Лабораторная работа №5.  
«Синхронизация потоков»

Немного теории

Рассмотрим случай работы нескольких потоков с единственным одноэлементным буфером. Одни потоки – писатели, они записывают в буфер некоторую информацию поэлементно из своего списка. Другие потоки – читатели, они извлекают их буфера всю хранящуюся там там информацию, сохраняя её в своём списке. Изначально списки писателей чем-то заполнены, а списки писателей – все пусты. После сеанса работы списки всех писателей пусты, а списки читателей заполнены.

Вот пример программной реализации такого сеанса.

В случае одноэлементного буфера достаточно использовать флаг типа bool для контроля состояния буфера. Читатели обращаются к буферу, только если он свободен (внимание! приводится схема фрагмента кода. Реальная реализация будет сложнее!):

// 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 бузаер, стирая предыдущее сообщение. 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)  
 {  
 MyMessage[i++] = 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 неизвестен, 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.Wait();  
 // 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 semReader = state as SemaphoreSlim;  while(!finish)  {  if(!bEmpty)  {  semReader.Wait();  if(!bEmpty)  {  bEmpty = true;  myMessages.Add(buffer);  }  semReader.Release();  }  } } | void Writer(object state) {  var semWriter = state as SemaphoreSlim;  while(i < myMessages.Length)  {  if(bEmpty)  {  semWriter.Wait();  if(bEmpty)  {  bEmpty = false;  buffer = myMessages[i];  }  semWriter.Release();  }  } } |

Порядок выполнения работы1. Изучить основные теоретические положения, и выполнить предоставленные задания.

2. Сделать выводы по итогам выполнения 1-ого задания.

3. По результатам задания 3 заполнить таблицу (форма таблицы – на Ваше усмотрение).

4. Сделать выводы по итогам выполнения 4-ого задания.

Задание (5 баллов)

Несколько потоков работают с общим одноэлементным буфером.

Потоки делятся на «писателей», осуществляющих запись сообщений в буфер, и «читателей», осуществляющих извлечение сообщений из буфера.

Работу с буфером может осуществлять только один поток. 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, например, типа string. 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. Проиллюстрируйте проблему совместного доступа. Почему возникает проблема доступа?

2. Реализуйте доступ "читателей" и "писателей" к буферу с применением следующих средств синхронизации:

a. блокировки (lock);

b. сигнальные сообщения (ManualResetEvent, AutoResetEvent, ManualResetEventSlim);

c. семафоры (Semaphore, SemaphoreSlim).

3. Исследуйте производительность средств синхронизации при разном числе сообщений, разном объеме сообщений, разном числе потоков. Сделайте выводы об эффективности применения средств синхронизации.

**Требования к отчету**

Отчет по лабораторной работе должен состоять из разделов:

1. информация о студенте – ФИО, группа;

2. цель работы (подумайте сами);

3. выводы по итогам выполнения п. 1 задания;

4. результаты в табличной форме по итогам выполнения п. 3 задания

(форма таблицы – на Ваше усмотрение);

5. выводы по п.3 задания;

6. исходный код, полученный при выполнении пп. 1, 2 и 3 задания;

7. скриншоты тех блоков вывода на консоль, на основании которых сделаны выводы и построена таблица.