

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

- 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 (\implies Daten) zugreifen können, ist **wechselseitiger Ausschluss** (*Mutual Exclusion*) nötig

[illegible]

— — — — —

$$in = \text{next free slot} + 1; \quad (17)$$

— — — — —

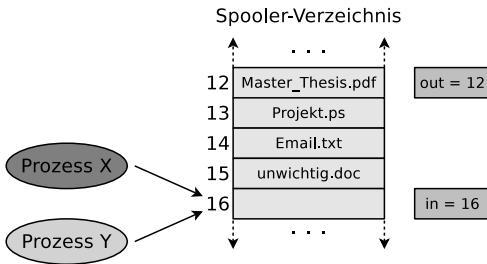
[illegible]

Speichere Eintrag in next

$$in = \text{next free slot} + 1; \quad (17)$$

— — — — — ()

References



- 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**

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

-

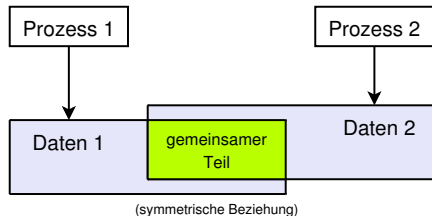


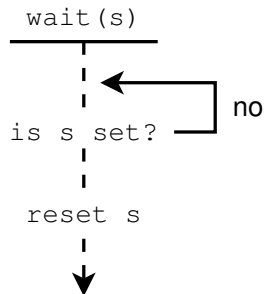
Prof. Dr. Christian Baun – 9. Foliensatz Betriebssysteme – Frankfurt University of Applied Sciences – WS1920

- Funktionaler Aspekt: **Kommunikation** und **Kooperation**
- Zeitlicher Aspekt: **Synchronisation**

Kooperation

(= Zugriff auf gemeinsame Daten)

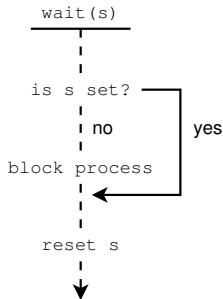
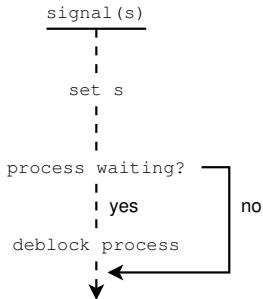




- Das aktive Warten heißt in der Literatur auch *Busy Waiting* oder *Polling*

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`.

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**)

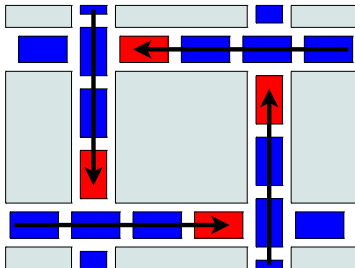
Zusammenfassung: Unterschied zwischen Signalisieren und Blockieren

- **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

Bildquelle: Google Bildersuche

- **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



Bedingungen für Deadlocks

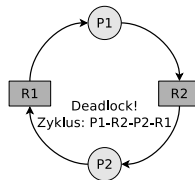
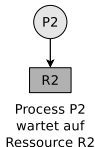
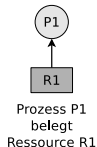
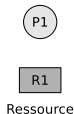
System Deadlocks. E. G. Coffman, M. J. Elphick, A. Shoshani. *Computing Surveys*, Vol. 3, No. 2, June 1971, S.67-78.
http://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf

- 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 \implies 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

Prozess



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

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

- 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

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

- 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

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

- 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

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

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

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

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

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

- Prozess 1 ist nicht blockiert** \implies 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)

