

## 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

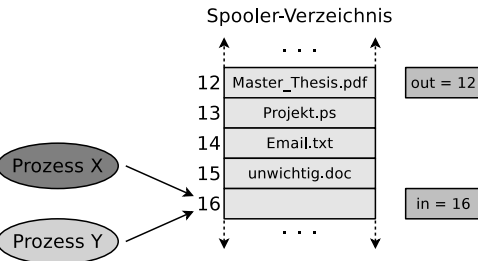
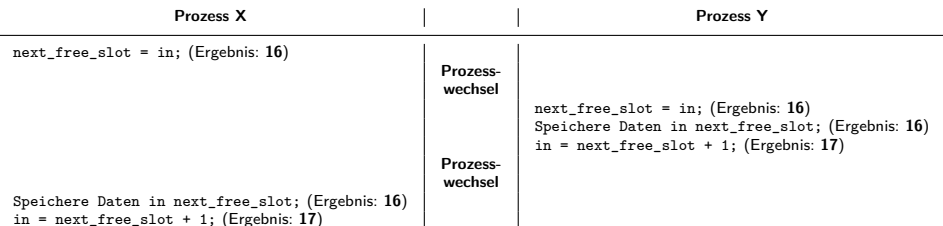




## Kritische Abschnitte

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

## Kritische Abschnitte – Beispiel: Drucker-Spooler

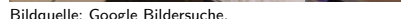


- Das Spooler-Verzeichnis ist konsistent
  - Aber der Eintrag von **Prozess Y** wurde von **Prozess X** überschrieben und ging verloren
- Eine solche Situation heißt **Race Condition**



\_\_\_\_\_

\_\_\_\_\_



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

<https://www.bugsnag.com/blog/bug-day-race-condition-therac-25>

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

[https://www-dssz.informatik.tu-cottbus.de/information/slides\\_studis/ss2009/mehner\\_RisikoComputer\\_zs09.pdf](https://www-dssz.informatik.tu-cottbus.de/information/slides_studis/ss2009/mehner_RisikoComputer_zs09.pdf)  
 Killer Bug. Therac-25: Quick-and-Dirty: <https://www.viva64.com/en/b/0438/>  
 Killed by a machine: The Therac-25: <https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/>

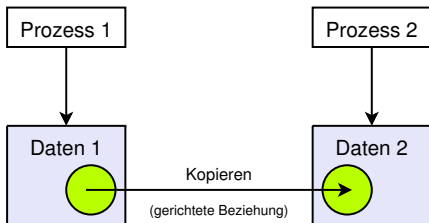


# Kommunikation vs. Kooperation

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

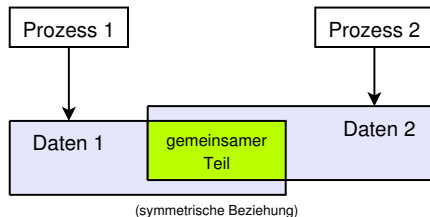
## Kommunikation

(= expliziter Datentransport)



## Kooperation

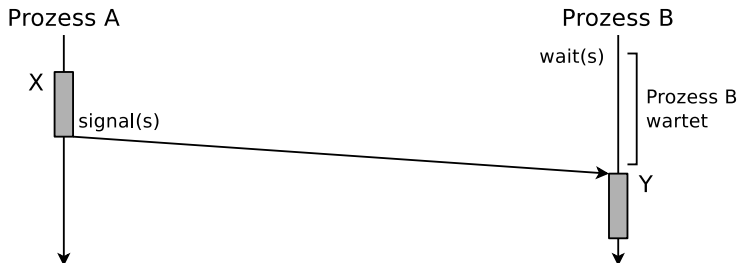
(= Zugriff auf gemeinsame Daten)



