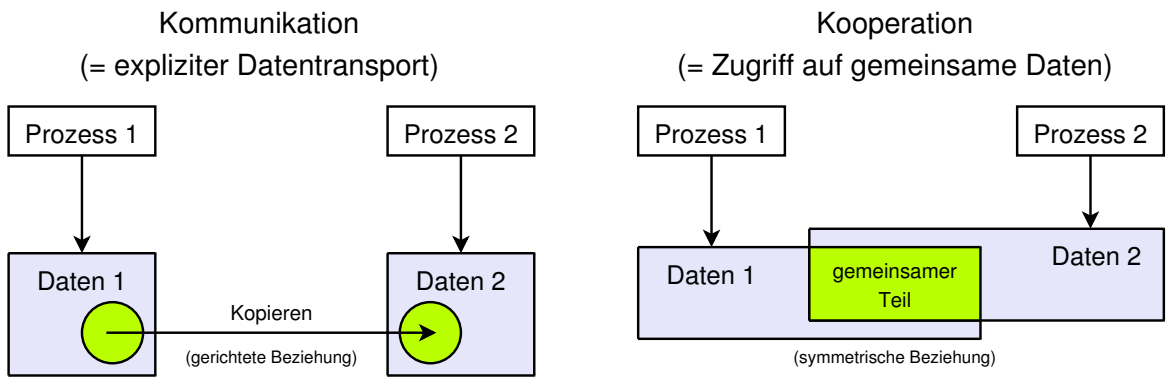
10. Foliensatz Betriebssysteme

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...
  - 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**



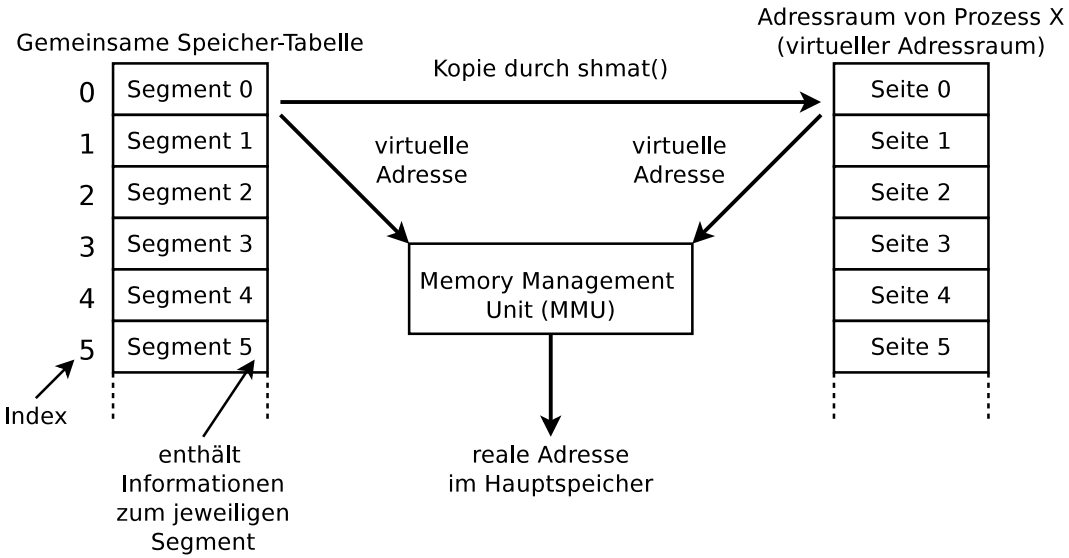
Übungsblatt 10 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes

Gemeinsamer Speicher – Shared Memory

- Prozesskommunikation über einen gemeinsamen Speicher (Shared Memory) nennt man 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

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
- `shmat()`: Gemeinsames Speichersegment an Prozesse binden
- `shmdt()`: Gemeinsames Speichersegment von Prozessen lösen/freigeben
- `shmctl()`: Status (u.a. Zugriffsrechte) eines gemeinsamen Speichersegments abfragen, ändern oder es löschen

Ein sehr gut erklärtes Beispiel zur Arbeit mit gemeinsamem Speicher enthält...

`http://openbook.rheinwerk-verlag.de/unix_guru/node393.html`

`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
```

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");
        }
    }
}
```

In ein Speichersegment schreiben und daraus lesen (in C)

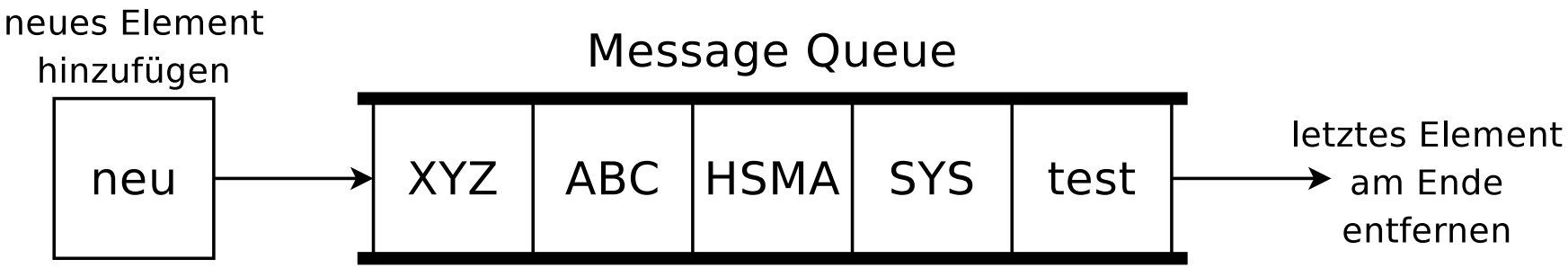
```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define MAXMEMSIZE 20

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

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)

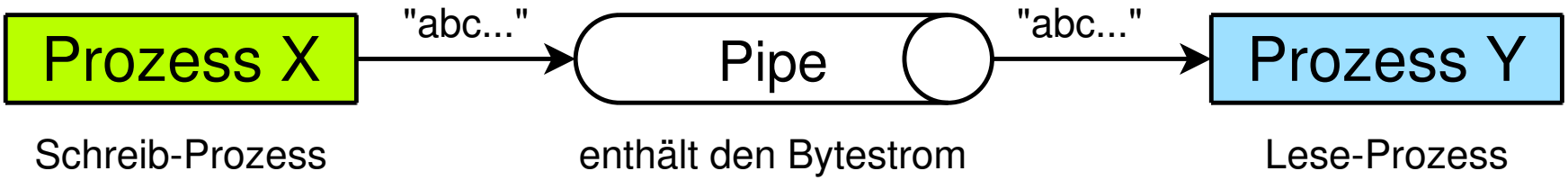
```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/msg.h>
#include <string.h>                 // Diese Header-Datei ist nötig für strcpy()
typedef struct msgbuf {             // Template eines Puffers fuer msgsnd und msgrcv
    long mtype;          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

