УДК 681.3

Б. М. Кирдяшкин, Е. Л. Калишенко
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Разработка неблокирующих конкурентных структур данных на основе временных меток

Рассмотрена реализация относительно нового подхода к построению конкурентных структур данных – подход на временных метках, а также разработка модификаций к этому подходу. Рассматриваются возможности освобождения памяти и использования структуры данных на временных метках с неограниченным числом потоков. Исследуется влияние разработанных модификаций на производительность структуры данных.

Неблокирующие конкурентные структуры данных, многопоточность

Подход к реализации конкурентных структур данных, основанный на временных метках, был впервые предложен в 2014 г. Mike Doddsa Andreas Haas Christoph M. Kirsch в [1].

Данный подход к построению конкурентных очередей, стеков и деков заключается в использовании временных меток для определения порядка элементов. Во время вставки с каждым элементом

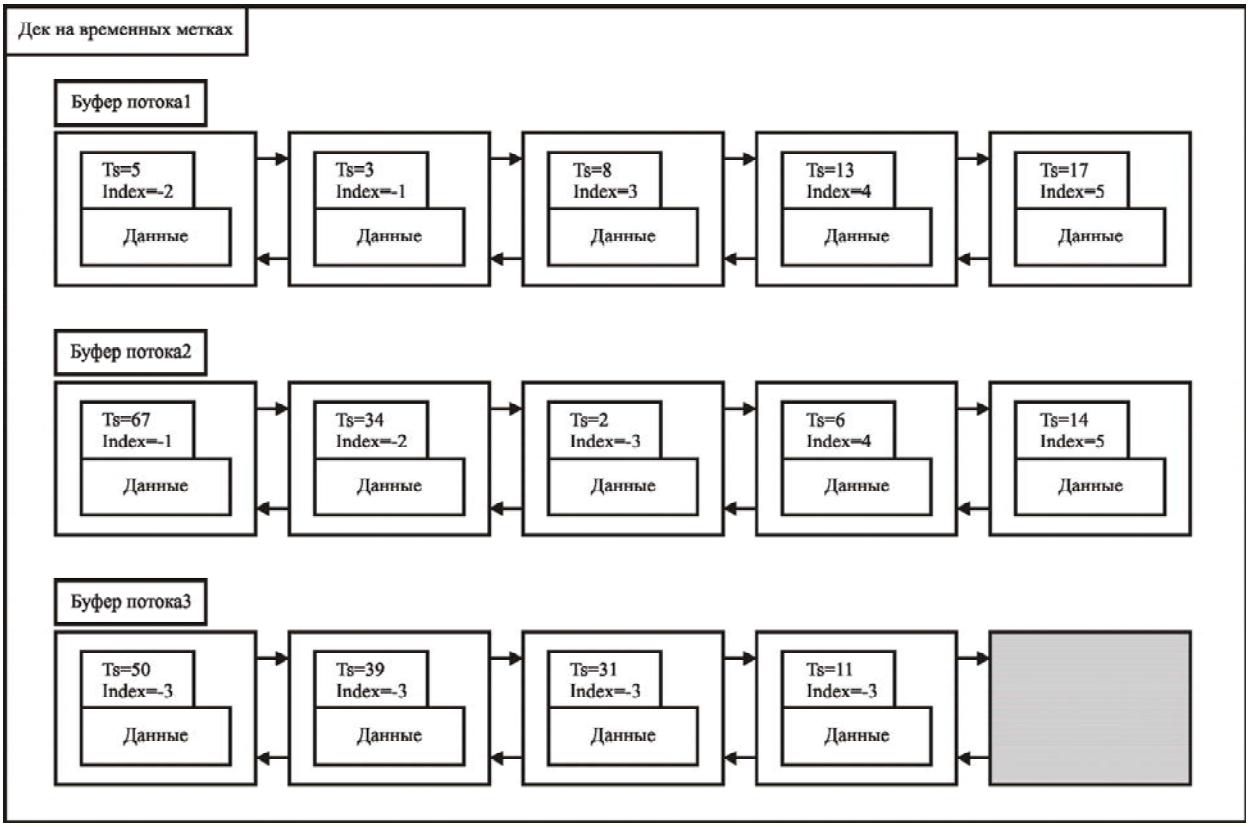

Рис. 1

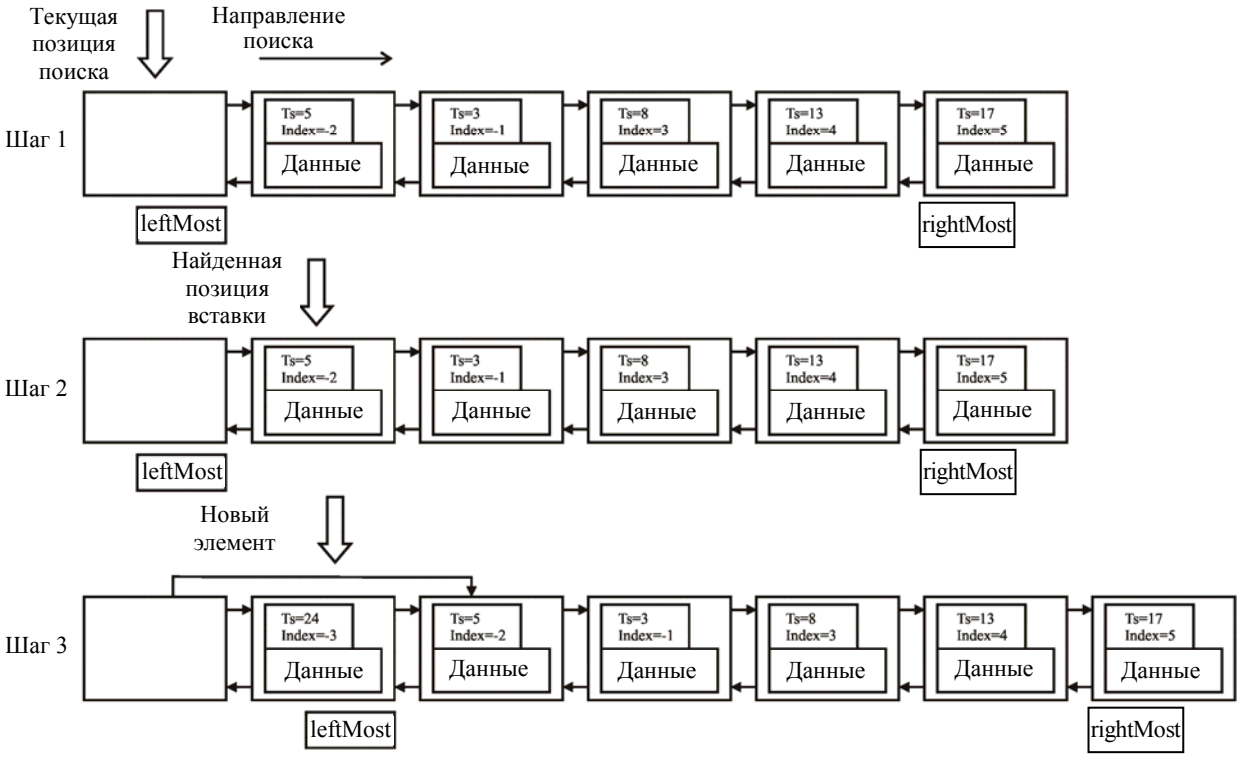

Рис. 2

ассоциируется временная метка, используемая во время извлечения для определения порядка элементов. В этом подходе используется множество буферов вида один читатель – много писателей, ассоциированных с потоками. Каждый поток пишет в свой собственный буфер, и благодаря этому обеспечивается малое время, затрачиваемое на вставку. В оригинальной структуре данных [1] количество потоков, способных работать со структурой данных, ограничивается значением, заданным в начале работы. На рис. 1 изображено графическое представление структуры данных.

Каждый буфер является двухсвязным списком, предоставляющим возможность доступа как к правому, так и к левому концу списка. Вставка элемента в список производится с помощью простой вставки в соответствующий буфер. Процесс вставки в буфер изображен на рис. 2.

