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 к.ф.-м.н.
Плинер Ю. А.

Екатеринбург

2017 год

Содержание

1 Реферат

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

Ключевые слова:

В данной работе описаны некоторые потокобезопасные неблокирующие структуры данных, приведены сценарии и результаты тестирования.

2 Введение

С ростом прогресса многие электронные устройства становятся многопроцессорными, а сами процессоры почти что везде многоядерные. Поэтому задача программиста, как человека, который пытается максимально эффективно использовать предоставленные ресурсы, - писать программы, способные масштабироваться при добавлении новых ядер и процессоров и параллелиться на них. Поэтому сейчас все чаще и чаще пишут программы, способные распараллелить свои инструкции, и используют многопоточные структуры данных.

Многопоточность — свойство приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени [1].

Процесс - это выполнение последовательностей инструкций программы [1].

Поток выполнения (поток) - подпроцесс, или легковесный процесс (lightweight process), выполняющийся в контексте полноценного процесса [1].

При выполнении некоторых задач распараллеливание может помочь достичь более эффективного использования ресурсов вычислительной машины, однако такой код обладает рядом проблем, связанных с доступом к общим ресурсам разными потоками.

Потоки внутри одного процесса могут иметь доступ к одному и тому же участку памяти (адресному пространству). Это позволяет нескольким независимым потокам обмениваться данными. Если два или более потока захотят изменить этот участок единовременно, может возникнуть состояние гонки - ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выпол-

няются части кода. Например, оба потока хотят прибавить единицу к какой-то переменной. Оба потока считывают эту переменную в локальную память единовременно, прибавляют единицу в своей локальной памяти,и поочереди в случайном порядке перезаписывают эту переменную из локальной памяти. В итоге переменная увеличилась на 1, а не на 2, как ожидалось [2]. Возникает вопрос: как в таком случае контролировать доступ к этому ресурсу? Требуется, чтобы в каждый момент времени, способом, очевидным для разработчика, ресурсом владел только один поток, а все остальные каким-то образом ждали своей очереди.

Синхронизация процессов — это механизм, позволяющий обеспечить целостность какого-либо ресурса (файл, данные в памяти), когда он используется несколькими потоками в случайном порядке [3].

Этот механизм можно осуществить несколькими способами. Самый простой из них - блокировка. Каждый раз, когда поток хочет осуществить операцию с ресурсом, он захватывает блокировку на этот ресурс. В каждый момент времени только один объект может владеть этой блокировкой. Как только поток заканчивает работу с ресурсом, он освобождает блокировку. Если другому потоку будет нужен доступ к переменной, защищённой блокировкой, то этот поток блокируется до тех пор, пока блокировка не будет освобождена [4].

Плюсы такого механизма синхронизации:

- прост в понимании и реализации, так как в большинстве современных ОС уже существует низкоуровневая реализация блокировок (Мьютекс и семафор [1])
- решает описанную выше проблему.

Минусы:

- при большом количестве потоков, желающих получить доступ к ресурсу, возникает «узкое горлышко», т. е. место в программе, которое тормозит выполнение программы в целом [5].
- при существовании больших участков программы с блокировкой теряется весь смысл многопоточности. В эти участки может заходить только один поток, как и в однопоточном программировании, следовательно никаких преимуществ многопоточности не наблюдается
- возможны ситуации, когда один поток захватил первый ресурс и ждет освобождение второго ресурса, в то время как второй поток захватил второй ресурс и ждет освобождения первого. Такая ситуация называется взаимная блокировка (deadlock [6], [7]). Программа в таком случае останавливает свое выполнение совсем и не может без каких-либо вмешательств извне разрешить эту ситуацию.

Эти минусы привели исследователей к созданию других способов синхронизации. Один из них - неблокирующая синхронизация.

Неблокирующая синхронизация - это способ, при котором каждый поток пытается применить низкоуровневые атомарные аппаратные примитивы, а не использовать блокировки [8].

