AG Betriebssysteme FB3
Kirsten Berkenkötter

Allerlei Nützliches

1 Semaphore

1.1 Allgemein

Semaphore sind unter System V IPC erweitert:

- Ganze Arrays von Semaphoren können auf einmal angelegt werden.
- In einer Operation können mehrere Semaphore auf einmal modifiziert werden.
- Das Inkrementieren und Dekrementieren kann in größeren Schritten als 1, bzw. -1 erfolgen.
- Semaphor-Operationen können nach Beendigung des Prozesses rückgängig gemacht werden, d.h. bei Aufruf von exit() werden alle rückgängig zu machenden Operationen zurückgesetzt.

1.2 Header

Benötigt werden:

- <sys/ipc.h>
- <sys/sem.h>
- <sys/types.h>

1.3 Anlegen eines Semaphors

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

- key: Schlüssel für das Semaphor-Array, IPC_PRIVATE als spezieller Wert, wenn keine existierendes Objekt genommen werden soll
- nsems: Anzahl der Semaphore
- semflg:
  - 0400: Leserecht Erzeuger
  - 0200: Schreibrecht Erzeuger
  - usw.
  - IPC_CREAT: anlegen, wenn es das Semaphor nicht gibt
  - IPC_EXCL: wenn IPC_CREAT gesetzt ist, und der Semaphor existiert, kehrt die Funktion mit dem Fehler EEXISTS zurück

- mehrere flags werden durch | verodert
- Rückgabewert: Filedescriptor, negativer Wert bei Fehlern

Typischer Aufruf:

```
int semid;

if (0 > (semid = semget(0x2a, 10, (IPC_CREAT | 0666))))
{
  perror(‘‘problem with semget()’’);
  exit(1);
}
```

1.4 Operationen auf Semaphoren

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

- semid: Id des Semaphorarrays
- Array mit den Operationen, Index 0 erste Operationen, Index 1 zweite, usw.
- Anzahl der Operationen
- Rückgabewert: negativer Wert bei Fehlern

Operationen:

```
struct sembuf
{
  ushort sem_num; //Index des Semaphors
  short sem_op;   //auszuführende Operation
  short sem_flg;  //Flags
}
```

Flags bei Operationen:

- IPC_NOWAIT: nichtblockierender Aufruf
- SEM_UNDO: diese Operation wird bei Beendigung des Prozesses rückgängig gemacht

Typischer Aufruf:

```
struct sembuf sops[SEM_NUM];

sops[0].sem_num = 1;
sops[0].sem_op = 1;
sops[0].sem_flg = 0;

if(0 > semop(semid, &(sops[0]), 1))
{
  perror(‘‘problem with semop()’’);
  exit(1);
}
```

1.5 Kommandos auf Semaphoren

int semctl(int semid, int semnum, int cmd, ...);

- semid: id des Semaphorarrays
- semnum: Nummer eines Semaphors
- cmd: auszuführendes Kommando
- ...: Argumente für das Kommando
- Rückgabewert: hängt vom Kommando ab, negativ bei Fehler

Kommandos:

- IPC_STAT: Infos, Argument wird benötigt
- IPC_SET: Werte für die Struktur angeben, z.B. Zugriffsrechte, Argument wird benötigt, semnum wird ignoriert
- IPC_RMID: Semaphor löschen, alle wartenden Prozesse werden aufgeweckt, kein Argument, semnum wird ignoriert
- GETALL: die Werte aller Semaphore zurückgeben, Argument wird für die Rückgabe benötigt, semnum wird ignoriert
- GETNCNT: Anzahl der Prozesse, die auf den semnum-ten Semaphor warten, Argument wird nicht benötigt, Rückgabewert ist die Anzahl
- GETPID: PID des Prozesses, der semop() auf dem semnum-ten Semaphor zuletzt ausgeführt hat, ist der Rückgabewert
- GETVAL: gibt den Wert des semnum-ten Semaphor zurück
- GETZCNT: Anzahl der Prozesse die auf SETVAL des semnum-ten Semaphors warten
- SETALL: alle Semaphore werden gesetzt, Argument wird benötigt, semnum wird ignoriert
- SETVAL: einen Semaphor setzen, Argument wird benötigt

