ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 2017
Seria: ORGANIZACJA I ZARZĄDZANIE z. 113 Nr kol. 1992

Paweł WICHARY, Ireneusz J. JÓŹWIAK, Michał SZCZEPANIK
Politechnika Wrocławska
Wydział Informatyki i Zarządzania
wicharypawel@gmail.com, ireneusz.jozwiak@pwr.edu.pl, michal.szczepanik@pwr.edu.pl

STRATEGIE NISKOPOZIOMOWEGO PROGRAMOWANIA WSPÓŁBIEŻNEGO PLATFORMY .NET

Streszczenie. W artykule przedstawiono mechanizmy związane z programowaniem współbieżnym na niskim poziomie abstrakcji. Na podstawie literatury światowej wyróżniono zbiór pojęć podstawowych dotyczących współbieżności. Wskazano mechanizmy pomagające w rozwiązaniu problemu wzajemnego wykluczania przy użyciu konstrukcji językowych platformy .NET. Opisano sposoby uzyskania synchronizacji wątków wraz z opisem ich wad i zalet.

Słowa kluczowe: programowanie współbieżne, współbieżność, problem wzajemnego wykluczania, współzawodnictwo, semafor, monitor, operacje atomowe, zagłodzenie, zakleszczenie, wątek

STRATEGIES FOR LOW-LEVEL CONCURRENT PROGRAMMING OF THE .NET FRAMEWORK

Abstract. The article presents issues related to low level concurrent programming. Based on world literature, a collection of basic notions of concurrency has been distinguished. Indicators that help to solve the mutual exclusion problem using the .NET language constructs. Describes ways to get thread synchronization once with a description of their drawbacks and advantages.

Keywords: concurrent programming, concurrency, mutual exclusion problem, semaphore, monitor, atomic operations, starvation, deadlock, thread

1. Wprowadzenie

W artykule przedstawiono wiedzę uzyskaną na podstawie pracy komercyjnej oraz badań własnych w dziedzinie programowania współbieżnego. Zagadnienia poruszane w kolejnych rozdziałach stanowią podstawę współbieżności w ogólności, a w szczególności pozwalają na

implementację wydajnych mechanizmów synchronizacji niskiego poziomu. Celem rozdziału drugiego jest omówienie podstawowych pojęć, niezależnych od języka, platformy oraz paradygmatu programowania, które są powszechnie stosowane podczas pracy z współbieżnością w różnych formach. Współbieżność dzieli się na podzagadnienia i techniki, takie jak asynchroniczność, równoległość, reaktywność itp., nie będą one opisywane w niniejszym artykule, ponieważ są rzadko stosowane w bezpośrednich implementacjach niskopoziomowych. W rozdziale trzecim przedstawiono klasyczne mechanizmy synchronizacji wątków, próbujących jednocześnie uzyskać dostęp do zasobów współdzielonych, które zostały zaimplementowane przy pomocy platformy .NET. Przedstawione rozwiązania znajdują bezpośrednie zastosowanie w niskopoziomowych implementacjach oraz wewnątrz mechanizmów wysokiego poziomu. Podsumowując, autorzy skupią się na opisaniu synchronizacji blokującej i nieblokującej oraz zaproponują dalszą lekturę rozszerzającą tematykę.

2. Pojęcia podstawowe w kontekście współbieżności

Programowanie sekwencyjne polega na definiowaniu pracy programu, w którym kolejne instrukcje oprogramowania wykonywane są kolejno jedna po drugiej. Rozumienie pracy procesora w tym sensie stanowi naturalne dla człowieka podejście, gdzie sekwencyjność jest przeciwieństwem zrównoleglenia.

