**Отчет**

**Задание №1:**

реализуйте взаимодействие потоков-читателей и потоков-писателей с общим буфером без каких-либо средств синхронизации. Проиллюстрируйте проблему совместного доступа. Почему возникает проблема доступа?

результат моделирования:

Писатель №1: BBBB - 1

Писатель №0: AAAA - 1

Читатель №0: AAAA - 1

Писатель №1: BBBB - 2

Писатель №2: CCCC - 1

Читатель №0: CCCC - 1

Писатель №1: BBBB - 3

Читатель №0: CCCC - 2

Писатель №1: BBBB - 4

Читатель №0: BBBB - 4

Писатель №1: BBBB - 5

Читатель №0: BBBB - 5

Читатель №1: BBBB - 4

Писатель №2: CCCC - 2

Читатель №1: BBBB - 5

Читатель №0: BBBB - 5

Писатель №2: CCCC - 3

Читатель №1: CCCC - 3

Читатель №0: CCCC - 3

Писатель №2: CCCC - 4

Читатель №0: CCCC - 4

Писатель №2: CCCC - 5

Читатель №0: CCCC - 5

Читатель №2: CCCC - 4

Читатель №1: CCCC - 4

Писатель №0: BBBB - 2

Читатель №1: CCCC - 5

Читатель №0: CCCC - 5

Писатель №0: AAAA - 3

Читатель №0: AAAA - 3

Писатель №0: AAAA - 4

Читатель №0: AAAA - 4

Писатель №0: AAAA - 5

Читатель №1: AAAA - 3

Читатель №0: AAAA - 5

Читатель №2: CCCC - 5

Читатель №1 результаты захвата данных:

AAAA - 3

BBBB - 4

BBBB - 5

CCCC - 3

CCCC - 4

CCCC - 5

Читатель №0 результаты захвата данных:

AAAA - 1

AAAA - 3

AAAA - 4

AAAA - 5

BBBB - 4

BBBB - 5

BBBB - 5

CCCC - 1

CCCC - 2

CCCC - 3

CCCC - 4

CCCC - 5

CCCC - 5

Читатель №2 результаты захвата данных:

CCCC - 4

CCCC – 5

**Задание №3:**

Исследуйте производительность средств синхронизации при разном числе сообщений, разном объеме сообщений, разном числе потоков.

Результаты моделирования:

Lock: число прогонов - 10, число писателей - 3, число читателей - 3, среднее время выполнение - 353,4264 мс

ResetEvent: число прогонов - 10, число писателей - 3, число читателей - 3, среднее время выполнение - 54,0078 мс

Semaphore: число прогонов - 10, число писателей - 3, число читателей - 3, среднее время выполнение - 57,2731 мс

Interlock: число прогонов - 10, число писателей - 3, число читателей - 3, среднее время выполнение - 165,8198 мс

Lock: число прогонов - 10, число писателей - 3, число читателей - 2, среднее время выполнение - 283,9538 мс

ResetEvent: число прогонов - 10, число писателей - 3, число читателей - 2, среднее время выполнение - 41,3354 мс

Semaphore: число прогонов - 10, число писателей - 3, число читателей - 2, среднее время выполнение - 51,5047 мс

Interlock: число прогонов - 10, число писателей - 3, число читателей - 2, среднее время выполнение - 128,2853 мс

Lock: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 293,6621 мс

ResetEvent: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 49,4379 мс

Semaphore: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 47,1960 мс

Interlock: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 806,5804 мс

Lock: число прогонов - 20, число писателей - 2, число читателей - 2, среднее время выполнение - 106,0697 мс

ResetEvent: число прогонов - 20, число писателей - 2, число читателей - 2, среднее время выполнение - 25,0912 мс

Semaphore: число прогонов - 20, число писателей - 2, число читателей - 2, среднее время выполнение - 25,1642 мс

Interlock: число прогонов - 20, число писателей - 2, число читателей - 2, среднее время выполнение - 42,4289 мс

**Задание №4:**

Сделайте выводы об эффективности применения средств синхронизации.

