C# Асинхронное программирование

Потокобезопасные коллекции. Producer-Consumer Pattern. Класс Parallel.

C# Асинхронное программирование

Автор курса

Гнатюк Владислав

MCID:16354168

Асинхронное программирование

После урока обязательно

Повторите этот урок в видео формате на ITVDN.com

Проверьте как Вы усвоили данный материал на TestProvider.com

Потокобезопасные коллекции. Producer-Consumer Pattern. Класс Parallel.

План урока

1) Потокобезопасные коллекции
2) Разновидности потокобезопасных коллекций
3) ConcurrentQueue<T>
4) ConcurrentStack<T>
5) ConcurrentBag<T>
6) Шаблон Producer-Consumer. IProducerConsumerCollection<T>
7) Класс BlockingCollection<T>
8) Параллельная обработка. Класс Parallel
9) Параллельные циклы (For, ForEach)

Коллекция и потокобезопасная коллекция

Коллекция – это объект, который содержит набор сгруппированных данных с поддержкой их перебора, изменения, добавления и удаления. Одним словом – удобный интерфейс для взаимодействия с набором данных.

Потокобезопасная коллекция – та же коллекция, но работа с ее элементами из нескольких потоков безопасна, в отличие от обычной коллекции. То есть разные потоки могут читать и писать в коллекцию без опасности потери или повреждения данных.

C# Асинхронное программирование

Техники синхронизации потокобезопасных коллекций

- Конструкции синхронизации (lock, Monitor, SpinWait…).
- Атомарные инструкции (Interlocked, volatile).
- Неблокирующие алгоритмы.

```
lock (syncRoot)
{
    queue.Enqueue(item);
}
```

Разновидности потокобезопасных коллекций

Эквиваленты потокобезопасным коллекциям

Потокобезопасные коллекции:

- ConcurrentDictionary<TKey, TValue>
  - Dictionary<TKey, TValue>
  - SortedDictionary<TKey, TValue>
  - SortedList<TKey, TValue>
- ConcurrentQueue<T>
  - Queue<T>
- ConcurrentStack<T>
  - Stack<T>
- ConcurrentBag<T>
  - Нет эквивалента из обычных коллекций

Обычные коллекции (нет потокобезопасного аналога):

- T[]
- List<T>
- LinkedList<T>
- HashSet<T>
- SortedSet<T>
- ObservableCollection<T>

Потокобезопасная очередь - ConcurrentQueue<T>

ConcurrentQueue<T> - потокобезопасная коллекция, работающая по принципу FIFO (First In First Out).

Коллекцию, работающую по принципу FIFO называют - **очередью**.

Принципы работы очереди:

1. При добавлении элемента, он будет добавлен в конец очереди.
2. При извлечении элемента, вы будете извлекать их из начала очереди, при этом удаляя его из очереди.

ConcurrentQueue<T> - работа с элементами

Открытые API для работы с элементами:

void Enqueue(T item) – добавляет элемент в конец очереди.

bool TryDequeue(out T item) – извлекает элемент из начала очереди, удаляя его.

bool TryPeek (out T item) – извлекает элемент из начала очереди для просмотра.

Методы, подобные «bool TryXXX» возвращают true – если метод выполнился успешно, false – если у метода не получилось выполниться. Элемент они возвращают в out параметре.

ConcurrentQueue<T> - работа с элементами

Открытые API для работы с элементами:

void Enqueue(T item) – добавляет элемент в конец очереди.

bool TryDequeue(out T item) – извлекает элемент из начала очереди, удаляя его.

bool TryPeek (out T item) – извлекает элемент из начала очереди для просмотра.

Методы, подобные «bool TryXXX» возвращают true – если метод выполнился успешно, false – если у метода не получилось выполниться. Элемент они возвращают в out параметре.

Потокобезопасный стек - ConcurrentStack<T>

ConcurrentStack<T> - потокобезопасная коллекция, работающая по принципу LIFO (Last In First Out).

Коллекцию, работающую по принципу LIFO называют - стеком.

Принципы работы стека:

1. При добавлении элемента, он будет добавлен в начало (на вершину) стека.
2. При извлечении элемента, вы будете извлекать последний добавленный элемент, при этом удаляя его из стека.

ConcurrentStack<T> - работа с элементами

Открытые API для работы с элементами:

void Push(T item) – добавляет элемент в начало (на вершину) стека.

void PushRange(T[] items) – добавляет элементы массива в начало (на вершину) стека.

void PushRange(T[] items, int startIndex, int count) – добавляет указанное (count) количество элементов массива, начиная с указанного индекса (startIndex) в начало (на вершину) стека.

bool TryPeek (out T item) – извлекает элемент из начала (вершины) стека для просмотра.

bool TryPop(out T item) – извлекает элемент из начала (вершины) стека, удаляя его.

int TryPopRange(T[] items) – извлекает элементы из начала (вершины) стека, добавляя их в массив. При этом элементы стека, добавленные в массив, из стека удаляются. Метод пытается извлечь количество элементов, совпадающих с размером переданного массива. TryPopRange возвращает количество извлеченных элементов.

int TryPopRange(T[] items, int startIndex, int count) – извлекает указанное (count) количество элементов из начала (вершины) стека, добавляя их в массив, начиная с указанного индекса (startIndex). При этом элементы стека, добавленные в массив, из стека удаляются. Метод пытается извлечь количество элементов, совпадающих с размером переданного массива. TryPopRange возвращает количество извлеченных элементов.

Методы «bool TryXXX» возвращают true – если метод выполнился успешно, false – если у метода не получилось выполниться. Элемент они возвращают в out параметре.

Потокобезопасная «сумка» - ConcurrentBag<T>

ConcurrentBag<T> - неупорядоченная потокобезопасная коллекция.

ConcurrentBag подходит в случаях, когда порядок элементов не имеет значения. Поэтому, изымая элемент вы можете получить совсем не то, что ожидали.

ConcurrentBag<T> - работа с элементами

Открытые API для работы с элементами:

void Add(T item) – добавляет элемент в коллекцию.

bool TryTake(out T item) – извлекает элемент из коллекции, удаляя его.

bool TryPeek (out T item) – извлекает элемент из коллекции для просмотра.

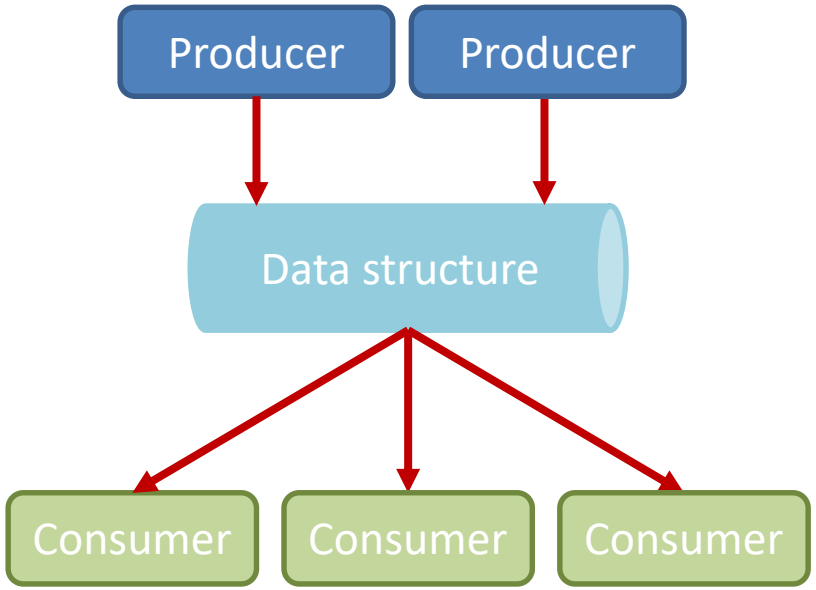
Методы «bool TryXXX» возвращают true – если метод выполнился успешно, false – если у метода не получилось выполниться. Элемент они возвращают в out параметре.

Producer-Consumer Pattern

Шаблон Producer-Consumer подходит для ситуаций, когда скорость получения (генерации) данных/задач отличается от скорости обработки данных/задач.

Producer – это изготовитель (поставщик) данных (задач), который создает или поставляет данные (задачи) в структуру данных.

Consumer – это потребитель, который берет данные (задачи) из структуры данных и выполняет над ними манипуляции (обрабатывает, выполняет, отправляет результаты…).



Интерфейс IProducerConsumerCollection<T>

Интерфейс IProducerConsumerCollection<T> реализует шаблон Producer-Consumer с помощью потокобезопасных коллекций.

Методы интерфейса IProducerConsumerCollection<T>:

void CopyTo(T[] array, int index) – скопировать содержимое коллекции в массив.
T[] ToArray() – превратить коллекцию в массив.
bool TryAdd(T item) – попытаться добавить элемент в коллекцию.
bool TryTake(out T item) – попытаться получить и удалить элемент из коллекции.

Класс BlockingCollection<T>

BlockingCollection<T> - объект, который является оболочкой для потокобезопасных коллекций интерфейса IProducerConsumerCollection<T>.

BlockingCollection<T> позволяет:

- Одновременное добавление (Add) и удаление (Take) элементов из нескольких потоков.
- Поддержка ограничения и блокировки. Блокирование операции Add или Take, когда коллекция заполнена или пустая.
- Возможность отмены выполнения методов Add/TryAdd и Take/TryTake с помощью CancellationToken или тайм-аута.

Коллекция BlockingCollection<T> реализует интерфейс IDisposable. Когда вы закончили работать с коллекцией, вы должны избавиться от нее, вызвав метод Dispose руками или через конструкцию using.

Метод Dispose не является потокобезопасным! Все остальные открытые и защищенные члены BlockingCollection<T> могут использоваться одновременно из нескольких потоков.

BlockingCollection<T>

При создании объекта BlockingCollection<T> вы можете задать несколько настроек для коллекции с помощью конструктора :

1. Указать какой тип коллекции будет использован внутри. Этот параметр принимает объект, реализующий интерфейс IProducerConsumerCollection<T>. По умолчанию при создании типа используется ConcurrentQueue<T>.

2. Установка максимальной вместимости коллекции. Это ограничение важно, потому что оно позволяет контролировать максимальный размер коллекции в памяти.

Конструкторы:

```
public BlockingCollection();
public BlockingCollection(int boundedCapacity);
public BlockingCollection(IProducerConsumerCollection<T> collection);
public BlockingCollection(IProducerConsumerCollection<T> collection, int boundedCapacity);
```

BlockingCollection<T> - Свойства

Свойства:

bool IsAddingCompleted { get; } – показывает, помечена ли коллекция как завершенная для добавления.

bool IsCompleted { get; } – показывает, помечена ли коллекция для добавления и является ли она пустой.

int BoundedCapacity { get; } – отдает установленную ограниченную емкость.

int Count { get; } – отдает количество элементов.

BlockingCollection<T> - Работа с элементами

Методы для добавления/удаления элементов коллекции (БЛОКИРУЮЩИЕ):

- Добавление элементов:

void Add (T item) – добавляет элемент в коллекцию. Если при инициализации коллекции была указана ограниченная емкость, выполнение метода Add может заблокировано, пока не освободится место для добавления элемента.

void Add (T item, CancellationToken cancellationToken) – делает то же, что и метод Add выше, но, при этом еще и поддерживает отмену выполнения.

- Удаление элементов:

T Take() – возвращает и удаляет элемент из коллекции. Если в коллекции нет элементов, метод Take заблокирует поток, пока он не получит элемент.

T Take(CancellationToken cancellationToken) – делает то же, что и метод Take выше, но при этом еще и поддерживает отмену выполнения.

BlockingCollection<T> - Работа с элементами

Методы для добавления/удаления элементов коллекции (НЕБЛОКИРУЮЩИЕ):

bool TryAdd – пытается добавить элемент в коллекцию. Если коллекция является ограниченной и она в данный момент заполнена до предела, то этот метод немедленно возвращает значение false, не добавляя элемент. Если элемент был успешно добавлен - возвращает значение true. У метода TryAdd есть различные перегрузки, которые поддерживают отмену или тайм-аут на выполнение:

```
(T item);
(T item, int millisecondsTimeout);
(T item, TimeSpan timeout);
(T item, int millisecondsTimeout), CancellationToken cancellationToken);
```

bool TryTake – пытается вернуть и удалить элемент из коллекции. Если коллекция пустая – метод немедленно возвращает значение false. Если метод TryTake смог найти элемент, поместить его значение в out параметр и удалить из коллекции, то метод возвращает значение true. У метода TryTake есть различные перегрузки, которые поддерживают отмену или тайм-аут на выполнение:

```
(out T item);
(out T item, int millisecondsTimeout);
(out T item, TimeSpan timeout);
(out T item, int millisecondsTimeout, CancellationToken cancellationToken);
```

BlockingCollection<T>

BlockingCollection<T> поддерживает ограничение и блокировку. Под ограничением имеется в виду то, что вы можете установить максимальную вместимость коллекции.

- Несколько потоков-поставщиков могут одновременно добавлять элементы в коллекцию. Если коллекция будет наполнена до максимально указанной емкости, то потоки-поставщики будут заблокированы на операции добавления, пока элемент не будет удален.

- Несколько потоков-потребителей могут одновременно удалять элементы из коллекции. Если коллекция станет пустой, потоки-потребители будут заблокированы, пока потоки-поставщики не добавят новые элементы.

- Если добавление элементов окончено, то для того чтобы разблокировать потоки-потребители, которые ждут новых элементов, необходимо вызвать метод CompleteAdding(). Он указывает, что больше новых элементов не будет добавлено и будит потоки исключением InvalidOperationException.

- Потоки-потребители могут узнать, что коллекция пуста и элементов больше не будет добавлено с помощью свойства IsCompleted.

BlockingCollection<T> - IEnumerable

BlockingCollection<T> реализует интерфейс IEnumerable<T>.

В BlockingCollection<T> есть две версии метода для получения перечислителя:

- Метод GetEnumerator() реализован явно (explicitly). Он возвращает перечислителя со «снимком» коллекции элементов. Под снимком имеется в виду получение элементов на момент вызова метода.

- Метод GetConsumingEnumerable() возвращает перечислитель, который будет отдавать (удалять из коллекции) элементы (если они есть в коллекции) до тех пор, пока значение свойства IsCompleted не станет равным true. Если элементов в коллекции нет и значение свойства IsCompleted равно false - цикл блокируется до тех пор, пока не появится доступный элемент или до отмены CancellationToken.

Метод GetConsumingEnumerable имеет две перегрузки. Первая - без параметров, вторая - принимает один параметр типа CancellationToken. Это показывает, что отмена является необязательной.

Параллельное программирование

.NET предлагает вам несколько стандартных решений для распараллеливания кода:

- Типы System.Threading.Tasks.Task и System.Threading.Tasks.Task<TResult>
- Тип System.Threading.Tasks.Parallel
- Параллельный LINQ (PLINQ), в виде типов System.Linq.ParallelEnumerable и System.Linq.ParallelQuery<T>.

Класс Parallel

Тип Parallel – это класс, который упрощает параллельное выполнение кода. Благодаря его методам можно быстро распараллелить вычислительный процесс.

У класса Parallel доступно всего 3 метода для параллельной обработки:

- Invoke – параллельное выполнение делегатов Action.
- For – параллельное выполнение итераций.
- ForEach – параллельный перебор коллекций.

При этом, параллельное выполнение этих методов можно настраивать с помощью выбора перегрузки, которая принимает в качестве параметров класс ParallelOptions.

ParallelOptions

С помощью класса ParallelOptions можно настроить выполнение параллельных методов. Для этого необходимо создать экземпляр класса ParallelOptions и задать необходимые параметры.

Класс ParallelOptions имеет 3 различных настройки:

1. CancellationToken – указание CancellationToken для возможности отмены выполнения параллельных операций.

2. MaxDegreeOfParallelism – возможность для установки или получения максимального количества одновременных задач.

3. TaskScheduler – установка своего планировщика задач для параллельного выполнения.

Параллельное выполнение класса Parallel

Класс Parallel пытается выполнить параллельный вызов, итерирование или перебор с помощью наименьшего количества задач, необходимых для максимально быстрого завершения, при этом учитывая ваши ограничения по распараллеливанию (MaxDegreeOfParallelism). Для этого класс Parallel использует самореплицирующиеся задачи (Self-Replicating Tasks).

Реплицируемая задача - это задача, делегат которой может быть выполнен несколькими потоками одновременно. На количество саморепликаций задачи влияет указание максимального значения по распараллеливанию (MaxDegreeOfParallelism). Если оно указано, то система не будет превышать число реплицирований задачи более чем значение MaxDegreeOfParallelism.

*Именно самореплицирующиеся задачи используются параллельными циклами **For** и **ForEach**. Метод **Invoke** использует самореплицирующиеся задачи, если количество делегатов на выполнение превышает 10 единиц или если указана настройка максимального распараллеливания.*

ParallelLoopState

Класс ParallelLoopState позволяет отдельным параллельным итерациям параллельных циклов взаимодействовать друг с другом, поскольку методы Parallel.For и Parallel.ForEach отдельные итерации выполняют параллельно.

Методы:

- Break – предотвращает выполнение любых итераций с индексом, превышающим текущий. Не влияет на выполняющиеся итерации.
- Stop – заканчивает текущую итерацию и запрещает запуск дополнительных итераций как только это возможно. Не влияет на выполняющиеся итерации.

Свойства:

- ShouldExitCurrentIteration – определяет, вызывала ли какая-то из итераций метод Stop, Break или было выброшено исключение.
- IsExceptional – определяет, произошло ли исключение в какой-то из итераций цикла.
- IsStopped – определяет, вызывала ли какая-то итерация цикла метод Stop.
- LowestBreakIteration – отдает индекс итерации, в которой был вызван Break.

Исключения в методе Invoke

Исключение, возникшее в одном из делегатов, выполняемого методом **Invoke(),** не прервет работу других делегатов или потоков. Делегат, в котором произошло исключение, прервет свою работу. Все исключения, возникшие в делегатах, собираются и помещаются в исключение AggregateException. Оно будет выброшено через вызов метод **Invoke()**.

Для обработки исключений метода **Invoke()** необходимо помещать его вызов в тело блока try конструкции try-catch.

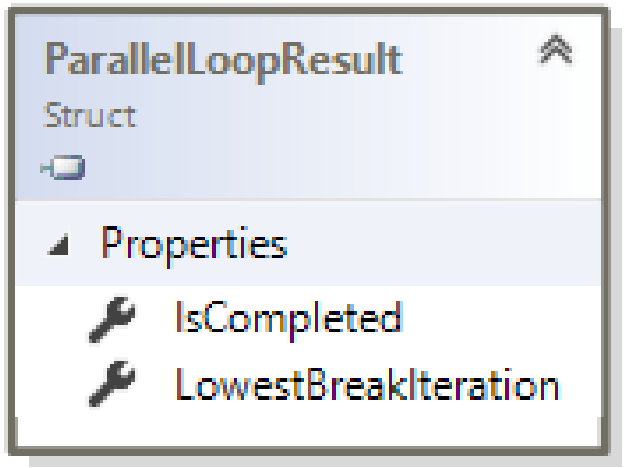
Исключения в параллельных циклах

Исключение в одной из итераций параллельного цикла приводит к полному прерыванию работы всего цикла. Все исключения из итераций собираются и помещаются в исключение AggregateException. Оно будет выброшено через вызов метода For или ForEach.

Для обработки исключений параллельных циклов необходимо помещать вызов метода For или ForEach в тело блока try конструкции try-catch.

ParallelLoopResult

Структура ParallelLoopResult предоставляет возможность просмотреть статус выполнения параллельного цикла. Все перегрузки методов Parallel.For и Parallel.ForEach возвращают после своего выполнения экземпляр структуры ParallelLoopResult.



Свойства:

- IsCompleted – возвращает значение true, если все параллельные итерации были выполнены, false – если параллельная обработка была нарушена вызовом метода Stop, Break или произошло исключение.

- LowestBreakIteration – возвращает значение null, если для остановки цикла был вызван метод Stop. Возвращает целочисленное значение, если для завершения цикла был использован метод Break.

Блокирование вызывающего потока в классе Parallel

Если блокирование вызывающего потока при использовании класса Parallel для вас неудобно, то вы можете использовать класс Task для решения.

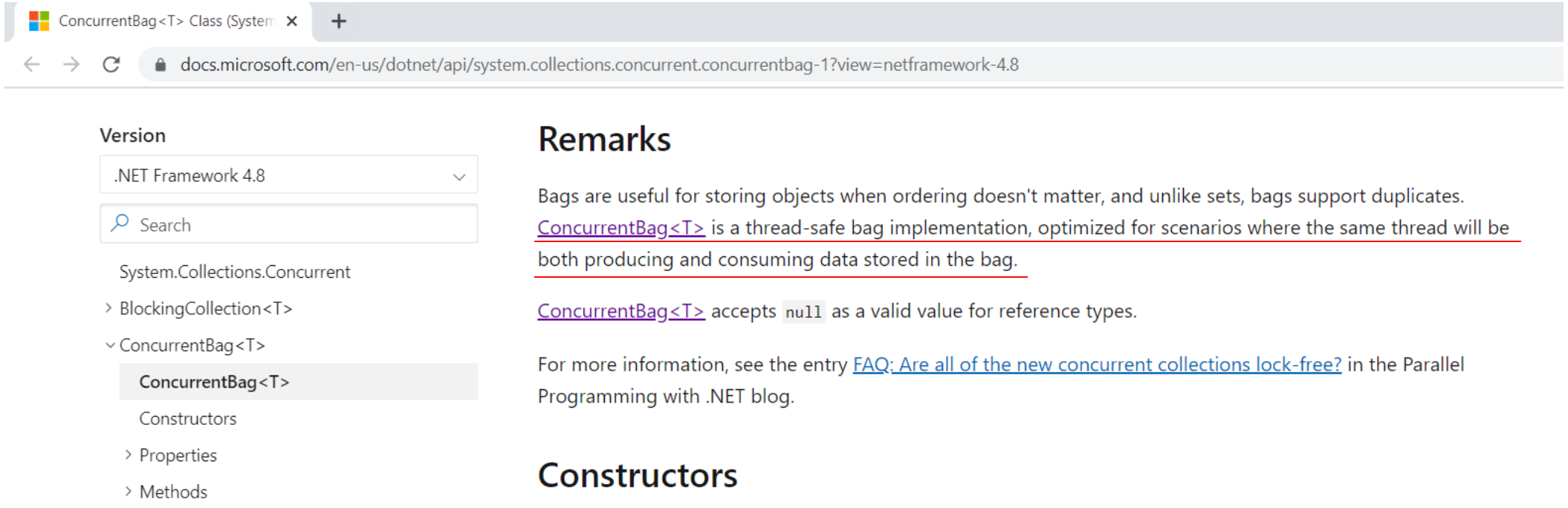
Поместите вызов метода класса Parallel в задачу (например: через метод Task.Run) и воспользуйтесь ключевыми словами async await для неблокирующего ожидания завершения выполнения задачи.

Пример:

```
public async Task OperationAsync<T>(IEnumerable<T> elements, Action<T> loopBody)
{
        await Task.Run(() => Parallel.ForEach(elements, loopBody);
}
```

Использование ConcurrentBag

На Microsoft docs в описании класса ConcurrentBag написано, что его следует использовать в сценариях, когда один и тот же поток как записывает данные, так и потребляет их.



В чем причина такого требования ConcurrentBag

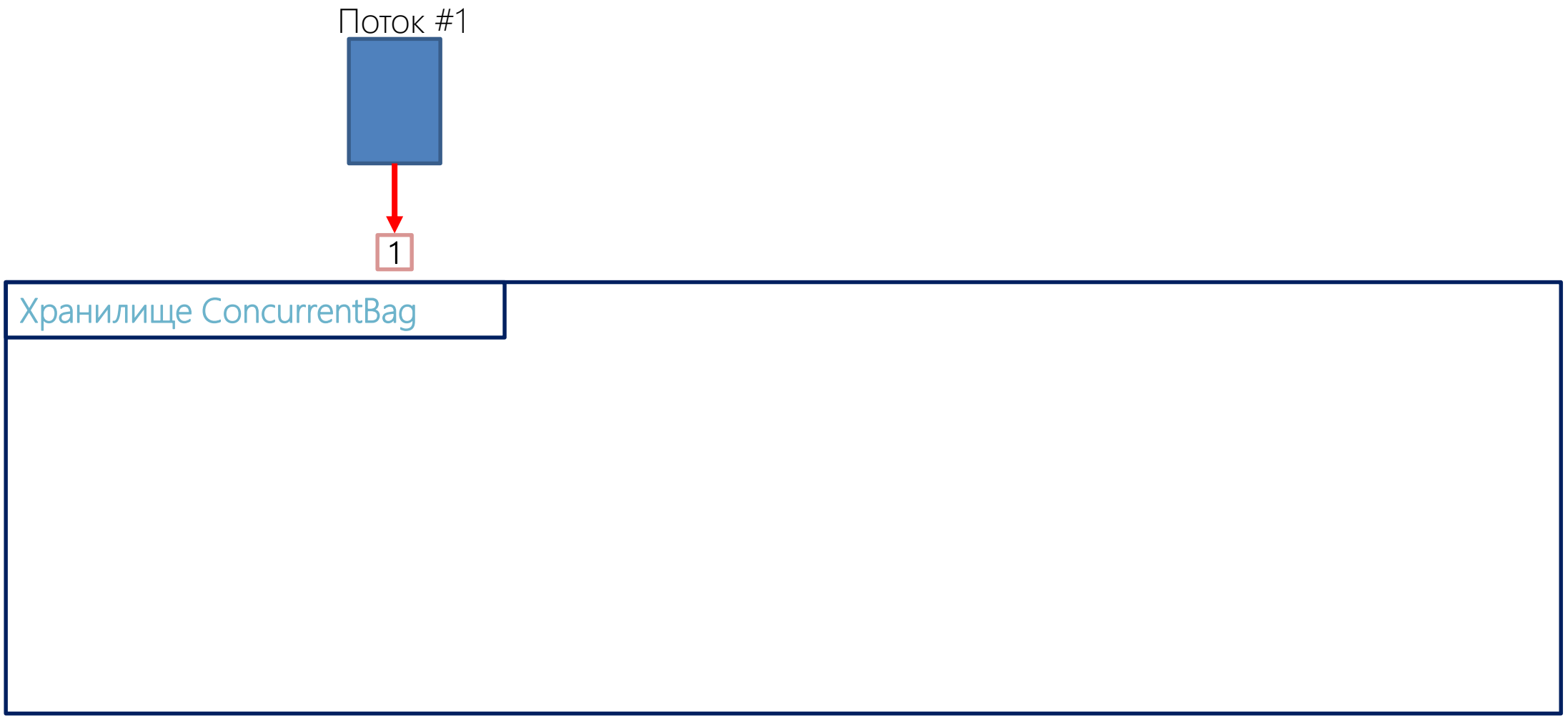
Все дело во внутреннем устройстве класса ConcurrentBag.

Все данные ConcurrentBag хранит в однонаправленном связном списке. Он представлен внутренним классом ThreadLocalList, экземпляр которого создается для каждого нового потока, который добавляет элемент в коллекцию. Каждый экземпляр этого класса имеет ссылку на созданный другим потоком экземпляр ThreadLocalList. Это и создает однонаправленный связный список из элементов ThreadLocalList.

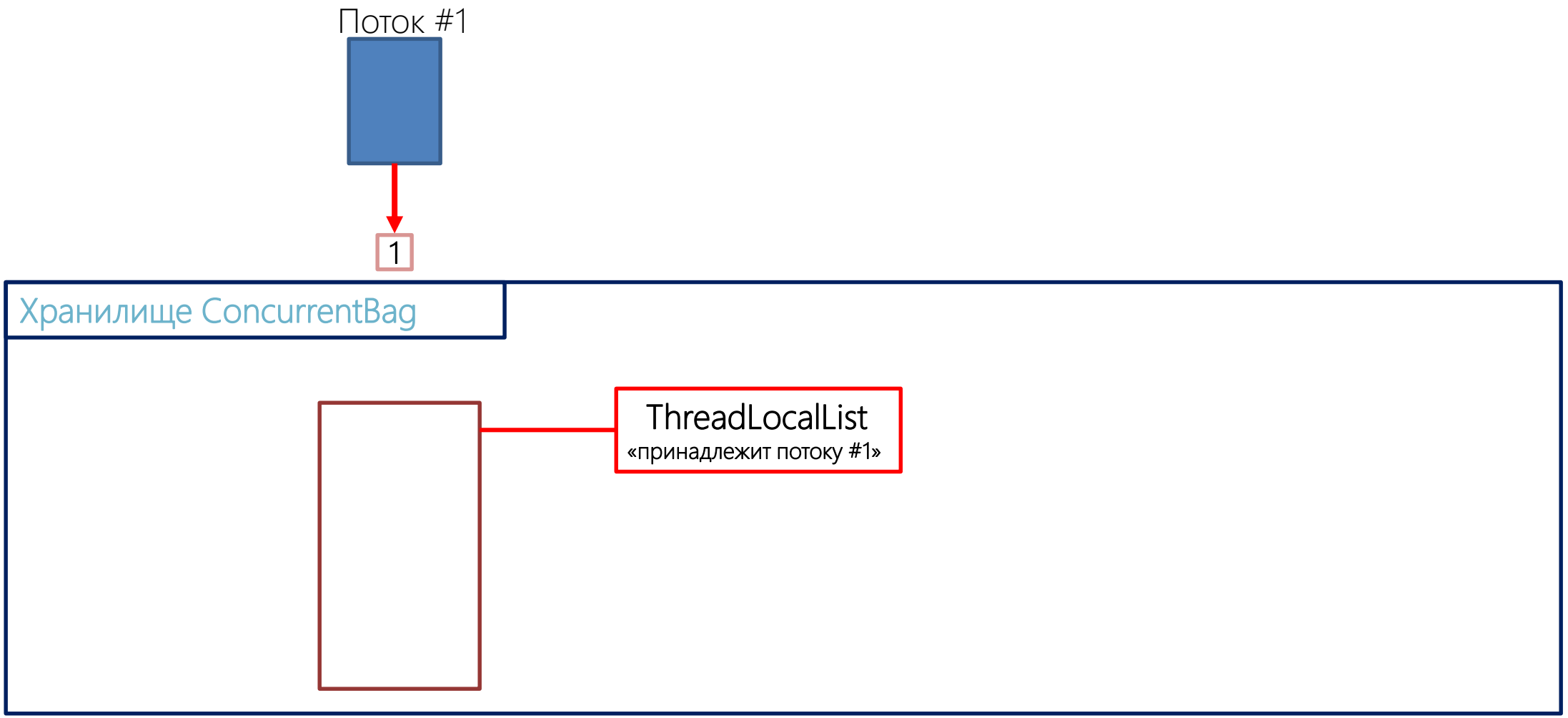
ThreadLocalList содержит внутри себя двунаправленный связный список. Он представлен внутренним классом Node. Этот класс хранит добавленные элементы в коллекцию. Каждый экземпляр класса Node имеет ссылку на следующий и предыдущий элементы Node. Из-за этого набор таких элементов становится двунаправленным связным списком.

