

Paralelní a distribuované výpočty (B4B36PDV)

Jakub Mareček, Michal Jakob

`jakub.marecek@fel.cvut.cz`

Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Dnešní přednáška

Souběžný přístup k datovým strukturám

Vícero vláken chce přistupovat ke společné datové struktuře

- např. zásobník, fronta, spojový seznam
- binární vyhledávací strom, jiné vyhledávací stromy
- haldy
- ...

Dnešní přednáška

Souběžný přístup k datovým strukturám

Vícero vláken chce přistupovat ke společné datové struktuře

- Potřebujeme zabezpečit korektní operace s datovou strukturou sdílenou několika vlákny
- Nechceme celou strukturu zamknout při každém přístupu, alebrž navrhnout co nejefektivnější práci více vláken nad společnou datovou strukturou bez zámků (tzv. lock-free datové struktury)
- Do C++17 to nebylo standardem. Od GCC 9.1 (květen 2019) a Microsoft Visual Studio 2017 standardní. Nezávisle: Intel Parallel STL, boost::compute, NVIDIA Thrust, atp.

Souběžný přístup k datovým strukturám

Co chceme dosáhnout?

- Hlavní myšlenka

Vlákna optimisticky předpokládají, že vše bude v pořádku
(modifikace DS budou konzistentní)

- ... ale nemůžeme se na to spolehnout, takže

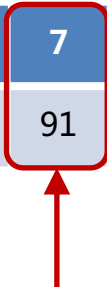
V případě detekce nekonzistence ji vlákno vyřeší/opraví

Souběžný přístup k datovým strukturám

Příklad 1: vyhledání maxima

- Vyhledání maxima v seznamu čísel
 - Chci najít maximální hodnotu a index na kterém se nachází
 - V případě rovnosti, chci co možná největší index

1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32



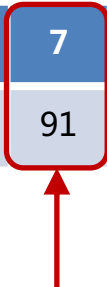
- Společná datová struktura
 - Dvě čísla – maximální hodnota & index

Souběžný přístup k datovým strukturám

Příklad 1 – vyhledání maxima

- Vyhledání maxima v seznamu čísel
 - Chci najít maximální hodnotu a index na kterém se nachází
 - V případě rovnosti, chci co možná největší index

1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32



- Společná datová struktura
 - Dvě čísla – maximální hodnota & index
 - Jedno číslo – index aktuální maximální hodnoty
- Jak na to?

Souběžný přístup k datovým strukturám

Příklad 1

- Vyhledání maxima v seznamu čísel
 - Chci najít maximální hodnotu a index na kterém se nachází
 - V případě rovnosti, chci co možná největší index
- První řešení - zámky

1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

max index

-1

Souběžný přístup k datovým strukturám

Příklad 1

- Vyhledání maxima v seznamu čísel
 - Chci najít maximální hodnotu a index na kterém se nachází
 - V případě rovnosti, chci co možná největší index
- První řešení - zámky

Vlákno 1					Vlákno 2				
1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

max index

-1

Souběžný přístup k datovým strukturám

Příklad 1

- Vyhledání maxima v seznamu čísel
 - Chci najít maximální hodnotu a index na kterém se nachází
 - V případě rovnosti, chci co možná největší index
- První řešení - zámky

Vlákno 1					Vlákno 2				
1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

max index

-1

Souběžný přístup k datovým strukturám

Příklad 1

- Vyhledání maxima v seznamu čísel
 - Chci najít maximální hodnotu a index na kterém se nachází
 - V případě rovnosti, chci co možná největší index
- První řešení - zámky

Vlákno 1					Vlákno 2				
1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

- vlákno 1 porovná hodnotu
- pokud je aktuální hodnota větší
 - zamkne max index
 - provede úpravu
 - odemkne max index

max index
1

Souběžný přístup k datovým strukturám

Příklad 1

- Vyhledání maxima v seznamu čísel
 - Chci najít maximální hodnotu a index na kterém se nachází
 - V případě rovnosti, chci co možná největší index
- První řešení - zámky

Vlákno 1					Vlákno 2				
1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

- vlákno 2 porovná hodnotu
- pokud je aktuální hodnota větší
 - zamkne max index
 - provede úpravu
 - odemkne max index

max index
6

Souběžný přístup k datovým strukturám

Příklad 1

- Vyhledání maxima v seznamu čísel
 - Chci najít maximální hodnotu a index na kterém se nachází
 - V případě rovnosti, chci co možná největší index
- První řešení - zámky

Vlákno 1					Vlákno 2				
1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

- vlákno 2 porovná hodnotu
- pokud je aktuální hodnota větší
 - zamkne max index
 - provede úpravu
 - odemkne max index

max index
6

Jak to bude fungovat?

Souběžný přístup k datovým strukturám

Příklad 1

- Může vzniknout nekonzistence

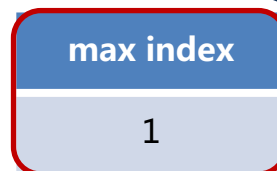
Vlákno 1					Vlákno 2				
1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

- vlákno 1 porovná hodnotu
- pokud je aktuální hodnota větší

- zamkne max index
- provede úpravu na hodnotu 1
- odemkne max index

- vlákno 2 porovná hodnotu
- pokud je aktuální hodnota větší

- zamkne max index
- provede úpravu na hodnotu 6
- odemkne max index



Konkurentní datové struktury

Příklad 1

- Může vzniknout nekonzistence

Vláknno 1					Vláknno 2				
1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

