Współbieżne wzorce projektowe

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:

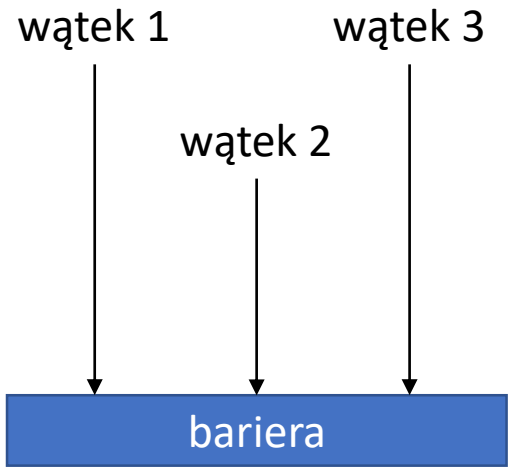
- Academic Championships in Team Programming
- code::dive conference
- code::dive community

Współbieżne wzorce projektowe - Agenda

- Barrier
- Guarded suspension
- Monitor Object
- Read write lock
- Scheduler
- Double checked locking
- Thread pool
- Active Object
- Reactor/Proactor

Bariera (barrier)

- Punkt synchronizacji kilku wątków
- Wątki, które dotrą do bariery czekają aż wszystkie wymagane wątki do niej dotrą
- Dopiero gdy wszystkie wymagane wątki dotarły do bariery, jest ona przełamywana i wszystkie wątki mogą ruszyć dalej
- https://en.cppreference.com/w/cpp/experimental/barrier (C++20?)
- Obecnie można zamodelować barierę jako czekanie na zakończenie kilku wątków za pomocą join(). Różnica jest taka, że musimy potem od nowa wystartować kolejne zadania. Bariera blokuje wykonanie wątków i wznawia ich pracę bez ich zakańczania
- https://en.wikipedia.org/wiki/Barrier_(computer_science)



Guarded suspension

- Oczekiwanie na blokadzie + spełniony warunek
- `std::condition_variable` implementuje wzorzec guarded suspension
- `void wait(std::unique_lock<std::mutex>& lock, Predicate pred)`
- https://en.wikipedia.org/wiki/Guarded_suspension

Monitor

- **Klasa** zawierająca `mutex` i `condition_variable`
- Przykład: problem producenta i konsumenta - thread-safe queue
- https://en.wikipedia.org/wiki/Monitor_(synchronization)

- Wzorzec Monitor jest w C++ szczególnym przypadkiem wzorca Guarded suspension
- Różnica - Guarded suspension nie musi mieć mutexu i zmiennej warunku w tej samej klasie

Read write lock

- Blokada posiadająca tryb do zapisu (wyłączny) oraz do odczytu (współdzielony)
- Implementowane przez `std::shared_mutex` (C++17)
- https://en.wikipedia.org/wiki/Readers–writer_lock
- https://en.cppreference.com/w/cpp/thread/shared_mutex

Scheduler

- Mechanizm kolejkujący zadania / wątki / zasoby
- Systemy operacyjne posiadają Scheduler, który sam zarządza tym, który wątek i przez jaki czas będzie się wykonywał
- Scheduler **powinien** zapobiegać zagłodzeniom
- Znane metody kolejkowania:
  - FIFO (`std::queue`)
  - LIFO (`std::stack`)
  - Priority based (`std::priority_queue`) – różne odmiany, np. Shortest Job First
  - Round Robin (algorytm karuzelowy) – każdy wątek dostaje taki sam czas i jest po nim wywłaszczany
- Im więcej wywłaszczeń tym więcej czasochłonnego przełączania kontekstów (przeładowanie rejestrów i pamięci cache)
- W C++ mechanizm do ręcznej implementacji, np. przy współpracy ze wzorcem Thread Pool (pula wątków)
- https://en.wikipedia.org/wiki/Scheduling_(computing)