Pipes (1/4)

- Eine Pipe ist wie ein Kanal bzw. eine Röhre, die einen gepufferten, unidirektionalen Datenaustausch zwischen 2 Prozessen realisiert
  - Können immer nur zwischen 2 Prozessen tätig sein
  - Arbeiten nach dem Prinzip FIFO
  - Haben 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
  - Werden mit dem Systemaufruf `pipe()` angelegt
    - Erzeugt einen Inode ($\Longrightarrow$ Foliensatz 6) und 2 Zugriffskennungen (*Handles*)
  - Prozesse greifen auf die Zugriffskennungen mit `read()` und `write()`-Systemaufrufen zu, um Daten aus der Pipe zu lesen bzw. um Daten in die Pipe zu schreiben



Pipes (2/4)

- Bei der Erzeugung von Kindprozessen mit `fork()` erben die Kindprozesse auch den Zugriff auf die Zugriffskennungen
- Man unterscheidet **anonyme Pipes** und **benannte Pipes**
- **Anonyme Pipes** ermöglichen Prozesskommunikation nur zwischen eng verwandten Prozessen
  - Kommunikation funktioniert nur in eine Richtung ($\Longrightarrow$ unidirektional)
  - 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

Pipes (3/4)

- Via **benannte Pipes** (Named Pipes), können auch nicht eng miteinander verwandte Prozesse kommunizieren
  - Auf diese Pipes kann mit Hilfe ihres Namens zugegriffen werden
  - 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`

Pipes in der Shell

Eine Pipe sorgt dafür, dass die Ausgabe eines Prozesses in die Eingabe eines anderen gelangt und wird auf der Shell mit | erzeugt. z.B.

```
cat /pfad/zu/Datei.txt | grep Suchmuster
```

Mit Pipes entwickeln (in C)

- Eine Pipe anlegen:

```
1 // Pipe testpipe anlegen
2 if (pipe(testpipe) < 0) {
3     // Falls die Pipe nicht angelegt werden konnte, wird das Programm beendet
4     printf("Das Anlegen der Pipe testpipe ist fehlgeschlagen.\n");
5     exit(1);
6 } else {
7     printf("Die Pipe testpipe wurde angelegt.\n");
8 }
```

- Pipe zum Schreiben vorbereiten (danach kann sie Daten aufnehmen):

```
1 close(testpipe[0]);                    // Lesekanal der Pipe testpipe blockieren
2 open(testpipe[1]);                     // Schreibkanal der Pipe testpipe oeffnen
```

- Pipe zum Lesen vorbereiten (danach kann sie ausgelesen werden):

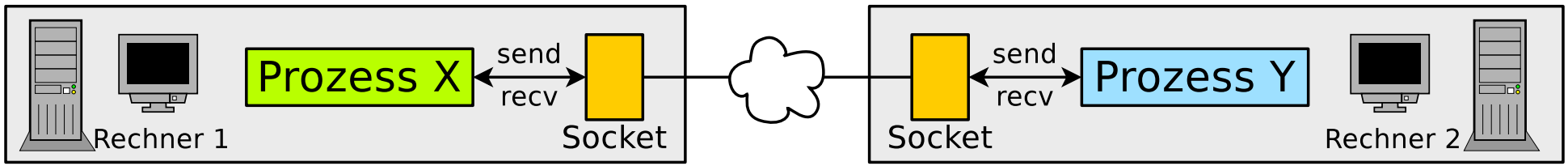
```
1 close(testpipe[1]);                    // Schreibkanall der Pipe testpipe blockieren
2 open(testpipe[0]);                     // Lesekanal der Pipe testpipe oeffnen
```

- Aus einer Pipe lesen und in eine Pipe schreiben:

```
1 read(testpipe[0], &puffervariable, sizeof(puffervariable));
2 write(testpipe[1], &puffervariable, sizeof(puffervariable));
```

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

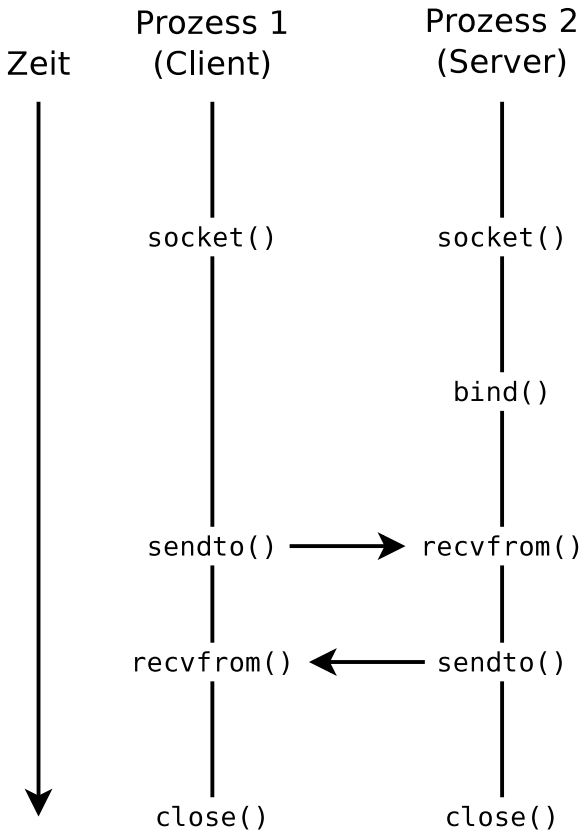
- **Verbindungslose Sockets** (bzw. **Datagram Sockets**)
  - Verwenden das Transportprotokoll UDP
  - Vorteil: Höhere Geschwindigkeit als bei TCP
    - Grund: Geringer Mehraufwand (Overhead) für das Protokoll
  - Nachteil: Segmente können einander überholen oder verloren gehen
- **Verbindungsorientierte 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()`