27 / 78

28/78

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     int shared_memory_id = 12345;
9     int returncode_shmget, returncode_shmdt, returncode_sprintf;
10    char *sharedmempointer;
11
12    // Gemeinsames Speichersegment erzeugen
13    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
14    ...
15
16    // Gemeinsames Speichersegment anhängen
17    sharedmempointer = shmat(returncode_shmget, 0, 0);
18    ...
19
20    // Eine Zeichenkette in das gemeinsame Speichersegment schreiben
21    returncode_sprintf = sprintf(sharedmempointer, "Hallo Welt.");
22    if (returncode_sprintf < 0) {
23        printf("Der Schreibzugriff ist fehlgeschlagen.\n");
24    } else {
25        printf("%i Zeichen in das Segment geschrieben.\n", returncode_sprintf);
26    }
27
28    // Die Zeichenkette im gemeinsamen Speichersegment ausgeben
29    if (printf("%s\n", sharedmempointer) < 0) {
30        printf("Der Lesezugriff ist fehlgeschlagen.\n");
31    }
32    ...
33}

```

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #define MAXMEMSIZE 20
6
7 int main(int argc, char **argv) {
8     int shared_memory_id = 12345;
9     int returncode_shmget;
10    int returncode_shmdt;
11    char *sharedmempointer;
12
13    // Gemeinsames Speichersegment erzeugen
14    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
15    ...
16
17    // Gemeinsames Speichersegment anhängen
18    sharedmempointer = shmat(returncode_shmget, 0, 0);
19    ...
20
21    // Gemeinsames Speichersegment lösen
22    returncode_shmdt = shmdt(sharedmempointer);
23    if (returncode_shmdt < 0) {
24        printf("Das gemeinsame Speichersegment konnte nicht gelöst werden.\n");
25        perror("shmdt");
26    } else {
27        printf("Das Segment wurde vom Prozess gelöst.\n");
28    }
29 }
30 }

```

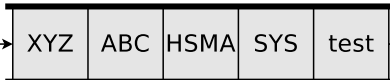

Nachrichtewarteschlangen - Message Queues

- 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 Nachrichtewarteschlange

neues Element
hinzufügen



Message Queue



letztes Element
am Ende
entfernen

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

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

Informationen über bestehende Nachrichtewarteschlangen liefert das Kommando `ipcs`

Nachrichtewarteschlangen erzeugen (in C)

```

1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6
7 int main(int argc, char **argv) {
8     int returncode_msgget;
9
10    // Nachrichtewarteschlange erzeugen oder auf eine bestehende zugreifen
11    // IPC_CREAT => neue Nachrichtewarteschlange erzeugen, wenn sie noch nicht existiert
12    // 0600 = Zugriffsrechte auf die neue Nachrichtewarteschlange
13    returncode_msgget = msgget(12345, IPC_CREAT | 0600);
14    if(returncode_msgget < 0) {
15        printf("Die Nachrichtewarteschlange konnte nicht erstellt werden.\n");
16        exit(1);
17    } else {
18        printf("Die Nachrichtewarteschlange 12345 mit der ID %i ist nun verfügbar.\n",
19               returncode_msgget);
20    }
21 }

```

```

$ 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)

```

1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6 #include <string.h>           // Diese Header-Datei ist nötig für strcpy()
7
8 struct msgbuf {              // Template eines Puffers fuer msgsnd und msgrcv
9     long mtype;              // Nachrichtentyp
10    char mtext[80];           // Sendepuffer
11 } msg;
12
13 int main(int argc, char **argv) {
14     int returncode_msgget;
15
16     // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
17     returncode_msgget = msgget(12345, IPC_CREAT | 0600);
18     ...
19
20     msg.mtype = 1;            // Nachrichtentyp festlegen
21     strcpy(msg.mtext, "Testnachricht"); // Nachricht in den Sendepuffer schreiben
22
23     // Eine Nachricht in die Nachrichtenwarteschlange schreiben
24     if (msgsnd(returncode_msgget, &msg, strlen(msg.mtext), 0) == -1) {
25         printf("In die Nachrichtenwarteschlange konnte nicht geschrieben werden.\n");
26         exit(1);
27     }
28 }

```

- 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)

```

1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6 #include <string.h>           // Diese Header-Datei ist nötig für strcpy()
7 typedef struct msgbuf {      // Template eines Puffers fuer msgsnd und msgrcv
8     long mtype;              // Nachrichtentyp
9     char mtext[80];          // Sendepuffer
10 } msg;
11
12 int main(int argc, char **argv) {
13     int returncode_msgget, returncode_msgrcv;
14     msg receivebuffer;        // Einen Empfangspuffer anlegen
15
16     // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
17     returncode_msgget = msgget(12345, IPC_CREAT | 0600)
18
19     msg.mtype = 1;            // Die erste Nachricht vom Typ 1 empfangen
20     // MSG_NOERROR => Nachrichten abschneiden, wenn sie zu lang sind
21     // IPC_NOWAIT  => Prozess nicht blockieren, wenn keine Nachricht vom Typ vorliegt
22     returncode_msgrcv = msgrcv(returncode_msgget, &msg, sizeof(msg.mtext), msg.mtype,
23                               MSG_NOERROR | IPC_NOWAIT);
24     if (returncode_msgrcv < 0) {
25         printf("Aus der Nachrichtenwarteschlange konnte nicht gelesen werden.\n");
26         perror("msgrcv");
27     } else {
28         printf("Diese Nachricht wurde aus der Warteschlange gelesen: %s\n", msg.mtext);
29         printf("Die empfangene Nachricht ist %i Zeichen lang.\n", returncode_msgrcv);
30     }
31 }

```