После вставки, при необходимости, от структуры данных отделяется хвост. Неиспользуемая память в оригинальном [1] алгоритме не освобождается. Для извлечения элемента, допустим, справа производится итерация по всем буферам для поиска самого правого элемента. Порядок элементов, найденных в процессе поиска, определяется с помощью временных меток – из двух элементов, вставленных справа, правее тот, чья временная метка больше.

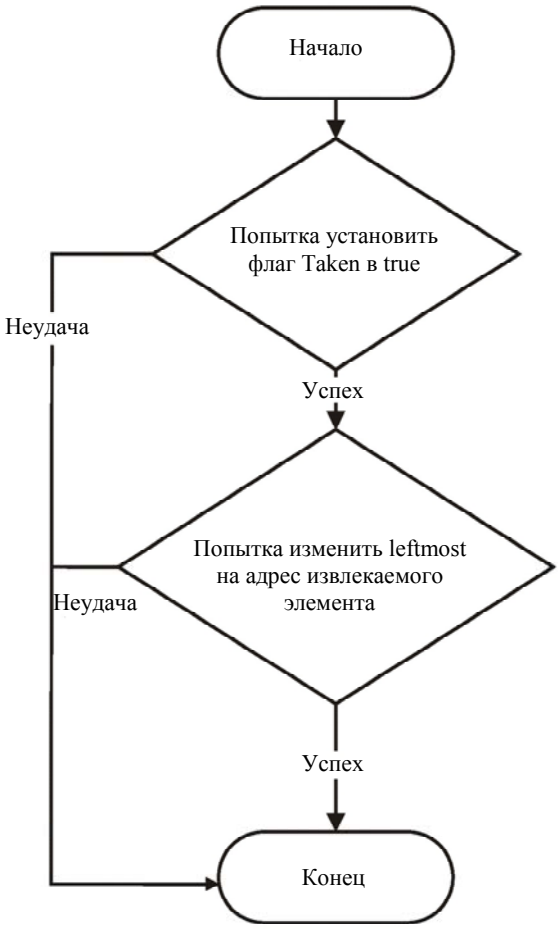

Рис. 3

Найденный элемент «удаляется» установкой флага Taken с помощью CAS-операции. Делается попытка перезаписи указателя на самый правый (или левый, для удаления слева) элемент с помощью CAS-операции (рис. 3).

Для уменьшения количества проваленных CAS-операций в [1] предлагается присваивать временную метку узлам *после* вставки в структуру данных. Такие узлы могут считаться неупорядоченными и могут быть извлечены в любом порядке. Стоит отметить, что количество таких узлов в отдельном буфере не может превышать единицу.

При нахождении неупорядоченного узла итерация по буферам прекращается и немедленно делается попытка извлечения. В случае провала CAS-операции во время извлечения узла производится повторный поиск элемента и попытка извлечения.

В описанной структуре данных [1] можно выделить несколько возможностей усовершенствования, среди которых:

– реализация освобождения использованной в процессе работы структуры данных памяти;

– реализация возможности использования структуры данных с неограниченным количеством потоков.

Алгоритм работы с неограниченным количеством потоков. Для обеспечения возможности работы с неограниченным количеством потоков необходимо при подключении нового потока создавать дополнительный буфер и добавлять его в некий список. В описываемой реализации оригинального алгоритма в качестве такого списка для хранения буферов используется массив, инициализирующийся в конструкторе. Использование обычного массива для реализации подобной модификации не представляется возможным, так как, во-первых, необходимо увеличивать размер массива по мере добавления потоков, а во-вторых, учитывая, что к такому списку будет осуществлять конкурентный доступ множество потоков, увеличение размера и добавление буфера в список должны быть атомарными операциями.

