Vlákna a přístup ke sdílené paměti

27. února 2019

B4B36PDV – Paralelní a distribuované výpočty

Minulé cvičení: *“Paralelizace nám může pomoct…”*

Minulé cvičení: *“Paralelizace nám může pomoct...”*

B4B36PDV: *“Ale ne všechny přístupy vedou ke stejně dobrým výsledkům!”*

Minulé cvičení: *"Paralelizace nám může pomoct..."*

B4B36PDV: *"Ale ne všechny přístupy vedou ke stejně dobrým výsledkům!"*

Dnešní menu: Vlákna a jejich synchronizace

Osnova

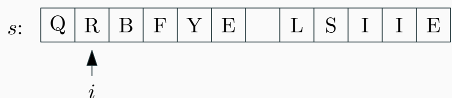
- Opakování z minulého cvičení
- Vlákna v C++ 11
- Přístup ke sdílené paměti a synchronizace

- Zadání první domácí úlohy

http://goo.gl/a6BEMb

Šifra PDVCrypt

Vzpomeňte si na šifru z minulého cvičení



Jeden krok dešifrování:

- $s_i \leftarrow \left[s_i + p_1 \times secret(\overbrace{s_{[i-2..i+2]}}^{\text{EQRBF}})\right] \bmod |\Sigma|$
- $i \leftarrow \left[i + p_2 \times secret(s_{[i-2..i+2]})\right] \bmod |s|$

… opakován N-krát

Šifra PDVCrypt

Jak vypadala paralelizace v OpenMP?

```
void decrypt_openmp(const PDVCrypt &crypt,
        std::vector<std::pair<std::string, enc_params>> &encrypted,
        unsigned int numThreads) {
    const unsigned long size = encrypted.size();

    #pragma omp parallel for num_threads(numThreads)
    for(unsigned long i = 0 ; i < size ; i++) {
        auto & enc = encrypted[i];
        crypt.decrypt(enc.first, enc.second);
    }
}
```

Šifra PDVCrypt

```
#pragma omp parallel for num_threads(numThreads)
for(...) {
  ...
}
```

Co se ve skutečnosti stalo?

Vlákna v C++ 11

C++11 (přes `#include <thread>`) poskytuje multiplatformní přístup k práci s vlákny:

```cpp
#include <iostream>
#include <thread>

void dummy_thread(int id, int n) {
  std::cout << "Thread " << id << " prints " << n << "\n";
}

int main() {
  std::thread t1(dummy_thread, 1, 2);
  std::thread t2(dummy_thread, 2, 42);
  t1.join();
  t2.join();

  return 0;
}
```

Kompaktnější zápis pomocí lambda funkcí

```
#include <iostream>
#include <thread>

void dummy_thread(int id, int n) {
  std::cout << "Thread " << id << " prints " << n << "\n";
}

std::thread t1(dummy_thread, 1, 2);
std::thread t2([&] (int id, int n) {
  std::cout << "Thread " << id << " prints " << n << "\n";
}, 2, 42);
```

Lambda funkce (uvozená pomocí `[&]`) má navíc přístup ke všem lokálním proměnným.

- Nemusíme si je tak předávat například pointery na lokální proměnné jako argumenty, pokud s nimi chceme pracovat

Vyřešte úlohu pomocí vláken

Doimplementujte tělo metody `decrypt_threads_1` v souboru `decryption.cpp`. Spusťte `numThreads` vláken, kdy každé vlákno bude vykonávat funkci `process`.

Vyřešte úlohu pomocí vláken

Doimplementujte tělo metody `decrypt_threads_1` v souboru `decryption.cpp`. Spusťte `numThreads` vláken, kdy každé vlákno bude vykonávat funkci `process`.

Co je na této implementaci špatně?

Synchronizace vláken při přístupu ke sdílené paměti

Varianta opravy č.1: Mutex

Mutex nám umožňuje zabránit více vláknům využívat stejný zdroj současně.

