Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Уральский федеральный университет
имени первого Президента России Б.Н. Ельцина»

Институт естественных наук и математики
Департамент математики, механики и компьютерных наук

Исследование потокобезопасных неблокирующих структур данных

«Допущен к защите»

Директор департамента
к.ф.-м.н., доцент
Асанов М. О.

_____
«___»_____2017 г.

Квалификационная работа на соискание степени бакалавра наук по направлению «Фундаментальная информатика и информационные технологии»
студента группы ФТ-401 (МЕН-430802)
Сваловой А. А.
Научный руководитель:
Ассистент департамента
Плинер Ю. А.

Екатеринбург
2017 год

Оглавление

Реферат

Свалова А. А. ИССЛЕДОВАНИЕ ПОТОКОБЕЗОПАСНЫХ НЕБЛОКИРУЮЩИХ СТРУКТУР ДАННЫХ, квалификационная работа: стр. 23, табл 1.

Ключевые слова: Многопоточность, структуры данных, Блокирующие и неблокирующие алгоритмы.

В данной работе описаны некоторые потокобезопасные неблокирующие структуры данных. Для этих структур данных приведены сценарии и результаты тестирования.

Введение

Технический прогресс в области электронных устройств привёл к появлению многоядерных процессоров и многопроцессорных систем. Одной из основных задач программистов стало написание программ, способных эффективно использовать эту особенность и выполняться на нескольких ядрах и процессорах. В наше время всё чаще возникает потребность в использовании специальных алгоритмов и структур данных для этого. Последним и посвящена эта работа.

Процессом называется последовательное выполнение инструкций заданной программы [1].

Многопоточностью называют определённое свойство процесса операционной системы. Многопочтоный процесс может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка [1].

Поток выполнения или поток — это подпроцесс, выполняющийся в контексте полноценного процесса. Потоки одного процесса работают параллельно, не зависимо друг от друга, но могут синхронизировать свою работу при необходимости. [1].

В общем случае использование многопоточности помогает ускорять выполнение вычислительных задач. Однако за это приходится платить проблемами, связанными с доступом к общим ресурсам (например, памяти) из разных по-

токов. Во всех современных операционных системах потоки одного процесса могут иметь доступ к общей памяти, что позволяет им обмениваться данными.

Если несколько потоков одновременно захотят изменить общую память, может возникнуть одна из ошибок параллельного программирования: потерянное обновление, грязное чтение, неповторяющееся чтение, чтение фантомов или состояние гонки [2]. В качестве примера можно рассмотреть следующий процесс: два потока хотят прибавить единицу к некоторой общей переменной. Они считывают значение переменной в локальную память, прибавляют единицу к полученному значению и по очереди в случайном порядке перезаписывают значение в общей памяти на значение из локальной памяти. В итоге переменная увеличивается только на единицу, а не на двойку, как ожидалось.

Чтобы разрешить эту проблему, необходимо разделение доступа к общей памяти и другим разделяемым ресурсам.

Синхронизация потоков — это механизм, позволяющий обеспечить корректную работу из нескольких потоков с общим ресурсом при условии, что доступ к нему может потребоваться в случайном порядке [3]. Общим ресурсом может выступать файл на файловой системе, общая память, открытый сокет и т. д.

Синхронизацию можно реализовать несколькими способами, но самым простым из них является блокировка. Каждый раз, когда поток хочет осуществить операцию с ресурсом, он блокирует этот ресурс. В любой момент времени только один птоок может владеть блокировкой заданного ресурса. После окончания работы с ресурсом блокировка снимается. Если другому потоку

нужен доступ к общему ресурсу, защищённому блокировкой, то выполнение этого поток останавливается до тех пор, пока блокировка не будет снята. [4].

Основное преимущество блокировки как механизма синхронизации заключается в его простоте. Большинство современных операционных систем содержат низкроуровневую реализацию блокировок — мьютексы и семафоры [1]. Благодаря этому в арсенале программиста оказывается простой и мощный инструмент для организации блокировок.

Однако у блокировок есть и минусы:

- если одновременно много потоков хотят получить доступ к общему ресурсу, то в процессе возникает «узкое горлышко» — все потоки ожидают снятия блокировки, а выполнение процесса практически полностью приостанавливается [5];
- если внутри блокировки выполняется достаточно большой участок программы, теряется основной смысл многопоточности. Внутри таких участков в каждый момент времени может находиться только один поток, а мы теряем преимущества многопоточности, когда, например, одна и та же задача для разных объектов может выполняться одновременно;
- возможна ситуации, когда поток захватил ресурс A и ждет освобождение ресурса B, в то время как второй поток захватил ресурс B и ждет освобождения ресурса A. Такая ситуация называется взаимной блокировкой (англ. deadlock [6], [7]). Программа останавливает свое выполнение и не может разрешить эту ситуацию без вмешательства извне.

Наличие недостатков блокировок привели исследователей к необходимости создания других методов синхронизации. Одним из них является неблокиру-

ющая синхронизация.

Неблокирующая синхронизация это метод синхронизации, в котором потоки не могут используть блокировки.

Синхронизация при этом обычно осуществлется с помощью специальных аппаратно-поддерживаемых атомарных операций [8]. Примером такой операции является Compare-And-Change от трёх аргументов x, y и z. Она сравнивает x с y и если они равны, то выполняет присваивание $x := z$. Процессором гарантируется, что эта операция будет выполнена атомарно, то есть между сравнением и присваиванием не может быть выполнена никакая другая операция. Поток может узнать, выполнилось ли присваивание в операции Compare-And-Change. Именно это свойство позволяет строить работающие неблокирующие синхронизации. Подробнее об атомарных операциях рассказывается в первой главе.

Область исследования неблокирующих алгоритмов является активно развивающейся, так как не всегда можно достаточно просто переписать блокирующую версию многопоточного алгоритма так, чтобы вместо блокировок использовать только неблокирующие атомарные операции. Для некоторых даже простых структур данных до сих пор не придуманы неблокирующие способы реализации. Примером такой структуры данных является обычный массив.

Однако задача придумывания неблокирующих алгоритмов имеет под самой практическую значимость: часто они оказываются быстрее блокирующих аналогов. Правда, на практике это нередко зависит от деталей реализации, области применение и др.