Istotne jest określenie relacji pomiędzy paradygmatami programowania a sposobami wykonania programu na procesorze lub procesorach. W mowie potocznej często stosowaną praktyką jest pomijanie słowa paradygmat, co może prowadzić do wrażenia, jakoby programowanie obiektowe oraz programowanie współbieżne stanowiły pojęcia o podobnej tematyce. Paradygmaty takie jak:

- paradygmat programowania obiektowego,
- paradygmat programowania imperatywnego,
- paradygmat programowania funkcyjnego,
- paradygmat programowania proceduralnego,
- paradygmat programowania aspektowego,

mówią o formie, w jakiej programista patrzy na przepływ sterowania i implementację programu komputerowego. Jednak kolejne paradygmaty programowania mogą pracować sekwencyjnie, np. „program sekwencyjny napisany przy użyciu paradygmatu obiektowego”, lub wykorzystywać techniki pozwalające na zrównoleglenie pracy. Do technik pozwalających na współbieżność pracy zaliczamy:

- programowanie wielowątkowe,
- programowanie równoległe,
- programowanie asynchroniczne,
- programowanie reaktywne,
- programowanie z użyciem aktorów.

Wymienione paradygmaty oraz techniki zrównoleglenia można ze sobą łączyć, co stanowi powszechną praktykę. W celu maksymalizacji wykorzystania zalet w ramach jednego programu można połączyć np. programowanie równoległe i asynchroniczne przy użyciu paradygmatów: obiektowego i funkcyjnego.

2.1. Wątki i procesy systemu operacyjnego

Wątek to ścieżka wykonania, która może być realizowana niezależnie od innych ścieżek. Proces systemu operacyjnego dostarcza odizolowane środowisko, w którym działa program. Jeżeli program jest jednowątkowy, to w odizolowaniu działa tylko jeden wątek, posiadający własne środowisko wykonania i pamięć.

Programy wielowątkowe uruchamiają wiele wątków w ramach jednego procesu, które współdzielą to samo środowisko wykonania wraz z pamięcią.

2.2. Przeplot, przerwanie i sekcja krytyczna

Przeplotem instrukcji nazywamy kolejne wywołania instrukcji w określonej kolejności na procesorze, gdzie instrukcje mogą pochodzić z różnych wątków. Ważne jest zaznaczenie, iż kolejność wykonywania instrukcji dla identycznego programu niesekwencyjnego dla kolejnych uruchomień rzadko kiedy będzie powtarzalna między kolejnymi wywołaniami. Brak determinizmu wynika z wewnętrznej implementacji zarządcy zadań procesora (implementowany w ramach systemu operacyjnego), który decyduje o zadaniach i kolejności ich wykonania.

Przerwaniem nazywamy moment, w którym praca określonego wątku na procesorze jest zatrzymana i zasoby przekazywane są dla innego wątku. Przerwania obsługiwane są przez implementację zarządcy.

Sekcja krytyczna jest obszarem kodu, w ramach którego różne wątki w tym samym czasie oczekują dostępu do zasobów współdzielonych, którymi może być urządzenie fizyczne, obszar pamięci, zasoby obliczeniowe.

2.3. Współzawodnictwo i synchronizacja

Współzawodnictwem nazywamy sytuację, gdy wątki ubiegają się o ten sam zasób. Poniżej w celu przedstawienia sytuacji patologicznej mogącej mieć miejsce przy jednoczesnej pracy wątków nad zasobem współdzielonym zostały przedstawione wywołania kolejnych instrukcji dwóch wątków. Zadaniem wątków będzie zwiększenie wartości zmiennej o jeden, gdzie początkowa wartość licznika wynosi zero. Poniżej zamieszczony został przykładowy możliwy przebieg programu:

- Wątek 1 odczytuje wartość zero zmiennej współdzielonej i zapisuje do pamięci lokalnej wątku.
- Wątek 2 odczytuje wartość zero zmiennej współdzielonej i zapisuje do pamięci lokalnej wątku.
- Wątek 1 dokonuje zwiększenia lokalnej wartości zmiennej o jeden (praca na pamięci lokalnej wątku nie ma wpływu na pamięć współdzieloną procesu).
- Wątek 2 dokonuje zwiększenia lokalnej wartości zmiennej o jeden.
- Wątek 1 zapisuje wynik zmiennej lokalnej do zmiennej współdzielonej, nadpisując wartość zero wartością jeden.
- Wątek 2 zapisuje wynik zmiennej lokalnej do zmiennej współdzielonej, nadpisując wartość jeden wartością jeden.

