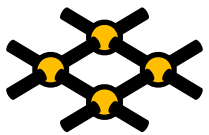
Vláknové programování
část II

Lukáš Hejmánek, Petr Holub
{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192
2011–03–10

Přehled přednášky

Kritické sekce

- Co je to kritická sekce?
  - Nereentrantní část kódu
- Ne vždy je na první pohled zřejmé, co je a není reentrantní.

Kritické sekce – ukázky

- Obecně je kritickou sekcí každá část kódu, která načte data, zpracuje je a zapíše zpět.
- Příklad:
  - Načti čítač ze souboru
  - Zvyš čítač o jedna
  - Zapiš současnou hodnotu čítače zpět do souboru

```
1
2  struct queue {
3          void *data;
4          struct queue *next;
5  } queue;
6
7  struct queue *head;
8  struct queue *tail;
9
10 void
11 add(void *data)
12 {
13         tail->next = malloc(sizeof(struct queue));
14         tail->data = data;
15         tail = tail->next;
16 }
```

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

volatile int x=0;

void *
foo(void *arg) {
        int i;
        while(x == 0);
        for(i = 0; i < 1000000; i++)
                asm("incl %0" : : "m" (x));
        printf("%d\n", x);
        return NULL;
}

int
main(void) {
        pthread_t t1, t2, t3;

        pthread_create(&t1, NULL, foo, NULL);
        pthread_create(&t2, NULL, foo, NULL);
        pthread_create(&t3, NULL, foo, NULL);
        x=1;
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
        pthread_join(t3, NULL);
        return 0;
}
```

- Příklad výstupu programu:
  - 1136215
  - 1355167
  - 1997368
- Očekávaný výstup:
  - xxxxxxx
  - yyyyyyy
  - 3000001
- Uvedené špatné chování se nazývá *race condition* (soupeření v běhu).

Řešení kritických sekcí

- Nejlépe změnou kódu na reentrantní verzi.
  - Ne vždy je to možné.
- Pomocí synchronizace = zamezení současného běhu kritické sekce
  - Snížení výkonu – přicházíme o výhodu paralelního běhu aplikace
- Synchronizační nástroje:
  - Mutexy (zámky)
  - Semafory
  - Podmíněné proměnné

Zámky

- Vzájemné vyloučení vláken
- Well-known algoritmy
  - Petersonův algoritmus
  - Dekkerův algoritmus
  - Lamportův algoritmus „pekařství“

Zámky

```
1  flag[0]   = 0;
2  flag[1]   = 0;
3  turn;
4
5  P0: flag[0] = 1;                          P1: flag[1] = 1;
6      turn = 1;                                 turn = 0;
7      while (flag[1] == 1 && turn == 1)         while (flag[0] == 1 && turn == 0)
8      {                                         {
9             // busy wait                          // busy
10            // wait
11     }                                         }
12     // critical section                       // critical
13     // section
14        ...                                    ...
15     // end of critical section                // end of critical
16     // section
17     flag[0] = 0;                              flag[1] = 0;
```

- Proč nefunguje:
  http://bartoszmilewski.wordpress.com/2008/11/05/who-ordered-memory-fences-on-an-x86/

Změny pořadí zápisů a čteni

- Současné procesory mohou měnit pořadí zápisů a čtení
- Čtení může prohozeno se starším zápisem

```
1 int a,b;
2
3  a = 5;
4  if(b) { }
```

- Důsledek:

```
1 init: x=0
2 Thread 1          Thread 2
3 x = 1             if ready == 1
4 ready = 1            R = x
```

Změny pořadí zápisů a čteni

- Současné procesory mohou měnit pořadí zápisů a čtení
- Čtení může prohozeno se starším zápisem

```
1 int a,b;
2
3  a = 5;
4  if(b) { }
```

- Důsledek:

```
1 init: x=0
2 Thread 1          Thread 2
3 x = 1             if ready == 1
4 ready = 1            R = x
```

  - **ready=1 && x=0**

Změny pořadí zápisů a čteni

- Současné procesory mohou měnit pořadí zápisů a čtení
- Čtení může prohozeno se starším zápisem

