Problemy współbieżności

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school

Łukasz Ziobroń

Not only programming experience:

- C++ and Python developer @ Nokia & Credit Suisse
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Webmaster (HTML, PHP, CSS) @ StarCraft Area

Training experience:

- C++ trainings @ Coders School
- Practial Aspects Of Software Engineering @ PWr, UWr
- Nokia Academy @ Nokia
- Internal corporate trainings

Public speaking experience:

- Academic Championships in Team Programming
- code::dive conference
- code::dive community

Problemy współbieżności - Agenda

- Wyścigi (race conditions, data races)
- Zakleszczenie (deadlock)
- Żywe zakleszczenie (livelock)
- Zagłodzenie (starvation)
- Fałszywe współdzielenie (false sharing)
- Cache ping-pong
- Typowe zagadnienia wielowątkowości
  - Problem producenta i konsumenta
  - Problem czytelników i pisarzy
  - Problem ucztujących filozofów

Wyścigi (race conditions, data races)

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

void abc(int &a) { a = 2; }
void def(int &a) { a = 3; }

int main()
{
    int x = 1;
    std::thread t1(abc, std::ref(x));
    std::thread t2(def, std::ref(x));

    t1.join();
    t2.join();

    std::cout << x << std::endl;
}
```

- Nieokreślona kolejność dostępu do danych
- Niezdefiniowane zachowanie
- Mogą skutkować różnymi scenariuszami zachowania programu
- Przykłady: jednoczesny, niesynchronizowany zapis do tej samej komórki pamięci
- Zapobieganie:
  - `std::mutex`
  - `std::atomic<T>`

Zakleszczenie (deadlock)

```cpp
#include <thread>
#include <mutex>
using namespace std;

class X {
    mutable mutex mtx_;
    int value_ = 0;
public:
    explicit X(int v) : value_(v) {}

    bool operator<(const X & other) const {
        lock_guard<mutex> ownGuard(mtx_);
        lock_guard<mutex> otherGuard(other.mtx_);
        return value_ < other.value_;
    }
};

int main() {
    X x1(5);
    X x2(6);
    thread t1([&](){ x1 < x2; });
    thread t2([&](){ x2 < x1; });
    t1.join();
    t2.join();
    return 0;
}
```

- Sytuacja, w której co najmniej dwa różne wątki czekają na siebie nawzajem, więc żaden nie może się zakończyć.
- Blokada programu
- Nie widać wykorzystania procesora
- Przykłady: problem ucztujących filozofów
- Zapobieganie:
  - `std::scoped_lock (C++17)`
  - `std::lock()`

Żywe zakleszczenie (livelock)

```
void threadAFunc() {
    unsigned counter = 0;
    while (true) {
        std::unique_lock<std::mutex> lockX(resourceX);
        std::this_thread::yield();

        std::unique_lock<std::mutex> lockY(resourceY, std::defer_lock);
        if (not lockY.try_lock())
            continue;

        std::cout << "threadA working: " << ++counter << "\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
    }
}

void threadBFunc() {
    unsigned counter = 0;
    while(true) {
        std::unique_lock<std::mutex> lockY(resourceY);
        std::this_thread::yield();

        std::unique_lock<std::mutex> lockX(resourceX, std::defer_lock);
        if (not lockX.try_lock())
            continue;

        std::cout << "threadB working: " << ++counter << "\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
    }
}
```

- Podobne do zakleszczenia
- Stan wątków zmienia się w czasie, ale program nie postępuje do przodu
- Może być widoczne wykorzystanie procesora
- „Zbyt miłe" wątki oddają swoje zasoby innym wątkom, dając im pierwszeństwo wykonania, przez co wątki nawzajem wywłaszczają sobie zasoby i nie postępują do przodu
- Przykład: powtarzanie cyklicznie tych samych ruchów w szachach
- Zapobieganie:
  - `std::scoped_lock (C++17)`
  - `std::lock()`

Zagłodzenie (starvation)

- Wątek spełnia warunki, aby mógł zostać dopuszczony do zasobów, ale na skutek priorytetów lub kolejkowania może nigdy nie zostać dopuszczony do potrzebnych zasobów, więc może się nigdy nie wykonać
- Przykłady: problem ucztujących filozofów
- Zapobieganie:
  - sprawiedliwy mechanizm schedulera (np. przypisywanie właśnie dopuszczonym wątkom/zadaniom kolejnych numerów rosnąco i faworyzowanie tych z najniższymi numerami)
  - inwersja priorytetów - co jakiś czas zmiana, że jako pierwsze wykonają się wątki/zadania z najniższymi numerami / najwyższymi numerami)

Fałszywe współdzielenie (false sharing)