Добавление элементов в ConcurrentBag

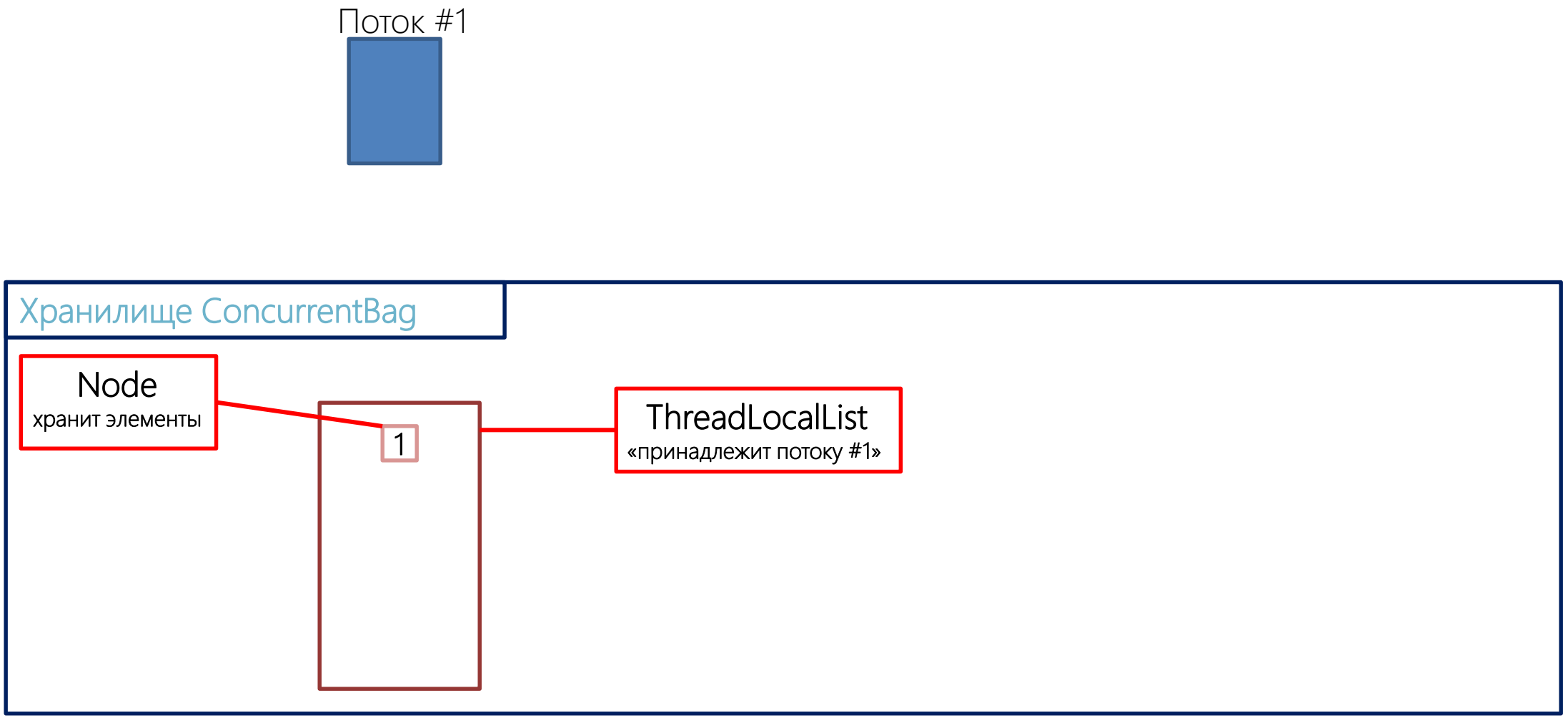
Добавление элементов в ConcurrentBag



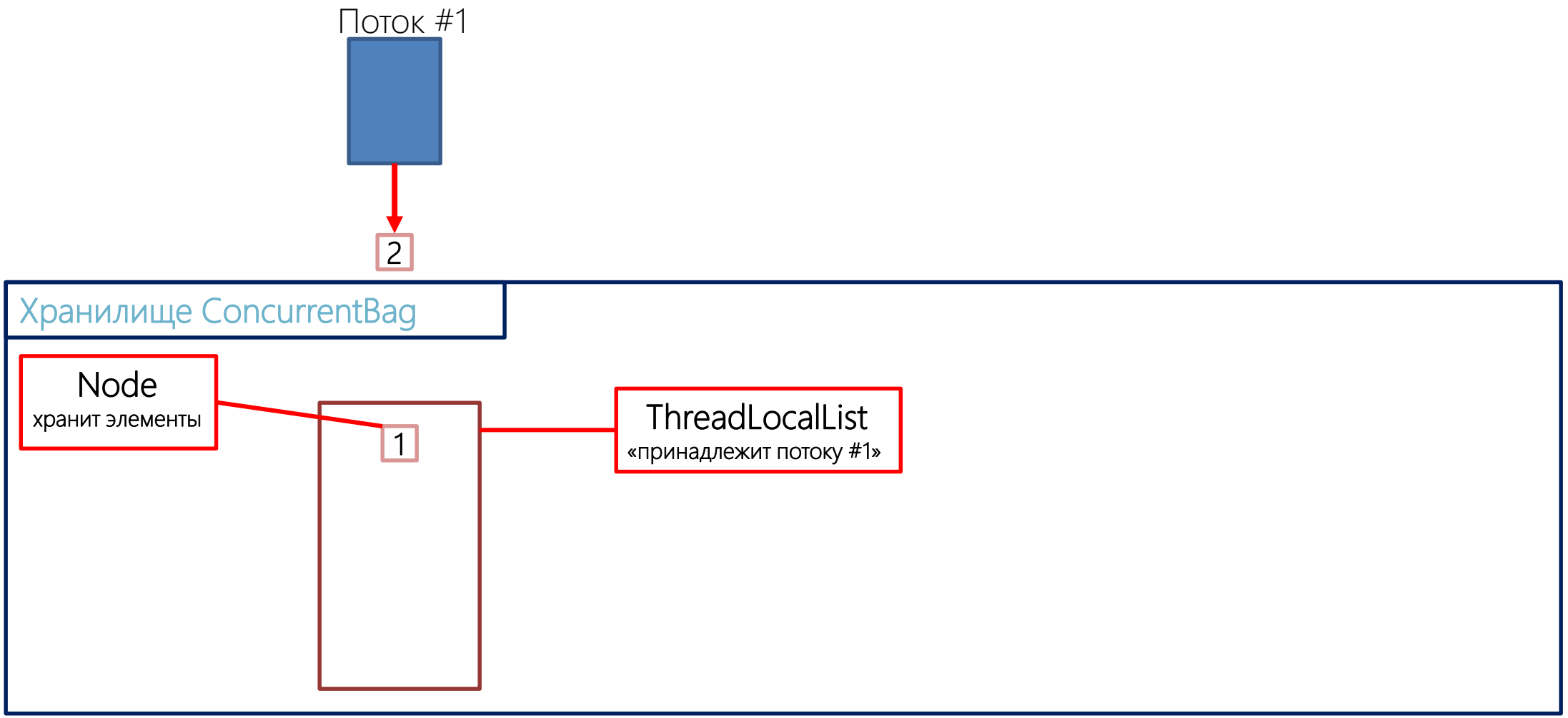
Добавление элементов в ConcurrentBag



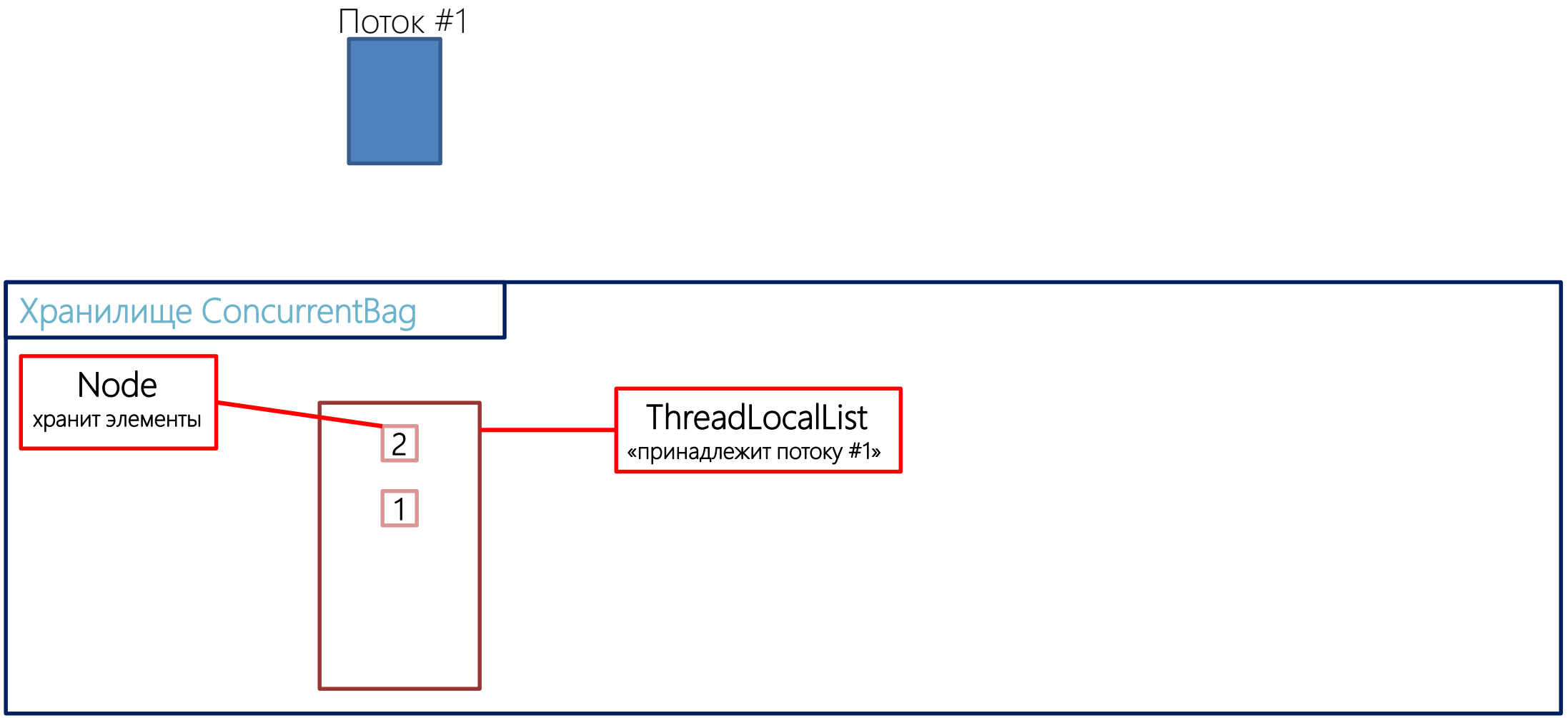
Добавление элементов в ConcurrentBag



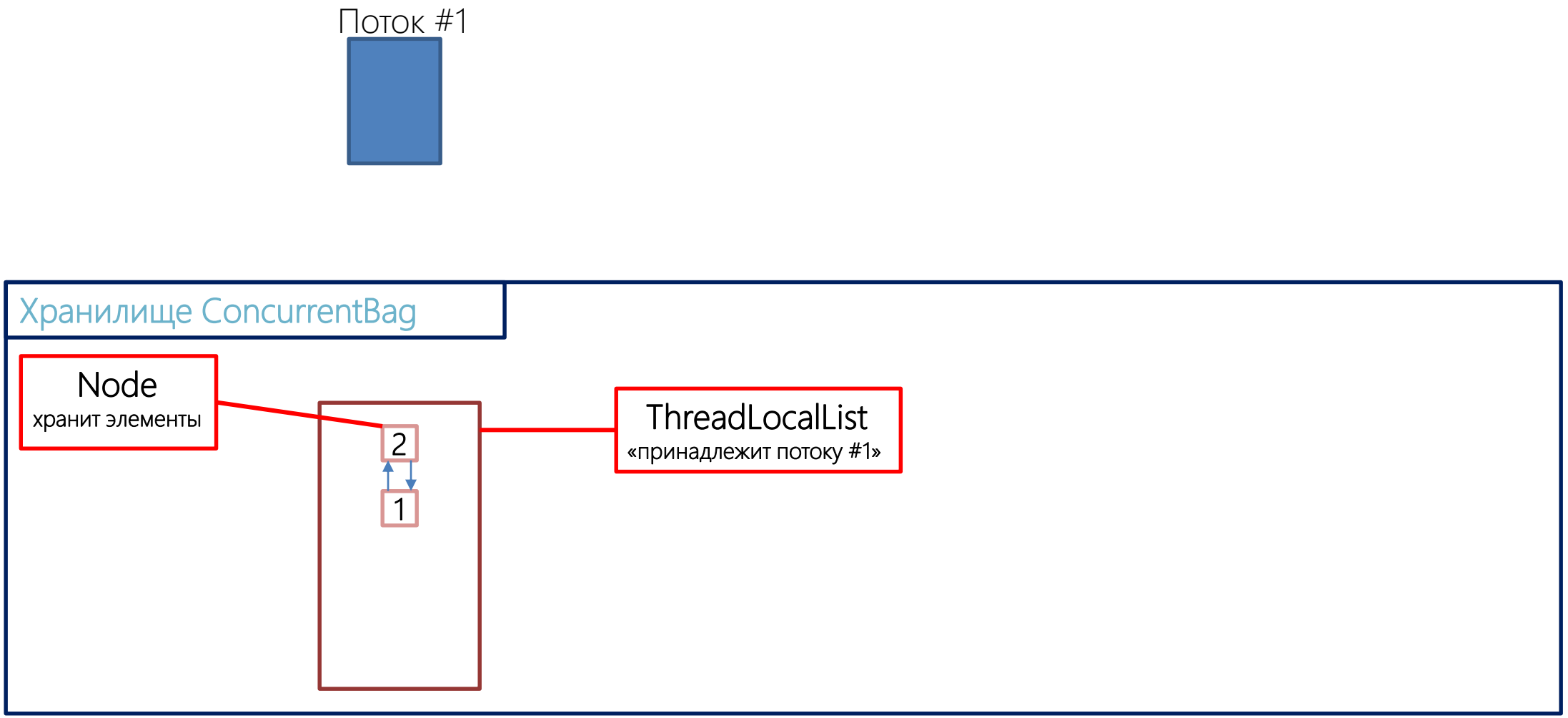
Добавление элементов в ConcurrentBag



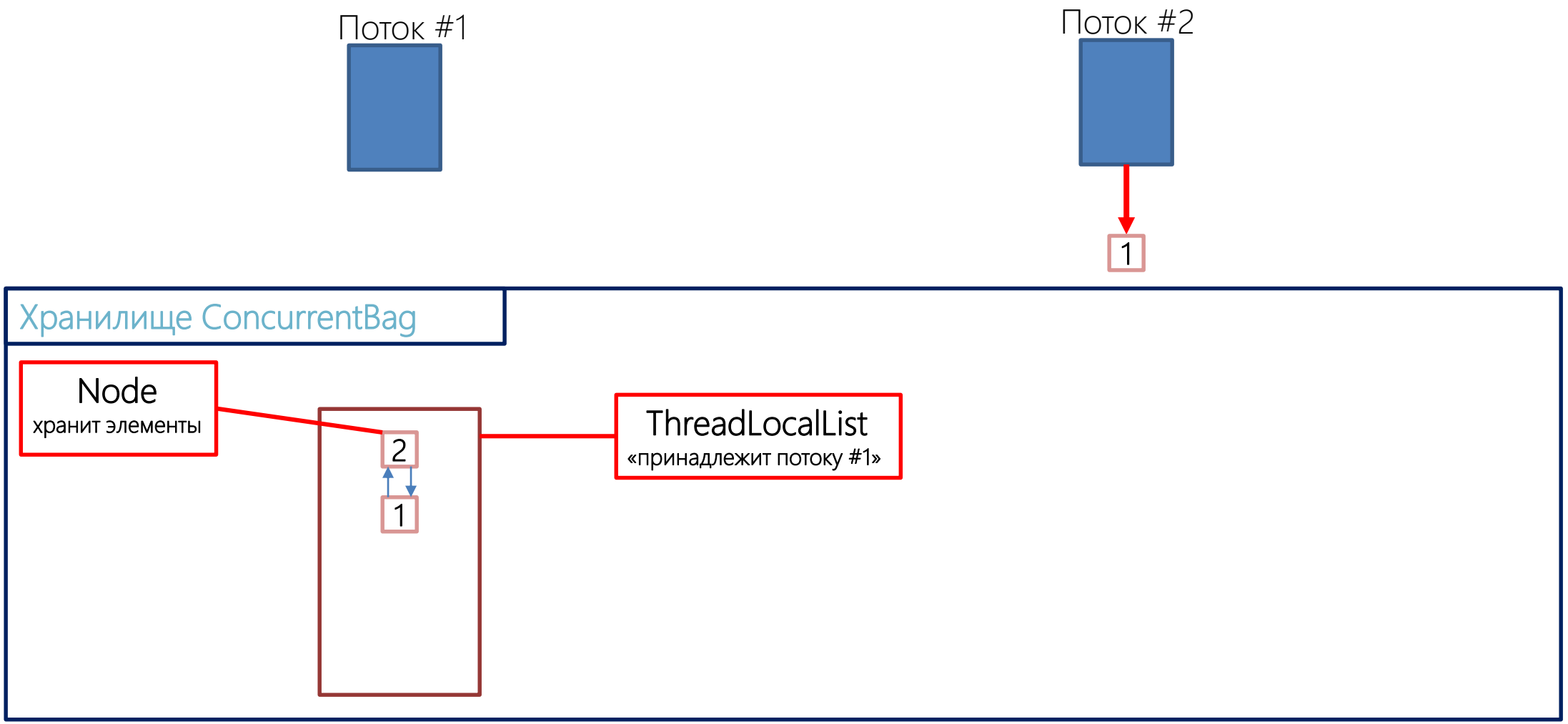
Добавление элементов в ConcurrentBag



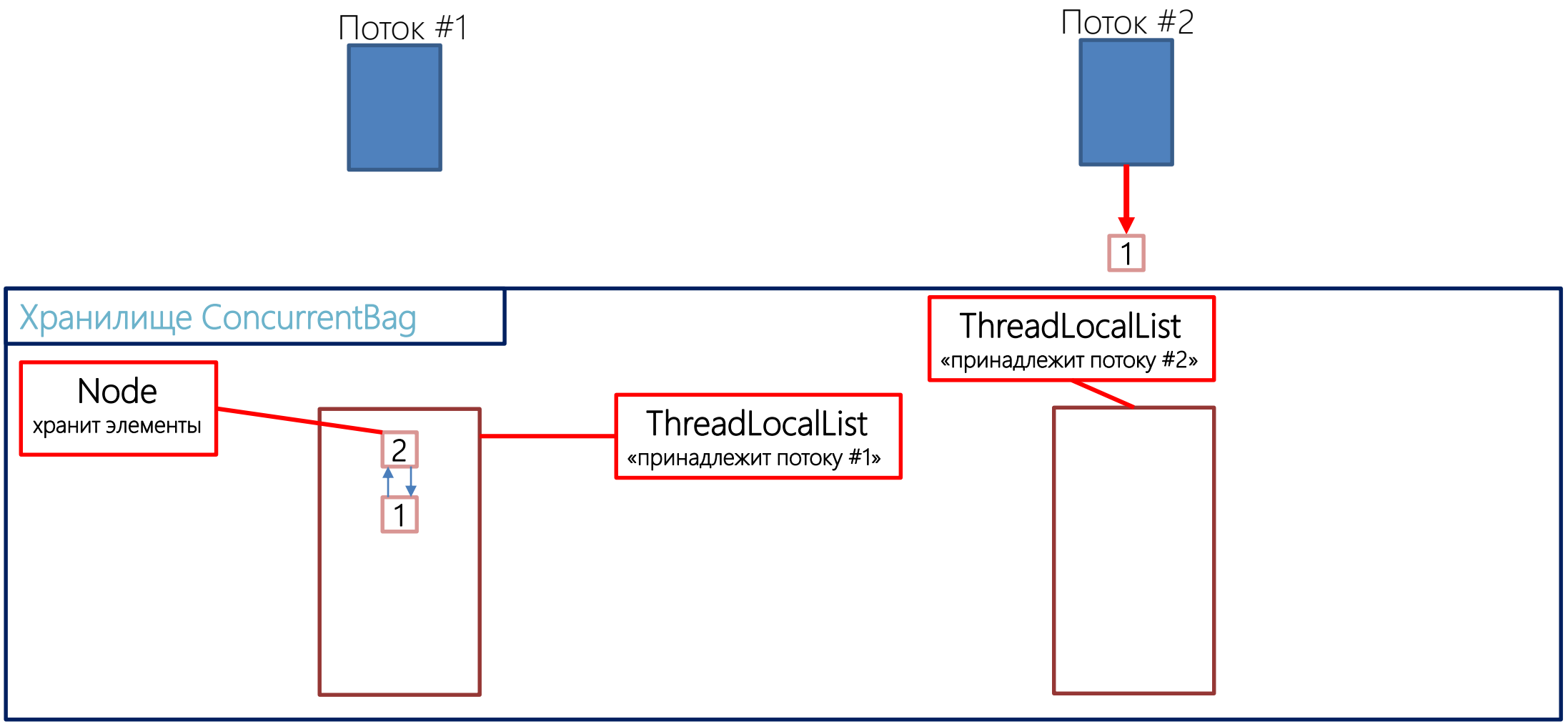
Добавление элементов в ConcurrentBag



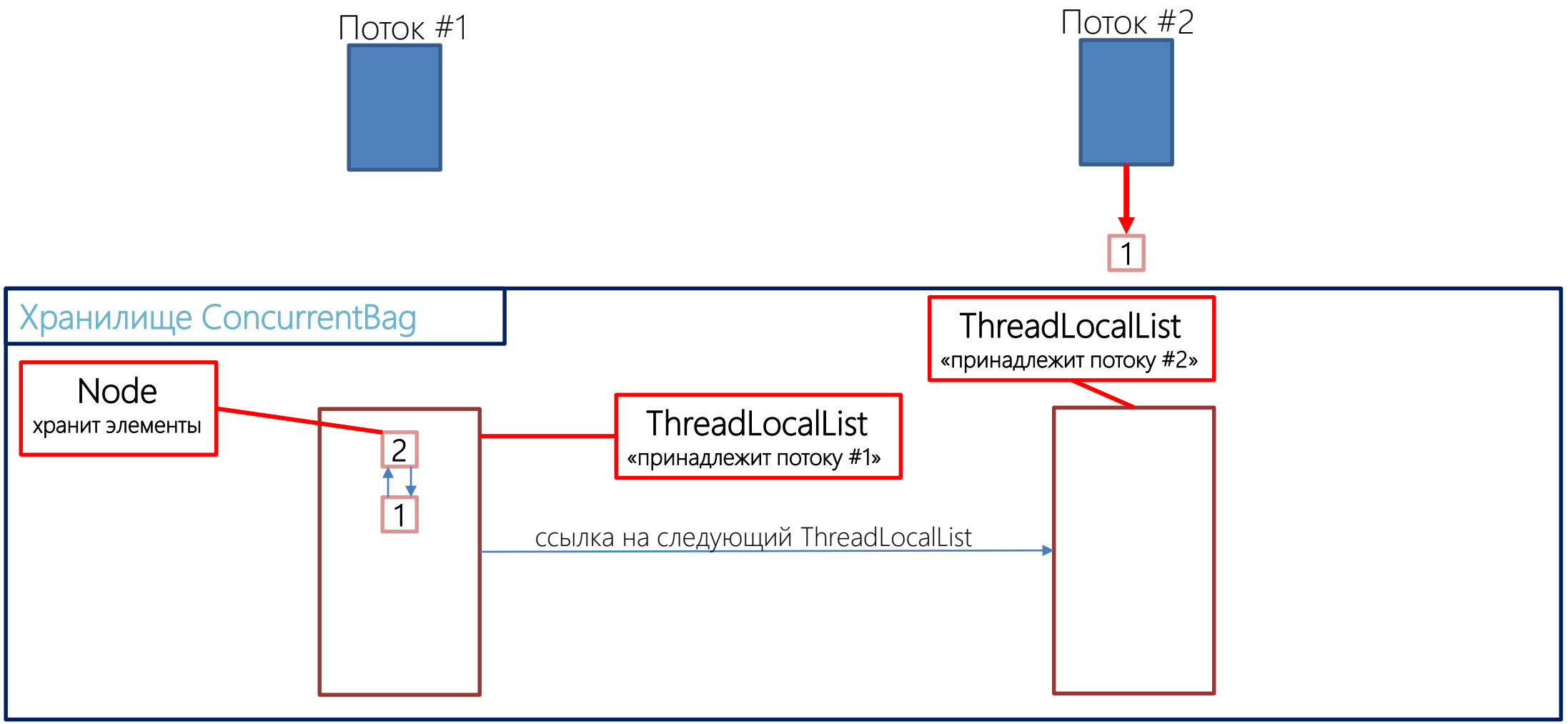
Добавление элементов в ConcurrentBag



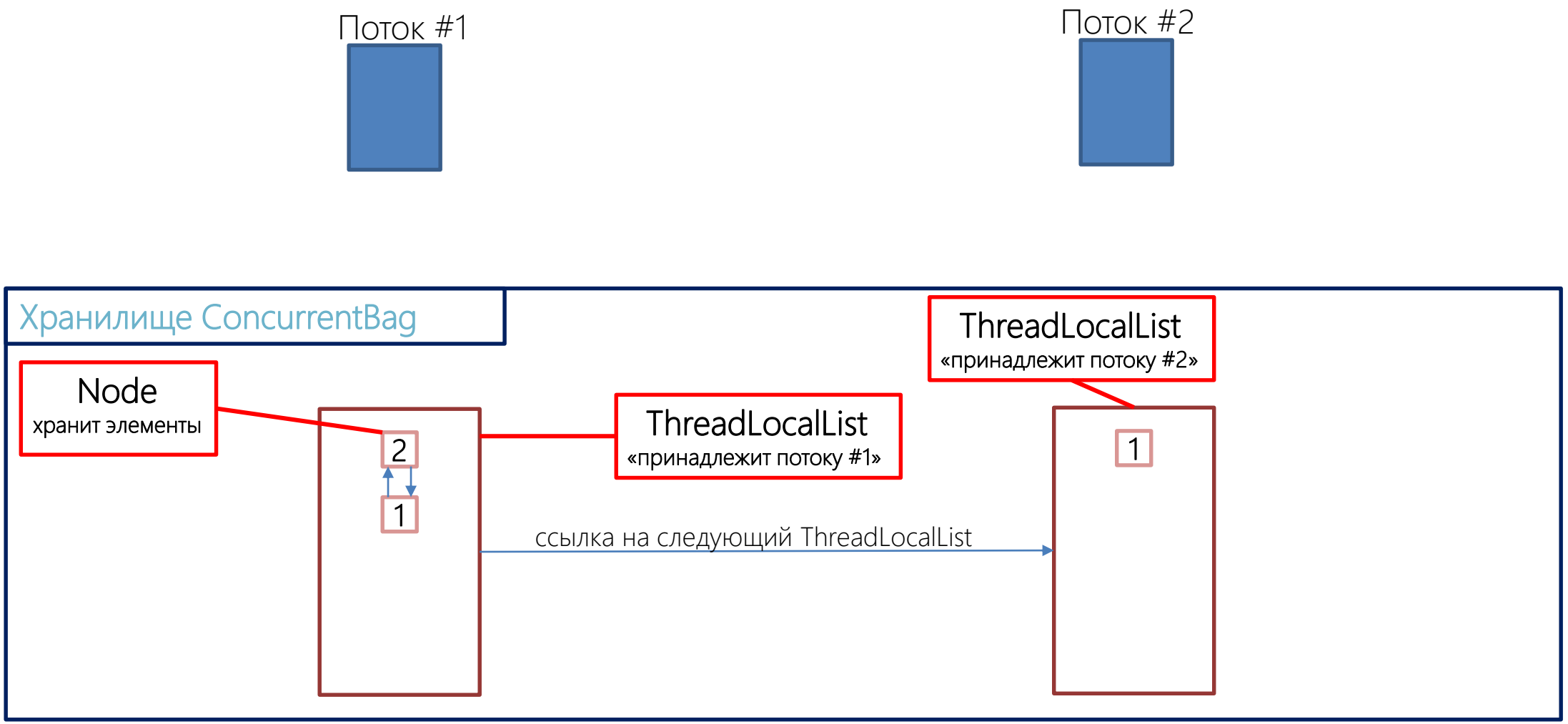
Добавление элементов в ConcurrentBag



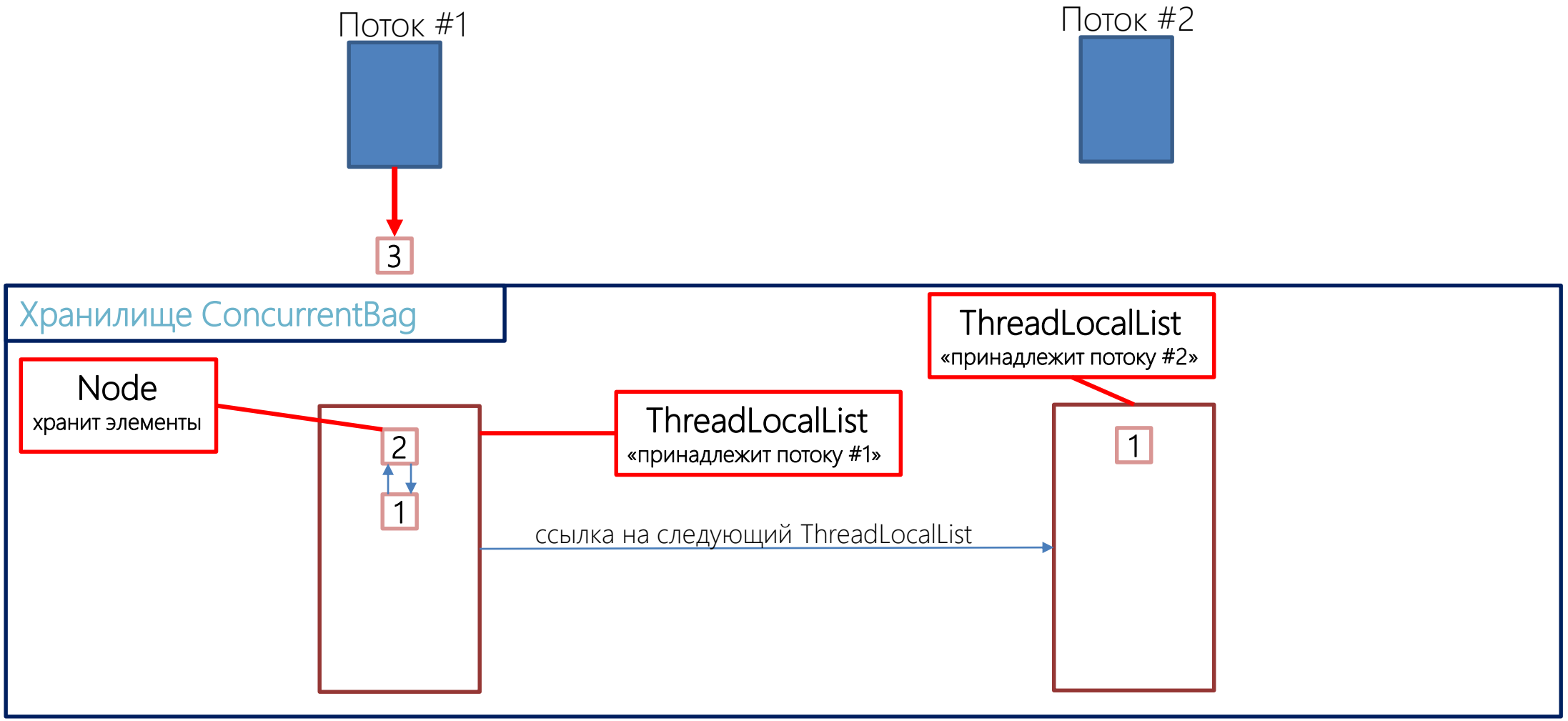
Добавление элементов в ConcurrentBag



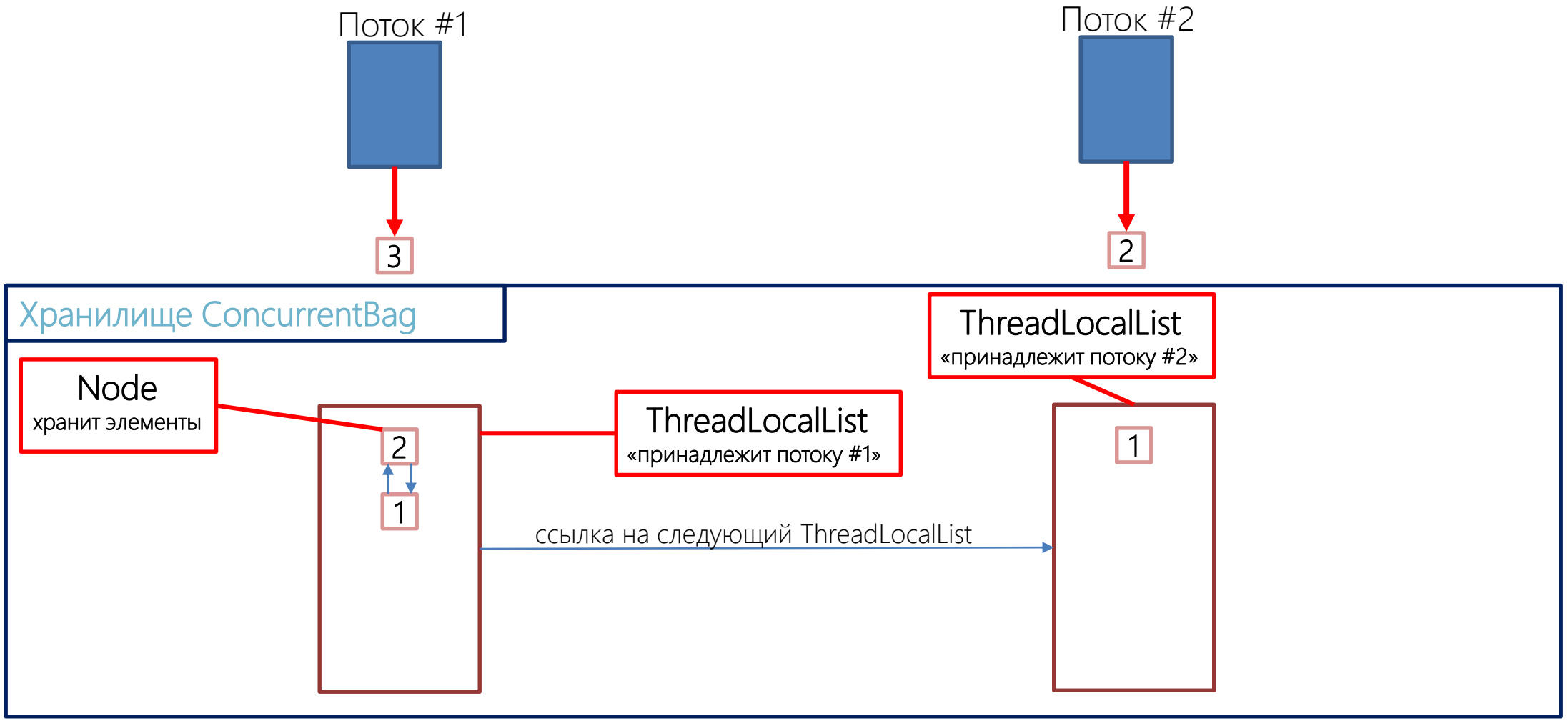
Добавление элементов в ConcurrentBag



Добавление элементов в ConcurrentBag



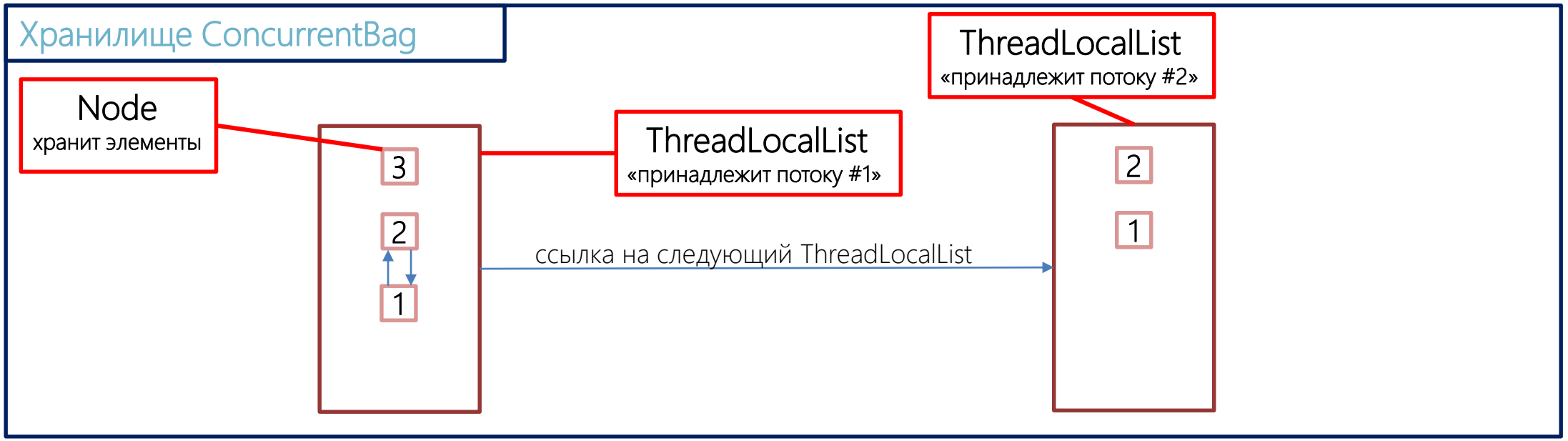