Wbrew oczekiwaniom wartość licznika po dwukrotnej inkrementacji nie wynosi dwa, lecz jeden, gdyż oba wątki wykonały pracę równocześnie, nadpisując swoje rozwiązania. W celu zagwarantowania bezpiecznego dostępu stosuje się różne formy synchronizacji. Prostym sposobem zagwarantowania bezpieczeństwa zasobów współdzielonych w ramach sekcji krytycznej jest nałożenie blokady, mechanizmu, który zagwarantuje, iż tylko jeden wątek w danym czasie będzie posiadał prawa dostępu.

Zadania poszczególnych wątków są następujące:

- Wątek 1 oczekuje na prawo dostępu do zmiennej współdzielonej.
- Wątek 2 oczekuje na prawo dostępu do zmiennej współdzielonej.
- Zarządca nadaje prawo dostępu wątkowi pierwszemu.
- Wątek 1 odczytuje wartość zero zmiennej współdzielonej i zapisuje do pamięci lokalnej wątku.
- Wątek 1 dokonuje zwiększenia lokalnej wartości zmiennej o jeden (praca na pamięci lokalnej wątku nie ma wpływu na pamięć współdzieloną procesu).
- Wątek 1 zapisuje wynik zmiennej lokalnej do zmiennej współdzielonej, nadpisując wartość zero wartością jeden.
- Wątek 1 zwalnia blokadę do zmiennej współdzielonej.
- Zarządca nadaje prawo dostępu wątkowi drugiemu.

- Wątek 2 odczytuje wartość jeden zmiennej współdzielonej i zapisuje do pamięci lokalnej wątku.
- Wątek 2 dokonuje zwiększenia lokalnej wartości zmiennej o jeden.
- Wątek 2 zapisuje wynik zmiennej lokalnej do zmiennej współdzielonej, nadpisując wartość jeden wartością dwa oraz zwalnia blokadę do zmiennej współdzielonej.

Kluczowe jest, aby wątek zwolnił zasób w momencie, w którym przestaje z niego korzystać.

2.4. Zagłodzenie i zakleszczenie

Zagłodzeniem nazywamy sytuację, w której wątek nie może dokończyć pracy, ponieważ nie ma dostępu do współdzielonego zasobu. Sytuacja może mieć miejsce w systemach, w których występuje przydzielanie zasobów według najwyższego priorytetu. Dla przykładu, jeżeli istnieje wątek z zadaniem o niskim priorytecie, początkowo jest on skierowany, aby oczekiwał na prawo dostępu do zasobu, prawo to nigdy nie zostanie mu przydzielone, jeżeli w czasie oczekiwania napływać będą nowe zadania o wyższych priorytetach.

Zakleszczenie (ang. *deadlock*) jest to stan, w którym dwa wątki lub więcej oczekuje nieskończenie długo na zasoby. Jako przykład posłuży system z dwoma zasobami współdzielonymi s1 oraz s2, który posiada dwa wątki w1 oraz w2. Poniższy ciąg instrukcji prowadzi do powstania zakleszczenia:

- Wątek 1 oczekuje na prawo do zasobu s1.
- Wątek 2 oczekuje na prawo do zasobu s2.
- Wątek 1 otrzymuje prawo do zasobu s1.
- Wątek 2 otrzymuje prawo do zasobu s2.
- Wątek 1 oczekuje na prawo do zasobu s2.
- Wątek 2 oczekuje na prawo do zasobu s1.

W powyższej sytuacji po ostatniej instrukcji występuje zakleszczenie, gdyż pierwszy wątek nie zwolnił jeszcze blokady, na którą oczekuje drugi wątek i odwrotnie. Posiadanie przez wątek kilku blokad nie jest sytuacją zabronioną. Blokady stanowią najprostszy sposób na synchronizację wątków, jednak ich nakładanie zwiększa prawdopodobieństwo powstania zakleszczenia.

Efektywna praca z blokadami polega na minimalizacji obszaru kodu, który jest nimi objęty oraz unikaniu tworzenia cykli pomiędzy wątkami oczekującymi.

