Uniwersytet Bielsko-Bialski

LABORATORIUM

Obliczeń Równoległych i Systemów Rozproszonych

Sprawozdanie nr 12

interface programowania OPENMPI

GRUPA: 2B / SEMESTR: 5 / ROK: 3

Igor Gawłowicz / 59096

Cel ćwiczenia

Celem tego ćwiczenia jest zaznajomienie się z tworzeniem i zarządzaniem wątkami w języku C# za pomocą klasy Thread. Ćwiczenie to ma na celu pokazanie, jak równolegle działające wątki mogą wpływać na siebie nawzajem oraz jak można zastosować mechanizmy synchronizacji, takie jak Mutex, aby uniknąć wyścigów o zasoby współdzielone.

Przebieg ćwiczenia

W sposób bezpośredni, tworzeniu i zarządzaniu wątkami służy klasa Thread.

Za nim wątek będzie mógł być użyty należy najpierw zdefiniować stosowny obiekt tej klasy. Odbywa się to w tradycyjny sposób, zaś funkcja wątku podawana jest za pośrednictwem konstruktora Thread().

Utwórzmy z wątku głównego procesu jeden wątek potomny o nazwie thread. Pierwszy będzie wypisywał na konsoli znaki 'x' zaś drugi (potomny) 'o'.

```
using System;
using System.Threading; //...konieczne włącznie przestrzeni nazw wątków
namespace Thread
{
    class Program
    {
        const int loops = 100; //...krotność wykonania
        static void Main(string[] args)
        {
            //...definicja obiektu typu wątek (Thread), z funkcją wątku o()
            System.Threading.Thread thread = new System.Threading.Thread(o);
            //...uruchomienie wątku thread
            thread.Start();
            //...działanie własne wątku procesu głównego
            for (int i = 0; i < loops; i++) { Console.Write("x"); }
        }
        static void o()
        {
            for (int i = 0; i < loops; i++) { Console.Write("o"); }
        }
    }
}
```

W sposób przewidywany możemy zauważyć że zadanie wykonało się w sposób bardzo chaotyczny w związku z wyścigiem o czas procesora

```
ooooooooooooooooooooooooxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
C:\Szkola\t1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 18840) exited
with code 0.
```

Warto jeszcze, na wstępie zauważyć, że podobnie jak w POSIX wątek otrzymuje swój unikalny identyfikator ale - ponieważ jest to zmienna obiektowa - także i szeregu innych atrybutów. Przykładowo można mu nadać nazwę Name.

```
using System;
using System.Threading;
namespace Thread
{
    class Program
    {
        static void Main(string[] args)
        {
            //...nazwa dla wątku procesu głównego
            System.Threading.Thread.CurrentThread.Name = "MASTER";
            //...inicjowanie obiektu Thread
            System.Threading.Thread thread = new System.Threading.Thread(go);
            thread.Name = "SLAVE";
            thread.Start();
            //...i jeszcze działanie dla głównego - niech także wykona funkcję
wątku
            go(); //...funkcję wątku wykonują oba, choć z różnym rezulatem.
        }
        static void go()
        { Console.WriteLine("Hello from {0}",
System.Threading.Thread.CurrentThread.Name); }
    }
}
```

```
Hello from SLAVE
Hello from MASTER

C:\Szkola\t1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 11756) exited
with code 0.
```

Tworząc nowy wątek bezpiecznie będzie sprawdzić, czy znajduje się już w stanie wykonania. Służy temu własność

public bool System.Threading.Thread.IsAl IsAlive { get; }

Wykonanie dowolnego wątku można wstrzymać na czas określony w milisekundach, metodą

public static void System.Threading.Thread.Sleep(int millisecondsTimeout);

Oczywiście można napisać także - przykładowo

Thread.Sleep (TimeSpan.FromHours (1));

co przyniesie wstrzymanie na 1 godzinę

Wątkowi można polecić natychmiastowe zatrzymanie

public void System.Threading.Thread.Abort();

które generuje ThreadAbortException.

```
using System;
using System.Threading;
class Program
{
    static void Main(string[] args)
    {
        System.Threading.Thread thread = new System.Threading.Thread(work);
        thread.Start();
        while (!thread.IsAlive) ;
        Thread.Sleep(2000);
        thread.Abort();
    }
    static void work()
    {
        try
        {
            Console.Write("[thread start]");
            while (true) { Console.Write("."); }
        }
        catch (Exception error)
        {
            Console.WriteLine("[thread stop]");
            //Console.WriteLine( "...catched exception {0}",error.ToString() );
        }
        finally
        {
            Console.WriteLine("[work done]");
        }
    }
}
```

