# Vlákna

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Přednášky byly připraveny s pomocí materiálů, které vyrobili Ladislav Vágner, Pavel Strnad, Martin Hořeňovský, Aleš Hrabalík a Martin Mazanec



Programování v C++, A7B36PJC
09/2015, Lekce xx

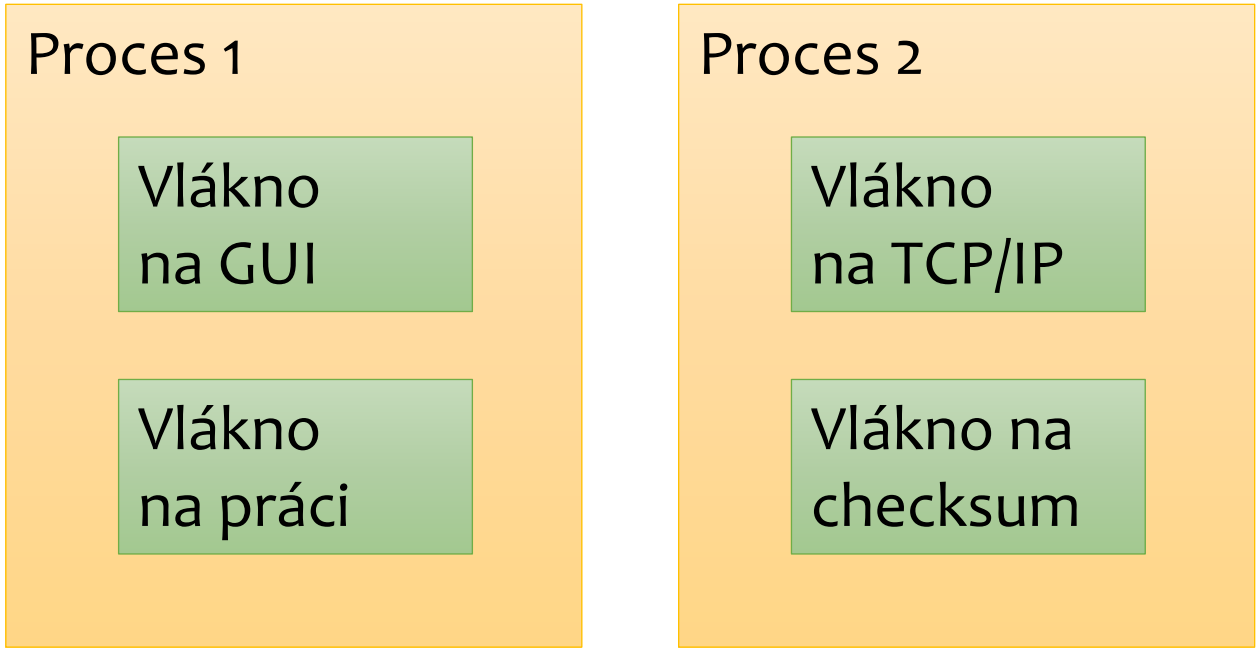
https://cw.fel.cvut.cz/wiki/courses/a7b36pjc/start

