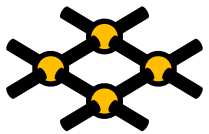
Vláknové programování
část VIII

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



Laboratoř pokročilých síťových technologií

PV192
2014–04–22

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 čteni

- 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, ready=0
2 Thread 1          Thread 2
3 x = 1             if ready == 1
4 ready = 1            R = x
```

Změny pořadí zápisů a čteni

- 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, ready=0
2 Thread 1          Thread 2
3 x = 1             if ready == 1
4 ready = 1            R = x
```

  - **R=1 && x=0**

Změny pořadí zápisů a čteni

- 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, ready=0
    2 Thread 1          Thread 2
    3 x = 1             if ready == 1
    4 ready = 1            R = x
    ```
  - **R=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

- 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, ready=0
    2 Thread 1          Thread 2
    3 x = 1             if ready == 1
    4 ready = 1            R = x
    ```
  - **R=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 }
```

GDB

- Kompilace s podporou GDB: **`gcc -g -o a.out foo.c -pthread -D__REENTRANT`**
- Spuštění pro ladění: **`gdb a.out`**
- Breakpoint – místo, kde je zastaven běh programu
  - **`br main`** – breakpoint na funkci main, tj. gdb zastaví na začátku programu
  - **`br foo.c:10`** – breakpoint v souboru foo.c na radku 10
  - **`del 1`** – smaže první breakpoint
  - **`del`** – smaže všechny breakpointy
- Běh programu: **`(gdb) run [parametry]`** – spustí nahraný program **`a.out`** s případnými parametry
  - Běh se zastaví a) na breakpointu, b) skončením programu, c) přerušením ctrl-c nebo jiným signálem
- **`help příkaz`** zobrazí nápovědu k zadnému příkazu

- Pohybování se mezi funkcemi (po zastavení běhu)
  - **`list`** – vypíše okolí místa zastavení
  - **`up`** – posune se do volající funkce (v sekvenci volání směrem k prvotní funkci **`main()`**)
  - **`down`** – posune se do volané funkce (směrem od prvotní funkce **`main()`**)
  - **`where`** – vypíše sekvenci volání od **`main()`** po aktuální funkci, kde byl přerušen běh
- Sledování proměnných
  - **`print [promenná]`** vypíše jednorázově obsah proměnné, lze používat přetypování, dereference, např. **`print (struct *foo)x->next`**.
  - Je-li použita volba **`-O2`** při překladu, nemusí některé proměnné existovat – optimalizace je nahradily. Pro přístupné proměnné je nutné použít **`-O0`**.
  - **`display [promenná]`** bude vypisovat obsah proměnné po každém příkazu pro gdb.

- Příkazy pro trasování programu
  - **`n [enter]`** krok dál, je-li funkce, provede se a krok skončí za ní
  - **`s [enter]`** krok dál, je-li funkce, vstoupí se do ní
  - Příkazy **`n`** a **`s`** není nutné stále psát, **`[enter]`** automaticky opakuje poslední příkaz

Ladění aplikací

- Ladící výpisy
- Debugger

Ladící výpisy

- Pozor na mixování výpisů jednotlivých vláken do sebe.
- Jeden print je obvykle atomický.
- **`getpid()`** vrací pro vlákna stejnou hodnotu.
- **`pthread_self()`** vrací identifikaci vlákna (**`pthread_t`** – lze vypsat jako integer).
- Ladící výpisy způsobují určitou synchronizaci!

Debugger

- Použití *gdb*:
  - **`info threads`** – vypíše základní informace o běžících vláknech.
  - **`thread ID`** – přepnutí se na konkrétní vlákno.
- Debugger způsobuje velkou synchronizaci!

Ladění aplikací

