Vlákna

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Programování v C++, B6B36PJC
12/2018, Lekce 12

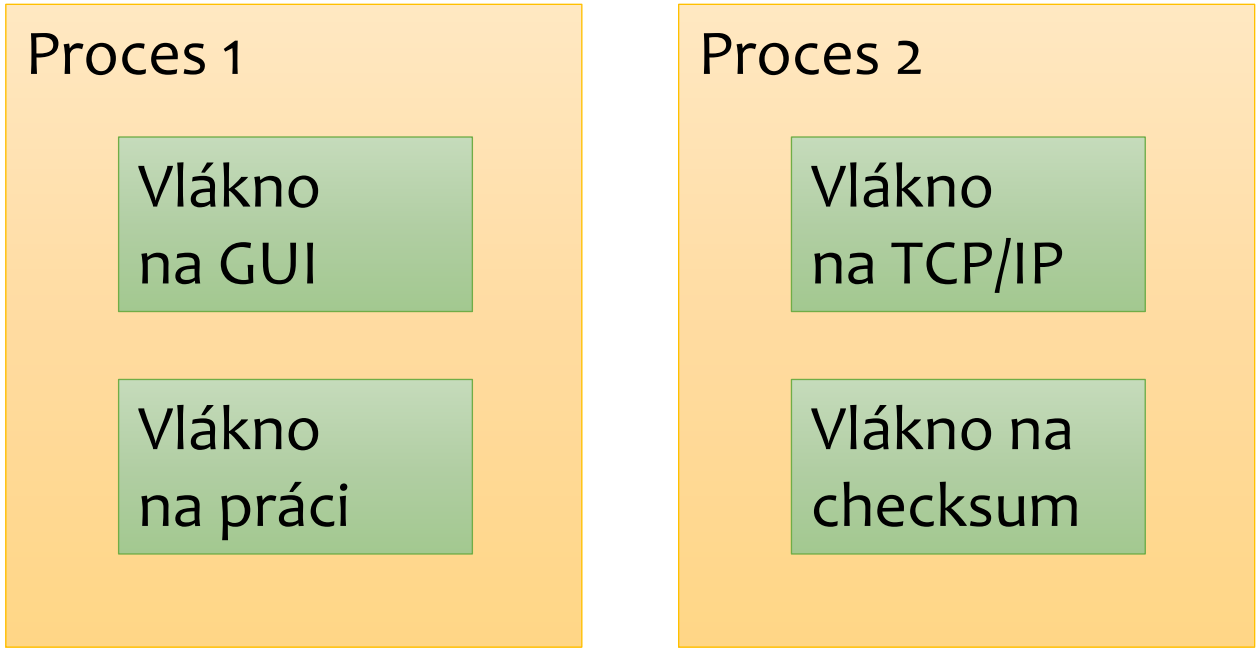
https://cw.fel.cvut.cz/wiki/courses/b6b36pjc/start

Motivace

- Všichni jsme už někdy potkali zamrzlé GUI.
- Obvykle se to děje kvůli čekání na dokončení práce nebo čtení z disku.
- Děje se to proto, že počítač dělá věci vždy jednu po druhé... ačkoliv by se během prodlev hodilo dělat něco jiného, trpělivě vyčkává.
- Vlákna umožňují dělat více věcí zároveň.
- Když tedy chceme, aby aplikace zároveň mohla pracovat i reagovat na uživatele, použijeme dvě nebo více vláken.
- Vlákna se též dají použít k urychlení běhu programu.