Объектом исследования данной работы являются неблокирующие алгорит-

мы для работы со структурами данных. Они описаны во второй главе этой работы.

Цели работы:

- изучить неблокирующие алгоритмы, реализующие интерфейс, включающий добавление элемента в множество, удаление и проверку на принадлежность к множеству;
- адаптировать алгоритмы, разработанные под языки программирования с неуправляемой памятью, к языкам с управляемой памятью;
- реализовать эти алгоритмы C#;
- проверить с помощью тестов производительности, являются ли рассмотренные неблокирующие алгоритмы эффективнее аналогичных блокирующих.

Для сравнения были выбраны следующие структуры данных, реализующие описанный выше функционал множества: сортирующийся лист, хеш-таблица, список с пропусками и самобалансирующееся дерево поиска. Для сравнения неблокирующих реализаций с блокирующими взяты реализации этих же структур данных из стандартной библиотеки .NET. О методиках тестировании и его результатах рассказывается в третьей главе.

В открытом репозитории GitHub [9] приведены результаты сравнений этих структур данных и разные варианты использования их на практике.

Глава 1

Общие сведения

1.1 Классификация алгоритмов и атомарная операция

Неблокирующие алгоритмы традиционно можно разделить на три класса: Wait-free, Lock-free и Obstruction-free алгоритмы [10].

В Wait-free алгоритмах каждый поток совершает операции за конечное число шагов вне зависимости от других потоков. Данное ограничение является очень сильным, поэтому, к сожалению, редко достигается.

В Lock-free алгоритмах конкретный поток может остановиться, но гарантируется, что система в целом двигается вперёд. Если какой-то поток не смог выполнить операцию, значит, что какой-то другой поток смог выполнить свою операцию, следовательно, в целом система продвинулась.

В третьем типе каждый может выполнить каждую операцию за конечное количество шагов, если ничего ему не помешает. В данном случае может случиться ситуация, когда ни один из потоков не двигается вперед, однако ни один заблокированный поток не может мешать работе всех остальных потоков, следовательно, это все равно более сильная гарантия, чем блокирущие алгоритмы.

Это самое сильное требование из-за чего редко реализуемое, однако существует алгоритм, преобразующий Obstruction-free в Wait-free [11]

Каждая из этих реализаций использует абстракцию «атомарная операция» - это операция, которая либо не выполняется совсем, либо выполняется как единое целое. В данной работе используется атомарная операция Compare And Swap (CAS) (Рис 1.1). Эта операция сравнивает две ссылки и, если они равны, меняет одну из них на новую. Эта операция предоставляется большинством операционных систем и уже встроена в язык C#.

```
CAS(ref reference, newReference, comparand)
```

Рис. 1.1: CAS сравнивает reference с comparand и, если они равны, заменяет reference на newReference

1.2 Известные алгоритмы и их особенности

Первой структурой данных, для которой был создан неблокирующий алгоритм вставки, удаления и проверки на принадлежность, был стек. Первое упоминание потокобезопасных неблокирующих структур данных было описано в статье Трейбера в 1986 году [12]. Это самый простой и самый известный алгоритм на данный момент. В данном алгоритме может возникнуть ситуация, когда большое количество потоков единовременно пытается изменить один и тот же ресурс. Тогда каждый поток в цикле будет пробовать атомарно изменить ресурс и при успешной попытке выходить из цикла. Чтобы уменьшить количество таких попыток используется стратегия Back-off - стратегия, при которой в случае неуспешной попытки применения атомарной операции поток на какое-то время засыпает или пытается сделать какую-то другую полезную работу, а потом снова пытается повторить атомарную операцию. Это

позволяет уменьшить давление на критические данные контейнера при большой нагрузке. Данная стратегия очень часто встречается в неблокирующих алгоритмах. Количество времени, на которое нужно заснуть, а так же выбор полезной работы выбирается в каждом конкретном случае индивидуально.

Другой не менее известный алгоритм - алгоритм очереди, описанный в статье Михаеля и Скотта в 1996 году [13]. В нем для поддержания двух ссылок на «голову» и на «хвост» используется еще одна популярная стратегия - вставка ложного (sentinel) элемента специальной структуры, который не хранит в себе никакого значения, а нужен исключительно для удобства использования и более простого описания алгоритма.

В двухтысячных годах тема стала популярной и начали появляться улучшения этих алгоритмов, а также создаваться новые алгоритмы других структур данных, таких как: списки[14], списки с пропусками [15], хеш-таблицы [16], некоторые деревья и др. Однако не все структуры данных можно реализовать неблокирующими. В таких случаях используют подход, объединяющий блокирующий и неблокирующий. Блокировка захватывается только на отдельные небольшие участки памяти, независимо друг от друга. В таких алгоритмах особо часто используемые операции производятся без блокировок, а некоторые операции производятся с блокировками, но на маленькие участки памяти. Такие алгоритмы оказываются проще в понимании и доказательстве, но не проигрывают в эффективности и применимости.

Один из способов такой локальной блокировки - это добавление особого маркера в ссылку на объект. Так, если один поток смог атомарно изменить ссылку на некоторый объект, пометив ее этим маркером, все остальные потоки понимают, что данный объект используется в какой-то операции и его нельзя

изменять.

В языках с неуправляемой памятью такой способ легко осуществим благодаря выравниванию указателей на объект. При выделении памяти компилятор, обычно, выравнивает длину указателя на максимально большой тип данных. Поэтому в указателе остаются реально неиспользуемые биты, которые можно как раз и использовать в качестве маркера.

