

## 9. Foliensatz Betriebssysteme

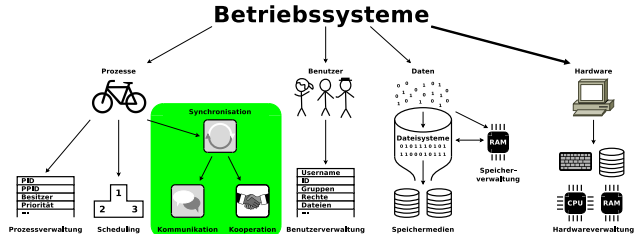
Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences  
(1971–2014: Fachhochschule Frankfurt am Main)  
Fachbereich Informatik und Ingenieurwissenschaften  
[christianbaun@fb2.fra-uas.de](mailto: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
    - mögliche Probleme (**Verhungern** und **Deadlocks**) beim Blockieren
    - wie **Deadlock-Erkennung mit Matrizen** funktioniert
  - verschiedene Möglichkeiten der **Kommunikation** zwischen Prozessen:
    - **Gemeinsamer Speicher, Nachrichtenwarteschlangen, Pipes, Sockets**
  - verschiedene Möglichkeiten der **Kooperation** von Prozessen
    - wie **Semaphore** und **Mutexe** kritische Abschnitte sichern können

Übungsblatt 9 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes



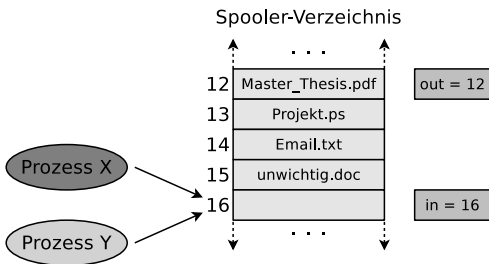


## Kritische Abschnitte

- 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

Prozess X		Prozess Y
next_free_slot = in; (Ergebnis: 16)		
	Prozesswechsel	next_free_slot = in; (Ergebnis: 16) Speichere Daten in next_free_slot; (Ergebnis: 16) in = next_free_slot + 1; (Ergebnis: 17)
Speichere Daten in next_free_slot; (Ergebnis: 16) in = next_free_slot + 1; (Ergebnis: 17)	Prozesswechsel	



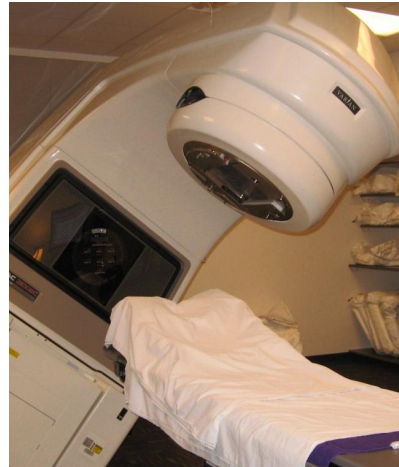
- Das Spooler-Verzeichnis ist konsistent
  - Aber der Eintrag von **Prozess Y** wurde von **Prozess X** überschrieben und ging verloren
- Eine solche Situation heißt **Race Condition**

## Race Condition (*Wettlaufsituation*)

- **Unbeabsichtigte Wettlaufsituation** zweier Prozesse, die den Wert der gleichen Speicherstelle ändern wollen
  - Das Ergebnis eines Prozesses hängt von der Reihenfolge oder dem zeitlichen Ablauf anderer Ereignisse ab
  - Häufiger Grund für schwer auffindbare Programmfehler
- Problem: Das Auftreten und die Symptome hängen von unterschiedlichen Ereignissen ab
  - Bei jedem Testdurchlauf können die Symptome unterschiedlich sein oder verschwinden
- Vermeidung ist u.a durch das Konzept der **Semaphore** ( $\implies$  Folie 60) möglich

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

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



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

*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](http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html)

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

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

The Worst Computer Bugs in History: Race conditions in Therac-25:

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

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

### Weitere interessante Quellen

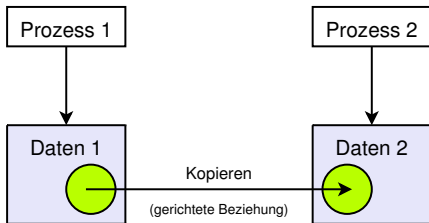
[https://www.dssz.informatik.tu-cottbus.de/information/slides\\_studis/ss2009/mehner\\_RisikoComputer\\_zs09.pdf](https://www.dssz.informatik.tu-cottbus.de/information/slides_studis/ss2009/mehner_RisikoComputer_zs09.pdf)

Killer Bug. Therac-25: Quick-and-Dirty: <https://www.viva64.com/en/b/0438/>

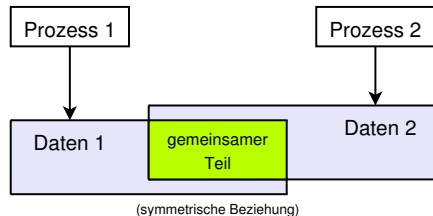
Killed by a machine: The Therac-25: <https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/>



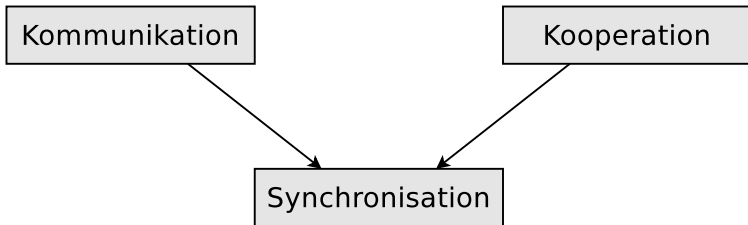
.....



1

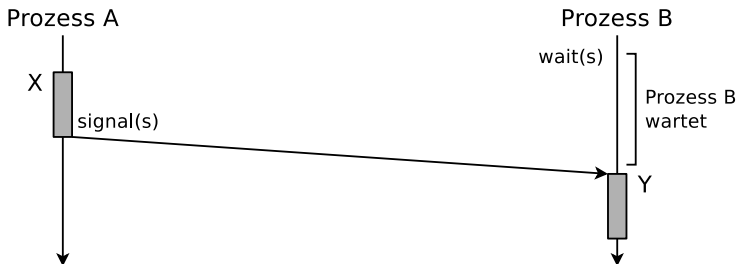


- Kommunikation und Kooperation basieren auf Synchronisation
  - Synchronisation ist die elementarste Form der Interaktion
    - Grund: Kommunikation und Kooperation benötigen eine zeitliche Abstimmung zwischen den Interaktionspartnern, um korrekte Ergebnisse zu erhalten
  - Darum behandeln wir zuerst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



# Signalisierung

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







## Schutz kritischer Abschnitte durch Sperren / Blockieren

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



## Sperren 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







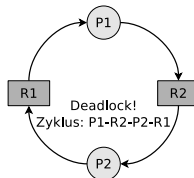
# Betriebsmittel-Graphen

- 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



Ressource

Prozess P1  
belegt  
Ressource R1Process P2  
wartet auf  
Ressource R2

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 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