В таком способе синхронизации тоже можно добиться, чтобы в каждый момент времени выполнялась только одна операция, только одного потока. Все остальные операции в других потоках либо завершаются ошибкой, либо выполняются сразу следом за предыдущей. Такие алгоритмы обеспечивают общее продвижение программы в целом: даже если какой-то поток не смог

выполнить операцию или завершился с ошибкой - значит, что какой-то другой поток успешно выполнил свою операцию. Не существует случаев, когда все потоки одновременно простаивают, и, как частный случай этого, невозможно существование взаимных блокировок. В первой главе будет подробно рассказано об атомарных операциях.

Однако, несмотря на все преимущества, данная область является до сих пор развивающейся. Не всегда возможно просто переписать неблокирующую версию алгоритма, основанного на блокировках. В некоторых случаях до сих пор не придумано неблокирующих аналогов, например, неблокирующие массивы. Причина: каждый раз нужно творчество, чтобы свести все операции над разделяемым ресурсом к последовательности независимых атомарных операций, т. е. не существует универсального способа написания неблокирующей реализации. Однако часто такие алгоритмы оказываются быстрее аналогов, основанных на блокировках. Но это только в теории. На практике скорость работы зависит от конкретной реализации, области применения, часто встречающихся запросов и т. д.

Объект исследования данной работы - потокобезопасные неблокирующие алгоритмы структур данных. Цели работы:

- реализовать на языке C# неблокирующие версии структур данных, реализующие интерфейс, включающий добавление элемента в множество, удаление и проверку на принадлежность к множеству,
- адаптировать алгоритмы, разработанные под языки программирования без неуправления памятью, к языкам с управлением памятью,
- проверить с помощью тестов производительности, являются ли рассмотренные неблокирующие алгоритмы эффективнее аналогичных блокиру-

ющих.

В работе представлены структуры данных, реализующие описанный выше интерфейс. Для сравнения были выбраны следующие реализации: сортирующийся лист, хэш-таблица, список с пропусками и дерево поиска. Также взяты готовые реализации всех этих структур из библиотеки языка C#, чтобы сравнить неблокирующие реализации с блокирующими. Все алгоритмы адаптированы под язык C# и собраны в один общий модуль с внешним интерфейсом ISet. В приложении 1 приведены результаты сравнений всех этих структур и вариации использования их в реальной жизни.

3 Глава 1. Общие сведения

3.1 Классификация алгоритмов и атомарная операция

Все неблокирующие алгоритмы можно разделить на три типа: Wait-free, Lock-free, Obstruction-free [9].

В первом типе каждый поток совершает каждую операцию за конечное число шагов, независимо от влияния других потоков. Это самое сильное требование из-за чего редко реализуемое, однако существует алгоритм, преобразующий Obstruction-free в Wait-free [10].

Во втором типе система в целом двигается вперед, даже если какой-то поток стоит на месте. 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.

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

3.2 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 структур данных было описано в статье Трейбера в 1986 году [11]. 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 [12]. 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[13], списки с пропусками [14], хеш-таблицы [15], 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 без управления 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 управлением 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 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>
{
        State state;

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

Рис. 2: AtomicMarkableReference

3.3 Проблемf ABA

При неправильной организации кода в неблокирующих алгоритмах могут возникнуть проблемы. Самая известная - проблема ABA [16]. 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. Ука-

затель на новый элемент совпадёт с указателем на старый, что и приведёт к проблеме. Такая проблема может возникнуть только в языках программирования без управления памятью. В языках программирования с управлением памятью существует абстракция «сборщик мусора», которая обеспечивает отсутствие потери и повторного использования ссылок.

3.4 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 3).

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

Рис. 3: Интерфейс ISet

Чаще всего set применяют для объединения объектов с какими-то общими свойствами. Эта структура данных близка к map (key-value pair), которая используется повсеместно. Например, все базы данных так или иначе создаются на основе key-value pair.

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.

