Средства синхронизации

Синхронизация необходима для координации выполнения потоков. Такая координация необходима для согласования порядка выполнения потоков или для согласования доступа потоков к разделяемому ресурсу.

Среда Framework .NET предоставляет широкий набор средств синхронизации.

Блокировка	`Join, Sleep, SpinWait`
Взаимно-исключительный доступ	`Lock, Monitor, Mutex, SpinLock`
Сигнальные сообщения	`AutoResetEvent, ManualResetEvent, ManualResetEventSlim`
Семафоры	`Semaphore, SemaphoreSlim`
Атомарные операторы	`Interlocked`
Конкурентные коллекции	`ConcurrentBag, ConcurrentQueue, ConcurrentDictionary, ConcurrentStack, BlockedCollection`
Блокировки чтения-записи	`ReaderWriterLock, ReaderWriterLockSlim`
Шаблоны синхронизации	`Barrier, CountdownEvent`

В основе синхронизации лежит понятие блокировки – один поток блокируется в ожидании определенного события от других потоков, например, завершения работы определенного потока или освобождения разделяемого ресурса.

Ожидание может быть активным или пассивным. При активном «ожидании» поток циклически проверяет статус ожидаемого события.

```
Thread thr = new Thread(SomeWork);
thr.Start();
while(thr.IsAlive) ;
```

Такая блокировка называется активной, так как фактически поток не прекращает своей работы и не освобождает процессорное время для других потоков. Активное ожидание эффективно ***только*** при незначительном времени ожидания.

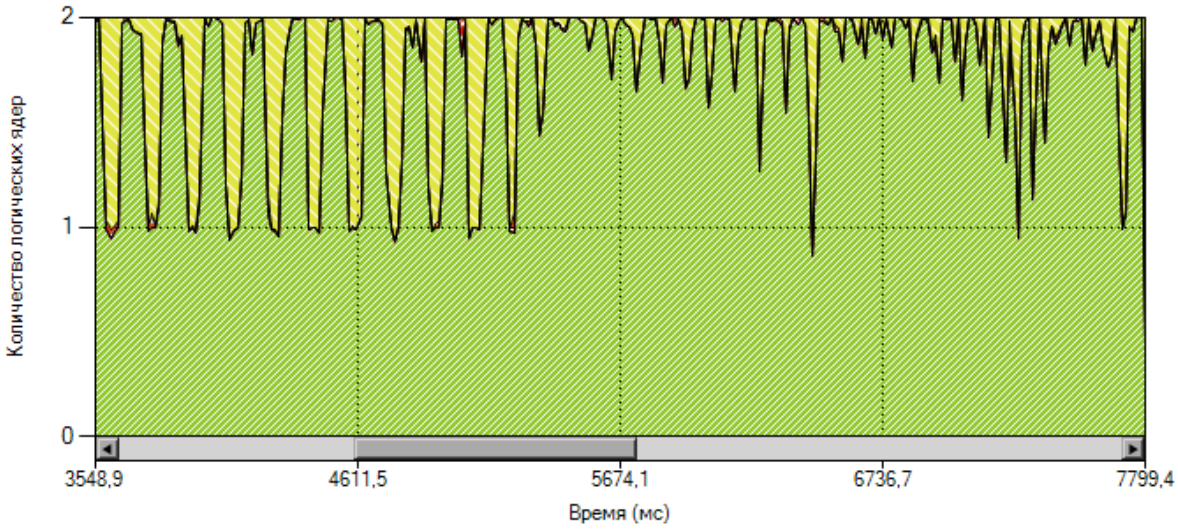
Пассивное ожидание реализуется с помощью операционной системы, которая сохраняет контекст потока и выгружает его, предоставляя возможность выполняться другим потокам. При наступлении ожидаемого события операционная система «будит» поток – загружает контекст потока и выделяет ему процессорное время. Пассивное ожидание требует времени на сохранение контекста потока при блокировке и загрузку контекста при возобновлении работы потока, но позволяет использовать вычислительные ресурсы во время ожидания для выполнения других задач.