Исходя из этого было бы логично использовать одну из множества структур данных, уже реализованных в библиотеке libcds. Однако это не представляется рациональным, так как указанные структуры данных – общего назначения. В них реализован весь спектр операций, необходимых для работы со структурой данных: добавление, удаление, очистка и т. д., а также алгоритмы освобождения памяти. Для создаваемой структуры данных это представляется излишним. Например, не требуется удаление буферов, так как даже если поток прекратил работу, данные, добавленные им в структуру данных, должны сохраниться. Пример

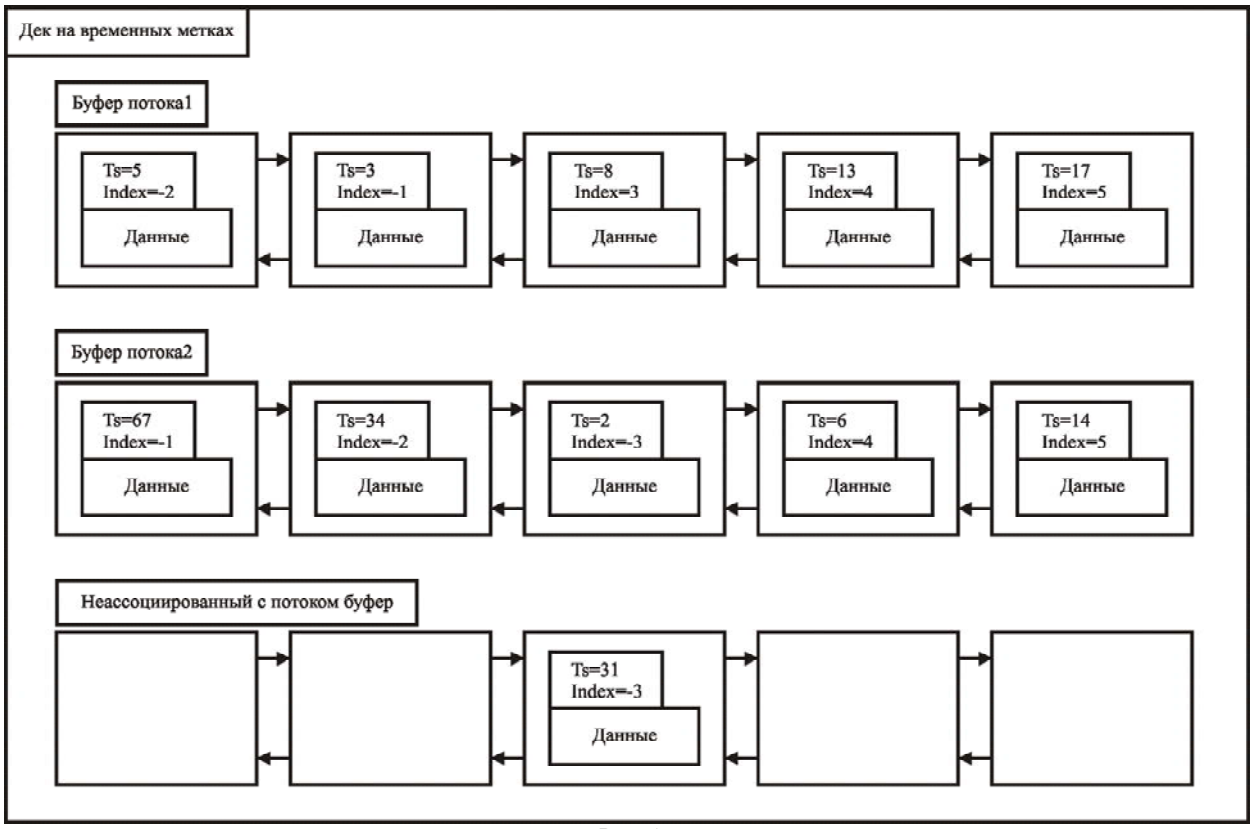

Рис. 4

такой ситуации представлен на рис. 4. Поток № 3 прекратил работу и больше не будет использовать структуру данных, однако в буфере, ассоциированном с данным потоком, еще остались не извлеченные элементы. Таким образом, невозможно извлечь такой буфер из списка, не потеряв данные, хранящиеся в нем.

Соответственно, если не требуется удаление буферов, то не нужны и алгоритмы освобождения памяти.

По этой причине для реализации возможности работы с неограниченным числом потоков предлагается реализовать однонаправленный связный список. Вставка в список производится с помощью CAS-операции над указателем на голову списка. И, помимо доступа к элементам, это все, что с таким списком можно делать.

Таким образом, идея данной модификации заключается в следующем:

1. В начале работы со структурой данных не инициализируется ни одного буфера. Создается пустой список буферов.

2. Во время вставки элемента в структуру данных каким-либо потоком проверяется, существует ли ассоциированный с этим потоком буфер. Если нет, создается новый буфер, ассоциируется с потоком, производящим вставку, и вставляется в общий список. Затем производится вставка элемента.

Данный подход является модификацией предыдущего с целью избавиться от главного недостатка – неиспользуемых буферов, накапливающихся во время работы со структурой данных.

Данный недостаток можно преодолеть несколькими способами:

1.Удалить буфера, в которых отсутствуют элементы.

2. Повторно использовать буфера, с которыми не ассоциирован поток.

Для удаления буферов, в которых отсутствуют элементы, необходимо:

– определить момент, когда в буфере уже не осталось элементов;

– реализовать конкурентное удаление буфера из списка буферов;

– обеспечить безопасное освобождение памяти, занимаемой удаленным буфером.

Первый пункт не представляет сложности в реализации: во время операции извлечения производится итерация по всем буферам и можно установить, что в буфере нет элементов. Второй и третий пункты достаточно сложны в реализации. Для удаления элемента из списка буферов и последующего освобождения памяти необходима синхронизация работы со списком буферов. Более того, помимо сложности реализации увеличатся накладные расходы на обеспечение работы со списком буферов.

В отличие от удаления повторное использование уже не использующихся буферов обещает быть проще в реализации и требует меньше накладных расходов на обеспечение своей работы. Идея заключается в добавлении флага каждому буферу для определения того, используется этот буфер каким-либо потоком или нет. Если флаг установлен – в данный момент буфер ассоциирован с каким-то потоком, если флаг не установлен – буфер свободен и может быть использован другим потоком. При этом, в отличие от идеи с удалением буферов, не надо отслеживать, есть в таких буферах элементы или нет. Необходимо обеспечить лишь одно условие – в определенный момент времени вставлять элементы в буфер может только один поток.

Упомянутый ранее флаг является простой булевской переменной и устанавливается с помощью CAS-операции во избежание «захвата» буфера двумя потоками одновременно.

Схематичное изображение списка потоков изображено на рис. 5.