4 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 [13], [14], [17], [15], [18]. Стандартные реализации однопоточных алгоритмов общедоступны, поэтому не будут описаны в данной работе. Познакомиться с ними можно, например, в книге «Структуры данных и алгоритмы» [19].

4.1 Односвязный список

Пусть односвязный список состоит из элементов, в каждом из которых есть значение этого элемента и ссылка на следующий элемент (Рис 4). 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. 4: 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 5(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 5(b)). Поток Б также находит место для вставки и тоже запомина-

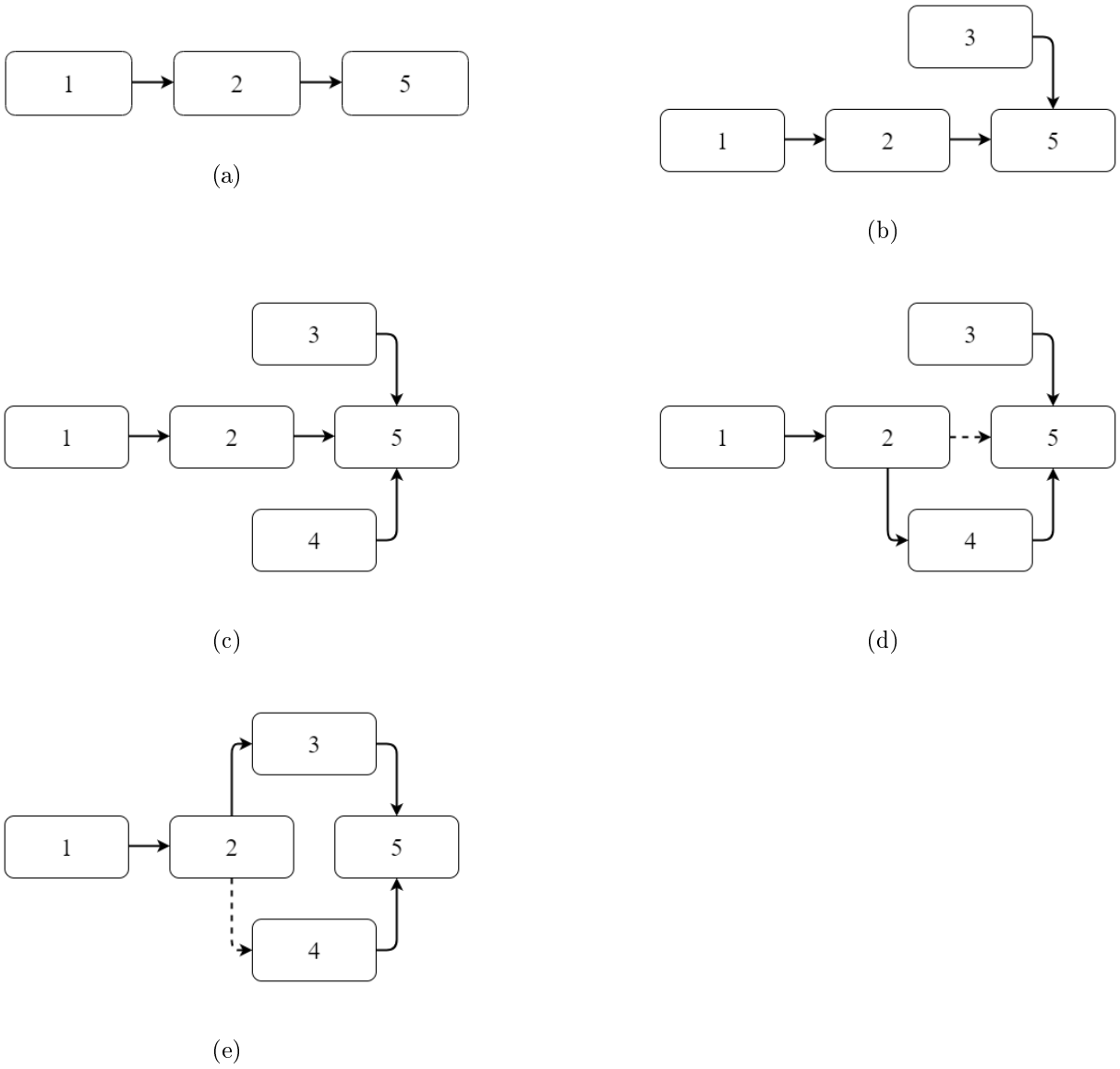


Рис. 5: Одновременное добавление в 2 потоках: a) начальное состояние, b) А поставил ссылку на 5, c) Б поставил ссылку на 5, d) А сменил ссылку «Next» у 2, e) Б сменил ссылку «Next» у 2.