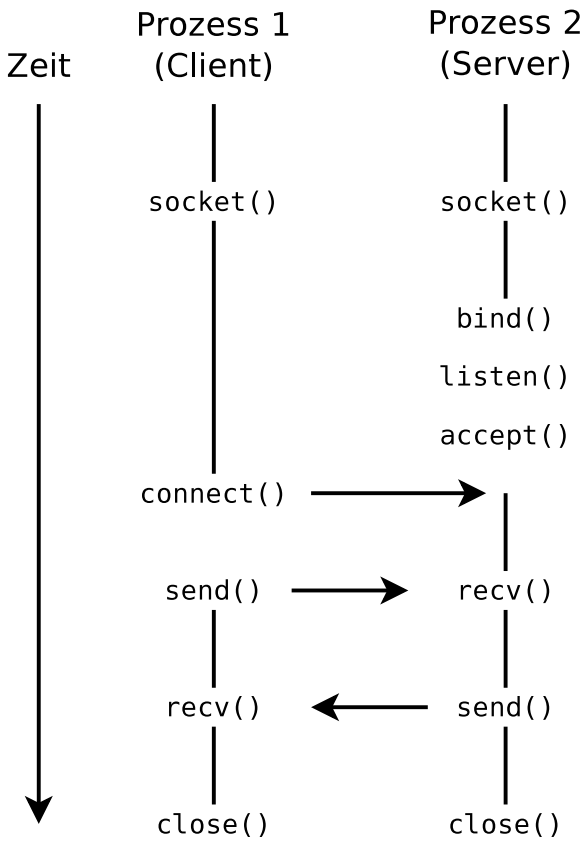
Übersicht der Sockets unter Linux/UNIX: `netstat -n` oder `lsof | grep socket`

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);
```

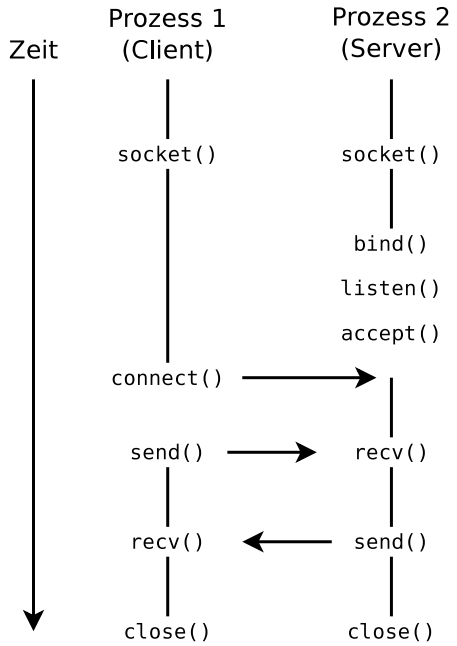
- Ein Aufruf von `socket()` liefert einen Integerwert zurück
  - Der Wert heißt **Socket-Deskriptor** (*socket file descriptor*)
- `domain`: Legt die Protokollfamilie fest
  - `PF_UNIX`: Lokale Prozesskommunikation unter Linux/UNIX
  - `PF_INET`: IPv4
  - `PF_INET6`: IPv6
- `type`: Legt den Typ des Sockets (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);
```

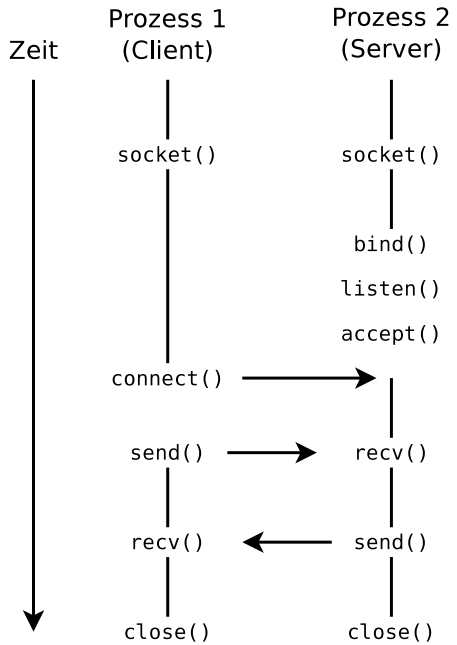
- `bind()` bindet den neu erstellen Socket (`sd`) an die Adresse (`address`) des Servers
  - `sd` ist der Socket-Deskriptor aus dem vorhergehenden Aufruf von `socket()`
  - `address` ist eine Datenstruktur, die die IP-Adresse des Server und eine Portnummer enthält
  - `addrlen` ist die Länge der Datenstruktur,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



Server empfangsbereit machen: `listen`

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

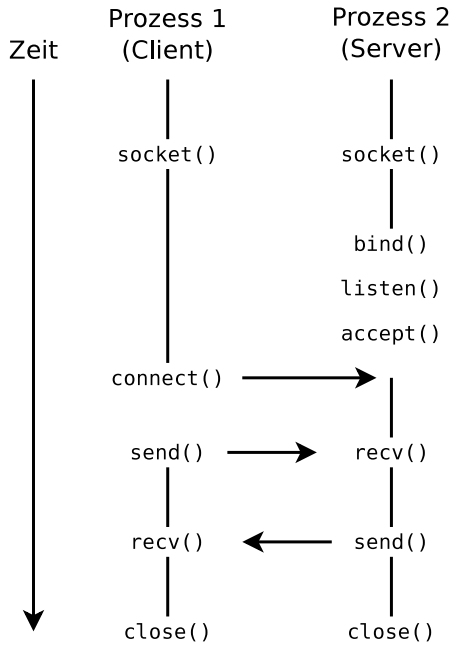
- `listen()` definiert, wie viele Verbindungsanfragen am Socket gepuffert werden können
  - Ist die `listen()`-Warteschlange voll, werden weitere Verbindungsanfragen von Clients abgewiesen
  - `sd` ist der Socket-Deskriptor aus dem vorhergehenden Aufruf von `socket()`
  - `backlog` enthält die Anzahl 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);
```

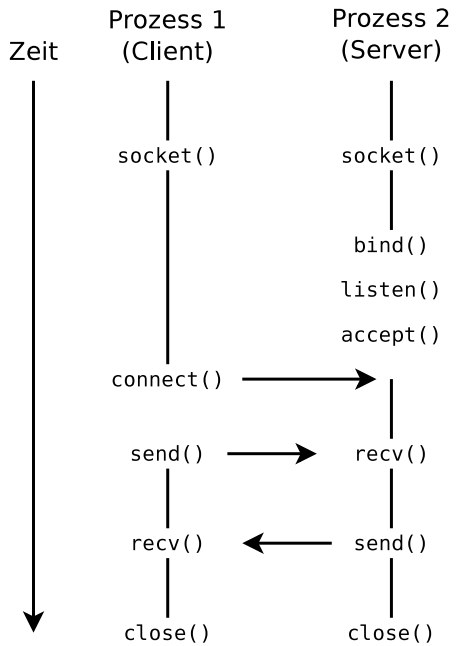
- Mit `accept()` holt der Server die erste Verbindungsanforderung aus der Warteschlange
- Der Rückgabewert ist der Socket-Deskriptor des neuen Sockets
- Enthält die Warteschlange keine Verbindungsanforderungen, ist der Prozess blockiert, bis eine Verbindungsanforderung eintrifft
- `address` enthält die 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);
```