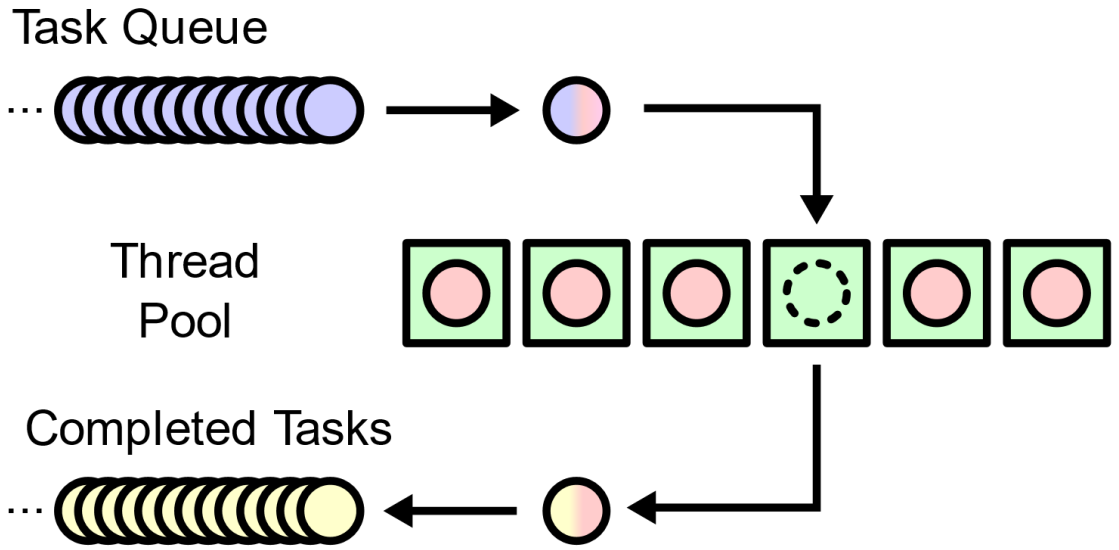
Double checked locking

```
static std::atomic<Singleton*> Singleton::m_instance = nullptr;
static std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_acquire);
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr) {
            tmp = new Singleton;
            m_instance.store(tmp, std::memory_order_release);
        }
    }
    return tmp;
}

Singleton& instance() {  // from C++ 11 the best Solution :)
    static Singleton s;
    return s;
}
```

- Od C++11 rzadko stosowany, z powodu gwarancji bezpieczeństwa wielowątkowego statycznej inicjalizacji
- Optymalizacja pozwalająca uniknąć kosztownego oczekiwania na zablokowanie mutexu, aby sprawdzić warunek
- https://en.wikipedia.org/wiki/Double-checked_locking

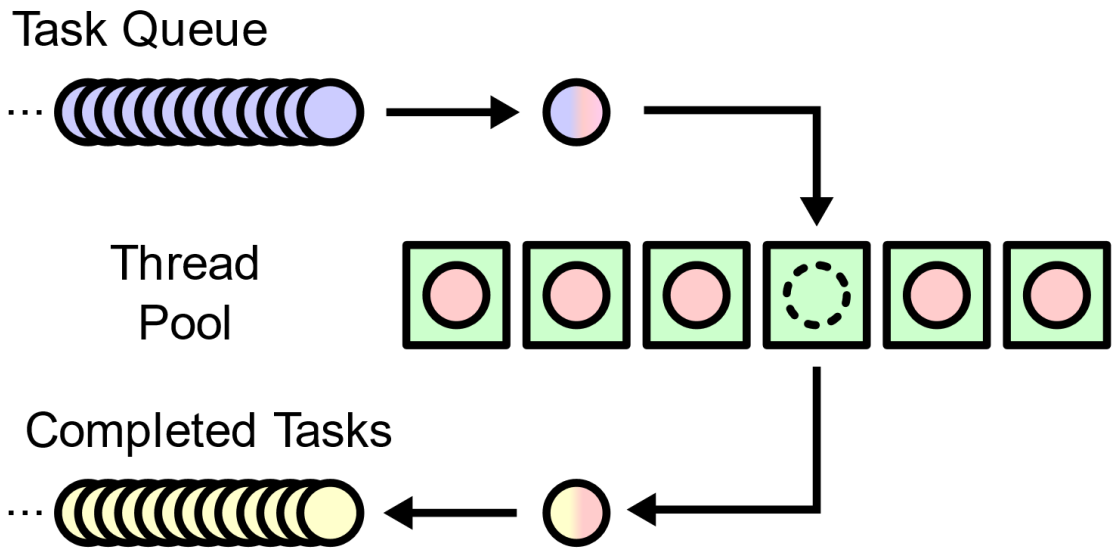
Thread pool (pula wątków)

