UNIX-Schnittstelle

4. Systemrufe für die lokale Prozesskommunikation

- **System V spezifische lokale Prozesskommunikation:**

  Messages
  Semaphore
  Shared Memory

- **POSIX konforme lokale Prozesskommunikation**

  Messages
  Semaphore
  Shared Memory

- **Lokale Remote Procedure Call - Doors (Solaris)**

4.1 System V IPC

4.1.0 Einführung

System V spezifische Mechanismen:

- **Messages:** Austausch von Nachricheten zwischen beliebigen Prozessen bzw. zwischen mehreren Clienten und einem Server.
  Realisierung: Messagequeue im Kern
- **Semaphore:** Synchronisation der Arbeit von verschiedenen Prozessen.
  Realisierung: Feld von Semaphoren im Kern
- **Shared Memory:** Erlaubt mehreren Prozessen die gemeinsame Nutzung eines Speichersegmentes.
  Realisierung: Shared Memory Segment

Ressourcen: Messagequeue, Array of Semaphore, Shared Memory Segment

Ressourcenidentifikation:

- extern: Schlüssel (key) der allen beteiligten Prozessen bekannt sein muss. Der Schlüssel ist im System eindeutig. key ist eine long int. Mittels der Bibliotheksfunktion "ftok()" kann man aus einem Pfadname oder String einen Schlüssel machen.
- intern: Numerischer ID, den man beim "Eröffnen" einer Ressource erhält.

Die Anzahl der jeweiligen Ressourcen im System werden im Kernel durch Konstanten festgelegt (msglimits, semlimits, shmlimits).

Zugriffsrechte zu Ressourcen: ähnlich wie bei Files

Headerfile <sys/ipc.h>:

```
        /* Common IPC Access Structure */

        struct ipc_perm {
                ushort  uid;    /* owner's user id */
                ushort  gid;    /* owner's group id */
                ushort  cuid;   /* creator's user id */
                ushort  cgid;   /* creator's group id */
                ushort  mode;   /* access modes */
                ushort  seq;    /* slot usage sequence number */
                key_t   key;    /* key */
        };


        /* Common IPC Definitions. */

        /* Mode bits. */
        #define IPC_ALLOC   0100000  /* entry currently allocated */
        #define IPC_CREAT   0001000  /* create entry  */
        #define IPC_EXCL    0002000  /* fail if key exists */
        #define IPC_NOWAIT  0004000  /* error if request must wait */

        /* Keys. */
        #define IPC_PRIVATE     (key_t)0   /* private key */

        /* Control Commands. */
        #define IPC_RMID        0       /* remove identifier */
        #define IPC_SET         1       /* set options */
        #define IPC_STAT        2       /* get options */
```

```
Prinzipieller Aufbau der System V IPC-Systemrufe
-----------------------------------------------

Der Systemruf  get :
      int XXXget(key_t key, int XXXflg);

    Holen einer Ressource (open).

        key:       Schlüssel, IPC_PRIVATE immer neue Ressource
        XXXflags:  IPC_CREATE - neue Ressource erzeugen, wenn
                                diese noch nicht existiert
                   IPC_CREATE|IPC_EXCL - Fehler, wenn Ressource
                                         bereits existiert.
                   Zugriffsrechte ebenfalls in XXXflags kodiert.
        Rückgabe:
                   >=0 - Identifier für weitere Systemrufe
                    <0 - Fehler

Der Systemruf  ctl() :
       int XXXctl(int XXXid, int cmd, struct XXXid_ds *buf);

    Ausführen einer Steueroperation für eine Ressource.

       XXXid:  RessourcenID ( von XXXget) für die Ressource, für
               die eine Steueroperation ausgeführt werden soll.
       cmd:    Auszuführende Steueroperation
               IPC_STAT - Lesen der Statusinformationen der Ressource
               IPC_SET  - Setzen der Statusinformationen der Ressource
               IPC_RMID - Streichen der Ressource (IPC-Ressourcen
                          bleiben über Prozessende hinaus erhalten)
```



```
Die Systemrufe für die eigentlichen Operationen sind je nach
Ressource individuell.

          msgsnd, msgrcv - Messages

          shmat, shmdt   - Shared Memory

          semop          - Semaphore

Gemeinsam ist nur das Flag  IPC_NOWAIT , das jeweils eine Wartefunktion
 unterdrückt.
```

```
Ein paar Kommandos
------------------
   ipcs  - Auflisten von Informationen über das IPC

           ipcs   [-a|-q|-m|-s] [-c|-l|-p|-t|-u]
            -q - MessageQueue
            -m - Shared Memory
            -s - Semaphore
            -a - all
            -c - Erzeuger und Eigentuemer
            -l - Limits
            -p - PID des Erzeugers und des letzten Zugriffs
            -t - Zeit des letzten Zugriffs
            -u - Zusammenfassung

   ipcrm - Streichen einer IPC-Einrichtung

      ipcrm { -s semid  | -m msgid  | -q msqid    |
              -S semkey | -M msgkey | -Q msqkey }

          xxxid  - interner Identifier
          xxxkey - externer Key (bei xxxget angegeben)
                   Achtung!! wird bei ipcs hexadezimal angezeigt!!
       Beispiele:

          ipcrm  -Q 0x50  # extern hexadezimal
          ipcrm  -Q 80    # extern dezimal
          ipcrm  -q  32769  # intern dezimal
```



```
           4.1.1. Messages
           ===============

  /*
   *  User message buffer template for msgsnd and msgrcv system calls.
   */
  struct msgbuf {
    long  mtype;     /* message type */
    char  mtext[1];  /* message text */
  };

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);

msgget()  gibt einen  msqid  für die Messagequeue  key  zurück.
msgflg  spezifiziert die Zugriffsrechte (niederwertige 9 Bit)
und die Art der "Eröffnung" (IPC_PRIVATE, IPC_CREAT, IPC_EXCL,
eröffnen einer bestehenden Messagequeue). Ist der Rückkehrwert
< 0, ist ein Fehler aufgetreten.

Fehler:

      EACCES   - keine Zugriffsrechte
      EEXIST   - IPC_CREATE|IPC_EXCL für eine existierende MQ
      EIDRM    - MQ ist removed
      ENOSPC   - alle ID's vergeben
      ENOENT   - MQ existiert nicht und kein IPC_CREAT
      ENOMEM   - kein Memory
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);
```

Senden einer Message an die Messagequeue msqid (von msgget()).
msgp zeigt auf einen Puffer mit der Struktur msgbuf.

msgsz gibt die Länge von mtext in msgbuf an. Ist in msgflg IPC_NOWAIT gesetzt, so setzt der aktuelle Prozess die Arbeit fort. Andernfalls wartet er, bis ein Empfängerprozess die Message abholt bzw. bis die Messagequeue durch msgctl() aus dem System entfernt wird oder ein Signal eintrifft.

Rückkehrwerte von msgsnd():

```
   >=0  -  Es wird  msgsz  zurückgegeben, wenn  msgsnd()
           erfolgreich war

   <0   -  Fehler (-1)
              EACCES   - Keine Zugriffsrechte
              EAGAIN   - IPC_NOWAIT wenn Queue ist voll
              EFAULT   - msgp  falsche Adresse
              EIDRM    - MQ ist removed
              EINTR    - Volle MQ, Prozess wartete aber Signal kam
              EINVAL   - mtype <1,  msgsz >MSGMAX,
                         msgsz <0  oder  msqid <0
```



```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv( int msqid, struct msgbuf *msgp,
            int msgsz, long msgtyp, int msgflg);
```

msgrcv() liest eine Message aus der MQ, die durch msqid (von msgget()) spezifiziert wurde. Die Message wird in den Puffer *msgp übertragen.

