

Лабораторная работа 1.7

«Мьютексы»

Когда программа работает в несколько потоков, возникает вопрос о коммуникации между ними, в частности проблема разделения памяти – попытка одновременного изменения несколькими потоками одной ячейки памяти приведет к некорректным данным в этой самой ячейке. Как правило, каждому потоку выделяется свой собственный пулл памяти, с которым он может безопасно работать. Однако иногда всё же возникают ситуации, когда потоку нужно обратиться к объекту в общей памяти, и для того, чтобы сделать это безопасно, необходима гарантия уникальности доступа к этому объекту в этот момент времени. Мьютексы призваны обеспечить эту гарантию.

Механизм работы всех мьютексов следующий – некий участок кода, в котором происходит доступ к общей памяти, помечается как критический, мьютексы регулируют, чтобы в один момент времени в критическом участке находилось не более N потоков.

Язык C# «из коробки» предоставляет следующие механизмы блокировок:

C# lock

Оператор lock позволяет привязать критическую секцию к любому объекту ссылочного типа:

```
static object locker = new object();  
.  
.  
.  
lock (locker) {  
    do_concurrency_stuff();  
}
```

В качестве примера использования lock, напомним класс, который суммирует элементы массива в несколько потоков и с определенной частотой выводит текущую посчитанную сумму.

Конструктор класса ParallelSum принимает на вход массив, элементы которого нужно просуммировать, и частоту вывода текущей посчитанной суммы. В конструктор массив разбивает на равные части, каждая часть будет привязана к своему потоку и будет просуммирована в нем:

```
public ParallelSum(int[] arr, int report_frequency=0, int nthreads=0)  
{  
    this.report_freq = report_frequency;  
    this.threads = new List<Thread>(nthreads);  
  
    // Определяем размер чанка на поток  
    int chunk_size = (int)(arr.Length / (double)nthreads + 0.5);  
    for (int i=0; i<arr.Length; i+=chunk_size)  
    {  
        threads.Add(  
            new Thread(  
                (n) => Compute(  
                    // Привязываем к потоку с индексом N соответствующий слайс  
                    // исходного массива  
                    arr, i, i+chunk_size, report_frequency  
                )  
            )  
        );  
    }  
}
```

```

        new ArraySegment<int>(
            arr,
            (int)n * chunk_size,
            Math.Min(chunk_size, arr.Length - (int)n * chunk_size)
        )
    ) { Name = $"Thread {threads.Count}"}
};
}
}

```

Метод Compute устроен следующим образом: он суммирует элементы переданного слайса в переменную своего потока, и с заданной частотой вызывает метод Report, который сообщает результат в основной поток:

```

void Report(ref int thread_res)
{
    lock (locker)
    {
        global_res += thread_res;
        thread_res = 0;
    }
}

```

Здесь понадобится блокировка, потому что наш поток меняет переменную, находящуюся в общей памяти.

Добавим выходы в консоль при входе и выходе и критической секции и посчитаем сумму последовательности $x_i = i$, $i = 0 \dots 1000$ используя 4 потока, который сообщают о результате раз в 200 операций суммирования:

```

Консоль отладки Microsoft Visual Studio
Thread 3 is going to report its result, waiting for permission to enter into critical section...
Thread 3 has just entered the critical section
Thread 3 leaving critical section...
Thread 0 is going to report its result, waiting for permission to enter into critical section...
Thread 1 is going to report its result, waiting for permission to enter into critical section...
Thread 2 is going to report its result, waiting for permission to enter into critical section...
Thread 0 has just entered the critical section
Thread 0 leaving critical section...
Thread 1 has just entered the critical section
Thread 1 leaving critical section...
Thread 3 is going to report its result, waiting for permission to enter into critical section...
Thread 2 has just entered the critical section
Thread 2 leaving critical section...
Thread 0 is going to report its result, waiting for permission to enter into critical section...
Thread 3 has just entered the critical section
Thread 3 leaving critical section...
Thread 1 is going to report its result, waiting for permission to enter into critical section...
Thread 0 has just entered the critical section
Thread 0 leaving critical section...
Thread 2 is going to report its result, waiting for permission to enter into critical section...
Thread 1 has just entered the critical section
Thread 1 leaving critical section...
Thread 2 has just entered the critical section
Thread 2 leaving critical section...
Основной поток завершил выполнение: 499500
C:\Users\rp-re\source\repos\blab1\blab7\bin\Debug\netcoreapp3.1\blab7.exe (процесс 11832) завершил работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" -> "Параметры" -> "Отладка" -> "Автоматически закрыть консоль при остановке отладки".

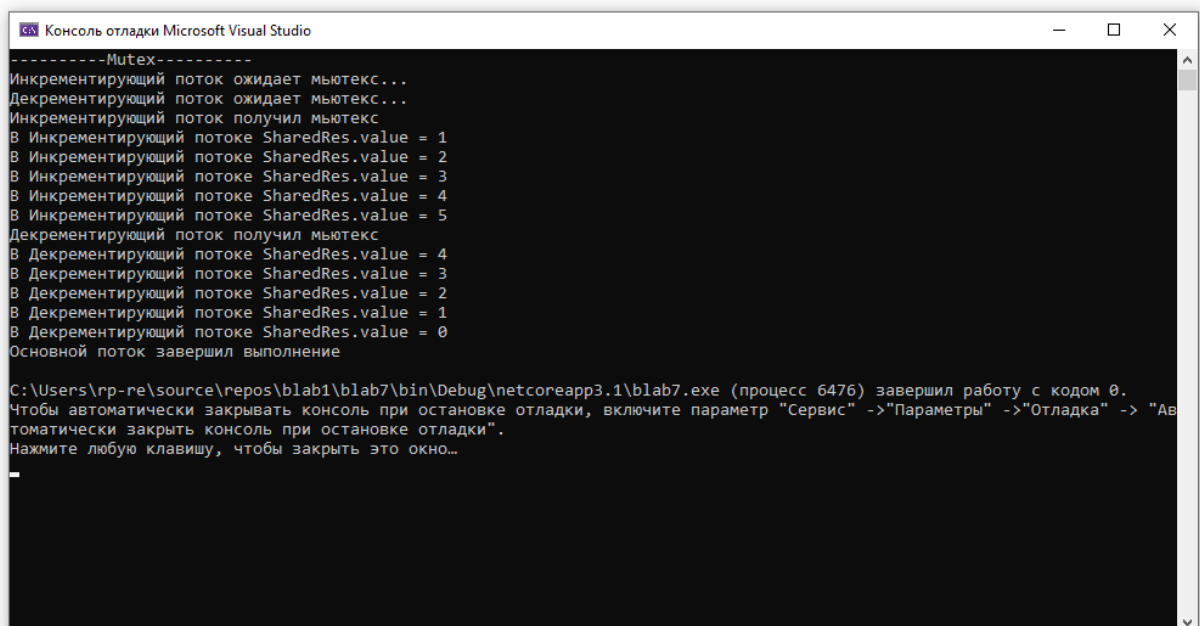
```

