Ekstremalny kod w języku C

Współbieżność i programowanie zorientowane obiektowo

Kamran Amini

Helion

Packt

Tytuł oryginału: Extreme C: Taking you to the limit in Concurrency, OOP, and the most advanced capabilities of C

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-7459-1

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/ekskod
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/ekskod.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 13

Współbieżność

W dwóch kolejnych rozdziałach zamierzam zająć się *współbieżnością*. Teoria dotycząca tej technologii jest wymagana podczas tworzenia programów współbieżnych nie tylko w C, ale także w innych językach programowania. Dlatego też w tych dwóch rozdziałach nie zamieściłem żadnego rzeczywistego kodu w języku C, a jedynie pseudokod przedstawiający systemy współbieżne oraz ich wewnętrzne właściwości.

Ze względu na dużą objętość temat współbieżności podzieliłem na dwa rozdziały. W tym przedstawię podstawowe koncepcje związane z samą współbieżnością, natomiast w rozdziale 14. przejdę do omówienia problemów związanych ze współbieżnością i zaprezentuję mechanizmy *synchronizacji* używane w programach współbieżnych do rozwiązywania wspomnianych kwestii. Ostatecznym celem tych dwóch rozdziałów jest dostarczenie Ci wiedzy wystarczającej do poradzenia sobie z tematami wielowątkowości i wieloprocesowości w kolejnych rozdziałach.

Wiedza przedstawiona w rozdziale przyda się także podczas pracy z *biblioteką wątków POSIX*, z której będziemy korzystać w książce.

W pierwszym rozdziale dotyczącym współbieżności zostaną omówione następujące tematy:

- Czym różnią się systemy równoległe od współbieżnych.
- Dlaczego w ogóle potrzebujemy współbieżności.
- Czym jest *zarządca procesów* i jakie są najczęściej stosowane przez niego algorytmy.
- Jak działa program współbieżny i czym jest przeplatanie.
- Co to jest stan współdzielony i jak różne zadania mogą uzyskać do niego dostęp.

Rozpoczynam omawianie współbieżności od zaprezentowania ogólnej koncepcji i jej znaczenia z perspektywy programisty.

Wprowadzenie do współbieżności

Współbieżność oznacza po prostu istnienie w programie wielu fragmentów logiki i ich jednoczesnego wykonywania. Nowoczesne systemy oprogramowania są często współbieżne, ponieważ programy wymagają jednoczesnego działania różnych fragmentów logiki. Dlatego też współbieżność to technologia, która przynajmniej do pewnego stopnia jest używana przez każdy obecnie tworzony program.

Można powiedzieć, że współbieżność to narzędzie o potężnych możliwościach, pozwalające na tworzenie programów, które mogą jednocześnie zarządzać wieloma różnymi zadaniami. Obsługa współbieżności zwykle zawarta jest w jądrze, czyli w sercu systemu operacyjnego.

Istnieje wiele przykładów pokazujących, jak zwykły program zarządza jednocześnie wieloma zadaniami. Na przykład można przeglądać strony internetowe i jednocześnie pobierać pliki. W takim przypadku zadania są wykonywane współbieżnie w kontekście procesu przeglądarki WWW. Kolejnym przykładem może być *strumieniowanie pliku wideo*, np. sytuacja, z którą masz do czynienia, oglądając film opublikowany w serwisie YouTube. Odtwarzacz wideo może być w środku procesu pobierania kolejnych fragmentów klipu, podczas gdy użytkownik nadal ogląda wcześniej pobrany fragment.

Nawet proste oprogramowanie typu procesor tekstu może mieć wiele współbieżnych zadań działających w tle. W chwili gdy piszę te słowa w aplikacji Microsoft Word, w tle działa moduł sprawdzania pisowni i formatowania tekstu. Jeżeli tę książkę czytasz w urządzeniu takim jak Kindle lub iPad, to jak sądzisz, które programy mogą działać współbieżnie jako część programu Kindle?

Duża ilość jednocześnie działających programów wydaje się być zadziwiającym osiągnięciem. Choć jednak współbieżność wykorzystuje najnowszą dostępną technologię, to oprócz zalet ma też wiele wad. Tak naprawdę współbieżność przyniosła jedne z najboleśniejszych problemów w historii informatyki. Te bolączki, do których powrócę w dalszej części rozdziału, mogą pozostać niewidoczne przez długi czas, nawet wiele miesięcy po wydaniu oprogramowania, a ponadto z reguły są trudne do znalezienia, powielenia i rozwiązania.

Rozpocząłem ten podrozdział od zdefiniowania współbieżności jako zadań wykonywanych w tym samym czasie, czyli współbieżnie. Taka definicja może wskazywać na równoległe wykonywanie zadań, choć nie do końca tak jest. Przedstawiona definicja jest zbyt prosta i niedokładna, ponieważ *współbieżność różni się od równoległości*, a różnice między nimi jeszcze nie zostały wyjaśnione. Dwa programy współbieżne różnią się od dwóch programów równoległych. Jednym z celów tego rozdziału jest pokazanie tych różnic, a także zaprezentowanie definicji, z którymi można spotkać się w oficjalnej literaturze.

W następnych podrozdziałach zamierzam przedstawić pewne podstawowe koncepcje związane ze współbieżnością, takie jak *zadania*, *szeregowanie*, *przeplatanie*, *stan* i *stan współdzielony*, ponieważ są to pojęcia, które będziesz dość często spotykać podczas lektury książki. Warto w tym miejscu dodać, że większość tych koncepcji jest abstrakcyjnych i może być zastosowanych względem dowolnego systemu współbieżnego, nie tylko w języku programowania C.

Aby poznać różnice między równoległością i współbieżnością, w następnym podrozdziale zaprezentuję w skrócie systemy równoległe.

W rozdziale pozostanę przy prostych definicjach. Moim celem jest tutaj przedstawienie ogólnego sposobu działania systemu współbieżnego, ponieważ dokładniejsze omówienie tego zagadnienia wykracza poza zakres tematyczny książki.

Równoległość

Równoległość oznacza, że dwa zadania działają w tym samym czasie, czyli *równolegle*. Słowo *równolegle* ma tutaj znaczenie kluczowe podczas odróżniania równoległości od współbieżności. Dlaczego tak się dzieje? Ponieważ równoległość wskazuje na to, że dwa zadania są wykonywane jednocześnie. To nie jest cecha systemu współbieżnego, w którym trzeba wstrzymać jedno zadanie, aby móc zezwolić innemu na kontynuowanie działania. Trzeba w tym miejscu dodać, że ta definicja może być zbyt uproszczona i niepełna w stosunku do nowoczesnych systemów współbieżnych, ale jest wystarczająca, aby pokazać, o co chodzi.

Z równoległością spotykasz się na co dzień. Gdy Ty i Twój kolega jednocześnie wykonujecie dwa oddzielne zadania, wówczas są one realizowane równolegle. Aby pewna liczba zadań była wykonywana równolegle, potrzebne są oddzielne i odizolowane *jednostki przetwarzania*, z których każda jest przypisana określonemu zadaniu. Na przykład w komputerze każdy *rdzeń* procesora jest jednostką procesora, która w danej chwili może wykonywać jedno zadanie.

Przez chwilę spójrz na siebie jako na jedynego czytelnika tej książki. Nie możesz czytać dwóch książek jednocześnie, musisz przerwać czytanie jednej, aby móc czytać inną. Jeżeli dołączy do Ciebie kolega, wówczas istnieje możliwość, że dwie książki będą czytane równolegle.

Co się stanie w sytuacji, gdy pojawi się trzecia książka, którą trzeba przeczytać? Skoro żaden z Was nie może czytać równolegle dwóch książek, więc jeden z Was musi odłożyć na bok bieżącą książkę i zająć się czytaniem tej trzeciej. To oznacza, że Ty lub Twój kolega musicie odpowiednio podzielić czas, aby razem przeczytać trzy książki.

W komputerze muszą istnieć przynajmniej dwie oddzielne i niezależne jednostki przetwarzania, aby możliwe było równoległe wykonywanie dwóch zadań. Nowoczesne procesory zawierają pewną liczbę *rdzeni*, które są rzeczywistymi jednostkami przetwarzania. Na przykład procesor czterordzeniowy ma cztery jednostki przetwarzania, a tym samym pozwala na jednoczesne wykonywanie 4 równoległych zadań. W celu zachowania prostoty w rozdziale przyjąłem założenie, że nasz wyimaginowany procesor zawiera tylko jeden rdzeń, więc nie może równolegle wykonywać zadań. W dalszej części rozdziału znajdziesz więcej informacji na temat procesorów wielordzeniowych.

