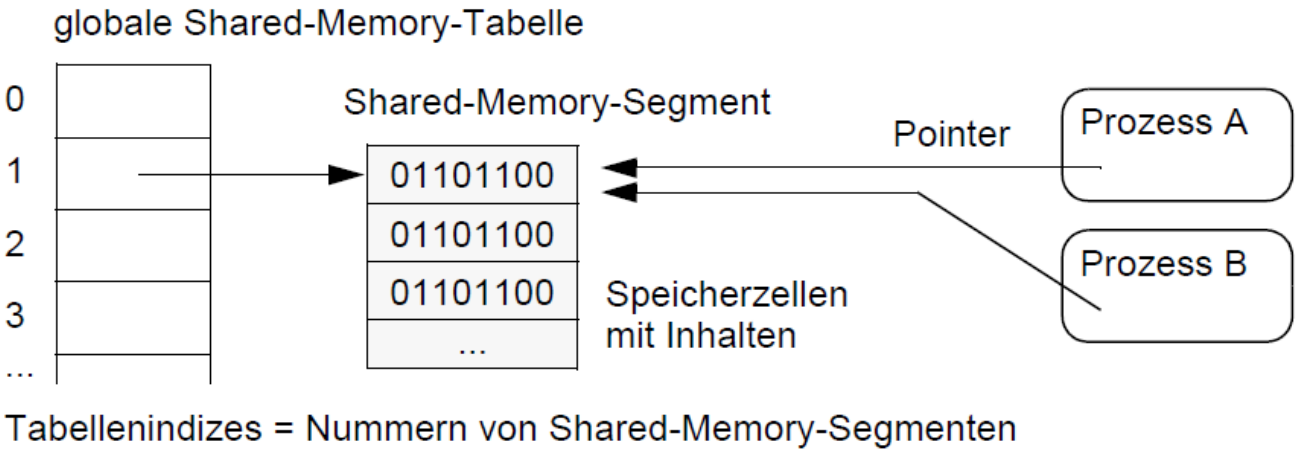
Shared Memory

- Speicherbereiche, auf die mehrere UNIX/Linux-Prozesse zugreifen können, werden Shared-Memory-Segmente genannt.
  - Diese Segmente werden wie Semaphore dynamisch erzeugt und gelöscht und mit Hilfe einer Tabelle verwaltet.
  - Prozesse können aus C-Programmen heraus über Pointer auf die Segmente zugreifen.



Quelle: [Vogt, 2012]

Shared Memory erzeugen bzw. öffnen: `shmget()`

- Mit **`shmget()`** kann ein Prozess ein neues Shared-Memory-Segment erzeugen oder sich den Tabellenindex eines bereits bestehenden Segments verschaffen:

```
int shmget(key_t key, size_t size, int shmflg)
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

Nummer des Segments
(= Index in der Shm.tabelle)
–1 bei Fehler

Schlüssel des Segments oder `IPC_PRIVATE`
(*`key_t` und `size_t` sind systemhängige Ganzzahltypen*)

Größe des Segments (in Bytes)

Flags zur Steuerung der Ausführung
(insbes. `IPC_CREAT` für Erzeugung)
sowie Zugriffsrechte auf das Segment

analog **`zu semget()`**
und Semaphorgruppen

- beispielhafter Aufruf: `int shmid = shmget(IPC_PRIVATE,5*sizeof(float),IPC_CREAT|0777);`
- liefert den Index der Shared-Memory-Tabelle zurück, über den nachfolgende Operationen auf das Segment zugreifen können. Dieser Index wird nur innerhalb von Programmen benutzt.
- Zusätzlich kann jedes Segment (wie eine Semaphorgruppe) einen ganzzahligen „Schlüssel“ besitzen (→ durch **`key`**-Parameter übergeben; IPC_PRIVATE → kein Schlüssel benannt)

Quelle: [Vogt, 2012]

Shared Memory einbinden: `shmat()`

- Mit **`shmat()`** kann ein Prozess ein Shared-Memory-Segment in seinen Adressraum einbinden.
  - Er erhält einen Pointer (nämlich die Adresse der ersten Speicherzelle des Segments) zurück, über den er anschließend auf das Segment zugreifen kann:

```
#include <sys/types.h>  #include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg)
```

- Adresse der ersten Speicherzelle des Segments, −1 bei Fehler
- Nummer des Segments (= Index in der Shared-Memory-Tabelle)
- üblicherweise 0
- Flags zur Steuerung der Ausführung (meist 0)

  - Beispiel:

```
float *f_pointer = (float *) shmat(shmid,0,0);
*f_pointer = 0.5;  *(f_pointer+1) = 1.0;
```

  - Der Rückgabewert ist **`void*`** → Daten im *Shared-Memory*-Segment sind nicht getypt → muss reinterpretiert werden → Segment kann flexibel zur Übertragung von Daten unterschiedlichen Typs benutzt werden.

Quelle: [Vogt, 2012]

Shared Memory steuern und löschen: `shmctl()`

- **`shmctl()`** ermöglicht verschiedene Steuerungsoperationen auf Shared-Memory-Segmenten:

```
#include <sys/ipc.h>  #include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

Rückgabewert abhängig von der ausgeführten Operation
–1 bei Fehler

Nummer des Segments (= Index in der Shared-Memory-Tabelle)

auszuführende Operation

Parameter für die Operation

  - Die auszuführende Operation wird durch den **`cmd`**-Parameter bestimmt (= symbol. Konstante), z. B. **`IPC_RMID`** um das Segment zu löschen → **`shmctl(shmid,IPC_RMID,0)`**.
  - Werden Shared-Memory-Segmente nicht explizit gelöscht, so bleiben sie über das Ende der Prozessausführung hinaus bestehen!

