Podstawy programownia (w języku C++)

Wątki

Marek Marecki

Polsko-Japońska Akademia Technik Komputerowych

4 października 2021

Overview

Po co wątki w C++?

Wątki w C++ są narzędziem, które można wykorzystać do kilku celów. Nie znaczy to, że są najlepszym możliwym rozwiązaniem! Są jedynie podstawą, która umożliwia stworzenie i wykorzystanie bardziej eleganckich mechanizmów.

Wątki są również sposobem na wprowadzenie do programów **współbieżności** (ang. *concurrency*) i **równoległości** (ang. *parallelism*).

Współbieżność (ang. concurrency)

Wykonywanie przez program kilku zadań naraz (ale niekoniecznie *jednocześnie*).

Iluzja wykonywania przez program kilku zadań naraz może być zapewniona, dzięki szybkiemu przełączaniu się między częściowo wykonanymi zadaniami.

Przykład: program kopiujący k plików. Program może skopiować plik n w całości, po czym przejść do kopiowania pliku $n + 1$ i działać sekwencyjnie. Może też kopiować fragmenty plików od n do $n + k$ i działać współbieżnie (dzięki temu małe pliki mogą być skopiowane w całości szybciej).

Równoległość (ang. parallelism)

Wykonywanie przez program kilku zadań *jednocześnie* (w tym samym kwancie czasu).

Równoległość możliwa jest do osiągnięcia dzięki wykorzystaniu w programie wielu procesorów (na tym samym fizycznym komputerze, lub w systemie rozproszonym).

Przykład: program kopiujący k plików. Program może podzielić listę plików do skopiowania na n procesorów, i każdemu przekazać do obsłużenia k/n plików.

Współbieżność *vs* równoległość

Współbieżność *nie zawsze* oznacza równoległość.
Równoległość *zawsze* oznacza współbieżność.

Współbieżność jest sposobem na modelowanie zachowania i podział zadań wewnątrz programu. Równoległość jest sposobem na pełniejsze wykorzystanie zasobów systemu.

Overview

std::thread

Wątek czyli std::thread

Wątki tworzy się za pomocą typu `std::thread` z nagłówka `<thread>`.

Utworzenie wątku oznacza rozpoczęcie współbieżnego wykonywania funkcji, która została przekazana do konstruktora typu `std::thread`

```
auto a_thread = std::thread{
      some_function
    , 42
    , "Hello, World!"
};
```

„Dołączanie" wątków

std::thread

Za pomocą funkcji składowej `std::thread::join` można „dołączyć" wątek reprezentowany jakąś zmienną do wątku, z którego ta funkcja jest wywołana.

```
auto errand_boy = std::thread{run_some_errands, std::move(errands)};
errand_boy.join();  // wait until the errand boy returns...
```

Wątek wywołujący funkcję `std::thread::join` jest zablokowany do momentu aż wątek „dołączany" nie zakończy działania.

„Odcinanie” wątków

`std::thread`

Za pomocą funkcji składowej `std::thread::detach` można „odciąć” wątek i uniemożliwić jego dołączenie do innego wątku. Pozwala to odciętemu wątkowi działać nawet jeśli żaden inny wątek nie posiada jego „uchwytu”[1]:

```
auto fire_and_forget = std::thread{launch_missile, std::move(target)};
fire_and_forget.detach();  // too late to change our mind now...
```

[1] *thread handle*

Zadanie: drukowanie napisów

std::thread

Napisz program, który uruchomi 42 wątki drukujące napis „`Hello, X!`”.
`X` musi być różne dla każdego wątku (mogą to być różne imiona, różne liczby, itp.).
Wykorzystaj funkcję `std::thread::detach`.

Kod w pliku `src/s05-print-thread.cpp`

Zadanie: drukowanie grup napisów

`std::thread`

Napisz program, który uruchomi 42 wątki drukujące napis „`Hello, X!`”.
`X` musi być różne dla każdego wątku (mogą to być różne imiona, różne liczby, itp.).
Wątki mają być uruchamiane grupami po 6 naraz. Wykorzystaj funkcję `std::thread::join` do czekania na zakończenie wątku.

