Паралелно програмиране

Упражнение 2 Нишки

Стартиране и приспиване на нишки

```
1 usage
public void DoTask1()
{
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine("Thread1:job({0})", i);
        Thread.Sleep( millisecondsTimeout: 1);
    }
}
```

```
1 usage
public void DoTask2()
{
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine("Thread2:job({0})", i);
        Thread.Sleep( millisecondsTimeout: 1);
    }
}

static void Main(string[] args)
{
    ThreadClass threadClass = new ThreadClass();

    Thread thread1 = new Thread(new ThreadStart(threadClass.DoTask1));
    Thread thread2 = new Thread(new ThreadStart(threadClass.DoTask2));

    thread1.Start();
    thread2.Start();
}
```

Thread(ThreadStart)

Създава инстанция. Подава се делегат с метод, който да се изпълни при стартиране.

Sleep(…)

“Приспива” текущата нишка за указания брой милисекунди. Методът е статичен и блокира текущото изпълняваната нишка. След изтичането на зададения интервал, тя продължава работата си.

Suspend()

Ако нишката работи, я преустановява временно. Ако е преустановена, не се случва нищо. За разлика от Sleep(), чрез който нишка преустановява себе си за някакъв фиксиран интервал от време, Suspend() преустановява нишка за неопределено време и тя остава в това състояние до извикването на Resume(), който подновява изпълнението ѝ.

Resume()

Подновява нишка, която е била преустановена (suspended). Ако нишката работи, не прави нищо.

Стартиране и изчакване на нишка

```
static void Main(string[] args)
{
    Console.WriteLine("Main thread started.");
    ThreadClass threadClass = new ThreadClass();
    Thread thread1 = new Thread(new ThreadStart(threadClass.DoTask1));
    Thread thread2 = new Thread(new ThreadStart(threadClass.DoTask2));

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

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

    Console.WriteLine("Main thread finished.");
}
```

Join()

Извикващата нишка изчаква, докато извиканата приключи. Може да се зададе и таймаут.

Abort()

Хвърля ThreadAbortException в извиканата нишка, с което обикновено прекратява нишката. При определени условия, Abort() може и да не прекрати нишката.

Interrupt()

Ако нишката е в състояние WaitSleepJoin, хвърля ThreadInterruptedException. Нишката може да прихване това изключение и да продължи изпълнението си. Ако тя не го прихване, CLR го прихваща и прекратява нишката. Ако нишката не е в състояние WaitSleepJoin, извикването на Interrupt() не прави нищо.

Приоритет на нишките

```
thread1.Priority = ThreadPriority.
thread1.Start();
thread2.Start();

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

Console.WriteLine("Main thread
}
```

Highest ThreadPriority
Lowest ThreadPriority
Normal ThreadPriority
AboveNormal ThreadPriority
BelowNormal ThreadPriority
TryParse bool
Press ⏎ to insert, ⇥ to replace Next Tip

Проблеми при работа с общи данни

Задача: Имаме клас банкова сметка (Account), с член променлива mBalance - текущият баланс по сметката и член метод Withdraw100(), който намалява баланса с 100. Проблемът настъпва, когато две нишки теглят едновременно пари от тази банкова сметка, остатъкът в нея остава некоректен. Първоначално да се реализира проблема и след това да се коментира как може да се подобри, така че балансът да бъде коректен в сметката.

Синхронизиращи конструкции

Тук се синхронизират само отделни участъци от кода - тези, които са рискови. Критична секция наричаме участък от кода, до който не трябва да бъде допускан едновременен достъп. За гарантиране безопасен достъп до критична секция може да използваме ключовата дума lock или класа Monitor.

Вариант 1

```
lock (obj)
{
    // code
}
```

Вариант 2

```
Monitor.Enter(obj);
try
{
    // code
}
finally
{
    Monitor.Exit(obj);
}
```

Вариант 3

Класът Mutex е наследник на WaitHandle и представлява “мутекс” - примитив за синхронизация на операционната система. Той наподобява Monitor, но не е обвързан с обект.

Обектът obj трябва да бъде от референтен тип (ако не е, се извършва опаковане, което ще доведе до безрезултатно заключване на различен новосъздаден обект при всяко влизане в секцията.

