WYŻSZA SZKOŁA BIZNESU – NATIONAL-LOUIS UNIVERSITY
W NOWYM SĄCZU

WYDZIAŁ INFORMATYKI

KIERUNEK: INFORMATYKA

SPECJALNOŚĆ: INŻYNIERIA OPROGRAMOWANIA

Bartłomiej Dawidów

Nr albumu: 9057

Wykrywanie deadlocków w aplikacjach .NET

Deadlock detection in .NET applications

Praca licencjacka

Promotor: dr Tomasz Gorazd

Nowy Sącz 2008

1. Spis treści

2. Wstęp

Wielowątkowość stała się nieodłączną cechą większości współczesnych programów. W dobie procesorów wielordzeniowych, obecnych w każdym komputerze osobistym, zyskuje jeszcze większą popularność. Patrząc na rozwój komputerów w przeciągu ostatnich lat można się spodziewać systemów komputerowych z coraz większą ilością procesorów i rdzeni. Aby móc w pełni wykorzystywać ich możliwości, potrzebne są odpowiednie narzędzia programistyczne, umiejętnie stosowane przez twórców oprogramowania.

Platforma Microsoft .NET, dzięki wygodnym narzędziom, dobrej dokumentacji i szerokiemu wachlarzowi dostępnych komponentów zdobyła akceptację wielu programistów. Zaawansowane prace nad implementacjami open source pod inne systemy operacyjne jeszcze bardziej poszerzyły grono użytkowników.

W tej pracy omawiam problem zakleszczenia wątków w kontekście platformy Microsoft .NET. W pierwszych rozdziałach przedstawiam różnorakie metody synchronizacji pracy wątków dostarczonych przez tą platformę. Następnie wprowadzony zostaje problem deadlocka i jego przyczyn. W końcu prezentuję możliwe sposoby wykrywania zakleszczeń oraz przykładową implementację jednego z nich.

3. Podstawowe metody synchronizacji wątków w .NET

Klasa System.Threading.Monitor

Jednym z najpopularniejszych sposobów synchronizacji pracy wątków są sekcje krytyczne. W języku C# sekcję krytyczną tworzymy z użyciem słowa kluczowego `lock`.

```
lock(someObject)
{
    Ciąg instrukcji...
}
```

W ten sposób uzyskujemy blokadę na wskazany obiekt (someObject) na początku bloku kodu i zwalniamy ją na końcu bloku. W danym momencie tylko jeden wątek może posiadać blokadę na dany obiekt. Jeżeli wątek spróbuje uzyskać blokadę na obiekt, który już jest zablokowany przez inny wątek, to zostanie dodany do kolejki wątków oczekujących i wstrzymany do czasu, aż przyjdzie jego kolej.

Wątek może zablokować ten sam obiekt wielokrotnie. Obiekt pozostanie zablokowany do momentu zwolnienia wszystkich blokad przez dany wątek.

Warto zwrócić uwagę, że uzyskanie blokady na obiekcie nie ma wpływu na działanie samego obiektu. Wszystkie jego metody i właściwości działają bez zmian i mogą być bez przeszkód wywoływane z dowolnego wątku. Obiekt służy nam tutaj tylko jako „kotwica", do której podczepiamy informacje o blokadach. W praktyce blokady zakładamy często na obiekty klasy `Object`, które tworzymy tylko po to, żeby służyły jako właśnie takie „kotwice".

Prześledźmy działanie słowa kluczowego `lock` podglądając kod CIL przykładowego programu.

```
static class Program
{
    static object obj = new object();

    static void Main()
    {
        lock (obj)
        {
        }
    }
}
```

W wyniku kompilacji powyższego kodu, dostajemy następujący kod CIL dla metody Main:

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       22 (0x16)
  .maxstack  2
  .locals init ([0] object CS$2$0000)
  IL_0000:  ldsfld     object SampleLock.Program::obj
  IL_0005:  dup
  IL_0006:  stloc.0
  IL_0007:  call void [mscorlib]System.Threading.Monitor::Enter(object)
  .try
  {
    IL_000c:  leave.s    IL_0015
  }  // end .try
  finally
  {
    IL_000e:  ldloc.0
    IL_000f:  call void [mscorlib]System.Threading.Monitor::Exit(object)
    IL_0014:  endfinally
  }  // end handler
  IL_0015:  ret
} // end of method Program::Main
```

Jak widać, w rzeczywistości słowo kluczowe `lock` podczas kompilacji jest zamieniane na wywołanie na początku bloku metody `System.Threading.Monitor.Enter`, a na końcu bloku metody `System.Threading.Monitor.Exit`. W efekcie słowo kluczowe `lock` daje taki sam efekt jak następujący kod:

```
System.Monitor.Enter(someObject);
try
{
     Ciąg instrukcji...
}
finally
{
     System.Monitor.Exit(someObject);
}
```

Umieszczenie zwolnienia blokady w bloku finally zabezpiecza nas przed pozostawieniem blokady w przypadku wystąpienia wyjątku.

Metoda `Monitor.TryEnter` pozwala ponadto na próbę uzyskania blokady z ustalonym limitem czasowym.

Klasa ReaderWriterLockSlim

Często zdarza się, że pewne wątki potrzebują tylko dostępu do odczytu danych. Dopóki żaden wątek nie modyfikuje danych, nie ma przeszkód, aby wiele wątków równocześnie czytało ten sam obszar pamięci. Na potrzeby takich sytuacji, .NET udostępnia klasę `System.Threading.ReaderWriterLockSlim`. Klasa ta pozwala jednoczesnie na blokadę dowolnej liczbie wątków czytających lub jednemu wątkowi piszącemu. Dodatkowo jeden z wątków czytających może być w trybie „upgradeowalnym" do trybu zapisu.

Blokady uzyskujemy i zwalniamy wywołując na obiekcie tej klasy metody m.in.

- `EnterReadLock, ExitReadLock`
- `EnterWriteLock, ExitWriteLock`
- `EnterUpgradeableReadLock, ExitUpgradeableReadLock`

Przykładowa blokada tylko do odczytu może być stworzona w następujący sposób:

```
ReaderWriterLockSlim lock = new ReaderWriterLockSlim();
lock.EnterReadLock();
try
{
     Ciąg instrukcji...
}
finally
{
     lock.ExitReadLock();
}
```

Synchronizacja bez blokad

Metody oparte o blokady, takie jak sekcje krytyczne, są często najprostszym rozwiązaniem problemu synchronizacji wątków. Jendak ich wadą jest stosunkowa powolność. Po pierwsze wynika to z czasochłonności samych operacji uzyskania i zwalniania blokad. Po drugie spowodowane jest faktem, że stosując blokady często zbyt restrykcyjnie przydzielamy wątkom dostęp do zasobów komputera. Wydajność naszych programów wielowątkowych możemy istotnie zwiększyć używając bardziej wyrafinowanych narzędzi oferowanych przez platformę Microsoft .NET.

VolatileRead, VolatileWrite

Metoda `System.Threading.Thread.VolatileRead` służy do odczytania aktualnej wartości pola klasy. Metoda `Thread.VolatileWrite` służy do zapisania wartości pola. Stosując te metody unikamy optymalizacji, która mogłaby pominąć niektóre operacje odczytu lub zapisu lub zamienić ich kolejność. Dzięki temu mamy pewność, że zawsze operujemy na najnowszej wartości pola, także na komputerze wieloprocesorowym. Odbywa się to jednak kosztem czasochłonności, gdyż `VolatileRead` i `VolatileWrite` wymagają odwołania się za każdym pamięci RAM.

W języku C# możemy deklarację pola klasy poprzedzić modyfikatorem `volatile`, który powoduje, że każde odwołanie do tego pola pomija optymalizację.

Przykładem zastosowania tych metod może być zatrzymanie pracy wątku z użyciem pola ustawianego przez inny wątek.

```
static class Program
{
    public static void Main()
    {
      Thread thread = new Thread(Start);
            thread.Start();
            Thread.Sleep(5000);
            _terminate = true;
            thread.Join();
    }
    private static volatile bool _terminate;
    public void Start()
      {
            while (!_terminate)
            {
                  Thread.Sleep(500);
            }
    }
}
```

Gdyby pole `_terminate` nie zostało poprzedzone modyfikatorem `volatile` istniałoby niebezpieczeństwo, że wątek nigdy się nie zatrzyma. W kolejnych obiegach pętli mógłby korzystać ze scacheowanej w rejestrze procesora wartości pola, nie wiedząc, że inny wątek ją zmodyfikował.

Klasa Interlocked

Klasa `System.Threading.Interlocked` dostarcza metody pozwalające wykonać pewne operacje jako operacje niepodzielne (atomowe). Te metody to:

- `Add, Increment, Decrement` – dodają zadaną liczbę do zmiennej i zwracają zmienioną wartość jako operację niepodzielną (Fetch-And-Add)
- `CompareExchange` – porównuje dwie zmienne i, jeśli są równe, ustawia pierwszą z nich na zadaną wartość oraz zwraca wartość oryginalną jako operację niepodzielną (Compare-And-Swap)
- `Read` – odczytuje wartość pola typu Int64 jako operację niepodzielną, nawet na komputerze 32-bitowym

Przeładowania tych metod operujące na zmiennych 64-bitowych są niepodzielne także na komputerach 32-bitowych, ale tylko jeśli wszystkie wątki przy dostępie do tych zmiennych używają metod klasy `Interlocked`.

Algorytmy lockless

Operacje udostępniane przez klasę `Interlocked` pozwalają tworzyć algorytmy i struktury (np. listy, kolejki, stosy), które nie wymagają stosowania blokad. Takie algorytmy

nazywane są algorytmami lockless. Tematyka ta została szeroko omówiona w pracy doktorskiej „Efficient and Practical Non-Blocking Data Structures”, Hakan Sundell.

MemoryBarrier

Metoda `System.Threading.Thread.MemoryBarrier` służy do zachowania kolejności zapisów i odczytów z pamięci poprzez synchronizację cache’a procesora z pamięcią główną. Jest ona wymagana w przypadku w przypadku niektórych systemów wieloprocesorowych.

Weźmy poniższy przykład. Niech pewien wątek wywołuje poniższy kod:

```
imię = "Jan";
nazwisko = "Kowalski";
waga = 75.0;
daneGotowe = true;
```

Chcemy, aby flaga `DaneGotowe` służyła do poinformowania inne wątki, że mogą już odczytywać dane osoby. Jednak w wyniku optymalizacji i sposobu działania niektórych procesorów, kolejność zapisu i odczytu danych nie zawsze jest zachowywana. Może to doprowadzić do sytuacji, w której wątki odczytają błędne dane. Aby temu zapobiec, dodamy wywołanie metody `MemoryBarrier`.

```
imię = "Jan";
nazwisko = "Kowalski";
waga = 75.0;
Thread.MemoryBarrier();
daneGotowe = true;
```

Przykładowy wątek czytający mógłby działać tak:

```
if (daneGotowe)
{
    Thread.MemoryBarrier();
    imięINazwisko = imię + " " + nazwisko;
}
else
{
     // Np. czekaj dalej albo zrezygnuj
}
```

W rzeczywistości niektóre z wymienionych w poprzednich rozdziałach sposobów synchronizacji wykorzystuje wewnętrznie metodę `MemoryBarrier`. Potwierdza to podgląd kodu CIL biblioteki systemowej mscorlib.dll.

Przykładowo, metoda `VolatileWrite(int32& address, int32 'value')`, ma następujący kod:

```
.method public hidebysig static void  VolatileWrite(int32& address,
                                                     int32 'value') cil
managed noinlining
{
  // Code size       9 (0x9)
  .maxstack  8
  IL_0000:  call       void System.Threading.Thread::MemoryBarrier()
  IL_0005:  ldarg.0
  IL_0006:  ldarg.1
```

```
  IL_0007:  stind.i4
  IL_0008:  ret
} // end of method Thread::VolatileWrite
```

Przyglądnijmy się kolejno wywoływanym instrukcjom:

1. `call`: Wywołujemy metodę `MemoryBarrier`. Dzięki temu mamy pewność, że wszystkie poprzednie zapisy znajdą się teraz w pamięci głównej.
2. `ldarg.0`: Ładujemy argument `address` na stos.
3. `ldarg.1`: Ładujemy argument `value` na stos.
4. `stind.i4`: Zapisujemy wartość `value` pod adresem `address`.
5. `ret`: Koniec metody.

Dla porównania metoda `VolatileRead(int32& address)` wygląda następująco:

```
.method public hidebysig static int32  VolatileRead(int32& address) cil
managed noinlining
{
  // Code size       10 (0xa)
  .maxstack  1
  .locals init (int32 V_0)
  IL_0000:  ldarg.0
  IL_0001:  ldind.i4
  IL_0002:  stloc.0
  IL_0003:  call       void System.Threading.Thread::MemoryBarrier()
  IL_0008:  ldloc.0
  IL_0009:  ret
} // end of method Thread::VolatileRead
```

Jak widać, gdyby w przykładzie z Janem Kowalskim pole `daneGotowe` było zadeklarowane jako `volatile`, nie byłoby potrzeby stosowania `MemoryBarrier`.

4. Deadlock

Przykład deadlocka w aplikacji .NET

Stosując blokady, możemy doprowadzić do sytuacji, w której dwa wątki będą czekać na siebie nawzajem.

