Współbieżność w C++

Kamil Kowalski

Model równoległości w C++

- Zapewnia równoległość ale nie współbieżność
- Opiera się na `std::thread`
- Korzysta z systemowej implementacji

std::thread

- Reprezentuje pojedyńczy wątek wykonania
- Argumentem konstruktora jest objekt funkcyjny który jest od razu uruchamiany

std::thread - funkcje

- `.join()` - czeka na zakończenie wątku
- `.detach()` - pozwala wątkowi pracować nawet po zniszczeniu oryginalnego std::thread
- `std::thread::hardware_concurrency()` - informuje o liczbie fizycznych wątków w systemie

std::async

- Funkcja przyjmuje obiekt funkcyjny i parametry do niego
- Pozwala na asynchroniczne wykonywanie funkcji
- Ma dwa tryby działania:
  - `std::launch::async`
  - `std::launch::deffered`
- Zwraca objekt typu `std::future<T>`

std::async - przykład

```
future<string> greeting = async([](int a,bool d) -> string {return “hello”;},10,false);
```

Funkcja ta bardzo upraszcza korzystanie z wielowątkowości, jednak jest też najbardziej ograniczona ze wszystkich.

std::future

- Reprezentuje wynik jakiejś operacji asynchronicznej
- Pozwala sprawdzić czy wynik jest gotowy, poczekać jakiś czas na wynik, lub całkowicie zablokować dalsze wykonanie podczas oczekiwania.
- Zwraca wartość lub wyjątek

std::shared_future

- Jest rozszerzeniem `std::future`, pozwala na oczekiwanie na jeden `std::future` przez kilka wątków
- Jest kopiowalne

std::future – przykład użycia

- `future.get()` - blokuje bieżący wątek w oczekiwaniu na wynik
- `future.wait_for(std::chrono::duration<>)` - czeka na wynik przez określony czas
- `future.wait_until(std::chrono::time_point<>)` - czeka na wynik do określonego momentu

std::promise

- Najbardziej podstawowy sposób realizacji wielowątkowości
- Działa jak ‘kanał’ którym wątek wykonujący obietnicę zwraca wynik do `std::future`
- Pozwala odseparować moment uruchomienia wątku od oczekiwania na jego wynik
- Zapewnia dużą elastyczność – `std::future` może być zwrócona także w trakcie działania funkcji

std::promise – przykład użycia

```
std::promise<int> promise;
std::future<int> result = promise.get_future();
std::thread thread([](std::promise<int> promise){promise.set_value(10);
                                    std::cout<<"exiting thread\n";},
                                    std::move(promise));
std::cout<<result.get();
```

std::packaged_task

- Podobny w działaniu do `std::promise`
- Automatyzuje zwracanie wartości lub wyjątków z funkcji
- Zapewnia mniejszą kontrolę niż `std::promise`, ale jest wygodniejszy w użyciu

std::packaged_task – przykład użycia

```
std::packaged_task<int> task([](int unnecessary){return 10;});
std::future<int> value = task.get_future();
std::thread thread(std::move(task),20);
std::cout<<value.get()<<std::endl;
thread.join();
```

Źródła:

https://en.cppreference.com

https://isocpp.org/wiki/faq/cpp11-library-concurrency