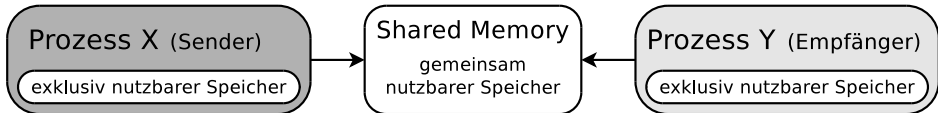




## Gemeinsamer Speicher – Shared Memory

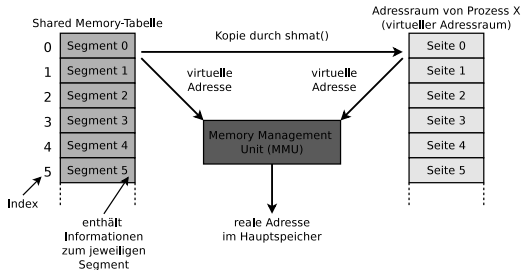
- Prozesskommunikation über einen gemeinsamen Speicher (Shared Memory) heißt auch **speicherbasierte Kommunikation**
- **Gemeinsame Speichersegmente** sind Speicherbereiche, auf die mehrere Prozesse direkt zugreifen können
  - Diese Speicherbereiche liegen im Adressraum mehrerer Prozesse
- Die Prozesse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

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



# Gemeinsamer Speicher unter Linux/UNIX

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



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

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

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

# Mit gemeinsamem Speicher arbeiten (System V vs. POSIX)

Linux/UNIX-Betriebssysteme stellen 4 Systemaufrufe (System V) für die Arbeit mit gemeinsamem Speicher bereit

- `shmget()`: Gemeinsames Speichersegment erzeugen oder auf ein bestehendes zugreifen
- `shmat()`: Gemeinsames Speichersegment an einen Prozess anhängen
- `shmdt()`: Gemeinsames Speichersegment von einem Prozess lösen/freigeben
- `shmctl()`: Status (u.a. Zugriffsrechte) eines gemeinsamen Speichersegments abfragen, ändern oder es löschen
- Informationen über bestehende gemeinsame Speichersegmente (System V) liefert das Kommando `ipcs`

Ein Beispiel zu gemeinsamen Speicherbereichen (**System V**) unter Linux finden auf der Webseite der Vorlesung

- Einige Entwickler bevorzugen die System V API und andere die POSIX API. ... ㄟ( ͡ʷ)ㄟ

C-Funktionsaufrufe für gemeinsame POSIX-Speichersegmente (teilweise in der Header-Datei `mman.h` definiert)

- `shm_open()`: Ein Segment erzeugen oder auf ein bestehendes zugreifen
- `ftruncate()`: Die Größe eines Speichersegments definieren
- `mmap()`: Ein Segment an einen Prozess anhängen
- `munmap()`: Ein Segment von einem Prozess lösen/freigeben
- `close()`: Den Deskriptor eines Speichersegments schließen
- `shm_unlink()`: Ein Segment löschen
- Unter Linux liegen POSIX-Speichersegmente im Verzeichnis `/dev/shm`

Ein Beispiel zu gemeinsamen POSIX-Speichersegmenten unter Linux finden sie auf der Webseite der Vorlesung



# Gemeins. Speichersegment (System V) 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	

## In ein Speichersegment (System V) schreiben und daraus lesen (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, returncode_shmctl, 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    ...
}
```

# Gemeinsames Speichersegment (System V) lösen (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_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 }
```

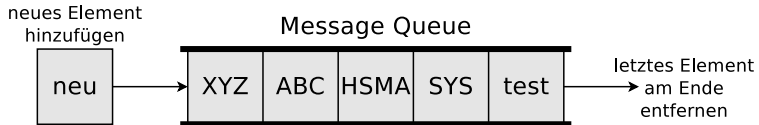
# Gemeinsames Speichersegment (System V) löschen (in C)

```
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 }
```



# 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



Linux/UNIX-Betriebssysteme stellen 4 Systemaufrufe (System V) für die Arbeit mit Nachrichtewarteschlangen bereit

- `msgget()`: Nachrichtewarteschlange erzeugen oder auf eine bestehende zugreifen
- `msgsnd()`: Nachricht in Nachrichtewarteschlange schreiben (schicken)
- `msgrcv()`: Nachricht aus Nachrichtewarteschlange lesen (empfangen)
- `msgctl()`: Status (u.a. Zugriffsrechte) einer Nachrichtewarteschlange abfragen, ändern oder sie löschen
- Informationen über bestehende Nachrichtewarteschlangen (System V) liefert das Kommando `ipcs`