- Mutex vlastní vždy pouze jedno vlákno a ostatní vlákna musí čekat (mutex = *mutually-exclusive*)

Varianta opravy č.1: Mutex

Mutex nám umožňuje zabránit více vláknům využívat stejný zdroj současně.

- Mutex vlastní vždy pouze jedno vlákno a ostatní vlákna musí čekat (mutex = *mutually-exclusive*)
- Můžeme tak naimplementovat kritickou sekci, kam může vstoupit jediné vlákno. V této sekci:
  - Zjistíme index, který máme zpracovat
  - Inkrementujeme hodnotu `i`

Varianta opravy č.1: Mutex

Mutex nám umožňuje zabránit více vláknům využívat stejný zdroj současně.

- Mutex vlastní vždy pouze jedno vlákno a ostatní vlákna musí čekat (mutex = *mutually-exclusive*)
- Můžeme tak naimplementovat kritickou sekci, kam může vstoupit jediné vlákno. V této sekci:
  - Zjistíme index, který máme zpracovat
  - Inkrementujeme hodnotu `i`

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m;
void dummy_thread() {
  std::cout << "Zde muze byt soucasne vice vlaken." << std::endl;
  {
    std::unique_lock<std::mutex> lock(m);
    std::cout << "Ale zde budu uplne sam..." << std::endl;
  }
  std::cout << "A tady opet nemusim byt sam...";
}
```

Varianta opravy č.1: Mutex

Doplňte mutex

Opravte metodu `decrypt_threads_1` za použití mutexu. Metodu `decrypt_threads_1` neupravujte, opravený kód zapište do metody `decrypt_threads_2`.

Pozor!

Použití mutexů skrývá hrozbu *dead-locků*. Kód musíme navrhovat tak, aby bylo garantované, že vlákno někdy mutex získá (a provede tak kritickou sekci). Jinak zůstane čekat navěky...

Varianta opravy č.2: Atomická proměnná

Pokud nám stačí v rámci kritické sekce provést *jednu* operaci nad *jednou* proměnnou, můžeme si vystačit s atomickou operací.

Příklady atomických operací:

- Inkrementování proměnné typu `int`
- Vynásobení proměnné typu `int` konstantou

Varianta opravy č.2: Atomická proměnná

Pokud nám stačí v rámci kritické sekce provést *jednu* operaci nad *jednou* proměnnou, můžeme si vystačit s atomickou operací.

Příklady atomických operací:

- Inkrementování proměnné typu `int`
- Vynásobení proměnné typu `int` konstantou

Jak na to v C++11: `#include <atomic>`

```
int x = 0;    →    std::atomic<int> x { 0 };
```

Varianta opravy č.2: Atomická proměnná

Nahraďte mutex atomickou proměnnou

Nahraďte mutex v `decrypt_threads_2` atomickou proměnnou. Nový kód zapište do funkce `decrypt_threads_3`.

Varianta opravy č.2: Atomická proměnná

Mutex vs. Atomická proměnná

Mutex je založený na systémovém volání jádra operačního systému

- To může být ale **drahé**!

Mutex vs. Atomická proměnná

Mutex je založený na systémovém volání jádra operačního systému

- To může být ale **drahé**!

Atomická proměnná je (většinou) implementovaná na hardwarové úrovni

- Speciální instrukce pro atomické operace nad některými typy

Mutex vs. Atomická proměnná

Mutex je založený na systémovém volání jádra operačního systému

- To může být ale **drahé**!

Atomická proměnná je (většinou) implementovaná na hardwarové úrovni

- Speciální instrukce pro atomické operace nad některými typy
- ⚠ **Nelze použít vždy!**
  Procesory zpravidla podoporují jenom základní typy.

Varianta opravy č.3: Disjunktní rozsahy

I atomická proměnná má ale nějakou režii...

Nemůžeme se vyhnout použití mutexů
a atomických proměnných úplně?

Varianta opravy č.3: Disjunktní rozsahy

I atomická proměnná má ale nějakou režii…

Nemůžeme se vyhnout použití mutexů
a atomických proměnných úplně?

Doplňte logiku výpočtu rozsahů

Ve funkci `decrypt_threads_4` chybí implementace výpočtu rozsahu `t`-tého vlákna. Doplňte výpočet hodnot proměnných `begin` a `end`.

Podmínkové proměnné

Jaký je problém následujícího kódu?