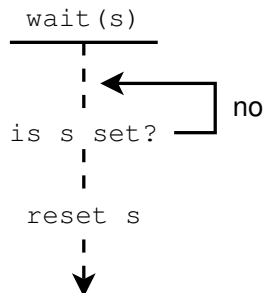
---

# Signalisierung

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



## Einfachste Form der Signalisierung (aktives Warten)

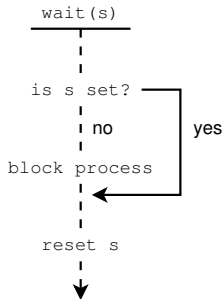
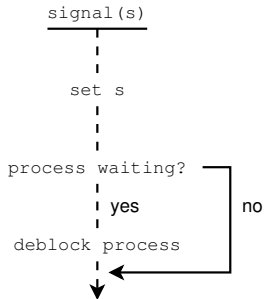


- Die Abbildung zeigt **aktives Warten** an der Signalvariable s
  - Die Signalvariable kann sich zum Beispiel in einer lokalen Datei befinden
  - Nachteil: Rechenzeit der CPU wird verschwendet, weil die wait-Operation den Prozessor in regelmäßigen Abständen belegt
- Diese Technik heißt auch **Warteschleife**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

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

# Signalisieren und Warten

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

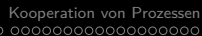


Eine Möglichkeit, um unter Linux eine Ausführungsreihenfolge mit passivem Warten festzulegen, ist die Funktion `sigsuspend`. Damit blockiert sich ein Prozess so lange selbst, bis ein anderer Prozess ihm mit der Funktion `kill` (oder dem gleichnamigen Systemaufruf) ein passendes Signal (meist `SIGUSR1`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

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

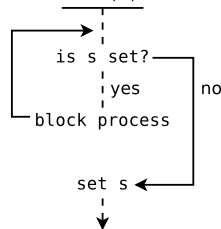
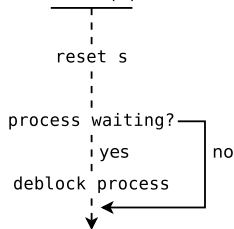
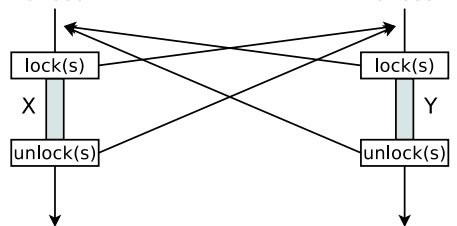
Kooperation von Prozessen  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

- Kooperation von Prozessen



- Kooperation von Prozessen  
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○

## Prozess A



sigsuspend, kill, pause und sleep

- 15/81





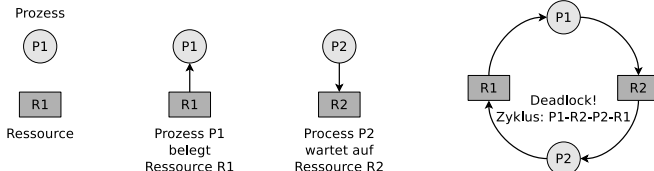


*System Deadlocks.* E. G. Coffman, M. J. Elphick, A. Shoshani. *Computing Surveys*, Vol. 3, No. 2, June 1971, S.67-78.  
[http://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman\\_deadlocks/coffman\\_deadlocks.pdf](http://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf)

- 18/81

# Betriebsmittel-Graphen

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



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

# Deadlock-Erkennung mit Matrizen

- Ein Nachteil der Deadlock-Erkennung mit Betriebsmittel-Graphen ist, dass man damit nur einzelne Ressourcen darstellen kann
  - Gibt es mehrere Kopien (Instanzen) einer Ressource, sind Graphen zur Darstellung bzw. Erkennung von Deadlocks ungeeignet
    - Existieren von einer Ressource mehrere Instanzen, kann ein matritzenbasiertes 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

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

- 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**

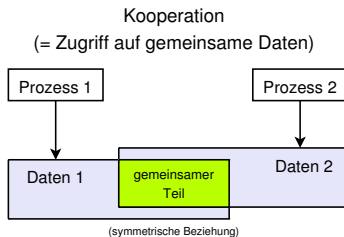
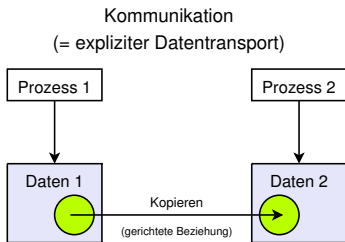




