**«Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И.Ульянова (Ленина)»**

**(СПбГЭТУ “ЛЭТИ”)**

|  |  |
| --- | --- |
| **Направление** | 09.04.01 - Информатика и вычислительная техника |
| **Программа** | Распределенные интеллектуальные системы и технологии |
| **Факультет** | Компьютерных технологий и информатики |
| **Кафедра** | Вычислительной техники |
| *К защите допустить* |  |
| Зав. кафедрой | д.т.н., профессор Куприянов М. С. |

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА**

**Тема: АНАЛИЗ ПОТОКОБЕЗОПАСНЫХ СТРУКТУР ДАННЫХ**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент |  | \_\_\_\_\_\_\_\_\_\_\_\_ |  | Кабин А. К. |
| Руководитель | к.т.н., доцент | \_\_\_\_\_\_\_\_\_\_\_\_ |  | Пазников А. А. |
| Консультанты | к.э.н., доцент | \_\_\_\_\_\_\_\_\_\_\_\_ |  | Магомедов М. Н. |
|  |  | \_\_\_\_\_\_\_\_\_\_\_\_ |  | Гречухин М. Н. |

Санкт-Петербург

2023

**ЗАДАНИЕ**

**НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Утверждаю | | |
|  | | Зав. кафедрой ВТ | | |
|  | | \_\_\_\_\_\_\_\_\_\_ Куприянов М. С. | | |
|  | | “\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г. | | |
| Студент: | Кабин А. К. | Группа: |  | 7307 |
| Тема работы: Анализ потокобезопасных структур данных  Место выполнения ВКР: СПБГЭТУ «ЛЭТИ»  Исходные данные: отсутствуют  Содержание ВКР: обзор литературы по теме, анализ общих методов анализа потокобезопасных структур данных, анализ производительности потокобезопасных и непотокобезопасных структур данных, реализация и тестирование различных lock-free очередей.  Дополнительные разделы: Оценка и защита результатов интеллектуальной деятельности  Перечень отчетных материалов: пояснительная записка, презентация | | | | |
| Дата выдачи задания | | Дата представления ВКР к защите | | |
| “\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г. | | “\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г. | | |
| Студент |  | \_\_\_\_\_\_\_\_\_\_\_\_ |  | Кабин А. К. |
| Руководитель | к.т.н., доцент | \_\_\_\_\_\_\_\_\_\_\_\_ |  | Пазников А. А. |

**КАЛЕНДАРНЫЙ ПЛАН ВЫПОЛНЕНИЯ**

**ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ**

|  |  |  |  |
| --- | --- | --- | --- |
|  | | Утверждаю | |
|  | | Зав. кафедрой ВТ | |
|  | | \_\_\_\_\_\_\_\_\_\_Куприянов М. С. | |
|  | | “\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г. | |
| Студент: | Кабин А. К. | Группа: | 7307 |
| Тема работы: Анализ потокобезопасных структур данных | | | |

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование работ | Срок выполнения |
| 1 | Обзор структур данных и параллелизма | 01.02.2023 - 07.02.2023 |
| 2 | Изучение общих методов анализа потокобезопасных структур данных | 08.02.2023 - 28.02.2023 |
| 3 | Сравнительный анализ производительности потокобезопасных и непотокобезопасных структур данных. | 01.03.2023 - 31.03.2023 |
| 4 | Реализация и тестирование различных lock-free очередей | 01.04.2023 - 30.04.2023 |
| 5 | Оценка и защита результатов интеллектуальной деятельности | 01.05.2023 - 14.05.2023 |
| 6 | Оформление пояснительной записки | 15.05.2023 - 21.05.2023 |
| 7 | Представление работы на предзащите | 22.05.2023 |
| 8 | Представление работы на защите | 25.05.2023 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент |  | \_\_\_\_\_\_\_\_\_\_\_\_ |  | Кабин А. К. |
| Руководитель | к.т.н., профессор | \_\_\_\_\_\_\_\_\_\_\_\_ |  | Пазников А. А. |

**РЕФЕРАТ**

Пояснительная записка включает 86 страниц, 14 рисунков, 3 таблицы, 25 источников и 2 приложения.

АНАЛИЗ ПОТОКОБЕЗОПАСНЫХ СТРУКТУР ДАННЫХ, СТРУКТУРЫ ДАННЫХ, ПОТОКОБЕЗОПАСНЫЕ СТРУКТУРЫ ДАННЫХ

Объектом исследования являются потокобезопасные структуры данных.

Предметом исследования является процесс анализа потокобезопасных структур данных.

Цель работы состоит в выявлении слабых мест потокобезопасных структур данных и оптимизации с точки зрения скорости работы.

Метод выполнения работы определяется следующей последовательностью задач.

1. Изучение различных структур данных и параллелизма.

2. Изучение общих методов анализа потокобезопасных структур данных.

3. Сравнительный анализ производительности потокобезопасных и непотокобезопасных структур данных.

4. Реализация различных lock-free очередей.

5. Тестирование lock-free очередей.

В результате была решена задача анализа потокобезопасных структур данных, в частности были реализован, протестированы и проанализированы различные lock-free реализации очередей.

**ABSTRACT**

The explanatory note includes 74 pages, 12 figures, 3 tables, 22 sources and 1 appendix.

ANALYSIS OF THREAD-SAFE DATA STRUCTURES, DATA STRUCTURES, THREAD-SAFE DATA STRUCTURES

The object of the study is thread-safe data structures.

The subject of the study is the process of analyzing thread-safe data structures.

The purpose of the work is to identify weaknesses of thread-safe data structures and optimize in terms of speed.

The method of performing the work is determined by the following sequence of tasks.

1. Study of various data structures and concurrency.

2. Study of general methods for analyzing thread-safe data structures.

3. Comparative analysis of the performance of thread-safe and non-thread-safe data structures.

4. Implementation of various lock-free queues.

5. Testing lock-free queues.

As a result, the problem of analyzing thread-safe data structures was solved, in particular, various lock-free queue implementations were implemented, tested and analyzed.

**СОДЕРЖАНИЕ**

[ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 8](#_Toc135239423)

[ВВЕДЕНИЕ 9](#_Toc135239424)

[1 Обзор параллелизма с учетом структур данных 12](#_Toc135239425)

[1.1 Параллелизм в операционной системе 12](#_Toc135239426)

[1.2 Принципы параллелизма 12](#_Toc135239427)

[1.3 Механизмы параллелизма 14](#_Toc135239428)

[1.4 Преимущества и проблемы параллелизма 15](#_Toc135239429)

[2 Методы общего анализа потокобезопасных структур данных 20](#_Toc135239430)

[2.1 Статический анализ 20](#_Toc135239431)

[2.2 Динамический анализ 20](#_Toc135239432)

[2.3 Методы формальной верификации 22](#_Toc135239433)

[3 Примеры потокобезопасных структур данных 27](#_Toc135239434)

[4 Применение потокобезопасных структур данных 33](#_Toc135239435)

[5 Сравнительный анализ производительности потокобезопасных и непотокобезопасных структур данных 35](#_Toc135239436)

[6 Разработка программы для анализа потокобезопасных структур данных 38](#_Toc135239437)

[6.1 Архитектура программного обеспечения 38](#_Toc135239438)

[6.2 Используемые библиотеки и инструменты 38](#_Toc135239439)

[6.2.1 Язык программирования 38](#_Toc135239440)

[6.2.2 Среда разработки 42](#_Toc135239441)

[6.3 Реализация программы для анализа потокобезопасных структур данных 44](#_Toc135239442)

[6.3.1 Компонент для анализа данных 45](#_Toc135239443)

[6.3.2 Компонент для визуализации результатов 56](#_Toc135239444)

[7 Оценка и защита результатов интеллектуальной деятельности 60](#_Toc135239445)

[7.1 Описание результата интеллектуальной деятельности 61](#_Toc135239446)

[7.2 Оценка рыночной стоимости результата интеллектуальной деятельности 62](#_Toc135239447)

[7.2.1 Расчет затрат на оплату труда 64](#_Toc135239448)

[7.2.2 Расчет накладных расходов 66](#_Toc135239449)

[7.2.3 Издержки на амортизацию оборудования 66](#_Toc135239450)

[7.2.4 Расходы на услуги сторонних организаций 67](#_Toc135239451)

[7.2.5 Себестоимость разработки программного обеспечения 68](#_Toc135239452)

[7.3 Правовая защита результатов интеллектуальной   
деятельности 68](#_Toc135239453)

[ЗАКЛЮЧЕНИЕ 70](#_Toc135239454)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 72](#_Toc135239455)

[ПРИЛОЖЕНИЕ А 76](#_Toc135239456)

[ПРИЛОЖЕНИЕ Б 77](#_Toc135239457)

# 

# **ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ**

В данной работе применяются следующие термины с соответствующими определениями.

Взаимная блокировка – ситуация в многозадачной среде или СУБД, при которой несколько процессов находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать свое выполнение.

Инверсия приоритетов – это сценарий в планировании, в котором высокоприоритетная задача косвенно заменяется задачей с более низким приоритетом, эффективно инвертируя назначенные приоритеты задач. Это нарушает модель приоритета, согласно которой выполнение высокоприоритетных задач может быть предотвращено только задачами с более высоким приоритетом. Инверсия происходит, когда возникает конфликт ресурсов с задачей низкого приоритета, которая затем вытесняется задачей среднего приоритета.

НИР – научно-исследовательская работа.

СД – структуры данных.

Состояние гонки ­– это состояние электроники, программного обеспечения или другой системы, при котором существенное поведение системы зависит от последовательности или времени других неконтролируемых событий.

# **ВВЕДЕНИЕ**

Любая современная программа состоит из программного кода. Программный код содержит внутри себя определенные идентичные сущности. Одной из таких сущностей являются структуры данных.

Структура данных — это вариант организации информации для более удобного и эффективного использования. Наиболее часто структурой данных в программировании называют набор связанных определенным образом данных. Например, массив — это структура.

Для каждой структуры свойственны определённые операции и алгоритмы реализации. Наиболее частыми из них являются: добавление, извлечение, модификация. Так же содержимое структур данных можно обрабатывать, например анализировать и сортировать. Для каждой структуры данных — свои алгоритмы. Задача программиста — уметь правильно применять уже реализованные структуры, либо реализовывать свои.

Главным свойством любой структуры данных является атомарность. Каждой единице данных принадлежит чёткое место, по которому её можно найти. Способ определения места, а также поиск по значению, зависит от конкретной структуры [1].

Характеристики структур данных представлены ниже.

1. Данные в памяти представлены определённым образом, который однозначно позволяет определить структуру.

2. Чаще всего внутрь структуры можно добавить элемент или извлечь оттуда. Это свойство не постоянное — бывают структуры, которые нельзя изменять после создания.

3. Существуют алгоритмы, которые позволяют взаимодействовать с этой структурой.

Однако, с развитием цифровых технологий и увеличением мощности вычислительных устройств, стали появляться многоядерные и многопроцессорные вычислительные системы, обладающие общей памятью, что позволило им одновременно обращаться к данным. Появление таких систем потребовало от разработчиков модификации существующего набора структур данных. Новым структурам данных было дано название «Потокобезопасные структуры данных»

Исторически такие структуры данных начали развиваться еще во времена однопроцессорных машин на операционных системах, поддерживающих несколько вычислительных потоков или процессов. В последствии такие потокобезопасные структуры данных так же стали называть параллельными структурами данных.

В современности, когда многопроцессорные компьютерные архитектуры, обеспечивающие параллелизм, выступают в роли доминирующей вычислительной платформы (благодаря распространению многоядерных процессоров), этот термин стал обозначать в основном структуры данных, доступ к которым может осуществляться несколькими потоками, которые фактически могут обращаться к данным одновременно, поскольку они работают на разных процессорах или процессорных ядрах, которые взаимодействуют друг с другом. Параллельная структура данных обычно считается находящейся в абстрактной среде хранения, называемой общей памятью, хотя эта память может быть физически реализована либо как «сильно связанная», либо как распределенная коллекция модулей хранения [2].

Анализ потокобезопасных структур данных позволит разработчикам ускорить процесс разработки параллельных программ, так как им не придется затрачивать время на изучение таких структур данных, а также позволит оптимизировать процесс разработки, так как не придется придумывать и применять специфичные алгоритмы для последовательных структур данных в контексте параллельных вычислений. Последующая оптимизация таких структур данных позволит повысить не только скорость разработки, но и эффективность итоговых программ.

Объектом является процесс анализа потокобезопасных структур данных.

Предметом разработки являются анализ потокобезопасных структур данных и реализация методов их оптимизации.

Цель работы состоит в анализе потокобезопасных структур данных и реализация методов их оптимизации.

Данная работа состоит из семи разделов.

В первом разделе приведена информация о структурах данных и параллелизме, описаны общие свойства и зависимости.

Второй раздел содержит методы общего анализа потокобезопасных структур данных.

Третий раздел посвящен примерам потокобезопасных структур данных.

В четвертом разделе изучаются сферы применения потокобезопасных структур данных.

В пятом разделе сравнительный анализ производительности потокобезопасных и непотокобезопасных структур данных.

В шестом разделе разрабатываются программы для анализа потокобезопасных структур данных.

В седьмом разделе проводится оценка и защита результатов интеллектуальной деятельности.

# **1 Обзор параллелизма с учетом структур данных**

Для последующего анализа потокобезопасных структур данных нужно подробно изучить, как работает параллелизм в операционной системе.

## **1.1 Параллелизм в операционной системе**

Параллелизм в операционных системах ­­– это способность операционной системы обрабатывать несколько процессов или задач одновременно. Увеличение спроса на высокопроизводительные вычисления сделало параллелизм одним из самых важных аспектов современных вычислительных систем. Поддержка параллелизма операционными системами позволяет выполнять несколько различных задач одновременно, что увеличивает эффективность использования ресурсов, повышает быстродействие и улучшает отзывчивость пользовательского интерфейса. Параллелизм нужен в современных операционных системах из-за высокого спроса на операции обработки данных в реальном времени, мультизадачность и параллельные вычисления. Веб-сервера, базы данных, научное моделирование, обработка мультимедиа, все эти сферы требуют повышения эффективности параллельных вычислений. Однако параллелизм также создает новые проблемы, такие как состояние гонки, взаимная блокировка и инверсия приоритетов, которыми необходимо эффективно управлять, чтобы обеспечить стабильность и надежность системы.

