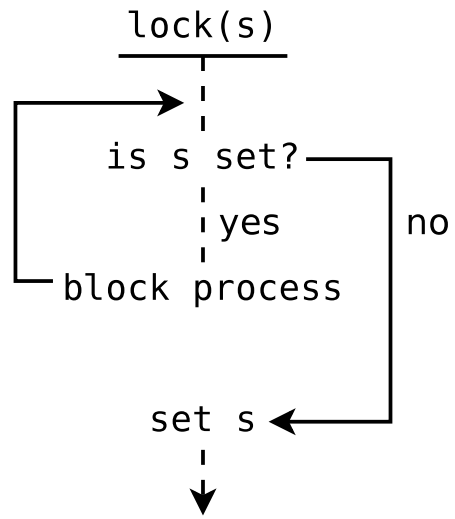


- Sperren (Blockieren) vermeidet Überlappungen bei der Abarbeitung von 2 kritischen Abschnitten
  - Beispiel: Kritische Abschnitte **X** von Prozess P_A und **Y** von Prozess P_B

Sperren und Freigeben von Prozessen unter Linux (1/2)







Hilfreiche Systemaufrufen und Bibliotheksfunktion um die Operationen `lock` und `unlock` unter Linux zu realisieren

`sigsuspend`, `kill`, `pause` und `sleep`

- Alternative 1: Realisierung von Sperren mit den Signalen `SIGSTOP` (Nr. 19) und `SIGCONT` (Nr. 18)
  - Mit `SIGSTOP` kann ein anderer Prozess gestoppt werden
  - Mit `SIGCONT` kann ein anderer Prozess reaktiviert werden

Sperren und Freigeben von Prozessen unter Linux (2/2)

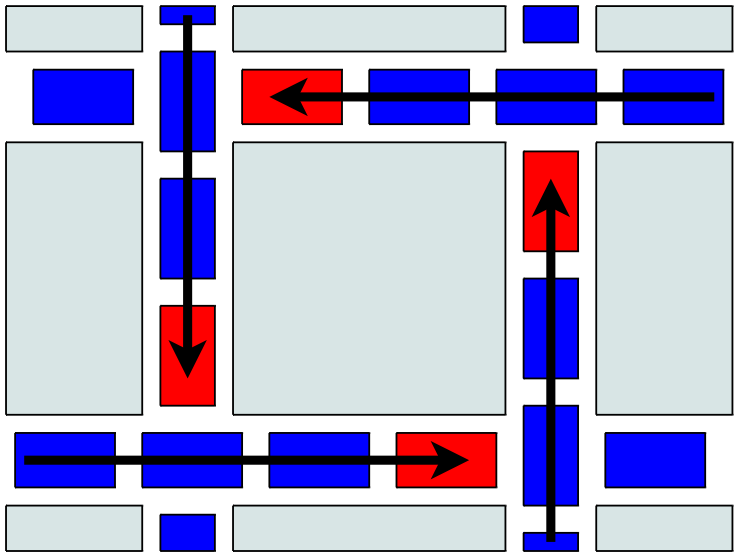
- Alternative 2: Eine lokale Datei dient als Sperrmechanismus für wechselseitigen Ausschluss
  - Jeder Prozess prüft vor dem Eintritt in seinen kritischen Abschnitt, ob er die Datei exklusiv öffnen kann
    - z.B. mit dem Systemaufruf `open` oder der Bibliotheksfunktion `fopen`
  - Ist das nicht der Fall, muss er für eine bestimmte Zeit pausieren (z.B. mit dem Systemaufruf `sleep`) und es danach erneut versuchen (**aktives Warten**)
    - Alternativ kann er sich mit `sleep` oder `pause` selbst pausieren und hoffen, dass der Prozess, der bereits die Datei geöffnet hat ihn nach Abschluss seines kritischen Abschnitts mit einem Signal deblockiert (**passives Warten**)

Zusammenfassung: Unterschied zwischen Signalisieren und Blockieren

- **Signalisieren** legt die Ausführungsreihenfolge fest
  Beispiel: Abschnitt X von Prozess P_A vor Abschnitt Y von P_B ausführen
- **Sperren / Blockieren** sichert kritische Abschnitte
  Die Reihenfolge, in der die Prozesse ihre kritische Abschnitte abarbeiten, ist nicht festgelegt! Es wird nur sichergestellt, dass es keine Überlappung in der Ausführung der kritischen Abschnitte gibt

Probleme, die durch Blockieren entstehen

- **Verhungern** (Starvation)
  - Hebt ein Prozess eine Sperre nicht wieder auf, müssen die anderen Prozesse unendlich lange auf die Freigabe warten
- **Verklemmung** (Deadlock)
  - Es warten mehrere Prozesse gegenseitig auf die von ihnen gesperrten Ressourcen, sperren sie sich gegenseitig
  - Da alle am Deadlock beteiligten Prozesse (ewig) warten, kann keiner ein Ereignis auslösen, dass die Situation auflöst



Quelle: `https://i.redd.it/vvu6v8pxvue11.jpg`
(Autor und Lizenz: unbekannt)

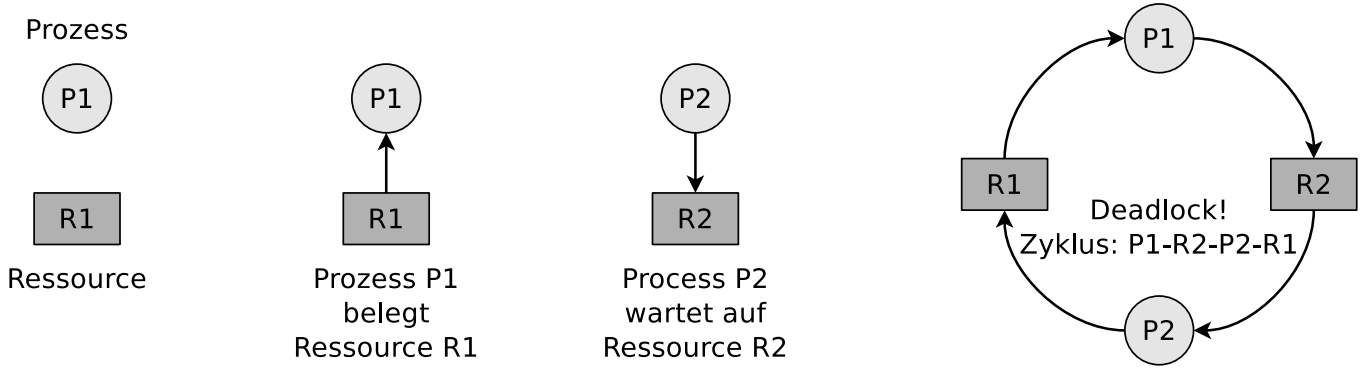
Bedingungen für Deadlocks

System Deadlocks. E. G. Coffman, M. J. Elphick, A. Shoshani. Computing Surveys, Vol. 3, No. 2, June 1971, S.67-78.
`http://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf`

- Damit ein Deadlock entstehen kann, müssen folgende Bedingungen gleichzeitig erfüllt sein:
  - **Wechselseitiger Ausschluss** (*mutual exclusion*)
    - Mindestens 1 Ressource wird von genau einem Prozess belegt oder ist verfügbar $\Longrightarrow$ nicht gemeinsam nutzbar (*non-sharable*)
  - **Anforderung weiterer Betriebsmittel** (*hold and wait*)
    - Ein Prozess, der bereits mindestens 1 Ressource belegt, fordert weitere Ressourcen an, die von einem anderen Prozess belegt sind
  - **Ununterbrechbarkeit** (*no preemption*)
    - Die Ressourcen, die ein Prozess besitzt, können nicht vom Betriebssystem entzogen, sondern nur durch ihn selbst freigegeben werden
  - **Zyklische Wartebedingung** (*circular wait*)
    - Es gibt eine zyklische Kette von Prozessen
    - Jeder Prozess fordert eine Ressource an, die der nächste Prozess in der Kette besitzt
- Fehlt eine Bedingung, ist ein Deadlock unmöglich

Betriebsmittel-Graphen

- Mit gerichteten Graphen können die Beziehungen von Prozessen und Ressourcen dargestellt werden
- So lassen sich auch Deadlocks modellieren
  - Die Knoten sind...
    - **Prozesse**: Sind als Kreise dargestellt
    - **Ressourcen**: Sind als Rechtecke dargestellt
  - Eine Kante von einem Prozess zu einer Ressource heißt:
    - Der Prozess ist blockiert, weil er auf die Ressource wartet
  - Eine Kante von einer Ressource zu einem Prozess heißt:
    - Der Prozess belegt die Ressource



Eine umfangreiche Beschreibung zu Betriebsmittel-Graphen enthält das Buch **Betriebssysteme – Eine Einführung**, *Uwe Baumgarten, Hans-Jürgen Siegert*, 6.Auflage, Oldenbourg Verlag (2007), Kapitel 6

Deadlock-Erkennung mit Matrizen

- Ein Nachteil der Deadlock-Erkennung mit Betriebsmittel-Graphen ist, dass man damit nur einzelne Ressourcen darstellen kann
  - Gibt es mehrere Kopien (Instanzen) einer Ressource, sind Graphen zur Darstellung bzw. Erkennung von Deadlocks ungeeignet
    - Existieren von einer Ressource mehrere Instanzen, kann ein matrizenbasiertes Verfahren verwendet werden, das 2 Vektoren und 2 Matrizen benötigt
- Wir definieren 2 Vektoren
  - **Ressourcenvektor** (*Existing Resource Vektor*)
    - Zeigt an, wie viele Ressourcen von jeder Klasse existieren
  - **Ressourcenrestvektor** (*Available Resource Vektor*)
    - Zeigt an, wie viele Ressourcen von jeder Klasse frei sind
- Zusätzlich sind 2 Matrizen nötig
  - **Belegungsmatrix** (*Current Allocation Matrix*)
    - Zeigt an, welche Ressourcen die Prozesse aktuell belegen
  - **Anforderungsmatrix** (*Request Matrix*)
    - Zeigt an, welche Ressourcen die Prozesse gerne hätten

Deadlock-Erkennung mit Matrizen – Beispiel (1/2)

Quelle des Beispiels: Tanenbaum. Moderne Betriebssysteme. Pearson. 2009

$$\text{Ressourcenvektor} = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

- 4 Ressourcen von Klasse 1 existieren
- 2 Ressourcen von Klasse 2 existieren
- 3 Ressourcen von Klasse 3 existieren
- 1 Ressource von Klasse 4 existiert

$$\text{Ressourcenrestvektor} = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

- 2 Ressourcen von Klasse 1 sind frei
- 1 Ressource von Klasse 2 ist frei
- Keine Ressourcen von Klasse 3 sind frei
- Keine Ressourcen von Klasse 4 sind frei

$$\text{Belegungsmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

- Prozess 1 belegt 1 Ressource von Klasse 3
- Prozess 2 belegt 2 Ressourcen von Klasse 1 und 1 Ressource von Klasse 4
- Prozess 3 belegt 1 Ressource von Klasse 2 und 2 Ressourcen von Klasse 3

$$\text{Anforderungsmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

- Prozess 1 ist blockiert, weil keine Ressource von Klasse 4 frei ist
- Prozess 2 ist blockiert, weil keine Ressource von Klasse 3 frei ist
- **Prozess 3 ist nicht blockiert**

