

System-Programmierung

10: POSIX IPC

CC BY-SA, Thomas Amberg, FHNW
(Soweit nicht anders vermerkt)

Slides: tmb.gr/syspr-10

Überblick

Heute geht's um *POSIX* Interprozesskommunikation*.

Datenübertragung mittels Message Queues.

Synchronisation mit Semaphoren.

Und Shared Memory.

*IEEE Standard für portable Anwendungen.

POSIX Message Queues

Eine *Message Queue* erlaubt es, Messages von einem Prozess an einen anderen zu übertragen.

Jede Leseoperation liest eine ganze *Message*, wie sie vom schreibenden Prozess geschrieben wurde.

POSIX Messages haben neben der Payload auch eine *Priorität* und "high priority" Messages können in der Queue nach vorne rücken.

Message Queue öffnen mit *mq_open()*

Message Queue mit Name *name*, Flags *oflag* öffnen:

```
mqd_t mq_open(const char *name, int oflag,  
              mode_t mode, // diese 2 Argumente braucht es  
              struct mq_attr *attr); // nur, falls O_CREAT
```

Wobei *oflag* einen der folgenden Werte haben muss:

O_RDONLY, O_WRONLY, O_RDWR

Dieser kann verodert werden mit folgenden Flags:

O_CLOEXEC, O_CREAT (und O_EXCL), O_NONBLOCK

Message Queue Attribute in *mq_attr*

Die Calls *mq_open()*, *mq_getattr()* und *mq_setattr()* nutzen *struct mq_attr* für Message Queue Attribute:

```
struct mq_attr {  
    long mq_flags; // Ignoriert bei mq_open()  
    long mq_maxmsg; // Max. Anzahl Messages  
    long mq_msgsize; // Message Grösse in Bytes  
    long mq_curmsgs; // Aktuelle Anz. Messages,  
};                // ignoriert bei mq_open()
```

Attribute setzen bei *mq_open()*

Default Attribute setzen mit *attr = NULL*.

Oder Attribute explizit setzen, z.B. mit:

```
struct mq_attr attr;  
attr.mq_maxmsg = 3; // ≤ HARD_MSGMAX  
attr.mq_msgsize = 1024;  
mqd_t mqd = mq_open("/mq", O_RDWR|O_CREAT,  
S_IRUSR|S_IWUSR, &attr);
```

Alle anderen Attribute in *attr* werden ignoriert.

Message Queue Tabellen im Kernel

MQDs/Proc.

0:	Info	
1:		
0:		
1:		
2:		

Open MQs/Sys.

0:	Flags	
23:	O_NON BLOCK	
41:		
42:		

MQ-Table/System

66:	MQ Name	MQ Attr.	...
98:	/mq	Max N: 10	Size: 128
	Messages		
99:			

Attribute lesen mit *mq_getattr()*

Attribute *attr* der Message Queue *mqd* auslesen:

```
int mq_getattr( // 0 oder -1, errno  
               mqd_t mqd, // Message Queue Deskriptor  
               struct mq_attr *attr);
```

Der Wert *attr.mq_curmsgs* enthält die aktuelle Anzahl Messages in der Message Queue.

Message senden mit *mq_send()*

Message *msg* senden an Message Queue *mqd*:

```
int mq_send( // 0 oder -1, errno
             mqd_t mqd, // Message Queue Deskriptor
             const char *msg, // Message Inhalt
             size_t msg_len, //  $0 \leq \text{msg\_len} \leq \text{mq\_msgsize}$ 
             unsigned int msg_prio); //  $0 \leq \text{msg\_prio}$ 
```

Messages mit hoher Priorität springen in der Message Queue nach vorne, d.h. Sie werden eher empfangen.

Message empfangen mit *mq_receive()*

Message *msg* empfangen aus Message Queue *mqd*:

```
ssize_t mq_receive( // # Bytes oder -1, errno  
    mqd_t mqd, // Message Queue Deskriptor  
    char *msg, // Zeiger auf Buffer für Message  
    size_t msg_len, // mq_getattr() => mq_msgsize  
    unsigned int *msg_prio); // gibt Prio. raus
```

mq_receive() blockiert, falls keine Message verfügbar.

Message Queue schliessen mit *mq_close()*

Message Queue *mqd* schliessen:

```
int mq_close( // 0 oder -1, errno  
             mqd_t mqd); // Message Queue Deskriptor
```

mq_close() gibt den Deskriptor frei, löscht aber die Message Queue nicht, wie bei File Deskriptoren.

Beim Beenden des Prozesses und wenn *exec()* aufgerufen wird, wird *mq_close()* automatisch ausgeführt.

Message Queue löschen mit *mq_unlink()*

Message Queue Tabelleneintrag von *name* löschen:

```
int mq_unlink( // 0 oder -1, errno  
               const char *name); // Message Queue Name
```

Sobald keine Message Queue Deskriptoren mehr auf die Message Queue *name* zeigen, wird sie gelöscht.