Przyjmuję założenie, że masz dwa laptopy z naszym jednordzeniowym procesorem. Jeden z nich odtwarza muzykę, zaś drugi jest używany do rozwiązania równania różniczkowego. Oba te laptopy działają równolegle, ale jeśli chcesz, aby te zadania były wykonywane w jednym

laptopie wyposażonym w procesor jednordzeniowy, wówczas *nie będzie* możliwości równoległego wykonywania tych zadań.

Równoległość wiąże się z zadaniami, które mogą być wykonywane równolegle. To oznacza, że rzeczywisty algorytm będzie można podzielić, a następnie uruchomić w wielu jednostkach przetwarzania. Jednak większość obecnie tworzonych algorytmów jest z natury *sekwencyjnych*, a nie równoległych. Nawet w przypadku wielowątkowości każdy z wątków otrzymuje pewną liczbę instrukcji sekwencyjnych, których nie można podzielić na pewnego rodzaju równoległe *procesy wykonywania*.

Innymi słowy algorytm sekwencyjny nie może w prosty sposób zostać automatycznie podzielony przez system operacyjny na kilka równolegle wykonywanych — to jest zadanie dla programisty. Dlatego też nawet jeśli masz wielordzeniowy procesor, nadal każdemu rdzeniowi trzeba przekazywać odpowiednią jednostkę wykonywania zadania. Jeżeli rdzeniowi przypiszesz więcej takich jednostek, one nie będą mogły być wykonywane równolegle i natychmiast zauważysz zachowanie współbieżne.

Ujmując rzecz najkrócej, jeżeli dwie jednostki wykonywania zostaną przypisane dwóm oddzielnym rdzeniom procesora, będą one wykonywane równolegle. Natomiast dwie jednostki wykonywania przypisane jednemu rdzeniowi procesora będą wykonywane współbieżnie. W procesorach wielordzeniowych można zaobserwować mieszane zachowanie, czyli równoległość między rdzeniami i współbieżność tego samego rdzenia.

Pomimo prostego wyjaśnienia i wielu przykładów zaczerpniętych z rzeczywistości równoległość to skomplikowany i ciężki temat w architekturze komputerowej. Tak naprawdę jest to dziedzina poddawana badaniom akademickim, na temat której powstało wiele prac, książek i różnej literatury. Nadal otwarta jest kwestia posiadania systemu operacyjnego, który będzie umiał podzielić algorytm sekwencyjny na równolegle wykonywane jednostki — ta kwestia wciąż jest przedmiotem badań, a obecnie istniejące systemy operacyjne nie mają takich możliwości.

Jak już wcześniej wspomniałem, w rozdziale nie zamierzam zbyt dokładnie omawiać tematu równoległości, a jedynie ograniczę się do dostarczenia początkowej definicji dla tej koncepcji. Dogłębne wyjaśnienie równoległości wykracza poza zakres tematyczny książki, dlatego przechodzę teraz do koncepcji współbieżności.

Przede wszystkim chciałbym poruszyć temat systemów współbieżnych i wyjaśnić, co to tak naprawdę oznacza w porównaniu z równoległością.

Współbieżność

Prawdopodobnie słyszałeś o *wielozadaniowości* — współbieżność można uznać za tę samą ideę. Jeżeli Twój system zarządza jednocześnie wieloma zadaniami, powinieneś zrozumieć, że to niekoniecznie oznacza równoległe wykonywanie tych zadań. Zamiast tego mamy tzw. *zarządcę zadań* (ang. *task scheduler*), który bardzo szybko przełącza się między poszczególnymi zadaniami i wykonuje niewielką część każdego z nich przez bardzo krótki okres czasu.

Z takim rozwiązaniem zdecydowanie mamy do czynienia w przypadku istnienia tylko jednej jednostki procesora. Na potrzeby dalszej analizy w rozdziale przyjmuję założenie, że działamy tylko w jednej jednostce procesora.

Jeżeli zarządca zadań będzie wystarczająco *szybki* i *sprawiedliwy*, wówczas nie zauważysz żadnego *przełączania* między zadaniami, więc z Twojej perspektywy będzie się wydawało, że są one wykonywane równolegle. To jest właśnie magia współbieżności i zarazem powód, dla którego współbieżność jest powszechnie stosowana w najpopularniejszych systemach operacyjnych, takich jak Linux, macOS i Microsoft Windows.

Współbieżność może być postrzegana jako symulacja równoległego wykonywania zadań za pomocą pojedynczej jednostki procesora. W rzeczywistości cała idea może być określana jako forma sztucznej równoległości. W przypadku starszych komputerów wyposażonych w jeden procesor jednordzeniowy takie rozwiązanie miało ogromną zaletę, ponieważ użytkownik miał możliwość używania tego rdzenia do symulacji wielozadaniowości.

Trzeba w tym miejscu dodać, że *Multics* był jednym z pierwszych systemów operacyjnych zaprojektowanych do obsługi wielozadaniowości i zarządzania jednocześnie działającymi procesami. Z lektury rozdziału 10. powinieneś pamiętać, że system UNIX został zbudowany na podstawie idei zaczerpniętych z projektu Multics.

Jak już wcześniej wyjaśniłem, praktycznie wszystkie systemy operacyjne mają możliwość współbieżnego wykonywania zadań dzięki wielozadaniowości. To dotyczy zwłaszcza systemów operacyjnych zgodnych ze standardem POSIX, ponieważ ta możliwość została wyraźnie udostępniona w wymienionym standardzie.

Jednostka zarządcy zadań

Jak już wcześniej wspomniałem, wszystkie systemy operacyjne zapewniające wielozadaniowość muszą mieć w jądrze jednostkę *zarządcy zadań* nazywaną również *jednostką szeregowania*. W tym podrozdziale pokażę, jak ta jednostka działa i jaki ma wkład na bezproblemowe wykonywanie pewnych zadań współbieżnych.

Oto wybrane fakty dotyczące jednostki zarządcy zadań:

- Posiada *kolejkę* zadań przeznaczonych do wykonania. Tzw. *zadania* to po prostu fragmenty pracy, która musi być wykonana w oddzielnych jednostkach wykonywania.
- Zadania umieszczone w kolejce zwykle mają przydzielany *priorytet*, a zadania o wyższym priorytecie są zwykle wykonywane jako pierwsze.
- Jednostka procesora jest zarządzana i współdzielona między zadania za pomocą zarządcy zadań. Gdy jednostka procesora nie jest zajęta (nie wykonuje żadnego zadania), wówczas zarządca zadań musi wybrać kolejne zadanie z kolejki, zanim będzie można użyć jednostki procesora. Po zakończeniu zadania następuje zwolnienie jednostki procesora i jej ponowne udostępnienie, a zarządca zadań wybiera kolejne zadanie do wykonania. Te operacje są wykonywane w pętli. Cały proces nosi nazwę *szeregowania zadań* i jest wykonywany jedynie przez zarządcę zadań.

- Istnieje wiele *algorytmów szeregowania* używanych przez zarządcę zadań, a każdy z tych algorytmów spełnia określone wymagania. Na przykład wszystkie powinny być *sprawiedliwe*, żadne z nich nie powinno być *zapomniane* w kolejce jako wynik jego niewybierania przez zbyt długi czas.
- Na podstawie wybranej strategii szeregowania zarządca zadań powinien wyznaczać określony *przedział czasu*, w trakcie którego zadanie może korzystać z jednostki procesora. Ewentualnie zarządca zadań musi czekać, aż zadanie zwolni jednostkę procesora.
- Jeżeli strategia szeregowania jest z *wywłaszczeniem*, zarządca zadań powinien mieć możliwość wymuszonego odebrania jednostki procesora od zadania, aby ta jednostka mogła zająć się wykonywaniem następnego zadania. To nosi nazwę *szeregowania z wywłaszczeniem*. Istnieje także inne rozwiązanie, w którym zadanie dobrowolnie zwalnia jednostkę procesora — wówczas mówimy o *szeregowaniu współpracującym*.
- Algorytmy szeregowania z wywłaszczeniem próbują współdzielić *przedziały czasu* równo i sprawiedliwie między poszczególnymi zadaniami. Charakteryzujące się większym priorytetem zadania mogą być wybierane znacznie częściej lub mogą otrzymywać dłuższe przedziały czasu, w zależności od sposobu implementacji zarządcy procesów.

Zadanie to ogólna koncepcja abstrakcyjna, używana w celu odwołania się do dowolnego fragmentu pracy, która powinna zostać wykonana w systemie współbieżnym, niekoniecznie w komputerze. Wkrótce dowiesz się, czym tak dokładnie są systemy inne niż komputerowe. Procesor nie jest jedynym typem zasobu, który może być współdzielony między zadaniami. Ludzie od samego początku istnienia szeregują zadania i nadają im priorytety, gdy stoją przed zadaniami, których nie można wykonać jednocześnie. W kilku następnych akapitach przedstawię dobry przykład pomagający w zrozumieniu szeregowania.