Die maximale Länge von mtext in msgbuf wird durch msgsz angegeben.

msgtyp beschreibt den Typ der Message, die geholt werden soll.

```
  msgtyp == ANY  - jede Message wird geholt
  msgtyp  >  0   - 1.Message mit  mtype == msgtyp  wird geholt
  msgtyp  <  0   - 1.Message mit  mtype  <= abs( msgtyp ) wird geholt
```

msflg spezifiziert ob auf eine Message gewartet wird (IPC_NOWAIT) bzw. ob längere Messages als in msgsz spezifiziert akzeptiert werden (MSG_NOERROR).

Rückkehrwerte von msgrcv():

```
     >=0  - Anzahl der Zeichen in  mtext  für die geholte Message
     <0   - Fehler (-1)
            E2BIG - Message zu lang und nicht MSG_NOERROR
            EACCES, EFAULT, EIDRM,
            EINTR, EINVAL, ENOMSG   - siehe msgsnd()
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);

msgctl()  liefert eine Vielzahl von Steuerfunktionen für Messages.
msqid  spezifiziert die Messagequeue (von  msgget()).
cmd  spezifiziert die gewünschte Steuerfunktion:
     IPC_RMID  -  Entfernen der Messagequeue aus dem System
     IPC_STAT  -  Lesen der Struktur msqid_ds in den Puffer *buf
     IPC_SET   -  Setzen einiger Werte aus der Struktur msqid_ds
                  im Kern
struct msqid_ds {
  struct ipc_perm  msg_perm;  /* operation permission struct */
  struct msg  *msg_first;     /* ptr to first message on q */
  struct msg  *msg_last;      /* ptr to last message on q */
  ushort msg_cbytes;          /* current # bytes on q */
  ushort msg_qnum;            /* # of messages on q */
  ushort msg_qbytes;          /* max # of bytes on q */
  ushort msg_lspid;           /* pid of last msgsnd */
  ushort msg_lrpid;           /* pid of last msgrcv */
  time_t msg_stime;           /* last msgsnd time */
  time_t msg_rtime;           /* last msgrcv time */
  time_t msg_ctime;           /* last change time */
};
Rückkehrwerte von  msgctl():
   0   - ok
   <0  - Fehler (-1)
         EACCES, EFAULT, EIDRM, EINVAL  -  siehe  msgsnd()
```



```
  Beispiel 1:

          Header: mipc1.h
          Server: msv1.c
                  msv1a.c  mit IPC_EXCL
          Client: mcl1.c

  Beispiel 2: mit Threads

          Header: mipc1.h
          Server: msv1b.c
          Client: mcl1b.c

  Beispiel 3: Zentraler Dienst

          Server: msv2.c
                  msv3.c  mit ordentlichem Ende
          Client: mcl2.c
```

4.1.2. Shared Memory

Headerfiles:

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

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

shmget() gibt einen **shmid** für das Shared Memory Segment **key** zurück. **size** spezifiziert die minimale Grösse des Segments (SHMMIN <= size <= SHMMAX).
shmflg spezifiziert die Zugriffsrechte (niederwertige 9 Bit) und die Art der "Eröffnung" (IPC_PRIVATE, IPC_CREAT, IPC_EXCL - eröffnen eines bestehenden Shared Memory Segmentes).
Ist der Rückkehrwert < 0, ist ein Fehler aufgetreten.

Fehler (shmget):

```
EINVAL   - size  nicht zulässig
EACCES   - keine Zugriffsrechte
EEXIST   - IPC_CREATE|IPC_EXCL für ein existierendes SMS
EIDRM    - SMS ist removed
ENOSPC   - alle ID's vergeben
ENOENT   - SMS existiert nicht und kein IPC_CREAT
ENOMEM   - kein Memory
```



```
char *virt_addr;
virt_addr shmat(int shmid, char *shmaddr, int shmflg);
```

shmat() blendet das durch **shmid** spezifizierte Shared Memory Segment in den Adressbereich des Datensegmentes des aktuellen Prozesses ein. **shmaddr** spezifiziert die Adresse und **shmflg** die Art der Einbindung und die Zugriffsrechte.

```
   shmaddr == NULL  - erste freie Mappingadresse wird als
                      Startadresse des Shared Memory Segments
                      benutzt.
   shmaddr != NULL und SHM_RND-Flag nicht gesetzt -
                      Shared Memory Segment wird an der
                      spezifizierten Adresse eingeblendet.
                      Adresse muss "page aligned" sein!!!
   shmaddr != NULL und SHM_RND-Flag gesetzt -
                      das Shared Memory Segment wird an der
                      Adresse (shmaddr - ( shmaddr modulo SHMLBA))
                      eingeblendet.
shmflg:  SHM_RDONLY   -  nur Leserechte für das Segment
         SHM_RND      -  Mappingadresse abrunden auf Seitengrenze
```

Rückkehrwert (shmat):

```
   != NULL   - Adresse des Shared Memory Segmentes
   == NULL   - Fehler
                EACCESS  -  keine Zugriffsrechte
                EINVAL   -  shmid  unbenutzt,
                            shmaddr  nicht auf Seitengrenzen
                EIDRM    -  Shared Memory Segment wird gestrichen
                ENOMEM   -  kein Speicher kann für das Segment
                            bereitgestellt werden
```

```
int shmdt(char *shmaddr);
```

shmdt() entfernt das an der Adresse shmaddr im Datensegment liegende Shared Memory Segment aus dem Adressbereich des aktuellen Prozesses.

Rückkehrwert (shmdt):

```
   0  - ok, Shared Memory Segment ausgeblendet
  <0  - Fehler
         EINVAL - kein Shared Memory Segment an der angegebenen
                  Adresse vorhanden
```



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

shmctl() liefert eine Vielzahl von Steuerfunktionen für Shared Memory Segmente. shmid spezifiziert das SMS (von shmget()).

cmd spezifiziert die gewünschte Steuerfunktion:

```
     IPC_RMID  -  Entfernen der SMS aus dem System
     IPC_STAT  -  Lesen der Struktur shmid_ds in den Puffer *buf
     IPC_SET   -  Setzen einiger Werte aus der Struktur shmid_ds
                  im Kern
     SHM_LOCK  -  Shared Memory Segment im Speicher fixieren
                  (nur für SU)
     SHM_UNLOCK - Fixierung des Shared Memory Segments im Speicher
                  aufheben (nur für SU)
buf  -  Adresse des Puffers bei IPC_STAT und IPC_SET
struct shmid_ds {
        struct ipc_perm shm_perm;       /* operation permission struct */
        int             shm_segsz;      /* size of segment in bytes */
        struct region   *shm_reg;       /* ptr to region structure */
        char            pad[4];         /* for swap compatibility */
        pid_t           shm_lpid;       /* pid of last shmop */
        pid_t           shm_cpid;       /* pid of creator */
        unsigned short  shm_nattch;     /* used only for shminfo */
        unsigned short  shm_cnattch;    /* used only for shminfo */
        time_t          shm_atime;      /* last shmat time */
        time_t          shm_dtime;      /* last shmdt time */
        time_t          shm_ctime;      /* last change time */
};
```

```
Rückkehrwerte (shmctl):

   0   - ok
   <0  - Fehler (-1)
           EACCES - keine Zugriffsrechte
           EFAULT - Puffer nicht adressierbar
           EIDRM  - Shared Memory Segment gestrichen
           EINVAL - shmid < 0  oder unbenutzt
           EPERM  - kein Erzeuger, Eigentümer oder Superuser

Grenzwerte:
   SHMMNI  -  maximale Zahl von Shared Memory Segments im System
   SHMMAX  -  Maximale Grösse eines Segmentes
   SHMMIN  -  minimale Grösse eines Segmentes
   SHMALL  -  maximaler Speicher für SMS im System
   SHMLBA  -  "page aligned"
   SHMSEG  -  maximale Zahl der SMS pro Prozess
```



```
Beispiel 1:  busy wait

     Header:  mipc.h
     Server:  svm.c
     client:  clm.c

Beispiel 2: mit Signal 1. Version

     Header: mipc2.h
     server:  svm_sig.c
     client:  clm_sig.c

Beispiel 3: mit Signal 2. Version mit Threads

     Header: mipc2.h
     server:  svm_sig1.c
     client:  clm_sig1.c
```