```
(gdb) info threads
  2 Thread 0x40da4950 (LWP 12809)  0x00007f9f852c7b99 in
pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
* 1 Thread 0x7f9f856de6e0 (LWP 12806)  0x00007f9f84ff8b81 in nanosleep ()
   from /lib/libc.so.6
(gdb) thread 2
[Switching to thread 2 (Thread 0x40da4950 (LWP 12809))]#0  0x00007f9f852c7b99
in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
(gdb) where
#0  0x00007f9f852c7b99 in pthread_cond_wait@@GLIBC_2.3.2 ()
   from /lib/libpthread.so.0
#1  0x0000000000400907 in worker (arg=0x0) at conditions.c:13
#2  0x00007f9f852c33f7 in start_thread () from /lib/libpthread.so.0
#3  0x00007f9f85032b2d in clone () from /lib/libc.so.6
#4  0x0000000000000000 in ?? ()
(gdb)
```

Ladění aplikací

- **`valgrind`** – ladící nástroj
- **`helgrind`** – režim **`valgrindu`**
- Použití: **`valgrind --tool=helgrind aplikace`**
- Detekuje
  - Chybné použití knihovny pthreads
  - Nekonzistentní použití zámků
  - Některé nezamknuté přístupy ke sdíleným datům (data races)

Ladění aplikací

```
1  ==20556== Possible data race during read of size 4 at 0x601040 by thread #3
2  ==20556==    at 0x400630: foo (critsec1.c:10)
3  ==20556==  This conflicts with a previous write of size 4 by thread #1
4  ==20556==    at 0x4006D9: main (critsec1.c:24)
5  ==20556==
6  ==20556== Possible data race during write of size 4 at 0x601040 by thread #3
7  ==20556==    at 0x40064C: foo (critsec1.c:12)
8  ==20556==  This conflicts with a previous write of size 4 by thread #1
9  ==20556==    at 0x4006D9: main (critsec1.c:24)
10 ==20556==
11 ==20556== Possible data race during read of size 4 at 0x601040 by thread #2
12 ==20556==    at 0x400643: foo (critsec1.c:12)
13 ==20556==  This conflicts with a previous write of size 4 by thread #4
14 ==20556==    at 0x40064C: foo (critsec1.c:12)
```

Ladění aplikací

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <math.h>
5
6  void * test_prime(void *arg) {
7          long i,j, prime = *(long*)arg;
8          int sieve[prime];
9          for(i = 0; i < prime; i++) sieve[i] = 1;
10         for(i = 2; i < prime; i++) {
11                 if(sieve[i]) {
12                         printf("%ld, ", i);
13                         for(j=i*2; j < prime; j+=i) sieve[j] = 0;
14                 }
15         }
16         printf("\n"); return NULL;
17 }
18
19 int main(int argc, char *argv[]){
20         if(argv[1] == NULL) {
21                 printf("usage: test prime\n"); return 1;
22         } else {
23                 long test = atoi(argv[1]);
24                 test_prime(&test);
25         }
26         return 0;
27 }
```

Ladění aplikací

```
1  pthread_mutex_t lock_x, lock_y, lock_z;
2  volatile int x = 20, y = 10, z = 0, quit = 0;
3
4  void * foo(void *arg) {
5          while(!quit) {
6                  if(*(int*)arg == 1) { // z = x + y
7                          pthread_mutex_lock(&lock_x);
8                          pthread_mutex_lock(&lock_y);
9                          pthread_mutex_lock(&lock_z);
10                         z = x + y;
11                         pthread_mutex_unlock(&lock_z);
12                         pthread_mutex_unlock(&lock_y);
13                         pthread_mutex_unlock(&lock_x);
14                 }
15                 if(*(int*)arg == 2) { // z = y + x
16                         pthread_mutex_lock(&lock_y);
17                         pthread_mutex_lock(&lock_x);
18                         pthread_mutex_lock(&lock_z);
19                         z = y + x;
20                         pthread_mutex_unlock(&lock_z);
21                         pthread_mutex_unlock(&lock_x);
22                         pthread_mutex_unlock(&lock_y);
23                 }
24         }
25         return NULL;
26 }
```

Ladění aplikací

```
1  int
2  main() {
3          pthread_t t1, t2;
4          int asoc1=1, asoc2=2;
5
6          pthread_create(&t1, NULL, foo, &asoc1);
7          pthread_create(&t2, NULL, foo, &asoc2);
8
9          sleep(10);
10         quit = 1;
11
12         pthread_join(t1, NULL);
13         pthread_join(t2, NULL);
14         return 0;
15 }
```

Start vlákna v C++11

- Třída **`thread`**
- **`#includlde <thread>`**
- Metody
  - **`join`** – odpovídá **`pthread_join`**
  - **`detach`** – osamostatní vlákno, obdoba **`PTHREAD_CREATE_DETACHED`**