В рамках рассматриваемого задания с разными набатами данных самыми эффективными средствами по обеспечению синхронизированного доступа были ResetEvent (в частности AutoResetEvent) и Semaphore результаты испытаний были примерно одинаковые. В данной задачи эффективность применения атомарных средстве синхронизации сильно зависит от общего числа потоков выполнения, а также количества физических ядер процессора. Максимальная эффективность в данном примере проявляется при 4 потоках выполнения. При большом числе потоков выполнения на времени выполнения программы начинает сказываться конкуренция за ресурсы процессора.

**Вопросы и упражнения**

**Вопрос №1:** почему проблема гонки данных проявляется не при каждом прогоне?

Это связано с тем, как планировщик задач ОС снимает и устанавливает потоки на выполнения. Теоретически для конкретного случая гонки за данными можно избежать если воспользоваться методом Thread.Sleep(time) если правильно подгадать и установить время приостановки потоков выполнения.

**Вопрос №2:** какие факторы увеличивают вероятность проявления проблемы гонки данных?

В данном задании на гонку за данными могут повлиять, общее число потоков выполнения, количество физических ядер процессора -> порождает гонку за вычислительный ресурс. Размер кода и времени его выполнения в критической секции (общие данные).

**Вопрос №3:** возможно ли в данной задаче при отсутствии средств синхронизации возникновение исключения и аварийное завершение программы?

Нет. Только исключения связанные с самими потоками выполнения.

**Вопрос №4:** можно ли в данной задаче использовать атомарные операторы для обеспечения согласованности доступа? Необходимы ли при этом дополнительные средства синхронизации?

В рамках данной задачи можно использовать атомарные операторы, дополнительные средства синхронизации использовать не требуется. В дополнении к атомарным используются только конструкции if (…) {…} и булевыми флагами. Ниже приведен значимые части исполняемого кода:

…

private Int32 m\_iFull;

private Int32 m\_iEmpty;

private bool m\_bwFinish; // статус завершения записи данных в буфер

…

void WriteInterlock(object obj)

{

TestData dTemp = (TestData)obj;

int cnt = 0;

while (cnt < dTemp.RecordsCount)

{

if (0 < m\_iEmpty)

{

if (0 < Interlocked.CompareExchange(ref m\_iEmpty, 0, 1))

{

m\_stringBuffer = string.Format("{0} - {1}", dTemp.StringData, cnt + 1);

Console.WriteLine(string.Format("{0}:\t{1}\n", Thread.CurrentThread.Name, m\_stringBuffer));

cnt++;

Interlocked.Increment(ref m\_iFull);

}

}

}

}

…

void ReadInterlock()

{

List<string> readerBuffer = new List<string>();

while (!m\_bwFinish)

{

if (0 < m\_iFull)

{

if (0 < Interlocked.CompareExchange(ref m\_iFull, 0, 1))

{

readerBuffer.Add(m\_stringBuffer);

Console.WriteLine(string.Format("\t{0}:\t{1}\n", Thread.CurrentThread.Name, m\_stringBuffer));

Interlocked.Increment(ref m\_iEmpty);

}

}

}

if (m\_brPrint)

{

// вывод информации читателей:

lock (m\_printLock)

{

readerBuffer.Sort();

Console.WriteLine(Thread.CurrentThread.Name + " результаты захвата данных:");

readerBuffer.ForEach((string s) => { Console.WriteLine(s); });

Console.WriteLine();

}

}

}

**Вопрос №5:** можно ли в данной задаче использовать потокобезопасные коллекции для обеспечения согласованного доступа?

Для данной задачи использование потокобезопастных коллекций применимо, в качестве примера использовался потокобезопастная очередь ConcurrentQueue. Ниже приведены значимые участки кода задания:

…

private static ConcurrentQueue<string> m\_queue;