Переменная occupied является булевской переменной и показывает, захвачен буфер каким-либо потоком либо нет. В примере буфера 1 и 4 не ассоциированы ни с одним потоком и могут быть использованы повторно.

Таким образом, идея данной модификации заключается в следующем:

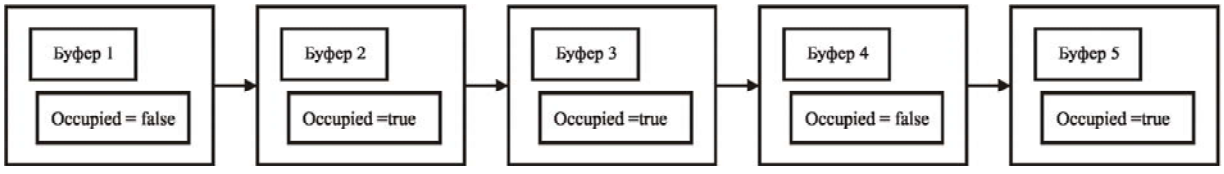

Рис. 5

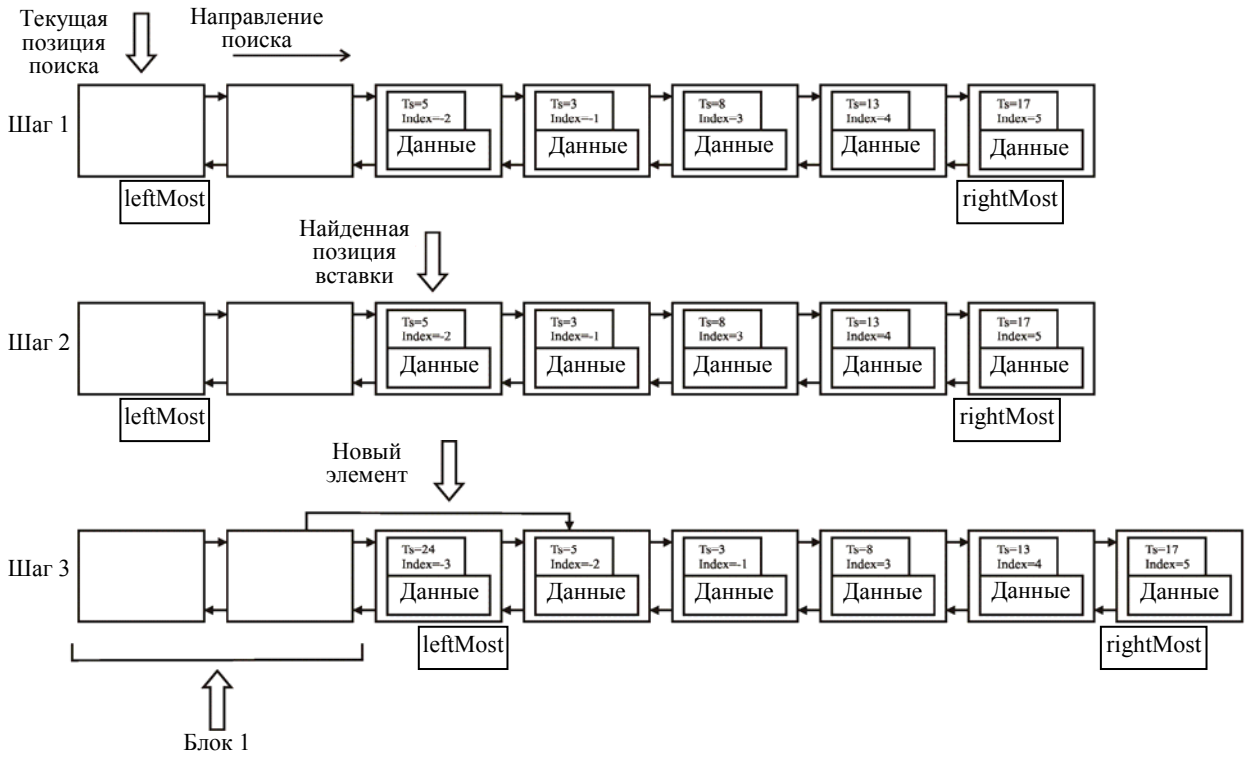

Рис. 6

1. В начале работы со структурой данных не инициализируется ни одного буфера. Создается пустой список буферов.

2. Во время вставки элемента в структуру данных каким-либо потоком проверяется, существует ли ассоциированный с этим потоком буфер. Если нет, производится поиск незанятого буфера в списке. Если такой найден, происходит его захват. Если свободного буфера нет, создается новый, ассоциируется с потоком и вставляется в список буферов.

3. Если во время окончания жизни потока существует буфер, ассоциированный с ним, флаг occupied этого буфера устанавливается в false.

Алгоритм освобождения неиспользуемой памяти. Во время работы структуры данных образуется значительное количество неиспользуемой памяти. Под неиспользуемой памятью подразумеваются уже «извлеченные» элементы структуры данных, т. е. элементы с установленным флагом «Taken».

Так как фактически описываемая структура данных, основанная на временных метках (дек, стек или очередь), состоит из множества других структур данных, называемых буферами, будем рассматривать способы освобождения памяти в этих буферах, и, найдя способ сделать это в одном буфере, сможем применить его к множеству буферов.

В оригинальной структуре данных неиспользуемая память не освобождается. Так, например, узлы, отделяемые от структуры во время вставки элементов, остаются в памяти программы до ее завершения.

На рис. 6 показан отсоединенный во время вставки блок элементов, который может быть освобожден.

Блок 1 на рисунке отсоединен от буфера и содержит элементы, которые могут быть освобождены.

В оригинальном алгоритме отсоединение узлов от двунаправленного списка, представляющего буфер, производится только в одной операции – вставке. Наиболее очевидным решением в данной ситуации будет попытка освободить память, занимаемую узлами, отсоединяемыми во время вставки. На рис. 6 показан блок узлов, отсоединенных от буфера.

Такой подход, однако, имеет недостатки, которые не позволяют использовать его для очистки памяти.

Так как описываемая структура данных может использоваться в многопоточных программах, с одним и тем же буфером могут работать множество потоков одновременно. Исходя из этого можно сделать вывод, что при таком подходе к очистке памяти, занимаемой неиспользуемыми элементами буфера, будет постоянно возникать ошибка доступа к памяти (segmentation fault).