В языках с управляемой памятью разработчик не имеет доступа к ссылке, поэтому стоит придумывать способы сымитировать эту ссылку с помощью объектов. В данной работе реализован примитив маркируемой ссылки (Atomic Markable Reference). Он будет хранить в себе текущее состояние этой ссылки (Рис 1.2). Само же состояние будет состоять из непосредственно ссылки и описанного выше маркера. В данном случае маркер как раз отвечает за неиспользуемые биты указателя. Теперь можно атомарно изменять ссылку на состояние, что равносильно изменению помеченного указателя. Изменяя этот примитив, можно сымитировать изменение указателя, что позволяет также использовать алгоритмы с локальными блокировками.

```
public class AtomicMarkableReference<TReference, TMark>
{
        volatile State state;

        private class State
        {
                TReference Reference;
                TMark Mark;
        }
}
```

Рис. 1.2: AtomicMarkableReference

1.3 Проблема ABA

При неправильной организации кода в неблокирующих алгоритмах могут возникнуть проблемы. Самая известная — проблема ABA [17]. Эта проблема возникает, когда множество потоков обращаются к разделяемой памяти поочерёдно. Алгоритм может рассчитывать, что если атомарная операция завершилась успехом - значит память не менялась. Но CAS всего лишь гарантирует, что ее значение в момент, когда он ее перезаписал, было равно тому значению, которое ему передано как ожидаемое. Если из списка удалить элемент, уничтожить его, а затем создать новый элемент и добавить обратно в список, есть вероятность, что новый элемент будет размещён на месте старого. Указатель на новый элемент совпадёт с указателем на старый, что и приведёт к проблеме. Такая проблема может возникнуть только в языках программирования с неуправляемой памятью. В языках программирования с управляемой памятью существует абстракция «сборщик мусора», которая обеспечивает отсутствие потери и повторного использования ссылок.

1.4 Set

Set — это коллекция для хранения неупорядоченного множества уникальных объектов. Это аналог математического понятия множество Эта структура данных позволяет добавлять элементы, удалять элементы и быстро проверять, существует ли уже такой элемент (Рис 1.3).

Чаще всего set применяют для объединения объектов с какими-то общими свойствами. Эта структура данных близка к map, которая используется повсеместно. Пример: индексы в базах данных.

```
public interface ISet<TElement>
{
        bool Add(TElement element);
        bool Contains(TElement element);
        bool Remove(TElement element);
}
```

Рис. 1.3: Интерфейс ISet

Хотя операции в set могут быть реализованы произвольным способом, но чаще всего его используют для быстрой проверки принадлежности элемента (черные, белые списки, базы данных). Поэтому в данной работе будет сделан акцент на быстром поиске. Быстрое добавление тоже будет играть роль, поэтому придется пожертвовать скоростью удаления.

Глава 2

Реализации алгоритмов

В данной главе будут приведены краткие описания неблокирующих алгоритмов, проблемы, которые они решают и сложности реализации. Полное описание алгоритмов можно найти в списке литературы [14], [15], [18], [16], [19]. Стандартные реализации однопоточных алгоритмов общедоступны, поэтому не будут описаны в данной работе. Познакомиться с ними можно, например, в книге «Структуры данных и алгоритмы» [20].

2.1 Односвязный список

Пусть односвязный список состоит из элементов, в каждом из которых есть значение этого элемента и ссылка на следующий элемент (Рис 2.1). Поиск элемента не будет рассмотрен, так как он совпадает с поиском в обычном сортирующимся списке.

```
public class LinkedListNode<TElement>
{
        TElement Element;
        LinkedListNode<TElement> Next;
}
```

Рис. 2.1: Вершина списка

Если реализовать добавление как в однопоточном варианте, то возможна проблема при одновременном добавлении двух последовательных элементов. Пусть, есть список с элементами 1-2-5 (Рис 2.2(a)). Поток А хочет вставить элемент 3, поток Б - элемент 4. Поток А понимает, что ему нужно вставить элемент между 2 и 5. Он запоминает ссылку на предыдущий и следующий элементы и в этот момент операционная система передает управление потоку Б (Рис 2.2(b)). Поток Б также находит место для вставки и тоже запоминает 2, как предыдущий элемент, 5, как следующий (Рис 2.2(c)). После этого он переписывает ссылку «Следующий» у элемента 2 на новосозданный элемент 4, а у элемента 4 на 5 (Рис 2.2(d)). Управление возвращается к потоку А. Он перезаписывает ссылку «Next» предыдущего элемента (2) на новосозданный элемент 3, а ссылку элемента 3 на следующий элемент (5) (Рис 2.2(e)). В результате элемент 4 «потеряется», т. е. не будет ни одной ссылки, указывающей на него.

Реализация неблокирующего доступа использует типичный прием для неблокирующих алгоритмов - вечный цикл с операцией CAS и применение стратегии Back-off, описанной ранее. На каждом шаге цикла алгоритм пытается найти два элемента а и б, между которыми должен быть вставлен новый, и атомарно перезаписать ссылку «Next» с предыдущего элемента (а) на новый, при этом сравнивая, является ли эта ссылка до сих пор ссылкой на следующий (б). Алгоритм выходит из цикла, когда попытка замены ссылки происходит успешно (Рис 2.3). Такая реализация полностью решает вышеописанную проблему. При попытке перезаписать ссылку элемента 2 с 5 на 3 (Рис 2.2(e)), CAS завершается неудачей, потому что ссылка уже не на 5, а на 4. Алгоритм заново находит соседние элементы, и следующий уже не 5, а 4. На этом шаге