➢ Man muss also auch hier ggf. Segmente durch Benutzerkommandos entfernen:
  - Befehl **`ipcs -m`** → Nummern der existierenden Shared-Memory-Segmentes feststellen
  - Befehl **`ipcrm -m nummer`** → Segment löschen.

Quelle: [Vogt, 2012]

Shared Memory → Erzeuger-Verbraucher-Problem

- bei verwandten Prozessen

```c
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <sys/wait.h>

#define PUFFERKAP 3      /* Kapazität des Puffers */
#define ANZAHL_RUNDEN 10 /* Anzahl der Erzeugungs- bzw. Verbrauchsvorgänge */

main() {

 int shmid;      /* Nummer des Shared-Memory-Segments */
 float *shmptr;  /* Pointer auf das Segment */
 int ix;         /* Lese- bzw. Schreibindex im Segment */
 float zwisch;   /* Zwischenspeicher */
 int i;          /* Schleifenzähler */
 int status;     /* Rückgabeparameter für wait() */

 int semid;                          /* Nummer der Semaphorgruppe zur Synchronisation */
 struct sembuf sem_p[2], sem_v[2]; /* P- und V-Operationen */
 unsigned short init_array[3];     /* Anfangswerte der Semaphore */

 /* Erzeugung des Shared-Memory-Segments */
 shmid = shmget(IPC_PRIVATE,PUFFERKAP*sizeof(float),IPC_CREAT|0777);

 /* Erzeugung und Vorbereitung der Semaphore */
 semid = semget(IPC_PRIVATE,3,IPC_CREAT|0777);
 init_array[0] = 0;   /* S_BELEGT==0: zählt # belegten Plätze, blockiert bei leerem Puffer  */
 init_array[1] = PUFFERKAP; /* S_FREI==1: zählt # freien Plätze, blockiert bei vollem Puffer */
 init_array[2] = 1;    /* S_WA==2: soll Pufferzugriff wechselseitig ausschließen */
 semctl(semid,0,SETALL,init_array);
 sem_p[0].sem_op = sem_p[1].sem_op = -1;
 sem_p[0].sem_flg = sem_p[1].sem_flg = 0;
 sem_v[0].sem_op = sem_v[1].sem_op = 1;
 sem_v[0].sem_flg = sem_v[1].sem_flg = 0;
```

Semaphor-Lösung

Shared Memory → Erzeuger-Verbraucher-Problem (2)

```
44  /* Verbraucher-Prozess */
45  if (fork()==0) {
46   ix = 0;  /* Leseindex auf Segmentanfang setzen */
47   shmptr = (float *) shmat(shmid,0,0); /* Pointer auf Segment */
48   for (i=0;i<ANZAHL_RUNDEN;i++) {
49    /* Blockierung, solange Puffer leer ist oder durch anderen Prozess
50       benutzt wird */
51    sem_p[0].sem_num = 0; sem_p[1].sem_num = 2;
52    semop(semid,sem_p,2);
53
54    zwisch = *(shmptr+ix); /* Lesen aus der ix-ten Segmentpos.  */
55    ix = (ix+1)%PUFFERKAP; /* Weiterschalten des Index */
56    printf("Index: %d\n",ix);
57    printf("Gelesen: %f\n",zwisch);
58
59    sem_v[0].sem_num = 1; sem_v[1].sem_num = 2;
60    semop(semid,sem_v,2);
61   }
62   exit(0);
63  }
```

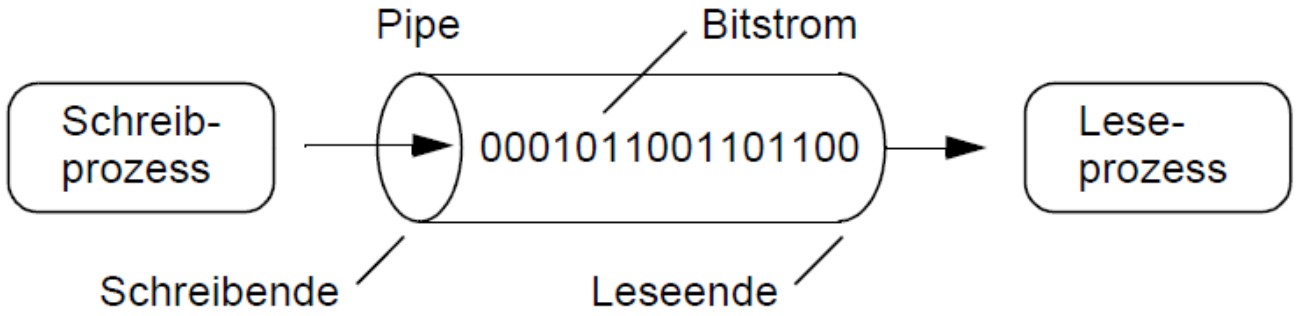
```
ifadmin@llc-off-site:~/Vogt_Beispielprogramme$ g++ prog_4_01.c
 -o prog_4_01
ifadmin@llc-off-site:~/Vogt_Beispielprogramme$ prog_4_01
Index: 1
Gelesen: 0.000000
Index: 2
Gelesen: 0.500000
Index: 0
Gelesen: 1.000000
Index: 1
Gelesen: 1.500000
Index: 2
Gelesen: 2.000000
Index: 0
Gelesen: 2.500000
Index: 1
Gelesen: 3.000000
Index: 2
Gelesen: 3.500000
Index: 0
Gelesen: 4.000000
Index: 1
Gelesen: 4.500000
ifadmin@llc-off-site:~/Vogt_Beispielprogramme$
```

```
65  /* Erzeuger-Prozess */
66  if (fork()==0) {
67   ix = 0;  /* Schreibindex auf Segmentanfang setzen */
68   shmptr = (float *) shmat(shmid,0,0); /* Pointer auf Segment */
69   for (i=0;i<ANZAHL_RUNDEN;i++) {
70    zwisch = 0.5*i;  /* zu schreibender Wert (im Prinzip beliebig) */
71    /* Blockierung, solange Puffer voll ist oder durch anderen Prozess
72       benutzt wird */
73    sem_p[0].sem_num = 1; sem_p[1].sem_num = 2;
74    semop(semid,sem_p,2);
75
76    *(shmptr+ix) = zwisch; /* Schreiben an die ix-te Segmentpos. */
77    ix = (ix+1)%PUFFERKAP; /* Zyklisches Weiterschalten des Index */
78
79    sem_v[0].sem_num = 0; sem_v[1].sem_num = 2;
80    semop(semid,sem_v,2);
81   }
82   exit(0);
83  }
```

```
85  /* Warten auf das Ende von Erzeuger und Verbraucher */
86  wait(&status);
87  wait(&status);
88
89  /* Löschung des Shared-Memory-Segments und der Semaphorgruppe */
90  shmctl(shmid,IPC_RMID,0);
91  semctl(semid,IPC_RMID,0);
92
93 }
```

