PB173 – Ovladače jádra – Linux

IV.

Jiří Slabý

ITI, Fakulta Informatiky

19. 10. 2010

Chyby souběhu, zámky

LDD3 kap. 5 (zastaralá)

Co je chyba souběhu

- Chyba závislá na načasování/prokládání operací

```
int *addr;
...
int a = fetch(addr);
a = a + 1;
store(a, addr);
```

Ukázkový kód

Příklad chyby souběhu

Vlákno A	Vlákno B

`int a = fetch(addr);`
`a = a + 1;`
`<schedule>`	

Vlákno A	Vlákno B

`int a = fetch(addr);`
`a = a + 1;`
	`int a = fetch(addr);`
	`a = a + 1;`
	`store(a, addr);`

Vlákno A	Vlákno B

`int a = fetch(addr);`
`a = a + 1;`
`store(a, addr);`
	`int a = fetch(addr);`
	`a = a + 1;`
	`store(a, addr);`

Řešení chyb souběhu

- Atomickou operací ve stylu `fetch_inc_store`
  - Podpora CPU (zamknutí sběrnice, cache koherence)
  - Ne na všechno jsou operace (vesměs +, -, set, get)
- Kritickou sekcí
  - Kus kódu vykonávaný max. jedním procesem
  - Zámky
- Read-copy-update (RCU)
  - Podrobnosti v LDD

Atomické operace

- **asm/atomic.h**, **Documentation/atomic_ops.txt**
- `atomic_t a = ATOMIC_INIT(5)`
- Pojme 32 bitů se znaménkem (`int`) (historicky jen 24)
- `atomic_read`, `atomic_set`
- `atomic_add`, `atomic_inc`, `atomic_sub`, `atomic_dec`, `atomic_*_return` a další (LXR)

```
int *addr;
...
int a = fetch(addr);
a = a + 1;
store(a, addr);
```

$\Rightarrow$

```
atomic_t a;
...
atomic_inc(&a);
/* nebo atomic_add(1, &a); */
```

Řešení pomocí atomických operací

- `atomic64_t` (drahý na 32-bitu)

Úkol

Práce s atomickými typy

1. Definice jednoho `atomic_t` v `module_init`
2. Nastavit hodnotu na -3 (nejlépe staticky)
3. Atomicky: přičíst 1 a přečíst hodnotu
4. Přečtenou hodnotu vypsat do logu
5. Přičíst 3
6. Odečíst 1
7. Přečíst hodnotu a vrátit jako návratovou

Atomické bitové operace

- Stačí-li 1 bit namísto `int`
- **`linux/bitops.h`**, **`Documentation/atomic_ops.txt`**
- `unsigned long a = 0`, popř. `DECLARE_BITMAP(a, 1000)`
- `set_bit`, `clear_bit`, `test_bit`
- `test_and_set_bit`, `test_and_clear_bit`

Bitmapy lze použít i NEATOMICKY (např. v kritických sekcích)

- **`linux/bitmap.h`**
- `__set_bit`, `__clear_bit`
- `bitmap_zero`, `bitmap_fill`, `bitmap_copy`
- `bitmap_OP`, kde `OP` $\in$ {`and`, `or`, `xor`, `andnot`, `complement`}
- `bitmap_empty`, `bitmap_full`, ...

Úkol

Práce s bitmapami

1. Definice bitového pole o 100 bitech
2. Výmaz pole (`bitmap_zero`)
3. Nastavení 2., 63. a 76. bitu
4. Výpis longu (`%lx`) s 63. bitem (`pole[BIT_WORD(63)]` z LXR)
5. Výpis celého pole (`bitmap_scnprintf`)
6. Výpis longů obsahující 1 bity (`for_each_set_bit` z LXR)
7. Výpis pozice 1. nastaveného bitu (`find_first_bit`)

Zámky

Vytvoření kritické sekce

- Spinlocky
  - Čekání ve smyčce (požírá strojový čas)
  - Rychlé, nesmí se uvnitř spát (čekat)
- Mutexy
  - Spící, fronta čekatelů
  - Pomalejší než spinlock (viz `__mutex_lock_common`)
- Semafory
  - Podobné mutexům
  - Počítadlo (jsou rekurzivní)
  - Dnes se používají vyjímečně
- Monitory (java) . . .