# Kommunikation von Prozessen

- Kommunikation

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



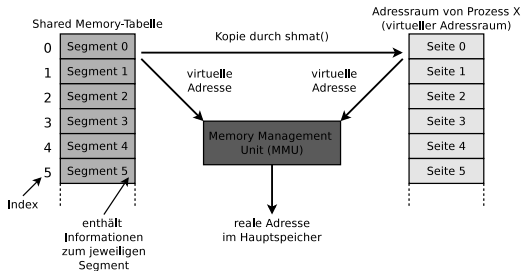


- Prozesskommunikation über einen gemeinsamen Speicher (Shared Memory) heißt auch **speicherbasierte Kommunikation**
- **Gemeinsame Speichersegmente** sind Speicherbereiche, auf die mehrere Prozesse direkt zugreifen können
  - Diese Speicherbereiche liegen im Adressraum mehrerer Prozesse
- Die Prozesse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

```
graph LR; A[Prozess X (Sender)  
exklusiv nutzbarer Speicher] --> B[Shared Memory  
gemeinsam nutzbarer Speicher]; B --> C[Prozess Y (Empfänger)  
exklusiv nutzbarer Speicher];
```

# 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

Beim Neustart des Betriebssystems sind die gemeinsamen Speichersegmente und deren Inhalte verloren



28/81

## Gemeinsames Speichersegment anhängen (in C)

```

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    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    if (sharedmempointer!=(char *)-1) {
19        printf("Das gemeinsame Speichersegment konnte nicht angehängt werden.\n");
20        perror("shmat");
21    } else {
22        printf("Das Segment wurde angehängt an Adresse %p\n", sharedmempointer);
23    }
24 }
25 }

```

```
$ ipcs -m
```

```
----- Shared Memory Segments -----
```

key	shmid	owner	perms	bytes	nattch	status
0x00003039	56393780	bnc	600	20	1	

```

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    // Gemeinsames Speichersegment anhängen
16    sharedmempointer = shmat(returncode_shmget, 0, 0);
17    ...
18
19    // Eine Zeichenkette in das gemeinsame Speichersegment schreiben
20    returncode_sprintf = sprintf(sharedmempointer, "Hallo Welt.");
21    if (returncode_sprintf < 0) {
22        printf("Der Schreibzugriff ist fehlgeschlagen.\n");
23    } else {
24        printf("%i Zeichen in das Segment geschrieben.\n", returncode_sprintf);
25    }
26
27    // Die Zeichenkette im gemeinsamen Speichersegment ausgeben
28    if (printf("%s\n", sharedmempointer) < 0) {
29        printf("Der Lesezugriff ist fehlgeschlagen.\n");
30    }
31

```

```

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 }

```

## Gemeinsames Speichersegment löschen (in C)

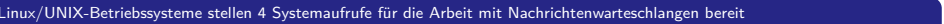
```

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_shmctl;
11    char *sharedmempointer;
12
13    // Gemeinsames Speichersegment erzeugen
14    returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
15    ...
16
17    // Gemeinsames Speichersegment löschen
18    returncode_shmctl = shmctl(returncode_shmget, IPC_RMID, 0);
19    if (returncode_shmctl == -1) {
20        printf("Das gemeinsame Speichersegment konnte nicht gelöscht werden.\n");
21        perror("semctl");
22    } else {
23        printf("Das Segment wurde gelöscht.\n");
24    }
25 }
26 }

```



- 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



- Informationen über bestehende Nachrichtenwarteschlangen 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    // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
11    // IPC_CREAT => neue Nachrichtenwarteschlange erzeugen, wenn sie noch nicht existiert
12    // 0600 = Zugriffsrechte auf die neue Nachrichtenwarteschlange
13    returncode_msgget = msgget(12345, IPC_CREAT | 0600);
14    if(returncode_msgget < 0) {
15        printf("Die Nachrichtenwarteschlange konnte nicht erstellt werden.\n");
16        exit(1);
17    } else {
18        printf("Die Nachrichtenwarteschlange 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
```



# 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 }

```

## Nachrichtenwarteschlangen löschen (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     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  $\implies$  der in die Pipe schreibende Prozess wird blockiert
  - Pipe = leer  $\implies$  der aus der Pipe lesende Prozess wird blockiert
- wird mit dem Systemaufruf `pipe()` angelegt
  - Dabei erzeugt der Betriebssystemkern einen Inode ( $\implies$  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 1/2

