

Współdzielenie danych

CODERS SCHOOL

<https://coders.school>



Łukasz Ziobroń
lukasz@coders.school

Łukasz Ziobroń & Bartosz Szurgot - autorzy



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
- Practical 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



Współdzielenie danych - Agenda

- Współdzielenie danych do odczytu
- Współdzielenie danych z ich modyfikacją
- Przykład: usuwanie węzła z listy dwukierunkowej
- Wyścigi (data races)
- Thread sanitizer
- Zadanie 1: reprodukcja wyścigów pod debuggerem
- Mutex – unikanie wyścigów
- Zadanie 2: zabezpieczenie zapisu do strumienia
- Sekcja krytyczna
- Rodzaje mutexów
- Blokada współdzielona (shared_mutex)
- Menadżery blokad
- Zadanie 3: zabezpieczenie programu za pomocą odpowiednich blokad
- Mutex – dobre praktyki
- Zakleszczenie (deadlock)
- Przykład: zakleszczenie w porównaniu
- Współdzielenie danych - podsumowanie

Współdzielenie danych do odczytu

- Wyobraź sobie, że oglądasz TV ze znajomymi. Wielu znajomych może oglądać TV razem z Tobą. Żadne z was nie wpływa w żaden sposób na oglądany materiał (nikt nie ma pilota ;)).
- Dokładnie tak samo jest ze współdzieleniem danych tylko do odczytu
 - Wątki = Ty i znajomi
 - Dane = film
 - Odczyt danych przez jeden wątek nie zakłóca odczytu danych przez inny wątek
- Brak modyfikacji danych = brak problemów

