Anthony Williams

Odkryj wszystkie tajemnice wielowątkowych aplikacji!

Język C++

i przetwarzanie współbieżne w akcji

Wydanie II

Tytuł oryginału: C++ Concurrency in Action, 2nd Edition

Tłumaczenie: Robert Górczyński
na podstawie: „Język C++ i przetwarzanie współbieżne w akcji"
w przekładzie Mikołaja Szczepaniaka

ISBN: 978-83-283-4448-8

Original edition copyright © 2019 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2020 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jcppw2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Słowo **wstępne** **11**
Podziękowania ***13***
O tej książce ***15***
O autorze ***19***

Rozdział 1. Witaj, świecie współbieżności w C++! **21**

1.1. Czym jest współbieżność? **22**
1.1.1. Współbieżność w systemach komputerowych 22
1.1.2. Modele współbieżności 25
1.1.3. Współbieżność kontra równoległość 27
1.2. Dlaczego warto stosować współbieżność? **27**
1.2.1. Stosowanie współbieżności do podziału zagadnień 27
1.2.2. Stosowanie współbieżności do podniesienia wydajności — równoległość zadań i danych 28
1.2.3. Kiedy nie należy stosować współbieżności 30
1.3. Współbieżność i wielowątkowość w języku C++ **31**
1.3.1. Historia przetwarzania wielowątkowego w języku C++ 31
1.3.2. Obsługa współbieżności w nowym standardzie 32
1.3.3. Większa obsługa współbieżności i równoległości w standardach C++14 i C++17 33
1.3.4. Efektywność biblioteki wątków języka C++ 33
1.3.5. Mechanizmy związane z poszczególnymi platformami 34
1.4. Do dzieła **35**
1.4.1. Witaj, świecie współbieżności! 35
1.5. Podsumowanie **36**

Rozdział 2. Zarządzanie wątkami **39**

2.1. Podstawowe zarządzanie wątkami **40**
2.1.1 Uruchamianie wątku 40
2.1.2. Oczekiwanie na zakończenie wątku 43
2.1.3. Oczekiwanie w razie wystąpienia wyjątku 44
2.1.4. Uruchamianie wątków w tle 46
2.2. Przekazywanie argumentów do funkcji wątku **47**
2.3. Przenoszenie własności wątku **50**
2.4. Wybór liczby wątków w czasie wykonywania **55**
2.5. Identyfikowanie wątków **57**
2.6. Podsumowanie **59**

Kup ksi k   Pole ksi k

Rozdział 3. Współdzielenie danych przez wątki 61

3.1. Problemy związane ze współdzieleniem danych przez wątki ... 62
3.1.1. Sytuacja wyścigu ... 64
3.1.2. Unikanie problematycznych sytuacji wyścigu ... 65
3.2. Ochrona współdzielonych danych za pomocą muteksów ... 66
3.2.1. Stosowanie muteksów w języku C++ ... 66
3.2.2. Projektowanie struktury kodu z myślą o ochronie współdzielonych danych ... 68
3.2.3. Wykrywanie sytuacji wyścigu związanych z interfejsami ... 70
3.2.4. Zakleszczenie: problem i rozwiązanie ... 77
3.2.5. Dodatkowe wskazówki dotyczące unikania zakleszczeń ... 80
3.2.6. Elastyczne blokowanie muteksów za pomocą szablonu std::unique_lock ... 87
3.2.7. Przenoszenie własności muteksu pomiędzy zasięgami ... 89
3.2.8. Dobór właściwej szczegółowości blokad ... 90
3.3. Alternatywne mechanizmy ochrony współdzielonych danych ... 93
3.3.1. Ochrona współdzielonych danych podczas inicjalizacji ... 93
3.3.2. Ochrona rzadko aktualizowanych struktur danych ... 97
3.3.3. Blokowanie rekurencyjne ... 99
3.4. Podsumowanie ... 100

Rozdział 4. Synchronizacja współbieżnych operacji 101

4.1. Oczekiwanie na zdarzenie lub inny warunek ... 102
4.1.1. Oczekiwanie na spełnienie warunku za pomocą zmiennych warunkowych ... 103
4.1.2. Budowa kolejki gwarantującej bezpieczne przetwarzanie wielowątkowe przy użyciu zmiennych warunkowych ... 106
4.2. Oczekiwanie na jednorazowe zdarzenia za pomocą przyszłości ... 111
4.2.1. Zwracanie wartości przez zadania wykonywane w tle ... 112
4.2.2. Wiązanie zadania z przyszłością ... 114
4.2.3. Obietnice (szablon std::promise) ... 117
4.2.4. Zapisywanie wyjątku na potrzeby przyszłości ... 119
4.2.5. Oczekiwanie na wiele wątków ... 121
4.3. Oczekiwanie z limitem czasowym ... 124
4.3.1. Zegary ... 124
4.3.2. Okresy ... 125
4.3.3. Punkty w czasie ... 127
4.3.4. Funkcje otrzymujące limity czasowe ... 129
4.4. Upraszczanie kodu za pomocą technik synchronizowania operacji ... 131
4.4.1. Programowanie funkcyjne przy użyciu przyszłości ... 131
4.4.2. Synchronizacja operacji za pomocą przesyłania komunikatów ... 136
4.4.3. Współbieżność w stylu kontynuacji dzięki użyciu Concurrency TS ... 141
4.4.4. Łączenie kontynuacji ze sobą ... 143
4.4.5. Oczekiwanie na więcej niż tylko jedną przyszłość ... 146
4.4.6. Oczekiwanie za pomocą when_any na pierwszą przyszłość w zbiorze ... 148
4.4.7. Zasuwy i bariery w Concurrency TS ... 151
4.4.8. Zasuwa typu podstawowego — std::experimental::latch ... 151
4.4.9. Podstawowa bariera — std::experimental::barrier ... 153
4.4.10. std::experimental::flex_barrier, czyli elastyczniejszy przyjaciel std::experimental:barrier ... 155
4.5. Podsumowanie ... 156

Rozdział 5. Model pamięci języka C++ i operacje na typach atomowych 157

5.1. Podstawowe elementy modelu pamięci ... 158
5.1.1. Obiekty i miejsca w pamięci ... 158
5.1.2. Obiekty, miejsca w pamięci i przetwarzanie współbieżne ... 159
5.1.3. Kolejność modyfikacji ... 161
5.2. Operacje i typy atomowe języka C++ ... 161
5.2.1. Standardowe typy atomowe ... 162
5.2.2. Operacje na typie std::atomic_flag ... 165
5.2.3. Operacje na typie std::atomic<bool> ... 167
5.2.4. Operacje na typie std::atomic<T*> — arytmetyka wskaźników ... 170
5.2.5. Operacje na standardowych atomowych typach całkowitoliczbowych ... 172
5.2.6. Główny szablon klasy std::atomic<> ... 172
5.2.7. Wolne funkcje dla operacji atomowych ... 174
5.3. Synchronizacja operacji i wymuszanie ich porządku ... 176
5.3.1. Relacja synchronizacji ... 178
5.3.2. Relacja poprzedzania ... 179
5.3.3. Porządkowanie pamięci na potrzeby operacji atomowych ... 181
5.3.4. Sekwencje zwalniania i relacja synchronizacji ... 201
5.3.5. Ogrodzenia ... 204
5.3.6. Porządkowanie operacji nieatomowych za pomocą operacji atomowych ... 206
5.3.7. Porządkowanie operacji nieatomowych ... 207
5.4. Podsumowanie ... 210

Rozdział 6. Projektowanie współbieżnych struktur danych przy użyciu blokad 211

6.1. Co oznacza projektowanie struktur danych pod kątem współbieżności? ... 212
6.1.1. Wskazówki dotyczące projektowania współbieżnych struktur danych ... 213
6.2. Projektowanie współbieżnych struktur danych przy użyciu blokad ... 214
6.2.1. Stos gwarantujący bezpieczeństwo przetwarzania wielowątkowego przy użyciu blokad ... 215
6.2.2. Kolejka gwarantująca bezpieczeństwo przetwarzania wielowątkowego przy użyciu blokad i zmiennych warunkowych ... 218
6.2.3. Kolejka gwarantująca bezpieczeństwo przetwarzania wielowątkowego przy użyciu szczegółowych blokad i zmiennych warunkowych ... 222
6.3. Projektowanie złożonych struktur danych przy użyciu blokad ... 235
6.3.1. Implementacja tablicy wyszukiwania gwarantującej bezpieczeństwo przetwarzania wielowątkowego przy użyciu blokad ... 235
6.3.2. Implementacja listy gwarantującej bezpieczeństwo przetwarzania wielowątkowego przy użyciu blokad ... 241
6.4. Podsumowanie ... 246

Rozdział 7. Projektowanie współbieżnych struktur danych bez blokad 247

7.1. Definicje i ich praktyczne znaczenie ... 248
7.1.1. Rodzaje nieblokujących struktur danych ... 248
7.1.2. Struktury danych bez blokad ... 249
7.1.3. Struktury danych bez oczekiwania ... 250
7.1.4. Zalety i wady struktur danych bez blokad ... 250

7.2. Przykłady struktur danych bez blokad ... 252
7.2.1. Implementacja stosu gwarantującego bezpieczeństwo przetwarzania wielowątkowego bez blokad ... 253
7.2.2. Eliminowanie niebezpiecznych wycieków — zarządzanie pamięcią w strukturach danych bez blokad ... 257
7.2.3. Wykrywanie węzłów, których nie można odzyskać, za pomocą wskaźników ryzyka ... 262
7.2.4. Wykrywanie używanych węzłów metodą zliczania referencji ... 271
7.2.5. Zmiana modelu pamięci używanego przez operacje na stosie bez blokad ... 277
7.2.6. Implementacja kolejki gwarantującej bezpieczeństwo przetwarzania wielowątkowego bez blokad ... 282
7.3. Wskazówki dotyczące pisania struktur danych bez blokad ... 295
7.3.1. Wskazówka: na etapie tworzenia prototypu należy stosować tryb std::memory_order_seq_cst ... 295
7.3.2. Wskazówka: należy używać schematu odzyskiwania pamięci bez blokad ... 296
7.3.3 Wskazówka: należy unikać problemu ABA ... 296
7.3.4. Wskazówka: należy identyfikować pętle aktywnego oczekiwania i wykorzystywać czas bezczynności na wspieranie innego wątku ... 297
7.4. Podsumowanie ... 298

Rozdział 8. Projektowanie współbieżnego kodu 299

8.1. Techniki dzielenia pracy pomiędzy wątki ... 300
8.1.1. Dzielenie danych pomiędzy wątki przed rozpoczęciem przetwarzania ... 301
8.1.2. Rekurencyjne dzielenie danych ... 302
8.1.3. Dzielenie pracy według typu zadania ... 307
8.2. Czynniki wpływające na wydajność współbieżnego kodu ... 310
8.2.1. Liczba procesorów ... 310
8.2.2. Współzawodnictwo o dane i ping-pong bufora ... 311
8.2.3. Fałszywe współdzielenie ... 314
8.2.4. Jak blisko należy rozmieścić dane? ... 315
8.2.5. Nadsubskrypcja i zbyt intensywne przełączanie zadań ... 316
8.3. Projektowanie struktur danych pod kątem wydajności przetwarzania wielowątkowego ... 317
8.3.1. Podział elementów tablicy na potrzeby złożonych operacji ... 317
8.3.2. Wzorce dostępu do danych w pozostałych strukturach ... 319
8.4. Dodatkowe aspekty projektowania współbieżnych struktur danych ... 321
8.4.1. Bezpieczeństwo wyjątków w algorytmach równoległych ... 321
8.4.2. Skalowalność i prawo Amdahla ... 329
8.4.3. Ukrywanie opóźnień za pomocą wielu wątków ... 330
8.4.4. Skracanie czasu reakcji za pomocą technik przetwarzania równoległego ... 332
8.5. Projektowanie współbieżnego kodu w praktyce ... 334
8.5.1. Równoległa implementacja funkcji std::for_each ... 334
8.5.2. Równoległa implementacja funkcji std::find ... 337
8.5.3. Równoległa implementacja funkcji std::partial_sum ... 343
8.6. Podsumowanie ... 353

Rozdział 9. Zaawansowane zarządzanie wątkami 355

9.1. Pule wątków ... 356
9.1.1. Najprostsza możliwa pula wątków ... 356
9.1.2. Oczekiwanie na zadania wysyłane do puli wątków ... 359
9.1.3. Zadania oczekujące na inne zadania ... 363
9.1.4. Unikanie współzawodnictwa w dostępie do kolejki zadań ... 366
9.1.5. Wykradanie zadań ... 368

9.2. Przerywanie wykonywania wątków 372
9.2.1. Uruchamianie i przerywanie innego wątku 373
9.2.2. Wykrywanie przerwania wątku 375
9.2.3. Przerywanie oczekiwania na zmienną warunkową 375
9.2.4. Przerywanie oczekiwania na zmienną typu std::condition_variable_any 379
9.2.5. Przerywanie pozostałych wywołań blokujących 381
9.2.6. Obsługa przerwań 382
9.2.7. Przerywanie zadań wykonywanych w tle podczas zamykania aplikacji 383
9.3. Podsumowanie 384

Rozdział 10. Algorytmy równoległości 385

10.1. Algorytmy równoległe w bibliotece standardowej 385
10.2. Polityki wykonywania 386
10.2.1. Ogólny efekt wyboru polityki wykonywania 386
10.2.2. std::execution::sequenced_policy 388
10.2.3. std::execution::parallel_policy 388
10.2.4. std::execution::parallel_unsequenced_policy 389
10.3. Algorytmy równoległości w bibliotece standardowej C++ 390
10.3.1. Przykłady używania algorytmów równoległych 392
10.3.2. Licznik odwiedzin 394
10.4. Podsumowanie 396

Rozdział 11. Testowanie i debugowanie aplikacji wielowątkowych 397

11.1. Rodzaje błędów związanych z przetwarzaniem współbieżnym 398
11.1.1. Niechciane blokowanie 398
11.1.2. Sytuacje wyścigu 399
11.2. Techniki lokalizacji błędów związanych z przetwarzaniem współbieżnym 400
11.2.1. Przeglądanie kodu w celu znalezienia ewentualnych błędów 401
11.2.2. Znajdowanie błędów związanych z przetwarzaniem współbieżnym poprzez testowanie kodu 403
11.2.3. Projektowanie kodu pod kątem łatwości testowania 405
11.2.4. Techniki testowania wielowątkowego kodu 407
11.2.5. Projektowanie struktury wielowątkowego kodu testowego 410
11.2.6. Testowanie wydajności wielowątkowego kodu 413
11.3. Podsumowanie 414

Dodatek A. Krótki przegląd wybranych elementów języka C++11 415

A.1. Referencje do r-wartości 415
A.2. Usunięte funkcje 420
A.3. Funkcje domyślne 421
A.4. Funkcje constexpr 425
A.5. Funkcje lambda 430
A.6. Szablony zmiennoargumentowe 436
A.7. Automatyczne określanie typu zmiennej 440
A.8. Zmienne lokalne wątków 441
A.9. Ustalanie argumentu szablonu klasy 442
A.10. Podsumowanie 443

Kup ksi k Pole ksi k

Dodatek B. Krótkie zestawienie bibliotek przetwarzania współbieżnego 445

Dodatek C. Framework przekazywania komunikatów i kompletny przykład implementacji systemu bankomatu 447

Dodatek D. Biblioteka wątków języka C++ 465

D.1. Nagłówek <chrono> ... 465
D.2. Nagłówek <condition_variable> ... 481
D.3. Nagłówek <atomic> ... 498
D.4. Nagłówek <future> ... 536
D.5. Nagłówek <mutex> ... 561
D.6. Nagłówek <ratio> ... 613
D.7. Nagłówek <thread> ... 619

Skorowidz 631

Kup ksi k Pole ksi k

4 Synchronizacja współbieżnych operacji

W tym rozdziale zostaną omówione następujące zagadnienia:

- oczekiwanie na zdarzenie;
- oczekiwanie na jednorazowe zdarzenia za pomocą przyszłości;
- oczekiwanie z limitem czasowym;
- upraszczanie kodu za pomocą technik synchronizowania operacji.

W poprzednim rozdziale przeanalizowaliśmy rozmaite sposoby ochrony danych współdzielonych przez wiele wątków. Okazuje się jednak, że w pewnych przypadkach jest potrzebna nie tyle ochrona danych, co synchronizacja działań podejmowanych przez różne wątki. Wątek może na przykład czekać z realizacją własnej operacji na zakończenie pewnego zadania przez inny wątek. Ogólnie w wielu przypadkach wątek oczekujący na określone zdarzenie lub spełnienie pewnego warunku jest najwygodniejszym rozwiązaniem. Mimo że analogiczne rozwiązanie można zaimplementować w formie mechanizmu okresowego sprawdzania flagi zakończonego zadania lub innej wartości zapisanej we współdzielonych danych, taki model byłby daleki od ideału. Konieczność synchronizacji operacji wykonywanych przez różne wątki jest dość typowym scenariuszem, zatem biblioteka standardowa języka C++ oferuje mechanizmy ułatwiające obsługę tego modelu, w tym **zmienne warunkowe** i tzw. **przyszłości**. Te funkcje zostały rozszerzone w specyfikacji technicznej współbieżności i dostarczają dodatkowe operacje dla *przyszłości* oraz nowe możliwości w postaci **zasuw** i **barier**.

Kup ksi k Pole ksi k

W tym rozdziale omówię techniki oczekiwania na zdarzenia przy użyciu zmiennych warunkowych, przyszłości, zasuw i barier, oraz sposoby upraszczania synchronizacji operacji.

4.1. Oczekiwanie na zdarzenie lub inny warunek

Przypuśćmy, że podróżujemy nocnym pociągiem. Jednym ze sposobów zagwarantowania, że wysiądziemy na właściwej stacji, jest unikanie snu i sprawdzanie wszystkich stacji, na których zatrzymuje się nasz pociąg. W ten sposób nie przegapimy naszej stacji, jednak po dotarciu na miejsce będziemy bardzo zmęczeni. Alternatywnym rozwiązaniem jest sprawdzenie rozkładu jazdy pod kątem godziny przyjazdu, ustawienie budzika z pewnym wyprzedzeniem względem tej godziny i pójście spać. To rozwiązanie jest dość bezpieczne — nie przegapimy naszej stacji, ale jeśli pociąg się spóźni, wstaniemy zbyt wcześnie. Nie można też wykluczyć sytuacji, w której wyczerpią się baterie w budziku — w takim przypadku możemy zaspać i przegapić swoją stację. Idealnym rozwiązaniem byłaby możliwość pójścia spać i skorzystania z pomocy czegoś (lub kogoś), co obudziłoby nas bezpośrednio przed osiągnięciem stacji docelowej.

Jaki to ma związek z wątkami? Jeśli jeden wątek czeka, aż inny wątek zakończy jakieś zadanie, ma do wyboru kilka możliwych rozwiązań. Po pierwsze, może stale sprawdzać odpowiednią flagę we współdzielonych danych (chronionych przez muteks); flaga zostanie ustawiona przez drugi wątek w momencie zakończenia zadania. Takie rozwiązanie jest nieefektywne z dwóch powodów: wątek, który wielokrotnie sprawdza wspomnianą flagę, zajmuje cenny czas procesora, a muteks zablokowany przez oczekujący wątek nie jest dostępny dla żadnego innego wątku. Oba te czynniki działają na niekorzyść oczekującego wątku, ponieważ ten wątek zajmuje zasoby potrzebne także do działania wątku, na który czeka, co opóźnia wykonanie zadania i ustawienie odpowiedniej flagi. Sytuacja przypomina unikanie snu przez całą podróż pociągiem i prowadzenie rozmowy z maszynistą — maszynista zajęty rozmową musi prowadzić pociąg nieco wolniej, zatem później dotrzemy na swoją stację. Podobnie wątek oczekujący zajmuje zasoby, które mogłyby być używane przez pozostałe wątki w systemie, przez co czas oczekiwania może być dłuższy, niż to konieczne.

Druga opcja polega na przechodzeniu wątku oczekującego w stan uśpienia na krótkie momenty i okresowym wykonywaniu testów za pomocą funkcji `std::this_thread::sleep_for()` (patrz punkt 4.3):

```
bool flag;
std::mutex m;

void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();    ← ❶ Odblokowuje muteks
        std::this_thread::sleep_for(std::chrono::milliseconds(100));    ← ❷ Czeka 100 ms
        lk.lock();    ← ❸ Ponownie blokuje muteks
    }
}
```

Kup książkę Poleć książkę

Wywołanie funkcji w pętli odblokowuje muteks ❶ przed przejściem w stan uśpienia ❷ i ponownie blokuje ten muteks po wyjściu z tego stanu ❸ — dzięki temu drugi wątek ma szanse uzyskania dostępu do flagi i jej ustawienia.

Opisane rozwiązanie jest o tyle dobre, że uśpiony wątek nie zajmuje bezproduktywnie czasu procesora. Warto jednak pamiętać, że dobór właściwego czasu uśpienia jest dość trudny. Zbyt krótki czas przebywania w tym stanie spowoduje, że wątek będzie tracił czas procesora na zbyt częste testy; zbyt długi czas uśpienia będzie oznaczał, że wątek będzie przebywał w tym stanie nawet po zakończeniu zadania, na które oczekuje, zatem opóźnienie w działaniu wątku oczekującego będzie zbyt duże. Takie „zaspanie" wątku rzadko ma bezpośredni wpływ na wynik operacji wykonywanych przez program, ale już w przypadku szybkiej gry może powodować pominięcie niektórych klatek animacji, a w przypadku aplikacji czasu rzeczywistego może oznaczać pominięcie przydziału czasu procesora.

Trzecim, najlepszym rozwiązaniem jest użycie gotowych elementów biblioteki standardowej języka C++ umożliwiających oczekiwanie na określone zdarzenie. Najprostszym mechanizmem oczekiwania na zdarzenie generowane przez inny wątek (na przykład zdarzenie polegające na umieszczeniu dodatkowego zadania w potoku) jest tzw. **zmienna warunkowa**. Zmienna warunkowa jest powiązana z pewnym zdarzeniem lub **warunkiem** oraz co najmniej jednym wątkiem, który **czeka** na spełnienie tego warunku. Wątek, który odkrywa, że warunek jest spełniony, może **powiadomić** pozostałe wątki oczekujące na tę zmienną warunkową, aby je obudzić i umożliwić im dalsze przetwarzanie.

4.1.1. Oczekiwanie na spełnienie warunku za pomocą zmiennych warunkowych

Biblioteka standardowa języka C++ udostępnia **dwie** implementacje mechanizmu zmiennych warunkowych w formie klas `std::condition_variable` i `std::condition_variable_any`. Obie klasy zostały zadeklarowane w pliku nagłówkowym `<condition_variable>`. W obu przypadkach zapewnienie właściwej synchronizacji wymaga użycia muteksu — pierwsza klasa jest przystosowana tylko do obsługi muteksów typu `std::mutex`, natomiast druga klasa obsługuje wszystkie rodzaje muteksów spełniających pewien minimalny zbiór kryteriów (stąd przyrostek `_any`). Ponieważ klasa `std::condition_variable_any` jest bardziej uniwersalna, z jej stosowaniem wiążą się dodatkowe koszty w wymiarze wielkości, wydajności i zasobów systemu operacyjnego. Jeśli więc nie potrzebujemy dodatkowej elastyczności, powinniśmy stosować klasę `std::condition_variable`.

Jak należałoby użyć klasy `std::condition_variable` do obsługi przykładu opisanego na początku tego podrozdziału — jak sprawić, że wątek oczekujący na wykonanie jakiegoś zadania będzie uśpiony do momentu, w którym będą dostępne dane do przetworzenia? Na listingu 4.1 pokazano przykład kodu implementującego odpowiednie rozwiązanie przy użyciu zmiennej warunkowej.