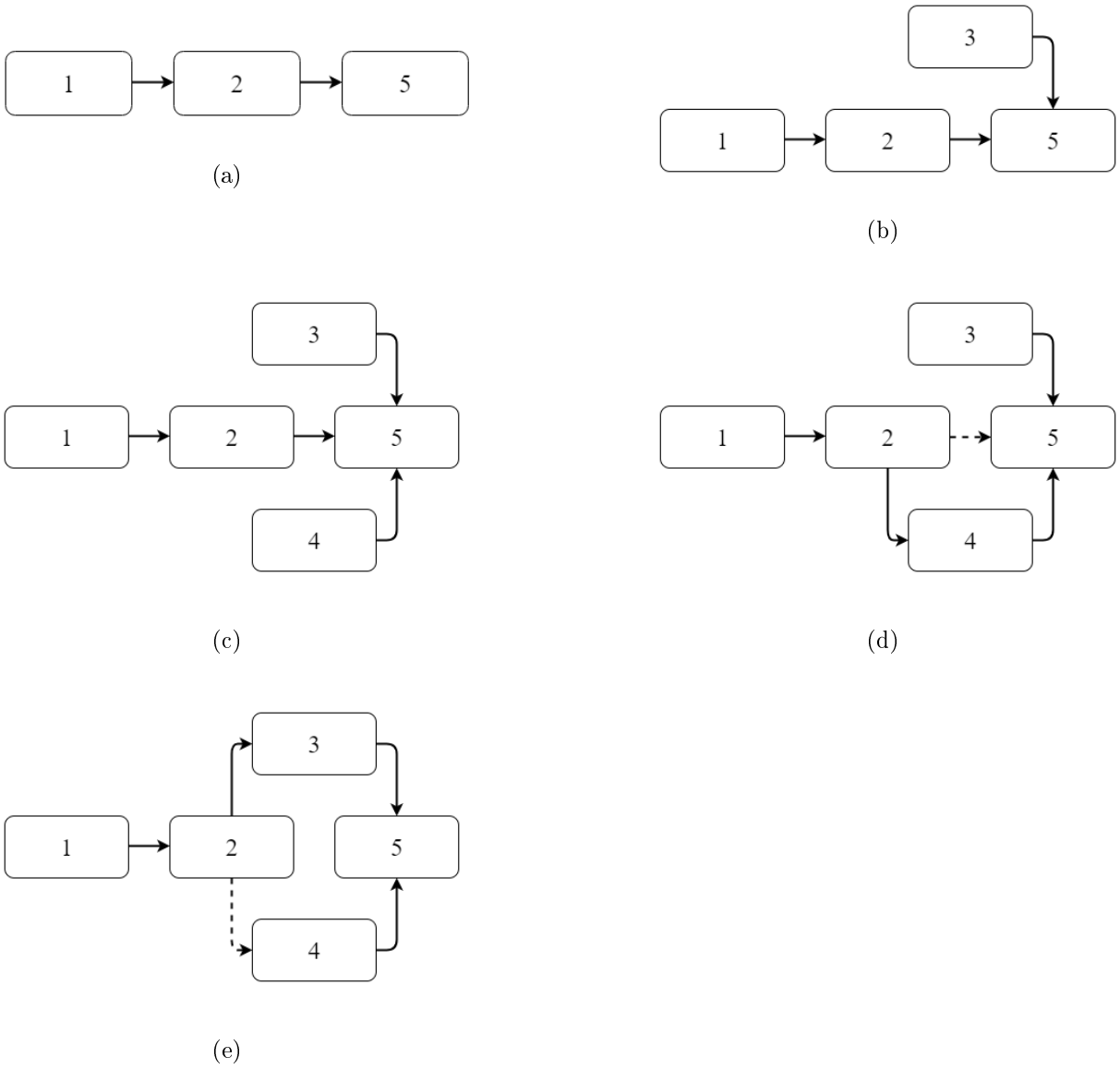

Рис. 2.2: Одновременное добавление в 2 потоках: a) начальное состояние, b) А поставил ссылку на 5, c) Б поставил ссылку на 5, d) А сменил ссылку «Next» у 2, e) Б сменил ссылку «Next» у 2.

цикла CAS уже выполняется успешно. Оба элемента вставлены правильно.

Еще одна проблема может возникнуть при одновременным удалении и вставке двух последовательных элементов. Пусть есть список 1-2-4 (Рис 2.4(a)). Поток А хочет добавить элемент 3, поток Б удалить элемент 2. Поток Б запоминает, что предыдущий элемент 1, следующий 4. Управление передается потоку А. Поток А вставляет элемент 3, как это было описано ранее (Рис 2.4(b)).

```
while (true)
{
        1) (predsessor, subsessor) = FindPlace(newElement);
        2) newElement.Next = subsessor;
        2) if (CAS(ref predsessor.Next, newElement, subsessor));
              break
}
```

Рис. 2.3: Шаг цикла. 1) находим соседние элементы, между которыми нужно вставить новый, 2) ссылку Next у нового элемента поместим на subsessor 3) пытаемся атомарно вставить

Управление возвращается к потоку Б. Он атомарно заменяет ссылку «Next» у элемента 1 на элемент 4 (Рис 2.4(c)). В итоге элемент 3 «потерялся».

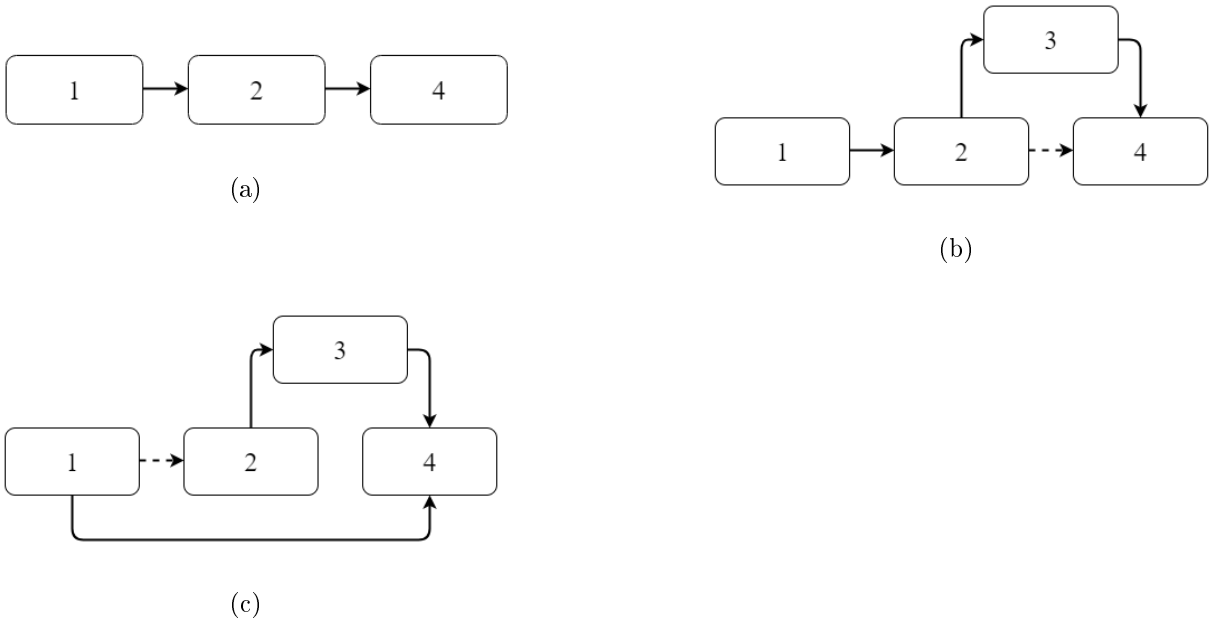

Рис. 2.4: Одновременное добавление и удаление в 2 потоках: a) начальное состояние, b) А поставил ссылку на 4, c) Б сменил ссылку с 2 на 4.

