

PB173 – Ovladače jádra – Linux IV.

Jiri Slaby

ITI, Fakulta Informatiky

10. 10. 2012

LDD3 kap. 5 (zastaralá)

Co je chyba souběhu

- Chyba závislá na načasování/prokládání operací

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

Ukázkový kód

Příklad chyby souběhu

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

Uvažujme $*addr == 0$

*addr	Vláknno A	Vláknno B
0	int a = load(addr);	<waiting> ~
0	a = a + 1;	
0	<schedule>	-

*addr	Vláknno A	Vláknno B
0	int a = load(addr);	int a = load(addr);
0	a = a + 1;	a = a + 1;
1	<waiting>	store(a, addr);

*addr	Vláknno A	Vláknno B
0	int a = load(addr);	<exited> ~
0	a = a + 1;	
/*>1<*/	store(a, addr);	-

- Atomickou operací ve stylu `load_inc_store`
 - Nutná podpora CPU
 - Ne na všechno jsou operace (vesměš jen +, -, load, store)
- Kritickou sekci
 - Kus kódu vykonávaný max. jedním procesem
 - Zámky
- Read-copy-update (RCU)
 - Podrobnosti v LDD

Část I

Atomické operace, bitmapy

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 = load(addr);  ⇒  atomic_t a;
a = a + 1;           atomic_inc(&a);
store(a, addr);      /* nebo atomic_add(1, &a); */
```

Řešení pomocí atomických operací

- `atomic64_t` (drahý na 32-bitu)

Práce s atomickými typy

- 1 Definice jednoho `atomic_t` v `module_init`
- 2 Nastavit hodnotu na -3 (nejlépe staticky)
- 3 Atomicky jednou operací „přičíst 1 a přečíst hodnotu”
- 4 Přečtenou hodnotu vypsát 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`
- `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` $\in \{\text{and, or, xor, andnot, complement}\}$
- `bitmap_empty`, `bitmap_full`, ...

Práce s bitmapami

- 1 Definice bitového pole o 100 bitech
- 2 Výmaz pole (`bitmap_zero`)
- 3 Nastavení 2., 63. a 76. bitu
- 4 Vypis *longu* (`%lx`) s 63. bitem (`bitmapa[BIT_WORD(63)]`)
- 5 Vypis celé bitmapy (`bitmap_scnprintf`)
- 6 Vypis *longů* obsahující „1” bity (`foreach_set_bit`)
- 7 Vypis *pozice* 1. nastaveného bitu (`find_first_bit`)

Část II

Zámky

Vytvoření kritické sekce

- Spinlocky
 - Čekání ve smyčce (požírání strojový čas)
 - Rychlé, nesmí se uvnitř spát (čekat)
- Mutexy
 - Spící, fronta čekatelů
 - Pomalejší než spinlock (viz tělo `--mutex_lock_common`)
- Semaforey
 - Podobné mutexům
 - Počítadlo (jsou rekurzivní)
 - Dnes se používají výjimečně

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`
- Podobné pthread spinlockům

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

Řešení pomocí spinlocků

Spinlocky a přerušení

Vláknno	Přerušení
<pre>spin_lock(&addr_lock); <interrupt> spin_unlock(&addr_lock); .</pre>	<pre>. spin_lock(&addr_lock); // ^^^ deadlock ^^^ spin_unlock(&addr_lock);</pre>

irq* varianty

- Zákaz přerušení, poté spinlock

Vláknno	Přerušení
<pre>spin_lock_irq(&addr_lock); // interrupt cannot trigger spin_unlock_irq(&addr_lock); . .</pre>	<pre>. . spin_lock(&addr_lock); ... spin_unlock(&addr_lock);</pre>

Mutexy

- `linux/mutex.h`
- `DEFINE_MUTEX(name)`
- `mutex_lock`, `mutex_unlock`
- Podobné `pthread` mutexům

```
int *addr;
```

```
...
```

```
int a = load(addr);
```

```
a = a + 1;
```

```
store(a, addr);
```

⇒

```
DEFINE_MUTEX(addr_lock);
```

```
int *addr;
```

```
...
```

```
mutex_lock(&addr_lock);
```

```
int a = load(addr);
```

```
a = a + 1;
```

```
store(a, addr);
```

```
mutex_unlock(&addr_lock);
```

Řešení pomocí mutexů

Semafor

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

Big Kernel Lock (BKL)

- NEPOUŽÍVAT
 - V nových jádrech už ani není
- Hrubozrnný zámek
- Pochází z dob počátku Linuxu
- `lock_kernel`, `unlock_kernel`

Atomické čtení/zápis bufferu o velikosti 128 bytů

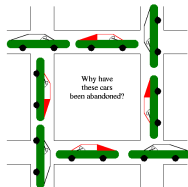
- Globální buffer
- 2 znaková (misc) zařízení
 - 1 implementuje `.read`
 - 1 implementuje `.write`
- Zápis
 - Umožněn max. po 5 znacích (`.write` vrací max. 5)
 - Spí 20 ms po každém zápisu znaku do bufferu (`msleep` z `linux/delay.h`)
- Čtení
 - Vrábí naráz celých 128 B (je-li `count` dostatečně velký)
 - Musí vidět změny pouze po 5 znacích (až na poslední pětici)
- Vyzkoušejte

Pozn. 1: práce s `offp` v pb173/04

Pozn. 2: odevzdat s domácím

Deadlock

- 4 podmínky uvážnutí
- 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`



- Zpomalují kritický kód
 - Řešení: odstranit zámký
 - Např. kruhovými buffery
- Nevhodná granularita
 - Jeden zámek na všechno vs. jeden zámek na jednu činnost
 - Např. BKL, nebo naopak zámký každého registru
- Zhlcení
 - Příliš mnoho procesů čeká na zámek
 - Lze řešit přechodem na COW, RCU, RW zámký, ...
 - Např. všechny procesy čekají na `tasklist_lock`