# 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.

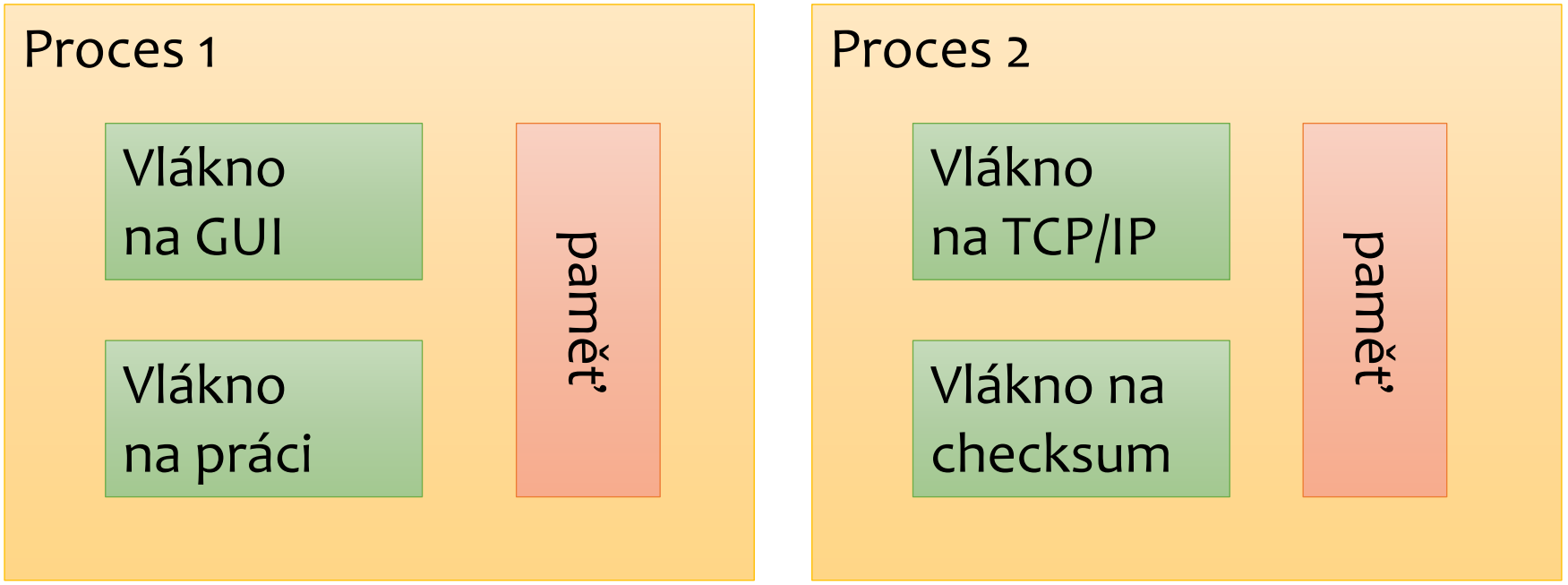
# Vlákna

- 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
  - Mutexy
  - RAII zámky
  - Futures
  - Condition Variables
  - Atomické proměnné
- Dále C++ definuje tzv. „Memory model“, model pro chování aplikace, pokud v ní běží více vláken zároveň.
  - Tím se nebudeme příliš zabývat.

# Vlákna v C++ (<thread>)

- Vlákna jsou exportována hlavičkou <thread>.
- Vlákno při své konstrukci bere funkci, kterou bude vykonávat, a další argumenty, které předá funkci.
- Jedná se o nejhrubší jednotku paralelismu.

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

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, 6);
    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
vlakno 2 rika ahoj
vlakno 2 rika ahoj
```

# Vlákna v C++ (<thread>)

- Vlákno musí být před svou destrukcí buďto tzv. „spojeno“ (joined), kdy spouštějící vlákno čeká, než spuštěné vlákno doběhne, nebo tzv. „odpojeno“ (detached), kdy je spuštěnému vláknu dána autonomie, než doběhne.

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

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, 10);
    t1.join(); // hlavní vlákno čeká na ukončení vlákna t1
    t2.join(); // hlavní vlákno čeká na ukončení vlákna t2
}
```

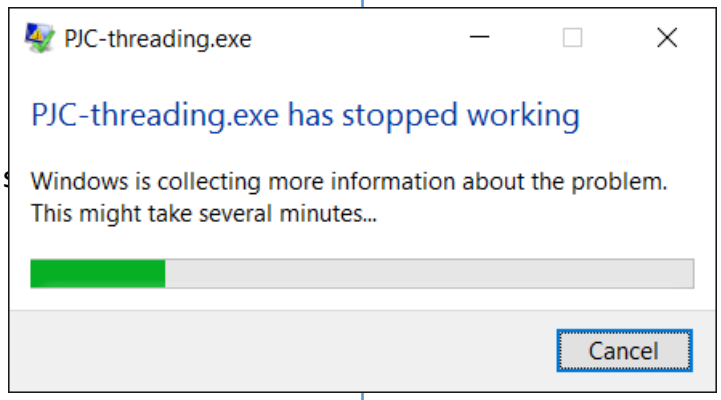
# Vlákna v C++ (<thread>)