Deadlock-Erkennung mit Matrizen – Beispiel (2/2)

- Wurde Prozess 3 fertig ausgeführt, gibt er seine Ressourcen frei

$$\text{Ressourcenrestvektor} = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix} \qquad \text{Anforderungsmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ - & - & - & - \end{bmatrix}$$

- 2 Ressourcen von Klasse 1 sind frei
- 2 Ressourcen von Klasse 2 sind frei
- 2 Ressourcen von Klasse 3 sind frei
- Keine Ressourcen von Klasse 4 sind frei
- Prozess 1 kann nicht laufen, weil keine Ressource vom Typ 4 frei ist
- **Prozess 2 ist nicht blockiert**
- Wurde Prozess 2 fertig ausgeführt, gibt er seine Ressourcen frei

$$\text{Ressourcenrestvektor} = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix} \qquad \text{Anforderungsmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$$

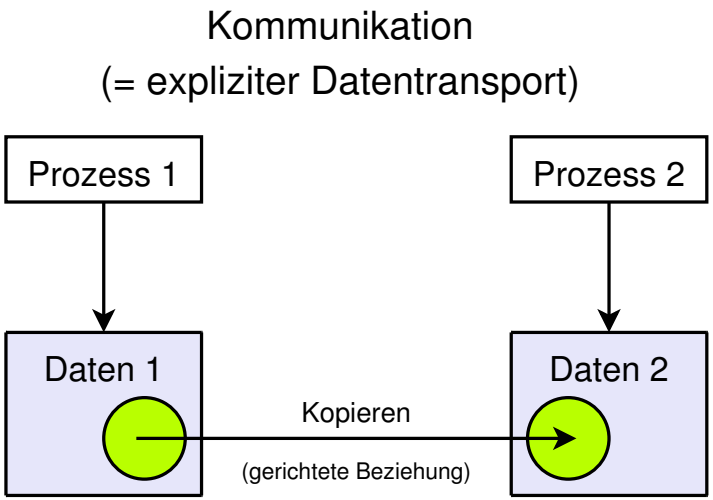
- **Prozess 1 ist nicht blockiert** $\Longrightarrow$ kein Deadlock in diesem Beispiel

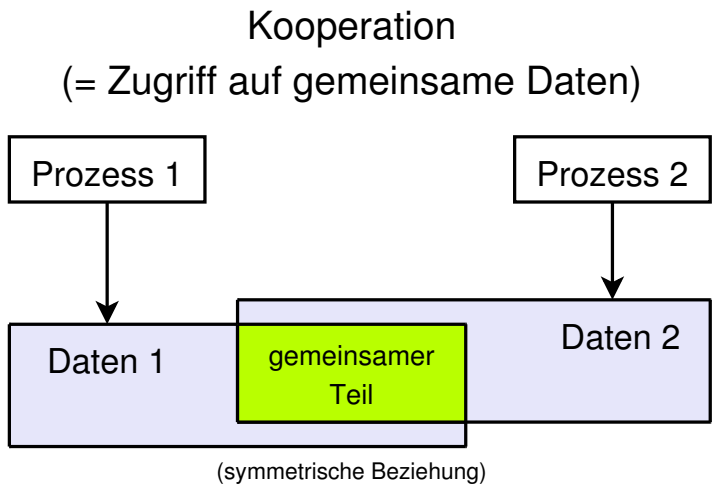
Fazit zu Deadlocks

- Manchmal wird die Möglichkeit von Deadlocks akzeptiert
  - Entscheidend ist, wie wichtig ein System ist
    - Ein Deadlock, der statistisch alle 5 Jahre auftritt, ist kein Problem in einem System das wegen Hardwareausfällen oder sonstigen Softwareproblemen jede Woche ein mal abstürzt
- Deadlock-Erkennung ist aufwendig und verursacht Overhead
- In allen Betriebssystemen sind Deadlocks möglich:
  - Prozesstabelle voll
    - Es können keine neuen Prozesse erzeugt werden
  - Maximale Anzahl von Inodes vergeben
    - Es können keine neuen Dateien und Verzeichnisse angelegt werden
- Die Wahrscheinlichkeit, dass so etwas passiert, ist gering, aber $\neq 0$
  - Solche potentiellen Deadlocks werden akzeptiert, weil ein gelegentlicher Deadlock nicht so lästig ist, wie die ansonsten nötigen Einschränkungen (z.B. nur 1 laufender Prozess, nur 1 offene Datei, mehr Overhead)

Kommunikation von Prozessen

- Kommunikation
  - Gemeinsamer Speicher (Shared Memory)
  - Nachrichtenwarteschlangen (Message Queues)
  - Pipes
  - Sockets





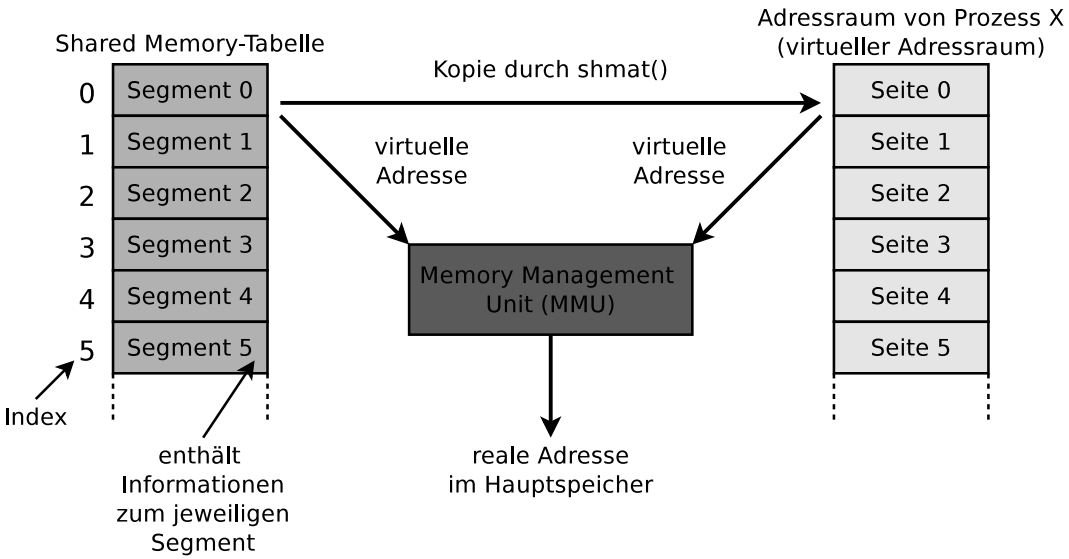
Gemeinsamer Speicher – Shared Memory

- Prozesskommunikation über einen gemeinsamen Speicher (Shared Memory) heißt auch **speicherbasierte Kommunikation**
- **Gemeinsame Speichersegmente** sind Speicherbereiche, auf die mehrere Prozesse direkt zugreifen können
  - Diese Speicherbereiche liegen im Adressraum mehrerer Prozesse
- Die Prozesse müssen die Zugriffe selbst koordinieren und sicherstellen, dass ihre Speicherzugriffe sich gegenseitig ausschließen
  - Der Empfänger-Prozess darf nichts aus dem gemeinsamen Speicher lesen, bevor der Sender-Prozess fertig geschrieben hat
  - Ist die Koordinierung der Zugriffe nicht sorgfältig $\Longrightarrow$ Inkonsistenzen

Bei den anderen Formen der Interprozesskommunikation garantiert das Betriebssystem die Synchronisation der Zugriffe

Gemeinsamer Speicher unter Linux/UNIX

- Unter Linux/UNIX speichert eine **Shared Memory Tabelle** mit Informationen über die existierenden gemeinsamen Speichersegmente
  - Zu diesen Informationen gehören: Anfangsadresse im Speicher, Größe, Besitzer (Benutzername und Gruppe) und Zugriffsrechte



- Ein gemeinsames Speichersegment wird immer über seine Indexnummer in der Shared Memory-Tabelle angesprochen

- Vorteil:
  - Ein gemeinsames Speichersegment, das an keinen Prozess gebunden ist, wird nicht automatisch vom Betriebssystem gelöscht

Mit gemeinsamem Speicher arbeiten

Linux/UNIX-Betriebssysteme stellen 4 Systemaufrufe für die Arbeit mit gemeinsamem Speicher bereit

- `shmget()`: Gemeinsames Speichersegment erzeugen oder auf ein bestehendes zugreifen
- `shmat()`: Gemeinsames Speichersegment an einen Prozess anhängen
- `shmdt()`: Gemeinsames Speichersegment von einem Prozess lösen/freigeben
- `shmctl()`: Status (u.a. Zugriffsrechte) eines gemeinsamen Speichersegments abfragen, ändern oder es löschen

Ein Beispiel zur Arbeit mit gemeinsamen Speicherbereichen unter Linux finden sie zum Beispiel auf der Webseite der Vorlesung

`ipcs`

Informationen über bestehende gemeinsame Speichersegmente liefert das Kommando `ipcs`

Gemeinsames Speichersegment erzeugen (in C)

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget;

    // Gemeinsames Speichersegment erzeugen
    // IPC_CREAT = Speichersegment erzeugen, wenn es noch nicht existiert
    // 0600 = Zugriffsrechte auf das neue gemeinsame Speichersegment
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);

    if (returncode_shmget < 0) {
        printf("Das gemeinsame Speichersegment konnte nicht erstellt werden.\n");
        perror("shmget");
    } else {
        printf("Das gemeinsame Speichersegment wurde erstellt.\n");
    }
}
```

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

0x00003039 56393780   bnc        600        20         0

$ printf "%d\n" 0x00003039        # Umrechnen von Hexadezimal in Dezimal
12345
```

Gemeinsames Speichersegment anhängen (in C)

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget;
    char *sharedmempointer;

    // Gemeinsames Speichersegment erzeugen
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
    ...

        // Gemeinsames Speichersegment anhängen
        sharedmempointer = shmat(returncode_shmget, 0, 0);
        if (sharedmempointer==(char *)-1) {
            printf("Das gemeinsame Speichersegment konnte nicht angehängt werden.\n");
            perror("shmat");
        } else {
            printf("Das Segment wurde angehängt an Adresse %p\n", sharedmempointer);
        }
    }
}
```

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00003039 56393780   bnc        600        20         1
```

In ein Speichersegment schreiben und daraus lesen (in C)

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget, returncode_shmdt, returncode_sprintf;
    char *sharedmempointer;

    // Gemeinsames Speichersegment erzeugen
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
    ...
        // Gemeinsames Speichersegment anhängen
        sharedmempointer = shmat(returncode_shmget, 0, 0);
        ...