```

1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6
7 int main(int argc, char **argv) {
8     int returncode_msgget;
9     int returncode_msgctl;
10
11     // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
12     returncode_msgget = msgget(12345, IPC_CREAT | 0600);
13     ...
14
15     // Nachrichtenwarteschlange löschen
16     returncode_msgctl = msgctl(returncode_msgget, IPC_RMID, 0);
17     if (returncode_msgctl < 0) {
18         printf("Die Nachrichtenwarteschlange mit der ID %i konnte nicht gelöscht werden.\n",
19             returncode_msgget);
20         perror("msgctl");
21         exit(1);
22     } else {
23         printf("Die Nachrichtenwarteschlange mit der ID %i wurde gelöscht.\n",
24             returncode_msgget);
25     }
26     exit(0);
27 }

```

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 \Rightarrow der in die Pipe schreibende Prozess wird blockiert
 - Pipe = leer \Rightarrow der aus der Pipe lesende Prozess wird blockiert
- wird mit dem Systemaufruf `pipe()` angelegt
 - Dabei erzeugt der Betriebssystemkern einen Inode (\Rightarrow 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

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 }

```

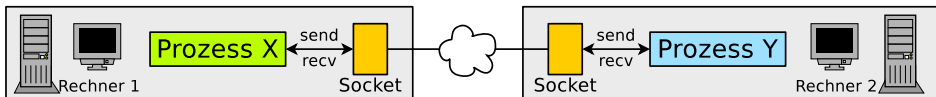
```

$ 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)

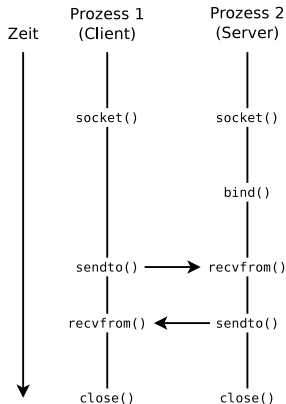
- Verwenden das Transportprotokoll TCP
- Vorteil: Höhere Verlässlichkeit
 - Segmente können nicht verloren gehen
 - Segmente kommen immer in der korrekten Reihenfolge an
- Nachteil: Geringere Geschwindigkeit als bei UDP
 - Grund: Höherer Mehraufwand (Overhead) für das Protokoll

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

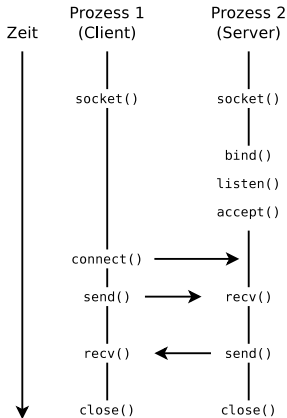
- Socket erstellen (`socket`)
- Socket an einen Port binden (`bind`)
- Daten senden (`sendto`) und empfangen (`recvfrom`)
- Socket schließen (`close`)

- **Client**

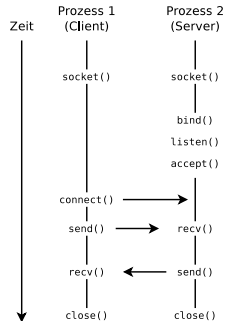
- Socket erstellen (`socket`)
- Client mit Server-Socket verbinden (`connect`)
- Daten senden (`send`) und empfangen (`recv`)
- Socket schließen (`close`)

- **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`)

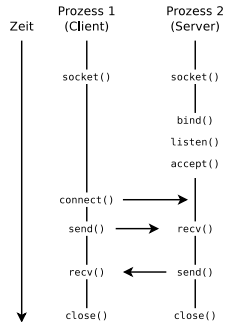


- `bind()` bindet den neu erstellen Socket (`sd`) an die Adresse (`address`) des Servers
 - `sd` ist der Socket-Deskriptor aus dem vorhergehenden Aufruf von `socket()`
 - `address` ist eine Datenstruktur, die die IP-Adresse des Server und eine Portnummer enthält
 - `addrlen` ist die Länge der Datenstruktur,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



100

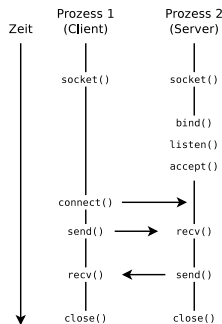
- `listen()` definiert, wie viele Verbindungsanfragen am Socket gepuffert werden können
 - Ist die `listen()`-Warteschlange voll, werden weitere Verbindungsanfragen von Clients abgewiesen
 - `sd` ist der Socket-Deskriptor aus dem vorhergehenden Aufruf von `socket()`
 - `backlog` enthält die Anzahl der möglichen Verbindungsanforderungen, die die Warteschlange maximal speichern kann
 - Standardwert: 5
 - Ein Server für Datagramme (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);
```