Добавление элементов в ConcurrentBag



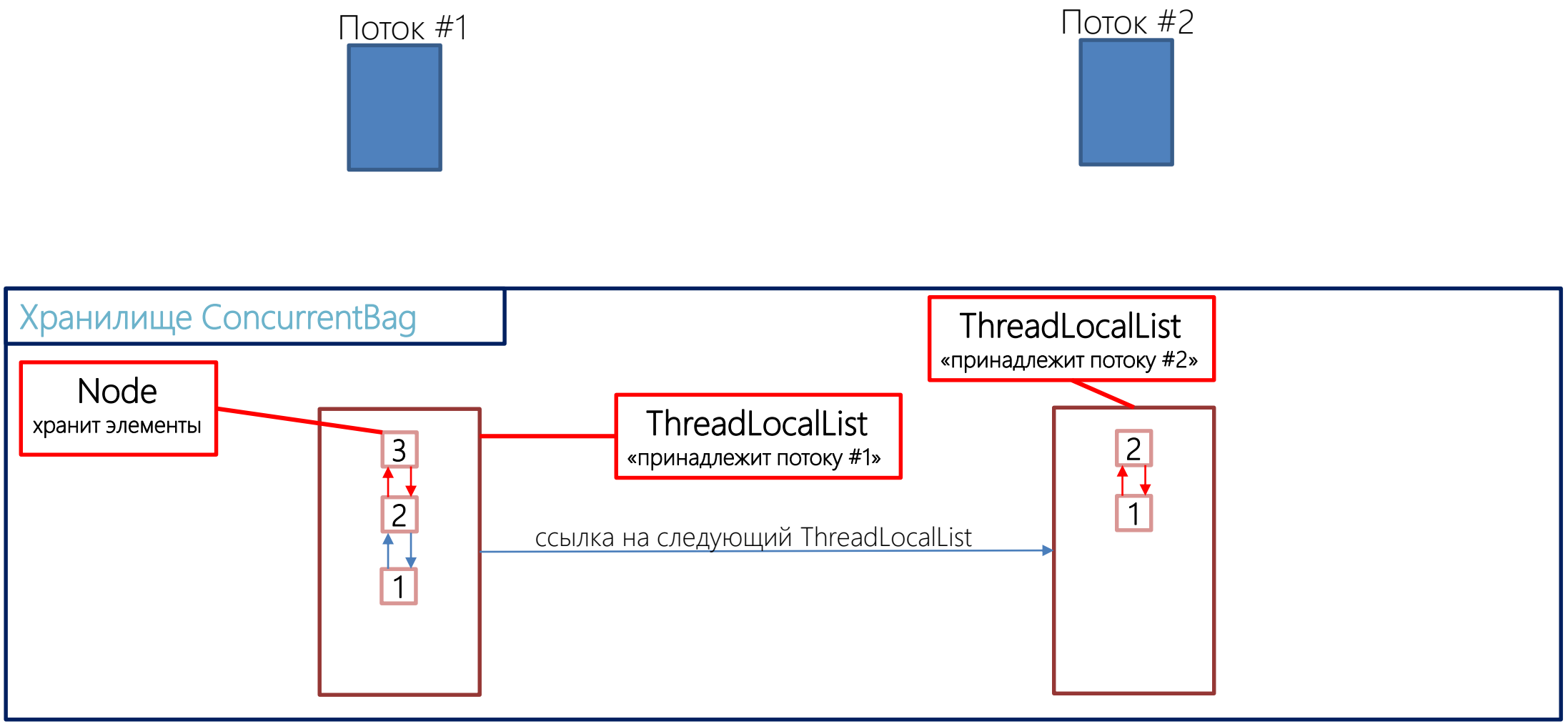
Добавление элементов в ConcurrentBag

Поток #1

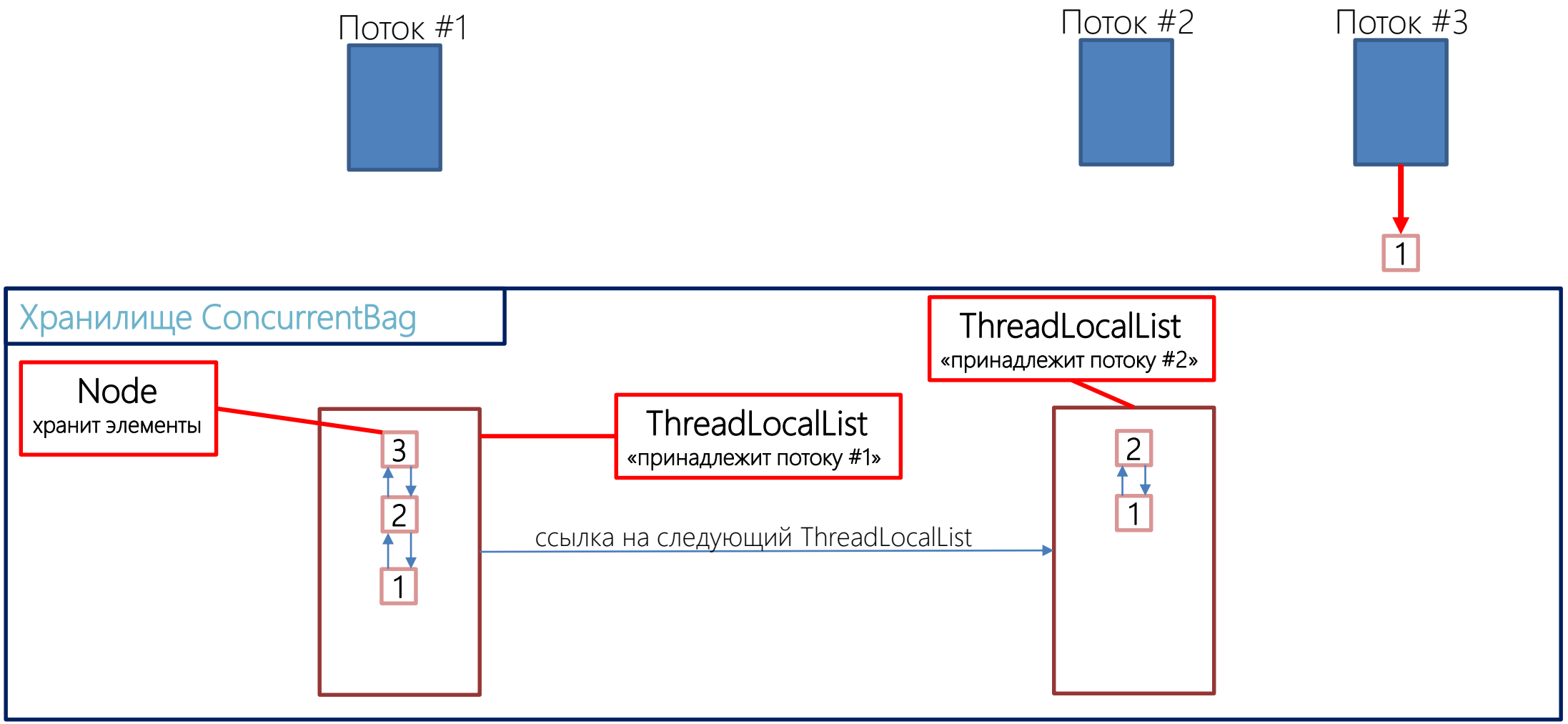
Поток #2



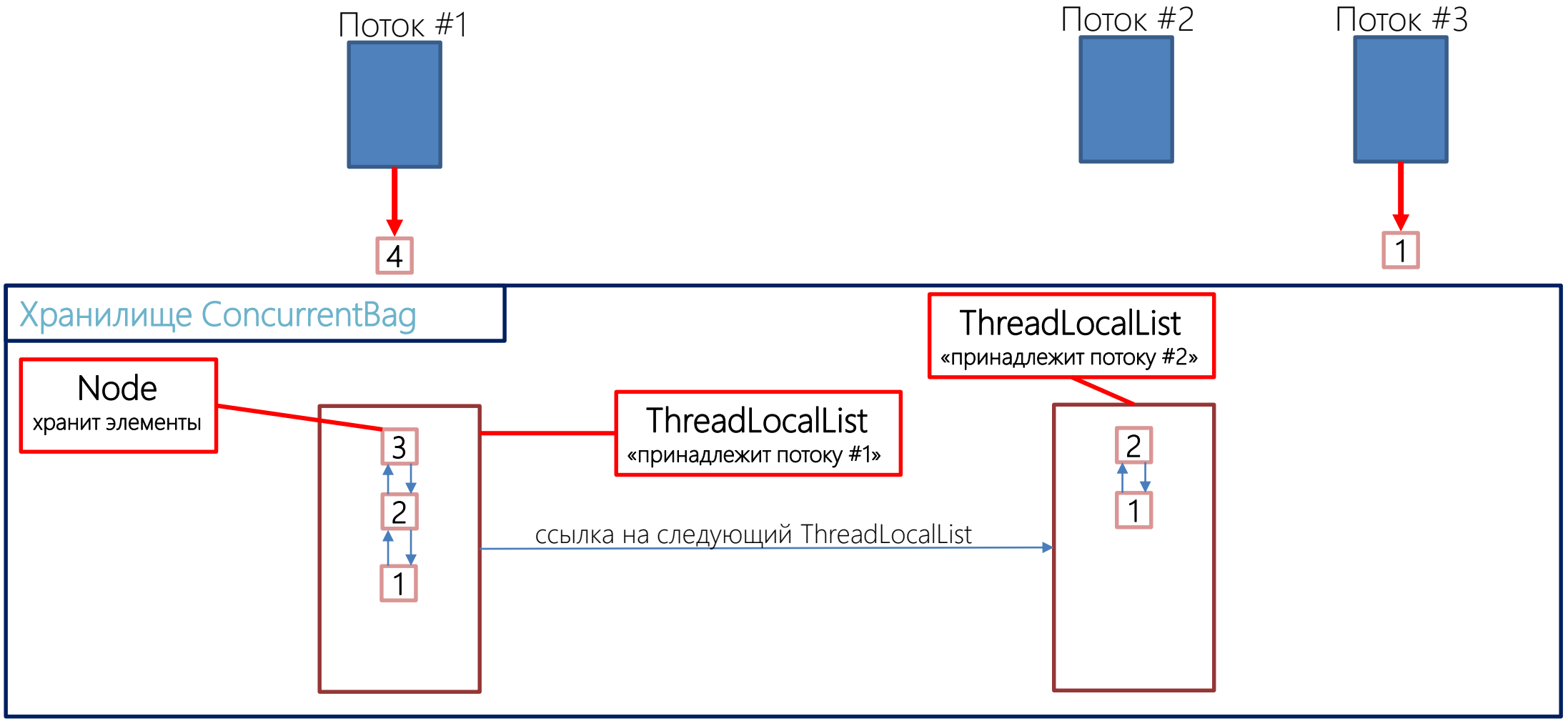
Добавление элементов в ConcurrentBag



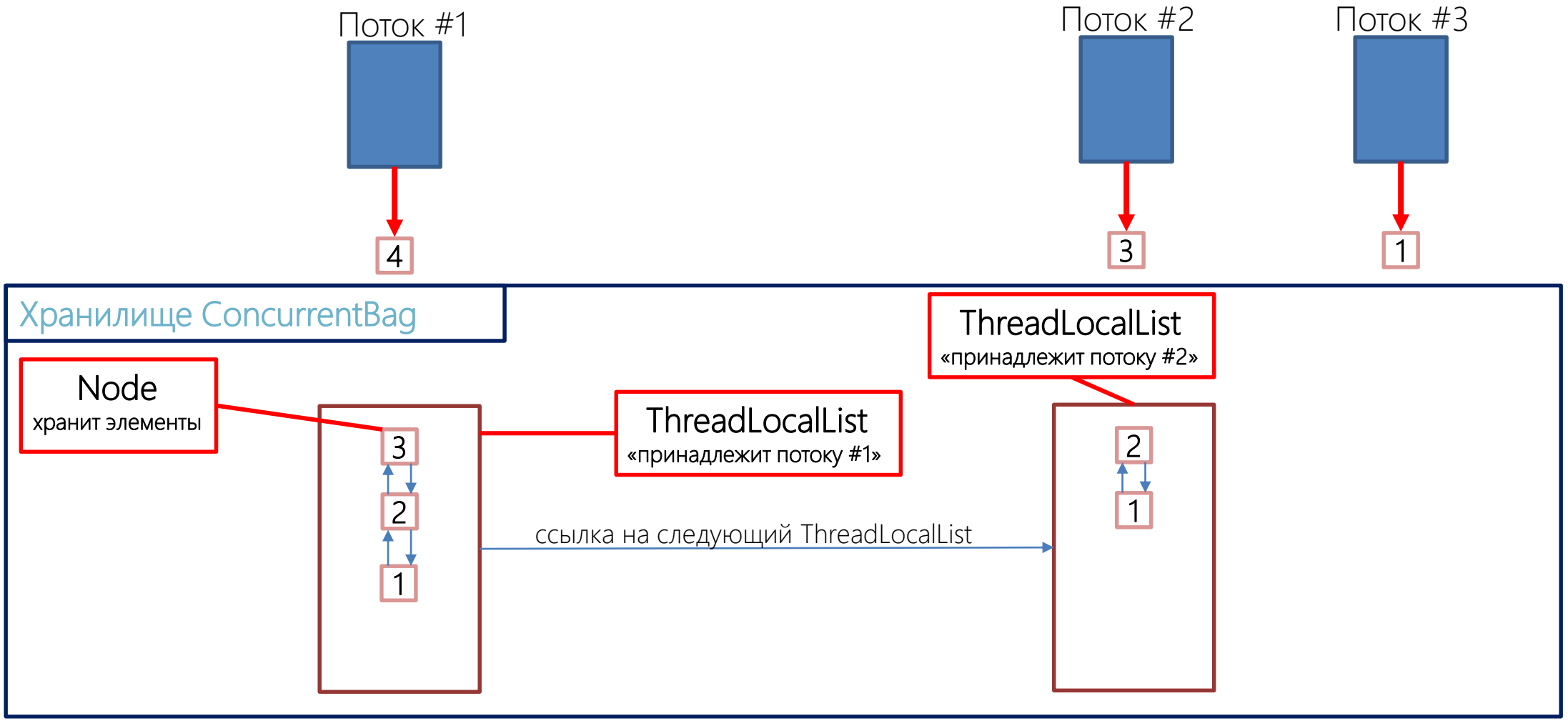
Добавление элементов в ConcurrentBag



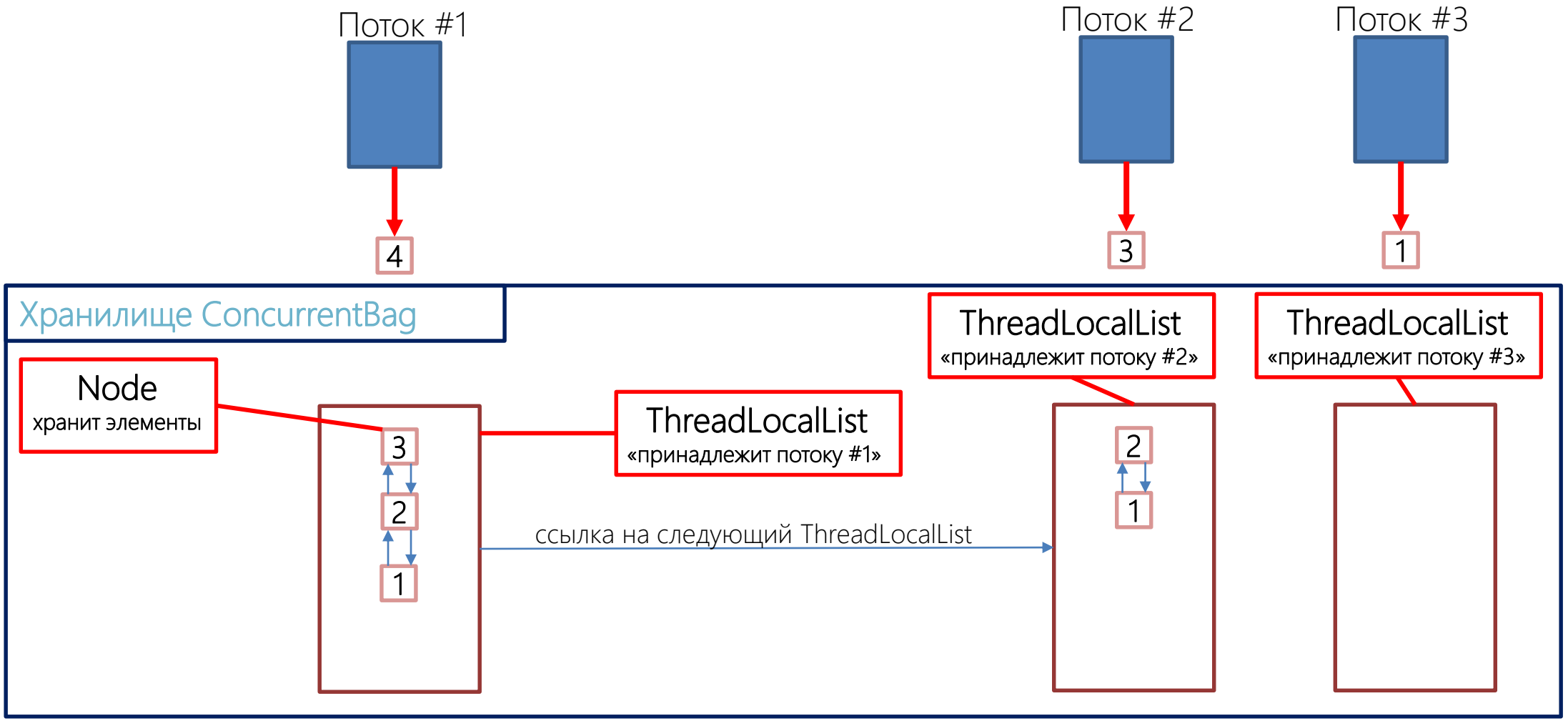
Добавление элементов в ConcurrentBag



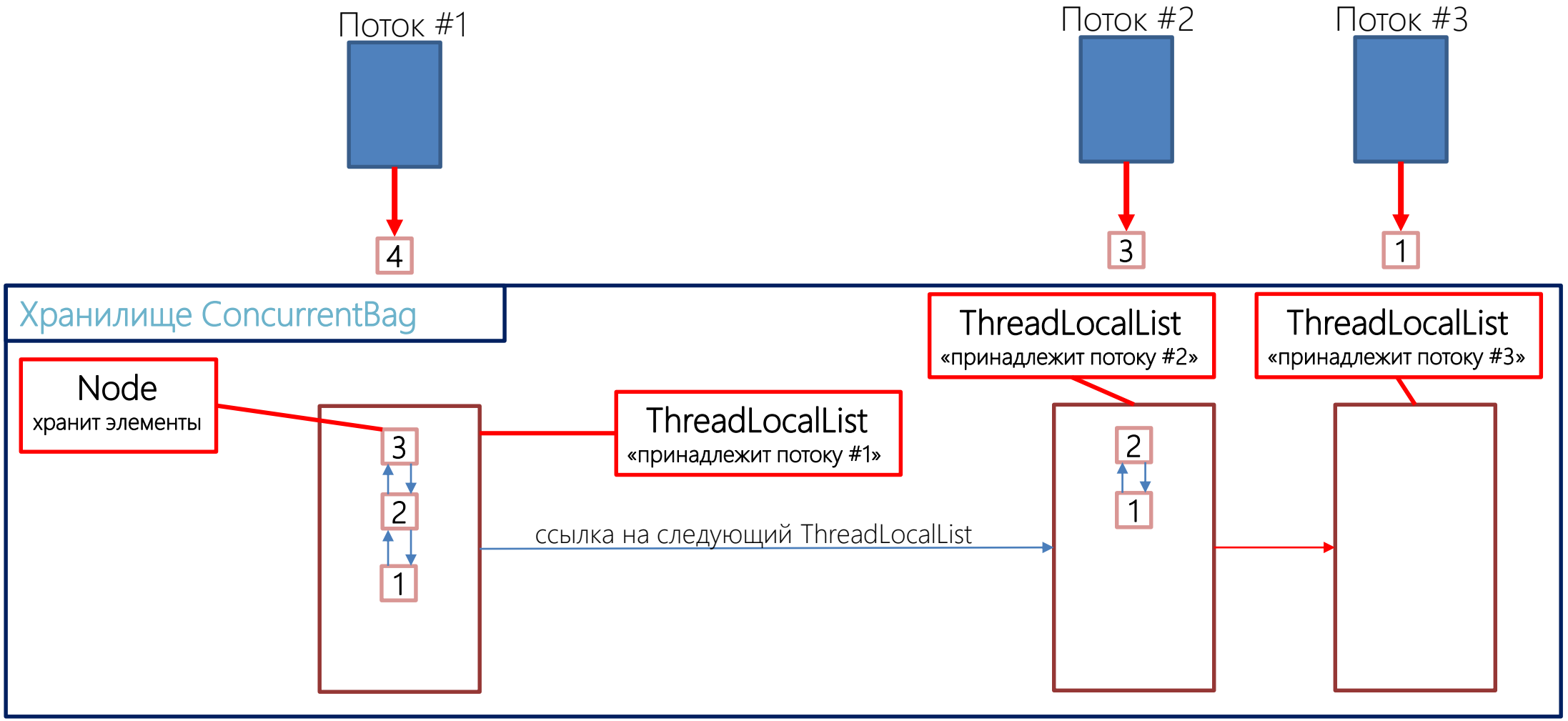
Добавление элементов в ConcurrentBag



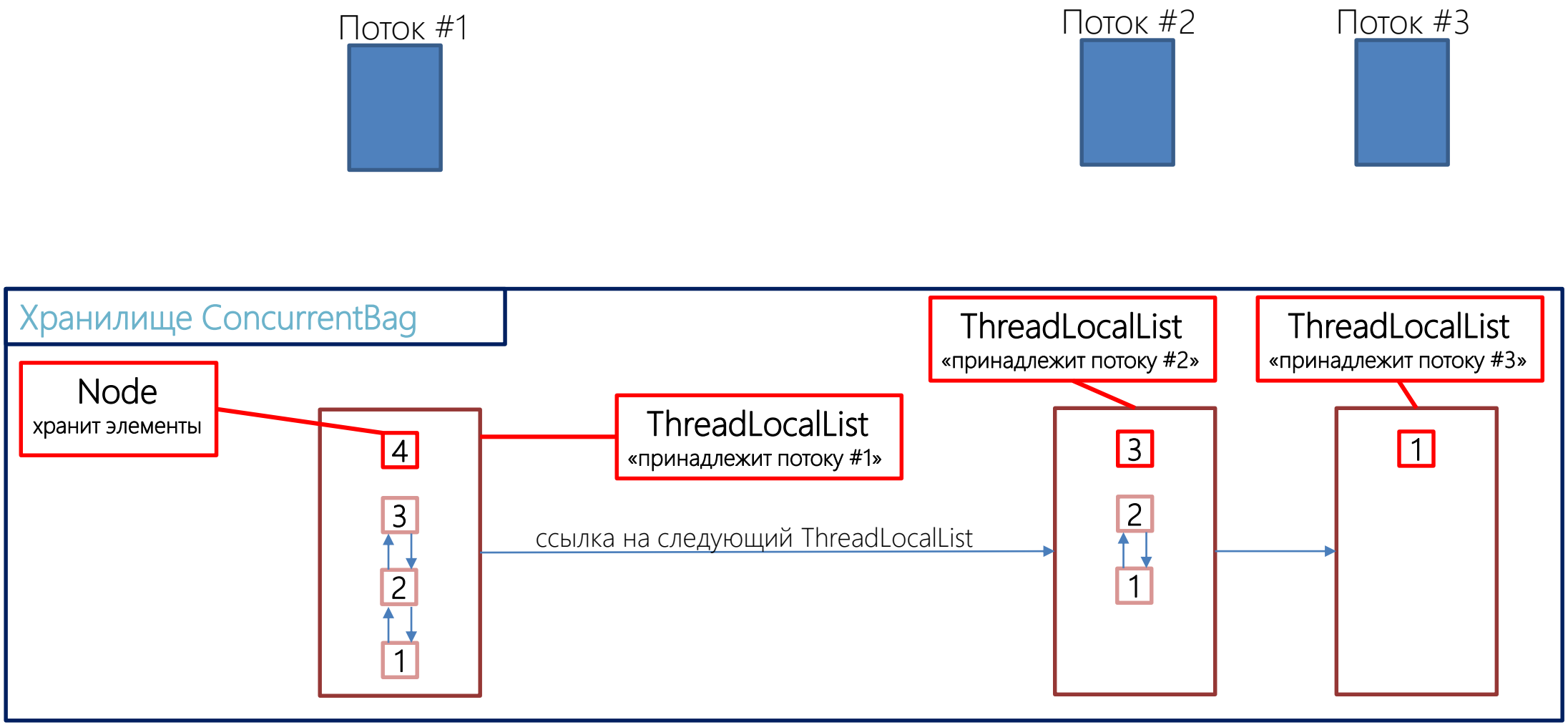
Добавление элементов в ConcurrentBag



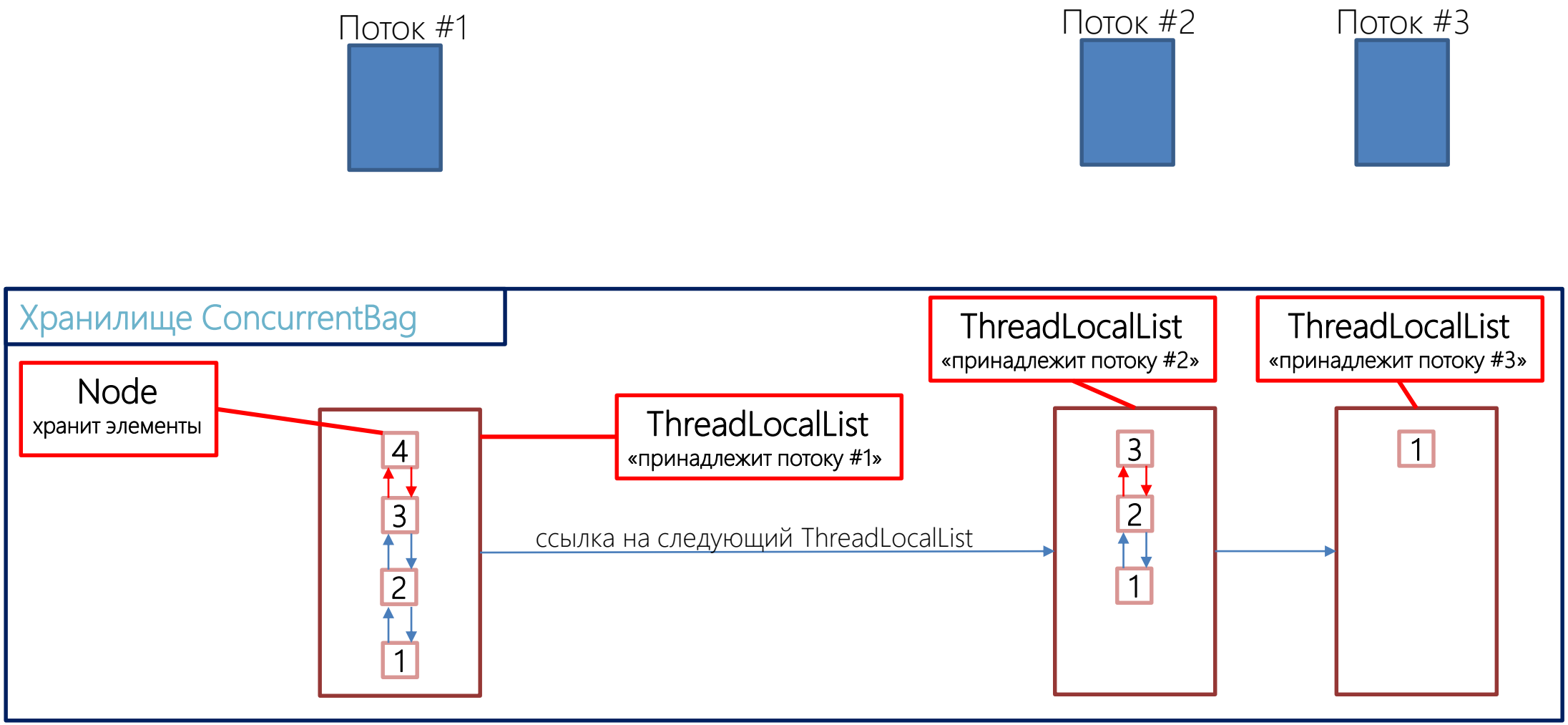
Добавление элементов в ConcurrentBag



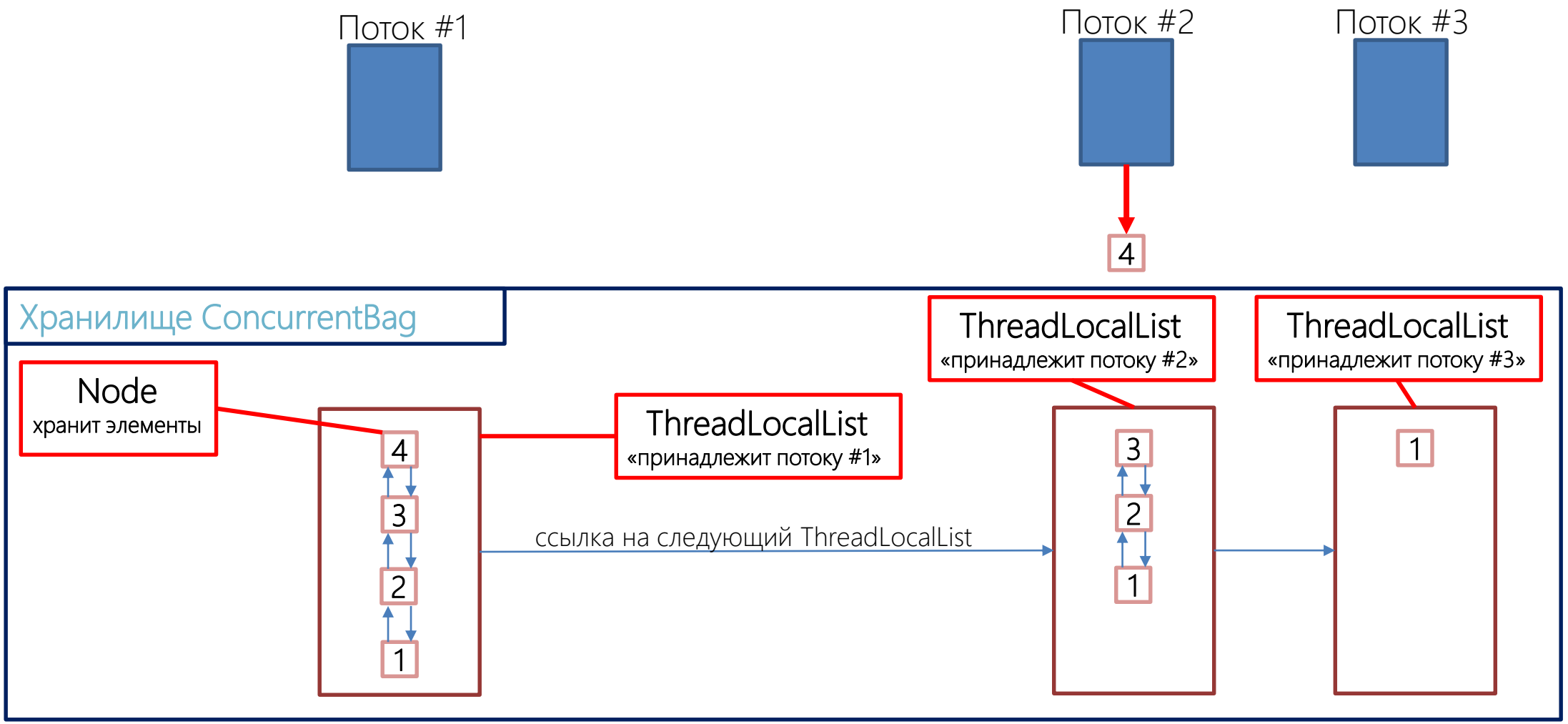
Добавление элементов в ConcurrentBag



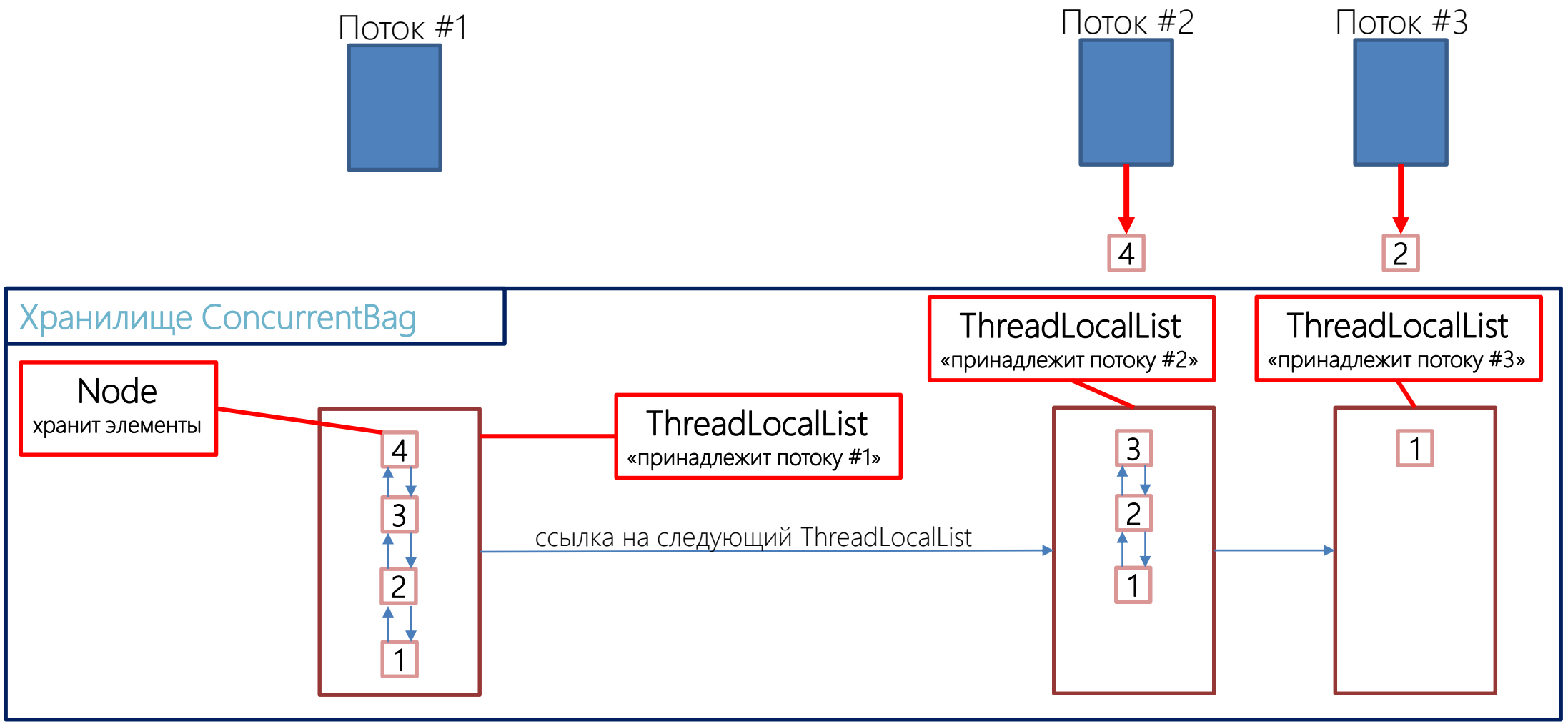
Добавление элементов в ConcurrentBag



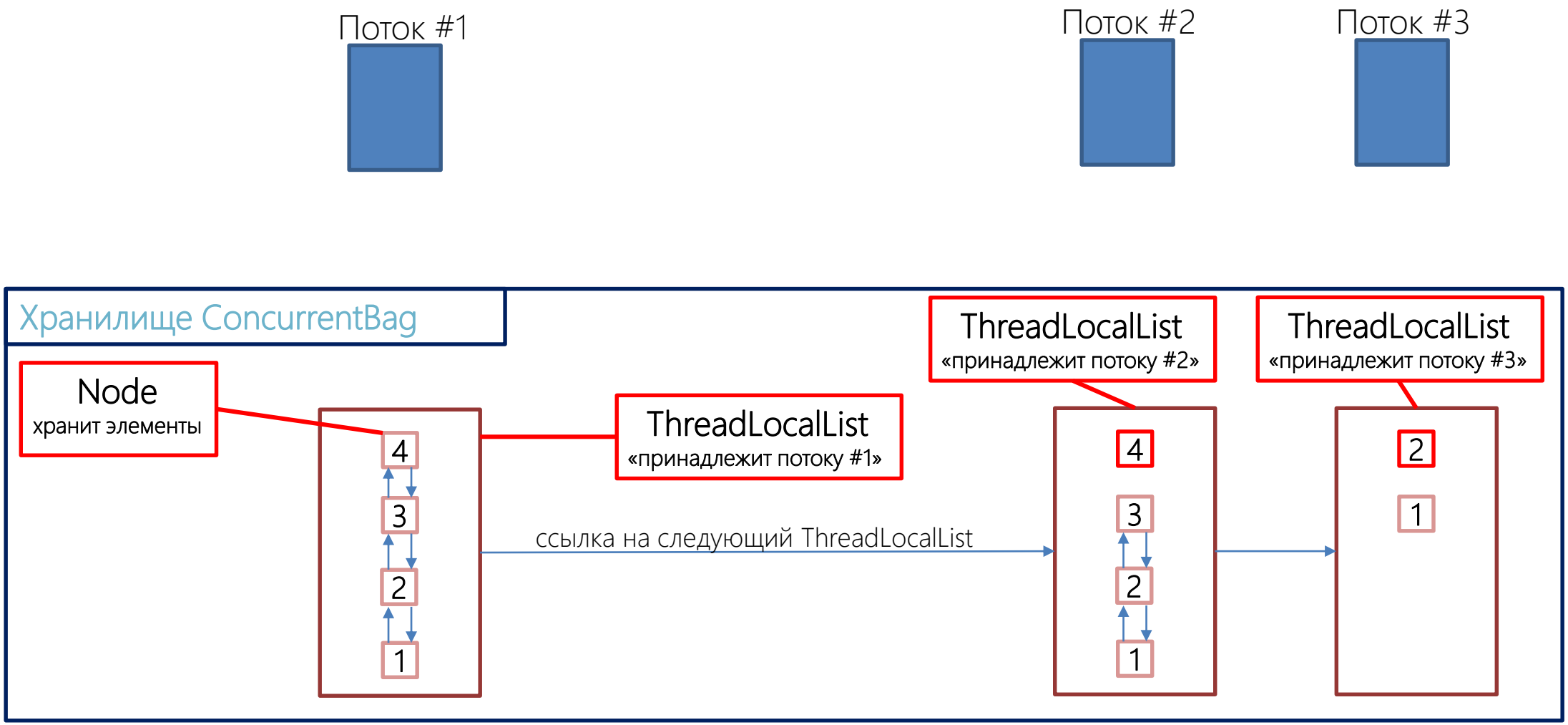
Добавление элементов в ConcurrentBag



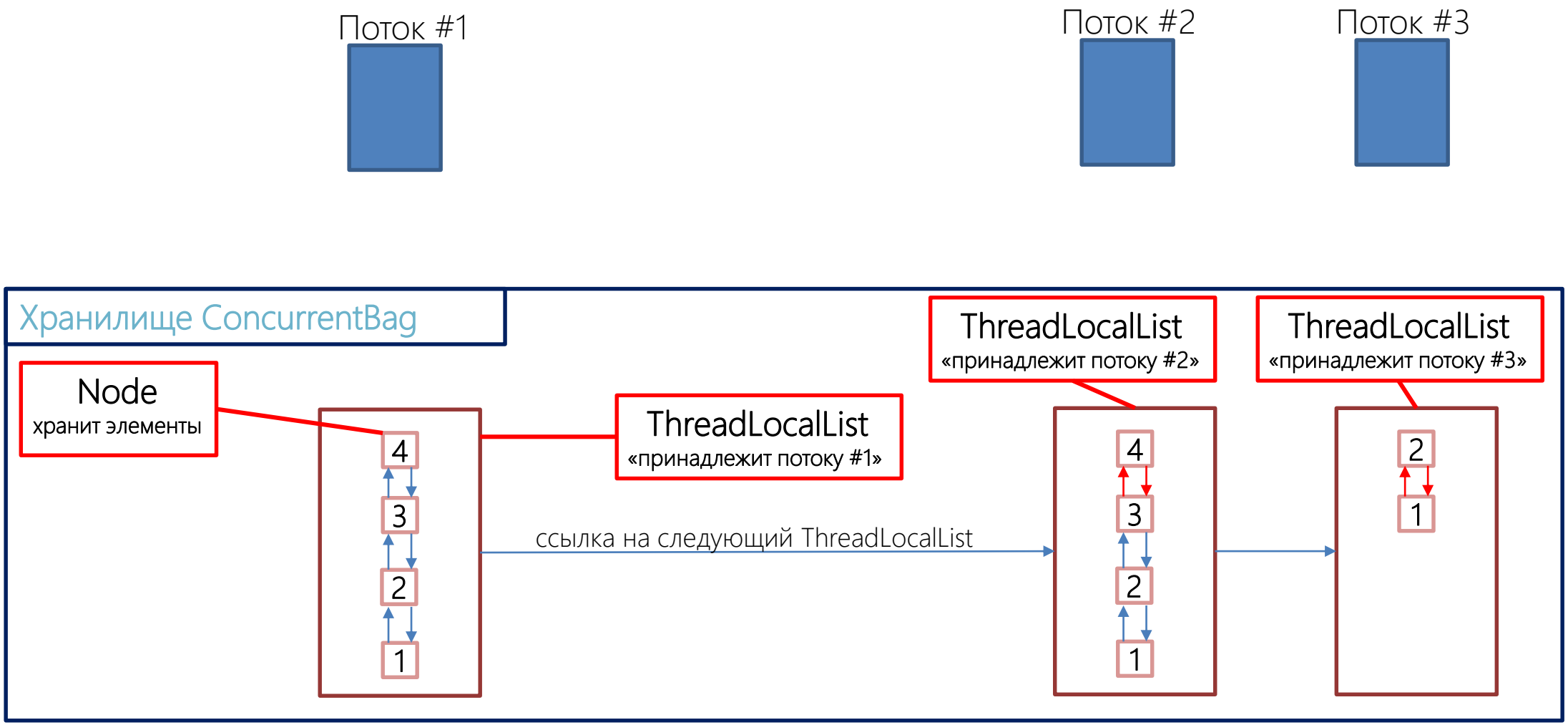