Przyjmijmy założenie, że znajdujemy się na początku XX wieku i na ulicy znajduje się tylko jedna budka telefoniczna, a 10 osób stoi w kolejce, aby zatelefonować. W takim przypadku te 10 osób stosuje algorytm szeregowania w celu sprawiedliwego współdzielenia tej budki telefonicznej między sobą.

Przede wszystkim wszyscy muszą stać w kolejce. To jest najprostsza decyzja, która przychodzi na myśl każdemu cywilizowanemu człowiekowi — ustawienie się w kolejce i zaczekanie na swoją kolej. Jednak to jest niewystarczające i potrzebne są pewne regulacje wspomagające tę metodę. Pierwsza osoba korzystająca z telefonu nie może rozmawiać dowolnie długo, gdy dziewięć innych osób czeka przed budką telefoniczną. Dlatego też pierwsza osoba musi opuścić budkę po upływie określonego czasu, aby kolejna osoba z kolejki mogła skorzystać z telefonu.

W tych rzadkich sytuacjach, gdy pierwsza osoba nie zakończyła konwersacji, po upływie określonego czasu powinna przestać używać telefonu, opuścić budkę i stanąć na końcu kolejki. Następnie musi czekać na swoją kolej, aby mogła kontynuować rozmowę. W ten sposób każda z dziesięciu osób będzie mogła powtarzać tę operację i kolejno wchodzić do budki telefonicznej aż do chwili zakończenia ich rozmów.

To jest tylko przykład. Każdego dnia spotykamy się z przykładami współdzielenia zasobów między pewną liczbą konsumentów. Człowiek wynalazł wiele sposobów pozwalających na sprawiedliwe współdzielenie tych zasobów — przynajmniej do stopnia, na jaki pozwala natura ludzka. W następnym podrozdziale powrócę do tematu szeregowania z udziałem komputera.

Procesy i wątki

W książce interesuje nas przede wszystkim szeregowanie zadań w systemach informatycznych. W przypadku systemu operacyjnego zadaniami są *procesy* lub *wątki*. Różnice między nimi przedstawię w kolejnych rozdziałach, a teraz wystarczy wiedzieć, że większość systemów operacyjnych traktuje je praktycznie w taki sam sposób: jako zadania wymagające wykonywania współbieżnego.

System operacyjny musi stosować zarządcę zadań, aby mieć możliwość współdzielenia rdzeni procesora między wieloma zadaniami (procesy lub wątki), które chcą skorzystać z procesora. Po utworzeniu nowego procesu lub wątku zostaje on umieszczony w kolejce szeregowania jako nowe zadanie i oczekuje na uzyskanie dostępu do rdzenia procesora, zanim to nowe zadanie będzie mogło być uruchomione.

W przypadku *współdzielenia czasu* lub zastosowania zarządcy *z wywłaszczeniem*, jeżeli zadanie nie będzie w stanie wykonać swojej logiki w trakcie określonego czasu, straci dostęp do procesora i ponownie trafi do kolejki, podobnie jak w przykładzie budki telefonicznej, gdy osoba musiała ją opuścić i udać się na koniec kolejki.

Wówczas zadanie powinno oczekiwać w kolejce na ponowne uzyskanie dostępu do procesora i wtedy będzie mogło kontynuować działanie. Gdy i tym razem nie uda się zakończyć wykonywania logiki, ten sam proces jest powtarzany aż do chwili zakończenia zadania.

Za każdym razem gdy zarządca z wywłaszczeniem wstrzymuje działanie procesu i wznawia wykonywanie innego, mówimy o tzw. *przełączeniu kontekstu*. Im szybciej następuje przełączanie kontekstu, tym większe odczucie u użytkownika, że zadania są wykonywane równolegle. Za interesujące można uznać to, że obecnie większość systemów operacyjnych używa zarządcy z wywłaszczeniem, na czym skoncentruję się w pozostałej części rozdziału.

> **Uwaga!**
>
> Począwszy od tego momentu, przyjmujemy założenie, że każdy zarządca jest z wywłaszczeniem. Wyraźnie zaznaczę, gdy będzie inaczej.

Podczas wykonywania zadania może dochodzić do setek lub nawet tysięcy operacji przełączania kontekstu, zanim dane zadanie zostanie zakończone. Jednak przełączanie kontekstu ma bardzo dziwną i unikatową cechę charakterystyczną — jest *nieprzewidywalne*. Innymi słowy nie ma możliwości przewidzenia, kiedy lub na jakiej instrukcji dojdzie do przełączenia kontekstu. Nawet w przypadku dwóch względnie bliskich, kolejnych uruchomień programu na tej samej platformie operacje przełączania kontekstu będą zachodziły inaczej.

To jest bardzo ważna cecha, której wpływ pozostaje niezwykle istotny. Przełączania kontekstu nie da się przewidzieć. Wkrótce na podstawie omawianych przykładów będziesz mógł samodzielnie obserwować konsekwencje tej operacji.

Przełączanie kontekstu jest nieprzewidywalne aż do tego stopnia, że najlepszym sposobem na poradzenie sobie z tą niepewnością jest przyjęcie założenia o takim samym prawdopodobieństwie jego wystąpienia dla każdej instrukcji. Innymi słowy należy się spodziewać, że przełączenie kontekstu może wystąpić podczas wykonywania każdej instrukcji. Mając to na uwadze, można się spodziewać występowania przerw między wykonywaniem dwóch kolejnych instrukcji.

Przechodzę teraz do omówienia jedynych pewników istniejących w środowisku współbieżnym.

Ograniczenie typu „zachodzi wcześniej”

W poprzednim podrozdziale dowiedziałeś się, że operacja przełączania kontekstu jest nieprzewidywalna. Nie ma absolutnie żadnej pewności dotyczącej tego, kiedy może ona wystąpić w programie. Mimo to mamy pewność, że instrukcje są wykonywane współbieżnie.

Przechodzimy teraz do prostego przykładu. Rozpoczynamy od pracy na podstawie zadania, takiego jak pokazane na listingu 13.1 i składającego się z pięciu instrukcji. Warto w tym miejscu dodać, że te instrukcje są całkowicie abstrakcyjne i nie przedstawiają żadnych rzeczywistych instrukcji języka C lub języka maszynowego.

Listing 13.1. Proste zadanie z pięcioma instrukcjami

```
Zadanie P {
    1. num = 5
    2. num++
    3. num = num – 2
    4. x = 10
    5. num = num + x
}
```

Jak możesz zobaczyć, w omawianym przykładzie instrukcje są ponumerowane, co oznacza, że *muszą* być wykonane w podanej kolejności, aby zadanie mogło zostać uznane za wykonane prawidłowo. Co do tego mamy pewność. Z technicznego punktu widzenia mamy do czynienia z *ograniczeniem typu „zachodzi wcześniej”* zachodzącym między dwiema kolejnymi instrukcjami. Instrukcja `num++` musi być wykonana przed instrukcją `num = num - 2` i to ograniczenie musi zostać spełnione, niezależnie od sposobu przełączania kontekstu.

Zwróć uwagę na to, że nadal istnieje niepewność związana z tym, że nie wiadomo kiedy dojdzie do przełączenia kontekstu. Trzeba pamiętać, że operacja przełączenia kontekstu może wystąpić w dowolnym momencie między instrukcjami.

Spójrz teraz na dwa możliwe sposoby wykonania omawianego zadania, w którym operacja przełączania kontekstu odbywa się inaczej (zobacz listing 13.2).

Listing 13.2. Jedno z możliwych uruchomień omawianego zadania i operacje przełączania kontekstu

```
Uruchomienie 1:
  1. num = 5
  2. num++
>>>>> Przełączenie kontekstu <<<<<
  3. num = num – 2
  4. x = 10
>>>>> Przełączenie kontekstu <<<<<
  5. num = num + x
```

Na listingu 13.3 przedstawiłem przebieg drugiego uruchomienia tego zadania i operacje przełączania kontekstu

Listing 13.3. Drugie uruchomienie naszego zadania i operacje przełączania kontekstu

```
Uruchomienie 2:
  num = 5
  >> Przełączenie kontekstu <<
  num++
  num = num – 2
  >> Przełączenie kontekstu <<
  x = 10
  >> Przełączenie kontekstu <<
  num = num + x
```

Jak możesz zobaczyć na listingu 13.2 liczba operacji przełączenia kontekstu i miejsce ich występowania może się zmieniać między poszczególnymi uruchomieniami. Jak wcześniej wspomniałem, istnieją pewne ograniczenia typu „zachodzi wcześniej”, które powinny być respektowane.