Start vlákna v C++11

```
1  #include <iostream>
2  #include <thread>
3
4  void foo1()
5  {
6          std::cout << "Foo1\n";
7  }
8
9  void foo2(int x)
10 {
11         std::cout << "Foo2\n";
12 }
13
14 int main()
15 {
16         std::thread first(foo1);
17         std::thread second(foo2,0);
18
19         second.detach();
20
21         std::cout << "main,␣foo␣and␣bar␣now␣execute␣concurrently...\n";
22
23         first.join();
24
25         std::cout << "foo␣and␣bar␣completed.\n";
26
27         return 0;
28 }
```

Start vlákna v C++11

- Překlad: **`g++ -o c11test c11test.C -std=c++11 -pthread`**
- Neuvedení **`-pthread`** způsobí runtime chybu

```
terminate called after throwing an instance of 'std::system_error'
  what():  Enable multithreading to use std::thread: Operation not
 permitted
Aborted (core dumped)
```

Zamykání

- Třída **`mutex`**
- **`#includlde <mutex>`**
- Metody
  - **`lock`** – odpovídá **`pthread_mutex_lock`**
  - **`unlock`** – odpovídá **`pthread_mutex_unlock`**

Zamykání

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  std::mutex mtx;
6
7  void print_block (int n, char c) {
8          mtx.lock();
9          for (int i=0; i<n; ++i) { std::cout << c; }
10         std::cout << '\n';
11         mtx.unlock();
12 }
13
14 int main ()
15 {
16         std::thread th1 (print_block,50,'*');
17         std::thread th2 (print_block,50,'$');
18
19         th1.join();
20         th2.join();
21
22         return 0;
23 }
```

Zamykání

- Šablona **`std::unique_lock`**
- **`#includlde <mutex>`**
- Šablona garantuje odemknutí zámku při destrukci objektu
- Použití: **`std::unique_lock<std::mutex> lck (mtx);`**
  - Zámek **`mtx`** je zamknut.
- Co podobného mají pthreads?

Podmínky

- Třída **`condition_variable`**
- **`#includlde <condition_variable>`**
- Metody
  - **`wait`** – odpovídá **`pthread_condition_wait`**
  - **`notify_all`** – odpovídá **`pthread_condition_broadcast`**
  - **`notify_one`** – odpovídá **`pthread_condition_signal`**

Zamykání

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4  #include <condition_variable>
5
6  std::mutex mtx;
7  std::condition_variable cv;
8  bool ready = false;
9
10 void print_id (int id)
11 {
12         std::unique_lock<std::mutex> lck(mtx);
13         while (!ready) cv.wait(lck);
14         std::cout << "thread␣" << id << '\n';
15 }
16
17 void go()
18 {
19         std::unique_lock<std::mutex> lck(mtx);
20         ready = true;
21         cv.notify_all();
22 }
```

Zamykání

```
1  int main ()
2  {
3          std::thread threads[10];
4          for (int i=0; i<10; ++i)
5                 threads[i] = std::thread(print_id,i);
6
7          std::cout << "10 threads ready to race...\n";
8          go();
9
10         for (auto& th : threads) th.join();
11
12         return 0;
13 }
```

Atomické typy

- Šablona **`atomic`**
- **`#includlde <atomic>`**
- V C++ šablona akceputuje libovolný typ (včetně objektů a-la 1MB)
- Operátory
  - Přiřazení **=**
  - **++**
  - **--**
  - Přiřazení s operátorem např. **+=**

Atomické typy

- Metody
  - **`is_lock_free`** – vrací true, pokud lze použít lock free mechanismus
  - **`exchange`** – atomický swap
  - **`load`** – atomické čtení objektu
  - **`store`** – atomický zápis objektu
    - Pro **`load`** a **`store`** lze specifikovat uspořádání operací
    - **`memory_order_relaxed`** – žádné bariéry
    - **`memory_order_consume`** – synchronizace proti závislým datům z **`memory_order_seq_cst`**
    - **`memory_order_acquire`** – synchronizace proti všem datům z **`memory_order_seq_cst`**
    - **`memory_order_seq_cst`** – úplná bariéra

Atomické typy

```
1  #include <iostream>
2  #include <thread>
3  #include <atomic>
4
5  std::atomic<int> x;
6
7  void foo()
8  {
9          for(int i = 0; i < 1000000; i++) {
10                 // x=x+1 <-- does NOT work!
11                 x+=1;
12         }
13         std::cout << x << "\n";
14 }
15
16 int main()
17 {
18         x.store(0, std::memory_order_relaxed);
19         std::thread first(foo);
20         std::thread second(foo);
21         first.join();
22         second.join();
23         std::cout << "foo completed with " << x << "\n";
24         return 0;
25 }
```