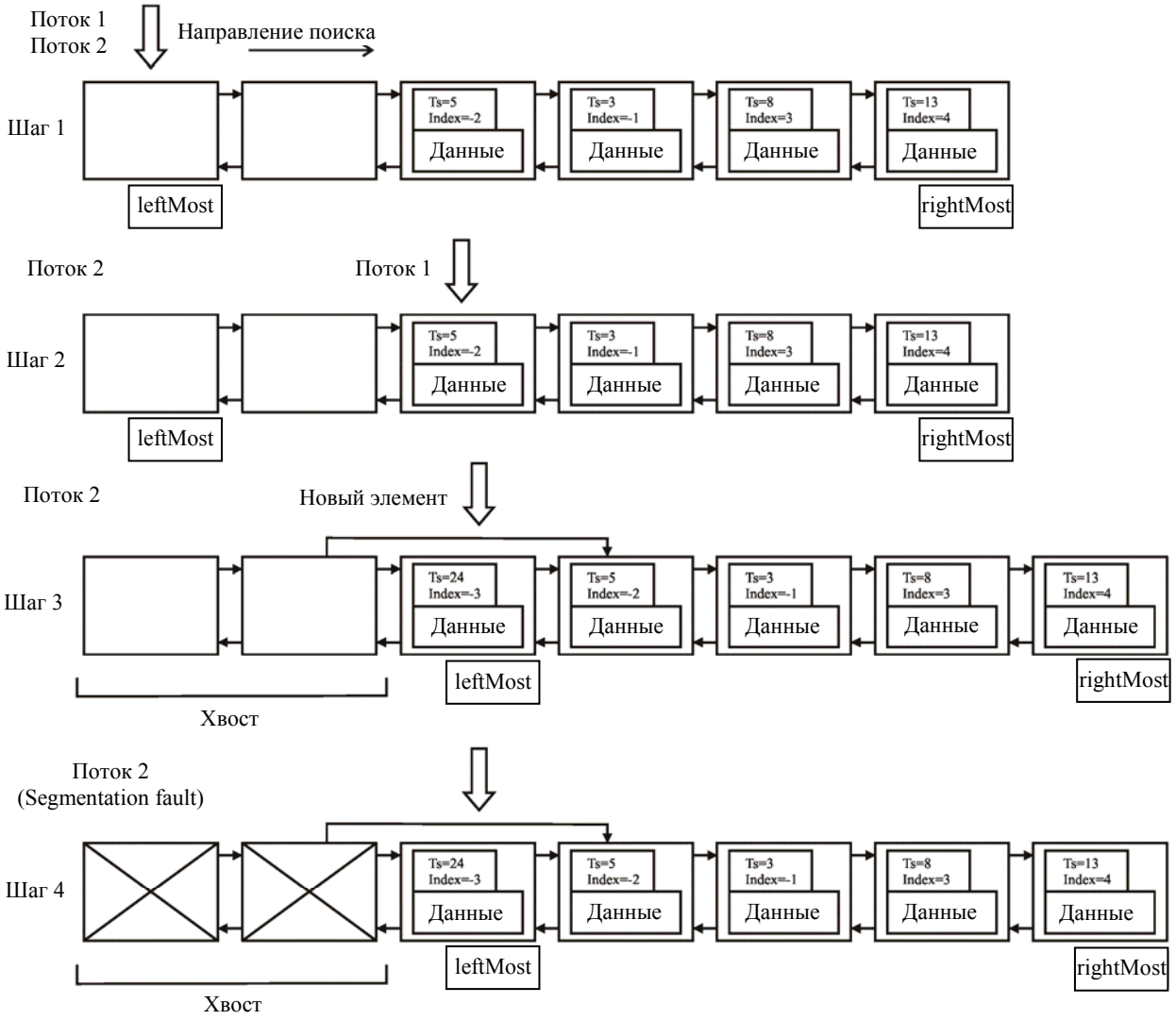

Рис. 7

Рассмотрим ситуацию, изображенную на рис. 7.

Поток 1 вставляет в буфер новый элемент. Поток 2 пытается из буфера элемент извлечь.

Перед попыткой извлечь элемент из буфера или вставить элемент в буфер оба потока ведут поиск первого неудаленного элемента. В данном случае вставка и удаление идут слева, поэтому поиск ведется слева направо.

Оба потока начали поиск одновременно, и поток 1 нашел элемент, перед которым будет производиться вставка, поток 2 не успел завершить поиск, так как время, выделенное для его работы, закончилось, и он заснул. Тем временем поток 1 отсоединяет «хвост» и освобождает занимаемую им память. После этого поток 2 просыпается и пытается продолжить поиск. Так как теперь он обращается к деаллоцированным участкам памяти, происходит ошибка доступа к памяти (segmentation fault).

Для того чтобы избежать данной ошибки, предлагается использовать счетчик потоков, работающих с буфером. В качестве такого счетчика можно использовать атомарную переменную. Данный счетчик предполагается инкрементировать перед началом поиска элемента в буфере и декрементировать после его окончания.

Таким образом, проверяя счетчик перед освобождением памяти, можно определить, работают ли в данный момент с буфером другие потоки или нет.

Перед началом поиска в буфере запоминаются текущие самый правый и самый левый элементы буфера, и поиск, например слева направо, ведется от самого левого элемента до самого правого. Учитывая также, что после вставки вставленный элемент считается самым левым (при вставке слева), можно проверить счетчик после окончания вставки и быть уверенным, что ни один поток не работает с кандидатами на удаление.

Основной недостаток данного подхода – ссылки на те элементы, которые не были удалены

во время вставки (счетчик был больше 0), теряются и не могут быть удалены.

Для того чтобы преодолеть этот недостаток, можно сохранять ссылки на те элементы, во время извлечения которых счетчик был равен нулю (откладывать элементы на освобождение памяти), однако в ходе экспериментов выяснилось, что при работе со структурой данных, несмотря на все улучшения, освобождается лишь малое количество памяти.

