Prof. Dr.-Ing. J. Quade

Inhalt

Schutz kritischer Abschnitte

Infos:

- Quade/Kunst: Linux-Treiber entwickeln. 3. Auflage, dpunkt-Verlag, 2006, S. 167-193.

Einführung

Übersicht

- Durch die Parallelität gibt es Konflikte beim Zugriff auf gemeinsame Betriebsmittel (z.B. auf Speicherbereiche) und bei der Synchronisation.

Bereits gesehen:

- Globale Variablen im Treiber
- Softirqs (Tasklets, Timer)
- Kernel-Threads (Workqueues, Event-Workqueue)

Einführung

Übersicht

- Bekannte Lösung:
  - Gegenseitiger Ausschluss (mutual exclusion)
  - Ausprägungen im Linux-Kernel:
    - Interruptsperre
    - Atomare Operationen
    - Semaphor/Mutex
    - Spinlocks
    - Memory Barriers

Interruptsperre

Interrupt- und Preemptionsperre

- Interruptsperre ist nur für Einprozessor-Systeme geeignet.
- Beide Sperren sollten generell vermieden werden.
- Funktionen:
  - `cli()`
  - `sti()`
  - `preempt_disable()`
  - `preempt_enable()`

Atomare Operationen

- Mit einer *atomaren Variable* kann atomar auf einen
  - Integerwert (32 Bit) oder einen
  - Bitwert zugegriffen werden.
- Der Zugriff auf die atomare Variable erfolgt über zugehörige Funktionen.
- Der Zugriff ist sowohl im Kernel-, Prozess- als auch im Interruptkontext möglich.
- Mit Kernel 2.6.19 gibt es zusätzlich einen `atomic64_t`.

Atomare Operationen

```
int atomic_read(atomic_t *v);

void atomic_set(atomic_t *v, int i);

void atomic_add_(int i, atomic_t *v)

void atomic_sub(int i, atomic_t *v)

void atomic_inc(atomic_t *v)

void atomic_dec(atomic_t *v)

int atomic_sub_and_test(int i, atomic_t
   *v)

int atomic_inc_and_test(atomic_t *v)

int atomic_dec_and_test(atomic_t *v)

int atomic_add_negative(int i, atomic_t
   *v)v
```

```
void set_bit(int nr, unsigned long *a)

void clear_bit(int nr, unsigned long *a)

void change_bit(int nr, unsigned long *a)

int test_and_set_bit(int nr, unsigned
   long *a)

int test_and_clear_bit(int nr, unsigned
   long *a)

int test_and_change_bit(int nr, unsigned
   long *a)

int test_bit(int nr, void *a)
```

Atomare Operationen

Kernel 2.6.19

```
int atomic_long_read(atomic_long_t *l);
void atomic_long_set(atomic_long_t *s, long i);
void atomic_long_add_(long i, atomic_long_t *l)
void atomic_long_sub(long i, atomic_long_t *l)
void atomic_long_inc(atomic_long_t *l)
void atomic_long_dec(atomic_long_t *l)
```

Semaphor

- Zum Schutz komplexer Datenstrukturen.
- Der Zugriff ist nur im Prozesskontext möglich.

```
• DECLARE_MUTEX( mutex );
• init_MUTEX( &mutex );
• sema_init( &mutx, n );
• down( &mutex );
• down_interruptible( &mutex );
• down_trylock( &mutex );
• up( &mutex );
```

Semaphor

Semaphor

```
while( down_interruptible(&Semaphor) == –EINTR ) {
                                  Betreten des kritischen Abschnitts.
        ... // Behandlung des Signals
}
... // Modifikation der Daten
                                  Der kritische Abschnitt selbst.

up( &Semaphor );Verlassen des kritischen Abschnitts.
```

Semaphor

```
#include <asm/semaphore.h>
...
static struct semaphore my_mutex;
...
    init_MUTEX( &my_mutex );
    ...
    if( down_interruptible( &my_mutex ) ) {
        return -ERESTART; // durch Signal unterbrochen
    }
    ... // Hier ist der kritische Abschnitt
    up( &my_mutex );
    ...
```