W przypadku tego zadania otrzymałem niespodziewany rezultat ponieważ pomimo tego że spodziewaliśmy się pewnego rodzaju komunikatu o błędzie, to w związku z nieobsługiwaną wersją środowiska .NET otrzymałem innego rodzaju wyjątek.

```
[thread
start]..........................................................................
..........................................................................
..........................................................................
..........................................................................
```

```
.....................................................................................
.....................................................................................
.....................................................................................
.....................................

...

.....................................System.PlatformNotSupportedException:
Thread abort is not supported on this platform.
   at System.Threading.Thread.Abort()
   at Program.Main(String[] args) in C:\Szkola\t1\ConsoleApp1\Program.cs:line 11

...
```

Prócz wymienionych a właściwie ponad nie, bo jest to zasadniczy sposób jaki należy użyć celem synchronizacji jest metoda

public void System.Threading.Thread.Join();

W kolejnym przykładzie wątek procesu głównego będzie tak długo kontynuował swe działanie do póki potomny nie zakończy.

```
using System;
using System.Threading;
class Program
{
    static void Main(string[] args)
    {
        System.Threading.Thread thread = new System.Threading.Thread(work);
        thread.Start();
        while (thread.IsAlive) { Console.Write("."); }
        thread.Join();
    }
    static void work()
    {
        Console.Write("[thread start]");
        Thread.Sleep(100);
        Console.WriteLine("[thread stop]");
    }
}
```

Tym razem po złączeniu wątków z procesem wszystko działa już poprawnie.

```
[thread
start].....................................................................................
.....................................................................................
.....................................................................................
```

```
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.[thread stop]
```

Nic nie stoi na przeszkodzie aby Join() dotyczyło wielu wątków, czyli

```
using System;
using System.Threading;
class Program
{
    static void Main(string[] args)
    {
        Thread[] thread = new Thread[10];
        for (int ID = 0; ID < thread.Length; ID++)
        {
            thread[ID] = new System.Threading.Thread(work);
            thread[ID].Name = ID.ToString();
            thread[ID].Start();
        }
        for (int ID = 0; ID < thread.Length; ID++) { thread[ID].Join(); }
    }
    static void work()
    {
        Console.WriteLine("{0}", Thread.CurrentThread.Name);
```

```
    }

}
```

Zgodnie z przewidywaniami otrzymaliśmy identyfikator dla każdego z działających wątków.

```
0
2
1
3
4
5
6
7
8
9

C:\Szkola\t1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 23252) exited
with code 0.
```

Podstawowym narzędziem synchronizacyjnym C# jest obiekt

System.Threading.Monitor;

Jego ogólny schemat użycia przedstawia się jako

```
Monitor.Enter( locked );
try
{
    //...kod sekcji krytycznej
}
finally
{
    //...protokół wyjścia
    Monitor.Exit( locked );
}
```

Równoważne wobec tych wywołań jest użycie słowa kluczowego lock, jako

```
using System;
using System.Threading;
class Job
{
    public Job() { /* czyli konstruktor domyślny */ }
    public void work()
    {
```

```
        for (int i = 0; i < 5; i++)
        { Console.Write("{0}-{1} ", Thread.CurrentThread.Name, i); }
        Console.WriteLine();
    }
}
class Program
{
    static void Main(string[] args)
    {
        Thread[] thread = new Thread[10];
        Job job = new Job();
        for (int ID = 0; ID < thread.Length; ID++)
        {
            thread[ID] = new Thread(new ThreadStart(job.work));
            thread[ID].Name = ID.ToString();
        }
        foreach (Thread t in thread) { t.Start(); }
        foreach (Thread t in thread) { t.Join(); }
    }
}
```