Hands-on, 15': Message Queues

Lesen Sie die folgenden [TLPI] Beispiel Programme:

`pmsg_create.c`, `pmsg_getattr.c`, `pmsg_unlink.c`,
`pmsg_send.c` und `pmsg_receive.c`

Testen Sie eine Message Queue mit den Kommandos:

```
$ ./pmsg_create -cx /my_mq
$ ./pmsg_send /my_mq "my msg a" 0 # Prio. 0
$ ./pmsg_send /my_mq "my msg b" 1 # >0 => Skip
$ ./pmsg_receive /my_mq # Blockierend
$ ./pmsg_unlink /my_mq
```

Message Queue Verwaltung in Linux

Linux implementiert POSIX Message Queues als Files in einem virtuellen Filesystem, das *mount*-bar ist:

```
$ mkdir /dev/mqueue  
$ sudo mount -t mqueue none /dev/mqueue  
$ exit
```

Message Queues mit *ls* auflisten:

```
$ ls -ld /dev/mqueue  
$ cat /dev/mqueue/my_mq
```

POSIX Semaphore

Semaphore erlauben es mehreren Prozessen, ihre Aktionen zu synchronisieren, mit "Kernel-Variablen".

Ein *Semaphor* ist eine Zahl deren Wert nicht unter 0 fallen kann. Beim Dekrementieren eines Semaphors das 0 ist, wird der Aufrufer vom Kernel blockiert.

Sobald ein anderer Prozess das Semaphor wieder erhöht, kann der blockierte Prozess weiterlaufen.

Named Semaphore

Benannte (named) Semaphore haben einen Namen, mit `sem_open()` können zwei beliebige Prozesse dasselbe Semaphore gemeinsam verwenden.

POSIX IPC Namen beginnen mit einem '/', gefolgt von `('a'-'z'|'_'*)`, für **Semaphore** ist `NAME_MAX` bzw. 255 minus 4 Zeichen das Limit, weil das System den Präfix `"sem."` davor hängt.

Semaphor öffnen mit *sem_open()*

Named Semaphor öffnen mit *sem_open()*:

```
sem_t *sem_open( // bzw. SEM_FAILED bei Error  
    const char *name, // z.B. "/my_sem"  
    int oflag, // oflag = O_CREAT ( | O_EXCL)  
    mode_t mode, // wie bei File, falls O_CREAT  
    unsigned int value); // > 0, falls O_CREAT
```

```
sem_t *sem_open( // bzw. SEM_FAILED bei Error  
    const char *name, int oflag); // oflag = 0
```

Auf Semaphor warten mit *sem_wait()*

Semaphor *sem* um 1 reduzieren mit *sem_wait()*:

```
int sem_wait(sem_t *sem); // blockierend
int sem_trywait(sem_t *sem); // non-blocking
int sem_timedwait(sem_t *sem, // mit Timeout
    const struct timespec *abs_timeout);

struct timespec {
    time_t tv_sec; // Sekunden
    long tv_nsec; // Nanosekunden
};
```

Semaphor erhöhen mit *sem_post()*

Semaphor *sem* um 1 erhöhen mit *sem_post()*:

```
int sem_post( // 0 oder -1, errno  
              sem_t *sem); // Semaphor
```

Falls das Semaphor dadurch > 0 wird, und bereits ein anderer Prozess am Warten ist, wird dieser geweckt.

Wert eines Semaphors auslesen

Wert des Semaphors *sem* auslesen in *value* rein:

```
int sem_getvalue( // 0 oder -1, errno  
    sem_t *sem, // Semaphor  
    int *value);
```

Falls N andere Prozesse mit *sem_wait()* am Warten sind, liefert Linux 0, andere Implementierungen $-N$.

Semaphor schliessen und löschen

Semaphor *sem* schliessen mit *sem_close()*:

```
int sem_close( // 0 oder -1, errno  
              sem_t *sem); // Semaphor
```

Semaphor löschen mit *sem_unlink()*:

```
int sem_unlink( // 0 oder -1, errno  
               const char *name);
```

Beide mit *-pthread* kompilieren.

Hands-on, 15': Semaphore

Lesen Sie die folgenden [TLPI] Beispiel Programme:

`psem_create.c`, `psem_wait.c`, `psem_getvalue.c`,
`psem_post.c` und `psem_unlink.c`

Testen Sie ein Semaphor, z.B. mit den Kommandos:

```
$ ./psem_create -c /my_sem 600 3
```

```
$ ./psem_wait /my_sem &
```

```
$ ./psem_getvalue /my_sem
```

```
$ ./psem_post /my_sem
```

```
$ ./psem_unlink /my_sem
```

Unbenannte Semaphore

Unbenannte (unnamed) Semaphore befinden sich an einer vereinbarten Speicherstelle. Sie können von Prozessen mit Shared Memory oder von Threads, via Heap oder globalen Speicher, geteilt werden.

Dazu wird vom Prozess eine Variable vom Typ *sem_t* alloziert, mit *sem_init()* initialisiert und zum Schluss mit *sem_destroy()* gelöscht. Der Rest ist wie vorher.