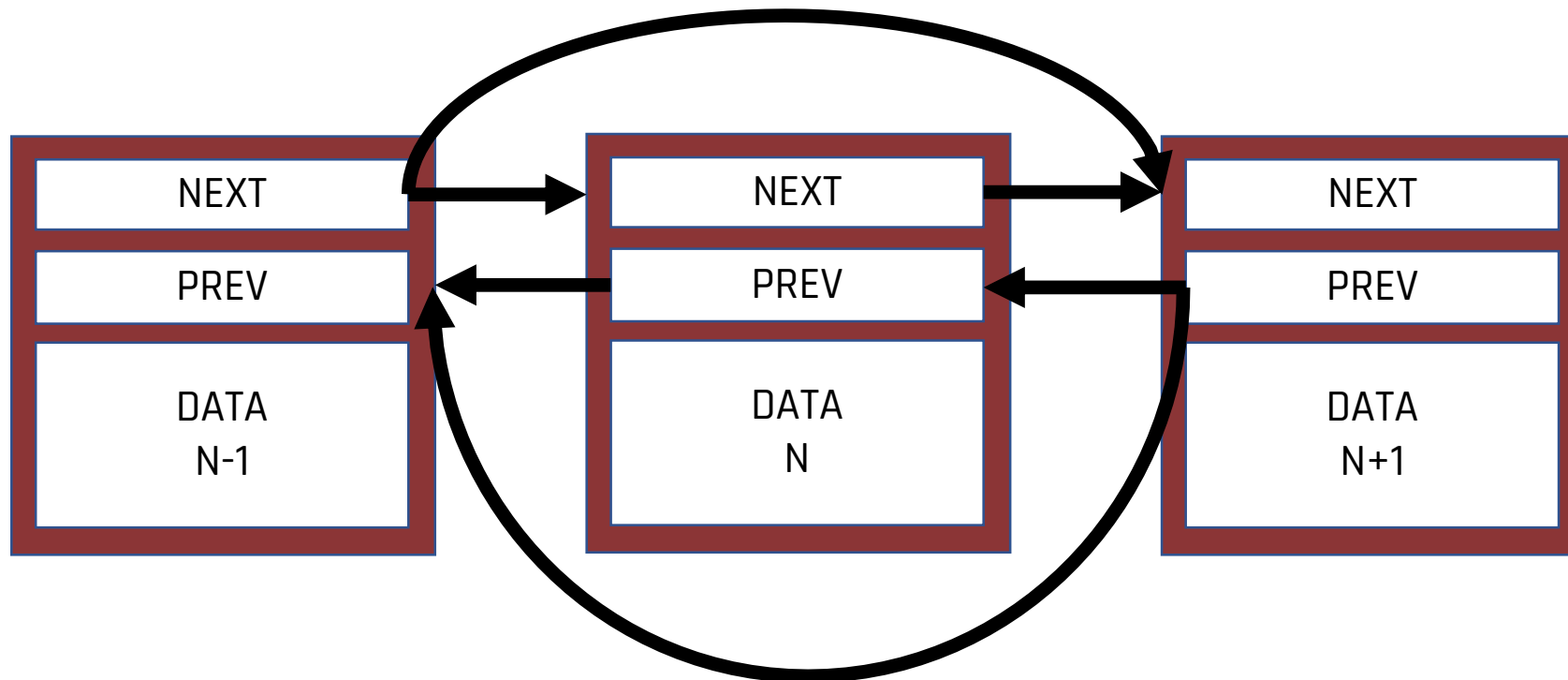
Współdzielenie danych z ich modyfikacją

- Wyobraź sobie, że współdzielisz mieszkanie ze współlokatorami. Każde z was ma własny pokój, ale toaleta jest wspólna. Gdy ktoś w niej jest, musi zablokować drzwi. Jeśli chcecie skorzystać z toalety w tym samym czasie nie możecie tego zrobić. Długie oczekiwanie, aż inna osoba przestanie okupować toaletę jest frustrujące. Toaleta po skorzystaniu jest w innym stanie niż przed (mniej papieru, zużyta woda, ciekawsze zapachy...)
- Dokładnie tak samo jest ze współdzieleniem danych 😊
 - Wątki = Ty i współlokator
 - Zasób (dane) = toaleta (papier, woda)
 - Mutex = blokada drzwi
- Modyfikacja danych (użycie toalety) = problem – wymagana jest synchronizacja
- Co gdyby nie było drzwi z blokadą do toalety? Ktoś mógłby nam spuścić wodę w trakcie jej użytkowania albo podebrać ostatni listek papieru toaletowego.

Przykład: usuwanie węzła z listy dwukierunkowej

1. Znajdź węzeł N do usunięcia
2. Ustaw wskaźnik NEXT w węźle N-1 na N+1
3. Ustaw wskaźnik PREV w węźle N+1 na N-1
4. Usuń węzeł N

Pomiędzy krokami 2 i 3 wskaźniki są ustawione w nieprawidłowy, niejednolity sposób



Przykład: usuwanie węzła z listy dwukierunkowej

- Usuwanie węzła z listy składa się z kilku kroków, następuje modyfikacja kilku węzłów
- Co jeśli inny wątek będzie iterował po liście gdy trwa usuwanie węzła N?
- Co jeśli inny wątek zacznie usuwać węzeł N+1 w czasie gdy trwa usuwanie węzła N?
- Efekty mogą być różne w zależności w którym momencie wątki weszły sobie w paradę
- Zjawisko to nosi nazwę *wyścigów (race conditions)*
- Wyścigi zazwyczaj występują, gdy trzeba zmodyfikować dwa lub więcej oddzielnych kawałków danych, tak jak dwa wskaźniki na przykładzie z listą

Czy są tu wyścigi?

```
#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;
}
```

Co wyświetli się na ekranie?

[illegible]

Wyścigi (race conditions)

- Wyścigi generują niedeterministyczne (losowe) zachowania programu
- Wyścig = Niezdefiniowane zachowanie
- Niechciane skutki uboczne zazwyczaj nie występują podczas większości uruchomień danej procedury.
- Problemy mogą występować rzadziej niż 1 na 1000 uruchomień
- Wyścigi niesamowicie trudno wykryć. Zazwyczaj jeśli procesor nie jest obciążony to wszystko działa jak należy. Im bardziej obciążony procesor tym bardziej rośnie ryzyko innej kolejności dostępu do danych. Takie problemy trudno jest zreprodukować.
- Problem wyścigów jest krytyczny czasowo i może być zupełnie niemożliwy do wychwycenia pod debuggerem. Debugger wpływa na czasy wykonywania poszczególnych instrukcji.
- Znacznie łatwiej jest zapobiegać wyścigom, bo leczenie jest długotrwałe i kosztowne
- Thread Sanitizer (TSan) – data race detector

