C++17 i STL

Programowanie współbieżne

Synchronizacja wątków

- Korzystanie z wielu wątków niemal zawsze wiąże się z równoległym dostępem do danych – rzadko się zdarza, aby wiele wątków działało całkowicie niezależnie od siebie.
- Wątki mogą dostarczać danych albo modyfikować dane, które są przetwarzane przez inne wątki – obliczenia te nie powinny jednocześnie pracować na tych samych danych.
- Techniki synchronizacji pracy wątków:
  - muteksy i blokady, włącznie z funkcją call_once();
  - zmienne warunkowe;
  - zmienne atomowe.

Synchronizacja wątków

- **Jedynym bezpiecznym sposobem współbieżnego dostępu do tych samych danych przez wiele wątków bez stosowania synchronizacji jest sytuacja, kiedy wszystkie wątki jedynie czytają dane.**
- Jednak jeśli więcej wątków współbieżnie korzysta z tych samych danych i co najmniej jeden z wątków wprowadza modyfikacje, możemy łatwo wpaść w kłopoty, jeśli nie zadbamy o synchronizację tego dostępu.
- **Wyścig o dane** zachodzi wtedy, gdy dwie ścieżki obliczeniowe w różnych wątkach kolidują ze sobą (współdzielą dane), z których co najmniej jedna nie jest atomowa i żadna nie jest wykonywana wcześniej niż druga – wyścig o dane zawsze kończy się niezdefiniowanym zachowaniem/stanem.

Problem jednoczesnego dostępu do danych

- **Niezsynchronizowany dostęp do danych**: gdy dwa wątki działające równolegle czytają i zapisują te same dane, pozostaje otwartą kwestią to, która instrukcja będzie wykonana w pierwszej kolejności.
- **Dane w trakcie zapisywania**: gdy jeden wątek czyta dane, które inny wątek modyfikuje, wątek czytający może odczytywać dane nawet w trakcie zapisywania przez drugi wątek – zatem odczytywana wartość nie jest ani nowa, ani stara.
- **Zmieniona kolejność instrukcji**: można zmienić kolejność instrukcji i operacji w taki sposób, że zachowanie każdego pojedynczego wątku będzie poprawne, ale po połączeniu wszystkich wątków zachowanie będzie różne od oczekiwanego.

Niezsynchronizowany dostęp do danych (1)

- Standardowe funkcje biblioteczne C++ zazwyczaj nie wspierają współbieżnego zapisywania lub odczytywania w przypadku realizowania zapisu do tej samej struktury danych.
- Współbieżny dostęp do różnych elementów tego samego kontenera jest możliwy (z wyjątkiem klasy `vector<bool>`) – tak więc różne wątki mogą jednocześnie czytać lub zapisywać różne elementy tego samego kontenera.
- Jednoczesny dostęp do strumienia łańcuchów znaków, strumienia pliku lub bufora strumienia skutkuje niezdefiniowanym zachowaniem.

Dane w trakcie zapisywania (2)

- Przykład:
  Załóżmy, że mamy następującą zmienną:
  ```
  long long x = 0;
  ```
  oraz jeden wątek zapisujący dane:
  ```
  x = -1;
  ```
  i drugi, który te dane odczytuje:
  ```
  std::cout << x;
  ```
  Jaki jest wynik działania programu?
- Wynikiem może być dowolna inna wartość, jeśli drugi wątek czyta zmienną x w czasie przypisywania wartości -1 przez pierwszy wątek.
- Standard C++ nie gwarantuje, że odczyt bądź zapis będzie atomowy nawet w przypadku podstawowych typów danych.

Zmieniona kolejność instrukcji (3)

- Przykład:
  Załóżmy, że mamy następujące zmienne:
  ```
  long data;
  bool ready = false;
  ```
  oraz w wątku dostarczającym dane znajdą się:
  ```
  data = 42;
  readyFlag = true;
  ```
  natomiast w wątku konsumującym dane:
  ```
  while (! readyFlag) { // do kiedy dane będą gotowe
  }
  foo(data);
  ```
- Uwaga, kompilator lub sprzęt mógł zmienić kolejność instrukcji, tak że w rezultacie sekwencja niekolidujących z pozoru instrukcji mogła zostać wykonana w odwrotnej kolejności.
- Zmiana kolejności instrukcji jest dozwolona ze względu na reguły języka C++, które wymagają jedynie, aby obserwowane zachowanie wewnątrz wątku wygenerowanego kodu było poprawne.