Listing 4.1. Oczekiwanie na dane do przetworzenia za pomocą klasy std::condition_variable

```
std::mutex mut;
std::queue<data_chunk> data_queue;        ←❶
std::condition_variable data_cond;
```

Kup książkę Poleć książkę

```
void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);        ←❷
        data_cond.notify_one();        ←❸
    }
}

void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut);        ←❹
        data_cond.wait(
            lk,[]{return !data_queue.empty();});        ←❺
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();        ←❻
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

Na początku kodu zdefiniowano kolejkę ❶, która będzie używana do przekazywania danych pomiędzy dwoma wątkami. Kiedy dane są gotowe do przetworzenia, wątek, który je przygotował, blokuje muteks chroniący kolejkę za pomocą klasy `std::lock_guard` i umieszcza nowe dane w kolejce ❷. Wątek wywołuje następnie funkcję składową `notify_one()` dla obiektu klasy `std::condition_variable`, aby powiadomić oczekujący wątek (jeśli taki istnieje) o dostępności nowych danych ❸. Zwróć uwagę na umieszczenie w mniejszym zasięgu kodu odpowiedzialnego z przekazywanie danych do kolejki. Zmienna warunkowa jest definiowana *po* odblokowaniu muteksu — jeżeli wątek oczekujący zostanie obudzony natychmiast, nie będzie musiał ponownie nakładać blokady w oczekiwaniu na odblokowanie muteksu.

W tym modelu drugą stroną komunikacji jest wątek przetwarzający te dane. Wątek przetwarzający najpierw blokuje muteks, jednak tym razem użyto do tego celu klasy `std::unique_lock` zamiast klasy `std::lock_guard` ❹ — przyczyny tej decyzji zostaną wyjaśnione za chwilę. Wątek wywołuje następnie funkcję `wait()` dla obiektu klasy `std::condition_variable`. Na wejściu tego wywołania wątek przekazuje obiekt blokady i funkcję lambda reprezentującą warunek, który musi zostać spełniony przed przystąpieniem do dalszego przetwarzania ❺. Funkcje lambda to stosunkowo nowy element (wprowadzony w standardzie C++11), który umożliwia pisanie funkcji anonimowych w ramach innych wyrażeń. Wspomniane rozwiązanie wprost idealnie nadaje się do wskazywania predykatów w wywołaniach takich funkcji biblioteki standardowej jak `wait()`. W tym przypadku prosta funkcja lambda `[]{return !data_queue.empty();}` sprawdza, czy struktura reprezentowana przez zmienną `data_queue` nie jest pusta, tj.

czy kolejka zawiera jakieś dane gotowe do przetworzenia. Funkcje lambda zostaną szczegółowo omówione w części A.5 dodatku A.

Implementacja funkcji `wait()` sprawdza warunek (wywołując przekazaną funkcję lambda), po czym zwraca sterowanie, jeśli ten warunek jest spełniony (jeśli funkcja lambda zwróciła wartość `true`). Jeśli warunek nie jest spełniony (jeśli funkcja lambda zwróciła wartość `false`), funkcja `wait()` odblokowuje muteks i wprowadza bieżący wątek w stan blokady (oczekiwania). Kiedy zmienna warunkowa jest powiadamiana za pomocą funkcji `notify_one()` wywołanej przez wątek przygotowujący dane, wątek oczekujący jest budzony (odblokowywany), ponownie uzyskuje blokadę muteksu i jeszcze raz sprawdza warunek. Jeśli warunek dalszego przetwarzania jest spełniony, funkcja `wait()` zwraca sterowanie z zachowaniem blokady muteksu. Jeśli warunek nie jest spełniony, wątek odblokowuje muteks i ponownie przechodzi w stan oczekiwania. Właśnie dlatego w przykładzie należało użyć klasy `std::unique_lock` zamiast klasy `std::lock_guard` — wątek oczekujący musi odblokować muteks na czas oczekiwania i zablokować go ponownie po otrzymaniu powiadomienia, a klasa `std::lock_guard` nie zapewnia takiej elastyczności. Gdyby muteks pozostał zablokowany przez cały czas uśpienia tego wątku, wątek przygotowujący dane nie mógłby zablokować tego muteksu i dodać elementu do kolejki, zatem warunek budzenia wątku oczekującego nigdy nie zostałby spełniony.

Na listingu 4.1 użyłem prostej funkcji lambda ❺, która sprawdza, czy struktura kolejki nie jest pusta. Okazuje się, że w tej roli równie dobrze można by użyć dowolnej funkcji lub obiektu wywoływalnego. Jeśli programista dysponuje już funkcją sprawdzającą odpowiedni warunek (funkcja może oczywiście być nieporównanie bardziej złożona niż prosty test z powyższego przykładu), może przekazać tę funkcję bezpośrednio na wejściu funkcji `wait()`, bez konieczności opakowywania jej w ramach wyrażenia lambda. Po wywołaniu funkcji `wait()` zmienna warunkowa może sprawdzić wskazany warunek na wiele różnych sposobów, jednak podczas tego testu muteks zawsze jest zablokowany, a funkcja `wait()` natychmiast zwraca sterowanie, pod warunkiem że przekazana funkcja sprawdzająca ten warunek zwróciła wartość `true`. Jeśli wątek oczekujący ponownie uzyskuje muteks i sprawdza warunek, mimo że nie otrzymał powiadomienia od innego wątku i jego działania nie są bezpośrednią odpowiedzią na takie powiadomienie, mamy do czynienia z tzw. **pozornym budzeniem** (ang. *spurious wake*). Ponieważ optymalna liczba i częstotliwość takich pozornych budzeń są z definicji trudne do oszacowania, funkcja sprawdzająca prawdziwość warunku nie powinna powodować żadnych skutków ubocznych. Gdyby ta funkcja powodowała skutki uboczne, programista musiałby przygotować swój kod na wielokrotne występowanie tych skutków przed spełnieniem warunku.

Ogólnie rzecz biorąc, `std::condition_variable::wait` to **rodzaj optymalizacji**. Istotnie tak jest: odpowiednio przygotowana (choć nieidealna) technika implementacji to po prostu zwykła pętla.

```
template<typename Predicate>
void minimal_wait(std::unique_lock<std::mutex>& lk,Predicate pred){
    while(!pred()){
        lk.unlock();
        lk.lock();
    }
}
```

Kup ksi k Pole ksi k

Kod musi być przygotowany do pracy z tak minimalną implementacją funkcji `wait()`, a także implementacją, która budzi wątek jedynie po wywołaniu `notify_one()` lub `notify_all()`.

Możliwość odblokowania obiektu klasy `std::unique_lock` nie jest używana tylko dla wywołania funkcji `wait()` — analogiczne rozwiązanie zastosowaliśmy po uzyskaniu danych do przetworzenia, ale przed przystąpieniem do właściwego przetwarzania ❻. Przetwarzanie danych może być czasochłonną operacją, a jak wiemy z rozdziału 3., utrzymywanie blokady muteksu dłużej, niż to konieczne, nie jest dobrym rozwiązaniem.

Stosowanie struktury kolejki do przekazywania danych pomiędzy wątkami (jak na listingu 4.1) jest dość typowym rozwiązaniem. Jeśli projekt aplikacji jest właściwy, synchronizacja powinna dotyczyć samej kolejki, co znacznie ogranicza liczbę potencjalnych problemów i problematycznych sytuacji wyścigu. Spróbujmy więc wyodrębnić z listingu 4.1 uniwersalną kolejkę gwarantującą bezpieczne przetwarzanie wielowątkowe.

4.1.2. Budowa kolejki gwarantującej bezpieczne przetwarzanie wielowątkowe przy użyciu zmiennych warunkowych

Przed przystąpieniem do projektowania uniwersalnej kolejki warto poświęcić kilka minut analizie operacji, które trzeba będzie zaimplementować dla tej struktury danych (podobnie jak w przypadku stosu gwarantującego bezpieczeństwo przetwarzania wielowątkowego z punktu 3.2.3). Przyjrzyjmy się kontenerowi `std::queue<>` dostępnemu w bibliotece standardowej języka C++ (patrz listing 4.2), który będzie stanowił punkt wyjścia dla naszej implementacji.

Listing 4.2. Interfejs kontenera std::queue

```
template <class T, class Container = std::deque<T> >
class queue {
public:
    explicit queue(const Container&);
    explicit queue(Container&& = Container());
    template <class Alloc> explicit queue(const Alloc&);
    template <class Alloc> queue(const Container&, const Alloc&);
    template <class Alloc> queue(Container&&, const Alloc&);
    template <class Alloc> queue(queue&&, const Alloc&);
    void swap(queue& q);
    bool empty() const;
    size_type size() const;
    T& front();
    const T& front() const;
    T& back();
    const T& back() const;
    void push(const T& x);
    void push(T&& x);
    void pop();
    template <class... Args> void emplace(Args&&... args);
};
```

Jeśli pominiemy operacje konstruowania, przypisywania i wymiany, pozostaną nam zaledwie trzy grupy operacji: operacje zwracające stan całej kolejki (`empty()` i `size()`), operacje zwracające pojedyncze elementy kolejki (`front()` i `back()`) oraz operacje

Kup ksi k Pole ksi k

modyfikujące kolejkę (`push()`, `pop()` i `emplace()`). Mamy więc do czynienia z sytuacją analogiczną do tej opisanej w punkcie 3.2.3 (gdzie omawialiśmy strukturę stosu), zatem opisany interfejs jest narażony na te same problemy związane z sytuacjami wyścigów. W tym przypadku należy połączyć funkcje `front()` i `pop()` w jedno wywołanie, tak jak wcześniej połączyliśmy funkcje `top()` i `pop()` dla struktury stosu. Warto jeszcze zwrócić uwagę na pewien nowy element w kodzie z listingu 4.1 — podczas używania kolejki do przekazywania danych pomiędzy wątkami wątek docelowy zwykle musi czekać na te dane. Warto więc zaimplementować funkcję `pop()` w dwóch wersjach — pierwsza funkcja, `try_pop()`, próbuje pobrać wartość z kolejki, ale zawsze zwraca sterowanie bezpośrednio po wywołaniu, nawet jeśli kolejka nie zawierała żadnej wartości (wtedy funkcja sygnalizuje błąd); druga funkcja, `wait_and_pop()`, czeka na pojawienie się w kolejce wartości do pobrania. Po wprowadzeniu zmian zgodnie ze schematem opisanym już przy okazji przykładu stosu interfejs struktury kolejki powinien wyglądać tak jak na listingu 4.3.

Listing 4.3. Interfejs struktury danych threadsafe_queue

```
#include <memory>        ← Dla typu std::shared_ptr

template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue&);
    threadsafe_queue& operator=(
        const threadsafe_queue&) = delete;   ← Dla uproszczenia wyklucza możliwość przypisywania

    void push(T new_value);

    bool try_pop(T& value);        ← ❶

    std::shared_ptr<T> try_pop();        ← ❷
    void wait_and_pop(T& value);
    std::shared_ptr<T> wait_and_pop();
    bool empty() const;
};
```

Podobnie jak w przypadku stosu, na listingu 4.3 usunięto konstruktory i operator przypisania, aby uprościć analizowany kod. Tak jak wcześniej, także tym razem funkcje `try_pop()` i `wait_for_pop()` występują w dwóch wersjach. Pierwsza przeciążona wersja funkcji `try_pop()` ❶ zapisuje pobraną wartość we wskazywanej zmiennej, tak aby można było użyć tej wartości w roli informacji o stanie; funkcja zwraca wartość `true`, jeśli uzyskała jakąś wartość — w przeciwnym razie funkcja zwraca wartość `false` (patrz część A.2 dodatku A). Druga przeciążona wersja ❷ nie może działać w ten sam sposób, ponieważ natychmiast zwraca uzyskaną wartość. Jeśli jednak funkcja nie uzyskała żadnej wartości, może zwrócić wskaźnik równy `NULL`.

Jaki to ma związek z listingiem 4.1? Okazuje się, że możemy wyodrębnić kod funkcji `push()` i `wait_and_pop()` z tamtego listingu i na tej podstawie przygotować nową implementację (patrz listing 4.4).

Kup ksi k Pole ksi k

Listing 4.4. Funkcje push() i wait_and_pop() wyodrębnione z listingu 4.1

```
#include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
    std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }
};

threadsafe_queue<data_chunk> data_queue;        ←❶

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        data_queue.push(data);        ←❷
    }
}

void data_processing_thread()
{
    while(true)
    {
        data_chunk data;
        data_queue.wait_and_pop(data);        ←❸
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

Muteks i zmienna warunkowa są teraz elementami składowymi obiektu klasy `threadsafe_queue`, zatem nie jest potrzebne stosowanie odrębnych zmiennych ❶, a wywołanie funkcji `push()` nie wymaga zewnętrznych mechanizmów synchronizacji ❷. Jak widać, także funkcja `wait_and_pop()` uwzględnia stan zmiennej warunkowej ❸.

Napisanie drugiej wersji przeciążonej funkcji `wait_and_pop()` nie stanowi żadnego problemu; także pozostałe funkcje można niemal skopiować z przykładu stosu pokazanego na listingu 3.5. Ostateczną wersję implementacji kolejki pokazano na listingu 4.5.

Listing 4.5. Kompletna definicja klasy kolejki gwarantującej bezpieczeństwo przetwarzania wielowątkowego (dzięki użyciu zmiennych warunkowych)

```
#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;          ←❶ Muteks musi być modyfikowalny
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    threadsafe_queue(threadsafe_queue const& other)
    {
        std::lock_guard<std::mutex> lk(other.mut);
        data_queue=other.data_queue;
    }

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }

    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
```

Kup książkę Poleć książkę

```
        value=data_queue.front();
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
        return res;
    }

    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

Mimo że `empty()` jest stałą funkcją składową i mimo że parametr `other` konstruktora kopiującego jest stałą referencją, pozostałe wątki mogą dysponować niestałymi referencjami do tego obiektu i wywoływać funkcje składowe zmieniające jego stan, zatem blokowanie muteksu wciąż jest konieczne. Ponieważ blokowanie muteksu jest operacją zmieniającą stan obiektu, obiekt muteksu należy oznaczyć jako modyfikowalny (ang. *mutable*) ❶, tak aby można było blokować ten muteks w ciele funkcji `empty()` i konstruktora kopiującego.

Zmienne warunkowe są przydatne także w sytuacji, w której wiele wątków czeka na to samo zdarzenie. Jeśli celem stosowania wątków jest dzielenie obciążenia i jeśli tylko jeden wątek powinien reagować na powiadomienie, można zastosować dokładnie taką samą strukturę jak ta z listingu 4.1 — wystarczy uruchomić wiele egzemplarzy wątku przetwarzającego dane. Po przygotowaniu nowych danych wywołanie funkcji `notify_one()` spowoduje, że jeden z wątków aktualnie wykonujących funkcję `wait()` sprawdzi warunek. Ponieważ do struktury `data_queue` właśnie dodano nowe dane, funkcja `wait()` zwróci sterowanie (z powodu dodania elementu do egzemplarza `data_queue`). Nie wiadomo, do którego wątku trafi powiadomienie ani nawet czy istnieje wątek oczekujący na to powiadomienie (nie można przecież wykluczyć, że wszystkie wątki w danej chwili przetwarzają swoje dane).

Warto też pamiętać o możliwości oczekiwania na to samo zdarzenie przez wiele wątków, z których każdy musi zareagować na powiadomienie. Opisany scenariusz może mieć związek z inicjalizacją współdzielonych danych, gdzie wszystkie wątki przetwarzające operują na tych samych danych i muszą czekać albo na ich inicjalizację (w takim przypadku istnieją lepsze mechanizmy — patrz punkt 3.3.1 w rozdziale 3.), albo na ich aktualizację (na przykład w ramach okresowej, wielokrotnej inicjalizacji). W opisanych przypadkach wątek przygotowujący dane może wywołać funkcję składową `notify_all()` dla zmiennej warunkowej (zamiast funkcji `notify_one()`). Jak nietrudno się domyślić, funkcja powoduje, że **wszystkie** wątki aktualnie wykonujące funkcję `wait()` sprawdzą warunek, na który czekają.

Kup ksi k Pole ksi k

Jeśli wątek wywołujący w założeniu ma oczekiwać na dane zdarzenie tylko raz, czyli jeśli po spełnieniu warunku wątek nie będzie ponownie czekał na tę samą zmienną warunkową, być może warto zastosować inny mechanizm synchronizacji niż zmienna warunkowa. Zmienne warunkowe są szczególnie nieefektywne w sytuacji, gdy warunkiem, na który oczekują wątki, jest dostępność określonego elementu danych. W takim przypadku lepszym rozwiązaniem jest użycie mechanizmu **przyszłości**.

4.2. Oczekiwanie na jednorazowe zdarzenia za pomocą przyszłości

Przypuśćmy, że planujemy podróż samolotem. Po przyjeździe na lotnisko i przejściu rozmaitych procedur wciąż musimy czekać na komunikat dotyczący gotowości naszego samolotu na przyjęcie pasażerów (zdarza się, że pasażerowie muszą czekać wiele godzin). Możemy oczywiście znaleźć sposób, aby ten czas minął nieco szybciej (możemy na przykład czytać książkę, przeglądać strony internetowe lub udać się na posiłek do drogiej lotniskowej kawiarni), jednak niezależnie od sposobu spędzania czasu czekamy na jedno — sygnał wzywający do udania się na pokład samolotu. Co więcej, interesujący nas lot odbędzie się tylko raz, zatem przy okazji następnego wyjazdu na wakacje będziemy czekali na inny lot.

Twórcy biblioteki standardowej języka C++ rozwiązali problem jednorazowych zdarzeń za pomocą mechanizmu nazwanego **przyszłością** (ang. *future*). Wątek, który musi czekać na określone jednorazowe zdarzenie, powinien uzyskać przyszłość reprezentującą to zdarzenie. Wątek oczekujący na tę przyszłość może następnie okresowo sprawdzać, czy odpowiednie zdarzenie nie nastąpiło (tak jak pasażerowie co jakiś czas zerkają na tablicę odlotów), i jednocześnie pomiędzy tymi testami wykonywać inne zadanie (spożywać drogi deser w lotniskowej kawiarni). Alternatywnym rozwiązaniem jest wykonywanie innego zadania do momentu, w którym dalsze działanie nie jest możliwe bez określonego zdarzenia, i przejście w stan **gotowości** na przyszłość. Przyszłość może, ale nie musi być powiązana z danymi (tak jak tablica odlotów może wskazywać rękawy prowadzące do właściwych samolotów). Po wystąpieniu zdarzenia (po osiągnięciu **gotowości** przez przyszłość) nie jest możliwe wyzerowanie tej przyszłości.

W bibliotece standardowej języka C++ istnieją dwa rodzaje przyszłości zaimplementowane w formie dwóch szablonów klas zadeklarowanych w nagłówku biblioteki <future>: **przyszłości unikatowe** (std::future<>) oraz **przyszłości współdzielone** (std::↪shared_future<>). Wymienione klasy opracowano na bazie typów std::unique_ptr i std::shared_ptr. Obiekt typu std::future jest jedynym egzemplarzem odwołującym się do powiązanego zdarzenia, natomiast do jednego zdarzenia może się odwoływać wiele egzemplarzy typu std::shared_future. W drugim przypadku wszystkie egzemplarze są **gotowe** jednocześnie i wszystkie mogą uzyskiwać dostęp do dowolnych danych powiązanych z danym zdarzeniem. Właśnie z myślą o powiązanych danych zaprojektowano te szablony klas — tak jak w przypadku szablonów std::unique_ptr i std::shared_ptr, parametry szablonów std::future<> i std::shared_future<> reprezentują właśnie typy powiązanych danych. W razie braku powiązanych danych należy stosować następujące specjalizacje tych szablonów: std:future<void> i std::shared_future<void>. Mimo że przyszłości służą do komunikacji pomiędzy wątkami, same obiekty przyszłości nie oferują mechanizmów synchronizowanego dostępu. Jeśli wiele wątków potrzebuje dostępu do

Kup ksi k Pole ksi k

jednego obiektu przyszłości, należy chronić ten dostęp za pomocą muteksu lub innego mechanizmu synchronizacji (patrz rozdział 3.). Jak napiszę w punkcie 4.2.5 w dalszej części tego podrozdziału, wiele wątków może uzyskiwać dostęp do własnej kopii obiektu typu `std::shared_future<>` bez konieczności dodatkowej synchronizacji, nawet jeśli wszystkie te kopie odwołują się do tego samego asynchronicznego wyniku.

Specyfikacja techniczna Concurrency TS dostarcza rozszerzane wersje tych szablonów klas w przestrzeni nazw `std::experimental` — `std::experimental::future<>` i `std::experimental::shared_future<>`. Działają one analogicznie do ich odpowiedników w przestrzeni nazw `std`, przy czym oferują dodatkowe funkcje składowe dostarczające kolejne możliwości. Trzeba koniecznie zwrócić uwagę na to, że przestrzeń nazw `std::` ↪`experimental` nie narzuca żadnych wymagań w zakresie jakości kodu (mam nadzieję, że implementacja będzie miała taką samą jakość jak pozostały kod znajdujący się w bibliotece danego dostawcy). Trzeba podkreślić fakt, że to nie są standardowe klasy i funkcje, więc mogą nie mieć dokładnie tej samej składni i semantyki, gdy wreszcie zostaną zaadaptowane (o ile w ogóle tak się stanie) w przyszłych wersjach standardu C++. Aby móc skorzystać z możliwości oferowanych przez te klasy i funkcje, konieczne jest umieszczenie w kodzie polecenia dodającego nagłówek `<experimental/future>`.

Najprostszym przykładem jednorazowego zdarzenia jest wynik obliczeń wykonywanych w tle. Już w rozdziale 2. napisałem, że klasa `std::thread` nie udostępnia prostych mechanizmów zwracania wartości wynikowych dla tego rodzaju zadań, i zapowiedziałem wprowadzenie odpowiednich rozwiązań w rozdziale 4. przy okazji omawiania przyszłości — czas zapoznać się z tymi rozwiązaniami.

4.2.1. Zwracanie wartości przez zadania wykonywane w tle

Przypuśćmy, że nasza aplikacja wykonuje czasochłonne obliczenia, które ostatecznie pozwolą uzyskać oczekiwany wynik. Załóżmy, że wartość wynikowa nie jest potrzebna na tym etapie działania programu. Być może udało nam się wymyślić sposób poszukiwania odpowiedzi na pytanie o życie, wszechświat i całą resztę stawiane w książkach Douglasa Adamsa[1]. Moglibyśmy oczywiście uruchomić nowy wątek, który wykona niezbędne obliczenia, jednak takie rozwiązanie wiązałoby się z koniecznością przekazania wyników z powrotem do wątku głównego, ponieważ klasa `std::thread` nie oferuje alternatywnego mechanizmu zwracania wartości wynikowych. W takim przypadku sporym ułatwieniem jest użycie szablonu funkcji `std::async` (zadeklarowanego w pliku nagłówkowym `<future>`).

Asynchroniczne zadanie, którego wynik nie jest potrzebny na bieżącym etapie działania programu, można rozpocząć za pomocą funkcji `std::async`. Zamiast zwracania obiektu klasy `std::thread`, który umożliwi oczekiwanie na zakończenie danego wątku, funkcja `std::async` zwraca obiekt klasy `std::future`, który w przyszłości będzie zawierał wartość wynikową. W miejscu, w którym aplikacja będzie potrzebowała tej wartości, należy wywołać funkcję `get()` dla obiektu przyszłości — wywołanie tej funkcji zablokuje wykonywanie bieżącego wątku do momentu osiągnięcia **gotowości** przez przyszłość, po czym zwróci uzyskaną wartość. Prosty przykład użycia tych elementów pokazano na listingu 4.6.

[1] W książce *Autostopem przez Galaktykę* zbudowano komputer Deep Thought, który miał odpowiedzieć na pytanie o życie, wszechświat i całą resztę. Odpowiedzią na to pytanie była liczba 42.

Kup ksi k Pole ksi k

Listing 4.6. Przykład użycia szablonu klasy std::future do uzyskania wartości wynikowej asynchronicznego zadania

```
#include <future>
#include <iostream>

int find_the_answer_to_ltuae();
void do_other_stuff();
int main()
{
    std::future<int> the_answer=std::async(find_the_answer_to_ltuae);
    do_other_stuff();
    std::cout<<"Odpowiedź brzmi "<<the_answer.get()<<std::endl;
}
```

Szablon funkcji `std::async` umożliwia przekazywanie dodatkowych argumentów na wejściu wywoływanej funkcji — wystarczy dodać te argumenty do wywołania (podobnie jak w przypadku klasy `std::thread`). Jeśli pierwszy argument reprezentuje wskaźnik do funkcji składowej, drugi argument zawiera obiekt, dla którego ma zostać wywołana ta funkcja składowa (bezpośrednio, za pośrednictwem wskaźnika lub poprzez opakowanie `std::ref`), a pozostałe argumenty są przekazywane na wejściu tej funkcji składowej. W przeciwnym razie drugi i kolejne argumenty są przekazywane na wejściu funkcji składowej lub wywoływalnego obiektu wskazanego za pośrednictwem pierwszego argumentu. Tak jak w przypadku klasy `std::thread`, jeśli argumenty mają postać r-wartości, zostaną utworzone kopie poprzez **przeniesienie** oryginalnych wartości. Dzięki temu możemy stosować typy oferujące tylko możliwość przenoszenia zarówno w roli obiektów funkcji, jak i w roli argumentów. Przykład takiego rozwiązania pokazano na listingu 4.7.

Listing 4.7. Przekazywanie argumentów na wejściu funkcji wątku std::async

```
#include <string>
#include <future>

struct X
{
    void foo(int,std::string const&);
    std::string bar(std::string const&);
};
X x;
auto f1=std::async(&X::foo,&x,42,"witaj");     <- Wywołuje p->foo(42,"witaj"), gdzie p jest reprezentowane przez &x
auto f2=std::async(&X::bar,x,"żegnaj");        <- Wywołuje tmpx.bar("żegnaj"), gdzie tmpx jest kopią x
struct Y
{
    double operator()(double);
};
Y y;
auto f3=std::async(Y(),3.141);                 <- Wywołuje tmpy(3.141), gdzie tmpy jest tworzone za pomocą konstruktora przenoszącego Y()
auto f4=std::async(std::ref(y),2.718);         <- Wywołuje y(2.718)
X baz(X&);
std::async(baz,std::ref(x));                   <- Wywołuje baz(x)
class move_only
{
public:
    move_only();
    move_only(move_only&&)
```

```
    move_only(move_only const&) = delete;
    move_only& operator=(move_only&&);
    move_only& operator=(move_only const&) = delete;

    void operator()();
};
auto f5=std::async(move_only());   ← Wywołuje tmp(), gdzie tmp jest konstruowany
                                     na podstawie wywołania std::move(move_only())
```

Domyślnie to od stosowanej implementacji zależy, czy funkcja `std::async` uruchamia nowy wątek, czy wskazane zadanie będzie wykonywane w sposób synchroniczny. W większości przypadków standardowe rozwiązanie jest wystarczające, jednak programista może wybrać właściwy tryb za pomocą dodatkowego parametru funkcji `std::async` przekazywanego przed funkcją do wywołania. Wspomniany parametr typu `std::launch` może mieć albo wartość `std::launch::deferred` (wówczas wywołanie funkcji jest odkładane do momentu wywołania funkcji `wait()` lub `get()` dla danej przyszłości), albo wartość `std::launch::async` (wówczas funkcja musi być wykonywana w odrębnym wątku), albo wartość `std::launch::deferred | std::launch::async` (wówczas decyzja należy do implementacji). Ostatnia opcja jest stosowana w roli wartości domyślnej. Jeśli wywołanie funkcji jest odkładane na przyszłość, może nigdy nie nastąpić. Na przykład:

```
auto f6=std::async(std::launch::async,Y(),1.2);   ← Wykonywane w nowym wątku
auto f7=std::async(std::launch::deferred,baz,std::ref(x));   ← Wykonywane w ramach
auto f8=std::async(                                            funkcji wait() lub get()
    std::launch::deferred | std::launch::async,   ← Wybór
    baz,std::ref(x));                                implementacji
auto f9=std::async(baz,std::ref(x));
f7.wait();   ← Wywołanie odroczonej funkcji
```

Jak się przekonasz w dalszej części tego rozdziału (i ponownie w rozdziale 8.), funkcja `std::async` ułatwia dzielenie algorytmów na współbieżnie wykonywane zadania. Okazuje się jednak, że nie jest to jedyny sposób kojarzenia obiektu typu `std::future` z zadaniem — alternatywnym rozwiązaniem jest opakowanie zadania w ramach egzemplarza szablonu klasy `std::packaged_task<>` lub napisanie kodu bezpośrednio ustawiającego wartości za pomocą szablonu klasy `std::promise<>`. Szablon klasy `std::packaged_task` jest abstrakcją wyższego poziomu (w porównaniu z szablonem `std::promise`), zatem właśnie ten szablon omówimy jako pierwszy.

4.2.2. *Wiązanie zadania z przyszłością*

Szablon klasy `std::packaged_task<>` wiąże przyszłość z funkcją lub wywoływalnym obiektem. W momencie wywołania obiektu typu `std::packaged_task<>` wywołana zostaje powiązana funkcja lub wywoływalny obiekt, a sama przyszłość przechodzi w stan **gotowości** (wartość wynikowa zostaje umieszczona w powiązanych danych). Opisaną strukturę można wykorzystać w roli elementu składowego podczas budowy puli wątków (patrz rozdział 9.) lub dowolnego innego schematu zarządzania zadaniami polegającego na przykład na wykonywaniu każdego zadania w osobnym wątku lub sekwencyjnym wykonywaniu zadań w jednym wątku działającym w tle. Jeśli jedną większą operację można podzielić na wiele autonomicznych podzadań, każde z tych podzadań można opakować w ramach obiektu klasy `std::packaged_task<>`, aby następnie przekazać ten obiekt do mechanizmu szeregowania zadań lub do puli wątków. W ten sposób można skutecznie ukryć szczegóły związane z poszczególnymi zadaniami — mechanizm szeregowania zadań operuje na obiektach klasy `std::packaged_task<>`, nie na poszczególnych funkcjach.

Kup ksi k Pole ksi k

Parametr szablonu klasy `std::packaged_task<>` reprezentuje sygnaturę funkcji — na przykład dla funkcji, która nie otrzymuje żadnych parametrów i nie zwraca wartości, należałoby użyć sygnatury `void()`, natomiast dla funkcji otrzymującej niestałą referencję do wartości typu `std::string` i wskaźnik do wartości typu `double` oraz zwracającej wartość typu `int` należałoby użyć sygnatury `int(std::string&,double*)`. Podczas konstruowania obiektu klasy `std::packaged_task` należy przekazać funkcję (lub wywoływalny obiekt) otrzymującą na wejściu wskazane parametry i zwracającą typ, który można przekonwertować na wskazany typ danych. Dokładne dopasowanie typów nie jest wymagane — istnieje możliwość skonstruowania obiektu klasy `std::packaged_task ↪<double(double)>` na podstawie funkcji otrzymującej na wejściu wartość typu `int` i zwracającej wartość typu `float`, ponieważ wymienione typy mogą być automatycznie konwertowane.

Typ wartości zwracanych przez wskazaną funkcję identyfikuje typ zwracany przez funkcję składową `get_future()` konstruowanego obiektu klasy `std::future<>`, natomiast lista argumentów zdefiniowana w ramach sygnatury funkcji jest używana do wyznaczania sygnatury operatora wywołania funkcji zadania reprezentowanego przez ten obiekt. Przykład częściowej definicji klasy `std::packaged_task<std::string(std:: ↪vector<char>*,int)>` pokazano na listingu 4.8.

Listing 4.8. Częściowa definicja specjalizacji szablonu klasy std::packaged_task<>

```
template<>
class packaged_task<std::string(std::vector<char>*,int)>
{
public:
    template<typename Callable>
    explicit packaged_task(Callable&& f);
    std::future<std::string> get_future();
    void operator()(std::vector<char>*,int);
};
```

Egzemplarz klasy `std::packaged_task` jest obiektem wywoływalnym i jako taki może być opakowany w ramach obiektu klasy `std::function`, przekazany do obiektu klasy `std::thread` w roli funkcji wątku, przekazany do dowolnej innej funkcji oczekującej wywoływalnego obiektu, a nawet bezpośrednio wywołany. W momencie wywołania obiektu klasy `std::packaged_task` jako obiektu funkcji argumenty przekazane na wejściu operatora wywołania są przekazywane do opakowanej funkcji, a zwracana wartość jest zapisywana jako asynchroniczny wynik w obiekcie typu `std::future` (obiekt można następnie uzyskać za pomocą funkcji `get_future()`). Oznacza to, że możemy opakować zadanie w obiekcie klasy `std::packaged_task` i uzyskać przyszłość przed przekazaniem tego obiektu do miejsca, gdzie zostanie wywołany. W momencie, w którym program będzie potrzebował wyniku, wystarczy poczekać na osiągnięcie gotowości przez tę przyszłość. Praktyczny przykład takiego rozwiązania opisano w następnym podpunkcie.

PRZEKAZYWANIE ZADAŃ POMIĘDZY WĄTKAMI

Wiele frameworków graficznego interfejsu użytkownika wymaga, aby aktualizacje tego interfejsu były wykonywane przez określone wątki. Oznacza to, że jeśli jakiś inny wątek musi zaktualizować graficzny interfejs użytkownika, powinien wysłać komunikat do

Kup ksi k Pole ksi k

właściwego wątku, aby wyznaczony wątek wykonał to zadanie w jego imieniu. Szablon klasy `std::packaged_task` oferuje odpowiednie rozwiązania bez konieczności stosowania niestandardowych komunikatów dla każdego zadania związanego z działaniem graficznego interfejsu użytkownika (patrz listing 4.9).

Listing 4.9. Uruchamianie kodu w wątku graficznego interfejsu użytkownika za pomocą szablonu klasy std::packaged_task

```
#include <deque>
#include <mutex>
#include <future>
#include <thread>
#include <utility>

std::mutex m;
std::deque<std::packaged_task<void()> > tasks;

bool gui_shutdown_message_received();
void get_and_process_gui_message();

void gui_thread()     ←❶
{
    while(!gui_shutdown_message_received())     ←❷
    {
        get_and_process_gui_message();     ←❸
        std::packaged_task<void()> task;
        {
            std::lock_guard<std::mutex> lk(m);
            if(tasks.empty())     ←❹
                continue;
            task=std::move(tasks.front());     ←❺
            tasks.pop_front();
        }
        task();     ←❻
    }
}

std::thread gui_bg_thread(gui_thread);

template<typename Func>
std::future<void> post_task_for_gui_thread(Func f)
{
    std::packaged_task<void()> task(f);     ←❼
    std::future<void> res=task.get_future();     ←❽
    std::lock_guard<std::mutex> lk(m);
    tasks.push_back(std::move(task));     ←❾
    return res;     ←❿
}
```

Powyższy kod jest bardzo prosty: wątek graficznego interfejsu użytkownika ❶ działa w pętli do momentu otrzymania komunikatu sygnalizującego konieczność zamknięcia tego interfejsu ❷. W ciele tej pętli wątek sprawdza komunikaty dotyczące graficznego

interfejsu użytkownika ❸ (na przykład tego, że użytkownik kliknął jakiś element interfejsu) oraz ewentualne zadania w kolejce zadań. Jeśli kolejka nie zawiera żadnych zadań ❹, wątek przechodzi do następnej iteracji pętli; w przeciwnym razie wątek odczytuje zadanie z kolejki ❺, zwalnia blokadę tej kolejki, po czym uruchamia to zadanie ❻. W momencie zakończenia zadania powiązana z nim przyszłość przechodzi w stan gotowości.

Umieszczenie zadania w kolejce jest równie proste: nowe, opakowane zadanie jest tworzone na podstawie wskazanej funkcji ❼, przyszłość jest uzyskiwana z obiektu zadania ❽ za pomocą funkcji składowej `get_future()` i wreszcie zadanie jest umieszczane na liście ❾ przed zwróceniem przyszłości do kodu wywołującego ❿. Kod, który wysyłał komunikat do wątku interfejsu użytkownika, może albo poczekać na przyszłość (jeśli wykonanie zadania jest niezbędne do dalszego działania), albo porzucić tę przyszłość (jeśli nie potrzebuje wyniku przetwarzania).

W tym przykładzie użyliśmy do reprezentacji zadań klasy `std::packaged_task` ↪`<void()>`. Klasa opakowuje funkcję (lub inny obiekt wywoływalny), która nie otrzymuje żadnych parametrów i zwraca `void` (jeśli wskazana funkcja zwraca inną wartość, wynik zostanie porzucony). W tym przypadku zastosowano najprostsze możliwe zadanie, jednak (jak już wiemy) szablon klasy `std::packaged_task` może być równie dobrze stosowany w implementacjach bardziej złożonych rozwiązań — wystarczy w roli parametru szablonu użyć innej sygnatury funkcji, zmienić typ zwracanych wartości (a więc także typ danych przechowywanych w ramach stanu przyszłości) i zmienić typy argumentów operatora wywołania funkcji. Przedstawiony przykład można by łatwo rozszerzyć o możliwość przekazywania argumentów do zadań, które mają być wykonywane przez wątek graficznego interfejsu użytkownika, i zwracania wartości w ramach obiektu typu `std::future` (zamiast samego sygnału o zakończeniu zadania).

Co należy zrobić z zadaniami, których nie można wyrazić w formie prostych wywołań funkcji, i zadaniami, których wyniki mogą pochodzić z wielu różnych miejsc? Obsługa takich przypadków wymaga jeszcze innego sposobu tworzenia przyszłości — bezpośredniego ustawiania wartości za pomocą szablonu `std::promise`.

4.2.3. Obietnice (szablon std::promise)

Programiści pracujący nad aplikacjami, które muszą obsługiwać wiele połączeń sieciowych, często ulegają pokusie obsługi każdego połączenia w osobnym wątku, ponieważ takie rozwiązanie ułatwia zrozumienie i zaimplementowanie mechanizmów komunikacji sieciowej. Takie rozwiązanie sprawdza się w przypadku niewielkiej liczby połączeń (a więc także niewielkiej liczby wątków). Okazuje się jednak, że w razie wzrostu liczby połączeń opisany model staje się nieefektywny, ponieważ duża liczba wątków zajmuje zbyt wiele zasobów systemu operacyjnego, a częste przełączanie kontekstu (jeśli liczba wątków przekracza współbieżność sprzętową) ma negatywny wpływ na wydajność aplikacji. W skrajnych przypadkach aplikacja uruchamiająca dużo nowych wątków może wyczerpać zasoby systemu operacyjnego przed osiągnięciem limitu połączeń sieciowych. Właśnie dlatego nawet w aplikacjach obsługujących bardzo dużo połączeń sieciowych stosuje się stosunkowo niewiele wątków (czasem tylko jeden wątek) odpowiedzialnych za obsługę tych połączeń, zatem każdy wątek musi obsługiwać wiele połączeń jednocześnie.

Kup ksi k Pole ksi k

Przeanalizujmy przykład wątku obsługującego połączenia. Pakiety danych przychodzą za pośrednictwem różnych połączeń w przypadkowej kolejności; podobnie pakiety danych przeznaczone do wysłania są kolejkowane w przypadkowej kolejności. W wielu przypadkach pozostałe elementy aplikacji będą oczekiwały albo na wysłanie danych, albo na otrzymanie nowego pakietu danych za pośrednictwem określonego połączenia sieciowego.

Szablon klasy `std::promise<T>` umożliwia ustawienie wartości (typu `T`), którą w przyszłości będzie można odczytać za pośrednictwem powiązanego obiektu klasy `std::future<T>`. Para klas `std::promise` i `std::future` to jeden z mechanizmów umożliwiających implementację interesującego nas rozwiązania — wątek oczekujący może wstrzymać działanie w oczekiwaniu na przyszłość, natomiast wątek udostępniający dane może użyć obiektu obietnicy do ustawienia powiązanej wartości, tak aby odpowiednia przyszłość przeszła w stan **gotowości**.

Obiekt klasy `std::future` powiązany z danym obiektem klasy `std::promise` można uzyskać za pomocą funkcji składowej `get_future()`, a więc tak samo jak w przypadku obiektu klasy `std::packaged_task`. W momencie ustawienia wartości obiektu obietnicy (za pomocą funkcji składowej `set_value()`) obiekt przyszłości przechodzi w stan **gotowości** i jako taki może zostać użyty do pobrania zapisanej wartości. Jeśli nastąpi zniszczenie obiektu klasy `std::promise` bez wcześniejszego ustawienia wartości, zamiast oczekiwanej wartości zostanie ustawiony stosowny wyjątek. Sposób przekazywania wyjątków pomiędzy wątkami zostanie opisany w punkcie 4.2.4.

Na listingu 4.10 pokazano przykład kodu wątku, który przetwarza połączenia w opisany powyżej sposób. W prezentowanym przykładzie użyliśmy pary klas `std::promise<bool>` i `std::future<bool>` do identyfikacji udanej transmisji bloku danych wychodzących; wartość powiązana z obiektem przyszłości ma postać prostej flagi sukcesu lub niepowodzenia. W przypadku pakietów przychodzących funkcję danych powiązanych z obiektem przyszłości pełni właściwa treść tych pakietów.

Listing 4.10. Obsługa wielu połączeń w jednym wątku przy użyciu obiektów obietnic

```
#include <future>

void process_connections(connection_set& connections)
{
    while(!done(connections))          ←❶
    {
        for(connection_iterator          ←❷
                connection=connections.begin(),end=connections.end();
            connection!=end;
            ++connection)
        {
            if(connection->has_incoming_data())          ←❸
            {
                data_packet data=connection->incoming();
                std::promise<payload_type>& p=
                    connection->get_promise(data.id);          ←❹
                p.set_value(data.payload);
            }
```

Kup książkę Poleć książkę

```
            if(connection->has_outgoing_data())      ←❺
            {
                outgoing_packet data=
                    connection->top_of_outgoing_queue();
                connection->send(data.payload);
                data.promise.set_value(true);      ←❻
            }
        }
    }
}
```

Funkcja `process_connections()` wykonuje pętlę do momentu, w którym funkcja `done()` zwróci wartość `true` ❶. W każdej iteracji tej pętli kod aplikacji sprawdza kolejno każde połączenie ❷ i pobiera dane przychodzące (jeśli istnieją) ❸ lub wysyła kolejkowane dane wychodzące ❺. Zakładamy, że pakiet przychodzący zawiera jakiś identyfikator i właściwe dane. Identyfikator jest odwzorowywany na odpowiedni obiekt klasy `std::promise` (na przykład metodą odnajdywania w kontenerze asocjacyjnym) ❹, natomiast wartość jest przypisywana do ciała pakietu. W przypadku pakietów wychodzących zastosowano mechanizm kolejki pakietów oczekujących na wysłanie — program sprawdza stan kolejki i wysyła ewentualne pakiety dla danego połączenia. Po wysłaniu pakietu w obiekcie obietnicy powiązanym z tymi danymi wychodzącymi jest ustawiana wartość `true`, która oznacza pomyślną transmisję danych ❻. Zgodność opisanego modelu z rzeczywistymi protokołami komunikacji sieciowej zależy tylko od tych protokołów. Struktura na bazie obietnicy i przyszłości nie pasuje co prawda do każdego scenariusza, ale pod wieloma względami przypomina model asynchronicznych operacji wejścia-wyjścia stosowany w niektórych systemach operacyjnych.

W dotychczas prezentowanym kodzie całkowicie ignorowaliśmy problem wyjątków. Wyobrażenie świata, w którym wszystko działa, jak należy, jest być może kuszące, ale nie ma wiele wspólnego z rzeczywistością. Nie można wykluczyć, że dysk zostanie zapełniony, że program nie będzie mógł znaleźć potrzebnych danych, że nastąpi awaria połączenia sieciowego lub że w wyniku błędu nie będzie dostępna baza danych. Jeśli operacja wykonywana w jednym wątku potrzebuje do działania wyniku innego wątku, warto uwzględnić możliwość zasygnalizowania błędu w formie wyjątku — zakładanie, że w kodzie stosującym obiekty klasy `std::packaged_task` czy `std::promise` wszystko zawsze będzie działało prawidłowo, byłoby zbyt optymistyczne. Biblioteka standardowa języka C++ oferuje wygodne mechanizmy obsługi wyjątków w tego rodzaju scenariuszach i umożliwia zapisywanie wyjątków w ramach wyników powiązanych z tymi obiektami.

4.2.4. Zapisywanie wyjątku na potrzeby przyszłości

Przeanalizujmy następujący fragment kodu. Jeśli na wejściu funkcji `square_root()` przekażemy wartość `-1`, zgłoszony zostanie wyjątek (to on trafi do kodu wywołującego tę funkcję):

```
double square_root(double x)
{
    if(x<0)
    {
```

Kup ksi k Pole ksi k

```
        throw std::out_of_range("x<0");
    }
    return sqrt(x);
}
```

Przypuśćmy teraz, że zamiast wywołać funkcję `square_root()` w bieżącym wątku, jak w poniższym wierszu:

```
double y=square_root(-1);
```

użyjemy wywołania asynchronicznego w następującej formie:

```
std::future<double> f=std::async(square_root,-1);
double y=f.get();
```

Idealnym rozwiązaniem byłoby zapewnienie dokładnie takiego samego zachowania jak w przypadku kodu jednowątkowego — skoro zmiennej `y` w obu przypadkach jest przypisywany wynik funkcji, wątek wywołujący funkcję `f.get()` powinien mieć dostęp także do ewentualnych wyjątków (tak jak odpowiedni kod jednowątkowy).

Okazuje się, że właśnie tak działa prezentowane rozwiązanie: jeśli funkcja `square_root` wywołana za pośrednictwem funkcji `std::async` zgłosi jakiś wyjątek, wyjątek ten zostanie zapisany w obiekcie przyszłości (w miejscu dla wartości wynikowej), przyszłość przejdzie w stan **gotowości**, a funkcja `get()` spowoduje ponowne zgłoszenie zapisanego wyjątku. (Uwaga: standard języka C++ nie określa, czy ponowne zgłoszenie dotyczy oryginalnego obiektu wyjątku, czy jego kopii; różne kompilatory i biblioteki stosują w tym względzie odmienne rozwiązania). To samo dotyczy funkcji opakowanej w ramach obiektu klasy `std::packaged_task` — jeśli po wywołaniu zadania opakowana funkcja zgłosi jakiś wyjątek, wyjątek jest zapisywany w obiekcie przyszłości zamiast właściwego wyniku. Aby ponownie zgłosić ten wyjątek, wystarczy wywołać funkcję `get()`.

Szablon klasy `std::promise` oferuje oczywiście analogiczne rozwiązanie, które wymaga bezpośredniego wywołania funkcji. Aby zapisać wyjątek zamiast wartości wynikowej, wystarczy wywołać funkcję składową `set_exception()` zamiast funkcji `set_value()`. Wspomniana funkcja jest zwykle stosowana w bloku `catch` odpowiedzialnym za obsługę wyjątku zgłoszonego w trakcie działania algorytmu — wyjątek jest umieszczany w obiekcie obietnicy:

```
extern std::promise<double> some_promise;

try
{
    some_promise.set_value(calculate_value());
}
catch(...)
{
    some_promise.set_exception(std::current_exception());
}
```

W powyższym kodzie użyto funkcji `std::current_exception()` do pobrania zgłoszonego wyjątku; alternatywnym rozwiązaniem byłoby wywołanie funkcji `std::copy_exception()` w celu zapisania nowego wyjątku bez jego bezpośredniego zgłaszania:

```
some_promise.set_exception(std::make_exception_ptr(std::logic_error("foo ")));
```

Opisane rozwiązanie jest nieporównanie bardziej czytelne niż stosowanie bloku `try-catch`, jeśli tylko potencjalny wyjątek jest znany z wyprzedzeniem, i powinno być preferowane. W ten sposób można nie tylko uprościć kod, ale też ułatwić optymalizację tego kodu przez kompilator.

Innym sposobem zapisywania wyjątku w przyszłości jest zniszczenie obiektu klasy `std::promise` lub `std::packaged_task` powiązanego z obiektem przyszłości bez uprzedniego wywołania funkcji ustawiającej (w przypadku obiektu obietnicy) lub uruchomienia opakowanego zadania. Jeśli obiekt przyszłości nie będzie **gotowy**, w obu przypadkach destruktor klasy `std::promise` lub `std::packaged_task` zapisze w powiązanym stanie wyjątek typu `std::future_error` z kodem błędu `std::future_errc::broken_promise`. Tworząc przyszłość, zapowiadamy (składamy obietnicę), że udostępnimy jakąś wartość lub jakiś wyjątek; zniszczenie źródła tej wartości lub tego wyjątku bez uprzedniego dostarczenia zapowiedzianego zasobu łamie tę obietnicę. Gdyby w opisanym przypadku kompilator niczego nie zapisał w obiekcie przyszłości, wątki oczekujące mogłyby czekać w nieskończoność.

Do tej pory we wszystkich przykładach stosowałem szablon klasy `std::future`. Warto jednak pamiętać o pewnych ograniczeniach szablonu `std::future`, w tym o możliwości oczekiwania na wynik przez zaledwie jeden wątek. W razie konieczności zaimplementowania modelu, w którym na jedno zdarzenie będzie oczekiwało wiele wątków, należy użyć raczej szablonu klasy `std::shared_future`.

4.2.5. Oczekiwanie na wiele wątków

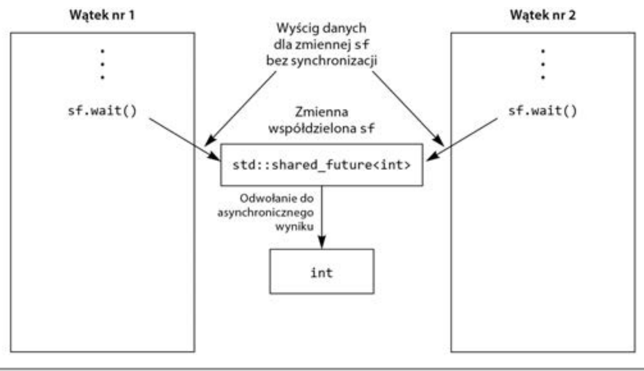
Mimo że szablon klasy `std::future` obsługuje wszystkie mechanizmy synchronizacji potrzebne do przesyłania danych pomiędzy wątkami, wywołania funkcji składowych określonego obiektu klasy `std::future` nie są synchronizowane z wywołaniami funkcji pozostałych obiektów tej klasy. Jeśli wiele wątków uzyskuje dostęp do jednego obiektu klasy `std::future` bez stosowania dodatkowych mechanizmów synchronizacji, aplikacja jest narażona na **wyścig danych** i niezdefiniowane zachowania. Problem wynika z projektu tego rozwiązania — szablon klasy `std::future` modeluje unikatową własność asynchronicznego wyniku, a jednorazowy charakter funkcji `get()` i tak wyklucza sens współbieżnego dostępu. Skoro po pierwszym wywołaniu funkcji `get()` nie można już pobrać żadnych danych, z natury rzeczy dane mogą być pobrane tylko przez jeden wątek.

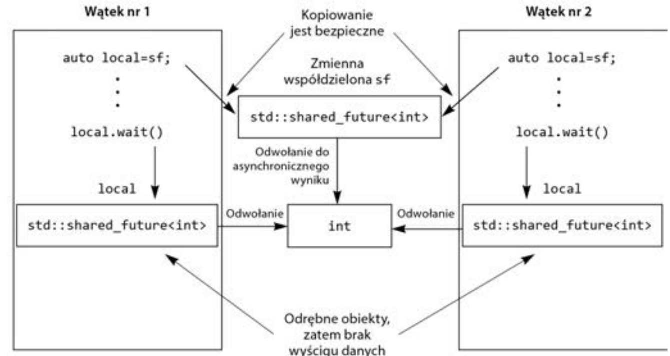
Jeśli jednak projekt naszej aplikacji współbieżnej wymaga, aby wiele wątków mogło czekać na to samo zdarzenie, nie wszystko stracone — wystarczy użyć szablonu klasy `std::shared_future`. O ile szablon klasy `std::future` oferuje tylko możliwość **przenoszenia**, zatem własność przyszłości można przenosić pomiędzy różnymi obiektami, ale tylko jeden obiekt może się jednocześnie odwoływać do jednego asynchronicznego wyniku, o tyle szablon klasy `std::shared_future` oferuje możliwość **kopiowania**, zatem może istnieć wiele obiektów odwołujących się do tego samego stanu.

W przypadku szablonu `std::shared_future` funkcje składowe wywoływane dla pojedynczego obiektu wciąż nie są synchronizowane, zatem warunkiem unikania wyścigów danych w związku z dostępem do tego samego obiektu z poziomu wielu wątków jest ochrona tego dostępu za pomocą blokady. Najlepszym sposobem jest kopiowanie tego obiektu, tak aby każdy wątek uzyskiwał bezpieczny dostęp do własnej kopii, ponieważ wewnętrzne dane są poprawnie synchronizowane przez bibliotekę. Dostęp do współ-

Kup ksi k Pole ksi k

dzielonego, asynchronicznego stanu z poziomu wielu wątków jest bezpieczny, jeśli tylko każdy z tych wątków uzyskuje dostęp do stanu za pośrednictwem własnego obiektu klasy `std::shared_future`. Przykład takiego rozwiązania pokazano na rysunku 4.1.





Rysunek 4.1. Użycie wielu obiektów klasy std::shared_future w celu uniknięcia wyścigów danych

Jednym z możliwych zastosowań szablonu klasy `std::shared_future` jest implementacja równoległego wykonywania jakiejś operacji w modelu zbliżonym do złożonego arkusza kalkulacyjnego, gdzie każda komórka zawiera wartość, która może być używana we wzorach w wielu pozostałych komórkach. Wzory potrzebne do obliczania wyników w komórkach zależnych mogą używać obiektu klasy `std::shared_future` podczas odwoływania się do pierwszej komórki. Jeśli wzory we wszystkich komórkach będą przetwarzane równolegle, zadania odwołujące się do gotowych wartości zostaną zrealizowane natychmiast, natomiast zadania zależne od innych, jeszcze przetwarzanych komórek będą musiały poczekać na osiągnięcie gotowości przez tamte komórki. Takie rozwiązanie umożliwia maksymalne wykorzystanie dostępnej współbieżności sprzętowej.

Obiekty klasy `std::shared_future`, które wskazują pewien asynchroniczny stan, są konstruowane na podstawie obiektów klasy `std::future` odwołujących się do tego stanu. Ponieważ obiekty klasy `std::future` nie współdzielą własności tego asynchronicznego stanu z żadnymi innymi obiektami, własność należy przenieść do obiektu klasy `std::` ↪`shared_future` za pomocą funkcji `std::move`, pozostawiając oryginalny obiekt klasy `std::future` z pustym stanem (jak w przypadku użycia konstruktora domyślnego):