# Nachrichtewarteschlangen (System V) 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    }
```

```
$ 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 (System V) 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 (System V) 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     }
```

# Nachrichtewarteschlangen (System V) 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     // Nachrichtewarteschlange erzeugen oder auf eine bestehende zugreifen
12     returncode_msgget = msgget(12345, IPC_CREAT | 0600);
13     ...
14
15     // Nachrichtewarteschlange löschen
16     returncode_msgctl = msgctl(returncode_msgget, IPC_RMID, 0);
17     if (returncode_msgctl < 0) {
18         printf("Die Nachrichtewarteschlange mit der ID %i konnte nicht gelöscht werden.\n",
19             returncode_msgget);
20         perror("msgctl");
21         exit(1);
22     } else {
23         printf("Die Nachrichtewarteschlange mit der ID %i wurde gelöscht.\n",
24             returncode_msgget);
25     }
26     exit(0);
27 }
```

Ein Beispiel zur Arbeit mit System V-Nachrichtewarteschlangen unter Linux finden sie auf der Webseite der Vorlesung

# Nachrichtenwarteschl. unter Linux (System V vs. POSIX)

- Die bislang beschriebenen Funktionen zur Arbeit mit Nachrichtenwarteschlangen sind Teil der **System V**-Schnittstelle
- Einige Entwickler bevorzugen die System V API und andere die POSIX API... ㄟ( ㄣ ) ㄟ

In der Header-Datei `mqueue.h` definierte C-Funktionsaufrufe der POSIX-Semaphoren

- `mq_open()`: Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
- `mq_send()`: Nachricht in eine Nachrichtenwarteschlange schreiben (schicken). Blockierende Anweisung
- `mq_timedsend()`: Nachricht in eine Nachrichtenwarteschlange schreiben (schicken). Blockierende Anweisung mit Timeout
- `mq_receive()`: Nachricht aus einer Nachrichtenwarteschlange lesen (empfangen). Blockierende Anweisung
- `mq_timedreceive()`: Nachricht aus einer Nachrichtenwarteschlange lesen (empfangen). Blockierende Anweisung mit Timeout
- `mq_getattr()`: Eigenschaften einer Nachrichtenwarteschlange abfragen. Diese sind: Anzahl der Nachrichten in der Warteschlange, maximale Nachrichtengröße, maximale Anzahl an Nachrichten, etc.
- `mq_setattr()`: Eigenschaften einer Nachrichtenwarteschlange ändern
- `mq_notify()`: Der Prozess soll benachrichtigt werden, sobald eine Nachricht vorliegt
- `mq_close()`: Nachrichtenwarteschlange schließen
- `mq_unlink()`: Nachrichtenwarteschlange löschen
- Unter Linux liegen POSIX-Speichersegmente im Verzeichnis `/dev/mqueue`

Ein Beispiel zur Arbeit mit benannten POSIX-Nachrichtenwarteschlangen unter Linux finden sie auf der Webseite der Vorlesung

# Anonyme Pipes (1/2)

- Pipes können **anonyme** oder **benannte Pipes** (siehe Folie 44) sein
- 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



## Anonyme 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

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

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

Sie können die anonyme Pipe unter Linux/UNIX mit `ls -l | grep <PID>` und im Verzeichnis `/proc/<PID>/fd` überwachen

```
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 anonymen Pipe ist fehlgeschlagen.\n");
13        // Programmabbruch
14        exit(1);
15    } else {
16        printf("Die anonyme Pipe testpipe wurde angelegt.\n");
17    }
18
19    // Einen Kindprozess erzeugen
20    pid_des_Kindes = fork();
21
22    if (pid_des_Kindes < 0) {
23        perror("Es kam bei fork zu einem Fehler!\n");
24        // Programmabbruch
25        exit(1);
26    }
```