4.1.3. Semaphore

```
Headerfiles:   #include <sys/types.h>
               #include <sys/ipc.h>
               #include <sys/sem.h>
```

Semaphore sind int-Variable, für die geschützte Operationen (+,-,test) existieren. Diese Operationen sind im Kern realisiert. Damit können sich mehrere Prozesse synchronisieren. Im UNIX werden nicht einzelne Semaphore benutzt, sondern Felder von Semaphoren. Für jede einzelne Semaphore existiert im Kern folgende Struktur:

```
struct sem {
        short sempid;      /* pid of last semop() */
        ushort semval;     /* current value */
        ushort semncnt;    /* # procs awaiting
                              increase in semval */
        ushort semzcnt;    /* # procs awaiting semval = 0 */
}
```



```
int semget(key_t key, int nsems, int semflg);
```

semget() gibt die zu key gehörende Semaphorekennung semid zurück. nsems gibt die Zahl der Semaphore an (0 <nsems <= SEMMSL). semflg spezifiziert die Zugriffsrechte (niederwertige 9 Bit) und die Art der "Eröffnung" (IPC_PRIVATE, IPC_CREAT, IPC_EXCL, eröffnen eines bestehenden Semaphorefeldes).

Ist der Rückkehrwert < 0, ist ein Fehler aufgetreten.

Fehler (semget()):

```
EINVAL   - size  nicht zulässig
EACCES   - keine Zugriffsrechte
EEXIST   - IPC_CREATE|IPC_EXCL für ein existierendes SEM
EIDRM    - SEM ist removed
ENOSPC   - alle ID's vergeben
ENOENT   - SEM existiert nicht und kein IPC_CREAT
ENOMEM   - kein Memory
```

```
int semop(int semid, struct sembuf *sops, unsignes nsops);
```

semop() wird zur Ausführung einer Reihe von Semaphore-operationen in dem zur Semaphorekennung semid gehörende Semaphorefeld benutzt. sops ist ein Zeiger auf ein Feld von Strukturen sembuf, die jeweils eine Semaphoreoperation beschreiben.

```
struct sembuf {
        unsigned short  sem_num;        /* semaphore # */
        short           sem_op;         /* semaphore operation */
        short           sem_flg;        /* operation flags */
};
```

nsops gibt die Zahl der Strukturen in dem Feld an.

```
Es gibt folgende drei Semaphoreoperationen:
sem_op  < 0 : Ist der Wert des Semaphores mit dem Index sem_num
              grösser oder gleich dem Absolutbetrag von sem_op,
              so wird der Absolutbetrag vom Semaphore subtrahiert
              und der Prozess setzt seine Arbeit fort.
              Anderfalls wird der Prozess angehalten, wenn in
              sem_flg  das Flag IPC_NOWAIT nicht gesetzt ist.
              Wird ein angehaltener Prozess wieder aktiviert, so
              wird die (das Stoppen verursachende) Semaphoreoperation
              wiederholt.
sem_op  > 0 : sem_op  wird zum Semaphore mit dem Index sem_num
              addiert und der Prozess setzt seine Arbeit fort.
              Sollte ein Prozess zuvor gestoppt worden sein, so
              wird der Prozess wieder gestartet.
```



```
sem_op == 0 : Wenn der Semaphore den Wert  0  hat, setzt der Prozess
              seine Arbeit fort. Andernfalls wird er gestoppt, wenn
              in sem_flg das Flag IPC_NOWAIT nicht gesetzt ist.

In  sem_flg  können  folgende Flags gesetzt werden:

IPC_NOWAIT  -   nicht warten, wenn Wartebedingung eintritt
SEM_UNDO    -   addiere  semval (aktuellen Wert des Semaphores)
                zu  un_aue  für den Semaphore, eventuell
                sem_undo Struktur anlegen.

     struct sem_undo {
        struct sem_undo *un_np; /* ptr to next active undo structure */
        short           un_cnt; /* # of active entries */
        struct undo {
                short   un_aue; /* adjust on exit values */
                short   un_num; /* semaphore # */
                int     un_id;  /* semid */
        }       un_ent[1];      /* undo entries (one minimum) */
     }
                Die sem_undo  Struktur wird bei exit() zum Zurück-
                setzen der Semaphore benutzt.
```

Rückkehrwert:

```
    0 - Alle Semaphoreoperationen erfolgreich abgeschlossen
   -1 - Semaphoreoperation nicht erfolgreich abgeschlossen.
      Fehler (semop()):
        E2BIG  - nsops > SEMOPM
        EACCES - keine Zugriffsrechte
        EAGAIN - IPC_NOWAIT spezifiziert und Operation nicht aus-
                 führbar
        EFAULT - sops  nicht adressierbar
        EFBIG  - sem_num  >= nsems
        EIDRM  - Semaphore momentan gestrichen
        EINTR  - Signal eingetroffen
        EINVAL - semid unzulässig
        ENOMEM - SEM_UNDO gefordert und kein Speicher im Kern frei.
        ERANGE - sem_op + semval  > SEMVMAX
```



```
int semctl(int semid, int semnum,int cmd, union semun arg);
```

semctl() stellt eine Vielzahl von Steuerfunktionen für Semaphore zur Verfügung. Diese werden durch cmd angegeben und für den Semaphore, der durch semid und semnum spezifiziert ist ausgeführt. arg enthält zusätzliche Informationen für die Steuerfunktionen, er ist wie folgt aufgebaut:

```
         union semun {
              int val;       /* value for SETVAL */
              struct semid_ds *buf; /* buffer for IPC_STAT and
                                       IPC_SET   */
              ushort *array;    /*array for GETALL and SETALL */
         }

struct semid_ds {
   struct ipc_perm  sem_perm;   /* operation permission struct */
   struct sem      *sem_base; /* ptr to first semaphore in set */
   unsigned short  sem_nsems; /* # of semaphores in set */
   time_t          sem_otime; /* last semop time */
   time_t          sem_ctime; /* last change time */
};
```

```
Folgende Kommandos sind für  cmd  zulässig:

    GETNCNT     - get semncnt  -> Rückkehrwert
    GETPID      - get sempid -> Rückkehrwert
    GETVAL      - get semval -> Rückkehrwert
    GETALL      - get all semval's -> arg.array
    GETZCNT     - get semzcnt -> Rückkehrwert
    SETVAL      - set semval  <- arg.val
    SETALL      - set all semval's <- arg.array
    IPC_RMID    - Streichen des Semaphore
    IPC_STAT    - get  semid_ds -> arg.buf
    IPC_SET     - set  semid_ds <- arg.buf (uid, gid, mode)

Rückkehrwerte (semctl()):
   >=0  ok, eventuell Wert der geforderten Variablen
   < 0  Fehler
          EACCESS - kein Zugriff
          EIDRM   - Semaphore werden momentan getrichen
          EINVAL  - unzulässiger int-Parameter
          EPERM   - IPC_RMID, IPC_SET nicht zulässig
          ERANGE  - unzulässiger Wert für semval
```



```
Beispiel 1:  Synchronisation mit Semaphors und Shared Memory

      Header:  mipc.h
      Server:  svsm.c
      Client:  clsm.c    - einzelne Semaphore
               clsm1.c   - Feld von Semaphoren, mehrere Operationen
               clsm2.c   - Feld von Semaphoren, eine Operation !!!
```