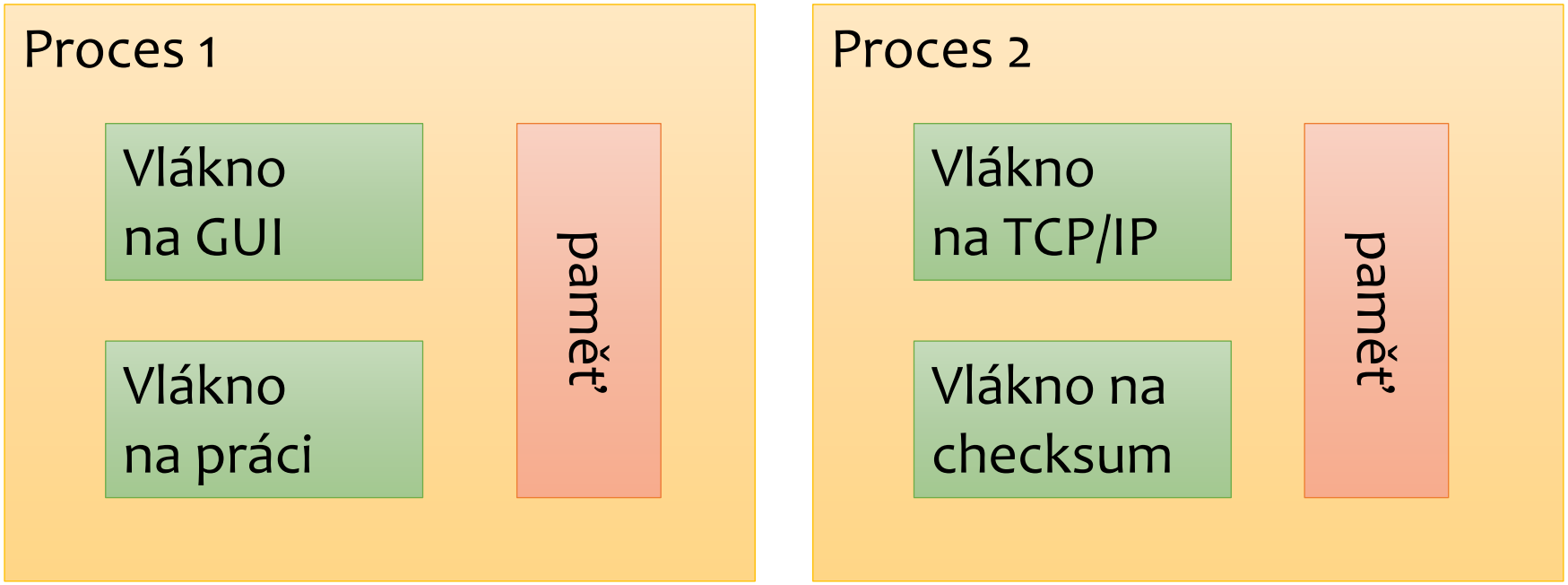
Vlákna

- Běh programu se nazývá proces.
- Proces obsahuje jedno nebo více vláken.



Vlákna a paměť

- Procesy žijí v navzájem oddělených adresních prostorech.
- Vlákna jednoho procesu naopak paměť sdílejí.



Podpora vláken ve standardním C++

- C++ standardní knihovna poskytuje pro práci s vlákny různé prostředky.
  - Vlákna – `std::thread`
  - Mutexy – `std::mutex`, `std::shared_mutex` a další
  - RAII zámky – `std::unique_lock` a další
  - `std::future`, `std::promise`, `std::async`
  - `std::condition_variable`
  - Atomické proměnné – `std::atomic<T>`
- Dále C++ definuje tzv. *memory model*, model chování vícevláknové aplikace.
  - Tím se nebudeme příliš zabývat.

Vlákna v C++ (<thread>)

- Vlákna jsou exportována hlavičkou <thread>.
- Konstruktor vlákna získá prvním argumentem funkci, kterou bude vykonávat, ostatní argumenty předá volané funkci.
- Vlákno je nejhrubší jednotka paralelismu.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std::chrono_literals;

void function(int id, int n) {
    for (int i = 0; i < n; ++i) {
        std::cout << "vlakno " << id << " rika ahoj\n";
        std::this_thread::sleep_for(10ms);
    }
}

int main() {
    std::thread t1(function, 1, 2);
    std::thread t2(function, 2, 4);
    t1.join();
    t2.join();
}
```

```
vlakno vlakno 2 rika ahoj
1 rika ahoj
vlakno 1 rika ahoj
vlakno 2 rika ahoj
vlakno 2 rika ahoj
vlakno 2 rika ahoj
```

Vlákna v C++ (<thread>)

- Dříve, než je zavolán destruktor `std::thread`, musíme
  - zavolat metodu `join` – spouštějící vlákno čeká, než spuštěné vlákno doběhne,
  - nebo zavolat metodu `detach` – spuštěné vlákno je autonomní.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std::chrono_literals;

void function(int id, int n) {
    for (int i = 0; i < n; ++i) {
        std::cout << "vlakno " << id << " rika ahoj\n";
        std::this_thread::sleep_for(10ms);
    }
}

int main() {
    std::thread t1(function, 1, 2);
    std::thread t2(function, 2, 4);
    t1.join(); // hlavní vlákno čeká na ukončení vlákna t1
    t2.join(); // hlavní vlákno čeká na ukončení vlákna t2
}
```

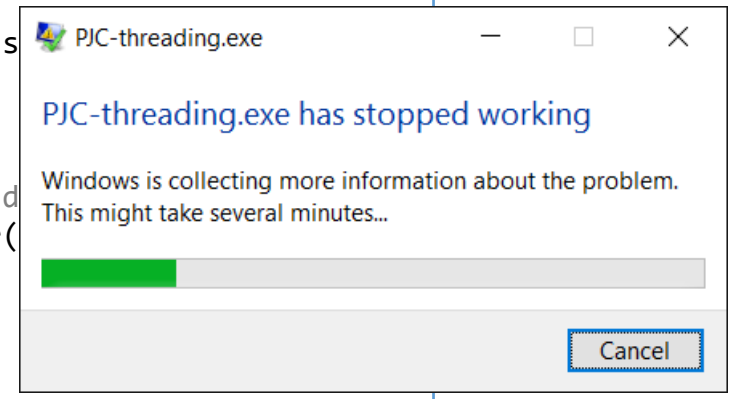
Vlákna v C++ (<thread>)