Mechanizmy pozwalające na rozwiązanie problemów

- Aby rozwiązać trzy główne problemy współbieżnego dostępu do danych, potrzebujemy następujących mechanizmów:
  - Niepodzielność: oznacza ona, że odczyt lub zapis do zmiennej lub wykonanie sekwencji instrukcji dzieje się w sposób wyłączny i bez żadnych przerw, dzięki czemu jeden wątek nie może czytać stanów pośrednich spowodowanych przez inny wątek.
  - Kolejność: potrzebujemy mechanizmów, które gwarantują kolejność wykonania określonych instrukcji bądź grup określonych instrukcji.

Mechanizmy pozwalające na rozwiązanie problemów

- Biblioteka standardowa C++ dostarcza różnych sposobów obsługi tych mechanizmów, dzięki czemu programy mogą korzystać z dodatkowych gwarancji dotyczących współbieżnego dostępu do danych:
  - Można skorzystać z **futur** i **promes**, które gwarantują zarówno niepodzielność, jak i kolejność.
  - Można skorzystać z **muteksów** i **blokad** w celu obsługi sekcji krytycznych lub stref chronionych, w których możemy zagwarantować wyłączny dostęp.
  - Można skorzystać ze **zmiennych warunkowych**, aby umożliwić jednemu wątkowi oczekiwanie na to, aż pewien predykat kontrolowany przez inny wątek stanie się prawdziwy.
  - Można skorzystać z **atomowych typów danych**, aby w ten sposób zapewnić niepodzielność każdego dostępu do zmiennej bądź obiektu.

Współbieżność

- Nowoczesne architektury systemów zazwyczaj umożliwiają uruchomienie wielu procesów i wielu wątków w obrębie każdego procesu jednocześnie.
- Wykorzystanie wielu wątków może przyczynić się do poprawy czasu realizacji zadania, zwłaszcza wtedy, gdy komputer dysponuje wieloma procesorami/rdzeniami.
- Obliczenia równoległe generują jednak problemy przy dostępie do tych samych zasobów w pamięci (operacje tworzenia, czytania, pisania i usuwania mogą odbywać się w nieoczekiwanej kolejności, co może prowadzić do nieoczekiwanych rezultatów).
- W C++11 wprowadzono klasę `thread` reprezentującą wątek w programie.

Współbieżność

- W C++11 wprowadzono obsługę programowania współbieżnego – biblioteka STL zapewnia wsparcie dla:
  - uruchamiania wielu wątków,
  - mechanizmy do synchronizacji wątków,
  - przekazywanie argumentów, zwracanie wartości i zgłaszanie wyjątków poza granicami wątków.
- Biblioteka STL zapewnia wsparcie dla współbieżności na wielu poziomach:
  - z jednej strony interfejs wysokiego poziomu umożliwia rozpoczęcie wątku, przekazanie do niego argumentów i obsługę wyników i wyjątków;
  - z drugiej strony, istnieją również własności niskiego poziomu, takie jak muteksy lub atomowe typy danych obsługujące swobodny dostęp do pamięci.

Interfejs wysokiego poziomu

- W pliku nagłówkowym `<future>` zdefiniowane są podstawowe narzędzia do wysokopoziomowej pracy z wątkami.
- Funkcja `async()` umożliwia uruchomienie pewnej funkcjonalności w osobnym wątku.
- Klasa `future<>` pozwala czekać na zakończenie wątku i zapewnia dostęp do jego wyników – zwróconej wartości lub wyjątku.

Interfejs wysokiego poziomu – futury

- Szablon klasy `future<>` udostępnia mechanizm dostępu do wyniku operacji asynchronicznych:
  - operacja asynchroniczna utworzona za pomocą funkcji `async()` lub obiektu `promise()` może dostarczyć obiekt futury do kreatora tej operacji asynchronicznej;
  - kreator operacji asynchronicznej może następnie użyć różnych metod do zapytania o wynik lub oczekiwania na wynik z futury (te metody mogą być blokowane, jeśli operacja asynchroniczna nie dostarczyła jeszcze żadnej wartości);
  - gdy operacja asynchroniczna jest gotowa na wysłanie wyniku do kreatora, może to zrobić poprzez modyfikację stanu współdzielonego, używając na przykład `promise<>::set_value()`, który jest powiązany z futurą kreatora.
- Obiekt futury odwołuje się do stanu współdzielonego, który nie jest współdzielony z żadnymi innymi asynchronicznymi obiektami zwracanymi (w przeciwieństwie do `shared_future<>`).