В следующем фрагменте используются два типа ожидания. В первом случае применяется циклическая проверка статуса. Во втором случае используется метод Join.

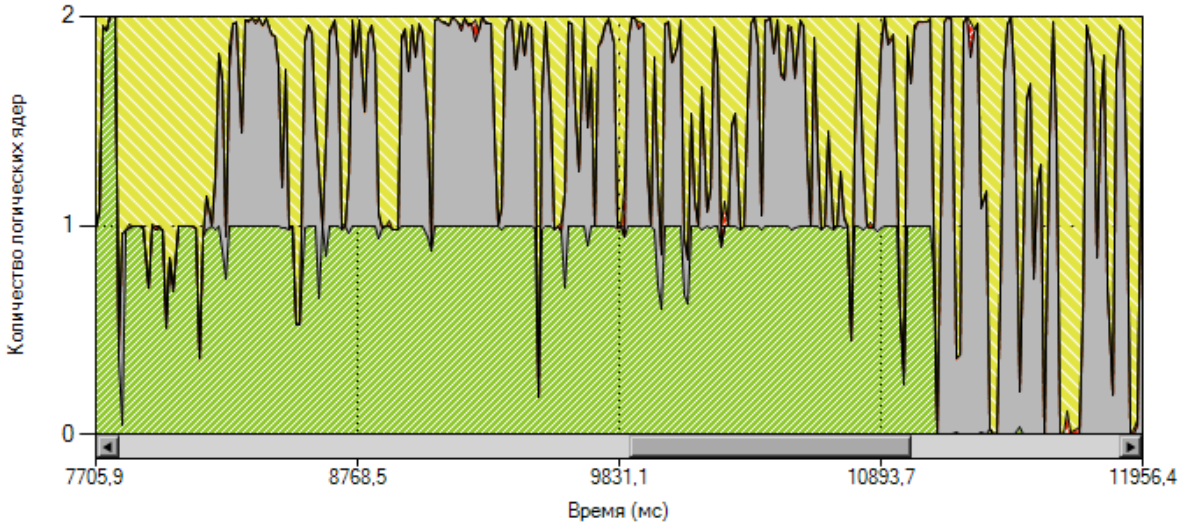
```
class Program
{
     static bool b;
     static double res;
     static void SomeWork()
     {
          for (int i=0; i<100000; i++)
               for(int j=0; j<20; j++)
               res += Math.Pow(i, 1.33);
          b = true;
     }
     static void Main()
     {
          Thread thr1 = new Thread(SomeWork);
          thr1.Start();
          // Активное ожидание в цикле
          while(!b) ;
          Console.WriteLine("Result = " + res);

          res = 0;
          Thread thr2 = new Thread(SomeWork);
          thr2.Start();
          // Ожидание с выгружением контекста
          thr2.Join();

     }
}
```

Анализ выполнения программы с помощью инструмента Visual Studio 12 «Визуализатор параллелизма» позволяет зафиксировать особенности загрузки вычислительной системы.

Загрузка процессора при активном ожидании в среднем равна 92%



При пассивном ожидании, основной поток выгружается и занятость ЦП в среднем равна 50%.



Существуют гибридные средства синхронизации, сочетающие в себе достоинства активного и пассивного ожидания. Гибридную блокировку используют объекты синхронизации, введенные в .NET 4.0: `SpinWait`, `SpinLock`, `SemaphoreSlim`, `ManualResetEventSlim`, `ReaderWriteLockSlim` и др. Потоки, блокируемые с помощью гибридных средств синхронизации, в начале фазы ожидания находятся в активном состоянии – не выгружаются, циклически проверяют статус ожидаемого события. Если ожидание затягивается, то активная блокировка становится не эффективной и операционная система выгружает ожидающий поток.