Ausprägungen

- „Normale“ Semaphore
- Schreib-/Lese-Semaphore
  - `down_read`
  - `down_read_trylock`
  - `down_write`
  - `down_write_trylock`
  - `downgrade_write`
  - `up_read`
  - `up_write`

Mutex

- Binäres Semaphor (passives Warten).
- Löst das Semaphor ab.
- Vergleich Semaphor - Mutex:
  - Geringerer Speicherbedarf.
  - Schneller.
- Headerdatei: `<linux/mutex.h>`

Mutex - Verwendungshinweise

- Freigabe nur durch den Besitzer möglich.
- Darf nicht mehrfach freigegeben werden.
- Darf nicht mehrfach (rekursiv) reserviert werden.
- Eine Task darf nicht mit einem gehaltenen Mutex beendet werden.
- Mutexe dürfen nicht im Interrupt-Kontext verwendet werden.
- Reservierte Mutexe dürfen nicht reinitialisiert werden.

Mutex - Funktionen

- DEFINE_MUTEX(name);
- mutex_init(mutex);
- void mutex_lock(struct mutex *lock);
- int mutex_lock_interruptible(struct mutex *lock);
  - Returnwert: 0 oder -EINTR
- int mutex_trylock(struct mutex *lock);
- void mutex_unlock(struct mutex *lock);
- int mutex_is_locked(struct mutex *lock);

Mutex - Verwendungsbeispiel

```
...
#include <linux/mutex.h>

static struct mutex tmutex;

static int __init mod_init(void)
{
    printk("mod_init()\n");
    mutex_init( &tmutex );

    mutex_lock( &tmutex );
    printk("kritischer Abschnitt...\n");
    mutex_unlock( &tmutex );
    return -EIO; // return 0;
}
...
```

Spinlock

- Synchronisationselement zum Schutz eines „kurzen“ kritischen Abschnitts.
- Realisiert ein aktives Warten.
- Im Prozess- und Interruptkontext einsetzbar!
- In der Theorie nur für Mehrprozessormaschinen geeignet.
- Praktisch auch auf Einprozessormaschinen einsetzbar.

Spinlock

- Initialisierung:
  - statisch:
    - `static spinlock_t my_lock = SPIN_LOCK_UNLOCKED;`
  - dynamisch:
    - `spin_lock_init( &my_lock );`

Spinlock

- Anwendung:
  - statisch:
    - `static spinlock_t my_lock = SPIN_LOCK_UNLOCKED;`
  - dynamisch:
    - `spin_lock_init( &my_lock );`

- `spin_lock`
- `spin_unlock`
- `spin_lock_irq`
- `spin_unlock_irq`
- `spin_lock_irqsave`
- `spin_unlock_irqsave`
- `spin_lock_bh`
- `spin_unlock_bh`

Spinlock

```
#include <asm/spinlock.h>
...
unsigned long irqflags;
static struct spinlock_t my_lock=SPIN_LOCK_UNLOCKED;
...

    ...
    spin_lock_irqsave( &my_lock, &irqflags );
    ... // kritischer Abschnitt
    spin_unlock_irqrestore( &my_lock, &irqflags );

    ...
```

Spinlock

Ausprägungen

- `read_lock, write_lock`
- `read_unlock, write_unlock`
- `read_lock_irq, write_lock_irq`
- `read_unlock_irq, write_unlock_irq`
- `read_lock_irqsave, write_lock_irqsave`
- `read_unlock_irqrestore, write_unlock_irqrestore`
- `read_lock_bh, write_lock_bh`
- `read_unlock_bh, write_unlock_bh`