4.2 POSIX IPC

4.2.0 Einführung

```
 POSIX  spezifische Mechanismen:

    Messages:      Austausch von Nachrichten zwischen beliebigen
                   Prozessen bzw. zwischen mehreren Clienten und
                   einem Server.
                   Realisierung: Messagequeue im Kern oder System V IPC
    Semaphore:     Synchronisation der Arbeit von verschiedenen
                   Prozessen.
                   Realisierung: Kern oder System V IPC
    Shared Memory: Erlaubt mehreren Prozessen die gemeinsame
                   Nutzung eines Speichersegmentes.
                   Realisierung: Kern oder System V IPC

Ressourcen: Messagequeue, Semaphore, Shared Memory Segment
Ressourcenidentifikation:
    Filename mit "/" beginnend.
Die Anzahl der jeweiligen Ressourcen im System werden im Kernel
durch Konstanten festgelegt (MQ_OPEN_MAX, SHM_OPEN_MAX, SEM_VALUE_MAX).
Zugriffsrechte zu Ressourcen: ähnlich wie bei Files
```



```
Gemeinsamkeiten von POSIX-Messagequeues, POSIX-Shraed-Memory und
POSIX-Semaphore ähnlich wie bei IPC.

xxx={mq,shm,sem}

xxx_t *xxx_open(const char *name, int oflag);

    Eröffnen einer bestehenden POSIX-Kommunikationseinrichtung

xxx_t *xxx_open(const char *name, int oflag,
                       mode_t mode, ...);

    Eröffnen einer neuen POSIX-Kommunikationseinrichtung

int xxx_unlink(const char *name);

    Löschen einer POSIX-Kommunikationseinrichtung

name  - Externer Name der Kommunikationseinrichtung
        Die Zeichenkette muß mit einem "/" beginnen.

oflag - Flags für Zugriffsrechte: O_RDONLY, O_WRONLY, O_RDWR,
        O_NONBLOCK, O_CREAT, O_EXCL

mode  - Zugriffsrechte nach dem Schließen.
```

4.2.1 POSIX Messages

Überblick

POSIX Message Queues erlauben Prozessen den Datenaustausch in Form von Messages. Die API ist von der API der Messagequeues von System V abgeleitet, aber mit einfacherer Funktionalität.

POSIX Message Queues werden mittels mq_open() erzeugt bzw. eröffnet. mq_open() liefert einen Queue-Descriptor vom Type mqd_t zurück, der von allen weiteren Calls zur Identifikation der PMQ benutzt wird. Jede PMQ wird mittels eines Namens der Form "/<name>" identifiziert. Wenn zwei Prozesse eine PMQ mit dem gleichen Namen eröffnen, benutzen sie die gleiche PMQ.

Zum Senden von Messages wird mq_send() benutzt.
Zum Empfangen von Messages wird mq_receive() benutzt.
Mittels mq_close() kann ein Prozess die Verbingung zu einer PMQ beenden, sie wird dadurch nicht gelöscht.
Zum Löschen einer PMQ wird mq_unlink() benutzt.
Ohne mq_unlink sind PMQs kernelresident, d.h. sie bleiben bis zum nächsten Reboot bestehen.

Die Eigenschaften von PMQs können mittels mq_getattr() abgefragt werden und mittels mq_setattr() gesetzt werden. mq_notify() dient zur asynchronen Information über das Eintreffen einer Message in einer PMQ.

Message Queue Descriptoren werden bei fork() vererbt.

Jede Message hat eine Priorität. Die Messages werden entsprechend ihrer Priorität abgeholt (0 - niedrigste Priorität, _SC_MQ_PRIO_MAX-1 - höchste Priorität).

Programme mit PMQs müssen immer mit -lrt gelinkt sein!!!

```
     gcc   ...  -lrt
```

```
#include <mqueue.h>

mqd_t mq_getattr(mqd_t mqdes, struct mq_attr *attr);
mqd_t mq_setattr(mqd_t mqdes, struct mq_attr *newattr,
                 struct mq_attr *oldattr);
```

Mittels mq_getattr() werden die Attribute einer bestehenden eröffneten Message-Queue ausgelesen (*attr). Mittels mq_setattr() können bei einer bestehenden eröffneten Message-Queue Attribute gesetzt bzw. modifiziert, dabei enthält *newattr die zu setzenden Attribute und *oldattr die vorher gültigen Attribute.

Die Datenstruktur mq_attr hat folgende Struktur

```
 struct mq_attr {
   long mq_flags;    /* Flags */
   long mq_maxmsg;   /* Max. Zahl der Messages in der Queue */
   long mq_msgsize;  /* Max. Kapazität in Bytes */
   long mq_curmsgs;  /* aktuelle Zahl der Messages in der Queue */
 };
```

flags enthält den bei mq_open angegeben Wert. Es kann aber nur der Wert für O_NONBLOCK verändert werden.
mq_maxmsg und mq_msgsize können nur bei mq_open() gesetzt werden. Sie müssen größer als Null sein.

```
Rückkehrcode:  0 - ok
              -1 - Fehler
```



```
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
                     struct mq_attr *attr);
```

mq_open() eröffnet eine bereits bestehende PMQ bzw. erzeugt eine neue PMQ. Beim Erzeugen einer neuen PMQ sollten die Parameter mode und attr angegeben werden.

```
name  - Name der PMQ
oflag - Basisflags
           O_RDONLY O_WRONLY O_RDWR
        Zusatzflags
           O_NONBLOCK  O_CREAT  O_EXCL
mode - Zugriffsrechte nach dem Erzeugen
attr - Attribute siehe mq_getattr()
       Nur hier können die Attribute  mq_maxmsg (max. Anzahl der
       Messages) und mq_msgsize (maximale Kapazität) gesetzt werden.

Rückkehrcode:  0 - ok
   >=0  - PMQ-Identifier
   <0   - Fehler: EACCESS, EEXIST, EINVAL, EMFILE,
                  ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENSPC
```

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

mq_close beendet den Zugriff eines Prozesses auf die POSIX-Messagequeue mqdes . Die Messagequeue muß vorher für den Prozess eröffnet gewesen sein. Die Messagequeue bleibt erhalten. Das Beenden eines Prozesses (exit()) beinhaltet ein mq_close().

Rückkehrkode:
- 0 - ok
- -1 - EBADF (falscher Descriptor)

```
#include <mqueue.h>

mqd_t  mq_unlink(const char *name);
```

Mittels mq_unlink wird die duch *name spezifizierte POSIX-Messagequeue gelöscht. *name muß der gleiche Filename sein, der bei mq_open angegeben wurde.

Rückkehrkode:
- 0 - ok
- -1 - Fehler: EACCESS, ENAMETOOLONG, ENOENT


```
#define _XOPEN_SOURCE 600
#include <time.h>
#include <mqueue.h>
#include <mqueue.h>

mqd_t mq_send(mqd_t mqdes, const char *msg_ptr,
                size_t msg_len, unsigned msg_prio);
mqd_t mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
                unsigned msg_prio, const struct timespec *abs_timeout);
```

mq_send() fügt die durch msg_ptr addressierte Messages in die durch mqdes angegebenen PMQ ein. msg_len gibt die Länge der Message an. Die Länge muß kleiner oder gleich dem Attribut mq_msgsize sein. Die Länge 0 ist erlaubt. msg_prio ist eine nicht negativer Zahl, die die Priorität angibt (0 niedrigste Priorität). Bei gleicher Priorität, wird die Messages hinter den anderen Messages gleicher Priorität eingereiht.