## **1.2 Принципы параллелизма**

Принципы параллелизма в операционных системах нужны для обеспечения эффективности выполнения нескольких процессов или потоков, не снижая эффективность и не блокируя друг друга.

1. Чередование − операция смены выполняющегося в текущий момент процесса или потока выполняющаяся для нескольких процессов или потоков. Планировщик операционной системы определяет, какой процесс или поток выполнять в любой момент времени. Чередование позволяет эффективно использовать ресурсы процессора и гарантирует, что все процессы или потоки получают справедливую долю процессорного времени.

2. Синхронизация – операция связи с целью координации нескольких процессов или потоков для обеспечения их работы без влияния друг на друга, понижающего эффективность. Данная операция выполняется с применением примитивов синхронизации, таких как блокировки, семафоры и мониторы. Эти примитивы позволяют процессам или потокам координировать доступ к общим ресурсам, например, таким как память и устройства ввода-вывода.

3. Взаимное исключение – операция, реализующая принцип обеспечения того, чтобы только один процесс или поток мог одновременно обращаться к общему ресурсу. Наиболее часто она реализуется с применением блокировок или семафоров, гарантируя, что несколько процессов или потоков не смогут получить одновременный доступ к общему ресурсу.

4. Предотвращение взаимоблокировки – операция, позволяющая избежать случай, когда два или более процессов или потоков ожидают друг друга, с целью освобождения ресурсов, что приводит к взаимоблокировке. Операционные системы используют различные методы, такие как графики распределения ресурсов и алгоритмы предотвращения взаимоблокировок, чтобы избежать взаимоблокировки.

5. Координация процессов или потоков – операция, координирующая действия процессов и потоков для достижения общей цели. Наиболее часто реализуется с помощью примитивов синхронизации, таких как семафоры или механизмы передачи сообщений, такие как каналы или сокеты.

6. Распределение ресурсов – операция, контролирующая справедливое и эффективное распределение ресурсов операционной системы, таких как память, процессорное время и устройства ввода-вывода, между несколькими процессами или потоками. Обычно это достигается с помощью алгоритмов планирования, таких как циклический перебор, планирование на основе приоритетов или в режиме реального времени [3].

## **1.3 Механизмы параллелизма**

Механизмы параллелизма необходимы для управления параллельными операциями в операционных системах и используются для обеспечения безопасного и эффективного доступа к системным ресурсам.

1. Процессы и Потоки − процессы и потоки являются основными ресурсами операционной системы, обеспечивающими параллелизм. Процесс — это экземпляр программы, который может выполняться независимо в изолированном пространстве памяти. Поток — это облегченный процесс, который использует то же пространство памяти, что и его родительский процесс и всегда является его частью.

2. Примитивы синхронизации − примитивы синхронизации являются частью операционной системы, использующейся для координации доступа к общим ресурсам между несколькими процессами или потоками. Распространенные примитивы синхронизации включают семафоры, мьютексы и переменные условия.

3. Алгоритмы планирования − алгоритмы планирования являются частью операционной системы, использующейся, чтобы определить, какой процесс или поток должен выполняться следующим. Распространенные алгоритмы планирования включают циклическое планирование, планирование на основе приоритетов и планирование в реальном времени.

4. Передача сообщений – механизм передачи сообщений используется для обмена данными между процессами или потоками. Сообщения могут отправляться синхронно или асинхронно и могут содержать в себе данные, сигналы или уведомления.

5. Управление памятью – механизм управления памятью является одной из наиболее важных частей операционной системы, используемой для выделения ресурсов памяти и управления ими. Этот механизм обеспечивает собственное пространство памяти для каждого процесса или поток и позволяет им безопасно обращаться к памяти, не создавая помех другим процессам или потокам.

6. Обработка прерываний – механизм обработки прерываний обеспечивает обработку особых сигналов (прерываний), посылаемых аппаратными устройствами операционной системе, указывающих на то, что они требуют внимания. Операционная система использует механизмы обработки прерываний для остановки текущего процесса или потока, сохранения его состояния и выполнения специального обработчика прерываний для обработки запроса устройства [4].

## **1.4 Преимущества и проблемы параллелизма**

Параллелизм обеспечивает несколько преимуществ в операционных системах.

1. Улучшенная производительность. Использование параллелизма позволяет увеличить производительность системы путем одновременного выполнения нескольких задач. Это достигается благодаря использованию нескольких процессоров или потоков, которые позволяют задачам выполняться параллельно, что в свою очередь уменьшает общее время обработки.

2. Использование ресурсов. Использование параллелизма позволяет эффективнее расходовать системные ресурсы, такие как процессор, память и устройства ввода-вывода. Благодаря возможности выполнения нескольких задач одновременно, система может более полно и эффективно использовать доступные ей ресурсы.

3. Отзывчивость. Использование параллелизма может повысить отзывчивость системы, позволяя одновременно выполнять несколько задач. Это особенно важно для систем реального времени и интерактивных приложений, таких как игры и мультимедиа, где быстрое реагирование на пользовательский ввод является критически важным.

4. Масштабируемость. Использование параллелизма может улучшить масштабируемость системы, позволяя ей обрабатывать все большее количество задач и пользователей без ухудшения производительности.

5. Отказоустойчивость. Использование параллелизма может улучшить отказоустойчивость системы, позволяя задачам выполняться независимо друг от друга. Если одна задача завершается неудачей, это не влияет на выполнение других задач, что позволяет системе продолжать работу без простоев и снижения общей производительности.

6. Отладка и диагностика проблем, связанных с параллелизмом, может быть сложной задачей и требует тщательной разработки и внедрения соответствующих механизмов, чтобы избежать подобных проблем.

7. Условия гонки возникают, когда порядок и время выполнения событий становятся критически важными для выходных данных системы, что приводит к непредсказуемому поведению. Если несколько процессов или потоков одновременно пытаются получить доступ к общим ресурсам, это может привести к состоянию гонки и неправильным результатам.

8. Взаимоблокировки возникают, когда два или более процессов или потоков ожидают друг друга, чтобы освободить ресурсы, которые нужны им для продолжения своей работы, что приводит к циклическому ожиданию. Это может произойти, когда несколько процессов или потоков конкурируют за эксклюзивный доступ к общим ресурсам, таким как файлы, базы данных или устройства ввода-вывода.

9. Нехватка ресурсов может привести к ситуации, когда процесс или поток не могут получить доступ к нужному для выполнения своей задачи ресурсу из-за его блокировки другими процессами или потоками. Это может привести к застреванию процесса или потока в цикле, не позволяющем процессам или потоками продвинуться в выполнении задачи.

10. Инверсия приоритета возникает, когда процесс или поток с более низким приоритетом удерживает ресурс, необходимый для выполнения задачи процесса или потока с более высоким приоритетом, что приводит к задержкам в работе высокоприоритетных процессов или потоков.

11. Согласованность памяти — это способность параллельной системы обеспечивать правильный порядок операций с памятью между различными процессами или потоками. Несоответствие порядка операций может привести к неправильному поведению системы, поэтому важно обеспечить согласованность памяти в параллельной среде.

12. Методы, предназначенные для предотвращения взаимоблокировок, могут помочь избежать данной проблемы, но в то же время могут привести к низкой эффективности использования ресурсов, а также даже к остановке некоторых процессов или потоков [5].

**1.5 Развитие параллелизма**

Перспективные направления развития параллелизма в операционных системах включают в себя разработку новых моделей программирования и языков, аппаратной поддержки и алгоритмов, которые будут способствовать более эффективному и безопасному использованию параллельных вычислений, а также решению проблем, таких как конфликты доступа к общим ресурсам, взаимоблокировки и нехватка ресурсов.

1. Новые модели и языки программирования. Новые модели программирования и языки становятся доступными для разработки параллельного программного обеспечения, с целью упрощения процесса и уменьшения возможных ошибок. Rust, как пример языка программирования, использует модель владения для обеспечения безопасности памяти и избежания перегрузки данных, а Go и Swift также поддерживают параллелизм встроенными средствами.

2. Аппаратная поддержка параллелизма. Быстрое развитие аппаратной поддержки параллелизма связано с возрастающей популярностью многоядерных процессоров, графических процессоров и специализированных ускорителей. В будущем, аппаратная поддержка будет улучшаться с целью обеспечения более эффективной интеграции между аппаратным и программным обеспечением, что позволит сократить затраты и повысить производительность. Кроме того, разрабатываются новые аппаратные архитектуры, такие как транзакционная память и механизмы принудительной синхронизации, чтобы уменьшить вероятность ошибок, связанных с параллелизмом.

3. Новые алгоритмы и структуры данных. Новые алгоритмы и структуры данных разрабатываются для обеспечения более эффективного и масштабируемого параллелизма. Например, применение безблокировочных структур данных и алгоритмов с примитивами синхронизации, которые не блокируют, позволяет нескольким задачам одновременно получать доступ к общим данным. В дополнение к этому, новые алгоритмы и структуры данных разрабатываются для обеспечения распределенных вычислений и параллельной обработки, что позволяет выполнять задачи на нескольких машинах или процессорах.

Параллелизм играет ключевую роль в современных операционных системах и обеспечивает высокую производительность, масштабируемость и надежность приложений и систем. Несмотря на многочисленные преимущества, параллелизм также может вызывать проблемы, описанные ранее. Для управления параллелизмом операционные системы предоставляют множество различных инструментов. Развитие новых технологий и приложений требует более эффективных и масштабируемых механизмов параллелизма [6].

# **2 Методы общего анализа потокобезопасных структур данных**

Существуют различные методы анализа потокобезопасных структур данных, которые помогают обнаружить и исправить ошибки параллелизма и согласованности памяти. Основными методами можно считать статический и динамический анализ, а также методы формальной верификации.

## **2.1 Статический анализ**

Статический анализ — это метод анализа программного кода без его фактического выполнения. Статический анализатор может проверять код на наличие ошибок, предупреждать о потенциальных проблемах, определять уязвимости и прочее.

В контексте потокобезопасных структур данных, статический анализ может помочь выявить возможные проблемы с параллельным доступом к данным, такие как условия гонки, гонки на запись и прочие ошибки синхронизации. Он также может помочь выявить потенциально небезопасные конструкции в коде, которые могут привести к конфликтам при параллельном доступе к данным.

Статический анализ может быть полезен для разработчиков, которые могут использовать его для улучшения качества своего кода и уменьшения вероятности ошибок в параллельной среде. Также статический анализ может быть встроен в процесс непрерывной интеграции и доставки, что позволяет автоматически проверять код на наличие проблем перед его релизом [7].

## **2.2 Динамический анализ**

Динамический анализ для потокобезопасных структур данных — это метод анализа, который основывается на наблюдении за выполнением программы во время ее выполнения. Он позволяет выявлять проблемы с потокобезопасностью в реальном времени, что может быть особенно полезным для обнаружения проблем, которые могут произойти только при определенных условиях во время выполнения программы.

Один из подходов к динамическому анализу потокобезопасных структур данных — это использование инструментов для отслеживания состояния памяти во время выполнения программы. Эти инструменты могут использоваться для обнаружения условий гонки, когда два или более потока пытаются одновременно получить доступ к общим данным. Они также могут использоваться для обнаружения других проблем, таких как взаимоблокировки и неправильное использование примитивов синхронизации.

Еще один подход к динамическому анализу потокобезопасных структур данных — это использование методов тестирования, таких как тестирование на основе свойств и моделирование поведения программы. Эти методы могут помочь выявить проблемы потокобезопасности, которые могут быть упущены при статическом анализе, например, когда состояние программы зависит от внешних факторов, которые нельзя точно предсказать на этапе компиляции.

Динамический анализ для потокобезопасных структур данных имеет свои ограничения. Он может быть очень ресурсоемким и может не обнаружить все проблемы с потокобезопасностью, особенно если они происходят только при очень редких условиях. Кроме того, динамический анализ может быть недостаточно точным, если программа имеет очень высокую сложность или если она работает в распределенной среде. Тем не менее, динамический анализ остается важным инструментом для обеспечения безопасности и надежности программного обеспечения, особенно в условиях постоянно меняющихся условий работы и конфигураций систем [8].

## **2.3 Методы формальной верификации**

Формальная верификация — это метод формального доказательства корректности программного обеспечения. Она используется для проверки соответствия программного кода спецификации, которая формулирует требования к программе. Формальная верификация может помочь гарантировать отсутствие ошибок, таких как условия гонки, при работе с потокобезопасными структурами данных.

Существует несколько методов формальной верификации для потокобезопасных структур данных:

1. Модельная проверка (Model checking)

Основная идея метода заключается в том, что модель программы представляется в виде конечного автомата или другой формализованной структуры данных, которая отображает все возможные состояния программы. Затем система модельной проверки проходит по всем возможным путям выполнения программы, проверяя каждый шаг на соответствие спецификации.

Для этого система модельной проверки использует формальные методы и техники, такие как символьное выполнение, анализ ограничений, анализ графов и другие. Эти методы позволяют проверять различные аспекты программы, такие как правильность работы алгоритмов, отсутствие доступов к несуществующим объектам памяти, отсутствие гонок данных и другие.

2. Доказательства программ (Program proofs)

Это метод формальной верификации, который использует математическую логику для доказательства корректности программного кода. С помощью этого метода можно доказать, что программа соответствует своей спецификации, что все возможные пути выполнения программы корректны и что программа работает правильно во всех возможных сценариях.

Для доказательства корректности программного кода используются формальные методы математической логики, такие как логика высказываний, логика предикатов, теория типов и другие. Эти методы позволяют формально доказать, что программа работает корректно, используя математические доказательства.

3. Система типов (Type systems)

Это метод формальной верификации, используемый в компьютерных науках, для классификации различных значений, которые могут быть присвоены переменным в рамках программы. Система типов определяет, какие типы значений могут быть использованы в конкретных контекстах, и позволяет обеспечить безопасность типов, проверяя правильность типов, используемых в программе.

Системы типов широко используются в компиляторах и интерпретаторах для проверки корректности программного кода и выявления ошибок типизации. Системы типов могут обнаруживать множество ошибок, таких как неправильное использование переменных, вызов несуществующих функций и методов, некорректное использование потокобезопасных структур данных и т.д.