Pipes

- Pipes werden zur lokalen, strombasierten Kommunikation benutzt.
  - Pipes sind von der Benutzerschnittstelle eines Betriebssystems her bekannt:
    - Man kann hier zwei Kommandos durch den Pipe-Operator | miteinander verknüpfen (zum Beispiel `ls -l | more`), wodurch die Daten der Standardausgabe des ersten Kommandos in die Standardeingabe des zweiten fließen.
  - Bildlich: "Rohrleitung", durch die ein Bitstrom von einem Schreibende zu einem Leseende fließt
  - Datenübertragung verläuft unidirektional
  - Es werden keine Nachrichten übertragen (die durch Grenzen voneinander getrennt sind) sondern Datenströme.



Quelle: [Vogt, 2012]

Benannte Pipes

- Benannte Pipes haben Namen, die im Dateisystem verzeichnet sind.
  - Über diese Namen können prinzipiell beliebige Prozesse auf sie zugreifen.
  - In der Ausgabe des Kommandos `ls -l` sind Pipes durch den Kennbuchstaben `p` als Dateityp gekennzeichnet:

```
prw-rw-rw-  1 gonzo muppets    0 2011-06-27 09:03 PIPE_1|
prw-rw-rw-  1 gonzo muppets    0 2011-06-27 09:05 PIPE_2|
-rwxr-xr-x  1 gonzo muppets 4096 2011-06-27 19:22 normale_datei
```

- Benannte Pipe mit der Funktion `mkfifo()` erzeugen:

```
#include <sys/types.h>  #include <sys/stat.h>
```

mode_t ist ein system-abhängiger Ganzzahltyp

```
int mkfifo(const char *pathname, mode_t mode)
```

0 bei Erfolg
–1 bei Fehler

Name der Pipe

Zugriffsrechte auf die Pipe

- Auf einer benannten Pipe arbeitet man mit Standard-Dateioperationen:
  `open()` → Öffnen; `write()` → Schreiben, `read()` → Lesen, `unlink()` → Löschen

Quelle: [Vogt, 2012]

Benannte Pipes → Programmbeispiel

- Sender einer Zeichenfolge:

```
 6 #include <stdio.h>
 7 #include <fcntl.h>
 8
 9 #include <unistd.h>
10 #include <sys/types.h>
11 #include <sys/stat.h>
12
13 main() {
14
15  int fd;  /* Deskriptor fuer die Pipe */
16  const char* pipe_file = "./PIPE_2"; /* Pipe mit Pfad */
17
18  printf("Erzeuge Pipe\n");
19  mkfifo(pipe_file,0666);  /* Erzeugung der
20                              und Schreibre
21
22  printf("Oeffne Pipe zum Schreiben\n");
23  fd=open(pipe_file,O_WRONLY);  /* Oeffnen der Pipe zum Schreiben */
24
25  printf("Schreibe 'HALLO'\n");
26  write(fd,"HAL",3); /* Schreiben zweier Zeichenfolgen */
27  write(fd,"LO",3);  /* in die Pipe (inkl. Stringendezeichen \0) */
28
29  printf("Programmende Sender\n");
30  unlink("PIPE_2");
31 }
```

`open()` blockiert, bis Pipe auch zum Lesen geöffnet wird!

`write()` blockiert nicht, bis zum Aufruf von `read()` → asynchron!