Добавление элементов в ConcurrentBag



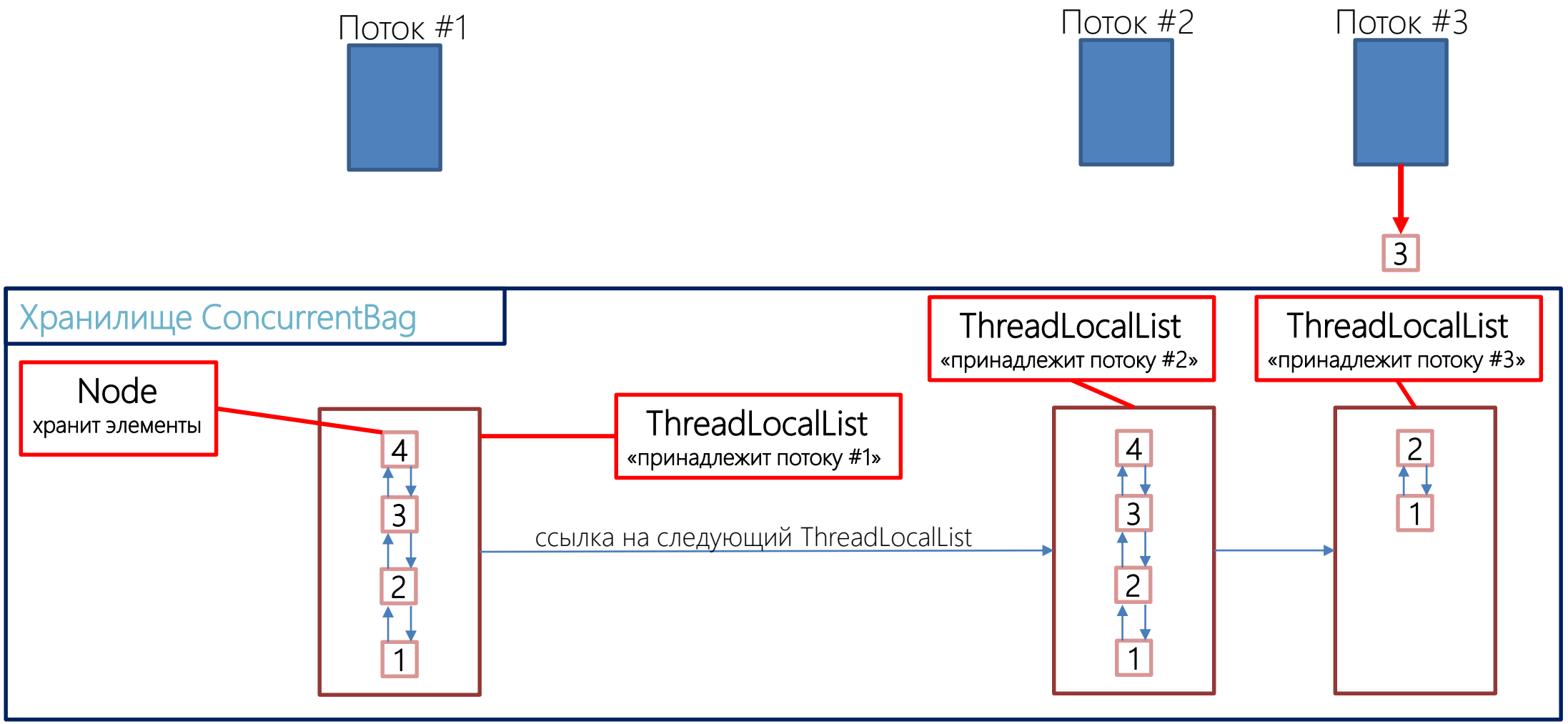
Добавление элементов в ConcurrentBag



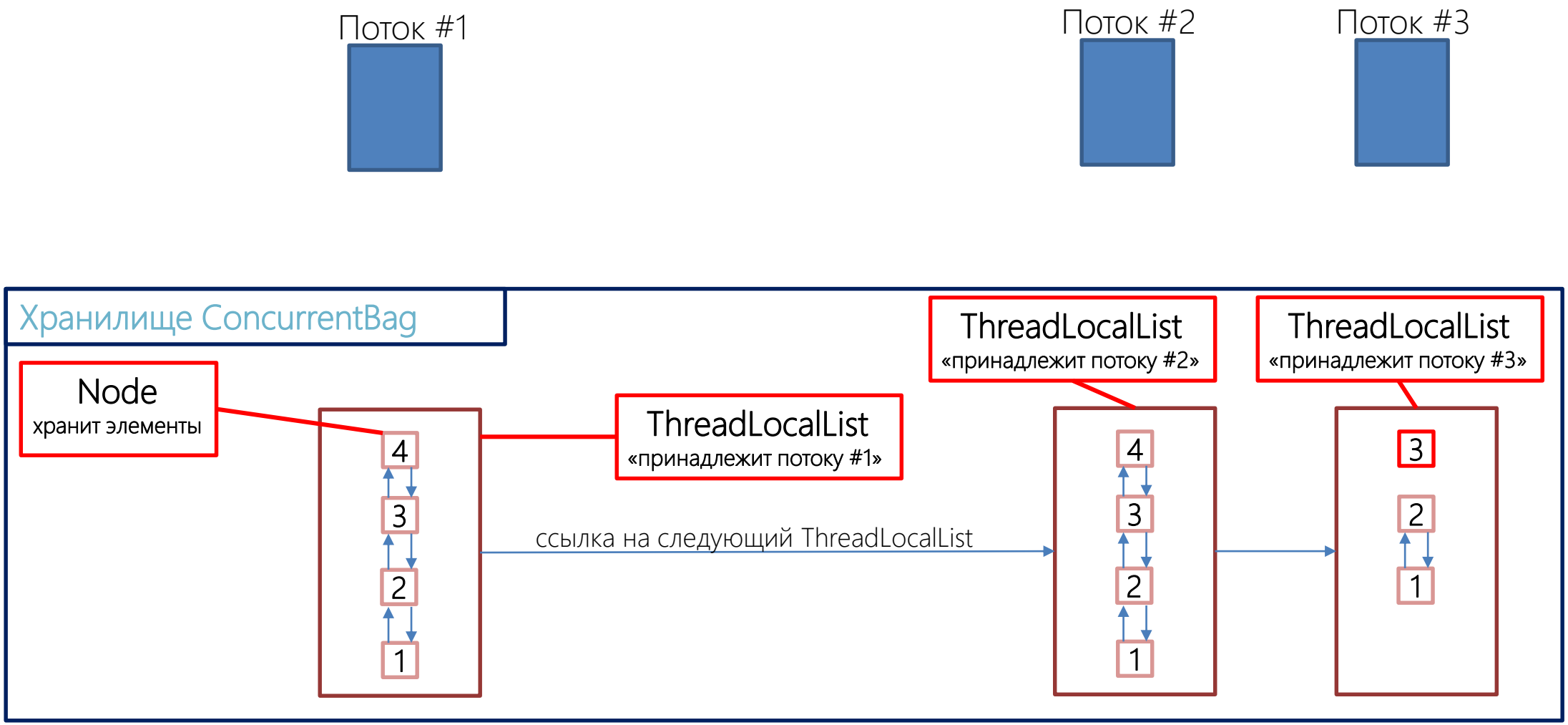
Добавление элементов в ConcurrentBag



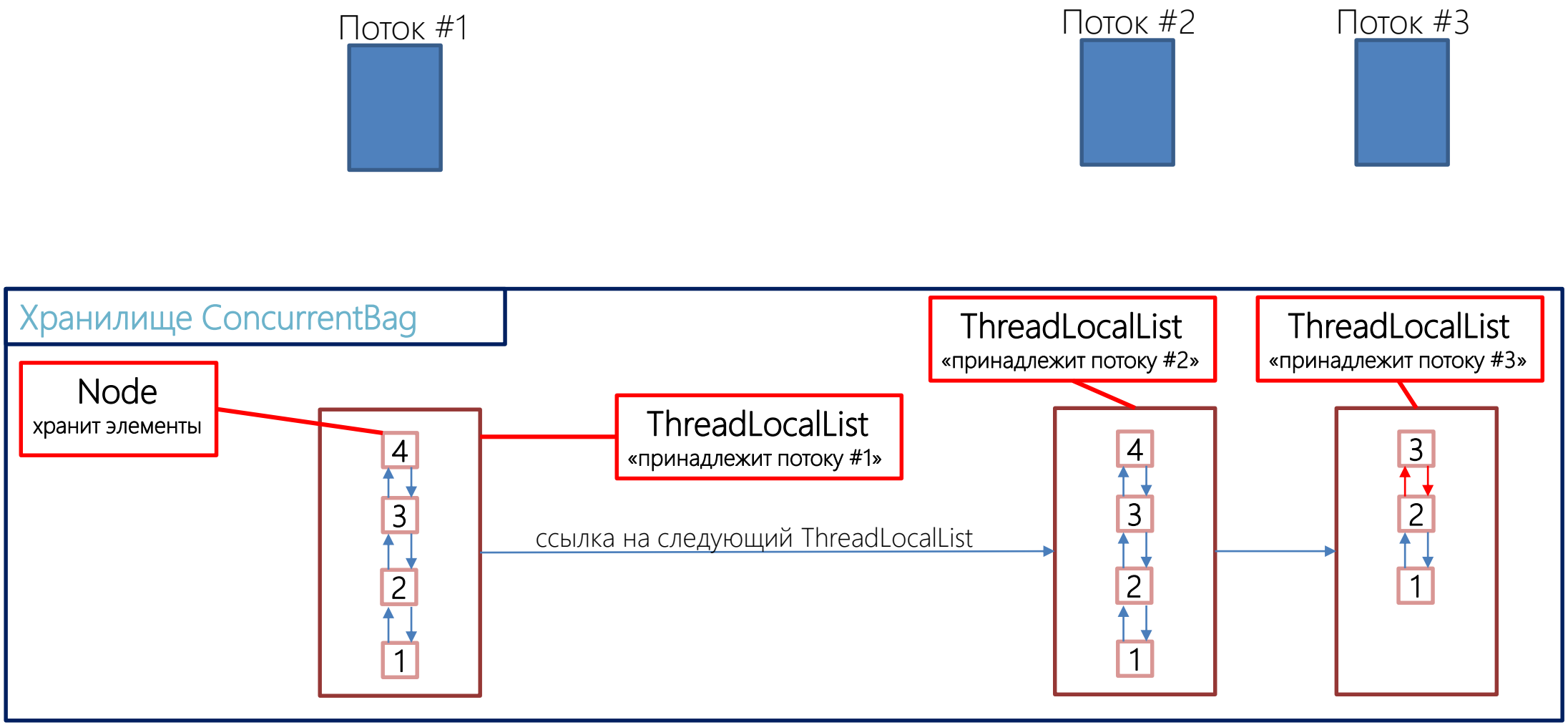
Добавление элементов в ConcurrentBag



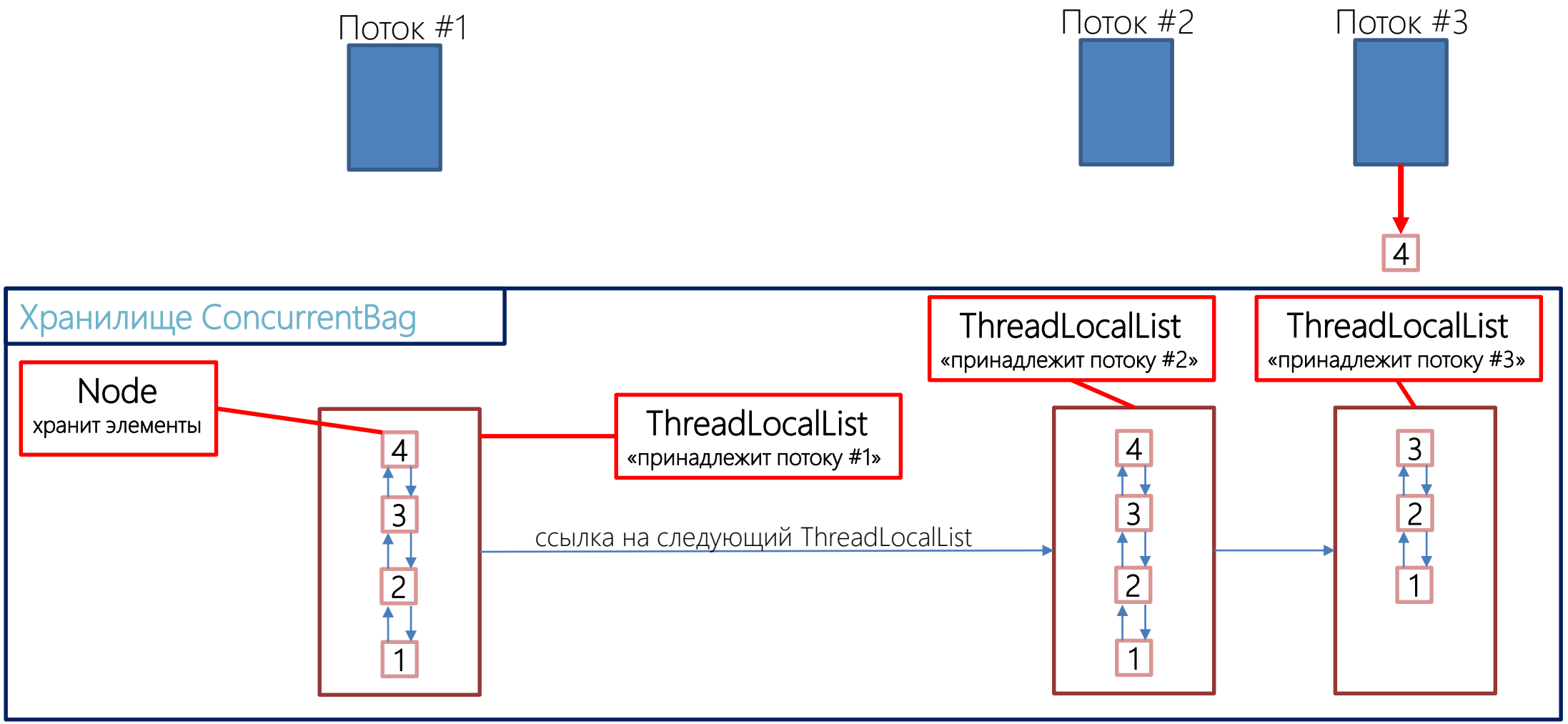
Добавление элементов в ConcurrentBag



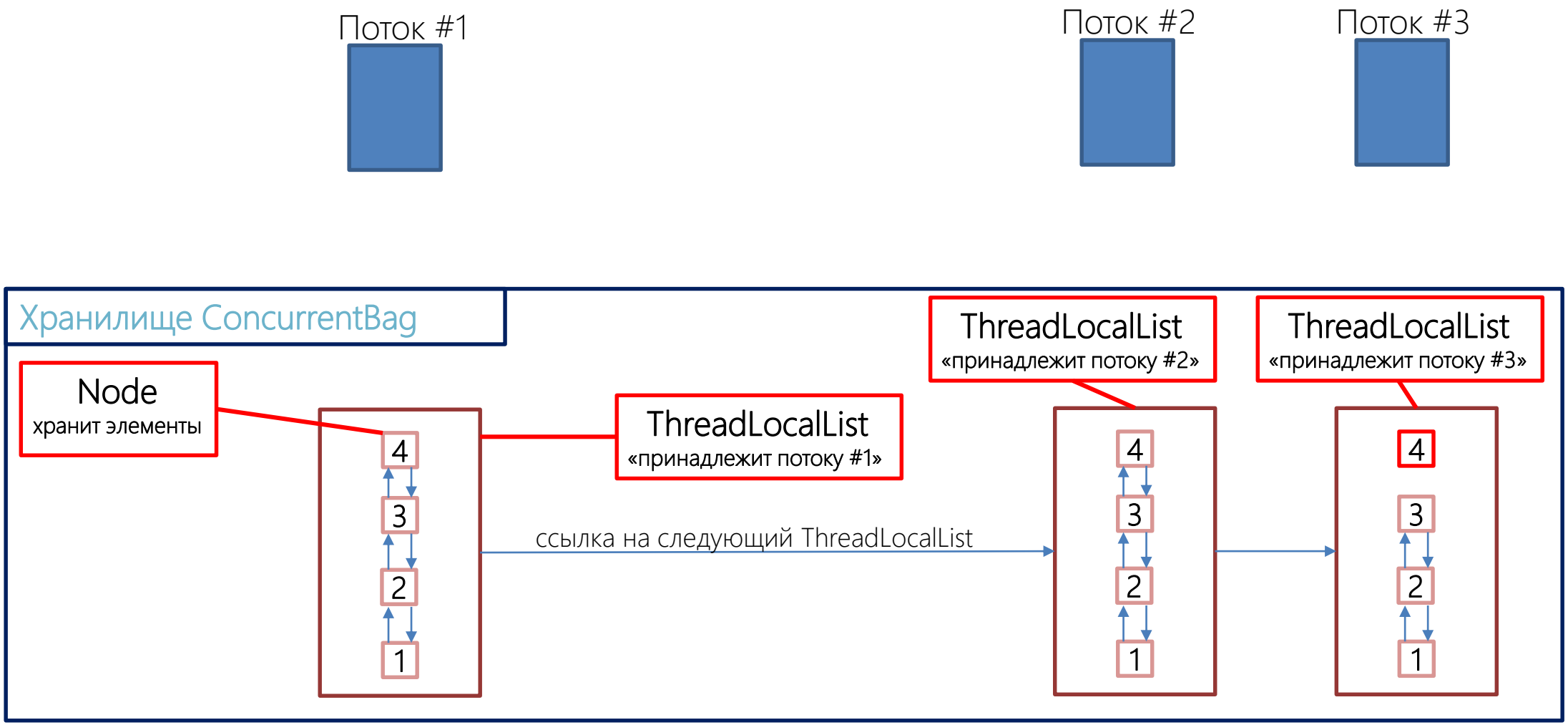
Добавление элементов в ConcurrentBag



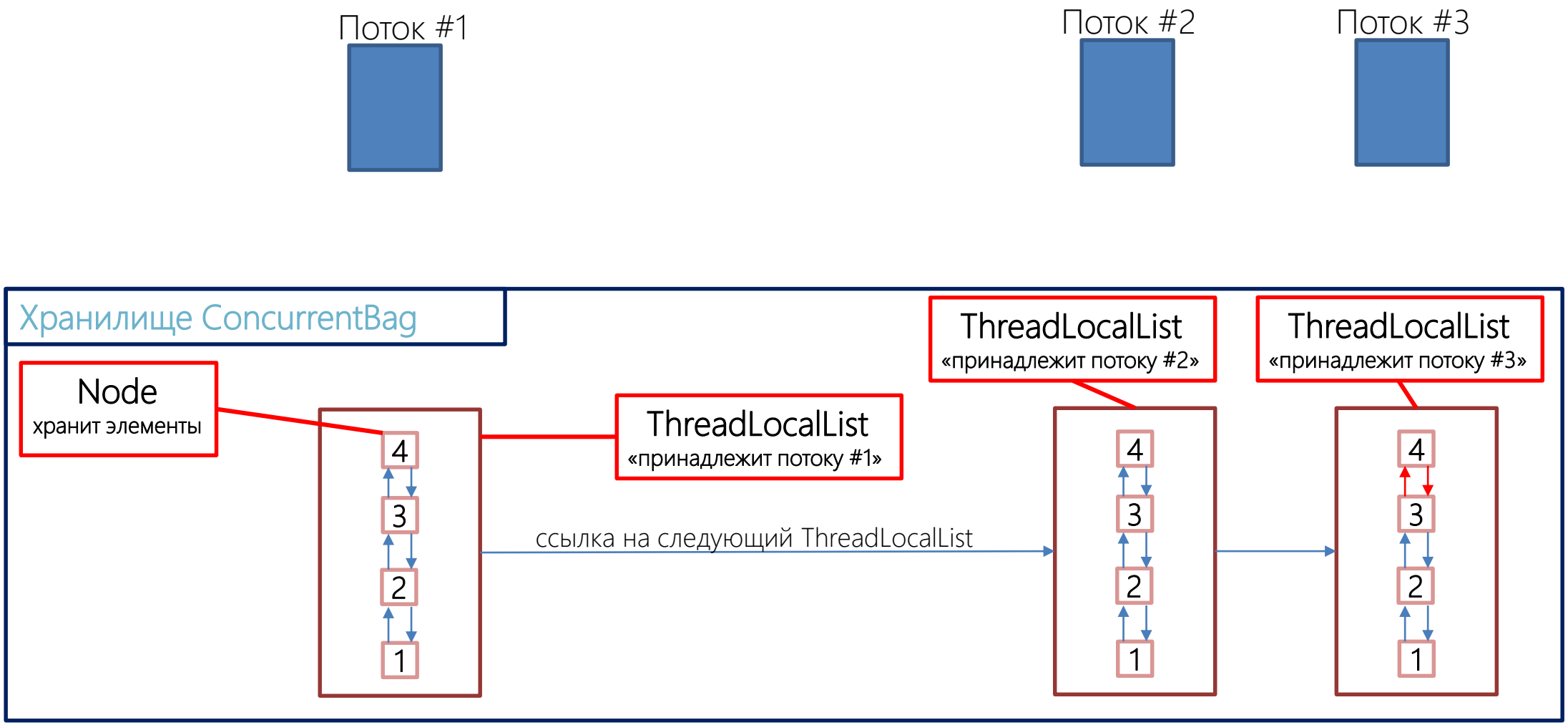
Добавление элементов в ConcurrentBag



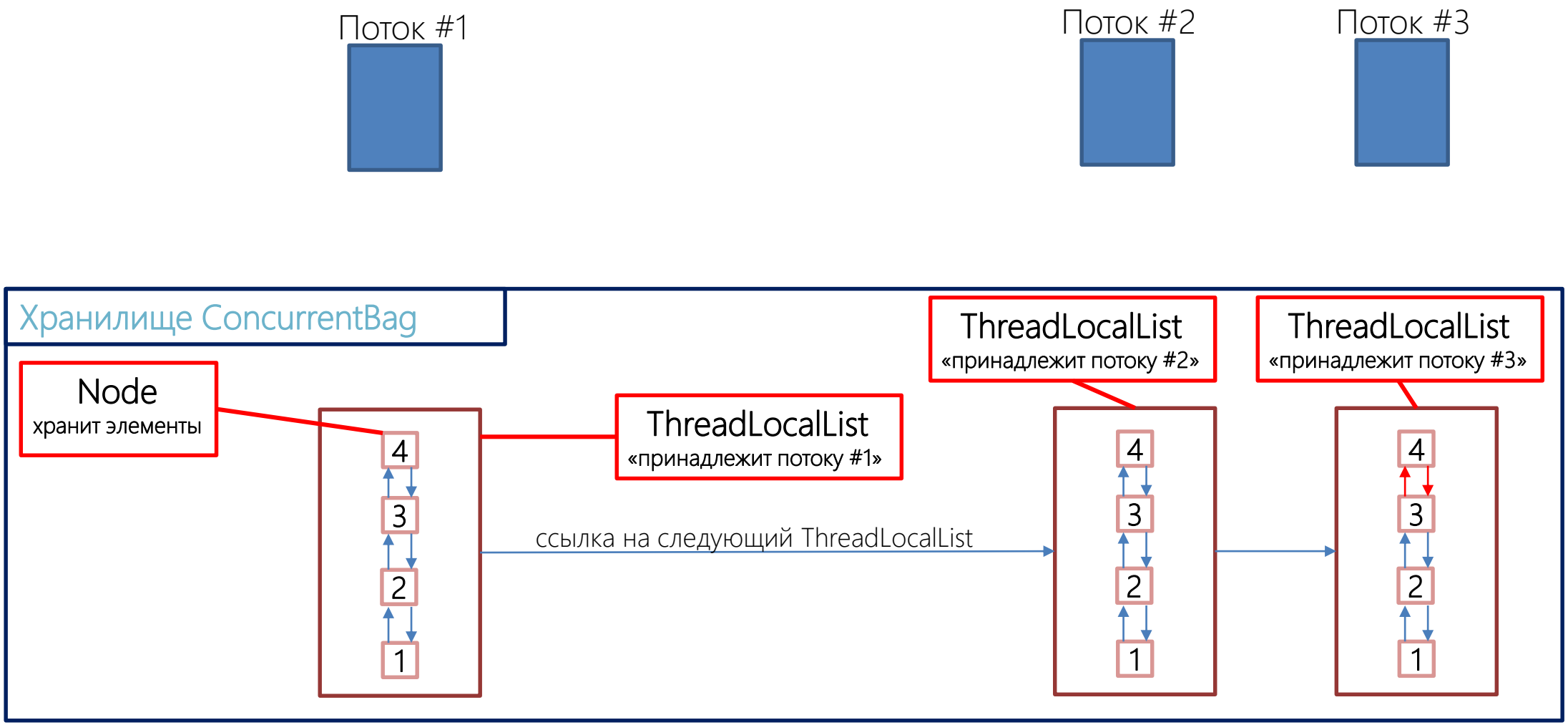
Добавление элементов в ConcurrentBag



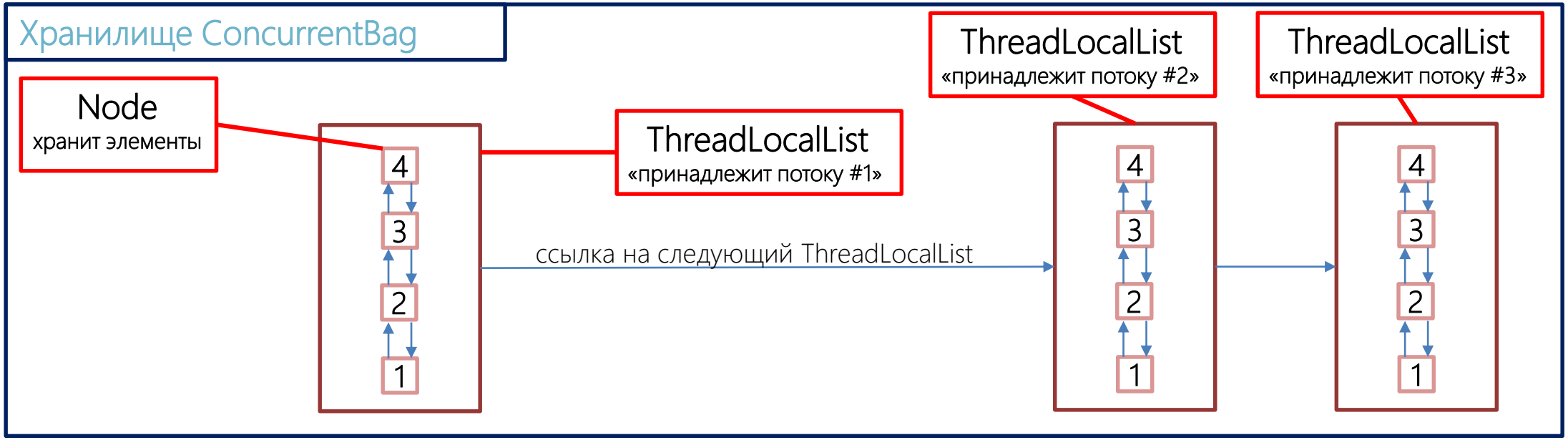
Добавление элементов в ConcurrentBag



Добавление элементов в ConcurrentBag



Добавление элементов в ConcurrentBag



Что происходит с ThreadLocalList при уничтожении потока

