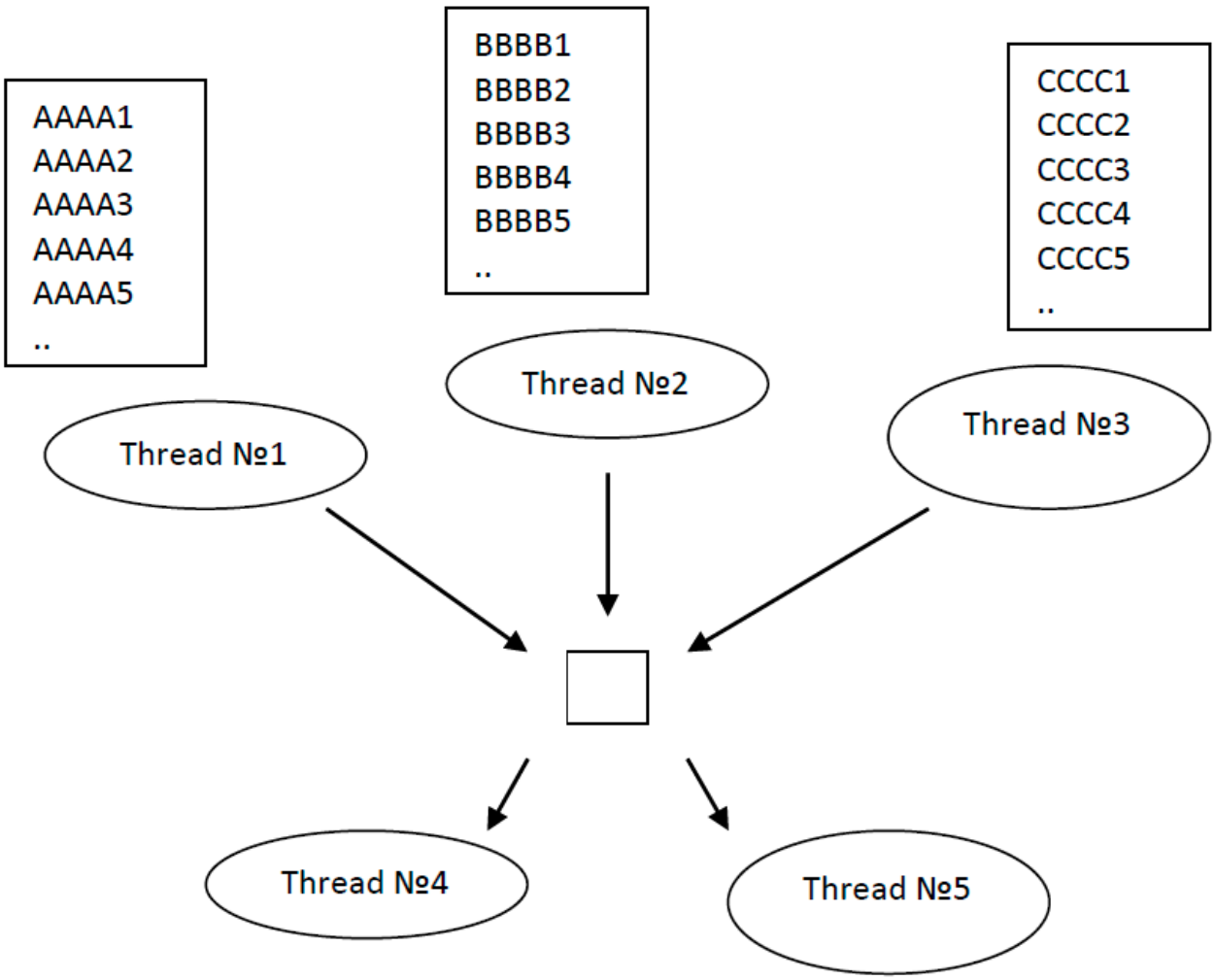
Laboratory work №3. Synchronization of access to one-element buffer

Task

Several threads work with a common one-element buffer. Threads are divided into "writers", recording messages in the buffer, and "readers", which take the extraction of messages from the buffer. Only one thread can work with a buffer. If the buffer is free, then only one writer can record it in the buffer. If the buffer is busy (filled), then only one reader can read the buffer. After reading, the buffer is released and accessible to record. A global variable is used as a buffer, such as String. The application ends after all writer messages through a common buffer will be processed by readers.