Средства для взаимного исключения

Одно из основных назначений средств синхронизации заключается в организации взаимно исключительного доступа к разделяемому ресурсу. Изменения общих данных одним потоком не должны прерываться другими потоками. Фрагмент кода, в котором

осуществляется работа с разделяемым ресурсом и который должен выполняться только в одном потоке одновременно, называется критической секцией.

```
public string data;
void DoSomeWork1()
{
     Thread.Name = "First";
     data = "AAAA";
     Console.WriteLine("Thread: {0}, Data: {1}",
                                Thread.Name, data);
}

void DoSomeWork2()
{
     Thread.Name = "Second";
     data = "BBBB";
     Console.WriteLine("Thread: {0}, Data: {1}",
                                   Thread.Name, data);

}
```

В этом фрагменте предполагается, что два потока изменяют значение общей переменной data. После внесения изменений поток выводит на экран новое значение data и имя потока. Отсутствие средств синхронизации могло бы привести к некорректному результату:

```
Thread: First, Data: BBBBB
Thread: Second, Data: BBBBB
```

Поток `First` внес свои изменения и собирался вывести сообщение, но второй поток успел вклиниться и изменить данные.

Выделение критической секции с помощью конструкции lock позволяет избежать такой ситуации:

```
lock(sync_obj)
{
     data = "AAAA";
     Console.WriteLine("Thread #1 has changed
                                  data to: {0}", data);
}
```

Когда один поток входит в критическую секцию (захватывает объект синхронизации), другой поток ожидает завершения всего блока (освобождения объекта синхронизации). Если заблокировано было несколько потоков, то при освобождении критической секции только один поток разблокируется и входит критическую секцию.

Для выделения критической секции с помощью конструкции lock необходимо указать объект синхронизации, который выступает в качестве идентификатора блокировки.

Monitor

Конструкция `lock` введена для удобства как аналог применения объекта синхронизации `Monitor`.

Итак, конструкция

```
lock(sync_obj)
{
     // Critical section
}
```

аналогична применению объекта `Monitor`:

```
try
{
     Monitor.Enter(sync_obj);
     // Critical section
}
finally
{
     Monitor.Exit(sync_obj);
}
```

Блоки `try-finally` формируются для того, чтобы гарантировать освобождение блокировки (критической секции) в случае возникновения какого-либо исключения внутри критической секции.

Кроме «обычного» входа в критическую секцию класс `Monitor` предоставляет «условные» входы:

```
     b = Monitor.TryEnter(sync_obj);
     if(!b)
     {
          // Выполняем полезную работу
          DoWork();
          // Снова пробуем войти в критическую секцию
          Monitor.Enter(sync_obj);
     }
     // Критическая секция
     ChangeData();
     // Выходим
     Monitor.Exit(sync_obj);
```

Если критическая секция уже выполняется кем-то другим, то поток не блокируется, а выполняет полезную работу. После завершения всех полезных работ поток пытается войти в критическую секцию с блокировкой.

Если доступ к критической секции достаточно интенсивен со стороны множества потоков, то полезным может быть метод `TryEnter` с указанием интервала в миллисекундах, в течение которого поток пытается захватить блокировку.

```
while(! Monitor.TryEnter(sync_obj, 100))
{
     // Полезная работа
     DoWork();
}
// Критическая секция
ChangeData();
// Выходим
Monitor.Exit(sync_obj);
```

Объект `Monitor` также предоставляет методы для обмена сигналами с ожидающими потоками `Pulse` и `Wait`, которые могут быть полезны для предотвращения взаимоблокировки в случае работы с несколькими разделяемыми ресурсами.