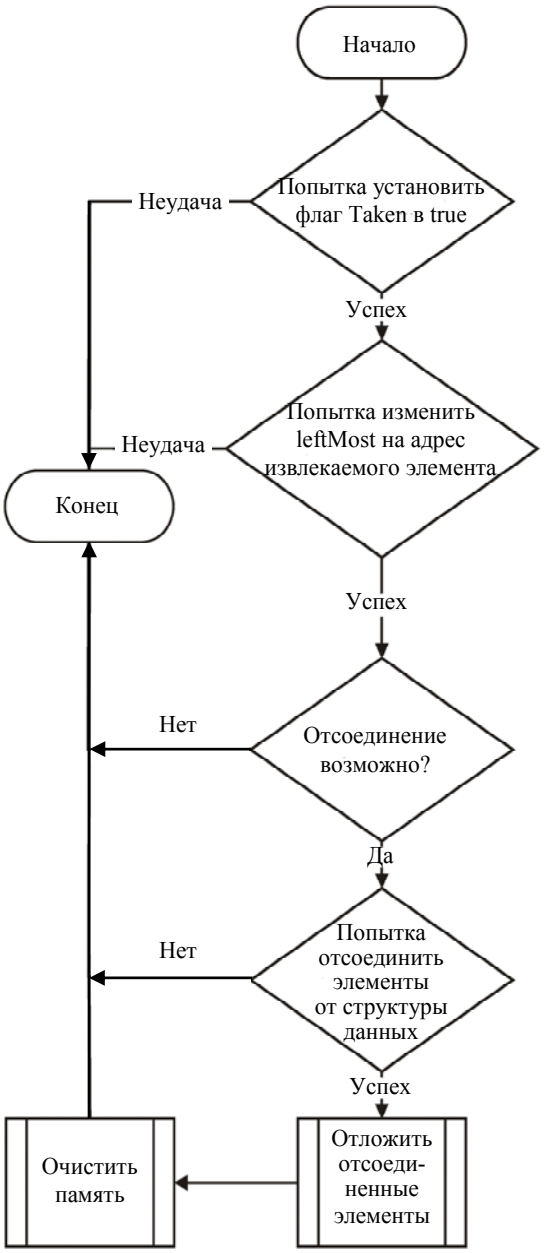

Рис. 8

Очевидным улучшением будет отсоединять элементы от буфера также и во время операции извлечения. Модификация операции извлечения из буфера представлена на рис. 8. Такое улучшение, однако, увеличит количество возможных ошибок, связанных с взаимодействием потоков (в отличие от операции вставки в буфер, которую может производить лишь один поток, операцию извлечения могут производить одновременно неограниченное количество потоков).

Основные проблемы:

1. Необходимость использования конкурентной структуры данных для отложенных элементов.

2. Необходимость синхронизации во время отсоединения при операции извлечения.

3. Необходимость синхронизации при освобождении памяти, занимаемой отложенными элементами.

Опишем последовательность действий, приводящих к segmentation fault в текущем подходе со всеми описанными модификациями. Segmentation fault появляется в момент, когда один поток (поток 1) пытается очистить память, занимаемую отложенными элементами, во время того как другой поток (поток 2) откладывает элементы на очистку, а третий поток (поток 3) начинает поиск. Следующая последовательность действий приводит к проблеме:

1. Поток 1 начинает выполнение операции освобождения отложенных элементов и проверяет счетчик (равный нулю), после чего засыпает.

2. Поток 3 начинает поиск и засыпает.

3. Поток 2 начинает операцию извлечения (или вставки), заканчивает ее и откладывает элементы на освобождение.

4. Просыпается поток 1 и освобождает память элементов, отложенных потоком 2.

5. Поток 3 продолжает поиск, что приводит к Segmentation fault.

Чтобы избежать этой проблемы, предлагается использовать временные метки. Каждый отложенный элемент ассоциируется с временной меткой, генерируемой в момент, когда этот элемент откладывается. Данная временная метка используется, чтобы определить, помещен этот элемент до того, как счетчик был проверен в операции извлечения (шаг 1), или после того. Если элемент отложен после того как счетчик был проверен, то память, занимаемая этим элементом, не освобождается.

Блок-схема освобождения отложенных элементов представлена на рис. 9.

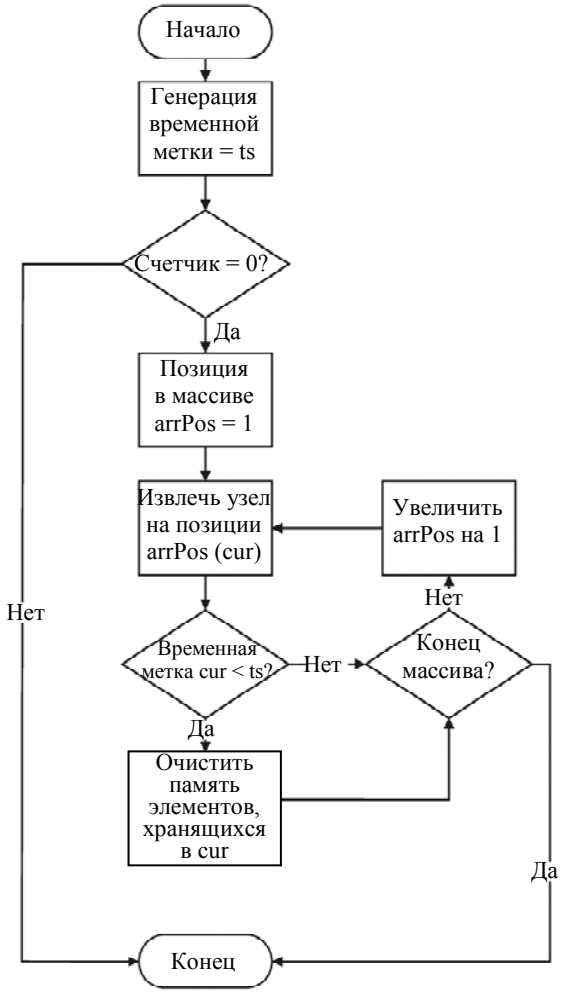

Рис. 9

Вместо обычной структуры данных для хранения отложенных элементов предлагается использовать массив ограниченного размера. Если во время операции откладывания массив переполнен, делается попытка выполнять операцию освобождения памяти до тех пор, пока в массиве не появится свободное место. Вставка в массив производится с помощью CAS-операции.