```
void Thread1()
{
    whlie (true)
    {
      lock (resource1)
      {
            lock (resource2)
            {
                  }
      }
    }
}
void Thread2()
{
        e (true)
```

```
        lock (resource2)
        {
              lock (resource1)
              {
              }
        }
     }
}
```

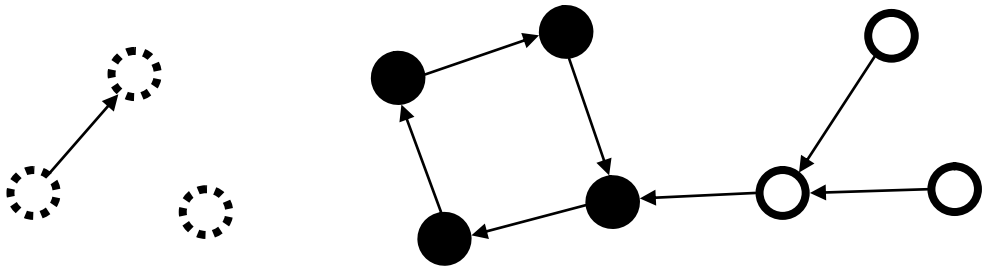
Powyższe dwie metody różnią się kolejnością zakładania blokad. Jeśli uruchomimy te metody każdą we własnym wątku, może wystąpić następujący bieg wydarzeń:

1. Wątek 1 blokuje zasób 1.
2. System wywłaszcza wątek 1, wątek 2 rozpoczyna pracę.
3. Wątek 2 blokuje zasób 2.
4. Wątek 2 próbuje zablokować zasób 1, jednak jest on już zablokowany, więc oczekuje na jego zwolnienie.
5. System wywłaszcza wątek 2, wątek 1 kontynuuje pracę.
6. Wątek 1 próbuje zablokowac zasób 2, jednak jest on już zablokowany, więc oczekuje na jego zwolnienie.

Jest to przykład deadlocka (zakleszczenia), w którym biorą udział dwa wątki. Analogiczna sytuacja może wystąpić w przypadku większej ilości wątków.

Graf oczekiwań

Aby formalnie zdefiniować pojęcie zakleszczenia, możemy użyć grafu oczekiwań. Jest to graf skierowany, w którym węzęł reprezentuje wątek, a krawędź wskazuje, że wątek na początku krawędzi oczekuje na zwolnienie zasobu przez wątek na końcu krawędzi. Jeżeli w takim grafie wystąpi cykl, to mamy doczynienia z deadlockiem. Zakleszczone zostają wszystkie wątki w cyklu. Pośrednio w zakleszczeniu biorą także udział wątki, od których istnieje ścieżka do wątków w cyklu.



● wątki bespośrednio uczestniczące w zakleszczeniu

 wątki pośrednio uczestniczące w zakleszczeniu

 wątki nie uczestniczące w zakleszczeniu

Zapobieganie deadlockom

Aby zapobiec deadlockom w naszych programach, należy niedopuścić do powstania cyklu. W praktyce można to osiągnąć poprzez unikanie zagnieżdżonych blokad. W sytuacji, gdy zagnieżdżona blokada jest konieczna, należy pilnować, aby kolejność w której zakładamy blokady na obiekty była taka sama we wszystkich wątkach. Możemy to zrobić np. poprzez przyporządkowanie każdemu blokowanemu obiektowi poziomu i zakładanie blokad zawsze w takiej kolejności, aby obiekty o wyższym poziomie były blokowane wcześniej. W przypadku obiektów tego samego typu jako poziom możemy przyjąć jakąś jego unikalną cechę, np. identyfikator lub adres w pamięci.

Przykładowo, gdybyśmy mieli dwa obiekty typu `Konto`i chcieli stworzyć bezpieczną metodę do przelewania środków z konta na konto, blokady moglibyśmy zakładać w kolejności uzależnionej od numeru konta.

```
class Konto
{
    public int numer;

    public int środki;