4. Автоматическое доказательство теорем (Automated theorem proving)

Это метод формальной верификации, основанный на использовании компьютерных программ для автоматического поиска и доказательства математических теорем. Этот метод может быть использован для проверки корректности программного кода, в том числе и алгоритмов работы с потокобезопасными структурами данных.

Автоматическое доказательство теорем позволяет проверять математические утверждения на их истинность, используя формальную логику и правила вывода. Это позволяет доказывать корректность программного кода, используя формальные методы, а не только тестирование и эмпирические методы.

Однако, автоматическое доказательство теорем является достаточно сложным процессом и может потребовать значительных вычислительных ресурсов и времени для поиска доказательств. Кроме того, этот метод не может гарантировать полную корректность программного кода, поскольку он ограничен формальной логикой и правилами вывода.

Каждый из этих методов формальной верификации может быть использован для проверки корректности работы с потокобезопасными структурами данных в программном коде. Однако, каждый метод имеет свои преимущества и недостатки, и его выбор зависит от конкретного случая и требований к программному коду [9].

5. Анализ потокобезопасных структур данных с точки зрения эффективности эффективности включает оценку того, насколько эффективны эти структуры данных при использовании в многопоточных приложениях.

Существует несколько критериев для оценки эффективности потокобезопасных структур данных.

1. Скорость работы — это один из самых важных критериев эффективности. Чем быстрее работает потокобезопасная структура данных, тем лучше. Скорость работы зависит от различных факторов, таких как алгоритм, используемый в структуре данных, размер данных, количество потоков, использующих структуру данных и т.д.

2. Использование ресурсов - эффективная потокобезопасная структура данных должна использовать минимальное количество ресурсов, таких как память, процессорное время и т.д.

3. Масштабируемость - эффективная потокобезопасная структура данных должна быть масштабируемой, то есть должна работать эффективно независимо от количества потоков, использующих ее.

4. Безопасность - эффективная потокобезопасная структура данных должна обеспечивать безопасность данных и исключать возможность появления состояний гонки и других ошибок, связанных с многопоточностью.

5. Простота использования - эффективная потокобезопасная структура данных должна быть проста в использовании и не требовать от программиста специальных навыков или знаний в области многопоточного программирования.

Для оценки эффективности потокобезопасных структур данных можно использовать различные тесты производительности, например, измерять время выполнения операций в многопоточной среде на разных размерах данных. Также можно провести анализ потребляемых ресурсов и масштабируемости структур данных.

Важно отметить, что выбор потокобезопасной структуры данных должен быть основан не только на эффективности, но и на требованиях конкретного приложения. Например, в некоторых случаях более важна безопасность данных, чем скорость и масштабируемость.

Также для оценки эффективности потокобезопасных структур данных можно использовать сравнительный анализ существующих решений. Например, можно сравнить производительность различных реализаций одной и той же структуры данных или сравнить производительность разных структур данных на одних и тех же тестовых данных и при одинаковых условиях использования.

Существует также множество инструментов и методик для оценки производительности и эффективности многопоточных приложений в целом, включая профилирование, бенчмаркинг и тестирование под нагрузкой.

Наконец, эффективность потокобезопасных структур данных может зависеть от специфики аппаратной и программной среды, на которой они используются. Поэтому при выборе структуры данных следует учитывать не только требования конкретного приложения, но и особенности используемой среды, такие как тип процессора, объем доступной памяти и т.д.

В целом, эффективный анализ потокобезопасных структур данных требует комплексного подхода, включающего не только оценку скорости работы и использования ресурсов, но и анализ многопоточности, безопасности и масштабируемости структур данных [10].

# **3 Примеры потокобезопасных структур данных**

Потокобезопасные структуры данных являются особенно важными для многопоточных приложений, поскольку они обеспечивают безопасный доступ к общей памяти из разных потоков исполнения. Некоторые из наиболее распространенных потокобезопасных структур данных включают в себя следующее [11].

**Блокирующая очередь (Locked Queue)**

Это потокобезопасная структура данных, которая обычно используется для обмена данными между потоками в многопоточных приложениях. Эта структура данных является очередью, которая имеет блокировки для синхронизации доступа к элементам очереди. Блокировки гарантируют, что только один поток может получить доступ к элементам очереди в один момент времени.

Блокирующие очереди часто используются для реализации шаблона проектирования "Producer-Consumer". В этом шаблоне один или несколько производителей (producer) добавляют элементы в очередь, а один или несколько потребителей (consumer) получают элементы из очереди и обрабатывают их. Блокирующие очереди обеспечивают синхронизацию между производителями и потребителями, чтобы гарантировать безопасность и правильность обработки данных.Очередь представлена на рисунке 3.1.

**Конвейерная очередь (Pipeline Queue)**

Это структура данных, которая позволяет организовать эффективную параллельную обработку данных в нескольких этапах, называемых также конвейерами. Каждый конвейер представляет собой отдельный поток, который может обрабатывать данные независимо от других конвейеров. Конвейеры могут выполнять разные операции над входными данными, например, преобразование данных, фильтрацию, анализ и т.д.

В конвейерной очереди, каждый конвейер принимает данные из предыдущего конвейера, обрабатывает их и передает результат следующему конвейеру. Конвейеры работают параллельно, каждый в своем потоке, что позволяет существенно ускорить обработку данных. Для того чтобы обеспечить эффективность работы конвейера, необходимо минимизировать задержки между этапами и избежать блокировки потоков.

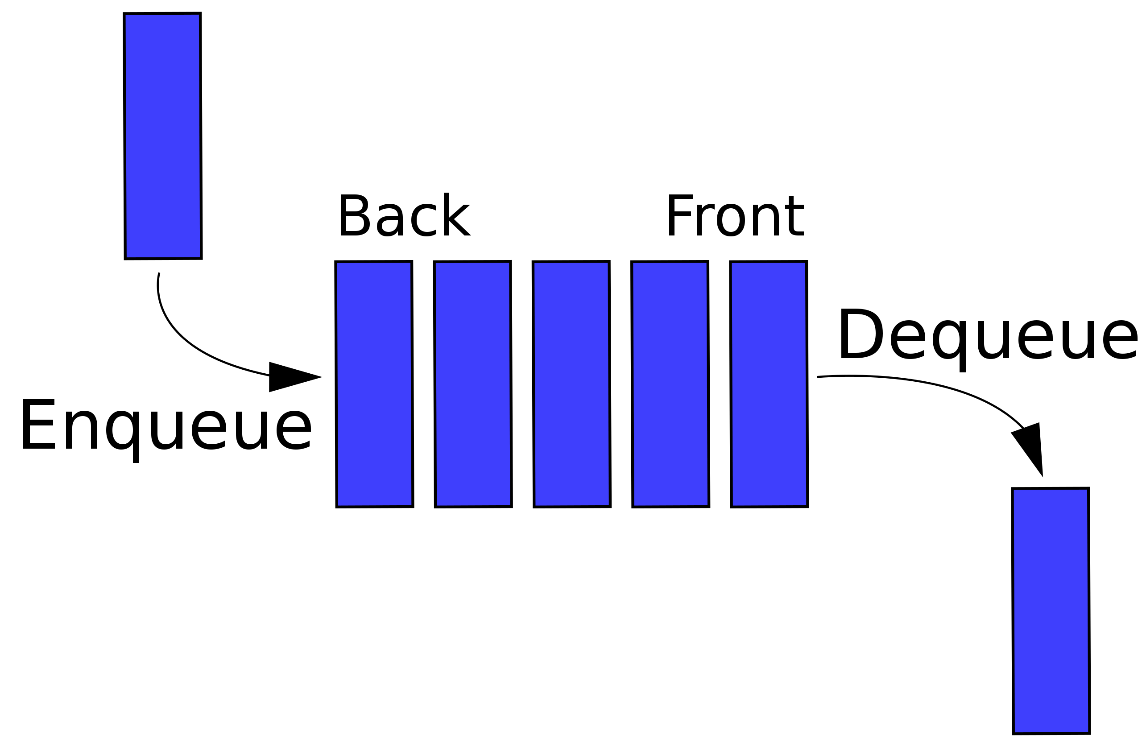


Рисунок 3.1 - Очередь

**Потокобезопасные хеш-таблицы (Thread-Safe Hash Table)**

Это структуры данных, которые обеспечивают потокобезопасный доступ к хеш-таблицам. Общий подход к реализации потокобезопасных хеш-таблиц заключается в использовании мьютекса или других механизмов блокировки для защиты доступа к таблице и ее элементам. В большинстве реализаций потокобезопасных хеш-таблиц используются мьютексы на уровне таблицы, чтобы гарантировать эксклюзивный доступ к таблице, когда один поток выполняет операцию вставки, удаления или обновления таблицы.

Однако этот подход может сильно замедлять работу приложения, особенно в случае больших таблиц, поскольку другие потоки должны ждать освобождения мьютекса, чтобы получить доступ к таблице. Чтобы уменьшить этот эффект, в некоторых реализациях используются мьютексы на уровне каждого элемента таблицы, что позволяет нескольким потокам выполнять операции над таблицей одновременно, если они работают с разными элементами таблицы. Хэш-таблица представлена на рисунке 3.2.

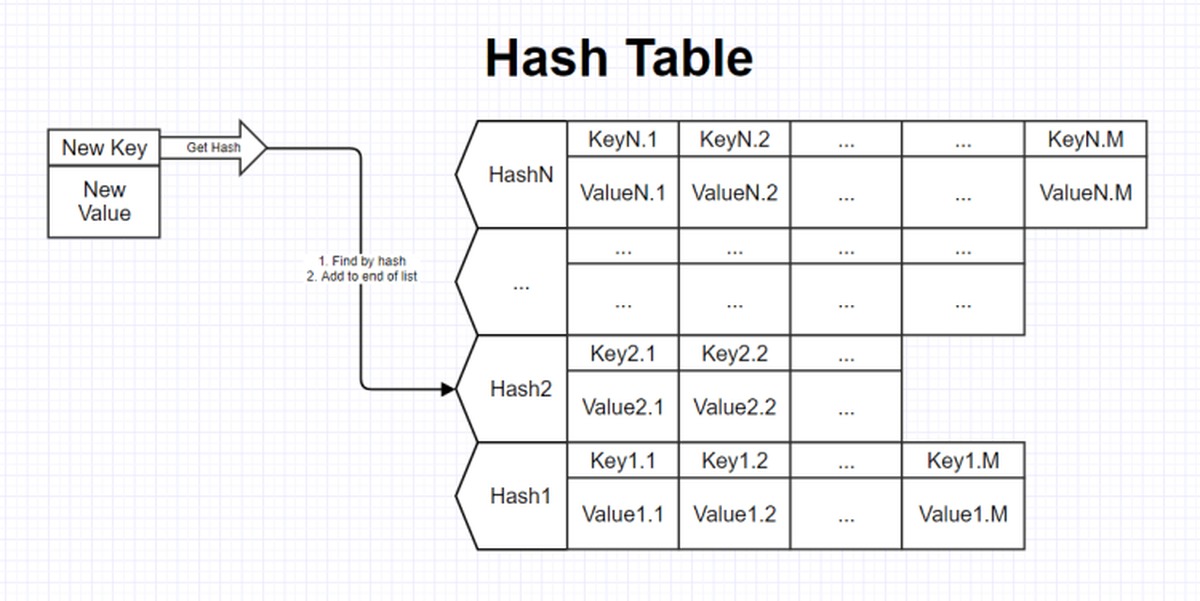


Рисунок 3.2 – Хэш-таблица

**Потокобезопасные связанные списки (Thread-Safe Linked List)**

Это структуры данных, которые позволяют безопасно добавлять, удалять и обновлять элементы в связанном списке из нескольких потоков. В потокобезопасных связанных списках обычно используется блокировка на уровне каждого элемента списка, чтобы гарантировать безопасный доступ к данным из нескольких потоков одновременно. Связанный список представлен на рисунке 3.3.

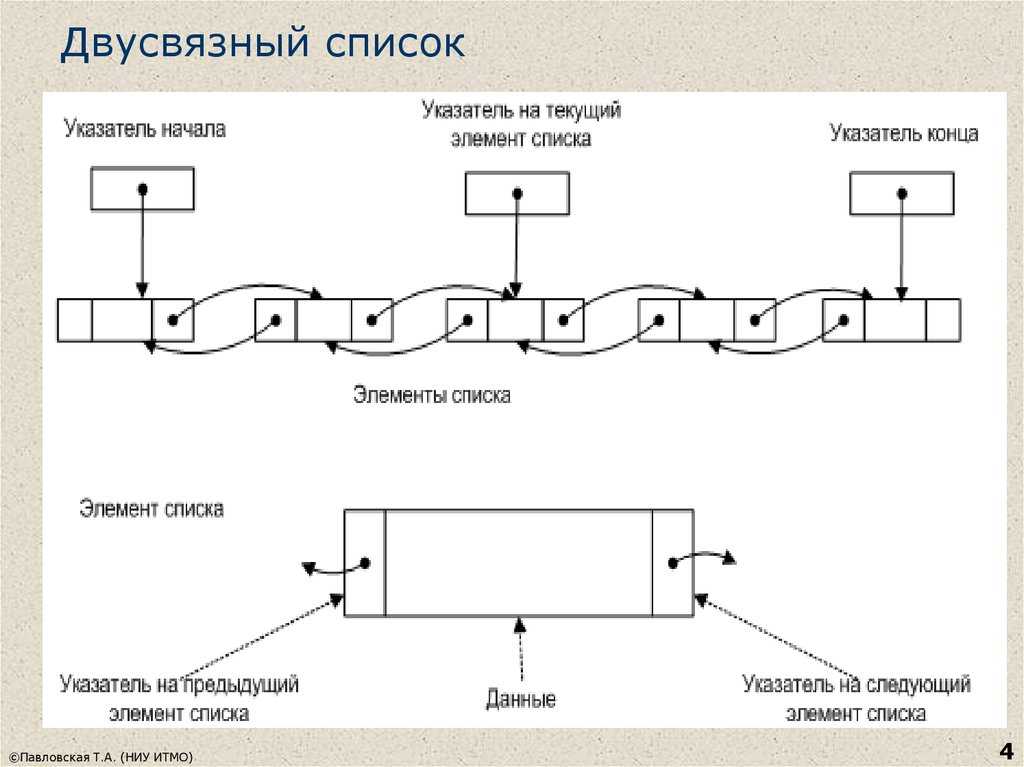


Рисунок 3.3 – Связанный список

**Атомарные переменные (Atomic Variables)**

Это специальные переменные, которые гарантируют безопасный доступ к данным из нескольких потоков одновременно. Атомарные переменные предоставляют примитивы синхронизации, которые позволяют выполнять операции на переменных в атомарной последовательности, так что состояние переменной всегда остается валидным.

