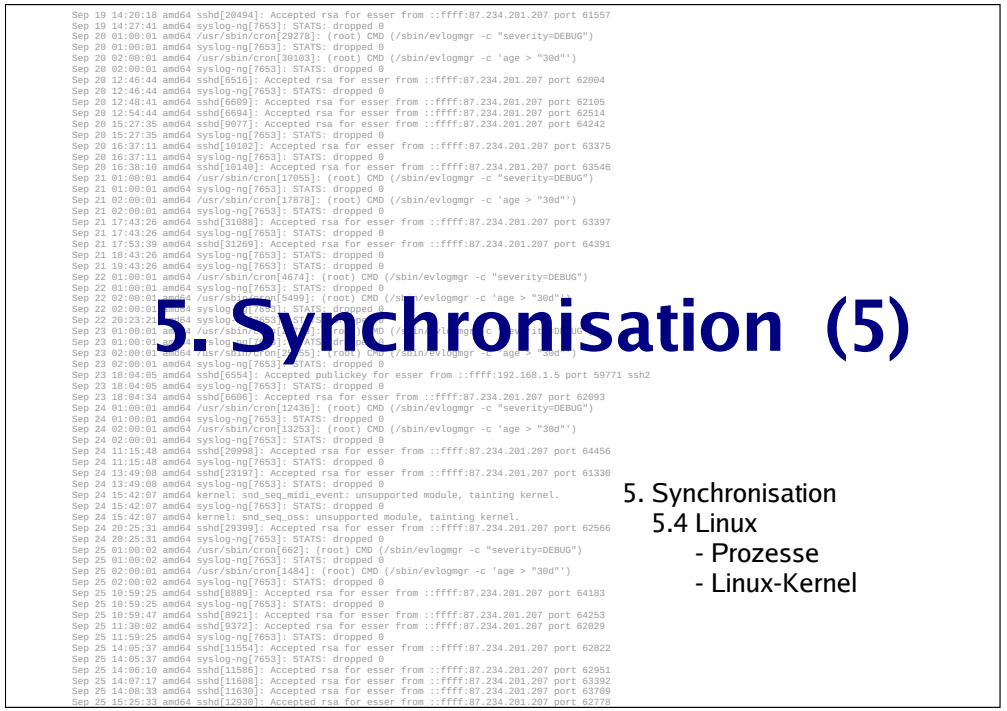
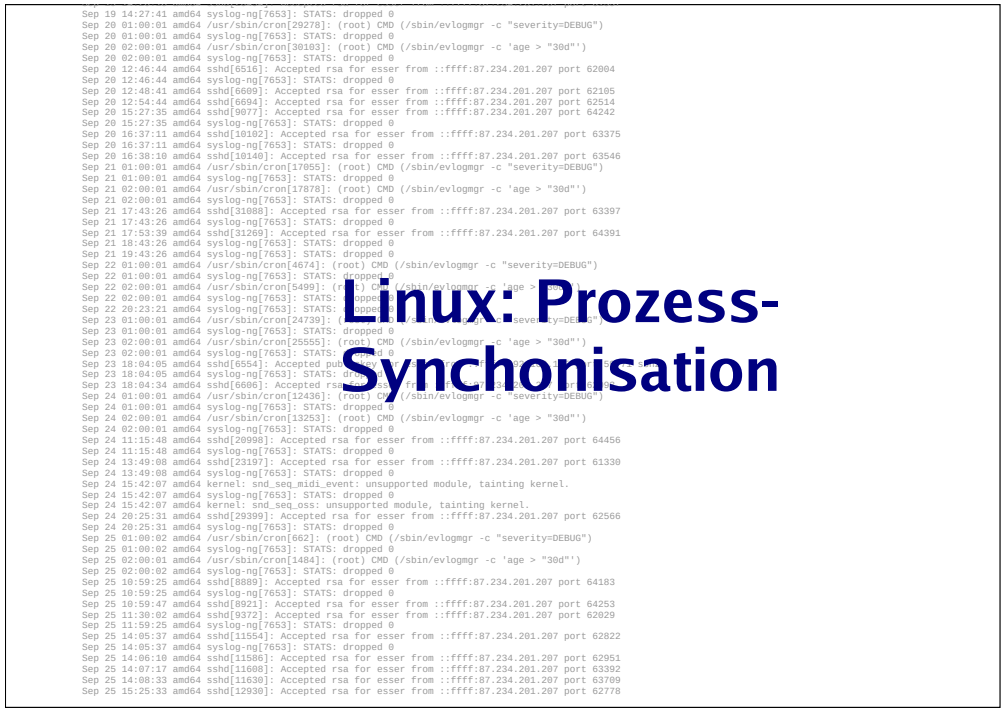