Wenn die PMQ voll ist (Attribut mq_maxmsg), blockiert mq_send() bis genügend Platz in der PMQ isti (standard). Durch Angabe von O_NONBLOCK kann dies verhindert werden (Fehler EAGAIN).

mq_timedsend() funktioniert wie mq_send(), außer die PMQ ist voll und O_NONBLOCK ist nicht gesetzt. abs_timout enthält dann den Zeitpunkt, bis wann der Call warten soll (Sekunden ab 1.1.1970).

```
    struct timespec {
             time_t tv_sec;        /* seconds */
             long   tv_nsec;       /* nanoseconds */
    };
```

Wenn die PMQ voll ist und der Zeitpunkt bereits vergangen ist, erfolgt eine sofortige Rückkehr.

```
Rückkehrkode:
      0 - ok
     -1 - Fehler: EAGAIN ( bei O_NONBLOCK ), EBADF, EINTR,
              EINVAL, EMSGSIZE, ETIMEDOUT (Timeout)
```



```
#define _XOPEN_SOURCE 600
#include <time.h>
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                          size_t msg_len, unsigned *msg_prio);
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned *msg_prio, const struct timespec *abs_timeout);
```

mq_receive holt die älteste Message mit der höchsten Priorität aus der PMQ und speichert sie in den mittels msg_ptr angegebenen Puffer. msg_len gibt die Länge des Puffers an (größer als mq_msgsize). Wenn msg_prio nicht NULL ist, wird dort die Priorität der Nachricht abgelegt. Wenn die PMQ leer ist, blockiert mq_receive() bis eine neue Message kommt, ausser wenn O_NONBLOCK gesetzt ist, dann wird mit Fehler EAGAIN zurückgekehrt.

mq_timedreceive() funktioniert wie mq_receive() außer die PMQ ist leer und O_NONLBLOCK ist nicht gesetzt. Dann wartet mq_timedreceive die angegeben Zeit auf das Eintreffen einer Nachricht. Ist die Zeit abgelaufen, wird mit einem Fehlerkode zurückgekehrt, siehe mq_timedsend().

```
              struct timespec {
                  time_t tv_sec;        /* seconds */
                  long   tv_nsec;       /* nanoseconds */
              };
```

```
Rückkehrkode:
    >= 0  - Länge der Message im Puffer
    <0    - Fehler: EBADF  EINTR EINVAL EMSGSIZE ETIMEDOUT
```

```
Beispiele:

  myipc.h       -  Headerfiles, Hilfsfunktionen

  mqsysconf.c   -  mqsysconf

  mqcreate.c    -  mqcreate [ -e ] [ -m maxmsg -z msgsize ]  /test1

  mqgetattr.c   -  mqgetattr  /test1

  mqsend.c      -  mqsend <name> <#bytes>   /test1

  mqreceive.c   -  mqreceive [ -n ]   /test1

  mqunlink.c    -  mqunlink   /test1
```



```
#include <mqueue.h>

mqd_t mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

mq_notify() ermöglicht dem Prozess einen Benachrichtigungsmechanismus für das Eintreffen von Messages in einer leeren PMQ zu aktivieren bzw. zu deaktivieren. mqdes spezifiziert die Messagequeue. notification verweist auf folgende Datenstruktur:

```
         struct sigevent {
             int    sigev_notify;      /* Benachrichtigungsmethode */
             int    sigev_signo;       /* Signalnummer */
             union sigval sigev_value; /* Data passed with notification */
             void (*sigev_notify_function) (union sigval);
                                       /* Function für Thread Ben. */
             void  *sigev_notify_attributes;
                                       /* Thread Function Attribute */
         };

         union sigval {            /* Data passed with notification */
             int     sival_int;    /* Integer value */
             void   *sival_ptr;    /* Pointer value */
         };
```

Wenn notification ungleich NULL ist, dann aktiviert mq_notify() den Benachrichtigungsmechanismus für den Prozess.
sigev_notify enthält die Benachrichtigungsmethode
- **SIGEV_NONE - Prozess ist registriert, aber beim Eintreffen der Message wird nichts gemacht.**

```
   SIGEV_SIGNAL - Es wird das Signal  (sigev_signo) gesendet.
         In einer mittels sigaction() registrierten Signal-Routine
         werden bei gesetztem SA_SIGINFO Flag folgende Informationen
         in der siginfo_t Struktur übergeben:
                 si_code  =  SIMESGQ
                 si_signo =  Signalnummer
                 si_value =  notificatio->sigev_value
                 si_pid   =  PID (Message-Sender)
                 si_uid   =  effektiver UID
         sigwaitinfo() liefert die selben Informationen.

   SIGEV_THREAD
         Ein neuer Thread wird gestartet.
         notification->sigev_thread_function  ist die Startfunktion.
         notification->sigev_value ist der Startwert.
         notification->sigev_notify_attributes  ist das Attribut
```

**Nur ein Prozess kann pro PMQ registriert werden.
Hat notification den Wert NULL und der Prozess ist registriert, wird die Registrierung gelöscht.
Eine Benachrichtigung erfolgt nur, wenn die PMQ vorher leer war. Wenn mq_notify() auf eine nicht leere PMQ angewendet wird, erfolgt die Benachrichtigung erst, wenn die PMQ leer war und eine neue Messages eingetroffen ist.
Wenn ein anderer Prozess/Thrad auf die PMQ mittels mq_receive() wartet, erfolgt keine Benachrichtigung. Die Registrierung bleibt aber aktiv.
Die Benachrichtigung erfolgt einmalig. Wenn eine Benachrichtigung erfolgt ist, wird der Benachrichtigungsmechanismus deaktiviert. mq_notify() muß erneut aufgerufen werden.**

Rückkehrwert:

```
    0   -  ok
   -1   -  Fehler: EBADF, EBUSY, EINVAL, ENOMEM
```

Beispiel:

```
     mqnotifysig1.c  - signal

     mqnotifysig2.c  - signal, signal-Maske

     mqnotifysig4.c  - sigwait
```

4.2.2 POSIX Semaphore

POSIX Semaphore (PSEM) ermöglichen die Synchronisation von Prozessen und Threads. Ein PSEM ist eine Interger-Variable, die niemals kleiner als Null werden kann. Es gibt zwei Operationen über POSIX Semaphore:

1. Erhöhen des Semaphore-Werteres (sem_post())
2. Verringern des Semaphore-Wertes (sem_wait()), nur möglich, wenn der Wert nicht kleiner als Null wird.

Es gibt zwei Arten von POSIX Semaphoren:

- Named Semaphore: Der Identifier wird über einen Namen zugeordnet (sem_op()). Die beteiligten Prozesse benutzen den gleichen Namen (/name). Löschen mittels sem_close() und sem_unlink().

- Unnamed Semaphore: Der Semaphore muß in einem gemeinsam benutzbaren Speicherbereich platziert sein (Datensegment bei Threads, Shared Memory Segment bei Prozessen). Er wird mittels sem_init() initialisiert und mittels sem_destroy() gelöscht.

Linking:

```
gcc -lrt  ...
```

Namensraum:
manchmal unter /dev/shm

Standard:
POSIX.1-2001.


```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

sem_open() eröffnet eine bereits bestehende Named PSEM bzw. erzeugt eine neue Named PSEM und initialisiert diese. Beim Erzeugen einer neuen PSEM müssen die Parameter mode und value angegeben werden.

```
name  - Name der PSEM
oflag - Basisflags
           O_RDONLY O_WRONLY O_RDWR
        Zusatzflags
           O_NONBLOCK  O_CREAT  O_EXCL
mode  - Zugriffsrechte nach dem Erzeugen
value - Anfangswert für den Semaphor
```

Rückkehrwert:
  Adresse des Semaphores (PSEM-Identifier)
  SEM_FAILED - Fehler: EACCESS, EEXIST, EINVAL, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

sem_close() schließt einen Named PSEM. Der Prozess gibt den PSEM frei.

Rückkehrwert
    0 - ok
   -1 - Fehler: EINVAL

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Löschen des angegebenen Named PSEM. Der Semaphore wird gelöscht, wenn alle beteiligten Prozesse den Semaphore geschlossen haben (sem_close()), bzw. die Prozesse beendet wurden.