Methodenwahl

r.-Ing. J. Quade

	Treiberinstanz	Kernel–Thread	Workqueue Event–Workqueue	Softirq	Tasklet Timer	Hardirq
Treiberinstanz	Semaphore RW–Semaphore Spinlock RW–Lock Seqlock	Semaphore RW–Semaphore Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Kernel–Thread	Semaphore RW–Semaphore Spinlock RW–Lock Seqlock	Semaphore RW–Semaphore Spinlock RW–Lock	Spinlock RW–Lock Seqlock	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Workqueue Event–Workqueue	Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Softirq	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock RW–Lock	Spinlock RW–Lock	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Tasklet Timer	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock RW–Lock	Spinlock RW–Lock 1)	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Hardirq	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock RW–Lock 2)

1) Nur bei unterschiedlichen Tasklets. Ein Tasklet läuft zu einem Zeitpunkt garantiert nicht zweimal.
2) Nur bei unterschiedlichen ISRs. Ein und dieselbe ISR läuft zu einem Zeitpunkt nicht zweimal ab.

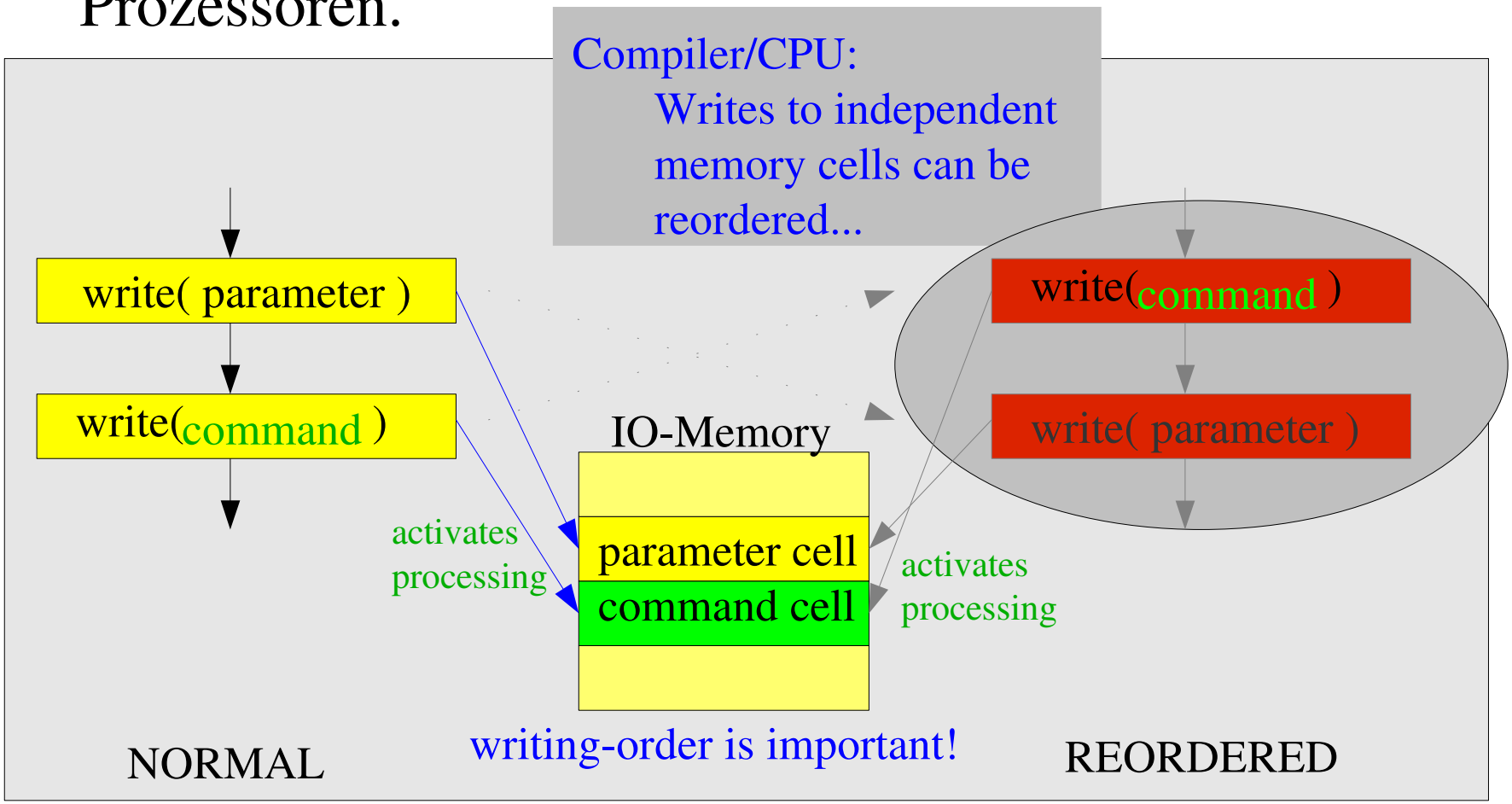
Memory Barrier

- Problem:
  - Reordering durch die CPU
  - Reordering durch den Compiler
- Beispiel Hardware-Zugriff:
  - Notwendige Reihenfolge: Daten, Kommando
  - Aus Sicht der CPU sind das zwei voneinander unabhängige Zugriffe – Reordern ist erlaubt.
  - Aus Sicht der Hardware: Das Kommando kommt vor den Daten.

```
...
outw( DATA_PORT, 0x0001 );
outw( COMMAND_PORT, 0x1122 );
...
```

Memory Barrier

- Befehls-Umsortierung moderner Compiler und Prozessoren.



Memory Barrier

- Lösung:
  - Spinlocks/Semaphore
  - Besser: Memory Barriers
- Memory Barriers stellen sicher, dass die programmierte Abarbeitungsreihenfolge eingehalten wird:
  - `mb()`
  - `rmb()`
  - `wmb()`
  - `barrier()` (verhindert Reordering durch den Compiler)

```
...