Kod w pliku `src/s05-print-thread-group.cpp`

Sekcje krytyczne

Sekcja krytyczna to fragment kodu, który może być wykonywany przez jeden wątek naraz. Takie fragmenty kodu powinny być zabezpieczone przed współbieżnym wykonaniem.

Jak rozpoznać, że coś jest sekcją krytyczną?
Sekcja krytyczna dotyka (manipuluje, odczytuje, itd.) danych, które widoczne są z potencjalnie wielu wątków.

Zabezpieczenie sekcji krytycznych

Sekcje krytyczne

Najprościej jest wykorzystać typy `std::mutex`[2] do zabezpieczenia danych, oraz `std::lock_guard` lub `std::unique_lock` do zablokowania wątku w oczekiwaniu na dostęp do sekcji krytycznej.

Zabezpieczenie danych:

```
auto foo = some_type{};
std::mutex foo_mtx;  // mutex for foo
```

Oczekiwanie na dostęp do sekcji krytycznej:

```
std::lock_guard<std::mutex> lck { foo_mtx };
// or...
std::unique_lock<std::mutex> lck { foo_mtx };
```

[2] *mutex* od *mutual exclusion*

Zadanie: lista zadań

`std::thread`

Napisz program, w którym uruchomisz cztery wątki pobierające ze wspólnej[3] kolejki `std::queue` liczby do wydrukowania. Pobieranie liczb powinno być sekcją krytyczną.
Każdemu z wątków przydziel ID (przekazywane jako argument podczas tworzenia wątku). Drukowane linie powinny mieć następujący format:

```
from thread THREAD-ID: NUMBER-TO-PRINT
```

Niech wątki zakończą działanie kiedy wykryją, że wektor z liczbami do wydrukowania jest pusty.

Kod w pliku `src/s05-job-queue.cpp`

[3] potrzebna będzie tu referencja lub wskaźnik

Overview

Zadanie: śpiące wątki

std::thread

Uruchom 4 wątki pobierające rzeczy do wydrukowania ze wspólnej kolejki (jak w poprzednim zadaniu). Niech wątki kończą pracę kiedy dostaną pusty `std::string` i drukują `thread X exiting` kiedy kończą pracę. Kiedy wątek wykryje, że kolejka jest pusta niech zwolni mutex i pójdzie spać na losową wartość milisekund między 10, a 100.

W funkcji `main()` odczytuj napisy ze standardowego strumienia wejścia i odczytane napisy odkładaj na kolejkę, z której czytają wątki drukujące. Zakończ pętlę po otrzymaniu 4 pustych napisów.

Kod w pliku `src/s05-sleeping-threads.cpp`

Oczekiwanie na, i sygnalizacja warunku

`std::condition_variable`

Kod taki jak w zadaniu ze śpiącymi wątkami jest nieefektywny, mało czytelny, i - powiedzmy to jasno - kiepski. Jeśli *oczekujemy* na wystąpienie jakiegoś warunku (np. nadejście zadania) to dużo lepszy jest kod, który realizuje to w sposób jawny.

Typ `std::condition_variable` z nagłówka `<condition_variable>` pozwala to zrealizować w prosty sposób.

Przygotowanie

`std::condition_variable`

Żeby móc oczekiwać na warunek potrzebujemy następujących rzeczy:

1. zmiennej typu `std::condition_variable`, której będziemy używać do oczekiwania i sygnalizacji warunku
2. zmiennej typu `std::mutex`, na której będziemy blokować nasz condition variable
3. dodatkowej zmiennej przechowującej dane, które powinniśmy sprawdzić po zajściu warunku (np. kolejki rzeczy do wydrukowania)

Przygotowanie (c.d.)