ет 2, как предыдущий элемент, 5, как следующий (Рис 5(c)). После этого он переписывает ссылку «Следующий» у элемента 2 на новосозданный элемент 4, а у элемента 4 на 5 (Рис 5(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 5(e)). В результате элемент 4 «потеряется», т. е. не будет ни одной ссылки, указывающей

на него.

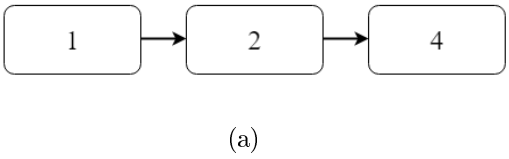
Реализация неблокирующего доступа использует типичный прием для неблокирующих алгоритмов - вечный цикл с операцией CAS. На каждом шаге цикла алгоритм пытается найти два элемента а и б, между которыми должен быть вставлен новый, и атомарно перезаписать ссылку «Next» с предыдущего элемента (а) на новый, 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 6). Такая реализация полностью решает вышеописанную проблему. При попытке перезаписать ссылку элемента 2 с 5 на 3 (Рис 5(e)), CAS не проходит, 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 цикла CAS уже выполняется успешно. Оба элемента вставлены правильно.

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

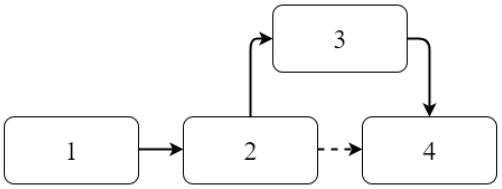
Рис. 6: Шаг цикла. 1) находим соседние элементы, между которыми нужно вставить новый, 2) ссылку Next у нового элемента поместим на subsessor 3) пытаемся атомарно вставить

Еще одна проблема может возникнуть при одновременным удалении и вставке двух последовательных элементов. Пусть есть список 1-2-4 (Рис 7(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 7(b)). Управление возвращается к потоку Б. Он атомарно заменяет ссылку «Next»

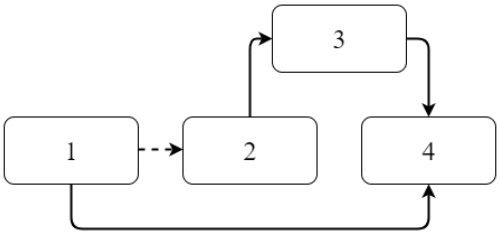
у элемента 1 на элемент 4 (Рис 7(c)). В итоге элемент 3 «потерялся».



(a)



(b)



(c)

Рис. 7: 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 8).

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

Рис. 8: AtomicMarkableReference - структура, описанная в предыдущей главе, Flag может быть никаким или помеченным

Теперь объект удаляется в два шага:

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

Можно заметить, что при помечивании ссылки на удаленный элемент, ситуация, изображенная на рисунке 2, существенно не изменится. Однако, при по-

мечивании ссылки «Next» у удаляемого объекта, можно избежать петерь 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.

4.2 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 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 9). Теперь

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

Рис. 9: 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 10(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 10(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.