Бонус пример - Singleton

```
public sealed class Singleton
{
    private static int counter = 0;
    private static Singleton instance = null;

    2 usages
    public static Singleton GetInstance
    {
        get
        {
            if(instance == null)
                instance = new Singleton();
            return instance;
        }
    }

    1 usage
    private Singleton()
    {
        counter++;
        Console.WriteLine("Counter Value " + counter.ToString());
    }

    2 usages
    public void PrintDetails(string message)
    {
        Console.WriteLine(message);
    }
}
```

```
class Program2
{
    static void Main(string[] args)
    {
        // Invoke multiple methods parallelly
        Parallel.Invoke(
            params actions: () => PrintTeacherDetails(),
            () => PrintStudentDetails()
            );
        Console.ReadLine();
    }

    1 usage
    private static void PrintTeacherDetails()
    {
        Singleton fromTeacher = Singleton.GetInstance;
        fromTeacher.PrintDetails( message: "From Teacher");
    }

    1 usage
    private static void PrintStudentDetails()
    {
        Singleton fromStudent = Singleton.GetInstance;
        fromStudent.PrintDetails( message: "From Student");
    }
```

Резултат

```
Counter Value 2
From Student
Counter Value 1
From Teacher
```

Singleton - Thread Safe

```
public sealed class Singleton
{
    private static int counter = 0;
    private static readonly object Instancelock = new object();

    1 usage
    private Singleton()
    {
        counter++;
        Console.WriteLine("Counter value " + counter.ToString());
    }

    private static Singleton instance = null;

    2 usages
    public static Singleton GetInstance
    {
        get
        {
            lock (Instancelock)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }

                return instance;
            }
        }
    }

    2 usages
    public void PrintDetails(string message)
    {
        Console.WriteLine(message);
    }
}
```

```
class Program3
{
    static void Main(string[] args)
    {
        Parallel.Invoke(
            params actions: () => PrintTeacherDetails(),
            () => PrintStudentDetails()
            );
        Console.ReadLine();
    }

    1 usage
    private static void PrintTeacherDetails()
    {
        Singleton fromTeacher = Singleton.GetInstance;
        fromTeacher.PrintDetails( message: "From Teacher");
    }

    1 usage
    private static void PrintStudentDetails()
    {
        Singleton fromStudent = Singleton.GetInstance;
        fromStudent.PrintDetails( message: "From Student");
    }
}
```

Резултат

```
Counter value 1
From Student
From Teacher
```

Задачи

Задача1: Решете проблема за “обядващите философи” чрез подходящи синхонизационни механизми.

В тази задача, няколко философа стоят около кръгла маса и всеки от тях извършва само 2 действия - храни се или мисли. За да започне даден философ да се храни, той се нуждае едновременно от двете вилици, които стоят вляво и вдясно от чинията му. Ако един философ вземе едната вилица, но не може да вземе в този момент и другата (защото тя е заета), той не може да започне да се храни докато не се сдобие и с нея. Има риск всеки философ да хване една от вилиците в даден момент и да чака безкрайно за другата. Това ще доведе до “мъртва хватка”. Задачата е да се измисли алгоритъм за хранене на философите, при който не се получават “мъртви хватки”.

Задача2: Решете проблема за “четци и писачи”. В тази задача имам един или повече “писачи”, които искат да пишат върху даден общ ресурс, например файл. Успоредно на тях, един или повече “четци” четат от същият ресурс. За да е коректен достъпът до общият ресурс, необходимо да се спазени следващите условия:

- **Пройзволен брой четци могат да имат едновременнен достъп до ресурса - това няма как да породи синхронизационни проблеми, защото в този момент ресурсът не се променя.**
- **Ако на писач е продставен достъп до ресурса, достъпът на всички останали трябва да бъде забранен - независимо дали четци или писачи.**
- **Нито един четец или писач не трябва да чака безкрайно дълго. Упътване на следващият слайд.**

Упътване “четци-писачи”

```
using System.Threading;

namespace ParallelPrograming
{
    public class Resource
    {
        ReaderWriterLock rwLock = new ReaderWriterLock();

        public void Read()
        {
            rwLock.AcquireReaderLock(Timeout.Infinite);
            try
            {

            }
            finally
            {
                rwLock.ReleaseReaderLock();
            }
        }

        public void Write()
        {
            rwLock.AcquireWriterLock(Timeout.Infinite);
            try
            {

            }
            finally
            {
                rwLock.ReleaseWriterLock();
            }
        }
    }
}
```

Използвайте класът ReaderWriterLock.

Критичният ресурс се заключва с методите: AcquireReaderLock и AcquireWriterLock, съответно за четец и писач.

Освобождаването става с ReleaseReaderLock() и ReleaseWriterLock().

Свойствата IsReaderLockHeld и IsWriterLockHeld информират дали ресурсът е текущо заключен от четец или писач.