- Składowe:
  - kolejka zadań do wykonania
  - kolekcja wątków
  - (opcjonalnie) kolejka wykonanych zadań/wyników
- Założenia:
  - zadania to funkcje o tej samej sygnaturze
  - nie wiemy, który wątek będzie przetwarzał które zadanie
- Działanie:
  - wszystkie wątki są uruchamiane i kończone razem
  - wątki działają w nieskończonych pętlach, dopóki nie zniszczymy całej puli
  - wątki pobierają zadania z kolejki
  - wątki wykonują zadania
  - jeśli żadne zadania nie są dostępne wątki czekają
  - (opcjonalnie) po wykonaniu zadania wątki odkładają wykonane zadanie lub rezultat do kolejki wyjściowej



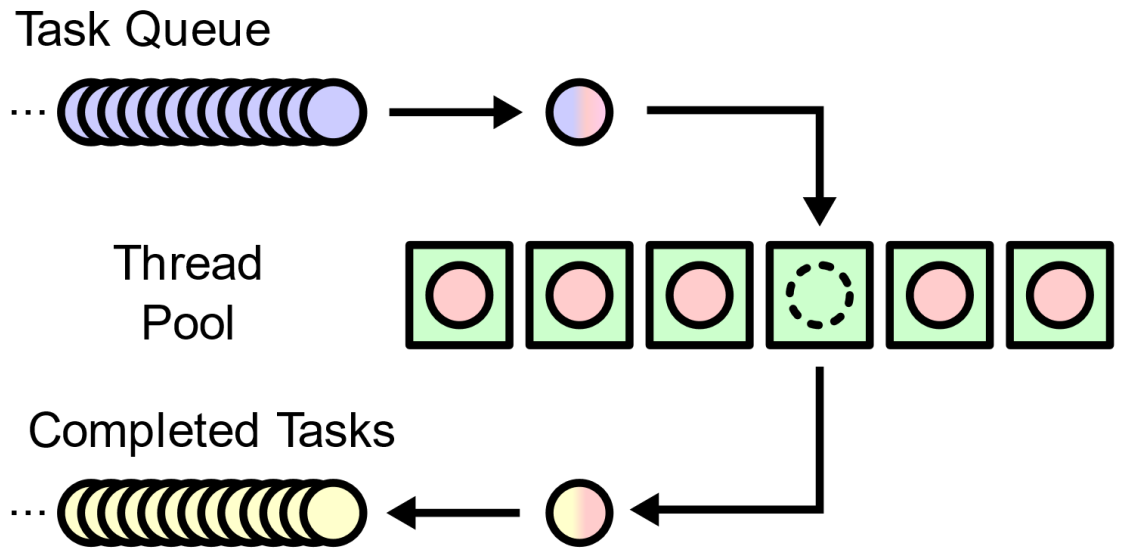
Thread pool (pula wątków) - zalety

- Dlaczego nie startować każdego zadania asynchronicznie w osobnych wątkach?
- Jeśli zadania są małe i jest ich dużo to tworzenie i niszczenie wątków zajmuje proporcjonalnie długi okres czasu
- Liczba wątków w puli jest z góry określona na podstawie możliwości maszyny, na której program jest uruchomiony
- Zapobiega to zjawisku *oversubscription* (więcej wątków niż rdzeni), które może mieć negatywny wpływ na wydajność w związku z wywłaszczaniem i przełączaniem kontekstów