C# Mutex

В стандартной библиотеке `Threading` содержится класс `Mutex`, который также позволяет организовывать критические секции, доступные только одному потоку в единицу времени. Для того, чтобы встать в очереди в критическую секцию у объекта типа `Mutex` необходимо вызвать метод `WaitOne()`, затем поток блокируется до того момента, пока до него не дойдет очередь. При выходе из критической секции поток должен явно освободить её, вызвав метод `Release()`:

```
static Mutex mutex = new Mutex();  
.  
.  
.  
mutex.WaitOne();  
do_concurrency_stuff();  
mutex.Release();
```

Сгенерируем ещё одну ситуацию, когда потокам необходимо делить общую память и разрешим её с помощью `Mutex`. Пусть есть два потока: один инкрементирует переданный объект, а второй декрементирует. Передадим один и тот же объект в оба потока, но оградим работу с ним критической секцией на основе `Mutex`:



```
-----Mutex-----  
Инкрементирующий поток ожидает мьютекс...  
Декрементирующий поток ожидает мьютекс...  
Инкрементирующий поток получил мьютекс  
В Инкрементирующий потоке SharedRes.value = 1  
В Инкрементирующий потоке SharedRes.value = 2  
В Инкрементирующий потоке SharedRes.value = 3  
В Инкрементирующий потоке SharedRes.value = 4  
В Инкрементирующий потоке SharedRes.value = 5  
Декрементирующий поток получил мьютекс  
В Декрементирующий потоке SharedRes.value = 4  
В Декрементирующий потоке SharedRes.value = 3  
В Декрементирующий потоке SharedRes.value = 2  
В Декрементирующий потоке SharedRes.value = 1  
В Декрементирующий потоке SharedRes.value = 0  
Основной поток завершил выполнение  
  
C:\Users\rp-ne\source\repos\blab1\blab7\bin\Debug\netcoreapp3.1\blab7.exe (процесс 6476) завершил работу с кодом 0.  
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Автоматически закрыть консоль при остановке отладки".  
Нажмите любую клавишу, чтобы закрыть это окно...
```