Для устранения этой проблемы можно ввести дополнительный флаг в ссылку на объект (Рис 2.5).

Теперь объект удаляется в два шага:

- пометить как удаленный, но не удалить
- физически удалить.

```
public class LinkedListNode<TElement>
{
        TElement Element;
        AtomicMarkableReference<LinkedListNode<TElement>, Flag> Next;
}
```

Рис. 2.5: AtomicMarkableReference - структура, описанная в предыдущей главе, Flag может быть никаким или помеченным

Можно заметить, что при помечивании ссылки на удаленный элемент, ситуация, изображенная на рисунке 5, существенно не изменится. Однако, при помечивании ссылки «Next» у удаляемого объекта, можно избежать потерь элементов. Теперь при вставке тройки из предыдущего примера ссылка «Next» у 2 уже будет помеченной. Это будет сообщать о том, что элемент в данный момент удаляется, а значит, манипулировать этой ссылкой пока что нельзя, надо заново перейти на новый виток в цикле и заново определить соседей. В итоге проблемы, описанной ранее при одновременной вставке и удалении не случится.

Данный алгоритм в описанной ранее классификации является lock-free, так как один поток может помешать работе других потоков, но это будет значить, что он сам выполнил свою работу.

2.2 Улучшенный односвязный список

Вышеописанная реализация односвязного списка является неблокирующей, что, возможно, может ускорить работу программы, однако у нее до сих пор существует недостаток: если операции удаления происходят достаточно часто, то операции вставки будут также часто заканчиваться не успехом, из-за чего они каждый раз будут начинать сначала. В результате в худшем случае может получиться, что программа каждый раз заново пробегает весь список.

Чтобы устранить эту проблему можно ввести еще две дополнительных абстракции. В ссылку «Next» добавить еще один флаг, который будет свидетельствовать, что следующий элемент в данный момент на стадии удаления. В сам элемент нужно добавить поле «Backlink», который будет указывать на предыдущий элемент, который еще не участвует в удалении (Рис 2.6). Теперь

```
public class LinkedListNode<TElement>
{
        TElement Element;
        LinkedListNode<TElement> Backlink;
        AtomicMarkableReference<LinkedListNode<TElement>, Flag> Next;
}
```

Рис. 2.6: AtomicMarkableReference - структура, описанная в предыдущей главе, Flag может быть никаким, или помеченным на удаление, или помеченным на невозможность удаления

операция удаления будет проходить не в два, а в три этапа. Между двумя этапами из предыдущего алгоритма появится новый этап. Теперь после помечивания удаляемой вершины на удаление (Рис 2.7(a)) алгоритм добавляет в ссылку «Next» у предыдущей вершины новый флаг, который будет обозначать, что в данный момент вершина участвует в удалении, и ее саму удалять нельзя. У удаляемой вершины алгоритм устанавливает ссылку «Backlink» на ближайшую предыдущую вершину, которая еще не помечена новым флагом (Рис 2.7(b)). Теперь каждый раз, когда вставка не может завершиться успехом, поток будет по ссылкам «Backlink» возвращаться не в самое начало, а в первую вершину, следующая за которой еще не удаляется. Это позволяет еще немного ускорить работу программы, так как при каждой неудачной вставке, возможно, больше не нужно проходить лист полностью заново.

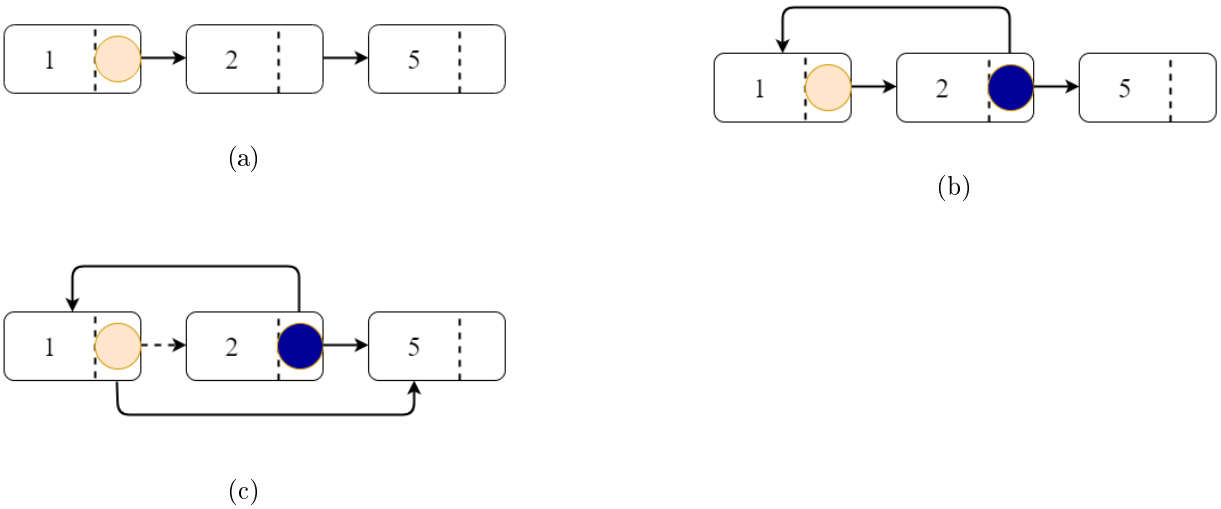

Рис. 2.7: Удаление в 3 этапа: a) помечивание на участие в удалении, b) помечивание на удаление, c) реальное удаление.

2.3 Список с пропусками

Список с пропусками в своей структуре содержит несколько самосортирующихся односвязных списков. Поэтому алгоритм неблокирующего списка с пропусками будет использовать все те же идеи, что и неблокирующий односвязный список. Остается разобраться, как применить все те же идеи, но вставляя и удаляя не 1 элемент, а сразу столбец.

Вставка, как и в однопоточном варианте осуществляется снизу-вверх (Рис 2.8). Однако, в данном случае на каждом уровне приходится искать заново, иначе можно запомнить элемент, который какой-то другой поток уже удалил.