    public static void Przelej(Konto skąd, Konto dokąd, int ile)
    {
        Konto a, b;
        if (skąd.numer > dokąd.numer)
        {
            a = skąd;
            b = dokąd;
        }
        else
        {
            a = dokąd;
            b = skąd;
        }
        lock (a)
        {
            lock (b)
            {
                skąd.środki -= ile;
                dokąd.środki += ile;
            }
        }
    }
}
```

5. Możliwe sposoby wykrywania deadlocków w aplikacjach .NET

Własne klasy do synchronizacji

Jednym ze sposobów wykrycia zakleszczenia w aplikacji mogłoby być stworzenie własnych klas do sekcji krytycznych. Metody tych klas, zakładając i zwalniając blokady budowałyby graf oczekiwań. Na podstawie tego grafu wykrywałyby zakleszczenie. Do napisania tych klas możnaby użyć m.in. metod klasy `Interlocked`.

Można także stworzyć klasy, które przypilnują kolejność, w której żądamy blokad. Każdy obiekt do zakładania blokady miałby przypisany poziom. Próba założenia zagnieżdżonej blokady na obiekcie o poziomie wyższym od poziomów posiadanych blokad rzucałaby wyjątek. Takie podejście uniemożliwiłoby wystąpienie zapętlenia i uczyniło z naszego grafu oczekiwań zbiór drzew.

Wadą tego rozwiązania jest konieczność używania tych klas w całym programie. Nie możemy wtedy używać np. słowa kluczowego `lock` w języku C#. W przypadku istniejących programów, takie podejście wymagałoby zmiany kodu całego programu. W sytuacji, gdy nie mamy kodu źródłowego programu (lub np. którejś biblioteki), byłoby bezużyteczne.

Zmodyfikowany host

Platforma .NET daje możliwość tworzenia własnych hostów, w których można uruchamiać nasze programy. Takie hosty mogą modyfikować działanie niektórych elementów platformy, w tym synchronizacji wątków. Tworząc własny host, moglibyśmy podobnie jak w przypadku własnych klas, monitorować blokady i wykrywać zakleszczenia.

Zaletą tego rozwiązania jest brak potrzeby modyfikowania samych programów. Monitorowana aplikacja może używać standardowych mechanizmów synchronizacji dostarczanych przez platformę .NET.

Wadą jest niemożność skorzystania z niektórych funkcji standardowego hosta, m.in. kontroli bezpieczeństwa. Ponadto, jeśli chcielibyśmy uruchomić aplikację ASP.NET, musielibyśmy stworzyć oddzielny host dla takich aplikacji. W przypadku aplikacji Silverlight, które uruchamiane są w przeglądarce internetowej na komputerze klienta, stosowanie własnego hosta nie wchodzi w grę, gdyż to Silverlight jest wtedy narzuconym hostem.

Modyfikacja kodu MSIL

Kolejnym podejściem jest modyfikowanie kodu MSIL skompilowanej aplikacji. Wywołania standardowych metod synchronizacji wątków, np. `Monitor.Enter` i `Monitor.Exit` zastępujemy własnymi metodami. Te metody poza funkcjonalnością oryginałów, zbierają informacje potrzebne o blokadach i wykrywają zakleszczenia.

Zaletą tego podejścia jest, podobnie jak w przypadku własnego hosta, możliwość zastosowania jej do aplikacji napisanych z użyciem metod synchronizacji dostarczonych przez platformę .NET. Ponadto aplikacja ze zmodyfikowanym kodem MSIL jest

samodzielnym, samokontrolującym się programem. Może on zostać uruchomiony pod dowolnym hostem, także ASP.NET lub Silverlight.

Wadą jest konieczność modyfikowania każdego modułu, w którym chcemy wykrywać zakleszczenia.

Przykładowa implementacja tego rozwiązania jest zaprezentowana w następnym rozdziale.

6. DotDeadlock

Cel systemu

Celem systemu DotDeadlock jest wykrywanie zakleszczeń w aplikacjach .NET. Wykrywanie odbywa się dzięki zastąpieniu wywołań standardowych metod synchronizacji wątków własnymi metodami. Te metody zastępcze opakowują funkcjonalność metod oryginalnych oraz dodają operacje potrzebne do znalezienia zakleszczeń.

Opis działania

System składa się z trzech podstawowych składników:

- DotDeadlock – moduł zawierający klasy służące do przetwarzania modułów
- DotDeadlockRuntime – moduł zawierający klasy monitorujące działanie aplikacji
- DotDeadlockCmd – interfejs wiersza poleceń do modułu DotDeadlock

DotDeadlock
Moduł DotDeadlock dostarcza klasę `DotDeadlockBuilder`, która służy do przetwarzania modułów z pomocą metody klasy `ProcessAssemblyFile`. Metoda ta ma następujące argumenty:

- `fileName` (typu `string`) – określa plik modułu do ptrzetworzenia
- `updateSymbols` (typu `bool`) – określa, czy po przetwarzaniu ma nastąpić uaktualnienie bazy symboli
- `callsReplaced` (typu `out int`) – zwraca ilość podmienionych metod

W wyniku tej metody nastąpi podmienienie wywołań metod `System.Threading.Monitor.Enter` i `Monitor.Exit` wywołaniami metod zastępczych z klasy `DotDeadlock.Runtime.Monitor`.

Uaktualnienie bazy symboli umożliwia debugowanie przetworzonego modułu.

Do przetwarzania modułów wykorzystywana jest biblioteka Mono.Cecil. Główna metoda podmnieniająca metody ma następującą postać:

```
/// <summary>
/// Przetwarza wybrany moduł.
/// Podmienia wywołania metod zgodnie z argumentem replacements.
```