Zámky lze držet jen v jádře (po dobu vykonávání syscallu)

Zámky v jádře – spinlocky

- **linux/spinlock.h**, **Documentation/spinlocks.txt**
- `DEFINE_SPINLOCK(lock)`, `spinlock_t lock`
- `spin_lock`, `spin_unlock`
- read/write varianty (**linux/rwlock.h**)
  - Jeden zapisovatel, více čitatelů (jen odůvodněné případy)
  - Dražší operace než obyčejný spinlock
- `_irq*` varianty
  - Zákaz přerušení, poté spinlock
- Podobné `pthread` spinlockům

```
int *addr;
...
int a = fetch(addr);
a = a + 1;
store(a, addr);
```

⇒

```
DEFINE_SPINLOCK(addr_lock);
int *addr;
...
spin_lock(&addr_lock);
int a = fetch(addr);
a = a + 1;
store(a, addr);
spin_unlock(&addr_lock);
```

Řešení pomocí spinlocků

Zámky v jádře – mutexy

Mutexy

- `linux/mutex.h`
- `DEFINE_MUTEX(name)`
- `mutex_lock`, `mutex_unlock`
- Přerušitelné varianty
  - Signál přeruší čekání
  - Tyto funkce vracejí hodnotu ne/mám zámek
- Podobné `pthread` mutexům

```
int *addr;
...
int a = fetch(addr);
a = a + 1;
store(a, addr);
```

⇒

```
DEFINE_MUTEX(addr_lock);
int *addr;
...
mutex_lock(&addr_lock);
int a = fetch(addr);
a = a + 1;
store(a, addr);
mutex_unlock(&addr_lock);
```

Řešení pomocí mutexů

Zámky v jádře – ostatní

Semafory

- `linux/semaphore.h`
- Víceméně nepoužívat
- Pozor: `DECLARE_MUTEX(lock)`
- `down`, `up`

Big Kernel Lock (BKL)

- NEPOUŽÍVAT
- Hrubozrnný zámek
- Pochází z dob počátku Linuxu
- `lock_kernel`, `unlock_kernel`

Úkol

Atomické čtení/zápis bufferu o velikosti 128 bytů

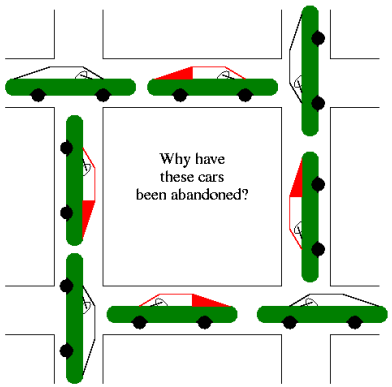
- Globální buffer
- 2 znaková (i misc) zařízení
  - 1 implementuje `.read`
  - 1 implementuje `.write`
- Zápis
  - Umožněn pouze po 5 znacích (vrací max. 5)
  - Spí 10 ms po každém zápisu znaku do bufferu (`msleep` z **`linux/delay.h`**)
- Čtení
  - Vrátí naráz celých 128 B
  - Musí vidět změny pouze po 5 znacích (až na poslední pětici)
- Vyzkoušejte

Pozn. 1: pb173/04

Pozn. 2: odevzdat s domácím

Deadlock

- 4 podmínky uváznutí
- Jádro spoléhá na programátora, že k němu nikdy nedojde
- LockDep
  - Dynamický mechanismus hledání chyb v zámcích
- Obvyklé typy chyb: ABBA, AA
- Obvyklé chyby: `lock` + `if` + `return`



Další problémy zámků

- Zpomalují kritický kód
  - Odstraní se zámky
  - Např. kruhové buffery
- Nevhodná granularita
  - Jeden zámek na všechno vs. jeden zámek na jednu činnost
  - Např. BKL, nebo naopak zámky každého registru
- Zahlcení
  - Příliš mnoho procesů čeká na zámek
  - Lze řešit přechodem na COW, RCU, RW zámky, . . .
  - Např. všechny procesy čekají na `tasklist_lock`