Praktikum angewandte Systemsoftwaretechnik

Synchronisierung im Linux-Kernel

Alexander Würstlein

Lehrstuhl Informatik 4

2018-02-01

Synchronisierung

- ...sichert Konsistenz von Daten und Abläufen
- vor...
  - unerwünschtem gleichzeitigem Zugriff
  - unerwünschtem wechselweisem Zugriff
- durch...
  - gegenseitigem Ausschluss
  - Serialisierung
  - atomare (unteilbare) Operationen

Synchronisierung im Betriebssystemkern

Situationen

- Interrupts und Traps (Prologe)
- Tasklets, threaded Interrupts (Epiloge)
- Userspace vs. Kernel
- Multiprozessorsysteme
- Userspace vs. Userspace z.B. Syscalls

wichtig

geschickte Auswahl der richtigen Synchronisationsprimitive

- „Selbstbau“ fast immer unnötig, oft fehleranfällig
- einfach, schön und billig bevorzugen

atomare Datentypen

atomic_t

```
atomic{,64,_long}_t x;

int y = atomic_read(&x);
atomic_set(y, &x);

atomic_inc(&x); atomic_dec(&x); /* RMW */

atomic_cmpxchg(...);
atomic_inc_unless_negative(...);
...
```

atomic bitops

```
test_bit();
{set,clear,change}_bit(); /* RMW */
test_and_{set,clear,change}_bit();
```

CPU-Primitiven

- `mfence, sfence, lfence` (x86_64)
- compare & swap
- `lock` (x86)
- atomare CPU-Instruktionen
- (hardware) transactional memory

Achtung

Eher nur als Bausteine für komfortablere Methoden geeignet!

RCU-Datenstrukturen

- überwiegend lesender Zugriff: Lesen vorhersagbar, warte- und sperrfrei
- dreiphasiges Entfernen eines Elements:
  - Verlinkung entfernen
  - auf Leser des Elements warten
  - Element zerstören/freigeben

```
rcu_read_lock(); /* Magie */
list_for_each_entry_rcu(e, &audit_tsklist, list) {
  if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
      rcu_read_unlock(); return state;
  }
}
rcu_read_unlock();
return AUDIT_BUILD_CONTEXT;
...
list_add_rcu(&entry->list, list);
list_del_rcu(&entry->list);
```

Semaphoren

- `struct mutex`: binäre Semaphore
  - Allzwecksperre
  - schlafend (z.B. blockierende Syscalls)
  - hohe Kosten
  - extra-Features: Warteliste, dreiphasig (Lock, aktives Warten, Schlafen)
- `struct semaphore`: zählend, weniger extra-Features
- weitere: rt-mutex, ww-mutex
- nur freigebende Operationen geeignet für Interrupt-Handler

Spinlocks

- sehr billig
- ausschliesslich für sehr kurze kritische Abschnitte
- implementiert durch aktives Warten („spinning“)
- synchronisiert gegen Interrupt-Handler durch Interrupt-Sperre
  - sperrt Interrupts nur auf lokaler CPU
  - andere CPUs warten aktiv

```
spin_lock{,_irqsave}(&spinlock);
kritisch();
spin_unlock{,_irqrestore}(&spinlock);
```

kritische Sequenz: warum Interrupts sperren?

```
kernel: spin_lock(&spinlock);
/* Interrupt wird ausgeloest */
interrupt: spin_lock(&spinlock);
/* deadlock */
```

weitere Ansätze

- Produzent/Konsument-Ringpuffer:
  `Documentation/circular-buffers.txt`
- uvm: die passende Datenstruktur für ein Problem wurde meist schon implementiert

Quellen: `Documentation/`