Systemprogrammierung
Test 2

Jan Fässler

3. Semester (HS 2012)

Inhaltsverzeichnis

1 Sockets **1**
- 1.1 Design der Socket Schnittstelle . . . 1
- 1.2 Grundkonzepte . . . 1
- 1.3 Erstellen eine Verbindung . . . 1
- 1.4 Senden / Empfangen . . . 1
- 1.5 Warten auf Socket-I/O . . . 2
- 1.6 Beenden einer Verbindung . . . 2
- 1.7 Datenstruktur für die Protokollauswahl . . . 3
- 1.8 Datenfluss durch die Schichten . . . 3

2 System V IPC **4**
- 2.1 Zentrale Konzepte . . . 4
- 2.2 Verwaltung der Objekte . . . 4
- 2.3 Kommandos . . . 4
- 2.4 Shared Memory . . . 5
  - 2.4.1 Einleitung . . . 5
  - 2.4.2 Beispiel-Adressraum eines Prozesses . . . 5
  - 2.4.3 Erstellen eines Segments . . . 5
  - 2.4.4 Einbinden eines Segments . . . 6
  - 2.4.5 Entfernen eines Segmentes . . . 7
- 2.5 Message Queues . . . 8
  - 2.5.1 Einleitung . . . 8
  - 2.5.2 Erzeugen einer Message Queue . . . 8
  - 2.5.3 Senden einer Nachricht . . . 8
  - 2.5.4 Empfang einer Nachricht . . . 9
- 2.6 Semaphore . . . 11
  - 2.6.1 Einleitung . . . 11
  - 2.6.2 Benutzung für gegenseitigen Ausschluss . . . 11
  - 2.6.3 Benutzung für Prozess-Synchronisation . . . 11
  - 2.6.4 Additive Semaphore . . . 11
  - 2.6.5 Semaphore in Unix . . . 11
  - 2.6.6 Datenstrukturen . . . 12
  - 2.6.7 Implementierung P and V . . . 13

3 Remote Procedure Calls **14**
- 3.1 Genereller Aufbau . . . 14
- 3.2 Server / Client Bindung . . . 14
- 3.3 RPC Portmapping Ablauf . . . 14
- 3.4 Auswahl des Transportprotokolls . . . 15
- 3.5 Programmierbeispiel . . . 15
- 3.6 Ausnahmebehandlung . . . 15
- 3.7 Aufrufsemantik . . . 16
- 3.8 Benutzung und Beschränkungen . . . 16

1 Sockets

1.1 Design der Socket Schnittstelle

Das Ziel der Schnittstelle ist eine offene, generische Programmierschnittstelle fu?r Interprozesskommunikation (lokal und u?ber verschiedenste Netzwerke) fu?r Unix. Die Anforderungen an die Socket-Schnittstelle sind:

Transparenz
bezüglich Netzwerkverhalten

Effizienz
um die Programmierer zu überzeugen

Kompatibilität
mit Unix Standard I/O

1.2 Grundkonzepte

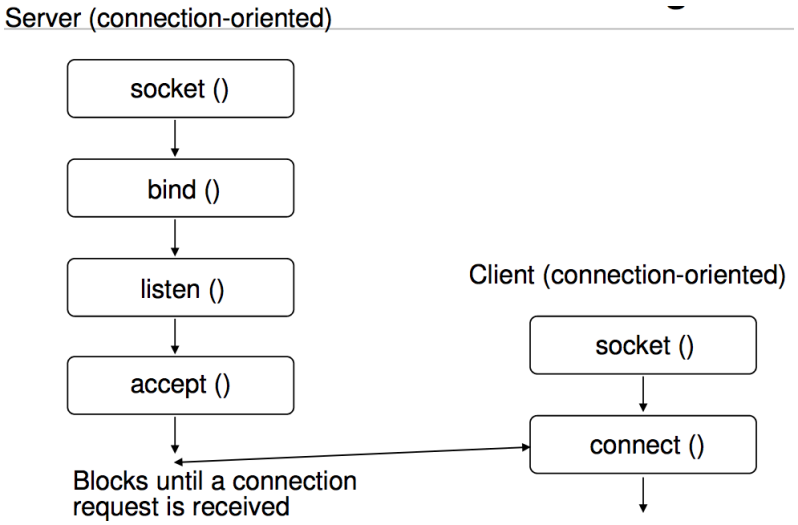
communication domains
Unterstützung verschiedener Netzwerkprotokolle

communication types
Klassifikation von Kommunikationseigenschaften

name binding
Benennung und Adressierung von Kommunikationsendpunkten

sockets
Einheitliche Abstraktion für Kommunikationsendpunkte in einem Programm/Prozess

