Vlákna a přístup ke sdílené paměti

Cvičení 2

B4B36PDV – Paralelní a distribuované výpočty
FEL ČVUT

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 cvičení:

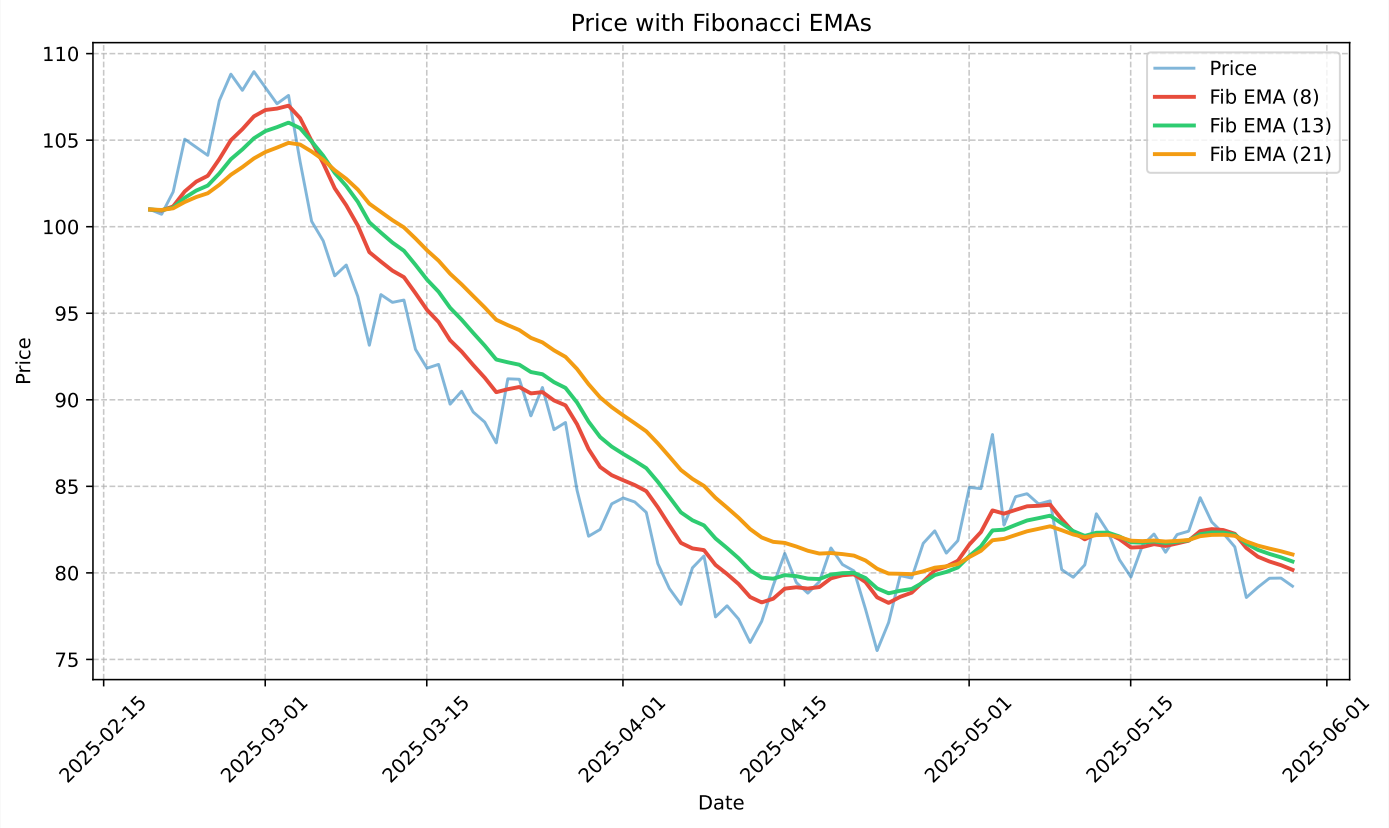
Vlákna a jejich synchronizace

Osnova

- Opakování z minulého cvičení
- Vlákna v C++20
- Přístup ke sdílené paměti
- Zadání první domácí úlohy

http://goo.gl/a6BEMb

Exponenciální klouzavý průměr



$$\mathrm{EMA}_0 = \mathrm{price}_0$$

$$\mathrm{EMA}_i = \mathrm{price}_i * \mathrm{k} + \mathrm{EMA}_{i-1} * (1 - \mathrm{k})$$

