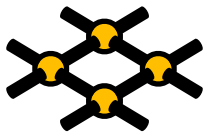
Vláknové programování
část VI

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



Laboratoř pokročilých síťových technologií

PV192
2014–03–25

Knihovna Pthreads

- POSIX Threads (Pthreads) je POSIX standard pro vlákna
- Vlákna v systémech na bázi jádra Linux
  - Linux threads – neúplná implementace POSIX Threads
    - Vlákno bylo obsluhováno stejně jako proces, mělo i vlastní PID (process ID).
    - Není nutná speciální podpora jádra, problémy s výkonem, pokud se vlákno samo nevzdá procesoru (yield()).
  - Nahrazena NPTL – Native POSIX threads library
    - Výrazně vyšší výkonnost
    - Vlákno je samo o sobě jednotkou plánování, tj. procesový plánovač plánuje i vlákna obvykle úplně stejně.
    - NPTL potřebuje speciální podporu jádra pro synchronizaci.
- Pthreads knihovna má implementaci pro řadu systémů: Linux, *BSD, Windows, MacOS, ...

Knihovna Pthreads

- Vytváření vláken a procesů v Linuxu
  - Vlákno vzniká systémovým voláním **`clone(2)`**
  - Proces vzniká systémovým voláním **`fork(2)`** (případně **`vfork`**)
    - Nejčastější použití **`fork(2)`** je pro spuštění nového programu
    - Po **`fork(2)`** dojde k rozštěpení rodiče, duplikaci adresního prostoru, atd.
    - Následně je pomocí **`execl(3)`** zrušen obsah paměti a puštěn nový program
    - Jednodušší volání **`system(3)`**, nelze ale použít vždy
- Procesy se obvykle vytváří přímo voláním **`fork(2)`**, vlákna pomocí knihovny pthreads.

Knihovna Pthreads

- Vytvoření procesu

```
1  #include <unistd.h>
2
3  void
4  run(char *name)
5  {
6          pid_t child;
7
8          if((child=fork())==0) {
9                  /* child */
10                 execlp(name, NULL);
11                 return;
12         }
13         if(child < 0) {
14                 perror("fork error");
15         }
16         /* parent */
17         return;
18 }
```

Vytváření vláken pomocí Pthreads

- Rukojeť vlákna **`pthread_t`**, používá se pro pro takřka všechna volání týkající se vytváření a manipulace s vlákny.
- **`int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void*), void* arg);`**

- Vytvoření vlákna v C

```
1  #include <pthread.h>
2
3  void *
4  runner(void *foo)
5  {
6          return NULL;
7  }
8
9  int
10 main(void)
11 {
12         pthread_t t;
13
14         pthread_create(&t, NULL, runner, NULL);
15         return 0;
16 }
```

- Nefunkční příklad pro C++

```
1  #include <pthread.h>
2
3  // not void*
4  void
5  runner(void *foo)
6  {
7          return;
8  }
9
10 int
11 main(void)
12 {
13         pthread_t t;
14
15         pthread_create(&t, NULL, runner, NULL);
16         return 0;
17 }
```

Ukončování vláken

- Možnosti ukončení vlákna samotným vláknem:
  - Návrat z hlavní funkce startu vlákna (třetí argument funkce **`pthread_create`**).
  - Explicitní zavolání funkce **`pthread_exit(void *value_ptr)`**.
- Možnosti ukončení vlákna jiným vláknem:
  - „Zabití“ vlákna **`pthread_kill(pthread_t thread, int sig)`**.
  - Zasláním signálu *cancel* **`pthread_cancel(pthread_t thread)`**
  - Nedoporučovaná možnost, není jisté, kde přesně se vlákno ukončí.
- Ukončení vlákna ukončením celého procesu
  - Zavoláním **`exit(3)`**
  - Posláním signálu **`SIGKILL`**, **`SIGTERM`**, ...

- Co s návratovou hodnotou ukončeného vlákna?
- Pro zjištění návratové hodnoty
  `int pthread_join(pthread_t thread, void **value)`.

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

void *
runner(void *foo)
{
        sleep(10);
        pthread_exit(NULL);
}

int
main(void)
{
        pthread_t t;

        pthread_create(&t, NULL, runner, NULL);

        pthread_kill(t, SIGKILL);
        return 0;
}
```

Reentrantní funkce

- Vícenásobný běh programu má určitá úskalí.
- Kód funkcí musí počítat s tím, že může být prováděn několikrát najednou.
- Problematické jsou globální proměnné a statické lokální proměnné.