To jest powód, dla którego możemy obserwować ogólnie deterministyczne zachowanie określonego procesu. Niezależnie od tego, jak operacja przełączania kontekstu będzie przeprowadzana w poszczególnych uruchomieniach, to jednak *ogólny stan* zadania pozostaje taki sam. Przez określenie ogólny stan zadania mam na myśli zbiór zmiennych i odpowiadających im wartości po wykonaniu ostatniej instrukcji zadania. Na przykład w omawianym zadaniu zawsze będziemy mieli stan ostateczny obejmujący zmienną `num` o wartości `14` i zmienną `x` o wartości `10` — niezależnie od liczby operacji przełączania kontekstu i miejsca ich występowania.

Skoro wiemy, że ogólny stan pojedynczego zadania nie ulega zmianie podczas poszczególnych jego uruchomień, kuszące może być przyjęcie następującego założenia: ze względu na konieczność stosowania pewnej kolejności wykonywania instrukcji i istnienie ograniczeń typu „zachodzi wcześniej” współbieżność nie ma wpływu na ogólny stan zadania. Trzeba jednak zachować ostrożność podczas formułowania takiego wniosku.

Przyjmujemy założenie o istnieniu systemu współbieżnych zadań, z których wszystkie mają uprawnienia odczytu i zapisu *zasobu współdzielonego*, np. zmiennej. Jeżeli wszystkie zadania jedynie odczytują tę zmienną współdzieloną i żadne z nich nie przeprowadza operacji jej zapisu (zmiany wartości zmiennej), wówczas można stwierdzić, że niezależnie od sposobu przełączania kontekstu i niezależnie od liczby uruchomień tego zadania, zawsze otrzymamy ten sam wynik. Trzeba w tym miejscu dodać, że takie założenie jest również prawdziwe w przypadku systemu zadań współbieżnych, które w ogóle nie mają współdzielonej zmiennej.

Jeżeli jedno z zadań będzie przeprowadzać operację zapisu zmiennej współdzielonej, wówczas przełączenie kontekstu wymuszone przez jednostkę zarządcy zadań będzie miało wpływ na ogólny stan wszystkich zadań. To oznacza możliwość otrzymania różnych wyników podczas poszczególnych uruchomień! W efekcie konieczne jest wprowadzenie odpowiednich mechanizmów kontroli w celu uniknięcia wszelkich nieoczekiwanych wyników. To wynika z tego, że operacji przełączenia kontekstu nie można przewidzieć, a *stany pośrednie* zadań mogą być różne w trakcie poszczególnych uruchomień. Stan pośredni, w przeciwieństwie do ogólnego, to zbiór zmiennych i ich wartości podczas wykonywania określonej instrukcji. Każde zadanie ma tylko jeden stan ogólny, który jest określany po zakończeniu jego wykonywania. Istnieje jednak wiele stanów pośrednich odpowiadających zmiennym i ich wartościom po wykonaniu określonej instrukcji.

Podsumowując, gdy nasz system współbieżny zawiera wiele zadań wykonywanych na zasobie współdzielonym, który może być modyfikowany przez te zadania, wówczas poszczególne uruchomienia takiego systemu będą kończyły się odmiennymi wynikami. Dlatego też należy używać prawidłowych metod *synchronizacji* w celu zniwelowania efektu przełączania kontekstu i otrzymywania tych samych deterministycznych wyników podczas poszczególnych uruchomień.

W ten sposób poznałeś niektóre podstawowe koncepcje współbieżności, która jest dominującym tematem tego rozdziału. Koncepcje wyjaśnione w tym podrozdziale mają ważne znaczenie pomagające w zrozumieniu wielu innych zagadnień. Do tych koncepcji będziemy powracać wielokrotnie w kolejnych rozdziałach książki.

Czy pamiętasz, jak wcześniej wspomniałem, że współbieżność może być problematyczna, a co za tym idzie, może również komplikować pracę programisty? Być może zastanawiasz się, czy współbieżność naprawdę jest nam potrzebna. W następnym podrozdziale spróbuję udzielić odpowiedzi na to pytanie.

Kiedy należy używać współbieżności

Na podstawie przedstawionych dotychczas wyjaśnień można odnieść wrażenie, że posiadanie tylko jednego zadania jest znacznie mniej problematyczne niż posiadanie wielu zadań wykonujących to samo współbieżnie. To prawda. Jeżeli można utworzyć program działający w sposób możliwy do zaakceptowania bez konieczności wprowadzania współbieżności, wówczas gorąco zachęcam do takiego właśnie rozwiązania. Istnieją pewne ogólne wzorce, o których trzeba wiedzieć podczas używania współbieżności.

W tym podrozdziale zamierzam omówić te ogólne wzorce i wyjaśnić, w jaki sposób prowadzą one do podzielenia programu na współbieżnie wykonywane jednostki.

Niezależnie od użytego języka programowania program składa się ze zbioru poleceń, które powinny być wykonane sekwencyjnie. Innymi słowy dana instrukcja nie zostanie wykonana, dopóki nie zakończy się wykonywanie poprzedniej instrukcji Tę koncepcję określamy mianem *wykonywania sekwencyjnego*. Nie ma znaczenia czas potrzebny na wykonanie bieżącej instrukcji, następna musi poczekać na zakończenie wykonywania bieżącej. Zwykle mówimy, że bieżąca instrukcja *blokuje* następną, co czasami prowadzi do określenia bieżącej instrukcji jako *instrukcji blokującej*.

W każdym programie wszystkie instrukcje są blokujące, a każda jednostka wykonuje zadanie sekwencyjnie. Można powiedzieć, że program działa szybko, gdy poszczególne instrukcje blokują inne przez bardzo krótki czas wyrażony w milisekundach. Co się dzieje w sytuacji, gdy instrukcja blokująca zabiera zbyt dużą ilość czasu, np. 2 sekundy czyli 2000 milisekund, lub jeśli ilość potrzebnego jej czasu nie może być ustalona? Te dwa wzorce wyraźnie wskazują, że konieczne jest zastosowanie współbieżności w programie.

Można pójść krok dalej i stwierdzić, że każda instrukcja blokująca zabiera pewną ilość czasu, gdy system próbuje ją wykonać. Z perspektywy użytkownika najlepszym rozwiązaniem będzie, gdy taka instrukcja zabiera względnie niewielką ilość czasu potrzebnego do jej wykonania, a następnie kolejna instrukcja może być wykonana bardzo szybko. Jednak takie rozwiązanie nie zawsze jest możliwe.

Zdarzają się pewne sytuacje, w których nie można ustalić ilości czasu potrzebnego do wykonania instrukcji blokującej. Tak się najczęściej dzieje, gdy instrukcja blokująca oczekuje na pewne zdarzenie lub udostępnienie określonych danych.

Przejdźmy do kolejnego przykładu. Przyjmujemy założenie o istnieniu programu serwerowego obsługującego pewną liczbę programów klienckich. W programie serwera znajduje się pewna instrukcja oczekująca na nawiązanie połączenia przez program klienta. Z perspektywy programu serwera nie można określić, kiedy nowy klient nawiąże połączenie. Dlatego też następna instrukcja po stronie serwera nie może zostać wykonana, ponieważ nie wiadomo kiedy zakończy się bieżąca. Wszystko zależy całkowicie od czasu, w którym nowy klient spróbuje nawiązać połączenie.

Z prostszym przykładem mamy do czynienia podczas odczytywania ciągu tekstowego pochodzącego od użytkownika. Z perspektywy programu nie można określić, kiedy użytkownik dostarczy dane wejściowe, więc następne instrukcje nie mogą zostać wykonane. To jest *pierwszy wzorzec* prowadzący do współbieżnego systemu zadań.

Z pierwszym wzorcem dla współbieżności mamy do czynienia, gdy istnieje instrukcja, która może przez nieokreśloną i nieskończoną ilość czasu blokować wykonywanie kolejnych. Na tym etapie należy dokonać podziału na dwie oddzielne jednostki wykonywania lub na dwa oddzielne zadania. Powinieneś to zrobić, jeśli chcesz, aby następne instrukcje zostały wykonane, i nie możesz przy tym czekać na zakończenie wykonywania pierwszej. W takim scenariuszu najważniejsza jest możliwość przyjęcia założenia o niezależności późniejszych instrukcji od wyniku wykonania pierwszej.

Dzięki podziałowi pracy na dwa zadania współbieżne, gdy jedno oczekuje na zakończenie wykonywania instrukcji blokującej, drugie może kontynuować działanie i wykonywać te instrukcje, które byłyby blokowane w przypadku niezastosowania podziału.

W kolejnym przykładzie skoncentruję się na pokazaniu, jak pierwszy wzorzec może prowadzić do powstania systemu zadań współbieżnych. Do przedstawienia poszczególnych instrukcji w zadaniach wykorzystam pseudokod.

> **Uwaga!**
>
> Do zrozumienia omawianego tutaj przykładu nie jest wymagana żadna wiedza z zakresu sposobu działania sieci komputerowych.