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

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6     int pid_des_Kindes;
7     // Zugriffskennungen zum Lesen (testpipe[0]) und Schreiben (testpipe[1]) anlegen
8     int testpipe[2];
9
10    // Die Pipe testpipe anlegen
11    if (pipe(testpipe) < 0) {
12        printf("Das Anlegen der Pipe ist fehlgeschlagen.\n");
13        // Programmabbruch
14        exit(1);
15    } else {
16        printf("Die Pipe testpipe wurde angelegt.\n");
17    }
18
19    // Einen Kindprozess erzeugen
20    pid_des_Kindes = fork();
21
22    // Es kam beim fork zu einem Fehler
23    if (pid_des_Kindes < 0) {
24        perror("Es kam bei fork zu einem Fehler!\n");
25        // Programmabbruch
26        exit(1);
27    }

```

# 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.
```

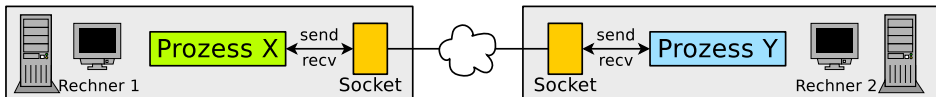
```
Elternprozess: PID: 6363
```

```
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



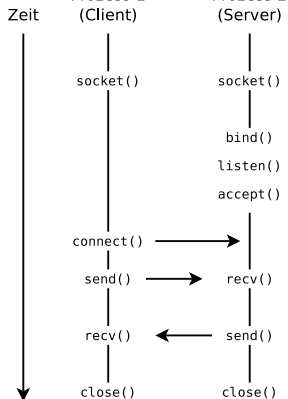
# Sockets nutzen

- 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`



- Client



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

- 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   }
```





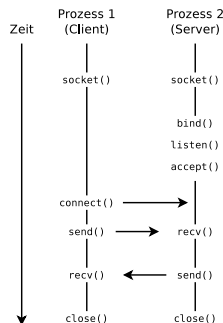
```
int listen(int sd, int backlog);
```

- 
- ```
sequenceDiagram
    participant P1 as Prozess 1 (Client)
    participant P2 as Prozess 2 (Server)
    P1->>P1: socket()
    P2->>P2: socket()
    P2->>P2: bind()
    P2->>P2: listen()
    P2->>P2: accept()
    P1->>P1: connect()
    P1->>P2: send()
    P2->>P1: recv()
    P1->>P1: recv()
    P1->>P1: close()
    P2->>P2: send()
    P2->>P2: close()
```

# 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

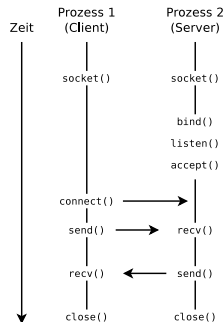




# Verbindungsorientierter Datenaustausch: send und recv

```
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











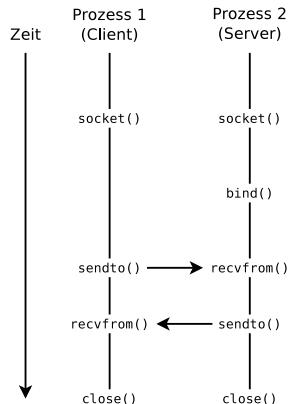
## Sockets via UDP – Beispiel (Server)

```

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)

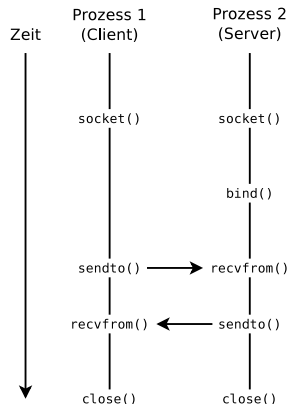
```

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))
```