- Empfänger der Zeichenfolge:

```
 6 #include <fcntl.h>
 7 #include <stdio.h>
 8
 9 #include <unistd.h>
10 #include <sys/types.h>
11 #include <sys/stat.h>
12
13 main() {
14
15  char buffer[6];  /* Speicher fuer die empfangenen Daten */
16
17  int fd;  /* Deskriptor fuer die Pipe */
18  const char* pipe_file = "./PIPE_2"; /* Pipe mit Pfad */
19
20  printf("Oeffne Pipe zum Lesen\n");
21  fd=open(pipe_file,O_RDONLY);  /* Oeffnen der Pipe zum Lesen */
22  if(!fd) {
23     printf("Oeffnen der Pipe gescheitert!");
24     return 0;
25  }
26
27  printf("Lese 6 Zeichen\n");
28  int anzahl = read(fd,buffer,6);  /* Lesen von 6 Zeichen aus der Pipe */
29
30  if(anzahl>0) /* Ausgabe hier: HALLO */
31      printf("Es wurden %d Zeichen gelesen: %s\n",anzahl, buffer);
32  else
33      printf("Es sind keine Daten in der Pipe vorhanden\n");
34
35  unlink("PIPE_2");  /* Loeschen der Pipe */
36 }
```

ist die Maximalzahl

Benannte Pipes → Programmbeispiel (2)

- Sender einer Zeichenfolge:

```
os@os:~/betriebssysteme/Vogt_Beispielprogramme$ ./prog_4_02_sender
Erzeuge Pipe
Oeffne Pipe zum Schreiben
Schreibe 'HALLO'
Programmende Sender
os@os:~/betriebssysteme/Vogt_Beispielprogramme$
```

- Empfänger der Zeichenfolge:

```
os@os:~/betriebssysteme/Vogt_Beispielprogramme$ ./prog_4_02_empfaenger
Oeffne Pipe zum Lesen
Lese 6 Zeichen
Es wurden 6 Zeichen gelesen: HALLO
os@os:~/betriebssysteme/Vogt_Beispielprogramme$
```

- Anzeige der benannten Pipe im Arbeitsverzeichnis der Prozesse während der Programmausführung:

```
os@os:~/betriebssysteme/Vogt_Beispielprogramme$ ls -l PIPE*
prw-rw-r-- 1 os os 0 Apr 29 14:41 PIPE_2
os@os:~/betriebssysteme/Vogt_Beispielprogramme$
```

Unbenannte Pipes

- Unbenannte Pipes haben (im Gegensatz zu benannten) keine Namen im Dateisystem.
  - Sie werden über zwei Deskriptoren identifiziert, die dem erzeugenden Prozess übergeben werden.
  - Damit ist die Pipe nur für den erzeugenden Prozess und seine Nachkommen zugreifbar.

- Erzeugung unbenannter Pipes mit der Funktion **`pipe()`**:

```
int pipe(int pipefd[2])
```

```
#include <unistd.h>
```

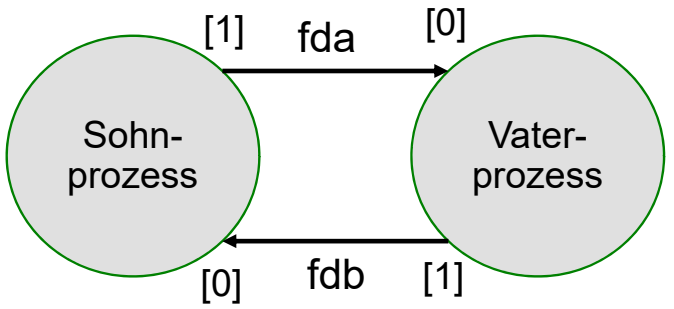
0 bei Erfolg
–1 bei Fehler

Pipe-Deskriptoren (Rückgabeparameter):
`filedes[0]`: Leseende der Pipe
`filedes[1]`: Schreibende der Pipe

Quelle: [Vogt, 2012]