```
std::promise<int> p;
std::future<int> f(p.get_future());
assert(f.valid());                    ← ❶ Przyszłość f jest prawidłowa
std::shared_future<int> sf(std::move(f));
assert(!f.valid());                   ← ❷ Przyszłość f już nie jest prawidłowa
assert(sf.valid());                   ← ❸ Przyszłość sf jest teraz prawidłowa
```

Obiekt przyszłości `f` jest początkowo prawidłowy ❶, ponieważ odwołuje się do asynchronicznego stanu obietnicy `p`, jednak po przeniesieniu tego stanu do obiektu `sf` to obiekt `sf` jest prawidłowy ❷, natomiast obiekt `f` jest już nieprawidłowy ❸.

Jak w przypadku wszystkich obiektów z możliwością przenoszenia, przeniesienie własności jest wykonywane automatycznie dla r-wartości, zatem możemy skonstruować obiekt klasy `std::shared_future` bezpośrednio na podstawie wartości zwróconej przez funkcję składową `get_future()` obiektu klasy `std::promise`:

```
std::promise<std::string> p;
std::shared_future<std::string> sf(p.get_future());    ← ❶ Automatyczne przeniesienie własności
```

W powyższym kodzie własność jest przenoszona automatycznie — obiekt klasy `std::↪shared_future<>` jest konstruowany na podstawie r-wartości typu `std::future<std::↪string>` ❶.

Szablon klasy `std::future` oferuje jeszcze inne rozwiązanie ułatwiające stosowanie obiektów klasy `std::shared_future` przy użyciu nowego mechanizmu automatycznego określania typu zmiennej na podstawie inicjalizatora (patrz część A.6 dodatku A). Szablon klasy `std::future` definiuje funkcję składową `share()`, która tworzy nowy obiekt klasy `std::shared_future` i bezpośrednio przenosi własność do tego obiektu. Użycie tego rozwiązania może nam oszczędzić sporo pisania i znacznie ułatwia modyfikowanie kodu:

```
std::promise< std::map< SomeIndexType, SomeDataType, SomeComparator,
    SomeAllocator>::iterator> p;
auto sf=p.get_future().share();
```

W tym przypadku typ zmiennej `sf` jest identyfikowany jako `std::shared_future< std::map<SomeIndexType, SomeDataType, SomeComparator, SomeAllocator>::iterator>`, czyli konstrukcja, której wielokrotne stosowanie w kodzie byłoby dość kłopotliwe. W razie zmiany komparatora lub alokatora wystarczy zmodyfikować typ obiektu obietnicy; typ obiektu przyszłości zostanie automatycznie zaktualizowany i dostosowany do tej zmiany.

W pewnych przypadkach dobrym rozwiązaniem jest ograniczanie maksymalnego czasu oczekiwania na zdarzenie (z uwagi na ograniczony czas działania określonej sekcji kodu lub ze względu na istnienie innych ważnych zadań, którymi dany wątek może się zająć, jeśli oczekiwane zdarzenie nie wystąpi odpowiednio wcześnie). Z myślą o takich przypadkach wiele funkcji oczekiwania oferuje wersje z możliwością określenia limitu czasowego.

Kup ksi k Pole ksi k

4.3. Oczekiwanie z limitem czasowym

Wszystkie wywołania blokujące, które stosowaliśmy w dotychczasowych przykładach, blokowały wykonywanie wątków przez nieokreślony czas, tj. do momentu wystąpienia oczekiwanego zdarzenia. W wielu przypadkach takie rozwiązanie jest wystarczające, jednak w niektórych sytuacjach lepszym wyjściem jest określenie maksymalnego czasu oczekiwania. Stosowanie takich limitów czasowych może mieć na celu potwierdzenie prawidłowego działania aplikacji (w formie komunikatu dla użytkownika lub innego procesu) lub przerwanie oczekiwania, jeśli na przykład użytkownik kliknął przycisk *Anuluj*.

Istnieją dwa rodzaje limitów czasowych stosowanych dla operacji blokujących: limity określające maksymalny **czas blokowania** wątku (na przykład 30 milisekund) oraz limity **bezwzględne**, gdzie oczekiwanie nie może trwać dłużej niż do określonego punktu w czasie (na przykład do godziny 17:30:15.045987023 dnia 30 listopada 2012 roku). Większość funkcji oczekujących występuje w wersjach obsługujących obie formy limitów czasowych. Wersje obsługujące względne limity czasowe (określające czas trwania operacji) są oznaczane przyrostkiem `_for`, natomiast bezwzględne limity czasowe oznacza się przyrostkiem `_until`.

Na przykład klasa `std::condition_variable` definiuje dwie przeciążone wersje funkcji składowej `wait_for()` i dwie przeciążone wersje funkcji składowej `wait_until()`, czyli odpowiedniki obu wersji funkcji `wait()` uzupełnione o obsługę względnych i bezwzględnych limitów czasowych — pierwsza wersja czeka na sygnał, upłynięcie limitu czasowego lub bezpośrednie budzenie; druga wersja w momencie budzenia sprawdza przekazany predykat i zwraca sterowanie, pod warunkiem że albo ten predykat jest prawdziwy (w wyniku sygnału umieszczonego w zmiennej warunkowej), albo osiągnięto limit czasowy.

Zanim przeanalizujemy szczegółowe aspekty stosowania funkcji uwzględniających limity czasowe, warto poświęcić chwilę na omówienie sposobu określania czasu w języku C++, w tym dostępnych zegarów.

4.3.1. Zegary

W kontekście elementów biblioteki standardowej języka C++ zegar jest dla programu źródłem informacji o bieżącej godzinie. W szczególności zegar jest klasą udostępniającą cztery odrębne informacje:

- **bieżąca** godzina;
- typ wartości używanych do reprezentowania godzin uzyskiwanych z obiektu zegara;
- okres reprezentowany przez jeden takt zegara;
- to, czy takty zegara mają stałą długość, czyli możliwość traktowania zegara jako **stabilnego**.

Bieżącą godzinę reprezentowaną przez zegar można uzyskać, wywołując statyczną funkcję składową `now()` klasy zegara; na przykład funkcja `std::chrono::system_clock::now()` zwróci bieżącą godzinę reprezentowaną przez zegar systemowy. Typ punktów w czasie dla poszczególnych zegarów jest reprezentowany przez składową definicję typu `time_point`, zatem każda funkcja `zegar::now()` zwraca wartość typu `zegar::time_point`.

Okres taktu zegara jest wyrażany w formie ułamka sekundy reprezentowanego przez składową definicję typu `period` — w przypadku zegara wykonującego 25 taktów w ciągu sekundy `period` definiuje typ `std::ratio<1,25>`, natomiast w przypadku zegara wykonującego jeden takt na 2,5 sekundy składowa `period` definiuje typ `std::ratio<5,2>`. Jeśli określenie okresu taktu nie jest możliwe do momentu uruchomienia programu lub jeśli ten okres może się zmieniać w czasie działania aplikacji, okres można zdefiniować w formie średniego czasu trwania taktu, najkrótszego możliwego taktu lub innej wartości przewidzianej przez twórców biblioteki. Nie można jednak zakładać, że obserwowany okres taktu zegara podczas jednej próby uruchomienia programu będzie odpowiadał rzeczywistemu okresowi danego zegara.

Jeśli **takty zegara mają stałą częstotliwość** (niezależnie od tego, czy ta częstotliwość pasuje do przyjętego okresu) i jeśli **nie możemy zmienić długości taktu**, mamy do czynienia z tzw. **stabilnym zegarem** (ang. *steady clock*). Składowa statyczna `is_steady` klasy stabilnego zegara ma wartość `true` (w przypadku niestabilnego zegara ta sama składowa ma wartość `false`). Zegar systemowy (reprezentowany przez klasę `std::chrono::system_clock`) zwykle **nie** jest stabilny, ponieważ można dostosowywać jego częstotliwość (nawet jeśli zmiany częstotliwości są wprowadzane automatycznie z uwzględnieniem przesunięć względem zegara lokalnego). Każda taka zmiana może spowodować, że wywołanie funkcji `now()` zwróci wartość wcześniejszą niż zwrócona przez poprzednie wywołanie tej funkcji, co oczywiście narusza wymaganie stałej częstotliwości zegara (i długości taktu). Jak się za chwilę przekonasz, stabilne zegary są szczególnie przydatne podczas obliczeń z uwzględnieniem limitów czasowych — z myślą o tych i podobnych zastosowaniach twórcy biblioteki standardowej udostępnili taki zegar w formie klasy `std::chrono::steady_clock`. Biblioteka standardowa języka C++ zawiera też inne klasy zegarów: wspomnianą wcześniej klasę `std::chrono::system_clock`, która reprezentuje zegar „czasu rzeczywistego” w danym systemie i która udostępnia funkcję konwersji punktów w czasie na i z wartości typu `time_t`, oraz klasę `std::chrono::high_resolution_clock`, która zapewnia najkrótszy możliwy takt zegara (a więc także najwyższą możliwą częstotliwość) spośród wszystkich zegarów tej biblioteki. Drugi z zegarów można wykorzystać w roli punktu wyjścia dla definicji alternatywnych rozwiązań. Wymienione zegary zdefiniowano (wraz z pozostałymi elementami związanymi z obsługą czasu) w pliku nagłówkowym `<chrono>`.

Zanim przystąpimy do omawiania metod reprezentowania punktów w czasie, warto poświęcić chwilę analizie technik reprezentowania okresów.

4.3.2. Okresy

Okres (czas trwania) to najprostszy aspekt obsługi czasu. Okres zaimplementowano w szablonie klasy `std::chrono::duration<>` (wszystkie elementy języka C++ związane z obsługą czasu i używane przez bibliotekę wątków należą do przestrzeni nazw `std::chrono`). Pierwszy parametr tego szablonu określa typ reprezentacji (na przykład `int`, `long` lub `double`); drugi parametr jest ułamkiem określającym liczbę sekund reprezentowanych przez jednostkę okresu. Na przykład liczba minut przechowywana w wartości typu `short` jest reprezentowana przez klasę `std::chrono::duration<short,std::ratio<60,1>>`, ponieważ minuta składa się z 60 sekund. Liczba milisekund przechowywanych w wartości typu `double` jest reprezentowana przez klasę `std::chrono::duration<double,std::ratio<1,1000>>`, ponieważ każda milisekunda trwa jedną tysięczną część sekundy.

Kup ksi k Pole ksi k

Biblioteka standardowa oferuje zbiór predefiniowanych definicji typów w przestrzeni nazw std::chrono dla różnych okresów (wyrażanych w nanosekundach, mikrosekundach, milisekundach, sekundach, minutach i godzinach). Wszystkie te definicje stosują na tyle elastyczne typy całkowitoliczbowe, że mogą reprezentować na przykład okresy ponad 500-letnie w wybranych jednostkach czasu. Przestrzeń nazw zawiera także definicje typów dla rzędów wielkości układu SI: od std::atto (10^{-18}) do std::exa (10^{18}) (i więcej, jeśli tylko dana platforma obsługuje 128-bitowe typy całkowitoliczbowe). Za pomocą tych typów można operować na niestandardowych okresach, na przykład klasa std::duration<double,std::centi> obsługuje liczbę setnych części sekundy reprezentowanych przez liczbę typu double.

Dla wygody programisty w standardzie C++14 wprowadzono w przestrzeni nazw std::chrono_literals pewną liczbę predefiniowanych literałów sufiksów dla operacji związanych z okresami czasu. Te prefiksy mogą uprościć tworzenie kodu używającego na stałe zdefiniowanych wartości, które określają okresy czasu. Spójrz na przedstawiony tutaj przykład.

```
using namespace std::chrono_literals;
auto one_day=24h;
auto half_an_hour=30min;
auto max_time_between_messages=30ms;
```

W przypadku użycia wraz z literałami liczb całkowitych te sufiksy są odpowiednikami używania predefiniowanych typów danych dla okresu czasu. Dlatego też 15ns i std::↪chrono::nanoseconds(15) to wartości identyczne. Natomiast w połączeniu z literałami w postaci liczb zmiennoprzecinkowych te sufiksy tworzą odpowiednio przeskalowaną wartość zmiennoprzecinkową nieokreślonego typu. Dlatego też 2.5min odpowiada std::↪chrono::duration<*dowolny-typ-liczb-zmiennoprzecinkowych*,std::ratio<60,1>>. Jeżeli masz obawy związane z zakresem lub dokładnością implementacji wybranego typu liczb zmiennoprzecinkowych, wówczas samodzielnie musisz przygotować obiekt wraz z odpowiednią reprezentacją, zamiast korzystać z wygody, jaką zapewniają literały sufiksów.

Konwersja pomiędzy okresami jest wykonywana automatycznie, pod warunkiem że nie wymaga obcięcia wartości źródłowej — oznacza to, że konwersja godzin na sekundy jest możliwa, ale już konwersja sekund na godziny nie zostanie wykonana automatycznie. Konwersję można też wykonać jawnie za pomocą funkcji std::chrono::↪duration_cast<>:

```
std::chrono::milliseconds ms(54802);
std::chrono::seconds s=
    std::chrono::duration_cast<std::chrono::seconds>(ms);
```

Ponieważ wynik jest obcinany (nie zaokrąglany), zmienna s będzie zawierała wartość 54.

Okresy obsługują działania arytmetyczne, zatem możemy dodawać i odejmować okresy, aby otrzymywać nowe okresy, bądź mnożyć lub dzielić okresy przez stałe wybranego typu danych (czyli pierwszego parametru szablonu klasy). Oznacza to, że wyrażenie 5*seconds(1) ma taką samą wartość jak wyrażenia seconds(5) i minutes(1) - seconds(55). Liczbę jednostek składających się na dany okres można uzyskać za pomocą funkcji składowej count(). Oznacza to, że wywołanie std::chrono::milliseconds(1234).count() zwróci wartość 1234.

Kup ksi k Pole ksi k

Wymuszanie oczekiwania na podstawie okresu (czasu trwania) wymaga stosowania egzemplarza typu `std::chrono::duration<>`. Możemy na przykład spowodować, że czas oczekiwania na gotowość obiektu przyszłości wyniesie 35 milisekund:

```
std::future<int> f=std::async(some_task);
if(f.wait_for(std::chrono::milliseconds(35))==std::future_status::ready)
    do_something_with(f.get());
```

Wszystkie funkcje oczekiwania zwracają status określający, czy koniec oczekiwania wynika z wyczerpania limitu czasowego, czy z wystąpienia zdarzenia, na które czekał dany wątek. W tym przypadku wątek czeka na przyszłość, zatem funkcja zwraca wartość `std::future_status::timeout` w przypadku wyczerpania limitu czasowego; wartość `std::future_status::ready`, jeśli przyszłość jest gotowa; lub wartość `std::future_↪status::deferred`, jeśli zadanie przyszłości zostało odłożone na później. Czas oczekiwania okresowego jest mierzony przy użyciu stabilnego, wewnętrznego zegara biblioteki, zatem 35 milisekund oznacza właśnie 35 milisekund, nawet jeśli w czasie oczekiwania zegar systemowy zostanie przestawiony (w przód lub w tył). Nie można oczywiście zapominać o kaprysach systemu szeregowania zadań i o zróżnicowanej precyzji zegarów systemów operacyjnych, które mogą spowodować, że rzeczywisty czas dzielący wywołanie funkcji `wait()` od zwrócenia sterowania będzie dużo dłuższy niż 35 ms.

Skoro potrafimy już stosować okresy, możemy przejść do analizy modelu punktów w czasie.

4.3.3. Punkty w czasie

Punkt w czasie jest reprezentowany przez egzemplarz szablonu klasy `std::chrono::↪time_point<>`. Pierwszy parametr tego szablonu wskazuje zegar, natomiast drugi parametr określa jednostki miary (w tej roli należy użyć specjalizacji szablonu klasy `std::chrono::duration<>`). Wartość punktu w czasie reprezentuje czas (w formie wielokrotności wskazanego okresu) od konkretnego punktu w czasie nazywanego **epoką** zegara. Epoka zegara jest prostą właściwością, która jednak nie jest bezpośrednio dostępna ani wprost definiowana przez standard języka C++. Do najczęściej stosowanych epok należy północ 1 stycznia 1970 roku i moment uruchomienia komputera, na którym działa dana aplikacja. Zegary mogą stosować jedną epokę lub różne, niezależne epoki. Jeśli dwa zegary stosują tę samą epokę, definicja typu `time_point` w klasie jednego zegara może wskazywać drugą klasę jako typ zegara powiązanego z daną definicją `time_point`. Mimo że nie można bezpośrednio uzyskać epoki, **mamy do dyspozycji** funkcję `time_↪since_epoch()`, którą możemy wywołać dla danego egzemplarza typu `time_point`. Funkcja składowa `time_since_epoch()` zwraca wartość okresu reprezentującą czas od epoki zegara do określonego punktu w czasie.

Punkt w czasie można zdefiniować na przykład w formie obiektu klasy `std::chrono::↪time_point<std::chrono::system_clock, std::chrono::minutes>`. Tak skonstruowany obiekt zawiera czas względem zegara systemowego, tyle że mierzony w minutach (nie według standardowej precyzji zegara systemowego, czyli sekund lub części sekundy).

Na obiektach klasy `std::chrono::time_point<>` można wykonywać operacje dodawania i odejmowania okresów, których wynikiem są nowe punkty w czasie. Oznacza to, że na przykład w wyniku dodawania `std::chrono::high_resolution_clock::now() + std::chrono::nanoseconds(500)` otrzymamy punkt w czasie, który nastąpi za 500 nanosekund (licząc od teraz). Wyrażenia tego typu są przydatne podczas obliczania

Kup ksi k Pole ksi k

bezwzględnego limitu czasowego w sytuacji, gdy maksymalny czas wykonywania bloku kodu jest znany z wyprzedzeniem. Takie rozwiązanie wymaga jednak wielu wywołań funkcji oczekujących lub funkcji poprzedzających funkcję oczekującą i zaliczanych do bloku, który podlega ograniczeniu czasowemu.

Istnieje też możliwość odjęcia jednego punktu w czasie od innego, pod warunkiem że oba punkty bazują na tym samym zegarze. Wynikiem tej operacji jest okres, który reprezentuje czas dzielący oba punkty. W ten sposób można na przykład sprawdzać czas wykonywania bloków kodu:

```
auto start=std::chrono::high_resolution_clock::now();
do_something();
auto stop=std::chrono::high_resolution_clock::now();
std::cout<<"Wykonanie funkcji do_something() zajęło "
  <<std::chrono::duration<double,std::chrono::seconds>(stop-start).count()
  <<" sekund."<<std::endl;
```

Parametr zegara szablonu klasy `std::chrono::time_point<>` decyduje nie tylko o epoce. W przypadku przekazania punktu w czasie na wejściu funkcji oczekującej, która stosuje bezwzględny limit czasowy, wybrany zegar będzie używany do mierzenia czasu. Warto pamiętać o możliwości zmiany wskazań zegara i o tym, że funkcja oczekująca nie zwróci sterowania do momentu, w którym funkcja `now()` tego zegara nie zwróci wartości późniejszej od wskazanego limitu czasowego. Jeśli zegar zostanie przestawiony w przód, łączny czas oczekiwania może być krótszy (w porównaniu z czasem mierzonym przez stabilny zegar); a jeśli zegar zostanie cofnięty, łączny czas oczekiwania zostanie wydłużony.

Jak nietrudno odgadnąć, punkty w czasie są używane w wersjach funkcji oczekujących z przyrostkiem `_until`. Typowym zastosowaniem tego rozwiązania jest wyznaczanie przesunięcia względem godziny zwracanej przez wywołanie *jakiś-zegar*`::now()` w wybranym punkcie programu. Punkty powiązane z zegarem systemowym można uzyskiwać, konwertując egzemplarze typu `time_t` za pomocą statycznej funkcji składowej `std::chrono::system_clock::to_time_point()`. Jeśli na przykład maksymalny czas oczekiwania na zdarzenie powiązane ze zmienną warunkową wynosi 500 milisekund, można zastosować konstrukcję podobną do tej z listingu 4.11.

Listing 4.11. Oczekiwanie na zmienną warunkową z uwzględnieniem limitu czasowego

```
#include <condition_variable>
#include <mutex>
#include <chrono>

std::condition_variable cv;
bool done;
std::mutex m;

bool wait_loop()
{
    auto const timeout= std::chrono::steady_clock::now()+
        std::chrono::milliseconds(500);
    std::unique_lock<std::mutex> lk(m);
    while(!done)
    {
```

Kup ksi k Pole ksi k