- vláknno 1 porovná hodnotu
- pokud je aktuální hodnota větší

- zamkne max index
- provede úpravu na hodnotu 1
- odemkne max index

- vláknno 2 porovná hodnotu
- pokud je aktuální hodnota větší

- zamkne max index
- provede úpravu na hodnotu 6
- odemkne max index

Výsledek může být nesprávný

max index
1

Souběžný přístup k datovým strukturám

Příklad 1

- možné řešení:
 - vlákna budou zamykat max index před kontrolou – pomalé ☹️
 - po získání zámku vlákno opět zkontroluje jestli je update aktuální

- vlákno 1 porovná hodnotu
- pokud je aktuální hodnota větší
 - zamkne max index
 - **zkontroluje, jestli je aktuální hodnota stále větší**
 - provede úpravu na hodnotu 1
 - odemkne max index

Vlákno 1					Vlákno 2				
1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

max index
1

Souběžný přístup k datovým strukturám

Příklad 1

- Jde to i bez zámků?
- K nekonzistenci může dojít mezi kontrolou jestli je aktuální hodnota větší než v maximu a případnou výměnou
 - Necht' je aktuální hodnota **max index** == 2 a vlákno 2 testuje podmínku
 - Výměnu vykoná pouze tehdy, pokud je **max index** pořád 2

Vlákno 1					Vlákno 2				
1	2	3	4	5	6	7	8	9	10
3	12	42	91	74	24	91	66	19	32

Můžeme použít atomické proměnné

max index
2

Parallel STL

Nová specifikace

Pro všechny algoritmy ve standardní šablonové knihovně bychom měli mít možnost specifikovat "Execution Policy Value" :

- seq: jako do C++17.
- par: paralelní verze s několika thready.
- unseq: vektorizace (SIMD), pokud jsou předané funkce SIMD-safe.
- par_unseq: paralelní a vektorizovaná verze.

Plus řada nových algoritmů (např. reduce, které odpovídá foldl v Haskellu).

Od verze GCC 9.1 je možné použít vestavěné paralelní STL, pokud člověk má nainstalované OpenMP 4.0 a Intel Threading Building Blocks (TBB) 2019.

```
sudo apt install gcc libtbb-dev  
g++ -ggdb3 -O3 -std=c++17 -Wall -Wextra -pedantic -o main.out  
main.cpp -ltbb ./main.out
```

<https://software.intel.com/content/www/us/en/develop/documentation/oneapi-dpcpp-library-guide/top/parallel-stl-overview.html>

Parallel STL

Nová specifikace

```
#include <algorithm>
#include <chrono>
#include <execution>
#include <iostream>
#include <random>
#include <vector>

using namespace std::chrono;

int main() {
    const int N = 1000000;
    std::vector<int> v(N);
    std::mt19937 rng;
    rng.seed(std::random_device()());
    std::uniform_int_distribution<int> dist(0, 255);
    std::generate(begin(v), end(v), [&]() { return dist(rng); });

    auto start = high_resolution_clock::now();
    std::sort(std::execution::par, begin(v), end(v));
    // std::reduce(std::execution::par, begin(v), end(v), 0.0, std::plus<>{});
    auto finish = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(finish - start);

    std::cout << "\nElapsed time = " << duration.count() << " ms\n";
    return 0;
}
```

Parallel STL

Nová specifikace

Intel® oneAPI Threading Building Blocks

Scalable Parallel Programming at Your Fingertips

Features

Documentation & Code Samples

Tutorials

Specifications

Help



Advanced Threading for Fast Applications

Intel® oneAPI Threading Building Blocks (oneTBB)[†] is a flexible performance library that simplifies the work of adding parallelism to complex applications across accelerated architectures, even if you're not a threading expert.

oneTBB is ideal for a wide range of compute-intensive domains, such as:

- Numeric weather prediction
- Oceanography
- Astrophysics

Develop in the Cloud

Get what you need to build, test, and optimize your oneAPI projects for free. With an Intel® DevCloud account, you get 120 days of access to the latest Intel® hardware—CPUs, GPUs, FPGAs—and Intel oneAPI tools and frameworks. No software downloads. No configuration steps. No installations.

[Get Access](#)

<https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html>

Compare and Swap

Hlavní nástroj

Hlavní nástroj: Atomické „compare and swap“ (CAS) atomicky porovná zda hodnota dereferencovaného ukazatele na atomickou proměnnou odpovídá očekávané hodnotě. Pokud ano, provede změnu na novou hodnotu. Typicky:

1. Deklarujeme atomickou proměnnou.
2. Uložíme její hodnotu atomické proměnné (např. dereferencováním ukazatele na atomickou proměnnou).
3. Na základě uložené hodnoty napočítáme novou hodnotu, kterou bychom chtěli uložit do atomické proměnné.
4. Použijeme CAS: Pokud dereferencováním ukazatele stále dostaneme uloženou hodnotu, přepíšeme uloženou hodnotu napočítanou hodnotou.
5. Pokud dereferencování ukazatele nevrací uloženou hodnotu, počkej (malé náhodné, ale rostoucí) a jdi na krok 2.

Compare and Swap

Hlavní nástroj

Hlavní nástroj: Atomické „compare and swap“ (CAS). V C++ v hlavičce `atomic` šablonové funkce:

- `bool compare_exchange_strong(atomic<T>* uka, T* chce, T val, std::memory_order ano, std::memory_order ne) noexcept;`
- `bool compare_exchange_weak(atomic<T>* uka, T* chce, T val, std::memory_order ano, std::memory_order ne) noexcept;`

kde „uka“ musí být ukazatel na atomický objekt,

kde nepovinné `std::memory_order` pro obě návratové hodnoty nabývá hodnot od `std::memory_order_relaxed` (bez záruk na vedlejší účinky) po `std::memory_order_seq_cst` (sekvenčně konsistentní)

kde funkce „weak“ mohou selhat ve smyslu, že i pokud „uka“ ukazuje na „chce“, můžeme občas dostat návratovou hodnotu `false`.