Thread sanitizer

```
$> g++ 01_threads_write.cpp -lpthread -fsanitize=thread -O2 -g
$> ./a.out
```

```
=====
WARNING: ThreadSanitizer: data race (pid=3180)
```

```
Write of size 4 at 0x7ffc806f24ac by thread T2:
```

```
#0 def(int&) /home/ziobron/coders.school/data_sharing/threads_write.cpp:6 (a.out+0x12c5)
#1 void std::__invoke_impl<void, void (*) (int&), std::reference_wrapper<int> >(std::__invoke_other, void (*&&)(int&), std::reference_wrapper<int>&&) /usr/include/c++/7/bits/invoke.h:60 (a.out+0x130e)
#2 std::__invoke_result<void (*) (int&), std::reference_wrapper<int> >::type std::__invoke<void (*) (int&), std::reference_wrapper<int> >(void (*&&)(int&), std::reference_wrapper<int>&&) /usr/include/c++/7/bits/invoke.h:95 (a.out+0x130e)
#3 decltype (__invoke((__S_declval<0ul>()), (__S_declval<1ul>()))) std::thread::Invoker<std::tuple<void (*) (int&), std::reference_wrapper<int> > >::__M_invoke<0ul, 1ul>(std::_Index_tuple<0ul, 1ul>) /usr/include/c++/7/thread:234 (a.out+0x130e)
#4 std::thread::Invoker<std::tuple<void (*) (int&), std::reference_wrapper<int> > >::operator()() /usr/include/c++/7/thread:243 (a.out+0x130e)
#5 std::thread::State_impl<std::thread::Invoker<std::tuple<void (*) (int&), std::reference_wrapper<int> > > >::__M_run() /usr/include/c++/7/thread:186 (a.out+0x130e)
#6 <null> <null> (libstdc++.so.6+0xbd57e)
```

```
Previous write of size 4 at 0x7ffc806f24ac by thread T1:
```

```
#0 abc(int&) /home/ziobron/coders.school/data_sharing/threads_write.cpp:5 (a.out+0x1295)
#1 void std::__invoke_impl<void, void (*) (int&), std::reference_wrapper<int> >(std::__invoke_other, void (*&&)(int&), std::reference_wrapper<int>&&) /usr/include/c++/7/bits/invoke.h:60 (a.out+0x130e)
#2 std::__invoke_result<void (*) (int&), std::reference_wrapper<int> >::type std::__invoke<void (*) (int&), std::reference_wrapper<int> >(void (*&&)(int&), std::reference_wrapper<int>&&) /usr/include/c++/7/bits/invoke.h:95 (a.out+0x130e)
#3 decltype (__invoke((__S_declval<0ul>()), (__S_declval<1ul>()))) std::thread::Invoker<std::tuple<void (*) (int&), std::reference_wrapper<int> > >::__M_invoke<0ul, 1ul>(std::_Index_tuple<0ul, 1ul>) /usr/include/c++/7/thread:234 (a.out+0x130e)
#4 std::thread::Invoker<std::tuple<void (*) (int&), std::reference_wrapper<int> > >::operator()() /usr/include/c++/7/thread:243 (a.out+0x130e)
#5 std::thread::State_impl<std::thread::Invoker<std::tuple<void (*) (int&), std::reference_wrapper<int> > > >::__M_run() /usr/include/c++/7/thread:186 (a.out+0x130e)
#6 <null> <null> (libstdc++.so.6+0xbd57e)
```

```
Location is stack of main thread.
```

```
Location is global '<null>' at 0x000000000000 ([stack]+0x00000001f4ac)
```

```
Thread T2 (tid=3183, running) created by main thread at:
```

```
#0 pthread_create <null> (libtsan.so.0+0x2bcfe)
#1 std::thread::M_start_thread(std::unique_ptr<std::thread::State, std::default_delete<std::thread::State> >, void (*)()) <null> (libstdc++.so.6+0xbd834)
#2 main /home/ziobron/coders.school/data_sharing/threads_write.cpp:12 (a.out+0x1065)
```

```
Thread T1 (tid=3182, finished) created by main thread at:
```

```
#0 pthread_create <null> (libtsan.so.0+0x2bcfe)
#1 std::thread::M_start_thread(std::unique_ptr<std::thread::State, std::default_delete<std::thread::State> >, void (*)()) <null> (libstdc++.so.6+0xbd834)
#2 main /home/ziobron/coders.school/data_sharing/threads_write.cpp:11 (a.out+0x1041)
```

```
SUMMARY: ThreadSanitizer: data race /home/ziobron/coders.school/data_sharing/threads_write.cpp:6 in def(int&)
```

```
=====
3
ThreadSanitizer: reported 1 warnings
```