```
/// Zwraca liczbę podmienionych wywołań w argumencie callsReplaced.
/// </summary>
/// <param name="assembly">Moduł do przetworzenia.</param>
/// <param name="replacements">
/// Podmiany wywołań.
/// Określa wywołania których metody mają być podmienione i na co.
/// </param>
/// <param name="updateSymbols">Jeśli true, symbole zostaną
/// uaktualnione.</param>
/// <param name="callsReplaced">Liczba podmienionych wywołań.</param>
private static void ProcessAssembly(
    AssemblyDefinition assembly,
    MethodCallReplacement[] replacements,
    bool updateSymbols,
    out int callsReplaced)
{
    // Sprawdza poprawność argumentów
    if (assembly == null)
        throw new ArgumentNullException("assembly");
    if (replacements == null)
        throw new ArgumentNullException("replacements");

    MethodReference[] oldMethods =
        new MethodReference[replacements.Length];
    MethodReference[] newMethods =
        new MethodReference[replacements.Length];

    // Pobieramy główny moduł
    ModuleDefinition module = assembly.MainModule;

    if (updateSymbols)
        module.LoadSymbols();

    // Przygotowuje tablice za zaimportowanymi metodami
    for (int i = 0; i < replacements.Length; ++i)
    {
        oldMethods[i] = module.Import(replacements[i].OldMethod);
        newMethods[i] = module.Import(replacements[i].NewMethod);
    }

    callsReplaced = 0;

    foreach (TypeDefinition type in module.Types)
    {
        foreach (MethodDefinition method in type.Methods)
        {
            MethodBody body = method.Body;
            CilWorker worker = body.CilWorker;

            int i = 0;
            while (i < body.Instructions.Count)
            {
                Instruction instruction = body.Instructions[i];

                // Sprawdzamy czy instrukcja jest wywołaniem metody
                if (instruction.OpCode == OpCodes.Call)
                {
                    // Sprawdzamy, czy wowływana jest któraś z metod do
                    // podmiany
                    for (int iReplacement = 0;
                        iReplacement < replacements.Length;
```

```
                        iReplacement++)
                    {
                        if (instruction.Operand.ToString() ==
                            oldMethods[iReplacement].ToString())
                        {
                            // Podmieniamy instrukcję
                            worker.Replace(instruction, worker.Create(
                                OpCodes.Call, newMethods[iReplacement]));
                            callsReplaced++;
                        }
                    }
                }