1.3 Erstellen eine Verbindung



1.4 Senden / Empfangen

Damit die Socktet-Schnittstelle möglichst kompatibel ist und und mehrere Domähnen und Protokolle unterstützt werden können gibt es für das Senden und Empfangen mehrere Paare von Systemaufrufen.

read/write

- Rückwärtskompatibilität zu Unix Standard I/O
- Ermöglichen von Pipelining mit dup()

send/recv

- zusätzliche Funktionalität und Parametrisierung
- für TCP

sento/recvfrom

- erlaubt das Senden von ganzer Datenportionen ohne Verbindung
- für UDP

sendmsg/recvmsg

- überträgt/signalisiert eine vollständige Nachricht, anstelle eines unstrukturierten Byte-Stroms
- die Applikation wird erst benarchichtigt, wenn eine ganze Nachricht angekommen ist

1.5 Warten auf Socket-I/O

accept()
Blockiert den aufrufenden Prozess solange, bis ein Verbindungsaufbauwunsch am Socket signalisiert wird.

select()
Erlaubt eine nicht blockierendes Warten auf neue Verbindungen bzw. nur kurze Blockaden. (kann zB über SIGALARM realisiert werden)

1.6 Beenden einer Verbindung

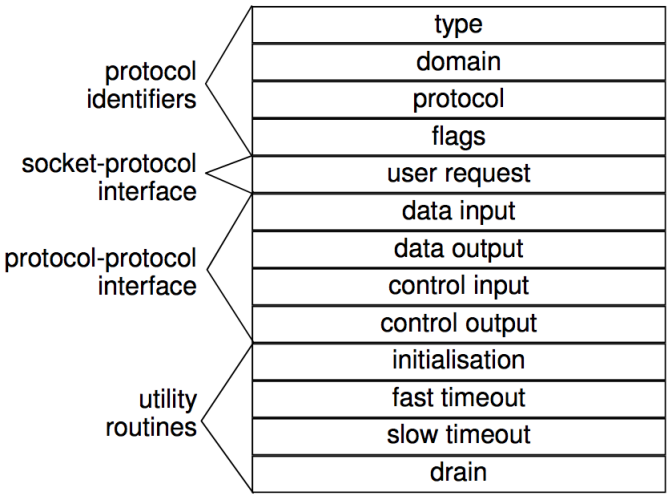
shutdown
Signalisiert eine Verbindungsabbruch-Anforderung zur anderen Seite und wartet dann auf die Bestätigung. Inzwischen noch ankommende Daten werden noch bestätigt, aber nicht an die Applikation weitergegeben.

close
Deralloziert die lokale Socket Datenstruktur und die zugeordneten Ressourcen (Puffer).

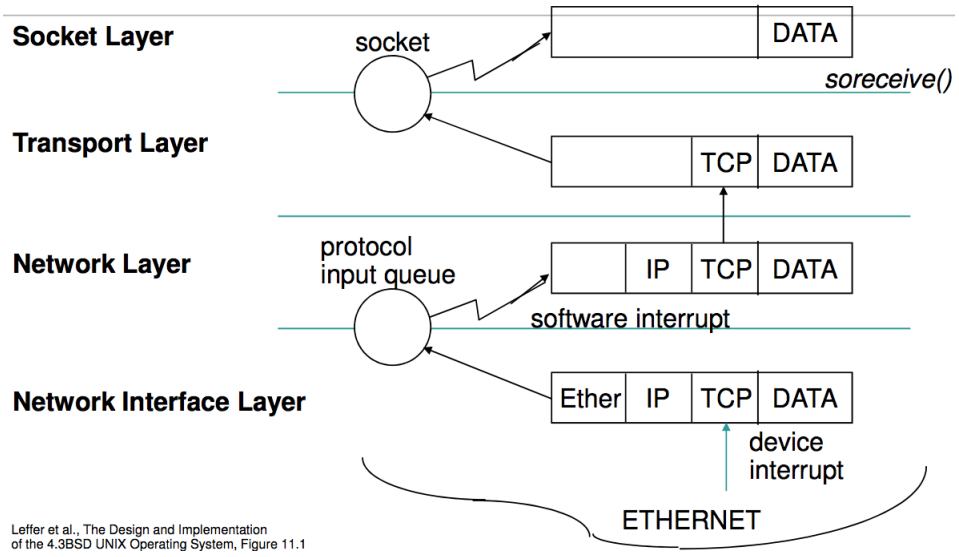
exit
Terminiert das PRogramm ohne explizites shutdown/close, überlässt das Aufräumen dem Betriebssystem und der Gegenpartei.