Zadanie 1: reprodukcja wyścigów pod debuggerem

- Skompiluj program `threads_write.cpp`
`g++ 01_threads_write.cpp -lpthread -g`
- Uruchom program pod debuggerem (gdb lub inny)
`gdb --tui ./a.out`
- Spraw, aby na końcu programu w zmiennej `x` była wartość 2
- Spraw, aby na końcu programu w zmiennej `x` była wartość 3
- Przydatne komendy
 - `b 5` – ustawia breakpoint w 5 linii
 - `watch x` – obserwowanie zmian zmiennej `x` (debugger zatrzyma się gdy nastąpi jej modyfikacja)
 - `c` – kontynuowanie debugowania
 - `info threads` – informacje o wątkach
 - `thread 3` – przełączenie na wątek 3
 - `n` – następna instrukcja
 - `fin` – wykonanie wszystkiego do końca bieżącej funkcji
 - `up` – przejście do wyższej ramki stosu
 - `down` – przejście do niższej ramki stosu
 - `del br` – usunięcie wszystkich breakpointów
 - `CTRL + L` – odświeżenie widoku

Jedna z możliwych sekwencji:

- `b 5`
- `b 17`
- `r`
- `c`
- `p x`
 - `$ = 2`
- `del br`
- `b 6`
- `b 17`
- `r`
- `c`
- `p x`
 - `$ = 3`

Mutex – unikanie wyścigów

- Mutex (Mutual Exclusion) – wzajemne wykluczanie
- Implementacja blokady
- `#include <mutex>`
- `std::mutex`
- Najważniejsze operacje:
 - `void lock()` – zablokowanie mutexu. Operacja blokująca. Jeśli mutex jest zablokowany przez inny wątek to oczekujemy na odblokowanie
 - `void unlock()` – odblokowanie mutexu
 - `bool try_lock()` – zablokowanie mutexu. Operacja nieblokująca. Jeśli mutex jest już zablokowany przez inny wątek to kontynuuje dalsze wykonanie wątku. Zwraca `true`, jeśli udało się zablokować mutex

Zadanie 2: zabezpieczenie zapisu do strumienia

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

void do_work(int id) {
    this_thread::sleep_for(100ms);
    cout << "Thread [" << id << "]: " << "Job done!" << endl;
}

int main() {
    vector<thread> threads;
    for (int i = 0; i < 20; i++) {
        threads.emplace_back(thread(do_work, i));
    }
    for (auto && t : threads) {
        t.join();
    }
    return 0;
}
```

- Zabezpiecz kod, tak aby każdy wątek mógł bezpiecznie wpisać do strumienia swój pełny tekst
- Nie powinny być możliwe przypadki takie jak poniższe

```
$> g++ 02_threads_in_collection.cpp -lpthread
$> ./a.out
```

```
...
Thread [10]: Job done!
Thread [9]: Thread [Job done!11]: Job done!
Thread [6]: Job done!
```

```
...
$> ./a.out
```

```
...
Thread [Thread [5]: Job done!
Thread [13]: Job done!
17]: Job done!
...
```


Zadanie 2 – rozwiązanie z użyciem mutexu