```
        if(cv.wait_until(lk,timeout)==std::cv_status::timeout)
            break;
    }
    return done;
}
```

Rozwiązanie z listingu 4.11 jest zalecanym sposobem oczekiwania na zmienne warunkowe z uwzględnieniem limitu czasowego w sytuacji, gdy funkcja oczekująca nie otrzymuje na wejściu żadnego predykatu. Maksymalny czas wykonywania pętli jest ograniczony. Jak napisałem w punkcie 4.1.1, jeśli nie stosujemy dodatkowego predykatu, operowanie na zmiennych warunkowych wymaga użycia pętli, która obsługuje nietypowe sposoby budzenia wątku. W przypadku wywołania funkcji `wait_for()` w ciele pętli może się zdarzyć, że warunek wznowienia działania zostanie spełniony bezpośrednio przed upływem limitu czasowego, a w następnej iteracji czas oczekiwania będzie liczony od początku. Taka sytuacja może mieć miejsce wielokrotnie, zatem łączny czas oczekiwania byłby nieograniczony.

Skoro dysponujemy już podstawowymi narzędziami umożliwiającymi stosowanie limitów czasowych, pora omówić funkcje, w których można używać tych limitów.

4.3.4. Funkcje otrzymujące limity czasowe

Najprostszym zastosowaniem limitu czasowego jest dodanie opóźnienia do przetwarzania określonego wątku, tak aby ten wątek nie zajmował czasu procesora (potrzebnego innym wątkom) w czasie, gdy nie wykonuje żadnych wartościowych zadań. Przykład takiego rozwiązania opisałem w podrozdziale 4.1, gdzie kod wielokrotnie sprawdzał stan flagi gotowości w pętli. Do opóźniania działania (usypiania) wątków służą funkcje `std::this_thread::sleep_ for()` i `std::this_thread::sleep_until()`. Obie funkcje działają jak proste budziki — wątek jest usypiany albo na określony okres (za pomocą funkcji `sleep_for()`), albo do wskazanego punktu w czasie (za pomocą funkcji `sleep_ ↪until()`). W przykładowych rozwiązaniach z podrozdziału 4.1 należałoby użyć funkcji `sleep_for()`, ponieważ w przypadku okresowo wykonywanych operacji wątek powinien być wstrzymywany na pewien czas (nie do określonego punktu w czasie). Funkcja `sleep_until()` umożliwia planowanie budzenia wątku w określonym punkcie w czasie. Funkcja `sleep_until()` może więc służyć na przykład do uruchamiania procedury tworzenia kopii zapasowej o północy, drukowania listy płac o 6 rano lub wstrzymywania wątku do momentu wyświetlenia następnej klatki podczas odtwarzania zapisu wideo.

Usypianie wątku to oczywiście nie jedyne zastosowanie limitów czasowych — jak już wspomniałem, limity tego typu można stosować łącznie ze zmiennymi warunkowymi i przyszłościami. Istnieje nawet możliwość stosowania limitów czasowych podczas prób uzyskania blokady muteksu, jeśli tylko użyty muteks obsługuje takie działanie. Standardowe muteksy typów `std::mutex` i `std::recursive_mutex` nie obsługują limitów czasowych dla operacji blokowania, ale już muteksy typów `std::timed_mutex` i `std::recursive_timed_mutex` dopuszczają takie działanie. Oba te typy udostępniają funkcje składowe `try_lock_for()` i `try_lock_until()`, które próbują uzyskać blokadę muteksu odpowiednio w określonym czasie lub przed osiągnięciem określonego punktu w czasie. W tabeli 4.1 opisano funkcje biblioteki standardowej języka C++, które obsługują limity czasowe, wraz z ich parametrami i typami zwracanych wartości. W miejsce parametru opisanego jako *`okres`* należy przekazać obiekt klasy `std::duration<>`, natomiast parametr *`punkt_w_czasie`* należy zastąpić obiektem klasy `std::time_point<>`.

Kup ksi k Pole ksi k

Tabela 4.1. Funkcje otrzymujące limity czasowe

Klasa lub przestrzeń nazw	Funkcje	Zwracane wartości
Przestrzeń nazw std::this_thread	sleep_for(*okres*) sleep_until(*punkt_w_czasie*)	Nie dotyczy
std::condition_variable lub std::condition_variable_any ↪wait_for(*blokada*, *okres*)	std::cv_status::timeout lub std::cv_status::no_timeout	
wait_until(*blokada*, ↪*punkt_w_czasie*)	wait_for(*blokada*, *okres*, ↪*predykat*) wait_until(*blokada*, *punkt_* ↪*w_czasie*, *predykat*)	bool — wartość zwrócona przez *predykat* po obudzeniu wątku.
std::timed_mutex, std:: ↪recursive_timed_mutex lub std::shared_timed_mutextry_ ↪lock_for(*okres*) try_lock_until(*punkt_* ↪*w_czasie*)	bool — wartość true, jeśli udało się nałożyć blokadę; w przeciwnym razie wartość false	
std::shared_timed_mutex	try_lock_shared_for(*okres*) try_lock_shared_until ↪(*punkt_w_czasie*)	bool — wartość true, jeśli udało się nałożyć blokadę; w przeciwnym razie wartość false
std::unique_lock<*Typ* ↪*ZMożliwościąBlokady* ↪*Czasowej*>unique_lock ↪(*typ_blokowalny*, *okres*)	Nie dotyczy — funkcja owns_lock() wywołana dla nowo skonstruowanego obiektu zwraca wartość true, jeśli udało się nałożyć blokadę (w przeciwnym razie zwraca wartość false).	
unique_lock(*typ_blokowalny*, *punkt_w_czasie*)	try_lock_for(*okres*) try_lock_until ↪(*punkt_w_czasie*)	bool — wartość true, jeśli udało się nałożyć blokadę; w przeciwnym razie wartość false
std::shared_lock<Współ ↪dzielony*TypZMożliwością* ↪*BlokadyCzasowej*>shared_ ↪lock(*typ_blokowalny*, *okres*)	Nie dotyczy — funkcja owns_lock() wywołana dla nowo skonstruowanego obiektu zwraca wartość true, jeśli udało się nałożyć blokadę (w przeciwnym razie zwraca wartość false).	
shared_lock(*typ_blokowalny*, ↪*punkt_w_czasie*)	try_lock_for(*okres*) try_lock_until ↪(*punkt_w_czasie*)	bool — wartość true, jeśli udało się nałożyć blokadę; w przeciwnym razie wartość false
std::future<*TypWartości*> lub std::shared_future< ↪*TypWartości*>wait_for(*okres*) wait_until(*punkt_w_czasie*)	Jeśli wyczerpano limit czasu funkcji oczekującej, zwraca wartość std::future_status::timeout; jeśli obiekt przyszłości jest gotowy, zwraca wartość std::future_ ↪status::ready; jeśli przyszłość zawiera odroczoną funkcję, która nie została jeszcze wywołana, zwraca wartość std::future_ ↪status::deferred.	

Skoro wiemy już, jak działają zmienne warunkowe, obiekty przyszłości i obietnic oraz opakowane zadania, czas przeanalizować te mechanizmy w nieco szerszym kontekście, w szczególności przyjrzeć się technikom upraszczania (za pomocą tych mechanizmów) synchronizacji operacji wykonywanych przez różne wątki.

4.4. Upraszczanie kodu za pomocą technik synchronizowania operacji

Stosowanie mechanizmów synchronizacji, które opisałem w poprzednich podrozdziałach, w roli gotowych elementów składowych umożliwia programiście koncentrowanie się na samych operacjach wymagających synchronizacji, nie na mechanice tej synchronizacji. Mechanizmy synchronizacji pozwalają uprościć kod aplikacji choćby dlatego, że wprowadzają do świata programowania współbieżnego dużo więcej elementów znanych z **programowania funkcyjnego**. Zamiast bezpośrednio współdzielić dane pomiędzy wątkami, każde zadanie może otrzymywać potrzebne dane, a wynik przetwarzania może być przekazywany do wielu innych wątków za pośrednictwem obiektów przyszłości.

4.4.1. Programowanie funkcyjne przy użyciu przyszłości

Termin **programowanie funkcyjne** (ang. *functional programming* — *FP*) odnosi się do stylu programowania, w którym wynik wywołania funkcji zależy wyłącznie od parametrów przekazanych na jej wejściu. Oznacza to, że na wynik funkcji nie ma wpływu zewnętrzny stan. Opisane działanie jest więc zgodne z matematycznym pojęciem funkcji, gdzie każde użycie jednej funkcji z tymi samymi parametrami spowoduje otrzymanie dokładnie takiego samego wyniku. W ten sposób działa wiele matematycznych funkcji biblioteki standardowej języka C++, jak `sin`, `cos` czy `sqrt`, oraz prostych operacji na typach podstawowych, jak `3+3`, `6*9` czy `1.3/4.7`. **Typowa** funkcja nie **modyfikuje** zewnętrznego stanu; skutki wykonywania tej funkcji ograniczają się tylko do zwracanej wartości.

Opisany model programowania ułatwia interpretację kodu, szczególnie jeśli program zawiera elementy przetwarzania współbieżnego, ponieważ wiele problemów związanych z pamięcią współdzieloną (opisanych w rozdziale 3.) w ogóle nie występuje w świecie programowania funkcyjnego. Skoro dane współdzielone nie są modyfikowane, nie mogą wystąpić sytuacje wyścigów, zatem ochrona tych danych za pomocą muteksów jest po prostu niepotrzebna. Właśnie prostota tego modelu powoduje, że takie języki programowania jak Haskell (*https://www.haskell.org/*), gdzie wszystkie funkcje **domyślnie** spełniają warunek programowania funkcyjnego, zyskują coraz większą popularność wśród programistów systemów współbieżnych. Ponieważ niemal cały kod jest zgodny z zasadami programowania funkcyjnego, **nieliczne** funkcje, które **modyfikują** współdzielony stan, na tyle wyróżniają się spośród pozostałych elementów, że można bez trudu ocenić ich udział w całej strukturze aplikacji.

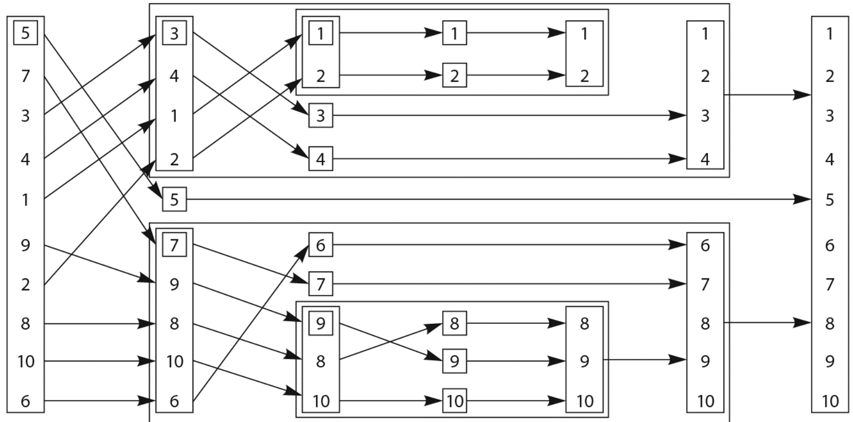
Zalety programowania funkcyjnego nie ograniczają się tylko do języków, w których ten model jest domyślnym paradygmatem. Język C++ łączy w sobie wiele paradygmatów, zatem także pisanie programów według zasad programowania funkcyjnego jest możliwe w tym języku. W wersji C++11 programowanie funkcyjne jest jeszcze prostsze niż w standardzie C++98 dzięki wprowadzeniu funkcji lambda (patrz część A.5 dodatku A), integracji funkcji `std::bind` zaczerpniętej z biblioteki Boost i dokumentu

Kup ksi k Pole ksi k

TR1 oraz dodaniu automatycznego wnioskowania typów zmiennych (patrz część A.7 dodatku A). Ostatnim elementem, który ułatwia programowanie funkcyjne w języku C++, są obiekty przyszłości — obiekt przyszłości można przekazywać pomiędzy wątkami, tak aby wynik jednej operacji mógł zależeć od wyniku innej operacji i aby ta zależność **nie wymagała bezpośredniego dostępu do współdzielonych danych**.

SZYBKIE SORTOWANIE W MODELU PROGRAMOWANIA FUNKCYJNEGO

Aby lepiej zrozumieć możliwe zastosowania przyszłości w modelu programowania funkcyjnego, przeanalizujmy prostą implementację algorytmu sortowania szybkiego (ang. *quicksort*). Podstawowa koncepcja tego algorytmu jest dość prosta — należy z listy wartości wybrać element dzielący, osiowy (ang. *pivot*), po czym podzielić listę na dwa zbiory, z których jeden zawiera elementy mniejsze od elementu dzielącego, a drugi zawiera elementy większe od wybranego elementu. Posortowana kopia listy jest uzyskiwana poprzez sortowanie obu podzbiorów i połączenie odpowiednio posortowanej listy złożonej z wartości mniejszych od elementu dzielącego, samego elementu dzielącego i posortowanej listy większej od elementu dzielącego. Przykład sortowania listy dziesięciu liczb całkowitych według opisanego schematu pokazano na rysunku 4.2. Na listingu 4.12 pokazano sekwencyjną implementację tego algorytmu opracowaną zgodnie z zasadami programowania funkcyjnego; funkcja `sequential_quick_sort()` otrzymuje listę i zwraca jej posortowaną kopię przez wartość (zamiast sortować listę przekazaną przez referencję, jak w przypadku funkcji `std::sort()`).



Rysunek 4.2. Rekurencyjne sortowanie w modelu programowania funkcyjnego

Listing 4.12. Sekwencyjna implementacja algorytmu sortowania szybkiego

```
template<typename T>
std::list<T> sequential_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
```

Kup ksi k Pole ksi k

```
    result.splice(result.begin(),input,input.begin());     ←❶
    T const& pivot=*result.begin();     ←❷

    auto divide_point=std::partition(input.begin(),input.end(),
            [&](T const& t){return t<pivot;});     ←❸

    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point);     ←❹

    auto new_lower(
        sequential_quick_sort(std::move(lower_part)));     ←❺
    auto new_higher(
        sequential_quick_sort(std::move(input)));     ←❻

    result.splice(result.end(),new_higher);     ←❼
    result.splice(result.begin(),new_lower);     ←❽
    return result;
}
```

Mimo że zewnętrzny interfejs tej implementacji jest zgodny z regułami programowania funkcyjnego, wewnętrzne mechanizmy zaimplementowano w „tradycyjny” sposób, ponieważ konsekwentne stosowanie modelu funkcyjnego wymagałoby wielu dodatkowych operacji kopiowania. W roli elementu dzielącego jest wybierany pierwszy element, który jest wyodrębniany z listy za pomocą funkcji `splice()`❶. Chociaż sortowanie na bazie tak wybranego elementu dzielącego może nie być optymalne (liczba operacji porównania i wymiany może być większa, niż to konieczne), pozostałe operacje na strukturze typu `std::list` będą wykonywane szybciej dzięki efektywnemu przeszukiwaniu listy. Wiadomo, że wyodrębniony element dzielący musi się znaleźć na liście wynikowej, zatem jest od razu umieszczany w odpowiedniej strukturze. Ponieważ element dzielący będzie teraz porównywany z pozostałymi elementami, przekazujemy referencję do tego elementu, aby uniknąć wielokrotnego kopiowania ❷. Możemy następnie użyć funkcji `std::partition` do podzielenia sekwencji na wartości **mniejsze** od elementu dzielącego i wartości **nie mniejsze** od tego elementu ❸. Najprostszym sposobem określenia kryterium podziału jest użycie funkcji lambda — aby uniknąć kopiowania wartości elementu dzielącego, zastosowano tutaj technikę przechwytywania referencji (więcej informacji na temat funkcji lambda można znaleźć w części A.5 dodatku A).

Funkcja `std::partition()` przetwarza przekazaną listę i zwraca iterator wskazujący pierwszy element, który **nie** jest mniejszy od wartości elementu dzielącego. Kompletny typ iteratora może być dość długi, zatem w powyższym kodzie użyto mechanizmu automatycznej identyfikacji typów, aby to kompilator automatycznie określił odpowiedni typ (patrz część A.7 dodatku A).

Ponieważ analizowana implementacja udostępnia interfejs zgodny z zasadami programowania funkcyjnego, warunkiem rekurencyjnego posortowania obu „połówek” listy jest utworzenie dwóch odrębnych list. Do tego celu możemy ponownie użyć funkcji `splice()`, aby skopiować wartości z listy wejściowej (do elementu `divide_point`) i umieścić

Kup ksi k Pole ksi k

na nowej liście: `lower_part` ❹. Reszta wartości pozostanie na liście wejściowej. Obie listy można następnie posortować za pomocą rekurencyjnych wywołań funkcji `sequential_quick_sort()` ❺ ❻. Użycie funkcji `std::move()` podczas przekazywania list na wejściu rekurencyjnych wywołań pozwala uniknąć kopiowania tych struktur (wyniki obu wywołań są kopiowane automatycznie). I wreszcie możemy ponownie użyć funkcji `splice()` w celu uporządkowania list reprezentujących podzbiory elementów oryginalnej struktury. Wartości z listy `new_higher` trafiają na koniec listy wynikowej ❼ (za element dzielący), natomiast wartości z listy `new_lower` są umieszczane na początku listy ❽ (przed elementem dzielącym).

Szybkie sortowanie równoległe w modelu programowania funkcyjnego

Ponieważ już w poprzednim przykładzie zastosowano reguły programowania funkcyjnego, konwersja tego algorytmu na wersję równoległą (korzystającą z obiektów przyszłości) nie jest specjalnie trudna (patrz listing 4.13). Zbiór operacji jest taki sam jak w poprzedniej wersji, tyle że teraz część tych operacji jest wykonywana równolegle. W tej wersji użyto implementacji algorytmu sortowania szybkiego łączącej obiekty przyszłości i model programowania funkcyjnego.

Listing 4.13. Równoległe sortowanie szybkie z wykorzystaniem przyszłości

```
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin());
    T const& pivot=*result.begin();

    auto divide_point=std::partition(input.begin(),input.end(),
         [&](T const& t){return t<pivot;});

    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point);

    std::future<std::list<T> > new_lower(        ←❶
        std::async(&parallel_quick_sort<T>,std::move(lower_part)));

    auto new_higher(
        parallel_quick_sort(std::move(input)));        ←❷

    result.splice(result.end(),new_higher);        ←❸
    result.splice(result.begin(),new_lower.get());        ←❹
    return result;
}
```

Jedną z najważniejszych różnic dzielących obie wersje jest to, że w wersji współbieżnej część listy sprzed elementu dzielącego nie jest sortowana w bieżącym wątku, tylko w dodatkowym wątku — w tym celu zastosowano funkcję `std::async()` ❶. Druga część listy jest sortowana tak jak w poprzedniej wersji, a więc przy użyciu bezpośredniego wywołania rekurencyjnego ❷. Rekurencyjne wywołanie funkcji `parallel_quick_sort()` pozwala wykorzystać dostępną współbieżność sprzętową. Jeśli wywołanie funkcji `std::async()` za każdym razem uruchamia nowy wątek, wystarczą trzy poziomy rekurencji, aby program został podzielony na osiem wątków; w przypadku dziesięciu poziomów rekurencji (czyli około tysiąca elementów) program w tej formie uruchomi 1024 wątki (o ile stosowany sprzęt poradzi sobie z taką liczbą). W razie wykrycia zbyt dużej liczby uruchomionych zadań (jeśli na przykład liczba równolegle realizowanych zadań przekroczy dostępną współbieżność sprzętową) biblioteka może przejść w tryb synchronicznego uruchamiania nowych zadań. Zadania będą wykonywane w wątku wywołującym funkcję `get()`, nie w nowym wątku, zatem program uniknie kosztów przekazywania zadania pomiędzy wątkami, jeśli koszty tej operacji nie są rekompensowane przez wzrost wydajności. Jeśli nie przekazano wprost wartości `std::launch::deferred`, uruchamianie nowego wątku dla każdego zadania jest w pełni zgodne z założeniami implementacji `std::async` (nawet jeśli prowadzi do nadsubskrypcji); podobnie jeśli nie przekazano wartości `std::launch::async`, najlepszym rozwiązaniem jest synchroniczne wykonywanie wszystkich zadań. W przypadku stosowania biblioteki oferującej mechanizmy automatycznego skalowania warto sprawdzić w dokumentacji, jak te mechanizmy będą działały w kontekście tego algorytmu.

Zamiast funkcji `std::async()` moglibyśmy użyć własnej funkcji `spawn_task()` w roli prostego opakowania szablonu klasy `std::packaged_task` i klasy `std::thread` (patrz listing 4.14). W takim przypadku należałoby utworzyć obiekt klasy `std::packaged_task` dla wyniku wywołania funkcji, odczytać obiekt przyszłości z obiektu zadania, uruchomić zadanie w odpowiednim wątku, po czym zwrócić obiekt przyszłości. Takie rozwiązanie samo w sobie nie przyniosłoby co prawda żadnych korzyści (prowadziłoby raczej do dużej nadsubskrypcji), ale może stanowić punkt wyjścia dla bardziej zaawansowanych implementacji, które będą dodawały zadania do kolejki w celu przetworzenia przez wątki robocze dostępne w puli. Zagadnienia związane z pulami wątków zostaną omówione w rozdziale 9. Wybór tego kierunku (zamiast stosowania funkcji `std::async`) jest uzasadniony tylko w przypadku programistów, którzy mają pełną świadomość skutków podejmowanych działań i chcą mieć pełną kontrolę nad sposobem budowy puli wątków i wykonywania zadań.

Listing 4.14. Prosta implementacja funkcji spawn_task

```
template<typename F,typename A>
std::future<std::result_of<F(A&&)>::type>
    spawn_task(F&& f,A&& a)
{
    typedef std::result_of<F(A&&)>::type result_type;
    std::packaged_task<result_type(A&&)>
        task(std::move(f)));
    std::future<result_type> res(task.get_future());
    std::thread t(std::move(task),std::move(a));
    t.detach();
    return res;
}
```

Kup ksi k Pole ksi k

Wróćmy teraz do funkcji `parallel_quick_sort()`. Ponieważ do uzyskania listy `new_higher` użyliśmy bezpośredniego wywołania rekurencyjnego, możemy użyć funkcji `splice()` tak jak w algorytmie jednowątkowym ❸. Okazuje się jednak, że zmienna `new_lower` zawiera obiekt klasy `std::future<std::list<T>>`, nie listę, zatem przed wywołaniem funkcji `splice()` musimy wywołać funkcję `get()`, aby uzyskać odpowiednią wartość ❹. Wywołanie w tej formie czeka na zakończenie zadania wykonywanego w tle, po czym **przenosi** wynik do wywołania funkcji `splice()`. Funkcja `get()` zwraca referencję do r-wartości wyniku, zatem lista może zostać przeniesiona (więcej informacji na temat referencji do r-wartości i operacji przenoszenia można znaleźć w części A.1.1 dodatku A).

Nawet jeśli przyjąć, że funkcja `std::async()` w optymalny sposób wykorzystuje dostępną współbieżność sprzętową, proponowana implementacja równoległego algorytmu sortowania szybkiego wciąż nie jest optymalna. Funkcja `std::partition` realizuje co prawda znaczną część zadań związanych z działaniem tego algorytmu, ale jej wywołanie ma charakter typowo sekwencyjny. Czytelnicy zainteresowani możliwie najszybszą, równoległą implementacją powinni sięgnąć po odpowiednią literaturę akademicką. Alternatywne rozwiązanie polega na użyciu równoległego przeciążenia oferowanego przez bibliotekę standardową C++17 (więcej informacji na ten temat znajdziesz w rozdziale 10.).

Programowanie funkcyjne nie jest jedynym paradygmatem programowania współbieżnego eliminującym problem współdzielenia zmiennych danych; innym przykładem takiego paradygmatu jest komunikacja procesów sekwencyjnych (ang. *Communicating Sequential Processes — CSP*)[2], gdzie wątki są w założeniu całkowicie niezależne i nie operują na współdzielonych danych — zamiast tego wymieniają komunikaty za pośrednictwem kanałów komunikacyjnych. Paradygmat CSP zastosowano w języku programowania Erlang (*http://www.erlang.org/*) oraz w środowisku MPI (od ang. *Message Passing Interface*) stosowanym w systemach implementowanych w językach C i C++, które muszą gwarantować najwyższą wydajność (*http://www.mpi-forum.org/*). Po tym, co już napisałem, jestem pewien, że wiadomość o możliwości implementacji tego paradygmatu także w języku C++ nie będzie dla czytelnika żadnym zaskoczeniem — wystarczy odrobina dyscypliny. Sposób implementacji tego modelu omówię w następnym punkcie.

4.4.2. Synchronizacja operacji za pomocą przesyłania komunikatów

Koncepcja paradygmatu CSP jest prosta — nie istnieją żadne współdzielone dane, a każdy wątek można traktować jako zupełnie niezależny byt. Zachowanie wątku zależy wyłącznie od komunikatów, które do niego trafiają. Każdy wątek jest więc swoistą maszyną stanów, która po otrzymaniu komunikatu aktualizuje swój stan i która może (ale nie musi) wysłać co najmniej jeden komunikat do pozostałych wątków. Sposób przetwarzania komunikatu zależy od stanu początkowego „maszyny”. Jednym ze sposobów pisania wątków tego typu jest stworzenie formalnego modelu i implementacja skończonej maszyny stanów, jednak istnieją też lepsze rozwiązania — do wyrażenia maszyny stanów można użyć odpowiedniej struktury aplikacji. To, która metoda sprawdza się

[2] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985. Książka jest dostępna za darmo pod adresem *http://www.usingcsp.com/cspbook.pdf*.

Kup ksi k Pole ksi k

lepiej w danym scenariuszu, zależy od szczegółowych wymagań dotyczących zachowań budowanego systemu i od umiejętności zespołu programistów. Jeśli jednak zdecydujemy się na implementację odrębnych wątków, sam podział na niezależne procesy może prowadzić do wyeliminowania wielu komplikacji związanych ze współbieżnym przetwarzaniem danych współdzielonych i tym samym ułatwić programowanie oraz ograniczyć liczbę błędów.

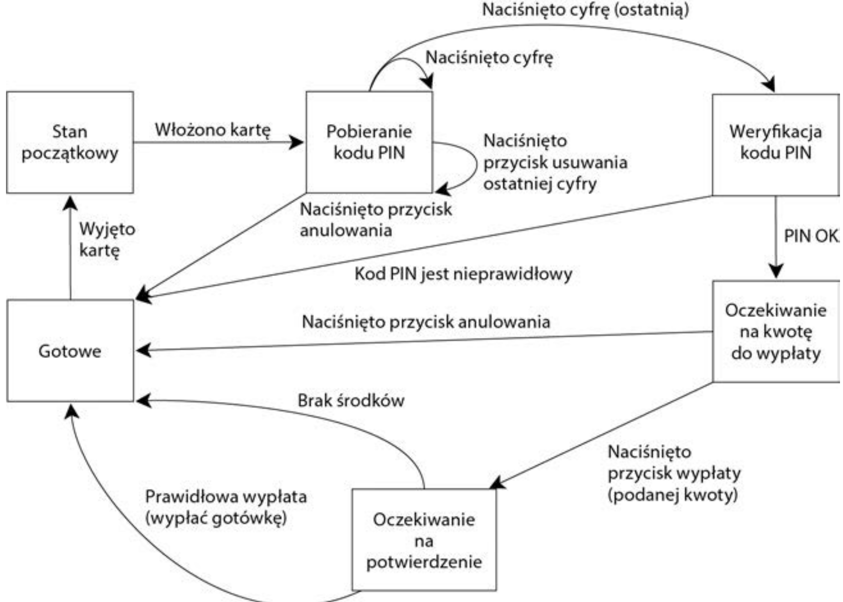
Procesy w pełni zgodne z paradygmatem CSP nie operują na żadnych współdzielonych danych, a cała komunikacja odbywa się za pośrednictwem kolejek komunikatów. Ponieważ jednak wątki języka C++ współdzielą przestrzeń adresową, wymuszenie tego wymagania jest niemożliwe. W tej sytuacji bardzo duże znaczenie ma dyscyplina autorów aplikacji i bibliotek, ponieważ to od nas zależy, czy nasze wątki będą się odwoływać do współdzielonych danych. Same kolejki komunikatów oczywiście muszą być współdzielone (w przeciwnym razie wątki nie mogłyby się komunikować), jednak szczegóły implementacji tych kolejek można opakować w ramach bibliotek.

Wyobraźmy sobie, że implementujemy kod dla bankomatu. Kod tego systemu musi obsługiwać interakcję z użytkownikiem, czyli osobą wypłacającą gotówkę, musi komunikować się z systemem odpowiedniego banku oraz musi sterować fizycznymi urządzeniami odpowiedzialnymi za akceptację karty, wyświetlanie stosownych komunikatów, obsługę klawiatury, wypłatę pieniędzy i zwracanie karty.

Jednym ze sposobów obsługi wszystkich tych zadań jest podzielenie kodu na trzy niezależne wątki: jeden obsługujący fizyczne urządzenia, drugi implementujący logikę samego bankomatu i trzeci odpowiedzialny za komunikację z bankiem. Wątki mogą się komunikować wyłącznie poprzez przekazywanie komunikatów (nie poprzez współdzielenie jakichkolwiek danych). Na przykład wątek obsługujący fizyczne urządzenia mógłby wysłać do wątku logiki bankomatu komunikat o włożeniu karty lub naciśnięciu przycisku przez użytkownika, natomiast wątek logiki mógłby wysłać do wątku obsługującego fizyczne urządzenia komunikat określający kwotę do wypłacenia.

Jednym ze sposobów modelowania logiki bankomatu jest opracowanie maszyny stanów. W każdym stanie wątek oczekuje na określony komunikat, który jest następnie odpowiednio przetwarzany. W wyniku przetwarzania tego komunikatu wątek może przejść w nowy stan i kontynuować cały cykl. Stany składające się na tę prostą implementację pokazano na rysunku 4.3. W tej uproszczonej implementacji system oczekuje na umieszczenie karty w bankomacie. Po włożeniu karty system czeka, aż użytkownik wpisze kod PIN, naciskając kolejno cyfry tego kodu. Użytkownik może usunąć ostatnią wpisaną cyfrę. Po wpisaniu odpowiedniej liczby cyfr system weryfikuje kod PIN. Jeśli PIN jest nieprawidłowy, cykl działania kończy się — bankomat zwraca kartę i przechodzi w stan oczekiwania na wsunięcie karty przez klienta. Jeśli kod PIN jest prawidłowy, system czeka na anulowanie transakcji albo na wybór kwoty do wypłacenia. W razie anulowania transakcji cykl pracy bankomatu kończy się i urządzenie zwraca kartę. Jeśli klient wybrał kwotę, przed wypłaceniem gotówki system czeka na potwierdzenie ze strony banku, po czym albo wypłaca gotówkę, albo wyświetla komunikat „brak środków” i (niezależnie od wyniku weryfikacji stanu konta) wysuwa kartę. Systemy prawdziwych bankomatów są oczywiście bardziej skomplikowane, jednak opisana powyżej maszyna stanów dobrze ilustruje istotę tego rozwiązania.

Kup ksi k Pole ksi k



Rysunek 4.3. Prosty model maszyny stanów dla bankomatu

Po zaprojektowaniu maszyny stanów dla logiki bankomatu możemy przystąpić do implementacji tego rozwiązania w formie klasy, która będzie definiowała po jednej funkcji składowej dla każdego stanu. Każda funkcja składowa może czekać na określony zbiór komunikatów przychodzących i odpowiednio obsługiwać te komunikaty (obsługa może polegać na przechodzeniu do innego stanu). Każdy typ komunikatów jest reprezentowany przez odrębną strukturę. Na listingu 4.15 pokazano fragment prostej implementacji logiki takiego systemu, w tym główną pętlę oraz implementację pierwszego stanu, w którym system oczekuje na włożenie karty.

Listing 4.15. Prosta implementacja klasy logiki systemu bankomatu