3. Techniki współbieżności niskiego poziomu

Przeplot wykonania wątków w kontekście sekcji krytycznych prowadzi do powstania problemu wzajemnego wykluczania. Zagadnienie to opiera się na zagwarantowaniu dostępu tylko dla jednego z wątków do sekcji, przy zachowaniu wydajności przetwarzania. Rozwiązanie tego problemu polega na wprowadzeniu do programu zestawu dodatkowych instrukcji wykonywanych przez wątek, który chce wejść lub opuścić sekcję krytyczną.

Problem wzajemnego wykluczania można uznać za rozwiązany, jeżeli:

- instrukcje z sekcji krytycznych dwóch lub więcej procesów nie są przeplatane;
- zatrzymanie (przerwanie) wykonywania pracy wątku niebędącego w sekcji krytycznej nie ma negatywnego wpływu na wykonywanie pozostałych wątków;
- nie może występować blokada w dostępie do sekcji krytycznych;
- nie może występować zagłodzenie lub efekt żywej blokady.

Możliwe jest wprowadzenie własnej implementacji problemu wzajemnego wykluczania, jednak nie stanowi to powszechnie zalecanej praktyki. Tworzenie własnych mechanizmów synchronizujących w ramach zespołu programistów może doprowadzić do powstania błędów trudnych do wykrycia. Zalecaną praktyką podczas pracy z platformą .NET jest używanie dostępnych mechanizmów, które mają na celu uporządkowanie zagadnienia oraz podniesienie niezawodności.

3.1. Semafory

Implementacje algorytmów, wspomniane podczas omawiania problemu wzajemnego wykluczania, pozwalają na zagwarantowanie synchronizacji pracy wątków na maszynie bez wsparcia ze strony mechanizmów systemu operacyjnego oraz wsparcia sprzętowego.

Semafory stanowią mechanizm rozwiązujący to samo zagadnienie na wyższym poziomie abstrakcji. Programista pracujący z semaforem może go używać przy pomocy dwóch zdefiniowanych metod Czekaj (ang. Wait) oraz Sygnalizuj (ang. Signal). Wywołanie powoduje zmianę wewnętrznego licznika Semafora, symbolizującego ilość dostępnych zasobów. Licznik przyjmować może tylko wartości całkowite nieujemne. Wywołanie przez wątek metody Czekaj służy do odnotowania chęci pozyskania zasobu, którym może być dostęp do sekcji krytycznej. Jeżeli licznik posiada wartość dodatnią, zostanie zmniejszony o jeden, następnie wątek wywołujący metodę otrzymuje pozwolenie na wejście. Przypadek, w którym wartość licznika wynosi zero, wymusza na wątku oczekiwanie. Sygnalizacja oznacza zwolnienie zasobu i zwiększenie wartości licznika semafora o jeden.

Semafor przyjmujący dla licznika dowolne nieujemne wartości nazywamy semaforem ogólnym. Semafor przyjmujący jedynie wartości 0 i 1 nazywamy semaforem binarnym.

Semafory są podstawowym mechanizmem synchronizującym, które nie wymaga aktywnego czekania na zasoby. Aktywnym czekaniem nazywamy pracę wątku w pętli, w której nie wykonuje operacji, gdzie warunkiem wyjścia z pętli jest pozwolenie na uzyskanie zasobów. Należy mieć świadomość zużywanych zasobów sprzętowych, takich jak czas pracy procesora, gdy mówimy o oczekiwaniu tego typu. Semafory rozwiązują ten problem, wstrzymując pracę wątku, która jest osiągnięta poprzez wewnętrzną implementację semafora.

W implementacji platformy .NET występują dwie klasy odpowiedzialne za implementację logiki semaforów. Pierwszą z nich jest klasa Semaphore, która do pracy używa metod WaitOne, odpowiednik Czekaj oraz Release będącej odpowiednikiem metody Sygnalizuj. Zasada działania klasy jest zgodna z tą opisaną w literaturze. Autor poprzez pozyskiwanie zasobów z semafora będzie rozumiał wywołanie metody Czekaj, natomiast zwolnieniem zasobów będzie określane wywołanie metody Release.