Удаление тоже происходит, начиная с удаления вершины на самом нижнем уровне. Этого действия достаточно, чтобы весь столбец считался удаленным. При каждом следующем поиске по списку нужно проверять не удалена ли текущая вершина, а удалена ли ее вершина с первого уровня (вершины реально не удаляются из памяти, но на них больше никто не ссылается, поэтому можно считать, что они больше не принадлежат к списку, так как они недостижи-

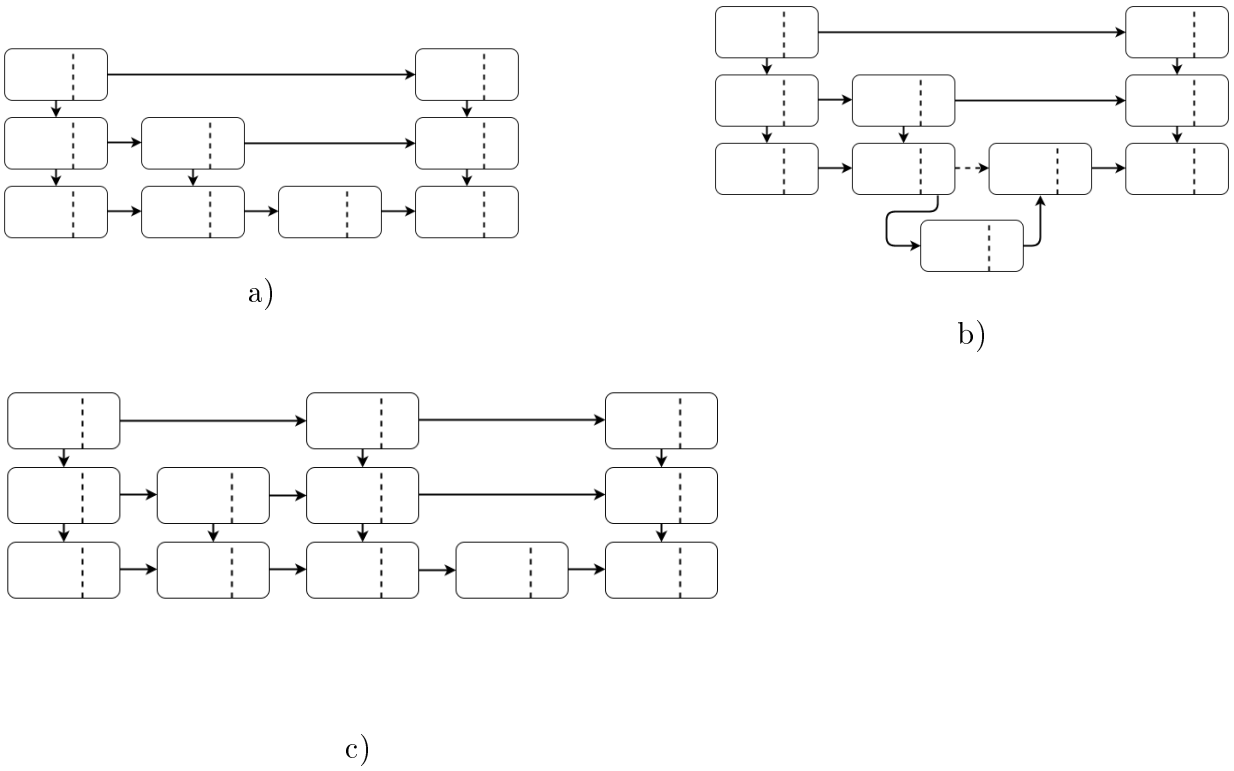

Рис. 2.8: Добавление в список с пропусками: a) начальное состояние, b) добавление элемента на самом нижнем уровне, c) добавление остальных элементов.

мы). Если вершина с первого уровня удалена, то нужно удалить и текущую вершину, а также больше не ссылаться на нее и не строить из нее ссылки на новые вершины (Рис 2.9).

Списки внутри списка с пропусками можно также улучшить с помощью второго алгоритма односвязного списка.

Так как список с пропусками основан на односвязных списках, то он тоже является lock-free.

2.4 Хеш-таблица

Хеш-таблица - структура данных, основыванная на массиве с произвольным доступом. На данный момент не придумано алгоритма, как можно реализовать строгий параллельный доступ к одной ячейке памяти на запись.