C# Semaphore

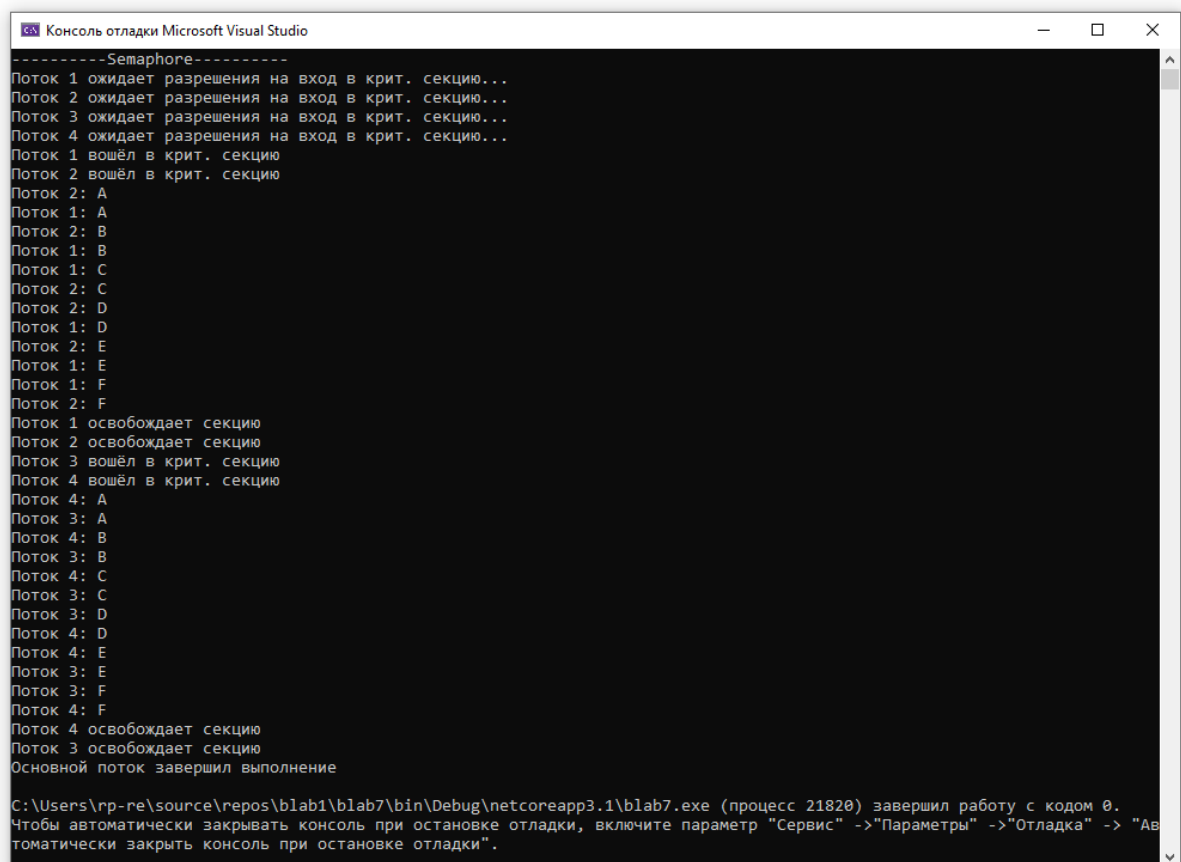
В той же стандартной библиотеке `Threading` представлен класс `Semaphore` – мьютекс, позволяющий находится в критической секции до N потоков.

В качестве аргументов конструктора класс принимает максимальное число потоков, которые могут находиться в критической секции (`maximum_count`) и сколько свободных мест в крит. секции будет изначально (`initial_count`). Остальной интерфейс эквивалентен классу `Mutex`.

```
static Semaphore mutex = new Semaphore(  
    /*initial_count=*/2,  
    /*maximum_count=*/2,
```

```
);  
.  
.  
.  
mutex.WaitOne();  
do_concurrency_stuff();  
mutex.Release();
```

Для примера напишем многопоточную программу, печатающую буквы от A-F, ограничим кол-во потоков одновременно печатающих буквы до двух с помощью Semaphore и запустим программу с четырьмя потоками:



```
-----Semaphore-----  
Поток 1 ожидает разрешения на вход в крит. секцию...  
Поток 2 ожидает разрешения на вход в крит. секцию...  
Поток 3 ожидает разрешения на вход в крит. секцию...  
Поток 4 ожидает разрешения на вход в крит. секцию...  
Поток 1 вошёл в крит. секцию  
Поток 2 вошёл в крит. секцию  
Поток 2: A  
Поток 1: A  
Поток 2: B  
Поток 1: B  
Поток 1: C  
Поток 2: C  
Поток 2: D  
Поток 1: D  
Поток 2: E  
Поток 1: E  
Поток 1: F  
Поток 2: F  
Поток 1 освобождает секцию  
Поток 2 освобождает секцию  
Поток 3 вошёл в крит. секцию  
Поток 4 вошёл в крит. секцию  
Поток 4: A  
Поток 3: A  
Поток 4: B  
Поток 3: B  
Поток 4: C  
Поток 3: C  
Поток 4: D  
Поток 3: D  
Поток 4: E  
Поток 3: E  
Поток 3: F  
Поток 4: F  
Поток 4 освобождает секцию  
Поток 3 освобождает секцию  
Основной поток завершил выполнение  
  
C:\Users\rp-re\source\repos\blab1\blab7\bin\Debug\netcoreapp3.1\blab7.exe (процесс 21820) завершил работу с кодом 0.  
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" -> "Параметры" -> "Отладка" -> "Автоматически закрыть консоль при остановке отладки".
```

Приложение

1. Исходный код программы: <https://github.com/proxodilka/csharp-labs/tree/master/blab7>