Если поток-владелец экземпляра ThreadLocalList уничтожается (получает состояние **Stopped**), значит его экземпляр ThreadLocalList становится свободным (без владельца). Новый поток, который добавляет элемент в первый раз, может получить этот существующий брошенный экземпляр. То есть он станет его новым владельцем.

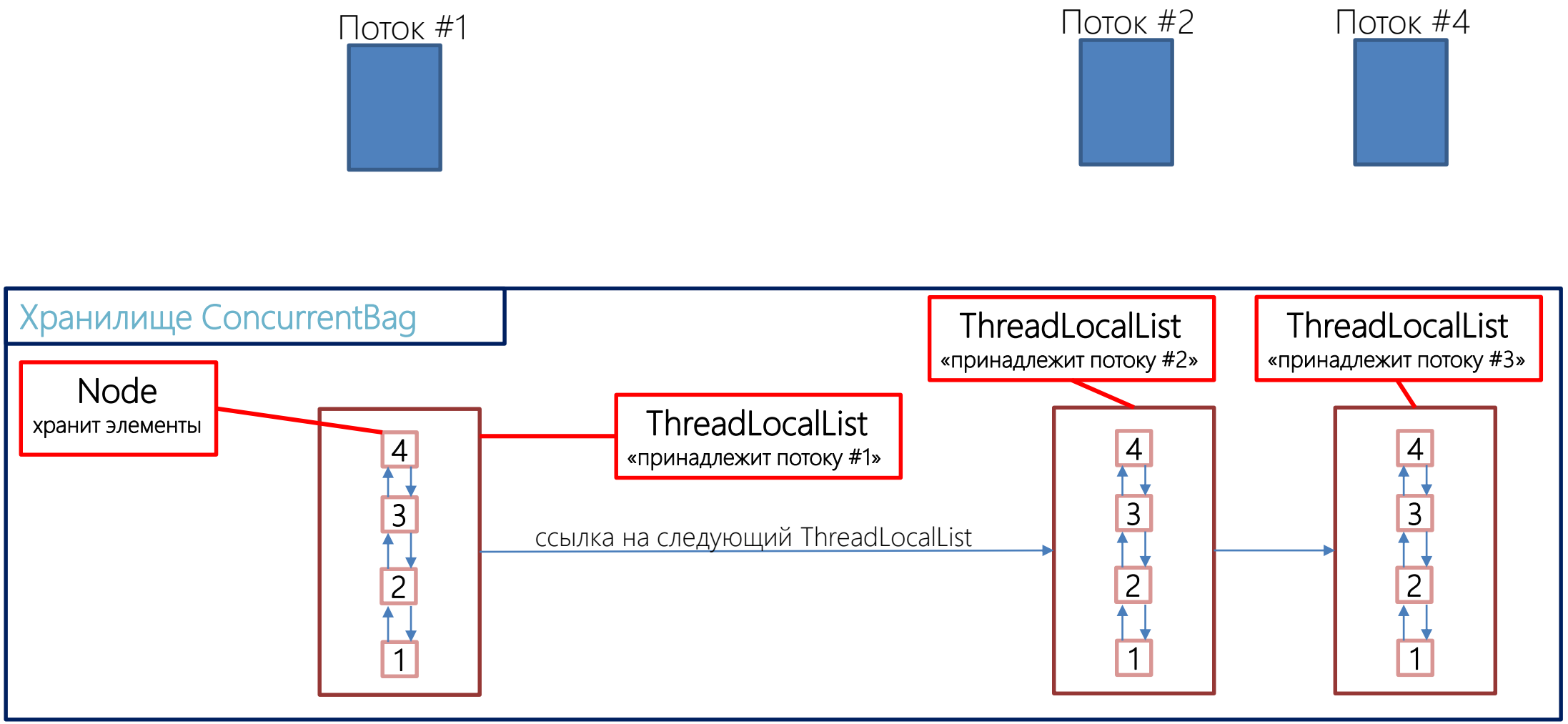
Как происходит извлечение из ConcurrentBag

Потоки извлекают элементы из своих хранилищ ThreadLocalList. Если у потока есть свое хранилище, он извлекает элемент с головы двунаправленного связного списка.

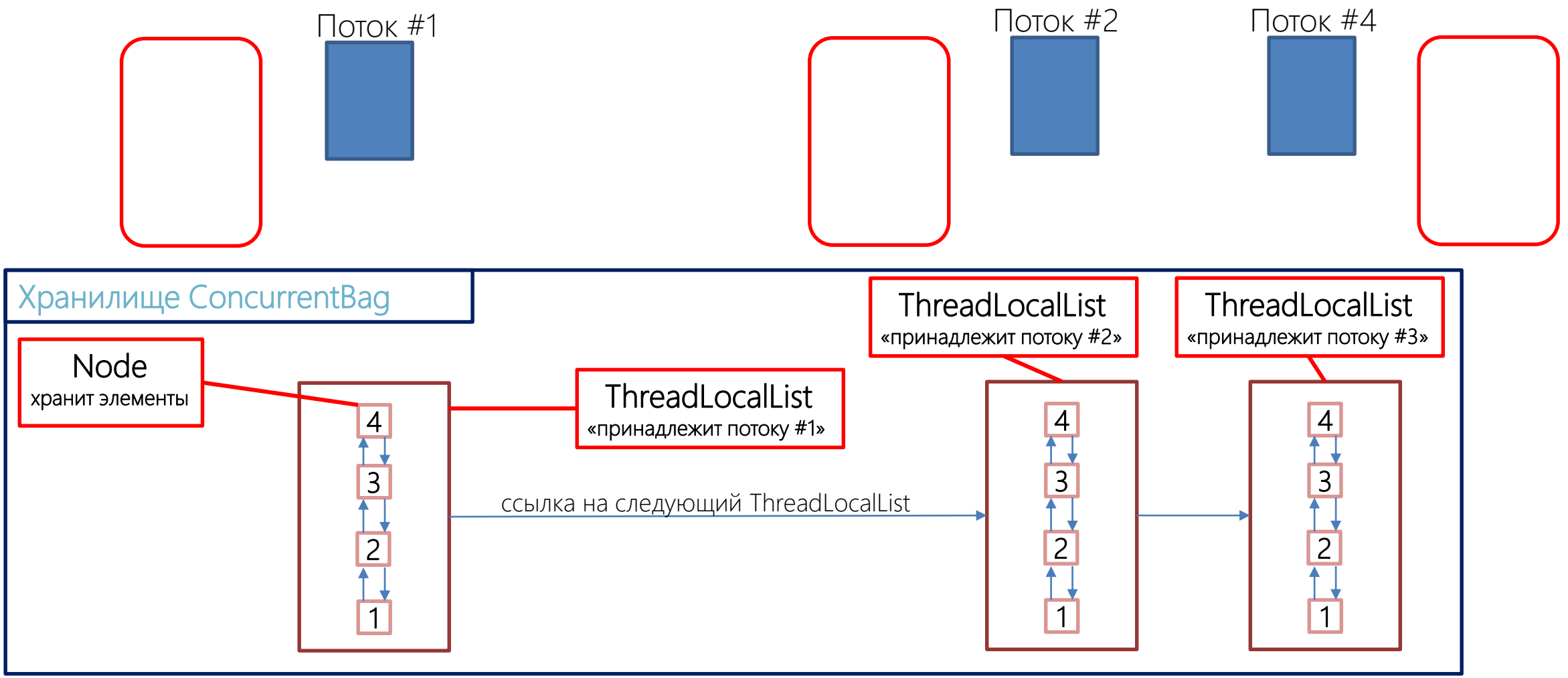
Когда поток опустошает свое хранилище, он не прекращает изымать элементы. Он по ссылке переходит с следующему ThreadLocalList, принадлежащему другому потоку. Там поток начинает красть элементы. Кража происходит с хвоста двунаправленного связного списка.

Если у потока изначально не было своего хранилища ThreadLocalList, он занимается кражами элементов из других хранилищ, начиная с самого первого ThreadLocalList.