        // Eine Zeichenkette in das gemeinsame Speichersegment schreiben
        returncode_sprintf = sprintf(sharedmempointer, "Hallo Welt.");
        if (returncode_sprintf < 0) {
            printf("Der Schreibzugriff ist fehlgeschlagen.\n");
        } else {
            printf("%i Zeichen in das Segment geschrieben.\n", returncode_sprintf);
        }

        // Die Zeichenkette im gemeinsamen Speichersegment ausgeben
        if (printf ("%s\n", sharedmempointer) < 0) {
            printf("Der Lesezugriff ist fehlgeschlagen.\n");
        }
    ...
```

Gemeinsames Speichersegment lösen (in C)

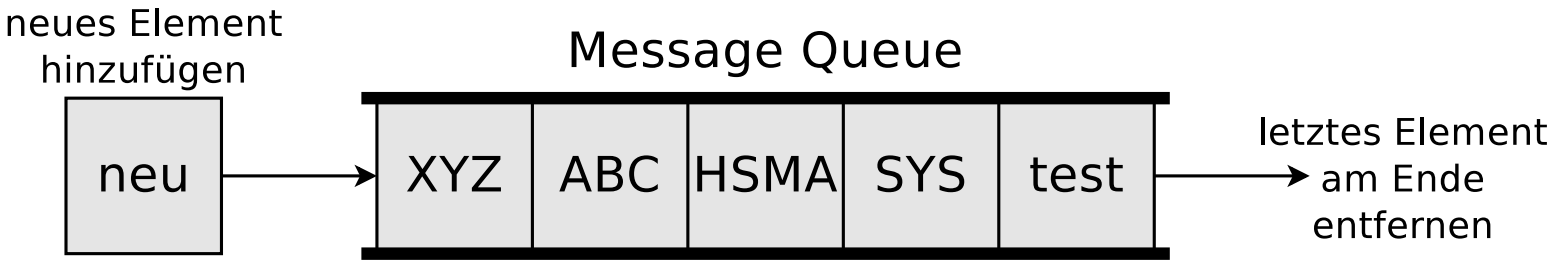
```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget;
    int returncode_shmdt;
    char *sharedmempointer;

    // Gemeinsames Speichersegment erzeugen
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
    ...

        // Gemeinsames Speichersegment anhängen
        sharedmempointer = shmat(returncode_shmget, 0, 0);
        ...

        // Gemeinsames Speichersegment lösen
        returncode_shmdt = shmdt(sharedmempointer);
        if (returncode_shmdt < 0) {
            printf("Das gemeinsame Speichersegment konnte nicht gelöst werden.\n");
            perror("shmdt");
        } else {
            printf("Das Segment wurde vom Prozess gelöst.\n");
        }
    }
}
```

Gemeinsames Speichersegment löschen (in C)

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

int main(int argc, char **argv) {
    int shared_memory_id = 12345;
    int returncode_shmget;
    int returncode_shmctl;
    char *sharedmempointer;

    // Gemeinsames Speichersegment erzeugen
    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
    ...

        // Gemeinsames Speichersegment löschen
        returncode_shmctl = shmctl(returncode_shmget, IPC_RMID, 0);
        if (returncode_shmctl == -1) {
            printf("Das gemeinsame Speichersegment konnte nicht gelöscht werden.\n");
            perror("semctl");
        } else {
            printf("Das Segment wurde gelöscht.\n");
        }
    }
}
```

Nachrichtenwarteschlangen - Message Queues

- Sind verketteten Listen mit Nachrichten
- Arbeiten nach dem Prinzip FIFO
- Prozesse können Daten darin ablegen und daraus abholen
- Vorteil: Auch nach Beendigung des Erzeuger-Prozesses verbleiben die Daten in der Nachrichtenwarteschlange



Linux/UNIX-Betriebssysteme stellen 4 Systemaufrufe für die Arbeit mit Nachrichtenwarteschlangen bereit

- `msgget()`: Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
- `msgsnd()`: Nachrichten in Nachrichtenwarteschlange schreiben (schicken)
- `msgrcv()`: Nachrichten aus Nachrichtenwarteschlange lesen (empfangen)
- `msgctl()`: Status (u.a. Zugriffsrechte) einer Nachrichtenwarteschlang abfragen, ändern oder sie löschen

Informationen über bestehende Nachrichtenwarteschlangen liefert das Kommando `ipcs`

Nachrichtenwarteschlangen erzeugen (in C)

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>

int main(int argc, char **argv) {
    int returncode_msgget;

    // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
    // IPC_CREAT => neue Nachrichtenwarteschlange erzeugen, wenn sie noch nicht existiert
    // 0600 = Zugriffsrechte auf die neue Nachrichtenwarteschlange
    returncode_msgget = msgget(12345, IPC_CREAT | 0600);
    if(returncode_msgget < 0) {
        printf("Die Nachrichtenwarteschlange konnte nicht erstellt werden.\n");
        exit(1);
    } else {
        printf("Die Nachrichtenwarteschlange 12345 mit der ID %i ist nun verfügbar.\n",
            returncode_msgget);
    }
}
```

```
$ ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00003039 98304      bnc        600        0            0

$ printf "%d\n" 0x00003039        # Umrechnen von Hexadezimal in Dezimal
12345
```

In Nachrichtenwarteschlangen schreiben (in C)

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>
#include <string.h>                         // Diese Header-Datei ist nötig für strcpy()

struct msgbuf {                             // Template eines Puffers fuer msgsnd und msgrcv
    long mtype;                             // Nachrichtentyp
    char mtext[80];                         // Sendepuffer
} msg;

int main(int argc, char **argv) {
    int returncode_msgget;

    // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
    returncode_msgget = msgget(12345, IPC_CREAT | 0600);
    ...

    msg.mtype = 1;                          // Nachrichtentyp festlegen
    strcpy(msg.mtext, "Testnachricht");     // Nachricht in den Sendepuffer schreiben

    // Eine Nachricht in die Nachrichtenwarteschlange schreiben
    if (msgsnd(returncode_msgget, &msg, strlen(msg.mtext), 0) == -1) {
        printf("In die Nachrichtenwarteschlange konnte nicht geschrieben werden.\n");
        exit(1);
    }
}
```

- Den Nachrichtentyp (eine positive ganze Zahl) definiert der Benutzer

Ergebnis des Schreibens in die Nachrichtenwarteschlange

- Vorher...

```
$ ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00003039 98304      bnc        600        0            0
```

- Nachher...

```
$ ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00003039 98304      bnc        600        80           1
```

Aus Nachrichtenwarteschlangen lesen (in C)

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>
#include <string.h>                 // Diese Header-Datei ist nötig für strcpy()
typedef struct msgbuf {             // Template eines Puffers fuer msgsnd und msgrcv
    long mtype;                     // Nachrichtentyp
    char mtext[80];                 // Sendepuffer
} msg;

int main(int argc, char **argv) {
    int returncode_msgget, returncode_msgrcv;
    msg receivebuffer;              // Einen Empfangspuffer anlegen

    // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
    returncode_msgget = msgget(12345, IPC_CREAT | 0600)

    msg.mtype = 1;                  // Die erste Nachricht vom Typ 1 empfangen
    // MSG_NOERROR => Nachrichten abschneiden, wenn sie zu lang sind
    // IPC_NOWAIT  => Prozess nicht blockieren, wenn keine Nachricht vom Typ vorliegt
    returncode_msgrcv = msgrcv(returncode_msgget, &msg, sizeof(msg.mtext), msg.mtype,
         MSG_NOERROR | IPC_NOWAIT);
    if (returncode_msgrcv < 0) {
        printf("Aus der Nachrichtenwarteschlange konnte nicht gelesen werden.\n");
        perror("msgrcv");
    } else {
        printf("Diese Nachricht wurde aus der Warteschlange gelesen: %s\n", msg.mtext);
        printf("Die empfangene Nachricht ist %i Zeichen lang.\n", returncode_msgrcv);
    }
}
```

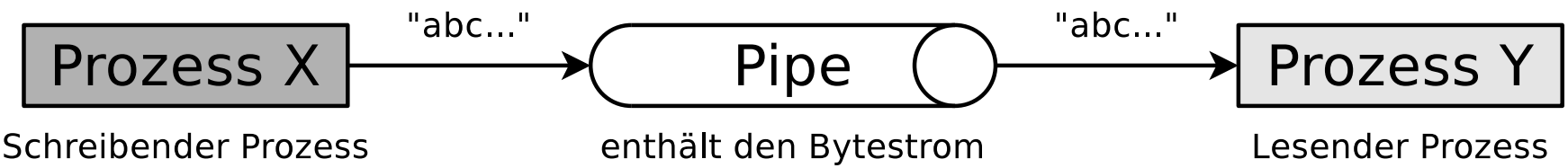
Nachrichtenwarteschlangen löschen (in C)

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>

int main(int argc, char **argv) {
    int returncode_msgget;
    int returncode_msgctl;

    // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
    returncode_msgget = msgget(12345, IPC_CREAT | 0600);
    ...

    // Nachrichtenwarteschlange löschen
    returncode_msgctl = msgctl(returncode_msgget, IPC_RMID, 0);
    if (returncode_msgctl < 0) {
        printf("Die Nachrichtenwarteschlange mit der ID %i konnte nicht gelöscht werden.\
            n", returncode_msgget);
        perror("msgctl");
        exit(1);
    } else {
        printf("Die Nachrichtenwarteschlange mit der ID %i wurde gelöscht.\n",
            returncode_msgget);
    }
    exit(0);
}
```

Ein Beispiel zur Arbeit mit Nachrichtenwarteschlangen unter Linux finden sie zum Beispiel auf der Webseite der Vorlesung

Pipes (1/2)

- Eine **anonyme Pipe**...
  - ist ein gepufferter unidirektionaler Kommunikationskanal zwischen 2 Prozessen
    - Soll Kommunikation in beide Richtungen gleichzeitig möglich sein, sind 2 Pipes nötig – eine für jede mögliche Kommunikationsrichtung
  - arbeitet nach dem FIFO-Prinzip
  - hat eine begrenzte Kapazität
    - Pipe = voll $\Longrightarrow$ der in die Pipe schreibende Prozess wird blockiert
    - Pipe = leer $\Longrightarrow$ der aus der Pipe lesende Prozess wird blockiert
  - wird mit dem Systemaufruf `pipe()` angelegt
    - Dabei erzeugt der Betriebssystemkern einen Inode ($\Longrightarrow$ Foliensatz 6) und 2 Zugriffskennungen (*Handles*)
    - Prozesse greifen auf die Zugriffskennungen mit `read()` und `write()`-Systemaufrufen (oder Bibliotheksfunktionen) zu, um Daten aus der Pipe zu lesen bzw. um Daten in die Pipe zu schreiben



Pipes (2/2)

- Bei der Erzeugung von Kindprozessen mit `fork()` erben die Kindprozesse auch den Zugriff auf die Zugriffskennungen
- **Anonyme Pipes** ermöglichen Prozesskommunikation nur zwischen eng verwandten Prozessen
  - Nur Prozesse, die via `fork()` eng verwandt sind, können über anonyme Pipes kommunizieren
  - Mit der Beendigung des letzten Prozesses, der Zugriff auf eine anonyme Pipe hat, wird diese vom Betriebssystem beendet
- Via **benannte Pipes** (Named Pipes), können auch nicht eng miteinander verwandte Prozesse kommunizieren
  - Auf diese Pipes kann mit Hilfe ihres Namens zugegriffen werden
    - Sie werden in C erzeugt via: `mkfifo("<pfadname>",<zugriffsrechte>)`
  - Jeder Prozess, der den Namen kennt, kann über diesen die Verbindung zur Pipe herstellen und darüber mit anderen Prozessen kommunizieren
- **Wechselseitigen Ausschluss** garantiert das Betriebssystem
  - Zu jedem Zeitpunkt kann nur 1 Prozess auf eine Pipe zugreifen