В следующем фрагменте два потока пытаются захватить ресурсы `P` и `Q`. Первый поток захватывает сначала `P`, затем пытается захватить Q. Второй поток сначала захватывает ресурс Q, а затем пытается захватить P. Применение обычной конструкции `lock` привело бы в некоторых случаях к взаимоблокировке потоков – потоки успели захватить по одному ресурсу и пытаются получить доступ к недостающему ресурсу. Следующий фрагмент решает проблему с помощью объекта `Monitor`:

```
void ThreadOne()
{
     // Получаем доступ к ресурсу P
     Monitor.Enter(P);
     // Пытаемся захватить ресурс Q
     if(!Monitor.TryEnter(Q))
     {
          // Если Q занят другим потоком,
          // освобождаем P и
          // ожидаем завершения работы потока
          Monitor.Wait(P);
          // Освободился ресурс P, смело захватываем и Q
          Monitor.Enter(Q);
     }
     // Теперь у потока есть и P, и Q, выполняем работу
     ..
     // Освобождаем ресурсы в обратной последовательности
     Monitor.Exit(Q);
     Monitor.Exit(P);
}

void ThtreadTwo()
{
     Monitor.Enter(Q);
```

```
            Monitor.Enter(P);
            // Выполняем необходимую работу
            ..
            // Обязательный сигнал для потока, который
            // заблокировался при вызове Monitor.Wait(P)
            Monitor.Pulse(P);
            Monitor.Exit(P);
            Monitor.Exit(Q);
      }
```

Первый поток после захвата ресурса P пытается захватить Q. Если `Q` уже занят, то первый поток, зная, что второму нужен еще и `P`, освобождает его и ждет завершения работы второго потока с обеими ресурсами. Вызов `Wait` блокирует первый поток и позволяет другому потоку (одному из ожидающих) войти в критическую секцию для работы с ресурсом P. Работа заблокированного потока может быть продолжена после того как выполняющийся поток вызовет метод `Pulse` и освободит критическую секцию. Таким образом, первый поток возобновляет работу не после вызова `Pulse`, а после вызова `Exit(P)`.

Mutex

Объект `Mutex` используется, как и `Monitor`, для обеспечения взаимно-исключительного доступа к фрагменту кода. В основе объекта Mutex лежит вызов функции ядра операционной системы, и поэтому блокировка с помощью Mutex является менее эффективной по сравнению с классом `Monitor` и конструкцией `lock`.

Отличие от `Monitor` заключается в возможности использования глобальных именованных блокировок, доступных в рамках нескольких приложений. Таким образом, с помощью объекта Mutex можно организовать синхронизацию нескольких приложений. Ядро операционной системы контролирует взаимную исключительность выполнения критических секций. Одним из примеров использования межпроцессной синхронизации является контроль количества запущенных копий приложения.

```
class MyApplication
{
      static void Main()
      {
            var mutex = new Mutex(false, "MyApp ver 2.0");

            if(!mutex.WaitOne(TimeSpan.FromSeconds(5), false))
            {
                  Console.WriteLine("Приложение уже запущено");
                  return;
            }
            Run();
            mutex.Dispose();
      }
```

```
     static void Run ()
     {
          Console.WriteLine("Welcome to MyApp ver 2.0");
          ..
     }
}
```

В этом фрагменте создается «именованный» мьютекс. Вызов метода `WaitOne` позволяет одному потоку или процессу войти в критическую секцию. Вызов метода `WaitOne` для дополнительных копий приложения возвращает признак занятости критической секции. По этому признаку приложение распознает факт, что копия уже запущена. Временной интервал в 5с используется для случаев, если запущенное приложение завершает выполнение. Освобождение мьютекса не выполняется, так как нет необходимости передавать управление другому процессу. Второй аргумент в вызове `WaitOne` указывает, что используется ожидание глобального мьютекса.

Сигнальные сообщения

Сигнальные сообщения позволяют реализовать разные схемы синхронизации, как взаимное исключение, так и условную синхронизацию. При условной синхронизации поток блокируется в ожидании события, которое генерируется в другом потоке. Платформа .NET предоставляет три типа сигнальных сообщений: `AutoResetEvent`, `ManualResetEvent и ManualResetEventSlim`, а также шаблоны синхронизации, построенные на сигнальных сообщениях `(CountdownEvent, Barrier)`. Первые два типа построены на объекте ядра операционной системы. Третий тип `ManualResetEventSlim` является облегченной версией объекта `ManualResetEvent`, является более производительным.