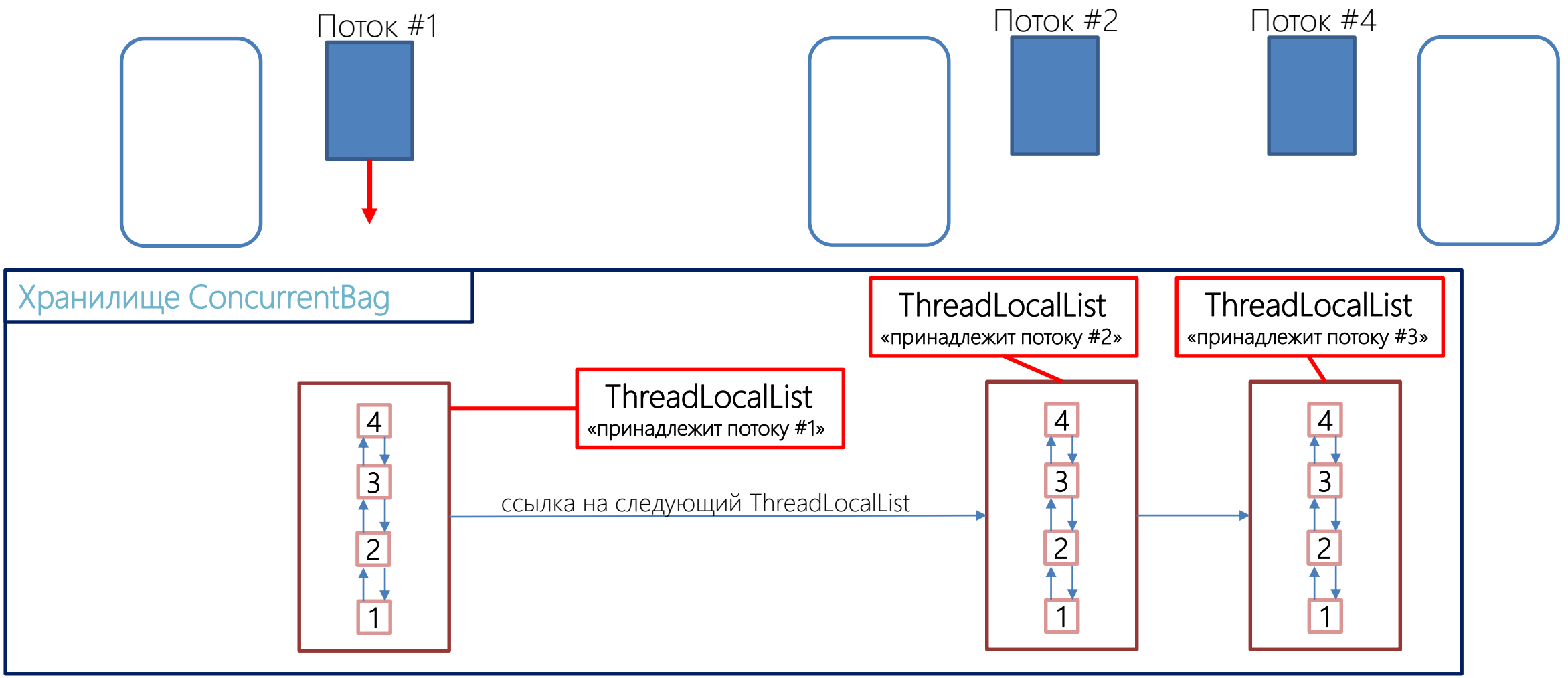
Извлечение элементов из ConcurrentBag



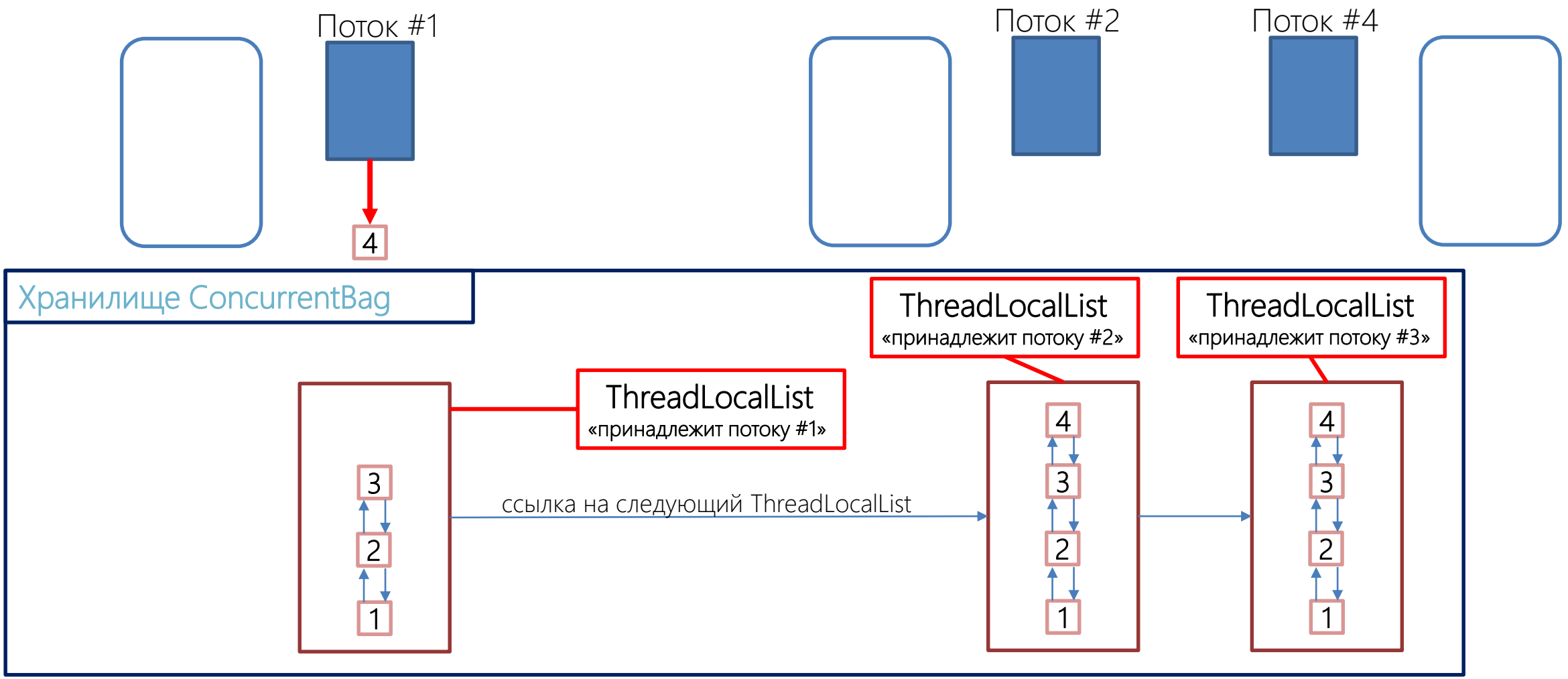
Извлечение элементов из ConcurrentBag



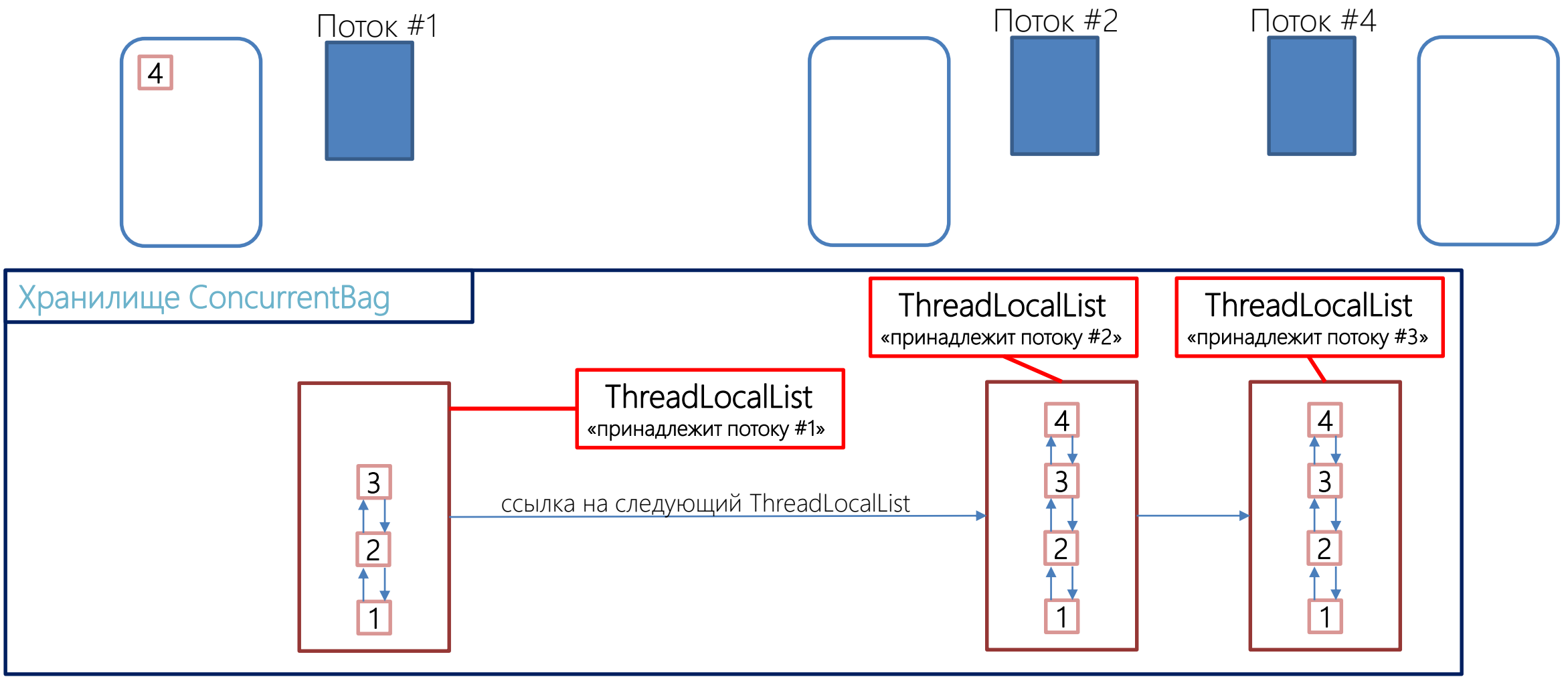
Извлечение элементов из ConcurrentBag



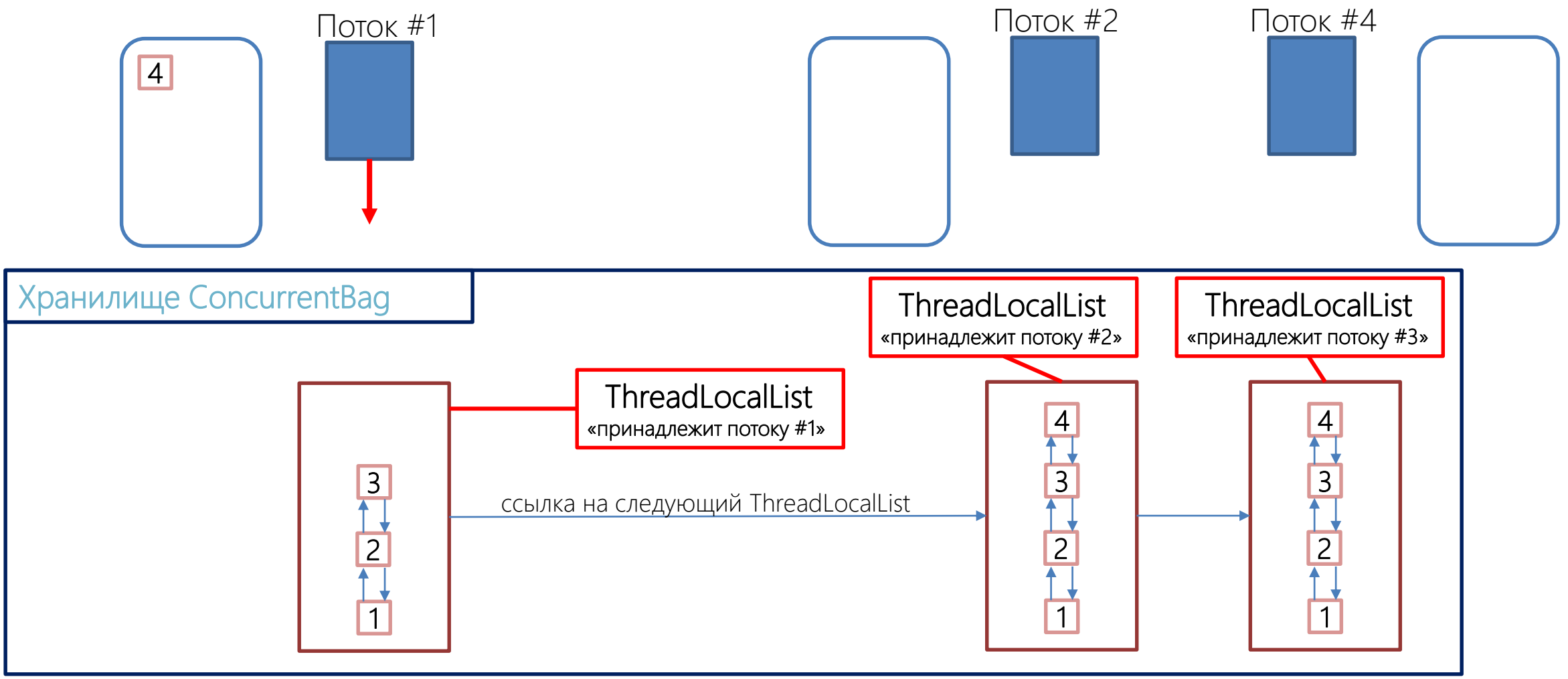
Извлечение элементов из ConcurrentBag



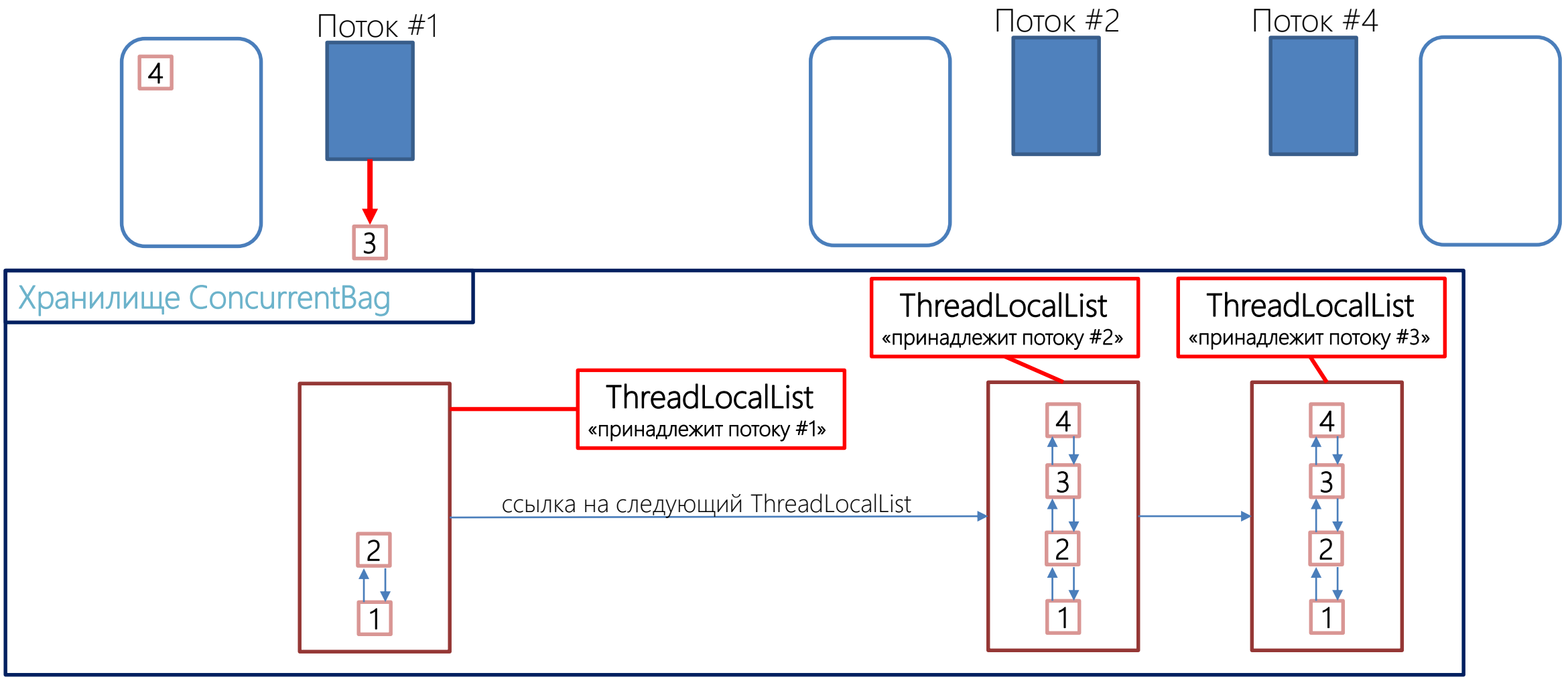
Извлечение элементов из ConcurrentBag



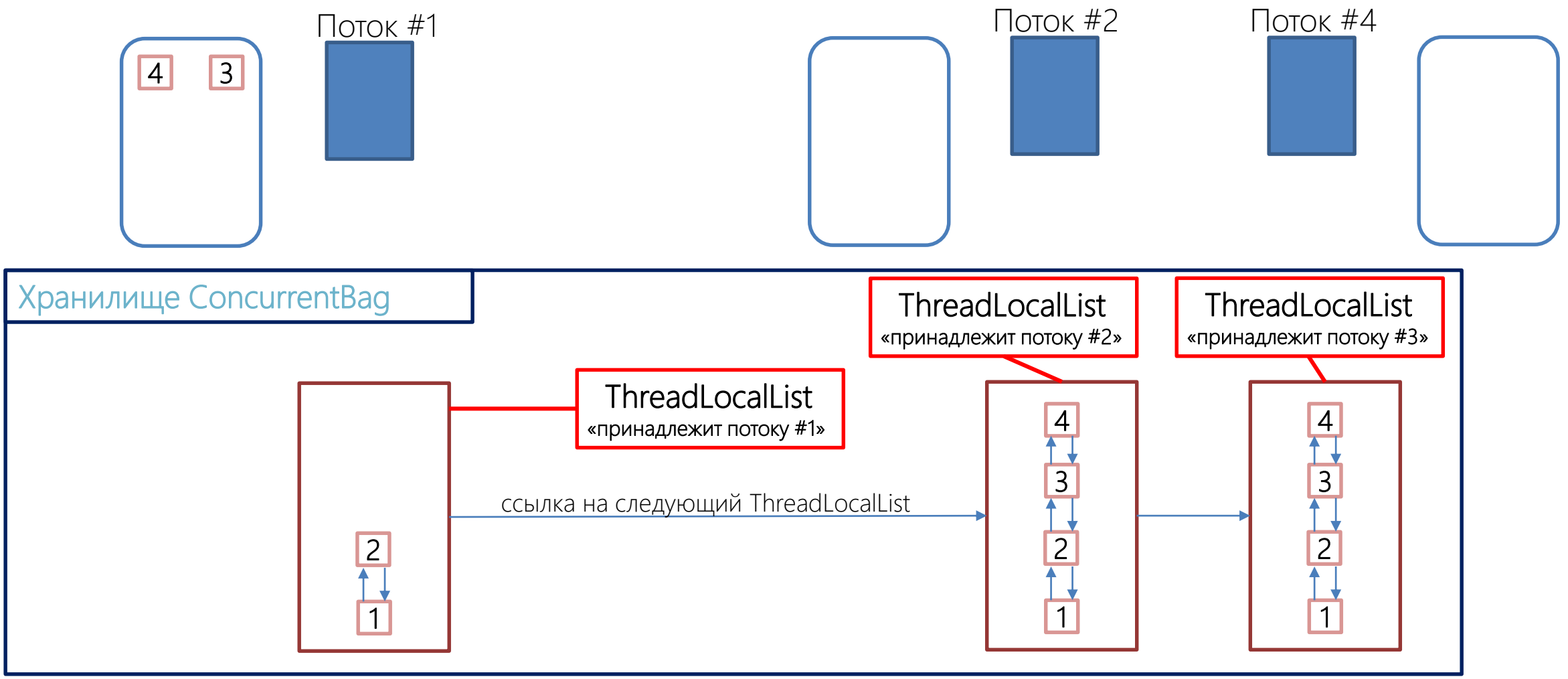
Извлечение элементов из ConcurrentBag



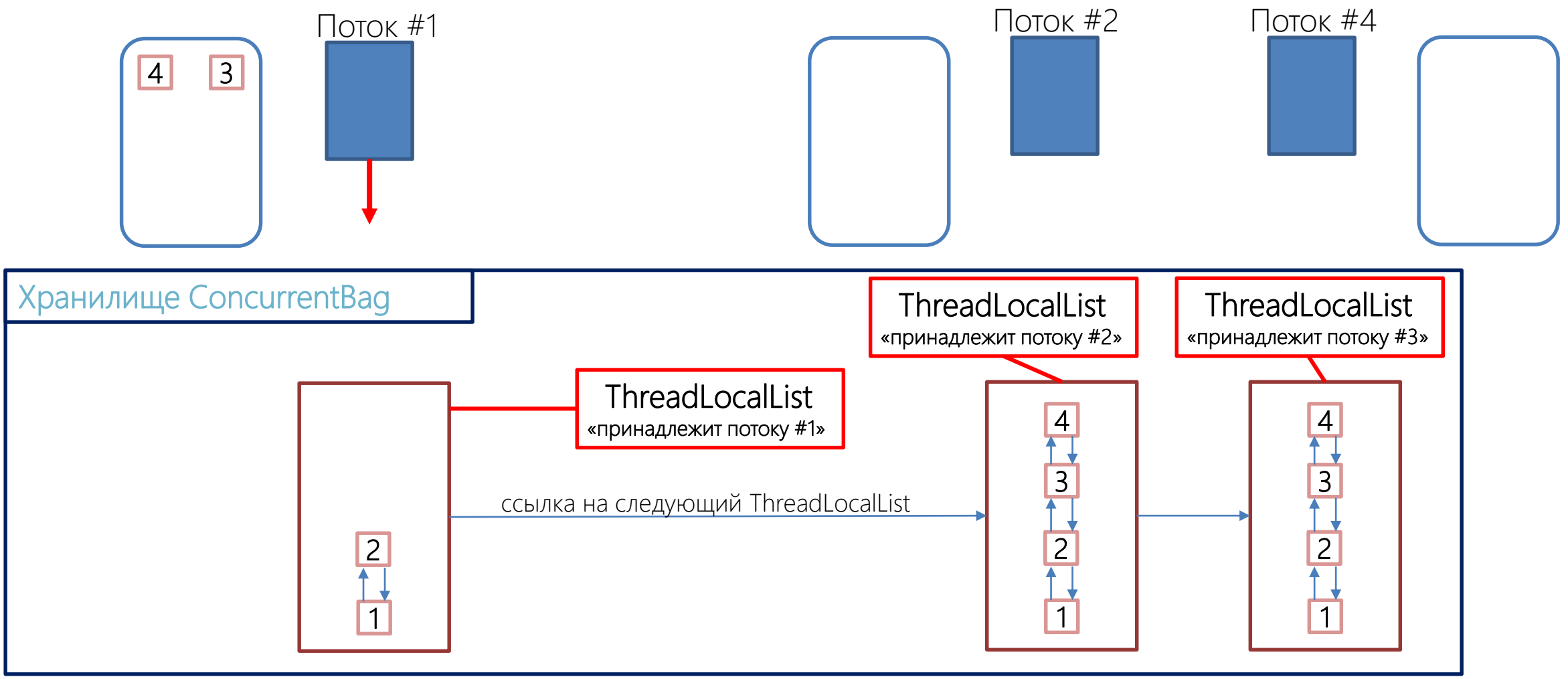
Извлечение элементов из ConcurrentBag



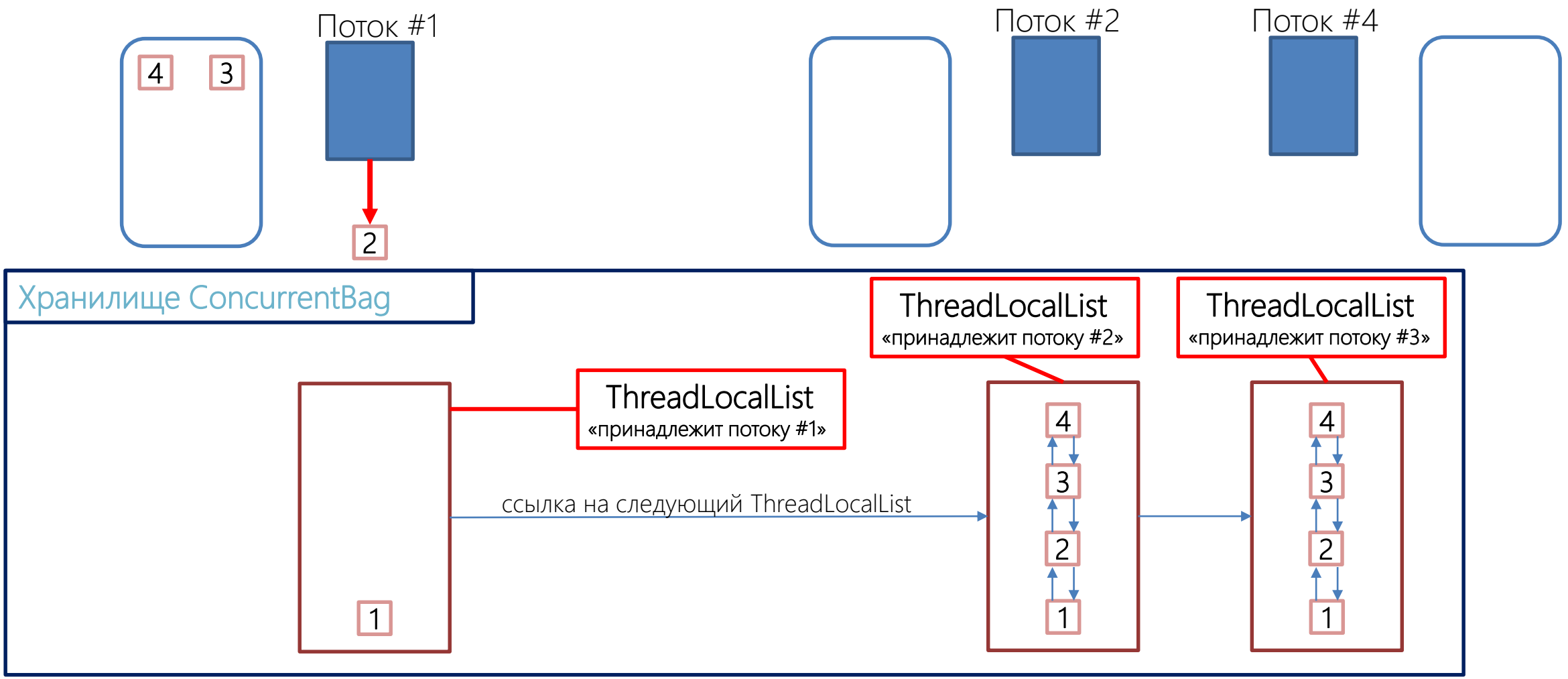
Извлечение элементов из ConcurrentBag



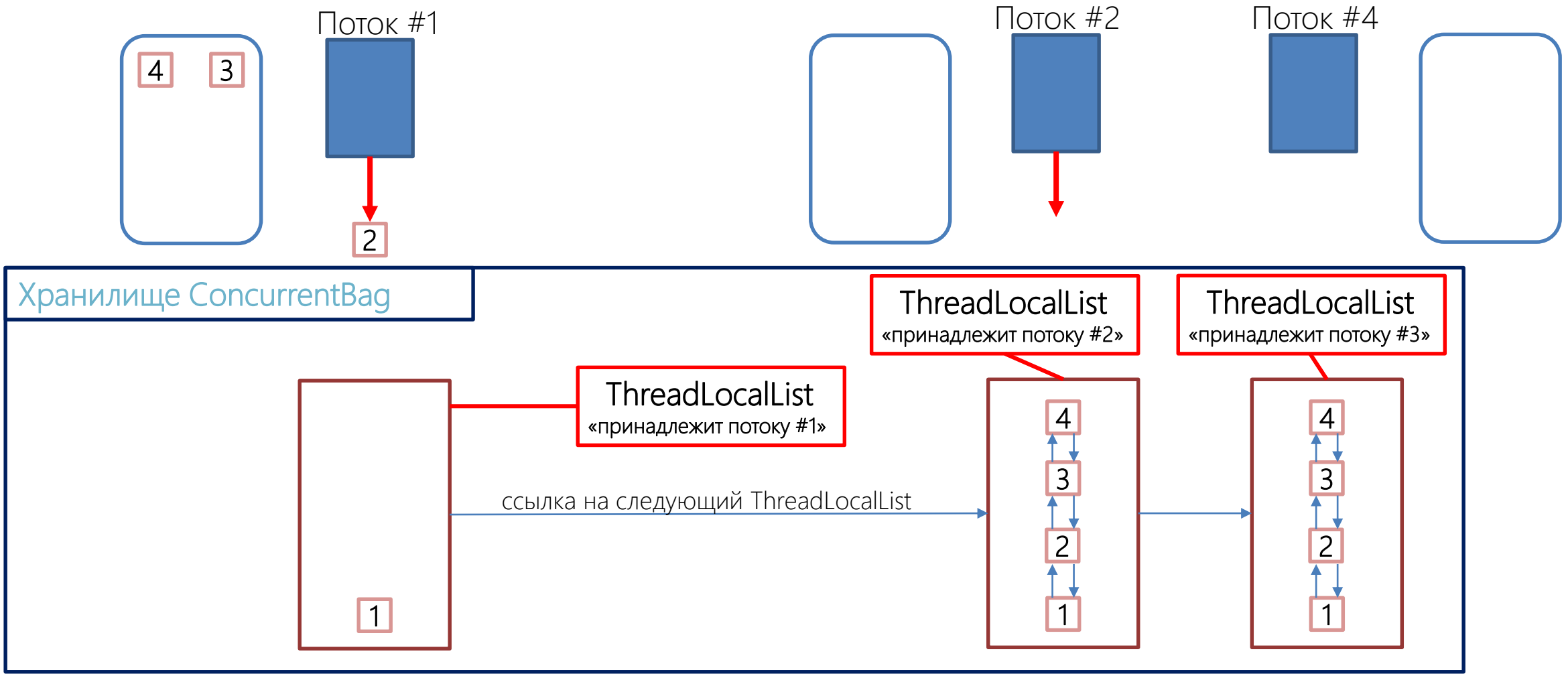
Извлечение элементов из ConcurrentBag



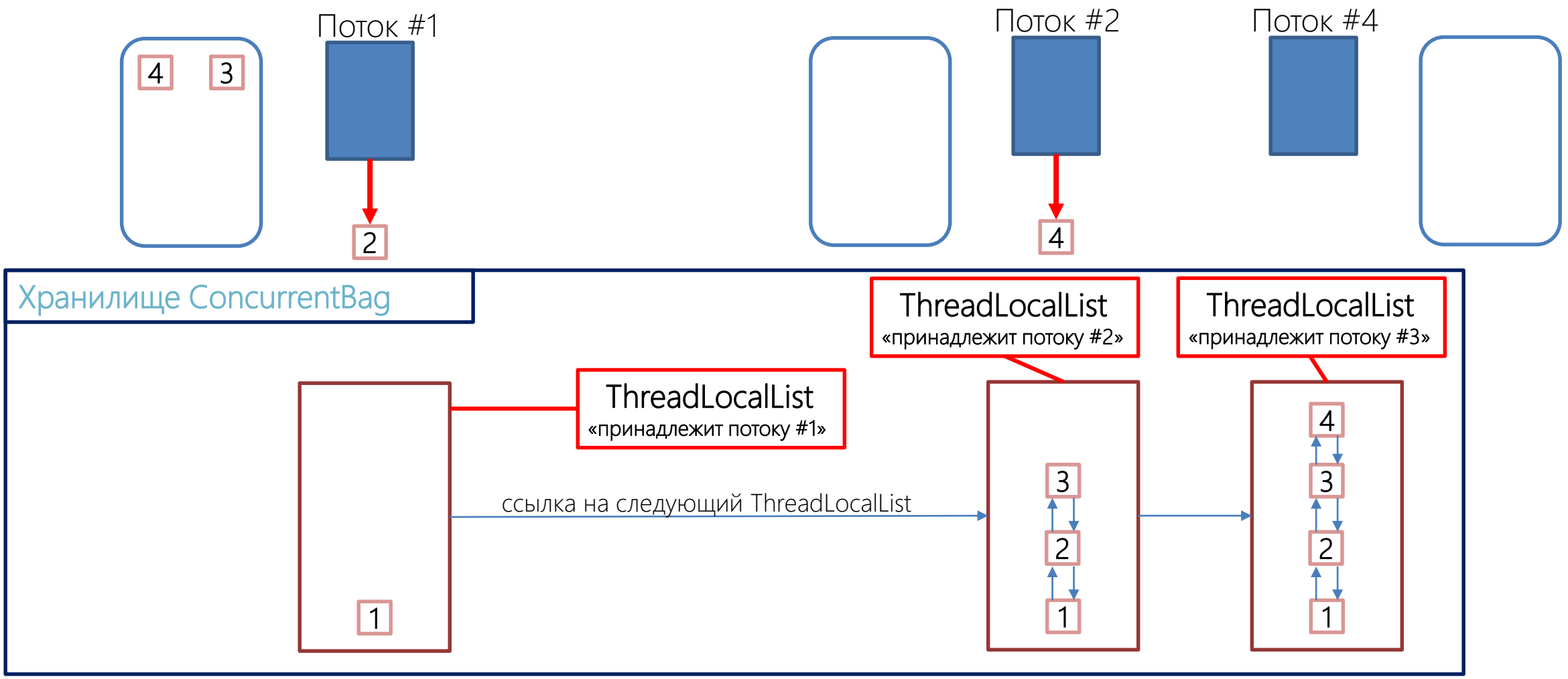
Извлечение элементов из ConcurrentBag



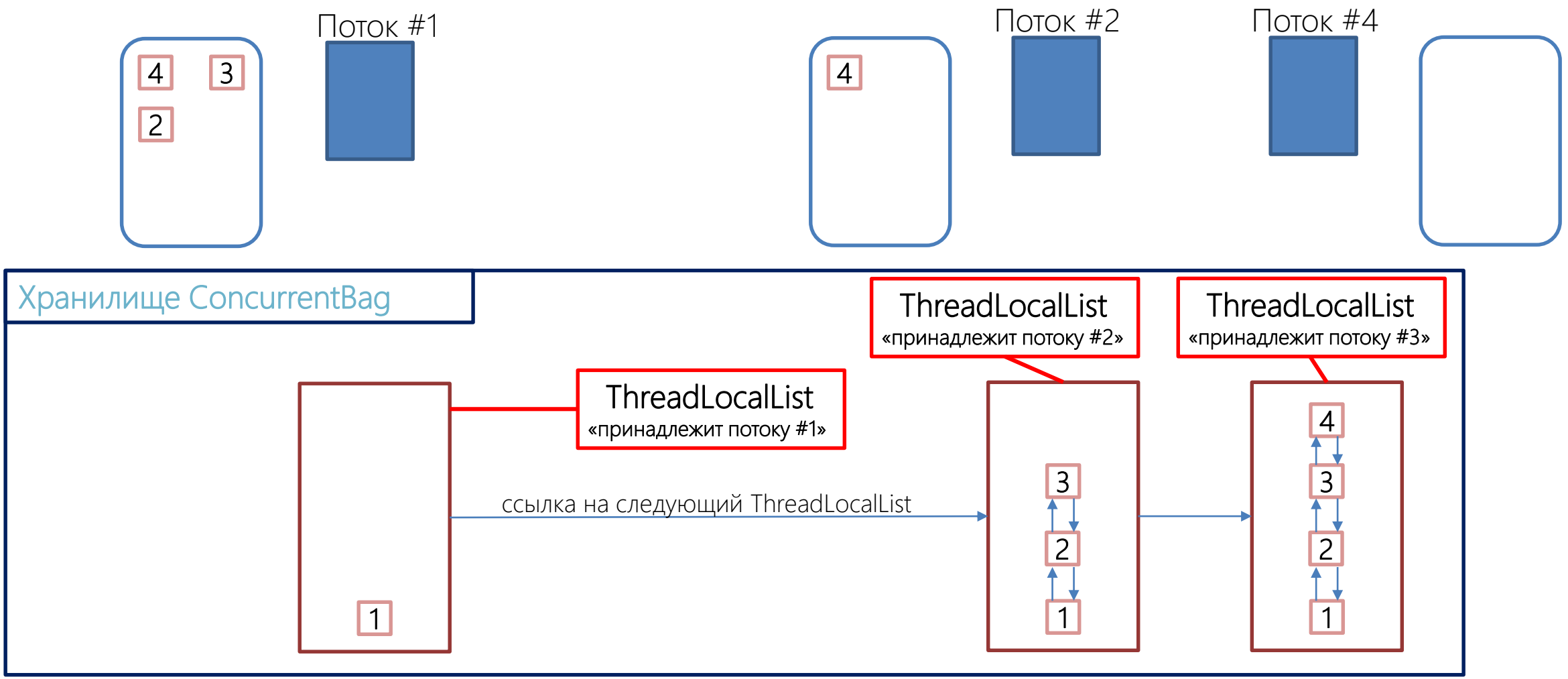
Извлечение элементов из ConcurrentBag



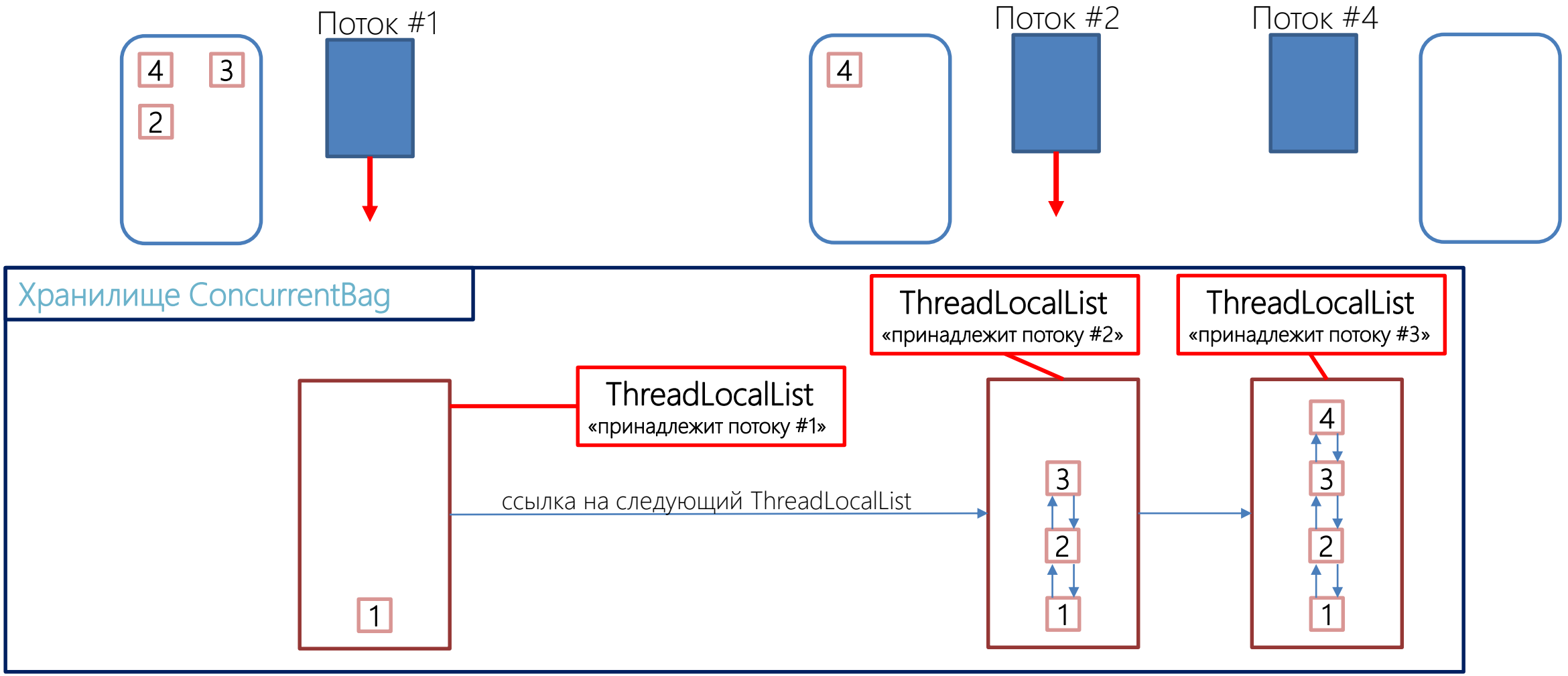
Извлечение элементов из ConcurrentBag



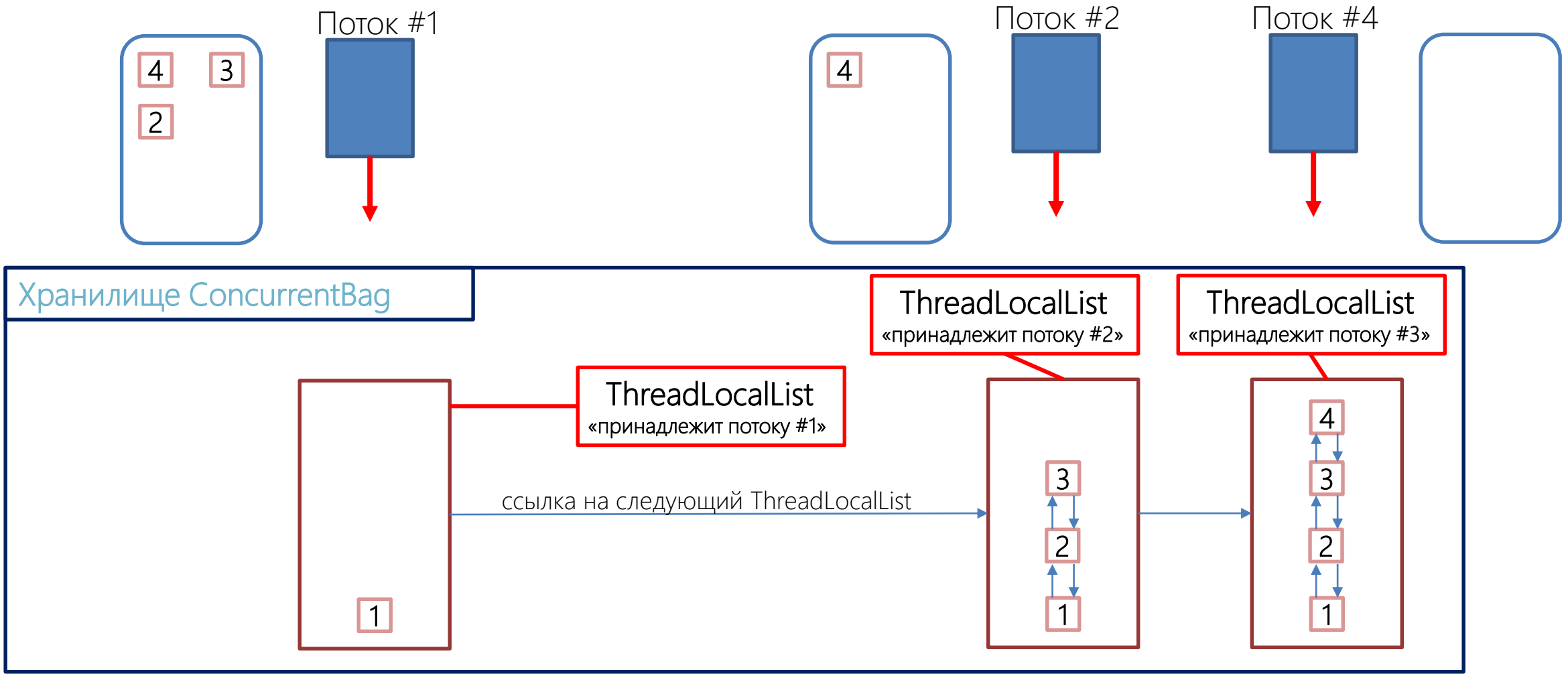
Извлечение элементов из ConcurrentBag



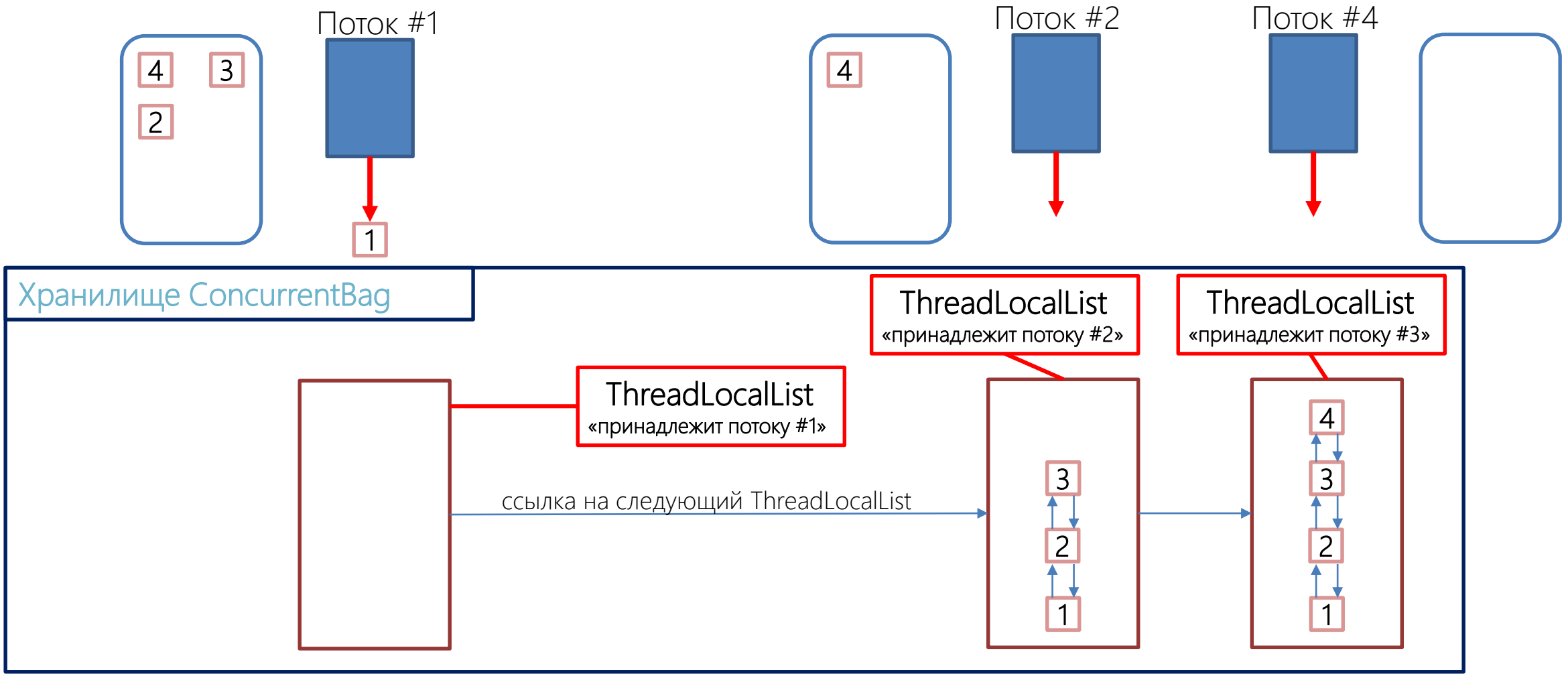
Извлечение элементов из ConcurrentBag



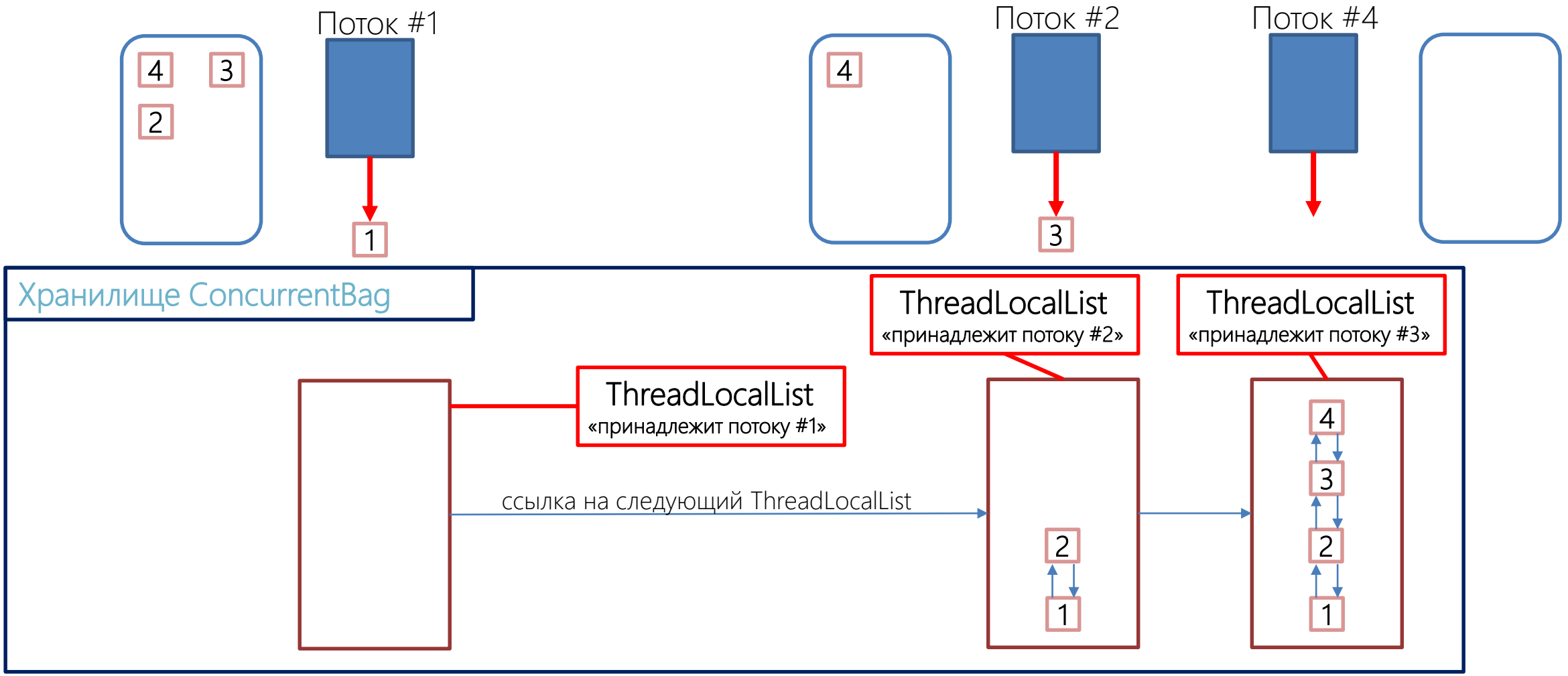
Извлечение элементов из ConcurrentBag



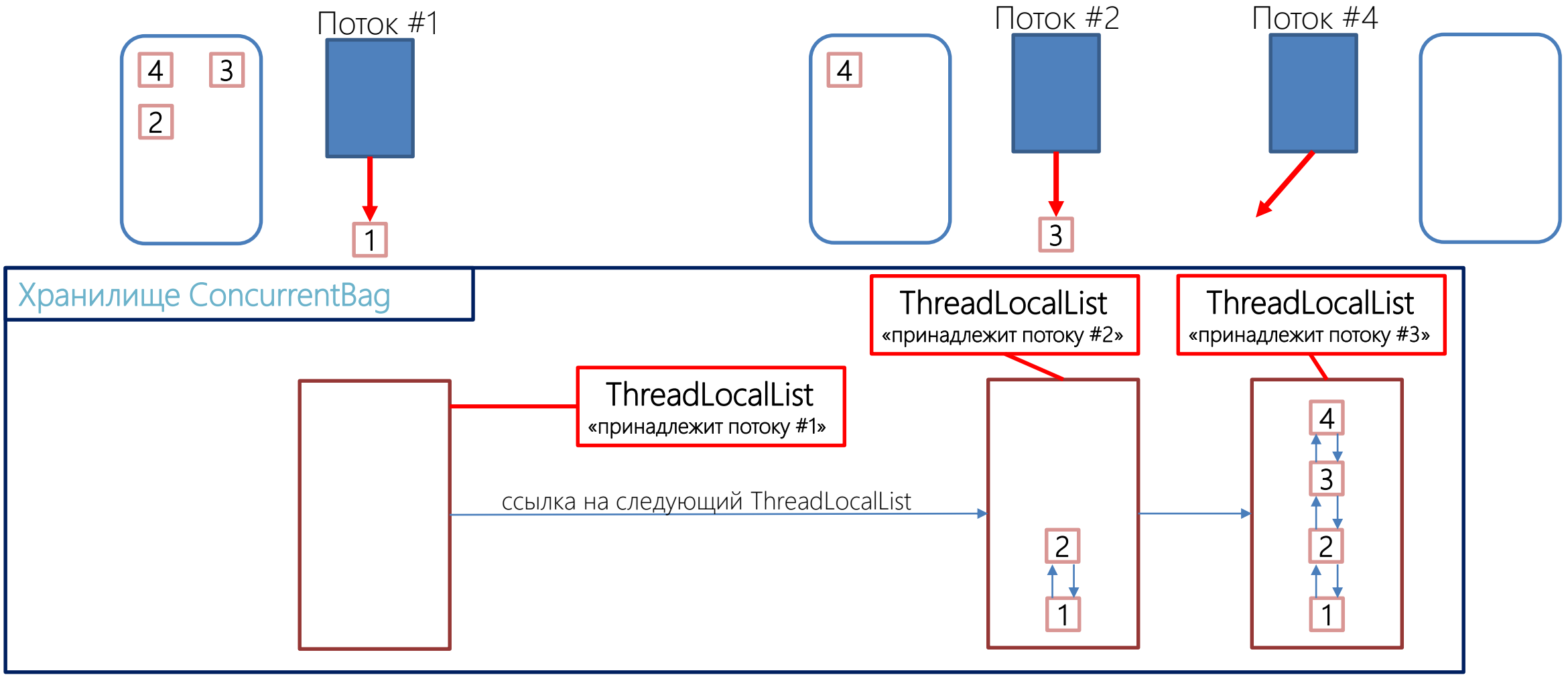
Извлечение элементов из ConcurrentBag



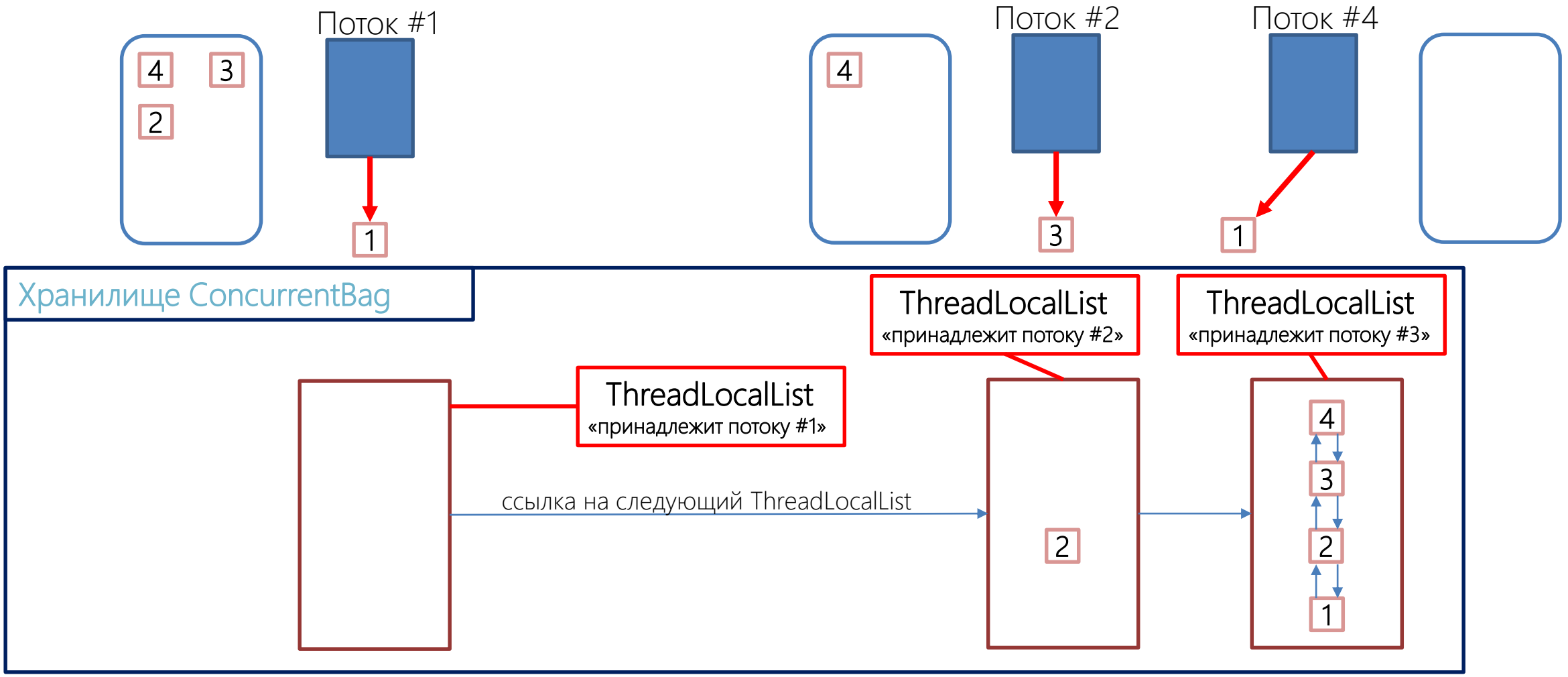
Извлечение элементов из ConcurrentBag



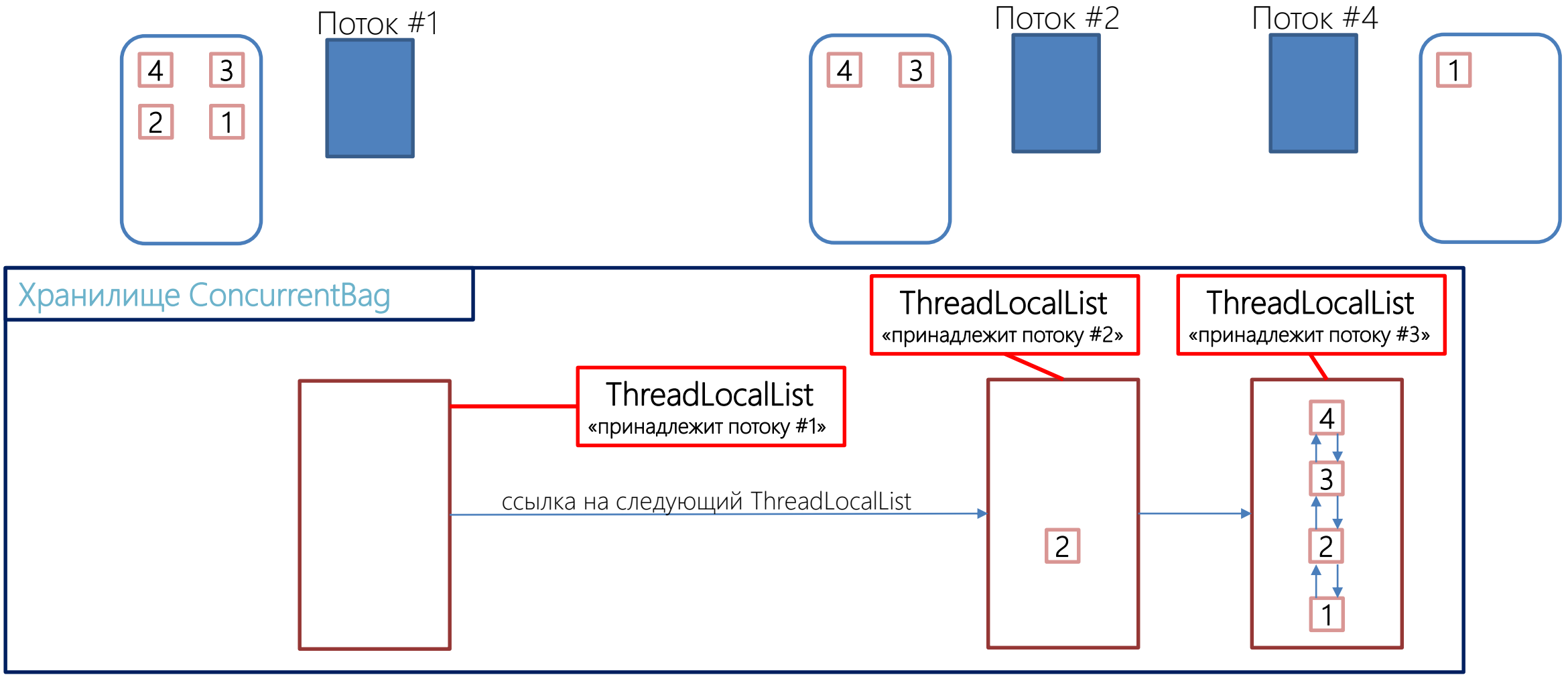