Provider Abort
überlässt das Aufräumen dem Betriebssystem beider Gegenparteiten.

1.7 Datenstruktur für die Protokollauswahl



1.8 Datenfluss durch die Schichten



Leffer et al., The Design and Implementation of the 4.3BSD UNIX Operating System, Figure 11.1

2 System V IPC

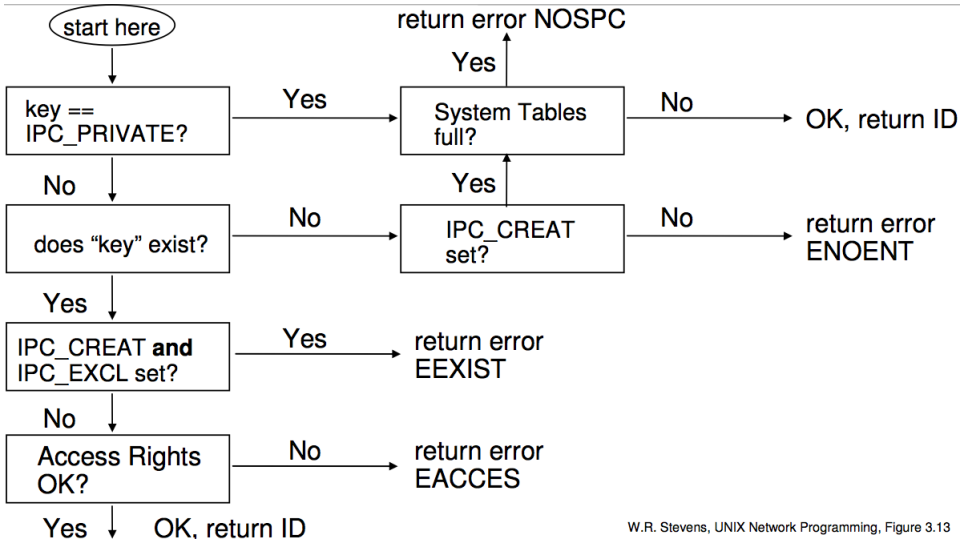
2.1 Zentrale Konzepte

Der Kernel verwaltet je eine separate Tabelle für jeden der drei System V IPC Mechanismen:

- Message Queues
- Shared Memory
- Semaphore

Für die Verwaltung der drei Mechanismen werden allerdings die **gleichen Prozeduren** verwendet. Der Zugriff erfolgt über einen **numerischen Schlüssel**. Es gibt keine Registratur für verwendete/reservierte Schlüssel, was **Kollisionen ausschliesst**. Die System V IPC Objekte sind **nicht kompatibel** mit Standard I/O-basierter Prozesskommunikation wie Dateien, Sockets, Pipes oder Geräte.

2.2 Verwaltung der Objekte



2.3 Kommandos

ipcs
: Listet alle aktiven System V IPC Objekte eines Systems auf

iprm
: Löscht ein System V IPC Objekt

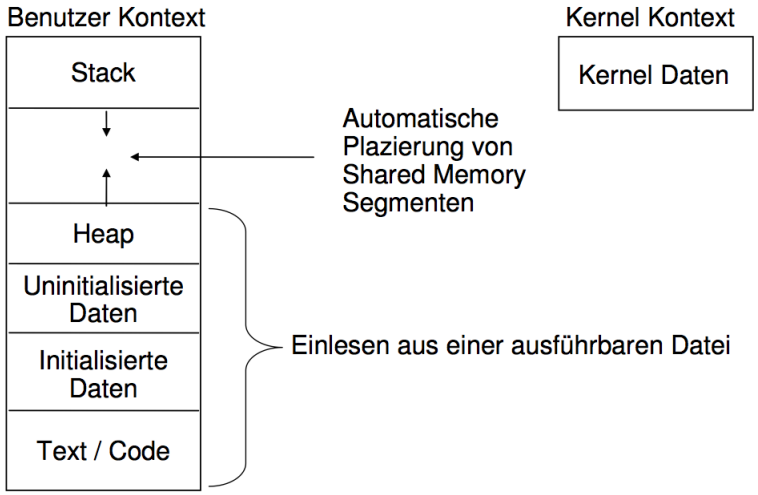
2.4 Shared Memory

2.4.1 Einleitung

Shared Memory erlaubt die gemeinsame Nutzung von Hauptspeicher-Seiten zwischen verschiedenen Prozessen. Es wird ein dedizierter Segment-Typ verwendet, der aber auf der normalen Unix-Speicherverwaltung basiert. Ein eNutzung eines Shared Memory's zwischen nicht verwandten PRozessen ist möglich aber limitiert auf ein lokales System. Es gibt keinen Synchronisationsmechanismus für Shared Memory. Als Synchronisation gilt das Gentlemen's Agreement:

- Signalbit im Shared Memory
- Signale
- Semaphore

2.4.2 Beispiel-Adressraum eines Prozesses



2.4.3 Erstellen eines Segments

Wenn ein Segment alloziert wird, werden folgende Schritte durchgeführt:

- remove region from linked list of free regions;
- assign region type;
- assign region inode pointer;
- if (inode pointer not null) increment inode reference count;
- place region on linked list of active regions;
- return (locked region);

Listing 1: Erstellen eines Shared Memory Segments

```
1 #include <sys/types.h> /* supplies key_t */
  #include <sys/ipc.h>   /* supplies ipc_perm */
  #include <sys/shm.h>   /* supplies structures and macros for
                          * shared mem. data structures etc. */
```

```
  int size, permflags, shm_id;
6 key_t key;
  ...
  shm_id = shmget (key, size, permflags);
```

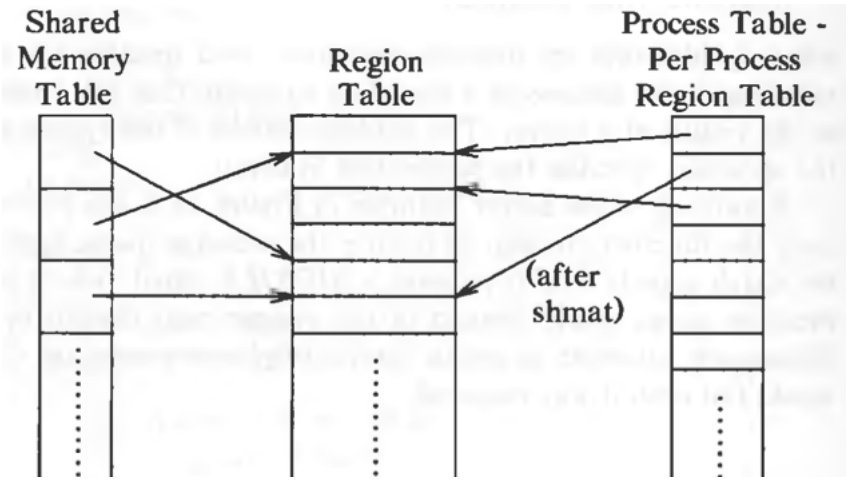
2.4.4 Einbinden eines Segments

Wenn ein Segment eingebunden wird, werden folgende Schritte durchgeführt durch den System Call shmat():

- check validity of descriptor, permissions
- if (user specified virtual address)
  - round off virtual address, as specified by flags;
  - check legality of virtual address, size of region;
- else
  - kernel picks virtual address: error if none available;
- attach region to process address space (algorithm attachreg);
- if (region being attached for the first time)
  - allocate page tables, memory for region (algorithm growreg);
- return (virtual address where attached);

Listing 2: Einbindung eines Shared Memory Segments

```
  #include <sys/types.h> /* supplies key_t */
2 #include <sys/ipc.h>   /* supplies ipc_perm */
  #include <sys/shm.h>   /* supplies structures and macros for
                          * shared mem. data structures etc. */
  int shm_id, shmflags;
  char *memptr, *daddr, *shmat();
7 ...
  memptr = shmat (shm_id, daddr, shmflags);
```



2.4.5 Entfernen eines Segmentes

Wenn ein Segment entfernt wird, werden folgende Schritte durchgeführt durch den System Call detachreg():

- get auxiliary memory management tables for process,
- release as appropriate;
- decrement process size;
- decrement region reference count;
- if (region count is 0 and region not sticky bit)
  - free region (algorithm freereg);
- else */* either reference count non-0 or region sticky bit on */*
  - free inode lock, if applicable (inode associated with region);
  - free region lock;

Listing 3: Entfernen eines Shared Memory Segments

```
  int       retval;
2 char      *memptr;
  ...
  retval = shmdt (memptr);
```

2.5 Message Queues

2.5.1 Einleitung

Das eigentliche IPC Objekt ist die Message Queue, nicht die einzelne Nachricht. Die MEssage Queue erlaubt den Austausch beliebig strukturierter Daten. Die Nachrichten können über ein Typenfeld identifiziert werden. Ein Empfänger kann Nachrichten auf unterschiedlichste Arten empfachen (Rheienfolge, Nachrichtentyp, ...). Message Queues erlaubt die Kommunikation zwischen nicht-verwandten Prozessen, ist aber beschränkt auf ein lokales System.