```
#include <vector>
#include <thread>
#include <chrono>
#include <iostream>
#include <mutex>
using namespace std;

mutex m;

void do_work(int id) {
    this_thread::sleep_for(100ms);
    m.lock();
    cout << "Thread [" << id << "]: " << "Job done!" << endl;
    m.unlock();
}

int main() {
    vector<thread> threads;
    for (int i = 0; i < 20; i++) {
        threads.emplace_back(thread(do_work, i));
    }
    for (auto && t : threads) {
        t.join();
    }
    return 0;
}
```

- Działa... w tym przypadku
- Kod zablokowany mutexem to tzw. sekcja krytyczna
- Czas trwania blokady musi być możliwie najkrótszy
 - błędem jest blokowanie funkcji `sleep_for()`
- Globalny mutex
 - zazwyczaj mutexy umieszcza się w klasach, których operacje mają być blokowane
- Co jeśli podczas trwania blokady wystąpi wyjątek?
 - nie zostanie zawołane `unlock()`
 - inne wątki nigdy nie skończą pracy
 - brak RAI 
- Czy można lepiej?

Zadanie 2 - rozwiązanie z użyciem lock_guard

```
#include <vector>
#include <thread>
#include <chrono>
#include <iostream>
#include <mutex>
using namespace std;

void do_work(int id, mutex & m) {
    this_thread::sleep_for(100ms);
    lock_guard<mutex> lock(m);
    cout << "Thread [" << id << "]: " << "Job done!" << endl;
}

int main() {
    mutex m;
    vector<thread> threads;
    for (int i = 0; i < 20; i++) {
        threads.emplace_back(thread(do_work, i, ref(m)));
    }
    for (auto && t : threads) {
        t.join();
    }
    return 0;
}
```

- Działa
- Bezpieczne w przypadku wystąpienia wyjątku - jest RAII 😊
- Mutex przekazany przez referencję
- Formatowanie i składanie tekstu podczas trwania blokady trochę trwa - strumienie nie słyną z wydajności
- Czy można szybciej?

Zadanie 2 - rozwiązanie z użyciem lock_guard i stringstream

```
#include <vector>
#include <thread>
#include <chrono>
#include <iostream>
#include <mutex>
#include <sstream>
using namespace std;

void do_work(int id, mutex & m) {
    this_thread::sleep_for(100ms);
    stringstream ss;
    ss << "Thread [" << id << "]: " << "Job done!" << endl;
    lock_guard<mutex> lock(m);
    cout << ss.rdbuf();
}

int main() {
    mutex m;
    vector<thread> threads;
    for (int i = 0; i < 20; i++) {
        threads.emplace_back(thread(do_work, i, ref(m)));
    }
    for (auto && t : threads) {
        t.join();
    }
    return 0;
}
```

- Działa
- Bezpieczne w przypadku wystąpienia wyjątku - Jest RAII 😊
- Mutex przekazany przez referencję
- Formatowanie tekstu poza blokadą
- Jedyna operacja podczas blokady to wypisanie zawartości stringa

Sekcja krytyczna

```
class SafeList {  
    mutex m;  
  
public:  
    void remove_safely(Node* node) {  
        lock_guard<mutex> lock(m);  
        node->prev->next = node->next;  
        node->next->prev = node->prev;  
        delete node;  
    } // automatic unlocking  
  
    // other methods  
};
```

- Sekcja krytyczna to fragment programu, który może być wykonywany tylko przez 1 wątek na raz
- Zazwyczaj jest realizowana z użyciem mutexu jako blokady dostępu
- Zawsze używaj menadżera blokady (np. `lock_guard<mutex>`) w celu zapewnienia mechanizmu RAII
- Całą sekcję krytyczną zazwyczaj wydziela się do osobnej funkcji

Rodzaje mutexów

- **mutex**
 - `void lock()` - bieżący wątek jest wstrzymany, aż do momentu pozyskania blokady
 - `void unlock()` - jeżeli bieżący wątek jest posiadaczem blokady, to następuje jej zwolnienie
 - `bool try_lock()` - próba pozyskania blokady bez wstrzymywania bieżącego wątku. Zwraca `true` jeśli blokada została pozyskana, w przeciwnym wypadku zwraca `false`
- **timed_mutex**
 - posiada metody umożliwiające zdefiniowanie maksymalnego czasu oczekiwania na pozyskanie blokady przez wątek
 - `bool try_lock_until(timeout_time)`
 - `bool try_lock_for(timeout_duration)`
- **recursive_mutex**
 - Ten sam wątek może wielokrotnie pozyskać mutex poprzez wywołanie metody `lock()` lub `try_lock()`
 - Aby zwolnić mutex wątek musi odpowiednią ilość razy wywołać `unlock()`
- **recursive_timed_mutex**
 - posiada właściwości `timed_mutex`
 - posiada właściwości `recursive_mutex`
- **shared_mutex**
 - możliwość pozyskiwania blokad współdzielonych przy pomocy metod:
 - `void lock_shared()`
 - `bool try_lock_shared()`
 - `bool try_lock_shared_for(rel_time)`
 - `bool try_lock_shared_until(abs_time)`
 - `void unlock_shared()`

Blokada współdzielona – shared_mutex

