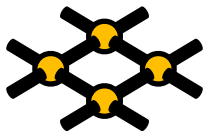
Vláknové programování
část III

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



Laboratoř pokročilých síťových technologií

PV192
2011–03–17

Přehled přednášky

Další nástroje pro synchronizaci

Spin locks

- Klasické zámky (mutexy) používají systémové volání **`futex()`**.
  - Podpora jádra pro NPTL implementaci POSIX threads.
  - Mutexy používají systémová volání ⇒ nutnost přepnutí kontextu.
- Zámky typu spin jsou implementovány kompletně v user space.
  - Nemusí se přepínat kontext.
  - Za cenu busy loopu při pokusu zamknout zámek (Vlákno se cyklicky dotazuje, zda je možno zámek 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 neurychlí 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

```
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 }
```

- Je zde ABA problém?

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 }
```

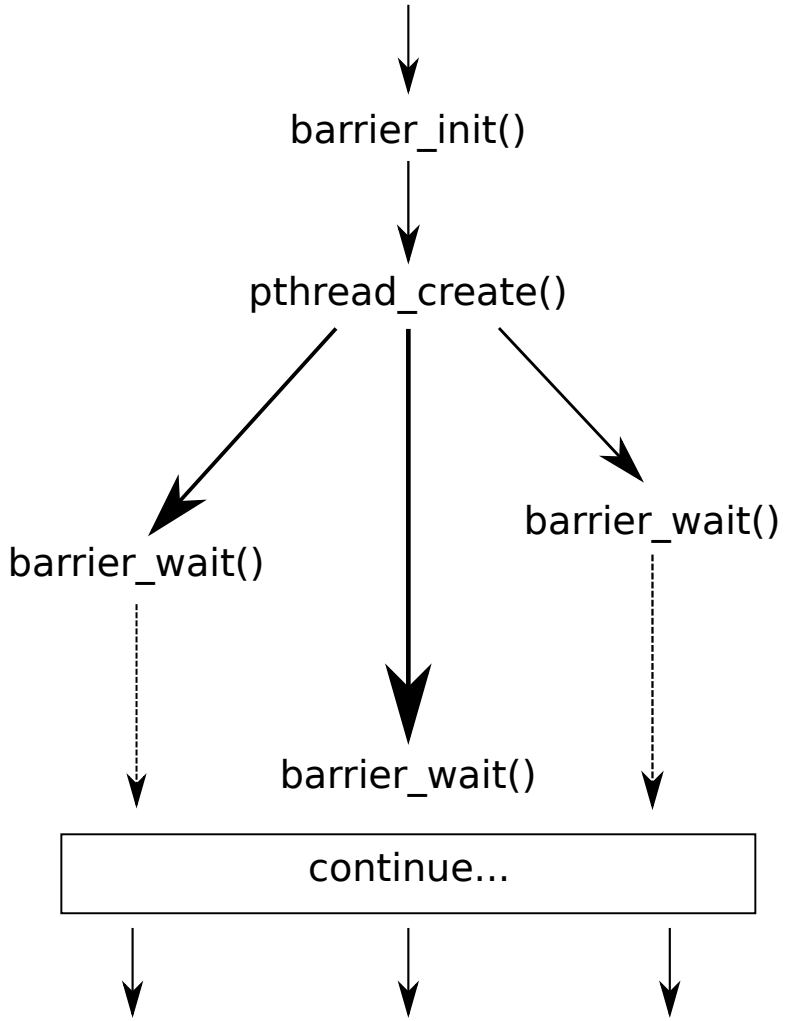
Doba běhu příkladu

- 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

Bariéry

- Bariéry jsou v podstatě místa setkání.
- Bariéra je místo, kde se vlákna setkají.
- Bariéra zablokuje vlákno do doby než k bariéře dorazí všechna vlákna.
- Příklad:
  - Vláknové násobení matic: $M \times N \times O \times P$
  - Každé vlákno násobí a sčítá příslušný sloupec a řádek.
  - Po vynásobení $M \times N$ se vlákna setkají u *bariéry*.
  - Vynásobí předchozí výsledek $\times O$, opět se setkají u bariéry.
  - Dokončí výpočet vynásobením výsledku $\times P$.
- Ne všechny implementace POSIX threads poskytují bariéry!

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 }
```

Read Write zámky

- Read Write zámky dovolují násobné čtení ale jediný zápis.
- Příklad:
  - Několik vláken čte nějakou strukturu (velmi často!).
  - Jedno vlákno ji může měnit (velmi zřídka!).
  - Pozorování:
    - Je zbytečné strukturu zamykat mezi čtecími vlákny
      Nemohou ji měnit a netvoří tedy kritickou sekci.
    - Je nutné strukturu zamknout, mění-li ji zapisovací vlákno
      V této chvíli nesmí strukturu ani nikdo číst (není změněna atomicky).
- Nastupují Read Write 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!