- Vlákno musí být před svou destrukcí buďto tzv. „spojeno“ (joined), kdy spouštějící vlákno čeká, než spuštěné vlákno doběhne, nebo tzv. „odpojeno“ (detached), kdy je spuštěnému vláknu dána autonomie než doběhne.

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

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

int main() {
    std::thread t1(function, 1, 2);
    std::thread t2(function, 2, 10);
    t1.join(); // hlavní vlákno čeká na ukončení vlákna t1
    t2.join(); // hlavní vlákno čeká na ukončení vlákna t2
}
```



# Problémy

- 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 zamčen pouze jedním vláknem.
- Mutexy a zámky jsou exportovány hlavičkou <mutex>.
- Zámky jsou RAII třída, která obsluhuje uzamykání a odemykání mutexů.
- Mutex je hlavní primitivum pro synchronizaci paralelního běhu.

```
    int counter = 0;
    std::mutex mutex;

    auto thread_func = [&counter]() {
        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
...
```

# 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
```

# 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 rsum = std::async(std::launch::async,
                           parallel_mult<RandomAccessIter>,
                           middle, end);
    int lsum = parallel_mult(beg, middle);
    return lsum * rsum.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.
  - Narozdíl od mutexů, které slouží k synchronizaci.
- Podmínkové proměnné jsou vždy spojené s nějakým mutexem.
- Na podmínkových proměnných vlákna mohou čekat na signál od jiného vlákna.

# Atomické proměnné (<atomic>)

- Atomické proměnné jsou alternativa k zámku nad jednou proměnnou.
  - Zamknutí mutexu je poměrně nákladná operace a pro spoustu věcí vlastně není potřeba.
- 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 nejsou zdarma, jsou ale mnohem levnější nežli zamknutí a odemknutí mutexu.
- Atomické operace mají různé „síly“.
  - Dávají se jako druhý parametr operacím nad atomickými proměnnými.
  - Budeme se zabývat pouze jednou – sekvenční konzistencí.

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

    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>)

- Pozor, pro synchronizaci nad více proměnnými je stále potřeba zámek.

```
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
...
```

# Proč chceme sekvenční konzistenci

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

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

| Thread 1 | Thread 2 |
| --- | --- |
| `x = 1;`<br>`r1 = y;` | `y = 1;`<br>`r2 = x;` |

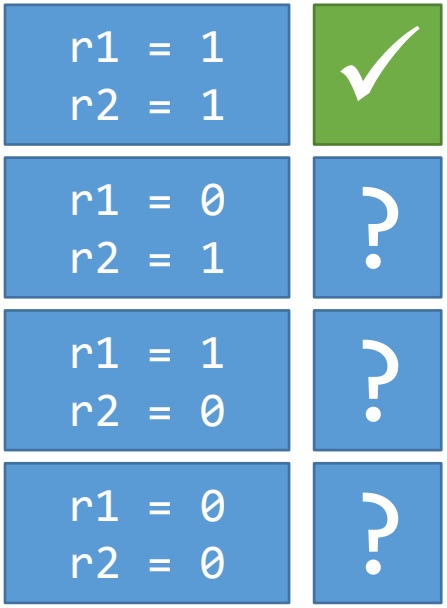
| | |
| --- | --- |
| `r1 = 1`<br>`r2 = 1` | ? |
| `r1 = 0`<br>`r2 = 1` | ? |
| `r1 = 1`<br>`r2 = 0` | ? |
| `r1 = 0`<br>`r2 = 0` | ? |

# Proč chceme sekvenční konzistenci

- Mějme kousek pseudokódu:
- Jaké různé výsledky by jste čekali?