- Dříve, než je zavolán destruktor `std::thread`, musíme
  - zavolat metodu `join` – spouštějící vlákno čeká, než spuštěné vlákno doběhne,
  - nebo zavolat metodu `detach` – spuštěné vlákno je autonomní.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std::chrono_literals;

void function(int id, int n) {
    for (int i = 0; i < n; ++i) {
        std::cout << "vlakno " << id
        std::this_thread::sleep_for(
    }
}

int main() {
    std::thread t1(function, 1, 2);
    std::thread t2(function, 2, 4);
    t1.join(); // hlavní vlákno čeká na ukončení vlákna t1
    t2.join(); // hlavní vlákno čeká na ukončení vlákna t2
}
```



když zapomeneme na `join` nebo `detach`...

...proces se okamžitě ukončí.

Synchronizace

- Vzhledem k tomu, že vlákna sdílejí paměť, je potřeba je synchronizovat.
  - Jinak se budou dít divné věci!

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

int main() {
    int counter = 0;
    auto thread_func = [&counter]() {
        for (int i = 0; i < 1'000'000; ++i) {
            counter++;
            counter--;
        }
    };

    auto t1 = std::thread(thread_func);
    auto t2 = std::thread(thread_func);

    t1.join(); t2.join();
    std::cout << counter << std::endl;
}
```

```
>>> 0
>>> 1
>>> -3053
>>> 4
>>> -2
...
```

Mutexy a zámky (<mutex>)

- *Mutex* je struktura pro synchronizaci vláken.
  - Vlákna žádají o vlastnictví (uzamčení) mutexu.
  - Mutex může být v danou chvíli uzamčen pouze jedním vláknem.
- *Zámky* jsou RAII třídy, které obsluhují uzamykání a odemykání mutexů.
- Mutexy a zámky jsou exportovány hlavičkou <mutex>.

```
...
int counter = 0;
std::mutex mutex;
auto thread_func = [&counter, &mutex]() {
    for (int i = 0; i < 1'000'000; ++i) {
        std::unique_lock<std::mutex> lock(mutex);
        counter++;
        counter--;
    } // destruktor lock odemkne mutex
};
...
```

```
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
...
```

Mutexy a zámky – problémy

- Používání zámků s sebou nese různé potenciální problémy.
- Jeden z nich je deadlock. Deadlock je stav ve kterém program nemůže pokračovat, protože se různá vlákna navzájem blokují.

```
int main() {
    std::mutex m1;
    std::mutex m2;

    auto thread_func = [](std::mutex& first_mutex, std::mutex& second_mutex, int id) {
        while (true) {
            std::lock_guard<std::mutex> l1(first_mutex);
            std::cout << "Vlakno " << id << " rika ahoj.\n";
            std::lock_guard<std::mutex> l2(second_mutex);
        }
    };

    std::thread t1(thread_func, std::ref(m1), std::ref(m2), 0);
    std::thread t2(thread_func, std::ref(m2), std::ref(m1), 1);

    t1.join();
    t2.join();
}
```

Mutexy a zámky – problémy

- K deadlocku obvykle dojde, pokud se více vláken pokusí zamknout stejné mutexy v různém pořadí.
- Pokud spustíme tento kód, výpisy přestanou okamžitě. Proč?

```
int main() {
    std::mutex m1;
    std::mutex m2;

    auto thread_func = [](std::mutex& first_mutex, std::mutex& second_mutex, int id) {
        while (true) {
            std::lock_guard<std::mutex> l1(first_mutex);
            std::cout << "Vlakno " << id << " rika ahoj.\n";
            std::lock_guard<std::mutex> l2(second_mutex);
        }
    };

    std::thread t1(thread_func, std::ref(m1), std::ref(m2), 0);
    std::thread t2(thread_func, std::ref(m2), std::ref(m1), 1);

    t1.join();
    t2.join();
}
```

Mutexy a zámky – problémy

vlákno 1:	`std::lock_guard<std::mutex> l1(first_mutex);  // vlákno 1 zamkne m1`
vlákno 1:	`std::cout << "Vlakno " << id << " rika ahoj.\n";`
vlákno 2:	`std::lock_guard<std::mutex> l1(first_mutex);  // vlákno 2 zamkne m2`
vlákno 2:	`std::cout << "Vlakno " << id << " rika ahoj.\n";`
vlákno 2:	`std::lock_guard<std::mutex> l2(second_mutex); // vlákno 2 čeká, chce zamknout m1`
vlákno 1:	`std::lock_guard<std::mutex> l2(second_mutex); // vlákno 1 čeká, chce zamknout m2`
...	**výsledek: deadlock**

```
int main() {
    std::mutex m1;
    std::mutex m2;

    auto thread_func = [](std::mutex& first_mutex, std::mutex& second_mutex, int id) {
        while (true) {
            std::lock_guard<std::mutex> l1(first_mutex);
            std::cout << "Vlakno " << id << " rika ahoj.\n";
            std::lock_guard<std::mutex> l2(second_mutex);
        }
    };

    std::thread t1(thread_func, std::ref(m1), std::ref(m2), 0);
    std::thread t2(thread_func, std::ref(m2), std::ref(m1), 1);

    t1.join();
    t2.join();
}
```