**Блокированные стеки (Locked Stack)**

Это потокобезопасная структура данных, которая поддерживает вставку и удаление элементов из стека из разных потоков. В блокированных стеках используются блокировки для обеспечения безопасного доступа к данным из нескольких потоков одновременно. Блокировки могут быть установлены на уровне стека или на уровне каждого элемента в стеке.

Когда поток пытается вставить или удалить элемент из стека, он блокирует стек и проверяет, можно ли выполнить операцию. Если операция возможна, то элемент вставляется или удаляется, а затем блокировка снимается. Если операция невозможна, то поток ожидает, пока другой поток не завершит операцию и не освободит блокировку.

Блокированные стеки могут быть реализованы с использованием различных алгоритмов блокировки, таких как блокировка на чтение/запись, блокировка на уровне элементов, и другие. Стек представлен на рисунке 3.4.

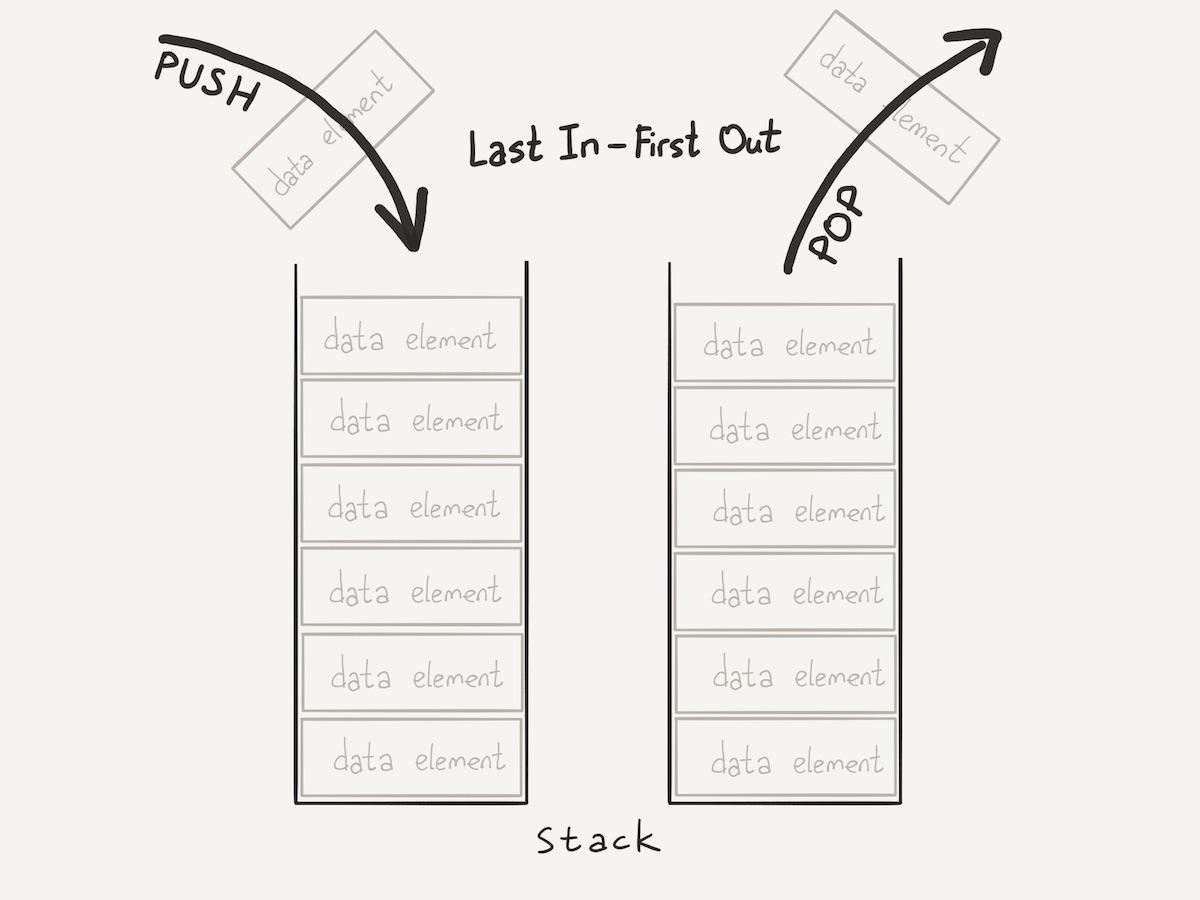


Рисунок 3.4 – Стек

**Кольцевой буфер (Ring Buffer)**

Это структура данных, которая используется для обмена данными между двумя или более потоками. Кольцевой буфер представляет собой буфер фиксированного размера, который образует кольцо. Каждый поток может записывать данные в буфер или считывать их из него, используя указатели на начало и конец буфера. Таким образом, кольцевой буфер позволяет многопоточным приложениям обмениваться данными без блокировки и ожидания друг друга. Кольцевой буфер представлен на рисунке 3.5.

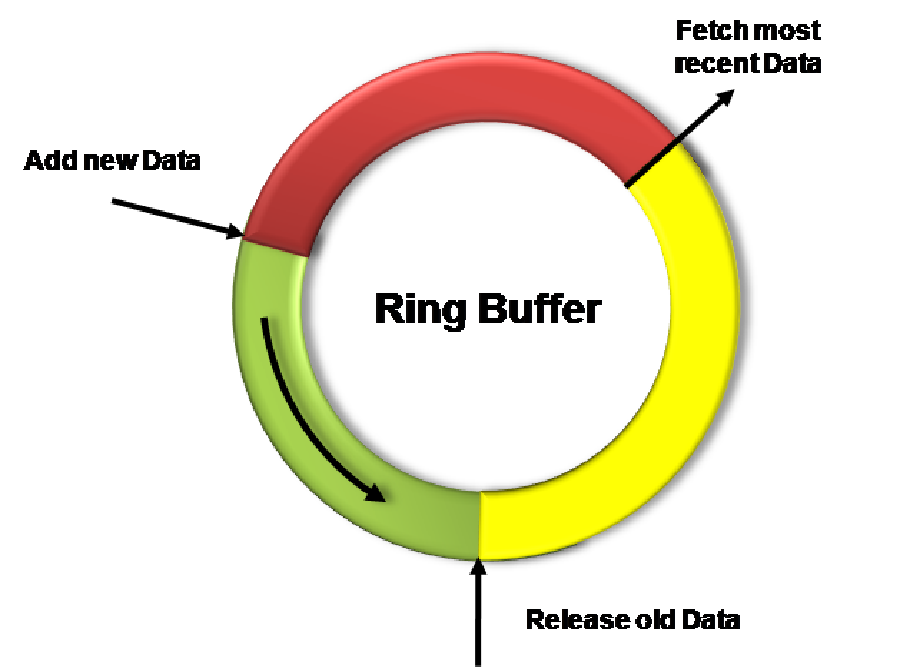


Рисунок 3.5 – Кольцевой буфер

# **4 Применение потокобезопасных структур данных**

Потокобезопасные структуры данных широко используются в различных областях, связанных с программированием, в задачах, где несколько потоков одновременно обращаются к одним и тем же данным. Ниже представлены способы применения потокобезопасных структур данных в различных областях.

1. Веб-разработка. В веб-разработке потокобезопасные структуры данных могут использоваться для обработки запросов клиентов на сервере. Например, в языке программирования Python для безопасной работы с базами данных используется библиотека SQLAlchemy, которая предоставляет потокобезопасный интерфейс для работы с базами данных.

2. Игровая индустрия. В игровой индустрии потокобезопасные структуры данных используются для разработки многопоточных игр. Например, в языке программирования C# для безопасной работы с данными используется класс ConcurrentQueue, который позволяет безопасно работать с очередями из нескольких потоков.

3. Медицинская техника. В медицинской технике потокобезопасные структуры данных используются для обработки медицинских данных и для управления медицинским оборудованием. Например, в языке программирования Java для безопасной работы с данными используется класс ConcurrentHashMap, который позволяет безопасно работать с хэш-таблицами из нескольких потоков.

4. Финансовые технологии. В финансовых технологиях потокобезопасные структуры данных используются для обработки финансовых данных, таких как транзакции и ордера. Например, в языке программирования C++ для безопасной работы с данными используется класс std::atomic, который позволяет безопасно работать с атомарными переменными из нескольких потоков.

5. Машинное обучение. В машинном обучении потокобезопасные структуры данных используются для параллельной обработки больших объемов данных. Например, в языке программирования Python для безопасной работы с данными используется библиотека PyTorch, которая предоставляет потокобезопасный интерфейс для работы с нейронными сетями.

6. Криптография. В криптографии потокобезопасные структуры данных используются для обработки криптографических ключей и секретной информации. Например, в языке программирования Java для безопасной работы с данными используется класс SecureRandom, который позволяет генерировать случайные числа из нескольких потоков без возможности получения одинаковых значений в разных потоках.

7. Большие данные. В области больших данных потокобезопасные структуры данных используются для параллельной обработки и анализа больших объемов данных. Например, в языке программирования Scala для безопасной работы с данными используется класс ConcurrentMap, который позволяет безопасно работать с хэш-таблицами из нескольких потоков.

8. Интернет вещей. В области интернета вещей потокобезопасные структуры данных используются для обработки данных, получаемых от различных датчиков и устройств. Например, в языке программирования C для безопасной работы с данными используется класс pthread\_mutex\_t, который позволяет синхронизировать доступ к данным из нескольких потоков.

И это только наиболее популярные сферы применения. Потокобезопасные структуры данных используются и развиваются практически во всех сферах программирования,

# **5 Сравнительный анализ производительности потокобезопасных и непотокобезопасных структур данных**

Сравнительный анализ производительности потокобезопасных и непотокобезопасных структур данных зависит от многих факторов, таких как язык программирования, используемый алгоритм, количество потоков, количество операций чтения и записи, размер данных и т.д. Однако, можно выделить несколько общих аспектов, которые влияют на производительность потокобезопасных и непотокобезопасных структур данных [12].

1. Затраты на синхронизацию. Потокобезопасные структуры данных требуют дополнительных затрат на синхронизацию доступа к данным из разных потоков, что может приводить к снижению производительности. В то же время, непотокобезопасные структуры данных не требуют таких затрат и могут работать быстрее в однопоточных приложениях.

2. Конкуренция за ресурсы. В многопоточных приложениях потокобезопасные структуры данных могут снижать конкуренцию за ресурсы, так как позволяют нескольким потокам одновременно обращаться к данным без блокировки. В то же время, непотокобезопасные структуры данных могут приводить к конкуренции за ресурсы, что может привести к блокировке и снижению производительности.

3. Параллелизм. Потокобезопасные структуры данных позволяют эффективно использовать параллелизм и ускорить выполнение операций в многопоточных приложениях. В то же время, непотокобезопасные структуры данных могут приводить к некорректным результатам при параллельном выполнении операций.

4. Размер данных. Потокобезопасные структуры данных могут иметь больший размер и большую сложность, чем непотокобезопасные структуры данных, что может приводить к снижению производительности при работе с большими объемами данных. Однако, это зависит от конкретной структуры данных и ее реализации.

5. Ошибки и безопасность. Потокобезопасные структуры данных предотвращают ошибки, которые могут возникать при работе с данными из нескольких потоков. Также они обеспечивают безопасность данных при работе в многопоточных приложениях. В то же время, непотокобезопасные структуры данных могут нарушать целостность данных при работе в многопоточной среде, что может привести к некорректным результатам и ошибкам.

В целом, сравнительный анализ производительности потокобезопасных и непотокобезопасных структур данных должен учитывать конкретные условия и требования приложения. Если в приложении используется многопоточность, то потокобезопасные структуры данных могут быть более эффективными. Если же приложение работает только в однопоточном режиме, то непотокобезопасные структуры данных могут быть более быстрыми [13].

Некоторые исследования показывают, что производительность потокобезопасных и непотокобезопасных структур данных зависит от типа операций, которые выполняются над данными. Например, при выполнении операций чтения потокобезопасные структуры данных могут работать медленнее, чем непотокобезопасные. Однако, при выполнении операций записи, потокобезопасные структуры данных могут работать быстрее, так как позволяют нескольким потокам одновременно записывать данные.

Также важно учитывать конкретную реализацию потокобезопасных и непотокобезопасных структур данных, так как разные реализации могут иметь различную производительность. Кроме того, производительность может зависеть от характеристик конкретного процессора и операционной системы, на которых запускается приложение.

В целом, выбор между потокобезопасными и непотокобезопасными структурами данных зависит от конкретных требований и условий приложения. Если в приложении используется многопоточность, то потокобезопасные структуры данных могут быть более предпочтительными. В остальных случаях можно использовать непотокобезопасные структуры данных, которые могут работать быстрее и иметь более простую реализацию.

# **6 Разработка программы для анализа потокобезопасных структур данных**

В рамках разработки программного обеспечения требуется выполнить несколько действий: определиться с архитектурой программного обеспечения, выбрать рабочие библиотеки, инструменты, язык программирования и среду программирования.

## **6.1 Архитектура программного обеспечения**

Архитектура программного обеспечения состоит из трех компонентов.

1. Компонент для реализации структур данных. Данный компонент отвечает за реализацию различных структур данных и получение текстовых результатов тестов.

2. Компонент для визуализации результатов. Данный компонент

отвечает визуализацию полученных данных в графическом виде для последующего анализа.

## **6.2 Используемые библиотеки и инструменты**

Для разработки программного обеспечения были использованы

следующие библиотеки и инструменты.