5. Synchronisation (5)

5. Synchronisation
   5.4 Linux
   - Prozesse
   - Linux-Kernel



Linux: Prozess-Synchonisation

Benannte POSIX-Semaphore (1)

Benutzung wie bei Threads, aber durch Vergabe eines (system-einheitlichen) Namens auf Prozesse ausdehnbar

```
#include <semaphore.h>
posix_sem = sem_open("/MeinSemaphor", O_CREAT,
                     0644, POSIX_UNLOCKED);
```

➞ erzeugt Eintrag in */dev/shm*:

```
$ ls -l /dev/shm/
-rw-r----- 1 esser users  16 2006-12-05 15:46 sem.MeinSemaphor
```


Benannte POSIX-Semaphore (2)

Zum Kennenlernen: 4 einfache Testprogramme:

- *named-sem-init.c* — Semaphor initialisieren benutzt *sem_open*, *sem_init*
- *named-sem-query.c* — Semaphor-Wert abfragen *sem_open*, *sem_getvalue*
- *named-sem-wait.c* — Semaphor erniedrigen (Wait-Operation), *sem_open*, *sem_wait*
- *named-sem-signal.c* — Semaphor erhöhen (Signal-Operation), *sem_open*, *sem_post*

Benannte POSIX-Semaphore (3)

```
/* named-sem-init.c */
#include <semaphore.h>
#include <asm/fcntl.h>
#define POSIX_LOCKED 0
#define POSIX_UNLOCKED 1

sem_t *posix_sem;

main () {
  posix_sem = sem_open("/MeinSemaphor", O_CREAT, 0644, POSIX_UNLOCKED);
  sem_init(posix_sem, 0, 5);     /* init: 5 */
}
```

```
$ ./named-sem-init
$ ls -l /dev/shm/
-rw-r-----  1 esser users    16 2006-12-05 15:46 sem.MeinSemaphor
$ hexdump /dev/shm/sem.MeinSemaphor
0000000 0005 0000 0000 0000 0000 0000 0000 0000
0000010
```


Benannte POSIX-Semaphore (4)

```
/* named-sem-query.c */
#include <semaphore.h>
#include <asm/fcntl.h>
#define POSIX_LOCKED 0
#define POSIX_UNLOCKED 1

sem_t *posix_sem;

main () {
  int ret;
  posix_sem = sem_open("/MeinSemaphor", O_CREAT, 0644, POSIX_UNLOCKED);
  sem_getvalue(posix_sem, &ret);
  printf ("Semaphor-Wert %d \n", ret);
}
```

```
$ ./named-sem-query
Semaphor-Wert 5
```


Benannte POSIX-Semaphore (5)

Signal-Operation

```
/* named-sem-signal.c */
#include <semaphore.h>
#include <asm/fcntl.h>

#define POSIX_LOCKED 0
#define POSIX_UNLOCKED 1

sem_t *posix_sem;

main () {
  posix_sem = sem_open("/MeinSemaphor", O_CREAT, 0644, POSIX_UNLOCKED);
  sem_post(posix_sem);
}
```

```
$ ./named-sem-query
Semaphor-Wert 5
$ ./named-sem-signal
$ ./named-sem-query
Semaphor-Wert 6
```


Benannte POSIX-Semaphore (6)

Wait-Operation

```
/* named-sem-wait.c */
#include <semaphore.h>
#include <asm/fcntl.h>

#define POSIX_LOCKED 0
#define POSIX_UNLOCKED 1

sem_t *posix_sem;

main () {
  posix_sem = sem_open("/MeinSemaphor", O_CREAT, 0644, POSIX_UNLOCKED);
  sem_wait(posix_sem);
}
```

```
$ ./named-sem-query
Semaphor-Wert 2
$ ./named-sem-wait
$ ./named-sem-query
Semaphor-Wert 1
$ ./named-sem-wait
```

```
$ ./named-sem-query
Semaphor-Wert 0
$ ./named-sem-wait
```

hier blockiert der Prozess, bis der Semaphor erhöht wird

Mutex für Prozesse

- benannten POSIX-Semaphor verwenden (Erinnerung: binärer Semaphor = Mutex)
- also:

```
posix_sem = sem_open("/mutex", O_CREAT,
                     0644, POSIX_UNLOCKED);
sem_init(posix_sem, 0, 1);      /* 1: Mutex */
```


