KIV/CPP – Programování v jazyce C++

08. Vlákna v C++, základní synchronizační primitiva, paralelní algoritmy

Martin Úbl

KIV ZČU

2023/2024

Vlákna

- vlákna jsou obecně a do hloubky probírány v jiných předmětech
- zde pokryjeme jen nutné minimum
- získání přehledu

Vlákna

- vlákno = samostatně vykonávaný tok instrukcí
- vlákna typicky mohou běžet paralelně
  - buď zdánlivě (střídají se – time-slicing, viz ZOS, OS)
  - nebo doopravdy (více jader CPU – viz ZOS, OS, PPR)
- typicky někdy chceme vlákna synchronizovat
  - vlákno čeká na dokončení práce jiného vlákna
  - vlákno čeká na kritickou sekci
    - pozn.: oblast kódu, ve které smí být nanejvýš jedno vlákno v jeden čas
  - atd...
- více viz KIV/ZOS, KIV/OS a KIV/PPR
- nebo od příštího akademického roku KIV/UPP
- my probereme jen tu jazykovou stránku věci

Vlákna

- `std::thread`
- `#include <thread>`
- třída, jejíž instance obalují právě jednu instanci vlákna
- nepodporuje copy sémantiku (nelze jen tak klonovat vlákna)
- jako první parametr přebírá funkci která se má vykonat
  - obyčejná funkce, metoda, lambda funkce, funktor, ...
- další parametry jsou parametry, které se do této funkce předají

```
void vlaknova_fce(int i) {
  // ...
}

std::thread thr(&vlaknova_fce, 4);
```

Vlákna

- metoda třídy jako parametr
- vlastně jen respektujeme `thiscall` konvenci a dodáme ukazatel na instanci třídy jako první parametr

```
void Worker::work(int i) {
  while (true) {
    //
  }
}

Worker w1;
std::thread thr(&Worker::work, &w1, 8);
```

- pozn.: pozor na životnost objektu `w1`, vlákno může teoreticky stále běžet i po opuštění scope

Vlákna

- destruktor `std::thread` předpokládá, že jsme vše korektně ukončili
  - to znamená přečíst si návratový kód vlákna
  - volání metody `join()`
  - zpravidla je dobrý nápad volat ještě předtím `joinable()` – tak se ujistíme, že si již někdo návratový kód nepřečetl
- bez toho to bude křičet destruktor (až za běhu)
- volá se `std::terminate` a aplikace spadne

```
std::thread thr(&do_some_work);

// ...

if (thr.joinable())
  thr.join();
```

Vlákna

- objekt vlákna se dá od zdroje „odpojit“
- `detach()`
- destruktor pak nevyžaduje přečtení návratové hodnoty
- může být ale problém s korektním ukončením
  - např. z funkce `main` už bychom se chtěli vrátit, ale vlákno pořád vesele běží

```
{
  std::thread thr(&do_some_work);

  thr.detach();
} // destruktor nekrici
```

Vlákna

- `std::jthread`
- v základu totéž, co `std::thread`
- ale...
  - destruktor automaticky provádí `join()`
  - standardní cesta pro ukončování vlákna sdíleným interrupt tokenem

```
std::jthread thr([] (std::stop_token st) {
  while (!st.stop_requested())
    work_work_work();
});

// nejaka prace

thr.request_stop(); // uz vlakno nechceme
thr.join();         // explicitne neni treba
```

Vlákna

- `std::jthread`
- pochopitelně lze docílit podobného efektu, kdybychom si stop token nahradili referencí na nějakou sdílenou proměnnou a použili obyčejný `std::thread`
- `std::jthread` jen tento flow standardizuje

Vlákna

- `std::jthread`
- lze přehledněji vyjádřit návaznost a posloupnost událostí
  - např. `task1` a `task2` mohou běžet paralelně, ale `task3` a `task4` závisí na jejich výsledku:

```
{
    std::jthread task1(&task_fnc1);
    std::jthread task2(&task_fnc2);
} // task1 a task2 bezi paralelne

// zde budou oba tasky hotove

{
    std::jthread task3(&task_fnc3);
    std::jthread task4(&task_fnc4);
} // task3 a task4 bezi paralelne
```