W tym przykładzie skoncentruję się na programie serwera, który ma trzy wymienione poniżej cele:

- Obliczenie sumy dwóch liczb pobranych od klienta i zwrócenie wyniku klientowi.
- Regularne zapisywanie w pliku liczb otrzymanych od klientów niezależnie od tego, czy akurat jest obsługiwany jakikolwiek klient.
- Możliwość jednoczesnego obsłużenia wielu klientów.

Zanim przejdę do ostatecznego systemu współbieżnego, który spełni wymienione tutaj cele, najpierw przyjmiemy założenie, że w omawianym przykładzie będzie użyte tylko jedno zadanie (lub jednostka wykonywania). Następnie pokażę, że pojedyncze zadanie nie jest w stanie spełnić zdefiniowanych tutaj celów. Na listingu 13.4 przedstawiłem pseudokod dla naszego programu serwera.

Listing 13.4. Program serwera oparty na pojedynczym zadaniu

```
Kalkulator serwera {
    Zadanie T1 {
        1. N = 0
        2. Przygotowanie serwera
        3. Wykonywanie w nieskończoność {
        4.     Oczekiwanie na klienta C
        5.     N = N + 1
        6.     Odczytanie pierwszej liczby z C i umieszczenie jej w X
        7.     Odczytanie drugiej liczby z C i umieszczenie jej w Y
        8.     Z = X + Y
        9.     Zapisanie Z do C
       10.     Zamknięcie połączenia z C
       11.     Zapisanie N do pliku
           }
    }
}
```

Jak możesz zobaczyć, w naszej pojedynczej jednostce wykonywania oczekujemy, aż klient sieci nawiąże połączenie. Następnie od klienta są pobierane dwie liczby, obliczana jest ich suma, a wynik zostaje zwrócony klientowi. Na koniec połączenie z serwerem jest zamykane, a przed przejściem do stanu oczekiwania na połączenie następnego klienta w pliku zostaje zapisana liczba obsłużonych klientów.

Ten pseudokod składa się tylko z jednego zadania, T1. W jego skład wchodzi 12 wierszy i jak już wcześniej stwierdziłem, są one wykonywane sekwencyjnie, a wszystkie instrukcje są blokujące. Co tak dokładnie ten kod nam pokazuje? Przeanalizujmy go nieco dokładniej.

- Pierwsza instrukcja, `N = 0`, jest bardzo prosta i szybko zostaje zakończona.
- Oczekuje się, że druga instrukcja, `Przygotowanie serwera`, zostanie zakończona w ciągu rozsądnego czasu, więc nie blokuje wykonywania programu serwera.
- Trzecia instrukcja rozpoczyna wykonywanie pętli głównej i powinna szybko się zakończyć po przejściu do wewnątrz pętli.
- Czwarta instrukcja, `Oczekiwanie na klienta C`, jest blokująca, a czas jej zakończenia pozostaje nieznany. Dlatego też instrukcje 5., 6. i pozostałe nie zostaną wykonane. Te instrukcje muszą czekać na dołączenie nowego klienta i dopiero wtedy będą mogły być wykonane.

Jak wcześniej wspomniałem, instrukcje od 5. do 10. muszą czekać na nawiązanie połączenia przez nowego klienta. Innymi słowy wykonanie tych instrukcji jest uzależnione od wyniku działania instrukcji 4. i nie mogą być one wykonane, dopóki nowy klient nie zostanie zaakceptowany. Jednak instrukcja 11., `Zapisanie N do pliku`, musi być wykonana niezależnie od tego, czy mamy klienta. To jest podyktowane drugim celem zdefiniowanym dla omawianego przykładu. W przedstawionej konfiguracji następuje zapisanie `N` do pliku tylko wtedy, gdy nowy klient został dołączony. To się odbywa pomimo początkowego wymagania, aby zapisać `N` do pliku *niezależnie* od tego, czy mamy klienta.

Przedstawiony kod ma jeszcze jeden problem w swojej jednostce wykonywania instrukcji, ponieważ instrukcje 6. i 7. mogą potencjalnie zablokować wykonywanie kolejnych. Te instrukcje czekają na podanie dwóch liczb przez użytkownika. Skoro to zadanie będzie wykonane po stronie klienta, nie można dokładnie przewidzieć, kiedy te instrukcje się zakończą. To uniemożliwia dalsze działanie programu.

Co więcej, te instrukcje mogą potencjalnie uniemożliwić programowi akceptowanie nowych klientów. To wiąże się z tym, że instrukcja 4. nie zostanie ponownie wykonana, jeśli zakończenie instrukcji 6. i 7. będzie wymagało bardzo długiego czasu. Dlatego też program serwera nie może jednocześnie obsługiwać wielu klientów, a to również jest sprzeczne ze zdefiniowanymi celami.

Aby usunąć te problemy, to pojedyncze zadanie trzeba podzielić na trzy współbieżne, które razem będą w stanie spełnić cele stawiane przed programem.

W pseudokodzie przedstawionym na listingu 13.5 możesz zobaczyć trzy jednostki wykonywania — `T1`, `T2` i `T3` — które spełniają zdefiniowane cele na podstawie rozwiązania współbieżnego.

Listing 13.5. Program serwera używającego trzech zadań współbieżnych

```
Kalkulator serwera {
    Zmienna współdzielona: N
    Zadanie T1 {
        1. N = 0
        2. Przygotowanie serwera
```

```
            3. Utworzenie zadania T2
            4. Wykonywanie w nieskończoność {
            5.     Zapisanie N do pliku
            6.     Oczekiwanie 30 sekund
               }
        }
        Zadanie T2 {
            1. Wykonywanie w nieskończoność {
            2.     Oczekiwanie na klienta C
            3.     N = N + 1
            4.     Utworzenie zadania T3 dla C
               }
        }
        Zadanie T3 {
            1. Odczytanie pierwszej liczby z C i umieszczenie jej w X
            2. Odczytanie pierwszej liczby z C i umieszczenie jej w Y
            3. Z = X + Y
            4. Zapisanie Z do C
            5. Zamknięcie połączenia z C
        }
    }
```

Ten program rozpoczyna działanie od uruchomienia zadania `T1`. To będzie zadanie główne programu, ponieważ zostało wykonane jako pierwsze. Warto zwrócić uwagę na to, że każdy program ma przynajmniej jedno zadanie, które będzie — bezpośrednio lub pośrednio — inicjowało wszystkie pozostałe zadania.

W omawianym przykładzie mamy jeszcze dwa inne zadania zainicjowane przez zadanie główne `T1`. W programie znajduje się również zmienna współdzielona, `N`, przechowująca liczbę obsłużonych klientów i dostępna (do odczytu i zapisu) dla wszystkich zadań.

Działanie programu rozpoczyna się od pierwszej instrukcji zadania `T1`, czyli inicjalizacji zmiennej `N` z wartością `0`. Instrukcja druga przygotowuje serwer. W ramach tej instrukcji powinny być podjęte pewne kroki wstępne, aby program serwera miał możliwość akceptowania połączeń przychodzących. Zauważ, że dotychczas nie zostało uruchomione żadne współbieżnie działające zadanie.

Trzecia instrukcja zadania `T1` powoduje utworzenie nowego *egzemplarza* zadania `T2`. Utworzenie nowego zadania jest zwykle szybką operacją niewymagającą czasu. Dlatego też zadanie `T1` natychmiast po utworzeniu zadania `T2` przechodzi do pętli działającej w nieskończoność, w której co 30 sekund będzie zapisywać w pliku wartość zmiennej współdzielonej `N`. To jest nasz pierwszy cel, który został spełniony dla zdefiniowanego programu serwera. Zadanie `T1` bez żadnych przerw lub blokowania ze strony innych instrukcji powoduje regularne zapisywanie do pliku wartości zmiennej `N`, co będzie się odbywało aż do zakończenia działania programu.

Chciałbym poświęcić chwilę na zagadnienie tworzenia nowego zadania. Jedynym celem zadania `T2` jest akceptowanie nowych klientów tuż po wysłaniu przez nich żądania nawiązania połączenia. Warto pamiętać, że wszystkie instrukcje wykonywane przez zadanie `T2` są uruchamiane w pętli działającej w nieskończoność. Drugie polecenie zadania `T2` czeka na nowego klienta,

więc powoduje zablokowanie pozostałych instrukcji. Jednak ta blokada dotyczy jedynie instrukcji zdefiniowanych w zadaniu T2. Zwróć uwagę na to, że jeśli zostaną utworzone dwa egzemplarze zadania T2 zamiast tylko jednego, wówczas zablokowanie instrukcji w jednym z nich nie oznacza zablokowania instrukcji w drugim egzemplarzu zadania.