Compare and Swap

- Atomická operace **compare and swap** (CAS)
 - Atomicky porovná jestli hodnota proměnné odpovídá očekávané hodnotě a pokud ano, provede změnu na novou hodnotu

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
    for (int i=0; i<SIZE; ++i) {
        int tmp_index = atomic_max_index.load();
        while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
            !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
            tmp_index = atomic_max_index.load();
        }
    }
}
```

Compare and Swap

- Atomická operace **compare and swap** (CAS)
 - Atomicky porovná jestli hodnota proměnné odpovídá očekávané hodnotě a pokud ano, provede změnu na novou hodnotu
- V C++, `compare_exchange_strong`

podmínka pro maximum

```
std::atomic_int atomic_max_index;  
  
void find_max_cas(std::vector<int> & vector) {  
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)  
for (int i=0; i<SIZE; ++i) {  
int tmp_index = atomic_max_index.load();  
while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&  
!atomic_max_index.compare_exchange_strong(tmp_index,i)) {  
tmp_index = atomic_max_index.load();  
}  
}  
}
```

hodnota indexu

Compare and Swap

Příklad 1

- Atomická operace **compare and swap** (CAS)
 - Atomicky porovná jestli hodnota proměnné odpovídá očekávané hodnotě a pokud ano, provede změnu na novou hodnotu
- V C++, `compare_exchange_strong`

```
std::atomic_int atomic_max_index;
```

```
void find_max_cas(std::vector<int> & vector) {
```

```
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
```

```
for (int i=0; i<SIZE; ++i) {
```

```
int tmp_index = atomic_max_index.load();
```

```
while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
```

```
!atomic_max_index.compare_exchange_strong(tmp_index,i)) {
```

```
tmp_index = atomic_max_index.load();
```

```
}
```

```
}
```

```
}
```

compare and swap – pokud je v proměnné **atomic_max_index** hodnota **tmp_index** (kterou jsme použili), tak provedeme update

Compare and Swap

Příklad 1

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
  for (int i=0; i<SIZE; ++i) {
    int tmp_index = atomic_max_index.load();
    while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
           !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
      tmp_index = atomic_max_index.load();
    }
  }
}
```

výsledek je **true** pokud se operace povede, **false** v opačném případě

Compare and Swap

Příklad 1

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
  for (int i=0; i<SIZE; ++i) {
    int tmp_index = atomic_max_index.load();
    while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
           !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
      tmp_index = atomic_max_index.load();
    }
  }
}
```

Pokud je výsledek **false**, jiné vlákno mezitím změnilo **atomic_max_index**. Musíme aktualizovat hodnotu **tmp_index** a provést kontrolu znovu.

výsledek je **true** pokud se operace povede, **false** v opačném případě

Compare and Swap

Příklad 1

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
  for (int i=0; i<SIZE; ++i) {
    int tmp_index = atomic_max_index.load();
    while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
           !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
      tmp_index = atomic_max_index.load();
    }
  }
}
```

Pokud je výsledek **false**, jiné vlákno mezitím změnilo **atomic_max_index**. Musíme aktualizovat hodnotu **tmp_index** a provést kontrolu znovu.

výsledek je **true** pokud se operace povede, **false** v opačném případě

Kontrola je ve **while cyklu!**
(ne Konzistence se může vyskytnout opakovaně)

Souběžný přístup k datovým strukturám

Příklad 2: zásobník

Zásobník

```
struct Node {  
    int value = 0;  
    Node* successor = nullptr;  
  
    Node(int _value, Node* _successor) : value(_value), successor(_successor) {}  
};
```



Kritické místo je při vkládání a odebírání do/z vrcholu zásobníku

Pro jednoduchost pracujme jen se zásobníkem celých čísel.

Souběžný přístup k datovým strukturám

Příklad 2

Zásobník



Řešení pomocí zámků

```
void add_to_stack_locks(int new_value) {
    m.lock();
    head = new Node(new_value, head);
    m.unlock();
}

int pop_from_stack_locks() {
    m.lock();
    if (head == nullptr) {
        m.unlock();
        throw std::out_of_range("The stack is empty.");
        return -1;
    } else {
        Node* tmp = head;
        int val = head->value;
        head = head->successor;
        delete tmp;
        m.unlock();
        return val;
    }
}
```

Souběžný přístup k datovým strukturám

Příklad 2

Řešení pomocí atomických proměnných



```
std::atomic<Node*> head2;

void add_to_stack_cas(int new_value) {
    Node* p = new Node(new_value, head2.load());
    while (!head2.compare_exchange_strong(p->successor, p)) {
        p->successor = head2.load();
    }
}

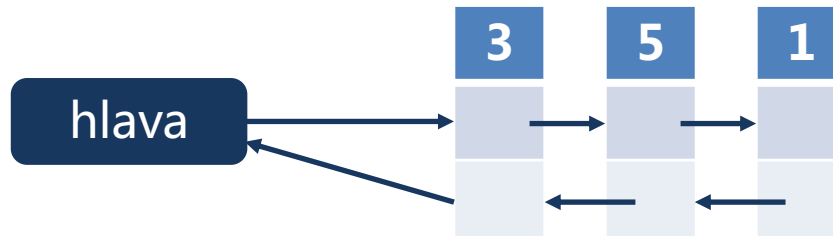
int pop_from_stack_cas() {
    if (head2.load() == nullptr) {
        throw std::out_of_range("The stack is empty.");
        return -1;
    } else {
        Node* h = head2.load();
        while (!head2.compare_exchange_strong(h, h->successor)) {
            h = head2.load();
        }
        int val = h->value;
        delete h;
        return val;
    }
}
```

Jak to bude fungovat?