```
1 int a,b;
2
3  a = 5;
4  if(b) { }
```

- Důsledek:
  - ```
    1 init: x=0
    2 Thread 1          Thread 2
    3 x = 1             if ready == 1
    4 ready = 1            R = x
    ```
  - **ready=1 && x=0**
  - ```
    1 init: x=0, y=0;
    2 Thread 0          Thread 1
    3 mov [x], 1        mov [y], 1
    4 mov r1, [y]       mov r2, [x]
    ```

Změny pořadí zápisů a čteni

- Současné procesory mohou měnit pořadí zápisů a čtení
- Čtení může prohozeno se starším zápisem

```
1 int a,b;
2
3  a = 5;
4  if(b) { }
```

- Důsledek:
  - ```
    1 init: x=0
    2 Thread 1          Thread 2
    3 x = 1             if ready == 1
    4 ready = 1            R = x
    ```
  - **ready=1 && x=0**
  - ```
    1 init: x=0, y=0;
    2 Thread 0          Thread 1
    3 mov [x], 1       mov [y], 1
    4 mov r1, [y]      mov r2, [x]
    ```
  - **r1=0 && r2=0**

Speciální instrukce CPU

- Atomické operace
  - Bit test (testandset())
  - Load lock/Store Conditional (LL/SC)
  - Compare and Swap (CAS) (x86 – **`cmpxchg`**)
    - **`__sync_bool_compare_and_swap()`**
    - **`__sync_value_compare_and_swap()`**
  - Atomická aritmetika – specialita x86, x86_64
    - Speciální instrukce **`lock`** formou prefixu
    - **`atomic_inc() { "lock xaddl %0, %1"}`**
      `__sync_fetch_and_add(val, 1)`
    - **`atomic_dec() { "lock xsubl %0, %1"}`**
      `__sync_fetch_and_sub(val, 1)`
    - **`xchg(int a, int b) { "xchgl %0, %1"}`**
- Paměťové bariéry
  - rmb(), wmb()
  - **`__sync_synchronize()`** – plná paměťová bariéra

Volatilní typy

- Nekonečná smyčka

```
1 int x=0;
2
3 void foo()
4 {
5         while(x==0);
6
7         x = 10;
8         //continue
9 }
```

```
1 foo:
2         movl    x(%rip), %eax
3         testl   %eax, %eax
4         je      .L2
5         movl    $10, x(%rip)
6         ret
7 .L2:
8 .L4:
9         jmp     .L4
```

Volatilní typy

- Funkční verze

```
1  volatile int x=0;
2
3  void foo()
4  {
5          while(x==0);
6
7          x = 10;
8          //continue
9  }
```

```
1  foo:
2  .L2:
3          movl    x(%rip), %eax
4          testl   %eax, %eax
5          je      .L2
6          movl    $10, x(%rip)
7          ret
```

Volatilní typy

- Volatilní proměnná: **`volatile int x;`** nebo **`int volatile x;`**
- Nevolatilní ukazatel na volatilní proměnnou: **`volatile int *x;`**
- Volatilní ukazatel na nevolatilní proměnnou: **`int *volatile x;`**
- Volatilní ukazatel na volatilní proměnnou: **`volatile int *volatile x;`**

Zámek

- Naivní algoritmus zámku

```
1  volatile int val=0;
2
3  void lock() {
4          while(atomic_inc(val)!=0) {
5                  //sleep
6          }
7  }
8
9  void unlock() {
10         val = 0;
11         // wake up
12 }
```

Zámek s podporou kernelu

- Podpora kernelu o „volání“ **`my_sleep_while()`**
  - Pozastaví proces právě tehdy když je podmínka splněna
- „volání“ **`my_wake()`**
  - Vzbudí pozastavený proces(y)

```
1  volatile int val=0;
2
3  void lock() {
4          int c=
5          while((c=atomic_inc(val))!=0) {
6                  my_sleep_while(val==(c+1));
7          }
8  }
9
10 void unlock() {
11         val = 0;
12         my_wake();
13 }
```

Mutexy

- Mechanismus zámků v knihovně Pthreads
- Datový typ **`pthread_mutex_t`**.
- Inicializace **`pthread_mutex_init`**
  (Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Zamykání/odemykání
  - **`pthread_mutex_lock`**
  - **`pthread_mutex_unlock`**
- Zrušení zámku **`pthread_mutex_destroy`**.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

volatile int x=0;

pthread_mutex_t x_lock;

void *
foo(void *arg)
{
        int i;
        while(x == 0);
        for(i = 0; i < 1000000; i++) {
                pthread_mutex_lock(&x_lock);
                x++;
                pthread_mutex_unlock(&x_lock);
        }
        printf("%d\n", x);
        return NULL;
}
```

```
1  int
2  main(void)
3  {
4          pthread_t t1, t2, t3;
5
6          pthread_mutex_init(&x_lock, NULL);
7          pthread_create(&t1, NULL, foo, NULL);
8          pthread_create(&t2, NULL, foo, NULL);
9          pthread_create(&t3, NULL, foo, NULL);
10         x=1;
11         pthread_join(t1, NULL);
12         pthread_join(t2, NULL);
13         pthread_join(t3, NULL);
14         pthread_mutex_destroy(&x_lock);
15         return 0;
16 }
```