### **6.2.1 Язык программирования**

Для реализации потокобезопасных структур данных был выбран языка программирования С++. С++ является статически типизированным, компилируемым языком программирования. Вот некоторые из свойств языка С++.

1. Объектно-ориентированное программирование. C++ поддерживает парадигму объектно-ориентированного программирования, что позволяет разработчикам создавать объекты, которые могут быть инкапсулированы, наследоваться и использоваться повторно.

2. Низкоуровневый доступ к памяти. C++ предоставляет низкоуровневый доступ к памяти, что позволяет программистам управлять использованием памяти и повысить эффективность программы.

3. Множественное наследование. C++ позволяет наследоваться от нескольких базовых классов, что дает возможность разработчикам создавать более гибкие и мощные классы.

4. Поддержка шаблонов. C++ поддерживает шаблоны функций и классов, что позволяет программистам создавать обобщенный код, который может работать с разными типами данных.

5. Многопоточность. C++ предоставляет поддержку многопоточности, что позволяет разработчикам создавать многопоточные приложения для параллельной обработки данных.

6. Быстродействие. C++ является одним из самых быстрых языков программирования, что делает его особенно подходящим для разработки системного программного обеспечения и игровых движков.

7. Платформенная независимость. C++ может быть скомпилирован и запущен на разных платформах, таких как Windows, Linux и macOS.

8. Большое сообщество разработчиков. C++ имеет огромное сообщество разработчиков и широкий спектр библиотек, что делает его более доступным и обширным языком программирования.

9. Оператор перегрузки. C++ позволяет программистам переопределять операторы для пользовательских типов данных, что позволяет более элегантно и удобно работать с объектами.

10. Указатели и ссылки. C++ поддерживает указатели и ссылки, которые предоставляют более точный контроль над памятью и производительностью программы.

11. Низкоуровневое программирование. C++ позволяет программистам писать низкоуровневый код, например, для управления аппаратным обеспечением, что делает его особенно подходящим для системного программирования.

12. Богатое наследие. C++ является продолжением языка С, который был разработан еще в 70-х годах и имеет богатое наследие в программировании операционных систем и системного программного обеспечения.

13. Кроссплатформенная библиотека стандартных шаблонов. C++ имеет стандартную библиотеку шаблонов, которая включает в себя множество структур данных, алгоритмов и контейнеров, что упрощает написание кода и ускоряет разработку.

14. Метапрограммирование. C++ позволяет программистам создавать программы, которые могут генерировать и изменять свой собственный код, что делает его особенно мощным инструментом для разработки программных систем.

15. Расширяемость. C++ позволяет программистам расширять язык путем создания пользовательских типов данных и функций, что позволяет создавать более выразительные и гибкие программы.

Главными плюсами использования данного языка в контексте решаемых задач являются наличие низкоуровневого доступа к памяти, расширяемость и низкий уровень абстракции. Язык обладает большим количеством стандартных и дополнительных библиотек [14].

Для визуализации результатов был выбран язык программирования Python. Python — это интерпретируемый язык программирования с открытым исходным кодом, который используется для различных целей, от веб-разработки до анализа данных и машинного обучения. Некоторые из основных свойств Python включают следующее.

1. Простота. Python имеет простой и понятный синтаксис, что делает его легким в изучении и использовании.

2. Интерпретируемость. Python является интерпретируемым языком, что означает, что он не требует компиляции перед выполнением, что упрощает и ускоряет процесс разработки.

3. Кроссплатформенность. Python может быть запущен на разных платформах, включая Windows, Linux, MacOS, что обеспечивает универсальность и переносимость разработанного кода.

4. Большая библиотека. Python имеет огромную библиотеку модулей и инструментов, которые могут быть использованы для различных задач, таких как работа с базами данных, создание графических интерфейсов пользователя, анализ данных и т.д.

5. Объектно-ориентированный подход. Python поддерживает объектно-ориентированный подход к программированию, что делает код более гибким и модульным.

6. Динамическая типизация. Python является языком с динамической типизацией, что означает, что тип переменной определяется автоматически во время выполнения программы.

7. Многопоточность. Python поддерживает многопоточность, что позволяет создавать многопоточные приложения для параллельной обработки данных.

8. Широкое применение. Python используется в различных областях, таких как разработка веб-приложений, научные и исследовательские задачи, анализ данных, машинное обучение и т.д.

9. Open source. Python — это язык с открытым исходным кодом, что делает его доступным для всех и позволяет разработчикам легко расширять его возможности.

10. Легкость интеграции. Python может легко интегрироваться с другими языками программирования, такими как C++, Java, что позволяет использовать Python для определенных задач в составе других приложений.

Главными плюсами использования Python в контексте решаемых задач являются простота, интерпретируемость и высокая скорость разработки. Язык обладает большим количеством стандартных и дополнительных библиотек [15]

### **6.2.2 Среда разработки**

Для реализации потокобезопасных структур данных была выбран Microsoft Visual Studio. Microsoft Visual Studio — это среда разработки программного обеспечения, которая имеет множество преимуществ.

1. Интегрированная среда разработки. Visual Studio предоставляет разработчикам все необходимые инструменты для создания, отладки, тестирования и развертывания программного обеспечения в одной удобной среде.

2. Поддержка множества языков программирования. Visual Studio поддерживает множество языков программирования, включая C#, C++, F#, Visual Basic и другие.

3. Интеллектуальная подсказка кода. Visual Studio предоставляет функцию автодополнения, которая помогает разработчикам написать код быстрее и с меньшими ошибками.

4. Интеграция с Git. Visual Studio имеет встроенную поддержку системы контроля версий Git, что облегчает работу в команде и управление исходным кодом.

5. Отладка. Visual Studio предоставляет мощный отладчик, который помогает быстро находить и исправлять ошибки в коде.

6. Расширяемость. Visual Studio имеет множество плагинов и расширений, которые помогают разработчикам улучшить свой опыт работы и повысить производительность.

7. Мощный редактор форм. Visual Studio имеет инструменты для создания профессиональных пользовательских интерфейсов, что делает разработку приложений более удобной и простой.

8. Поддержка многопоточности. Visual Studio имеет мощные инструменты для разработки многопоточных приложений, что позволяет создавать более эффективное и быстрое программное обеспечение.

В целом, Microsoft Visual Studio является мощной и полезной средой разработки программного обеспечения, которая облегчает и ускоряет процесс разработки, улучшает качество кода и повышает производительность разработчика.

Для визуализации результатов был выбран Microsoft Visual Studio Code. Microsoft Visual Studio Code — это бесплатный и легковесный редактор кода, который также имеет множество преимуществ [16].

1. Кросс-платформенность. Visual Studio Code работает на Windows, macOS и Linux, что позволяет разработчикам использовать его на любой платформе.

2. Интегрированная среда разработки. в отличие от обычного редактора кода, Visual Studio Code также является интегрированной средой разработки и предоставляет множество инструментов для создания, отладки, тестирования и развертывания программного обеспечения.

3. Поддержка множества языков программирования. Visual Studio Code поддерживает множество языков программирования, включая JavaScript, TypeScript, Python, Java, Go и многие другие.

4. Интеллектуальная подсказка кода. Visual Studio Code предоставляет функцию автодополнения, которая помогает разработчикам написать код быстрее и с меньшими ошибками.

5. Расширяемость. Visual Studio Code имеет множество плагинов и расширений, которые позволяют разработчикам настроить редактор под свои потребности и повысить производительность.

6. Интеграция с Git. Visual Studio Code имеет встроенную поддержку системы контроля версий Git, что облегчает работу в команде и управление исходным кодом.

7. Мощный отладчик. Visual Studio Code предоставляет мощный отладчик, который помогает быстро находить и исправлять ошибки в коде.

8. Множество полезных функций. Visual Studio Code имеет множество полезных функций, таких как возможность сравнения файлов, поиск и замена, поддержка сниппетов кода и многие другие.

В целом, Microsoft Visual Studio Code является отличным выбором для разработчиков, которые ищут легковесный, кроссплатформенный редактор кода с множеством полезных функций и инструментов для разработки программного обеспечения [17].

## **6.3 Реализация программы для анализа потокобезопасных структур данных**

Для итоговой реализации программы для анализа потокобезопасных структур данных было принято решение использовать lock-free queue или же потокобезопасные (не блокирующиеся очереди), так как они содержат внутри себя как минимум 2 граничных условия, что позволяет получить более валидные результаты анализа, а также имеют больше сценариев тестирования, чем, например, lock-free stack (не блокирующийся стек)

### **6.3.1 Компонент для анализа данных**

Существует большое количество видов и реализаций очередей. В данной работе мы будем использовать стандартную не приоритетную очередь, а также несколько вариантов lock free очередей.

Для стандартной очереди возможно три наиболее частых варианта реализации, на базе массива, на базе связного списка, на базе двух стеков.

1. На базе массива.

Преимущества: требуется малое количество памяти; высокая скорость работы; проще в разработке.

Недостатки: максимальное количество элементов в очереди ограничено размером массива, что может потребовать перевыделеления.

2. На базе связного списка.

Преимущества: размер ограничен лишь объемом памяти; эффективная масштабируемость.

Недостатки: сложнее в разработке; требуется больше памяти; работает медленнее реализации на массиве.

3. На базе двух стеков.

Преимущества: аналогично реализации на массиве, либо на связном списке; может обладать спец свойствами, например быстрым определением минимум

Недостатки: аналогично реализации на массиве, либо на связном списке; сложнее в разработке.

Пример кода на С++ для реализации стандартной очереди на базе связного списка:

struct Node {

Node \* m\_pNext;};

class queue {

Node \* m\_pHead;

Node \* m\_pTail;

public:

queue(): m\_pHead( NULL ), m\_pTail( NULL ) {}

void enqueue( Node \* p ){

p->m\_pNext = nullptr;

if ( m\_pTail )

m\_pTail->m\_pNext = p;

else

m\_pHead = p;

m\_pTail = p;}

Node \* dequeue(){

if ( !m\_pHead ) return nullptr;

Node \* p = m\_pHead;

m\_pHead = p->m\_pNext;

if ( !m\_pHead )

m\_pTail = nullptr;

return p;}};

Очереди, поддерживающие потокобезопасность, имеют несколько вариантов в зависимости от количества производителей и потребителей - single/multi producer — single/multi consumer. Они могут быть ограниченными, используя предопределенный буфер, или неограниченными на основе списка. Также существуют различные реализации очередей, lock-free, wait-free или lock-based, с соблюдением строгой или нестрогой FIFO. Обычно, чем больше специализированы требования к очереди, тем более эффективным оказывается ее алгоритм.

В контексте решаемых задач будем использовать алгоритм CAS и HP и их модификации.

CAS или же Сompare And Swap это атомарная инструкция, используемая в многопоточности для достижения синхронизации. Она сравнивает содержимое ячейки памяти с заданным значением и, только если они совпадают, изменяет содержимое этой ячейки памяти на новое заданное значение. Это выполняется как одна атомарная операция. Атомарность гарантирует, что новое значение вычисляется на основе актуальной информации; если бы значение было обновлено другим потоком за это время, запись завершилась бы ошибкой.

HP или же Hazard Pointer является наиболее популярной и детально проработанной схемой отложенного удаления. Она использует только простые операции чтения и записи данных, не прибегая к сложным механизмам синхронизации, например, CAS.

Ключевым элементом данного механизма является необходимость пометить указатель на элемент контейнера как опасный (hazard) внутри операции структуры данных, которая не использует блокировки. То есть перед тем, как работать с указателем на элемент, следует поместить его в массив hazard-указателей (HP) текущего потока. Массив HP - приватный для потока, владелец которого пишет в него, а остальные потоки могут читать. Преобладающее большинство контейнеров, использующих данную схему, имеет фиксированный размер массив hazard-указателей одного потока, не превышающий 3-4, что показывает, что расходы на поддержку данной схемы минимальны.

Классическим алгоритмом потокобезопасной очереди является алгоритм Майкла-Скотта. В последствии будем называть его MSQueue.

Пример кода на С++ для реализации метода enqueue MSQueue на базе связного списка:

bool enqueue( value\_type& val ) {

node\_type \* pNew = node\_traits::to\_node\_ptr( val );

typename gc::Guard guard; // Защитник, например, Hazard Pointer

back\_off bkoff;

node\_type \* t;

while ( true ) {

t = guard.protect( m\_pTail, node\_to\_value() );

node\_type \* pNext = t->m\_pNext.load(

memory\_model::memory\_order\_acquire);

if ( pNext != nullptr ) {

m\_pTail.compare\_exchange\_weak( t, pNext,

std::memory\_order\_release, std::memory\_order\_relaxed);

continue;

}

node\_type \* tmp = nullptr;

if ( t->m\_pNext.compare\_exchange\_strong( tmp, pNew,

std::memory\_order\_release,

std::memory\_order\_relaxed )) {

break;}

bkoff();}

++m\_ItemCounter;

m\_pTail.compare\_exchange\_strong( t, pNew,

std::memory\_order\_acq\_rel, std::memory\_order\_relaxed );

return true; }

Пример кода на С++ для реализации метода dequeue MSQueue на базе связного списка:

value\_type \* dequeue() {

node\_type \* pNext;

back\_off bkoff;

typename gc::template GuardArray<2> guards;

node\_type \* h;

while ( true ) {

h = guards.protect( 0, m\_pHead, node\_to\_value() );

pNext = guards.protect( 1, h->m\_pNext, node\_to\_value() );

if ( m\_pHead.load(std::memory\_order\_relaxed) != h ) {

continue;}

if ( pNext == nullptr )

return nullptr; // очередь пуста

node\_type \* t = m\_pTail.load(std::memory\_order\_acquire);

if ( h == t ) {

m\_pTail.compare\_exchange\_strong( t, pNext,

std::memory\_order\_release, std::memory\_order\_relaxed);

continue;}

if ( m\_pHead.compare\_exchange\_strong( h, pNext,

std::memory\_order\_release, std::memory\_order\_relaxed ))

{break;}

bkoff(); }

--m\_ItemCounter;

dispose\_node( h );

return pNext;}

Очередь реализована на базе односвязного списка от головы к хвосту.