                i++;
            }
        }
    }

    if (updateSymbols)
        module.SaveSymbols();
}
```

Za pośrednictwem klas udostępnianych przez bibliotekę Mono.Cecil, możemy iterować przez poszczególne typy, metody. Ostatecznie, z użyciem obiektu klasy CilWorker iterujemy instrukcje z danej metody. Iterując instrukcje szukamy instrukcji typu call. Operandem takich instrukcji jest metoda do wywołania. W momencie, gdy znajdziemy wywołanie pasujące do naszego wzorca, podmieniamy je na wywołanie do metody zastępczej z użyciem metody CilWorker.Replace.

DotDeadlockRuntime

Moduł DotDeadlockRuntime to moduł dostarczający klasy służące do wykrywania zakleszczeń w trakcie działania monitorowanego programu.

Podstawowe elementy modułu to:

- Klasa `DotDeadlock.Runtime.Monitor` zbierająca dane o zakładanych przez wątki blokadach
- Klasa `DotDeadlock.Runtime.WorkerThread`, która w wątku roboczym monitoruje blokady i wykrywa deadlocki

Faktyczne wykrywanie zakleszczeń odbywa się tylko w sytuacji, gdy któryś wątek przez zbyt długi czas nie będzie mógł uzyskać blokady.

Klasa `Monitor` dostarcza metody zastępcze dla metod `Monitor.Enter` i `Monitor.Exit`, które opakowują funkcjonalność oryginalnych metod i zbierają informacje potrzebne do wykrycia deadlocka. Działanie zastępczej metody `Enter` odbywa się w następujący sposób:

1. Zapisanie czasu oraz obiektu, na którym wątek próbuje uzyskać blokadę i dodanie wątku do listy wątków oczekujących.
2. Wywołanie oryginalnej metody `Monitor.Enter`.
3. Dodanie zablokowanego obiektu do listy obiektów zablokowanych przez aktualny wątek i zdjęcie wątku z listy wątków oczekujących.

Metoda `Exit` działa następująco:

1. Wywołanie oryginalnej metody `Monitor.Exit`.
2. Usunięcie obiektu z listy obiektów zablokowanych przez aktualny wątek.

O każdym wątku zbierane są następujące informacje:

- Lista aktualnie zablokowanych obiektów. Jeżeli wątek zablokował ten sam obiekt dwukrotnie (w zagnieżdżeniu), to obiekt dwukrotnie znajdzie się na liście.
- Obiekt, na który wątek obecnie oczekuje.
- Czas, od którego wątek oczekuje na obiekt.
- Stos wywołań w momencie próby uzyskania blokady, jeśli w konfiguracji włączono śledzenie stosów wywołań.

Te dane każdy wątek zbiera dla siebie jako pola `ThreadStatic`. Pole oznaczone atrybutem `ThreadStatic` ma wartość przechowywaną oddzielnie dla każdego wątku.

Wątek roboczy monitoruje listę wątków oczekujących. Jeżeli stwierdzi, że jakiś wątek znajduje się na liście dłużej niż określony próg czasu, to uruchamia faktyczny algorytm wykrycia zakleszczeń. W oparciu o dane zbierane przez wątki wykrywa zapętlenia w grafie oczekiwań. W przypadku wykrycia zakleszczenia zapisuje informacje o nim do pliku dotdeadlock.log.

Działanie systemu wykrywania deadlocków można skonfigurować dodając do katalogu z modułem plik dotdeadlock.xml, zgodny ze schematem Configuration.xsd. Przykładowy plik konfiguracyjny jest dołączony do programu. Plik konfiguracyjny pozwala ustawić następujące parametry:

- logFileName - ścieżka do pliku log, domyślnie dotdeadlock.log
- workerThreadWaitTimeoutMilliseconds - odstępy, w jakich wątek roboczy szuka deadlocków, domyślnie 1000
- deadlockDetectionDeferralMilliseconds - opóźnienie wykrywania deadlocków, domyślnie 1000. Opóźnienie ma na celu zwiększenie wydajności systemu. Faktyczne budowanie grafu wątków rozpoczyna się dopiero w momencie, gdy któremuś wątkowi nie uda się założyć blokady na obiekt po czasie dłuższym niż ten parametr.
- printStackTrace - true lub false. Jeśli true, w pliku log zapisane zostaną stosy wywołań wątków, które powodują deadlocka. Domyślnie true.
- printLockedObject - true lub false. Jeśli true, w pliku log zapisane zostaną wartości ToString() obiektów, które biorą udział w deadlocku. Domyślnie true.

DotDeadlockCmd

DotDeadlockCmd.exe to program pozwalający przetwarzać moduły z wiersza poleceń. Wykorzystuje w tym celu klasę `DotDeadlockBuilder` z modułu DotDeadlock.

Aby przetworzyć moduł należy wywołać komendę:

```
.ockCmd.exe TwójModuł.exe
```

Aby odświerzyć bazę symboli w przetwarzanym module należy wywołać powyższą komendę z opcją `-s`:

```
DotDeadlockCmd.exe -s TwójModuł.exe
```

Do poprawnego działania, w katalogu z naszym modułem musi się znajdować moduł runtime DotDeadlockRuntime.dll. Opcja -r spowoduje automatyczne utworzenie tego modułu.

```
DotDeadlockCmd.exe -rs TwójModuł.exe
```

Przykład 1

Pierwszy przykładowy program powoduje zakleszczenie w wyniku próby założenia przez dwa wątki dwóch blokad w odwrotnej kolejności. Program jest dołączony do pracy jako projekt Sample1.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Diagnostics;

namespace Sample
{
    static class Program
    {
        static void Main(string[] args)
        {
            Thread thread1 = new Thread(Start1);
            Thread thread2 = new Thread(Start2);

            thread1.Start();
            thread2.Start();

            thread1.Join();
            thread2.Join();
        }

        static object _object1 = new NamedObject("1");
        static object _object2 = new NamedObject("2");

        const int SLEEP_TIMEOUT = 1000;

        static void Start1()
        {
            while (true)
            {
                Console.WriteLine("Thread 1 locking object 1");
                lock (_object1)
                {
                    Console.WriteLine("Thread 1 locked object 1");
                    Thread.Sleep(SLEEP_TIMEOUT);
                    Console.WriteLine("Thread 1 locking object 2");
                    lock (_object2)
                    {
                        Console.WriteLine("Thread 1 locked object 2");
                        Thread.Sleep(SLEEP_TIMEOUT);
                    }
                    Console.WriteLine("Thread 1 unlocked object 2");
                }
```

```
                Console.WriteLine("Thread 1 unlocked object 1");
            }
        }

        static void Start2()
        {
            while (true)
            {
                Console.WriteLine("Thread 2 locking object 2");
                lock (_object2)
                {
                    Console.WriteLine("Thread 2 locked object 2");
                    Thread.Sleep(SLEEP_TIMEOUT);
                    Console.WriteLine("Thread 2 locking object 1");
                    lock (_object1)
                    {
                        Console.WriteLine("Thread 2 locked object 1");
                        Thread.Sleep(SLEEP_TIMEOUT);
                    }
                    Console.WriteLine("Thread 2 unlocked object 1");
                }
                Console.WriteLine("Thread 2 unlocked object 2");
            }
        }
    }

    class NamedObject
    {
        public NamedObject(string name)
        {
            _name = name;
        }

        private string _name;

        public override string ToString()
        {
            return _name;
        }
    }
}
```