Nasz wynik będzie chaotyczny w związku z tym że procesy zaczęły ze sobą kolidować, aby rozwiązac ten problem zastosujemy poznany przed chwilą **Monitor**, lub wprowadzając wzajemne wykluczanie.

```
0-0 4-0 2-0 2-1 2-2 2-3 2-4 0-1 0-2 0-3 0-4
4-1 4-2 4-3 4-4
9-0 9-1 9-2 9-3 9-4

5-0 5-1 5-2 5-3 8-0 8-1 8-2 8-3 8-4
3-0 3-1 3-2 3-3 5-4
7-0 7-1 7-2 7-3 7-4
1-0 1-1 1-2 1-3 1-4
3-4
6-0 6-1 6-2 6-3 6-4

C:\Szkola\t1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 4024) exited
with code 0.
```

Przykład z użyciem wykluczenia

```
using System;
using System.Threading;
class Job
{
 public Job() { /* czyli konstruktor domyślny */ }
 public void work()
 {
 lock (this)
 {
```

```
 for (int i = 0; i < 5; i++)
{ Console.Write("{0}-{1} ", Thread.CurrentThread.Name, i); }
 Console.WriteLine();
 }
 }
}
class Program
{
    static void Main(string[] args)
    {
        Thread[] thread = new Thread[10];
        Job job = new Job();
        for (int ID = 0; ID < thread.Length; ID++)
        {
            thread[ID] = new Thread( new ThreadStart( job.work ) );
            thread[ID].Name = ID.ToString();
        }
        foreach (Thread t in thread ) { t.Start(); }
        foreach (Thread t in thread) { t.Join(); }
    }
}
```

Tym razem wynik jest poprawny i zdecydowanie bardziej czytelny

```
0-0 0-1 0-2 0-3 0-4
1-0 1-1 1-2 1-3 1-4
2-0 2-1 2-2 2-3 2-4
3-0 3-1 3-2 3-3 3-4
4-0 4-1 4-2 4-3 4-4
5-0 5-1 5-2 5-3 5-4
7-0 7-1 7-2 7-3 7-4
6-0 6-1 6-2 6-3 6-4
8-0 8-1 8-2 8-3 8-4
9-0 9-1 9-2 9-3 9-4

C:\Szkola\t1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 25388) exited
with code 0.
```

Jak wspomniano wcześniej - identyczny efekt można uzyskać wprowadzając obiekt Monitor, czyli re-definiując metodą work() klasy Job public void work()

```
public void work()
{
    Monitor.Enter(this);
    try
    {
        for (int i = 0; i < 5; i++)
        {
            Console.Write("{0}-{1} ", Thread.CurrentThread.Name, i);
```

```
            }
            Console.WriteLine();
        }
        finally { Monitor.Exit(this); }
    }
```

Innym sposobem ochrony zasobów współdzielonych jest użycie obiektu

System.Threading.Mutex

Obiekt ten wyposażony jest w zestaw konstruktorów, z czego najczęściej używanym jest domyślny

public Mutex()

oraz dwie zgłaszające żądanie dostępu do zasobu współdzielonego oraz zwalniające ten zasób

public virtual bool WaitOne()

public void ReleaseMutex()

Ogólną zasadą użycia obiektu Mutex można przedstawić na przykładzie poniższego kodu.

```
using System;
using System.Threading;
class Program
{
    private static Mutex mutex = new Mutex(); //...definiujemy obiekt Mutex
    static void Main()
    {
        for (int ID = 0; ID < 5; ID++) //...tworzymy 5 wątków potomnych
        { // funkcją wątku będzie Go()
            Thread thread = new Thread(new ThreadStart(Go));
            thread.Name = ID.ToString();
            thread.Start();
        }
    }
    private static void Go()
    {
        for (int cycle = 0; cycle < 3; cycle++)
        {
            Use();
        }
    }
     private static void Use()
        {
            mutex.WaitOne(); //...czyli żądanie dostępu
            Console.WriteLine("thread {0}...use shared resource",
Thread.CurrentThread.Name);
            //...sekcja krytyczna, początek
            Thread.Sleep(100);
            //...sekcja krytyczna, koniec
            mutex.ReleaseMutex();//...zwolnienie zasobu
```

```
        }
}
```

Po wykonaniu 3 cykli zobaczymy przewidywany rezultat

```
thread 0...use shared resource
thread 4...use shared resource
thread 1...use shared resource
thread 2...use shared resource
thread 3...use shared resource
thread 0...use shared resource
thread 4...use shared resource
thread 1...use shared resource
thread 2...use shared resource
thread 3...use shared resource
thread 0...use shared resource
thread 4...use shared resource
thread 1...use shared resource
thread 2...use shared resource
thread 3...use shared resource

C:\Szkola\t1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 23768) exited
with code 0.
```

Zwróćmy tu jeszcze raz uwagę na sekwencję wykonania - przykładowo zaznaczono wątek ostatni o ID 5.