Jak vypadala paralelizace v OpenMP?

```
#pragma omp parallel for
for (size_t i = 0; i < stock_prices.size(); i++) {
  ema[i] = exponential_moving_average(stock_prices[i], 0.1);
}
```

Parallel for

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

Co se ve skutečnosti stalo?

Vlákna v C++20

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() {
 auto t1 = std::thread(dummy_thread, 1, 2);
 auto t2 = std::thread(dummy_thread, 2, 42);
 t1.join();
 t2.join();
}
```

C++20 (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() {
 auto t1 = std::jthread(dummy_thread, 1, 2);
 auto t2 = std::jthread(dummy_thread, 2, 42);
}
```

Kompaktnější zápis pomocí lambda funkcí

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

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

int main() {
 auto t1 = std::jthread(dummy_thread, 1, 2);
 auto t2 = std::jthread([&](auto id, auto 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.

Operace `map`

`Map` aplikuje funkci na každý prvek v poli.

```
void map_sequential(std::vector<float>& data, MapFn map_fn) {
  for (float& f : data) {
    f = map_fn(f);
  }

  // C ekvivalent:
  for (size_t i = 0; i < data.size(); i++) {
    data[i] = map_fn(data[i]);
  }
}
```

Operace `map` je ideální pro paralelizaci!

Soubor `1threads.cpp`

```
void map_openmp(std::vector<float>& data, MapFn map_fn);

void map_manual(std::vector<float>& data, MapFn map_fn);
```

Soubor `1threads.cpp`

```
void map_openmp(std::vector<float>& data, MapFn map_fn);

void map_manual(std::vector<float>& data, MapFn map_fn);
```

Co je na naší manuální implementaci paralelizace 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`

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

Varianta opravy č.1: Mutex

Doplňte mutex

Doimplementujte metodu `map_manual_locking` za použití mutexu.

Varianta opravy č.1: Mutex

Doplňte mutex

Doimplementujte metodu `map_manual_locking` za použití mutexu.

⚠ **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++20:

```
#include <atomic>
int x = 0;      →      std::atomic<int> x = 0;
```

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++20:

```
#include <atomic>
int x = 0;      →      std::atomic<int> x = 0;
```

Nahraďte mutex atomickou proměnnou

Doimplementujte metodu `map_manual_atomic` za použití atomické proměnné.

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

Doimplementujte metodu `map_manual_ranges` za použití disjunktních rozsahů.

Podmínkové proměnné

Jaký je problém následujícího programu?

```
void logger() {
 bool last_value = true;
 while(true) {
  auto lock = std::unique_lock(m);
  if(last_value ≠ value) {
   std::cout << "Value changed to " << value << "\n";
   last_value = value;
  }
 }
}
```

Jaký je problém následujícího programu?

```
void logger() {
 bool last_value = true;
 while(true) {
  auto lock = std::unique_lock(m);
  if(last_value ≠ value) {
   std::cout << "Value changed to " << value << "\n";
   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();
```