2.5.2 Erzeugen einer Message Queue

Listing 4: Erzeugen einer Message Queue

```
#include <sys/types.h> /* supplies key_t */
#include <sys/ipc.h>   /* supplies ipc_perm */
#include <sys/msg.h>   /* supplies structures and macros
                        * for Message Queues, Messages etc. */
int msg_qid, permflags;
key_t key;
...
msg_qid = msgget (key, permflags);
```

2.5.3 Senden einer Nachricht

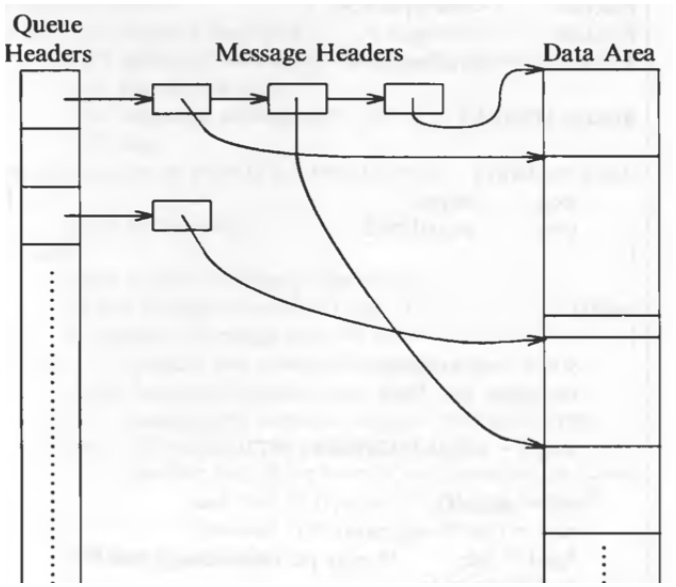
Wenn eine Nachricht versendet wird, werden folgende Schritte durchgeführt durch den System Call msgsnd():

- check legality of descriptor, permissions;
- while (not enough space to store message)
  - if (flags specify not to wait) return;
  - sleep (until event enough space is available);
- get message header;
- read message text from user space to kernel;
- adjust data structures:
  - enqueue message header,
  - message header points to data,
  - counts,
  - time stamps,
  - process ID;
- wakeup all processes waiting to read message from queue;

Listing 5: Versenden einer Nachricht

```
#include <sys/types.h> /* supplies key_t */
#include <sys/ipc.h>   /* supplies ipc_perm */
#include <sys/msg.h>   /* supplies structures and macros
```

```
                        * for Message Queues, Messages etc. */
int msg_qid, size, flags, retval;
struct my_msg {
  long mtype;
  char mtext[SOMEVALUE];
} message;
...
retval = msgsnd (msg_qid, &message, size, flags);
```



2.5.4 Empfang einer Nachricht

Wenn eine Nachricht empfangen wird, werden folgende Schritte durchgeführt durch den System Call msgsnd():

- check permissions;
- **loop;**
- check legality of message descriptor;
- */* find message to return to user */*
- if (requested message type == 0)
  - consider first message on queue;
- else if (requested message type > 0)
  - consider first message on queue with given type;
- else */* requested message type < 0 */*
- consider first of the lowest typed messages on queue, such that its type is <= absolute value of requested type;
- if (there is a message)

  - adjust message size or return error if user size too small;
  - copy message type, text from kernel space to user space;
  - unlink message from queue;
  - return;
- */* no message */*
- if (flags specify not to sleep) return with error;
- sleep (event message arrives on queue);
- **goto loop;**

Listing 6: Empfangen einer Nachricht

```
#include <sys/types.h> /* supplies key_t */
#include <sys/ipc.h>   /* supplies ipc_perm */
#include <sys/msg.h>   /* supplies structures and macros
                        * for Message Queues, Messages etc. */
int msg_qid, size, flags, retval;
struct my_msg {
  long mtype;
  char mtext[SOMEVALUE];
} message;
long msg_type;
...
retval = msgrcv (msg_qid, &message, size, msg_type, flags);
```

2.6 Semaphore

2.6.1 Einleitung

Semaphoren erlauben die Synchronisation von nicht-verwandten PRozessen sowie den Schutz von kritischen Abschnitten durch gegenseitigen Ausschluss. In Unix/Linux sind sie additiv und unterstützen Semaphore Sets. Sie sind aber auch begrenzt auf das lokale System.

