

# Vláknové programování

## část II

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



Laboratoř pokročilých síťových technologií

PV192  
2012-03-06

# Přehled přednášky

## Základy synchronizace

Zámky

Mutexy

Spin Locks

Semaforey

Podmíněné proměnné

# 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;**

# 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     tail->next = NULL;
17 }
```



- Co se stane, když 2 procesy přidají prvek do stejné fronty?
- Co obsahuje **tail**->**next**?

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4
5 volatile int x=0;
6
7 void *
8 foo(void *arg) {
9     int i;
10    while(x == 0);
11    for(i = 0; i < 1000000; i++)
12        asm("incl_%0" : : "m" (x));
13    printf("%d\n", x);
14    return NULL;
15 }
16
17 int
18 main(void) {
19     pthread_t t1, t2, t3;
20
21     pthread_create(&t1, NULL, foo, NULL);
22     pthread_create(&t2, NULL, foo, NULL);
23     pthread_create(&t3, NULL, foo, NULL);
24     x=1;
25     pthread_join(t1, NULL);
26     pthread_join(t2, NULL);
27     pthread_join(t3, NULL);
28     return 0;
29 }
```

- Příklad výstupu programu:
  - 1136215
  - 1355167
  - 1997368
- Očekávaný výstup:
  - xxxxxxxx
  - yyyyyyyy
  - 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)
  - Semaforey
  - Podmíněné proměnné

# Zámky

- Vzájemné vyloučení vláken
- Well-known algoritmy (ze „staré školy“)
  - Petersonův algoritmus
  - Dekkerův algoritmus
  - Lamportův algoritmus „pekařství“

# Petersonův algoritmus

```
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 &&   while (flag[0] == 1 &&
8         turn == 1)         turn == 0)
9   {                         {
10      // busy wait          // busy wait
11   }                          }
12   // critical section      // critical section
13   ...                        ...
14   // end of critical section // end of critical section
15   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 čtení

- Proč algoritmy ze „staré školy“?
- 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 čtení

- Proč algoritmy ze „staré školy“?
- 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 čtení

- Proč algoritmy ze „staré školy“?
- 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 čtení

- Proč algoritmy ze „staré školy“?
- 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

- Paměťové bariéry
  - `rmb()`, `wmb()`
  - `__sync_synchronize()` – plná paměťová bariéra

# 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"`

# 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`.

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4
5 volatile int x=0;
6
7 pthread_mutex_t x_lock;
8
9 void *
10 foo(void *arg)
11 {
12     int i;
13     while(x == 0);
14     for(i = 0; i < 1000000; i++) {
15         pthread_mutex_lock(&x_lock);
16         x++;
17         pthread_mutex_unlock(&x_lock);
18     }
19     printf("%d\n", x);
20     return NULL;
21 }
```



```
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.

# Spin locks

- Klasické zámky (mutexy) používají systémové volání **futex()**.
  - Podpora jádra pro NPTL implementaci POSIX threads.
  - Mutexy používají systémová volání  $\Rightarrow$  nutnost přepnutí kontextu.
- Zámky typu spin jsou implementovány kompletně v user space.
  - Nemusí se přepínat kontext.
  - Za cenu busy loopu při pokusu zamknout zámeček (Vlákno se cyklicky dotazuje, zda je možno zámeček zamknout – spinning).
  - Jsou situace, kdy přepnutí kontextu trvá déle než busy loop pro zamčení.
- Kdy je vhodné použít spin locks?
  - Při velmi krátké kritické sekci (typicky zvýšení/snížení proměnné).
  - Nedojde-li k přepnutí kontextu jinou cestou (máme-li více vláken než procesorů, spin lock neurýchlí běh).
- Ne všechny implementace POSIX threads poskytují spin locks!

# Spin locks

```
1 void spin_lock(volatile int *lock)
2 {
3     __sync_synchronize();
4     while(! __sync_bool_compare_and_swap(lock, 0, 1));
5 }
6
7 void spin_unlock(volatile int *lock)
8 {
9     *lock = 0;
10    __sync_synchronize();
11 }
```

# Spin locks

- Datový typ **pthread\_spin\_t**.
- Inicializace **pthread\_spin\_init**  
(Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Zamykání/odemykání
  - **pthread\_spin\_lock**
  - **pthread\_spin\_unlock**
- Zrušení zámku **pthread\_spin\_destroy**.



# Příklad

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4
5 int x=0;
6
7 pthread_spinlock_t x_lock;
8
9 void *
10 foo(void *arg)
11 {
12     int i;
13     while(x == 0);
14     for(i = 0; i < 100000000; i++) {
15         pthread_spin_lock(&x_lock);
16         x++;
17         pthread_spin_unlock(&x_lock);
18     }
19     printf("%d\n", x);
20     return NULL;
21 }
```

# Příklad

```
22 int
23 main(void)
24 {
25     pthread_t t1, t2;
26
27     pthread_spin_init(&x_lock, 0);
28     pthread_create(&t1, NULL, foo, NULL);
29     pthread_create(&t2, NULL, foo, NULL);
30     x=1;
31     pthread_join(t1, NULL);
32     pthread_join(t2, NULL);
33     pthread_spin_destroy(&x_lock);
34     return 0;
35 }
```

## Doba běhu příkladu

- Test na 2 procesorovém systému.
- V případě 2 vláken:
  - Za použití mutexů: 29 sec
  - Za použití spinů: 11 sec
- V případě 3 vláken:
  - Za použití mutexů: 28 sec
  - Za použití spinů: 29 sec

# Semaforey

- Mutexy řeší vzájemné vyloučení v kritických sekcích.
- Semaforey 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()**.

```
1 #include <semaphore.h>
2 #include <pthread.h>
3 #include <stdio.h>
4 #include <unistd.h>
5
6 sem_t sem;
7
8 int quit=0;
9
10 void *
11 producer(void *arg)
12 {
13     int i=0;
14     while(!quit) {
15         /* produce same data */
16         printf("Sending_data_%d\n",i++);
17         sem_post(&sem);
18     }
19 }
```



```
1 void *
2 consumer(void *arg)
3 {
4     int i=0;
5     while(!quit) {
6         /* wait for data */
7         sem_wait(&sem);
8         printf("Data_ready_%d\n",i++);
9         /* consume data */
10    }
11 }
12
13 int
14 main(void)
15 {
16     pthread_t p, c;
17
18     sem_init(&sem, 0, 0);
19     pthread_create(&c, NULL, consumer, NULL);
20     pthread_create(&p, NULL, producer, NULL);
21
22     sleep(1);
23     quit = 1;
24     pthread_join(c, NULL);
25     pthread_join(p, NULL);
26     sem_destroy(&sem);
27 }
```

## 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()`.

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 pthread_cond_t condition;
6 pthread_mutex_t cond_lock;
7
8 void *
9 worker(void *arg)
10 {
11     pthread_mutex_lock(&cond_lock);
12     printf("Waiting_for_condition\n");
13     pthread_cond_wait(&condition, &cond_lock);
14     printf("Condition_true\n");
15     pthread_mutex_unlock(&cond_lock);
16     return NULL;
17 }
```

```
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 }
```