```
#include <deque>
#include <shared_mutex>

std::deque<int> ids;
std::shared_mutex mtxIds;

int getIdsIndex() { /* ... */ }
void process(int) { /* ... */ }
int newValue()    { /* ... */ }

void reader() {
    int index = getIdsIndex();
    std::shared_lock<std::shared_mutex> lock(mtxIds);
    int value = ids[index];
    lock.unlock();
    process(value);
}

void writer() {
    int index = getIdsIndex();
    std::lock_guard<std::shared_mutex> lock(mtxIds);
    ids[index] = newValue();
}
```

- Współdzielone blokady używane są w trakcie czytania danych
 - shared_lock<shared_mutex>
- Do zapisu danych trzeba pozyskać wyłączną blokadę
 - lock_guard<shared_mutex>
 - unique_lock<shared_mutex>

Menadżery blokad

- `lock_guard<Mutex>`
 - najprostszy, główny wybór
 - konstruktor wywołuje `lock()` na mutexie
 - destruktor woła `unlock()`
 - jest niekopiowalny
- `unique_lock<Mutex>`
 - opóźnione blokowanie
 - próby zablokowania ograniczone czasowo
 - blokowanie rekursywne
 - podejmowanie nieblokujących prób pozyskania blokady (`try_lock`)
 - korzystanie z `timed_mutex`
 - korzystanie ze zmiennych warunkowych (`condition variable`)
 - jest niekopiowalny
 - jest przenaszalny (`move`)
- `scoped_lock<Mutexes...>`
 - blokuje kilka mutexów
 - zapobiega zakleszczeniom (`deadlock`)
 - konstruktor blokuje wszystkie mutexy w bezpiecznej kolejności, unikając blokad
 - destruktor odblokowuje je w kolejności odwrotnej
 - jest niekopiowalny
- `shared_lock<SharedMutex>`
 - menadżer współdzielonych blokad do odczytu zmiennych
 - kilka wątków może współdzielić blokadę `SharedMutex`
 - inny wątek może pozyskać blokadę `SharedMutex` na wyłączność za pomocą menadżera `unique_lock`
 - te same własności co `unique_lock`

Zadanie 3: zabezpieczenie programu za pomocą odpowiednich blokad

```
#include <vector>
#include <iostream>
#include <thread>
using namespace std;

vector<int> numbers = {};

int getNextValue() {
    static int i = 0;
    return i+=10;
}

void read(int index) {
    int value = numbers[index];
    cout << value << " ";
}

void write() {
    int newValue = getNextValue();
    numbers.emplace_back(newValue);
}

int main() {
    vector<thread> writers;
    for(int i = 0; i < 10; i++)
        writers.emplace_back(write);
    for(auto && writer : writers)
        writer.join();

    cout << "Writers produced: ";
    for(const auto & n : numbers)
        cout << n << " ";
    cout << endl;

    cout << "Readers consumed: ";
    vector<thread> readers;
    for(int i = 0; i < 10; i++)
        readers.emplace_back(read, i);
    for(auto && reader : readers)
        reader.join();

    cout << endl;
    return 0;
}
```

- Użyj blokad współdzielonych i/lub zwykłych
- Użyj odpowiednich menadżerów blokad
- Skompiluj w C++17 i z Tsanem

Wskazówki:

- Współdzielone blokady używane są w trakcie czytania danych
 - `shared_lock<shared_mutex>`
- Do zapisu danych trzeba pozyskać wyłączną blokadę
 - `lock_guard<shared_mutex>`
 - `unique_lock<shared_mutex>`

```
$> g++ 03_shared_mutex.cpp -lpthread -std=c++17 -fsanitize=thread
```

Zadanie 3 - rozwiązanie