Подводя итог, отметим, что для реализации освобождения памяти были разработаны следующие модификации оригинальной структуры данных:

1. Отсоединение узлов от буфера во время извлечения.

2. Счетчик работающих с буфером потоков.

3. Массив отложенных на очистку узлов.

В общих чертах алгоритм очистки неиспользуемой памяти во время работы структуры данных выглядит следующим образом:

1) отсоединение узлов от буфера во время операции вставки или извлечения;

2) добавление отсоединенных узлов в массив отложенных узлов;

3) очистка памяти, занимаемой отложенными узлами, если это возможно (счетчик = 0).

Нагрузочное тестирование. Для сбора различных данных о работе реализованного дека на временных метках выполнялись следующие тесты:

1. Тест с равным количеством потоков-читателей и потоков-писателей, одновременно работающих с деком (тест «читатель-писатель»).

2. Тест только с потоками-писателями (тест «писатель»).

3. Тест только с потоками-читателями, работающими с деком, предварительно заполненным элементами (тест «читатель»).

Тест 1 выполнялся следующим образом: для N потоков, работающих в тесте, $N/2$ являются читателями, $N/2$ – писателями. Каждый писатель вставляет в структуру данных 1 000 000 элементов без искусственных задержек. Каждый писатель извлекает 1 000 000 элементов. Синхронизация начала работы потоков со структурой данных выполняется с помощью барьера.

Тест 2 работает с N потоками, каждый из которых является писателем. Каждый поток вставляет в структуру данных 1 000 000 элементов. Синхронизация начала работы потоков со структурой данных выполняется с помощью барьера.

Тест 3 работает с N потоками, каждый из которых является читателем. Каждый поток читает из структуры данных 1 000 000 элементов. Перед началом работы тестовых потоков структура заполняется элементами. Для заполнения структуры данных элементами используется N потоков, каждый из которых вставляет в структуру данных 1 000 000 элементов. Синхронизация начала работы тестовых потоков со структурой данных выполняется с помощью барьера.

Из графика на рис. 10 видно, что в тесте «читатель-писатель» версии структуры данных с модификациями проигрывают в скорости оригинальной структуре данных. При этом модификация освобождением памяти и неограниченным числом потоков имеет лучшую производительность, чем модификация только с освобождением памяти. Это объясняется выделением буферов только при необходимости, что приводит к меньшему количеству буферов и меньшему времени поиска.