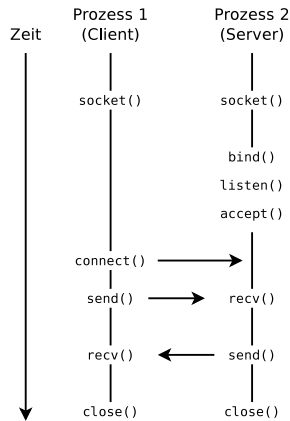


## Sockets via TCP – Beispiel (Server)

```

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 sd.close()                                  # Socket schließen

```



```
$ python tcp_server.py
```

## Sockets via TCP – Beispiel (Client)

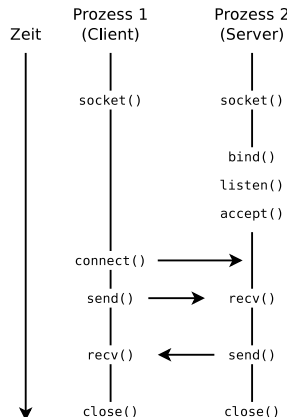
```

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)
```













# Semaphoren

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

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

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

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

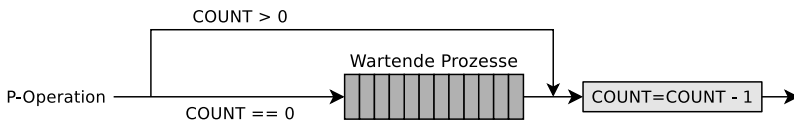
Ein Semaphore besteht aus 2 Datenstrukturen

- ```
// Operation INIT auf Semaphor SEM anwenden
SEM.INIT(unsigned int init_wert) {

    // Variable COUNT des Semaphors SEM mit einem
    // nichtnegativen Anfangswert initialisieren
    SEM.COUNT = init_wert;
}
```

Bildquelle: Carsten Vogt

- ```
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 }
```

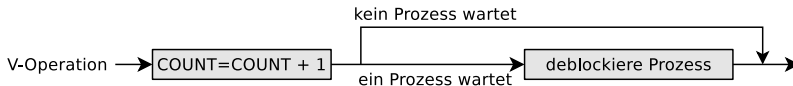


Bildquelle: Carsten Vogt

- ```

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 }

```



Das Programm erstellt einen Kindprozess. Eltern- und Kindprozess versuchen beide, Zeichen in der Shell auszugeben (kritischer Abschnitt). Die Prozesse sollen abwechselnd je ein Zeichen ausgeben. Zwei Semaphore ermöglichen den gegenseitigen Ausschluss.

```

1 #include <stdio.h>    // für printf
2 #include <stdlib.h>   // für exit
3 #include <unistd.h>   // für read, write, close
4 #include <sys/wait.h> // für wait
5 #include <sys/sem.h>  // für semget, semctl, semop
6
7 void main() {
8     int pid_des_kindes;
9     int sem_key1=12345;
10    int sem_key2=54321;
11    int returncode_semget1, returncode_semget2, returncode_semctl;
12    int output;
13
14    setbuf(stdout, NULL); // Das Puffern Standardausgabe (stdout) unterbinden
15
16    // Neue Semaphorgruppe 12345 mit einer Semaphore erstellen
17    // IPC_CREAT = Semaphore erzeugen, wenn Sie noch nicht existiert
18    // IPC_EXCL = Neuen Semaphorgruppe anlegen und nicht auf evtl. existierende Gruppe zugreifen
19    returncode_semget1 = semget(sem_key1, 1, IPC_CREAT | IPC_EXCL | 0600);
20    if (returncode_semget1 < 0) {
21        printf("Die Semaphorgruppe %i konnte nicht erstellt werden.\n", sem_key1);
22        perror("semget");
23        exit(1);
24    }

```

## Gute Dokumentation von semget

<https://www.nt.th-koeln.de/fachgebiete/inf/diplom/semwork/unix/semget/semget.html>