Pracując z Semaforem, można kilkukrotnie wywołać metodę Czekaj, co pobierze z puli dostępnych zasobów (reprezentowanych przez licznik semafora) tyle zasobów, o ile wątek poprosi. Sytuacja tego typu wystąpić może, przykładowo, jeżeli wątek wymaga do pracy trzech zasobów z puli. Elementem koniecznym po wykonaniu pracy jest sygnalizacja semafora odpowiedzialnego za dostęp o ilości zwalnianych zasobów.

Problemy mogące wystąpić przy nadmiarowym pozyskiwaniu zasobów to spadki wydajności oraz blokady. Poprzez nadmiarowość autor rozumie sytuację, w której wątek rezerwuje zasoby i nie prowadzi do ich zwolnienia (wskutek przepływu programu lub błędu programisty), co skutkuje trwałym zablokowaniem zasobów. Sytuacją odwrotną jest niedomiar w rezerwacji zasobów, który może prowadzić do jednoczesnego dostępu do zasobu przez dwa wątki.

Sygnalizowanie zwalnianych zasobów należy przeprowadzać równie uważnie jak ich rezerwację, ponieważ zdarzyć się mogą tutaj dwie sytuacje niepożądane. Pierwszą sytuacją jest poinformowanie semafora o zwolnieniu zasobów, gdy ten jest już pełny, tzn. licznik semafora jest ustawiony na wartość maksymalną. Implementacja semafora platformy .NET poinformuje o tym fakcie wyjątkiem w czasie wykonania, jednak świadczyć to może o błędnej implementacji wątków, która zwraca zasoby częściej niż powinna. Druga sytuacja wynika z mechaniki semafora, który nie sprawdza tożsamości wątku wywołującego metodę sygnalizującą. Przykładowo pierwszy wątek może wywołać metodę sygnalizującą zwracającą cztery zasoby, choć faktycznie rezerwował mniejszą ich liczbę. Zwrócenie nieposiadanej wartości zasobów spowoduje niezgodne z logiką systemu wpuszczenie wątków oczekujących.

Omawiając kwestie implementacyjne, istotny jest brak gwarancji na kolejność wpuszczania wątków do semafora. Nie występuje relacja pierwszeństwa wątku, który wcześniej zgłosił chęć uzyskania dostępu, brak implementacji w tym zakresie nie stanowi o wadzie klasy Semaphore.

Pracując z platformą .NET, wyróżniamy semafory lokalne oraz nazwane semafory globalne. Semaforem lokalnym nazywamy mechanizm pracujący w obrębie jednego procesu.

Globalny semafor nazwany jest widoczny między procesami i pozwala na synchronizację między nimi.

Drugą klasą realizującą logikę semaforów jest SemaphoreSlim. Różnica między wymienionymi klasami polega na zakresie ich działania, pierwsza pozwala na nazwane semafory globalne, druga nie. Zaletą wynikającą z korzystania z SemaphoreSlim jest prędkość działania, która w testach wydajnościowych jest lepsza o rząd wielkości.

Podsumowując, klasy SemaphoreSlim używamy jako preferowanej w celu uzyskania logiki semafora w ramach jednej aplikacji, używanie klasy Semaphore wskazane jest, gdy wymagana jest synchronizacja między procesami. Odpowiedzialnością programisty jest odpowiednie oczekiwanie oraz zwalnianie zasobów.

3.2. Monitory

Wadą semaforów w kontekście budowania złożonego systemu jest rozproszenie logiki odpowiedzialnej za synchronizację po różnych miejscach w kodzie systemu. Przyjmując hipotetyczną sytuację, w której programista zapomniał umieścić wywołanie metody sygnalizującej, może to doprowadzić do braku dostępu do zasobów dla kolejnych wątków, prowadząc do blokady. Monitory stanowią rozwiniętą koncepcję semaforów, dostarczając mechanizmu centralizującego logikę odpowiedzialną za uprawnienia.

Zaletą monitorów jest enkapsulacja danych oraz logiki odpowiedzialnej za synchronizację pracy wątków. Semafory gwarantują synchronizację pracy wywoływanych metod, zapewniając wzajemne wykluczanie ich wykonania. Inżynier oprogramowania korzystający z monitora zwolniony jest z obowiązku implementowania niezawodnego, wydajnego mechanizmu. Dostawcy implementacji monitorów, którymi mogą być niezależni programiści jak również biblioteki, kończąc na platformach, np. .NET, gwarantują poprawność dostarczanych mechanizmów.

