Министерство науки и высшего образования

Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Российский государственный университет им. А.Н. Косыгина

(Технологии.Дизайн.Искусство)»

Кафедра автоматизированных систем обработки информации и управления

Лабораторная работа №5

Выполнил: Букша Кирилл Владимирович

Группа: МАГ-В-221

Вариант 4

Проверил: Кузьмина Тамара Михайловна

Москва 2021

Задание:

Несколько потоков работают с общим одноэлементным буфером. Потоки делятся на "писателей", осуществляющих запись сообщений в буфер, и "читателей", осуществляющих извлечение сообщений из буфера. Только один поток может осуществлять работу с буфером. Если буфер свободен, то только один писатель может осуществлять запись в буфер. Если буфер занят, то только один читатель может осуществлять чтение из буфера. После чтения буфер освобождается и доступен для записи. В качестве буфера используется глобальная переменная. Работа приложения заканчивается после того, как все сообщения писателей через общий буфер будут обработаны читателями.

1) Реализуйте взаимодействие потоков-читателей и потоков-писателей с общим буфером без каких-либо средств синхронизации
2) Реализуйте доступ "читателей" и "писателей" к буферу с применением следующих средств синхронизации:
    a. блокировки (lock, Monitor)
    b. сигнальные сообщения (ManualResetEvent, AutoResetEvent, ManualResetEventSlim)
    c. класс Mutex
3) Исследуйте производительность средств синхронизации, для этого оцените время работы
4) Сделайте выводы об эффективности применения средств синхронизации.

Вариант задает число писателей и читателей. Первое число – количество писателей, второе число количество читателей.

Вариант 4: 14,15.

Решение:

Были реализованы 4 класса, каждый из которых реализует определенный механизм работы буфера. Простой подход, не учитывающий гонку между процессами:

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Lab5_mag
{
    class SimpleBuffer
    {
        private bool _finish = false;
        private bool _bEmpty = true;
        private int _buffer;

        private readonly List<Task> _writers = new();
        private readonly List<Task<int>> _readers = new();
```

```
        private readonly int _writersCount;
        private readonly int _readersCount;
        private readonly int _messagesCount;

        public SimpleBuffer(int writersCount, int readersCount, int n)
        {
            _writersCount = writersCount;
            _readersCount = readersCount;
            _messagesCount = n;
        }

        public async Task<IEnumerable<int>> DoWork()
        {
            for (var i = 0; i < _writersCount; i++)
            {
                _writers.Add(Task.Run(() => WriterJob(i)));
            }
            for (var i = 0; i < _readersCount; i++)
            {
                _readers.Add(Task.Run(() => ReaderJob()));
            }

            await Task.WhenAll(_writers);
            _finish = true;
            await Task.WhenAll(_readers);
            var readedMessages = _readers.Select(task => task.Result);
            return readedMessages;
        }

        private int ReaderJob()
        {
            var myMessages = new List<int>();

            while (!_finish)
            {
                if (!_bEmpty)
                {
                    myMessages.Add(_buffer);
                    _bEmpty = true;
                }
            }

            return myMessages.Count;
        }

        private void WriterJob(int index)
        {
            var myMessages = new List<int>();
            for (var message = _messagesCount * index; message < _messagesCount * (index
+ 1); message++)
            {
                myMessages.Add(message);
            }

            var i = 0;
            while (i < _messagesCount)
            {
                if (_bEmpty)
                {
                    _buffer = myMessages[i++];
                    _bEmpty = false;
                }
            }
        }
```

```
    }
}
```

Реализация, использующая Monitor:

```
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace Lab5_mag
{
    class MonitorBuffer
    {
        private bool _finish = false;
        private bool _bEmpty = true;
        private int _buffer;

        private readonly object _locker = new object();

        private readonly List<Task> _writers = new();
        private readonly List<Task<int>> _readers = new();

        private readonly int _writersCount;
        private readonly int _readersCount;
        private readonly int _messagesCount;

        public MonitorBuffer(int writersCount, int readersCount, int n)
        {
            _writersCount = writersCount;
            _readersCount = readersCount;
            _messagesCount = n;
        }

        public async Task<IEnumerable<int>> DoWork()
        {
            for (var i = 0; i < _writersCount; i++)
            {
                _writers.Add(Task.Run(() => WriterJob(i)));
            }
            for (var i = 0; i < _readersCount; i++)
            {
                _readers.Add(Task.Run(() => ReaderJob()));
            }

            await Task.WhenAll(_writers);
            _finish = true;
            await Task.WhenAll(_readers);
            var readedMessages = _readers.Select(task => task.Result);
            return readedMessages;
        }

        private int ReaderJob()
        {
            var myMessages = new List<int>();

            while (!_finish)
            {
                if (!_bEmpty)
                {
                    Monitor.Enter(_locker);
                    try
                    {
                        if (!_bEmpty)
```