```
struct foo {
    int x;
    int y;
};

static struct foo f;

int sum_a() {
    int s = 0;
    for (int i = 0; i < 100'000'000; ++i)
        s += f.x;
    return s;
}

void inc_b() {
    for (int i = 0; i < 100'000'000; ++i)
    ++f.y;
}

int main() {
    thread t1(sum_a);
    thread t2(inc_b);
```

```
$> ./02_false_sharing
Threads: 536699
Sequential: 290001
```

- Prawdziwe współdzielenie (true sharing) - współdzielenie tych samych danych przez różne wątki / zadania
- Fałszywe współdzielenie (false sharing) - nie współdzielenie danych, ale linii cache'u
- Wątki / zadania nie współdzielą zasobów, ale ich zasoby są położone blisko siebie w pamięci i mieszczą się jednej linii cache'u.
- Zapisanie czegoś do cache'u wymaga aktualizacji wszystkich kopii tej linii we wszystkich innych pamięciach cache, np. w pozostałych procesorach (cache ping-pong).
- Problem fałszywego współdzielenie nie zostanie wykryty przez thread sanitizer. Można go zauważyć po spadku wydajności.
- Typowo problem będzie występował dla położonych obok siebie danych, takich jak struktury, tablice, wektory.
- Przykłady: 1 wątek modyfikuje wartości pod parzystymi indeksami tablicy, 2 pod nieparzystymi.
- Zapobieganie:
  - podział zadań w taki sposób, aby wątki pracowały na lokalnie odległych fragmentach pamięci.

Cache ping-pong

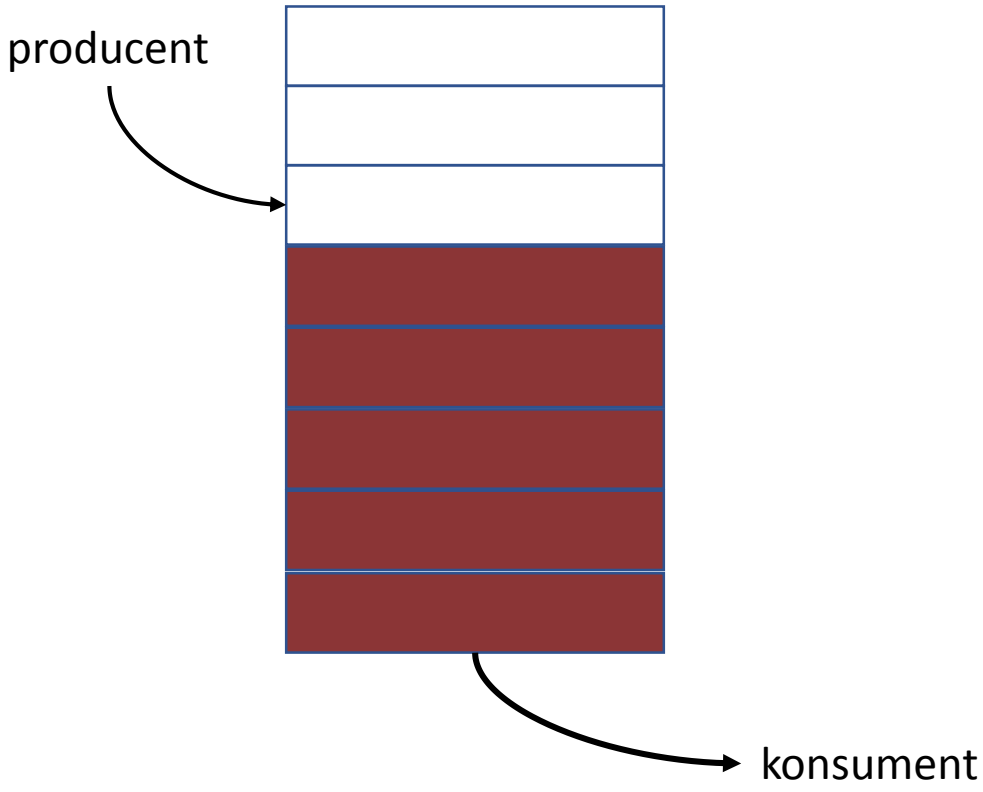
```
struct foo {
    int x;
    int y;
};

static struct foo f;

int sum_a() {
    int s = 0;
    for (int i = 0; i < 100'000'000; ++i)
        s += f.x;
    return s;
}

