

Konkurentní datové struktury

B4B36PDV – Paralelní a distribuované výpočty

Minulé cvičení:

“Paralelní programování v OpenMP..”

Minulé cvičení:

“Paralelní programování v OpenMP..”



Minulé cvičení:

“Paralelní programování v OpenMP..”



Minulé cvičení:

“Paralelní programování v OpenMP..”



Dnešní menu: Konkurentní datové struktury

- Opakování z minulého cvičení
- Zámková architektura datových struktur
- Bezzámková architektura datových struktur

- Zadání třetí domácí úlohy

Opakování z minulého cvičení

<http://goo.gl/a6BEMb>

Co provádí následující kód? Co bude po skončení v data?

```
unsigned int num_threads = omp_get_num_threads();
unsigned int thread_id = omp_get_thread_num();
std::vector<int> data(100000);
#pragma omp parallel
{
    int chunk_size = 1 + data.size() / num_threads;
    int begin = thread_id * chunk_size;
    int end = std::min ( data.size(),
                        (thread_id + 1) * chunk_size );
    for (unsigned int i = begin; i < end; i++)
        data[i] ++ ;
}
```

Napište odpověď

Jakým způsobem bude následující kód proveden?

```
std::vector<int> data(100000);  
int size = data.size();  
#pragma omp parallel  
{  
    #pragma omp parallel for  
    for (unsigned int i = 0; i < size; i++)  
        data[i] ++ ;  
}
```

Zvolte co se může stát

1. Kód nelze zkompileovat
2. OpenMP rozdělí práci na for cyklu mezi dostupná fyzická vlákna
3. Vnitřní smyčka bude provedena seriálně
4. OpenMP vytvoří více vláken než fyzicky lze a dojde k degradaci výkonu
5. For smyčka bude provedena každým vláknem celá

K čemu dojde u následujícího kódu?

```
int k = 0;
std::vector<int> data = getRandomVectorOfSize(400);
#pragma omp parallel num_threads(4)
{
    int begin = omp_get_thread_num() * 100;
    int end = (1 + omp_get_thread_num()) * 100;
    for (unsigned int i = begin; i < end; i++)
        #pragma omp critical
        k += data[i];
}
```

Zvolte co se může stát

1. Dojde k efektivní paralelizaci výpočtu
2. OpenMP zvládne distribuovat sčítání mezi vlákna aby nedošlo k degradaci výkonu
3. OpenMP bude zbytečně často serializovat vlákna pomocí critical
4. OpenMP bude naprosto zbytečně serializovat vlákna pomocí critical

Tvorba konkurentních datových struktur

Aby jednu strukturu používalo více vláken **současně**.

Co musíme změnit oproti frontě z prvního domácího úkolu?

Aby jednu strukturu používalo více vláken **současně**.

Co musíme změnit oproti frontě z prvního domácího úkolu?

- Nesmíme zamykat **celou** datovou strukturu!
- Se zamykáním zámků musíme šetřit

 Jinak se datová struktura stane brzdou výpočtu!

Možný přístup:

1. Vezmu kód existující jednovláknové datové struktury
2. Ve chvíli, kdy strukturu dělám nějaké zásahy, zamknu si část struktury pro sebe

Je to opravdu takto lehké?

Možný přístup:

1. Vezmu kód existující jednovláknové datové struktury
2. Ve chvíli, kdy strukturu dělám nějaké zásahy, zamknu si část struktury pro sebe

Je to opravdu takto lehké?

Typický vzor práce s jednovláknovými strukturami:


Příprava → “Poškození” dat  Oprava → Hotovo!

Možný přístup:

1. Vezmu kód existující jednovláknové datové struktury
2. Ve chvíli, kdy strukturu dělám nějaké zásahy, zamknu si část struktury pro sebe

Je to opravdu takto lehké?

Typický vzor práce s jednovláknovými strukturami:

Příprava → “Poškození” dat  Oprava → Hotovo!

Musíme zabránit použití “rozbité” části = vyloučit i čtenáře
(a zamykat si části struktury, i když to není potřeba – např. při čtení)

Možný přístup:

1. Vezmu kód existující jednovláknové datové struktury
2. Ve chvíli, kdy strukturu dělám nějaké zásahy, zamknu si část struktury pro sebe

Je to opravdu takto lehké?