```
                        {
                            myMessages.Add(_buffer);
                            _bEmpty = true;
                        }
                    }
                    finally
                    {
                        Monitor.Exit(_locker);
                    }
                }
            }

            return myMessages.Count;
        }

        private void WriterJob(int index)
        {
            var myMessages = new List<int>();
            for (var message = _messagesCount * index; message < _messagesCount * (index
+ 1); message++)
            {
                myMessages.Add(message);
            }

            var i = 0;
            while (i < _messagesCount)
            {
                Monitor.Enter(_locker);
                try
                {
                    if (_bEmpty)
                    {
                        _buffer = myMessages[i++];
                        _bEmpty = false;
                    }
                }
                finally
                {
                    Monitor.Exit(_locker);
                }
            }
        }
    }
}
```

Класс, использующий событийное программирование:

```
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace Lab5_mag
{
    class EventBuffer
    {
        private bool _finish = false;
        private int _buffer;

        private static AutoResetEvent eventReadyToRead = new AutoResetEvent(false);
        private static AutoResetEvent eventReadyToWrite = new AutoResetEvent(true);

        private readonly List<Task> _writers = new();
        private readonly List<Task<int>> _readers = new();
```

```
        private readonly int _writersCount;
        private readonly int _readersCount;
        private readonly int _messagesCount;

        public EventBuffer(int writersCount, int readersCount, int n)
        {
            _writersCount = writersCount;
            _readersCount = readersCount;
            _messagesCount = n;
        }

        public async Task<IEnumerable<int>> DoWork()
        {
            for (var i = 0; i < _writersCount; i++)
            {
                _writers.Add(Task.Run(() => WriterJob(i)));
            }
            for (var i = 0; i < _readersCount; i++)
            {
                _readers.Add(Task.Run(() => ReaderJob()));
            }

            await Task.WhenAll(_writers);
            _finish = true;
            for (var i = 0; i < _readersCount; i++)
                eventReadyToRead.Set();
            await Task.WhenAll(_readers);
            var readedMessages = _readers.Select(task => task.Result);
            return readedMessages;
        }

        private int ReaderJob()
        {
            var myMessages = new List<int>();

            while (true)
            {
                eventReadyToRead.WaitOne();
                if (_finish)
                    break;
                myMessages.Add(_buffer);
                eventReadyToWrite.Set();
            }

            return myMessages.Count;
        }

        private void WriterJob(int index)
        {
            var myMessages = new List<int>();
            for (var message = _messagesCount * index; message < _messagesCount * (index
+ 1); message++)
            {
                myMessages.Add(message);
            }

            var i = 0;
            while (i < _messagesCount)
            {
                eventReadyToWrite.WaitOne();
                _buffer = myMessages[i++];
                eventReadyToRead.Set();

            }
```

```
        }
    }

}
```

Класс, использующий Mutex:

```
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace Lab5_mag
{
    class MutexBuffer
    {
        private bool _finish = false;
        private bool _bEmpty = true;
        private int _buffer;

        private readonly Mutex _mutex = new Mutex();

        private readonly List<Task> _writers = new();
        private readonly List<Task<int>> _readers = new();

        private readonly int _writersCount;
        private readonly int _readersCount;
        private readonly int _messagesCount;

        public MutexBuffer(int writersCount, int readersCount, int n)
        {
            _writersCount = writersCount;
            _readersCount = readersCount;
            _messagesCount = n;
        }

        public async Task<IEnumerable<int>> DoWork()
        {
            for (var i = 0; i < _writersCount; i++)
            {
                _writers.Add(Task.Run(() => WriterJob(i)));
            }
            for (var i = 0; i < _readersCount; i++)
            {
                _readers.Add(Task.Run(() => ReaderJob()));
            }

            await Task.WhenAll(_writers);
            _finish = true;
            await Task.WhenAll(_readers);
            var readedMessages = _readers.Select(task => task.Result);
            return readedMessages;
        }

        private int ReaderJob()
        {
            var myMessages = new List<int>();

            while (!_finish)
            {
                if (!_bEmpty)
                {
                    _mutex.WaitOne();
                    try
```

```
                    {
                        if (!_bEmpty)
                        {
                            myMessages.Add(_buffer);
                            _bEmpty = true;
                        }
                    }
                    finally
                    {
                        _mutex.ReleaseMutex();
                    }
                }
            }

            return myMessages.Count;
        }

        private void WriterJob(int index)
        {
            var myMessages = new List<int>();
            for (var message = _messagesCount * index; message < _messagesCount * (index
+ 1); message++)
            {
                myMessages.Add(message);
            }

            var i = 0;
            while (i < _messagesCount)
            {
                _mutex.WaitOne();
                try
                {
                    if (_bEmpty)
                    {
                        _buffer = myMessages[i++];
                        _bEmpty = false;
                    }
                }
                finally
                {
                    _mutex.ReleaseMutex();
                }
            }
        }
    }
}
```

Далее был написан код функции Main, в котором происходит последовательный вызов всех описанных ранее методов и сравнение результатов работы:

```
static async Task Main(string[] args)
        {
            const int writersCount = 14;
            const int readersCount = 15;

            Console.WriteLine("Enter N:");
            var n = Convert.ToInt32(Console.ReadLine());

            Console.WriteLine("Simple buffer.");

            var simpleBuffer = new SimpleBuffer(writersCount, readersCount, n);
            var watch = Stopwatch.StartNew();
```

```
            var results = (await simpleBuffer.DoWork()).ToList();
            watch.Stop();
            Console.WriteLine($"Simple buffer, time of execution
{watch.ElapsedMilliseconds} ms");
            Console.WriteLine($"Readed messages by each reader: {string.Join(" ",
results)}. Sum: {results.Sum()}");
            Console.WriteLine();
            Console.WriteLine();

            var mutexBuffer = new MutexBuffer(writersCount, readersCount, n);
            watch = Stopwatch.StartNew();
            results = (await mutexBuffer.DoWork()).ToList();
            watch.Stop();
            Console.WriteLine($"Mutex buffer, time of execution
{watch.ElapsedMilliseconds} ms");
            Console.WriteLine($"Readed messages by each reader: {string.Join(" ",
results)}. Sum: {results.Sum()}");
            Console.WriteLine();
            Console.WriteLine();

            var monitorBuffer = new MonitorBuffer(writersCount, readersCount, n);
            watch = Stopwatch.StartNew();
            results = (await monitorBuffer.DoWork()).ToList();
            watch.Stop();
            Console.WriteLine($"Monitor buffer, time of execution
{watch.ElapsedMilliseconds} ms");
            Console.WriteLine($"Readed messages by each reader: {string.Join(" ",
results)}. Sum: {results.Sum()}");
            Console.WriteLine();
            Console.WriteLine();

            var eventBuffer = new EventBuffer(writersCount, readersCount, n);
            watch = Stopwatch.StartNew();
            results = (await eventBuffer.DoWork()).ToList();
            watch.Stop();
            Console.WriteLine($"Event buffer, time of execution
{watch.ElapsedMilliseconds} ms");
            Console.WriteLine($"Readed messages by each reader: {string.Join(" ",
results)}. Sum: {results.Sum()}");
            Console.WriteLine();
            Console.WriteLine();
        }
```

После запуска программы при использовании первоначального количества читателей и писателей было замечено падение быстродействия. Время исполнения заняло порядка 80 секунд, большая часть времени была затрачена «простой» реализацией:

```
Enter N:
10
Simple buffer.
Simple buffer, time of execution 80511 ms
Readed messages by each reader: 90 0 0 0 0 0 0 0 0 0 0 0 0 0 0. Sum: 90


Mutex buffer, time of execution 155 ms
Readed messages by each reader: 30 16 7 3 10 13 5 11 4 1 1 0 0 0 39. Sum: 140


Monitor buffer, time of execution 56 ms
Readed messages by each reader: 6 12 12 2 16 7 7 7 13 6 2 0 1 0 49. Sum: 140
```

```
Event buffer, time of execution 11 ms
Readed messages by each reader: 64 1 2 1 1 1 1 0 0 0 0 0 0 0 69. Sum: 140
```

При снижении количества читателей и писателей до 4 и 5 соответственно, появляется возможность запустить программу на больших исходных данных и получить следующие результаты:

```
Enter N:
10000
Simple buffer.
Simple buffer, time of execution 63 ms
Readed messages by each reader: 12600 13172 2153 1759 872. Sum: 30556


Mutex buffer, time of execution 1165 ms
Readed messages by each reader: 9367 9985 9953 738 9957. Sum: 40000


Monitor buffer, time of execution 112 ms
Readed messages by each reader: 10607 9940 9151 686 9616. Sum: 40000


Event buffer, time of execution 557 ms
Readed messages by each reader: 10002 10005 9989 0 10004. Sum: 40000
```

Исходя из результатов можно сделать вывод, что простая реализация не может быть применена на практике, т.к. результаты ее работы не предсказуемы и не корректны. Реализация, использующая Monitor, показала наибольшее быстродействие, Mutex – наименьшее. Данные результаты совпадают с результатами, полученными в предыдущей лабораторной работе. Также можно заметить, что в некоторых случаях один из читательских потоков был практически не задействован. Это связано с архитектурными особенностями платформы .Net, операционной системой Windows, а также возможностями компьютера, на котором происходил запуск кода.

Вывод: была написана программа, использующая многопоточный подход к программированию, а также классы Monitor и Mutex. Программа позволяет сравнить различные подходы к решению задачи и сделать выводы. Программа отлажена и протестирована ручными методами тестирования. Исходный код программы залит на Github и доступен по ссылке: https://github.com/bukSHA1024/RSU_TRPO_Lab5