Важным достоинством данного алгоритма является управление двумя указателям на хвост и голову без использования двойного CAS. Такого свойства получилось добиться за счет того, что очередь никогда не пуста. Это обеспечивается при помощи физической не пустоты. При инициализации очереди, в неё добавляется ложный элемент, выступающий в роли хвоста и головы. Логически же данная реализация ничем не отличается от классической. Работает это так: при извлечении возвращается элемент, который становится новым ложным элементом (новой головой), а прежний ложный элемент (предыдущая голова) удаляется.

Также алгоритм явно допускает, что хвост может указывать не на последний элемент. При всяком обращение к хвосту мы проверяем у него наличии следующего элемента. Если элемент присутствует, значит указатель указывает не на хвост, и его надо сдвинуть. Возможна опасная ситуация, когда хвост будет указывать на элемент перед головой, то есть на пересечение головы и хвоста. Чтобы предотвратить это, в методе dequeue проверяем наличие следующего элемента у хвоста: мы прочли голову, проверяем наличие следующего элемента у головы, а затем видим, что голова равна хвосту. Значит, за хвостом что-то есть, и хвост надо продвинуть вперед. Схематическое изображение данных действий отображено на рисунке 6.1.

Другим вариантом потокобезопасной очереди является Baskets queue. Наиболее часто такой алгоритм применяется в задачах высокой конкуренции. Внутри себя он является модификацией MSQueue. В данном алгоритме применяются наборы логических корзин. Каждая из этих корзин привязана к определенному короткому интервалу времени, в рамках которого доступно добавление нового элемента. Интервал прошел — создается новая корзина. Общий вид Baskets queue представлен на рисунке 6.2.

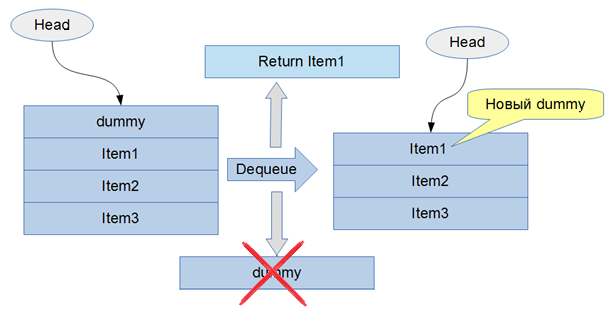


Рисунок 6.1 – Схема

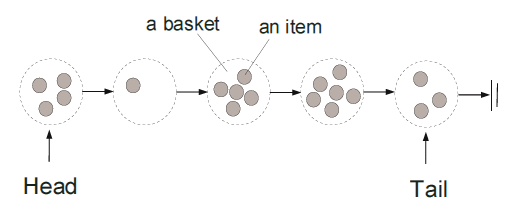


Рисунок 6.2 – Baskets queue

Каждая корзина является черным ящиком для обычного разработчика, так как содержит неупорядоченный набор элементов. Казалось бы, при такой логике перестает действовать правило FIFO, что приведет к тому, что очередь становится нечестной(unfair). FIFO будет выполняться только для корзин, но не для их содержимого. Получается, что чтобы пренебречь неупорядоченностью элементов в корзине, интервал доступности корзины для добавления должен быть достаточно мал.

Логически складывается впечатление, что этот интервал надо динамически определять, однако это не так. Рассмотрим очередь MSQueue. В случае высокой конкуренции при извлечении из очереди, когда механизм изменения хвоста с помощью CAS не сработал и back-off был вызван, мы не можем предсказать, в каком порядке элементы будут добавлены в очередь, так как они добавляются одновременно. Таким образом, логическая корзина является разновидностью стратегии back-off.

Данный алгоритм имеет достаточно объемный код, поэтому будет содержаться в приложении А. Более же наглядное описание работы Baskets queue будет содержаться ниже на рисунке 6.3.

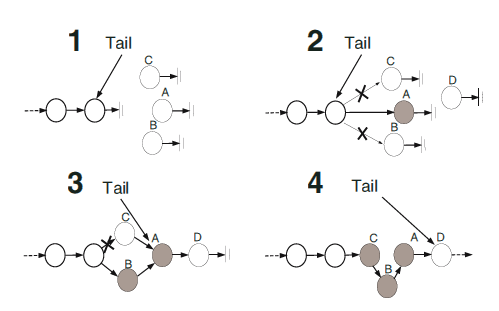


Рисунок 6.3 – Добавление в Baskets queue

1. Потоки A, B и C хотят добавить элементы в очередь. Они замечают, что хвост находится на своем месте (как известно, в MSQueue хвост может указывать на любой элемент, а не на последний). Таким образом, они пытаются изменить его одновременно.
2. Поток A побеждает и успешно добавляет элемент. В то время как потоки B и C оказываются неудачниками - их CAS с хвостом проваливается. В результате они начинают добавлять свои элементы в корзину, используя старое значение хвоста, которое было прочитано ранее.
3. Поток B быстрее всех и успешно добавляет свой элемент. А в это же время поток D также вызывает enqueue и добавляет свой элемент, изменяя хвост.
4. Поток C тоже завершает добавление, но в середину очереди. Он использует старый указатель на хвост, который он прочитал при первоначальном входе в операцию, перед неудачной попыткой выполнения CAS.

Следует отметить, что добавление элемента может привести к тому, что он будет помещен перед головой очереди, возможно, как показано на рисунке 6.3.1.3.4, где элемент добавляется перед C во время вставки C и может быть удален другим потоком. Чтобы избежать этой ситуации, используется логическое удаление с пометкой удаленных элементов флагом deleted. Чтобы флаг и сам указатель на элемент могли быть прочитаны атомарно будем использовать младший бит указателя на следующий элемент pNext. Ресурсы современных систем позволяют это сделать память, так как память выделяется выравненной как минимум по 4 байтам, следовательно младшие 2 бита указателя всегда будут нулями. Такое решение называется marked pointer. Применяя логическое удаление с помощью CAS и выставляя младший бит pNext в значение 1, мы исключаем возможность вставки элемента перед головой. Вставка также осуществляется с помощью CAS, и удаленный элемент содержит 1 в младшем бите указателя, что делает операцию CAS неудачной. При вставке мы берем только старшие биты marked pointer, содержащие адрес, а младший бит полагаем равным 0.

Ещё одним вариантом модификации MSQueue является применение оптимистического подхода. В рамках алгоритма Майкла-Скотта был обнаружено, что операция добавления работает за один CAS, однако операция извлечения требует два CAS. Наличие двух CAS операций может крайне сильно понижать производительность даже при малой нагрузке, так как CAS довольно тяжелая операция даже для современных процессоров. На рисунке 6.4 показано как это выглядит до оптимизации.

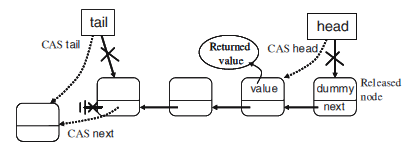


Рисунок 6.4 – Двойной CAS

Возникает вопрос, что выполняют эти два CAS и как их оптимизировать. Один из CAS нужен для связывания. Он соединяет новый элемент с хвостом посредством изменения pTail->pNext. Второй CAS перемещает сам pTail. Складывается впечатление, что от второго CAS можно избавиться за счет применение атомарной записи. Чтобы это сделать, нужно чтобы наш односвязный список был направлен в противоположную сторону, то есть от хвоста к голове. Вариант изменения направления постоянно будет работать слишком медленно, а сохранение направления от хвоста к голове приведет к невозможности применения операция извлечения, так как связи pHead->pNext не будет. Значит, наиболее эффективным решением данной проблемы будет являться применение двусвязного списка. Способ решения представлен на рисунке 6.5.

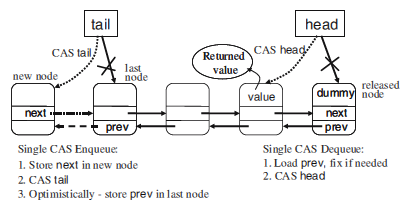


Рисунок 6.5 – Двойной CAS

Общая структура данных ясна, это двусвязный список, однако идеального алгоритма для lock-free двусвязного списка не было разработано. Текущие реализации, такие как DCAS и MCAS являются теоретическими. Первая требует аппаратной поддержки, которой пока нигде нет. Вторая является эмуляцией и требует больше ресурсов.

Nir Shavit и Edya Ladan Mozes предложили самовосстанавливающееся решение. В рамках их алгоритма прямая связь, то есть прямое направление от головы к хвосту, будет всегда существовать и являться актуальной и корректной. Обратная же связь, то есть направление от хвоста к голове, будет являться динамической и может иметь нарушения в порядке. Однако, при обнаружении таких нарушений в порядке, мы всегда сможем их устранить. При обнаружении нарушения, алгоритм просто пройдет по next ссылкам, и на их основании восстановит порядок prev ссылок. Фактором нарушения является невыполнение неравенства pHead->prev->next != pHead. Для реализации данного исправления был реализован метод fix\_list, код которого представлен ниже.

void fix\_list(node\_type \* pTail, node\_type \* pHead){

node\_type\* pCurNode;

node\_type\* pCurNodeNext;

typename gc::template GuardArray<2> guards;

pCurNode = pTail;

while (pCurNode != pHead){

pCurNodeNext = guards.protect(0, pCurNode->m\_pNext, node\_to\_value());

if ( pHead != m\_pHead.load(std::memory\_order\_relaxed) )

break;

pCurNodeNext->m\_pPrev.store(pCurNode, std::memory\_order\_release);

guards.assign(1, node\_traits::to\_value\_ptr( pCurNode = pCurNodeNext));

}

}

fix\_list обходит всю очередь по гарантировано корректным ссылкам pNext и исправляет pPrev.

Нарушение порядка pPrev возникает из-за задержек, а не из-за сильной нагрузки. Наиболее частой причиной задержки является вытеснение потока операционной системой или прерывание.

В итоге имеем то, что мы строим pPrev список со знанием возможного отсутствия корректности и валидности, однако при нахождении рассинхронизации порядка для pNext и pPrev вызываем метод fix\_list. Так же, в лучшем случае операции извлечения и добавления используют только по одному CAS. В худшем случае аналогично используется только по одному CAS, но с какой-то периодичностью вызывается метод fix\_list.

Конечно, бывают и другие подходы к потокобезопасным очередям, однако в данной работе они не рассматриваются, по причине их низкой эффективности из-за высокой строгости и большого количества накладных операций. Так же в большинстве своем эти алгоритмы являются достаточно сложными в реализации и последующей поддержке разработчиком [18].

Для тестирования различных реализаций было принято решение реализовать два тестовых сценария: endurance test и producer-consumer test.

Тестовая машина — ноутбук на Windows, AMD Ryzen 7 5800H, 8 ядер на процессор + hyperthreading, итого 16 логических процессоров. Для проведения тестов, все внешние программы и службы были закрыты.

Компилятор — GCC 4.8.2, без оптимизации.

Для сравнения эффективности, использовались стандартные реализации std::queue<T, std::deque<T>> и std::queue<T, std::list<T>> с синхронизацией мьютексом. Тип T – структура из двух целых. Все реализованные lock-free queue используют Hazard Pointer.

### **6.3.2 Компонент для визуализации результатов**

Данный компонент был реализован на языке Python. Внутри себя он содержит 2 метода read\_data и plot отвечающие за правильное извлечение данных и отрисовку графиков соответственно. Исходные коды данных методов приведены ниже.

def read\_data(file\_name):

    with open(file\_name, 'r') as f\_in:

        d = deque(

            line.rstrip()

            for line in f\_in.readlines()

        )

    threads = list(map(float, d.popleft().split()))

    line\_labels = []

    data\_list = []

    while d:

        line\_labels.append(d.popleft())

        data\_list.append(list(map(float, d.popleft().split())))

    return threads, data\_list, line\_labels

def plot(threads, data\_list, colors, line\_labels, title, x\_label, y\_label):

for i in range(len(data\_list)):

plt.plot(threads, data\_list[i], colors[i], label=line\_labels[i])

plt.title(title, fontsize=15)

plt.xlabel(x\_label)

plt.ylabel(y\_label)

plt.grid()

plt.legend()

plt.savefig(f"{title}.png")

plt.show()

Сначала проведем endurance test, чтобы понять, как себя ведет очередь при большом наполнении данными. Тест будет разделен на 2 части. Сначала выполним параллельно 20 млн операций добавления и 5 млн операций извлечения. По окончанию теста у нас будет очередь на 15 млн элементов. Выполним извлечение оставшихся 15 млн элементом, чтобы опустошить очередь полностью. Тестирование проводилось для 2, 4, 6..16 потоков. Для тестирования были выбраны две зависимости MOp/s от Threads и Secs от Threads. Результаты тестирования представлены на рисунках 6.6, 6.7.

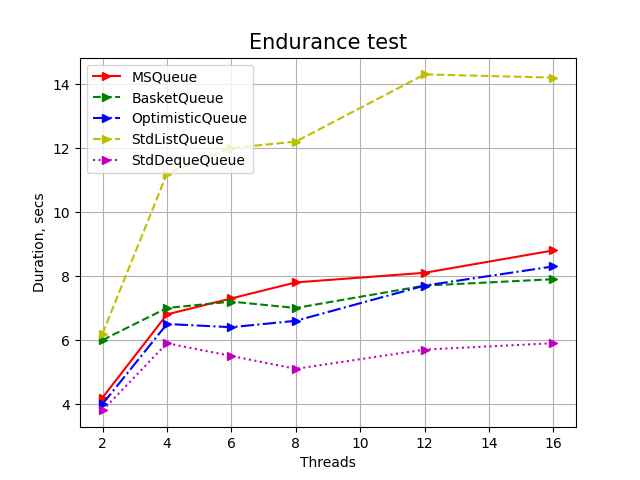


Рисунок 6.6 – Endurance test, secs

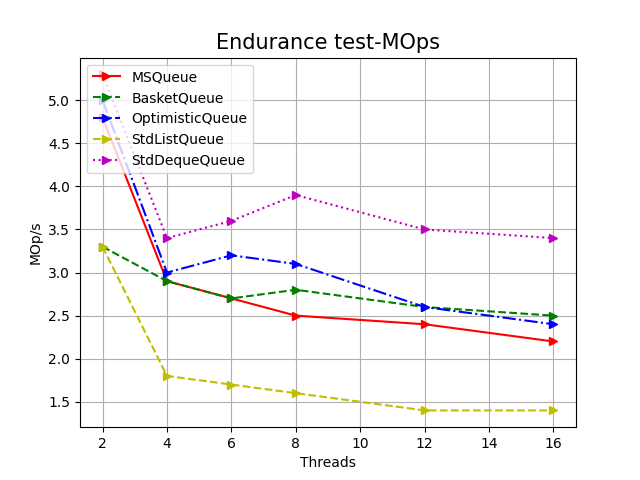


Рисунок 6.7 – Endurance test, MOp/s

Необычно, но наиболее быстрой реализацией очереди стала блокируемая std::queue<T, std::deque<T>>. С высоким шансом это связано с логикой работы std::deque. Он выделяет память не под каждый элемент, а создает блоки элементов, что позволяет вызывать меньше операций аллокации. Так же, вполне возможно, что проблема в самом аллокаторе, так как чаще всего аллокаторы работают на мьютексах, чего мы пытались избежать.