Pozostałe zadania współbieżne (w omawianym przykładzie tylko T1) bez żadnych problemów kontynuują wykonywanie swoich instrukcji. Jest to możliwość oferowana przez współbieżność. Gdy pewne zadania są blokowane dla danego zdarzenia, pozostałe działają bez żadnych przerw. Jak już wcześniej wspomniałem, jest to bardzo ważna zasada projektowa: *gdy masz operację blokującą, której czas zakończenia jest nieznany lub bardzo długi, wówczas zadanie powinieneś podzielić na dwa współbieżne*.

Teraz przyjmujemy założenie o dołączeniu nowego klienta. Miałeś już okazję zobaczyć to w kodzie na listingu 13.4 w niezapewniającej obsługi współbieżności wersji programu serwerowego, w której operacja odczytu mogła uniemożliwić akceptowanie nowych klientów. Skoro operacje odczytu są blokujące, to na podstawie wymienionej wcześniej reguły projektowej konieczny jest podział logiki na dwa zadania współbieżne, stąd wprowadzenie zadania T3.

Gdy dołącza nowy klient, zadanie T2 powoduje utworzenie egzemplarza zadania T3, aby zapewnić możliwość komunikacji z klientem, który właśnie dołączył. To jest możliwe dzięki czwartej instrukcji w zadaniu T2, którą przypomniałem na listingu 13.6.

Listing 13.6. Czwarta instrukcja zadania T2

```
4.    Utworzenie zadania T3 dla C
```

Trzeba koniecznie zwrócić uwagę na to, że przed utworzeniem nowego zadania zadanie T2 inkrementuje wartość zmiennej współdzielonej N, co ma wskazywać na obsłużenie nowego klienta. Wykonanie tej instrukcji odbywa się dość szybko i nie powoduje blokowania operacji zaakceptowania nowego klienta.

Gdy kończy się wykonywanie czwartej instrukcji w zadaniu T2, działanie pętli jest kontynuowane i ponownie zostaje wykonana druga instrukcja, która czeka na dołączenie kolejnego klienta. W omawianym tutaj pseudokodzie mamy tylko po jednym egzemplarzu zadań T1 i T2, ale możemy mieć wiele egzemplarzy zadania T3 — po jednym dla każdego klienta.

Jedynym przeznaczeniem zadania T3 jest komunikacja z klientem i odczytywanie podawanych przez niego liczb. Następnie egzemplarz T3 oblicza sumę otrzymanych liczb i zwraca ją klientowi. Jak już wcześniej wspomniałem, blokujące instrukcje w zadaniu T3 nie mogą blokować wykonywania innych zadań, a to blokujące zachowanie jest ograniczone do tego samego egzemplarza T3. Nawet instrukcje blokujące w konkretnym egzemplarzu T3 nie mogą prowadzić do blokowania instrukcji w innym egzemplarzu T3. W ten sposób program serwera może w sposób współbieżny spełnić wszystkie wyznaczone mu cele.

Następne pytanie może dotyczyć tego, kiedy zakończy się wykonywanie zadań. Ogólnie rzecz biorąc, wiadomo, że wykonanie wszystkich instrukcji zadania oznacza zakończenie danego zadania. Dlatego też gdy mamy pętlę działającą w nieskończoność opakowującą wszystkie

instrukcje w zadaniu, takie zadanie nie zostanie zakończone, a jego cykl życiowy zależy od *zadania nadrzędnego*, które je utworzyło. Procesy i wątki zostaną dokładniej omówione w kolejnych rozdziałach. Na potrzeby omawianego przykładu zadaniem nadrzędnym wszystkich zadań `T3` jest jedyny istniejący egzemplarz `T2`. Jak możesz zobaczyć, określony egzemplarz zadania `T3` kończy zadanie po zamknięciu połączenia z klientem, czyli po wykonaniu dwóch blokujących instrukcji pobrania danych wejściowych lub też po zakończeniu działania przez jedyny istniejący egzemplarz `T2`.

W rzadkiej (choć możliwej) sytuacji jeżeli wykonanie wszystkich operacji odczytu zabiera zbyt dużo czasu (to może być przypadkowe lub celowe), a liczba nowych klientów gwałtownie rośnie, wówczas może się zdarzyć, że będzie zbyt wiele uruchomionych egzemplarzy zadania `T3` i wszystkie będą oczekiwały na dostarczenie danych wejściowych przez klientów. W takiej sytuacji będziemy mieli do czynienia z użyciem znacznej ilości zasobów. Dlatego też po pojawieniu się kolejnych połączeń program serwera zostanie zakończony przez system operacyjny lub też serwer nie będzie w stanie obsłużyć nowych klientów.

Jeżeli dojdzie do sytuacji wymienionej w poprzednim akapicie, program serwera będzie musiał wstrzymać obsługę klientów. W takim przypadku mówimy o tzw. **odmowie usług** (ang. *denial of service*, DoS). Systemy z zadaniami działającymi współbieżnie powinny być projektowane w taki sposób, aby przezwyciężać takie sytuacje ekstremalne, które uniemożliwiają obsługę klientów w rozsądny sposób.

> **Uwaga!**
>
> W przypadku ataku typu DoS wyczerpanie dostępnych zasobów w komputerze serwera powoduje wstrzymanie jego działania, a system przestaje reagować na działania użytkownika. Ataki typu DoS zaliczają się do grupy ataków sieciowych, których celem jest zablokowanie danej usługi w taki sposób, aby stała się niedostępna dla jej klientów. W tej grupie mamy wiele różnych ataków, m.in. określanych jako *exploits* i mających na celu zatrzymanie usługi. Inny atak to tzw. *zalewanie sieci* (ang. *flooding*), a jego celem jest przeciążenie całej infrastruktury sieciowej.

W poprzednim przykładzie programu serwera przedstawiłem sytuację, w której mamy instrukcje blokujące o niemożliwym do ustalenia czasie ich wykonania. To jest pierwszy wzorzec stanowiący przesłankę do użycia współbieżności. Jest jeszcze inny wzorzec podobny do tego, choć nieco inny.

Jeżeli wykonanie instrukcji lub grupy instrukcji wymaga zbyt dużej ilości czasu, wówczas można je umieścić w oddzielnym zadaniu, a następnie uruchomić to nowe zadanie współbieżnie do zadania głównego. Ten wzorzec jest nieco inny niż pierwszy — wprawdzie dysponujemy szacunkowym (choć niezbyt dokładnym) czasem ukończenia, ale wiemy, że nie nastąpi to szybko.

W przedstawionym przykładzie jest jeszcze jedna kwestia, na którą należy zwrócić uwagę. Dotyczy ona zmiennej współdzielonej `N` i wiąże się z tym, że jedno z zadań, w szczególności egzemplarz `T2`, może zmienić jej wartość. Opierając się na informacjach zamieszczonych w rozdziale, można stwierdzić, że taki system zadań współbieżnych jest podatny na problemy związane z obsługą współbieżności ze względu na istnienie zmiennej współdzielonej, która może być zmodyfikowana przez jedno z zadań.

Trzeba koniecznie zwrócić uwagę na to, że rozwiązanie zaproponowane w programie serwera jest dalekie od idealnego. W następnym rozdziale omówię problemy powodowane przez współbieżność. Dzięki tym wyjaśnieniom będziesz mógł dostrzec, że w omówionym tutaj przykładzie pojawił się poważny problem, jakim jest *stan wyścigu* dotyczącego zmiennej współdzielonej `N`. Dlatego też konieczne jest zastosowanie odpowiednich mechanizmów kontroli pozwalających na rozwiązywanie problemów powodowanych przez współbieżność.

W ostatnim podrozdziale zamierzam skoncentrować się na *stanie* współdzielonym między zadaniami współbieżnymi. Poznasz również koncepcję *przeplotu* i jej ważne konsekwencje w systemie współbieżnym, który zawiera modyfikowalny stan współdzielony.

Stan współdzielony

W poprzednim podrozdziale omawiałem wzorce sugerujące potrzebę implementacji współbieżnego systemu zadań. Jednak wcześniej pokrótce wyjaśniłem, jak niepewność we wzorcu przełączania kontekstu podczas wykonywania wielu zadań współbieżnych w połączeniu z istnieniem modyfikowalnego stanu współdzielonego może prowadzić do niedeterministyczności w ogólnym stanie wszystkich zadań. W tym podrozdziale chciałbym przedstawić przykład potwierdzający, że brak deterministyczności może być problemem.

W podrozdziale będę kontynuował wcześniej rozpoczęty wątek i dołączę temat *stanu współdzielonego*, aby pokazać jego wpływ na brak deterministyczności. Jako programista powinieneś pojęcie *stanu* rozumieć jako zbiór zmiennych i odpowiadających im wartości w określonym czasie. Dlatego też mówiąc o *ogólnym stanie* zadania, jak to zdefiniowałem nieco wcześniej, mam na myśli zbiór istniejących zmiennych niewspółdzielonych razem z odpowiadającymi im wartościami w konkretnym momencie czasu wykonania ostatniej instrukcji zadania.