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

| Thread 1 | Thread 2 |
|---|---|
| `x = 1;`<br>`r1 = y;` | `y = 1;`<br>`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;
```

| Thread 1 | Thread 2 |
|---|---|
| `x = 1;`<br>`r1 = y;` | `y = 1;`<br>`r2 = x;` |

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

| | |
|---|---|
| r1 = 1<br>r2 = 1 | ✓ |
| r1 = 0<br>r2 = 1 | ✓ |
| r1 = 1<br>r2 = 0 | ? |
| r1 = 0<br>r2 = 0 | ? |

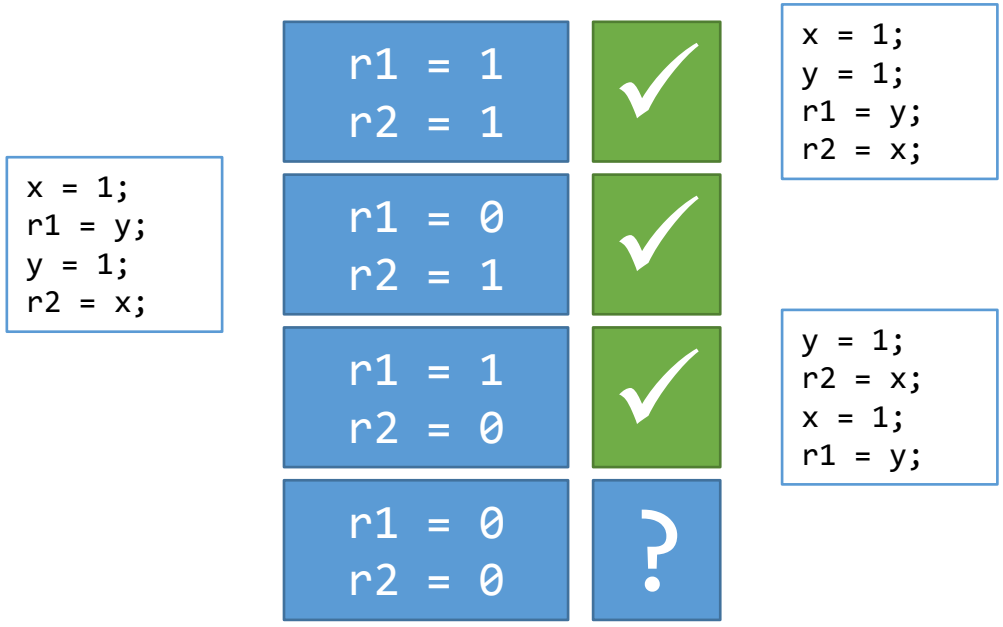
```
x = 1;
y = 1;
r1 = y;
r2 = x;
```

# Proč chceme sekvenční konzistenci

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

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

| Thread 1 | Thread 2 |
|---|---|
| `x = 1;`<br>`r1 = y;` | `y = 1;`<br>`r2 = x;` |

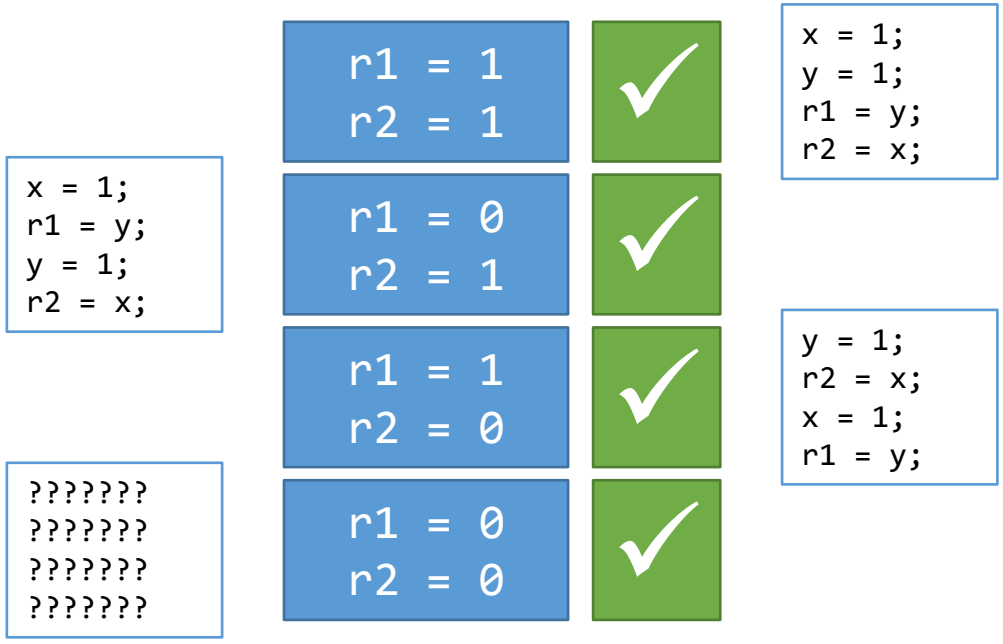