Souběžný přístup k datovým strukturám

Příklad 3

- Obousměrný spojový seznam



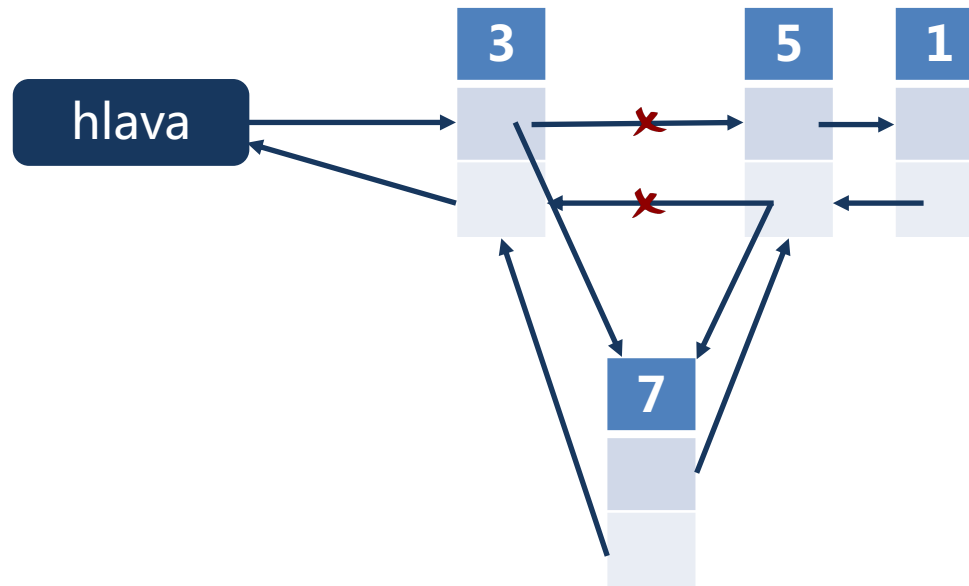
Operace přidání



Souběžný přístup k datovým strukturám

Příklad 3

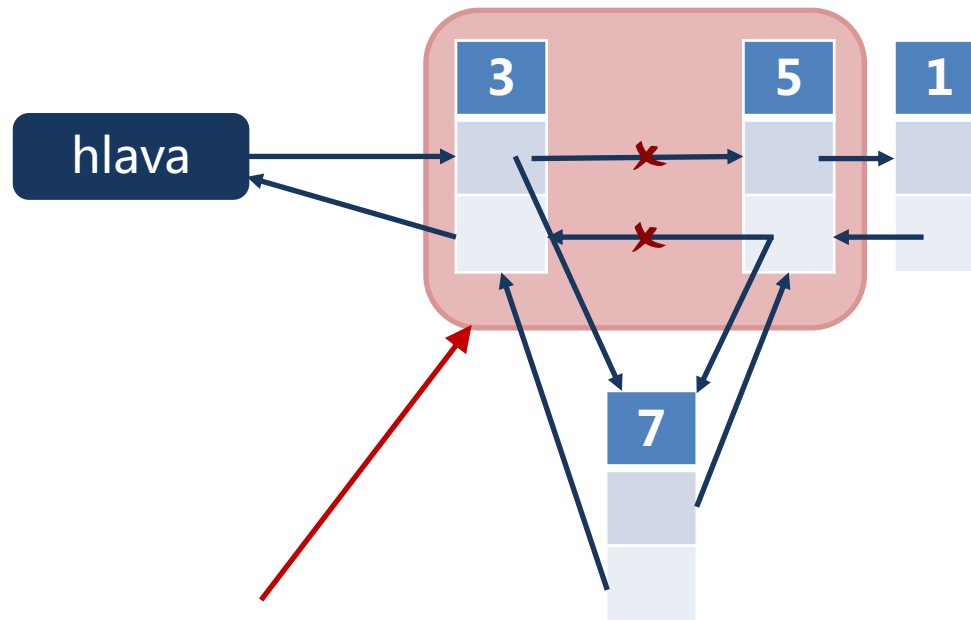
- Operace přidání pro obousměrný spojový seznam