```
struct card_inserted
{
    std::string account;
};
class atm
{
    messaging::receiver incoming;
    messaging::sender bank;
    messaging::sender interface_hardware;
    void (atm::*state)();

    std::string account;
    std::string pin;

    void waiting_for_card()          ←❶
    {
        interface_hardware.send(display_enter_card());          ←❷
```

```
        incoming.wait()      ❸
            .handle<card_inserted>(
                [&](card_inserted const& msg)      ❹
                {
                    account=msg.account;
                    pin="";
                    interface_hardware.send(display_enter_pin());
                    state=&atm::getting_pin;
                }
                );
    }
    void getting_pin();
public:
    void run()      ❺
    {
        state=&atm::waiting_for_card;      ❻
        try
        {
            for(;;)
            {
                (this->*state)();      ❼
            }
        }
        catch(messaging::close_queue const&)
        {
        }
    }
};
```

Jak widać, wszystkie niezbędne operacje związane z synchronizacją przekazywania komunikatów zostały ukryte w odpowiedniej bibliotece (implementacja tej biblioteki zostanie pokazana w dodatku C wraz z kompletnym kodem tego przykładu).

Jak już wspomniałem, opisana tutaj implementacja jest mocno uproszczona w stosunku do logiki obowiązującej w prawdziwych bankomatach, jednak przykład w tej formie wystarczy do zrozumienia stylu programowania na bazie przekazywania komunikatów. Nie musimy tracić czasu na projektowanie synchronizacji i rozwiązywanie problemów związanych z przetwarzaniem współbieżnym — wystarczy ustalić, jakie komunikaty mogą być odbierane i przetwarzane na poszczególnych etapach oraz które komunikaty należy wysyłać. Maszyna stanów logiki bankomatu jest przetwarzana przez jeden wątek; pozostałe elementy systemu, jak interfejs łączący się z bankiem czy interfejs terminala, są obsługiwane przez odrębne wątki. Ten styl projektowania oprogramowania określa się mianem **modelu aktorów** — w systemie istnieje wiele odrębnych **aktorów** (każdy działa w osobnym wątku), które wymieniają pomiędzy sobą komunikaty niezbędne do realizacji swoich zadań. W modelu aktorów nie istnieje współdzielony stan (wyjątkiem jest mechanizm potrzebny do bezpośredniego przekazywania komunikatów).

Działanie programu rozpoczyna się od funkcji składowej `run()`❺, która ustawia stan początkowy, czyli `waiting_for_card` ❻, po czym wielokrotnie wywołuje funkcję składową reprezentującą bieżący stan ❼ (niezależnie od tego, który to stan). Funkcje stanów mają postać prostych funkcji składowych klasy `atm`. Także funkcja stanu

waiting_for_card ❶ jest dość prosta — jej działanie ogranicza się do wysłania komunikatu do interfejsu w celu wyświetlenia na ekranie tekstu *Czekam na kartę* ❷; zaraz potem funkcja rozpoczyna oczekiwanie na komunikat do obsłużenia ❸. Jedynym rodzajem komunikatów, który może być obsługiwany w tej części kodu, jest komunikat card_inserted. Do jego obsługi używamy funkcji lambda ❹. Na wejściu funkcji handle można przekazać dowolną funkcję lub dowolny obiekt funkcji, jednak najprostszym rozwiązaniem jest użycie funkcji lambda. Łatwo zauważyć, że wywołanie funkcji handle() znalazło się w łańcuchu obejmującym wywołanie funkcji wait(), zatem w razie otrzymania komunikatu, który nie pasuje do wskazanego typu, wątek zignoruje ten komunikat i będzie dalej czekał na komunikat właściwego typu.

Sama funkcja lambda zapisuje numer konta w zmiennej składowej, zeruje bieżący kod PIN, wysyła komunikat do sprzętu odpowiedzialnego za obsługę interfejsu bankomatu, aby wyświetlić prośbę o podanie kodu PIN, po czym przechodzi w stan pobierania tego kodu. Po zakończeniu działania przez funkcję obsługującą komunikat funkcja stanu zwraca sterowanie, a główna pętla programu wywołuje funkcję nowego stanu ❼.

Funkcja stanu getting_pin jest nieco bardziej skomplikowana, ponieważ może obsługiwać trzy różne typy komunikatów (patrz rysunek 4.3). Kod tej funkcji pokazano na listingu 4.16.

Listing 4.16. Funkcja stanu getting_pin na potrzeby prostej implementacji systemu bankomatu

```
void atm::getting_pin()
{
    incoming.wait()
        .handle<digit_pressed>(      ←❶
            [&](digit_pressed const& msg)
            {
                unsigned const pin_length=4;
                pin+=msg.digit;
                if(pin.length()==pin_length)
                {
                    bank.send(verify_pin(account,pin,incoming));
                    state=&atm::verifying_pin;
                }
            }
            )
        .handle<clear_last_pressed>(      ←❷
            [&](clear_last_pressed const& msg)
            {
                if(!pin.empty())
                {
                    pin.resize(pin.length()-1);
                }
            }
            )
        .handle<cancel_pressed>(      ←❸
            [&](cancel_pressed const& msg)
```

```
        {
            state=&atm::done_processing;
        }
        );
}
```

Tym razem funkcja musi przetwarzać trzy różne typy komunikatów, zatem łańcuch wywołania funkcji `wait()` obejmuje aż trzy wywołania funkcji `handle()` ❶ ❷ ❸. Każde wywołanie funkcji `handle()` określa typ komunikatu w formie parametru szablonu, po czym przekazuje funkcję lambda, która otrzymuje na wejściu komunikat określonego typu. Ponieważ wszystkie te wywołania umieszczono w jednym łańcuchu wywołań, implementacja funkcji `wait()` „wie”, że czeka na komunikat `digit_pressed`, `clear_`
↪`last_pressed` lub `cancel_pressed`. Komunikaty wszystkich innych typów będą (tak jak wcześniej) ignorowane.

W tym przypadku otrzymanie komunikatu nie musi prowadzić do zmiany stanu. Jeśli na przykład wątek otrzyma komunikat `digit_pressed` i jeśli wpisana cyfra nie będzie ostatnią cyfrą kodu, wystarczy tę cyfrę dopisać do łańcucha `pin`. Główna pętla na listingu 4.15 ❼ ponownie wywoła funkcję `getting_pin()`, aby czekać na następną cyfrę (bądź wyzerować kod PIN lub anulować całą operację).

Opisana procedura jest zgodna ze schematem pokazanym na rysunku 4.3. Każdy stan przedstawiony na tym rysunku jest implementowany przez inną funkcję składową, która czeka na odpowiednie komunikaty i na tej podstawie aktualizuje stan systemu.

Jak widać, opisany styl programowania może znacznie uprościć zadanie projektowania systemu współbieżnego, ponieważ każdy wątek można traktować całkowicie niezależnie od pozostałych. W opisanym modelu wiele wątków ma za zadanie oddzielanie zagadnień i jako takie wymagają od projektanta jasnych decyzji o podziale zadań pomiędzy wątki.

W podrozdziale 4.2 wspomniałem, że specyfikacja techniczna Concurrency TS zapewnia rozbudowaną obsługę przyszłości. Kluczowym elementem dostępnych rozszerzeń jest możliwość określenia tzw. *kontynuacji*, czyli funkcji dodatkowych wykonywanych automatycznie, gdy przyszłość stanie się *gotowa*. Zobaczysz teraz, jak można wykorzystać tę funkcjonalność do uproszczenia kodu.

4.4.3. Współbieżność w stylu kontynuacji dzięki użyciu Concurrency TS

Specyfikacja techniczna Concurrency TS oferuje w przestrzeni nazw `std::experimental` nowe wersje `std::promise` i `std::packaged_task`, które w taki sam sposób różnią się od ich odpowiedników w `std` — zwracają egzemplarze `std::experimental::future` zamiast `std::future`. To pozwala użytkownikom na wykorzystanie nowej ważnej funkcji w `std::experimental::future` — *kontynuacji*.

Przyjmuję założenie o istnieniu zadania, które powoduje wygenerowanie wyniku, oraz istnienia przyszłości, która będzie przechowywała wynik, gdy tylko stanie się on dostępny. Następnie potrzebny jest pewien kod odpowiedzialny za przetworzenie wyniku. Dzięki `std::future` trzeba poczekać do chwili, aż przyszłość będzie gotowa, co oznacza wywołanie w pełni blokującej funkcji składowej `wait_for()` lub `wait_until()`, aby umożliwić czekanie wraz z `timeout`. Takie rozwiązanie może być niewygodne i skomplikować kod. Potrzebujesz rozwiązania w stylu „gdy dane będą gotowe, **wtedy**

należy rozpocząć przetwarzanie". Dokładnie w taki sposób działa kontynuacja. Nie powinno być zaskakujące, że funkcja składowa pozwalająca na dodanie kontynuacji do przyszłości nosi nazwę `then()` (z ang. *wtedy*). W przypadku przyszłości `fut` dodanie kontynuacji odbywa się za pomocą wywołania `fut.then(continuation)`.

Podobnie jak w przypadku `std::future`, także `std::experimental::future` pozwala na tylko jednokrotne pobranie przechowywanej wartości. Jeżeli wartość jest używana przez kontynuację, nie będzie można uzyskać do niej dostępu z poziomu innego kodu. Gdy kontynuacja zostanie dodana za pomocą wywołania `fut.then()`, wówczas pierwotna przyszłość, `fut`, stanie się **nieprawidłowa**. Dlatego też wywołanie `fut.then()` zwróci nową przyszłość przeznaczoną do przechowywania wyniku kontynuacji. Takie rozwiązanie przedstawiłem w kolejnym fragmencie kodu.

```
std::experimental::future<int> find_the_answer;
auto fut=find_the_answer();
auto fut2=fut.then(find_the_question);
assert(!fut.valid());
assert(fut2.valid());
```

Funkcja kontynuacji `find_the_question` jest przeznaczona do uruchomienia w „nieokreślonym wątku", gdy przyszłość będzie **gotowa**. Dzięki temu implementacja może korzystać z puli wątków lub innego wątku zarządzanego przez bibliotekę. To oznacza dość dużą elastyczność, która została zapewniona celowo. Intencje są takie, że po dodaniu kontynuacji do przyszłych wydań standardu C++ programiści będą w stanie lepiej wykorzystać swoje możliwości podczas wyboru wątków i zapewnić użytkownikom odpowiednie mechanizmy przeznaczone do kontrolowania wyboru wątków.
W przeciwieństwie do bezpośrednich wywołań `std::async` lub `std::thread` nie można przekazywać argumentów do funkcji kontynuacji, ponieważ argument jest już zdefiniowany przez bibliotekę — kontynuacja otrzymuje **gotową** przyszłość przechowującą wynik, który spowodował wywołanie kontynuacji. Przyjmując założenie, że funkcja `find_the_answer` zwraca wartość `int`, użyta w poprzednim przykładzie funkcja `find_the_question` musi pobierać `std::experimental::future<int>` jako jedyny parametr.

```
std::string find_the_question(std::experimental::future<int> the_answer);
```

Powód takiego rozwiązania jest prosty — funkcja, do której została dołączona kontynuacja, może zakończyć działanie, generując wartość do przechowania lub zgłaszając wyjątek. Jeżeli przyszłość została niejawnie wywołana do bezpośredniego przekazania wartości do kontynuacji, wówczas do biblioteki należy wybór sposobu obsługi wyjątku. Natomiast jeśli przyszłość jest przekazywana do kontynuacji, wtedy to kontynuacja określa sposób obsługi wyjątku. W prostych przypadkach to może odbywać się za pomocą wywołania `fut.get()` i możliwości ponownego zgłoszenia wyjątku, aby został propagowany poza funkcję kontynuacji. Podobnie jak w przypadku funkcji przekazywanej do `std::async`, wyjątek opuszczający kontynuację jest przechowywany w przyszłości, która przechowuje także wynik kontynuacji.
Warto zwrócić uwagę na to, że specyfikacja techniczna Concurrency TS nie określa, iż istnieje odpowiednik `std::async`, choć za pomocą rozszerzenia implementacja może dostarczać taki odpowiednik. Utworzenie odpowiedniej funkcji jest dość łatwym zadaniem — należy wykorzystać `std::experimental::promise` do pobrania przyszłości, następnie trzeba utworzyć nowy wątek za pomocą funkcji lambda przypisującej wartość obietnicy do wartości zwrotnej podanej funkcji, jak pokazałem w listingu 4.17.

Kup ksi k Pole ksi k

Listing 4.17. Prosty odpowiednik std::async dla funkcji Concurrency TS

```
template<typename Func>
std::experimental::future<decltype(std::declval<Func>()())>
spawn_async(Func&& func){
    std::experimental::promise<
        decltype(std::declval<Func>()())> p;
    auto res=p.get_future();
    std::thread t(
        [p=std::move(p),f=std::decay_t<Func>(func)]()
            mutable{
            try{
                p.set_value_at_thread_exit(f());
            } catch(...){
                p.set_exception_at_thread_exit(std::current_exception());
            }
    });
    t.detach();
    return res;
}
```

Wynik działania funkcji jest przechowywany w przyszłości lub następuje przechwycenie wyjątku zgłoszonego przez funkcję i jego przechowywanie w przyszłości, podobnie jak ma to miejsce w przypadku `std::async`. Ponadto w omawianym przykładzie mamy użycie funkcji `set_value_at_thread_exit` i `set_exception_at_thread_exit` gwarantujących, że zmienne `thread_local` zostaną poprawnie uprzątnięte, zanim przyszłość będzie gotowa.

Wartość zwracana przez wywołanie `then()` jest sama w sobie w pełni wyposażoną przyszłością. To oznacza możliwość łączenia kontynuacji ze sobą.

4.4.4. Łączenie kontynuacji ze sobą

Przyjmuję założenie o konieczności wykonania serii czasochłonnych zadań, które na dodatek mają być zrealizowane asynchronicznie, aby wątek główny pozostał wolny dla innych zadań. Przykładowo po zalogowaniu się użytkownika do aplikacji może zachodzić potrzeba przekazania danych uwierzytelniających do back-endu w celu uwierzytelnienia. Po uwierzytelnieniu będzie wykonane następne żądanie do back-endu, tym razem mające na celu pobranie informacji o koncie użytkownika. Później po pobraniu żądanych informacji następuje uaktualnienie danych wyświetlanych przez aplikację. W przypadku kodu działającego sekwencyjnie rozwiązanie może mieć postać podobną do przedstawionej na listingu 4.18.

Listing 4.18. Prosta funkcja sekwencyjna odpowiedzialna za przetworzenie logowania użytkownika

```
void process_login(std::string const& username,std::string const& password)
{
    try {
        user_id const id=backend.authenticate_user(username,password);
        user_data const info_to_display=backend.request_current_info(id);
        update_display(info_to_display);
    } catch(std::exception& e){
        display_error(e);
    }
}
```

Jednak kod sekwencyjny Cię nie interesuje, chcesz mieć kod asynchroniczny, aby nie blokować wątku interfejsu użytkownika. W przypadku `std::async` można przekazać zadanie do wątku działającego w tle, podobnie jak to pokazałem w listingu 4.19, ale takie rozwiązanie nadal będzie blokowało wątek i zabierało zasoby podczas oczekiwania na zakończenie zadań. Jeżeli jest wiele zadań do wykonania, skutkiem może być ogromna liczbą wątków, które nic nie robią i po prostu czekają.

Listing 4.19. Przetwarzanie logowania użytkownika za pomocą pojedynczego zadania asynchronicznego

```
std::future<void> process_login(
    std::string const& username,std::string const& password)
{
    return std::async(std::launch::async,[=](){
        try {
            user_id const id=backend.authenticate_user(username,password);
            user_data const info_to_display=
                backend.request_current_info(id);
            update_display(info_to_display);
        } catch(std::exception& e){
            display_error(e);
        }
    });
}
```

Aby uniknąć tych wszystkich blokujących wątków, potrzebny jest mechanizm pozwalający na łączenie zadań po zakończeniu ich wykonywania: kontynuacja. Na listingu 4.20 przedstawiłem ten sam ogólny proces, choć tym razem podzielony na serię zadań, z których każde jest łączone z poprzednim w ramach kontynuacji.

Listing 4.16. Funkcja przetwarzająca za pomocą kontynuacji logowanie użytkownika

```
std::experimental::future<void> process_login(
    std::string const& username,std::string const& password)
{
    return spawn_async([=](){
        return backend.authenticate_user(username,password);
    }).then([](std::experimental::future<user_id> id){
        return backend.request_current_info(id.get());
    }).then([](std::experimental::future<user_data> info_to_display){
       try{
            update_display(info_to_display.get());
        } catch(std::exception& e){
            display_error(e);
        }
    });
}
```

Zwróć uwagę na to, jak kontynuacja pobiera `std::experimental::feature` jako jedyny parametr, a następnie używa `.get()` do pobrania znajdującej się w nim wartości. To oznacza, że wyjątki są propagowane w dół łańcucha, więc wywołanie `info_to_display.get()` w ostatniej kontynuacji zgłosi wyjątek, jeśli którakolwiek funkcja w łańcuchu zgłosiła

Kup ksi k Pole ksi k

wyjątek. Natomiast blok `catch` potrafi obsłużyć wszystkie wyjątki, podobnie jak miało to miejsce w przypadku bloku `catch` przedstawionego na listingu 4.18.

Jeżeli wywołania funkcji back-endu powodują wewnętrzne blokowanie ze względu na konieczność oczekiwania na dostarczenie komunikatów przez sieć lub zakończenia operacji w bazie danych, to zadanie nie zostało jeszcze zakończone. Może wystąpić konieczność podziału zadania na mniejsze, ale nadal będziesz miał do czynienia z blokującymi wątkami. Wywołania back-endu muszą zwracać obietnice, które będą gotowe, gdy dane staną się gotowe — to odbędzie się bez blokowania jakichkolwiek wątków. W omawianym przykładzie wywołanie `backend.async_authenticate_user(username, ↪password)` zwróci egzemplarz `std::experimental::future<user_id>` zamiast zwykłego `user_id`.

Być może uważasz, że to skomplikuje kod, ponieważ wartością zwrotną kontynuacji jest `future<future<dowolna_wartość>>` lub trzeba umieścić wywołania `.then` wewnątrz kontynuacji. Jeśli tak pomyślałeś, to na szczęście jesteś w błędzie: obsługa kontynuacji oferuje użyteczną funkcję o nazwie rozpakowania przyszłości. Jeżeli funkcja kontynuacji przekazana do funkcji `.then()` zwróci `future<dowolny_typ>`, wówczas wartością zwrotną wywołania `.then()` będzie `future<dowolny_typ>`. Ostateczna postać kodu przedstawia się tak, jak pokazałem na listingu 4.21, natomiast w łańcuchu funkcji asynchronicznej nie zachodzi blokowanie.

Listing 4.21. Funkcja przetwarzająca logowanie użytkownika za pomocą w pełni asynchronicznej operacji

```
std::experimental::future<void> process_login(
    std::string const& username,std::string const& password)
{
    return backend.async_authenticate_user(username,password).then(
        [](std::experimental::future<user_id> id){
            return backend.async_request_current_info(id.get());
        }).then([](std::experimental::future<user_data> info_to_display){
            try{
                update_display(info_to_display.get());
            } catch(std::exception& e){
                display_error(e);
            }
        });
}
```

To jest proste niemalże jak kod sekwencyjny przedstawiony na listingu 4.18, choć zawiera nieco większą ilość kodu związanego z wywołaniami `.then()` i deklaracjami lambda. Jeżeli używany kompilator obsługuje generyki lambda ze specyfikacji C++14, wówczas typ przyszłości w parametrach lambda można zastąpić przez `auto`, co pozwala na dalsze uproszczenie kodu.

```
return backend.async_authenticate_user(username,password).then( [](auto id){
            return backend.async_request_current_info(id.get());
});
```

Jeżeli potrzebujesz czegoś bardziej skomplikowanego niż prosta, liniowa kontrola, wówczas żądane rozwiązanie możesz zaimplementować przez umieszczenie logiki w jednej

Kup ksi k Pole ksi k

z funkcji lambda. W przypadku naprawdę skomplikowanego rozwiązania kontrolnego prawdopodobnie konieczne będzie utworzenie oddzielnej funkcji.

Dotychczas skoncentrowałem się na obsłudze kontynuacji w std::experimental::future. Jak możesz się spodziewać, std::experimental::shared_future również obsługuje kontynuację. Różnica między nimi polega na tym, że obiekt typu std::experimental::shared_future może mieć więcej niż tylko jedną kontynuację, a parametrem kontynuacji jest std::experimental::shared_future zamiast std::experimental::future. To oczywiście kłóci się ze współdzieloną naturą std::experimental::shared_future — ponieważ wiele obiektów może odwoływać się do tego samego współdzielonego stanu. Jeżeli byłaby dozwolona tylko jedna kontynuacja, powstałby stan wyścigu między dwoma wątkami, które próbowałyby dodawać kontynuacje do własnych obiektów std::experimental::shared_future. Takie zachowanie jest niepożądane, stąd możliwość stosowania wielu kontynuacji. Gdy dozwolonych jest wiele kontynuacji, można je dodawać za pomocą tego samego egzemplarza std::experimental::shared_future, zamiast zezwalać na tylko jedną kontynuację dla każdego obiektu. Ponadto nie można umieścić współdzielonego stanu w jednym egzemplarzu std::experimental::future przekazywanym do pierwszej kontynuacji, gdy chcesz go przekazać także drugiej kontynuacji. Dlatego też parametr przekazywany do funkcji kontynuacji również musi być typu std::experimental::shared_future.

```
auto fut=spawn_async(some_function).share();
auto fut2=fut.then([](std::experimental::shared_future<some_data> data){
    do_stuff(data);
});
auto fut3=fut.then([](std::experimental::shared_future<some_data> data){
    return do_other_stuff(data);
});
```

Obiekt fut jest typu std::experimental::shared_future, co wynika z natury wywołania share(), więc funkcja kontynuacji musi pobrać parametr typu std::experimental::shared_future. Natomiast wartością zwrotną kontynuacji jest zwykły obiekt typu std::experimental::future — ta wartość nie jest aktualnie współdzielona, o ile nie wykonasz pewnego kroku, aby to zmienić. Dlatego też obiekty fut2 i fut3 są egzemplarzami std::experimental::future.

Kontynuacja nie jest jedynym usprawnieniem w przyszłości wprowadzonym w specyfikacji technicznej Concurrency TS, choć to prawdopodobnie najważniejsza zmiana. Dostarczone są również dwie przeciążone funkcje pozwalające oczekiwać na *dowolną* z zestawu gotowych przyszłości lub *wszystkich* gotowych przyszłości.

4.4.5. Oczekiwanie na więcej niż tylko jedną przyszłość

Przyjmuję założenie o istnieniu ogromnej ilości danych do przetworzenia, przy czym każdy element może być przetwarzany niezależnie. To jest doskonała okazja do użycia dostępnego sprzętu przez zdefiniowanie przeznaczonych do przetwarzania danych zadań asynchronicznych, z których każde zwróci przetworzone dane za pomocą przyszłości. Jeżeli jednak zachodzi potrzeba poczekania na zakończenie wszystkich zadań i zebrania wszystkich wyników, aby przeprowadzić pewne przetwarzanie końcowe, wówczas wymienione rozwiązanie okaże się niewygodne — trzeba będzie po kolei czekać na

Kup ksi k Pole ksi k

każdą przyszłość i zebrać wyniki. Jeżeli zebranie wyników ma się odbywać za pomocą kolejnego zadania asynchronicznego, trzeba je utworzyć wcześniej, aby zajęło wątek i czekało na swoją kolej, lub zapewnić pulę przyszłości i utworzyć nowe zadanie, gdy wszystkie przyszłości są *gotowe*. Przykład kodu dla takiego rozwiązania przedstawiłem na listingu 4.22.

Listing 4.22. Używanie std::async do pobierania wyników z przyszłości

```
std::future<FinalResult> process_data(std::vector<MyData>& vec)
{
    size_t const chunk_size=cokolwiek;
    std::vector<std::future<ChunkResult>> results;
    for(auto begin=vec.begin(),end=vec.end();beg!=end;){
        size_t const remaining_size=end-begin;
        size_t const this_chunk_size=std::min(remaining_size,chunk_size);
        results.push_back(
            std::async(process_chunk,begin,begin+this_chunk_size));
        begin+=this_chunk_size;
    }
    return std::async([all_results=std::move(results)](){
        std::vector<ChunkResult> v;
        v.reserve(all_results.size());
        for(auto& f: all_results)
        {
            v.push_back(f.get());          ←❶
        }
        return gather_results(v);
    });
}
```

Ten kod tworzy nowe zadanie asynchroniczne oczekujące na wyniki, a następnie przetwarza je, gdy wszystkie są już dostępne. Ponieważ oczekuje na każde zadanie oddzielnie, wątek wykonujący zadanie będzie nieustannie budzony ❶ po pojawieniu się wyniku, a następnie zostanie uśpiony, gdy kolejny wynik nie jest jeszcze gotowy. To nie tylko zabiera jeden wątek, ale jeszcze powoduje wiele dodatkowych operacji przełączania kontekstu, gdy kolejna przyszłość stanie się gotowa, co ogólnie zwiększa obciążenie nakładane przez całe rozwiązanie.

Dzięki `std::experimental::when_all` można uniknąć oczekiwania i przełączania kontekstu. Wystarczy przekazać do `when_all` zbiór oczekiwanych przyszłości, a wartością zwrotną będzie nowa przyszłość, która stanie się gotowa, gdy gotowe będą wszystkie przyszłości w zbiorze. Następnie ta przyszłość może być używana wraz z kontynuacją do zdefiniowania kolejnego zadania do wykonania, gdy wszystkie przyszłości będą już gotowe. Przykład takiego rozwiązania przedstawiłem na listingu 4.23.

Listing 4.23. Pobieranie wyników z przyszłości za pomocą std::experimental::when_all