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

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

```
$ gcc anonyme_pipe_beispiel.c -o anonyme_pipe_beispiel
$ ./anonyme_pipe_beispiel
Die anonyme Pipe testpipe wurde angelegt.
Elternprozess: PID: 6363
Kindprozess: PID: 6364
Empfangen: Testnachricht
```

# Benannte Pipes

- 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
- Benannte Pipes werden vom Betriebssystem nicht automatisch gelöscht (im Gegensatz zu anonymen Pipes)

# Ein Beispiel zu benannten Pipes (in C) – Teil 1/4

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <fcntl.h>
5 #include <sys/stat.h>
6
7 void main() {
8     int pid_des_Kindes;
9
10    // Die benannte Pipe anlegen
11    if (mkfifo("testfifo", 0666) < 0) {
12        printf("Das Anlegen der benannten Pipe ist fehlgeschlagen.\n");
13        exit(1);
14    } else {
15        printf("Die benannte Pipe testfifo wurde angelegt.\n");
16    }
17
18    // Einen Kindprozess erzeugen
19    pid_des_Kindes = fork();
20
21    if (pid_des_Kindes < 0) {
22        perror("Es kam bei fork zu einem Fehler!\n");
23        exit(1);
24    }
```

Der Funktionsaufruf erzeugt im aktuellen Verzeichnis einen Dateisystemeintrag mit dem Namen testfifo. Der erste Buchstabe in der Ausgabe des Kommandos ls zeigt, dass testfifo eine benannte Pipe ist.

```
$ ls -la testfifo
```

```
prw-r--r-- 1 bnc bnc 0 1. Feb 10:15 testfifo
```

# Ein Beispiel zu benannten Pipes (in C) – Teil 2/4

```
25 // Elternprozess
26 if (pid_des_Kindes > 0) {
27     printf("Elternprozess: PID: %i\n", getpid());
28
29     // Zugriffskennung für die benannte Pipe anlegen
30     int fd;
31
32     // Die zu übertragene Nachricht definieren
33     char nachricht[] = "Testnachricht";
34
35     // Die benannte Pipe für Schreibzugriffe öffnen
36     fd = open("testfifo", O_WRONLY);
37
38     // Daten in die benannte Pipe schreiben
39     write(fd, &nachricht, sizeof(nachricht));
40
41     // Die benannte Pipe schließen
42     close(fd);
43 }
```

# Ein Beispiel zu benannten Pipes (in C) – Teil 3/4