1. Implement the interaction of readers and speaker streams with a common service without any synchronization tools. Illustrate the scenery access. Why does access the problem occur?
2. Implement the access of "readers" and "writers" to the buffer using the synchronization assistances:

- `lock`;
- `ManualResetEvent` or `AutoResetEvent`;
- `Semaphore`;
- `Interlocked`.

3. Explore the performance of synchronization tools with different number of messages, different message volume, and different number of threads.
4. Make conclusions about the effectiveness of the use of synchronization tools.

Methodical instructions

In the case of one-element buffer, it is enough to use the `bool` type flag to control the buffer state. Readers turn to the buffer only if it is free:

```
// The work of a reader
while (!finish)
{
  if (!bEmpty)
  {
    MyMessages.Add(buffer);
    bEmpty = true;
  }
}
```

Writers turn to the buffer only if it is empty:

```
// The work of a writer
while(i < n)
{
  if (bEmpty)
  {
    buffer = MyMessages[i++];
    bEmpty = false;
  }
}
```

Writers work until they write all their messages. At the end of the writers, the main flow can change the status of the `finish` variable, which is a sign of the end of the readers.

```
static void Main()
{
  // Run readers and writers
  ..
  // Waiting for the completion of writers
  for(int i=0; i< writers.Length; i++)
    writers[i].Join();
  // Exposure signal for readers
  finish = true;
  // Waiting for the completion of readers
  for(int i=0; i< readers.Length; i++)
    readers[i].Join();
}
```

The lack of synchronization means when accessing a buffer leads to the appearance of data racing - several readers can read the same message before you have time to update the status of the buffer; Several writers can simultaneously record on the buffer. In this task, the consequence of the data racing is the loss of one messages and duplication of others. To fix the problem, it is proposed to display the number of repetitive and lost messages. The easiest solution to the problem is to use the critical section (`lock` or `Monitor`).

```
// The work of a reader
while (!finish)
{
  lock (“read”)
  {
    if (!bEmpty)
    {
      MyMessage[i++] = buffer;
      bEmpty = true;
    }
  }
}
```

For writers, there is its own critical section:

```
// The work of a writer
while(i < n)
```

```
{
  lock(“write”)
  {
    if (bEmpty)
    {
      buffer = MyMessage[i++];
      bEmpty = false;
    }
  }
}
```

This implementation is not optimal. Each readers alternately enters the critical section and checks the state of the buffer, at this time other readers are blocked, waiting for the release of the section. If the buffer is free, then synchronization of readers is redundant. More efficient is the option of double checking:

```
// The work of a reader
while (!finish)
{
  if (!bEmpty)
  {
    lock (“read”)
    {
      if (!bEmpty)
      {
        bEmpty = true;
        MyMessage[i++] = buffer;
      }
    }
  }
}
```

If the buffer is free, then the readers are "spin" in the cycle, checking the state of the buffer. At the same time, readers are not blocked. As soon as the buffer is filled, several readers, but not all, have time to enter the first `if` block before the fastest reader will have time to change the status of the buffer `bEmpty = true`.

The use of signaling messages allows you to simplify the logic of access synchronization. Readers are waiting for a signal that a message has arrived, and writers are waiting for a signal that the buffer has

been emptied. The reader freeing the buffer signals emptying. The writer filling the buffer signals that the buffer is full. Auto-reset messages `AutoResetEvent` have a useful property – when multiple threads are blocked on the same `AutoResetEvent` object, the appearance of a signal releases only one thread, while other threads remain blocked. The order of releasing threads when a signal arrives is not known, but this is not essential in this problem.

```
// The work of a reader
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();
  }
}
// The work of a writer
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();
  }
}
```

This fragment causes the readers' work to hang. Writers have finished work, and readers are waiting for an `evFull` buffer signal. To unlock readers, you need to form `evFull.Set()` signals from writers when completing work or from the main stream. To distinguish the termination situation, you can check the `finish` status immediately after unlocking.

```
// Working cycle of readers
while(true)
{
```

```
  evFull.WaitOne();
  // Сигнал о завершении работы
  if(finish) break;
  MyMessage.Add(buffer);
  evEmpty.Set();
}
```

Application of semaphores in this task similar to the use of signaling messages `AutoResetEvent`. In addition to the proposed exchange version of the signals between readers and writers, semaphores and signaling messages can be used as a critical section of readers and writers.

```
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();
    }
  }
}
```