System-V-IPC-Semaphore (1)

- Alternative zu benannten Posix-Semaphoren: System-V-IPC-Semaphore
- System-V-IPC: Methoden für die Inter-Prozess-Kommunikation (IPC) (mehr dazu: Kap. 6, IPC)
- Etwas komplexer:
  - Semaphor-Set (kann mehrere Semaphore enthalten)
  - private Semaphore (nur für Prozess und Kinder)
  - Public-Semaphore (mit Namen)

System-V-IPC-Semaphore (2)

- Semaphor erzeugen: *semget()*
- Semaphor initialisieren: *semctl()*
- Semaphor verwenden: *semop()*
  - erlaubt u. a. Signal- und Wait-Operationen:

```
struct sembuf                    /* in <sys/sem.h> */
{
  unsigned short int sem_num;    /* semaphore number */
  short int sem_op;              /* semaphore operation */
  short int sem_flg;             /* operation flag */
};
```

sem_op: -1 = wait, 1 = signal
sem_flg: IPC_NOWAIT➞ Fehler statt Warten

System-V-IPC-Semaphore (3)

Producer-Consumer-Problem mit SysV-Semaphoren (1)

```
/*
 * sem-producer-consumer.c
 */

#include <stdio.h>       /* standard I/O routines.              */
#include <stdlib.h>      /* rand() and srand() functions        */
#include <unistd.h>      /* fork(), etc.                        */
#include <time.h>        /* nanosleep(), etc.                   */
#include <sys/types.h>   /* various type definitions.           */
#include <sys/ipc.h>     /* general SysV IPC structures         */
#include <sys/sem.h>     /* semaphore functions and structs.    */

#define NUM_LOOPS  20    /* number of loops to perform.         */

union semun { int val; struct semid_ds *buf; unsigned short  *array; };

int main(int argc, char* argv[]) {
    int sem_set_id;           /* ID of the semaphore set.       */
    union semun sem_val;      /* semaphore value, for semctl(). */
    int child_pid;            /* PID of our child process.      */
    int i;                    /* counter for loop operation.    */
    struct sembuf sem_op;     /* structure for semaphore ops.   */
    int rc;                   /* return value of system calls.  */
    struct timespec delay;    /* used for wasting time.         */
```

System-V-IPC-Semaphore (4)

Producer-Consumer-Problem mit SysV-Semaphoren (2)

```
/* create private sem. set with one sem. in it, access only to the owner. */
sem_set_id = semget(IPC_PRIVATE, 1, 0600);
if (sem_set_id == -1) { perror("main: semget"); exit(1); }
printf("semaphore set created, semaphore set id '%d'.\n", sem_set_id);

/* intialize the first (and single) semaphore in our set to '0'. */
sem_val.val = 0;
rc = semctl(sem_set_id, 0, SETVAL, sem_val);

/* fork-off a child process, and start a producer/consumer job. */
child_pid = fork();
switch (child_pid) {
    case -1: perror("fork"); exit(1);
    case 0:  /* child process: consumer */
        for (i=0; i<NUM_LOOPS; i++) {
            /* block on the semaphore, unless its value is non-negative. */
            sem_op.sem_num = 0;
            sem_op.sem_op = -1;             /* <- -1: count down */
            sem_op.sem_flg = 0;
            semop(sem_set_id, &sem_op, 1);  /*  wait (semaphore)  */
            printf("consumer: '%d'\n", i); fflush(stdout);
        }
        break;
```


System-V-IPC-Semaphore (5)

Producer-Consumer-Problem mit SysV-Semaphoren (3)

```
        default:   /* parent process: producer */
            for (i=0; i<NUM_LOOPS; i++) {
                printf("producer: '%d'\n", i); fflush(stdout);
                /* increase the value of the semaphore by 1. */
                sem_op.sem_num = 0;
                sem_op.sem_op = 1; /* <-  +1: count up */
                sem_op.sem_flg = 0;
                semop(sem_set_id, &sem_op, 1);   /*  signal (semaphore)  */
                /* pause execution for a bit, to allow the child process to run */
                /* and handle some requests. this is done about 25% of the time.*/
                if (rand() > 3*(RAND_MAX/4)) {
                    delay.tv_sec = 0;
                    delay.tv_nsec = 10;
                    nanosleep(&delay, NULL);
                }
            }
            break;
    }

    return 0;
}
```