- Výstup změněného programu:
  - 2424575
  - 2552907
  - 3000001
- Což je očekávaný výsledek

Zámky „bolí“

- Mějme tři varianty předchozího příkladu:

```
1 void foo(void *arg) {
2         int i;
3         while(x == 0);
4         for(i = 0; i < 100000000; i++)
5                 x++;
6 }
```

```
1 void foo(void *arg) {
2         int i;
3         while(x == 0);
4         for(i = 0; i < 100000000; i++)
5                 asm("lock incl %0" : : "m" (x));
6 }
```

```
1 void foo(void *arg) {
2         int i;
3         while(x == 0);
4         for(i = 0; i < 100000000; i++) {
5                 pthread_mutex_lock(&x_lock);
6                 x++;
7                 pthread_mutex_unlock(&x_lock);
8         }
9 }
```

- Délky trvání jednotlivých variant (real time, 3 vlákna)
  - Bez zámku (nekorektní verze)
    1.052sec
  - „Fast lock“ pomocí assembleru
    5.716sec
  - pthread mutex
    66.414sec

Big kernel lock vs. Spin locking

- Vlákna a procesy mohou mít velké množství zámků.
- Koncepce *Big kernel lock*
  - Pro všechny kritické sekce máme jeden společný zámek
  - Název pochází z koncepce Linux kernelu verze 2.0
  - Jednoduchá implementace
  - Může dojít k velkému omezení paralelismu
- Koncepce *Spin locking*
  - Pro každou kritickou sekci zvláštní zámek
  - Název pochází z koncepce Linux kernelu verze 2.4 a dalších
  - Složitější implementace
  - Omezení paralelismu je většinou minimální
  - Velké nebezpečí vzniku skrytých race conditions.
    - Některé kritické sekce mohou spolu dohromady tvořit další kritickou sekci.

Semafory

- Mutexy řeší vzájemné vyloučení v kritických sekcích.
- Semafory řeší synchronizaci úloh typu producent/konzument.
- Producent/konzument úloha:
  - Producent vyrábí
  - Konzument(i) spotřebovává
  - Problém synchronizace - konzument může spotřebovat nejvýše tolik, co producent vytvořil
  - Příklad:
    - Producent přidává objekty do fronty
    - Konzument odebírá objekty z fronty
    - Synchronizace: konzument může odebrat pouze, je-li fronta neprázdná, jinak má čekat.
    - Není vhodné čekat pomocí tzv. busy loop, tj. neustále zjišťovat stav front, vlákno zbytečně spotřebovává procesor.

Synchronizace s použitím busy loop

```
1  producer:
2
3          while( ) {
4                  pridej prvek do fronty;
5          }
6
7  consumer:
8          while( ) {
9                  /* busy loop */
10                 while(fronta prazdna) nedelej nic;
11
12                 odeber prvek z fronty
13         }
```

Semafor

- Semafor je synchronizovaný čítač.
- Producent čítač zvyšuje
- Konzument čítač snižuje
- Čítač nelze snížit k záporné hodnotě
- Pokus o snížení k záporné hodnotě zablokuje vlákno, dokud není čítač zvýšen.