Извлечение элементов из ConcurrentBag



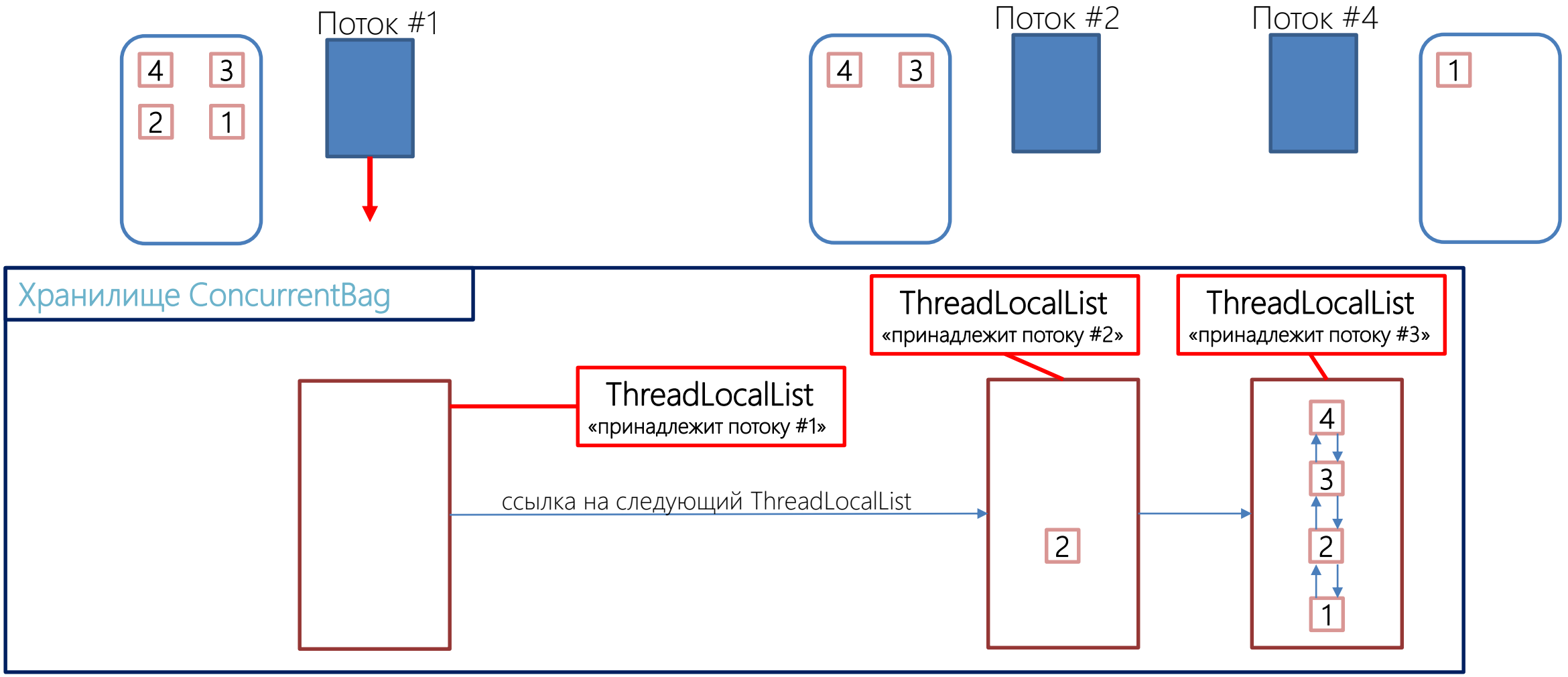
Извлечение элементов из ConcurrentBag



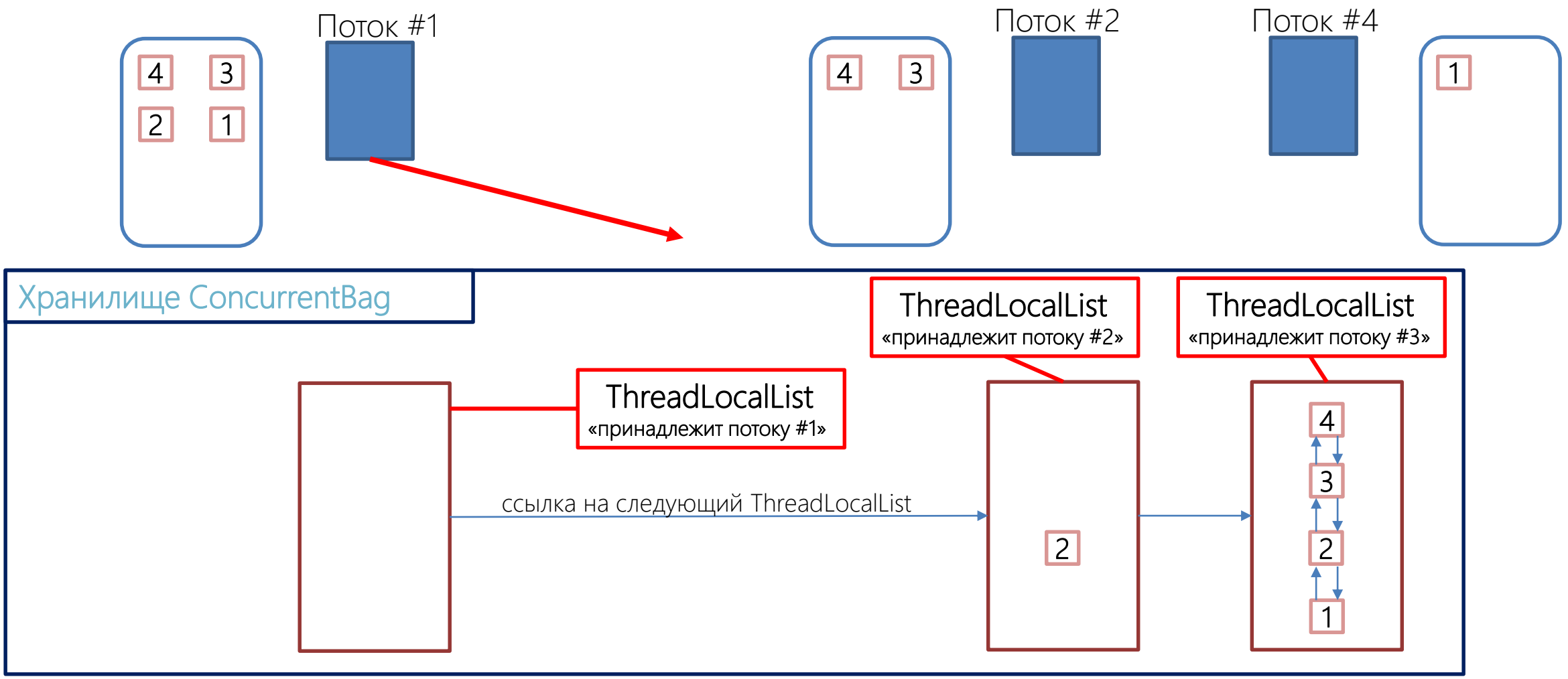
Извлечение элементов из ConcurrentBag



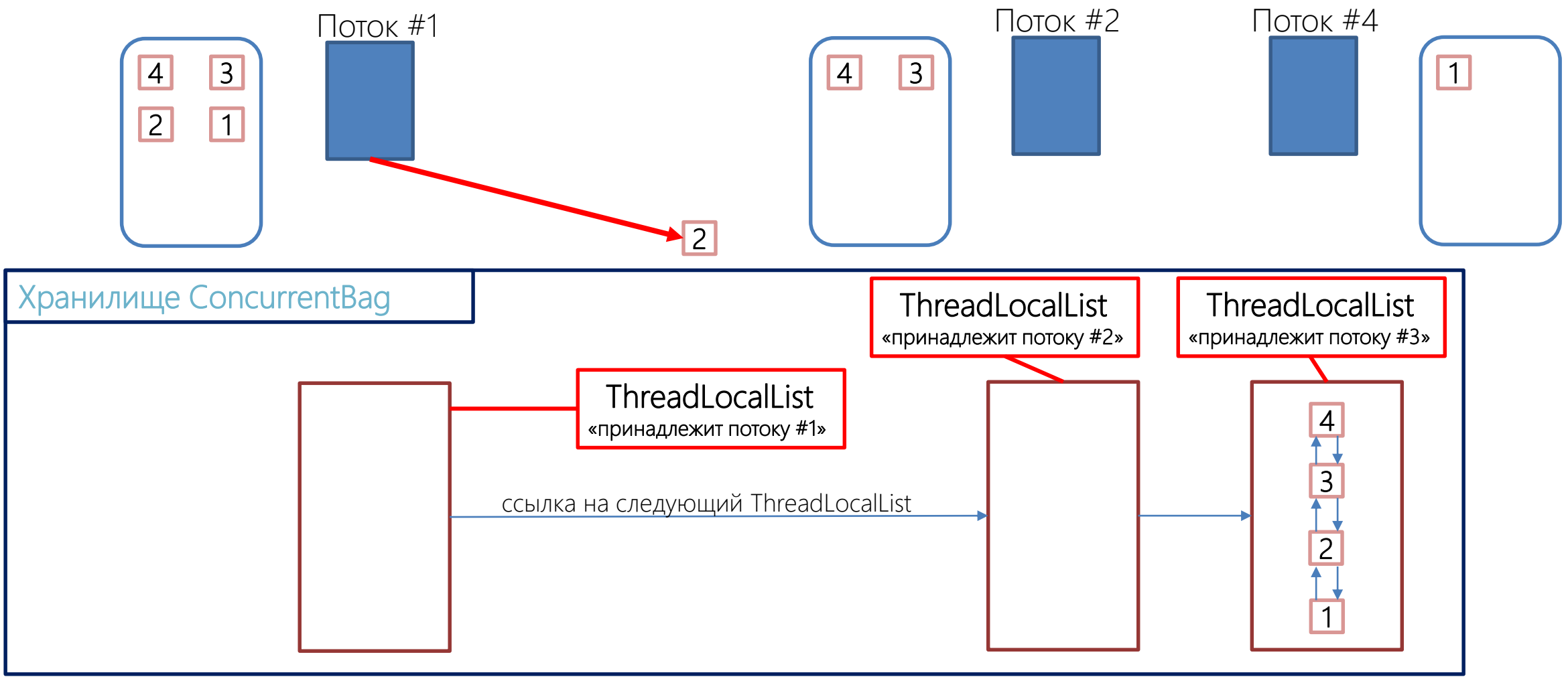
Извлечение элементов из ConcurrentBag



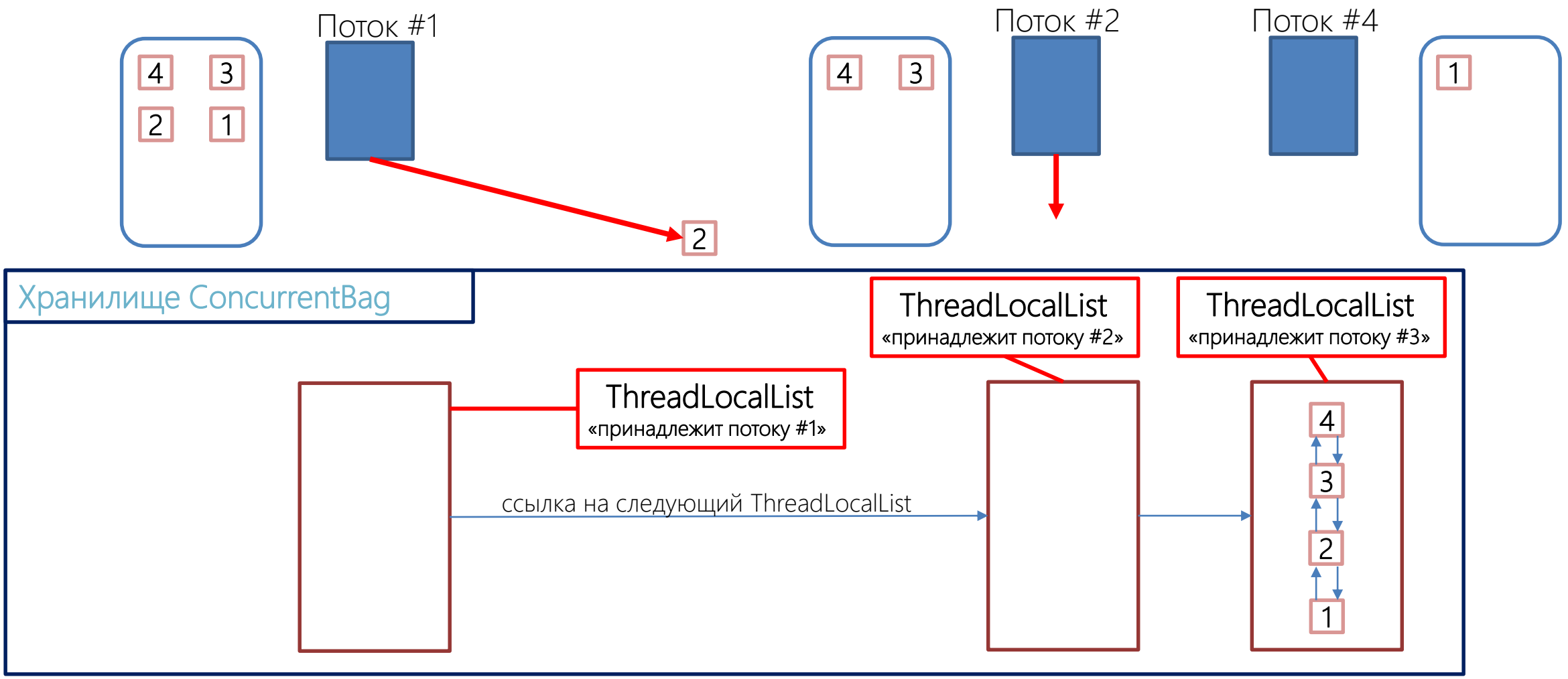
Извлечение элементов из ConcurrentBag



Извлечение элементов из ConcurrentBag

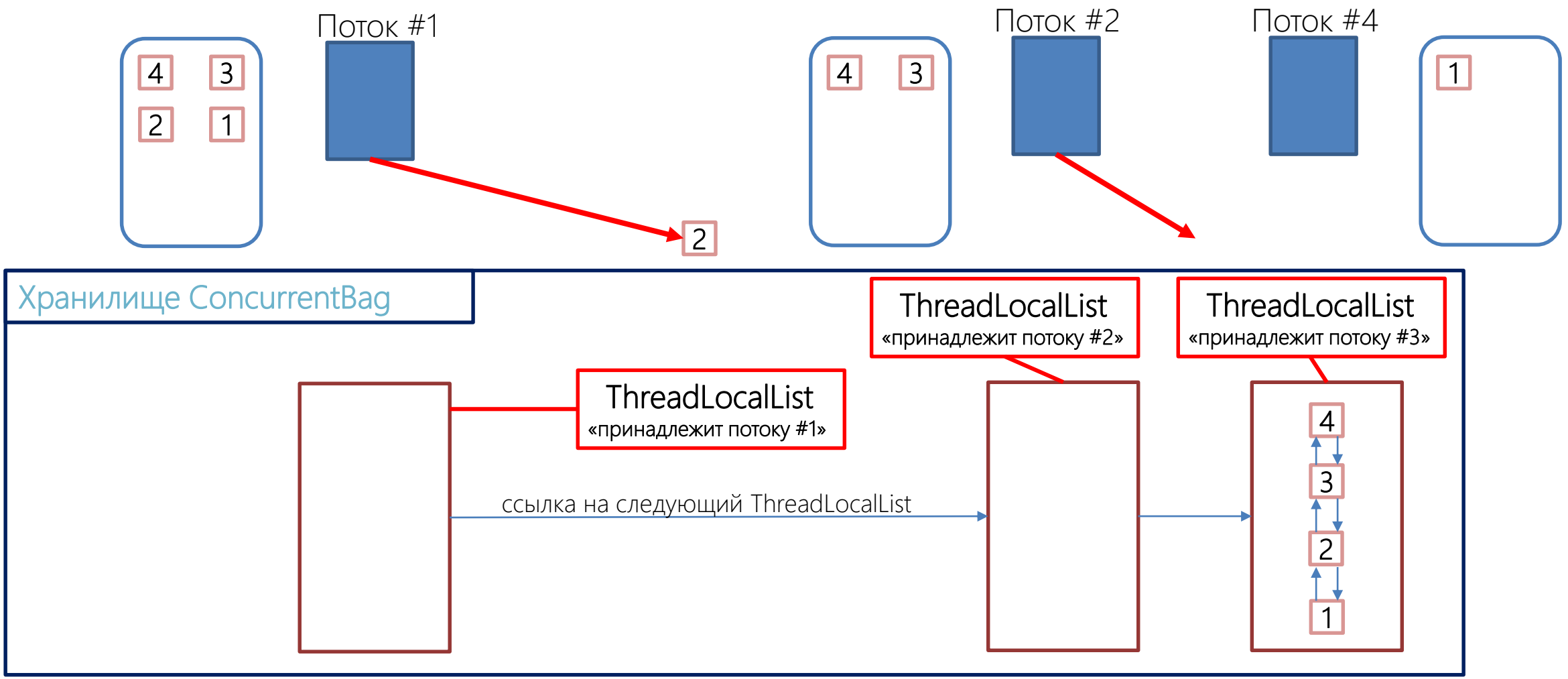


Извлечение элементов из ConcurrentBag

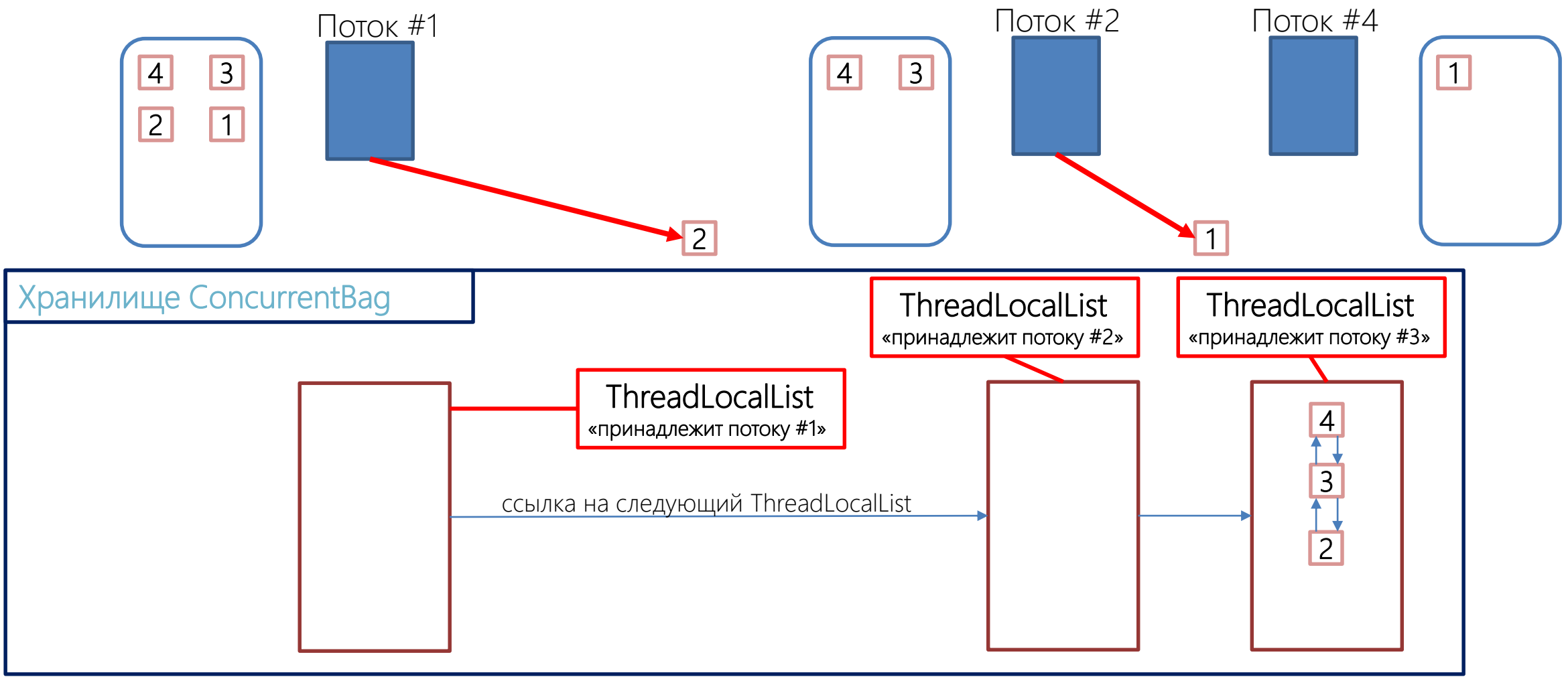


C# Асинхронное программирование

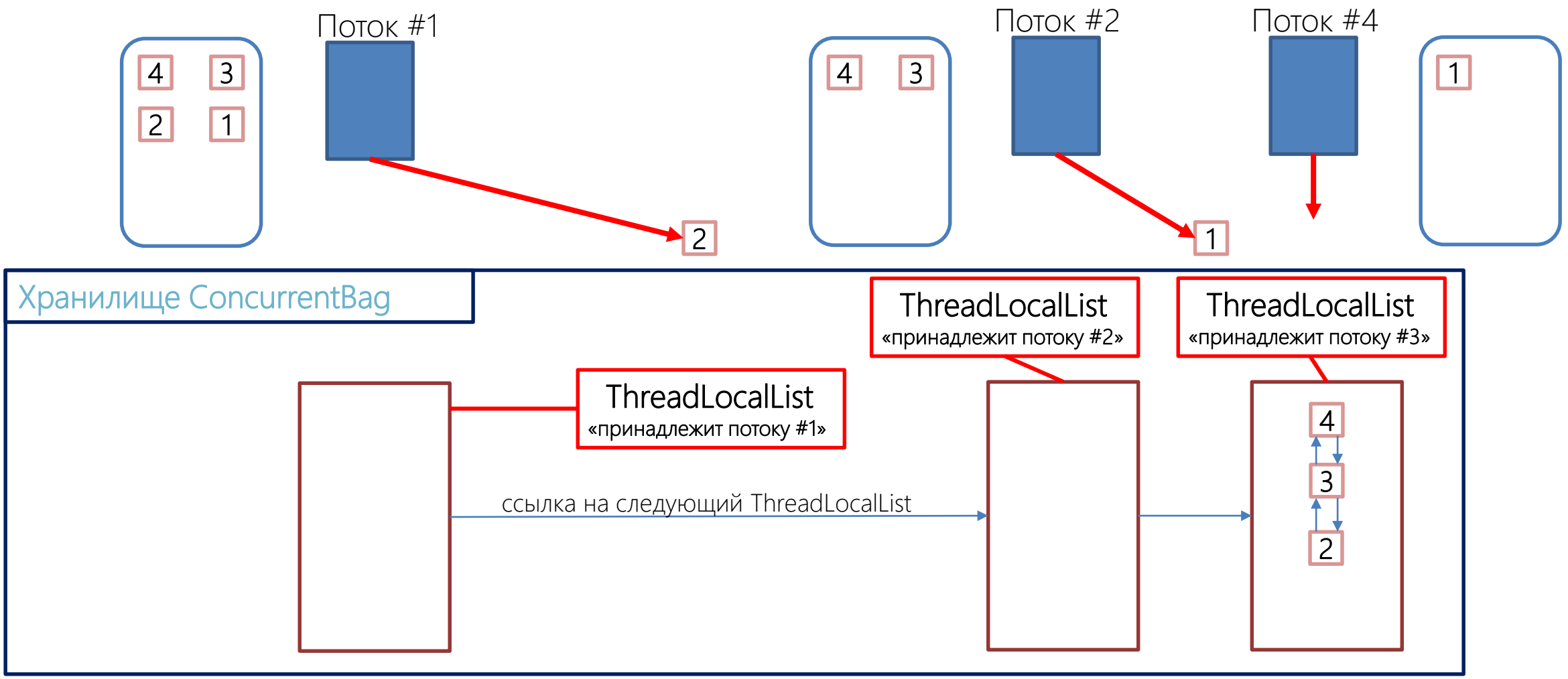
Извлечение элементов из ConcurrentBag



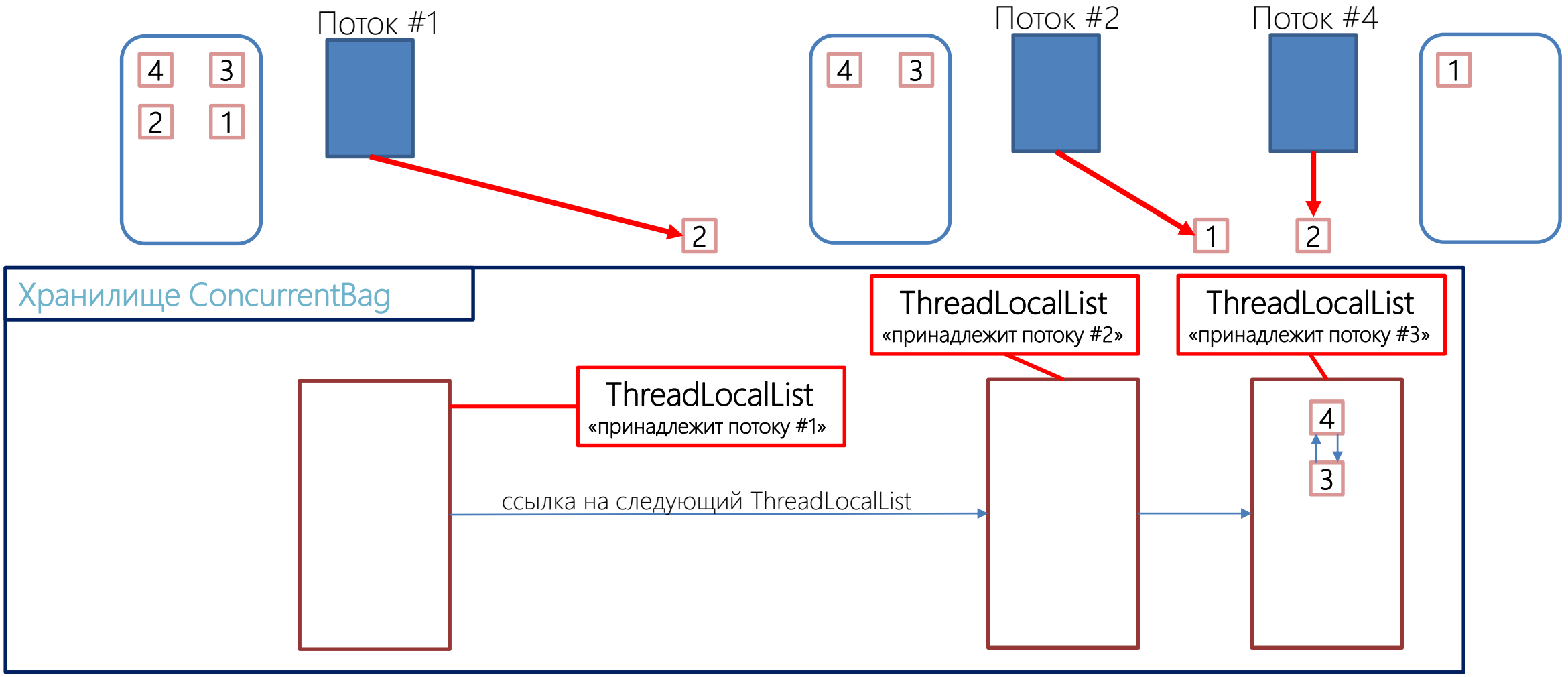
Извлечение элементов из ConcurrentBag



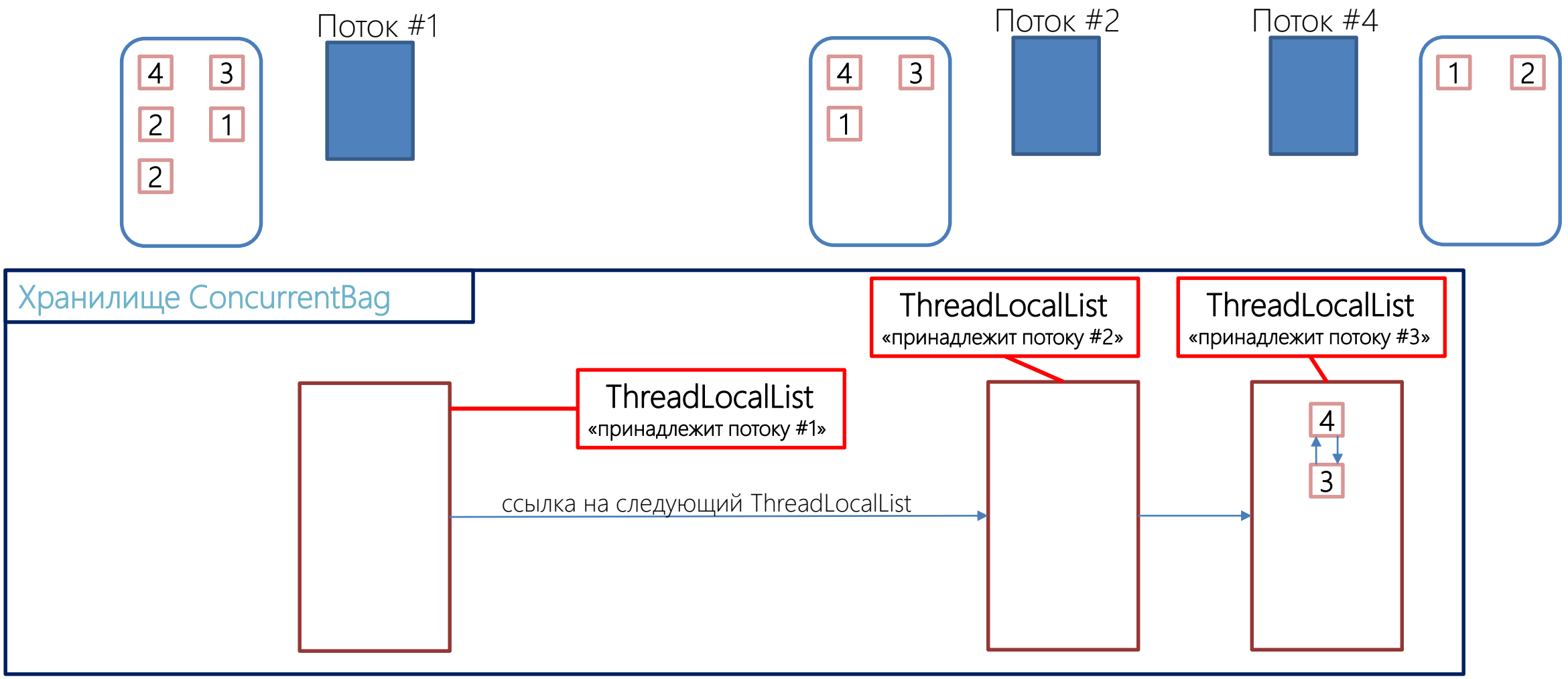
Извлечение элементов из ConcurrentBag



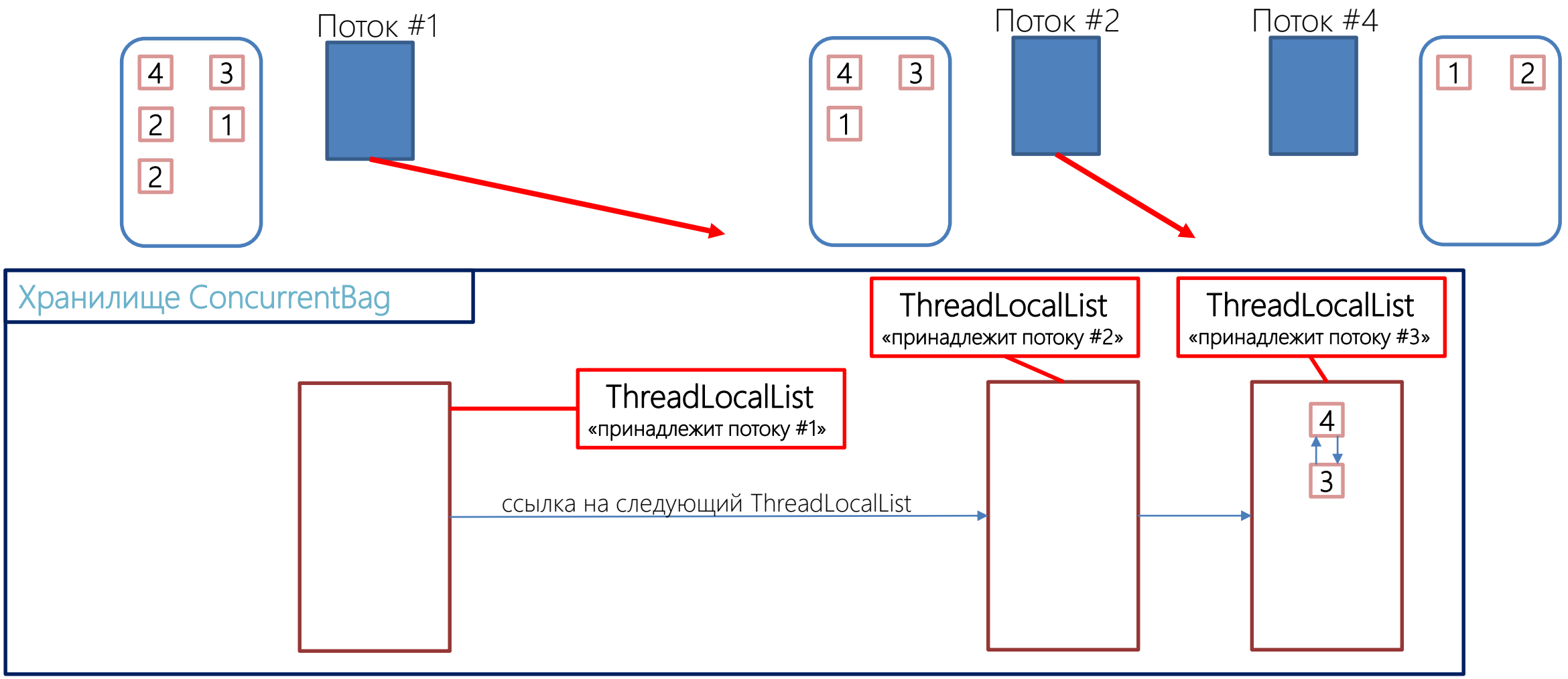
Извлечение элементов из ConcurrentBag



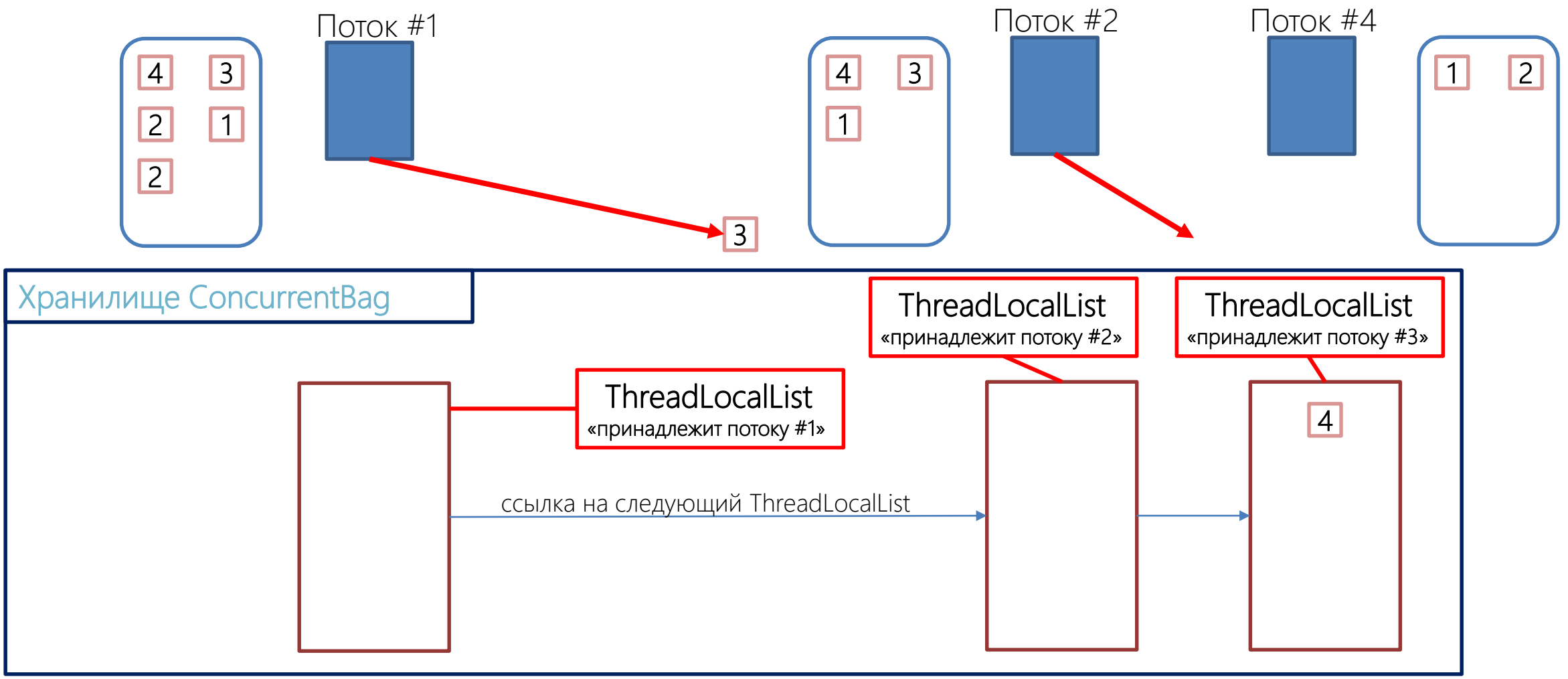
Извлечение элементов из ConcurrentBag



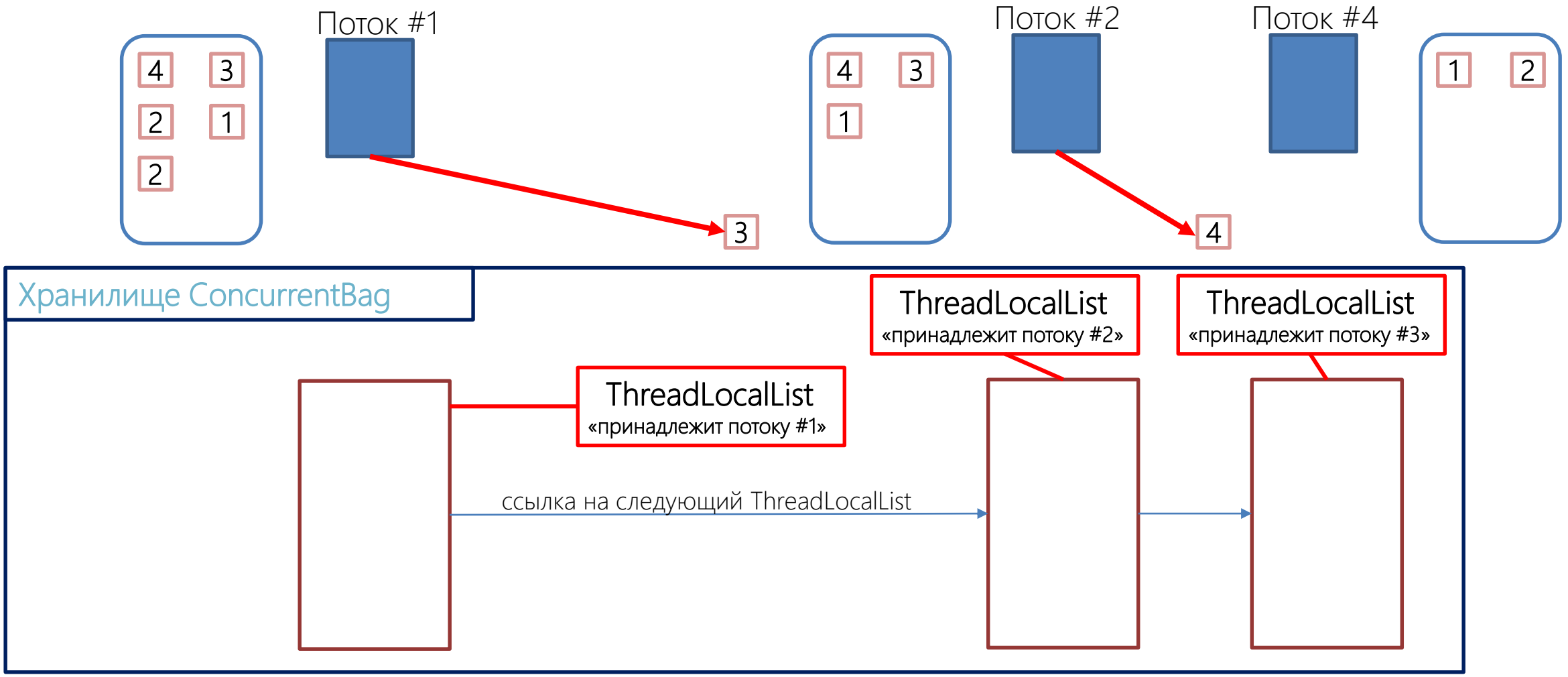
Извлечение элементов из ConcurrentBag



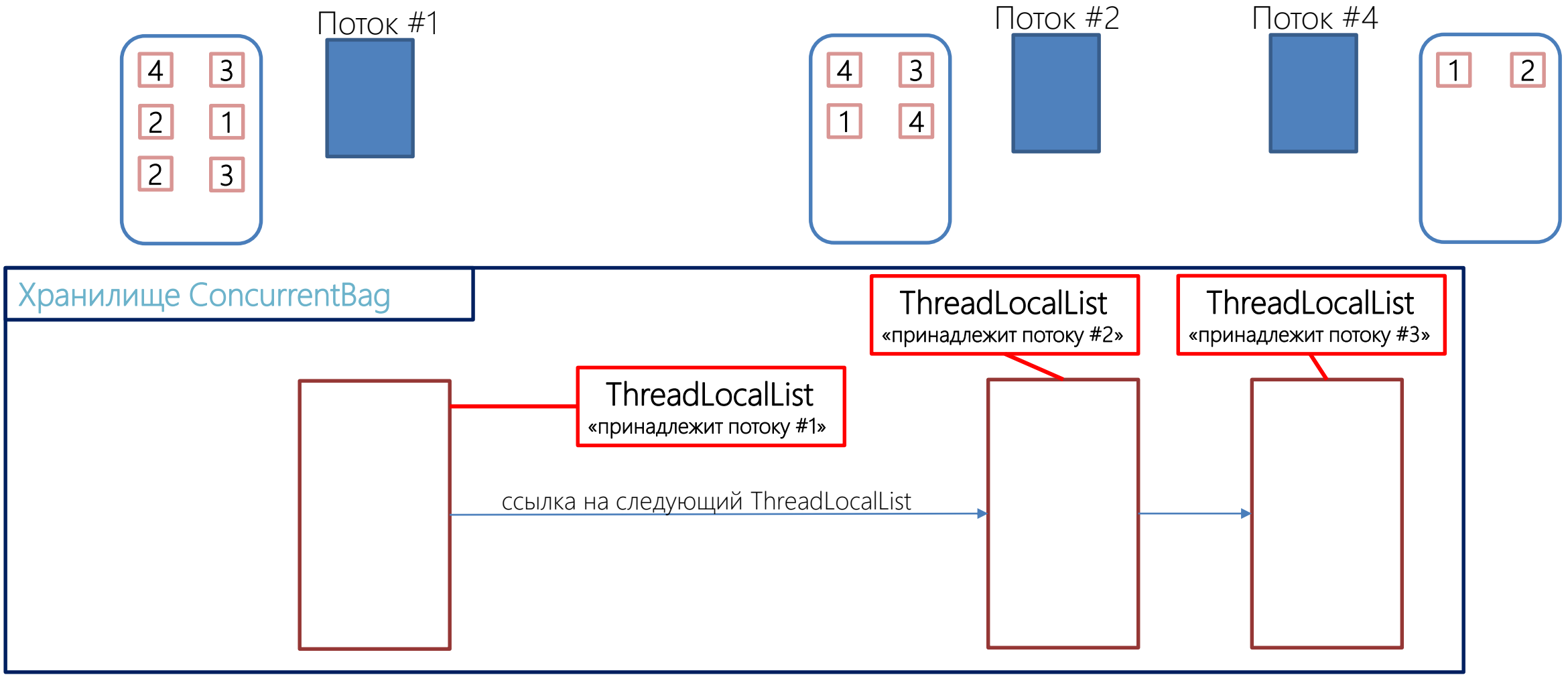
Извлечение элементов из ConcurrentBag



Извлечение элементов из ConcurrentBag



Извлечение элементов из ConcurrentBag



Работа ConcurrentQueue и ConcurrentStack

ConcurrentQueue и ConcurrentStack реализованы в виде однонаправленного связного списка.

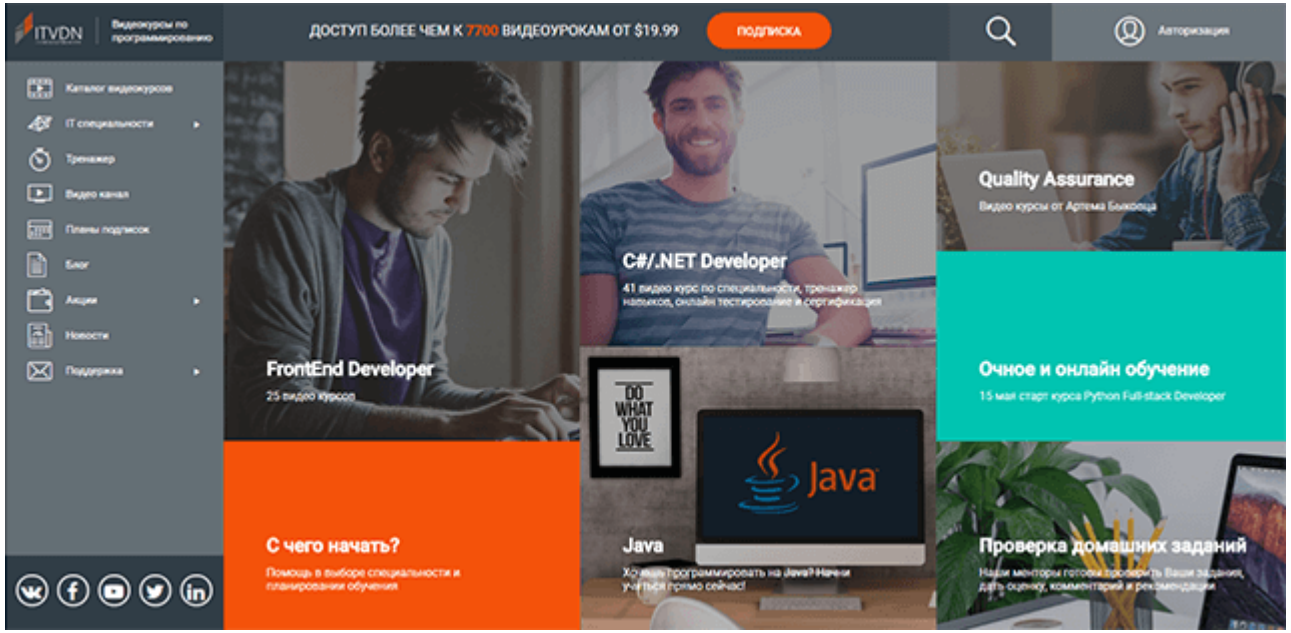