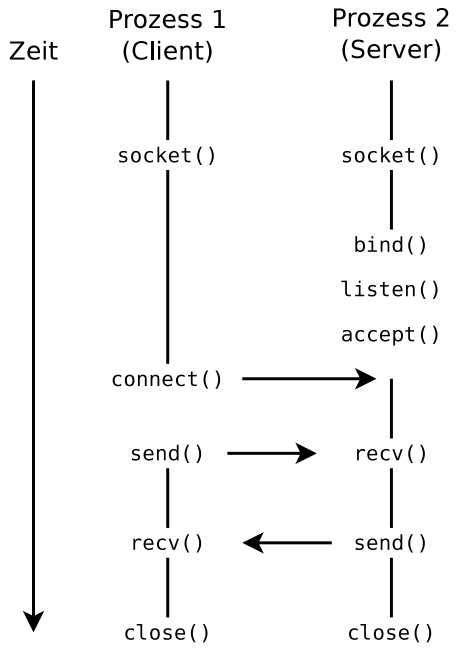
- Via `connect()` versucht der Client eine Verbindung mit einem Server-Socket herzustellen
- `sd` ist der Socket-Deskriptor
- `servaddr` ist die Adresse des Servers
- `addrlen` ist die Länge der Datenstruktur, die die Adresse enthält



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`

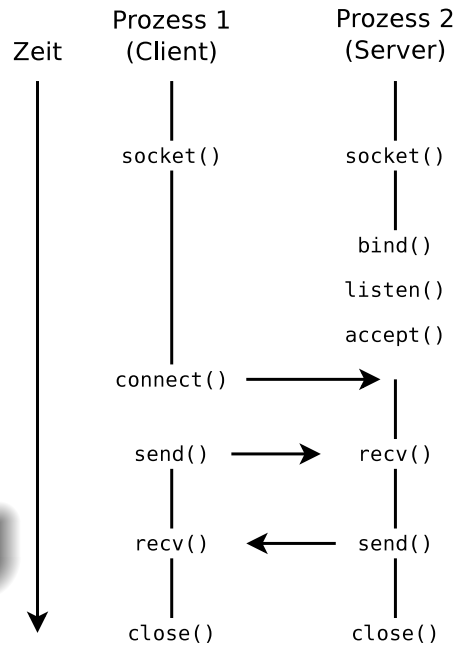
```
int sendto(int sd, char *buffer, int nbytes, int flags,
           struct sockaddr *to, int addrlen);
int recvfrom(int sd, char *buffer, int nbytes, int flags,
             struct sockaddr *from, int addrlen);
```

- 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)

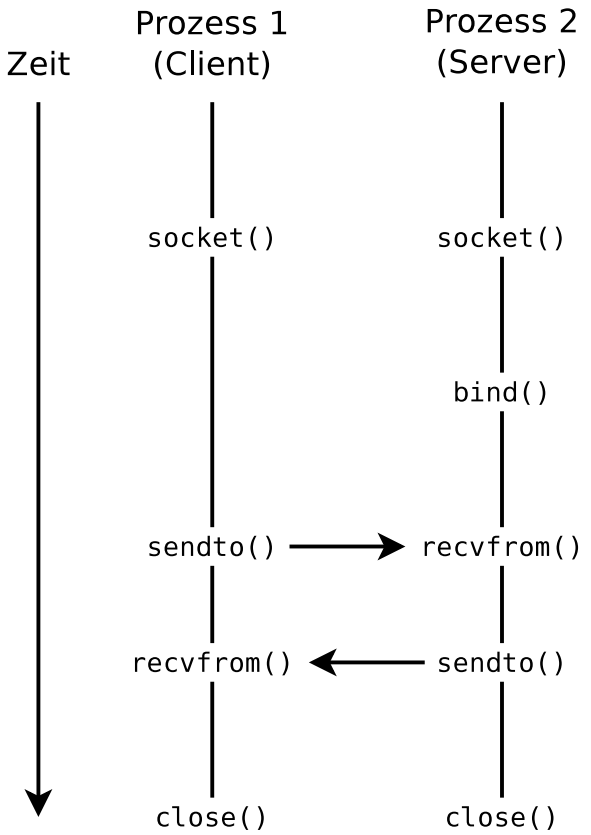



```
int close(int sd);
```

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

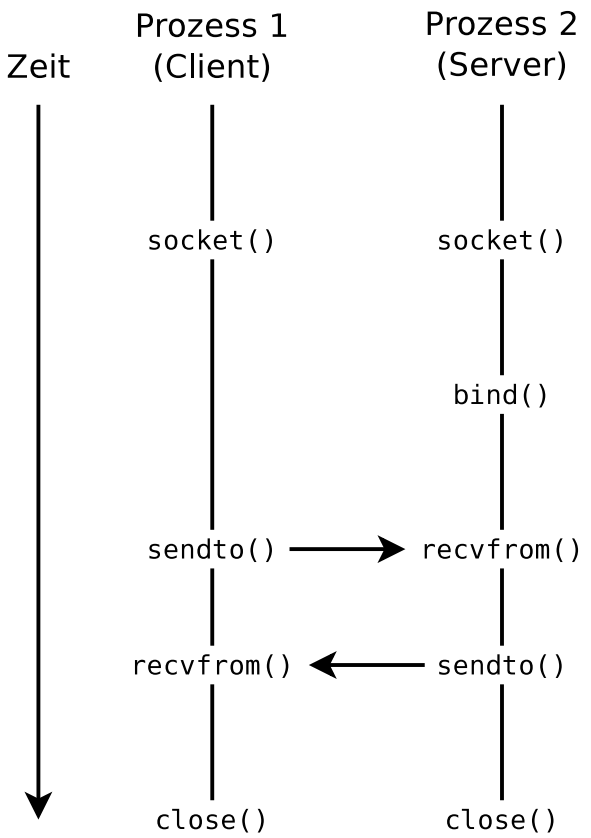
Sockets via UDP – Beispiel (Server)