В следующем фрагменте два потока используют один и тот же объект типа `ManualResetEvent`. Первый поток выводит сообщение от второго потока. Сообщение записывается в разделяемую переменную. Вызов метода `WaitOne` блокирует первый поток в ожидании сигнала от второго потока. Сигнал генерируется при вызове метода Set.

```
          void OneThread(object o)
          {
               ManualResetEvent mre = (ManualResetEvent)o;
               mre.WaitOne();
               Console.WriteLine("Data from thread #2: " + data);
          }
          void SecondThread(object o)
          {
               ManualResetEvent mre = (ManualResetEvent)o;
               Console.WriteLine("Writing data");
               data = "BBBBBB";
               mre.Set();
          }
```

Вывод:

```
Writing data..
Data from thread#2: BBBB
```

Отличия инструментов AutoResetEvent и ManualResetEvent заключаются в режиме сброса статуса сигнального события: автоматическое (auto reset) или ручное (manual reset). Сигнал с автоматическим сбросом снимается сразу же после освобождения потока, блокированного вызовом WaitOne. Сигнал с ручным сбросом не снимается до тех пор, пока какой-либо поток не вызовет метод Reset.

В следующем фрагменте рассматриваются отличия сигнальных сообщений. Управляющий поток Manager запускает пять рабочих потоков и каждому передает один и тот же сигнальный объект. Рабочие потоки ожидают сигнала от управляющего потока.

```
void Worker(object initWorker)
{
     string name = ((object[])initWorker)[0] as string;
     ManualResetEvent mre =
          (object[])initWorker)[1] as ManualResetEvent;
     // Waiting to start work
     mre.WaitOne();
     Console.WriteLine("Worker {0} starts ..", name);
     // useful work
}
void Manager()
{
     int nWorkers = 5;
     Thread[] worker = new Thread[nWorkers];
     ManualResetEvent mre = new ManualResetEvent(false);
     for(int i=0; i<nWorkers; i++)
     {
          worker[i] = new Thread(Worker);
          worker[i].Start(new object[]{"#" + i, mre});
     }
     // preparing data in shared variables for workers
     // let start work
     mre.Set();
}
```

При установлении события mre работу начнут все ожидающие рабочие потоки. При замене объекта на AutoResetEvent событие будет сбрасываться автоматически и «поймает» его только какой-то один поток. Таким образом, объект AutoResetEvent можно использовать для реализации взаимно исключительного доступа.

```
static void ThreadFunc(object o)
{
     var lockEvent = o as AutoResetEvent;
     ParallelWork();
```

```
     lockEvent.WaitOne();
     CriticalWork();
     lockEvent.Set();

}
static void Main()
{
     Thread[] workers = new Thread[5];
     for(int i=0; i<5; i++)
          workers[i] = new Thread(ThreadFunc);
     var lockEvent = new AutoResetEvent(true);
     for(int i=0; i<5; i++)
          workers[i].Start(lockEvent);

}
```

В этом примере пять рабочих потоков часть работы могут выполнять параллельно, но какой-то фрагмент должны выполнять последовательно (критическая секция). Сообщение типа `AutoResetEvent` используется для организации взаимно-исключительного доступа к критической секции. Объект инициализируется с установленным сигналом для того, чтобы вначале работы один из потоков вошел в критическую секцию. При завершении выполнения критической секции поток дает сигнал одному из ожидающих потоков. Порядок вхождения потоков в критическую секцию, также как и при использовании объекта Monitor, не определен.

Объект `ManualResetEventSlim` функционально соответствует сигнальному событию с ручным сбросом. Применение гибридной блокировки повышает производительность в сценариях с малым временем ожидания. Вызов метода `Wait` в течение ограниченного промежутка времени сохраняет поток в активном состоянии (циклическая проверка статуса сигнала), если сигнал не поступил, то осуществляется вызов дескриптора ожидания ядра операционной системы.

