English Edition

5. Synchronization (5)

5. Synchronization
   5.4 Linux
      - Processes
      - Linux Kernel

Named POSIX Semaphores (1)

can be used like the variant for threads, but extends to processes by using a semaphore name that is unique on the system

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

➡ creates entry in */dev/shm*:

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


Linux: Process Synchonization

Named POSIX Semaphores (2)

as introduction: 4 simple test programs:

- *named-sem-init.c* — initialize semaphore uses *sem_open*, *sem_init*
- *named-sem-query.c* — query semaphore value *sem_open*, *sem_getvalue*
- *named-sem-wait.c* — decrement semaphore (Wait operation), *sem_open*, *sem_wait*
- *named-sem-signal.c* — increment semaphore (Signal operation), *sem_open*, *sem_post*

Named POSIX Semaphores (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("/MySemaphore", 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.MySemaphore
$ hexdump /dev/shm/sem.MySemaphore
0000000 0005 0000 0000 0000 0000 0000 0000 0000
0000010
```


Named POSIX Semaphores (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("/MySemaphore", O_CREAT, 0644, POSIX_UNLOCKED);
  sem_getvalue(posix_sem, &ret);
  printf ("Semaphore value %d \n", ret);
}
```

```
$ ./named-sem-query
Semaphore value 5
```


Named POSIX Semaphores (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("/MySemaphore", O_CREAT, 0644, POSIX_UNLOCKED);
  sem_post(posix_sem);
}
```

```
$ ./named-sem-query
Semaphore value 5
$ ./named-sem-signal
$ ./named-sem-query
Semaphore value 6
```


Named POSIX Semaphores (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("/MySemaphore", O_CREAT, 0644, POSIX_UNLOCKED);
  sem_wait(posix_sem);
}
```

```
$ ./named-sem-query
Semaphore value 2
$ ./named-sem-wait
$ ./named-sem-query
Semaphore value 1
$ ./named-sem-wait
```

```
$ ./named-sem-query
Semaphore value 0
$ ./named-sem-wait
```

here the process blocks until the semaphore is incremented

Mutex for Processes

- use a named POSIX semaphore
  (reminder: binary semaphore = mutex)
- so:

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


System V IPC Semaphores (1)

- Alternative to named Posix semaphores:
  System V IPC Semaphores
- System V IPC: Methods for Inter Process Communication (IPC)
  (more about this: Ch. 6, IPC)
- A little more complex:
  - semaphore sets (can contain several semaphores)
  - private semaphores (only for process and children)
  - public semaphores (with identifiers)

System V IPC Semaphores (2)

- create semaphore: *semget()*
- initialize semaphore: *semctl()*
- use semaphore: *semop()*
  - includes **signal** and **wait** operations:

```
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→ error instead of waiting

System V IPC Semaphores (3)

Producer Consumer Problem with SysV semaphores (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 Semaphores (4)

Producer Consumer Problem with SysV semaphores (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 Semaphores (5)

Producer Consumer Problem with SysV semaphores (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;
}
```

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

System V IPC Semaphores (6)

Variant with two separare programs:

- common key allows accessing the (same) semaphore
- create key with *ftok()* („filename to key"):

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

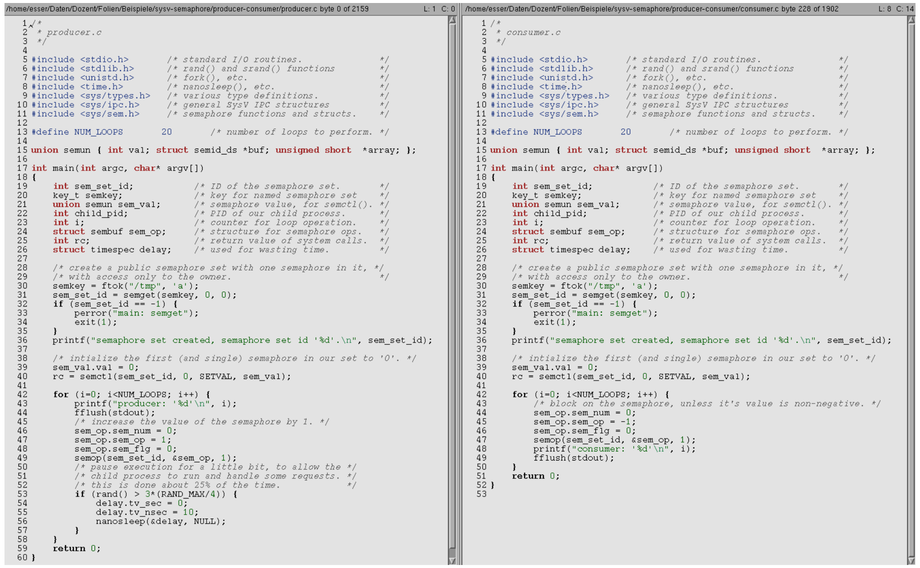
- change *semget()* call:

```
/* create private semaphore set */
sem_set_id = semget(IPC_PRIVATE, 1, 0600);
```