Mutexy a zámky – problémy

- K deadlocku nemůže dojít, použijeme-li `std::lock`.
- `std::lock` uzamkne všechny mutexy, které mu poskytneme.
- Mutexy pak musíme umístit do `std::lock_guard` s druhým parametrem `std::adopt_lock`, jinak se mutexy neodemknou.

```
int main() {
    std::mutex m1;
    std::mutex m2;

    auto thread_func = [](std::mutex& first_mutex, std::mutex& second_mutex, int id) {
        while (true) {
            std::lock(first_mutex, second_mutex);
            std::lock_guard<std::mutex> l1(first_mutex, std::adopt_lock);
            std::cout << "Vlakno " << id << " rika ahoj.\n";
            std::lock_guard<std::mutex> l2(second_mutex, std::adopt_lock);
        }
    };

    std::thread t1(thread_func, std::ref(m1), std::ref(m2), 0);
    std::thread t2(thread_func, std::ref(m2), std::ref(m1), 1);

    t1.join();
    t2.join();
}
```

Future, Promise a Async v C++ (<future>)

- Protože vlákna neumí vracet hodnotu, existují tzv. futures.
- S futures jsou spojeny tzv. "přísliby" (promise) a async.
- Future a promise reprezentují budoucí výsledek.
  - Future umožňuje jeho čtení.
  - Promise jeho zápis.

```
std::vector<int> numbers;
// ziskame cisla

// pripravime si future a promise
std::promise<int> sum_promise;
std::future<int> sum_future = sum_promise.get_future();

// Nechame jine vlakno, aby je secetlo
std::thread([](std::promise<int> promise, std::vector<int> numbers) {
    promise.set_value(std::accumulate(begin(numbers), end(numbers), 0));
}, std::move(sum_promise), std::move(numbers)).detach();
// zatimco delame jinou praci

int sum = sum_future.get(); // ziskame vysledek, pokud jeste neni, blokujeme
```

Future, Promise a Async v C++ (<future>)

- Pro ulehčení práce se `std::promise` a `std::future` existuje ještě `std::async`.
- `std::async` bere při konstrukci způsob vykonání, funkci, kterou má vykonat, a argumenty, které předá funkci.
- `std::async` vrátí `std::future` a následně „nějak zařídí“, aby se funkce vykonala.

```
template <typename RandomAccessIter>
int parallel_mult(RandomAccessIter beg, RandomAccessIter end) {
    auto dist = std::distance(beg, end);
    if (dist <= 1000) {
        return std::accumulate(beg, end, 1, std::multiplies<>{});
    }

    RandomAccessIter middle = beg + dist / 2;

    auto rmul = std::async(std::launch::async,
                           parallel_mult<RandomAccessIter>,
                           middle, end);
    int lmul = parallel_mult(beg, middle);
    return lmul * rmul.get();
}
```

Future, Promise a Async v C++ (<future>)

- POZOR: Destruktor `std::future` vytvořené přes `std::async` je blokující a čeká na ukončení vykonávání funkce uvnitř `std::async`.

```
std::async(std::launch::async, foo); // vytvoří dočasnou future a čeká na jejím destruktoru
std::async(std::launch::async, bar); // funkce bar nemůže začít, dokud neskončí funkce foo
```

- Je potřeba uložit future do dočasné proměnné, nebo jiným způsobem prodloužit její život.

```
{
    auto t1 = std::async(std::launch::async, foo); // začne probíhat foo
    auto t2 = std::async(std::launch::async, bar); // začne probíhat bar bez ohledu na foo
} // zde se spustí destruktor t2 a t1, čeká se na ukončení foo i bar.
```

```
std::future<void> qux() {
    return std::async(std::launch::async, foo); // začne probíhat foo
                      // nedojde k blokování, protože je future vrácena z funkce ven
}
```

Condition Variables (<condition_variable>)

- Podmínkové proměnné slouží ke komunikaci mezi vlákny.
  - Na rozdíl od mutexů, které slouží k synchronizaci.
- Podmínkové proměnné jsou vždy spojené s nějakým mutexem.
- Na podmínkových proměnných mohou vlákna čekat na signál od jiného vlákna.
- S čekáním na podmínkové proměnné vždy souvisí predikát, který kontroluje, jestli se vlákno mělo probudit – vlákno se mohlo probudit i samovolně, nebo nemusí být schopno v současném stavu systému pokračovat.