Platforma .NET udostępnia klasę Monitor znajdującą się w przestrzeni nazw System.Threading, która reprezentuje opisaną logikę. Wywołania statycznych metod Enter, TryEnter oraz Exit pozwalają na wejście do sekcji krytycznej, próbę wejścia oraz wyjście z sekcji krytycznej. Praca monitora opiera się na obiekcie, który reprezentuje prawo wejścia do sekcji krytycznej. W każdym czasie tylko jeden wątek może być właścicielem obiektu będącego zamkiem monitora i tym samym przebywać w sekcji krytycznej. Przykład pozyskania zamka przedstawiono poniżej:

```
1   private static void ThreadOperation()
2   {
3       for (var i = 0; i < 100000; i++)
4       {
5           System.Threading.Monitor.Enter(LockObject);
6           try
7           {
8               CriticalOperation();
9           }
10          finally
11          {
12              System.Threading.Monitor.Exit(LockObject);
13          }
14      }
15  }
```

Powyższy kod przedstawia metodę, która w kompletnym przykładzie zostaje wywołana przez wiele wątków jednocześnie. W celu zabezpieczenia procesu krytycznego, który nie jest operacją atomową, zastosowany został monitor. Sekcją krytyczną jest obszar wyznaczony między pozyskaniem zamka dla monitora w linii piątej oraz zwolnieniem w linii dwunastej. Istotne jest opakowanie sekcji krytycznej w blok try-finally, który przy ewentualnym wystąpieniu sytuacji wyjątkowej gwarantuje zwolnienie zamku przez monitor.

Praca z monitorem stanowi podstawowy i zalecany przez Microsoft sposób synchronizacji niskopoziomowej, jednak jest ona najczęściej stosowana przez skrót syntaktyczny wprowadzony w .NET, opierający się na słowie kluczowym lock. Poniżej przedstawiono sekcję krytyczną zabezpieczoną przez lock:

```
1   private static void ThreadOperation()
2   {
3       for (var i = 0; i < 100000; i++)
4       {
5           lock(LockObject);
6           {
7               CriticalOperation();
8           }
9       }
10  }
```

Zastosowanie skrótu syntaktycznego wprowadza czytelny mechanizm wyznaczający sekcję krytyczną, gwarantując poprawność zwolnienia zamku w przypadku sytuacji wyjątkowej.

Jawna praca z monitorem (nie poprzez słowo kluczowe lock) w całości opiera się na metodach statycznych pochodzących z klasy Monitor. W przypadku zwiększonych wymagań wobec monitora należy zapoznać się z przeciążeniami metod, które dostarczają funkcjonalności, takie jak specyfikacja czasu oczekiwania, możliwość bezzwłocznego informowania wątków oczekujących na zamek oraz sprawdzenie stanu monitora.

Jako najlepszą praktykę należy zapamiętać, iż sekcja krytyczna powinna być możliwie najkrótsza, co pozwala na szybsze przyznawanie praw dostępu oraz zmniejszenie szansy zakleszczenia. Monitory reprezentują mechanizm synchronizacji blokującej, ich zastosowanie jest powszechne, jednak nie należy ich stosować do operacji, które mogą być obsłużone przez operacje atomowe, które są formą synchronizacji nieblokującej.

3.3. Operacja atomowa

Operacją atomową jest operacja, która na określonym poziomie abstrakcji jest wykonywana w sposób niepodzielny. Operacje atomowe pozwalają rozwiązać problemy związane z zadaniami, które w sposób logiczny stanowią całość, jak np. inkrementacja. Zwiększanie licznika w językach wysokiego poziomu oznacza logiczną całość, jednak wykonywana jest na procesorze przy pomocy 3 kolejnych instrukcji, (wczytania zmiennej do rejestru tymczasowego, zwiększenie wartości rejestru tymczasowego o jeden, zapisania wyniku do rejestru) między którymi może nastąpić przerwanie.