Quelle: http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html#semaphores

System-V-IPC-Semaphore (6)

Variante mit zwei getrennten Programmen:

- gemeinsamer Key erlaubt Zugriff auf (gleichen) Semaphor
- Key erzeugen mit *ftok()* („filename to key“):

```
key_t semkey = ftok("/tmp", 'a');
```

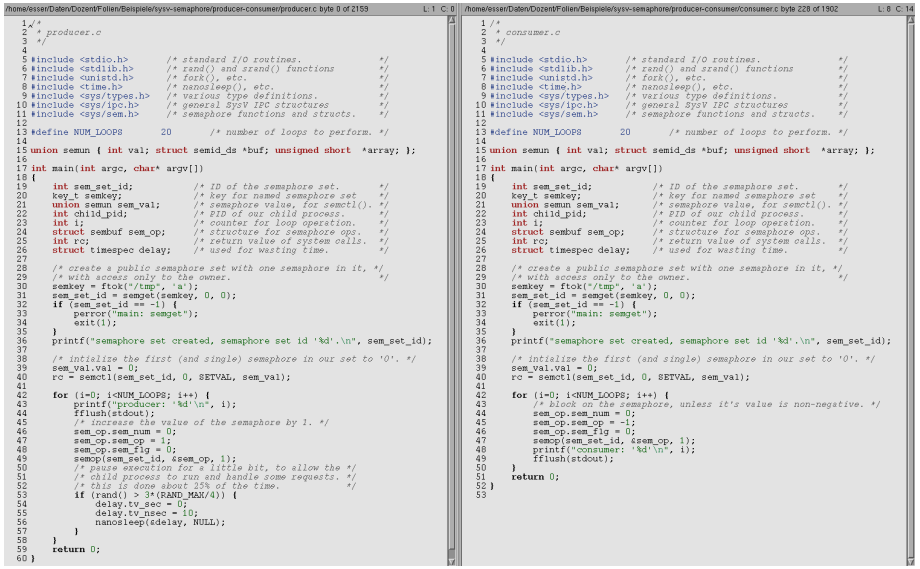
- *semget()*-Aufruf anpassen: Aus

```
/* privates Semaphor-Set erzeugen */
sem_set_id = semget(IPC_PRIVATE, 1, 0600);
```

wird

```
/* öffentliches Semaphor-Set erzeugen */
semkey = ftok("/tmp", 'a');
sem_set_id = semget(semkey, 1, 0);
```


System-V-IPC-Semaphore (7)



producer.c:

```
/*
 * producer.c
 */

#include <stdio.h>       /* standard I/O routines.             */
#include <stdlib.h>      /* rand() and srand() functions       */
#include <unistd.h>      /* fork(), etc.                       */
#include <time.h>        /* nanosleep(), etc.                  */
#include <sys/types.h>   /* various type definitions.          */
#include <sys/ipc.h>     /* general SysV IPC structures        */
#include <sys/sem.h>     /* semaphore functions and structs.   */

#define NUM_LOOPS       20      /* number of loops to perform. */

union semun { int val; struct semid_ds *buf; unsigned short  *array; };

int main(int argc, char* argv[])
{
    int sem_set_id;           /* ID of the semaphore set.          */
    key_t semkey;             /* key for named semaphore set       */
    union semun sem_val;      /* semaphore value, for semctl().    */
    int child_pid;            /* PID of our child process.         */
    int i;                    /* counter for loop operation.       */
    struct sembuf sem_op;     /* structure for semaphore ops.      */
    int rc;                   /* return value of system calls.     */
    struct timespec delay;    /* used for wasting time.            */

    /* create a public semaphore set with one semaphore in it, */
    /* with access only to the owner.                          */
    semkey = ftok("/tmp", 'a');
    sem_set_id = semget(semkey, 0, 0);
    if (sem_set_id == -1) {
        perror("main: semget");
        exit(1);
    }
    printf("semaphore set created, semaphore set id '%d'.\n", sem_set_id);

    /* intialize the first (and single) semaphore in our set to '0'. */
    sem_val.val = 0;
    rc = semctl(sem_set_id, 0, SETVAL, sem_val);

    for (i=0; i<NUM_LOOPS; i++) {
        printf("producer: '%d'\n", i);
        fflush(stdout);
        /* increase the value of the semaphore by 1. */
        sem_op.sem_num = 0;
        sem_op.sem_op = 1;
        sem_op.sem_flg = 0;
        semop(sem_set_id, &sem_op, 1);
        /* pause execution for a little bit, to allow the */
        /* child process to run and handle some requests. */
        /* this is done about 25% of the time.            */
        if (rand() > 3*(RAND_MAX/4)) {
            delay.tv_sec = 0;
            delay.tv_nsec = 10;
            nanosleep(&delay, NULL);
        }
    }
    return 0;
}
```