Condition Variables (<condition_variable>)

```
template <typename T>
class synchronized_box {
    T element;
    std::mutex mutex;
    std::condition_variable cv;
    bool is_ready, closed;

public:
   synchronized_box():is_ready(false),closed(false) {}
    void insert(const T& e) {
        std::unique_lock<std::mutex> lock(mutex);
        cv.wait(lock, [&]() {
            return !is_ready || closed; });
        if (closed) { return; }
        element = e;
        is_ready = true;
        cv.notify_one();
    }
    bool take_out(T& out) {
        std::unique_lock<std::mutex> lock(mutex);
        cv.wait(lock, [&]() {
            return is_ready || closed; });
        if (closed) { return false; }
        out = std::move(element);
        is_ready = false;
        cv.notify_one();
        return true;
    }
    ...
```

```
    ...
    void close() {
        std::unique_lock<std::mutex>
            lock(mutex);
        closed = true;
        cv.notify_all();
    }

    ~synchronized_box(){
        close();
    }
};



int main() {
    synchronized_box<int> box;
    auto thread_func = [&]() {
        int temp;
        box.take_out(temp);
        std::cout << temp << '\n';
    };
    std::thread t(thread_func);

    std::this_thread::sleep_for(10ms);
    box.insert(1);
    t.join();
}
```

Atomické proměnné (<atomic>)

- Atomické proměnné jsou alternativa k zámku nad jednou proměnnou.
  - Zamknutí mutexu je poměrně nákladná operace.
  - Atomické operace nejsou zdarma, ale jsou levnější.
- Pracuje se s nimi podobně jako s neatomickými proměnnými, ale operace nad nimi jsou nedělitelné.

```
int main() {
    std::atomic<int> counter = 0;
    auto thread_func = [&counter]() {
        for (int i = 0; i < 1'000'000; ++i) {
            counter++;
            counter--;
        }
    };

    auto t1 = std::thread(thread_func);
    auto t2 = std::thread(thread_func);

    t1.join(); t2.join();
    std::cout << counter << std::endl;
}
```

```
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
...
```

Atomické proměnné (<atomic>)

- Atomické operace mohou mít různou „sílu“.
  - Můžeme ji určit pomocí druhého parametru, který poskytneme dané atomické operaci.
- Jednotlivé stupně mění, jaké optimalizace smí kompilátor provést.
  - Výchozí hodnota požaduje, aby byla dodržena *sekvenční konzistence*. To je rozumná varianta, ačkoliv může být pomalá.

```
int main() {
    std::atomic<int> counter = 0;
    auto thread_func = [&counter]() {
        for (int i = 0; i < 1'000'000; ++i) {
            counter.fetch_add(1, memory_order_seq_cst);
            counter.fetch_sub(1, memory_order_seq_cst);
        }
    };

    auto t1 = std::thread(thread_func);
    auto t2 = std::thread(thread_func);

    t1.join(); t2.join();
    std::cout << counter << std::endl;
}
```

```
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
...
```

Proč chceme sekvenční konzistenci

- Mějme kousek pseudokódu:
- Jaké různé výsledky byste čekali?

```
x = 0;
y = 0;
```

Vlákno 1	Vlákno 2
`x = 1;` `r1 = y;`	`y = 1;` `r2 = x;`

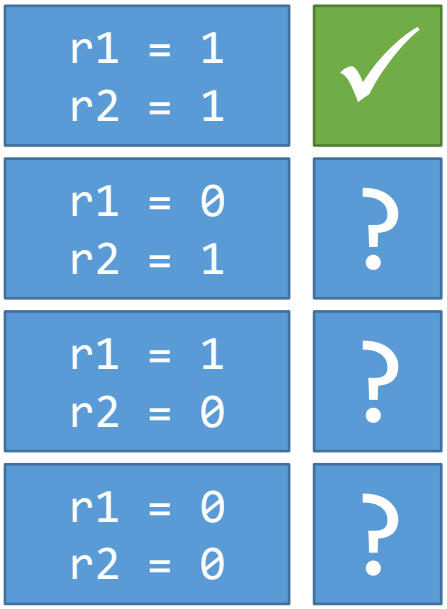
`r1 = 1` `r2 = 1`	?
`r1 = 0` `r2 = 1`	?
`r1 = 1` `r2 = 0`	?
`r1 = 0` `r2 = 0`	?

