

# Synchronisation im Kern

Reinhard Bündgen  
[buendgen@de.ibm.com](mailto:buendgen@de.ibm.com)

# Warum Synchronisieren?

Thread 1

`i++;`

-----

`get i (7)`

`increment i (7->8)`

`write back i (8)`

Thread 2 [1]

`i++;`

-----

`get i (8)`

`increment i (8->9)`

`write back i (9)`

Thread 2 [2]

`i++;`

-----

`get i (7)`

`increment i (7->8)`

`write back i (8)`

# Ursachen von Nebenläufigkeit im Kern

- UP & SMP (Pseudo-Nebenläufigkeit)
  - Unterbrechungen
  - Kern-Preemption
  - Schlafen/Warten
- SMP (echte Nebenläufigkeit, Parallelität)
  - paralleles Arbeiten von CPUs

# Was muß synchronisiert werden?

- ✓ Der Zugriff auf Ressourcen (Daten)
- ✗ Nicht der Ablauf von bestimmten Codeabschnitten

## Kritischer Abschnitt:

- Codeabschnitt in dem auf kritische Ressource zugegriffen wird.
- Merke, unterschiedliche Codeabschnitte können auf die gleiche kritische Ressource zugreifen

# Synchronisationsmechanismen

- UP
  - Verhindern von Unterbrechungen
  - Kein Aufruf von `schedule( )` in kritischen Abschnitten
- SMP
  - Verhindern von Unterbrechungen
  - Schlösser (Locks)

# Probleme mit der Synchronisation

- Verklemmungen (dead locks)

Thread 1

Thread 2

Lock A

Lock B

Lock B

Lock A

- Skalierbarkeit
  - Wenn mehrere Threads gleichzeitig das gleiche Lock anfordern, kann nur einer weiterarbeiten (Sequenzialisierung)
  - Je länger ein Lock gehalten wird, desto höher die Wahrscheinlichkeit, dass Threads sequenzialisiert werden.

# Überblick über Synchronisationsmechanismen im Linuxkern

- Atomare Operationen
  - Auf ints
  - auf longs (ab 2.6.16)
  - Auf bits in einem Wort
- Spin Locks
  - Einfach
  - Lese-/Schreiblocks
  - Sequence Locks
- Semaphoren
  - Standard
  - Mutexe
  - Lese-/Schreib Semaphore
- Completion Variablen
- RCU
- The Big Kernel Lock (BLK)
- Per-CPU Daten & Preemption disabling
- Barrieren

# Atomare Operationen

- Auf Integers (24 bit)
  - Datentyp `atomic_t`
  - z.B. `atomic_add(2,&v); /* v=v+2 atomically */`
  - `<asm/atomic.h>`
- Auf Longs (64 bit)
  - Datentyp `atomic_long_t`
  - z.B. `atomic_long_read(atomic_long_t *l)`
- Bit Operationen
  - z.B. `set_bit(3,&word);`  
`/* bit 3 of word is set atomically */`
  - `<asm/bitops.h>`



# Spin Locks

- Basiert auf Synchronisations Instruktion
  - z.B. test&set oder compare&swap
  - <asm/spinlock.h> <linux/spinlock.h>
- Busy-waiting an einem Schloss

- Nicht rekursiv

```
spinlock_t datax_lock = SPIN_LOCK_UNLOCKED;
```

```
...
```

```
spin_lock(&datax_lock);
```

```
/* critical region for work on datax */
```

```
spin_unlock(&datax_lock);
```

- Spin Lock Nutzung
  - Kurz!
  - kein Kontextwechsel, insbesondere *nicht* schlafen

# Spin Locks und Interrupts

- Wenn ein Spin Lock von einem Interrupt Handler angefordert werden kann muss es immer gegen Interrupts geschützt werden!

```
spinlock_t datax_lock = SPIN_LOCK_UNLOCKED;  
unsigned long myflags;  
...  
spin_lock_irqsave(&datax_lock, myflags);  
/* critical region for work on datax */  
spin_unlock_irqrestore(&datax_lock, myflags);
```

- Wenn keine IRQs maskiert:

```
spin_lock_irq()  
spin_unlock_irq()
```

# Lese/Schreib Spin Locks

- Initialisierung
- `rwlock_t myrwlock = RW_LOCK_UNLOCKED;`
- Lese Lock (gemeinsamer Zugriff)
  - `read_lock()` / `read_unlock()`
- Schreib Lock (exklusiver Zugriff)
  - `write_lock()` / `write_unlock()`
- Kein „upgrade“ von Lese auf Schreiblock möglich
- Lese/Schreib Locks favorisieren Leser
- Jeweils interruptmaskierende Varianten vorhanden
  - `{read|write}_lock_{irq|irqsave}`
  - `{read|write}_unlock_{irq|irqrestore}`