Futures

- *Futures*
- aneb když si nechceme špinit ruce se správou vláken
- mechanismy *futures* a *promises* jsou známé z jiných jazyků
  - např. JavaScript
- `#include <future>`

Futures

- `std::async`
- spustí funkci dodanou jako parametr
  - buď asynchronně (`std::launch::async`)
  - nebo odloženě (až si někdo vyžádá výsledek) (`std::launch::deferred`)
- samotné volání `async` ihned končí
- je vrácena instance `std::future` (šablonový typ)
  - šablonový typ je volen podle návratové hodnoty spouštěné funkce

Futures

- `std::future`
- šablonový typ
- slouží pro signalizaci (a předání výsledků) mezi vlákny
- v podstatě dvě základní metody:
  - `wait()` – zablokuje se, dokud není k dispozici výsledek
    - varianta `wait_for()` a `wait_until()`
  - `get()` – získá výsledek; pokud není k dispozici, volá `wait()`

Futures

- `std::async` a `std::future`

```
auto result = std::async(std::launch::async,
  []() {
    int x;

    // ...

    return x;
});

// nejaka dalsi prace ktera nevyzaduje vysledek

int big_result = local_result + result.get();
```

Futures

- `std::async` může být poněkud omezující
- může být potřeba použít `std::thread` kvůli lepší správě
- např. *thread pooling* a podobně
- pokud chceme i tak použít `std::future`, potřebujeme o mechanismus navíc
  - *promises*
  - `std::promise`
  - rovněž šablonový typ
  - poskytuje instance `std::future`
  - pracovní vlákno „pošle“ výsledek do promise
  - vnější vlákno si „vyzvedne“ výsledek z future
  - každé vlákno si může vyzvednout vlastní instanci future (a tedy si po vlastní ose počkat na výsledek)

Futures

- vnější kód – vytváří promise, vyzvedává future, čeká na výsledek

```
std::promise<int> res_promise;
std::future<int> local_future
             = res_promise.get_future();

std::thread thr(&ThreadFnc,
                std::move(res_promise));

// nejaka prace

int mega_result = local_result
                + local_future.get();

thr.join();
```

- pracovní vlákno – přebírá promise, nastavuje výsledek

```
void ThreadFnc(std::promise<int> pr) {
  // work

  pr.set_value(9001);

  // other work?
}
```

- `set_value()` signalizuje příslušné futures asociované s danou promise

Vlákna

- shrnutí první části
- `std::thread` – vlákno, nejnižší abstrakce, takřka neomezuje uživatele - vhodné pro dlouhou životnost
- `std::jthread` – varianta `std::thread`, lépe vyjadřuje návaznost úkolů, ale může být omezující
- `std::async` – asynchronní zpracování úkolu (hned nebo odloženě) – vhodné pro kratší úlohy, které nepotřebují složitější synchronizaci, než nad výsledkem výpočtu
- `std::promise` – prostředek pro synchronizaci vlákna produkujícího výsledek a vláken konzumujících výsledek
- `std::future` – konzumenti výsledku nad tímto objektem čekají a vyzvedají si výsledek

Vlákna

- instance současného vlákna: `std::this_thread`
- v kontextu vlákna lze nad tímto objektem volat několik statických metod
  - `sleep_for()` – uspí se na zadaný čas
  - `sleep_until()` – uspí se do zadaného času
  - `yield()` – vzdá se časového kvanta (viz jiné předměty)
  - `get_id()` – získá identifikátor vlákna (ne nutně skutečné ID ze systému)

```
using namespace std::chrono_literals;

std::this_thread::sleep_for(100ms);
```

Synchronizace

- C++11 přidalo rovněž základní synchronizační primitiva
- v podstatě jen obalují standardní primitiva
- C++ jen přidává další drobná omezení, aby byla zvýšena bezpečnost používání
- opět – do detailu vč. funkce v jiných předmětech (KIV/ZOS, OS, PPR, UPP)
- zde probereme spíše jazykové implikace