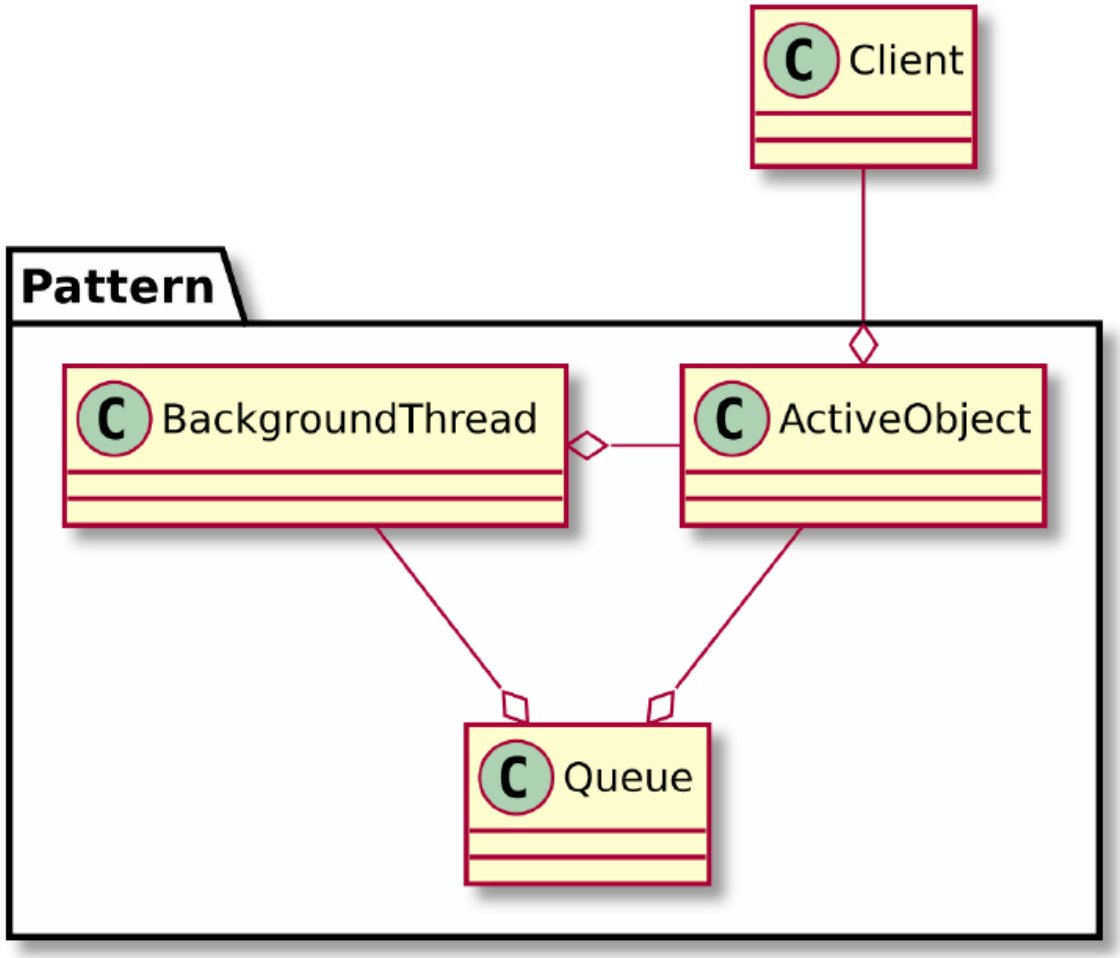
Thread pool (pula wątków) – zadanie w grupach

- Zaimplementuj pulę wątków, która będzie obliczać dowolne operacje na wektorze wejściowym i zapisywać wynik w wektorze wyjściowym (struktura `Task`)
- Jako kolejkę zadań wykorzystaj `ThreadSafeQueue<Task>`
- Pula ma przyjąć w konstruktorze liczbę wątków do uruchomienia.
- Do zwrócenia wyników użyj promise + future ze struktury `Task`