Argumente:

```
union semun
{
  int val;                //Wert für SETVAL
  struct semid_ds *buf;   //Puffer für IPC_STAT und IPC_SET
  ushort *array,          //Feld für GETALL und SETALL
  struct seminfo *__buf;  //Puffer für IPC_INFO
}
```

Typische Aufrufe:

```
if(0 > semctl(semid, 10, IPC\_RMID))
{
  perror(‘‘problem with semctl’’);
  exit(1);
}

union semun args;
args.val = 3;

if(0 > semctl(semid, 1, SETVAL, args))
{
  perror(‘‘problem with semctl’’);
  exit(1);
}
```

2 Shared Memory

2.1 Header

Benötigt werden:

- <sys/ipc.h>
- <sys/shm.h>
- <sys/types.h>

2.2 Anlegen von Shared Memory

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

- key: Schlüssel für das Shared Memory, IPC_PRIVATE als spezieller Wert, wenn keine existierendes Objekt genommen werden soll
- size: Größe des Shared Memory
- shmflg:
  - 0400: Leserecht Erzeuger
  - 0200: Schreibrecht Erzeuger
  - usw.
  - IPC_CREAT: anlegen, wenn es das Semaphor nicht gibt
  - IPC_EXCL: wenn IPC_CREAT gesetzt ist, und der Semaphor existiert, kehrt die Funktion mit dem Fehler EEXISTS zurück
- mehrere flags werden durch | verodert
- Rückgabewert: Filedescriptor, negativer Wert bei Fehlern

Typischer Aufruf:

```
typedef struct
{
  int numberOfReindeers;
  int numberOfElfes;
} shared_t;

int shmid;
shared_t *shmPtr;

if(0 > (shmid = shmat(0x2a, sizeof(shared_t), (IPC_CREAT | 0666))))
{
  perror(‘‘problem with semget()’’);
  exit(1);
}
```

2.3 Shared Memory in Memory des Prozesses einblenden

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

- shmid: ID des Shared Memory
- shmaddr: Zeiger auf Shared Memory im Prozess, bei 0 wird eine freie ausgewählt
- shmflg: Flags
- Rückgabewert: Filedescriptor, negativ, wenn's nicht klappt

Flags:

- SHM_RND: wenn die Adresse nicht 0 ist, dann wird die in shmaddr angegebene Adresse abgerundet auf den nächsten mehrfachen SHMLBA-Wert, sonst muss shmaddr in den Seitenrahmen passen, d.h. Anfang einer Page sein
- SHM_REMAP: Mapping soll bestehendes Mapping ersetzen (nur Linux)
- SHM_RDONLY: readonly

Typischer Aufruf:

```
if(0 > (shmPtr = shmat(shmid), 0, 0)))
{
  perror(‘‘problem with shmat()’’);
  exit(1);
}
```

2.4 Shared Memory aus Memory des Prozesses ausblenden

int shmdt(const void *shmaddr);

- shmaddr: Zeiger auf Shared Memory im Prozess
- Rückgabewert: negativ, wenn's nicht klappt

Typischer Aufruf:

```
if(0 > shmdt(shmPtr)) //bei exit automatisch ausgeführt, aber Memory besteht
{
  perror(‘‘problem with shmdt()’’);
  exit(1);
}
```

2.5 Kommandos auf Shared Memory

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

- shmid: id des Shared Memory
- cmd: auszuführendes Kommando
- buf: Argument für Kommando
- Rückgabewert: hängt vom Kommando ab, negativ bei Fehler

Kommandos:

- IPC_STAT: Infos, Argument wird benötigt
- IPC_SET: Werte für die Struktur angeben, z.B. Zugriffsrechte, Argument wird benötigt
- IPC_RMID: Shared Memory löschen, nach dem letzten shmdt(), kein Argument
- SHM_LOCK: nur Superuser, kein Swapping möglich
- SHM_UNLOCK: nur Superuser, Swapping möglich

Typischer Aufruf:

```
if(0 > shmctl(semid, IPC\_RMID, NULL)
{
  perror(‘‘problem with shmctl’’);
  exit(1);
}
```

3 UDP-Sockets

3.1 Allgemein

- Zum Erstellen von UDP-Sockets werden die Funtionen socket() und bind() verwendet.
- Zum Erstellen von TCP/IP-Scokets müssen zusätzlich listen() und accept() verwendet werden (s. man-Pages).
- Bei UDP können die Funtionen read() und write() nicht zum Senden und Empfangen verwendet werden (im Gegensatz zu TCP/IP), da keine permanente Verbindung besteht.