outw( DATA_PORT, 0x0001 );
wmb();
outw( COMMAND_PORT, 0x1122 );
wmb();
...
```

Completion-Objekt

- Dient der Synchronisation zwischen zwei Jobs:
  - Job 2 wartet darauf, dass Job 1 eine Aktion beendet.
- `DECLARE_COMPLETION`
- `init_completion`
- `complete`
- `complete_all`
- `complete_and_exit`
- `wait_for_completion`

Completion-Object

```
DECLARE_COMPLETION( on_exit );
...
static int kernelthread( void *data )
{
    ...
    if( signal_pending( current ) ) {
        complete_and_exit( &on_exit, 0 );
        return 0;
    }
    ...
    complete_and_exit( &on_exit, 0 );
    return 0;
}
...
static void __exit mod_exit( void )
{
    kill_proc( thread_id, SIGTERM );
    wait_for_completion( &on_exit );
    ...
}
```

Übung 15: Kritischer Abschnitt

- Kopieren Sie den Treiber „openclose.c“ in die Datei „semaphor.c“.
- Modifizieren Sie die Funktion driver_open() so, dass die aufrufende Applikation für 3 Sekunden schlafen gelegt wird.
- Stellen Sie mit Hilfe eines Semaphors sicher, dass die Funktion driver_open() nur jeweils von einer Instanz aufgerufen werden kann.
- Um den erfolglosen Versuch, driver_open() parallel ablaufen zu lassen, zu protokollieren, verwenden Sie die Funktion down_try_lock().

Übung 15: Kritischer Abschnitt

- Ist der kritische Abschnitt belegt, soll in einer Schleife alle 200ms geprüft werden, ob der kritische Abschnitt frei ist.
- Testen Sie den generierten und installierten Treiber durch parallele Zugriffe mit „cat“.

Zusammenfassung

- Linux bietet eine Reihe unterschiedlicher Methoden zum Schutz kritischer Abschnitte.
- Die Vielfalt resultiert aus dem Wunsch, ein möglichst **preiswertes** Verfahren anbieten zu können.
- Das Erkennen und das daraus erfolgende Absichern von kritischen Abschnitten ist eine der großen Herausforderungen bei der Kernel-Programmierung.
- Zusätzlich zu den vorgestellten Möglichkeiten gibt es noch das Sequence-Lock.