## Ein einfaches Beispiel zu Semaphore (in C) – Teil 2/5

```

25 // Neue Semaphoregruppe 54321 mit einer Semaphore erstellen
26 returncode_semget2 = semget(sem_key2, 1, IPC_CREAT | IPC_EXCL | 0600);
27 if (returncode_semget2 < 0) {
28     printf("Die Semaphoregruppe %i konnte nicht erstellt werden.\n", sem_key2);
29     perror("semget");
30     exit(1);
31 }
32
33 // P-Operation definieren. Wert der Semaphore um eins dekrementieren
34 struct sembuf p_operation = {0, -1, 0};
35
36 // V-Operation definieren. Wert der Semaphore um eins inkrementieren
37 struct sembuf v_operation = {0, 1, 0};
38
39 // Erste Semaphore der Semaphoregruppe 12345 initial auf Wert 1 setzen
40 returncode_semctl = semctl(returncode_semget1, 0, SETVAL, 1);
41
42 // Erste Semaphore der Semaphoregruppe 54321 initial auf Wert 0 setzen
43 returncode_semctl = semctl(returncode_semget2, 0, SETVAL, 0);
44
45 // Initialen Wert der ersten Semaphore der Semaphoregruppe 12345 zur Kontrolle ausgeben
46 output = semctl(returncode_semget1, 0, GETVAL, 0);
47 printf("Wert der Semaphore mit ID %i und Key %i: %i\n", returncode_semget1, sem_key1, output);
48
49 // Initialen Wert der ersten Semaphore der Semaphoregruppe 54321 zur Kontrolle ausgeben
50 output = semctl(returncode_semget2, 0, GETVAL, 0);
51 printf("Wert der Semaphore mit ID %i und Key %i: %i\n", returncode_semget2, sem_key2, output);

```

Gute Dokumentation von semctl

<https://www.nt.th-koeln.de/fachgebiete/inf/diplom/semwork/unix/semctl/semctl.html>

```

52 // Einen Kindprozess erzeugen
53 pid_des_kindess = fork();
54
55 // Kindprozess
56 if (pid_des_kindess == 0) {
57     for (int i=0;i<5;i++) {
58         semop(returncode_semget2, &p_operation, 1); // P-Operation Semaphore 54321
59         // Kritischer Abschnitt (Anfang)
60         printf("B");
61         sleep(1);
62         // Kritischer Abschnitt (Ende)
63         semop(returncode_semget1, &v_operation, 1); // V-Operation Semaphore 12345
64     }
65     exit(0);
66 }
67
68 // Elternprozess
69 if (pid_des_kindess > 0) {
70     for (int i=0;i<5;i++) {
71         semop(returncode_semget1, &p_operation, 1); // P-Operation Semaphore 12345
72         // Kritischer Abschnitt (Anfang)
73         printf("A");
74         sleep(1);
75         // Kritischer Abschnitt (Ende)
76         semop(returncode_semget2, &v_operation, 1); // V-Operation Semaphore 54321
77     }
78 }

```

## Gute Dokumentation von semop

<https://www.nt.th-koeln.de/fachgebiete/inf/diplom/semwork/unix/semop/semop.html>





```
$ gcc semaphore_beispiel_systemv.c -o semaphore_beispiel_systemv
Wert der Semaphore mit ID 98362 und Key 12345: 1
Wert der Semaphore mit ID 98363 und Key 54321: 0
ABABABABAB
Die Semaphorgruppe mit ID 98362 und Key 12345 wurde entfernt.
Die Semaphorgruppe mit ID 98363 und Key 54321 wurde entfernt.
```

```
$ printf "%d\n" 0x00003039          # Convert from hexadecimal to decimal
12345
$ printf "%d\n" 0x0000d431
54321
```

- 73/81







## 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 }

```

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, 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**
    - Mutexe ermöglichen Intra- sowie intra-Prozesssynchronisation

## Funktionen zum Zugriff auf einen Mutex

pthread\_mutex\_lock  $\Rightarrow$  entspricht der P-Operation  
pthread\_mutex\_unlock  $\Rightarrow$  entspricht der V-Operation

- Will ein Prozess auf den kritischen Abschnitt zugreifen, ruft er `pthread_mutex_lock` auf
  - Kritischer Abschnitt **gesperrt**  $\Rightarrow$  Prozess ist blockiert bis der Prozess im kritischen Abschnitt fertig ist und `pthread_mutex_unlock` aufruft
  - Kritischer Abschnitt **nicht gesperrt**  $\Rightarrow$  Prozess kann eintreten

Mehr Informationen über Mutexe

<https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>