Однако, так же, если отбросить влияние аллокаторов, и сравнить только lock-free queue реализации, то можно сделать вывод, что оптимизации Basket Queue и Optimistic Queue имеет более высокую эффективность, чем базовый MSQueue. Так же явно видно, что в контексте данного теста Basket Queue алгоритм работает эффективнее при количестве потоков от 4х и выше, что, предположительно, более востребовано из-за хорошей масштабируемости. С другой стороны, Optimistic Queue показывает более стабильные результаты, так как даже при 2 потоках работает уже достаточно быстро.

Проведем producer/consumer test. Нужно воссоздать более реалистичный сценарий. Чаще всего очередь применяется для хранения всплесков активности, для последующего последовательного извлечения. Следовательно, нормальным состоянием очереди в преобладающие моменты времени является её пустота. Пусть будет N писателей и N читателей очереди. Всего исполним 20 млн операций добавления, и, соответственно, 20 млн операций извлечения. Соответственно, аналогично предыдущему тесту будем использовать 2, 4, 6..16 потоков, где первая половина потоков используется для добавления, вторая половина для извлечения. Для тестирования были выбраны две зависимости MOp/s от Threads и Secs от Threads. Результаты тестирования представлены на рисунке 6.8, 6.9.

В данном случае явно видно превосходство lock-free queue алгоритма Opimistic Queue над всеми остальными. Следовательно, идеи, заложенные авторами алгоритма в него, в реальном сценарии действительно себя оправдывают. Важным моментом является то, что очередь на базе std::deque все ещё хорошо себя показывает, что говорит о том, что аллокация под каждый элемент является узким горлышком в контексте данных реализаций lock-free очередей. Последующие варианты развития и ускорения lock-free очередей скорей всего будут связаны с улучшением алгоритмов аллокации, либо изменения способа масштабирования запросов.

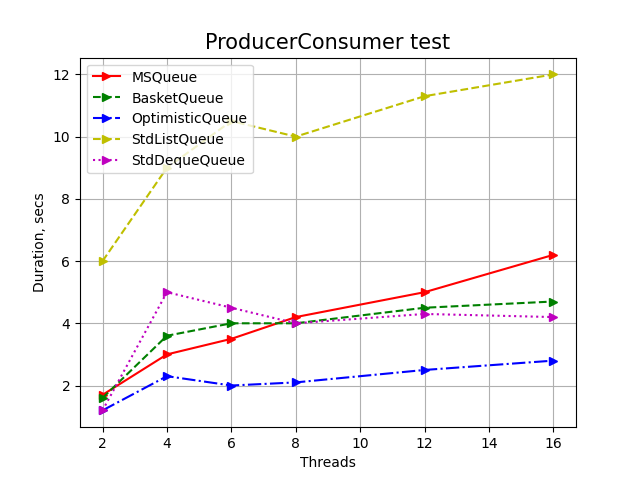


Рисунок 6.8 – Producer/consumer test, secs

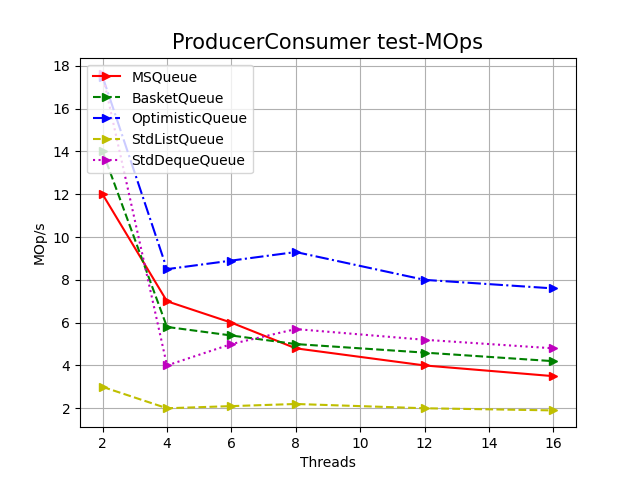


Рисунок 6.9 – Producer/consumer test, MOp/s

# **7 Оценка и защита результатов интеллектуальной деятельности**

Одной из главных задач государства, соблюдающего права, является защита законных прав и интересов в сфере интеллектуальной собственности. Для успешного вовлечения результатов научно-технической деятельности в хозяйственный оборот и формирования инновационных рынков необходима правовая охрана. Правовая охрана является основой защиты интересов авторов, инвесторов и производителей от недобросовестной конкуренции при распространении этих результатов.

Исследования, связанные с оценкой и охраной результатов интеллектуальной деятельности, имеют большое значение в связи с тем, что переход России к рыночной экономике и развитие предпринимательства привели к кардинальным изменениям в правовом регулировании, увеличению доли наукоемкой продукции и повышению потребности в ее оценке и правовой защите.

Этот раздел направлен на получение практических навыков оценки и защиты результатов интеллектуальной деятельности. Для достижения этой цели необходимо решить следующие задачи.

1. Выделить объект исследования дополнительного раздела на основании полученных результатов в основной части ВКР.
2. Произвести оценку рыночной стоимости объекта исследования (результата интеллектуальной деятельности) на основании предложенной методики.
3. Произвести исследование законодательной основы правовой защиты результатов интеллектуальной деятельности.

Методика проведения оценки и защиты результатов интеллектуальной деятельности осуществлялась в соответствии с методическими указаниями по оценке и защите результатов интеллектуальной деятельности выпускных квалификационных работ [19].

## **7.1 Описание результата интеллектуальной деятельности**

Целью выпускной квалификационной работы (ВКР) является анализ потокобезопасных структур данных с предложениями по оптимизации. Проведенная работа на языке программирования C++ может повысить эффективность работы параллельных вычислительных систем, а также повысить скорость разработки программного обеспечения в области параллельных вычислений на распределённых вычислительных машинах. Важными достоинствами потокобезопасных структур являются их универсальность, стабильность и высокая скорость работы, а также более высокая масштабируемость.

Выбор объекта исследования должен осуществляться на основании части четвёртой Гражданского кодекса Российской Федерации (ГК РФ) [20], согласно которому, результатами интеллектуальной деятельности и приравненными к ним средствами индивидуализации юридических лиц, товаров, работ, услуг и предприятий, которым предоставляется правовая охрана (интеллектуальной собственностью), являются следующие пункты.

1) произведения науки, литературы и искусства;

2) программы для электронных вычислительных машин (программы для ЭВМ);

3) базы данных;

4) топологии интегральных микросхем;

5) секреты производства (ноу-хау);

6) исполнения;

7) изобретения;

8) фонограммы;

9) сообщение в эфир или по кабелю радио- или телепередач (вещание организаций эфирного или кабельного вещания);

10) полезные модели;

11) промышленные образцы;

12) селекционные достижения;

13) фирменные наименования;

14) товарные знаки и знаки обслуживания;

15) наименования мест происхождения товаров;

16) коммерческие обозначения.

Из представленного списка объектов, подлежащих оценке результатов интеллектуальной деятельности, результаты, полученные в процессе выполнения выпускной квалификационной работы, относятся к объекту «программы для электронных вычислительных машин (программы для ЭВМ)».

## **7.2 Оценка рыночной стоимости результата интеллектуальной** **деятельности**

Для оценки рыночной стоимости результатов интеллектуальной деятельности, в основном, применяются доходный, сравнительный и затратный подходы.

Доходный подход к оценке следует использовать в случае наличия достоверной информации, позволяющей предсказать будущие доходы, которые может принести объект оценки, а также соответствующие расходы.

При наличии достаточной и достоверной информации о ценах и характеристиках объектов-аналогов, следует использовать сравнительный подход к оценке. При этом могут быть использованы как цены заключенных сделок, так и цены предложений.

Затратный подход к оценке объекта чаще всего используется, когда имеется достоверная информация о затратах на его приобретение, воспроизводство или замещение.

Область потокобезопасных структур данных и алгоритмов является достаточно перспективной, так как открывает возможности эффективной реализации и применения большего количества вычислительных ресурсов, чем вычислительные системы с общей памятью, а также уменьшает временные и финансовые потери компании разработчика. Тем не менее, исследований и удачных реализаций универсальных потокобезопасных структур крайне мало, что приводит к тому, что изучение оптимальности разных решений занимает крайне большое количество времени. Из-за этого коммерческий рынок программного обеспечения не наполнен большим количество продуктов, использующих потокобезопасные структуры данных, так как многие компании не готовы затрачивать ресурсы на разработку и анализ таких структур. Из-за недостатка данных спрогнозировать фактическую прибыль, которую данный анализ способен принести, невозможно. Следовательно, доходный и сравнительный подходы не могут быть использованы в данном случае. Однако имеющиеся сведения о расходах на разработку программного обеспечения (ПО) позволяют применить затратный подход.

Методика проведения затратного подхода осуществлялась согласно методическим указаниям по экономическому обоснованию выпускных квалификационных работ [21].

Ниже приведены основные категории расходов, связанных с разработкой программного обеспечения:

1) расходы на оплату труда;

2) отчисления на социальные нужды;

3) накладные расходы;

4) амортизация оборудования;

5) услуги сторонних организаций.

### **7.2.1 Расчет затрат на оплату труда**

Для расчёта расходов на оплату труда был сформирован план проводимых работ.

В связи с тем, что характер проводимых работ предполагает глубокие знания в области параллельных вычислений и компьютерной математики, а также свободное владение языками программирования низкого уровня, данные о средней заработной плате студента были собраны для C++-программиста уровня middle, а для научного руководителя – lead. По оценке «Хабр Карьера» медианная заработная плата таких специалистов в РФ во второй половине 2022 года составила 140 тыс. рублей для уровня middle и 250 тыс. рублей – для lead [22].

Таким образом, если принять, что в месяце двадцать один рабочий день, то дневная ставка студента и научного руководителя составят:

руб./день,

руб./день.

В таблице 7.1 приведён план работ студента.

Таблица 7.1 – План работ студента

|  |  |  |  |
| --- | --- | --- | --- |
| № | Этапы и содержание выполняемых работ | Трудоемкость, чел./день | Ставка, руб./день |
| 1 | Составление ТЗ | 1 | 7000 |
| 2 | Изучение технической литературы | 35 | 7000 |
| 3 | Изучение существующих работ | 12 | 7000 |
| 4 | Проведение анализа потокобезопасных структур данных | 60 | 7000 |
| 5 | Тестирование и отладка | 25 | 7000 |
| 6 | Согласование с руководителем и исправление ошибок | 7 | 7000 |
| 7 | Оформление дипломного проекта | 14 | 7000 |

В таблице 7.2 приведён план работ научного руководителя.

Таблица 7.2 – План работ научного руководителя

|  |  |  |  |
| --- | --- | --- | --- |
| № | Этапы и содержание выполняемых работ | Трудоемкость, чел./день | Ставка, руб./день |
| 1 | Составление ТЗ | 1 | 12500 |
| 2 | Изучение существующих работ | 12 | 12500 |
| 3 | Согласование с руководителем и исправление ошибок | 7 | 12500 |

Трудозатраты научного руководителя составят:

.

Трудозатраты студента составят:

.

Общая формула расчёта заработной платы:

,

где – время, затраченное 𝑖-ым исполнителем на проведение работ в днях;

– ставка 𝑖-го исполнителя в рублях в день.

Применительно к конкретно нашему случаю основная заработная плата рассчитывается по формуле:

Основная заработная плата составит:

руб.

Дополнительная заработная плата рассчитывается по формуле:

,

где – норматив дополнительной заработной платы.

Дополнительная заработная плата составит:

руб.

Формула расчёта отчислений на социальные нужды:

,

где – норматив отчислений на страховые взносы на обязательное социальное, пенсионное и медицинское страхование. Отчисления на социальные нужды от основной и дополнительной заработной платы равны:

руб.

### **7.2.2 Расчет накладных расходов**

В этой работе главные накладные расходы составляют 20% от общей суммы основной и дополнительной заработной платы. Накладные расходы рассчитываются по формуле:

,

где – норматив накладных расходов.

Накладные расходы равны:

руб.

### **7.2.3 Издержки на амортизацию оборудования**

В процессе разработке ПО использовался ноутбук. Стоимость ноутбука Lenovo Legion 7 16ACHg6 – 167 102 рублей [23].

Согласно постановлению правительства РФ от 01.01.2002 №1 (ред. От 28.04.2018) «О Классификации основных средств, включаемых в амортизационные группы» [24] персональные компьютеры и печатающие устройства к ним, относятся ко второй группе. Срок полезного использования оборудования равен от двух до трех лет включительно. Срок эксплуатации три года. Норма амортизации = 33,3%.

Годовая амортизация рассчитывается по формуле:

,

где − стоимость оборудования;

− норма амортизации.

Годовая амортизация составит:

руб.

Сумма амортизации за рабочий день рассчитывается по формуле:

.

В 2023 году 247 рабочих дней, соответственно сумма амортизации за один рабочий день составит:

28 руб.

Таким образом, амортизация оборудования за всё время на разработку ПО составит:

### **7.2.4 Расходы на услуги сторонних организаций**

В процессе разработки ПО был необходим доступ к интернету. Оператором связи выступил Skynet, интернет 400 Мбит/с – 680 руб. в месяц в рамках тарифа «Т-400», оформленного на 6 месяцев [25]. Таким образом, затраты на услуги сторонних организаций составили 4080 руб.