Proč chceme sekvenční konzistenci

- Mějme kousek pseudokódu:
- Jaké různé výsledky byste čekali?

```
x = 0;
y = 0;
```

Vlákno 1	Vlákno 2
`x = 1;` `r1 = y;`	`y = 1;` `r2 = x;`




```
x = 1;
y = 1;
r1 = y;
r2 = x;
```

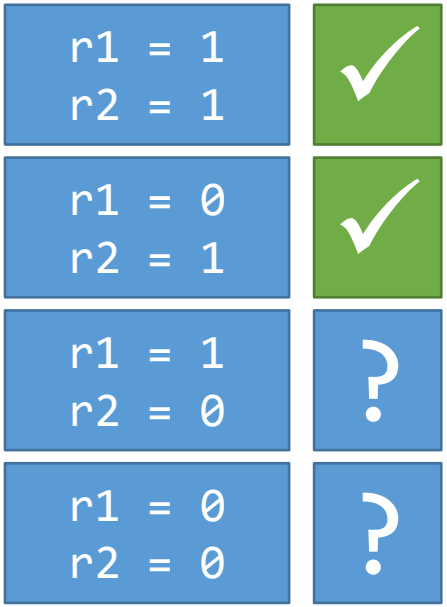
Proč chceme sekvenční konzistenci

- Mějme kousek pseudokódu:
- Jaké různé výsledky byste čekali?

```
x = 0;
y = 0;
```

Vlákno 1	Vlákno 2
`x = 1;` `r1 = y;`	`y = 1;` `r2 = x;`

```
x = 1;
r1 = y;
y = 1;
r2 = x;
```




```
x = 1;
y = 1;
r1 = y;
r2 = x;
```

Proč chceme sekvenční konzistenci

- Mějme kousek pseudokódu:
- Jaké různé výsledky byste čekali?

```
x = 0;
y = 0;
```

Vlákno 1	Vlákno 2
`x = 1;` `r1 = y;`	`y = 1;` `r2 = x;`

```
x = 1;
r1 = y;
y = 1;
r2 = x;
```

`r1 = 1` `r2 = 1`	✓
`r1 = 0` `r2 = 1`	✓
`r1 = 1` `r2 = 0`	✓
`r1 = 0` `r2 = 0`	?

```
x = 1;
y = 1;
r1 = y;
r2 = x;
```

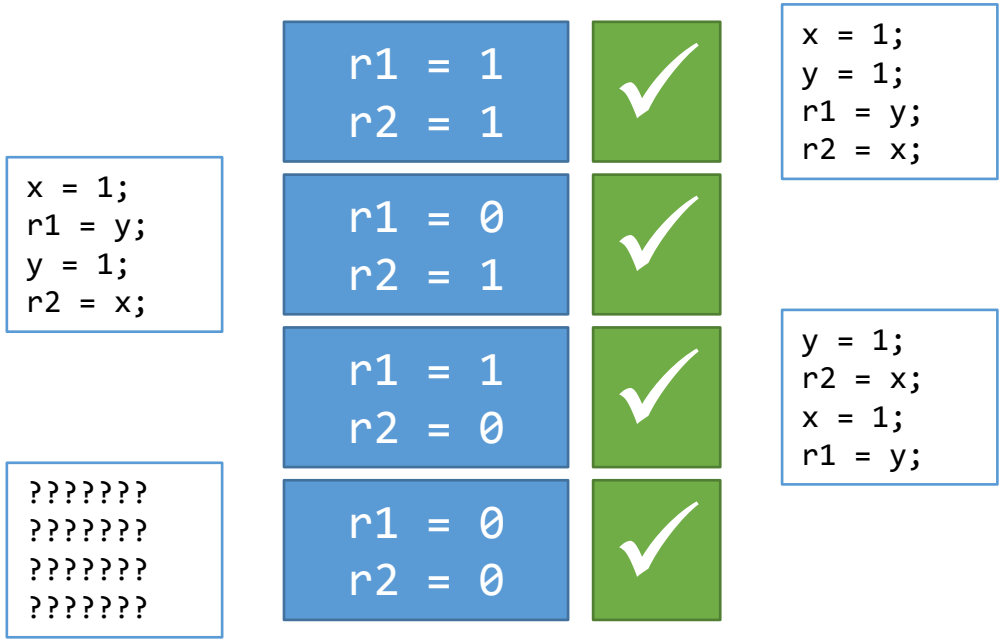
```
y = 1;
r2 = x;
x = 1;
r1 = y;
```

Proč chceme sekvenční konzistenci

- Mějme kousek pseudokódu:
- Jaké různé výsledky byste čekali?

```
x = 0;
y = 0;
```