void inc_b() {
    for (int i = 0; i < 100'000'000; ++i)
    ++f.y;
}
```

- Powodowany m.in. przez zjawisko false sharingu, ale też true sharingu.
- Ciągła aktualizacja linii cache'u pomiędzy procesorami.
- Na skutek cache ping-pong program wielowątkowy jest wolniejszy niż jego sekwencyjna wersja.
- Zapobieganie:
  - odpowiednie modelowanie danych
- Czy false sharing i cache ping pong są tym samym - StackOverflow

Typowe zagadnienia wielowątkowości

- Problem producenta i konsumenta
- Problem czytelników i pisarzy
- Problem ucztujących filozofów

Problem producenta i konsumenta



- Zmienne:
  - p - liczba producentów
  - k - liczba konsumentów
  - bufor - współdzielony zasób
  - N – rozmiar bufora
- Producenci wrzucają produkty do bufora. Producenci czekają, gdy bufor jest pełny.
- Konsumenci pobierają produkty. Konsumenci czekają, gdy bufor jest pusty.
- Wiki

Problem producenta i konsumenta - rozwiązanie

```
template <typename T>
class WaitQueue {
    deque<T> queue_;
    mutable mutex m_;
    condition_variable nonEmpty_;
    using Lock = unique_lock<mutex>;

public:
    void push(const T & element) {
        Lock l(m_);
        queue_.push_front(element);
        nonEmpty_.notify_all();
    }

    T pop() {
        Lock l(m_);
        auto hasData = [&]{
            return not queue_.empty();
        };
        nonEmpty_.wait(l, hasData);
        auto top = queue_.back();
        queue_.pop_back();
        return top;
    }
};
```

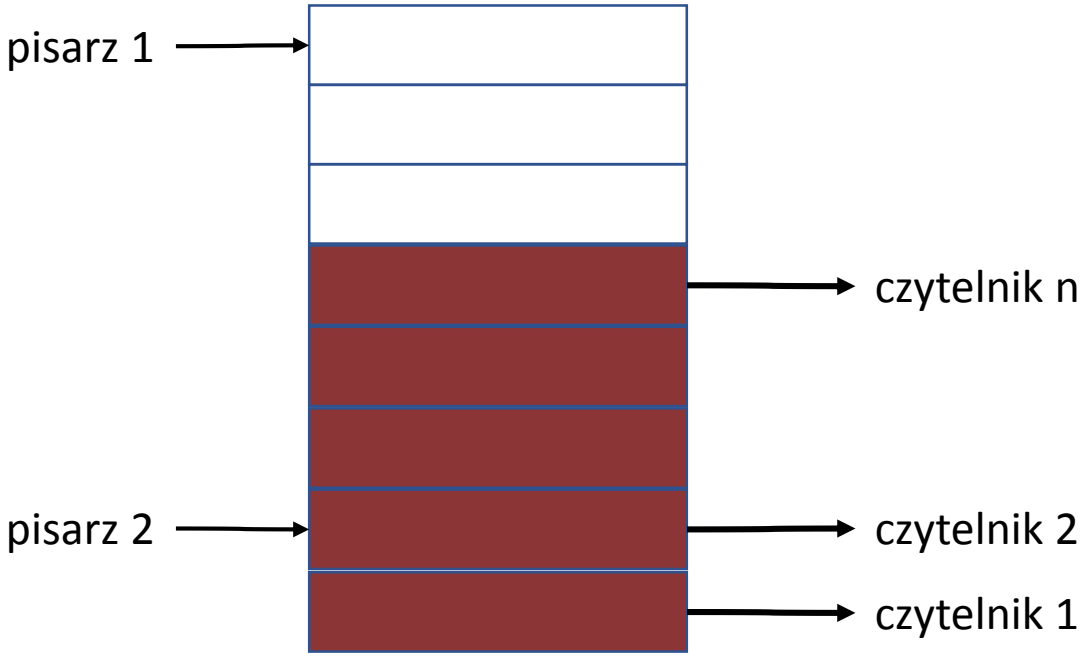
- Kolejka ze zmiennymi warunku
- Konsumenci czekają na zmiennej warunku mówiącej, czy bufor jest pusty
- Producenci informują, że coś wpisali i konsumenci mogą to pobrać
- Dostęp do kolejki jest synchronizowany mutexem
- Nasze rozwiązanie: 04_condition_variable/solutions/02_wait_queue.cpp
- Czego brakuje?
  - Ograniczony rozmiar bufora
  - Producenci czekają na zmiennej warunku mówiącej, czy bufor jest pełny
  - Konsumenci informują, że coś pobrali i producenci mają miejsce, aby coś wyprodukować

Problem producenta i konsumenta - zadanie