Объекты `AutoResetEvent`, `ManualResetEvent`, а также объекты `Semaphore`, `Mutex` происходят от объекта, инкапсулирующего дескриптор ожидания ядра `WaitHandle`. Тип `WaitHandle` содержит полезные статические методы ожидания нескольких объектов синхронизации ядра:

```
var ev1 = new ManualResetEvent(false);
var ev2 = new ManualResetEvent(false);
(new Thread(SomeFunc).Start(ev1);
(new Thread(SomeFunc).Start(ev2);
// Ожидаем все сигналы
WaitHandle.WaitAll(new ManualResetEvent[] {ev1, ev2});
ev1.Reset(); ev2.Reset();
// Ожидаем хотя бы один сигнал
int iFirst = WaitHandle.WaitAny(new ManualResetEvent[]
{ev1, ev2});
```

Семафоры

Объект синхронизации `Semaphore` отличается от сигнальных событий наличием внутреннего счетчика с устанавливаемым максимальным значением. Объект `AutoResetEvent` можно интерпретировать как семафор с максимальным счетчиком равным 1 (двоичный семафор).

В следующем фрагменте рассматривается применение семафоров. В коде используется объект `SemaphoreSlim`. Вместо него можно использовать объект `Semaphore`.

```
// Применение семафоров
class SemaphoreSlimTesting
{
     private static SemaphoreSlim sem;
     private static void Worker(object num)
     {
          // Ждем сигнала от управляющего
          sem.Wait();
          // Начинаем работу
          Console.WriteLine("Worker {0} starting", num);
     }
     private static void Main()
     {
          // Максимальная емкость семафора: 5
          // Начальное состояние: 0 (все блокируются)
          sem = new SemaphoreSlim(0, 5);
          Thread[] workers = new Thread[10];
          for(int i=0; i<workers.Length; i++)
          {
               workers[i] = new Thread(Worker);
               workers[i].Start(i);
          }
          Thread.Sleep(300);
          Console.WriteLine("Разрешаем работу трем рабочим");
          sem.Release(3);
          Thread.Sleep(200);
          Console.WriteLine("Разрешаем работу еще двум рабочим");
          sem.Release(2);

     }
}
```

В методе Main инициализируется семафор `SemaphoreSlim`. Начальное значение внутреннего счетчика равно 0, максимальное значение – 5. Рабочие потоки блокируются, так как счетчик семафора равен нулю. Главный поток увеличивает счетчик на три единицы, тем самым освобождая три потока. После небольшой паузы главный поток освобождает еще два потока.

Вывод программы:

```
Разрешаем работу трем рабочим
Worker 9 starting
Worker 6 starting
Worker 0 starting
Разрешаем работу еще двум рабочим
Worker 8 starting
Worker 1 starting
```

Атомарные операторы

Библиотека .NET 4.0 предоставляет высокоэффективные атомарные операторы, которые реализованы как статические методы класса `System.Threading.Interlocked`. Атомарные операторы предназначены для потокобезопасного неблокирующего выполнения операций над данными, преимущественно целочисленного типа.

Оператор	Метод	Типы данных
Увеличение счетчика на единицу	`Increment`	`Int32, Int64`
Уменьшение счетчика на единицу	`Decrement`	`Int32, Int64`
Добавление	`Add`	`Int32, Int64`
Обмен значениями	`Exchange`	`Int32, Int64, double, single, object`
Условный обмен	`CompareExchange`	`Int32, Int64, double, single, object`
Чтение 64-разрядного целого	`Read`	`Int64`

Атомарность означает, что при выполнении оператора никто не вмешается в работу потока. Функционально, атомарные операторы равносильны критической секции, выделенной с помощью `lock, Monitor` или других средств синхронизации.

```
lock (sync_obj)
{
     counter++;
}
// можно выполнить с помощью атомарного оператора
Interlocked.Increment(ref counter);
```

Атомарные операторы являются неблокирующими - поток не выгружается и не ожидает, поэтому обеспечивают высокую эффективность. Выполнение оператора `Interlocked` занимает вдвое меньшее время, чем выполнение критической секции с `lock`-блокировкой без конкуренции.