Vlákno 1	Vlákno 2
`x = 1;` `r1 = y;`	`y = 1;` `r2 = x;`

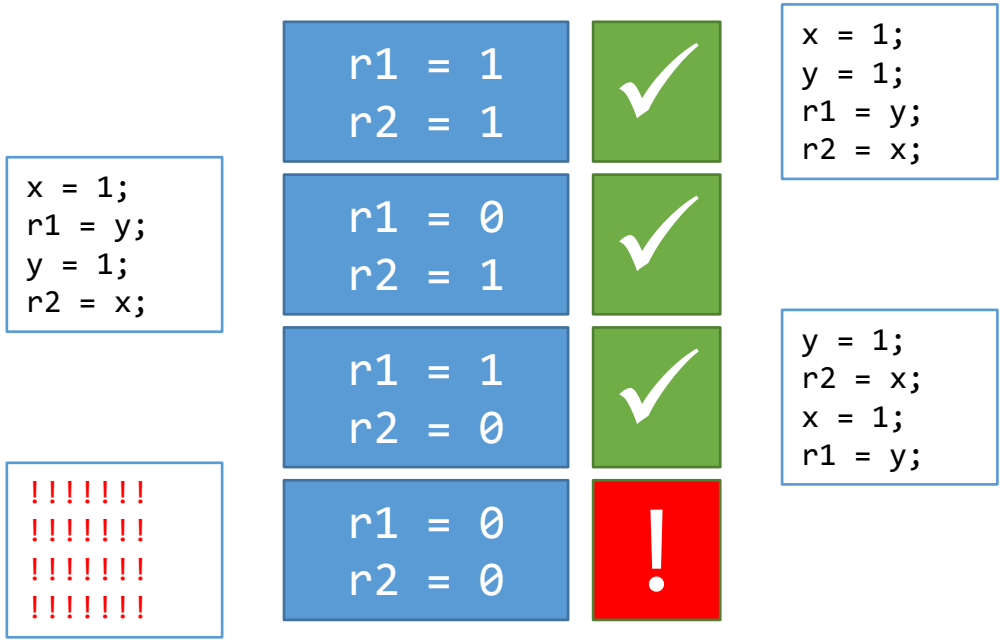


Proč chceme sekvenční konzistenci

- Mějme kousek pseudokódu:
- Jaké různé výsledky byste čekali?

```
x = 0;
y = 0;
```

Vlákno 1	Vlákno 2
`x = 1;` `r1 = y;`	`y = 1;` `r2 = x;`

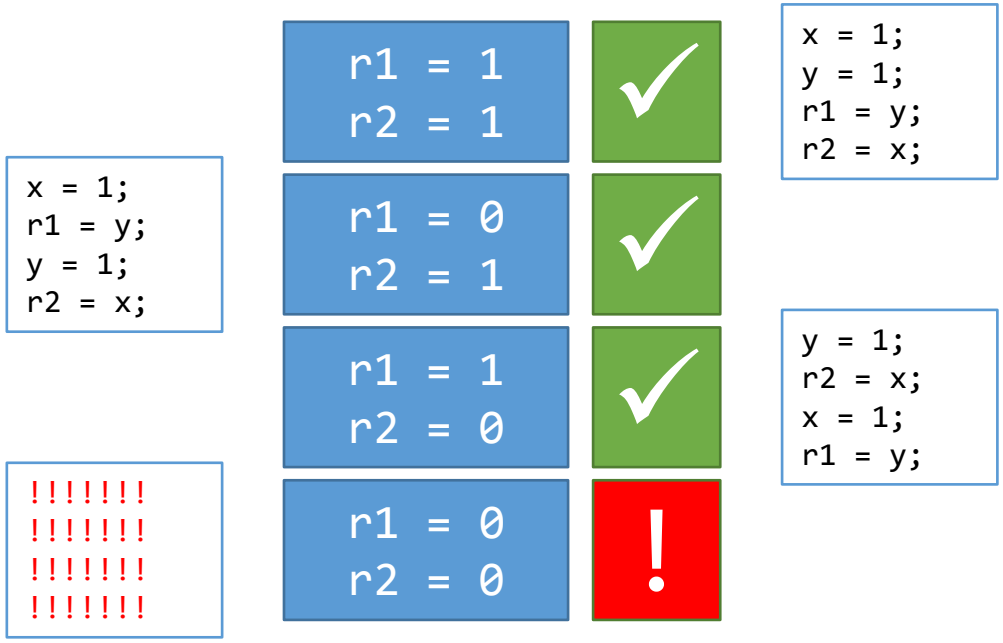


Proč chceme sekvenční konzistenci

- Mějme kousek pseudokódu:
- Jaké různé výsledky byste čekali?

```
x = 0;
y = 0;
```

Vlákno 1	Vlákno 2
`x = 1;` `r1 = y;`	`y = 1;` `r2 = x;`



- Poslední případ je tzv. relaxovaný. Tomuto výsledku sekvenční konzistence zabrání.

Synchronizace nad více proměnnými

- Pozor, pro synchronizaci nad více proměnnými je stále potřeba zámek.
- Uvažte tuto funkci pro převod peněz. Pokud se pokusíme provést více plateb zároveň, nemůže se nám stát, že se nějaký účet dostane do mínusu?

```
bool transaction(std::atomic<int>& balance1, std::atomic<int>& balance2, int amount) {
    if (balance1 >= amount) {
        balance1 -= amount;
        balance2 += amount;
        return true;
    }
    return false;
}
```

Synchronizace nad více proměnnými

- Pozor, pro synchronizaci nad více proměnnými je stále potřeba zámek.
- Uvažte tuto funkci pro převod peněz. Pokud se pokusíme provést více plateb zároveň, nemůže se nám stát, že se nějaký účet dostane do mínusu?

```
bool transaction(std::atomic<int>& balance1, std::atomic<int>& balance2, int amount) {
    if (balance1 >= amount) {
        balance1 -= amount;
        balance2 += amount;
        return true;
    }
    return false;
}
```

vlákno 1:	`if (balance1 >= amount) {`
vlákno 2:	`if (balance1 >= amount) {`
vlákno 1:	`balance1 -= amount;`
vlákno 1:	`balance2 += amount;`
vlákno 2:	`balance1 -= amount;`
vlákno 2:	`balance2 += amount;`
vlákno 2:	`return true;`
vlákno 1:	`return true;`
...	...