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



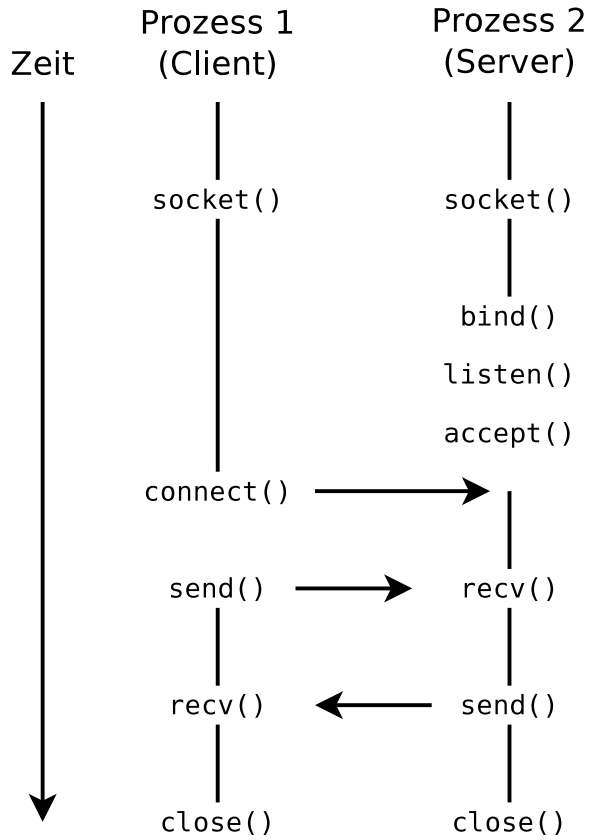
Sockets via UDP – Beispiel (Client)

```
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 = 'Hallo Welt'              # Nachricht
10
11 # Socket erzeugen und Socket Deskriptor zurückliefern
12 sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
13
14 sd.sendto(MESSAGE, (HOST, PORT)) # Nachricht an Socket senden
15
16 sd.close()                         # Socket schließen
```



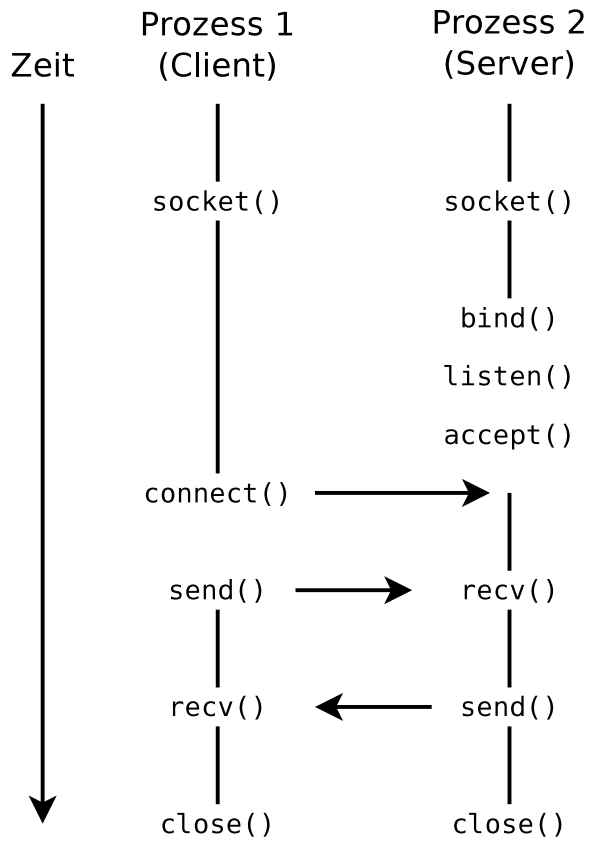
Sockets via TCP – Beispiel (Server)

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



Sockets via TCP – Beispiel (Client)

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



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

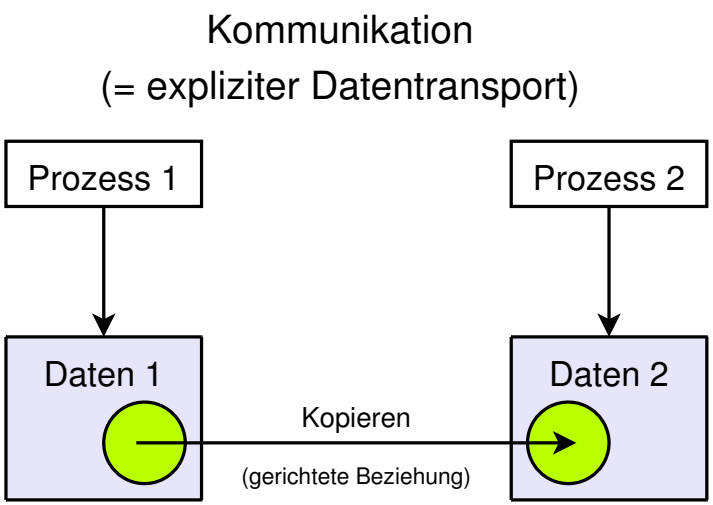
- Vorteile nachrichtenbasierter Kommunikation gegenüber speicherbasierter Kommunikation:
  - Das Betriebssystem nimmt den Benutzerprozessen die Synchronisation der Zugriffe ab $\Longrightarrow$ komfortabel
  - Einsetzbar in verteilten Systemen ohne gemeinsamen Speicher
  - Bessere Portabilität der Anwendungen

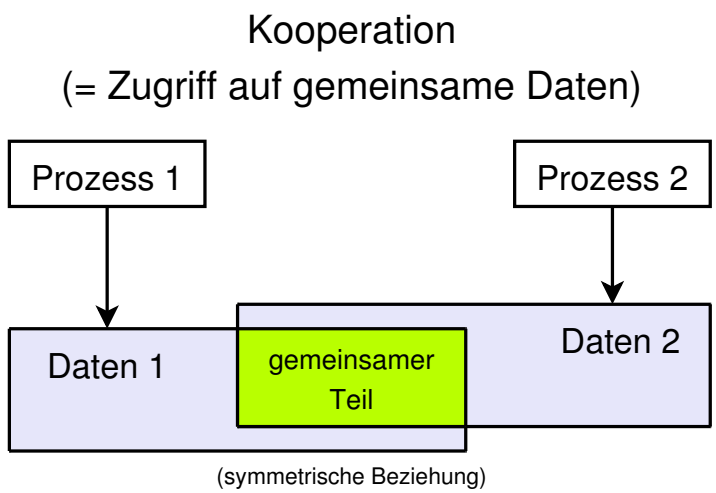
Speicher kann über Netzwerkverbindungen eingebunden werden