consumer.c:

```
/*
 * consumer.c
 */

#include <stdio.h>       /* standard I/O routines.             */
#include <stdlib.h>      /* rand() and srand() functions       */
#include <unistd.h>      /* fork(), etc.                       */
#include <time.h>        /* nanosleep(), etc.                  */
#include <sys/types.h>   /* various type definitions.          */
#include <sys/ipc.h>     /* general SysV IPC structures        */
#include <sys/sem.h>     /* semaphore functions and structs.   */

#define NUM_LOOPS       20      /* number of loops to perform. */

union semun { int val; struct semid_ds *buf; unsigned short  *array; };

int main(int argc, char* argv[])
{
    int sem_set_id;           /* ID of the semaphore set.          */
    key_t semkey;             /* key for named semaphore set       */
    union semun sem_val;      /* semaphore value, for semctl().    */
    int child_pid;            /* PID of our child process.         */
    int i;                    /* counter for loop operation.       */
    struct sembuf sem_op;     /* structure for semaphore ops.      */
    int rc;                   /* return value of system calls.     */
    struct timespec delay;    /* used for wasting time.            */

    /* create a public semaphore set with one semaphore in it, */
    /* with access only to the owner.                          */
    semkey = ftok("/tmp", 'a');
    sem_set_id = semget(semkey, 0, 0);
    if (sem_set_id == -1) {
        perror("main: semget");
        exit(1);
    }
    printf("semaphore set created, semaphore set id '%d'.\n", sem_set_id);

    /* intialize the first (and single) semaphore in our set to '0'. */
    sem_val.val = 0;
    rc = semctl(sem_set_id, 0, SETVAL, sem_val);

    for (i=0; i<NUM_LOOPS; i++) {
        /* block on the semaphore, unless it's value is non-negative. */
        sem_op.sem_num = 0;
        sem_op.sem_op = -1;
        sem_op.sem_flg = 0;
        semop(sem_set_id, &sem_op, 1);
        printf("consumer: '%d'\n", i);
        fflush(stdout);
    }
    return 0;
}
```

System-V-IPC-Semaphore (8)

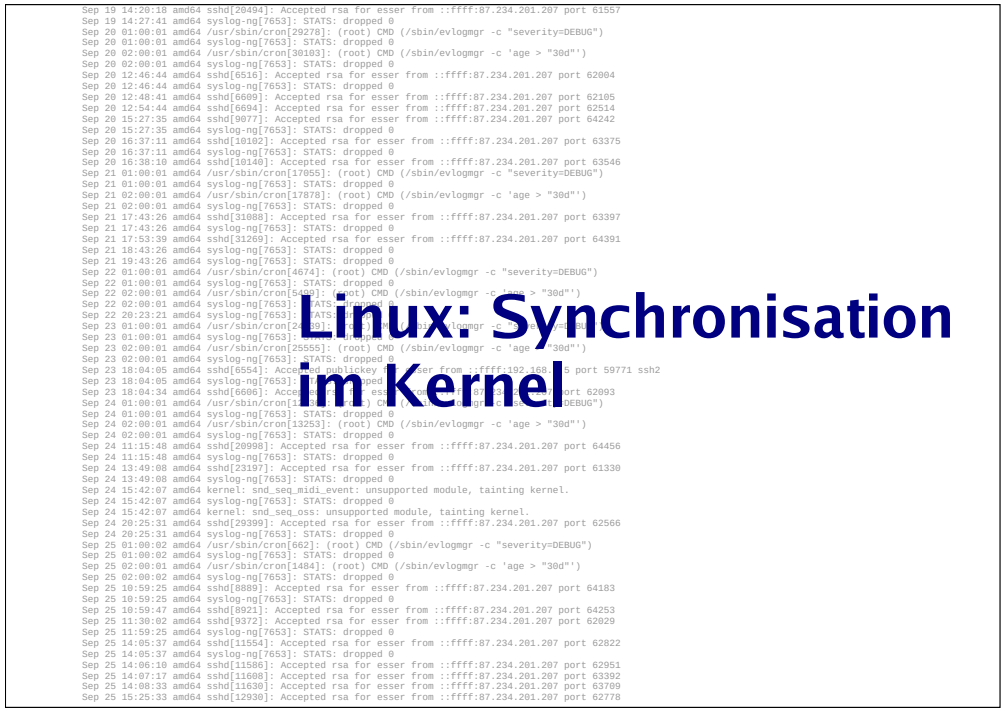
Terminal 1

```
$ gcc -o consumer consumer.c
$ gcc -o producer producer.c
$ ./consumer
semaphore set created,
semaphore set id '374964228'.
consumer: '0'
consumer: '1'
consumer: '2'
consumer: '3'
consumer: '4'
consumer: '5'
consumer: '6'
consumer: '7'
consumer: '8'
consumer: '9'
consumer: '10'
consumer: '11'
consumer: '12'
consumer: '13'
consumer: '14'
consumer: '15'
consumer: '16'
consumer: '17'
consumer: '18'
consumer: '19'
$ _
```