Rückkehrwert:
   0 - ok
  -1 - Fehler: EACCES, ENAMETOOLONG, ENOENT


```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

sem_init() initialisiert einen Unnamed PSEM an der angegebenen Adresse *sem . Der Initalwert ist value. pshared spezifiziert die Partner:
0 - Threads eines Prozesses, ungleich 0 - verschiedene Prozesse, der PSEM liegt in einem Shared Memory Segment. Das Ergebnis einer Initialisierung eines bereits initialisierten PSEM ist undefiniert.

Rückkehrwert
      0 - ok
     -1 - Fehler: EINVAL, ENOSYS

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

sem_destroy() löscht einen Unnamed PSEM, der durch *sem angegeben wurde. Nur durch sem_init() initialisierte PSEMs sollten durch sem_destroy() gelöscht werden. Wartet eine anderer Prozess oder Thread mittels sem_wait() auf einen gelöschten PSEM, ist die Reaktion darauf undefiniert. Vor einer weiteren Benutzung des PSEM ist dieser mit sem_init() wieder zu initialisieren.

Rückkehrwert:
    0 - ok
   -1 - Fehler: EINVAL

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

sem_post() erhöht den Wert des PSEM sem um Eins. Wenn ein Prozess darauf wartet, wird dieser aktiviert (unlock).

Rückkehrwert:
```
    0 - ok
   -1 - Fehler: EINVAL
```



```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
     _POSIX_C_SOURCE >= 200112L || _XOPEN_SOURCE >= 600
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

sem_wait() verringert den Wert des mittels sem spezifizierten PSEMs (lock) wenn der Wert zuvor größer als Null war und kehrt unmittelbar zurück. Wenn der Wert gleich Null war, blockiert (lock) der Systemruf, bis der Wert erhöht wird oder ein Signal eintrifft.

sem_trywait() arbeitet analog, blockiert aber nicht, sondern liefert den Rückkehrwert -1 und setzt die Fehlerbedingung EAGAIN.

sem_timedwait() arbeitet ebenfalls wie sem_wait(). Der Wartezustand wird aber zusätzlich durch eine Zeitinformation begrenzt (abs_timeout). Die Zeitinformation wird bezüglich 1.1.1970 0:00 Uhr angegeben.

```
     struct timespec {
            time_t tv_sec;       /* Seconds */
            long   tv_nsec;      /* Nanoseconds [0 .. 999999999] */
    };
```

Nach Ablauf der Zeit kehrt sem_timedwait() mit dem Fehlerkode ETIMEDOUT zurück. Liegt die Zeitangabe in der Vergangenheit, wird nicht gewartet und ebenfalls der Fehlerkode ETIMEDOUT angezeigt (Rückkehrwert -1). Wenn der Wert des PSEM größer als Null ist, wird nicht gewartet und die Zeitangabe nicht geprüft.

```
Rückkehrwert:
    0 - ok
   -1 - Fehler: alle: EINTR EINVAL
            sem_trywait(): EAGAIN
            sem_timedwait(): EINVAL, ETIMEDOUT

----------------------------------------------------------------------

#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);

sem_getvalue() speichert den aktuellen Wert des PSEM nach  sval.
Wenn mehrere Prozesse oder Threads auf den Semaphore warten, wird eine
negative Zahl zurückgegeben, deren Betrag die Anzahl der wartenden
Prozesse/Threads angibt.

Rückkehrwert:
    0 - ok
   -1 - Fehler: EINVAL
```



```
   Beispiele:

      semsysconf.c  - Konstanten lesen

      semcreate.c   - Erzeugen eines Semaphores

      semgetvalue.c - Wert auslesen

      sempost.c     - Erhöhen

      semwait.c     - Warten
```

```
                  4.2.3 POSIX Shared Memory (PSM)
                  ===============================

#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>           /* For O_* constants */
gcc -lrt

int shm_open(const char *name, int oflag, mode_t mode);

shm_open() erzeugt und eröffnet ein neues bzw. ein existierendes
POSIX Shared Memory Objekt.
   name  - Name der PSEM
   oflag - Basisflags
              O_RDONLY O_WRONLY O_RDWR
           Zusatzflags
              O_NONBLOCK  O_CREAT  O_EXCL O_TRUNC FD_CLOEXEC
   mode  - Zugriffsrechte nach dem Erzeugen
Ein neues PSM hat die Länge 0. Mittels  ftruncate() kann die Länge
explizite gesetzt werden. PSM werden mit Nullen initialisiert.

FD_CLOEXEC wird für das PSM automatisch gesetzt.

Rückkehrwert:
     >= 0  - Filedescriptor für PSM
      < 0  - Fehler: EACCES, EEXIST, EINVAL, EMFILE,
             ENAMETOOLONG, ENFILE, ENOENT

POSIX.1-2001 konform.
```



```
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>           /* For O_* constants */
gcc -lrt

int shm_unlink(const char *name);

shm_unlink() löscht ein PSM. Das PSM wird im System gelöscht, wenn
der letzte Prozesse, der dieses Segment benutzt hat, dieses freigegeben
hat (munmap()). Nach einem shm_unlink() ist ein neues shm_open()
mit O_CREAT notwendig, wenn das Segment erneut benutzt werden soll.

Rückkehrwert: VALUE
     0  -  ok
    -1  -  Fehler: EACCES, ENAMETOOLONG, ENOENT
```

```
#include <unistd.h>
#include <sys/mman.h>

void  *mmap(void *start, size_t length, int prot , int flags, int fd,
            off_t offset);
```

Die Funktion mmap projiziert length Bytes einer Datei oder eines PSM fd ab Offset offset in einen Speicherbereich, vorzugsweise ab der Adresse start. Die Adresse start ist nur ein Wunsch und wird normalerweise nicht angegeben, indem 0 eingetragen wird. Der tatsächliche Platz, an den das Objekt projiziert wurde, wird von mmap zurückgegeben. Der Parameter prot beschreibt den gewünschte Speicherschutz. Er besteht aus folgenden Bits:

- PROT_EXEC - Die Seiten können ausgeführt werden.
- PROT_READ - Die Seiten dürfen gelesen werden.
- PROT_WRITE - Die Seiten dürfen beschrieben werden.

Der Parameter flags gibt den Typ des zu projizierenden Objekts und Projektionsoptionen an, sowie ob Veränderungen an der Kopie des projizierten Objekts für den Prozess privat sind oder mit anderen Referenzen gemeinsam genutzt werden. Er besteht aus folgenden Bits:

- MAP_FIXED Verwende angegebene Adresse. Wenn die angegebene Adresse nicht benutzt werden kann, schlägt mmap() fehl. Wenn MAP_FIXED angegeben ist, muss start ein Vielfaches der Seitengröße sein.
- MAP_SHARED Die Seiten dürfen mit anderen Prozessen, die dieses Objekt ebenfalls in den Speicher projizieren, gemeinsam benutzt werden.
- MAP_PRIVATE Lege eine private Copy-on-Write-Projektion des Objekts an.

Rückkehrwert

- ok - Adresse des projizierten Speicherbereiches
- Fehler - MAP_FAILED (-1): EBADF, EACCES, EINVAL, ETXTBUSY, EAGAIN, ENOMEM

Konform zu POSIX.4.

```
#include <unistd.h>
#include <sys/mman.h>

int munmap(void *start, size_t length);
```

Der munmap-Systemaufruf löscht die Projektionen im angegebenen Speicherbereich. Zukünftige Zugriffe auf diesen Adressraum erzeugen einen Fehler vom Typ "invalid memory reference" - Ungültiger Speicherzugriff.

Rückkehrwert
  0 - ok
 -1 - Fehler: EINVAL

Konform zu POSIX.4.