Futures

- Šablona **`future`**
- **`#includle <future>`**
- Metody
  - **`get`** – vrátí hodnotu future, lze volat nejvýše jednou
  - **`valid`** – vrací true pokud je možné zavolat **`get()`**, tj. future objekt je platný
  - **`wait`** – čeká (a blokuje) dokud není k dispozici výsledek
  - **`wait_for`** – čeka (a blokuje) dokud není k dispozici výsledek nebo nevyprší časový limit
  - **`share`** – vrátí sdílenou instanci future **`get()`** lze pak volat vícekrát

Futures

```
1  #include <iostream>
2  #include <future>
3  #include <chrono>
4
5  bool is_prime (int x)
6  {
7          for (int i=2; i<x; ++i) if (x%i==0) return false;
8                  return true;
9  }
10
11 int main ()
12 {
13         std::future<bool> fut = std::async (is_prime,444444443);
14
15         std::cout << "checking, please wait";
16         std::chrono::milliseconds span (100);
17         while (fut.wait_for(span)==std::future_status::timeout)
18                 std::cout << '.';
19
20         bool x = fut.get();
21
22         std::cout << "\n444444443 " << (x?"is":"is not") << " prime.\n";
23
24         return 0;
25 }
```

Futures

```
1  #include <iostream>
2  #include <future>
3
4  int get_value() { return 10; }
5
6  int main ()
7  {
8          std::future<int> fut = std::async (get_value);
9          std::shared_future<int> shfut = fut.share();
10
11         std::cout << "value: " << shfut.get() << '\n';
12         std::cout << "its double: " << shfut.get()*2 << '\n';
13
14         return 0;
15 }
```

Způsoby vykonávání futures

- `async`
- `promise::get_future`
- `packaged_task::get_future`

Async

- Asynchronní volání funkce
- **`std::async(funkce, [argumenty, [...]])`**
- Návratový typ je **`future`** objekt

promise::get_future

- **`promise`** je objekt, který umožňuje uchovávat hodnoty, které obdrží **`future`** objekt
- Hodnota pro **`future`** je explicitně nastavena metodou **`promise::set_value`**
- Konstrukcí jde o jakési future "naruby"

promise::get_future

```
1  #include <functional>
2  #include <thread>
3  #include <future>
4  #include <iostream>
5
6  void print_int (std::future<int>& fut)
7  {
8          int x = fut.get();
9          std::cout << "value: " << x << '\n';
10 }
11
12 int main ()
13 {
14         std::promise<int> prom;
15
16         std::future<int> fut = prom.get_future();
17
18         std::thread th1 (print_int, std::ref(fut));
19
20         prom.set_value (10);
21
22         th1.join();
23         return 0;
24 }
```

packaged_task::get_future

- **`packaged_task`** je objekt, který obaluje funkce a umožňuje je volat asynchronně
- Místo kopírovacího konstruktoru lze použít *move* konstrukci **`std::move`**

packaged_task::get_future

```
1  #include <iostream>
2  #include <utility>
3  #include <future>
4  #include <thread>
5
6  int triple (int x) { return x*3; }
7
8  int main ()
9  {
10         std::packaged_task<int(int)> tsk (triple);
11
12         std::future<int> fut = tsk.get_future();
13         std::thread(std::move(tsk),33).detach();
14
15         int value = fut.get();
16
17         std::cout << "The␣triple␣of␣33␣is␣" << value << ".\n";
18
19         return 0;
20 }
```

Domácí úkol

- Vhodným způsobem paralelizujte hledání prvočísel pomocí Eratosthenova síta
- Lze využít sekvenční verz z těchto slídů, musí však fungovat i pro velká čísla

```
1  void * test_prime(void *arg) {
2          long i,j, prime = *(long*)arg;
3          int sieve[prime];
4          for(i = 0; i < prime; i++) sieve[i] = 1;
5          for(i = 2; i < prime; i++) {
6                  if(sieve[i]) {
7                          printf("%ld, ", i);
8                          for(j=i*2; j < prime; j+=i) sieve[j] = 0;
9                  }
10         }
11         printf("\n"); return NULL;
12 }
```