```
std::experimental::future<FinalResult> process_data(
    std::vector<MyData>& vec)
{
    size_t const chunk_size=cokolwiek;
    std::vector<std::experimental::future<ChunkResult>> results;
```

Kup ksi k Pole ksi k

```
    for(auto begin=vec.begin(),end=vec.end();beg!=end;){
        size_t const remaining_size=end-begin;
        size_t const this_chunk_size=std::min(remaining_size,chunk_size);
        results.push_back(
            spawn_async(
            process_chunk,begin,begin+this_chunk_size));
         begin+=this_chunk_size;
     }
     return std::experimental::when_all(
          results.begin(),results.end()).then(      ←❶
          [](std::future<std::vector<
               std::experimental::future<ChunkResult>>> ready_results)
          {
               std::vector<std::experimental::future<ChunkResult>>
                    all_results=ready_results .get();
               std::vector<ChunkResult> v;
               v.reserve(all_results.size());
               for(auto& f: all_results)
               {
                    v.push_back(f.get());      ←❷
               }
               return gather_results(v);
          });
}
```

W tym przykładzie mamy użyte `when_all()` w oczekiwaniu na zakończenie przygotowywania wszystkich przyszłości. Następnie zdefiniowanie harmonogramu wykonywania funkcji odbywa się za pomocą `.then()`, a nie `async` ❶. Wprawdzie wyrażenie lambda pozostało takie samo, ale pobiera parametr w postaci wektora `results` (opakowanego przyszłością) i wywołuje funkcję `get()` dla tej przyszłości ❷, która nie jest blokująca. Zanim nastąpi wywołanie tej funkcji, wszystkie wartości są już gotowe do użycia. Dzięki przedstawionemu rozwiązaniu można zmniejszyć obciążenie systemu, wprowadzając jedynie niewielkie zmiany w kodzie.

Uzupełnieniem dla wywołania `when_all()` jest `when_any()` powodujące utworzenie przyszłości, gdy *dowolna* z podanych przyszłości stanie się gotowa. Takie rozwiązanie sprawdza się w sytuacjach, gdy utworzonych zostało wiele zadań wykorzystujących dostępną współbieżność i trzeba coś zrobić, gdy pierwsza wartość stanie się gotowa.

4.4.6. Oczekiwanie za pomocą when_any na pierwszą przyszłość w zbiorze

Przyjmuję założenie o przeszukiwaniu ogromnego zbioru danych pod kątem wartości spełniającej określone kryteria, przy czym jeśli istnieje wiele takich wartości, dowolna z nich może zostać użyta. Takie wymaganie jest doskonałym przykładem dla równoległości — można utworzyć wiele wątków, z których każdy sprawdza pewien podzbiór danych. Jeżeli dany wątek znajdzie odpowiednią wartość, ustawia flagę wskazującą, że inne wątki mogą zakończyć operację przeszukiwania, a następnie mamy zwrot ostatecznej wartości. W omawianym przykładzie dalsze przetwarzanie ma odbywać się po zakończeniu wyszukiwania przez pierwsze zadanie, nawet jeśli inne zadania jeszcze nie zakończyły operacji czyszczących.

Kup książkę Poleć książkę

Istnieje możliwość użycia `std::experimental::when_any` do zebrania wszystkich przyszłości i dostarczenia nowej, która jest gotowa po przygotowaniu co najmniej jednego oryginalnego zbioru. Podczas gdy wywołanie `when_all()` dostarcza przyszłość opakowującą kolekcję przekazanych mu przyszłości, połączenie kolekcji wraz z wartością indeksu wskazuje, która przyszłość spowodowała przekazanie gotowej przyszłości do egzemplarza szablonu klasy `std::experimental::when_any_result()`.

Przykład użycia `when_any()` zgodnie z przedstawionym tutaj opisem zaprezentowałem w listingu 4.24.

Listing 4.24. Użycie wywołania std::experimental::when_any do przetworzenia pierwszej dostępnej wartości

```
std::experimental::future<FinalResult>
find_and_process_value(std::vector<MyData> &data)
{
    unsigned const concurrency = std::thread::hardware_concurrency();
    unsigned const num_tasks = (concurrency > 0) ? concurrency : 2;
    std::vector<std::experimental::future<MyData *>> results;
    auto const chunk_size = (data.size() + num_tasks - 1) / num_tasks;
    auto chunk_begin = data.begin();
    std::shared_ptr<std::atomic<bool>> done_flag =
        std::make_shared<std::atomic<bool>>(false);
    for (unsigned i = 0; i < num_tasks; ++i) {         ←❶
         auto chunk_end =
            (i < (num_tasks - 1)) ? chunk_begin + chunk_size : data.end();
        results.push_back(spawn_async([=] {          ←❷
            for (auto entry = chunk_begin;
                !*done_flag && (entry != chunk_end);
                 ++entry) {
                if (matches_find_criteria(*entry)) {
                    *done_flag = true;
                    return &*entry;
                }
            }
            return (MyData *)nullptr;
        }));
        chunk_begin = chunk_end;
    }
    std::shared_ptr<std::experimental::promise<FinalResult>> final_result =
        std::make_shared<std::experimental::promise<FinalResult>>();
    struct DoneCheck {
        std::shared_ptr<std::experimental::promise<FinalResult>>
            final_result;

        DoneCheck(
            std::shared_ptr<std::experimental::promise<FinalResult>>
                final_result_)
            : final_result(std::move(final_result_)) {}

        void operator()(         ←❹
            std::experimental::future<std::experimental::when_any_result<
                std::vector<std::experimental::future<MyData *>>>>
                results_param) {
            auto results = results_param.get();
```

Kup książkę Poleć książkę

```
                MyData *const ready_result =
                    results.futures[results.index].get();      ←5
                if (ready_result)
                    final_result->set_value(          ←6
                        process_found_value(*ready_result));
                else {
                    results.futures.erase(
                        results.futures.begin() + results.index);      ←7
                    if (!results.futures.empty()) {
                        std::experimental::when_any(        ←8
                            results.futures.begin(), results.futures.end())
                            .then(std::move(*this));
                } else {
                    final_result->set_exception(
                        std::make_exception_ptr(         ←9
                            std::runtime_error("Nie znaleziono")));
                }
            }
        };
        std::experimental::when_any(results.begin(), results.end())    ←3
            .then(DoneCheck(final_result));
        return final_result->get_future();      ←10
}
```

Pętla początkowa ❶ tworzy podaną liczbę (`num_tasks`) zadań asynchronicznych, z których każde wykonuje funkcję lambda ❷. Ta funkcja lambda jest pobierana przez kopiowanie, więc każde zadanie będzie miało własne wartości dla `chunk_begin` i `chunk_end`, a także kopię współdzielonego wskaźnika `done_flag`. Dzięki temu można uniknąć wszelkich obaw związanych z cyklem życiowym.

Po utworzeniu wszystkich zadań trzeba się zająć przypadkiem zwróconym przez zadanie. Odbywa się to przez połączenie kontynuacji w wywołaniu `when_any()`❸. Tym razem zapiszesz kontynuację w postaci klasy, ponieważ będziesz chciał jej rekurencyjnie wielokrotnie używać. Gdy jedno z zadań początkowych jest gotowe, następuje wywołanie operatora funkcji `DoneCheck()`❹. Przede wszystkim wyodrębniana jest wartość z gotowej przyszłości ❺ i jeśli wartość będzie znaleziona, należy ją przetworzyć i przygotować wynik końcowy ❻. W przeciwnym razie gotowa przyszłość zostaje usunięta z kolekcji ❼. Jeżeli istnieje więcej przyszłości do sprawdzenia, wykonywane jest kolejne wywołanie `when_any()`❽, które zainicjuje jego kontynuację, gdy następna przyszłość będzie gotowa. Jeżeli nie pozostała żadna przyszłość, to oznacza, że żadna z nich nie zawierała wartości, więc przechowywany będzie wyjątek ❾. Wartość zwrotna funkcji jest przyszłością dla ostatecznego wyniku ❿. Wprawdzie istnieją alternatywne sposoby na rozwiązanie przedstawionego tutaj problemu, ale chciałem w tym miejscu pokazać przykład użycia `when_any()`.

W obu przykładach użycia `when_all()` i `when_any()` zastosowano przeciążenie operatora zakresu — użyta jest para iteratorów określających początek i koniec zbioru oczekiwanych przyszłości. Obie funkcje mają również formy wariadyczne, które akceptują

Kup ksi k Pole ksi k

bezpośrednio pewną liczbę przyszłości jako parametrów funkcji. W tym przypadku wynikiem jest przyszłość przechowująca krotkę (`when_any_result` przechowuje krotkę) zamiast wektora.

```
std::experimental::future<int> f1=spawn_async(func1);
std::experimental::future<std::string> f2=spawn_async(func2);
std::experimental::future<double> f3=spawn_async(func3);
std::experimental::future<
    std::tuple<
        std::experimental::future<int>,
        std::experimental::future<std::string>,
        std::experimental::future<double>>> result=
    std::experimental::when_all(std::move(f1),std::move(f2),std::move(f3));
```

Ten przykład podkreśla ważny aspekt związany z używaniem `when_any()` i `when_all()` — zawsze następuje odejście od egzemplarza `std::experimental::futures` przekazanego w kontenerze, a ponadto wywołania te pobierają parametry poprzez wartość, więc konieczne jest wyraźne przekazanie przyszłości lub wartości tymczasowych.

Czasami oczekiwane zdarzenie zdefiniowane dla zbioru wątków polega na dotarciu do pewnego punktu w kodzie lub przetworzeniu pewnej liczby elementów danych przez te wątki. W takim przypadku lepszym rozwiązaniem może być użycie zasuwy bądź bariery, a nie przyszłości. W następnej sekcji zapoznasz się z zasuwami i barierami dostarczanymi przez specyfikację techniczną Concurrency TS.

4.4.7. Zasuwy i bariery w Concurrency TS

Przede wszystkim trzeba sobie odpowiedzieć na pytanie, co oznacza *zasuwa* lub *bariera*. *Zasuwa* to obiekt synchronizacji, który staje się gotowy, gdy wartość dekrementowanego licznika spadnie do zera. Nazwa wzięła się stąd, że obiekt *zasuwa* dane wyjściowe, a gdy jest gotowy, pozostaje w takiej postaci aż do momentu, gdy zostanie usunięty. Zasuwa jest więc lekkim mechanizmem pozwalającym oczekiwać na serię zdarzeń.

Z kolei *bariera* to komponent synchronizacji wielokrotnego użycia, wykorzystywany do wewnętrznej synchronizacji między zbiorem wątków. Podczas gdy dla zasuwy nie ma znaczenia, dla którego wątku wartość licznika spadła do zera — ten sam wątek może dekrementować licznik wielokrotnie, wiele wątków może raz dekrementować licznik lub możemy mieć do czynienia z połączeniem tych sytuacji — w przypadku bariery każdy wątek może w trakcie cyklu tylko raz dotrzeć do bariery. Gdy dwa wątki dotrą do bariery, pozostaną zablokowane aż do chwili, gdy wszystkie wątki dotrą do bariery i dopiero wówczas nastąpi zwolnienie wszystkich wątków. Następnie bariery można użyć ponownie — wątki znów mogą dotrzeć do bariery i czekać na wszystkie pozostałe wątki do następnego cyklu.

Zasuwa jest znacznie prostsza niż bariera, więc zacznę od przedstawienia typu zasuwy pochodzącego ze specyfikacji technicznej Concurrency TS — `std::experimental::latch`.

4.4.8. Zasuwa typu podstawowego — std::experimental::latch

Egzemplarz `std::experimental::latch` pochodzi z pliku nagłówkowego <experimental/latch>. Gdy konstruujesz `std::experimental::latch`, wartość początkową licznika musisz podać jako jedyny argument konstruktora. Po wystąpieniu zdarzenia, na które czekasz, należy wywołać funkcję `count_down()`, a zasuwa stanie się gotowa, gdy wartość

Kup ksi k Pole ksi k

licznika spadnie do zera. Jeżeli trzeba zaczekać na gotowość zasuwy, można w jej obiekcie wywołać `wait()`. Natomiast jeśli chcesz tylko sprawdzić, czy zasuwa jest gotowa, wówczas możesz wywołać `is_ready()`. W przypadku konieczności odliczania i oczekiwania, aż wartość licznika spadnie do zera, można wywołać `count_down_and_wait()`. Prosty przykład użycia zasuwy pokazałem na listingu 4.25.

Listing 4.25. Oczekiwanie za pomocą std::experimental::latch na zdarzenia

```
void foo(){
    unsigned const thread_count=...;
    latch done(thread_count);          ←❶
    my_data data[thread_count];
    std::vector<std::future<void> > threads;
    for(unsigned i=0;i<thread_count;++i)
        threads.push_back(std::async(std::launch::async,[&,i]{   ←❷
            data[i]=make_data(i);
            done.count_down();         ←❸
            do_more_stuff();           ←❹
    }));
    done.wait();         ←❺
    process_data(data,thread_count);      ←❻
}   ←❼
```

To polecenie przygotowuje egzemplarz `done` wraz z liczbą zdarzeń, na które trzeba oczekiwać ❶, i za pomocą `std::async` tworzy odpowiednią liczbę wątków ❷. Następnie każdy wątek zmniejsza wartość licznika po wygenerowaniu odpowiedniego fragmentu danych ❸, dopóki przetwarzanie nie będzie kontynuowane ❹. Wątek główny może czekać na przygotowanie wszystkich danych przez wstrzymanie zasuwy ❺ przed przystąpieniem do przetwarzania wygenerowanych danych ❻. Przetwarzanie danych ❻ będzie potencjalnie odbywało się równolegle wraz z ostatnimi krokami przetwarzania w poszczególnych wątkach ❹ — nie ma gwarancji, że działanie wątków zostanie ukończone przed uruchomieniem destruktorów `std::future` na końcu funkcji ❼.

Warto zwrócić uwagę na to, że wyrażenie lambda przekazane do `std::async` ❷ przechwytuje wszystko przez referencję z wyjątkiem egzemplarza `i` przechwytywanego przez wartość. To wynika z tego, że `i` to licznik pętli, więc przechwycenie go przez referencję spowodowałoby stan wyścigu danych i niezdefiniowane zachowanie, podczas gdy `data` i `done` to elementy, do których trzeba mieć współdzielony dostęp. Ponadto w omawianym scenariuszu zasuwa jest potrzebna, ponieważ wątki muszą przeprowadzić dodatkowe operacje przetwarzania po przygotowaniu danych. Gdyby nie zasuwa, trzeba by było poczekać na gotowość wszystkich przyszłości, aby mieć pewność co do zakończenia zadań przed przystąpieniem do przetwarzania danych.

W wywołaniu `process_data()`❻ można bezpiecznie uzyskać dostęp do `data` nawet pomimo przechowywania tego obiektu przez zadania działające w innych wątkach. To wynika z tego, że zasuwa to obiekt synchronizacji, więc zmiany widoczne dla wątku wywołującego `count_down()` będą widoczne także dla wątku kończącego wywołanie `wait()` w tym samym obiekcie zasuwy. Formalnie wywołanie `count_down()` jest *zsynchronizowa-*

Kup ksi k Pole ksi k

ne z wywołaniem `wait()` — dokładnie zobaczysz, co to oznacza, gdy w rozdziale 5. przeanalizujesz działające na niskim poziomie ograniczenia synchronizacji.

Poza zasuwami specyfikacja techniczna Concurrency TS udostępnia również bariery — obiekty wielokrotnego użycia służące synchronizacji grupy wątków. Barierę przedstawię dokładniej w następnej sekcji.

4.4.9. Podstawowa bariera — std::experimental::barrier

Specyfikacja techniczna Concurrency TS dostarcza dwa rodzaje barier w pliku nagłówkowym `<experimental/barrier>`: `std::experimental::barrier` i `std::experimental::flex_barrier`. Pierwsza z wymienionych jest znacznie prostsza i dlatego potencjalnie powoduje mniejsze obciążenie, natomiast druga z wymienionych jest znacznie elastyczniejsza, choć potencjalnie prowadzi do znacznie większego obciążenia.

Przyjmuję założenie o istnieniu grupy wątków operujących na pewnych danych. Każdy z nich może przetwarzać dane niezależnie od pozostałych, więc nie jest wymagana synchronizacja, choć wszystkie wątki muszą zakończyć działanie przed rozpoczęciem przetwarzania następnego elementu danych. W omawianym przypadku doskonale sprawdza się użycie `std::experimental::barrier`. Tworzysz barierę wraz z liczbą określającą ilość wątków używanych w grupie synchronizacji. Gdy wątek zakończy przetwarzanie danych, dociera do bariery i czeka na pozostałe wątki grupy, wywołując `arrive_and_wait()` w obiekcie bariery. Wątki w grupie mogą wznowić przetwarzanie i zająć się przetwarzaniem następnego elementu danych bądź też przejść do kolejnego etapu, w zależności od potrzeb.

Podczas gdy zasuwa może być użyta jednokrotnie, więc po osiągnięciu gotowości pozostaje w tym stanie, barierę można wykorzystać wielokrotnie — zwalnia oczekujące wątki, a potem może być wyzerowana i użyta ponownie. Synchronizacja odbywa się tylko *wewnątrz* grupy wątków — dlatego też wątek nie może czekać na gotowość bariery, o ile nie jest jednym z wątków w grupie synchronizacji. Wątek może wyraźnie porzucić grupę za pomocą wywołania `arrive_and_drop()` w barierze. W takiej sytuacji dany wątek nie może dłużej czekać na osiągnięcie gotowości przez barierę, a liczba wątków w następnym cyklu musi być o jeden mniejsza od liczby wątków w bieżącym cyklu.

Listing 4.26. Używanie bariery typu std::experimental::barrier

```
result_chunk process(data_chunk);
std::vector<data_chunk>
divide_into_chunks(data_block data, unsigned num_threads);

void process_data(data_source &source, data_sink &sink) {
    unsigned const concurrency = std::thread::hardware_concurrency();
    unsigned const num_threads = (concurrency > 0) ? concurrency : 2;

    std::experimental::barrier sync(num_threads);
    std::vector<joining_thread> threads(num_threads);

    std::vector<data_chunk> chunks;
    result_block result;

    for (unsigned i = 0; i < num_threads; ++i) {
        threads[i] = joining_thread([&, i] {
```

```
            while (!source.done()) {        ←❻
                if (!i) {       ←❶
                    data_block current_block =
                        source.get_next_data_block();
                        chunks = divide_into_chunks(
                            current_block, num_threads);
                }
                sync.arrive_and_wait();       ←❷
                result.set_chunk(i, num_threads, process(chunks[i]));   ←❸
                sync.arrive_and_wait();       ←❹
                if (!i) {      ←❺
                    sink.write_data(std::move(result));
                }
            }
        });
    }
}    ←❼
```

Na listingu 4.26 przedstawiłem przykład użycia bariery do synchronizacji grupy wątków. Dane pochodzą ze źródła (source) i są przekazywane do wyjścia (sink), ale aby wykorzystać dostępną współbieżność w systemie, każdy blok danych jest dzielony na podaną liczbę fragmentów (num_chunks). To musi się odbyć szeregowo, więc masz blok początkowy ❶ działający jedynie w wątku, dla którego warunek i==0 jest prawdziwy. Następnie wszystkie wątki czekają w barierze na zakończenie działania tego kodu szeregowego ❷, zanim dotrą do fragmentu równoległego, gdzie każdy wątek przetwarza jego pojedynczy fragment kodu i uaktualnia wynik (result) ❸ przed ponowną synchronizacją ❹. Następnie mamy drugi fragment szeregowo wykonywanego kodu, w którym tylko wątek 0 przeprowadza operację zapisu do sink ❺. Wszystkie wątki kontynuują działanie aż do chwili, gdy źródło (source) poinformuje o przetworzeniu wszystkich danych (done) ❻. Zwróć uwagę na to, że w trakcie pętli wątku szeregowa sekcja kodu w dolnej części pętli łączy się z sekcją na początku. To nie stanowi problemu, ponieważ tylko wątek 0 może działać w dowolnej z tych sekcji i wszystkie wątki będą zsynchronizowane podczas pierwszego użycia bariery ❷. Po zakończeniu przetwarzania wątki opuszczają pętlę, a destruktory obiektów joining_thread czekają na zakończenie ich działania na końcu funkcji zewnętrznej ❼ (joining_thread poznałeś w listingu 2.7 w rozdziale 2.).

Ważną kwestią do zapamiętania tutaj jest to, że wywołania arrive_and_wait() znajdują się w tych miejscach kodu, w których jest ważne, aby żaden wątek nie kontynuował działania, dopóki wszystkie wątki nie będą gotowe. W pierwszym punkcie synchronizacji wszystkie wątki oczekują na przybycie wątku 0, a użycie bariery pozwala na wyraźne wyznaczenie linii oczekiwania. W drugim punkcie synchronizacji mamy sytuację odwrotną: wątek 0 czeka na przybycie wszystkich pozostałych wątków, zanim zakończy zapis pełnego wyniku (result) w sink.

Specyfikacja techniczna Concurrency TS nie dostarcza tylko jednego typu bariery. Dlatego też poza std::experimental::barrier masz również std::experimental::flex_
↪barrier, która charakteryzuje się znacznie większą elastycznością. Jednym z przejawów

większej elastyczności jest możliwość uruchomienia ostatniego regionu szeregowego, gdy wszystkie wątki dotrą do bariery, ale jeszcze zanim zostaną zwolnione.

4.4.10. std::experimental::flex_barrier, czyli elastyczniejszy przyjaciel std::experimental:barrier

Interfejs `std::experimental::flex_barrier` różni się od `std::experimental::barrier` tylko pod jednym względem — istnieniem dodatkowego konstruktora pobierającego funkcję ukończenia. Oczywiście pobierana jest także liczba wątków. Wymieniona funkcja jest wykonywana w dokładnie jednym wątku, który dotarł do bariery, już po dotarciu wszystkich wątków do bariery. To oznacza konieczność dostarczenia nie tylko kodu wykonywanego szeregowo, ale również zmianę liczby wątków, które muszą dotrzeć do bariery w następnym cyklu. Liczba wątków może być zmieniona dowolnie, w górę lub w dół względem wartości bieżącej. Programista używający omawianej funkcjonalności powinien zagwarantować, że w następnym cyklu do bariery dotrze odpowiednia liczba wątków.

Na listingu 4.27 przedstawiłem poprzedni program zmodyfikowany do użycia bariery typu `std::experimental::flex_barrier`.

Listing 4.27. Używanie bariery typu std::flex_barrier do dostarczenia obszaru szeregowego

```
void process_data(data_source &source, data_sink &sink) {
    unsigned const concurrency = std::thread::hardware_concurrency();
    unsigned const num_threads = (concurrency > 0) ? concurrency : 2;

    std::vector<data_chunk> chunks;

    auto split_source = [&] {        ←❶
        if (!source.done()) {
            data_block current_block = source.get_next_data_block();
            chunks = divide_into_chunks(current_block, num_threads);
        }
    };

    split_source();        ←❷

    result_block result;

    std::experimental::flex_barrier sync(num_threads, [&] {    ←❸
        sink.write_data(std::move(result));
        split_source();    ←❹
        return -1;    ←❺
    });
    std::vector<joining_thread> threads(num_threads);

    for (unsigned i = 0; i < num_threads; ++i) {
        threads[i] = joining_thread([&, i] {
             while (!source.done()) {        ←❻
                result.set_chunk(i, num_threads, process(chunks[i]));
```