Unbenannte Pipes → Programmbeispiel

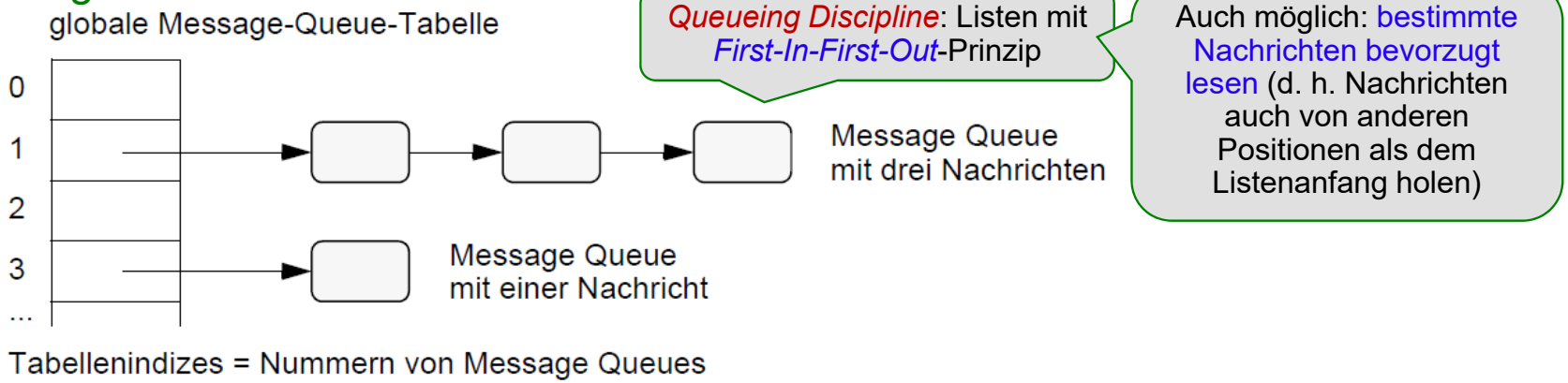
```c
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>

int main() {

 char buffer[20];     /* Puffer zum Datenempfang */
 int fda[2], fdb[2];  /* Deskriptoren fuer Leseenden (fdx[0]) */
                      /* und Schreibenden (fdx[1]) der Pipes */
 pipe(fda); /* Erzeugung zweier unbenannter Pipes; Speichern der */
 pipe(fdb); /* Deskriptoren fuer Lese-/Schreibenden in fda und fdb */

 if (fork()==0) {
  /* Sohn: sendet Stringnachricht und empfaengt Rueckantwort */
  close(fda[0]);  /* nicht benoetigte Lese- und */
  close(fdb[1]);  /* Schreibdeskriptoren schließen */
  /* 6 Bytes in Pipe A schreiben (5 Zeichen + Ende-Zeichen \0) */
  write(fda[1],"HALLO",6);
  /* Rueckantwort aus Pipe B lesen und ausgeben */
  read(fdb[0],buffer,20);
  printf("\nSohn liest %s aus der Pipe B\n\n",buffer);
  exit(0);
 }

 /* Vater: empfaengt Stringnachricht und sendet Rueckantwort */
 close(fda[1]);  /* nicht benoetigte Lese- und */
 close(fdb[0]);  /* Schreibdeskriptoren schließen */
 /* String aus Pipe A lesen und ausgeben */
 read(fda[0],buffer,20);
 printf("\nVater liest %s aus der Pipe A\n\n",buffer);
 /* Rueckantwort in Pipe B schreiben */
 write(fdb[1],"HALLO ZURUECK",14);
}
```




```
ifadmin@llc-off-site:~/Vogt_Beispielprogramme$
g++ prog_4_03.c -o prog_4_03.out
ifadmin@llc-off-site:~/Vogt_Beispielprogramme$
./prog_4_03.out

Vater liest HALLO aus der Pipe A


Sohn liest HALLO ZURUECK aus der Pipe B

ifadmin@llc-off-site:~/Vogt_Beispielprogramme$
```

Message Queues

- Message Queues dienen unter UNIX/Linux zur lokalen nachrichtenbasierten Kommunikation.
  - Unterschied zu Pipes: Es werden keine Datenströme transportieren, sondern die übertragenen Daten sind in einzelnen Nachrichten untergliedert und es wird Nachricht für Nachricht gesendet und empfangen.
  - Eine Message Queue realisiert damit eine Mailbox oder einen Port
- Message Queues werden (analog zu Semaphore und *Shared Memory*) durch Tabellen im Betriebssystem geführt:



- Die Struktur der Nachrichten ist nicht vorgegeben, sondern kann vom Programmierer frei festgelegt werden (→ typisierte Nachrichten)

Quelle: [Vogt, 2012]

Message Queues erzeugen bzw. öffnen: `msgget()`

- Wie für Semaphore und Shared-Memory-Segmente gibt es auch für Message Queues eine grundlegende Get-Funktion: **`msgget()`**:

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

int msgget(key_t key, int msgflg)
```

Nummer der Queue
(= Index in der Tabelle)
–1 bei Fehler

Schlüssel der Queue oder IPC_PRIVATE
(*key_t ist ein systemabhängiger Ganzzahltyp*)

Flags zur Steuerung der Ausführung
(insbesondere `IPC_CREAT` für Erzeugung)
sowie Zugriffsrechte auf die Queue

- Analog zu **`semget()`** und **`shmget()`** kann über den ersten **`msgget()`**-Parameter entweder die Konstante **`IPC_PRIVATE`** oder ein Queue-Schlüssel übergeben werden.
- Damit bieten sich auch hier die Möglichkeiten, die bei der Einführung von **`semget()`** diskutiert wurden.

Quelle: [Vogt, 2012]

Message Queues steuern und löschen: `msgctl()`

- **`msgctl()`** ist eine Steuerungsfunktion für Message Queues:

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

int msgctl(int msqid, int cmd, struct msqid_ds *buf)
```

wird analog zu **`semctl()`** und **`shmctl()`** eingesetzt

Rückgabewert abhängig von der ausgeführten Operation
-1 bei Fehler

Nummer der Queue
(= Index in der Queue-Tabelle)

auszuführende Operation

Parameter für die Operation

- Der wichtigste `cmd`-Parameter ist wiederum **`IPC_RMID`**, mit dem eine Message Queue gelöscht wird: **`msgctl(msgid,IPC_RMID,0)`**
- Werden Message Queues nicht explizit gelöscht, so bleiben sie über das Ende der Prozess-ausführung hinaus bestehen

➢ ggf. durch Benutzerkommandos entfernen:

- **`ipcs -q`** → Nummern der existierenden Message Queues ermitteln
- **`ipcrm -q nummer`** → Message Queue mit dieser Nummer löschen.

Quelle: [Vogt, 2012]

Message Queues → Struktur einer Nachricht

- Nachrichten, die über Message Queues übertragen werden, sind aus Sicht des BS (im Prinzip) beliebige Bitmuster.
- Aus der Sicht von Anwendungen sind sie C-Strukturen, mit (fast) beliebigen Aufbau:
  - einzige Einschränkung: erste Komponente einer Nachricht muss vom Typ `long` sein!