Beispiele:

```
   shmcreate.c  - Erzeugen Shared Memory Segment

   shmwrite.c   - Schreiben

   shmread.c    - Lesen

   shmunlink.c  - Löschen

   test1.c      - Zugriffstest
```


4.3 Lokale Remote Prozedure Call - Door (Solaris)

```
#include <door.h>
gcc [ flag... ] file... -ldoor [ library... ]

int door_create(void (*server_procedure) (void *cookie, char *argp,
                size_t  arg_size,  door_desc_t  *dp, uint_t n_desc),
                void *cookie, uint_t attributes);
```

The door_create() function creates a door descriptor that describes the procedure specified by the function server_procedure. The data item, cookie, is associated with the door descriptor, and is passed as an argument to the invoked function server_procedure during door_call(3DOOR) invocations. Other arguments passed to server_procedure from an associated door_call() are placed on the stack and include argp and dp. The argp argument points to arg_size bytes of data and the dp argument points to n_desc door_desc_t-structures. The attributes argument specifies attributes associated with the newly created door. Valid values for attributes are constructed by OR-ing one or more of the following values:

DOOR_UNREF
: Delivers a special invocation on the door when the numberof descriptors that refer to this door drops to one. In order to trigger this condition, more than one descriptor must have referred to this door at some time. DOOR_UNREF_DATA designates an unreferenced invocation, as the argp argument passed to server_procedure. In the case of an unreferenced invocation, the values for arg_size , dp and n_did are 0. Only one unreferenced invocation is delivered on behalf of a door.

```
  DOOR_UNREF_MULTI  Similar to DOOR_UNREF, except multiple unreferenced
                    invocations can be delivered on the same door if the
                    number  of  descriptors referring to the door  drops
                    to one more than once.  Since an additional reference
                    may have been passed by the time an unreferenced
                    invocation arrives, the DOOR_IS_UNREF attribute
                    returned by the door_info(3DOOR) call can be used
                    to determine  if  the  door  is still unreferenced.
  DOOR_PRIVATE      Maintains a separate pool of  server threads  on
                    behalf of the door.  Server threads are associated
                    with a door's  private server pool using
                    door_bind(3DOOR).
  DOOR_REFUSE_DESC  Any attempt to door_call(3DOOR) this door with
                    argument descriptors will fail with ENOTSUP.  When
                    this flag is set, the door's server procedure will
                    always  be  invoked  with  an n_desc argument of 0.
The descriptor returned from door_create() will be marked as close on exec
(FD_CLOEXEC).  Information  about a door is available for all clients of
a door using  door_info(3DOOR).  Applications concerned with security
should not place secure information in door data that is accessible by
door_info().  In  particular, secure data should not be stored in the data
item cookie.

By default, additional threads are created as needed to handle concurrent
door_call(3DOOR) invocations. See door_server_create(3DOOR) for information
on how  to  change this behavior.

A process can advertise a door in the file system name space using
fattach(3C).
```



```
RETURN VALUES
Upon successful completion,  door_create()  returns  a  nonnegative value.
Otherwise, door_create returns  -1 and sets errno to indicate the error.

ERRORS
The door_create() function will fail if:
  EINVAL      Invalid attributes are passed.
  EMFILE      The process has too many open descriptors.
```

```
#include <door.h>
gcc [ flag... ] file... -ldoor [ library... ]

int door_call(int d, door_arg_t *params);A

  typedef struct {
     char        *data_ptr;     /* Argument/result buf ptr*/
     size_t       data_size;    /* Argument/result buf size */
     door_desc_t *desc_ptr;     /* Argument/result descriptors */
     uint_t       desc_num;     /* Argument/result num desc */
     char        *rbuf;         /* Result buffer */
     size_t       rsize;        /* Result buffer size */
  } door_arg_t;
```

The door_call() function invokes the function associated with the door descriptor d, and passes the arguments (if any) specified in params. All of the params members are treated as in/out parameters during a door invocation and may be updated upon returning from a door call. Passing NULL for params indicates there are no arguments to be passed and no results expected.

Arguments are specified using the data_ptr and desc_ptr members of params. The size of the argument data in bytes is passed in data_size and the number of argument descriptors is passed in desc_num.

Results from the door invocation are placed in the buffer, rbuf. See door_return(3DOOR). The data_ptr and desc_ptr members of params are updated to reflect the location of the results within the rbuf buffer. The size of the data results and number of descriptors returned are updated in the data_size and desc_num members. It is acceptable to use the same buffer for input argument data and results, so door_call() may be called with data_ptr and desc_ptr pointing to the buffer rbuf.

If the results of a door invocation exceed the size of the buffer specified by rsize, the system automatically allocates a new buffer in the caller's address space and updates the rbuf and rsize members to reflect this location. In this case, the caller is responsible for reclaiming this area using munmap(rbuf, rsize) when the buffer is no longer required. See munmap(2).

Descriptors passed in a door_desc_t structure are identified by the d_attributes member. The client marks the d_attributes member with the type of object being passed by logically OR-ing the value of object type. Currently, the only object type that can be passed or returned is a file descriptor, denoted by the DOOR_DESCRIPTOR attribute. Additionally, the DOOR_RELEASE attribute can be set, causing the descriptor to be closed in the caller's address space after it is passed to the target. The descriptor will be closed even if door_call() returns an error, unless that error is EFAULT or EBADF.

The door_desc_t structure includes the following members:

```
  typedef struct {
    door_attr_t d_attributes;   /* Describes the parameter */
    union {
        struct {
                int d_descriptor;   /* Descriptor */
                door_id_t d_id;    /* Unique door id */
        } d_desc;
    } d_data;
  } door_desc_t;
```

When file descriptors are passed or returned, a new descriptor is created in the target address space and the d_descriptor member in the target argument is updated to reflect the new descriptor. In addition, the system passes a system-wide unique number associated with each door in the door_id member and marks the d_attributes member with other attributes associated with a door including the following:

- **DOOR_LOCAL** The door received was created by this process using door_create(). See door_create(3DOOR).
- **DOOR_PRIVATE** The door received has a private pool of server threads associated with the door.
- **DOOR_UNREF** The door received is expecting an unreferenced notification.

- **DOOR_UNREF_MULTI** Similar to DOOR_UNREF, except multiple unreferenced notifications may be delivered for the same door.
- **DOOR_REFUSE_DESC** This door does not accept argument descriptors.
- **DOOR_REVOKED** The door received has been revoked by the server.

The door_call() function is not a restartable system call. It returns EINTR if a signal was caught and handled by this thread. If the door invocation is not idempotent the caller should mask any signals that may be generated during a door_call() operation. If the client aborts in the middle of a door_call(), the server thread is notified using the POSIX (see standards(5)) thread cancellation mechanism. See cancellation(5).

The descriptor returned from door_create() is marked as close on exec (FD_CLOEXEC). Information about a door is available for all clients of a door using door_info(). Applications concerned with security should not place secure information in door data that is accessible by door_info(). In particular, secure data should not be stored in the data item cookie. See door_info(3DOOR).

```
RETURN VALUES

Upon successful completion, 0 is returned. Otherwise, -1  is
returned and errno is set to indicate the error.

ERRORS

The  door_call() function will fail if:
  E2BIG      Arguments were too  big  for  server  thread
             stack.
  EAGAIN     Server was out of available resources.
  EBADF      Invalid door descriptor was passed.
  EFAULT     Argument pointers pointed outside the  allo-
            cated address space.
  EINTR     A signal  was  caught  in  the  client,  the
            client  called fork(2), or the server exited
            during invocation.
  EINVAL    Bad arguments were passed.
  EMFILE    The client  or  server  has  too  many  open
            descriptors.
  ENOTSUP   The desc_num argument is  non-zero  and  the
            door has the DOOR_REFUSE_DESC flag set.
  EOVERFLOW System could not  create  overflow  area  in
                caller for results.
```



```
#include <door.h>
gcc -mt [ flag ... ] file ... -ldoor  [ library ... ]

int   door_return(char    *data_ptr,    size_t    data_size,
                  door_desc_t *desc_ptr, uint_t num_desc);

The door_return() function returns from a  door  invocation.
It  returns control to the thread that issued the associated
door_call() and blocks waiting for the next door invocation.
See  door_call(3DOOR).  Results, if any, from the door invo-
cation are passed back  to the client in the buffers pointed
to  by data_ptr and desc_ptr. If there is not a client asso-
ciated with the door_return(), the calling  thread  discards
the  results,  releases  any  passed  descriptors  with  the
DOOR_RELEASE attribute, and blocks waiting for the next door
invocation.

RETURN VALUES
Upon successful completion, door_return() does not return to
the  calling process. Otherwise, door_return() returns -1 to
the calling process and sets errno to indicate the error.

ERRORS
The  door_return() function fails and returns to the calling
process if:
  E2BIG, EFAULT, EINVAL, EMFILE
```

```
#include <door.h>
gcc -mt [ flag ... ] file ... -ldoor  [ library ... ]

int door_cred(door_cred_t *info);
```

The door_cred() function returns credential information associated with the client (if any) of the current door invocation.