```
#include <shared_mutex>
#include <mutex>
#include <vector>
#include <iostream>
#include <thread>
using namespace std;

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

int getNextValue() {
    static int i = 0;
    return i+=10;
}

void read(int index) {
    shared_lock<shared_mutex> lock(numbersMtx);
    int value = numbers[index];
    lock.unlock();
    lock_guard<mutex> coutLock(coutMtx);
    cout << value << " ";
}

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

```
int main() {
    vector<thread> writers;
    for(int i = 0; i < 10; i++)
        writers.emplace_back(write);
    for(auto && writer : writers)
        writer.join();

    cout << "Writers produced: ";
    for(const auto & n : numbers)
        cout << n << " ";
    cout << endl;

    cout << "Readers consumed: ";
    vector<thread> readers;
    for(int i = 0; i < 10; i++)
        readers.emplace_back(read, i);
    for(auto && reader : readers)
        reader.join();

    cout << endl;
    return 0;
}
```

Mutex – dobre praktyki

- Zawsze używaj mutexu z odpowiednim menadżerem (wrapperem RAII):
 - `lock_guard` – najprostszy, główny wybór
 - `unique_lock` – opóźnione blokowanie, próby zablokowania ograniczone czasowo, ...
 - `scoped_lock` – blokuje kilka mutexów, zapobiega zakleszczeniom (deadlock)
 - `shared_lock` – współdzielona blokada do odczytu zmiennych
- Blokuj zawsze gdy to konieczne
- Unikaj blokowania gdzie tylko się da
- Czas trwania blokady powinien być jak najkrótszy
 - Jeśli danych do modyfikacji jest dużo i nie są one ułożone obok siebie w pamięci, to czas trwania blokady mutexu będzie bardzo długi, aby wszystkie dane zdążyły się pobrać do pamięci cache.
 - Operacje I/O (wejścia/wyjścia) takie jak odczyty i zapisy plików, pobieranie danych przez kartę sieciową to długo trwające procesy
- Jeśli wątki potrzebują zablokować kilka mutexów, to powinny być one zawsze blokowane w tej samej kolejności w każdym używającym ich wątku. Zapobiega to zakleszczeniom (deadlock)

Zakleszczenie (deadlock)



sytuacja, w której co najmniej dwa różne wątki czekają na siebie nawzajem, więc żaden nie może się zakończyć.

Zadanie 4: zakleszczenie w porównaniu

```
#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;
}
```

- Wątek t1:
 - $x1 < x2$
 - `x1: mtx_.lock()`
 - `x2: mtx_.lock()`
 - DEADLOCK
- Wątek t2:
 - $x2 < x1$
 - `x2: mtx_.lock()`
 - `x1: mtx_.lock()`
 - DEADLOCK
- Zakleszczenie występuje losowo przy niektórych uruchomieniach
- Użyj `std::scoped_lock` do rozwiązania problemu zakleszczenia

```
$> g++ 04_deadlock.cpp -lpthread -fsanitize=thread
$> ./a.out
...
WARNING: ThreadSanitizer: lock-order-inversion
(potential deadlock) (pid=5509)
    Cycle in lock order graph: M6 (0x7ffffac4d4430)
=> M7 (0x7ffffac4d4460) => M6
...
$> for i in {1..20}; do ./a.out; done
```


Rozwiązania: zakleszczenie w porównaniu

```
// original example with deadlock
bool operator<(const X & other) const {
    lock_guard<mutex> ownGuard(mtx_);
    lock_guard<mutex> otherGuard(other.mtx_);
    return value_ < other.value_;
}

// defer unique_lock + lock
bool operator<(const X & other) const {
    unique_lock<mutex> l1(mtx_, defer_lock);
    unique_lock<mutex> l2(other.mtx_, defer_lock);
    lock(ownLock, otherLock);
    return value_ < other.value_;
}

// lock + adopt_lock_guard
bool operator<(const X & other) const {
    lock(mtx_, other.mtx_);
    lock_guard<mutex> l1(mtx_, adopt_lock);
    lock_guard<mutex> l2(other.mtx_, adopt_lock);
    return value_ < other.value_;
}

// scoped lock (C++17) - preferred solution
bool operator<(const X & other) const {
    scoped_lock bothLock(mtx_, other.mtx_);
    return value_ < other.value_;
}
```