Interfejs wysokiego poziomu – wykonanie asynchroniczne

- Szablon funkcji `async()` uruchamia zadaną funkcję asynchronicznie (potencjalnie w oddzielnym wątku, który może być częścią puli wątków) i zwraca futurę, która ostatecznie przechowa wynik wywołania tej funkcji.
- Funkcja `async()` jest przeciążona:
  - template< class Function, class... Args >
future<std::invoke_result_t<decay_t<Function>, decay_t<Args>...>>
async(Function&& f, Args&&... args);
  - template< class Function, class... Args >
std::future<invoke_result_t<decay_t<Function>, decay_t<Args>...>>
async(launch policy, Function&& f, Args&&... args);

Interfejs wysokiego poziomu – wykonanie asynchroniczne

- Strategię uruchamiania obliczeń w wątku możemy komponować za pomocą dwóch strategii:
  - `launch::async` – włącza obliczenia asynchroniczne,
  - `launch::deferred` – włącza obliczenia leniwe.
- Jeśli przy wywołaniu funkcji async() strategia nie zostanie określona, to funkcja zachowuje się tak, jakby obydwie flagi zostały włączone.
- Jeśli flaga `launch::async` jest ustawiona, to funkcja `async()` wykonuje funkcję zdefiniowaną w obiekcie `Function` w nowym wątku.
- Jeśli flaga `launch::deferred` jest ustawiona, to funkcja `async()` wykonuje funkcję zdefiniowaną w obiekcie `Function` w bieżącym wątku w odroczonym terminie.
- Jeśli obie flagi `launch::async` i `launch::deferred` są ustawione, to od implementacji zależy, czy zadanie wykona się asynchroniczne czy z opóźnieniem.

Interfejs wysokiego poziomu – przykład

- Przykład 1 [C++ biblioteka standardowa. Josuttis]:

```
int doSomething(char c) {
    default_random_engine dre(c);
    uniform_int_distribution<int> id(10,1000);
    for (int i=0; i<10; ++i) {
        this_thread::sleep_for(chrono::milliseconds(id(dre)));
        cout.put(c).flush();
    }
    return c;
}
int func1() {
    return doSomething('–');
}
int func2() {
    return doSomething('+');
}
```

Interfejs wysokiego poziomu – przykład

- Przykład 1 [C++ biblioteka standardowa. Josuttis]:

```
int main() {
    cout << "uruchomienie funkcji func1() w tle,"
        << " a funkcji func2() na pierwszym planie:" << endl;
    future<int> result1(async(func1));
    int result2 = func2();
    int result = result1.get() + result2;
    cout << "\nwynik sumy func1()+func2(): " << result << endl;
}
```

Interfejs wysokiego poziomu – przykład

- Przykład 1 [C++ biblioteka standardowa. Josuttis]:
  ```
  future<int> result1(async(func1));
  ```
  Jeśli funkcja `async()` nie może uruchomić przekazanej funkcjonalności natychmiast, odracza wywołanie do chwili jawnego zażądania wyniku przekazanej funkcjonalności za pomocą funkcji `get()` lub `wait()`.
- Przykład 1[C++ biblioteka standardowa. Josuttis]:
  ```
  future<int> result1(launch::async, async(func1));
  ```
  Jeśli należy wykonać przekazaną do funkcji `async()` funkcjonalność natychmiast, to trzeba określić strategię wykonania na `launch::async`.

Interfejs wysokiego poziomu – przykład

- Przykład 2 [C++ biblioteka standardowa. Josuttis]:
```
auto f1 = async(launch::deferred, task1);
auto f2 = async(launch::deferred, task2);
...
auto val = thisOrThatIsTheCase() ? f1.get() : f2.get();
```
- W tym przypadku mamy gwarancję, że funkcje `func1()` oraz `func2()` nie zostaną wywołane bez wywołania `get()` lub `wait()` – ta strategia umożliwia leniwe wartościowanie.