Souběžný přístup k datovým strukturám

Příklad 3

- Operace přidání pro obousměrný spojový seznam

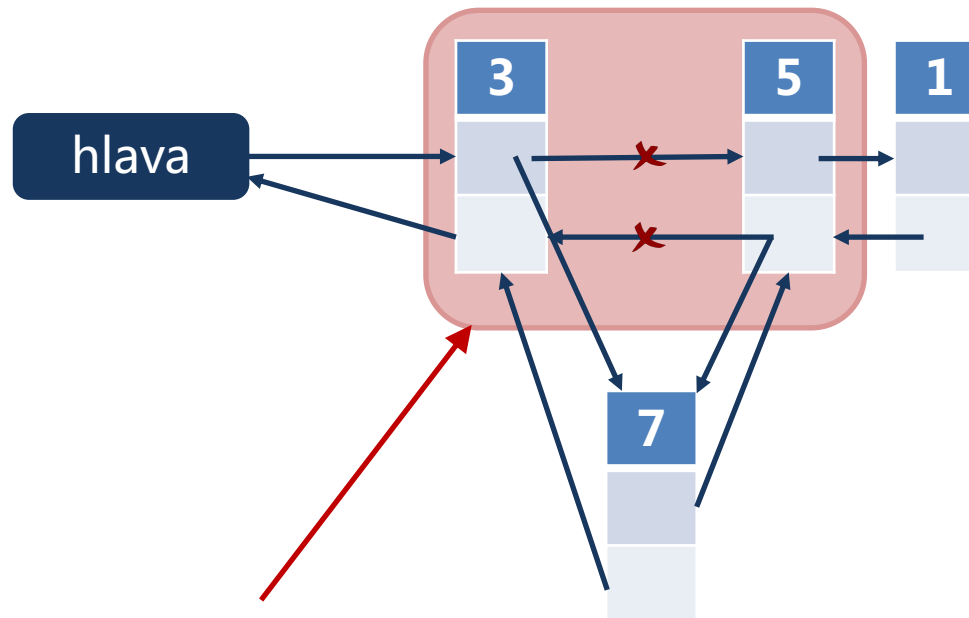


musíme zamknout oba prvky

Souběžný přístup k datovým strukturám

Příklad 3

- Operace přidání pro obousměrný spojový seznam



musíme zamknout oba prvky

Musíme zamykat oba současně nebo stačí vždy nejdřív prvek blíž k hlavě a pak jeho následovníka?

Souběžný přístup k datovým strukturám

Příklad 3

- Řešení pomocí zámků

```
struct Node {  
    std::mutex m;  
    int value = 0;  
    Node* successor = nullptr;  
    Node* predecessor = nullptr;  
    Node(int _value, Node* _predecessor, Node* _successor) :  
        value(_value), predecessor(_predecessor), successor(_successor) {}  
};
```

Souběžný přístup k datovým strukturám

Příklad 3

- Řešení pomocí zámků

```
Node* add_to_list_after(Node* _previous_node, int _new_value) {
    assert (_previous_node != nullptr);
    _previous_node->m.lock();

    Node* new_successor = _previous_node->successor;
    bool is_there_successor = new_successor != nullptr;

    if (is_there_successor) {
        new_successor->m.lock();
    }

    Node* new_node = new Node(_new_value, _previous_node, new_successor);

    if (is_there_successor)
        _previous_node->successor->predecessor = new_node;
    _previous_node->successor = new_node;

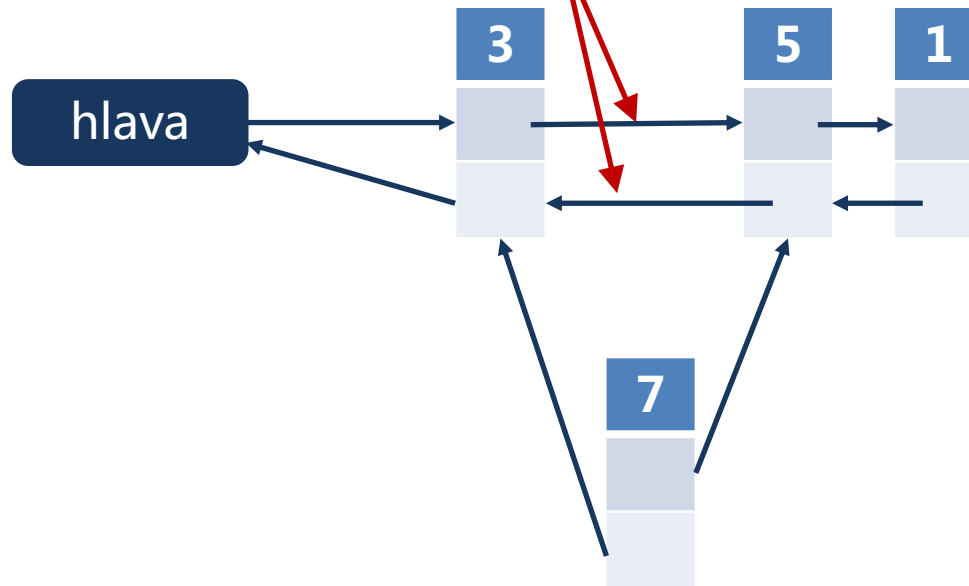
    if (is_there_successor)
        new_successor->m.unlock();
    _previous_node->m.unlock();
    return new_node;
}
```

Souběžný přístup k datovým strukturám

Příklad 3

- Jak řešit pomocí atomických operací?