```
void logger() {
  bool last_value = true;
  while(true) {
    std::unique_lock<std::mutex> lock(m);
    if(last_value != value) {
      std::cout << "Value changed to " << value << std::endl;
      last_value = value;
    }
  }
}
```

Jaký je problém následujícího kódu?

```
void logger() {
  bool last_value = true;
  while(true) {
    std::unique_lock<std::mutex> lock(m);
    if(last_value != value) {
      std::cout << "Value changed to " << value << std::endl;
      last_value = value;
    }
  }
}
```

Vlákno které čeká na splnění podmínky **vytěžuje procesor** (tzv. *busy waiting*)!

Podmínkové proměnné

Podmínkové proměnné (`#include <condition_variable>`) slouží ke komunikaci mezi vlákny

- Umožňují nám čekat na splnění podmínky jiným vláknem (a na signál od něj)

Podmínkové proměnné

Podmínkové proměnné (`#include <condition_variable>`) slouží ke komunikaci mezi vlákny

- Umožňují nám čekat na splnění podmínky jiným vláknem (a na signál od něj)

- Vytvoření podmínkové proměnné:
  `std::condition_variable cv;`

Podmínkové proměnné

Podmínkové proměnné (`#include <condition_variable>`) slouží ke komunikaci mezi vlákny

- Umožňují nám čekat na splnění podmínky jiným vláknem (a na signál od něj)

- Vytvoření podmínkové proměnné:
  ```
  std::condition_variable cv;
  ```
- Čekání na splnění podmínky:
  ```
  cv.wait(lock, [&] { return value != last_value; });
  ```

Podmínkové proměnné

Podmínkové proměnné (`#include <condition_variable>`) slouží ke komunikaci mezi vlákny

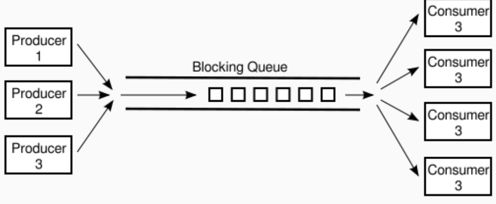
- Umožňují nám čekat na splnění podmínky jiným vláknem (a na signál od něj)

- Vytvoření podmínkové proměnné:
  ```
  std::condition_variable cv;
  ```
- Čekání na splnění podmínky:
  ```
  cv.wait(lock, [&] { return value != last_value; });
  ```
- Notifikace o změně stavu:
  ```
  cv.notify_one();
  cv.notify_all();
  ```

Zadání první domácí úlohy

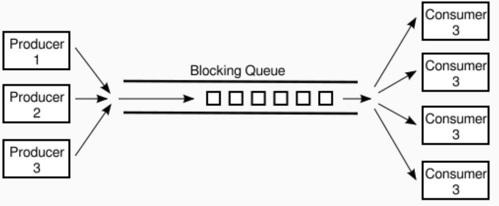
Producent – konzument

- Producent vyrábí určitá data a vkládá je do fronty
- Konzument je zase z fronty odebírá
- Každý pracuje svým tempem



Producent – konzument

- Producent vyrábí určitá data a vkládá je do fronty
- Konzument je zase z fronty odebírá
- Každý pracuje svým tempem



Proč bychom něco takového chtěli dělat?

Doimplementujte metody v `ThreadPool.h` a zajistěte, že

1. Výpočet úloh je paralelní a každá úloha (přidaná pomocí metody `process`) je zpracována právě jednou (1 bod)
2. Thread pool nečeká na přidání nových úloh pomocí busy-waitingu (1 bod)

`main.cpp`, `ThreadPool.h` / `make threadpool` `hw01_threadpool.zip`

Díky za pozornost!

Budeme rádi za Vaši zpětnou vazbu! $\rightarrow$

http://bit.ly/2VmAZjB