`std::condition_variable`

```
// this is our communication channel
std::queue<std::string> items_to_print;

// the mutex protects it from concurrent access
std::mutex mtx;

// the condition_variable is used to signal
// arrival of new data
std::condition_variable cv;
```

Oczekiwanie na zajście warunku

`std::condition_variable`

Oczekiwanie na zajście warunku jest realizowane funkcjami składowymi `std::condition_variable::wait` (czekanie w nieskończoność) i `std::condition_variable::wait_for` (czekanie przez określony czas).

Obie funkcje żądają przekazania im jako argumentu wartości typu `std::unique_lock` zablokowanej na muteksie, który chroni dane będące kanałem komunikacji:

```
// lock the mutex
std::unique_lock<std::mutex> lck { mtx };
```

Oczekiwanie na zajście warunku (c.d.)

`std::condition_variable`

Jeśli jesteśmy w stanie pozwolić sobie na czekanie w nieskończoność możemy użyć prostego `wait`:

```
// unlock the mutex, block calling thread, and
// wait for a notification (ad inifinitum)
cv.wait(lck);
```

Jeśli jednak w określonym czasie musimy podjąć jakieś działanie niezależnie od tego czy otrzymaliśmy komunikat czy nie, należy użyć `wait_for`:

```
// unlock the mutex, block calling thread, and
// wait for a notification for about 1 second
cv.wait(lck, std::chrono::seconds{1});
```

Sygnalizacja warunku

std::condition_variable

```
// lock the mutex and modify communication channel...
{
    std::lock_guard<std::mutex> lck { mtx };
    items_to_print.push("Hello, World!");
}

// notify just one of the waiting threads...
cv.notify_one();

// ...or notify all of them
cv.notify_all();
```

Zadanie: śpiące wątki 2

std::thread

Uruchom 4 wątki pobierające rzeczy do wydrukowania ze wspólnej kolejki (jak w poprzednim zadaniu). Niech wątki kończą pracę kiedy dostaną pusty `std::string` i drukują `thread X exiting` kiedy kończą pracę.
Niech wątki oczekują na rzeczy do wydrukowania z wykorzystaniem funkcji `std::condition_variable::wait`.

W funkcji `main()` odczytuj napisy ze standardowego strumienia wejścia i odczytane napisy odkładaj na kolejkę, z której czytają wątki drukujące. Zakończ pętlę po otrzymaniu 4 pustych napisów.

Kod w pliku `src/s05-sleeping-threads-2.cpp`

Zadanie: śpiące wątki 3

std::thread

Tak jak w „śpiące wątki 2”, ale z użyciem typu `itp::channel` z repozytorium szablonowego.

Kod w pliku `src/s05-sleeping-threads-3.cpp`

Zadanie: ping-pong

`std::thread`

Uruchom dwa wątki - jeden z identyfikatorem „ping”, drugi „pong”. Niech oba wątki wymieniają się liczbą całkowitą „odbijając” ją między sobą.
Wątek powinien relizować w pętli następującą logikę:

1. czekać na komunikat o otrzymaniu liczby
2. wydrukować liczbę poprzedzoną swoim identyfikatorem (np. `ping 7`)
3. zwiększyć liczbę o losową wartość między 1, a 42
4. poinformować drugi wątek o tym, że teraz jego kolej
5. jeśli liczba była większa niż 1024 zakończyć działanie

Pierwszą liczbę wyślij do wątku „ping” z funkcji `main()`.

Kod w pliku `src/s05-ping-pong.cpp`

Overview

Podsumowanie

Student powinien umieć:

1. wykorzystać wątki do wprowadzenie współbieżności i równoległości w programie
2. wytłumaczyć do czego służą typy `std::mutex`, `std::unique_lock`, i `std::lock_guard`
3. wytłumaczyć czym jest sekcja krytyczna

Zadania

Podsumowanie

Zadania znajdują się na slajdach 11, 12, 15, 17, 24, 25, 26.