Synchronizace

- `std::mutex`
- `#include <mutex>`
- klasický systémový mutex
  - vzájemné vyloučení
  - ochrana kritické sekce
  - má vlastníka – odemknout může jen ten, kdo ho zamknul
- má svoje metody, ale ty by se neměly používat přímo
  - `lock()` – zamkne mutex; pokud je již zamčený, zablokuje se, dokud není uvolněný
  - `unlock()` – odemkne mutex; pokud není zamčený, nebo ho zamknul někdo jiný, vyhodí chybu
  - `try_lock()` – pokusí se zamknout mutex; pokud se nepovede, hned se vrátí a indikuje neúspěch
- co když zapomeneme odemknout mutex?

Synchronizace

- `std::lock_guard`
- RAII obal nad zámky (a mutexy)
- konstruktor zamyká
- destruktor odemyká

```
std::mutex mtx;

// pro demonstraci - scope
{
  std::lock_guard<std::mutex> lck(mtx);

  // kriticka sekce
}
```

Synchronizace

- např. metoda, která je celá kritickou sekcí

```
class Crit {
  protected:
    std::mutex mMtx;

  public:
    void work()
    {
      std::lock_guard<std::mutex> lck(mMtx);
      // kriticka sekce
    }

    // ...
};
```

Synchronizace

- pozn.: zde je jedno z mála využití klíčového slova `mutable`

```
class Crit {
  protected:
    mutable std::mutex mMtx;

  public:
    void work() const
    {
      std::lock_guard<std::mutex> lck(mMtx);
      // kriticka sekce
    }

    // ...
};
```

- další varianty RAII obalů nad zámky
- `std::unique_lock` – podobné jako `lock_guard`, ale lze ho zamknout odloženě a odemknout explicitně
- `std::shared_lock` – sdílený zámek např. pro přístup pro čtení
  - v kombinaci s `unique_lock` lze implementovat řešení problému „čtenář-písař“
- `std::scoped_lock` – RAII obal nad vícečetným zamykáním (potřebujeme více zámků naráz)

Synchronizace

- další varianty mutexů
- `std::recursive_mutex` – mutex který může vlákno zamknout vícekrát
- `std::timed_mutex` – mutex s timeoutem
- `std::shared_mutex` – sdílený mutex, realizace zmíněného problému „čtenář-písař“
- jejich kombinace

Synchronizace

- skupinové zamykání
- `std::lock`, `std::try_lock`
- na vstupu několik instancí zamykatelných objektů
- zamkne vždy všechny naráz
- pokud nějaký není k dispozici, tak všechny odemkne a zablokuje se dokud nebudou všechny k dispozici
  - popř. `std::try_lock` neblokuje a vrací chybu

Synchronizace

- podmínková proměnná
- `std::condition_variable`
- `#include <condition_variable>`
- opět pouhý obal nad systémovou podmínkovou proměnnou
- lze nad ní čekat
  - `wait()`
  - potřebuje instanci `std::unique_lock`, obsahuje vlastně kritickou sekci
  - odblokuje se, když někdo tutéž podmínkovou proměnnou signalizuje (notifikuje)
  - a nebo *spurious wakeup* (viz jiné předměty)
    - poskytuje možnost, jak se s tím vypořádat v podobě dodatečné podmínky

Synchronizace

- podmínková proměnná
- `std::condition_variable`
- lze čekat po určitý čas
  - `wait_for()`, `wait_until()`
  - vrací, zda byl čekající probuzen kvůli signalizaci nebo vypršení času
  - vhodné například pro implementaci timeoutu nad odloženou operací

Synchronizace

- podmínková proměnná
- `std::condition_variable`
- lze notifikovat (signalizovat) čekajícího
  - `notify_one()` – probudí jednoho čekajícího z fronty
  - `notify_all()` – probudí všechny čekající z fronty