2.6.2 Benutzung für gegenseitigen Ausschluss

```
s: semaphore (1);

P1 : process                          P2 : process
      ...                                   ...
      P(s);                                 P(s);
      ... -- critical section               ... - critical section
      V(s);                                 V(s);
      ...                                   ...
end  process                          end  process
```

2.6.3 Benutzung für Prozess-Synchronisation

```
s: semaphore (0);

P1 : process                          P2 : process
      ...                                   ...
      V(s); -- signal event                 P(s); -- wait for event
      ...                                   ...
end  process                          end  process
```

2.6.4 Additive Semaphore

```
exclusion : add_semaphore (N);

     type reader = process
     ...
     P (exclusion, 1);
          ... - read within critical section
     V (exclusion, 1);
     end process;

     type writer = process
     ...
     P (exclusion, N);
          ... -- write within critical section
     V (exclusion, N);
     end process;
```

2.6.5 Semaphore in Unix

```
P(sem)
     if (sem != 0)
        decrement sem value by one
     else
        wait until sem becomes non-zero


V (sem)
     if (queue of waiting processes not empty)
        restart first process in wait queue
     else
        increment sem value by one
```

2.6.6 Datenstrukturen

Listing 7: Semctl Command Definitions

```
#define GETNCNT 3  /* get semncnt */
#define GETPID  4  /* get sempid */
#define GETVAL  5  /* get semval */
#define GETALL  6  /* get all semval's */
#define GETZCNT 7  /* get semzcnt */
#define SETVAL  8  /* set semval */
#define SETALL  9  /* set all semval's */
```

Listing 8: set of semaphores

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

Listing 9: one semaphore

```
struct sem {
  ushort          semval;  /* semaphore text map address */
  short           sempid;  /* pid of last operation */
  ushort          semncnt; /* # awaiting semval > cval */
  ushort          semzcnt; /* # awaiting semval = 0 */
};
```

Listing 10: template for semctl system calls

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

Listing 11: template for semop system calls

```
union semun {
    short   sem_num;  /* semaphore # */
    short   sem_op;   /* semaphore operation */
    short   sem_flg;  /* operation flags */
};
```

Listing 12: There is one undo structure per process

```
union sem_undo {
    struct sem_undo *un_np; /* ptr to next active undo structure */
    short           un_cnt; /* # of active entries */
    struct undo {
        short       un_aoe; /* ?adjust on exit?-values */
```

```
            short       un_num; /* semaphore # */
            int         un_id;  /* semid */
        }   un_ent[1];          /* (semume) undo entries (one min.) */

};
```