# Proč chceme sekvenční konzistenci

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

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

| Thread 1 | Thread 2 |
|---|---|
| `x = 1;`<br>`r1 = y;` | `y = 1;`<br>`r2 = x;` |

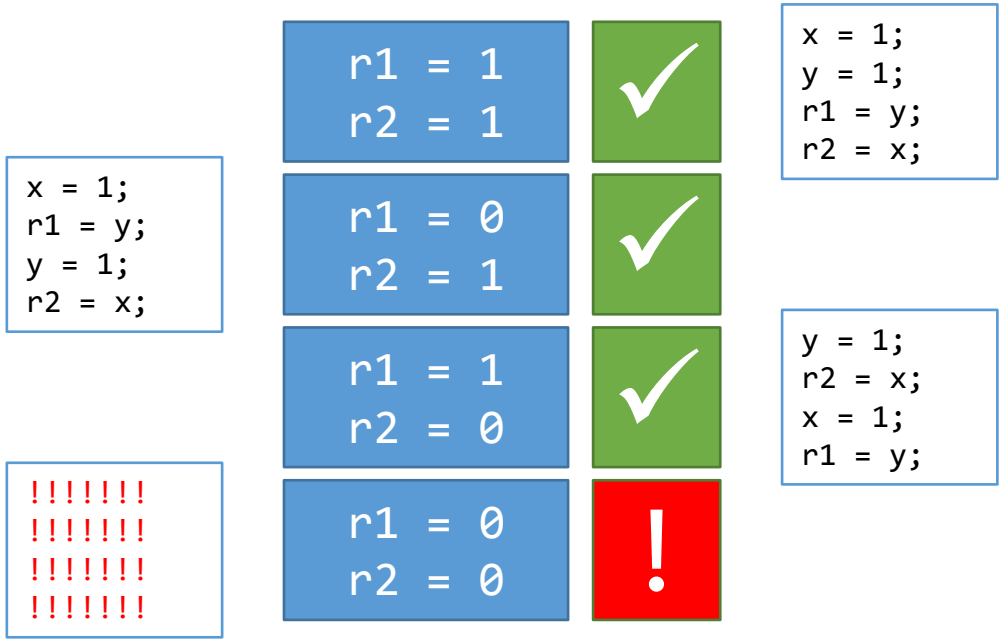


# Proč chceme sekvenční konzistenci

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

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

| Thread 1 | Thread 2 |
|---|---|
| `x = 1;`<br>`r1 = y;` | `y = 1;`<br>`r2 = x;` |

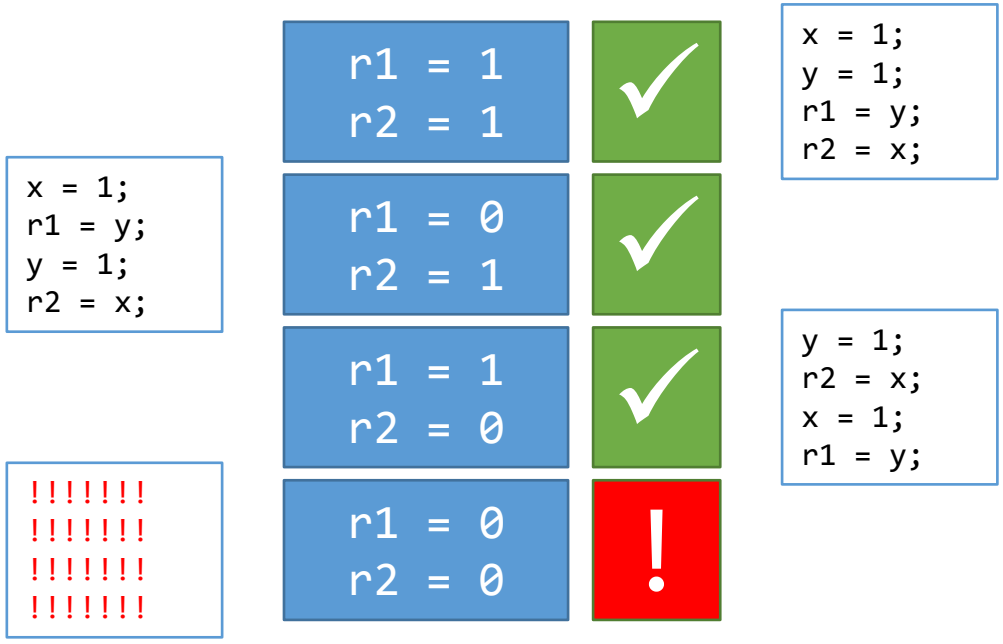


# Proč chceme sekvenční konzistenci

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

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

| Thread 1 | Thread 2 |
|---|---|
| `x = 1;`<br>`r1 = y;` | `y = 1;`<br>`r2 = x;` |

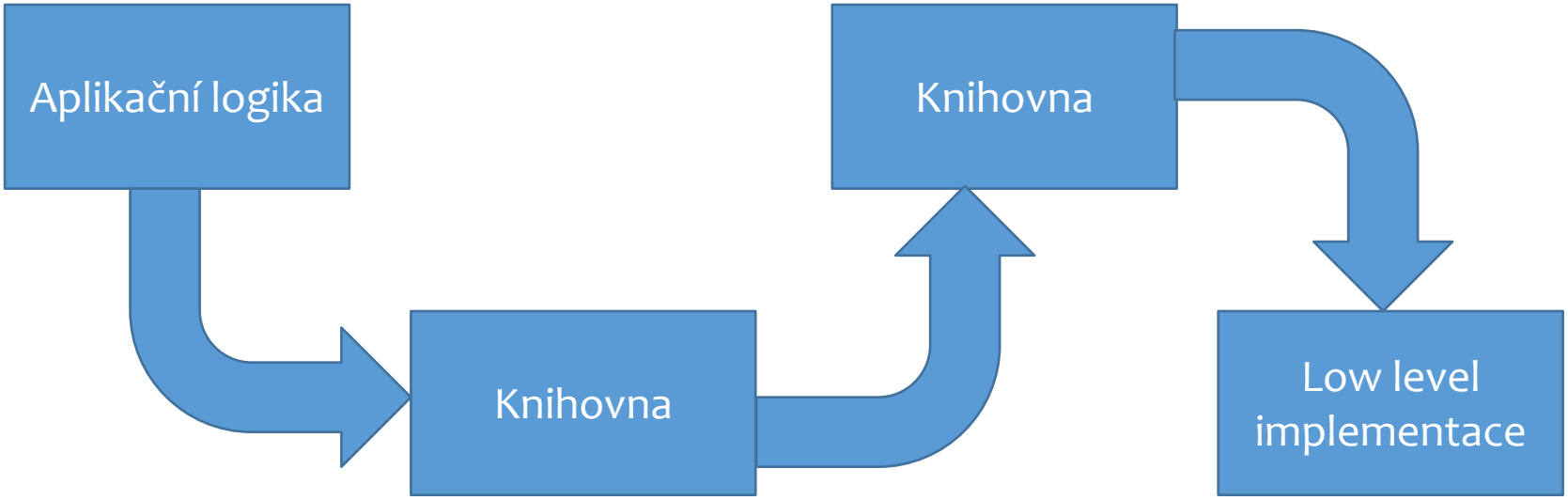



- Poslední výsledek je tzv. Relaxovaný. Jeho existence je důvod, proč doporučujeme používat pouze SC atomické proměnné.

# Děkuji za pozornost.

# Problém

- Při běhu programu mohou vznikat různé chyby. Například nemusíme mít práva ke čtení/zápisu souboru.
- Tyto chyby je potřeba nějak ošetřit.
- Místo, které si umí s chybou poradit je obvykle jiné, než místo kde chyba vznikla.



# Chybové příznaky – Problémy

- Chyby mohou být ignorovány.
  - Programátor musí udělat extra práci, aby ignorovány nebyly.
- Není jisté, kdy a kde k chybě došlo.
  - Pokud po volání funkce nezkontrolujeme errno, už se nedá určit zda k chybě došlo.
  - Ani není možné zjistit, jestli se chyb nevyskytlo více.
- Je to spousta kódu která se špatně čte a špatně píše.
  - Volání funkce následuje mnoho řádků, které čtou errno a reagují na chybu.
  - Funkce se dají skládat, ale pouze za cenu možného přemazání chyby.