```
44 // Kindprozess
45 if (pid_des_Kindes == 0) {
46     printf("Kindprozess: PID: %i\n", getpid());
47
48     // Zugriffskennung für die benannte Pipe anlegen
49     int fd;
50     // Einen Empfangspuffer anlegen
51     char puffer[80];
52
53     // Die benannte Pipe für Lesezugriffe öffnen
54     fd = open("testfifo", O_RDONLY);
55
56     // Daten aus der Pipe auslesen
57     read(fd, puffer, sizeof(puffer));
58     printf("Empfangen: %s\n", puffer);
59
60     // Die beannte Pipe schließen
61     close(fd);
62
63     // Die benannte Pipe löschen
64     if (unlink("testfifo") < 0) {
65         printf("Das Löschen der benannten Pipe ist fehlgeschlagen.\n");
66         exit(1);
67     } else {
68         printf("Die benannte Pipe wurde gelöscht.\n");
69     }
70 }
71 }
```

# Ein Beispiel zu benannten Pipes (in C) – Teil 4/4

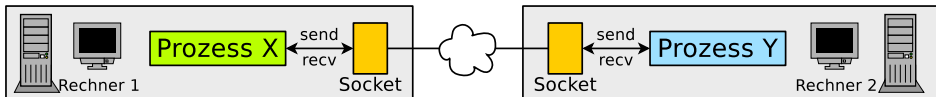
```
$ gcc benannte_pipe_beispiel.c -o benannte_pipe_beispiel
$ ./benannte_pipe_beispiel
Die benannte Pipe testfifo wurde angelegt.
Elternprozess: PID: 403660
Kindprozess: PID: 403661
Empfangen: Testnachricht
Die benannte Pipe wurde gelöscht.
```

Sie können die benannte Pipe unter Linux/UNIX mit `lsuf -n -P | grep <PID>` und im Verzeichnis `/proc/<PID>/fd` überwachen



# 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

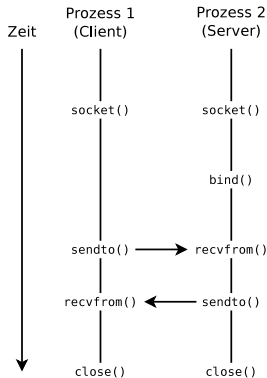
# Sockets nutzen

- 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`

Beispiele zur Interprozesskommunikation via Sockets (TCP and UDP) unter Linux finden Sie auf der Webseite der Vorlesung

# 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`)

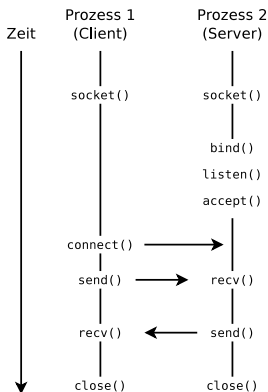
# Verbindungsorientierte Kommunikation mit Sockets – TCP

## • Client

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

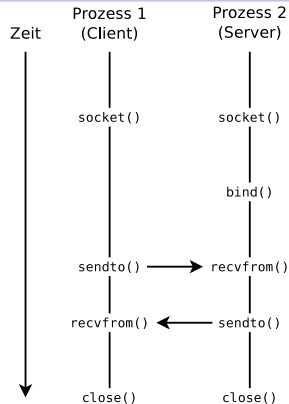
## • Server

- Socket erstellen (`socket`)
- Socket an einen Port binden (`bind`)
- Socket empfangsbereit machen (`listen`)
  - Warteschlange für Verbindungsanfragen einrichten. Definiert wie viele Verbindungsanfragen gepuffert werden können
- Verbindungsanforderung akzeptieren (`accept`)
  - Erste Verbindungsanforderung aus der Warteschlange holen
- Daten senden (`send`) und empfangen (`recv`)
- Socket schließen (`close`)



# Sockets via UDP – Beispiel (Server)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[]) {
10     int sd, adresse_laenge;
11     char puffer[1024] = { 0 };
12     struct sockaddr_in adresse, client_adresse;
13     memset(&adresse, 0, sizeof(adresse));
14     memset(&client_adresse, 0, sizeof(client_adresse));
15     adresse.sin_family = AF_INET;
16     adresse.sin_addr.s_addr = INADDR_ANY;
17     adresse.sin_port = htons(atoi(argv[1]));
18
19     sd = socket(AF_INET, SOCK_DGRAM, 0);
20     bind(sd, (struct sockaddr *) &adresse, sizeof(adresse));
21     adresse_laenge = sizeof(client_adresse);
22     recvfrom(sd, (char *)puffer, sizeof(puffer), 0,
23             (struct sockaddr *) &client_adresse, &adresse_laenge);
24     printf("Empfangene Nachricht: %s\n", puffer);
25     char antwort[] = "Server: Nachricht empfangen.\n";
26     sendto(sd, (const char *)antwort, sizeof(antwort), 0,
27           (struct sockaddr *) &client_adresse, adresse_laenge);
28     close(sd);
29     exit(0);
30 }
```



```
$ gcc udp_server.c -o udp_server
$ ./udp_server 50002
```