```
                sync.arrive_and_wait();     ←❼
            }
        });
    }
}
```

Pierwsza różnica między omawianym tutaj kodem i tym przedstawionym na listingu 4.26 polega na wyodrębnieniu wyrażenia lambda dzielącego na fragmenty blok następnych danych ❶. To wywołanie następuje jeszcze przed rozpoczęciem operacji ❷ i hermetyzuje kod uruchamiany w wątku 0 na początku każdej iteracji.

Druga różnica polega na tym, że obiektem `sync` jest teraz `std::experimental::flex_barrier`, a ponadto przekazywana jest funkcja ukończenia i liczba wątków ❸. Ta funkcja ukończenia jest wykonywana w jednym z wątków po przybyciu wątku, więc może hermetyzować kod przeznaczony do wykonania w wątku 0 na końcu każdej iteracji. Następnie mamy wywołanie nowo wyodrębnionego wyrażenia lambda `split_source`, które zostanie wykonane na początku następnej iteracji ❹. Wartość zwrotna -1 ❺ wskazuje, że liczba wątków uczestniczących w operacji pozostała niezmieniona. Z kolei wartość zwrotna wynosząca zero lub więcej oznacza liczbę wątków w następnym cyklu.

Pętla główna ❻ jest teraz uproszczona: zawiera jedynie równolegle wykonywany kod i dlatego wymaga tylko pojedynczego punktu synchronizacji ❼. Można stwierdzić, że użycie `std::experimental::flex_barrier` pozwoliło na uproszczenie kodu.

Użycie funkcji ukończenia, aby dostarczyć kod wykonywany szeregowo, oferuje naprawdę potężne możliwości, a także pozwala na zmianę liczby wątków uczestniczących w operacji. Przykładowo omówione rozwiązanie może być wykorzystane przez kod w stylu potoku, w którym liczba wątków jest na początku i na końcu operacji mniejsza niż w trakcie głównego przetwarzania, gdy operują wszystkie etapy potoku.

4.5. Podsumowanie

Synchronizacja operacji pomiędzy wątkami jest ważnym aspektem pisania aplikacji stosującej techniki przetwarzania współbieżnego — w razie braku synchronizacji wątki działałyby całkowicie niezależnie, zatem równie dobrze mogłyby mieć postać odrębnych aplikacji uruchamianych w formie pewnej grupy (z racji pokrewnych zadań). W tym rozdziale omówiłem rozmaite sposoby synchronizacji operacji, w tym podstawowe zmienne warunkowe, przyszłości, obietnice, opakowywane zadania, zasuwy i bariery. Opisałem też sposoby rozwiązywania problemów związanych z synchronizacją, w szczególności programowanie funkcyjne, gdzie każde zadanie generuje wynik wyłącznie na podstawie danych wejściowych (nie uwzględnia zewnętrznego środowiska), przekazywanie komunikatów, gdzie komunikacja pomiędzy wątkami odbywa się za pośrednictwem asynchronicznych komunikatów (wysyłanych przy użyciu podsystemu przekazywania komunikatów) oraz stylu kontynuacji, w którym podane są zadania dla poszczególnych operacji, a system zajmuje się ich harmonogramem.

Skoro omówiłem już wiele wysokopoziomowych rozwiązań dostępnych w języku C++, czas przyjrzeć się odpowiednim elementom niskopoziomowym, dzięki którym opisane mechanizmy mogą funkcjonować — w następnym rozdziale skoncentrujemy się na modelu pamięci języka C++ i operacjach atomowych.

Kup ksi k Pole ksi k

Skorowidz

A

algorytm
blokujący, 248
naturalnie równoległy, 29
obliczania sum częściowych, 343, 344, 346, 347, 350, 352
otwarty na zrównoleglanie, 29
poziom skomplikowania, 386, 387
przystosowany do współbieżności, 29
równoległy, 321, 342, 385, 386
przykład, 392, 394
sortowania szybkiego, 132, 302, 303, 306, 363
rekurencja, 133, 135
równoległego, 134, 135, 136
zachowanie w chwili zgłaszania wyjątku, 386, 387
aplikacja wielowątkowa, *Patrz:* wielowątkowość
argument przekazywanie, 47
przenoszenie, 49, 50
przez referencję, 48, 49, 68, 73

B

bariera, 151, 153, 154, 347
bariera pamięci, 204
biblioteka
ACE, 32
Boost, 32, 445
Boost Thread Library, 32, 445
narzędzi testerskich, 409
standardowa, 34
wątków, 32, 33, 34
blok
catch, 120
try-catch, 44, 121
błąd
kompilacji, 420
lokalizowanie, 400, 401
łączenia, 420
niechciane blokowanie, 398, 399
std::future_errc::broken_promise, 121
sytuacja wyścigu, 398, 399, 400
usuwanie, 397
błądzenie, *Patrz:* uwięzienie

C

cache line, *Patrz:* linia pamięci podręcznej
Communicating Sequential Processes, *Patrz:* CSP
compare/exchange, *Patrz:* operacja porównania-wymiany
Concurrency TS, 141, 142, 151, 153, 176

CSP, 136
czas trwania, *Patrz:* okres

D

daemon thread, *Patrz:* wątek demon
dane
 dzielenie pomiędzy wątki, 309, 343
  rekurencja, 302, 303, 306, 328
  według typu zadania, 307, 308
  wydajność, 310
 fałszywe współdzielenie, 314, 317, 320
 kopiowanie, 416, 418
 projektowanie pod kątem współbieżności, 318, 319, 321
 przenoszenie, 416, 418
 rzadko aktualizowane ochrona, 97
 sąsiedztwo, 315
 struktura, 211
  bez blokad, 248, 249, 250, 251, 252, 295
  bez oczekiwania, 250, 251
  bezpieczeństwo przetwarzania wielowątkowego, 212, 213, 217, 218, 226, 234, 235, 241, 242, 253, 255, 282, 288, 291, 294, 295
  blokująca, 248
  nieblokująca, 162
  projektowanie pod kątem współbieżności, 212, 213, 214, 215, 218, 226, 234, 235, 237, 282, 288, 291, 294, 295, 314, 315
 wejściowe, 311
  iterator, *Patrz:* iterator
 współdzielone, 61, 62, 131, 160, 301, 308, 311, *Patrz też:* dane dzielenie pomiędzy wątki
  fałszywie, *Patrz:* dane fałszywe współdzielenie
  inicjalizacja, 110
  kolejka, 106
  ochrona, 66, 68, 69, 70, 93
  problemy, 62, 63
 wyjściowe, 311
  iterator, *Patrz:* iterator
 zależności, 199, 201
data race, *Patrz:* wyścig danych
deadlock, *Patrz:* zakleszczenie
debugowanie, 397
double-word-compare-and-swap, *Patrz:* operacja porównywania i wymiany podwójnego słowa
drzewo binarne, 237
DWCAS, *Patrz:* operacja porównywania i wymiany podwójnego słowa
dziennik transakcji, 65

E

edytor tekstu, 46
embarrassingly parallel, *Patrz:* algorytm otwarty na zrównoleglanie
epoka zegara, 127
exchange-and-add, *Patrz:* operacja wymiany i dodania

F

false sharing, *Patrz:* dane fałszywe współdzielenie
forward iterator, *Patrz:* iterator postępujący
FP, *Patrz:* programowanie funkcyjne
functional programming, *Patrz:* programowanie funkcyjne
funkcja
 add_to_list, 68
 argument, *Patrz:* argument przekazywanie
 arrive_and_drop, 153
 back, 106
 clear, 166, 168
 compare_exchange_strong, 164, 165, 168, 169, 170, 173, 175
 compare_exchange_weak, 164, 165, 168, 169, 170, 175
 constexpr, 427, 428
  wymagania, 429
 count_down, 151
 count_down_and_wait, 152
 data.pop, 217
 data_cond.wait, 220
 deklarowanie, 41
 detach, 42, 46

distance, 323
domyślna, 421, 422, 423
dostępność, 422
emplace, 107
empty, 70, 71, 72, 106, 110, 217
exchange, 164, 165, 168, 170
fetch_add, 165, 171, 312
fetch_or, 165
fetch_sub, 171, 203
find_entry_for, 240
front, 106
get, 112, 136
get_bucket, 240
get_future, 115
get_id, 57
is_lock_free, 162, 170
is_ready, 152
jako argument funkcji, 68
join, 80
lambda, 104, 105, 133, 140, 146, 374, 430, 431, 435
argument, 432, 433
list_contains, 68
load, 164, 165, 168, 170
lock, 66, 167
lokalna, 41
main, 36, 40
memcmp, 173
memcpy, 172, 173
native_handle, 34
naturalna, 429
notify_one, 105
now, 125
open_connection, 96
paczka parametrów, *Patrz:* paczka parametrów
pop, 73, 107, 257, 288, 295
push, 70, 71, 107, 108, 257, 284, 285, 294, 295
set_exception, 120
share, 123
size, 70, 71, 106
sleep_for, 129, 130
sleep_until, 129, 130
spawn_task, 135
splice, 133, 136
std::async, 112, 113, 114, 135, 311, 328
std::atomic_flag_clear_explicit, 175
std::atomic_is_lock_free, 175
std::atomic_load, 174, 175
std::atomic_load_explicit, 175
std::atomic_signal_fence, 503
std::atomic_thread_fence, 502
std::bind, 49
std::call_once, 96
std::current_exception, 120
std::find, 337
std::for_each, 301, 334
std::kill_dependency, 201
std::lock, 78, 79
std::make_shared, 217
std::move, 89, 122, 134, 418
std::partial_sum, 343
std::ref, 49
std::terminate, 323
std::this_thread::get_id, 58
std::this_thread::sleep_for, 102
std::thread::hardware_concurrency, 55, 56, 303, 306, 310, 311, 334
store, 164, 165, 168, 170, 203
swap, 70, 79
test_and_set, 166, 167, 168
then, 142
top, 70, 72, 73
try_lock, 88
try_lock_for, 129, 130
try_lock_shared_for, 130
try_lock_shared_until, 130
try_lock_until, 129, 130
try_pop, 107, 219, 225, 226, 228, 229, 231
unlock, 66, 88, 91
usunięta, 420, 421
wait, 105, 110
wait_and_dispatch, 453
wait_and_pop, 107, 108, 219
wait_for, 124, 130, 141
wait_until, 124, 130, 141
when_all, 148, 149
when_any, 148, 149
wywołanie blokujące, 248
zegar::now, 124
zmiennoargumentowa, 436

Kup ksi k Pole ksi k

H

hash table, *Patrz:* tablica mieszająca
high contention, *Patrz:* współzawodnictwo wysokie

I

inicjalizacja leniwa, 93, 95
interfejs
  POSIX C, 31, 445
  użytkownika graficzny, 115, 116
invariant, *Patrz:* niezmiennik
iterator, 236, 241, 323, 389
  danych wejściowych, 391
  danych wyjściowych, 391
  dwukierunkowy, 391
  jednokrotnego przebiegu, 57
  losowego dostępu, 391
  poruszający się do przodu, 391
  postępujący, 57

J

język programowania
  Erlang, 136
  Haskell, 131
  Ruby, 158
  Smalltalk, 158

K

klasa, 158
  agregująca, 423
  boost::shared_mutex, 98
  constexpr, 430
  dispatcher, 450, 451, 453
  joining_thread, 53
  std::async, 209, 560
  std::atomic, 163, 165, 172, 506
    specjalizacja, 518, 530
  std::atomic_flag, 503
  std::call_once, 613
  std::chrono ::high_resolution_clock, 125
  std::chrono::duration, 466
  std::chrono::steady_clock, 125, 479
  std::chrono::system_clock, 125, 477
  std::chrono::time_point, 127, 128, 475
  std::condition_variable, 103, 124, 210, 431, 482
  std::condition_variable_any, 103, 210, 490
  std::experimental::barrier, 153, 210
  std::experimental::flex_barrier, 153, 155, 210
  std::experimental::future, 141, 142, 209
  std::experimental::latch, 151, 210
  std::experimental::shared_future, 209
  std::future, 111, 118, 121, 123, 209, 324, 418, 537
    ograniczenia, 121
  std::ifstream, 50
  std::lock, 611
  std::lock_guard, 66, 67, 69, 79, 104, 105, 167, 587
  std::mutex, 66, 99, 174, 208, 420, 429, 562
  std::once_flag, 612
  std::packaged_task, 114, 115, 116, 117, 135, 141, 209, 324, 359, 418, 548
  std::promise, 117, 118, 119, 120, 141, 418, 554
  std::queue, 106
  std::ratio, 614
  std::ratio_equal, 617
  std::ratio_greater, 618
  std::ratio_greater_equal, 619
  std::ratio_less, 618
  std::ratio_less_equal, 619
  std::ratio_not_equal, 617
  std::recursive_mutex, 79, 99, 208, 564
  std::recursive_timed_mutex, 208, 571
  std::scoped_lock, 67, 79, 80, 589
  std::shared_future, 111, 121, 122, 209, 542
  std::shared_lock, 601
  std::shared_lock<std::shared_mutex>, 98
  std::shared_mutex, 208, 575
  std::shared_timed_mutex, 208, 580
  std::thread, 40, 43, 46, 49, 50, 52, 80, 112, 135, 208, 355, 418, 620
  std::thread::id, 57, 58
  std::timed_mutex, 208, 567
  std::try_lock, 611

std::unique_lock, 87, 88, 89, 90, 98, 105, 106, 418, 420, 591
std::unique_ptr, 49, 50
thread_guard, 46, 52
klucz kontenera asocjacyjnego, 58
kod
analizowanie
przez inną osobę, 401
pytania, 402, 403
samodzielne, 401, 402
wyjaśnianie działania, 402
skalowalność, 321, 329, 330
testowanie, 403, 404
wykonywanie spekulatywne, 161
kolejka, 72, 106, 107, 117
bez blokad, 234
gwarantująca bezpieczeństwo przetwarzania wielowątkowego, 218, 222, 234, 235, 282, 288, 291, 294
komunikatów, 137, 447, 453
nieograniczona, 234, 235
ograniczona, 234
wykradanie zadań, 369, 372
komunikacja procesów sekwencyjnych, *Patrz:* CSP
komunikat, 136, 447
implementacja, 453
kolejka, *Patrz:* kolejka komunikatów
otrzymywanie, 449
wysyłanie, 448
konstruktor
domyślny, 57, 424, 427
kopiujący, 418, 419
przenoszący, 418, 419, 421
kontener std::stack, 70
kontynuacja, 142, 146
łączenie, 143, 144, 145

L

linia pamięci podręcznej, 314, 315
lista
dwukierunkowa, 62, 63, 66
gwarantująca bezpieczeństwo przetwarzania wielowątkowego, 241, 242
jednokierunkowa, 222
livelock, *Patrz:* uwięzienie
lock, 88
lock-free programming, *Patrz:* programowanie bez blokad
lookup table, *Patrz:* tablica wyszukiwania
low contention, *Patrz:* współzawodnictwo niskie
l-wartość, 416, 417

M

macierz, 317
podział danych, 318, 319
wątki, 318
mapa projektowanie pod kątem szczegółowych blokad, 237
maszyna stanów, 136, 137, 138
mechanizm przyszłości, 57, *Patrz:* przyszłość
memory barrier, *Patrz:* bariera pamięci
metoda
detach, 44
join, 43, 44, 45
joinable, 44, 45
std::terminate, 46
model
aktorów, 139
pamięci, 158
relacja, *Patrz:* relacja
wydań, 21
muteks, 66, 72, 112, 162, 212, 214, 248, 320, 398, 561
blokowanie, 66, 77, 89, 110, 178, 198, 214, 215, 216, 228, 312, 379
limit czasowy, 124, 129
rekurencyjne, 99
szczegółowość, 70, 90, 91
czytelników-pisarzy, 97
odblokowanie, 103, 216, 230, 248
tworzenie, 66
wirujący, 167, 249, 349
własności przekazywanie, 89
zgrożenia, 68, 69, 70, 77, 80, 81, 83, 87

N

nadsubskrypcja, 56, 135, 310, 316

nagłówek
<algorithm>, 390
<atomic>, 498
<chrono>, 465
<condition_variable>, 481
<future>, 536
<mutex>, 561
<numeric>, 390
<ratio>, 613
<thread>, 619
naruszenie niezmienników, *Patrz:* niezmiennik naruszony
niezmiennik, 62, 213
naruszony, 62, 63, 399
ochrona, 66

O

obiekt, 158
const, 429
czas życia, 158
polityki std::execution::par, 386
porządek modyfikacji, 161
przyszłości, 398
std::string, 48
typ, 158
właściwości, 158
wywoływalny, 43
zapobieganie kopiowaniu, 420
obietnica, 117, 120, 436
ogrodzenie, 204, 205, 206, 207
okres, 125, 126
operacja
atomowa, 160, 161, 165, 173, 175, 252
synchronizacja, 178
nieatomowa, 161
jako atomowa, 162
porządkowanie, 206, 207
odczytu-modyfikacji-zapisu, 192, 197, 198, 201, 203, 312, 320
porównania-wymiany, 168, 169, 250
podwójnego słowa, 173
uzyskiwania, 192
wymiany i dodania, 171
zwalniania, 192
operator przypisania, 165
z kopiowaniem, 172
złożony, 164
oversubscription, *Patrz:* nadsubskrypcja

P

pack expansion, *Patrz:* rozwinięcie paczki
paczka parametrów, 437, 439
wielkość, 439
pamięć
bariera, *Patrz:* bariera pamięci, ogrodzenie
dostęp, 159
model, *Patrz:* model pamięci
odzyskiwanie, 296
podręczna linia, *Patrz:* linia pamięci podręcznej
porządkowanie, 166, 168, 171, 176, 248, 277, 279, 295
koszty, 181, 198
opcje, 165, 181
poprzez wzajemne wykluczanie, 181, 191, 193, 195, 196, 198, 199
przechodniość, 196
sekwencja zwalniania, 201
spójne niesekwencyjnie, 185
spójne sekwencyjnie, 181, 182, 183, 184
wydajność, 181
złagodzone, 181, 185, 187, 190, 191
transakcyjna, 65
wyciek, 74
parameter pack, *Patrz:* paczka parametrów
parametr zmiennoargumentowy, 437
plik nagłówkowy, 40
podział zagadnień, 27
pole bitowe, 158
polityka wykonywania, 385, 387, 391
std::execution::parallel_policy, 386, 388, 392
std::execution::parallel_unsequenced_policy, 386, 389, 392, 393
std::execution::sequenced_policy, 386, 388
wybór, 392
potok, 308
prawo Amdahla, 330

problem
 ABA, 296, 297
 ping-ponga bufora, 312, 313, 366, 367
  koszt, 316
  zapobieganie, 313, 314
proces sekwencyjny komunikacja, *Patrz:* CSP
program wielowątkowy, *Patrz:* wielowątkowość
programowanie
 bez blokad, 65
 funkcyjne, 131, 132
projektowanie obiektowe, 68
przełączanie kontekstu, 23
przestrzeń nazw
 std::chrono, 125, 126
 std::chrono_literals, 126
 std::experimental, 112, 141
 std::this_thread, 629
przetwarzanie współbieżne, *Patrz:* współbieżność
przyszłość, 111, 129, 142, 325
 programowanie funkcyjne, 132
 stan gotowości, 111, 112, 114, 118, 120
 unikatowa, 111
 więcej niż jedna, 146, 147, 148
 współdzielona, 111
pula wątków, 306, 356, 358, 361, 363
 bezpieczeństwo, 362
 koszty, 362
punkt
 przerwania, 384
 w czasie, 127, 128

R

race condition, *Patrz:* sytuacja wyścigu
referencja, 415
 do l-wartości, 416, 417
 do obiektu docelowego, 165, 168
 do r-wartości, 415, 416, 419
 do zmiennej lokalnej, 42
relacja
 poprzedzania, 177, 179, 180, 196
  międzywątkowa, 180
  silnego, 179, 180
  według zależności, 199
 synchronizacji, 177, 178, 201
 wprowadzania zależności, 199
rozwinięcie paczki, 437, 438, 439
równoległość, 27, 29
r-wartość, 416

S

separation of concerns, *Patrz:* podział zagadnień
single responsibility principle, *Patrz:* zasada jednej odpowiedzialności
single-pass input iterator, *Patrz:* iterator jednokrotnego przebiegu
skalowalność, 413
skaner antywirusowy, 331
słownik, 235, *Patrz też:* tablica wyszukiwania
słowo kluczowe
 constexpr, 425, 426, 427, 428, 429
  ignorowanie, 430
 static, 96
 thread_local, 441
software transactional memory, *Patrz:* pamięć transakcyjna
spinlock mutex, *Patrz:* muteks wirujący
standard
 C++ z 1998 r., 31
 C++11, 97, 415, 425
 C++14, 33, 97, 425
 C++17, 33, 53, 57, 67, 315, 385, 425
 C++20, 53
steady clock, *Patrz:* zegar stabilny
STM, *Patrz:* pamięć transakcyjna
stos, 70
 chroniony przez muteks, 72
 gwarantujący bezpieczeństwo przetwarzania wielowątkowego, 215, 219, 235, 253, 255
 współdzielony, 71
sytuacja wyścigu, 64, 70, 71, 160, 213, 376, 399
 błąd, *Patrz:* błąd sytuacja wyścigu
 debugowanie, 65, 400
 problematyczna, 64, 65
 wykrywanie, 70
 zapobieganie, 73, 74, 75, 80, 176

szablon
 wariadyczny, 79
 zmiennoargumentowy, 436, 437

Ś

środowisko testowe, 404

T

tablica, 235, 236, 237
task switching, *Patrz:* przełączanie zadań
test
 planowanie, 405, 410
 siłowy, 407, 408
 struktura, 404
testowanie, 397, 401
 scenariusz, 404, 410
 struktura kodu, 405, 410
 symulowanych kombinacji, 408
 środowisko, *Patrz:* środowisko testowe
 wydajności, 413
train model, *Patrz:* model wydań
transakcja, 65
tryb std::memory_order_seq_cst, 295
typ
 atomowy, 158, 160, 162, 164, 500
  bez blokad, 162, 163
  całkowitoliczbowy, 167, 172
 automatyczne określanie, 440, 442
 całkowitoliczbowy, 164
 char, 159
 int, 159
 stały, 426
 std::atomic_, 500
 std::atomic_flag, 162, 163, 165, 166, 167, 170, 175
 std::atomic<bool>, 167, 168, 170
 std::atomic<T*>, 170, 171
 std::atomic<UDT>, 172
 std::chrono::high_resolution_clock, 481
 std::experimental::atomic_shared_ptr<T>, 176
 std::memory_order, 165, 501
 std::ratio, 615, 616
 std::ratio_add, 615
 std::ratio_divide, 616
 std::ratio_multiply, 616
 std::shared_ptr, 221
 std::size_t, 164
 UDT, 172
 wrapped_message, 447

U

uwięzienie, 251, 398, 399, 400

W

warstwa abstrakcji, 33
wątek, 40, 307
 budzenie, 220
  pozorne, 105
 czas życia, 400
 demon, 46
 funkcja początkowa, 36
 główny, 359
 graficznego interfejsu użytkownika, 115, 116
 hierarchia, 87
 identyfikator, 57, 58
 liczba współbieżnych, 55, 56
 łączenie, 308
 oczekiwanie
  na zakończenie, 43
  w razie wyjątku, 44
 odłączenie, 46
 początkowy, 58
 przerwanie wykonywania, 372, 373, 382, 383
  wykrywanie, 375
 pula, *Patrz:* pula wątków
 punkt przerwania, 373, 375
 roboczy, 356
 sprzętowy, 24
 szeregowanie, 213, 218
 uruchamianie, 40, 358
 uśpienie, 102, 103, 129, 147
 uwięzienie, *Patrz:* uwięzienie
 własności przenoszenie, 50, 51, 52
 współdzielenie danych, *Patrz:* dane współdzielone
 wykradanie zadań, 368, 370, 372
 zagłodzenie, 250
 zmienna lokalna, *Patrz:* zmienna lokalna wątku

Kup ksi k Pole ksi k

wektor, 73
węzeł
odzyskiwanie, 269
zliczanie referencji, 271, 273, 276
wyciek, 257, 262
wielowątkowość, 22, 157, 159, 180, 300, 403, 445
historia w języku C++, 31, 32
optymalizacja, 330, 331, 332, 334, 337, 343
wydajność
liczba procesorów, 310, 311, 315, 316
liczba wątków, 311, 315, 316, 317
wielozadaniowość, 23
wskaźnik, 68
do elementu zdjętego ze stosu, 74
do zmiennej lokalnej, 42
niejawna konwersja, 48
ryzyka, 262, 268, 269, 271
std::shared_ptr, 75
współbieżność, 22, 27, 31, 158, 300, 334, 384, 404, 445
eliminowanie, 406
iluzja, 23
model, 25
z wieloma procesami, 25, 26
z wieloma wątkami, 26, 27
ograniczenia, 30
podział zagadnień, 27
sprzętowa, 23
struktura danych, 212
wpływ na wydajność oprogramowania, 29, 31
wydajność, 27, 28
liczba procesorów, 310, 311, 315, 316
liczba wątków, 311, 315, 316, 317
zagrożenia, 77
zalety, 27
zwiększanie poprzez oddzielanie danych, 224, 226
współzawodnictwo, 306, 317
niskie, 312
skutki, 313
wysokie, 312, 313
wyjątek, 44, 213
bezpieczeństwo, 321, 323, 328, 342
brak elementów na stosie, 71
std::bad_alloc, 73
std::future_error, 121
w obiekcie przyszłości, 120, 121
wyrażenie
lambda, 41, 431
typ, 432
stałe, 425, 428
typ, 426
wynik przechwytywanie uogólnione, 435
wyścig, 64
danych, 64, 94, 121, 160, 161, 399
wzorzec projektowy
Double-Checked Locking, 94
RAII, 32, 45, 87, 98
dla muteksu, 66

Z

zadanie
anulowanie, 308
asynchroniczne, 112
przełączanie, 315, 316
wykonywane w tle, 112, 308
zakleszczenie, 66, 77, 213, 215, 217, 248, 363, 398, 399, 400
unikanie, 78, 79, 80, 81, 83, 87
zasada jednej odpowiedzialności, 307
zasuwa, 151, 152, 153
zdarzenie, 332
seria, 151
zegar, 124
epoka, *Patrz:* epoka zegara
stabilny, 124, 125
systemowy, 125
takt, 124, 125
złączenie, 43, 45
zmienna, 159
lokalna
przechwycenie, 432, 433, 434
wątku, 441, 442
nazwana, 41
składowa static constexpr, 162
tymczasowa, 41
warunkowa, 103, 105, 110, 129, 398
implementacja, 103

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

Programuj elegancko, twórz wydajny i czysty kod. Oto współbieżność w C++!

Jeśli aplikacja ma działać szybko i niezawodnie, najlepiej wybrać C++, dojrzały i wszechstronny język programowania, konsekwentnie rozwijany przez mistrzów kodowania. Wymaga on zachowania pewnej dyscypliny podczas pracy, jednak pozwala na uzyskanie kodu o znakomitej wydajności. Nowy standard C++17 zapewnia doskonałą obsługę wielowątkowości oraz programowania wieloprocesorowego wymaganego podczas szybkiego przetwarzania grafiki, uczenia maszynowego czy też wykonywania innych zadań, w których kluczową sprawą okazuje się wydajność.

Ta książka jest drugim, zaktualizowanym i uzupełnionym wydaniem doskonałego podręcznika dla profesjonalistów. Szczegółowo opisano w niej wszystkie etapy programowania współbieżnego: od utworzenia wątków po projektowanie wielowątkowych algorytmów i struktur danych. Przedstawiono zastosowania klas std::thread i std::mutex oraz funkcji std::async, a także złożone zagadnienia związane z operacjami atomowymi i modelem pamięci. Sporo miejsca poświęcono diagnozowaniu kodu i analizie rodzajów błędów. Opisano techniki lokalizowania błędów oraz metody testowania kodu. Prezentowany materiał został uzupełniony przykładami kodu i praktycznymi ćwiczeniami. Znalazły się tu również porady i wskazówki, które docenią wszyscy programiści C++.

W tej książce między innymi:

1. nowości w standardzie C++17
2. zagadnienia niskiego poziomu: model pamięci i operacje atomowe
3. zagadnienia wyższego poziomu: złożone struktury danych
4. zagadnienia wysokiego poziomu: kod wielowątkowy i zarządzanie wątkami
5. obsługa równoległości za pomocą funkcji dodanych w standardzie C++17
6. debugowanie kodu wielowątkowego

Anthony Williams od ponad dwudziestu lat programuje w C++, a także udziela porad i szkoli w tym zakresie. Od 2001 roku jest aktywnym członkiem organizacji BSI C++ Standards Panel. Brał udział w opracowywaniu i implementacji standardów C++. Kontynuuje prace nad rozszerzaniem pakietu współbieżności w tym języku.