```
template <typename T>
class WaitQueue {
    deque<T> queue_;
    mutable mutex m_;
    condition_variable nonEmpty_;
    using Lock = unique_lock<mutex>;

public:
    void push(const T & element) {
        Lock l(m_);
        queue_.push_front(element);
        nonEmpty_.notify_all();
    }

    T pop() {
        Lock l(m_);
        auto hasData = [&]{
            return not queue_.empty();
        };
        nonEmpty_.wait(l, hasData);
        auto top = queue_.back();
        queue_.pop_back();
        return top;
    }
};
```

- Rozszerz rozwiązanie 08_concurrency_problems/exercises/01_producers_consumers.cpp o obsługę ograniczonego bufora:
  - klasa `WaitQueue` powinna mieć dodatkowy parametr szablonowy – rozmiar bufora N
  - Producenci czekają na zmiennej warunku mówiącej, czy bufor jest pełny
  - Konsumenci informują, że coś pobrali i producenci mają miejsce, aby coś wyprodukować

Problem czytelników i pisarzy



- Zmienne:
  - c - liczba czytelników
  - p - liczba pisarzy
  - współdzielony zasób (plik, rekord, inne)
- Jednoczesny dostęp do zasobu może uzyskać dowolna liczba czytelników. Czytelnicy nie modyfikują zasobu.
- Pisarz modyfikuje stan zasobu i może otrzymać tylko wyłączny dostęp.
- Wiki

Problem czytelników i pisarzy - rozwiązanie

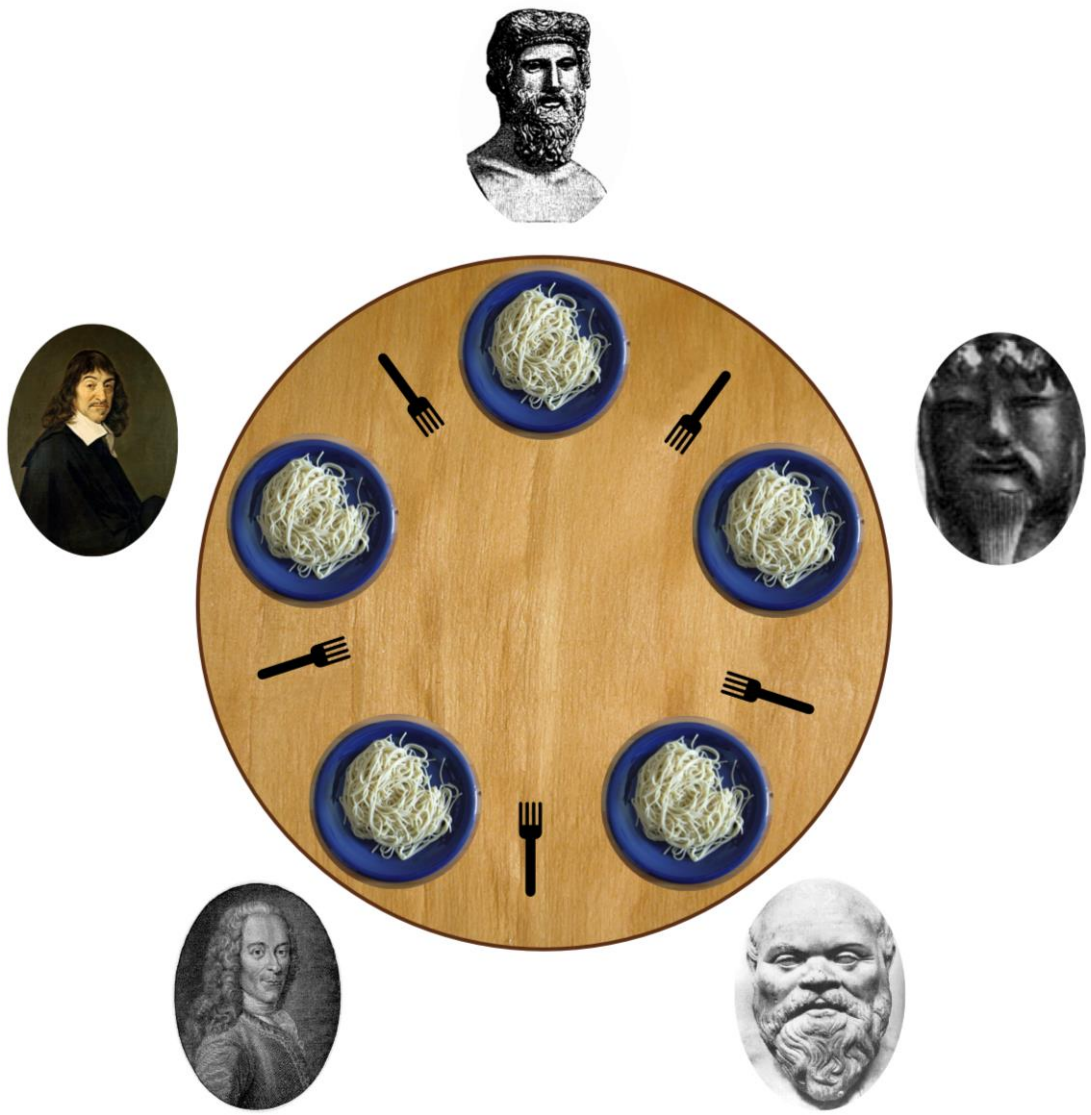
```
vector<int> numbers = {};
shared_mutex numbersMtx;
mutex coutMtx;