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

Kooperation

- Kooperation
  - Semaphor
  - Mutex





Semaphore

- Zur Sicherung (Sperrung) kritischer Abschnitte können außer den bekannten Sperren auch **Semaphore** 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 Semaphore können Sperren 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`

Semaphor: Arbeitsweise

- Folgendes Szenario macht die **Arbeitsweise** deutlich:
  - Vor einem Geschäft steht ein Stapel Einkaufskörbe
  - Will ein Kunde in das Geschäft, muss er einen Korb vom Stapel nehmen
  - Ist ein Kunde mit dem Einkauf fertig, muss er seinen Einkaufskorb wieder auf den Stapel zurückstellen
  - Ist der Stapel leer ($\Longrightarrow$ alle Einkaufskörbe sind vergeben), kann so lange kein neuer Kunde den Laden betreten, bis ein Einkaufskorb frei ist und auf dem Stapel liegt

Ein Semaphor besteht aus 2 Datenstrukturen

- `COUNT`: Eine ganzzahlige, nichtnegative Zählvariable
  - Gibt an, wie viele Prozesse das Semaphor aktuell ohne Blockierung passieren dürfen

Der Wert entspricht, gemäß dem einführenden Beispiel, der Anzahl der Körbe, die sich aktuell auf dem Stapel vor dem Laden befinden

- 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

Das Semaphor gibt den Weg frei, wenn wieder Körbe frei sind

Die Länge der Warteschlange entspricht der Anzahl der Kunden, die vor dem Laden warten, weil keine Körbe mehr frei sind

3 Zugriffsoperationen sind möglich (1/3)

- **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

Dieser Wert ist die Anzahl der Körbe, die bei Ladenöffnung vor dem Laden bereitgestellt werden

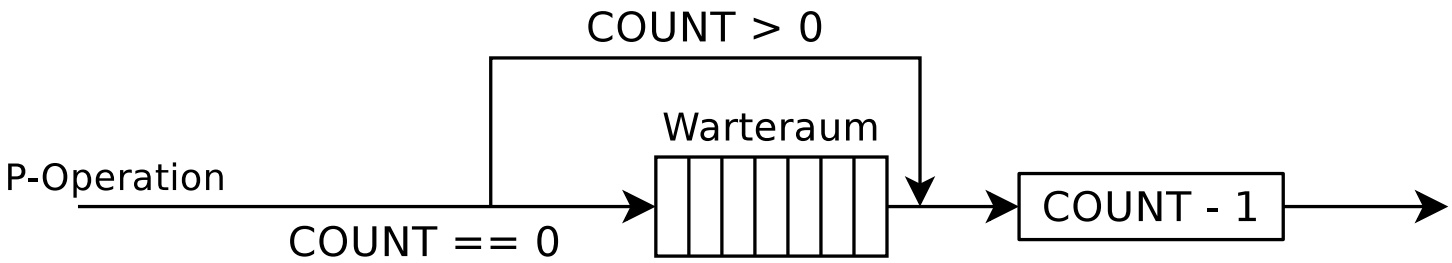
```
1 // Operation INIT auf Semaphor SEM anwenden
2 SEM.INIT(unsigned int init_wert) {
3
4     // Variable COUNT des Semaphors SEM mit einem
5     // nichtnegativen Anfangswert initialisieren
6     SEM.COUNT = init_wert;
7 }
```

3 Zugriffsoperationen sind möglich (2/3)

Bildquelle: Carsten Vogt

- **P-Operation** (*verringern*): Prüft den Wert der Zählvariable
  - Ist der Wert 0, wird der Prozess blockiert
    - *Der Kunde muss in der Warteschlange vor dem Laden warten*
  - Ist der Wert > 0, wird er um 1 erniedrigt
    - *Der Kunde nimmt einen Korb*

```
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 }
```

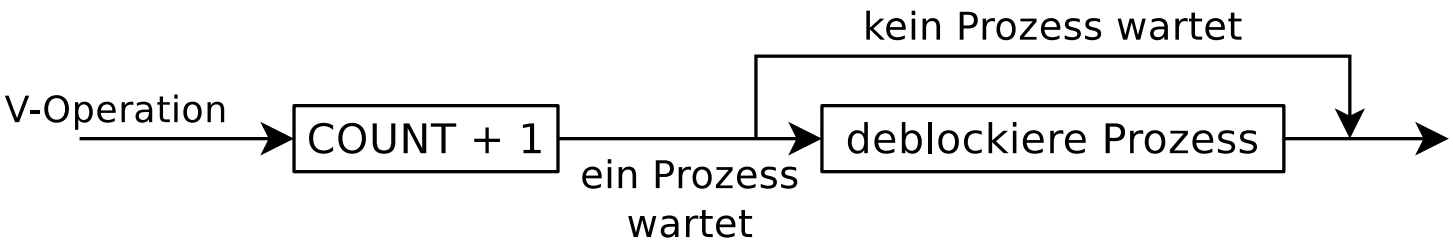


3 Zugriffsoperationen sind möglich (3/3)

Bildquelle: Carsten Vogt

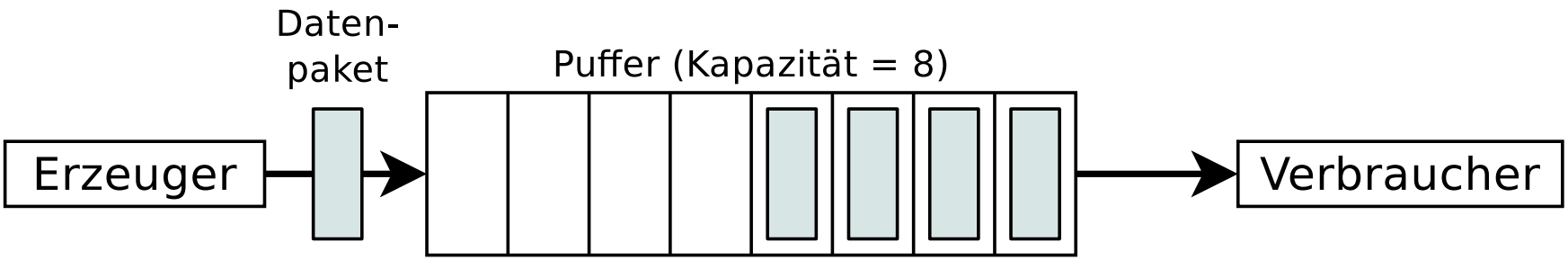
- **V-Operation** (*erhöhen*): Erhöht als erstes die Zählvariable um 1
  - *Es wird ein Korb auf den Stapel zurückgelegt*
  - Befinden sich Prozesse im Warteraum, wird ein Prozess deblockiert
    - *Ein Kunde kann jetzt einen Korb holen*
  - Der gerade deblockierte Prozess setzt dann seine P-Operation fort und erniedrigt als erstes die Zählvariable
    - *Der Kunde nimmt einen Korb*