3.2 Header

Benötigt werden:

- <sys/types.h>
- <sys/socket.h>
- <unistd.h>

3.3 Socket anlegen

int socket(int domain, int type, int protocol);

- domain: Protokollfamilie, die ich verwende
  - PF_INET: IP Protokoll
  - PF_INET6: IP version 6 Protokoll
  - PF_IPX: Novell Internet Protokoll
  - PF_BLUETOOTH: Bluetooth Protokoll
  - ...
- type: Art des Socket
  - SOCK_STREAM: verbindungsorientierter Bytestream
  - SOCK_DGRAM: verbindslos, Datagramme fester Größe
  - SOCK_SEQPACKET, verbindungsorientiert, Datagramme fester Größe
  - ...
- protocol: ergibt sich in der Regel aus den ersten beiden Parameters, wenn mehrere Protokoll möglich sind, kann man es hier angeben, in der Regel 0
- Rückgabewert: Socket-Filedescriptor oder negativ bei Fehler

Typischer Aufruf:

```
int sfd;

if(0 > (sfd = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)))
{
  perror(‘‘Problem with socket()’’);
  exit(1);
}
```

3.4 Kommandos auf Sockets

int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);

- s: Socket-Filedescriptor
- level: üblich SOL_SOCKET, Manipulation auf Socket-Ebene, auf Protokollebene die Nummer des Protokolls

- optname: Name der Option, z.B. SO_REUSEADDR
- optval: Value, z.B. int optval = 1, Aufruf mit &optval
- optlen: Länge von optval, z.B. sizeof(optval)
- kann in <bits/socket.h> und <asm/socket.h> eingesehen werden
- Rückgabewert: -1 bei Fehler, 0 bei Erfolg

Typischer Aufruf:

```
int optval = 1;

if(0 > setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)))
{
  perror(‘‘Problem with setsockopt()’’);
  exit(1);
}
```

3.5 Adresse an Socket binden

int bind(ind sockfd, struct sockaddr *my_addr, socklen_t addrlen);

- sockfd: Socket-Filedescriptor
- my_addr: Adresse, an die der Port gebunden wird, folgende Angaben müssen gemacht werden:
  - sin_family: Protokolltyp, z.B AF_INET, AF_INET6,usw.
  - sin_addr.s_addr: Was empfange ich, z.B. IN_ADDR_ANY, IN_ADDR_BROADCAST, IN_ADDR_NONE, INADDR_LOOPBACK, mit htonl()
  - sin_port: der zugehörige Port, will ja erreichbar sein, mit htons()
- addrlen: Länge der Adresse
- Rückgabewert: -1 bei Fehler, 0 bei Erfolg

Typischer Aufruf:

```
struct sockaddr_in saddr;
short port = 6666;

memset((void *)&saddr, 0, sizeof(saddr));
saddr.sin_familiy = AF_INET;
saddr.sin_addr.s_addr = htonl(INADDR_ANY);
saddr.sin_port = htons(port);

if(0 > bind(sfd, (struct sockaddr *),&saddr, sizeof(saddr)))
{
  perror(‘‘Problem with bind()’’);
  exit(1);
}
```

3.6 Netzwerkadressen

uint32_t htonl(uint32_t hostlong);

Host to network long, bekommt Adresse in “little endian” und konvertiert nach “big endian”

Analog dazu: htons(), ntohl(), ntohs()

Typischer Aufruf: s. bind()

struct hostent *gethostbyname(const char *name);

Ermittelt die IP-Adresse zum angegebenen Hostname, NULL, wenn nicht bekannt.

Typischer Aufruf: s. sendto()

3.7 Senden und Empfangen

ssize_t sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);

- s: Socket File-Descriptor
- msg: Zeiger auf die zu sendende Nachricht
- len: Länge der Nachricht
- flags: Flags, z.B. MSG_DONTWAIT (nichtblockieren)
- to: Adresse des Empfängers
  - sin_family: Protokollfamilie, z.B. AF_INET
  - sin_port: der zugehörige Port, mit htons()
  - sin_addr.s_addr: der Host
- tolen: Länge der Adresse
- Rückgabewert: -1 bei Fehlern, sonst die Anzahl der gesendeten Zeichen