Try varianty synchronizace

- Pomocí try variant volání lze zjistit, zda vstup do kritické sekce je volný či nikoli.
- Funkce atomicky zkusí provést synchronizaci (např. zamknout zámek).
- V případě neúspěchu není funkce blokující, ale okamžitě provede návrat.
- Neúspěch je signalizován návratovým kódem funkce (dle manuálové stránky, pro jednotlivá volání se může *lišit*!).
- Try varianty:
  - **`pthread_mutex_trylock()`**
  - **`pthread_spin_trylock()`**
  - **`pthread_rwlock_tryrdlock()`**
  - **`pthread_rwlock_trywrlock()`**
  - **`sem_trywait()`**

Příklad try zámku

```
1  #include <errno.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  pthread_mutex_t lock;
6
7  void
8  foo(void)
9  {
10         if(pthread_mutex_trylock(&lock) == EBUSY) {
11                 printf("Cannot acquire the lock right now\n");
12         } else {
13                 printf("Locked\n");
14         }
15 }
16
17 int
18 main()
19 {
20         pthread_mutex_init(&lock, NULL);
21
22         foo();
23
24         foo();
25 }
```

Timed varianty synchronizace

- Nástroje zabraňující „věčnému“ čekání.
- Příklad:
  - Jak ukončit vlákno čekající na zámek pomocí globální proměnné?
- K většině blokujících synchronizačních rozhraní existují ekvivalentní rozhraní s časovým omezením.
- Po vypršení časového omezení je vrácena chyba návratovým kódem (dle manuálové stránky, pro jednotlivá volání se může *lišit*!).
- Timed varianty:
  - **`pthread_cond_timedwait()`**
  - **`pthread_mutex_timedlock()`**
  - **`pthread_rwlock_timedrdlock()`**
  - **`pthread_rwlock_timedwrlock()`**
  - **`sem_timedwait()`**
- Ne všechny implementace poskytují *timed* varianty.

Příklad timed zámku

```
1  #include <time.h>
2  #include <errno.h>
3  #include <stdio.h>
4  #include <pthread.h>
5
6  pthread_mutex_t lock;
7
8  void
9  foo(void)
10 {
11         struct timespec tsp;
12
13         tsp.tv_sec = time(NULL)+5;
14         tsp.tv_nsec = 0;
15         if(pthread_mutex_timedlock(&lock, &tsp) == ETIMEDOUT) {
16                 printf("Timeout expired\n");
17         } else {
18                 printf("Locked\n");
19         }
20 }
21
22 int
23 main()
24 {
25         pthread_mutex_init(&lock, NULL);
26         foo();
27         foo();
28 }
```

Pojmenované semafory

- Semafor nemusí být nutně jen paměťový objekt.
- Semaforem může být „soubor“.
- Zůstává zachován datový typ **`sem_t`**.
- Semafor může být sdílen i mezi procesy.
- Založení nebo otevření semaforu **`sem_open()`**.
  - Otevření či založení souboru se podobá volaní **`open()`**, akceptuje příznaky **`O_CREAT`**, **`O_EXCL`**.
  - Jméno semaforu *není* soubor ve filesytému, ale je pouze virtuálním jménem.
- Zavření semaforu **`sem_close()`**.
- Zrušení semaforu **`sem_unlink()`**.

Příklad pojmenovaného semaforu

```
1  #include <semaphore.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4  #include <stdio.h>
5
6  int
7  main()
8  {
9          sem_t* sema = sem_open("/mysem", O_CREAT, 0644, 0);
10
11         if(sema == SEM_FAILED) {
12                 perror("Cannot create semaphore /mysem");
13                 return 1;
14         }
15
16         sem_post(sema);
17         sem_wait(sema);
18
19         sem_close(sema);
20         sem_unlink("/mysem");
21
22         return 0;
23 }
```

Jednorázové zavolání funkce

- Funkce typu inicializace chceme zavolat jen jednou.
- Konstrukce s příznakem, zda inicializace ještě nebyla provedena a následná inicializace je race condition.
- Zamykaní zbytečně snižuje paralelismus.
- **`pthread_once()`** zavolá jednou danou funkci.

