

# Vlákna a přístup ke sdílené paměti

---

B4B36PDV – Paralelní a distribuované výpočty

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

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

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

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

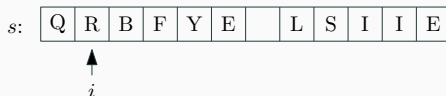
B4B36PDV: *“Ale ne všechny přístupy vedou  
ke stejně dobrým výsledkům!”*

Dnešní menu: Vlákna a jejich synchronizace

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

<http://goo.gl/a6BEMb>

Vzpomeňte si na šifru z minulého cvičení



Jeden krok dešifrování:

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

... opakován  $N$ -krát

Jak vypadala paralelizace v OpenMP?



Co se ve skutečnosti stalo?

## Vlákna v C++ 11

---

C++11 (přes ) poskytuje multiplatformní přístup k práci s vlákny:

---

Lambda funkce (uvozená pomocí `lambda`) 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

---

Mutex nám umožňuje zabránit více vláknům využívat stejný zdroj současně.

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



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`

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`

### Doplňte mutex

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

### 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...

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
- Vynásobení proměnné typu konstantou

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
- Vynásobení proměnné typu konstantou

---

Jak na to v C++11:

→

### Nahrad'te mutex atomickou proměnnou

Nahrad'te mutex v `decrypt_threads_2` atomickou proměnnou. Nový kód zapište do funkce `decrypt_threads_3`.



### 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 podporují jenom základní typy.

*I atomická proměnná má ale nějakou režii...*

Nemůžeme se vyhnout použití mutexů  
a atomických proměnných úplně?

*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?

Jaký je problém následujícího kódu?

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



Podmínkové proměnné () slouží ke komunikaci mezi vlákny

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

Podmínkové proměnné () slouží ke komunikaci mezi vlákny

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

- 
- Vytvoření podmínkové proměnné:

Podmínkové proměnné () slouží ke komunikaci mezi vlákny

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

- 
- Vytvoření podmínkové proměnné:
  - Čekání na splnění podmínky:

Podmínkové proměnné () slouží ke komunikaci mezi vlákny

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

- 
- Vytvoření podmínkové proměnné:
  - Čekání na splnění podmínky:
  - Notifikace o změně stavu:

Nahradte aktivní čekání podmínkovou proměnnou

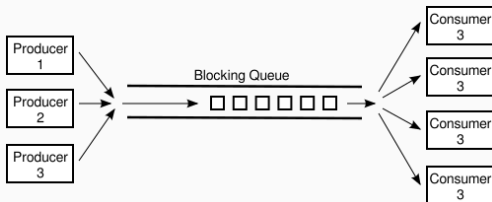
V souboru `conditional_variable.cpp` v metodách `logger` a `setter` nahradte aktivní čekání podmínkovou proměnnou.

## Zadání první domácí úlohy

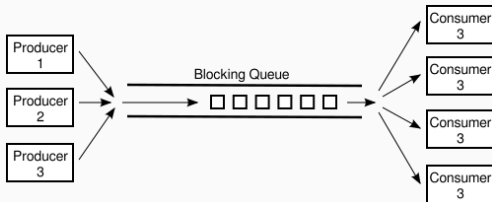
---

# Producent – konzument

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



- Producent 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)

Díky za pozornost!

Budeme rádi za Vaši  
zpětnou vazbu! →



[https://forms.gle/  
yi7FWBEw3mxgnJ9P7](https://forms.gle/yi7FWBEw3mxgnJ9P7)