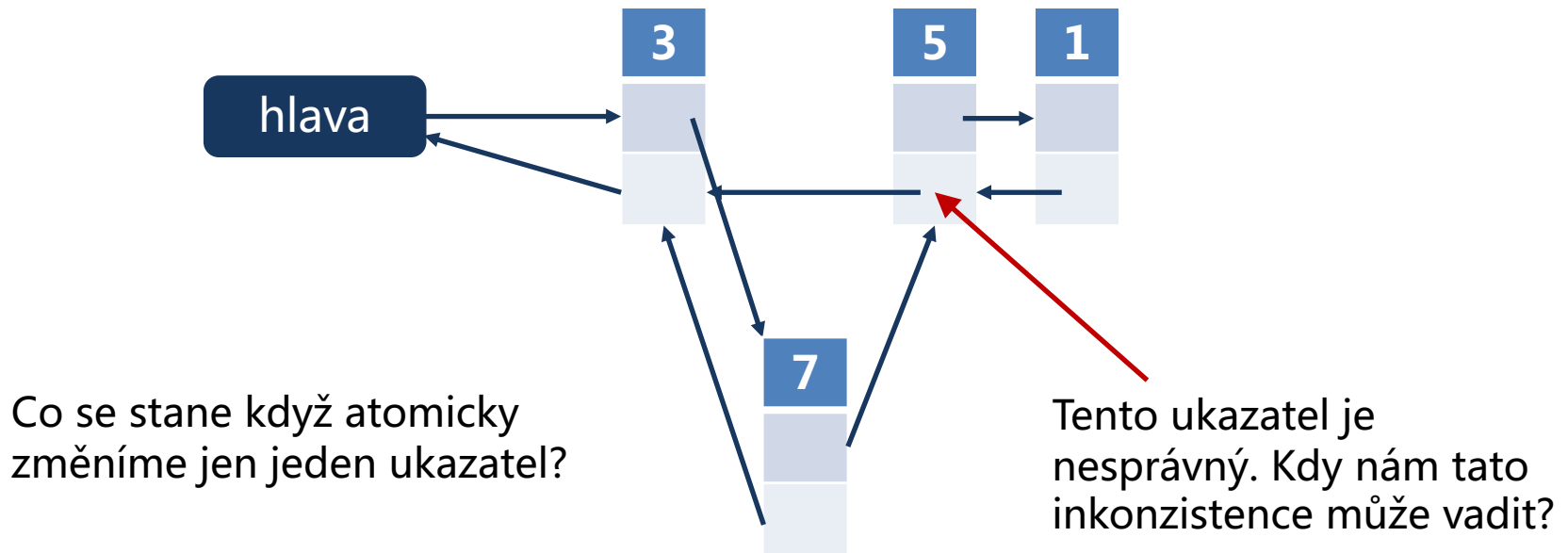
potřebovali bychom atomicky
změnit 2 ukazatele ... což nejde



Souběžný přístup k datovým strukturám

Příklad 3

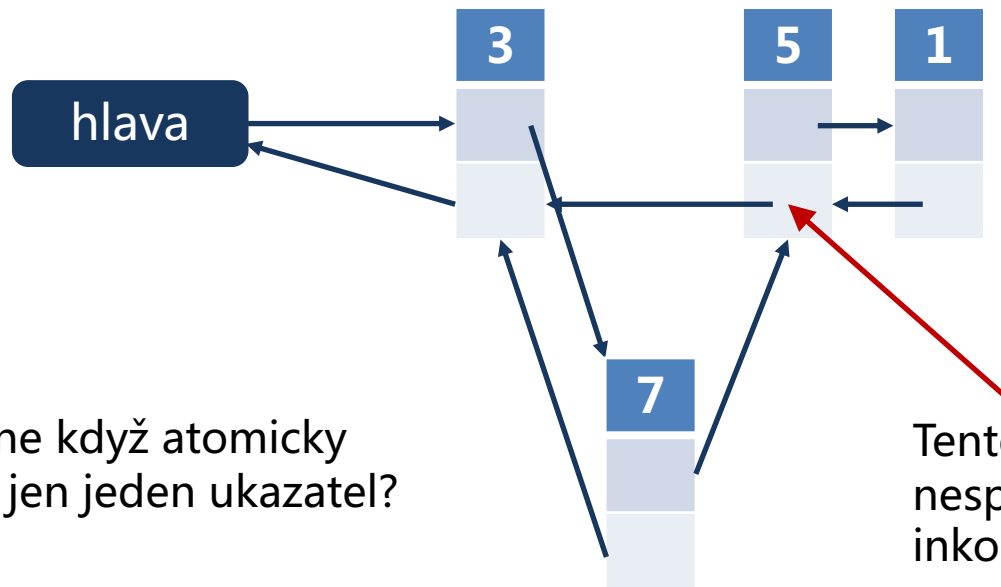
- Jak řešit pomocí atomických operací?



Souběžný přístup k datovým strukturám

Příklad 3

- Jak řešit pomocí atomických operací?



Co se stane když atomicky změníme jen jeden ukazatel?

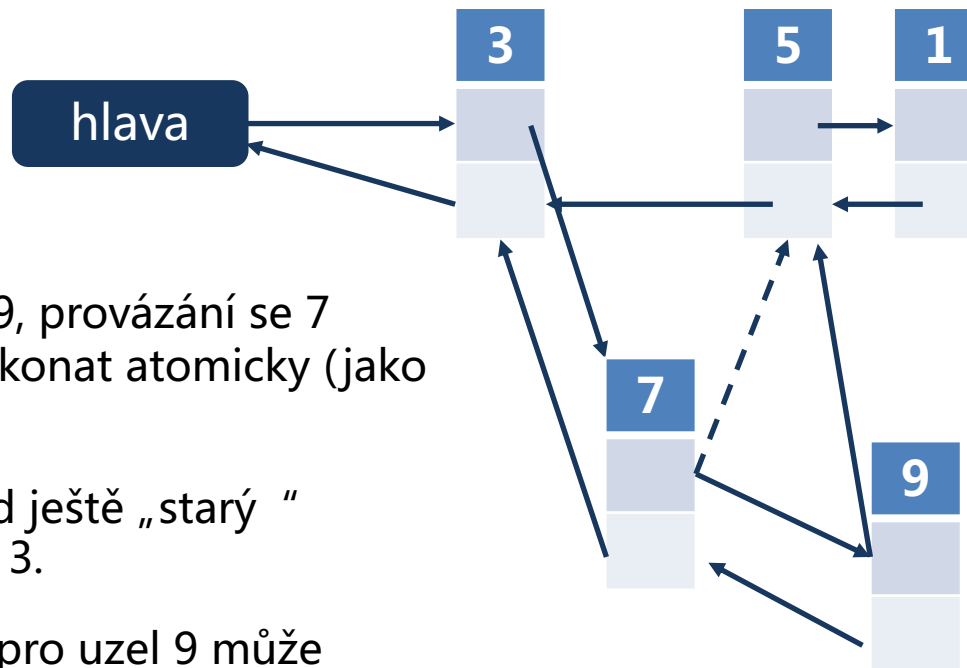
Tento ukazatel je nesprávný. Kdy nám tato inkonzistence může vadit?