```
1  char buffer_2[200];
2
3  char *
4  foo1(char * a)
5  {
6          static char buffer_1[200];
7
8          snprintf(buffer_1, 200, "Text: %s\n", a);
9
10         return buffer_1;
11 }
12
13 char *
14 foo2(char *a)
15 {
16         snprintf(buffer_2, 200, "Text: %s\n", a);
17
18         return buffer_2;
19 }
```

- *Reentrantní funkce* – funkce schopná násobného běhu.
- Příklad funkce **`foo(char *)`** – implementace alokuje dynamický kus paměti.
- Makro **`__REENTRANT`** – je-li definováno, říkáme překladači a hlavičkovým souborům, že funkce mohou být vykonávány násobně.
- Knihovní funkce:
  - Thread safe – lze používat z více vláken
  - Not thread safe – nelze používat z více vláken
  - Některé funkce nemohou být thread safe z podstaty věci – **`strtok(3)`**
  - POSIX.1c rozšiřuje množinu funkcí o thread safe varianty, např. **`strtok_r(3)`**

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.

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

int x=0;

void *
foo(void *arg)
{
        int i;
        while(x == 0);
        for(i = 0; i < 1000000; i++) {
                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;
        sleep(2);
        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

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>

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         sleep(2);
12         pthread_mutex_destroy(&x_lock);
13         return 0;
14 }
```

- 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

Semafory

- 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()`**.

Semafory

```
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 }
```

Semafory

```
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 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()`**.

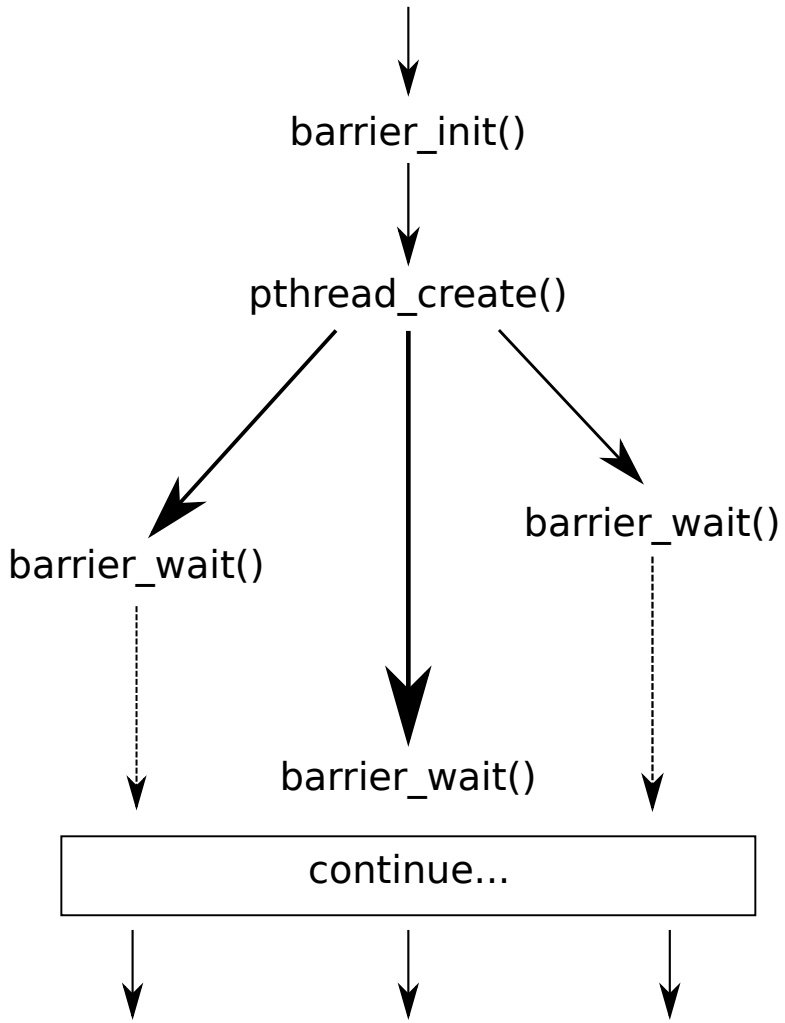
```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

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 }
```

Bariéry


barrier_init()
pthread_create()
barrier_wait()
barrier_wait()
barrier_wait()
continue...

Bariéry

- Datový typ **`pthread_barrier_t`**.
- Inicializace **`pthread_barrier_init()`**
  (Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Při inicializaci specifikujeme, pro kolik vláken bude bariéra sloužit.
- Zastavení na bariéře **`pthread_barrier_wait()`**.
- Zrušení bariéry **`pthread_barrier_destroy`**.

Příklad bariéry