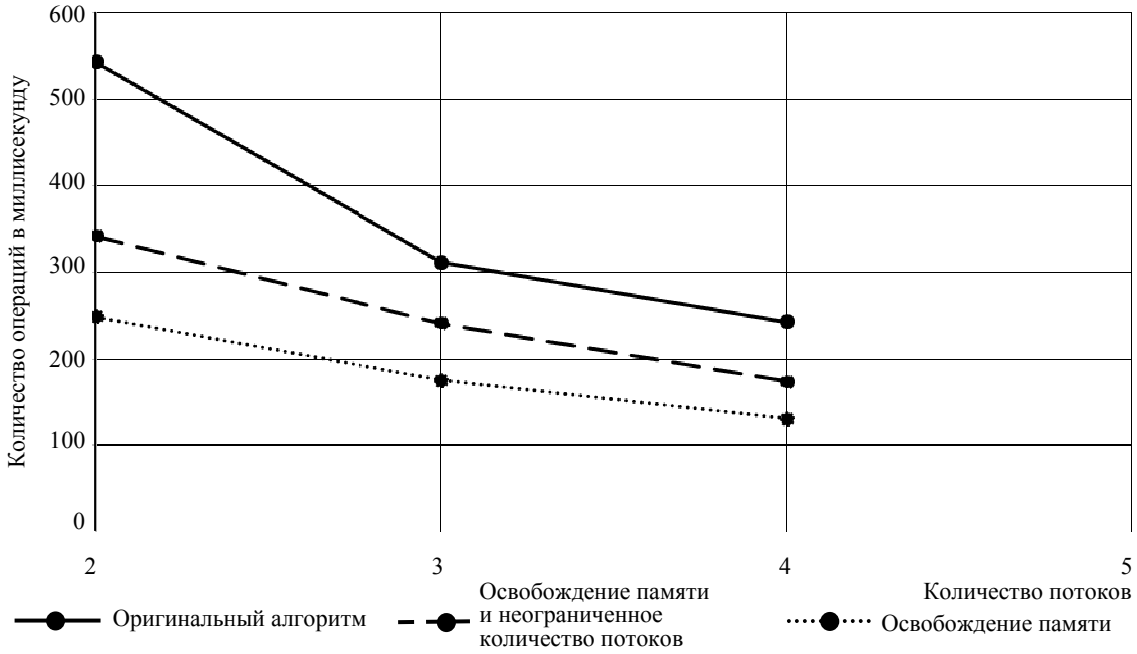

Рис. 10

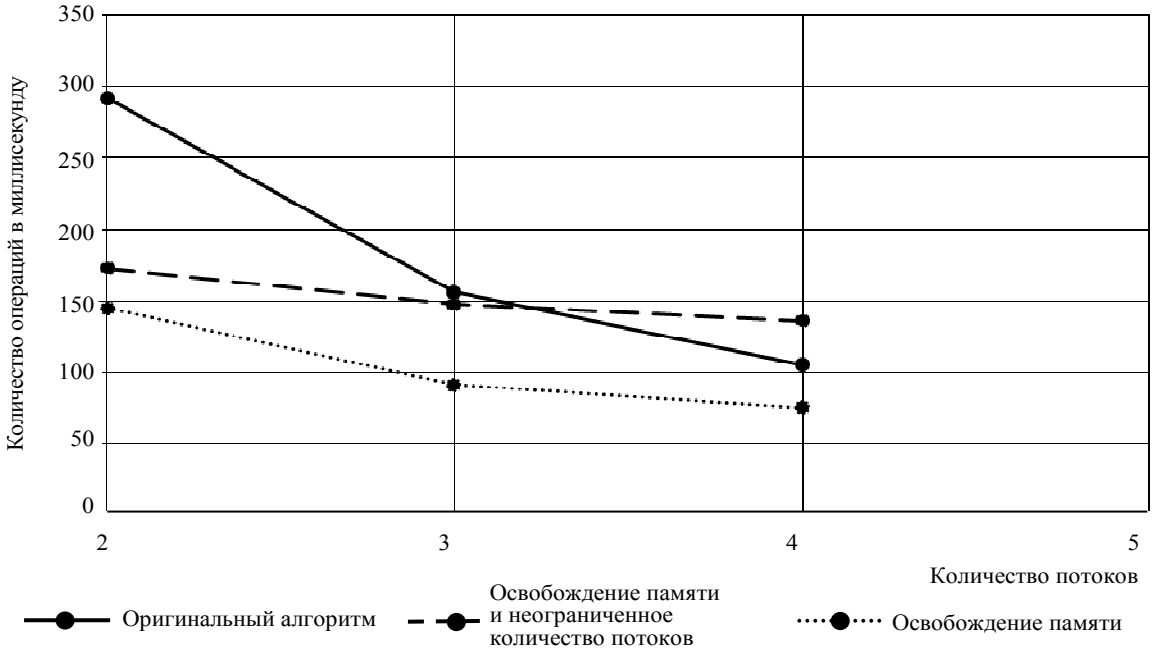

Рис. 11

Как следует из графика на рис. 11, для теста «читатель» версия структуры данных с обеими модификациями работает практически с такой же скоростью, что и без модификаций. Различия с тестом «читатель-писатель» объясняются меньшим количеством работающих потоков (у компьютера, на котором проводились тесты, только 4 ядра), а также отсутствием «неупорядоченных» элементов в процессе выполнения, что может замедлять как модифицированную, так и немодифицированную версии.

Как видно из графика на рис. 12, в тесте «писатель» обе версии с модификациями проходят тесты с примерно одинаковыми результатами, различие может объясняться неизбежными погрешностями в измерениях. Также обе модификации работают значительно медленнее версии без модификаций, что может объясняться дополнительной синхронизацией, необходимой для освобождения памяти. Скорость работы всех версий структуры данных возрастает при увеличении количества потоков в тесте «писатель».

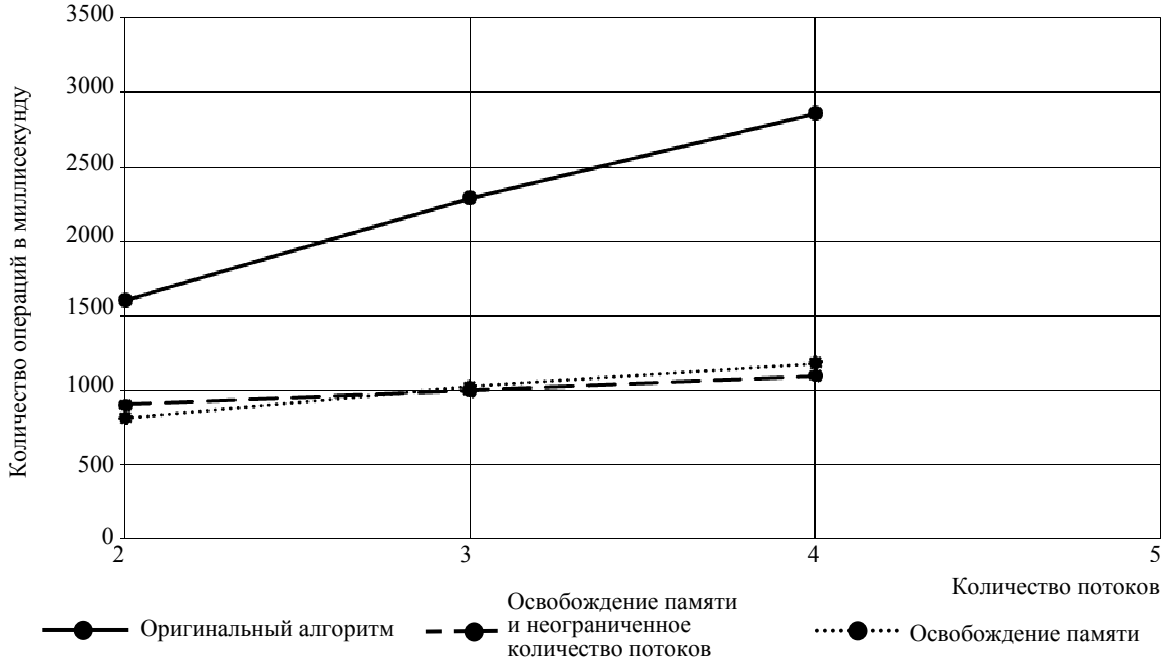

Рис. 12

На основе проделанной работы можно сделать вывод о возможности реализации освобождения памяти в рамках структуры данных, основанной на временных метках, а также возможности использования структуры данных с неограниченным числом потоков.

Как показали тесты, разработанные модификации замедляют работу структуры данных, что объясняется накладными расходами на синхронизацию и дополнительными, внутренними, структурами данных, использующихся для реализации модификаций.

В качестве будущих направлений работы можно отметить, в первую очередь, более подробное тестирование на большем количестве потоков, оптимизацию работы структуры данных и разработанных модификаций.

СПИСОК ЛИТЕРАТУРЫ

1. Dodds M., Haas A., Kirsch C. M. Fast Concurrent Data-Structures Through Explicit Timestamping / University of Salzburg. Salzburg, 2014.

2. Lock-free data structures. Stack evolution. URL: http://habrahabr.ru/company/ifree/blog/216013/.

3. Herlihy M., Shavit N. The Art of Multiprocessor Programming. San Francisco: Morgan Kaufmann, 2012.

B. M. Kirdyashkin, E. L. Kalishenko
Saint Petersburg Electrotechnical University «LETI»

DEVELOPMENT OF LOCK-FREE CONCURRENT DATA STRUCTURES BASED ON TIMESTAMPING

In this work relatively new approach to implement concurrent data structures is being considered. This approach is «Data structures through explicit timestamping». This work concentrated on developing modification of this approach. These modifications are «Ability to deallocate unused memory» and «Ability to work with unlimited amount of threads». To understand effect which these modifications render on data structure, performance of data structure with implemented modifications is being investigated.

Lock-free concurrent data structures, multithreading