Übersicht der Pipes unter Linux/UNIX: `lsof | grep pipe`

Ein Beispiel zu anonymen Pipes (in C) – Teil 1/2

Ein Beispiel zur Arbeit mit benannten Pipes unter Linux finden sie zum Beispiel auf der Webseite der Vorlesung

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void main() {
  int pid_des_Kindes;
  // Zugriffskennungen zum Lesen (testpipe[0]) und Schreiben (testpipe[1]) anlegen
  int testpipe[2];

  // Die Pipe testpipe anlegen
  if (pipe(testpipe) < 0) {
    printf("Das Anlegen der Pipe ist fehlgeschlagen.\n");
    // Programmabbruch
    exit(1);
  } else {
    printf("Die Pipe testpipe wurde angelegt.\n");
  }

  // Einen Kindprozess erzeugen
  pid_des_Kindes = fork();

  // Es kam beim fork zu einem Fehler
  if (pid_des_Kindes < 0) {
    perror("Es kam bei fork zu einem Fehler!\n");
    // Programmabbruch
    exit(1);
  }
```

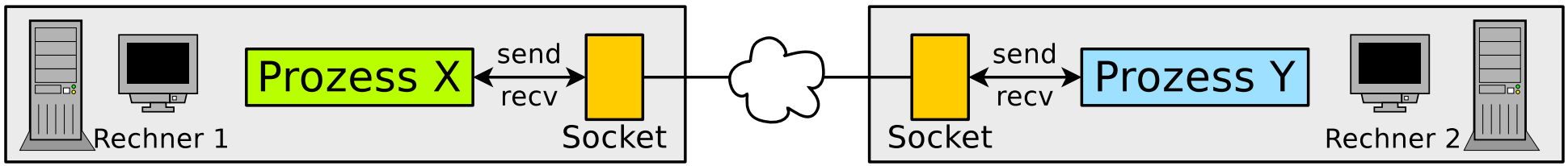
Ein Beispiel zu anonymen Pipes (in C) – Teil 2/2

```
28    // Elternprozess
29    if (pid_des_Kindes > 0) {
30      printf("Elternprozess: PID: %i\n", getpid());
31      // Lesekanal der Pipe testpipe blockieren
32      close(testpipe[0]);
33      char nachricht[] = "Testnachricht";
34      // Daten in den Schreibkanal der Pipe schreiben
35      write(testpipe[1], &nachricht, sizeof(nachricht));
36    }
37
38    // Kindprozess
39    if (pid_des_Kindes == 0) {
40      printf("Kindprozess: PID: %i\n", getpid());
41      // Schreibkanal der Pipe testpipe blockieren
42      close(testpipe[1]);
43      // Einen Empfangspuffer mit 80 Zeichen Kapazität anlegen
44      char puffer[80];
45      // Daten aus dem Lesekanal der Pipe auslesen
46      read(testpipe[0], puffer, sizeof(puffer));
47      // Empfangene Daten ausgeben
48      printf("Empfangene Daten: %s\n", puffer);
49    }
50  }
```

```
$ gcc pipe_beispiel.c -o pipe_beispiel
$ ./pipe_beispiel
Die Pipe testpipe wurde angelegt.
Elternprozess: PID: 6363
Kindprozess: PID: 6364
Empfangene Daten: Testnachricht
```

Sockets

- Vollduplexfähige Alternative zu Pipes und gemeinsamem Speicher
  - Ermöglichen Interprozesskommunikation in verteilten Systemen
- Ein Benutzerprozess kann einen Socket vom Betriebssystem anfordern, und über diesen anschließend Daten verschicken und empfangen
  - Das Betriebssystem verwaltet alle benutzten Sockets und die zugehörigen Verbindungsinformationen



- Zur Kommunikation über Sockets werden Ports verwendet
  - Die Vergabe der Portnummern erfolgt beim Verbindungsaufbau
  - Portnummern werden vom Betriebssystem zufällig vergeben
    - Ausnahmen sind Ports bekannter Anwendungen, wie z.B. HTTP (80), SMTP (25), Telnet (23), SSH (22), FTP (21),...
- Einsatz von Sockets ist blockierend (synchron) und nicht-blockierend (asynchron) möglich

Verschiedene Arten von Sockets

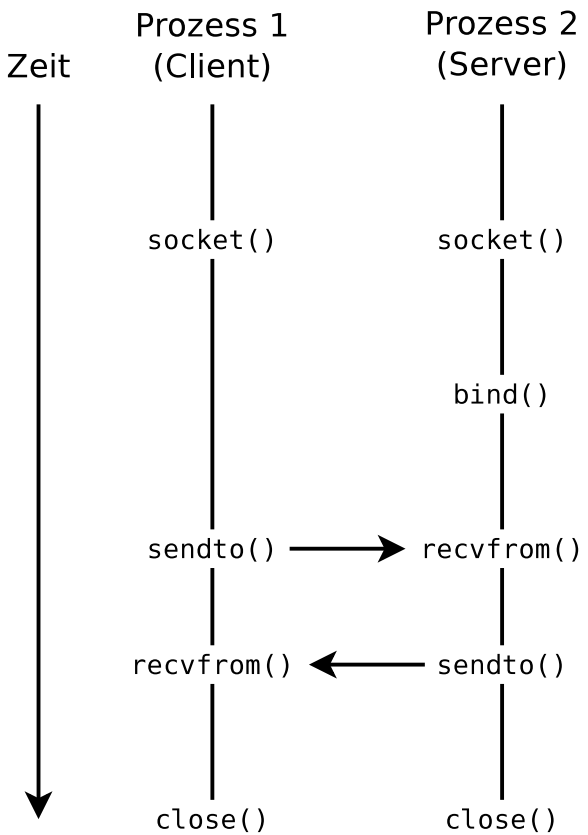
- **Verbindungslose Sockets** (bzw. **Datagram Sockets**)
  - Verwenden das Transportprotokoll UDP
  - Vorteil: Höhere Geschwindigkeit als bei TCP
    - Grund: Geringer Mehraufwand (Overhead) für das Protokoll
  - Nachteil: Segmente können einander überholen oder verloren gehen
- **Verbindungsorientierte Sockets** (bzw. **Stream Sockets**)
  - Verwenden das Transportprotokoll TCP
  - Vorteil: Höhere Verlässlichkeit
    - Segmente können nicht verloren gehen
    - Segmente kommen immer in der korrekten Reihenfolge an
  - Nachteil: Geringere Geschwindigkeit als bei UDP
    - Grund: Höherer Mehraufwand (Overhead) für das Protokoll

Sockets nutzen

- Praktisch alle gängigen Betriebssystemen unterstützen Sockets
  - Vorteil: Bessere Portabilität der Anwendungen
- Funktionen für Kommunikation via Sockets:
  - Erstellen eines Sockets:
    `socket()`
  - Anbinden eines Sockets an eine Portnummer und empfangsbereit machen:
    `bind()`, `listen()`, `accept()` und `connect()`
  - Senden/Empfangen von Nachrichten über den Socket:
    `send()`, `sendto()`, `recv()` und `recvfrom()`
  - Schließen eines Sockets:
    `shutdown()` oder `close()`

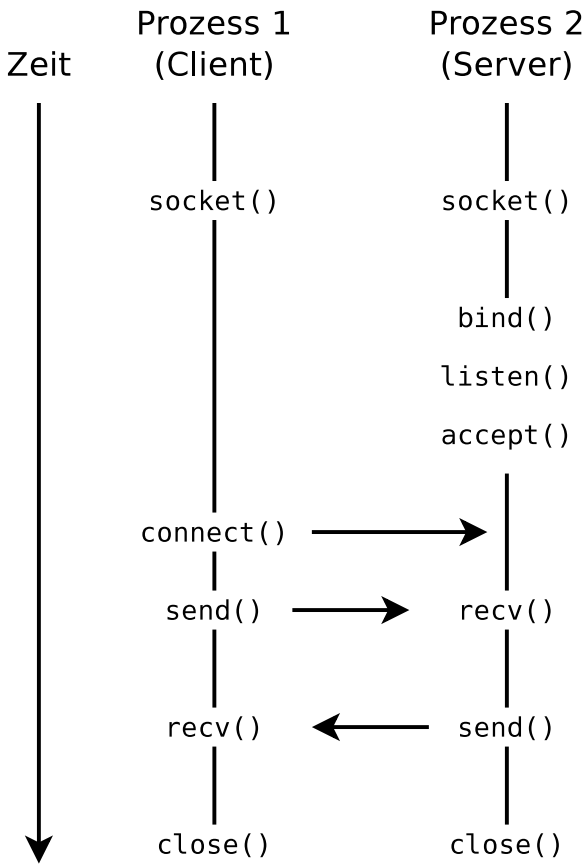
Übersicht der Sockets unter Linux/UNIX: `netstat -n` oder `lsof | grep socket`

Verbindungslose Kommunikation mit Sockets – UDP



- **Client**
  - Socket erstellen (`socket`)
  - Daten senden (`sendto`) und empfangen (`recvfrom`)
  - Socket schließen (`close`)
- **Server**
  - Socket erstellen (`socket`)
  - Socket an einen Port binden (`bind`)
  - Daten senden (`sendto`) und empfangen (`recvfrom`)
  - Socket schließen (`close`)

Verbindungsorientierte Kommunikation mit Sockets – TCP



- **Client**
  - Socket erstellen (`socket`)
  - Client mit Server-Socket verbinden (`connect`)
  - Daten senden (`send`) und empfangen (`recv`)
  - Socket schließen (`close`)
- **Server**
  - Socket erstellen (`socket`)
  - Socket an einen Port binden (`bind`)
  - Socket empfangsbereit machen (`listen`)
    - Richtete eine Warteschlange für Verbindungen mit Clients ein
  - Server akzeptiert Verbindungsanforderung (`accept`)
  - Daten senden (`send`) und empfangen (`recv`)
  - Socket schließen (`close`)

Einen Socket erzeugen: socket

```
int socket(int domain, int type, int protocol);
```

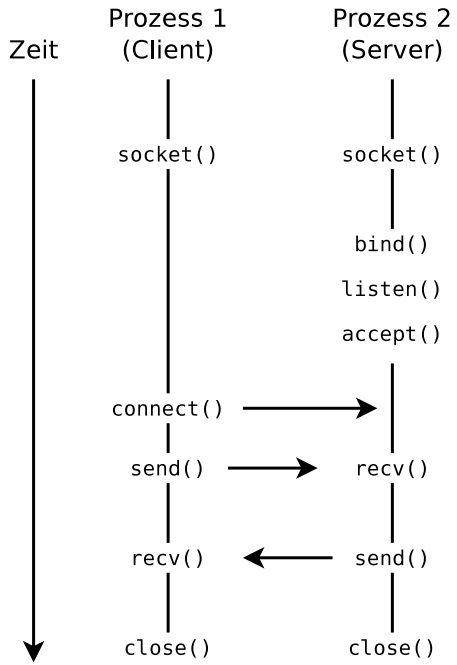
- Ein Aufruf von socket() liefert einen Integerwert zurück
  - Der Wert heißt **Socket-Deskriptor** (*socket file descriptor*)
- domain: Legt die Protokollfamilie fest
  - PF_UNIX: Lokale Prozesskommunikation unter Linux/UNIX
  - PF_INET: IPv4
  - PF_INET6: IPv6
- type: Legt den Typ des Sockets (und damit auch das Protokoll) fest:
  - SOCK_STREAM: Stream Socket (TCP)
  - SOCK_DGRAM: Datagram Socket (UDP)
  - SOCK_RAW: RAW-Socket (IP)
- Der Parameter protocol hat meist den Wert Null
- Einen Socket mit socket() erzeugen:

```
1 sd = socket(PF_INET, SOCK_STREAM, 0);
2     if (sd < 0) {
3         perror("Der Socket konnte nicht erzeugt werden");
4         return 1;
5     }
```

Adresse und Portnummer binden: `bind`

```
int bind(int sd, struct sockaddr *address, int addrlen);
```

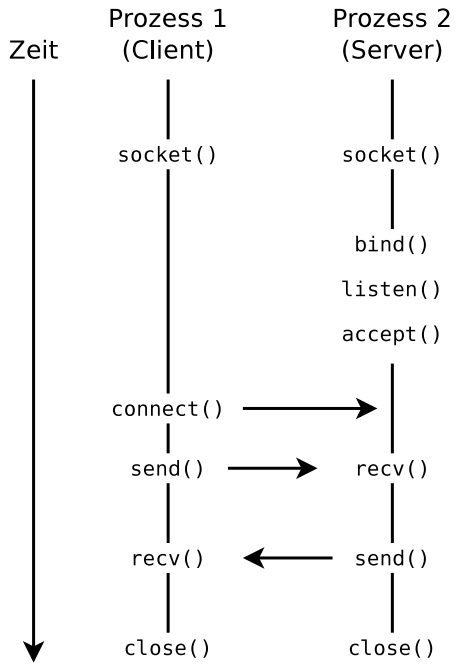
- `bind()` bindet den neu erstellen Socket (`sd`) an die Adresse (`address`) des Servers
  - `sd` ist der Socket-Deskriptor aus dem vorhergehenden Aufruf von `socket()`
  - `address` ist eine Datenstruktur, die die IP-Adresse des Server und eine Portnummer enthält
  - `addrlen` ist die Länge der Datenstruktur, die die IP-Adresse und Portnummer enthält



Server empfangsbereit machen: `listen`

```
int listen(int sd, int backlog);
```

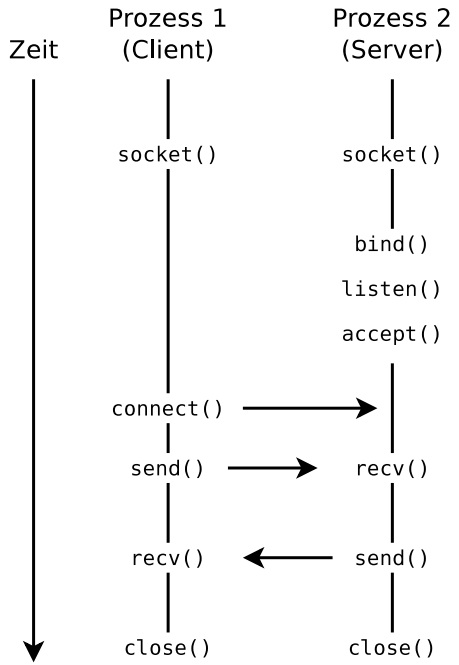
- `listen()` definiert, wie viele Verbindungsanfragen am Socket gepuffert werden können
  - Ist die `listen()`-Warteschlange voll, werden weitere Verbindungsanfragen von Clients abgewiesen
  - `sd` ist der Socket-Deskriptor aus dem vorhergehenden Aufruf von `socket()`
  - `backlog` enthält die Anzahl der möglichen Verbindungsanforderungen, die die Warteschlange maximal speichern kann
    - Standardwert: 5
  - Ein Server für Datagrame (UDP) braucht `listen()` nicht aufzurufen, da er keine Verbindungen zu Clients einrichtet



Eine Verbindungsanforderung akzeptieren: accept

```
int accept(int sd, struct sockaddr *address, int *addrlen);
```

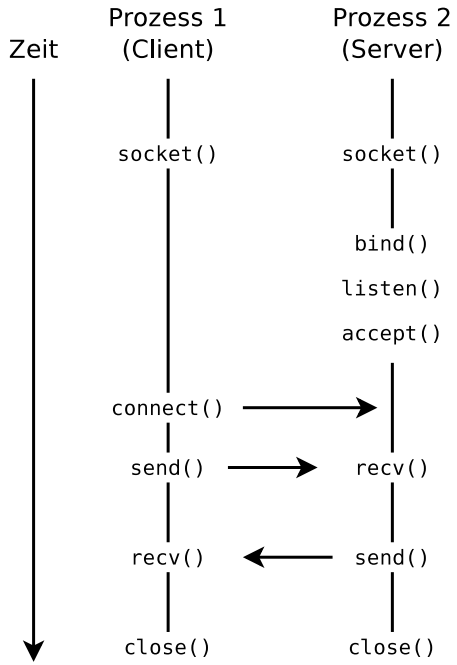
- Mit `accept()` holt der Server die erste Verbindungsanforderung aus der Warteschlange
- Der Rückgabewert ist der Socket-Deskriptor des neuen Sockets
- Enthält die Warteschlange keine Verbindungsanforderungen, ist der Prozess blockiert, bis eine Verbindungsanforderung eintrifft
- `address` enthält die Adresse des Clients
- Nachdem eine Verbindungsanforderungen mit `accept()` angenommen wurde, ist die Verbindung mit dem Client vollständig aufgebaut



Verbindung durch den Client herstellen

```
int connect(int sd, struct sockaddr *servaddr,
            socklen_t addrlen);
```

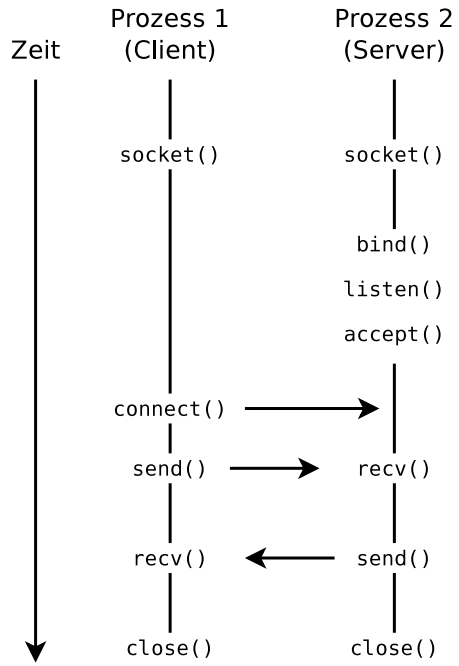
- Via `connect()` versucht der Client eine Verbindung mit einem Server-Socket herzustellen
- `sd` ist der Socket-Deskriptor
- `servaddr` ist die Adresse des Servers
- `addrlen` ist die Länge der Datenstruktur, die die Adresse enthält



Verbindungsorientierter Datenaustausch: send und recv

```
int send(int sd, char *buffer, int nbytes, int flags);
int recv(int sd, char *buffer, int nbytes, int flags);
```

- Mit `send()` und `recv()` werden über eine bestehende Verbindung Daten ausgetauscht
- `send()` sendet eine Nachricht (`buffer`) über den Socket (`sd`)
- `recv()` empfängt eine Nachricht vom Socket `sd` und legt diese in den Puffer (`buffer`)
- `sd` ist der Socket-Deskriptor
- `buffer` enthält die zu sendenden bzw. empfangenen Daten
- `nbytes` gibt die Anzahl der Bytes im Puffer an
- Der Wert von `flags` ist in der Regel Null



Verbindungsorientierter Datenaustausch: `read` und `write`

```
int read(int sd, char *buffer, int nbytes);
int write(int sd, char *buffer, int nbytes);
```

- Unter UNIX könnten im Normalfall auch `read()` und `write()` zum Empfangen und Senden über einen Socket verwendet werden
  - Der *Normalfall* ist, wenn der Parameter `flags` bei `send()` und `recv()` den Wert 0 hat
- Folgende Aufrufe haben das gleiche Ergebnis:

```
1 send(socket,"Hello World",11,0);
2 write(socket,"Hello World",11);
```

Verbindungsloser Datenaustausch: `sendto` und `recvfrom`

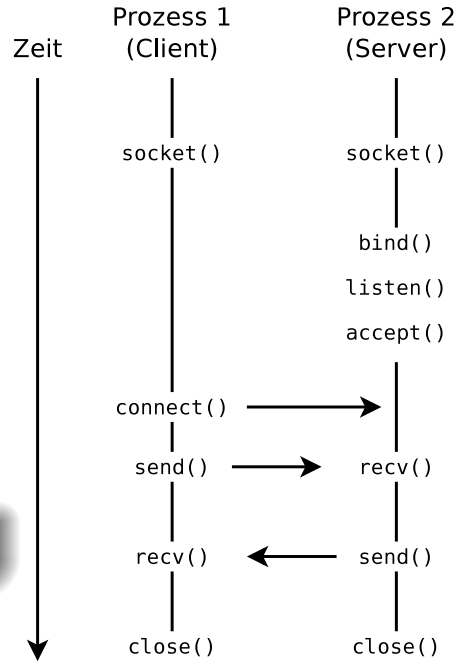
```
int sendto(int sd, char *buffer, int nbytes, int flags,
           struct sockaddr *to, int addrlen);
int recvfrom(int sd, char *buffer, int nbytes, int flags,
             struct sockaddr *from, int addrlen);
```

- Weiß ein Prozess, an welche Adresse (Host und Port), also an welchen Socket er Daten senden soll, verwendet er dafür `sendto()`
- `sendto()` übermittelt mit den Daten immer die lokale Adresse
- `sd` ist der Socket-Deskriptor
- `buffer` enthält die zu sendenden bzw. empfangenen Daten
- `nbytes` gibt die Anzahl der Bytes im Puffer an
- `to` enthält die Adresse des Empfängers
- `from` enthält die Adresse des Senders
- `addrlen` ist die Länge der Datenstruktur, die die Adresse enthält

Socket schließen: `close`

```
int shutdown(int sd, int how);
```

- `shutdown()` schließt eine bidirektionale Socket-Verbindung
- Der Parameter `how` legt fest, ob künftig keine Daten mehr empfangen werden sollen (`how=0`), keine mehr gesendet werden (`how=1`), oder beides (`how=2`)

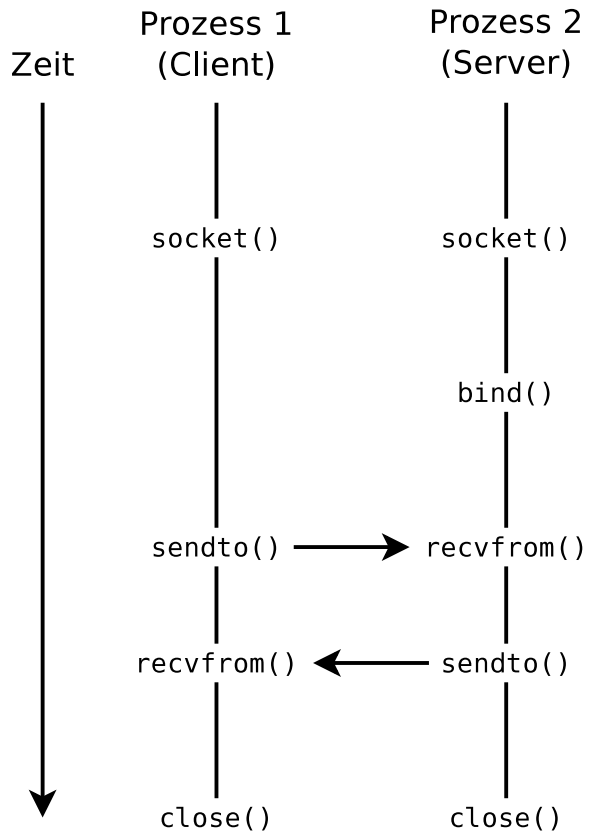