Operacje atomowe to ciąg instrukcji wykonywanych przez procesor, które dzięki wsparciu ze strony sprzętowej nie mogą być rozdzielone przeplotem. Mechanizm ten stanowi najniższą formę synchronizacji, do której ma dostęp programista. Operacje atomowe są przykładem synchronizacji nieblokującej, czyli takiej, w której wątki, pracując nad sekcjami krytycznymi, nie wywłaszczają (technika, w której algorytm szeregujący może wstrzymać aktualnie wykonywane zadanie, aby umożliwić działanie innemu) się wzajemnie, powodując oczekiwanie na wejście.

Klasą reprezentującą operacje nieblokujące w .NET jest typ Interlocked umożliwiający następujące metody jako operacje atomowe:

- inkrementacja,
- wymiana wartości zmiennych,
- porównanie wartości zmiennych,
- dekrementacja,
- odczyt,
- utworzenie bariery pamięci.

Jeżeli jedyną operacją wykonywaną w sekcji krytycznej w ramach przykładu z rozdziału o monitorach byłaby inkrementacja, możliwa byłaby następująca zmiana kodu, na wykorzystujący operacje atomowe. Wielowątkowa inkrementacja wartości przez operację atomową jest przedstawiona następująco:

```
1  private static void ThreadOperation()
2  {
3      for (var i = 0; i < 100000; i++)
4      {
5          System.Threading.Interlocked.Increment(ref Counter);
6      }
7  }
```

Operacje nieblokujące cechują się znacznie lepszym czasem wykonania, ponieważ nie ma oczekiwania wątków na pozwolenie wejścia do sekcji krytycznej.

4. Podsumowanie

Platforma .NET dostarcza mechanizmy synchronizacji niskiego poziomu, umożliwiające wydajne synchronizowanie sekcji krytycznych oraz współdzielonych zasobów. Tworzenie własnych mechanizmów synchronizujących w ramach zespołu programistów może doprowadzić do powstania błędów trudnych do wykrycia, co nie jest zalecaną praktyką. Wybierając między synchronizacją blokującą oraz nieblokującą, należy wybrać drugą ze względu na szybsze przetwarzanie. Niestety wybór nieblokujących mechanizmów nie jest często możliwy, gdyż wymaga wsparcia sprzętowego, w postaci operacji atomowych, które mogą być wykorzystane do jednoczesnej nieblokującej pracy. Zalecaną dalszą lekturą jest zgłębienie wiedzy o zaawansowanych mechanizmach synchronizacji niskopoziomowej, np. bariery pamięci oraz strategie wysokopoziomowego programowania współbieżnego.

Bibliografia

1. Albahari B.A.J.: C# 6.0 w pigułce. Wyd. Helion, ISBN: 978-83-283-2424-7, Gliwice 2016.
2. Ben-Ari M.: Podstawy programowania współbieżnego i rozproszonego. Wydawnictwo Naukowo-Techniczne, ISBN 83-204-1996-4, Warszawa1996.
3. Padua D.: Encyclopedia of Parallel Computing. Springer, ISBN 9780387097657, 2011, p. 524.
4. Raynal M.: Algorithms for Mutual Exclusion. Massachusetts: MIT Press, ISBN 0-262-18119-3, 1986.
5. Raynal M.: Concurrent Programming: Algorithms, Principles, and Foundations. Springer Science & Business Media. p. 9. ISBN 3642320279, 2012, p. 9.
6. Tanenbaum A.S.: Interprocess comminucation. [w:] Modern operating systems. Upper Saddle River, Pearson Prentice Hall, ISBN Q-1B-filBMST-L, 2009, p. 115-142.
7. Unger S.: Hazards, Critical Races, and Metastability. "IEEE Transactions on Computers", 1995, p. 754–768.
8. Akka.NET: Terminology and Concepts, [Online]. Available: http://getakka.net/docs/concepts/terminology. [Data uzyskania dostępu: 16.08.2017].
9. Microsoft Documents, "Overview of Synchronization Primitives", [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives. [Data uzyskania dostępu: 16.08.2017].