Terminal 2

```
$ ./producer
semaphore set created,
semaphore set id '374964228'.
producer: '0'
producer: '1'
producer: '2'
producer: '3'
producer: '4'
producer: '5'
producer: '6'
producer: '7'
producer: '8'
producer: '9'
producer: '10'
producer: '11'
producer: '12'
producer: '13'
producer: '14'
producer: '15'
producer: '16'
producer: '17'
producer: '18'
producer: '19'
$ _
```

consumer.c und *producer.c* auf der Vorlesungs-Web-Seite verfügbar



Linux: Synchronisation im Kernel

Synchronisation im Linux-Kernel

- Atomare Operationen
  - auf Integer-Variablen (atomic_set, atomic_add, atomic_inc, ...)
  - Bit-Operationen auf Bitvektoren (set_bit, clear_bit, test_and_set, ...)
- Spin Locks / Reader-Writer Spin Locks
- Semaphore / Reader-Writer-Semaphore
- „Big Kernel Lock“

Atomare Integer-Operationen (1)

- Neuer Typ *atomic_t* (24 Bit Integer)
- Initialisierung: *atomic_t var =* ***ATOMIC_INIT****(0);*
- Wert setzen: ***atomic_set*** *(&var, wert);*
- Addieren: ***atomic_add*** *(wert, &var);*
- ++: ***atomic_inc*** *(&var);*
- Subtrahieren: ***atomic_sub*** *(wert, &var);*
- --: ***atomic_dec*** *(&var);*
- Auslesen: *int i =* ***atomic_read*** *(&var);*

Atomare Integer-Operationen (2)

- *res =* ***atomic_sub_and_test*** *(i, &var);*
  zieht atomar i von var ab.
  - Rückgabewert true, falls Ergebnis 0 ist;
  - Rückgabewert false, falls Ergebnis nicht 0
- *res =* ***atomic_dec_and_test*** *(&var);*
  res = ***atomic_inc_and_test*** *(&var);*
  führt atomar *var--;* bzw. *var++;* aus.
  - Rückgabewert true, falls Ergebnis 0 ist;
  - Rückgabewert false, falls Ergebnis nicht 0

Atomare Integer-Operationen (3)

- *res =* ***atomic_add_negative*** *(i, &var);*
  addiert atomar *i* zu *var*.
  - Rückgabewert true, falls Ergebnis negativ ist;
  - Rückgabewert false, falls Ergebnis ≥ 0 ist

Atomare Bit-Operationen (1)

- Einzelne Bits in Bitvektoren setzen
- Datentyp: beliebig, z. B.
  unsigned long bitvektor = 0;
  - nur über Pointer ansprechen
  - Anzahl der setz-/testbaren Bits hängt von Größe des verwendeten Datentyps ab
- ***set_bit*** *(i, &bitvektor);* *i*-tes Bit setzen
- ***clear_bit*** *(i, &bitvektor);* *i*-tes Bit löschen
- ***change_bit*** *(i, &bitvektor);* *i*-tes Bit kippen

Atomare Bit-Operationen (2)