```
int close(int sd);
```

- Wird `close()` anstatt `shutdown()` verwendet, entspricht dies einem `shutdown(sd,2)`

Sockets via UDP – Beispiel (Server)

```
1  #!/usr/bin/env python
2  # -*- coding: iso-8859-15 -*-
3  # Server: Empfängt eine Nachricht via UDP
4
5  # Modul socket importieren
6  import socket
7
8  # Stellvertretend für alle Schnittstellen des Hosts
9  # '' = alle Schnittstellen
10 HOST = ''
11 # Portnummer des Servers
12 PORT = 50000
13
14 # Socket erzeugen und Socket-Deskriptor zurückliefern
15 sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
16
17 try:
18   sd.bind((HOST, PORT))            # Socket an Port
         binden
19   while True:
20     data = sd.recvfrom(1024)       # Daten empfangen
21     # Empfangene Daten ausgeben
22     print 'Received:', repr(data)
23 finally:
24     sd.close()                     # Socket schließen
```



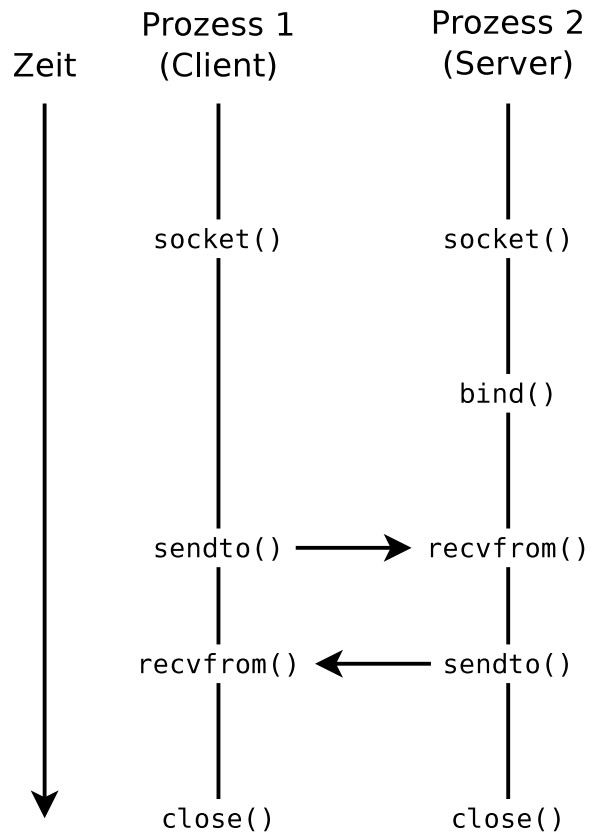

```
$ python udp_server.py
```

Sockets via UDP – Beispiel (Client)

```
1  #!/usr/bin/env python
2  # -*- coding: iso-8859-15 -*-
3  # Client: Schickt eine Nachricht via UDP
4
5  import socket               # Modul socket importieren
6
7  HOST = 'localhost'          # Hostname des Servers
8  PORT = 50000                # Portnummer des Servers
9  MESSAGE = 'Hello World'     # Nachricht
10
11 # Socket erzeugen und Socket-Deskriptor zurückliefern
12 sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
13
14 # Nachricht an Socket senden
15 sd.sendto(MESSAGE, (HOST, PORT))
16
17 sd.close()                  # Socket schließen
```

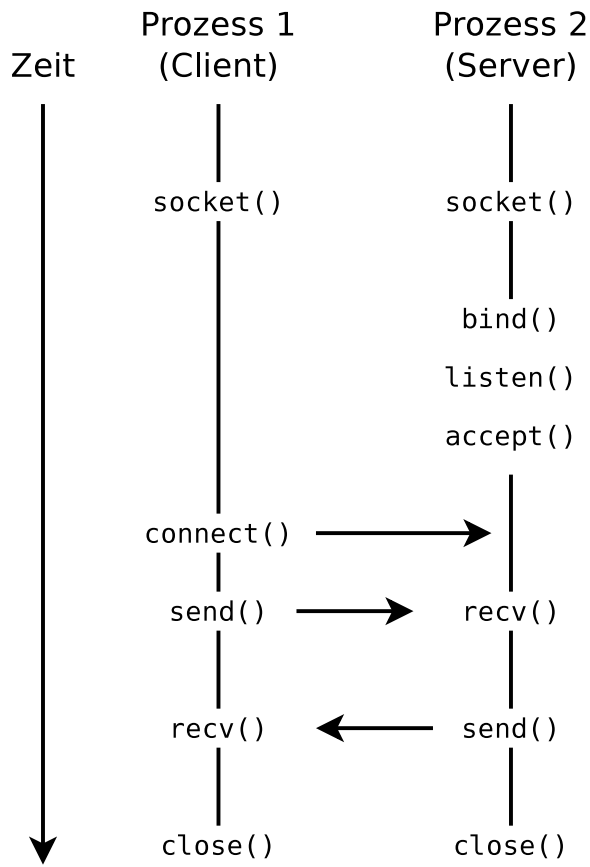
```
$ python udp_client.py
```

```
$ python udp_server.py
Received: ('Hello World', ('127.0.0.1', 39834))
```



Sockets via TCP – Beispiel (Server)

```
1  #!/usr/bin/env python
2  # -*- coding: iso-8859-15 -*-
3  # Echo Server via TCP
4  import socket               # Modul socket importieren
5  HOST = ''                   # '' = alle Schnittstellen
6  PORT = 50007                # Portnummer des Servers
7
8  # Socket erzeugen und Socket-Deskriptor zurückliefern
9  sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 # Socket an Port binden
11 sd.bind((HOST, PORT))
12 # Socket empfangsbereit machen
13 # Max. Anzahl Verbindungen = 1
14 sd.listen(1)
15 # Socket akzeptiert Verbindungen
16 conn, addr = sd.accept()
17
18 print 'Connected by', addr
19
20 while 1:                    # Endlosschleife
21     data = conn.recv(1024) # Daten empfangen
22     if not data: break     # Endlosschleife abbrechen
23     # Empfangene Daten zurücksenden
24     conn.send(data)
25
26 conn.close()                # Socket schließen
```



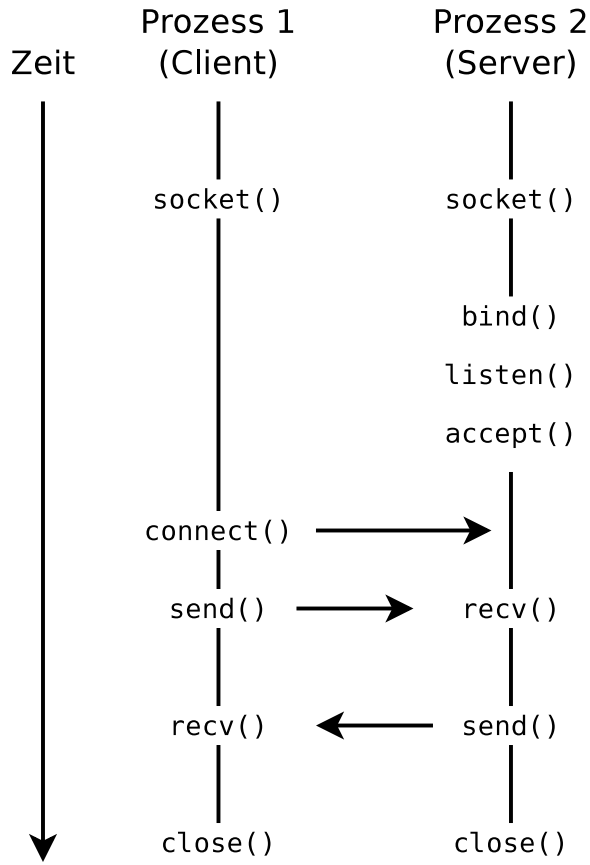

```
$ python tcp_server.py
```

Sockets via TCP – Beispiel (Client)

```
1  #!/usr/bin/env python
2  # -*- coding: iso-8859-15 -*-
3  # Echo Client via TCP
4  # Modul socket importieren
5  import socket
6
7  HOST = 'localhost'          # Hostname des Servers
8  PORT = 50007                # Portnummer des Servers
9
10 # Socket erzeugen und Socket-Deskriptor zurückliefern
11 sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12 # Mit Server-Socket verbinden
13 sd.connect((HOST, PORT))
14
15 sd.send('Hello, world')     # Daten senden
16 data = sd.recv(1024)        # Daten empfangen
17 sd.close()                  # Socket schließen
18
19 # Empfangene Daten ausgeben
20 print 'Empfangen:', repr(data)
```