Příklad - worker vlákno

```
std::mutex workMtx;
std::condition_variable workCv;
std::queue<Work> workQueue;
std::atomic<bool> running = true;

void process_inputs() {
  while (running) {
    std::unique_lock<std::mutex> lck(workMtx);

    workCv.wait(lck, []() {     // pocka na praci nebo na konec
      return !workQueue.empty() && running;
    });

    if (!running)               // byla signalizace kvuli
      break;                    // ukonceni?

    Work w = workQueue.top();   // vybereme z fronty praci
    workQueue.pop();

    lck.unlock(); // ted uz nam praci nikdo nevezme,
                  // muzeme odemknout
    process(w);   // a v klidu zpracovat,
  }               // abychom neblokovali zamek zbytecne
}
```

Synchronizace

- podmínková proměnná
- do detailu je subjektem jiných předmětů
- má mnohem více implikací a nástrah, než jsme zmínili

Synchronizace

- `std::atomic`
- šablonový datový typ s atomickými operacemi
- používá vlastnosti dané architektury
  - např. vlastnosti datových typů, transakční paměť, zamykání sběrnice, ...
- fallback na zámky, pokud není dostupná HW podpora dané atomické operace

```
std::atomic<int> i;

i++;
```

Synchronizace

- `std::call_once` + `std::once_flag`
- thread-safe jednorázové provedení akce
- podobného efektu lze docílit i kritickou sekcí a bool příznakem nebo atomickou proměnnou
  - tento přístup ale garantuje provedení skutečně dostupnými prostředky v systému přenositelně

```
std::once_flag fl;

for (int i = 0; i < 10; i++) {
  std::call_once(fl, [](){
    std::cout << "Hello␣once!" << std::endl;
  });
}
```

- pozn.: pochopitelně má význam hlavně ve vícevláknovém prostředí

Parallel STL

- od C++17 poskytuje standardní knihovna paralelní verze algoritmů `<algorithm>` knihovny
- jako první parametr stačí přidat `std::execution` příznak
- lze tak za předpokladu správného použití získat paralelní zpracování takřka zadarmo

```
std::for_each(vec.begin(), vec.end(),
              [](int a) {
  // seriove provedeni
});

std::for_each(std::execution::par_unseq,
              vec.begin(), vec.end(),
              [](int a) {
  // velmi pravdepodobne paralelni provedeni
});
```

Parallel STL

- *Execution policy*
- jakým způsobem se bude provádět daný algoritmus?
- 4 předdedinované hodnoty
  - `std::execution::seq` – sekvenční (sériové)
  - `std::execution::unseq` – nesekvenční (sériové)
  - `std::execution::par` – sekvenční paralelní
  - `std::execution::par_unseq` – nesekvenční paralelní

Parallel STL

- Parallel STL
- více v jiných předmětech
- zmíněným jednoduchým trikem jde pro vybrané druhy problémů získat urychlení v podstatě téměř zadarmo
- problém může nastat, pokud je potřeba větší míra synchronizace
- nebo když chceme mít kontrolu nad vlákny

Paralelizace

- Co C++ dál nabízí?
- atomické proměnné (`std::atomic`)
- semafory (`std::counting_semaphore`, `std::binary_semaphore`)
- závory a bariéry (`std::latch`, `std::barrier`)
- odkládání zamykání
- získání informací o prostředí
  - počet CPU jader
  - zarovnání pro zamezení falešnému sdílení (využití cache) apod.
- a další...

Paralelizace

- pod pokličkou jde stále o prostředky systému
- např. implementace vláken jsou na Linuxu/Mac stále pthreads
  - proto je potřeba stále přidávat ke kompilaci příznak `-lpthread` (nebo obdobu)
- zdaleka jsme neprobrali problematiku vláken a synchronizace
- jen jsme představili povrchně prvky jazyka a knihovny, díky kterým lze realizovat paralelní program
- více o principech paralelizace v KIV/ZOS, KIV/OS, KIV/PPR a KIV/UPP