The contents of the info argument include the following fields:

```
   uid_t   dc_euid;        /* Effective uid of client */
   gid_t   dc_egid;        /* Effective gid of client */
   uid_t   dc_ruid;        /* Real uid of client */
   gid_t   dc_rgid;        /* Real gid of client */
   pid_t   dc_pid;         /* pid of client */
```

The credential information associated with the client refers to the information from the immediate caller; not necessarily from the first thread in a chain of door calls.

RETURN VALUES

Upon successful completion, door_cred() returns 0. Otherwise, door_cred() returns -1 and sets errno to indicate the error.

ERRORS

The door_cred() function will fail if:
  EFAULT, EINVAL


```
#include <door.h>
gcc [ flag ... ] file ... -ldoor [ library ... ]

int door_info(int d, struct door_info *info);
```

The door_info() function returns information associated with a door descriptor. It obtains information about the door descriptor d and places the information that is relevant to the door in the structure pointed to by the info argument.

The door_info structure pointed to by the info argument contains the following members:

```
  pid_t         di_target;      /* door server pid */
  door_ptr_t    di_proc;        /* server function */
  door_ptr_t    di_data;        /* data cookie for invocation */
  door_attr_t   di_attributes;  /* door attributes */
  door_id_t     di_uniquifier;  /* unique id among all doors */
```

The di_target member is the process ID of the door server, or -1 if the door server process has exited.

The values for di_attributes may be composed of the following:

DOOR_LOCAL	The door descriptor refers to a service procedure in this process.
DOOR_UNREF	The door has requested notification when all but the last reference has gone away.

```
  DOOR_UNREF_MULTI  Similar to DOOR_UNREF, except multi-
                    ple  unreferenced  notifications may
                    be delivered for this door.
  DOOR_IS_UNREF     There is currently only one descrip-
                    tor referring to the door.
  DOOR_REFUSE_DESC  The  door  refuses  any  attempt  to
                    door_call(3DOOR)  it  with  argument
                    descriptors.
  DOOR_REVOKED      The door descriptor refers to a door
                    that has been revoked.
  DOOR_PRIVATE      The door  has  a  separate  pool  of
                    server threads associated with it.
```

The di_proc and di_data members are returned as door_ptr_t objects rather than void * pointers to allow clients and servers to interoperate in environments where the pointer sizes may vary in size (for example, 32-bit clients and 64-bit servers). Each door has a system-wide unique number associated with it that is set when the door is created by door_create(). This number is returned in di_uniquifier.

RETURN VALUES

Upon successful completion, 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The door_info() function will fail if:
  EFAULT, EBADF


```
#include <door.h>
gcc -mt [ flag ... ] file ... -ldoor  [ library ... ]

void (*)() door_server_create(void(*create_proc)(door_info_t*));
```

Normally, the doors library creates new door server threads in response to incoming concurrent door invocations automatically. There is no pre-defined upper limit on the number of server threads that the system creates in response to incoming invocations (1 server thread for each active door invocation). These threads are created with the default thread stack size and POSIX (see standards(5)) threads cancellation disabled. The created threads also have the THR_BOUND | THR_DETACHED attributes for Solaris threads and the PTHREAD_SCOPE_SYSTEM | PTHREAD_CREATE_DETACHED attributes for POSIX threads. The signal disposition, and scheduling class of the newly created thread are inherited from the calling thread (initially from the thread calling door_create(), and subsequently from the current active door server thread).

The door_server_create() function allows control over the creation of server threads needed for door invocations. The procedure create_proc is called every time the available server thread pool is depleted. In the case of private server pools associated with a door (see the DOOR_PRIVATE attribute in door_create()), information on which pool is depleted is passed to the create function in the form of a door_info_t structure. The di_proc and di_data members of the door_info_t structure can be used as a door identifier

associated with the depleted pool. The create_proc procedure may limit the number of server threads created and may also create server threads with appropriate attributes (stack size, thread-specific data, POSIX thread cancellation, signal mask, scheduling attributes, and so forth) for use with door invocations.

The specified server creation function should create user level threads using thr_create() with the THR_BOUND flag, or in the case of POSIX threads, pthread_create() with the PTHREAD_SCOPE_SYSTEM attribute. The server threads make themselves available for incoming door invocations on this process by issuing a door_return(NULL, 0, NULL, 0). In this case, the door_return() arguments are ignored. See door_return(3DOOR) and thr_create(3C).

The server threads created by default are enabled for POSIX thread cancellations which may lead to unexpected thread terminations while holding resources (such as locks) if the client aborts the associated door_call(). See door_call(3DOOR). Unless the server code is truly interested in notifications of client aborts during a door invocation and is prepared to handle such notifications using cancellation handlers, POSIX thread cancellation should be disabled for server threads using pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, NULL).

The create_proc procedure need not create any additional server threads if there is at least one server thread currently active in the process (perhaps handling another

door invocation) or it may create as many as seen fit each time it is called. If there are no available server threads during an incoming door invocation, the associated door_call() blocks until a server thread becomes available. The create_proc procedure must be MT-Safe.

RETURN VALUES

Upon successful completion, door_server_create() returns a pointer to the previous server creation function. This function has no failure mode (it cannot fail).

```
#include <door.h>
gcc -mt [ flag... ] file... -ldoor  [ library... ]

int door_bind(int did);

int door_unbind(void);
```

The door_bind() function associates the current thread with a door server pool. A door server pool is a private pool of server threads that is available to serve door invocations associated with the door did.

The door_unbind() function breaks the association of door_bind() by removing any private door pool binding that is associated with the current thread.

Normally, door server threads are placed in a global pool of available threads that invocations on any door can use to dispatch a door invocation. A door that has been created with DOOR_PRIVATE only uses server threads that have been associated with the door by door_bind(). It is therefore necessary to bind at least one server thread to doors created with DOOR_PRIVATE.

The server thread create function, door_server_create(), is initially called by the system during a door_create() operation. See door_server_create(3DOOR) and door_create(3DOOR).

The current thread is added to the private pool of server threads associated with a door during the next door_return() (that has been issued by the current thread after an associated door_bind()). See door_return(3DOOR). A server thread performing a door_bind() on a door that is already bound to a different door performs an implicit door_unbind() of the previous door.

If a process containing threads that have been bound to a door calls fork(2), the threads in the child process will be bound to an invalid door, and any calls to door_return(3DOOR) will result in an error.

RETURN VALUES
Upon successful completion, a 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS
The door_bind() and door_unbind() functions fail if:

```
  EBADF    The did argument is not a valid door.
  EBADF    The door_unbind() function was called by a
           thread that is currently not bound.
  EINVAL   did was not created with the DOOR_PRIVATE
           attribute.
```

```
#include <door.h>
gcc -mt [ flag ... ] file ... -ldoor  [ library ... ]

int door_revoke(int d);
```

The door_revoke() function revokes access to a door descriptor. Door descriptors are created with door_create(3DOOR). The door_revoke() function performs an implicit call to close(2), marking the door descriptor d as invalid.

A door descriptor can only be revoked by the process that created it. Door invocations that are in progress during a door_revoke() invocation are allowed to complete normally.

RETURN VALUES

Upon successful completion, door_revoke() returns 0. Otherwise, door_revoke() returns -1 and sets errno to indicate the error.

ERRORS

The door_revoke() function will fail if:

- EBADF An invalid door descriptor was passed.
- EPERM The door descriptor was not created by this process (with door_create(3DOOR)).

Beispiele:

```
client1.c  - einfach
server1.c

client4.c  - Anzeigen Puffer und Informationen
server4.c  - Anzeigen Puffer und Informationen

client7.c  - zwei Funktionen
server7.c

doorinfo.c - Informationen auslesen
```