Po uruchomieniu program wypisuje na konsolę informacje o przebiegu wydarzeń:

```
Thread 2 locking object 2
Thread 1 locking object 1
Thread 2 locked object 2
Thread 1 locked object 1
Thread 1 locking object 2
Thread 2 locking object 1
```

Dalsze działanie programu jest niemożliwe ze względu na deadlock. Oba wątki uzyskały po jednej blokadzie i zostały wstrzymane próbując uzyskać drugą blokadę.

Aby zdiagnozować program z użyciem systemu DotDeadlock musimy najpierw przetworzyć go programem DotDeadlockCmd. W tym celu wywołujemy komendę w katalogu ze skompilowanym programem:

```
.ockCmd.exe -rs Sample1.exe
```

Biblioteka DotDeadlockRuntime zostanie automatycznie umieszczona w katalogu. W pliku dotdeadlock.xml, w którym uaktywniamy śledzenie stosu wywołań dla wątków oraz zapisywanie nazwy obiektu.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://dawidow.pl/dotdeadlock/configuration">
<logFileName>dotdeadlock.log</logFileName>
<workerThreadWaitTimeoutMilliseconds>1000</workerThreadWaitTimeoutMillisec
onds>
<deadlockDetectionDeferralMilliseconds>1000</deadlockDetectionDeferralMill
iseconds>
<printStackTrace>true</printStackTrace>
<printLockedObject>true</printLockedObject>
</configuration>
```

To wszystko czego potrzebujemy do przeprowadzenia diagnozy. Jeżeli teraz uruchomimy program, to w jego katalogu zostanie utworzony plik dotdeadlock.log. Zawartość tego pliku informuje nas o wykrytym deadlocku i pozwala zlokalizować jego źródło:

```
Started at 2008-11-16 20:36:30
...
2008-11-16 20:36:32 Potential deadlock detected:
Thread 4 is waiting for thread 3 to release object "1"
   at System.Environment.get_StackTrace()
   at DotDeadlock.Runtime.ThreadData.AcquiringLock(Object obj)
   at DotDeadlock.Runtime.Monitor.Enter(Object obj)
   at Sample.Program.Start2() in C:\...\Program.cs:line 64
   at System.Threading.ExecutionContext.Run(ExecutionContext
   executionContext, ContextCallback callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
Thread 3 is waiting for thread 4 to release object "2"
   at System.Environment.get_StackTrace()
   at DotDeadlock.Runtime.ThreadData.AcquiringLock(Object obj)
   at DotDeadlock.Runtime.Monitor.Enter(Object obj)
   at Sample.Program.Start1() in C:\...\Program.cs:line 43
   at System.Threading.ExecutionContext.Run(ExecutionContext
   executionContext, ContextCallback callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
```

Przykład 2

Program jest dołączony do pracy jako projekt Sample2. W tym przykładzie możemy doprowadzić do deadlocka złożonego z zadanej liczby wątków. Każdy wątek blokuje dwa obiekty. Wątek `i` blokuje obiekt `i` oraz w zagnieżdżeniu obiekt `i + 1`. Ostatni wątek blokuje obiekty `i` oraz `0`. W ten sposób otrzymujemy cykl, w którym uczestniczą wszystkie wątki.

7. Podsumowanie

Wielowątkowość może być bardzo skutecznym narzędziem w rękach programistów. Jednak należy być świadomym związanych z nią zagrożeń i niuansów. Mam nadzieję, że w tej pracy udało mi się przybliżyć podstawowe problemy programowania wielowątkowego pod .NET, ze szczególnym uwzględnieniem kwestii zakleszczeń.

8. Bibliografia

Duffy, J. (2006). *Advanced Techniques To Avoid And Detect Deadlocks In .NET Apps*. Pobrano 17 listopada, 2008 z lokalizacji http://msdn.microsoft.com/en-gb/magazine/cc163618.aspx

Ecma International. (2006). *ECMA-335 Common Language Infrastructure (CLI), 4th edition (June 2006).* Pobrano 14 listopada, 2008 z lokalizacji http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf

Sundell, H. (2004). *Efficient and Practical Non-Blocking Data Structures.* Pobrano 14 listopada, 2008 z lokalizacji http://www.cs.chalmers.se/~phs/phd.pdf