Semaphor initialisieren mit *sem_init()*

Semaphor *sem* initialisieren mit *sem_init()*:

```
int sem_init( // 0 oder -1, errno
             sem_t *sem, // Semaphor
             int pshared, // 0: Threads, sonst Shared Mem.
             unsigned int value); // Semaphor-Initialwert
```

Diese Funktion ist nur für *unnamed* Semaphore, das Resultat *sem* kann aber "normal" mit *sem_getvalue()*, *sem_wait()* und *sem_post()* verwendet werden.

Semaphor löschen mit *sem_destroy()*

Semaphor *sem* löschen mit *sem_destroy()*:

```
int sem_destroy( // 0 oder -1, errno  
    sem_t *sem); // Semaphor
```

Diese Funktion ist speziell für *unnamed* Semaphore, dafür braucht es dann keinen Aufruf von *sem_close()* oder *sem_unlink()* weil es keinen Deskriptor gibt.

Named vs. unnamed Semaphore

Unnamed Semaphore können zwischen Threads im selben Prozess verwendet werden, ohne einen Namen.

Zudem können unnamed Semaphore vom Parent zu einem Child Prozess "vererbt" werden, mit *fork()*.

Die Speicherverwaltung für unnamed Semaphore ist manchmal einfacher als die Verwaltung von Namen, das Semaphor kann Teil z.B. eines Struct Typs sein.

POSIX Shared Memory

Shared Memory ist gemeinsam genutzter Speicher, auf den mehrere Prozesse gleichzeitig Zugriff haben.

Ein *POSIX Shared Memory Objekt* erlaubt Prozessen Speicher zu teilen, ohne ein Disk File zu erstellen.

Shared Memory ist für alle Prozesse sichtbar, die sich den Speicher teilen, das Lesen ist nicht destruktiv.

Shared Memory Objekt kreieren

Shared Memory Objekt kreieren mit *shm_open()*:

```
int shm_open( // File Deskriptor od. -1, errno
    const char *name, // POSIX Name
    int oflag, // O_RDWR oder O_RDONLY, |...
    mode_t mode); // wie bei File open()
```

Der "File" Deskriptor kann normal verwendet werden, insbesondere auch mit *mmap()* und *ftruncate()*.

Grösse setzen mit *ftruncate()*

Shared Memory Objekt Grösse setzen mit *ftruncate()*:

```
int ftruncate(int fd, off_t length);
```

Nach dem Erzeugen mit *shm_open()* hat das Shared Memory Objekt "File" die Grösse 0.

Shared Memory Objekt mappen

Shared Memory Objekt in den Speicher mappen:

```
void *mmap( // Speicheradresse oder MAP_FAILED
            void *addr, // NULL => Kernel-alloziert
            size_t length, // Grösse
            int prot, // z.B. PROT_READ|PROT_WRITE
            int flags, // z.B. MAP_SHARED
            int fd, // Shared Memory File Deskriptor
            off_t offset); // z.B. 0
```

Shared Memory schreiben

Nach dem Öffnen und Mappen des Shared Memory Objekts kann man *addr* direkt schreiben, z.B.:

```
char *buf = "hello";  
int fd = shm_open(name, O_RDWR, 0);  
size_t len = sizeof(buf) * sizeof(buf[0]);  
ftruncate(fd, len);  
void *addr = mmap(NULL, len,  
    PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);  
memcpy(addr, buf, len);
```

Shared Memory lesen

Nach dem Öffnen und Mappen des Shared Memory Objekts kann man direkt von *addr* lesen, z.B.:

```
int fd = shm_open(name, O_RDWR, 0);
struct stat sb;
fstat(fd, &sb); // (Shared Memory) File Stats
int len = sb.st_size; // File Grösse
char *addr = mmap(NULL, len,
    PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
write(STDOUT_FILENO, addr, len); // liest a. 32
```


Shared Memory Objekt löschen

Shared Memory Objekt löschen mit *shm_unlink()*:

```
int shm_unlink( // 0 oder -1, errno  
               const char *name);
```

Entfernt den Namen. Das Objekt selbst besteht weiter, bis alle Prozesse es mit *munmap()* freigegeben haben.

Das Objekt *name* kann nicht mehr mit *shm_open()* geöffnet werden; bloss neu erzeugt, mit *O_CREAT*.

Hands-on, 15': Shared Memory

Lesen Sie die folgenden [TLPI] Beispiel Programme:
`pshm_create.c`, `pshm_write.c`, `pshm_read.c` und
`pshm_unlink.c`

Testen Sie Shared Memory mit den Kommandos:

```
$ ./pshm_create -c /my_shm 0
$ ls -l /dev/shm
$ ./pshm_write /my_shm "hello"
$ ./pshm_read /my_shm
$ ./pshm_unlink /my_shm
```

Selbststudium, 3h: Message Queues

Zur Vertiefung der heutigen Lektion, lesen Sie im Buch [\[TLPI\]](#) *Chapter 52: POSIX Message Queues*.

(Das [PDF des Kapitels 52](#) ist verfügbar als Teil der offiziellen "Downloadable samples from the book".)

Feedback oder Fragen?

Gerne im Slack <https://fhnw-syspr.slack.com/>

Oder per Email an thomas.amberg@fhnw.ch

Danke für Ihre Zeit.