Typ

```
struct bestellung {
 long mtype;
 char warenname[20];
 int kennziffer;
 float preis;
} meineBestellung;
```

- Diese erste Komponente ist der „Typ“ der Nachricht:
  - wird beim Lesen der Nachricht ausgewertet
  - erlaubt gezielt Nachrichten eines bestimmten Typs zu empfangen → um z. B. wichtige Nachrichten bevorzugt aus der Queue zu lesen → Priorisierung möglich
  - Die Bedeutung der Typnummern legt allein der Programmierer fest.
  - Der Typ muss > 0 sein, auch dann, wenn das Programm den Nachrichtentyp nicht weiter berücksichtigt.

Message Queues → Senden und Empfangen

- Funktionen **`msgsnd()`** u. **`msgrcv()`**:

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

```
int msgsnd(
  int msqid,
  const void *msgp,
  size_t msgsz,
  int msgflg)
```

- Nummer der Queue (= Index der Queue-Tabelle)
- zu versendende Nachricht
- Größe der Nachricht (in Bytes, ohne die erste `long`-Komponente)
- Flags zur Steuerung der Ausführung (insbes. `IPC_NOWAIT` für nichtblockierendes Senden)

0 bei Erfolg
–1 bei Fehler

```
ssize_t msgrcv(
  int msqid,
  void *msgp,
  size_t msgsz,
  long msgtyp,
  int msgflg)
```

- Nummer der Queue (= Index der Queue-Tabelle)
- empfangene Nachricht (Rückgabeparameter)
- maximal zulässige Größe der empfangenen Nachricht (in Bytes, ohne die erste `long`-Komponente)
- gewünschter Typ der empfangenen Nachricht (0, wenn Nachrichten aller Typen akzeptiert werden)
- Flags zur Steuerung der Ausführung (insbes. `IPC_NOWAIT` für nichtblockierendes Empfangen)

Größe der empfangenen Nachricht (in Bytes, ohne die erste `long`-Komponente) oder –1 bei Fehler (*`ssize_t` ist ein systemabhängiger Ganzzahltyp*)

- Beispiel:

```
struct bestellung {
 long mtype;
 char warenname[20];
 int kennziffer;
 float preis;
} meineBestellung;
```

- senden:

```
meineBestellung.mtype = 1;
strcpy(meineBestellung.warenname,"USB-Stick");
meineBestellung.kennziffer = 1234;
meineBestellung.preis = 9.95;
msgsnd(msqid,&meineBestellung,
     sizeof(meineBestellung)-sizeof(long),0);
```

- empfangen:

```
struct bestellung deineBestellung;
msgrcv(msqid,&deineBestellung,
   sizeof(deineBestellung)-sizeof(long),0,0);
```

Quelle: [Vogt, 2012]

Message Queues → Verhalten beim Senden

- Bei der Ausführung von `msgsnd()` wird die Nachricht in einen Speicherbereich des Betriebssystemkerns kopiert.
  - Die übergebene Variable wird also wieder frei zur Zusammenstellung der nächsten Nachricht.
  - Eine einmal abgeschickte Nachricht kann über sie nicht nachträglich verändert werden.
- Das Betriebssystem fügt die Nachricht an das Ende der Queue an und entblockiert alle Prozesse, die auf eine Nachricht dieses Typs warten (siehe unten).
- Es kommt dann zu einem Wettrennen: Ein Prozess gewinnt das Rennen und erhält die Nachricht, die anderen blockieren sich wieder.
- Die Länge einer Nachricht sollte stets mit dem Ausdruck `sizeof(variablenname)-sizeof(long)` berechnet werden.
- Eine Message Queue kann nur eine bestimmte Maximalzahl von Bytes aufnehmen.
  - Maximalzahl von Bytes ändern mit `msgctl()`
  - Flag IPC_NOWAIT bestimmt das Verhalten, wenn neue Nachricht die Grenze überschreitet:
    - nicht gesetzt → sendender Prozess wird blockiert, bis wieder genug Platz frei ist.
    - gesetzt (also `msgsnd(...,IPC_NOWAIT)` → Sendeaufruf kehrt sofort mit Rückgabe `-1` zurück

Message Queues → Verhalten beim Empfangen

- **`msgrcv()`** empfängt und entfernt die erste Nachricht aus der Queue, deren Typeintrag mit dem gewünschten zu empfangenden Typ übereinstimmt:

```
struct bestellung deineBestellung;

msgrcv(msqid,&deineBestellung,
   sizeof(deineBestellung)-sizeof(long),0,0);
```

msgsz **msgtyp** **msgflg**

Parameter zur Steuerung der Übertragung (s. nächste Seite)

  - Die **`struct`**-Variablen, die bei **`msgsnd()`** und **`msgrcv()`** übergeben werden, müssen in ihrem Aufbau übereinstimmen.
  - UNIX/Linux überträgt nur den reinen Byte-Inhalt einer Nachricht und kann daher nicht prüfen, ob der Empfänger die Bytes korrekt interpretiert!