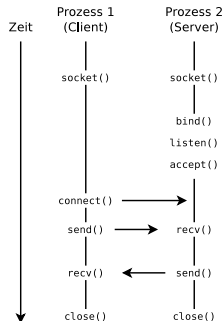
- Mit `accept()` holt der Server die erste Verbindungsanforderung aus der Warteschlange
- Der Rückgabewert ist der Socket-Deskriptor des neuen Sockets
- Enthält die Warteschlange keine Verbindungsanforderungen, ist der Prozess blockiert, bis eine Verbindungsanforderung eintrifft
- `address` enthält die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

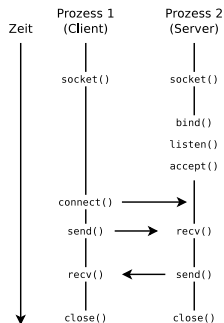
- Via `connect()` versucht der Client eine Verbindung mit einem Server-Socket herzustellen
- `sd` ist der Socket-Deskriptor
- `servaddr` ist die Adresse des Servers
- `addrlen` ist die Länge der Datenstruktur, die die Adresse enthält



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:

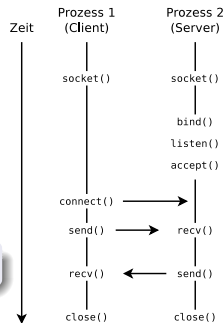
```
1 send(socket, "Hello World", 11, 0);  
2 write(socket, "Hello World", 11);
```

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

- _____



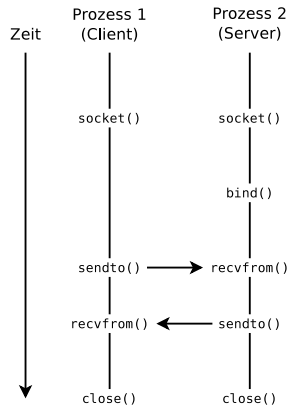
Sockets via UDP – Beispiel (Server)

```

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
19                                         binden
20     while True:
21         data = sd.recvfrom(1024)        # Daten empfangen
22         # Empfangene Daten ausgeben
23         print 'Received:', repr(data)
24 finally:
25     sd.close()                          # Socket schließen

```

```
$ python udp_server.py
```



Sockets via UDP – Beispiel (Client)

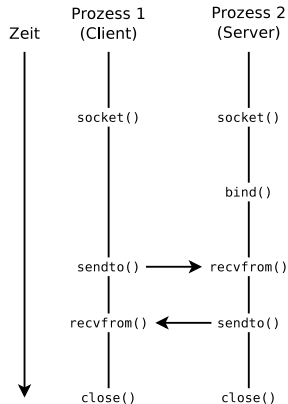
```

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))
```

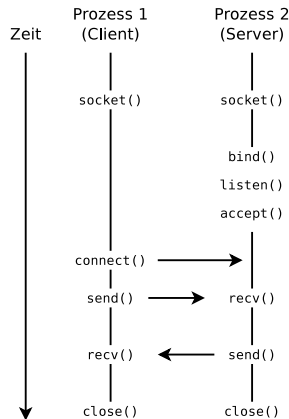


Sockets via TCP – Beispiel (Server)

```

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