2.6.7 Implementierung *P* and *V*

Listing 13: Programmierbeispiel: Implementierung

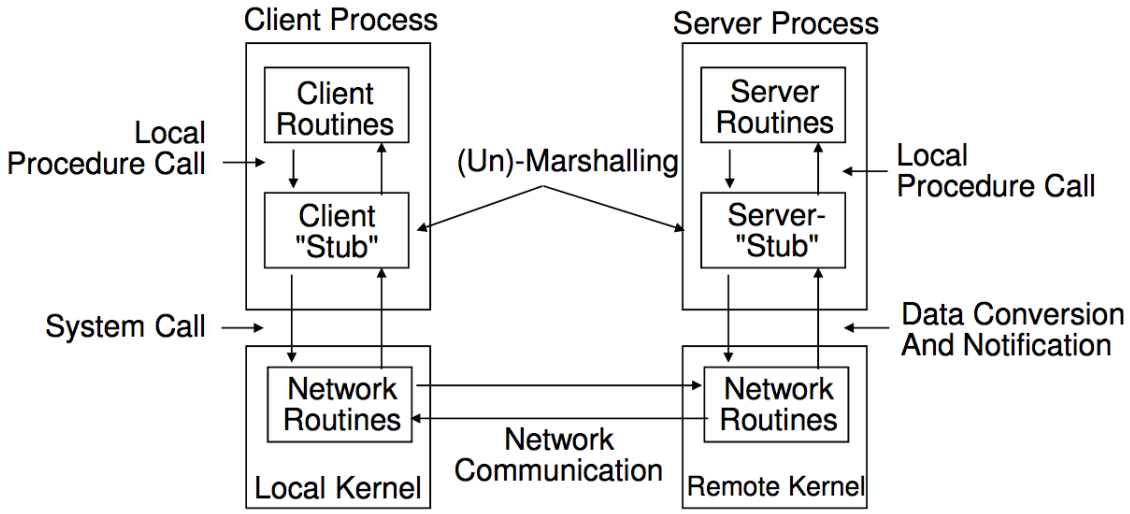
```
P (semid) int semid; {
      struct sembuf p_buf;
      p_buf.sem_num = 0;
      p_buf.sem_op = -1;         /* negativer Wert, also Fall 1 = P() */
      p_buf.sem_flg = SEM_UNDO;
      if (semop (semid, &p_buf, 1) == -1) {
           perror (?p(semid) failed?);
           exit (1);
      } else return (0);
}
V (semid) int semid; {
      struct sembuf v_buf;
      v_buf.sem_num = 0;
      v_buf.sem_op = 1;         /* positiver Wert, also Fall 2 = P() */
      v_buf.sem_flg = SEM_UNDO;
      if (semop (semid, &v_buf, 1) == -1) {
           perror (?v(semid) failed?);
           exit (1);
      } else return (0);
}
```

Listing 14: Programmierbeispiel: Benutzung

```
main () {
      key_t semkey = 0x200;
      if (fork () == 0) handlesem (semkey);
      if (fork () == 0) handlesem (semkey);
      if (fork () == 0) handlesem (semkey);
}
handlesem (skey) key_t skey; {
      int semid, pid = getpid();
      if ((semid = initsem (skey)) > 0) exit (1);
      printf (?\nprocess %d before critical section\n?, pid); P (semid);
      printf (?process %d in critical section\n?, pid);
      /* in real life do something interesting */
      sleep (2);
      printf (?process %d leaving critical section\n?, pid); V (semid);
      printf (?process %d exiting\n?, pid);
      exit (0);
}
```

3 Remote Procedure Calls

3.1 Genereller Aufbau



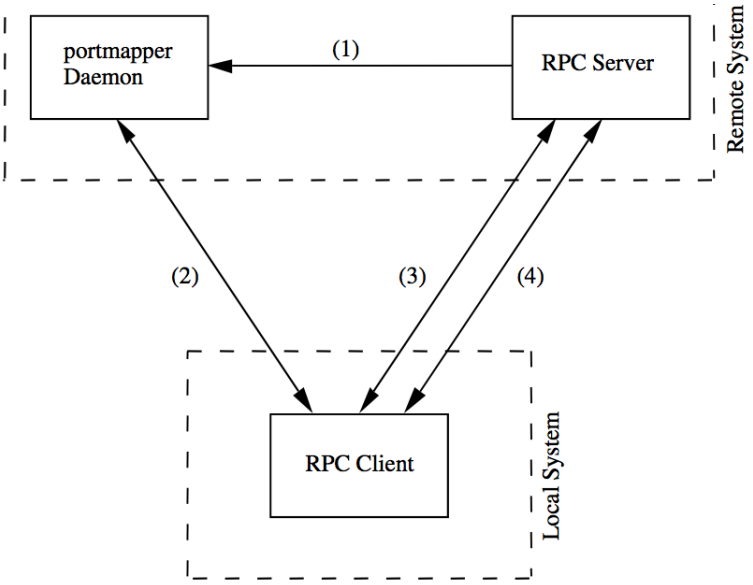
3.2 Server / Client Bindung

- Finden eines Servers im Netz
- Finden des gewünschten Service auf dem Server im Netz
- Sun RPC benutzt die Standard-Unix-Methode für das Finden von Servern im Internet (DNS).
- Alle Serverprogramme, Programmversionen und angebotenen Prozeduren werden mit eindeutigen Nummern gekennzeichnet.
- Ein Prozess kann eine oder mehrere Prozeduren anbieten.
- Der protmapper Prozess (Linux: rpcbind) auf Porst 111 auf jedem Serversystem dient als zentrale, lokale Registratur für verfügbare RPC-Dienste.