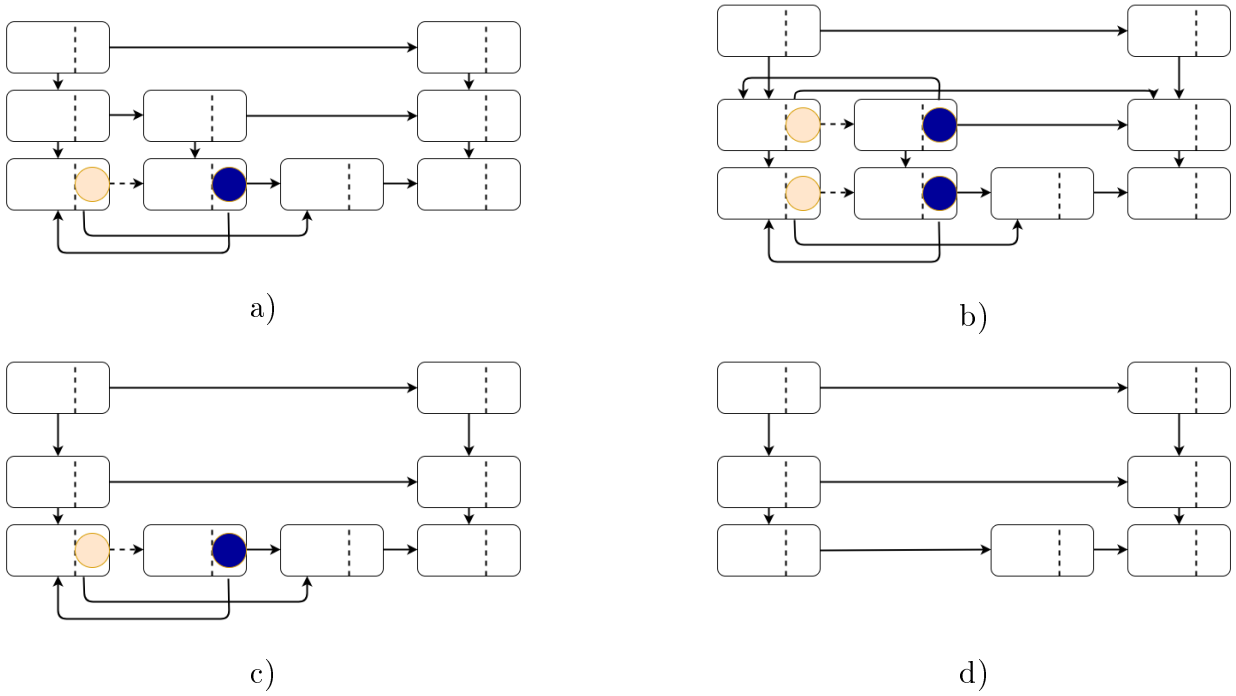

Рис. 2.9: Удаление из списка с пропусками: a) логическое удаление вершины на нижнем уровне, b) логическое удаление вершин на остальных уровнях, c) физическое удаление вершин на уровнях, выше первого, d) физическое удаление вершины на первом уровне.

Однако блокировать каждый раз весь массив, очевидно, неправильно. В таких случаях используют другой подход.

Весь массив разбивают на кусочки. Чаще всего используют куски одинаковой длины, распределяя их равномерно по массиву. Куски могут пересекаться или не пересекаться (Рис 2.10). В технике striping имеется два массива — массив блокировок и хеш-таблица. Во время операции модификации высчитывается нужный хеш, находится место в массиве, где этот элемент должен быть изменен, и блокируется только тот кусок, которому принадлежит данный элемент. После модификации блокировка отпускается.

При увеличении количества элементов количество кусков списки коллизий становятся очень большие и все преимущества хеш-таблиц становятся невозможными. Для расширения хеш-таблицы на весь массив в массиве блокировок захватывается блокировка, массив хеш-таблицы обновляется, после чего все

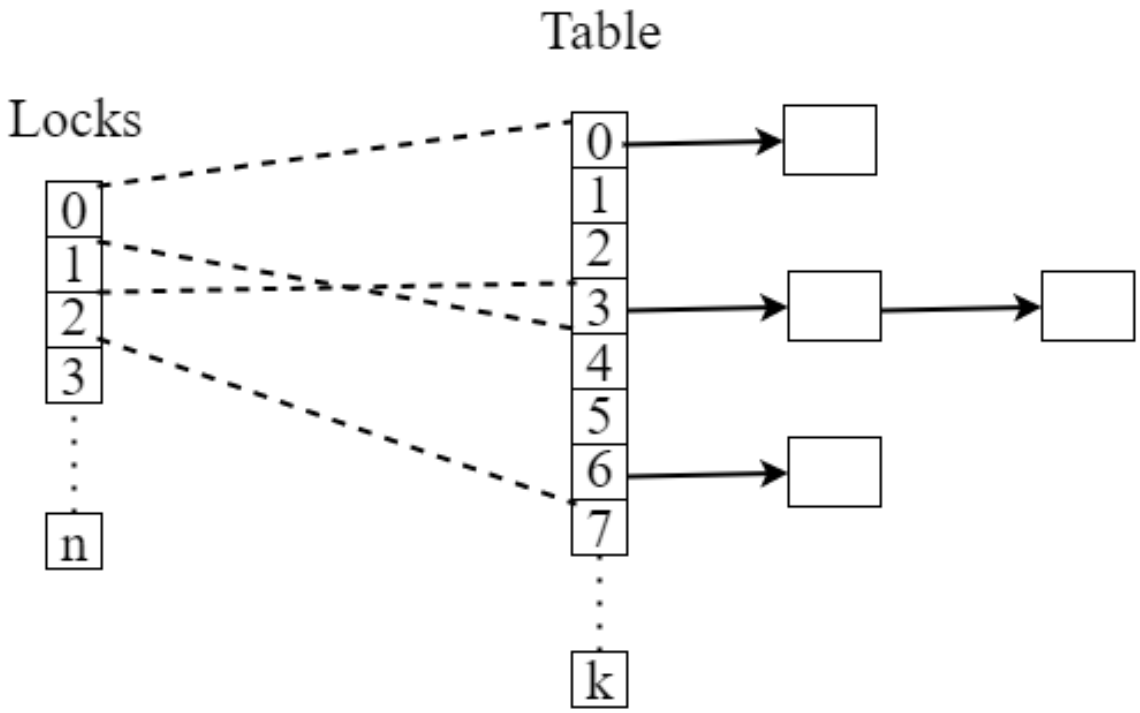

Рис. 2.10: Хеш-таблица. Слева массив блокировок, справа сама хеш-таблица

блокировки освобождаются. Важно заметить, что массив блокировок остается неизменного размера, но с каждым увеличением массива хеш-таблицы длина каждого куска увеличивается.

Такой подход не обеспечивает неблокирующего доступа, этот алгоритм хеш-таблицы является комбинацией неблокирующего и блокирующего алгоритмов. Так же он сильно зависит от длины массива блокировок. Если он слишком большой, то каждый его элемент покрывает очень маленький кусок в основном массиве, что близко к блокировке каждого элемента в основном массиве. Если он слишком маленький, то каждый его элемент покрывает слишком большой кусок в основном массиве, что близко к блокировке всего основного массива.

Глава 3

3. Тестирование

3.1 Модульное тестирование

Тестирование параллельного алгоритма всегда трудно. Приходится каждый раз задумываться о том, как будут работать те или иные функциональности вместе. В данной работе использовано два основных подхода. Первый заключается в изолированном тестировании каждого метода в одном потоке. Тестирование параллельного кода всегда должно начинаться с проверки функциональности в однопоточном искусственно-созданном окружении. Далее проверяются простые сценарии в нескольких потоках, чтобы проверить, что они вообще корректно взаимодействуют между собой, т. е. делают то, что от них «ожидают» в каждом конкретном сценарии.

Второй подход: «тестирование грубой силой». В этом случае запускается большое количество потоков или выполняется большое количество операций одновременно. При увеличении числа операций вероятность ошибки увеличивается, в этом и заключается данный метод. Однако даже это не гарантирует, что программа работает правильно. В некоторых случаях сценарии неправильной работы кода настолько редки, что можно вообще никогда их

не получить ошибку в тестировании.