```
$ python tcp_client.py
Empfangen: 'Hello, world'
```

```
$ python tcp_server.py
Connected by ('127.0.0.1', 49898)
```



Blockierende und nicht-blockierende Sockets

- Wird ein Socket erstellt, ist er standardmäßig im **blockierenden Modus**
  - Alle Methodenaufrufe warten, bis die von ihnen angestoßene Operation durchgeführt wurde
    - z.B. blockiert ein Aufruf von `recv()` den Prozess bis Daten eingegangen sind und aus dem internen Puffer des Sockets gelesen werden können
- Die Methode `setblocking()` **ändert** den Modus eines Sockets
  - `sd.setblocking(0)` $\Longrightarrow$ versetzt in den nicht-blockierenden Modus
  - `sd.setblocking(1)` $\Longrightarrow$ versetzt in den blockierenden Modus
- Es ist möglich, während des Betriebs den Modus **jederzeit** umzuschalten
  - z.B. könnte man die Methode `connect()` blockierend und anschließend `read()` nicht-blockierend verwenden

Quelle: Peter Kaiser, Johannes Ernesti. Python – Das umfassende Handbuch. Galileo (2008)

Nicht-blockierende Sockets – Einige Auswirkungen

- `recv()` und `recvfrom()`
  - Die Methoden geben nur dann Daten zurück, wenn sich diese bereits im internen Puffer des Sockets befinden
  - Sind keine Daten im Puffer, werfen die Methoden eine **Exception** und die Programmausführung läuft **weiter**
- `send()` und `sendto()`
  - Die Methoden versenden die angegebenen Daten nur, wenn sie direkt in den Ausgangspuffer des Sockets geschrieben werden können
  - Ist der Puffer schon voll, werfen die Methoden eine **Exception** und die Programmausführung läuft **weiter**
- `connect()`
  - Die Methode sendet eine Verbindungsanfrage an den Zielsocket und **wartet nicht**, bis diese Verbindung zustande kommt
  - Wird `connect()` aufgerufen, während die Verbindungsanfrage noch läuft, wird eine **Exception** geworfen
    - Durch mehrmaliges Aufrufen von `connect()` kann man überprüfen, ob die Operation immer noch durchgeführt wird

Vergleich der Kommunikations-Systeme

	Gemeinsamer Speicher	Nachrichten-warteschlangen	(anon./benannte) Pipes	Sockets
Art der Kommunikation	Speicherbasiert	Nachrichtenbasiert	Nachrichtenbasiert	Nachrichtenbasiert
Bidirektional	ja	nein	nein	ja
Plattformunabhäng	nein	nein	nein	ja
Prozesse müssen verwandt sein	nein	nein	bei anonymen Pipes	nein
Kommunikation über Rechnergrenzen	nein	nein	nein	ja
Bleiben ohne gebundenen Prozess erhalten	ja	ja	nein	nein
Automatische Synchronisierung	nein	ja	ja	ja

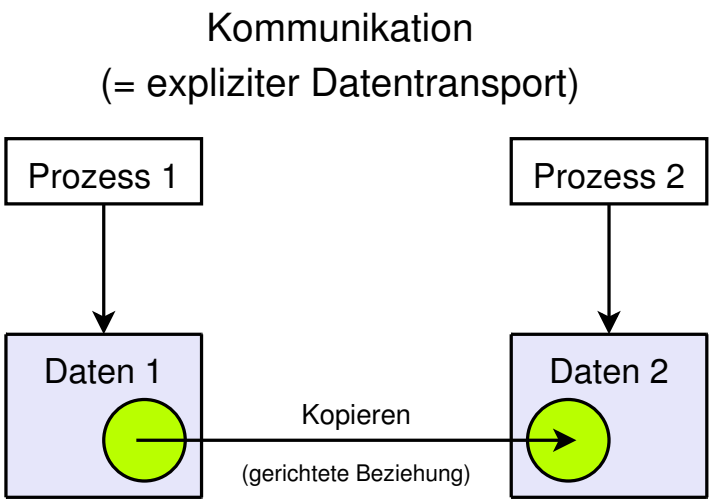
- Vorteile nachrichtenbasierte vs. speicherbasierte Kommunikation:
  - Das Betriebssystem nimmt den Benutzerprozessen die Synchronisation der Zugriffe ab $\Longrightarrow$ komfortabel
  - Einsetzbar in verteilten Systemen ohne gemeinsamen Speicher
  - Bessere Portabilität der Anwendungen

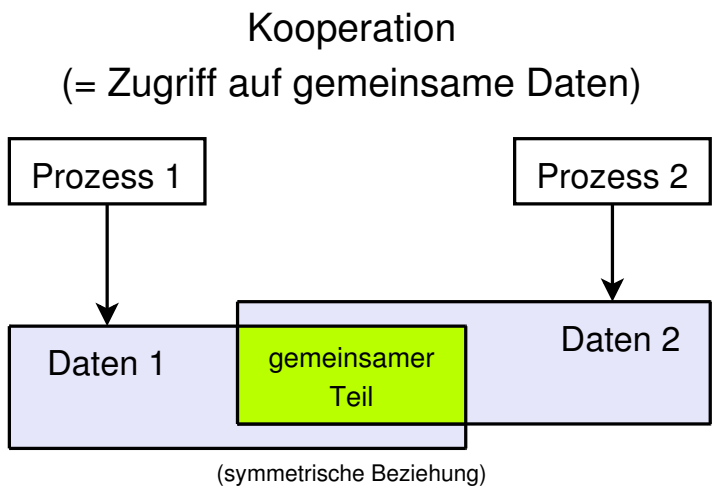
Speicher kann über Netzwerkverbindungen eingebunden werden

- Das ermöglicht speicherbasierte Kommunikation zwischen Prozessen auf verschiedenen, unabhängigen Systemen
- Das Problem der Synchronisation der Zugriffe besteht aber auch hier

Kooperation

- Kooperation
  - Semaphor
  - Mutex





Semaphoren

- Zur Sicherung (Sperrung) kritischer Abschnitte können außer den bekannten Sperren auch **Semaphoren** eingesetzt werden
- 1965: Veröffentlicht von Edsger W. Dijkstra
- Ein Semaphor ist eine Zählersperre **S** mit Operationen **P(S)** und **V(S)**
  - **V** kommt vom holländischen *verhogen* = erhöhen
  - **P** kommt vom holländischen *proberen* = versuchen (zu verringern)
- Die **Zugriffsoperationen sind atomar** $\Longrightarrow$ nicht unterbrechbar (unteilbar)
- Kann auch mehreren Prozessen das Betreten des kritischen Abschnitts erlauben
  - Im Gegensatz zu Semaphoren können Sperren ($\Longrightarrow$ Folie 14) immer nur einem Prozess das Betreten des kritischen Abschnitts erlauben

Die korrekte Grammatik ist *das Semaphor*, Plural *die Semaphore*

Cooperating sequential processes. *Edsger W. Dijkstra* (1965)

`https://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF`

Zugriffsoperationen auf Semaphoren (1/3)

Ein Semaphor besteht aus 2 Datenstrukturen

- `COUNT`: Eine **ganzzahlige, nichtnegative Zählvariable**.
  Gibt an, wie viele Prozesse das Semaphor aktuell ohne Blockierung passieren dürfen
- Ein `Warteraum` für die Prozesse, die darauf **warten**, das Semaphor passieren zu dürfen.
  Die Prozesse sind im Zustand `blockiert` und warten darauf, vom Betriebssystem in den Zustand `bereit` überführt zu werden, wenn das Semaphor den Weg freigibt

- **Initialisierung**: Zuerst wird ein Semaphor erzeugt oder ein bestehendes Semaphor geöffnet
  - Bei einem neuen Semaphor wird zu Beginn die Zählvariable mit einem nichtnegativen Anfangswert initialisiert

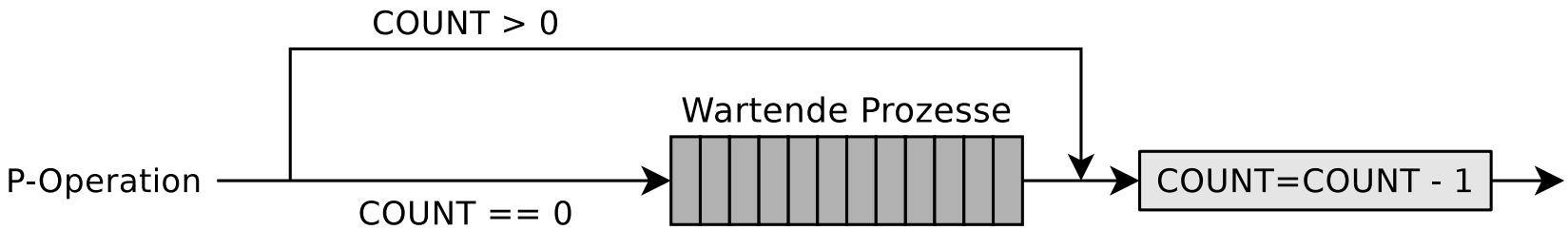
```
1 // Operation INIT auf Semaphor SEM anwenden
2 SEM.INIT(unsigned int init_wert) {
3
4     // Variable COUNT des Semaphors SEM mit einem
5     // nichtnegativen Anfangswert initialisieren
6     SEM.COUNT = init_wert;
7 }
```

Zugriffsoperationen auf Semaphoren (2/3)

Bildquelle: Carsten Vogt

- **P-Operation** (*verringern*): Prüft den Wert der Zählvariable
  - Ist der Wert 0, wird der Prozess blockiert
  - Ist der Wert > 0, wird er um 1 erniedrigt