```
1  #include <pthread.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  pthread_barrier_t barrier;
6
7  void *
8  foo(void *arg) {
9          int slp = (int)arg;
10         printf("Working..\n");
11         sleep(slp);
12         printf("Waiting on barrier\n");
13         pthread_barrier_wait(&barrier);
14         printf("Synchronized\n");
15         return NULL;
16 }
```

Příklad bariéry

```
17 int
18 main(void)
19 {
20         pthread_t t1, t2;
21
22         pthread_barrier_init(&barrier, NULL, 2);
23         pthread_create(&t1, NULL, foo, (void*)2);
24         pthread_create(&t2, NULL, foo, (void*)4);
25
26         pthread_join(t1, NULL);
27         pthread_join(t2, NULL);
28         pthread_barrier_destroy(&barrier);
29         return 0;
30 }
```

RW Zámky

- Pravidla:
  - Není-li zámek zamčen v režimu *Write*, může být libovolněkrát zamčen v režimu *Read*.
  - Je-li zámek zamčen v režimu *Write*, nelze jej už zamknout v žádném režimu.
  - Je-li zámek zamčen v režimu *Read*, nelze jej zamknout v režimu *Write*.
- Opět ne všechny implementace POSIX threads implementují RW zámky (korektně)!

RW zámky

- Datový typ **`pthread_rwlock_t`**.
- Inicializace **`pthread_rwlock_init()`**
  (Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Zamknutí v režimu *Read* **`pthread_rwlock_rdlock()`**.
- Zamknutí v režimu *Write* **`pthread_rwlock_wrlock()`**.
- Opakované zamčení jednoho zámku stejným vláknem skončí chybou **`EDEADLK`**.
  Není možné povýšit *Read* zámek na *Write* zámek a naopak.
- Odemknutí v libovolném režimu **`pthread_rwlock_unlock()`**
  Pthreads nerozlišují odemknutí dle režimů, některé implementace vláken párují rdlock s příslušným rdunlock, stejně tak pro wrlock.
- Zrušení rw zámku **`pthread_rwlock_destroy`**.

Příklad

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4
5  struct x_t {
6          int a;
7          int b;
8          pthread_rwlock_t lock;
9  };
10
11 struct x_t x;
12
13 int quit = 0;
14
15 pthread_barrier_t start;
```

Příklad

```
16 void *
17 reader(void *arg)
18 {
19         int n = (int)arg;
20         pthread_barrier_wait(&start);
21
22         while(!quit) {
23                 pthread_rwlock_rdlock(&x.lock);
24                 if((x.a + x.b)%n == 0)
25                         printf(".");
26                 else
27                         printf("+");
28                 pthread_rwlock_unlock(&x.lock);
29                 fflush(stdout);
30                 sleep(1);
31
32         }
33         return NULL;
34 }
```

Příklad

```
35
36 void *
37 writer(void *arg)
38 {
39         int i;
40         pthread_barrier_wait(&start);
41         for(i=0; i < 10; i++) {
42                 pthread_rwlock_wrlock(&x.lock);
43                 x.a = i;
44                 x.b = (i % 2)+1;
45                 pthread_rwlock_unlock(&x.lock);
46                 sleep(5);
47         }
48         quit = 1;
49         return NULL;
50 }
```

Příklad

```
52
53 int
54 main(void)
55 {
56         pthread_t t1, t2, t3;
57
58         x.a = 1;
59         x.b = 2;
60         pthread_rwlock_init(&x.lock, 0);
61         pthread_barrier_init(&start, NULL, 3);
62         pthread_create(&t1, NULL, reader, (void*)2);
63         pthread_create(&t2, NULL, reader, (void*)3);
64         pthread_create(&t3, NULL, writer, NULL);
65         pthread_join(t1, NULL);
66         pthread_join(t2, NULL);
67         pthread_join(t3, NULL);
68         pthread_rwlock_destroy(&x.lock);
69         pthread_barrier_destroy(&start);
70         return 0;
71 }
```

Problémy RW zámků

- Nebezpečí stárnutí zámků.
- Pokud je zamčená část kódu vykonávána déle než nezamčená, nemusí se nikdy podařit získat některý ze zámků.
- V předchozím příkladě nesmí být **`sleep()`** v zamčené části kódu!

```
1       for(i=0; i < 10; i++) {
2               pthread_rwlock_wrlock(&x.lock);
3               x.a = i;
4               x.b = (i % 2)+1;
5               pthread_rwlock_unlock(&x.lock);
6               sleep(5);
7       }
```