Podobnie *stan przejściowy* zadania to zbiór wszystkich istniejących zmiennych niewspółdzielonych z ich wartościami w chwili wykonywania określonej instrukcji. Dlatego też zadanie ma wiele różnych stanów przejściowych dla poszczególnych instrukcji, a liczba tych stanów odpowiada liczbie instrukcji. Zgodnie z naszymi definicjami ostatni stan przejściowy to również ogólny stan zadania.

Stan współdzielony to również zbiór zmiennych z odpowiadającymi im wartościami w określonym momencie czasu. Te zmienne mogą być odczytywane lub modyfikowane przez system zadań współbieżnych. Właścicielem stanu współdzielonego nie jest zadanie (ten stan nie jest lokalny dla zadania). Może być on odczytywany lub modyfikowany przez każde z zadań uruchomionych w systemie, i to w dowolnej chwili.

Ogólnie rzecz biorąc, nie interesują nas stany współdzielone określone jako tylko do odczytu. Z reguły pozwalają one na bezpieczny odczyt danych przez wiele zadań współbieżnych i nie prowadzą do żadnych problemów. Jednak modyfikowalny stan współdzielony zwykle oznacza pojawienie się poważnych problemów, o ile nie zostaną zachowane odpowiednie środki ostrożności. Dlatego też wszystkie stany współdzielone omawiane w podrozdziale są uznawane za modyfikowalne przez przynajmniej jedno z zadań.

Zadaj sobie następujące pytanie: co może pójść źle, jeśli stan współdzielony zostanie zmodyfikowany przez jedno z zadań współbieżnych w systemie? Aby udzielić odpowiedzi na to pytanie, trzeba przeanalizować przykład systemu dwóch zadań współbieżnych, które próbują uzyskać dostęp do pojedynczej zmiennej współdzielonej. W omawianym przykładzie jest to zmienna w postaci liczby całkowitej.

Przyjmujemy założenie o istnieniu systemu przedstawionego na listingu 13.7.

Listing 13.7. System składający się z dwóch zadań współbieżnych i modyfikowalnego stanu współdzielonego

```
System współbieżny {

    Stan współdzielony {
        X : Liczba całkowita = 0
    }

    Zadanie P {
        A : Liczba całkowita
            1. A = X
            2. A = A + 1
            3. X = A
            4. wyświetl X
    }

    Zadanie Q {
        B : Liczba całkowita
            1. B = X
            2. B = B + 2
            3. X = B
            4. wyświetl X
    }
}
```

Przyjmujemy założenie, że w przedstawionym systemie zadania `P` i `Q` nie są uruchomione współbieżnie. To oznacza ich sekwencyjne wykonywanie. Przyjmujemy założenie, że najpierw są wykonywane polecenia w `P`, a dopiero później w zadaniu `Q`. Jeżeli tak faktycznie będzie, wówczas ogólny stan całego systemu, niezależnie od ogólnego stanu poszczególnych zadań, będzie miał postać współdzielonej zmiennej `X` o wartości `3`.

Jeżeli ten system zostanie uruchomiony w odwrotnej kolejności, czyli najpierw instrukcje w `Q`, a dopiero później instrukcje w zadaniu `P`, wówczas wynikiem będzie ten sam stan ogólny. Jednak zwykle tak nie jest, a wykonanie dwóch różnych zadań w kolejności odwrotnej prawdopodobnie prowadzi do innego stanu ogólnego.

Jak możesz zobaczyć, sekwencyjne wykonanie zadań zapewnia wynik deterministyczny, bez konieczności przejmowania się operacjami przełączania kontekstu.

Przyjmujemy teraz założenie, że zadania są wykonywane współbieżnie przez ten sam rdzeń procesora. Istnieje wiele możliwych scenariuszy ułożenia instrukcji zadań `P` i `Q` na podstawie różnych operacji przełączania kontekstu zachodzących w poszczególnych instrukcjach.

Jeden z możliwych scenariuszy przedstawiłem na listingu 13.8.

Listing 13.8. Możliwy przeplot instrukcji podczas współbieżnego wykonywania zadań P i Q

```
  Zadanie P    |    Zarządca zadań          |    Zadanie Q
---------------------------------------------------------------
               |    Przełączenie kontekstu  |
               |                            |  B = X
               |                            |  B = B + 2
               |    Przełączenie kontekstu  |
A = X          |                            |
               |    Przełączenie kontekstu  |
               |                            |  X = B
               |    Przełączenie kontekstu  |
A = A + 1      |                            |
X = A          |                            |
               |    Przełączenie kontekstu  |
               |                            |  wyświetl X
               |    Przełączenie kontekstu  |
wyświetl X     |                            |
               |    Przełączenie kontekstu  |
```

To jest tylko jeden z wielu możliwych scenariuszy przełączania kontekstu zachodzącego podczas wykonywania poszczególnych instrukcji. Każdy scenariusz jest określany mianem *przeplotu* (ang. *interleaving*). Dlatego też dla systemu zadań współbieżnych istnieje wiele różnych przeplotów na podstawie miejsc, w których może dojść do operacji przełączania kontekstu. W trakcie każdego z uruchomień może być użyty tylko jeden z dostępnych przeplotów. Dlatego też wynik jest nieprzewidywalny.

Jak możesz zobaczyć na podstawie przykładowego przeplotu, w pierwszej i ostatniej kolumnie została zachowana kolejność instrukcji i ograniczeń typu „zachodzi wcześniej", ale mogą pojawić się *przerwy* między wykonaniami. Te przerwy są nieprzewidywalne i, jak wynika z przykładowego przebiegu wykonania, dany przeplot może prowadzić do otrzymania zadziwiającego wyniku. Proces `P` wyświetla wartość `1`, zaś proces `Q` wyświetla wartość `2`, choć oczekiwana była ostateczna wartość wynosząca `3`.

W omawianym przykładzie zwróć uwagę na to, że ograniczenie związane z akceptacją ostatecznego wyniku zostało zdefiniowane następująco — program powinien wyświetlić dwie cyfry `3` w danych wyjściowych. To ograniczenie mogłoby mieć inną postać i niezależną widoczność danych wyjściowych w programie. Co więcej, wszelkie inne istniejące ograniczenia o znaczeniu krytycznym powinny pozostać *niezmiennikami* podczas występowania nieprzewidywalnych operacji przełączania kontekstu. To obejmuje brak wszelkich *stanów wyścigu*, brak wycieku pamięci, a nawet brak awarii. Wszystkie te ograniczenia są znacznie ważniejsze niż widoczność danych wyjściowych programu. W wielu rzeczywistych aplikacjach program nawet nie musi generować żadnych danych wyjściowych.

Na listingu 13.9 pokazałem przykład innego przeplotu prowadzący do wygenerowania odmiennego wyniku.

Listing 13.9. Inny możliwy przeplot zadań P i Q wykonywanych współbieżnie

```
  Zadanie P  |    Zarządca zadań           |    Zadanie Q
-------------------------------------------------------------
             |    Przełączenie kontekstu   |
             |                             |    B = X
             |                             |    B = B + 2
             |                             |    X = B
             |    Przełączenie kontekstu   |
A = X        |                             |
A = A + 1    |                             |
             |    Przełączenie kontekstu   |
             |                             |    wyświetl X
             |    Przełączenie kontekstu   |
X = A        |                             |
wyświetl X   |                             |
             |    Przełączenie kontekstu   |
```

W tym przeplocie zadanie `P` powoduje wyświetlenie wartości `3`, natomiast zadanie `Q` wyświetla wartość `2`. Tak się zdarzyło, ponieważ zadanie `P` nie miało na tyle szczęścia, aby zdążyć z uaktualnieniem wartości zmiennej współdzielonej `X` przed operacją przełączenia kontekstu. Dlatego też zadanie `Q` jedynie wyświetliło wartość zmiennej `X`, która w danym momencie wynosiła `2`. To nosi nazwę *stanu wyścigu* dotyczącego zmiennej `X` —więcej informacji na ten temat znajdziesz w następnym rozdziale.

W rzeczywistym programie zwykle zapisujemy `X++` lub `X = X + 1`, zamiast najpierw kopiować wartość `A` do `X`, a następnie inkrementować `A` i ostatecznie umieścić nową wartość w zmiennej `X`. Przykład takiego rozwiązania zobaczysz w rozdziale 15.