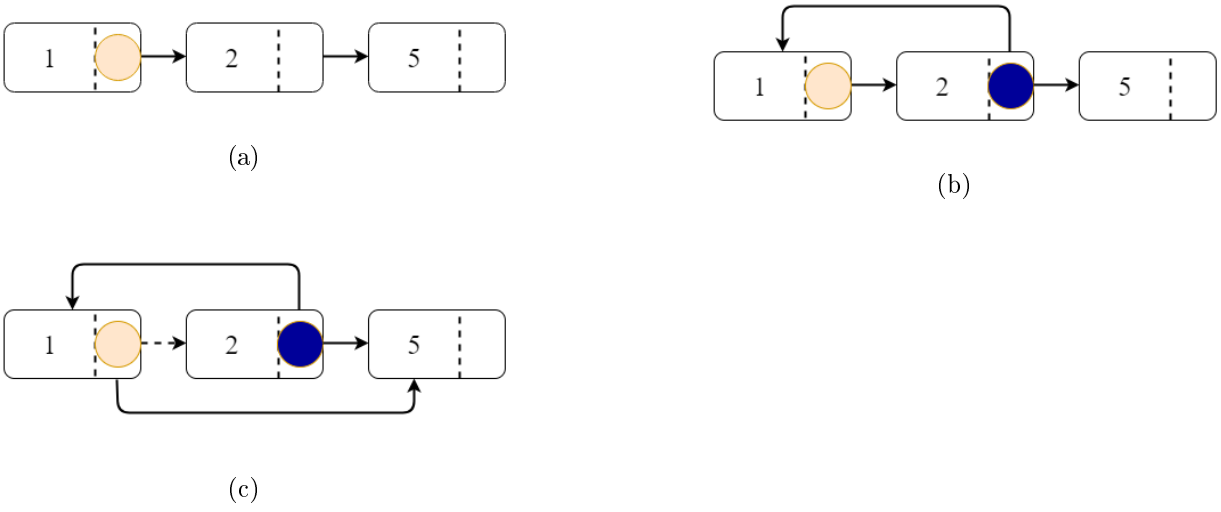


Рис. 10: Удаление в 3 этапа: a) помечивание на участие в удалении, b) помечивание на удаление, c) реальное удаление.

4.3 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. 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 недостижимы). 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.

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

4.4 Хэш-таблица

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

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 одновременно по массиву. Куски могут пересекаться или не пересекаться. Во время операции модификации высчитывается нужный хэш, 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 остается неизменным, однако, длина каждого из них увеличивается.

5 Глава 3. Тестирование

5.1 Модульное тестирование

Во время модульного тестирования проверяются максимально изолированные от внешнего мира части системы. Чаще всего такие тесты мелкие и целенаправленные. Такие тесты как раз определяют, изменилась ли функциональность программы или работает ли данная функция в целом.

Тестирование многопоточного приложения 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 тщательном тестировании каждого метода в одном потоке. Тестирование многопоточного приложения 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, но получить в работе с пользователем.

Отсюда плавно вытекает третий подход: тестирование аналитически. Он заключается в тщательном продумывании всех возможных сценариев, про-

верки 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 в ссылках на алгоритмы приведен подробный анализ корректности работы алгоритма, заключающийся просто в разборе всех возможных сценариев.

5.2 Тестирование производительности

Если тестирование работоспособности программы нужна для проверки, что программа работает корректно, то оценка производительности нужна для представления, реализуема ли вообще эта программа в жизни. 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 несколько различных структур данных и варианты работы в 1, 2 и 8 потоках.

Были произведены тесты на 1, 2 и 8 потоках всех вышеописанных структур данных а также их блокирующих аналогов. В качестве тестового окружения был выбран компьютер Intel Core i7-4790 CPU 3.60GHz (Haswell), ProcessorCount=8 Frequency=3507505 Hz, Resolution=285.1029 ns, Timer=TSC. C# Clr 4.0.30319.42000,

64bit LegacyJIT/clrjit-v4.6.127.1;compatjit-v4.6.1055.0. В качестве тестовых сценариев были выбраны такие как:

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

5.3 Результаты

Полные результаты представлены в приложении 1.

Из результатов

6 Заключение

В данной работе произведены

Список литературы

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

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

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

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

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

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

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

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

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

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

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

7 Приложения