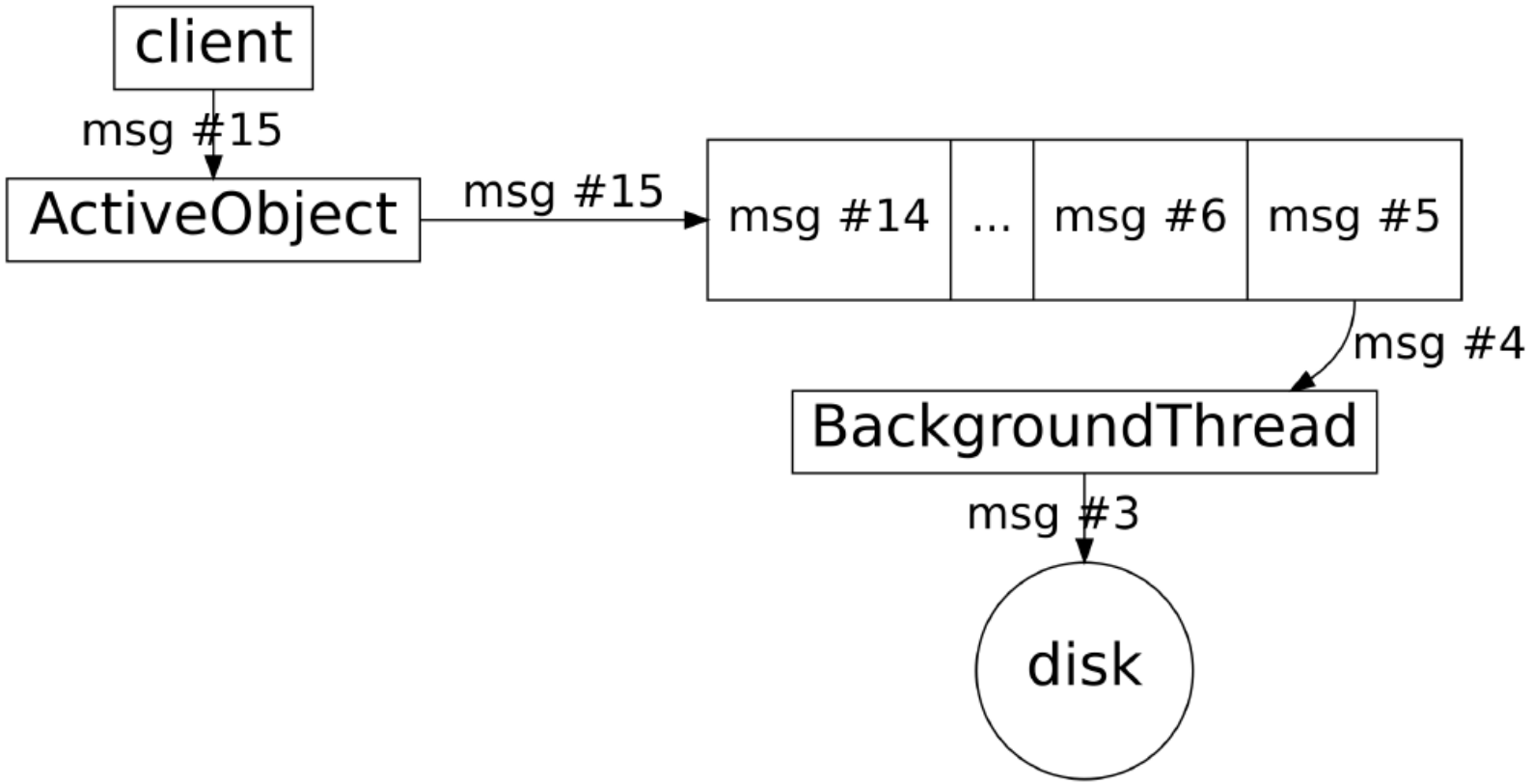


Active Object

- Współbieżne wykonywanie metod obiektu
- Oddzielenie wykonywanie metody od jej wywołania – metoda jest wykonywana asynchronicznie w oddzielnym wątku
- Upraszcza implementację synchronizowanego dostępu do obiektów
- Umożliwia użycie Schedulera, który odpowiednio kolejkuje wywołania metod
- Może być zaimplementowane z użyciem Thread Pool
- https://pl.wikipedia.org/wiki/Active_object



Active Object



Reactor / Proactor

- Służą do rozpropagowania zdarzenia ze źródła lub wielu źródeł do innego miejsca, w którym żądanie będzie odpowiednio obsłużone, zazwyczaj w innym wątku.
- Podobne do wzorca Observer (są to jego rozszerzenia).
- Reactor jest jednowątkowy / synchroniczny.
- Proactor jest wielowątkowy / asynchroniczny.
- Przykład: dowolna architektura serwer-klient.
- Często współpracują z Thread Pool wrzucając zadania do kolejki.
- https://en.wikipedia.org/wiki/Proactor_pattern
- https://en.wikipedia.org/wiki/Reactor_pattern

Dodatkowe zadania domowe – współpracujące wzorce

- Zmień nasza implementację wzorca Thread Pool, aby przyjmował dodatkowy parametr, mówiący o tym, jakiego Schedulera będzie używał (FIFO, LIFO, Priority Queue)
- Zaimplementuj wzorzec Reactor, którego zadaniem będzie wrzucanie zadań na pulę wątków ThreadPool co określony czas
- Napisz testy jednostkowe

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school