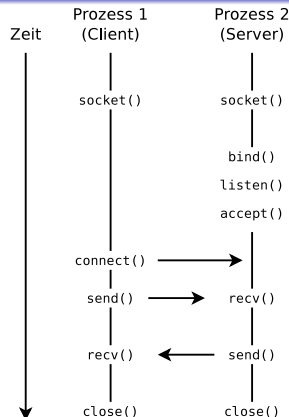


## Sockets via TCP – Beispiel (Client)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[]) {
10     int sd;
11     char puffer[1024] = { 0 };
12     struct sockaddr_in adresse;
13     memset(&adresse, 0, sizeof(adresse));
14     adresse.sin_family = AF_INET;
15     adresse.sin_port = htons(atoi(argv[2]));
16     adresse.sin_addr.s_addr = inet_addr(argv[1]);
17
18     sd = socket(AF_INET, SOCK_STREAM, 0);
19     connect(sd, (struct sockaddr *) &adresse, sizeof(adresse));
20
21     printf("Bitte Nachricht eingeben: ");
22     fgets(puffer, sizeof(puffer), stdin);
23     write(sd, puffer, strlen(puffer));
24     memset(puffer, 0, sizeof(puffer));
25     read(sd, puffer, sizeof(puffer));
26     printf("%s\n", puffer);
27
28     close(sd);
29     exit(0);
30 }

```



```
$ gcc tcp_client.c -o tcp_client
$ ./tcp_client 127.0.0.1 50003
Bitte Nachricht eingeben: Test
Server: Nachricht empfangen.
```

```
$ ./tcp_server 50003
Empfangene Nachricht: Test
```

# Vergleich der Kommunikations-Systeme

	Gemeinsamer Speicher	Nachrichtenwarteschlangen	(anon./benannte) Pipes	Sockets
<b>Art der Kommunikation</b>	Speicherbasiert	Nachrichtenbasiert	Nachrichtenbasiert	Nachrichtenbasiert
<b>Bidirektional</b>	ja	nein	nein	ja
<b>Plattformunabhängig</b>	nein	nein	nein	ja
<b>Prozesse müssen verwandt sein</b>	nein	nein	bei anonymen Pipes	nein
<b>Kommunikation über Rechnergrenzen</b>	nein	nein	nein	ja
<b>Bleiben ohne gebundenen Prozess erhalten</b>	ja	ja	nein	nein
<b>Automatische Synchronisierung</b>	nein	ja	ja	ja

- Vorteile nachrichtenbasierte vs. speicherbasierte Kommunikation:
  - Das Betriebssystem nimmt den Benutzerprozessen die Synchronisation der Zugriffe ab  $\implies$  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





# Zugriffsoperationen auf Semaphoren (1/3)

Ein Semaphor besteht aus 2 Datenstrukturen

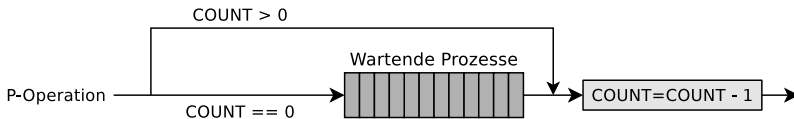
- **COUNT:** Eine **ganzzahlige, nichtnegative Zählvariable**.  
Gibt an, wie viele Prozesse das Semaphor aktuell ohne Blockierung passieren dürfen
- Ein Warteraum für die Prozesse, die darauf **warten**, das Semaphor passieren zu dürfen.  
Die Prozesse sind im Zustand **blockiert** und warten darauf, 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 }
```

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 }
```



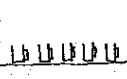
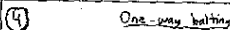
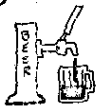






### A GRAPHIC EXAMPLE OF THE PRODUCER/CONSUMER PROBLEM

Michael Vignaro



The consumer must wait for producer to produce before it can consume...



DUFFER.  
(Infinite # of Mugs)



### PROBLEM -



Consumer takes from buffer before producer is done adding to it - trouble!  
This is solved by \_\_\_\_\_

Fill in the blanks



## BOUNDED BUFFER



(Fired # of mugs)

(IN MEN'S ROOM)  $\Rightarrow$

If the consumer is busy (can't consume), the producer must wait, if the buffer is full, for the consumer to start consuming again. The processes are now full in

(File in)

## 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
    - $\text{leer} = 0 \implies$  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
    - $\text{voll} = 0 \implies$  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