```
1 SEM.P() {
2     // Ist die Zaehlvariable = 0, wird blockiert
3     if (SEM.COUNT == 0)
4         < blockiere >
5
6     // Ist die Zaehlvariable > 0, wird die
7     // Zaehlvariable unmittelbar um 1 erniedrigt
8     SEM.COUNT = SEM.COUNT - 1;
9 }
```

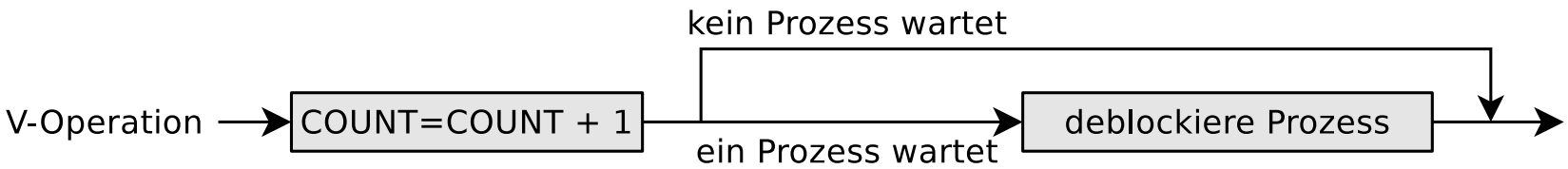


Zugriffsoperationen auf Semaphoren (3/3)

Bildquelle: Carsten Vogt

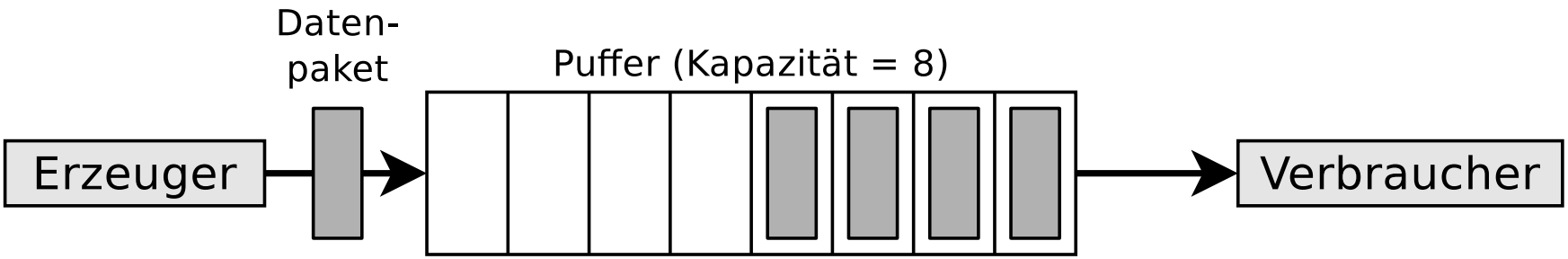
- **V-Operation** (*erhöhen*): Erhöht als erstes die Zählvariable um 1
  - Befinden sich Prozesse im Warteraum, wird ein Prozess deblockiert
  - Der gerade deblockierte Prozess setzt dann seine P-Operation fort und erniedrigt als erstes die Zählvariable

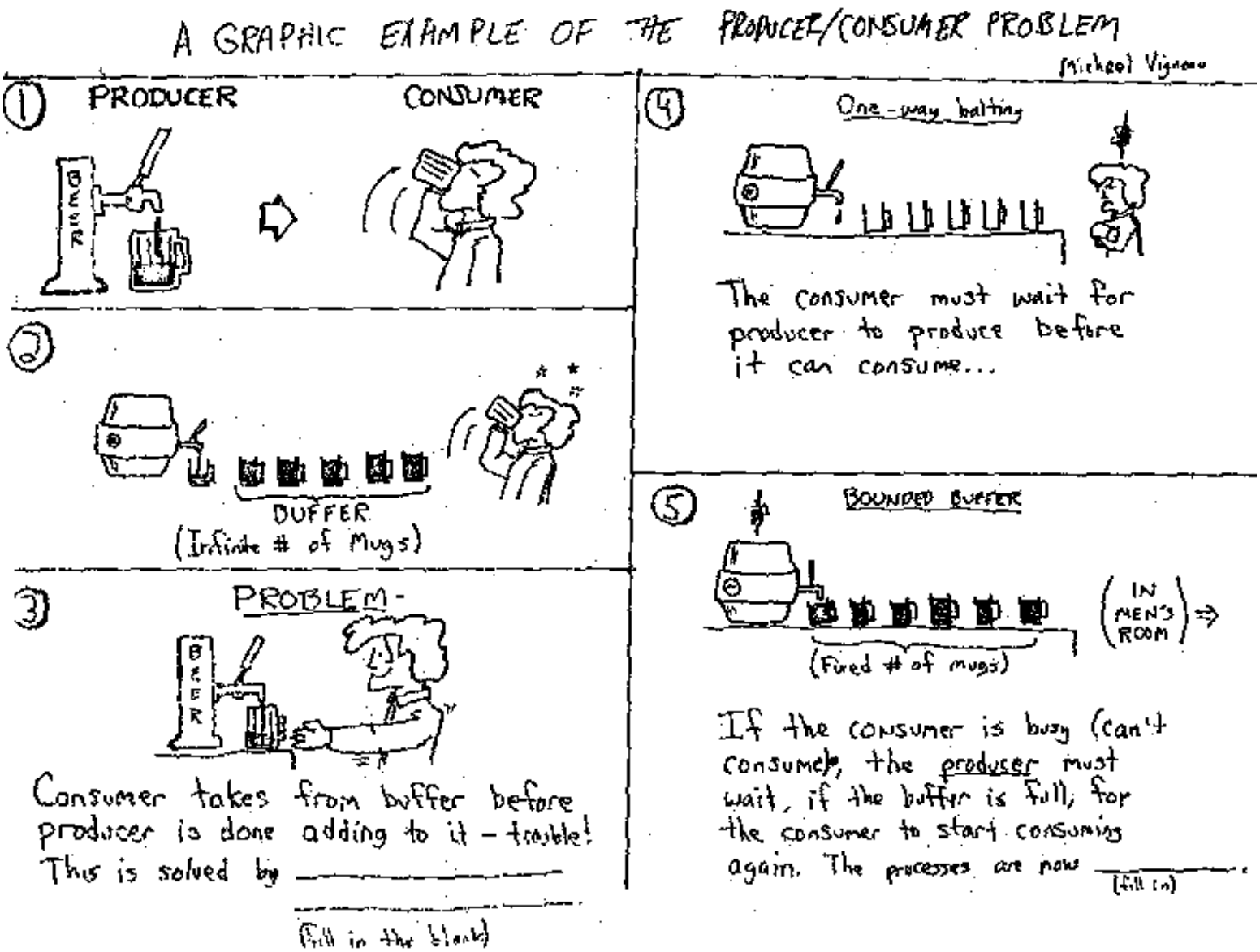
```
1 SEM.V() {
2     // Zaehlvariable = Zaehlvariable + 1
3     SEM.COUNT = SEM.COUNT + 1;
4
5     // Sind Prozesse im Warteraum, wird einer deblockiert
6     if ( < SEM-Warteraum ist nicht leer > )
7         < deblockiere einen wartenden Prozess >
8 }
```



Erzeuger/Verbraucher-Beispiel (1/3)

- Ein Erzeuger schickt Daten an einen Verbraucher
- Ein endlicher Zwischenspeicher (Puffer) soll Wartezeiten des Verbrauchers minimieren
- Daten werden vom Erzeuger in den Puffer gelegt und vom Verbraucher aus diesem entfernt
- Gegenseitiger Ausschluss ist notwendig, um Inkonsistenzen zu vermeiden
- Puffer = voll $\Longrightarrow$ Erzeuger muss blockieren
- Puffer = leer $\Longrightarrow$ Verbraucher muss blockieren





Quelle: Kenneth Baclawski (Northeastern University in Boston), Bildquelle: Michael Vigneau (Lizenz: unbekannt)
`http://www.ccs.neu.edu/home/kenb/tutorial/example.gif`

Erzeuger/Verbraucher-Beispiel (2/3)

- Zur Synchronisation der Zugriffe werden 3 Semaphore verwendet:
  - `leer`
  - `voll`
  - `mutex`
- Semaphore `voll` und `leer` werden gegenläufig zueinander eingesetzt
  - `leer` zählt die freien Plätze im Puffer, wird vom Erzeuger (P-Operation) erniedrigt und vom Verbraucher (V-Operation) erhöht
    - `leer` $= 0 \Longrightarrow$ Puffer vollständig belegt $\Longrightarrow$ Erzeuger blockieren
  - `voll` zählt die Datenpakete (belegte Plätze) im Puffer, wird vom Erzeuger (V-Operation) erhöht und vom Verbraucher (P-Operation) erniedrigt
    - `voll` $= 0 \Longrightarrow$ Puffer leer $\Longrightarrow$ Verbraucher blockieren
- Semaphor `mutex` ist für den wechselseitigen Ausschluss zuständig

Binäre Semaphore

- **Binäre Semaphore** werden mit dem Wert 1 initialisiert und garantieren, dass 2 oder mehr Prozesse nicht gleichzeitig in ihre kritischen Bereiche eintreten können
- Beispiel: Das Semaphor `mutex` aus dem Erzeuger/Verbraucher-Beispiel

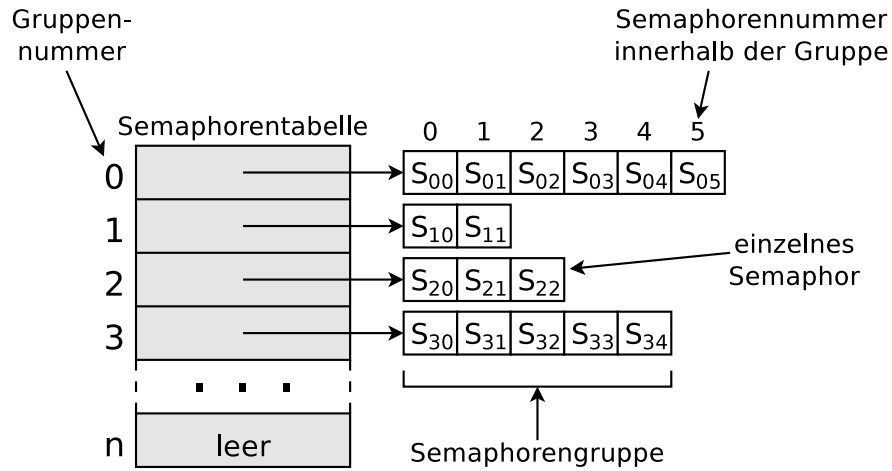
Erzeuger/Verbraucher-Beispiel (3/3)

```
1  typedef int semaphore;                    // Semaphore sind von Typ Integer
2  semaphore voll  = 0;                      // zählt die belegten Plätze im Puffer
3  semaphore leer  = 8;                      // zählt die freien Plätze im Puffer
4  semaphore mutex = 1;                      // steuert Zugriff auf kritische Bereiche
5
6  void erzeuger (void) {
7    int daten;
8
9    while (TRUE) {                          // Endlosschleife
10     erzeugeDatenpaket(daten);             // erzeuge Datenpaket
11     P(leer);                              // Zähler "leere Plätze" erniedrigen
12     P(mutex);                             // in kritischen Bereich eintreten
13     einfuegenDatenpaket(daten);           // Datenpaket in den Puffer schreiben
14     V(mutex);                             // kritischen Bereich verlassen
15     V(voll);                              // Zähler für volle Plätze erhöhen
16   }
17 }
18
19 void verbraucher (void) {
20   int daten;
21
22   while (TRUE) {                          // Endlosschleife
23     P(voll);                              // Zähler "volle Plätze" erniedrigen
24     P(mutex);                             // in kritischen Bereich eintreten
25     entferneDatenpaket(daten);            // Datenpaket aus dem Puffer holen
26     V(mutex);                             // kritischen Bereich verlassen
27     V(leer);                              // Zähler für leere Plätze erhöhen
28     verbraucheDatenpaket(daten);          // Datenpaket nutzen
29   }
30 }
```

Semaphoren unter Linux

Bildquelle: Carsten Vogt

- Linux weicht vom Konzept der Semaphore nach Dijkstra ab
  - Die Zählvariable kann mit einer P- oder V-Operation um mehr als 1 erhöht bzw. erniedrigt werden
  - Es können mehrere Zugriffsoperationen auf verschiedenen Semaphoren atomar, also unteilbar, durchgeführt werden
- Linux-Systeme verwalten eine Semaphortabelle, die Verweise auf Arrays mit Semaphoren enthält
  - Einzelne Semaphoren werden über den Tabellenindex und die Position in der Gruppe angesprochen



Linux/UNIX-Betriebssysteme stellen 3 Systemaufrufe für die Arbeit mit Semaphoren bereit

- `semget()`: Neues Semaphor oder eine Gruppe von Semaphoren erzeugen oder ein bestehendes Semaphor öffnen
- `semctl()`: Wert eines existierenden Semaphors oder einer Semaphorengruppe abfragen, ändern oder ein Semaphor löschen
- `semop()`: P- und V-Operationen auf Semaphoren durchführen
- Informationen über bestehende Semaphore liefert das Kommando `ipcs`

Mutexe

- Wird die Möglichkeit eines Semaphors zu zählen nicht benötigt, kann die vereinfachte Version eines Semaphors, der Mutex, verwendet werden
  - **Mutexe** (abgeleitet von **Mut**ual **Ex**clusion = wechselseitiger Ausschluss) dienen dem Schutz kritischer Abschnitte, auf die zu jedem Zeitpunkt immer nur **ein Prozess** zugreifen darf
    - Mutexe können nur 2 Zustände annehmen: **belegt** und **nicht belegt**
    - Mutexe haben die gleiche Funktionalität wie **binäre Semaphore**

2 Funktion zum Zugriff existieren

`mutex_lock`	$\Longrightarrow$	entspricht der P-Operation
`mutex_unlock`	$\Longrightarrow$	entspricht der V-Operation

- Will ein Prozess auf den kritischen Abschnitt zugreifen, ruft er `mutex_lock` auf
  - Ist der kritische Abschnitt **gesperrt**, wird der Prozess blockiert, bis der Prozess im kritischen Abschnitt fertig ist und `mutex_unlock` aufruft
  - Ist der kritische Abschnitt **nicht gesperrt** kann der Prozess eintreten

IPC-Objekte kontrollieren und löschen

- Informationen über bestehende gemeinsame Speichersegmente liefert das Kommando `ipcs`
- Die einfachste Möglichkeit, Semaphore, gemeinsame Speichersegmente und Nachrichtenwarteschlangen auf der Kommandozeile zu löschen, ist das Kommando `ipcrm`

```
ipcrm [-m shmid] [-q msqid] [-s semid]
      [-M shmkey] [-Q msgkey] [-S semkey]
```

- Oder alternativ einfach...
  - `ipcrm shm SharedMemoryID`
  - `ipcrm sem SemaphorID`
  - `ipcrm msg MessageQueueID`