

# PB173 – Ovladače jádra – Linux

## IV. Chyby souběhu

Jiri Slaby

Fakulta informatiky  
Masarykova univerzita

15. 10. 2015

- 1 Chyby souběhu
- 2 Atomické operace, bitmapy
- 3 Zámky
  - Spinlocky
  - Mutexy

## LDD3 kap. 5 (zastaralá)

### Co je chyba souběhu

- Chyba závislá na načasování/prokládání operací

### Ukázkový kód

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

# Příklad chyby souběhu

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

Uvažujme při startu `*addr == 0`

*addr	Vláknno A	Vláknno B
0	<code>int a = load(addr);</code>	<code>&lt;waiting&gt;</code> ~
0	<code>a = a + 1;</code>	
0	<code>&lt;schedule&gt;</code>	

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

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

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

## Sekce 2

# Atomické operace, bitmapy

# Atomické operace

- `linux/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)

## Řešení pomocí atomických operací

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

⇒

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

- `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

Pozn.: tento kód nemažte, budeme s ním nadále pracovat



# 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 (čísla považujte za indexy v C)
- 4 Vypis *longu* (`%lx`) s 63. bitem
  - `bitmapa[BIT_WORD(63)]`
- 5 Vypis celé bitmapy jako řetězce
  - `bitmap_scanf` (do 4.0), `%*bp` (od 4.0)
- 6 Vypis *longů* obsahující bity nastavené na 1
  - `for_each_set_bit`
- 7 Vypis *pozice* 1. nastaveného bitu
  - `find_first_bit`

## Sekce 3

# Zámky

## Vytvoření kritické sekce

- Spinlocky
  - Čekání ve smyčce (požírání strojový čas)
  - Rychlé, ale **POZOR**: 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

**POZOR**

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

# Zámky v jádře – spinlocky

- Čekání ve smyčce
- `linux/spinlock.h`, `Documentation/spinlocks.txt`
- `DEFINE_SPINLOCK(lock)`, `spinlock_t lock`
- `spin_lock`, `spin_unlock`
- Podobné pthread spinlockům

## Řešení pomocí spinlocků

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

⇒

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

## Práce se spinlocky

- 1 Úkol s atomickými operacemi přepište
- 2 `atomic_t` změňte na obyčejný `int`
- 3 A celý kód obalte spinlockem
- 4 Vyzkoušejte

## Mutexy

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

## Řešení pomocí mutexů

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

⇒

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

## Semaforey

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

## Big Kernel Lock (BKL)

- Historický
  - Dnes už v jádře nenajdeme (commit 4ba8216cd905 v 2.6.39)
  - Jen jeho relikty
- Hrubozrnný
- Pochází z dob počátku Linuxu
- `lock_kernel`, `unlock_kernel`



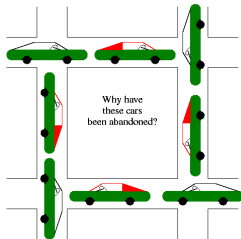
## Atomické čtení/zápis bufferu o velikosti 128 bytů

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

Pozn.: inspirace v pb173/04

# 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` (*POZOR*)



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