Když chceme přidávat mezi 7 a 5.

Souběžný přístup k datovým strukturám

Příklad 3

- Jak řešit pomocí atomických operací?



Přidáváme 9, provázání se 7
můžeme vykonat atomicky (jako
předtím).

U 5 je pořád ještě „starý“
ukazatel na 3.

Dokončení pro uzel 9 může
počkat, až bude v 5 správný
ukazatel (na 7).

Souběžný přístup k datovým strukturám

Příklad 3

- Řešení pomocí CAS

```
struct AtomicNode {
    int value = 0;
    std::atomic<AtomicNode*> successor;
    std::atomic<AtomicNode*> predecessor;

    AtomicNode(int value) {
        successor.store(nullptr);
        predecessor.store(nullptr);
    }

    AtomicNode(int _value, AtomicNode* _predecessor, AtomicNode* _successor) : value(_value) {
        successor.store(_successor);
        predecessor.store(_predecessor);
    }
};
```

Souběžný přístup k datovým strukturám

Příklad 3

- Řešení pomocí CAS

```
AtomicNode* atomic_add_to_list_after(AtomicNode* _previous_node, int _new_value) {
    assert (_previous_node != nullptr);

    AtomicNode* old_successor = _previous_node->successor;
    AtomicNode* new_node = new AtomicNode(_new_value, _previous_node, old_successor);

    while (!_previous_node->successor.compare_exchange_strong(old_successor, new_node)) {
        old_successor = _previous_node->successor;
        new_node->successor.store(old_successor);
    }

    if (old_successor != nullptr) {
        while (!old_successor->predecessor.compare_exchange_strong(_previous_node, new_node))
            ;
    }

    return new_node;
}
```

Souběžný přístup k datovým strukturám

Příklad 3

- Řešení pomocí CAS

```
AtomicNode* atomic_add_to_list_after(AtomicNode* _previous_node, int _new_value) {
    assert (_previous_node != nullptr);

    AtomicNode* old_successor = _previous_node->successor;
    AtomicNode* new_node = new AtomicNode(_new_value, _previous_node, old_successor);

    while (!_previous_node->successor.compare_exchange_strong(old_successor, new_node)) {
        old_successor = _previous_node->successor;
        new_node->successor.store(old_successor);
    }

    if (old_successor != nullptr) {
        while (!old_successor->predecessor.compare_exchange_strong(_previous_node, new_node))
            ;
    }

    return new_node;
}
```

Změna ukazatele v
prvním prvku (před
vkládaným uzlem).

Souběžný přístup k datovým strukturám

Příklad 3

- Řešení pomocí CAS

```
AtomicNode* atomic_add_to_list_after(AtomicNode* _previous_node, int _new_value) {
    assert (_previous_node != nullptr);

    AtomicNode* old_successor = _previous_node->successor;
    AtomicNode* new_node = new AtomicNode(_new_value, _previous_node, old_successor);

    while (!_previous_node->successor.compare_exchange_strong(old_successor, new_node)) {
        old_successor = _previous_node->successor;
        new_node->successor.store(old_successor);
    }

    if (old_successor != nullptr) {
        while (!old_successor->predecessor.compare_exchange_strong(_previous_node, new_node))
            ;
    }

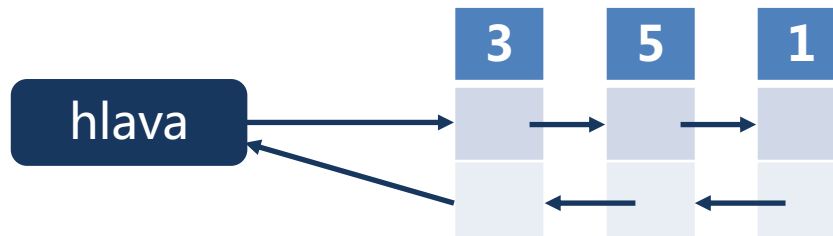
    return new_node;
}
```

Změna ukazatele v
druhého prvku (za
vkládaným uzlem).

Souběžný přístup k datovým strukturám

Příklad 3

- Obousměrný spojový seznam



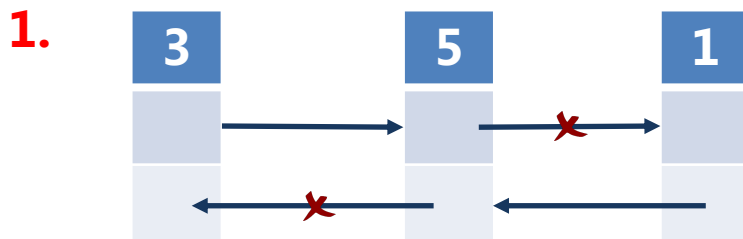
Operace mazání – komplexnější

Musíme označit které uzly (ukazatele) budou smazány

Souběžný přístup k datovým strukturám

Příklad 3

- Mazání v obousměrném spojovém seznamu



Provedeme následující atomické operace:

- Označíme ukazatele mazaného uzlu jako „ke smazání “

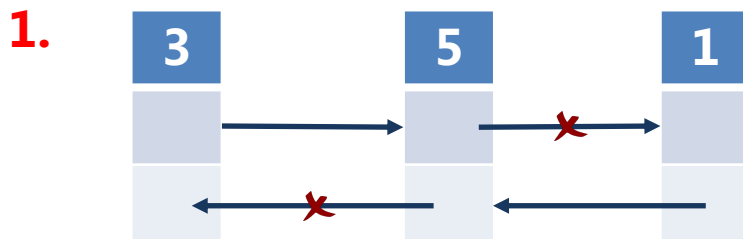
2.