- funkcja `std::lock()`
 - gwarantuje zablokowanie wszystkich muteksów bez zakleszczenia niezależnie od kolejności ich pozyskiwania
 - wymaga przekazania jako parametrów opóźnionych blokad (`defer_lock`) typu `std::unique_lock`
 - alternatywnie wymaga przekazania muteksów, a następnie utworzenia zablokowanych blokad (`adopt_lock`) typu `std::lock_guard`
 - `l1` i `l2` nie blokują muteksów w konstruktorze, robi to funkcja `std::lock()`
- menadżer RAII `std::scoped_lock` (C++17)
 - wymaga przekazania muteksów w konstruktorze, które blokuje tak samo jak funkcja `std::lock()`
 - nie wymaga tworzenia dodatkowych obiektów blokad

Zakleszczenie

- Może wystąpić gdy mamy 2 lub więcej muteksów blokowanych w różnych kolejnościach
- Blokuj muteksy wszędzie w tej samej kolejności, ale ręczna zmiana kolejności lock_guardów nie zawsze naprawi program (jak na przykładzie z operatorem<)
- Używaj std::scoped_lock – blokady, która przyjmuje kilka muteksów i blokuje je zawsze w określonej (prawidłowej) kolejności

Współdzielenie danych - podsumowanie

- Współdzielenie danych tylko do odczytu (read-only) jest bezpieczne i nie wymaga stosowania blokad. Słowo `const` oznacza bezpieczeństwo danych (read-only).
- Gdy jakiś wątek zablokuje mutex, to każdy inny wątek chcący go zablokować musi poczekać na odblokowanie. Blokowanie zbyt dużych sekcji krytycznych jest błędem.
- Problem zakleszczenia (deadlock) występuje, kiedy 2 mutexy (lub więcej) są blokowane przez wątki w różnych kolejnościach albo mamy 1 mutex, który nie zostanie odblokowany np. z powodu wyjątku i program nie przejdzie do funkcji `unlock()`.
- Mutexy znacznie spowalniają wykonywanie programu, ale zabezpieczają dane, jeśli są użyte w poprawny sposób.
- W małych programach mutexy można trzymać jako obiekty globalne, ale w większych powinny być trzymane w odpowiedniej klasie, której operacje mają być blokowane. Jeśli taka klasa ma kilka zasobów, które mogą być modyfikowane niezależnie, to należy mieć kilka mutexów (1 na każdy zasób). Trzeba uważać na zakleszczenie.
- Pomimo zablokowanych mutexów, jeśli używamy gdzieś wskaźników lub referencji do obiektów, które modyfikujemy w sekcji krytycznej, to cała ochrona jest na nic. Ochrona danych mutexami wymaga więc uważnego modelowania interfejsów.
- Zawsze używaj muteksów poprzez menadżery blokad (wrappery RAII)
- Używaj `std::atomic<T>` (zmiennych atomowych). Dają one wygodny sposób lekkiej synchronizacji danych bez użycia mutexów, ale o tym w kolejnej części 😊

Praca domowa

- Zaimplementuj [problem uczających filozofów](#) z użyciem wątków i mutexów.
 - Każdy filozof ma być kontrolowany przez oddzielny wątek.
 - Każdy sztuciec ma być chroniony przez 1 mutex
 - Postaraj się o wizualizację problemu
 - Strzeż się zakleszczeń
 - Pobaw się liczbą filozofów i zobacz czy zmienia się zachowanie programu
 - Zadanie dodatkowe: Strzeż się zagłodzenia (starvation). Jest to sytuacja w której przynajmniej 1 wątek z powodu implementacji lub swojego niższego priorytetu nigdy nie dostanie wszystkich wymaganych zasobów. Doimplementuj w tym celu pewien mechanizm, który zapobiegnie zagłodzeniu.

Przydatne linki

- [What every programmer should know about memory](#) – Ulrich Drepper
- [The C++ memory model](#)
- [Problem uczących filozofów](#)

CODERS SCHOOL

<https://coders.school>

ASK A NINJA



Łukasz Ziobroń
lukasz@coders.school