3.3 RPC Portmapping Ablauf

1. Der Server erstellt einen Socket und registriert Programmnummer, Versionsnummer und Portnummer beim Portmapper.
2. Ein Client kontaktiert den Portmapper und fragt nach einer Programm-, Versions- und Prozedurnummer. Falls lokal bekannt, sendet der Portmapper die Portnummer zurück.

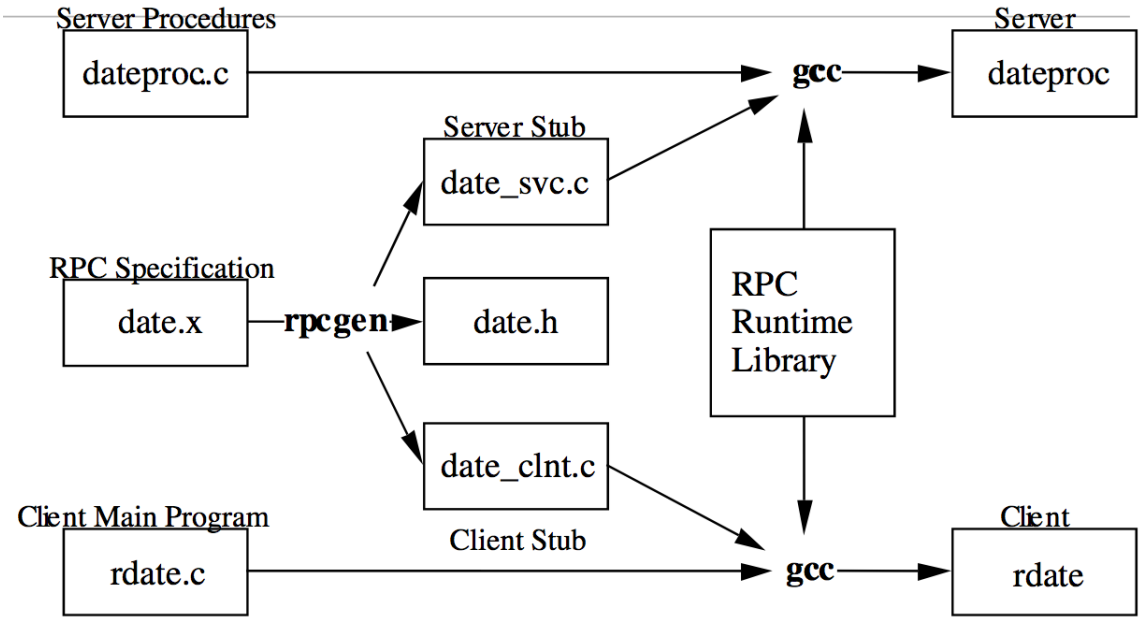
3./4. Der Client kann nun die gewünschte Prozedur direkt beim Server aufrufen.



3.4 Auswahl des Transportprotokolls

- RPC ist unabhängig von spezifischen Transportdiensten oder Protokollen. (Sun RPC unterstützt TCP und UDP)
- Es werden Abbildungen auf die üblichen Transportprotokolle angeboten.

3.5 Programmierbeispiel



3.6 Ausnahmebehandlung

- Zusätzlich zu lokalen Fehlern können in RPC weitere Fehler auf dem Serversystem und bei der Datenübermittlung durch das Netz auftreten.

- Abbruch von bereits übermittelten oder gestarteten Prozeduraufrufen durch den Klienten beim Server.
- Terminieren des Klienten bevor der Server den Ablauf der entfernten Prozdedur beendet hat.
- Sun RPC verwendet das automatische Neusenden von Anfragen im Fall der Benutzung von UDP, und erkennt verlorene Verbindungen in TCP.
- Sun RPC unterstützt keinen separaten Kontrollkanal.

3.7 Aufrufsemantik

- Prozedur wird genau einmal ausgeführt
- Prozedur wird höchstens einmal ausgeführt
- Prozedur wird mindestens einmal ausgeführt
- Jeder Server unterhält einen Cache mit kürzlich erhaltenen Prozeduraufrufen und den zurückgesendeten Resultaten, und sendet das gespeicherte Resultat zurück, wenn ein Duplikat eines PRozeduraufrufs entdeckt wird.

3.8 Benutzung und Beschränkungen

- Die Server sind meist zustandslos:
  - alle Operationen sind unabhängig voneinander
  - Bobustheit gegen Fehler im Klienten, im Server und im Netz
- Performance (lokale vs. entfernte Prozeduren)
- Service-Strategien (ein Server, ein Server pro Klient, ...)
- Verteilungsstrategien (wo im Netz werden Server plaziert.