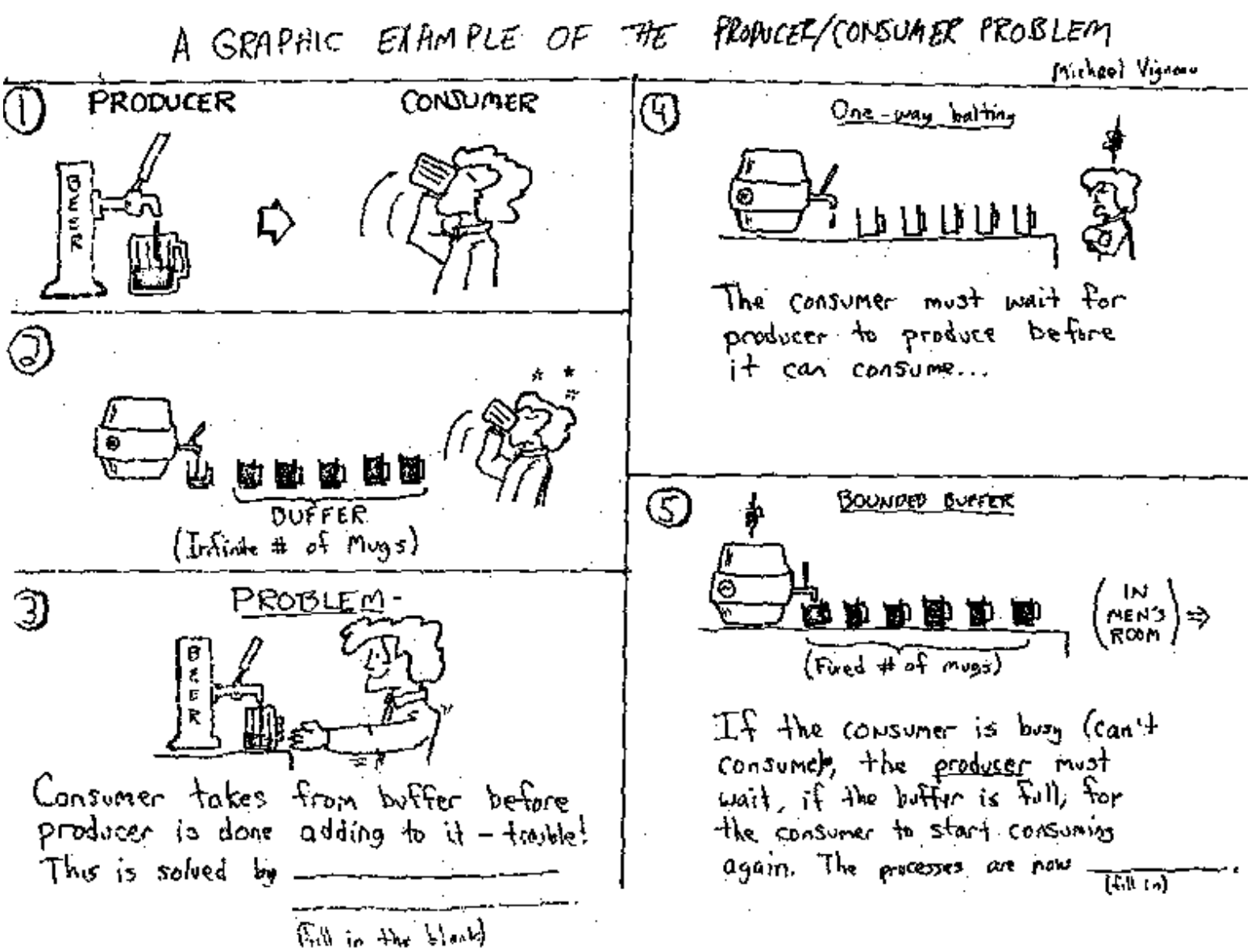
```
1 SEM.V() {
2     // Zaehlvariable = Zaehlvariable + 1
3     SEM.COUNT = SEM.COUNT + 1;
4
5     // Sind Prozesse im Warteraum, wird einer deblockiert
6     if ( < SEM-Warteraum ist nicht leer > )
7         < deblockiere einen wartenden Prozess >
8 }
```



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: `http://www.ccs.neu.edu/home/kenb/synchronize.html`

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

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 }
```

Beispiel zu Semaphore: PingPong

```
1  // Initialisierung der Semaphore
2  s_init (Sema_Ping, 1);
3  s_init (Sema_Pong, 0);
4
5  task Ping is
6  begin
7      loop
8          P(Sema_Ping);
9          print("Ping");
10         V(Sema_Pong);
11     end loop;
12 end Ping;
13
14 task Pong is
15 begin
16     loop
17         P(Sema_Pong);
18         print("Pong, ");
19         V(Sema_Ping);
20     end loop;
21 end Pong;
```

- Die beiden Endlosprozesse Ping und Pong geben endlos folgendes aus: PingPong, PingPong, PingPong...

Beispiel zu Semaphore: 3 Läufer (1/3)

- 3 Läufer sollen hintereinander eine bestimmte Strecke laufen
  - Der zweite Läufer darf erst starten, wenn der erste Läufer im Ziel ist
  - Der dritte Läufer darf erst starten, wenn der zweite Läufer im Ziel ist
- Ist diese Lösung korrekt?