Futury – obsługa wyjątków

- Zgłoszenie wyjątku spowoduje zakończenie pracy wątku, o ile wyjątek ten nie zostanie złapany w samej funkcji realizującej zadanie.
- Wywołanie funkcji `get()` dla futur obsługuje również wyjątki. Jeśli nastąpi wywołanie `get()`, a operacja w tle zostanie przerwana przez wyjątek, który nie został obsłużony wewnątrz wątku, to ten wyjątek będzie propagowany dalej.
- W celu obsługi wyjątków operacji wykonywanych w tle należy postępować z funkcją `get()` w taki sposób, w jaki postąpilibyśmy, gdyby operacja była uruchomiona synchronicznie.
- Przykład:

```
auto f = async(task);
...
try { f1.get(); }
catch (const exception &e) { … }
```

Futury – oczekiwanie i odpytywanie

- Funkcję `get()` w odniesieniu do obiektu `future<>` można wywołać tylko raz.
- Po wywołaniu funkcji `get()` futura jest nieważna.
- Ważność futury można sprawdzić jedynie poprzez wywołanie funkcji `valid()`.

Futury – oczekiwanie i odpytywanie

- Wywołanie funkcji `wait()` wymusza uruchomienie wątku reprezentowanego przez futurę i oczekiwanie na zakończenie operacji w tle.
- Funkcje `wait_for()` i `wait_until()` nie wymuszają uruchomienia wątku, jeśli nie został on uruchomiony wcześniej.
- Zarówno funkcja `wait_for()` jak i `wait_until()` zwracają status futury:
  - `future_status::deferred` – jeśli funkcja `async()` odroczyła operację i żadne z wywołań `wait()` lub `get()` jeszcze nie wymusiło jej startu;
  - `future_status::timeout` – jeśli operacja została uruchomiona asynchronicznie, ale jeszcze się nie zakończyła;
  - `future_status::ready` – jeśli operacja zakończyła się.

Futury – oczekiwanie i odpytywanie

- Funkcja `wait_for()`, do której przekazujemy czas oczekiwania, pozwala czekać na asynchroniczne uruchomienie operacji przez ograniczony czas.
  Przykład:
  ```
  future<...> f(async(func));
  ...
  f.wait_for(chrono::seconds(10));
  ```
- Funkcja `wait_until()` pozwala czekać do określonego punktu w czasie.
  Przykład:
  ```
  future<...> f(async(func));
  ...
  f.wait_until(system_clock::now() + chrono::minutes(1));
  ```

Futury – przekazywanie argumentów

- Do przekazania argumentów do zadania można wykorzystać lambdy.
- Przykład 1:
```
auto f = async([]{ doSomething('.'); });
```
- Przykład 2:
```
char c = '@';
auto f = async([=]{doSomething(c); });
```

Futury – przekazywanie argumentów

- Funkcja `async()` dostarcza standardowego interfejsu obiektów wywoływalnych – jeśli przekażemy wskaźnik do funkcji zadaniowej jako pierwszy argument, argumenty samego zadania możemy przekazać jako kolejne argumenty funkcji `async()`.
- Przykład 3 (przekazanie argumentu przez wartość):
```
char c = '#';
auto f = async(doSomething, c);
```
- Przykład 4 (przekazanie argumentu przez referencję):
```
char c = '#';
auto f = async(doSomething, ref(c));
```

Futury współdzielone

- Obiekt klasy `future<>` zapewnia możliwość przetwarzania wyniku futury dla współbieżnych obliczeń – jednak ten wynik możemy przetwarzać tylko raz, ponieważ drugie i kolejne wywołanie funkcji `get()` spowoduje niezdefiniowane zachowanie.
- Czasem jednak potrzebne jest przetwarzanie wyniku współbieżnych obliczeń w wielu różnych wątkach – do tego celu należy użyć obiektu klasy `shared_future<>`, wtedy możliwe jest kilkukrotne wywołanie funkcji `get()`.
- Funkcja `get()` w klasie `future<>` ma sygnaturę:
  ```
  T future<T>::get();
  ```
  a w klasie `shared_future<>` jej sygnatura wygląda następująco:
  ```
  const T& shared_future<T>::get();
  ```