…

void WriteConcurrent(object obj)

{

TestData dTemp = (TestData)obj;

int cnt = 0;

while (cnt < dTemp.RecordsCount)

{

// if (m\_queue.Count == 0)

// {

string sTemp = string.Format("{0} - {1}", dTemp.StringData, cnt + 1);

m\_queue.Enqueue(sTemp);

Console.WriteLine(string.Format("{0}:\t{1}\n", Thread.CurrentThread.Name, sTemp));

cnt++;

// }

}

}

…

void ReadConcurrent()

{

List<string> readerBuffer = new List<string>();

string sTemp = string.Empty;

while (!m\_bwFinish)

{

if (m\_queue.TryDequeue(out sTemp))

{

readerBuffer.Add(sTemp);

Console.WriteLine(string.Format("\t{0}:\t{1}\n", Thread.CurrentThread.Name, sTemp));

}

}

if (m\_brPrint)

{

// вывод информации читателей:

lock (m\_printLock)

{

readerBuffer.Sort();

Console.WriteLine(Thread.CurrentThread.Name + " результаты захвата данных:");

readerBuffer.ForEach((string s) => { Console.WriteLine(s); });

Console.WriteLine();

}

}

}

Результаты выполнения:

Писатель №0: AAAA - 1

Писатель №0: AAAA - 2

Писатель №0: AAAA - 3

Писатель №0: AAAA - 4

Писатель №0: AAAA - 5

Писатель №0: AAAA - 6

Писатель №1: BBBB - 1

Писатель №2: CCCC - 1

Писатель №2: CCCC - 2

Писатель №2: CCCC - 3

Писатель №2: CCCC - 4

Писатель №2: CCCC - 5

Писатель №2: CCCC - 6

Писатель №1: BBBB - 2

Писатель №1: BBBB - 3

Писатель №1: BBBB - 4

Писатель №1: BBBB - 5

Писатель №1: BBBB - 6

Читатель №0: AAAA - 1

Читатель №0: AAAA - 2

Читатель №0: BBBB - 1

Читатель №0: AAAA - 4

Писатель №3: DDDD - 1

Писатель №3: DDDD - 2

Писатель №3: DDDD - 3

Писатель №3: DDDD - 4

Писатель №3: DDDD - 5

Читатель №1: AAAA - 3

Читатель №1: AAAA - 6

Читатель №0: AAAA - 5

Читатель №0: BBBB - 2

Читатель №0: CCCC - 2

Читатель №0: CCCC - 3

Читатель №0: CCCC - 4

Писатель №3: DDDD - 6

Читатель №1: CCCC - 1

Читатель №1: CCCC - 5

Читатель №1 результаты захвата данных:

AAAA - 3

Читатель №0: DDDD - 1

AAAA - 6

CCCC - 1

CCCC - 5

Читатель №0 результаты захвата данных:

AAAA - 1

AAAA - 2

AAAA - 4

AAAA - 5

BBBB - 1

BBBB - 2

CCCC - 2

CCCC - 3

CCCC - 4

DDDD – 1

Concurrent: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 69,1607 мс

**Вопрос №6:** какие средства синхронизации обеспечивают наилучшее быстродействие в данной задаче? Объясните с чем это связано.

Результаты выполнения:

Lock: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 303,6999 мс

ResetEvent: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 52,5461 мс

Semaphore: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 51,8975 мс

Interlock: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 830,6574 мс

Concurrent: число прогонов - 10, число писателей - 4, число читателей - 2, среднее время выполнение - 69,1607 мс

Самыми эффективными способами синхронизации в данной задачи оказались: ResetEvent (AutoResetEvent), Semaphore – так как они пробуждают один из ожидающих потоков выполнения в очереди, и не порождают конкуренцию за вычислительный ресурс. Также достаточно эффективным оказалось применение объекта ConcurrentQueue – очередь из стандартной библиотеки классов.