Quelle: [Vogt, 2012]

Message Queues → Verhalten beim Empfangen (2)

- Der 3. Parameter **`msgsz`** verhindert einen Speicherüberlauf, da das BS nie mehr Bytes als hier angegeben überträgt:
  - **`MSG_NOERROR`**-Flag gesetzt → Inhalt der Nachricht wird abgeschnitten
  - **`MSG_NOERROR`**-Flag nicht gesetzt → Nachricht verbleibt in der Queue und **`msgrcv()`** kehrt mit dem Rückgabewert **`-1`** zurück
- Der 4. Parameter **`msgtyp`** gibt den Typ an, den die empfangene Nachricht haben soll.
  - **`msgtyp == 0`** → jede Nachricht wird akzeptiert
  - **`msgtyp > 0`** → erste Nachricht in der Queue wird empfangen, die diesen Typeintrag hat.
    → Nachrichten lassen sich mit Prioritäten versehen
- Der 5. Parameter **`msgflg`** legt unter anderem fest, was geschehen soll, wenn keine passende Nachricht vorliegt.
  - **`IPC_NOWAIT`**-Flag gesetzt → Aufruf kehrt sofort mit dem Rückgabewert **`-1`** zurück. → aktives Warten (Polling) möglich
  - **`IPC_NOWAIT`**-Flag nicht gesetzt → Prozess wird blockiert und wartet passiv.

Messages Queues → Erzeuger-Verbraucher-Problem

- Erzeuger:

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

#include <unistd.h>
#include <sys/wait.h>

int main() {

 int msqid, verbr_id, i, status;

 struct bestellung {  /* Typ fuer Bestellungen */
  long mtype;
  char warenname[20];
  int kennziffer;
  float preis;
 };

 /* Vater: Erzeugt die Message Queue */
 msqid = msgget(IPC_PRIVATE,IPC_CREAT|0777);

 /* Erzeuger: Schickt in Sekundenabstaenden 5 Bestellungen ab */
 if (fork()==0) {
  struct bestellung meineBestellung;
  for (i=0;i<5;i++) {
   meineBestellung.mtype = 1;
   strcpy(meineBestellung.warenname,"USB-Stick");
   meineBestellung.kennziffer = 4711;
   meineBestellung.preis = 9.95;
   msgsnd(msqid,&meineBestellung,
            sizeof(meineBestellung)-sizeof(long),0);
   sleep(1); /* Verzoegerung bis zum naechsten Senden */
  }
  exit(0);
 }
```

- Verbraucher:

```c
 /* Verbraucher: Nimmt beliebig viele Bestellungen entgegen */
 if ((verbr_id=fork())==0) {
  struct bestellung deineBestellung;
  while(1) {
   msgrcv(msqid,&deineBestellung,
                    sizeof(deineBestellung)-sizeof(long),0,0);
   printf("Gelesen: %s %d %f\n",
                   deineBestellung.warenname,
                   deineBestellung.kennziffer,
                   deineBestellung.preis);
  }
 }

 /* Vater: Raeumt auf */
 wait(&status);              /* Warten auf Ende des Erzeugers */
 kill(verbr_id,SIGKILL);     /* Terminieren des Verbrauchers */
 msgctl(msqid,IPC_RMID,0);   /* Loeschen der Message Queue */

}
```

```
ifadmin@llc-off-site:~/Vogt_Beispielprogramme$
g++ prog_4_04.c -o prog_4_04.out
ifadmin@llc-off-site:~/Vogt_Beispielprogramme$
./prog_4_04.out
Gelesen: USB-Stick 4711 9.950000
Gelesen: USB-Stick 4711 9.950000
Gelesen: USB-Stick 4711 9.950000
Gelesen: USB-Stick 4711 9.950000
Gelesen: USB-Stick 4711 9.950000
ifadmin@llc-off-site:~/Vogt_Beispielprogramme$
```

vgl. Lös. mit Shared Memory

Messages Queues → Prioritätengesteuertes Senden

- Das vorhergehende Beispiel kann so erweitert werden, dass Nachrichten prioritäten-gesteuert gesendet und empfangen werden.
- Beispiel:
  - Sender versendet dringende Nachrichten mit dem Typwert 2 und normale Nachrichten mit dem Typwert 1.
  - zweistufiger Empfangsvorgang:

```
if (msgrcv(msqid,&deineBestellung,...,2,IPC_NOWAIT)==-1)
 msgrcv(msqid,&deineBestellung,...,1,0);
```

Typ 2-Nachrichten nicht-blockierend empfangen

alle Nachrichten blockierend empfangen

Achtung:

Hier darf nicht der Typ 1 stehen, da …

1.) zwischen den beiden **`msgrcv()`**-Aufrufen eine Typ 2-Nachricht eintreffen könnte, die dann nicht empfangen würde.

2.) Typ 2-Nachrichten, die später eintreffen, den blockierten Prozess nicht wieder wecken würde!

Quelle: [Vogt, 2012]

Shared Memory mit der Boost-Bibliothek

- *Shared Memory* mit Namen **`sm_name`** und der Struktur
  `{ int, double[3], float }`
  erzeugen, den Adressbereich mappen, schreibend zugreifen und löschen:

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
using namespace boost::interprocess;

shared_memory_object sm(create_only, "sm_name", read_write);  // erzeugen
sm.truncate(sizeof(int)+3*sizeof(double)+sizeof(float));  // Größe festlegen
mapped_region region(sm, read_write);   // mappen zum Lesen und Schreiben

int* pi = NULL; double* pad = NULL; float* pf = NULL;  // Zeiger für Zugriff
pi = static_cast<int*>(region.get_address());
pad = reinterpret_cast<double*>(pi+sizeof(int));
pf = reinterpret_cast<float*>(pad+3*sizeof(double));

*pi=2; pad[0]=1.1; pad[1]=2.2; pad[2]=3.3; *pf=7.1f;  // schreiben
shared_memory_object::remove("sm_name");   // löschen
```

`SharedMemory`

Shared Memory mit der Boost-Bibliothek (2)