Weźmy teraz typowy przykład wyścigu w dostępie do zmiennej współdzielonej. Tutaj będzie to zmienna całkowita N, której nadamy wartość początkową 0 a każdy z wątków zwiększy ją o 1.

```
using System;
using System.Threading;
class Program
{
    private static Mutex mutex = new Mutex();
    private static int N = 0;
    static void Main()
    {
        Thread[] thread = new Thread[5];
        for (int ID = 0; ID < thread.Length; ID++)
        {
            thread[ID] = new Thread(new ThreadStart(Inc));
            thread[ID].Name = ID.ToString();
        }
        foreach (Thread t in thread) { t.Start(); }
        foreach (Thread t in thread) { t.Join(); }
        Console.WriteLine("N = {0}", N);
    }
    private static void Inc()
    {
```

```
            int n;
            //mutex.WaitOne();
            n = N;
            Console.WriteLine("thread {0} take N={1}", Thread.CurrentThread.Name, n);
            n = n + 1;
            N = n;
            Console.WriteLine("thread {0} send N={1}", Thread.CurrentThread.Name, n);
            //mutex.ReleaseMutex();
        }

    }
```

Z racji istnienia wyścigu i braku synchronizacji między pobraniem wartości N a przypisaniem własnego wyniku, efekt

```
thread 2 take N=0
thread 2 send N=1
thread 3 take N=0
thread 3 send N=1
thread 4 take N=0
thread 4 send N=1
thread 1 take N=0
thread 1 send N=1
thread 0 take N=0
thread 0 send N=1
N = 1

C:\Szkola\t1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 8396) exited
with code 0.
```

Gdyby wprowadzić ochronę sekcji krytycznej obiektem Mutex, czyli

```
private static void Inc()
{
    int n;
    mutex.WaitOne();
    n = N;
    Console.WriteLine("thread {0} take N={1}", Thread.CurrentThread.Name, n);
    n = n + 1;
    N = n;
    Console.WriteLine("thread {0} send N={1}", Thread.CurrentThread.Name, n);
    mutex.ReleaseMutex();
}
```

to uzyskamy wynik w pełni poprawny

```
```

```
thread 0 take N=0
thread 0 send N=1
thread 1 take N=1
thread 1 send N=2
thread 2 take N=2
thread 2 send N=3
thread 3 take N=3
thread 3 send N=4
thread 4 take N=4
thread 4 send N=5
N = 5

C:\Szkola\t1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 3552) exited
with code 0.
```

oraz jak można zastosować mechanizmy synchronizacji, takie jak Mutex, aby uniknąć wyścigów o zasoby współdzielone.

Wnioski:

1. **Tworzenie Wątków:** Utworzenie wątków w języku C# jest możliwe za pomocą klasy Thread. Każdy wątek może być inicjowany przez konstruktor klasy Thread, który przyjmuje funkcję, którą ma wykonywać.

2. **Synchronizacja z Mutex:** Do zabezpieczenia dostępu do zasobów współdzielonych między wątkami użyto obiektu Mutex. Mechanizm ten pozwala na blokadę dostępu do sekcji krytycznej, co eliminuje wyścigi o zasoby.

3. **Join i Oczekiwanie na Zakończenie Wątków:** Mechanizm Join umożliwia oczekiwanie na zakończenie działania danego wątku. To jest ważne, aby upewnić się, że wszystkie wątki zakończyły swoje zadania przed zakończeniem programu głównego.

4. **Wpływ Wyścigów na Wyniki:** Brak synchronizacji może prowadzić do wyścigów o zasoby, co skutkuje nieprzewidywalnymi wynikami. Wprowadzenie mechanizmów synchronizacji, takich jak Mutex, jest kluczowe dla zapewnienia poprawności i spójności działania programu wielowątkowego.

5. **Monitor jako Mechanizm Synchronizacji:** Oprócz Mutex, do synchronizacji dostępu do sekcji krytycznych można użyć klasy Monitor. Monitor dostarcza prosty sposób na oznaczanie krytycznych fragmentów kodu, które wymagają ekskluzywnego dostępu.