# Sequence Locks

- Lese/Schreib Lock
- favorisiert Schreiber
- Implementierung mit Hilfe von Zähler
  - Schreibreservierung: inkrementiert Zähler auf ungerade Zahl
  - Freigabe einer Schreibreservierung: Inkrementiert Zähler auf gerade Zahl

```
seqlock_t mylock = SEQLOCK_UNLOCKED;
```

```
write_seqlock(&mylock); ... write_sequnlock(&mylock);
```

- Lese Reservierung
- Klammere Zugriff und stelle sicher, dass Zähler unverändert bleibt

```
unsigned long seq;
```

```
do {
```

```
    seq = read_seqbegin(&mylock);
```

```
    /* lese Daten - keine Seiteneffekte!!! */
```

```
} while (read_seqretry(&mylock, seq));
```

# Semaphoren

- Edsger Wybe Dijkstra (1930-2002)
- Anfordernde Task schläft, wenn sie auf Lockfreigabe warten muss
  - Kann nur in Prozess Kontext benutzt werden
- Semaphorwert:
  - Zahl der Semaphor-Halter (Maximum deklarierbar)
  - 0: Semaphore ist frei
- Operationen
  - Reservieren: P() [poberen], down()
    - falls frei, inkrementiere Zahl der Halter, sonst warte bis frei
  - Freigeben: V() [verhogen], up()
    - dekementiere Zahl der Halter
- Binäre Semophore / Mutex
  - Maximale Zahl der Halter ist 1

# Semaphorfunktionen in Linux

- `<asm/semaphore.h>`
- `sema_init(struct semaphore *, int)`
- `init_MUTEX(struct semaphore *)`
- `init_MUTEX_LOCKED(struct semaphore *)`
- `down(struct semaphore *)`
- `down_interruptible(struct semaphore *)`
- `down_trylock(struct semaphore *)`
- `up(struct semaphore *)`

# Lese/Schreib-Semaphore

- `<linux/rwsem.h>`
- `init_rwsem(struct rw_semaphore *)`
- `down_read(struct rw_semaphore *)`
- `up_read(struct rw_semaphore *)`
- `down_read_trylock(struct rw_semaphore *)`
- `down_write(struct rw_semaphore *)`
- `up_write(struct rw_semaphore *)`
- `down_write_trylock(struct rw_semaphore *)`
- `downgrade_writer(struct rw_semaphore *)`

# Read Copy Update (RCU)

- Ziel: verhindere Blockaden bei Synchronisation
- Idee:
  - Leser lesen aktuelles Objekt ohne zu blockieren
  - Schreiber modifizieren Kopie des aktuellen Objekts und ersetzen ursprüngliches Objekt *atomar* durch aktuelle Kopie
- Anforderung
  - Leser und Schreiber hängen nicht von einander ab
  - Objekt, dynamisch, nur durch einen Zeiger erreichbar
- Herausforderung
  - Wann kann man ursprüngliches Objekt frei geben?



# RCU – Linux Implementierung

- `<linux/rcupdate.h>`
- Leser: schläft nicht, nicht preemptiv
  - `rcu_read_lock()` / `rcu_read_unlock()`
- Schreiber:
  - alloziert Kopie der Struktur und füllt sie aus
  - ersetzt atomar ursprüngliche Struktur durch neue Kopie,
  - schedule-t Freigabe mit
    - `void call_rcu(struct rcu_head head, void (*func) void *arg), void arg)`
    - `rcu_head` beschreibt „Freigabetask“
- Freigabe der ursprünglichen Struktur:
  - nach dem alle CPUs durch quiescent Zustand gegangen sind:
    - nach Task Kontextswitch
    - in User Space
    - in Idle Task

# Sonstige Locks

- The Big Kernel Lock: BLK
  - Schlafen erlaubt, wird automatisch freigegeben und wieder reserviert
  - Rekursives Lock (so oft freigeben, wie reservieren)
  - `lock_kernel()`, `unlock_kernel()`, `kernel_locked()`
- Completion variables
  - `DECLARE_COMPLETION(name)`
  - `init_completion(struct completion *)`
  - `wait_for_completion(struct completion *)`
  - `complete(struct completion *)`
- Verhindern von Preemption
  - implizit mit Spin Locks
  - Synchronisation für CPU-lokale Variablen
    - `preempt_disable()` / `preempt_enable()`
    - `preempt_enable_no_resched()`, `preempt_count()`
    - `int get_cpu()` / `put_cpu()`

# Barrieren

- Prozessor Scheduling Barrieren
  - `rmb()`, `wmb()`, `mb()`,  
`read_barrier_depends()`
  - `smp_rmb()`, `smp_wmb()`, `smp_mb()`,  
`smp_read_barrier_depends()`
- Compile Scheduling Barrieren
  - `barrier()`