becomes

```
/* create public semaphore set  */
semkey = ftok("/tmp", 'a');
sem_set_id = semget(semkey, 1, 0);
```


System V IPC Semaphores (7)


/home/esser/Daten/Dozent/Folien/Beispiele/sysv-semaphore/producer-consumer/producer.c byte 0 of 2159   L: 1 C: 0

```
/*
 * 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;
}
```

/home/esser/Daten/Dozent/Folien/Beispiele/sysv-semaphore/producer-consumer/consumer.c byte 228 of 1902   L: 8 C: 14

```
/*
 * 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 Semaphores (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

Synchronization in the Linux Kernel

- atomic operations
  - on Integer variables (atomic_set, atomic_add, atomic_inc, ...)
  - bit operations on bit vectors (set_bit, clear_bit, test_and_set, ...)
- Spin Locks / Reader Writer Spin Locks
- Semaphores / Reader Writer Semaphores
- „Big Kernel Lock“

Linux: Synchronization in the Kernel

Atomic Integer Operations (1)

- new type *atomic_t* (24 bit integer)
- initialization: *atomic_t var =* ***ATOMIC_INIT****(0);*
- set value: ***atomic_set*** *(&var, wert);*
- add: ***atomic_add*** *(wert, &var);*
- increment: ***atomic_inc*** *(&var);*
- subtract: ***atomic_sub*** *(wert, &var);*
- decrement: ***atomic_dec*** *(&var);*
- read: *int i =* ***atomic_read*** *(&var);*

Atomic Integer Operations (2)

- *res = **atomic_sub_and_test** (i, &var);*
  atomically subtracts *i* from *var*
  - return value true, if result is 0;
  - return value false, if result is not 0
- *res = **atomic_dec_and_test** (&var);*
  *res = **atomic_inc_and_test** (&var);*
  atomically decrements or increments *var*
  - return value true, if result is 0;
  - return value false, if result is not 0

Atomic Integer Operations (3)

- *res = **atomic_add_negative** (i, &var);*
  atomically adds *i* to *var*.
  - return value true, if result is negative;
  - return value false, if result is ≥ 0

Atomic Bit Operations (1)

- set and clear single bits in bit vector
- datatype: arbitrary, e.g.
  unsigned long bitvector = 0;
  - access through pointer
  - number of settable / clearable bits depends on size of the used datatype
- ***set_bit** (i, &bitvector);* set *i*-th bit
- ***clear_bit** (i, &bitvector);* clear *i*-th bit
- ***change_bit** (i, &bitvector);* change *i*-th bit

Atomic Bit Operations (2)

- Test-and-Set operations return the previous value of the bit
  - *b = **test_and_set_bit** (i, &bitvector);*
  - *b = **test_and_clear_bit** (i, &bitvector);*
  - *b = **test_and_change_bit** (i, &bitvector);*
- read single bit
  - *b = **test_bit** (i, &bitvector);*
- search functions
  - *pos = **find_first_bit** (&bitvector, length);*
  - *pos = **find_first_zero_bit** (&bitvector, length);*

Spin Locks (1)

- Lock with Mutex functionality: mutual exclusion
- Code which requests a Spin Lock but is not successful, will spin in a loop until the lock becomes available
- Type: *spinlock_t*

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED

spin_lock (&xy_lock);
/* critical region */
spin_unlock (&xy_lock);
```


Spin Locks (2)

- since Spin Locks don't sleep they can be used inside interrupt handlers
- In that case: also disable interrupts:

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED
unsigned long flags;

spin_lock_irqsave (&xy_lock, flags);
/* critical region */
spin_unlock_irqrestore (&xy_lock, flags);
```

(save current interrupt settings in *flags*, then disable; restore original state)

Spin Locks (3)

- when all interrupts are on before aquiring the spin lock, there's a simpler method:

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED

spin_lock_irq (&xy_lock);
/* critical section */
spin_unlock_irq (&xy_lock);
```

disables / reenables all interrupts

- spin locks are not „recursive“, i.e.: it's impossible to aquire the same spin lock twice, e.g. when calling a function recursively

Spin Locks (4)

- to avoid blocking, it is possible to query the spin lock state with *spin_is_locked (&xy_lock);*
- locking attempt with *spin_try_lock*:

```
if ( spin_try_lock (&xy_lock) ) {
  /* critical region */
  spin_unlock (&xy_lock);
} else {
  /* was not allowed to enter the critical region */
}
```

- both functions should not be used: either you need the lock (and in that case will possibly have to wait), or you don't need it ...

Reader Writer Locks (1)

- alternative to normal locks, allowing several concurrent read accesses – but exclusive for write access (like a standard lock):

```
rwlock_t xy_rwlock = RW_LOCK_UNLOCKED;
```

reading code

```
read_lock (&xy_rwlock) ) {
  /* critical region,
     read-only */
read_unlock (&xy_rwlock);
```

writing code

```
write_lock (&xy_rwlock) ) {
  /* critical region,
     read & write */
write_unlock (&xy_rwlock);
```

- only use in case of strict separation of reading and writing code parts

Reader Writer Locks (2)

-

	there is already a reader	there is already a writer	no locks yet
read_lock(&lck)	successful	fails	successful
write_lock(&lck)	fails	fails	successful

- there are variants for disabling interrupts, too:
  - read_lock_irq read_unlock_irq
  - read_lock_irqsave read_unlock_irqrestore
  - write_lock_irq write_unlock_irq
  - write_lock_irqsave write_unpock_irqrestore

Semaphores (1)

- Kernel semaphores are „sleeping“ locks
- if a semaphore is already locked, new requesters are put into a queue.
- when freeing a semaphore, the first waiting thread of the queue will be awoken
- semaphores are useful for locks that shall be kept over a longer period of time
  - no waste of CPU time

Semaphores (2)

- semaphores can only be used in process context, not in interrupt handlers (the scheduler does not deal with interrupt handlers)
- code which wants to use a semaphore, must not hold a normal lock (semaphore access can cause the thread to be put to sleep)
- semaphores can let more than one thread access the resource

Semaphores (3)

Type: *semaphore*

Static declaration

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

dynamic semaphore creation

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

- use with *up()* and *down()*

```
down (&sem);
/* critical section */
up (&sem);
```


Semaphores (4)

- Variants of *down()*
  - *down (&sem);*
    non-interruptible sleep, if semaphore is not available
  - *down_interruptible (&sem);*
    interruptible sleep, if semaphore is not available
  - *down_trylock (&sem);*
    tries to aquire the semaphore – if that fails, this function will return with a false value

Semaphores (5)

- example for *down_trylock()*

```
/* taken from /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();
    /* function release_console_sem() calls up(&console_sem); */
} 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 Semaphores (1)

- similar to Reader Writer Locks:
  Type *rw_semaphore*, allowing special Up and Down operations for read and write access
- all Reader Writer Semaphores are mutexes (counter is always 1 at initialization)

static declaration

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

dynamical semaphore creation

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

Reader Writer Semaphores (2)

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

reading code

```
down_read (&xy_rwsem) ) {
  /* critical region,
     read-only */
up_read (&xy_rwsem);
```

writing code

```
down_write (&xy_rwsem) ) {
  /* critical region,
     read & write */
up_write (&xy_rwsem);
```

just like Reader Writer Locks:

	there is already a reader	there is already a writer	no locks yet
down_read(&sem)	successful	fails	successful
down_write(&sem)	fails	fails	successful

„Big Kernel Lock“ (BKL) (1)

- relict from older kernel versions
- global lock for the whole kernel (which affects all kernel regions that protect data access with it)

```
lock_kernel ();
/* critical region */
unlock_kernel ();


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


„Big Kernel Lock“ (BKL) (2)

- BKL can only be used in process context (not in interrupt routines)
- a process holding the BKL may sleep
  - when going to sleep, the BKL is automatically released
  - on wake-up it will be re-aquired
- BKL is recursive: a process which already holds the BKL may call *lock_kernel()* again
- <u>don't use it!</u>