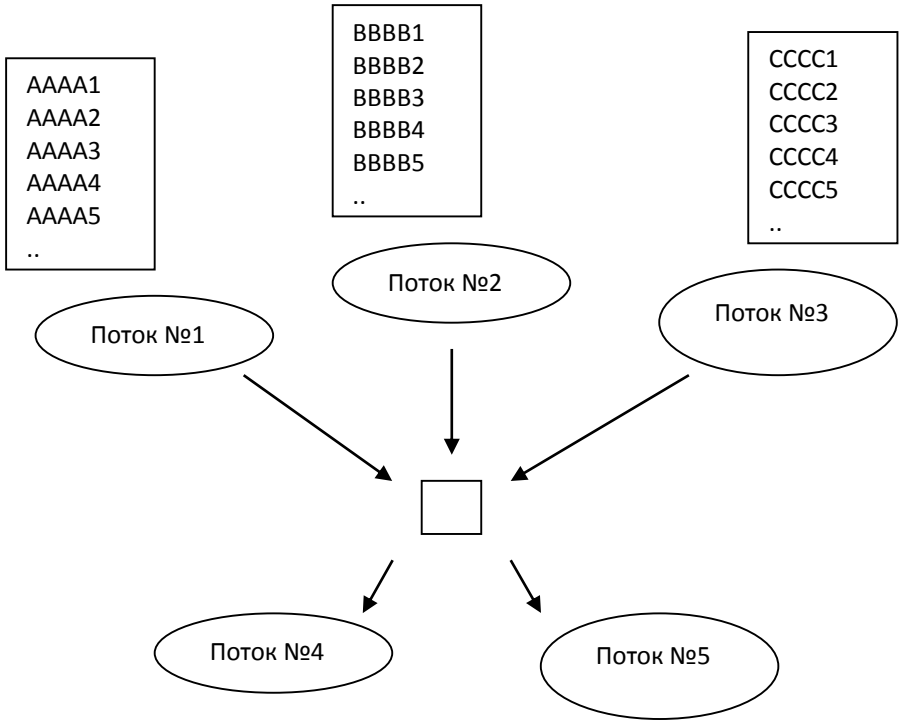
Лабораторная работа №3.
Синхронизация доступа к одноэлементному буферу

Задача

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 (заполнен), то только один читатель может осуществлять чтение из буфера. После чтения буфер освобождается и доступен для записи. В качестве буфера используется глобальная переменная, например, типа string. Работа приложения заканчивается после того, как все сообщения писателей через общий буфер будут обработаны читателями.



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?
2. Реализуйте доступ «читателей» и «писателей» к буферу с применением следующих средств синхронизации:

- блокировки (`lock`);
- сигнальные сообщения (`AutoResetEvent, ManualResetEventSlim`);
- семафоры (`Semaphore, SemaphoreSlim`);
- атомарные операторы (`Interlocked`).

3. Исследуйте производительность средств синхронизации при разном числе сообщений, при разной длине каждого сообщения, при разном числе потоков.
4. Сделайте выводы об эффективности применения средств синхронизации.

Методические указания

В случае одноэлементного буфера достаточно использовать флаг типа `bool` для контроля состояния буфера. Читатели обращаются к буферу, только если он свободен:

```
// Работа читателя
while (!finish)
{
     if (!bEmpty)
     {
          MyMessages.Add(buffer);
          bEmpty = true;
     }

}
```

Писатели обращаются к буферу, только если он пуст:

```
// Работа писателя
while(i < n)
{
     if (bEmpty)
     {
          buffer = MyMessages[i++];
          bEmpty = false;
     }
}
```

Писатели работают, пока не запишут все свои сообщения. По окончании работы писателей основной поток может изменить статус переменной `finish`, который является признаком окончания работы читателей.

```
static void Main()
{
     // Запускаем читателей и писателей
     ..
```

```
        // Ожидаем завершения работы писателей
        for(int i=0; i< writers.Length; i++)
             writers[i].Join();
        // Сигнал о завершении работы для читателей
        finish = true;

        // Ожидаем завершения работы читателей
        for(int i=0; i< readers.Length; i++)
             readers[i].Join();

}
```

Отсутствие средств синхронизации при обращении к буферу приводит к появлению гонки данных – несколько читателей могут прочитать одно и то же сообщение, прежде чем успеют обновить статус буфера; несколько писателей могут одновременно осуществить запись в буфер. В данной задаче следствием гонки данных является потеря одних сообщений и дублирование других. Для фиксации проблемы предлагается выводить на экран число повторяющихся и потерянных сообщений.