Отсюда плавно вытекает третий подход: тестирование аналитически. Он заключается в тщательном продумывании всех возможных сценариев, проверки каждой строчки кода, попытки смоделировать выполнение программы и найти потенциальные ошибки. Однако большая проблема данного подхода: «человеческий фактор». Иногда такую проверку все же можно сделать формально и наглядно. В данной статье в ссылках на алгоритмы приведен подробный анализ корректности работы алгоритма, заключающийся в разборе всех возможных сценариев.

3.2 Тестирование производительности

Если тестирование работоспособности алгоритма нужно для проверки, что алгоритм работает корректно, то оценка производительности нужна для представления, реализуем ли этот алгоритм на практике. Проверяется, соответствует ли время работы или количество используемой памяти теоретическим оценкам. Чаще всего такое тестирование проходит в сравнении с эталоном. В данной работе эталоном представлялся дополнительный алгоритм, основанный на блокировании структуры. Ожидается, что на большом количестве потоков, этот алгоритм будет работать в среднем хуже, чем неблокирующий алгоритм.

Для оценки производительности можно использовать различные метрики. В данной работе рассматривалось несколько различных тестовых окружений, в каждом из которых все реализованные структуры данных и варианты работы в 1, 2 и 8 потоках.

В качестве тестового окружения был выбран компьютер Intel Core i7-4790

CPU 3.60GHz (Haswell), ProcessorCount=8 Frequency=3507505 Hz, Resolution=285.102 ns, Timer=TSC. Операционная система: Windows 10. C# Clr 4.0.30319.42000, 64bit LegacyJIT/clrjit-v4.6.127.1;compatjit-v4.6.1055.0. В качестве тестовых сценариев были выбраны:

- вставка, удаление, поиск по-отдельности
- только вставка и поиск в соотношении 9:1
- вставка, поиск и удаление в соотношении 2:7:1

3.3 Результаты

Полные результаты представлены по ссылке [9].

Из результатов можно отметить, что неблокирующие структуры данных, действительно, с ростом количества потоков и числа обрабатываемых элементов часто работают лучше, чем блокирующие аналоги. Например, поиск в 8 потоках 100000 элементов неблокирующая хеш-таблица осуществляет за 2,894.9 мксек., а блокирующая за 5,222.1 мксек., добавление и поиск 10000 элементов в блокирующей реализации односвязного списка осуществляется за 625.1 мксек., а в блокирующем за 656.7 мксек.

Также можно привести некоторые рекомендации использования. Если чаще всего происходит операция проверки на принадлежность, то быстрее всего в среднем работает хеш-таблица, как и в неблокирующей реализации. Хуже всего поиск осуществляется в односвязном списке, опять как в неблокирующей реализации. При частой вставке элементов лучше всего работает односвязный список, так как он меньше всего задействует вспомогательных элементов структуры. Хуже всего работает хеш-таблица. При частых опе-

рациях лучше всего работает хеш-таблица, потому что при малых коллизиях очень редко приходится синхронизировать параллельные потоки. Если вставка и поиск происходят примерно в пропорциях 1:9, то список с пропусками и односвязный список справляются примерно одинаково, хеш-таблица тоже незначительно проигрывает. А вот при вставке, поиске и удалении в соотношении 2:7:1 неожиданно выигрывает односвязный список, однако все остальные структуры данных осуществляют эти операции за такой же порядок времени.

Заключение

Результатом данной работы является готовый протестированный модуль с реализованными потокобезопасными неблокирующими структурами данных. Были выбраны алгоритмы, способные реализовать интерфейс set и адаптированы под язык с управляемой памятью. Также были произведены тысты производительности, которые показали практическую применимость выбранных алгоритмов.

Литература

[1] Эндрю Таненбаум Х. Бос. Современные операционные системы. СПб.: Питер, 2005. 1038 с.

[2] Race conditions. Официальная документация microsoft. https://support.microsoft.com/en-us/help/317723/description-of-race-conditions-and-deadlocks.

[3] С. Шляхтина. Синхронизация данных. КомпьютерПресс. №7, 2005.

[4] Lock. Официальная документация C#. https://msdn.microsoft.com/en-us/library/c5kehkcz$v = vs$.100.aspx.

[5] Эндрюс. Грегори Р. Основы многопоточного, параллельного и распределённого программирования. Вильямс, 2003.

[6] Nima Kaveh W. E. Deadlock Detection in Distributed Object Systems. 2001.

[7] JoAnne L. Holliday A. E. A. Distributed Deadlock Detection.

[8] Preshing Jeff. An Introduction to Lock-Free Programming. Issue #29 of Hacker Monthly, 2012. 06.

[9] Андреевна Свалова Анастасия. Исходный код и результаты работы. 2017. urlhttps://github.com/CodeDonkeys/Lockness.

[10] Herlihy M. Wait-free synchronization // ACM Transactions on Programming Languages and Systems. 1991. p. 124–149.

[11] Fich F. L. V. M. M. S. N. Obstruction-free algorithms can be practically wait-free. // ACM Journal. p. 78–92.

[12] Treiber R. Systems programming: Coping with parallelism. International Business Machines Incorporated // Thomas J. Watson Research Center. 1986.

[13] Scott M. M. M. M. L. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. 1996.

[14] Harris T. L. A Pragmatic Implementation of Non-Blocking Linked-Lists // Proceedings of the 15th International Symposium on Distributed Computing. 2001. P. 300–314.

[15] Mikhail Fomitchev E. R. Lock-Free Linked Lists and Skip Lists. 2003.

[16] Michael M. M. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets // IBM Thomas J. Watson Research Center.

[17] Dechev Damian; Pirkelbauer P. S. B. Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs. 2010.

[18] Fomitchev M. Lock-free linked lists ans skip lists. 2003. p. 124–149.

[19] Maurice Herlihy N. S. The Art of Multiprocessor Programming. Morgan Kaufmann, 2012. 552 p.

[20] Alfred V. Aho Jeffrey D. Ullman J. E. H. Data Structures and Algorithms. Addison-Wesley publishing company, 2003. 382 p.