ConcurrentQueue состоит из экземпляров класса Segment, в которых сохраняются значения элементов по принципу FIFO. Каждый экземпляр Segment может хранить ссылку на следующий Segment.

ConcurrentStack состоит из экземпляров класса Node, в которых сохраняются значения элементов по принципу LIFO. Каждый экземпляр Node может хранить ссылку на следующий Node.

Q&A

Смотрите наши уроки в видео формате

ITVDN.com



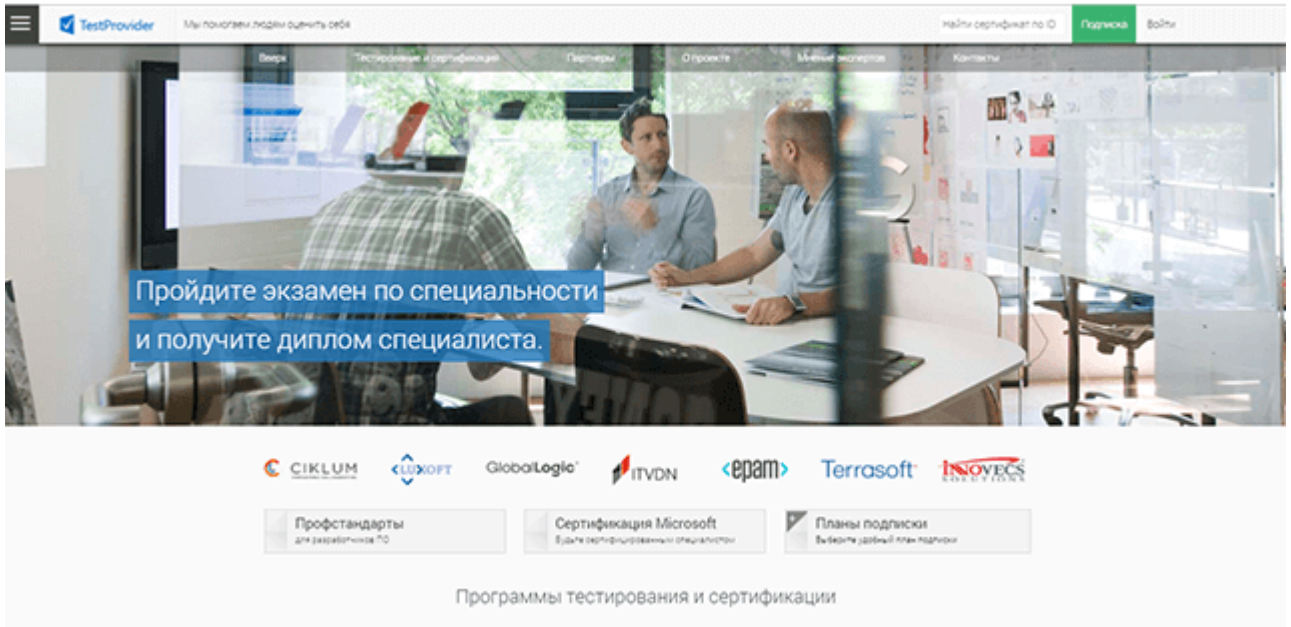
Посмотрите этот урок в видео формате на образовательном портале ITVDN.com для закрепления пройденного материала.

Курсы записаны сертифицированными тренерами, которые работают в учебном центре CyberBionic Systematics и другими высококвалифицированными разработчиками.

ITVDN
IT VIDEO DEVELOPERS NETWORK

Проверка знаний

TestProvider.com



TestProvider – это online сервис проверки знаний по информационным технологиям. С его помощью Вы можете оценить Ваш уровень и выявить слабые места. Он будет полезен как в процессе изучения технологии, так и для общей оценки знаний IT специалиста.

После каждого урока проходите тестирование для проверки знаний на TestProvider.com

Успешное прохождение финального тестирования позволит Вам получить соответствующий Сертификат.

TestProvider

Информационный видеосервис для разработчиков программного обеспечения

ITVDN

IT VIDEO DEVELOPERS NETWORK

TestProvider PROMETRIC TEST CENTER CyberBionic systematics Microsoft Partner Silver Learning Windows Azure Cloud Partner EBA European Business Association