```

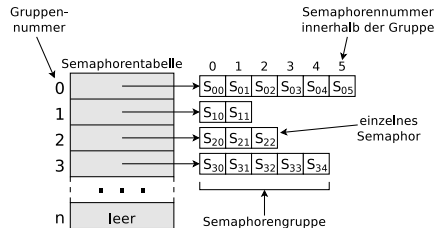
1 typedef int semaphore;           // Semaphore sind von Typ Integer
2 semaphore voll = 0;              // zählt die belegten Plätze im Puffer
3 semaphore leer = 8;              // zählt die freien Plätze im Puffer
4 semaphore mutex = 1;             // steuert Zugriff auf kritische Bereiche
5
6 void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

```

# Semaphoren unter Linux (System V)

Bildquelle: Carsten Vogt

- Linux weicht vom Konzept der Semaphore nach Dijkstra ab
    - Die Zählvariable kann mit einer P- oder V-Operation um mehr als 1 erhöht bzw. erniedrigt werden
    - Es können mehrere Zugriffsoperationen auf verschiedenen Semaphoren atomar, also unteilbar, durchgeführt werden
  - Linux-Systeme verwalten eine Semaphortabelle, die Verweise auf Arrays mit Semaphoren enthält
    - Einzelne Semaphore werden über den Tabellenindex und die Position in der Gruppe angesprochen
- 
- Das Diagramm zeigt die Struktur einer Semaphortabelle in Linux. Die Tabelle ist eine Matrix, die in Gruppen und Semaphoren unterteilt ist. Die Spaltenüberschriften sind die Semaphorennummern (0 bis 5), und die Zeilenüberschriften sind die Gruppennummern (0 bis n). Die Einträge in der Tabelle sind Arrays von Semaphoren:
- Gruppe 0: S<sub>00</sub>, S<sub>01</sub>, S<sub>02</sub>, S<sub>03</sub>, S<sub>04</sub>, S<sub>05</sub>
  - Gruppe 1: S<sub>10</sub>, S<sub>11</sub>
  - Gruppe 2: S<sub>20</sub>, S<sub>21</sub>, S<sub>22</sub>
  - Gruppe 3: S<sub>30</sub>, S<sub>31</sub>, S<sub>32</sub>, S<sub>33</sub>, S<sub>34</sub>
  - Gruppe n: leer
- Ein Pfeil zeigt auf die Spalte 2, die die Semaphoren S<sub>20</sub>, S<sub>21</sub> und S<sub>22</sub> enthält, und ist mit "einzelne Semaphore" beschriftet. Ein Pfeil zeigt auf die gesamte Zeile 3 und ist mit "Semaphorengruppe" beschriftet. Ein Pfeil zeigt auf die Spalte 5 und ist mit "Semaphorennun innerhalb der Gruppe" beschriftet. Ein Pfeil zeigt auf die Spalte 0 und ist mit "Semaphorentabelle" beschriftet.



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

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

















## IPC-Objekte kontrollieren und löschen

- Informationen über bestehende (**System V**)-Speichersegmente, (**System V**)-Nachrichtenwarteschlangen und (**System V**)-Semaphoren liefert das Kommando `ipcs`
- Die einfachste Möglichkeit, solche gemeinsamen Speichersegmente, Nachrichtenwarteschlangen und Semaphoren auf der Kommandozeile zu löschen, ist das Kommando `ipcrm`

```
ipcrm [-m shmids] [-q msgids] [-s semids]
      [-M shmkeys] [-Q msgkeys] [-S semkeys]
```

- **POSIX**-Speichersegmente und **POSIX**-Semaphoren können im Verzeichnis `/dev/shm` untersucht und von Hand gelöscht werden
- **POSIX**-Nachrichtenwarteschlangen im Verzeichnis `/dev/mqueue` untersucht und von Hand gelöscht werden