Synchronizace nad více proměnnými

- Pro synchronizaci více proměnných je stále potřeba použít zamykání.

```
bool transaction(account& account1, account& account2, int amount) {
    std::lock(account1.mutex, account2.mutex); // Zamkneme pristup k oboum uctum
    // Prevedeme vlastnictvi do lock guardu
    std::lock_guard<std::mutex> lg1(account1.mutex, std::adopt_lock);
    std::lock_guard<std::mutex> lg2(account2.mutex, std::adopt_lock);

    // Ted muzeme prevest penize aniz by hrozilo, ze se dostaneme do zaporu
    if (account1.balance >= amount) {
        account1.balance -= amount;
        account2.balance += amount;
        return true;
    }

    return false;
}
```

```
struct account {
    std::mutex mutex;
    std::atomic<int> balance;
};
```

Běžné modely paralelismu

- Existuje mnoho běžných modelů paralelismu v programech.
  - My si probereme 3 základní.

- Boss-Worker
  - Hlavní vlákno rozděluje úkoly ostatním vláknům
- Pipeline
  - Každé vlákno provede nějakou práci nad daty a pošle výsledek dalšímu.
- Work crew
  - Více vláken vykonává stejnou práci nad různými daty.

Work crew model

- Tento model je používaný například v OpenMP.
- Vlákna uvnitř jedné skupiny sdílejí kód, ale nesdílejí data.

```
void process_data(std::vector<foobar>& data) {
    auto thread_func = [&](int from, int to) {
        for (int i = from; i < to; ++i) {
            process_foobar(data[i]);
        }
    };
    int num_threads = std::thread::hardware_concurrency();
    int part_size = data.size() / num_threads;
    int mod = data.size() % num_threads;
    std::vector<std::thread> threads;
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(thread_func,
                             i * part_size + std::min(i, mod),
                             (i + 1) * part_size + std::min(i+1, mod));
    }
    for (auto& t : threads) {
        t.join();
    }
}
```

Pipeline model

- Zřetězení po sobě následujících úkolů, jako pipe v UNIXu:
  `find . | grep "homework" | grep "PJC"`
- Každá část vykoná jednoduchý úkol s daty a pošle výsledek dál.

```
void pipeline(const std::vector<spam>& data) {
    std::vector<std::thread> threads;
    synchronized_box<spam> pipe_start;
    synchronized_box<ham> pipe_mid;
    synchronized_box<lunch> pipe_end;

    threads.emplace_back([&]() {
        spam temp;
        while (pipe_start.take_out(temp)) {
            pipe_mid.insert(process_spam(temp));
        }
        pipe_mid.close();
    });
    threads.emplace_back([&]() {
        ham temp;
        while (pipe_mid.take_out(temp)) {
            pipe_end.insert(process_ham(temp));
        }
        pipe_end.close();
    });
    threads.emplace_back([&]() {
        lunch lunch;
        while(pipe_end.take_out(lunch)){
            eat(lunch);
        }
    });

    for (auto& d : data) {
        pipe_start.insert(d);
    }

    pipe_start.close();
    for (auto& t : threads) {
        t.join();
    }
}
```

Boss-Worker model

- Jedno vlákno je privilegované a rozdává práci. Ostatní vlákna zpracovávají práci a vrací výsledky.
- Typicky používané třeba pro uživatelské rozhraní aplikace.
  - „Boss“ vlákno přijímá uživatelovy požadavky a dává je jiným vláknům k vykonání. Tím se zamezí zaseknutí aplikace během provádění práce.
- Též se často používá pro obsluhu HTTP dotazů.
  - Nadřízený přijímá požadavky a dává je podřízeným, aby je vyřídili.

Boss-Worker model (příklad)

```
class worker {
    worker(synchronized_queue<request>& q):queue(q) { ... }
    // ...
    std::thread thread;
    synchronized_queue<request>& queue;
};


int main() {
    // Prepare workers
    synchronized_queue<request> work_queue;
    std::vector<worker> workers;
    for (int w = 0; w < std::thread::hardware_concurrency(); ++w) {
        workers.push_back(worker(work_queue));
    }

    // Open socket
    socket s("127:0:0:1:80");
    // Keep adding work
    while (true) {
        auto request = read_request(s);
        work_queue.insert(request);
    }
}
```

Děkuji za pozornost.