- Rukojeť semaforu **`sem_t`**.
- Inicializace semaforu **`sem_init()`**
  (Inicializaci je vhodné provádět před vytvořením vláken).
- Zvýšení hodnoty semaforu **`sem_post()`**.
- Snížení hodnoty semaforu a případně čekání na zvýšení jeho hodnoty **`sem_wait()`**.
- Zrušení semaforu **`sem_destroy()`**.

```c
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

sem_t sem;

int quit=0;

void *
producer(void *arg)
{
        int i=0;
        while(!quit) {
                /* produce same data */
                printf("Sending data %d\n",i++);
                sem_post(&sem);
        }
}
```

```c
void *
consumer(void *arg)
{
        int i=0;
        while(!quit) {
                /* wait for data */
                sem_wait(&sem);
                printf("Data ready %d\n",i++);
                /* consume data */
        }
}

int
main(void)
{
        pthread_t p, c;

        sem_init(&sem, 0, 0);
        pthread_create(&c, NULL, consumer, NULL);
        pthread_create(&p, NULL, producer, NULL);

        sleep(1);
        quit = 1;
        pthread_join(c, NULL);
        pthread_join(p, NULL);
        sem_destroy(&sem);
}
```

Ukázka části výstupu programu

```
1  Sending data 0
2  Sending data 1
3  Sending data 2
4  Sending data 3
5  Sending data 4
6  Sending data 5
7  Sending data 6
8  Sending data 7
9  Data ready 0
10 Data ready 1
11 Data ready 2
12 Data ready 3
13 Data ready 4
14 Data ready 5
15 Data ready 6
16 Data ready 7
17 Sending data 8
18 Sending data 9
19 Sending data 10
```

Podmíněné proměnné

- Synchronizace násobení matic
  - $A \times B \times C$
    1. Workery vynásobí $A \times B$
    2. Musí počkat
    3. Pokračují v násobení maticí C

Podmíněné proměnné

- Synchronizace pomocí podmínek:
  - A: Čekej na splnění podmínky
  - B: Oznam splnění podmínky

Podmíněné proměnné

- Základní rukojeť podmínky **`pthread_cond_t`**.
- Inicializace podmínky **`pthread_cond_init()`**.
- Čekání na podmínku **`pthread_cond_wait()`**.
- Signalizace splnění podmínky
  - **`pthread_cond_signal()`** – probudí alespoň jednoho čekatele.
  - **`pthread_cond_broadcast()`** – probudí všechny čekatele.
- Zrušení podmínky **`pthread_cond_destroy()`**.

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

pthread_cond_t condition;
pthread_mutex_t cond_lock;

void *
worker(void *arg)
{
        pthread_mutex_lock(&cond_lock);
        printf("Waiting for condition\n");
        pthread_cond_wait(&condition, &cond_lock);
        printf("Condition true\n");
        pthread_mutex_unlock(&cond_lock);
        return NULL;
}
```

```
18 int
19 main(void)
20 {
21         pthread_t p;
22
23         pthread_mutex_init(&cond_lock, NULL);
24         pthread_cond_init(&condition, NULL);
25
26         pthread_create(&p, NULL, worker, NULL);
27
28         sleep(1);
29         printf("Signaling condition\n");
30         pthread_mutex_lock(&cond_lock);
31         pthread_cond_signal(&condition);
32         pthread_mutex_unlock(&cond_lock);
33         printf("Condition done\n");
34
35         pthread_join(p, NULL);
36         pthread_cond_destroy(&condition);
37         pthread_mutex_destroy(&cond_lock);
38         return 0;
39 }
```

RCU struktury

- Read-Copy-Update struktura
- Zbytečné zamykat každý přístup
- Zřejmě
  - Lze číst paralelně
  - Aktualizovat může jen jeden v době, kdy jiný nečte
  - Musíme zamykat pro čtení?
- RCU struktury jsou hojně využívané v jádru Linuxu

RCU struktury

- Využití RCU struktur
  - Aktualizace seznamů
  - Seznam lze paralelně číst
  - Seznam nelze paralelně aktualizovat
- Princip RCU
  - Čtení seznamu „běžným“ způsobem
  - Modifikace:
    - Zduplikování starého prvku seznamu (`malloc()`)
    - Modifikace duplikátu
    - Záměna původního prvku s duplikátem
    - Co s původním prvkem?

RCU struktury

- API RCU struktur:
  - `rcu_read_lock()`
  - `rcu_read_unlock()`
  - `synchronize_rcu()`
  - `rcu_assign_pointer()`
  - `rcu_dereference()`
  - `call_rcu()`

RCU struktury

- Koncepce RW-zámků
- Lze držet několik zámků typu *read*
- Lze držet nejvýše jeden zámek typu *write* pokud není držen žádný zámek typu *read*
- Implementace v (nepreemptivním) jádře
  - `rcu_read_lock() { }`
  - `rcu_read_unlock() { }`
  - Write zámek: `synchronize_rcu() { for_each_cpu(cpu) run_on_cpu(cpu); }`