Typischer Aufruf

```
char *dummy = ‘‘Hallo Welt’’;
struct sockaddr raddr;
int port = 5555;
struct hostent *hostinfo = gethostbyname(auenland);

memset((void *)&raddr, 0, sizeof(raddr));
raddr.sin_family = AF_INET;
raddr.sin_port = htons(port);
memcpy(&raddr.sin_addr.s_addr, hostinfo->h_addr, hostinfo->h_length);

if(0 > sendto(sfd, dummy, sizeof(dummy), 0, (struct sockaddr *) & raddr,
   sizeof(raddr)))
{
  perror(‘‘Problem with sendto()’’);
  exit(1);
}
```

ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);

- s: Socket File-Descriptor
- buf: Puffer für die empfangene Nachricht
- len: Länge des Puffers
- flags: z.B. MSG_WAITALL
- from: Addresse des Empfangssockets, wie bei sendto()
- fromlen: Länge des Empfangssockets
- Rückgabewert: -1 bei Fehlern, sonst die Anzahl der empfangenen Zeichen

Typischer Aufruf:

```
char recvbuffer[4000];
int messagelenght = sizeof(raddr);

if(0 > recvfrom(sfd, recvbuffer, sizeof(recvbuffer), 0,
                (struct sockaddr *)&raddr, &length))
{
  perror(‘‘Problem with recvfrom()’’);
  exit(1);
}
```

3.8 Socket schließen

int close(int fd);

- fd: Filedescriptor
- Rückgabewert: -1 bei

Typischer Aufruf

```
if(0 > close(sfd))
{
  perror(‘‘Problem with close()’’);
  exit(1);
}
```

4 Signalhandler

4.1 Header

- <signal.h>

4.2 Auf Signale hören

sighandler_t signal(int signum, sighandler_t handler);

- signum: Nummer des Signals
- handler: Zeiger auf Funktion
- typedef void (*sighandler_t)(int) , d.h. Funktion mit Parameter int und Rückgabetyp void, Name der Funktion ist Zeiger auf die Funktion

Typischer Aufruf:

```
void sighandler(int sig)
{
  printf(‘‘Signal %d caught!\n’’, sig);

  //Aufräumarbeiten

  exit(0);
}

//im Programm
int main(int argc, char **argv)
{
  if(0 > signal(sighandler))
  {
    perror(‘‘Problem with signal()’’);
    exit(1);
  }

  //...
}
```

5 Kindprozesse

5.1 Header

- <unidst.h>
- <sys/types.h>

pid_t fork();

- Rückgabewert: 0 bei Kind, Kind-PID bei Vater, im Fehlerfall -1

5.2 Kindprozesse erzeugen

Typischer Aufruf:

```
int childpid;

if(0 > (childpid = fork()))
{
  perror(‘‘Problem with fork()’’);
  exit(1);
}

if(childpid)
{
  //alles, was der Vater macht
}
else
{
  //alles, was das Kind macht
}
```

6 Misc

6.1 Header

- <string.h> für memset()
- <sys/types.h> für getpid() und waitpid()
- <unistd.h> für getpid()
- <wait.h> für waitpid()

6.2 Speicherbereich initialisieren

void *memset(void *s, int c, size_t n);

- s: Zeiger auf Speicherbereich
- c: Zeichen, mit dem gesetzt werden soll
- n: Anzahl der zu setzenden Bytes
- Rückgabewert: Zeiger auf s

Typischer Aufruf: s. bind()

6.3 PID ermitteln

pid_t getpid();

- Rückgabewert: eigene PID

Typischer Aufruf:

```
pid_t pid;

if(0 > (pid = getpid()) )
{
  perror(‘‘Problem with getpid()’’);
  exit(1);
}
```

6.4 Auf Terminierung von Kindern warten

pid_t waitpid(pid_t pid, int *status, int options);

- pid: pid auf die gewartet werden soll, -1 meint alle Kindprozesse
- status: Statusinformationen, können hinterher ausgelesen werden
- options: WNOHANG (nonblocking), WUNTRACED (stopped, not traced)
- Rückgabewert: > 0 ist PID des beendeten Kindprozesses, 0 bei Verwendung von WNOHANG, -1 bei Fehler

Typischer Aufruf:

```
if(0 > waitpid(childpid, NULL, 0))
{
  perror(‘‘Problem with waitpid()’’);
  exit(1);
}
```