Čekání s podmínkovou proměnnou

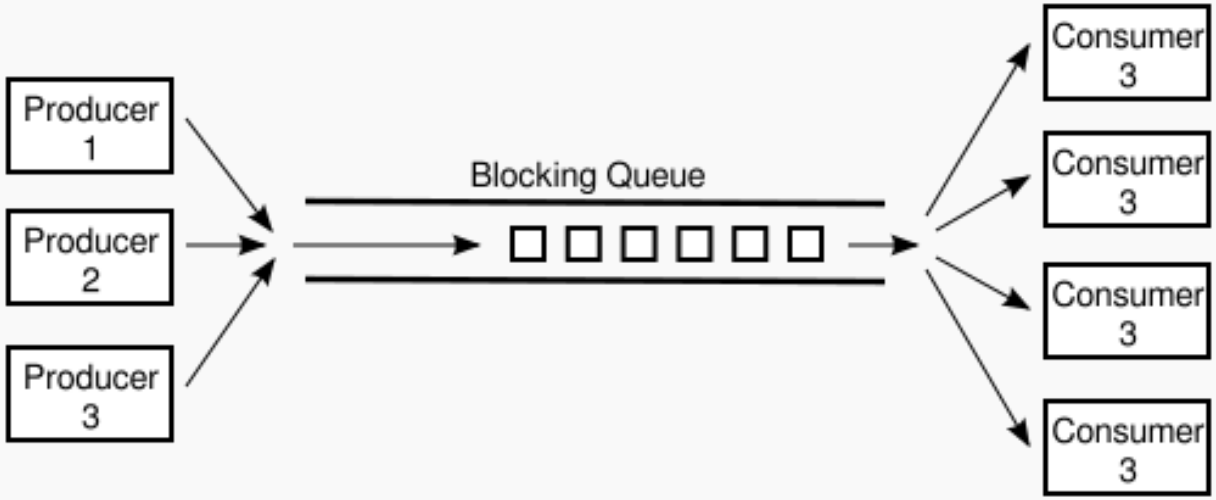
Nahraďte aktivní čekání podmínkovou proměnnou

V souboru `2conditional_variable.cpp` v metodách `logger` a `setter` nahraďte aktivní čekání podmínkovou proměnnou.

Zadání domácího úkolu

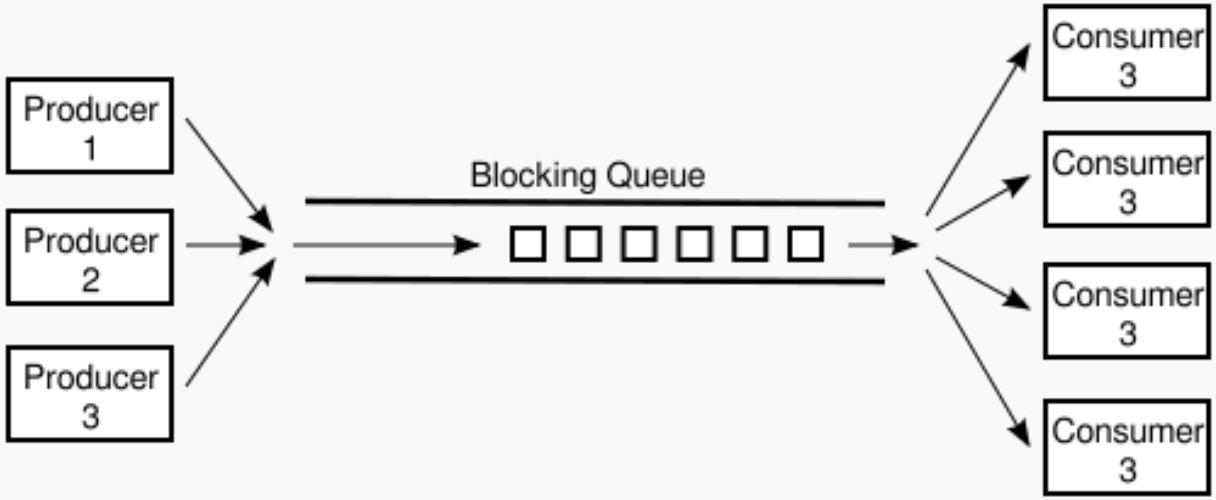
Producent – konzument

1. Producent vyrábí určitá data a vkládá je do fronty
2. Konzument je zase z fronty odebírá
3. Každý pracuje svým tempem



Producent – konzument

1. Producent vyrábí určitá data a vkládá je do fronty
2. Konzument je zase z fronty odebírá
3. Každý pracuje svým tempem



Co je na tomto přístupu zajímavé?

Zadání domácí úlohy

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)