C++; delete Java;

Část 7: vlákna a synchronizace

Kennny

srpen 2017

Vlákno

- `#include <thread>`
- konstruktor vytváří i spouští vlákno
- destruktor - pokud má objekt stále přiřazeno fyzické vlákno, hází výjimku
  - lze vyřešit např. užitím metody `detach()`
- lze předat vláknu parametry takřka 1:1 s konstruktorem `std::thread`
- konstruktor má jako první argument funkci, kterou má provádět; zbytek argumentů jsou argumenty této funkce pro dané vlákno
- metoda `join()` - "připojí" aktuální vlákno k tomuto

Vlákno

- Vytvoření vlákna a počkání na ukončení

```
// vytvori vlakno
std::thread vlakno(&zpracujFrontu, 5);

// pocka na jeho ukonceni
vlakno.join();

// napr. zde je uz bezpecne volat destruktor
```

- Vytvoření vlákna a "odtržení" od objektu, aby mohl být bezpečně volán destruktor

```
{
    // vytvori vlakno
    std::thread vlakno(&zpracujFrontu, 5);

    // odtrzeni od objektu
    vlakno.detach();
}
```

Explicitní předání řízení a spánek

- `std::this_thread::yield()` - vzdá se svého časového kvanta ve prospěch jiného vlákna (kterého, to řídí plánovač)
- `std::this_thread::sleep_for()` - spánek na určitou dobu
- `std::this_thread::sleep_until()` - spánek do určitého momentu

Vsuvka: std::chrono

- `#include <chrono>`
- modul STL pro práci s časovými úseky a body
- datové typy zaobaleny metodami podle jednotek
  - např. `std::chrono::milliseconds()`, `std::chrono::seconds()`
  - výsledný typ vždy `std::chrono::duration`
- od C++14 podporuje speciální literály pro eliminaci dlouhé syntaxe

```
using namespace std::chrono_literals;

std::this_thread::sleep_for(1250ms);
```

- spousty dalších věcí, ale pro nás důležité zatím jen tohle

Vsuvka: thread_local

- k dispozici je malé úložiště pro vlákno (zásobník)
- lze deklarovat proměnnou jako `thread_local`
- pak má každé vlákno svou vlastní instanci pod stejným jménem

```
thread_local int threadCounter = 0;
```

- platnost sdružená s kontextem vlákna

Příklad

- Prostor pro příklad 07_a_basics

std::mutex

- `#include <mutex>`
- klasický mutex pro ochranu kritické sekce
- varianty
  - `std::mutex` - klasický mutex
  - `std::recursive_mutex` - mutex který podporuje rekurzivní zamykání ze stejného vlákna
  - `std::timed_mutex` - mutex s podporou timeoutu při čekání
  - `std::shared_timed_mutex` - sdílený mutex s podporou timeoutu (např. čtenář-písař))
  - (`std::shared_mutex` - sdílený mutex); (C++17)
  - a další...

std::mutex

- zamykání by mělo být obaleno RAII zámkem
- dostupné zámky
  - `std::lock_guard` - jednoduchý lock/unlock zámek
  - `std::unique_lock` - exkluzivní zámek (lze odložit zamčení, pracovat se sdíleným mutexem, ..)
  - `std::shared_lock` - sdílený zámek (význam u shared mutexů)
- pro potřeby kurzu se omezíme na klasický mutex, `std::lock_guard` a `std::unique_lock`

Zamčení mutexu

- klasické jednoduché použití pro kritickou sekci chráněnou jedním mutexem

```
// sdileny mutex, deklarovan nekde viditelne
std::mutex mtx;

// scope kriticke sekce
{
    std::lock_guard<std::mutex> lck(mtx);

    // kriticka sekce
}
// uvolneni zamku s koncem platnosti (scope)
```

Zamčení více mutexů

- použití pro (vícenásobnou) kritickou sekci chráněnou více mutexy

```
// sdilene mutexy
std::mutex mtx1, mtx2;

// scope kriticke sekce
{
    std::unique_lock<std::mutex> lck1(mtx, std::defer_lock);
    std::unique_lock<std::mutex> lck2(mtx, std::defer_lock);

    // zatim nezamceno

    std::lock(lck1, lck2);

    // kriticka sekce
}
// uvolneni obou zamku s koncem platnosti (scope)
```

Příklad

- Prostor pro příklad 07_b_mutex

std::condition_variable

- `#include <condition_variable>`
- klasická podmínková proměnná
- k fungování potřebuje zámek a odpovídající mutex
- metody
  - `wait()` - čeká dokud není probuzena
  - `wait_for()` - čeká dokud není probuzena nebo neuplyne zadaná doba
  - `wait_until()` - čeká dokud není probuzena nebo do zadaného momentu
  - `notify_one()` - probudí jedno blokované vlákno
  - `notify_all()` - probudí všechny

std::condition_variable

- `wait()` jako argument bere zámek a volitelně i podmínku k probuzení ve formě funkce, lambda funkce nebo funktoru
- tato podmínka dokáže odstínit spurious wakeup

```
std::unique_lock<std::mutex> lck(mtx);

std::condition_variable cv;

cv.wait(lck, &canWakeUp);

...

bool canWakeUp()
{
    return (count == 0);
}
```

Příklad

- Prostor pro příklad 07_c_cond_variables

std::atomic

- `#include <atomic>`
- šablonový typ nad primitivními datovými typy
- zaručuje atomické provedení takto proveditelných operací
  - ++, −
  - +=, -=, &=, |=, ^=
- není potřeba explicitně nic zamykat

std::atomic

- všechny zmíněné operátory jsou přetíženy
- není potřeba speciálního zacházení

```
std::atomic<int> myCounter = 0;

myCounter++;
```

Příklad

- Prostor pro příklad 07_d_atomic

Konec 7. části

```
exit(0);
```