- *Shared Memory* mit Namen **`sm_name`** und der Struktur
  `{ int, double[3], float }`
  öffnen, den Adressbereich mappen und lesend darauf zugreifen:

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
using namespace boost::interprocess;

shared_memory_object sm(open_only, "sm_name", read_write); // öffnen
sm.truncate(sizeof(int)+3*sizeof(double)+sizeof(float));   // Größe festlegen
mapped_region region(sm, read_only);   // mappen zum Lesen

int* pi = NULL; double* pad = NULL; float* pf = NULL; // Zeiger für Zugriff
pi = static_cast<int*>(region.get_address());
pad = reinterpret_cast<double*>(pi+sizeof(int));
pf = reinterpret_cast<float*>(pad+3*sizeof(double));

cout << *pi << " " << pad[0] << " " << [1] << " " << [2] << " " << *pf << endl;       // lesen
```

`SharedMemory`

Managed Shared Memory mit der Boost-Bibliothek

- *Managed Shared Memory* mit Namen **msm_name** erzeugen, Variablen mit Namen in den Adressbereich mappen, schreibend zugreifen und löschen:

```
#include <boost/interprocess/managed_shared_memory.hpp>
using namespace boost::interprocess;

/* managed shared memory mit 1024 byte Länge erzeugen */
managed_shared_memory msm(create_only, "msm_name", 1024);

/* Daten in den Adressbereich mappen */
int* pi = NULL; float* pf = NULL; int* ai = NULL; // Zeiger für Zugriff
pi = msm.construct<int>("i_name")(7);  // int-Variable mit Namen i_name
                                       // erzeugen und mit 7 initialisieren
pf = msm.construct<float>("f_name")(0.11);    // analog für float-Variable
ai = msm.construct<int>("ia_name")[10](99);  // für int-Array mit 10 Elem.

*pi=2; *pf=1.1; ai[0]=2.2; ai[9]=3.3; // schreibender Zugriff

shared_memory_object::remove("msm_name");   // Löschen des shared mem.-Bereiches
```

SharedMemoryManaged

Managed Shared Memory mit der Boost-Bibliothek (2)

- *Managed Shared Memory* mit Namen **msm_name** öffnen, Variablen mit Namen in den Adressbereich mappen und lesend darauf zugreifen:

```
#include <boost/interprocess/managed_shared_memory.hpp>
using namespace boost::interprocess;

managed_shared_memory sm(open_only, "msm_name", 1024); // öffnen

/* Daten in den Adressbereich mappen */
int* pi = NULL; float* pf = NULL; int* ai = NULL; // Zeiger für Zugriff
pi = managed_shm.find<int>("i_name").first;
if(pi == NULL) { /* Fehlerbehandlung */ }
pf = managed_shm.find<float>("f_name").first;
pair<int*, size_t> p = managed_shm.find<int>("ia_name");
if(p.first == NULL || p.second != 10) { /* Fehlerbehandlung */ }
else ai = p.first;

cout << *pi << " " << *pf << " "
```

p.first → 1. Element und **p.second → Array-Größe**

SharedMemoryManaged

Managed Shared Memory mit der Boost-Bibliothek (3)

- Kommunikation über *Managed Shared Memory* mit Synchronisation mittels Semaphore mit Namen **`sem_name`**:

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/named_semaphore.hpp>
using namespace boost::interprocess;

managed_shared_memory msm(create_only, "msm_name", 1024); // öffnen
boost::interprocess::named_semaphore sem(create_only, "sem_name", 1);

/* Daten in den Adressbereich mappen */
int* pi = NULL; float* pf = NULL; int* ai = NULL; // Zeiger für Zugriff
pi = msm.construct<int>("i_name")(7);

sem.wait()
cout << *pi << endl; /* Zugriff */
sem.post();
```

`SharedMemorySync`

Die Klasse `message_queue` der Boost-Bibliothek

- Eine Message Queue **`mq_name`** für **`int`**-Werte mit einer Länge von 100 erzeugen, zwei **`int`**-Werte senden und Queue wieder entfernen:

```
#include <boost/interprocess/ipc/message_queue.hpp>
using namespace boost::interprocess;

message_queue mq(create_only, "mq_name", 100, sizeof(int));
/* other options: open_only, open_or_create, open_read_only */

unsigned int priority = 0; /* all messages with same priority */
int my_int1=6, my_int2=33;

mq.send(&my_int1, sizeof(my_int1), priority);
mq.send(&my_int2, sizeof(my_int2), priority);
/* If the message queue is full the sender is blocked. */

message_queue::remove(mq);
```

MessageQueue

Die Klasse `message_queue` der Boost-Bibliothek (2)

- Eine Message Queue `mq_name` öffnen und zwei `int`-Werte empfangen:

```
#include <boost/interprocess/ipc/message_queue.hpp>
using namespace boost::interprocess;
unsigned int priority = 0;
message_queue::size_type recvd_size;
int my_int1=0, my_int2=0;

message_queue mq(open_only, "mq_name");
/* other options: open_only, open_or_create, open_read_only */

mq.receive(&my_int1, sizeof(my_int1), recvd_size, priority);
if(recvd_size != sizeof(my_int1)) { /* error handling */ }
/* if message queue is empty the receiver is blocked */

mq.receive(&my_int2, sizeof(my_int2), recvd_size, priority);
if(recvd_size != sizeof(my_int2)) { /* error handling */ }
```

`MessageQueue` (siehe auch **`MessageQueue_mixed`**)