Оператор `Interlocked.CompareExchange` позволяет атомарно выполнить конструкцию «проверить-присвоить»:

```
lock(LockObj)
{
     if(x == curVal)
          x = newVal;
}

oldVal = Interlocked.CompareExchange(ref x, newVal, curVal);
```

Если значение переменной x равно значению, задаваемому третьим аргументом `curVal`, то переменной присваивается значение второго аргумента `newVal`. Возвращаемое значение позволяет установить, осуществилась ли замена значения.

Атомарный оператор `Read` предназначен для потокобезопасного чтения 64-разрядных целых чисел `(Int64)`. Чтение значений типа `long (Int64)` на 32-разрядной вычислительной системе не является атомарной операцией на аппаратном уровне. Поэтому многопоточная работа с 64-разрядными переменными может приводить к некорректным результатам. В следующем фрагменте проиллюстрируем проблему чтения переменных типа Int64:

```
Int64 bigInt = Int64.MinValue;
Thread t = new Thread(() => {
               while(true) {
                    if(bigInt == Int64.MinValue)
                         bigInt = Int64.MaxValue;
                    else
                         bigInt = Int64.MinValue;
               });
t.Start();
List<Int64> lstBig = new List<64>();
for(int i=0; i < 1000; i++)
{
     Thread.Sleep(100);
     lstBig.Add(bigInt);
}
t.Abort();

Console.WriteLine("Distinct values: "
                                        + lstBig.Distinct().Count());
lstBig.Distinct().AsParallel().ForAll(Console.WriteLine);
```

В этом примере значение переменной `bigInt` изменяется только в одном потоке. Основной поток периодически читает текущие значения `bigInt`. Поток t циклически меняет значение переменной `bigInt` с `MinValue` на `MaxValue` и с `MaxValue` на `MinValue`. Тем не менее, вывод показывает, что основной поток прочитал и другие значения. Эти «промежуточные» значения появились из-за не атомарности действий над 64-разрядными переменными – пока основной поток прочитал первые 32 бита числа, дочерний поток изменил следующие 32 бита. Предпоследняя строчка выводит число различных значений переменной `bigInt`, прочитанных в основном потоке. Последняя строчка выводит на консоль все различные значения.

```
Distinct values: 4
-9223372036854775808
-9223372032559808513
9223372036854775807
9223372032559808512
```

Для устранения проблемы необходимо сделать атомарным запись и чтение переменной `bigInt`:

```
Int64 bigInt = Int64.MinValue;
Thread t = new Thread(() => {
     Int64 oldValue = Interlocked.CompareExchange(ref bigInt,
                          Int64.MinValue, Int64.MaxValue);
     Interlocked.CompareExchange(ref bigInt,
                          Int64.MaxValue, oldValue);
               });
t.Start();
List<Int64> lstBig = new List<Int64>();
for(int i=0; i < 1000; i++)
{
     Thread.Sleep(100);
     lstBig.Add(Interlocked.Read(ref bigInt));
}
t.Abort();

Console.WriteLine("Distinct values: "
                                     + lstBig.Distinct().Count());
lstBig.Distinct().AsParallel().ForAll(Console.WriteLine);
```

Изменение `bigInt` реализовано с помощью двух операторов `CompareExchange`. Первый оператор пытается присвоить `MinValue`, если текущее значение равно `MaxValue`. Оператор возвращает старое значение. Сравнивая текущее со старым значением, определяем, произошло ли изменение. Если изменения не было, то присваиваем максимальное значение `MaxValue`. Атомарное чтение реализовано с помощью оператора `Interlocked.Read`. Вывод результатов свидетельствует о решении проблемы:

```
Distinct value: 2
-9223372036854775808
9223372036854775807
```

Операции над 64 разрядными целыми на 64-разрядной системе являются атомарными на аппаратном уровне, поэтому не требуют средств синхронизации при параллельной записи и чтении. Но при параллельной записи в нескольких потоках, возникает проблема гонки данных.