```



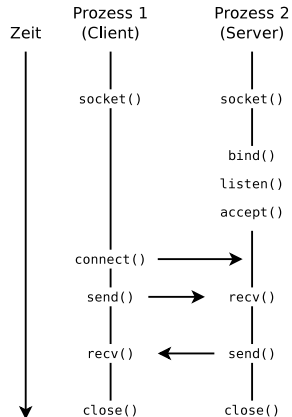
```
$ python tcp_server.py
```

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)
```

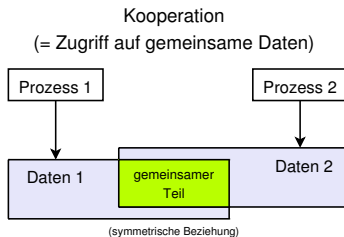
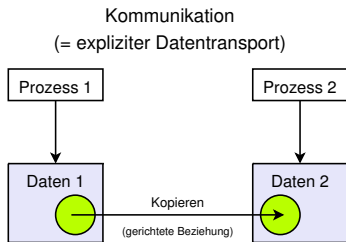
```
$ python tcp_client.py
Empfangen: 'Hello, world'
```

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



Kooperation

- Kooperation
 - Semaphore
 - Mutex



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

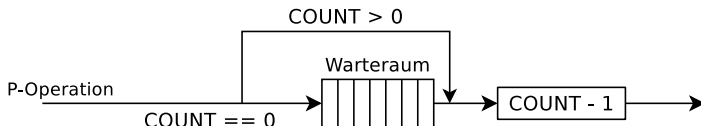
```
1 // Operation INIT auf Semaphor SEM anwenden
2 SEM.INIT(unsigned int init_wert) {
3
4     // Variable COUNT des Semaphors SEM mit einem
5     // nichtnegativen Anfangswert initialisieren
6     SEM.COUNT = init_wert;
7 }
```

Zugriffoperationen 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 }
```

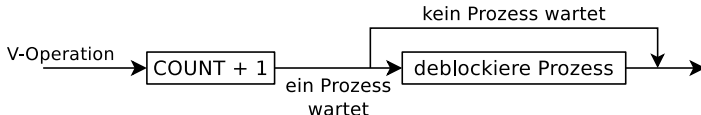


Zugriffsoperationen auf Semaphoren (3/3)

Bildquelle: Carsten Vogt

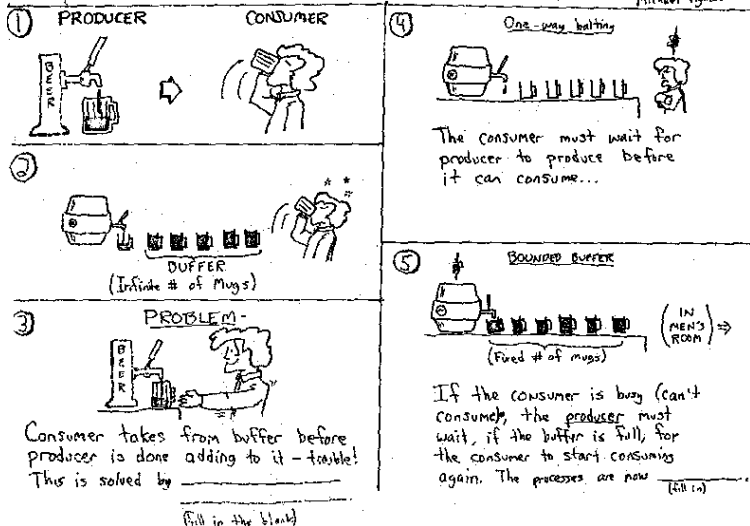
- **V-Operation** (*erhöhen*): Erhöht als erstes die Zählvariable um 1
 - Befinden sich Prozesse im Warteraum, wird ein Prozess deblockiert
 - Der gerade deblockierte Prozess setzt dann seine P-Operation fort und erniedrigt als erstes die Zählvariable

```
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 }
```



A GRAPHIC EXAMPLE OF THE PRODUCER/CONSUMER PROBLEM

Michael Vignaux



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` \implies Puffer vollständig belegt \implies Erzeuger blockieren
 - `voll` zählt die Datenpakete (belegte Plätze) im Puffer, wird vom Erzeuger (V-Operation) erhöht und vom Verbraucher (P-Operation) erniedrigt
 - `voll = 0` \implies Puffer leer \implies Verbraucher blockieren
- Semaphore `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 Semaphore `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 }

```

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 (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 >

```

Bildquelle: Carsten Vogt

-
- Gruppennummer
- Semaphorennummer innerhalb der Gruppe
- Semaphorentabelle
- 0 1 2 3 4 5
- 0 S_{00} S_{01} S_{02} S_{03} S_{04} S_{05}
- 1 S_{10} S_{11}
- 2 S_{20} S_{21} S_{22}
- 3 S_{30} S_{31} S_{32} S_{33} S_{34}
- ...
- n leer
- einzelnes Semaphor
- Semaphorengruppe

- `semget()`: Neues Semaphore oder eine Gruppe von Semaphoren erzeugen oder ein bestehendes Semaphore öffnen
- `semctl()`: Wert eines existierenden Semaphors oder einer Semaphorengruppe abfragen, ändern oder ein Semaphore 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 **Mutual Exclusion** = 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	⇒	entspricht der P-Operation
mutex_unlock	⇒	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

- 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`

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