- Test-and-Set-Operationen geben zusätzlich den vorherigen Wert des jeweiligen Bits zurück
  - *b =* ***test_and_set_bit*** *(i, &bitvektor);*
  - *b =* ***test_and_clear_bit*** *(i, &bitvektor);*
  - *b =* ***test_and_change_bit*** *(i, &bitvektor);*
- Einzelne Bits auslesen
  - *b =* ***test_bit*** *(i, &bitvektor);*
- Suchfunktionen
  - *pos =* ***find_first_bit*** *(&bitvektor, laenge);*
  - *pos =* ***find_first_zero_bit*** *(&bitvektor, laenge);*

Spin Locks (1)

- Lock mit Mutex-Funktion:
  Gegenseitiger Ausschluss
- Code, der ein Spin Lock anfordert und nicht erhält, läuft in Schleife weiter, bis das Lock verfügbar wird („spinning“)
- Typ: *spinlock_t*

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED

spin_lock (&xy_lock);
/* kritischer Abschnitt */
spin_unlock (&xy_lock);
```


Spin Locks (2)

- Da Spin Locks nicht schlafen, kann man sie in Interrupt-Handlern verwenden
- In dem Fall: zusätzlich Interrupts sperren:

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED
unsigned long flags;

spin_lock_irqsave (&xy_lock, flags);
/* kritischer Abschnitt */
spin_unlock_irqrestore (&xy_lock, flags);
```

(aktuelle Interrupts in *flags* sichern, dann sperren bzw. ursprünglichen Zustand wiederherstellen)

Spin Locks (3)

- Wenn zu Beginn alle Interrupts aktiviert sind, geht es auch einfacher:

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED

spin_lock_irq (&xy_lock);
/* kritischer Abschnitt */
spin_unlock_irq (&xy_lock);
```

  schaltet alle Interrupts aus bzw. wieder an
- Spin Locks sind nicht „rekursiv“,
  d.h.: es ist nicht möglich, das gleiche Spin Lock zweimal nacheinander anzufordern, etwa beim rekursiven Aufruf einer Funktion

Spin Locks (4)

- Um Blockieren zu vermeiden, ist Lock-Abfrage mit *spin_is_locked (&xy_lock);* möglich
- Locking-Versuch mit *spin_try_lock*:

```
if ( spin_try_lock (&xy_lock) ) {
  /* kritischer Abschnitt */
  spin_unlock (&xy_lock);
} else {
  /* durfte nicht in den kritischen Abschnitt */
}
```

- Beide Funktionen sollte man nicht verwenden: Entweder braucht man das Lock (und muss dann ggf. warten), oder man braucht es nicht...

Reader Writer Locks (1)

- Alternative zu normalen Locks, die mehrere Lesezugriffe zulässt – bei schreibendem Zugriff aber exklusiv (wie ein normales Lock) ist

```
rwlock_t xy_rwlock = RW_LOCK_UNLOCKED;
```

Lesender Code

```
read_lock (&xy_rwlock) ) {
  /* kritischer Abschnitt,
     read-only */
read_unlock (&xy_rwlock);
```

Schreibender Code

```
write_lock (&xy_rwlock) ) {
  /* kritischer Abschnitt,
     read & write */
write_unlock (&xy_rwlock);
```

- Nur bei klarer Trennung zwischen lesenden / schreibenden Programmteilen!

Reader Writer Locks (2)

-

	Es gibt schon einen Leser	Es gibt schon einen Schreiber	Noch keine Sperre
`read_lock(&lck)`	erfolgreich	schlägt fehl	erfolgreich
`write_lock(&lck)`	schlägt fehl	schlägt fehl	erfolgreich

- Auch hier Varianten für Interrupt-Behandlung:
  - read_lock_irq read_unlock_irq
  - read_lock_irqsave read_unlock_irqrestore
  - write_lock_irq write_unlock_irq
  - write_lock_irqsave write_unpock_irqrestore

Semaphore (1)

- Kernel-Semaphore sind „schlafende“ Locks
- Ist ein Semaphor schon gelockt, werden weitere Interessenten in eine Warteschlange eingereiht.
- Bei Freigabe eines Semaphors wird der erste wartende Thread in der Warteschlange geweckt
- Semaphore eignen sich für Sperren, die über einen längeren Zeitraum gehalten werden
  - keine Verschwendung von Rechenzeit

Semaphore (2)

- Semaphore sind nur im Prozess-Kontext einsetzbar, nicht in Interrupt-Handlern (Interrupt-Handler werden nicht vom Scheduler behandelt)
- Code, der einen Semaphor verwenden will, darf nicht bereits ein normales Lock besitzen (Semaphor-Zugriff kann dazu führen, dass der Thread sich schlafen legt.)
- Semaphore können auch mehr als einen Thread auf die Ressource zugreifen lassen

Semaphore (3)

Typ: *semaphore*

Statische Deklaration