void read(int index) {
    shared_lock<shared_mutex> lock(numbersMtx);
    int value = numbers[index];
    lock.unlock();
    process(value);
}

void write() {
    lock_guard<shared_mutex> lock(numbersMtx);
    int newValue = getNextValue();
    numbers.emplace_back(newValue);
}
```

- Dostęp do zasobu synchronizowany mutexem - `std::shared_mutex`
- Czytelnicy blokują mutex za pomocą `shared_lock<shared_mutex>`
- Pisarze blokują mutex za pomocą `unique_lock<shared_mutex>` lub `lock_guard<shared_mutex>`
- Nasze rozwiązanie: 03_data_sharing/solutions/03_shared_mutex.cpp

Problem czytelników i pisarzy - warianty

- Wariant faworyzujący czytelników - pisarze muszą poczekać, aż nie będzie żadnego czytelnika blokującego mutex. Dopóki czytelnicy blokują mutex, każdy nowy czytelnik również go pozyska.
  - Problem - może dojść do zagłodzenia pisarza.
- Wariant faworyzujący pisarzy - jeśli pisarz czeka na mutexie, to nowi czytelnicy ustawiają się w kolejce za pisarzem.
  - Problem - może dojść do zagłodzenia czytelników.
- Inne warianty - np. z użyciem kolejki FIFO, aby zapobiec zagłodzeniom. Każdy wątek (niezależnie, czy będzie to pisarz czy czytelnik) zostaje obsłużony według kolejności w której zażądał zasobu. Czytelnicy oczywiście mogą być obsługiwani jednocześnie, dopóki w kolejce nie pojawi się pisarz.
- Który wariant zaimplementowaliśmy?
  - `std::shared_mutex` nie faworyzuje ani czytelników ani pisarzy. Używa on kolejki dostępu, dzięki czemu zapobiega zagłodzeniom.

Problem ucztujących filozofów



- Filozofowie jedzą lub myślą
- Wersja domyślna: 5 filozofów, 5 sztućców, każdy filozof potrzebuje 2 sztućców do jedzenia.
- Inne warianty: f - liczba filozofów, s - liczba sztućców
- Problemy
  - zakleszczenie (deadlock)
  - żywa zakleszczenie (livelock)
  - zagłodzenie (starvation)
- Wiki

Problem ucztujących filozofów – warianty rozwiązań

- Przy pomocy kelnera - kelner zarządza sztućcami i wie, komu może dać dostęp do sztućców, a komu nie (zapobiega zakleszczeniom)
- Przy użyciu hierarchii zasobów - numeracja sztućców (priorytet), filozof zawsze podnosi najpierw sztuciec z niższym numerem, a odkłada najpierw ten z wyższym. W przypadku C++ wystarczy użyć `std::lock()` / `std::scoped_lock`. Przykład.
- Chandy/Misra - sztućce mogą być czyste lub brudne (na starcie są brudne). Jeśli filozof ma brudny sztuciec, a inny filozof o niego poprosi to musi go umyć i go oddać. Jeśli ma czysty sztuciec, to go nie oddaje. Po skończonym jedzeniu sztućce stają się brudne. To rozwiązanie dobrze się skaluje na więcej filozofów/sztućców. Przykład.

Przydatne linki

- Wyścigi
- Zakleszczenie
- Żywe zakleszczenie
- Zagłodzenie
- Fałszywe współdzielenie
- Scott Meyers – CPU Caches and why you care
- Are cache ping-pong and false sharing the same?

- Problem producenta i konsumenta
- Problem czytelników i pisarzy
- Problem ucztujących filozofów
  - Rozwiązanie z użyciem hierarchii zasobów
  - Rozwiązanie Chandy/Misra

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school