Příklad jednorázového zavolání

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4
5  pthread_once_t once_init = PTHREAD_ONCE_INIT;
6  char initialized=0;
7
8  void
9  init(void)
10 {
11         /* do initialization */
12         printf("Initialized\n");
13         initialized = 1;
14 }
15
16 int
17 main(void)
18 {
19         if(!initialized) {
20                 pthread_once(&once_init, init);
21         }
22         return 0;
23 }
```

Futexy

- Synchronizační nástroj Linuxu
- Systémové volání **`long sys_futex(void *addr1, int op, int val1, struct timespec *timeout, void *addr2, int val3)`**
- Operace:
  - **`FUTEX_WAIT`** pozastaví vlákno pokud obsah paměťového místa **`addr1`** je stejný jako **`val1`**, v opačném případě je vrácena chyba **`EWOULDBLOCK`**. Vlákno pokračuje v běhu, obdrželo-li signál.
  - **`FUTEX_WAKE`** Vzbudí jedno nebo více vláken (počet udává proměnná **`val1`**).
    Kernel nesleduje prioritu čekajících vláken.
  - **`FUTEX_CMP_REQUEUE`** Podobně jako WAKE, vzbudí daný počet vláken a daný počet z ostatních vláken přeřadí do fronty čekaní z **`addr1`** na **`addr2`**. To vše za předpokladu, že **`*addr1`** má stejnou hodnotu jako **`val3`**.
  - **`FUTEX_WAKE_OP`**

Futexy

- **FUTEX_WAKE_OP**

```
1 int oldval = *(int*)addr2;
2 *(int*)addr2 = oldval OP OPARG;
3 futex_wake(addr1, val1);
4 if(oldval CMP CMPARG)
5         futex_wake(addr2, (int)timeout);
6
7 /* OP, OPARG, CMP, CMPARG jsou kodovane ve val3 */
```

Mutex pomocí futexu

- `futex_wait(int *val, int val) { return syscall(SYS_futex, val, FUTEX_WAIT, 1, NULL, NULL, 0) }`
- `futex_wake(int *val, 1) { return syscall(SYS_futex, val, FUTEX_WAKE, 1, NULL, NULL, 0) }`
- `val = 0` nezamčeno
- `val != 0` zamčeno

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

Problémy mutexu

- **`unlock()`** vždy používá syscall
  - Kdy je to zbytečné?
- Co se stane, když 2 vlákna vstoupí do **`lock()`** a budou ho provádět paralelně?
- Stačí nám proměnná typu **`int`**?

Mutex pomocí futexu verze 2

- **`cmpxchg(int var, int old, int new): tmp = var; if(var==old) var=new; return tmp;`**
- **`val = 0`** nezamčeno
- **`val = 1`** zamčeno a nikdo nečeká
- **`val = 2`** zamčeno a někdo čeká

```
1  volatile int val=0;
2
3  void lock() {
4          int c;
5          /* try lock */
6          if((c = cmpxchg(val, 0, 1))!=0) {
7                  /* already locked */
8                  do {
9                          /* c==2 -> somebody is waiting already */
10                         if(c==2 || cmpxchg(val, 1, 2)!=0)
11                                 futex_wait(&val, 2);
12                 } while((c = cmpxchg(val, 0, 2))!=0);
13                  /* why val?0:val=*2*? */
14         }
15 }
16
17 void unlock() {
18         if(atomic_dec(val) != 1) {
19                 val = 0;
20                 futex_wake(&val, 1);
21         }
22 }
```

Problémy mutexu

- Zámek bez soupeření

		mutex	mutex v2
lock	atomic op	1	1
	futex syscall	0	0
unlock	atomic op	0	1
	futex syscall	1	0

- Zámek se soupeřením

		mutex	mutex v2
lock	atomic op	$1 + 1$	$\begin{matrix}2+1\\3+2\end{matrix}$
	futex syscall	$1 + 1$	$1 + 1$
unlock	atomic op	0	1
	futex syscall	1	1

Mutex pomocí futexu verze 3

```
1  volatile int val=0;
2
3  void lock() {
4          int c;
5          if((c = cmpxchg(val, 0, 1))!=0) {
6                  if(c!=2)
7                          c = xchg(val,2);
8                  while(c!=0) {
9                          futex_wait(&val, 2);
10                         c = xchg(val, 2);
11                 }
12         }
13 }
14
15 void unlock() {
16         if(atomic_dec(val) != 1) {
17                 val = 0;
18                 futex_wake(&val, 1);
19         }
20 }
```

Finální řešení

- Zámek bez soupeření

		mutex	mutex v2	mutex v3
lock	atomic op	1	1	1
	futex syscall	0	0	0
unlock	atomic op	0	1	1
	futex syscall	1	0	0

- Zámek se soupeřením

		mutex	mutex v2	
lock	atomic op	$1+1$	$\begin{smallmatrix}2+1\\3+2\end{smallmatrix}$	$\begin{smallmatrix}1\\2\end{smallmatrix}+1$
	futex syscall	$1+1$	$1+1$	$1+1$
unlock	atomic op	0	1	1
	futex syscall	1	1	1

Výkon různých variant zamykání

Bez zámků	0.39s
`lock` instrukce	5.72s
Futex verze 2	28.93s
Futex verze 3	22.22s
Pthread mutex (un)lock	52.31s