To wyraźnie pokazuje, że proste polecenie `X++` w języku C składa się z trzech mniejszych instrukcji, które nie będą wykonywane w pojedynczym przedziale czasu. Innymi słowy nie jest to *instrukcja niepodzielna*, ale składa się z trzech mniejszych instrukcji niepodzielnych. Taka instrukcja nie może być podzielona dalej na mniejsze operacje i nie będzie przerywana przez operacje przełączania kontekstu. Więcej informacji na ten temat znajdziesz w późniejszych rozdziałach dotyczących wielowątkowości.

Jest jeszcze jedna kwestia związana z omawianym przykładem. Zadania `P` i `Q` nie były jedynymi zadaniami uruchomionymi w systemie. Mieliśmy jeszcze inne współbieżnie wykonywane zadania, ale nie zostały one uwzględnione w analizie, dlatego skoncentrowałem się tylko na zadaniach `P` i `Q`. Dlaczego tak się stało?

Odpowiedź na to pytanie kryje się w fakcie, że poszczególne przeploty między tymi dwoma zadaniami i pozostałymi zadaniami w systemie nie mogły spowodować zmiany stanów przejściowych zadań `P` i `Q`. Innymi słowy pozostałe zadania nie współdzieliły stanu z `P` i `Q`, więc jak to już wyjaśniłem wcześniej, brak zasobów współdzielonych między zasobami oznacza, że przeplot nie ma znaczenia. Tak też było w omawianym przykładzie. Można było przyjąć założenie, że w naszym hipotetycznym systemie nie istniały inne zadania poza `P` i `Q`.

Jedyną przyczyną wpływów, jakie inne zadania mogły mieć na P i Q, była zbyt duża ich liczba, która mogła spowodować zmniejszenie się szybkości wykonywania zadań P i Q. Jest to po prostu efekt istnienia długich przerw między dwiema kolejnymi instrukcjami w zadaniu P lub Q. Innymi słowy rdzeń procesora jest współdzielony przez większą liczbę zadań. W takiej sytuacji zadania P i Q czekają w kolejce dłużej niż zwykle, co z kolei przekłada się na wydłużenie czasu ich wykonywania.

Opierając się na omówionym przykładzie, mogłeś zauważyć, że nawet pojedynczy stan współdzielony między zaledwie dwoma zadaniami współbieżnymi może ostatecznie prowadzić do braku deterministyczności. Nie chciałbyś chyba mieć programu, który będzie generował inny wynik po każdym uruchomieniu. Wprawdzie zadania w omówionym przykładzie były względnie proste i składały się z czterech trywialnych instrukcji, ale rzeczywiste aplikacje współbieżne spotykane w środowisku produkcyjnym są znacznie bardziej skomplikowane niż omówiony przykład.

Ponadto mamy różne rodzaje zasobów współdzielonych, które niekoniecznie znajdują się w pamięci, np. pliki lub usługi dostępne w sieci.

Liczba zadań próbujących uzyskać dostęp do zasobu współdzielonego może być wysoka, co wymusi dokładniejszą analizę problemów związanych ze współbieżnością oraz znalezienie mechanizmów pozwalających na przywrócenie deterministyczności. W następnym rozdziale będę kontynuował tę analizę, omówię problemy powodowane przez współbieżność i przedstawię ich rozwiązania.

Zanim zakończysz lekturę rozdziału, chciałbym jeszcze krótko wspomnieć o zarządcy zadań i sposobie jego działania. Jeżeli miałbyś tylko jeden rdzeń procesora, wówczas w dowolnym momencie mógłbyś mieć tylko jedno zadanie używające tego rdzenia procesora.

Doskonale wiesz, że zarządca zadań to program potrzebujący do swojego działania przedziału czasu rdzenia procesora. Na czym polega różnica w zarządzaniu przez zarządcę zadań różnymi zadaniami w celu uzyskania dostępu do rdzenia procesora, gdy jest on używany przez inne zadanie? Przyjmujemy założenie, że zarządca zadań sam używa rdzenia procesora. Przede wszystkim wybiera zadanie z kolejki i ustawia zegar dla tzw. *przerwania zegara*, a następnie opuszcza rdzeń procesora, oddając jego zasoby wskazanemu zadaniu.

Skoro przyjęliśmy założenie, że zarządca zadań będzie każdemu zadaniu przydzielał pewną ilość czasu, w pewnym momencie zadziała przerwanie zegara, a rdzeń procesora zatrzyma wykonywanie bieżącego zadania i natychmiast wczyta zarządcę zadań z powrotem do procesora. Ten zarządca przechowuje informacje o stanie poprzedniego zadania i wczytuje następne z kolejki. Wszystko to dzieje się dopóty, dopóki działa jądro systemu. W przypadku urządzenia wyposażonego w procesor wielordzeniowy może to ulec zmianie i jądro będzie wykorzystywało różne rdzenie podczas szeregowania zadań do poszczególnych rdzeni.

W tym podrozdziale pokrótce przedstawiłem koncepcję stanu współdzielonego oraz jego wpływ na działanie systemu współbieżnego. Ten temat będę kontynuował w następnym rozdziale, w którym skoncentruję się na kwestiach związanych ze współbieżnością oraz na technikach stosowanych podczas synchronizacji.

Podsumowanie

W rozdziale zaprezentowałem podstawy współbieżności oraz istotne koncepcje i terminologię. Ta wiedza jest niezbędna do zrozumienia tematów wielowątkowości i wieloprocesowości, które będą omawiane w kolejnych rozdziałach.

W rozdziale zostały poruszone wymienione poniżej zagadnienia:

- Omówiłem definicje współbieżności i równoległości — podkreśliłem fakt, że zadanie równoległe wymaga własnej jednostki procesora, podczas gdy zadania współbieżne mogą współdzielić pojedynczy procesor.
- Wyjaśniłem, że zadania współbieżne używają pojedynczej jednostki procesora, podczas gdy zarządca zadań zarządza czasem procesora i dzieli go między różne zadania. To prowadzi do wielu operacji przełączania kontekstu i różnych przeplotów dla poszczególnych zadań.
- Zaprezentowałem wprowadzenie do instrukcji blokujących. Omówiłem wzorce sugerujące konieczność zastosowania współbieżności i pokazałem, w jaki sposób zadanie można podzielić na dwa lub trzy zadania współbieżne.
- Wyjaśniłem, czym jest stan współdzielony. Pokazałem, jak stan współdzielony może prowadzić do powstania poważnych problemów związanych ze współbieżnością, takich jak stan wyścigu, gdy wiele zadań próbuje odczytywać i zapisywać ten sam stan współdzielony.

W następnym rozdziale będę kontynuował omawianie tematu współbieżności oraz zaprezentuję różne rodzaje problemów, które można napotkać w środowisku współbieżnym. Pokażę również rozwiązania problemów związanych ze współbieżnością.

Programowanie w C: przejdź na najwyższy poziom!

Jeśli uważasz, że język C dawno odszedł do lamusa, jesteś w błędzie. Wielu inżynierów oprogramowania o nim zapomniało, jednak C wciąż cieszy się popularnością. Jest przy tym uważany za dość trudny język programowania, gdyż samo opanowanie jego składni to za mało, aby efektywnie go wykorzystywać. Właśnie dlatego ceni się programistów z wnikliwym i naukowym podejściem do jego reguł i praktyk. Tylko wtedy można wykorzystać możliwości języka C do tworzenia efektywnych systemów.

To książka przeznaczona dla programistów, którzy chcą stać się ekspertami języka C. Przedstawia zasady pracy z dyrektywami preprocesora, makrami, kompilacją warunkową i ze wskaźnikami. Omawia ważne aspekty projektowania algorytmów, funkcji i struktur. Sporo miejsca poświęcono tu kwestii uzyskiwania maksimum wydajności z aplikacji działających w środowisku o ograniczonych zasobach. Starannie opisano, jak C współpracuje z systemem Unix, w jaki sposób zaimplementowano reguły zorientowane obiektowo w języku C, a także jak wykorzystać wieloprocesowość. To świetny materiał bazowy do samodzielnego badania, zadawania pytań i eksperymentowania z kodem.

W książce między innymi:

- zaawansowane elementy języka C
- struktury pamięci i proces kompilacji
- programowanie zorientowane obiektowo w proceduralnym kodzie C
- tworzenie kodu na niskim poziomie
- współbieżność, wielowątkowość i integracja z innymi językami programowania
- testy jednostkowe i debugowanie oraz komunikacja międzyprocesowa

Kamran Amini

specjalizuje się w programowaniu jądra systemu operacyjnego i tworzeniu rozwiązań osadzonych. Pracował dla wielu doskonale znanych firm irańskich. Pasjonuje się teorią obliczeń, systemami rozproszonymi, uczeniem maszynowym i informatyką kwantową. Interesuje się również powstaniem wszechświata, geometrią czarnych dziur, kwantową teorią pola i teorią strun.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
INFORMATYKA W NAJLEPSZYM WYDANIU

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-7459-1
9 788328 374591
Cena: 129,00 zł
Packt