Самый простой вариант решения проблемы заключается в использовании критической секции (`lock` или `Monitor`).

```
// Работа читателя
while (!finish)
{
     lock (“read”)
     {
          if (!bEmpty)
          {
               MyMessages.Add(buffer);
               bEmpty = true;
          }
     }

}
```

Для писателей существует своя критическая секция:

```
// Работа писателя
while(i < n)
{
     lock(“write”)
     {
          if (bEmpty)
          {
               buffer = MyMessage[i++];
```

```
                    bEmpty = false;
                }
            }
        }
```

Данная реализация не является оптимальной. Каждый из читателей поочередно входит в критическую секцию и проверяет состояние буфера, в это время другие читатели блокируются, ожидая освобождения секции. Если буфер свободен, то синхронизация читателей избыточна. Более эффективным является вариант двойной проверки:

```
// Работа читателя
     while (!finish)
     {    if (!bEmpty)
          {
               lock (“read”)
               {
                    if (!bEmpty)
                    {
                         bEmpty = true;
                         MyMessage[i++] = buffer;
                    }
               }
          }
     }
```

Если буфер свободен, то читатели «крутятся» в цикле, проверяя состояние буфера. При этом читатели не блокируются. Как только буфер заполняется, несколько читателей, но не все, успевают войти в первый `if`-блок, прежде чем самый быстрый читатель успеет изменить статус буфера `bEmpty = true`.

Применение сигнальных сообщений позволяет упростить логику синхронизации доступа. Читатели ожидают сигнала о поступлении сообщения, писатели – сигнала об опустошении буфера. Читатель, освобождающий буфер, сигнализирует об опустошении. Писатель, заполняющий буфер, сигнализирует о наполнении буфера. Сообщения с автоматическим сбросом `AutoResetEvent` обладают полезным свойством – при блокировке нескольких потоков на одном и том же объекте `AutoResetEvent` появление сигнала освобождает только один поток, другие потоки остаются заблокированными. Порядок освобождения потоков при поступлении сигнала не известен, но в данной задаче это не существенно.

```
// Работа читателя
void Reader(object state)
{
     var evFull = state[0] as AutoResetEvent;
```

```
     var evEmpty = state[1] as AutoResetEvent;
     while(!finish)
     {
          evFull.WaitOne();
          MyMessage.Add(buffer);
          evEmpty.Set();
     }


// Работа писателя
void Writer(object state)
{
     var evFull = state[0] as AutoResetEvent;
     var evEmpty = state[1] as AutoResetEvent;
     while(i < n)
     {
          evEmpty.WaitOne();
          buffer = MyMessage[i++];
          evFull.Set();
     }
}
```

Данный фрагмент приводит к зависанию работы читателей. Писатели закончили работу, а читатели ждут сигнала о наполненности буфера `evFull`. Для разблокировки читателей необходимо сформировать сигналы `evFull.Set()` от писателей при завершении работы или от главного потока. Чтобы отличить ситуацию завершения можно осуществлять проверку статуса `finish` непосредственно после разблокировки.

```
     // Рабочий цикл читателей
     while(true)
     {
          evFull.WaitOne();
          // Сигнал о завершении работы
          if(finish) break;
          MyMessage.Add(buffer);
          evEmpty.Set();
     }
```

Применение семафоров (`Semaphore`, `SemaphoreSlim`) в данной задаче аналогично использованию сигнальных сообщений `AutoResetEvent`. Кроме предложенного варианта обмена сигналами между читателями и писателями, семафоры и сигнальные сообщения могут использоваться в качестве критической секции читателей и писателей.

```
void Reader(object state)
{
```

```
     var sem = state as SemaphoreSlim;
     while(!finish)
     {
          if(!bEmpty)
          {
               sem.Wait();
               if(!bEmpty)
               {
                    bEmpty = true;
                    myMessages.Add(buffer);
               }
               sem.Release();
          }
     }
}

void Writer(object state)
{
     var sem = state as SemaphoreSlim;
     while(i < myMessages.Length)
     {
          if(bEmpty)
          {
               sem.Wait();
               if(bEmpty)
               {
                    bEmpty = false;
                    buffer = myMessages[i];
               }
               sem.Release();
          }
     }
}
```