```
static DECLARE_SEMAPHORE_GENERIC (name, count);
static DECLARE_MUTEX (name);            /* count=1 */
```

Dynamische Semaphor-Erzeugung

```
sema_init (&sem,count);
init_MUTEX (&sem);                       /* count=1 */
```

- Verwendung mit *up()* und *down()*

```
down (&sem);
/* kritischer Abschnitt */
up (&sem);
```


Semaphore (4)

- Varianten von *down()*
  - *down (&sem);*
    nicht unterbrechbarer Schlaf, falls Semaphor nicht verfügbar
  - *down_interruptible (&sem);*
    unterbrechbarer Schlaf, falls Sem. nicht verfügbar
  - *down_trylock (&sem);*
    versucht, den Semaphor zu erhalten – falls das nicht gelingt, kehrt die Funktion sofort mit False-Wert zurück

Semaphore (5)

- Beispiel für *down_trylock()*

```
/* Auszug aus /usr/src/linux/kernel/printk.c */

if (!down_trylock(&console_sem)) {
    console_locked = 1;
    /*
     * We own the drivers.  We can drop the spinlock and let
     * release_console_sem() print the text
     */
    spin_unlock_irqrestore(&logbuf_lock, flags);
    console_may_schedule = 0;
    release_console_sem();
    /* Funktion release_console_sem() führt up(&console_sem); aus */
} else {
    /*
     * Someone else owns the drivers.  We drop the spinlock, which
     * allows the semaphore holder to proceed and to call the
     * console drivers with the output which we just produced.
     */
    spin_unlock_irqrestore(&logbuf_lock, flags);
}
```


Reader-Writer-Semaphore (1)

- Analog zu Reader Writer Locks:
  Typ *rw_semaphore*, der spezielle Up- und Down-Operationen für Lese- und Schreibzugriff erlaubt
- Alle Reader-Writer-Semaphore sind Mutexe (Zähler ist bei Initialisierung immer 1)

Statische Deklaration

```
static DECLARE_RWSEM (name);
```

Dynamische Semaphor-Erzeugung

```
init_rwsem (&sem);
```

Reader-Writer-Semaphore (2)

```
static DECLARE_RWSEM (xy_rwsem);
```

Lesender Code

```
down_read (&xy_rwsem) ) {
  /* kritischer Abschnitt,
     read-only */
up_read (&xy_rwsem);
```

Schreibender Code

```
down_write (&xy_rwsem) ) {
  /* kritischer Abschnitt,
     lesen und schreiben */
up_write (&xy_rwsem);
```

Genau wie bei Reader Writer Locks:

	Es gibt schon einen Leser	Es gibt schon einen Schreiber	Noch keine Sperre
`down_read(&sem)`	erfolgreich	schlägt fehl	erfolgreich
`down_write(&sem)`	schlägt fehl	schlägt fehl	erfolgreich

„Big Kernel Lock“ (BKL) (1)

- Überbleibsel aus älteren Kernel-Versionen
- Globales Lock für den gesamten Kernel (das alle Code-Teile betrifft, die damit Datenzugriff schützen)

```
lock_kernel ();
/* kritischer Abschnitt */
unlock_kernel ();


if ( kernel_locked() ) {
   ...
}
```


„Big Kernel Lock“ (BKL) (2)

- BKL nur im Prozess-Kontext benutzbar (nicht in Interrupt-Routinen)
- Prozess, der das BKL hält, darf schlafen
  - Beim Schlafenlegen wird das BKL automatisch aufgegeben
  - Beim Aufwecken wird es wieder erworben
- BKL ist rekursiv: Prozess, der bereits das BKL hält, darf also erneut *lock_kernel()* ausführen
- Nicht benutzen!

Vorschau

Nächstes Mal:

Synchronisation von Prozessen und Threads unter Windows

(inkl. „Crashkurs“ zu Prozessen und Threads)