Typický vzor práce s jednovláknovými strukturami:

Příprava → “Poškození” dat  Oprava → Hotovo!

Musíme zabránit použití “rozbité” části = vyloučit i čtenáře
(a zamykat si části struktury, i když to není potřeba – např. při čtení)

To si ale moc nepomůžeme :-)

Jak to tedy udělat lépe?

1. Strčit hlavu do písku a (téměř) nezamykat
2. Když nastane problém, tak ho (nějak) vyřešit

To se snáz řekne, než udělá...

Jak to tedy udělat lépe?

1. Strčit hlavu do písku a (téměř) nezamykat
2. Když nastane problém, tak ho (nějak) vyřešit

To se snáz řekne, než udělá...

Některé z možných problémů:

- **Datová struktura se může nacházet v mezistavu:**
Buď musí být použitelná, nebo si musíme být jistí, že problém detekujeme, než poškodíme data
- **Nesmíme uvolnit paměť, pokud s ní pracuje jiné vlákno:**
Složitější datové struktury často využívají techniky podobné garbage-collectoru v Javě.

O takových datových strukturách se těžko přemýšlí...

... a ještě hůř se v nich hledají chyby!

👁 <http://libcdfs.sourceforge.net>

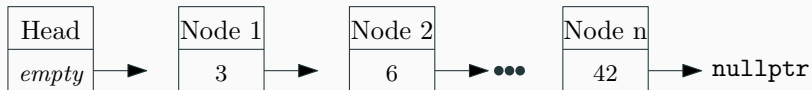
👁 C++ Concurrency In Action: Practical Multithreading

Cvičení: konkurenční spojový seznam

Konkurentní spojový seznam

Reprezentace seznamu prvků

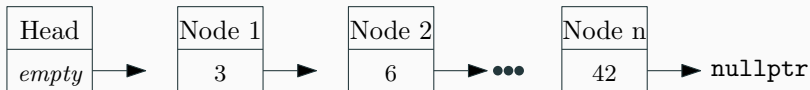
- My ho budeme chtít mít seřazený vzestupně...
- Vložení prvku = nalezení správné pozice + vložení nového uzlu



Konkurentní spojový seznam

Reprezentace seznamu prvků

- My ho budeme chtít mít seřazený vzestupně...
- Vložení prvku = nalezení správné pozice + vložení nového uzlu



Doimplementujte metodu `insert`

Doimplementujte tělo metody `insert` v souboru `lockBased.h`. Pro synchronizaci vláken použijte `spin_lock` (používá se stejně jako `std::mutex`), který umístíte ke každému uzlu seznamu. Snažte se zámky zamykat pouze na čas modifikace seznamu a pouze tam, kde jsou potřeba!

Nesynchronizované přístupy do paměti (s alespoň jedním zápisem) jsou ve standardu C++ vedené jako

 *undefined behavior* 

Může se nám stát spousta špatných věcí, například:
(ty navíc závisí na kompilátoru a platformě)

- Můžeme přečíst částečně zapsaná data (*dirty read*)
- Vlákno se nedozví o změně provedené jiným vláknem
- Vlákno se dozví pouze o části provedených změn

Nesynchronizované přístupy do paměti (s alespoň jedním zápisem) jsou ve standardu C++ vedené jako

 *undefined behavior* 

Může se nám stát spousta špatných věcí, například:
(ty navíc závisí na kompilátoru a platformě)

- Můžeme přečíst částečně zapsaná data (*dirty read*)
- Vlákno se nedozví o změně provedené jiným vláknem
- Vlákno se dozví pouze o části provedených změn

`std::atomic`

Synchronizace přístupů ke stejné proměnné zajištěna

Můžeme se zámků zbavit úplně?

Bezzámková architektura datových struktur

Navrhnout správně zamykání je náročné

- Špatné použití může vést k deadlocku
- Velké množství zámků snižuje potenciál opravdové konkurence
- Paměťový overhead (`std::mutex` na Linuxu má 40B!)

Navrhnout správně zamykání je náročné

- Špatné použití může vést k deadlocku
- Velké množství zámků snižuje potenciál opravdové konkurence
- Paměťový overhead (`std::mutex` na Linuxu má 40B!)

Jak na to?

Navrhnout správně zamykání je náročné

- Špatné použití může vést k deadlocku
- Velké množství zámků snižuje potenciál opravdové konkurence
- Paměťový overhead (`std::mutex` na Linuxu má 40B!)