```
1  // Initialisierung der Semaphore
2  s_init (Sema, 0);
3
4  task Erster is
5          < laufen >
6          V(Sema);
7
8  task Zweiter is
9          P(Sema);
10         < laufen >
11         V(Sema);
12
13 task Dritter is
14         P(Sema);
15         < laufen >
```

Beispiel zu Semaphore: 3 Läufer (2/3)

- Die Lösung ist nicht korrekt!
- Es existieren 2 Reihenfolgebeziehungen:
  - Läufer 1 vor Läufer 2
  - Läufer 2 vor Läufer 3
- Beide Reihenfolgebeziehungen verwenden das gleiche Semaphor
  - Es ist nicht ausgeschlossen, dass Läufer 3 mit seiner P-Operation vor Läufer 2 das Semaphor um den Wert 1 erniedrigt
- Wie könnte eine korrekte

```
1  // Initialisierung der Semaphore
2  s_init (Sema, 0);
3
4  task Erster is
5          < laufen >
6          V(Sema);
7
8  task Zweiter is
9          P(Sema);
10         < laufen >
11         V(Sema);
12
13 task Dritter is
14         P(Sema);
15         < laufen >
```

Beispiel zu Semaphore: 3 Läufer (3/3)

- Lösungsmöglichkeit:
  - Zweiten Semaphor einführen
  - Das zweites Semaphor wird ebenfalls mit dem Wert 0 initialisiert
  - Läufer 2 erhöht mit seiner V-Operation das zweite Semaphor und Läufer 3 erniedrigt dieses mit seiner P-Operation

```
1  // Initialisierung der Semaphore
2  s_init (Sema1, 0);
3  s_init (Sema2, 0);
4
5  task Erster is
6          < laufen >
7          V(Sema1);
8
9  task Zweiter is
10         P(Sema1);
11         < laufen >
12         V(Sema2);
13
14 task Dritter is
15         P(Sema2);
16         < laufen >
```

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

Starke und schwache Semaphore

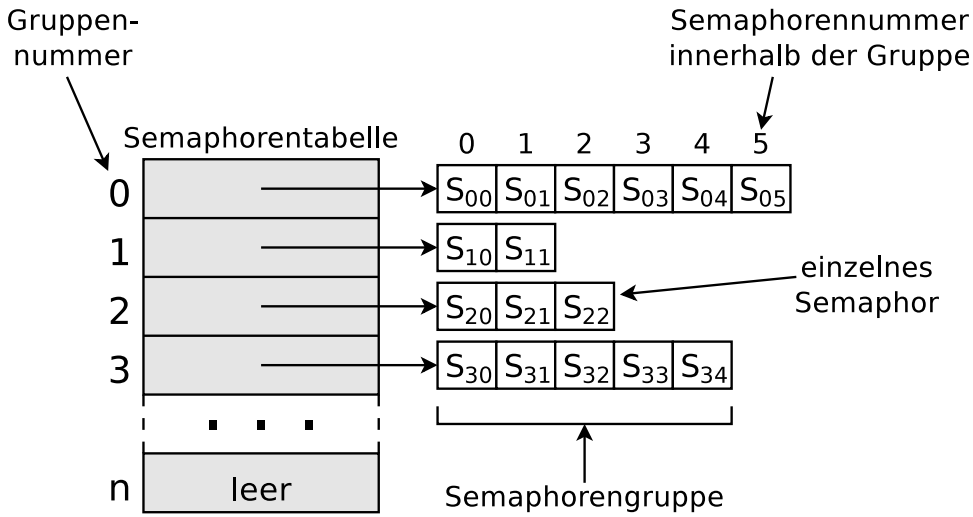
- Für jede Semaphore oder binäre Semaphore gibt es eine Warteschlange, die wartende Prozesse aufnimmt
  - **Starke Semaphore**
    - Prozesse werden nach dem Prinzip FIFO aus der Warteschlange geholt
    - Typische Form des Semaphor, die Betriebssysteme bereitstellen
    - Vorteil: Es kann nicht zum Verhungern kommen
  - **Schwache Semaphore** legen die Reihenfolge, in der die Prozesse aus der Warteschlange geholt werden, nicht fest
    - Werden bei Echtzeitbetrieb eingesetzt, da das Deblockieren von Prozessen sich an deren Priorität und nicht am Zeitpunkt der Blockierung orientiert

Semaphore unter Linux/UNIX (1/2)

Bildquelle: Carsten Vogt

- Das Konzept der Semaphore unter Linux/UNIX weicht vom Konzept der Semaphore nach Dijkstra ab
  - Unter Linux/UNIX kann die Zählvariable 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
    - Mehrere P-Operationen können z.B. zusammengefasst und nur dann durchgeführt werden, wenn keine der P-Operationen blockiert

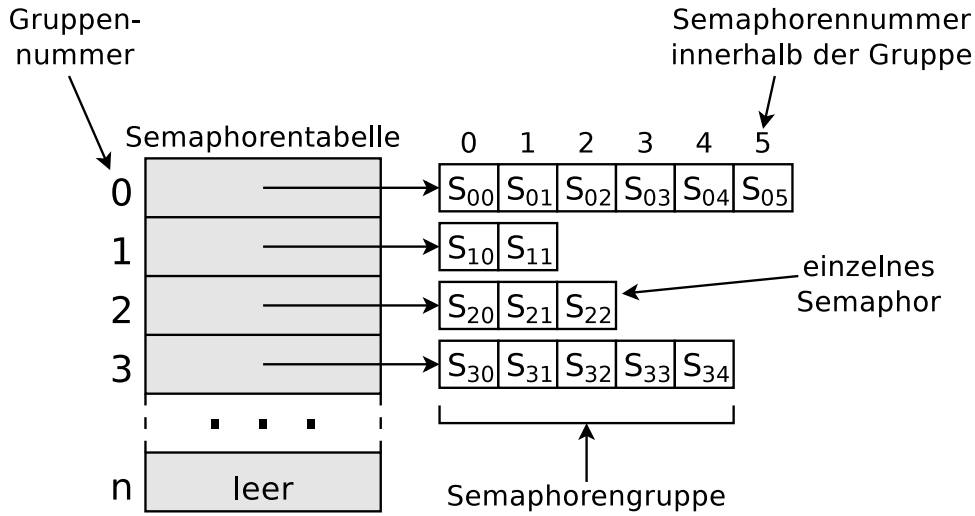
- Linux/UNIX-Systeme führen im Kernel eine Semaphortabelle, die Verweise auf Arrays mit Semaphore enthält
  - Jedes Array enthält eine Gruppe von Semaphoren, die über den Index der Tabelle identifiziert wird



Semaphore unter Linux/UNIX (2/2)

Bildquelle: Carsten Vogt

- Einzelne Semaphore werden über den Tabellenindex und die Position in der Gruppe (beginnend bei 0) angesprochen
- Atomare Operationen auf mehreren Semaphoren können nur dann durchgeführt werden, wenn alle Semaphore der gleichen Gruppe angehören



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`