```
double sumlogs(const std::vector<double> numbers) {
    double result = 0;
    for (auto n : numbers) {
        result += std::log(n);
    }
    return result;
}
```

# Jak psát bezpečný kód?

- ~~Vždy kontrolujte návratové hodnoty.~~
- ~~Zjistěte které funkce mohou hodit výjimku a rozmyslete si jak na ně reagovat.~~
- ~~Používejte specifikace výjimek~~
- ~~Používejte `try{} catch(){}` bloky.~~
- Vždy zachovejte invarianty.
  - I v případě chyby.
- Rozdělte kód na část, která může selhat a která selhat nemůže.
  - V části která selhat může se připravují změny.
  - V části která selhat nemůže se změny ukládají.

Transakce?

# Cargillův zásobník – kód

Implementace

```
template <class T>
void Stack<T>::push(T element) {
    top++;
    if (top == nelems - 1) {
        T* new_buffer = new
T[nelems += 10];
        if (new_buffer == 0)
            throw "out of memory";
        for (int i = 0; i < top;
i++)
            new_buffer[i] = v[i];
        delete[] v;
        v = new_buffer;
    }
    v[top] = element;
}

template <class T>
T Stack<T>::pop() {
    if (top < 0)
        throw "pop on empty stack";
    return v[top--];
}
```

- V dnešní době operátor new při nepodařené alokaci hodí výjimku.
- Jinak zde není žádný problém.

# Cargillův widget

- Jedná se o další Cargillův "rébus"
- Lze implementovat kopírující přiřazení, pokud libovolná operace na T1 a T2 může hodit výjimku?
  - A pokud povolíme změnit deklaraci `Widgetu`?

Interface

```
template <typename T1
class Widget {
public:
// Poskytuje silnou záruku
Widget& operator=(const Widget& rhs);

private:
    T1 t1_;
    T2 t2_;
};
```