Jak na to?

→ Pomocí atomických operací s pamětí

Klíčová operace pro *lock-free* datové struktury.

Porovnej a prohod' (neboli *compare-and-swap*) je atomická operace s paměti na objektu `std::atomic<T> X`, definovaná v C++ jako

```
bool X.compare_exchange_strong( T& expected, T desired)
```

která má funkcionalitu ekvivalentní

```
if ( X == expected ){ X = desired; return true; }  
else{ expected = X; return false; }
```


Klíčová operace pro *lock-free* datové struktury.

Porovnej a prohod' (neboli *compare-and-swap*) je atomická operace s paměti na objektu `std::atomic<T> X`, definovaná v C++ jako

```
bool X.compare_exchange_strong( T& expected, T desired)
```

která má funkcionalitu ekvivalentní

```
if ( X == expected ){ X = desired; return true; }  
else{ expected = X; return false; }
```

Kontrolu a změnu datové struktury lze provést **atomicky!**

Doimplementujte metodu `insert`

Doimplementujte tělo metody `insert` v souboru `lockFree.h`. Namísto použití zámků nyní použijte atomickou operaci *compare-and-swap* pro úpravu pointerů ve spojovém seznamu.

Doimplementujte metodu `insert`

Doimplementujte tělo metody `insert` v souboru `lockFree.h`. Namísto použití zámků nyní použijte atomickou operaci *compare-and-swap* pro úpravu pointerů ve spojovém seznamu.

Bonusové úlohy:

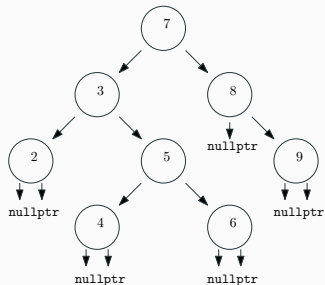
1. Zkuste se zamyslet, zda byste dokázali naimplementovat i dvousměrný spojový seznam
2. Zkuste naimplementovat spojový seznam, ve kterém může kromě přidávání docházet konkurentně i k mazání prvků

Zadání třetí domácí úlohy

Konkurentní binární vyhledávací strom

Struktura, v níž jsou jednotlivé prvky uspořádány tak, aby bylo možné rychle vyhledávat

- každý uzel tedy má nanejvýš dva potomky;
- každému uzlu je přiřazen určitý klíč;
- levý podstrom uzlu obsahuje klíče menší než je klíč uzlu; a
- pravý podstrom uzlu obsahuje klíče větší než je klíč uzlu.



Naimplementujte metody v `bst_tree.cpp` a `bst_tree.h` a zajistěte, že

1. každý prvek je vložen právě jednou; a
2. žádný vložený prvek se neztratí.

Zpracování musí být **konkurentní**, nikoli serielní!

Naimplementujte metody v `bst_tree.cpp` a `bst_tree.h` a zajistěte, že

1. každý prvek je vložen právě jednou; a
2. žádný vložený prvek se neztratí.

Zpracování musí být **konkurentní**, nikoli serielní!

Za správné výsledky a vysoký stupeň konkurence dostanete až **2b**.

Naimplementujte metody v `bst_tree.cpp` a `bst_tree.h` a zajistěte, že

1. každý prvek je vložen právě jednou; a
2. žádný vložený prvek se neztratí.

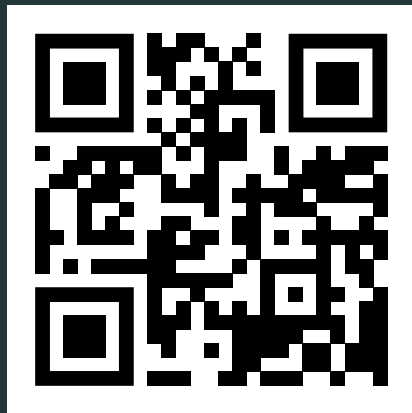
Zpracování musí být **konkurentní**, nikoli seriální!

Za správné výsledky a vysoký stupeň konkurence dostanete až **2b**.

Soubory `bst_tree.cpp` a `bst_tree.h` nahrajte do systému BRUTE.

Díky za pozornost!

Budeme rádi za Vaši
zpětnou vazbu! →



<http://bit.ly/2XTZhUo>