3.

Souběžný přístup k datovým strukturám

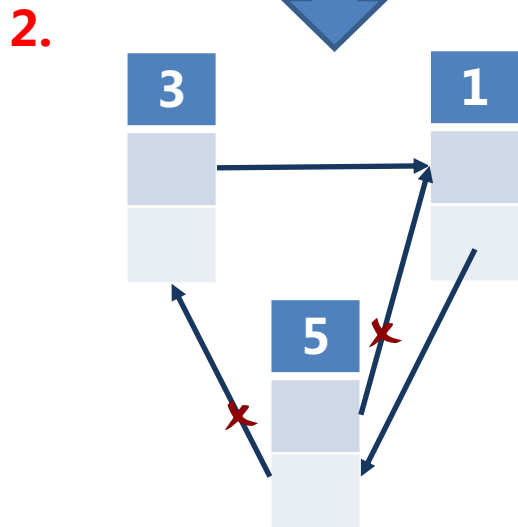
Příklad 3

- Mazání v obousměrném spojovém seznamu



Provedeme následující atomické operace:

- Označíme ukazatele mazaného uzlu jako „ke smazání“
- Převědeme ukazatel předchůdce

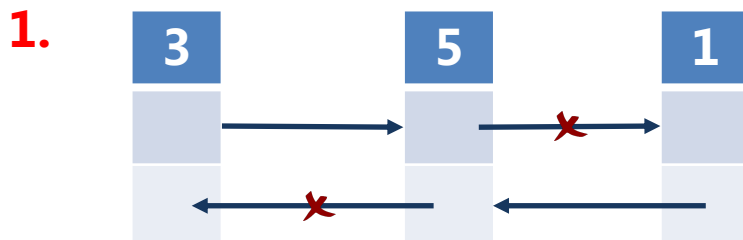


3.

Souběžný přístup k datovým strukturám

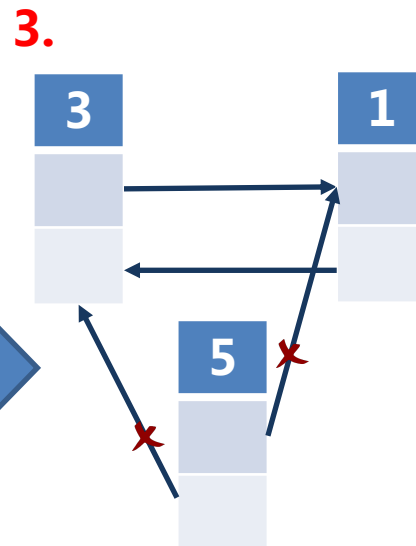
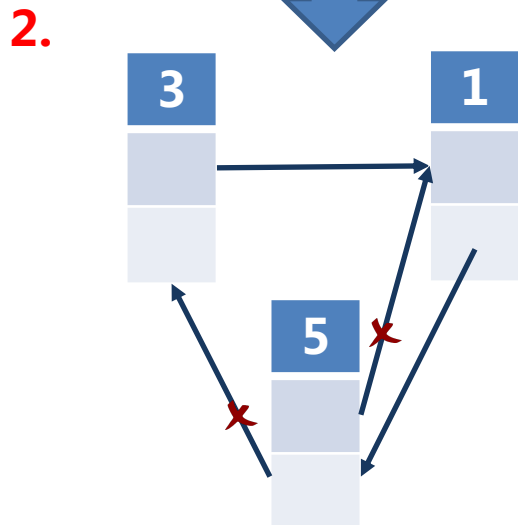
Příklad 3

- Mazání v obousměrném spojovém seznamu



Provedeme následující atomické operace:

- Označíme ukazatele mazaného uzlu jako „ke smazání“
- Převédeme ukazatel předchůdce
- Převédeme ukazatel následovníka



Souběžný přístup k datovým strukturám

Příklad 4

V posledku všechny případy výše jde zobecnit s pomocí šablon

```
/* podle https://en.cppreference.com/w/cpp/atomic/atomic/compare\_exchange */  
#include <atomic>  
template<typename T>  
struct Node {  
    T data;  
    Node* next;  
    Node(const T& data) : data(data), next(nullptr) {}  
};  
  
template<typename T> class stack {  
    std::atomic<Node<T>*> head;  
public:  
    void push(const T& data) {  
        Node<T>* new_node = new Node<T>(data);  
        new_node->next = head.load(std::memory_order_relaxed);  
        while(!head.compare_exchange_weak(new_node->next, new_node,  
                                           std::memory_order_release,  
                                           std::memory_order_relaxed));  
    }  
};  
  
int main() {  
    stack<int> s; s.push(1); s.push(2); s.push(3);  
}
```

Souběžný přístup k datovým strukturám

Závěr

V posledku všechny případy výše jde zobecnit s pomocí šablon. Tím se bychom se začali blížit STL.

V některých překladačích můžeme využít paralelní STL.

Naším cílem ale bylo poznat atomické „compare and swap “ (CAS), které atomicky porovná zda hodnota dereferencovaného ukazatele na atomickou proměnnou odpovídá očekávané hodnotě, a porozumět jeho použití v návrhu datových struktur pro souběžný přístup.