### **7.2.5 Себестоимость** **разработки программного обеспечения**

Себестоимость разработки ПО с учетом расходов и в целом указана в таблице 7.3.

Таблица 7.3 – Смета затрат на разработку ПО

|  |  |  |  |
| --- | --- | --- | --- |
| № | Наименование статьи | Сумма, руб. | Структура себестоимости, (%) |
| 1 | Расходы на оплату труда | 1438224,00 | 65,40 |
| 2 | Отчисления на социальные нужды | 434343,65 | 19,75 |
| 3 | Накладные расходы | 287644,80 | 13,08 |
| 4 | Издержки на амортизацию оборудования | 34693,62253 | 1,58 |
| 5 | Расходы на услуги сторонних организаций | 4080 | 0,19 |
| ИТОГО: | | 2198986,07 | 100,00 |

В итоге, себестоимость разработки программного обеспечения оценивается в 2198986,07 рублей.

## **7.3 Правовая защита результатов интеллектуальной деятельности**

Правовая охрана объектов интеллектуальной собственности обеспечивается на основе патентного законодательства и законодательства по защите недобросовестной конкуренции, авторского права, а также законодательства о средствах индивидуализации.

Нарушения прав интеллектуальной собственности наказывается в соответствии с гражданским, уголовным и административным законодательством.

Технические новшества могут быть защищены от использования другими лицами с помощью патентов на изобретения, патентов на промышленные образцы, свидетельств на полезные модели и свидетельств на ноу-хау.

Согласно пункту 1 статьи 1259 части 4 ГК РФ программы для ЭВМ относятся к объектам авторских прав, которые охраняются как литературные произведения. Следовательно, в данной работе результат интеллектуальной деятельности защищается авторским правом. Авторское право распространяется на все компьютерные программы и базы данных, независимо от того, находятся ли они в общественном достоянии и независимо от их материального носителя, цели или полезности.

Согласно пункту 2 статьи 1255 части 4 ГК РФ автору произведения принадлежат следующие права:

1) исключительное право на произведение;

2) право авторства;

3) право автора на имя;

4) право на неприкосновенность произведения;

5) право на обнародование произведения.

В процессе разработки ПО студента консультировал научный руководитель. В связи с этим согласно пункту 1 статьи 1258 части 4 ГК РФ, студент и научный руководитель признаются соавторами.

Согласно пункту 1 статьи 1281 части 4 ГК РФ исключительное право на произведение, созданное в соавторстве, действует в течение всей жизни автора, пережившего других соавторов, и 70 лет, считая с 1 января года, следующего за годом его смерти.

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения выпускной квалификационной работы был проведен анализ потокобезопасных структур данных. В частности, были реализованы две программы. Первая содержит внутри себя реализации различных алгоритмов lock-free очередей, а также тесты. Вторая используется для отображения и анализы графиков производительности.

Анализ полученных данных выявил недочеты существующих алгоритмов, в частности, связанные с блокирующей аллокацией.

В рамках данной работы был проведен обзор структур данных и параллелизма. Для последующего выявления слабых мест, были изучены методы общего анализа потокобезопасных структур данных и примеры потокобезопасных структур данных. На основе изученного, были проанализированые способы и сферы применения потокобезопасных структур данных.

В последствии был проведен сравнительный анализ производительности потокобезопасных и непотокобезопасных структур данных.

Для дальнейшего анализа были выбраны, реализованы и протестированы различные алгоритмы lock-free очередей. Также, на базе полученных данных, были построены графики производительности.

Это позволило более детально изучить узкие места подобных структур данных, провести анализ, и выделить некоторые общие закономерности.

Таким образом, реализованные структуры данных, тесты и проведенный анализ графиков, позволят разработчикам в будущем выбирать более быстрые алгоритмы для многопоточных приложений.

Благодаря проведенном анализу, компании и команды снизят затраты на разработку и реализацию параллельных приложений. Это произойдет за счет увеличения скорости разработки, так как не потребуется подробного изучения и анализа потокобезопасных структур данных.

В дальнейшем данная работа может быть доработана и улучшена, с целью коммерческого применения в информационных компаниях

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. 10 структур данных, которые должен знать каждый разработчик [Электронный ресурс] – URL: https://practicum.yandex.ru/blog/10-osnovnyh-struktur-dannyh/#id1 (дата обращения 06.02.2023 г)
2. Concurrent Data Structures [Электронный ресурс] – URL: https://www.cs.tau.ac.il/~afek/ConcurrentDS-MS04.pdf (дата обращения 13.02.2023г)
3. Concurrency in Operating System [Электронный ресурс] – URL: https://www.geeksforgeeks.org/concurrency-in-operating-system/ (дата обращения 20.02.2023г)
4. Концепция: Параллелизм [Электронный ресурс] – URL: https://dit.isuct.ru/Publish\_RUP/core.base\_rup/guidances/concepts/concurrency\_EE2E011A.html (дата обращения 20.02.2023г)
5. Когда параллелизм превосходит конкурентность [Электронный ресурс] – URL: https://nuancesprog.ru/p/10744/ (дата обращения 20.02.2023г)
6. Concurrency in Operating System [Электронный ресурс] – URL: https://www.tutorialspoint.com/concurrency-in-operating-system (дата обращения 20.02.2023г)
7. Статический анализ – от знакомства до интеграции [Электронный ресурс] – URL: https://habr.com/ru/companies/pvs-studio/articles/514124/ (дата обращения 27.02.2023г)
8. Зачем нужен динамический анализ кода, на примере проекта PVS-Studio [Электронный ресурс] – URL: https://habr.com/ru/companies/pvs-studio/articles/580196/ (дата обращения 06.03.2023г)
9. Введение в формальные методы верификации программ А. С. Камкин – М.: МАКС Пресс, 2018. – 272 с.
10. "The Art of Concurrency" by Clay Breshears [Электронный ресурс] – URL: http://www.nylxs.com/docs/ The%20Art%20of%20 Concurrency.pdf (дата обращения 13.03.2023 г)
11. "Concurrent Data Structures and Algorithms" by Maged M. Michael [Электронный ресурс] – URL: https://dspace.cvut.cz/bitstream/handle/ 10467/87820/F8-DP-2020-Kroilov-Viacheslav-thesis.pdf?sequence=-1&i sAllowed=y (дата обращения 20.03.2022г.)
12. Thread-safe collections [Электронный ресурс] – URL: https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/ (дата обращения 27.03.2022г.)
13. Data Structures for Parallel Programming [Электронный ресурс] – URL: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/data-structures-for-parallel-programming (дата обращения 03.04.2022г.)
14. Язык программирования C++ [Электронный ресурс] – URL: https://www.sravni.ru/kursy/info/yazyk-programmirovaniya-c/ (дата обращения 10.04.2022г.)
15. Почему Python – это отличный выбор для новичков? [Электронный ресурс] – URL: https://learn.microsoft.com/ru-RU/visualstudio/get-started/visual-studio-ide?view=vs-2022 (дата обращения 17.04.2022г.)
16. Что такое Visual Studio? [Электронный ресурс] – URL: https://itproger.com/news/pochemu-python-eto-otlichniy-vibor-dlya-novichkov (дата обращения 24.04.2022г.)
17. Why did we build Visual Studio Code? [Электронный ресурс] – URL: https://code.visualstudio.com/Docs/editor/whyvscode(дата обращения 24.04.2022г.)
18. Lock-free структуры данных. Очередной трактат [Электронный ресурс] – URL: https://habr.com/ru/articles/219201/ (дата обращения 24.04.2022г.)
19. Магомедов М. Н., Чигирь М. В. Оценка и защита результатов интеллектуальной деятельности: учебно-методическое пособие по выполнению дополнительного раздела выпускных квалификационных работ магистров – СПб.: Изд-во СПбГЭТУ “ЛЭТИ”, 2019 –26 с.
20. Гражданский кодекс Российской Федерации часть 4 [Электронный ресурс] – URL: http://www.consultant.ru/document/ Cons\_doc\_LAW\_64629 (дата обращения 01.05.2022г.)
21. Алексеева О. Г. Методические указания по экономическому обоснованию выпускных квалификационных работ бакалавров: Метод. указания – СПб.: Изд-во СПбГЭТУ “ЛЭТИ”, 2013. – 12 с.
22. Хабр Карьера [Зарплаты разработчиков во второй половине 2022: языки и квалификации] [Электронный ресурс] – URL: https://habr.com/ru/companies/habr\_career/articles/719730/ (дата обращения 01.05.2022г.)
23. Стоимость ноутбука Lenovo Legion 7 16ACHg6 [Электронный ресурс] – URL: https://www.kns.ru/product/noutbuk-lenovo-legion-7-16achg6-82n6000grk/ (дата обращения 01.05.2022г.)
24. О классификации основных средств, включаемых в амортизационные группы: постановление Правительства Рос. Федерации от 01.01.2002 №1 (ред. от 28.04.2018) // Собрание законодательства Российской Федерации [Электронный ресурс] – URL: https://www.consultant.ru/document/cons\_doc\_LAW\_34710/ (дата обращения 01.05.2022г.)
25. Тариф «T-400» [Электронный ресурс] – URL: https://skynet.ru/internet/tarify/tarif-internet-t-400/ (дата обращения 01.05.2022г.)

# **ПРИЛОЖЕНИЕ А**

**Отрисовка графиков**

import matplotlib.pyplot as plt

from collections import deque

colors = ['r->', 'g-->', 'b-.>', 'y-->', 'm:>']

files\_names = ["ProducerConsumer test", "Endurance test", "ProducerConsumer test-MOps", "Endurance test-MOps"]

def plot(threads, data\_list, colors, line\_labels, title, x\_label, y\_label):

for i in range(len(data\_list)):

plt.plot(threads, data\_list[i], colors[i], label=line\_labels[i])

plt.title(title, fontsize=15)

plt.xlabel(x\_label)

plt.ylabel(y\_label)

plt.grid()

plt.legend(loc='upper left')

plt.savefig(f"{title}.png")

plt.show()

def read\_data(file\_name):

with open(file\_name, 'r') as f\_in:

d = deque(

line.rstrip()

for line in f\_in.readlines()

)

x\_label, y\_label = d.popleft().split()

threads = list(map(float, d.popleft().split()))

line\_labels = []

data\_list = []

while d:

line\_labels.append(d.popleft())

data\_list.append(list(map(float, d.popleft().split())))

return (x\_label, y\_lab–el), threads, data\_list, line\_labels

if \_\_name\_\_ == "\_\_main\_\_":

for file\_name in files\_names:

labels, threads, data\_list, line\_labels = read\_data(file\_name)

plot(threads, data\_list, colors, line\_labels, file\_name, labels[0], labels[1])

# **ПРИЛОЖЕНИЕ Б**

**Потокобезопасные очереди**

template <typename GC, typename T, typename Traits = cds::container::msqueue::traits>

class MSQueue:

{

//@cond

typedef details::make\_msqueue< GC, T, Traits > maker;

typedef typename maker::type base\_class;

//@endcond

public:

/// Rebind template arguments

template <typename GC2, typename T2, typename Traits2>

struct rebind {

typedef MSQueue< GC2, T2, Traits2> other ; ///< Rebinding result

};

public:

typedef T value\_type; ///< Value type stored in the queue

typedef Traits traits; ///< Queue traits

typedef typename base\_class::gc gc; ///< Garbage collector used

typedef typename base\_class::back\_off back\_off; ///< Back-off strategy used

typedef typename maker::allocator\_type allocator\_type; ///< Allocator type used for allocate/deallocate the nodes

typedef typename base\_class::item\_counter item\_counter; ///< Item counting policy used

typedef typename base\_class::stat stat; ///< Internal statistics policy used

typedef typename base\_class::memory\_model memory\_model; ///< Memory ordering. See cds::opt::memory\_model option

static constexpr const size\_t c\_nHazardPtrCount = base\_class::c\_nHazardPtrCount; ///< Count of hazard pointer required for the algorithm

protected:

//@cond

typedef typename maker::node\_type node\_type; ///< queue node type (derived from \p intrusive::msqueue::node)

typedef typename maker::cxx\_allocator cxx\_allocator;

typedef typename maker::node\_deallocator node\_deallocator; // deallocate node

typedef typename base\_class::node\_traits node\_traits;

//@endcond

protected:

///@cond

static node\_type \* alloc\_node()

{

return cxx\_allocator().New();

}

static node\_type \* alloc\_node( value\_type const& val )

{

return cxx\_allocator().New( val );

}

template <typename... Args>

static node\_type \* alloc\_node\_move( Args&&... args )

{

return cxx\_allocator().MoveNew( std::forward<Args>( args )... );

}

static void free\_node( node\_type \* p )

{

node\_deallocator()( p );

}

struct node\_disposer {

void operator()( node\_type \* pNode )

{

free\_node( pNode );

}

};

typedef std::unique\_ptr< node\_type, node\_disposer > scoped\_node\_ptr;

//@endcond

public:

/// Initializes empty queue

MSQueue()

{}

/// Destructor clears the queue

~MSQueue()

{}

/// Enqueues \p val value into the queue.

/\*\*

The function makes queue node in dynamic memory calling copy constructor for \p val

and then it calls \p intrusive::MSQueue::enqueue.

Returns \p true if success, \p false otherwise.

\*/

bool enqueue( value\_type const& val )

{

scoped\_node\_ptr p( alloc\_node(val));

if ( base\_class::enqueue( \*p )) {

p.release();

return true;

}

return false;

}

/// Enqueues \p val in the queue, move semantics

bool enqueue( value\_type&& val )

{

scoped\_node\_ptr p( alloc\_node\_move( std::move( val )));

if ( base\_class::enqueue( \*p )) {

p.release();

return true;

}

return false;

}

/// Enqueues data to the queue using a functor

/\*\*

\p Func is a functor called to create node.

The functor \p f takes one argument - a reference to a new node of type \ref value\_type :

\code

cds::container::MSQueue< cds::gc::HP, Foo > myQueue;

Bar bar;

myQueue.enqueue\_with( [&bar]( Foo& dest ) { dest = bar; } );

\endcode

\*/

template <typename Func>

bool enqueue\_with( Func f )

{

scoped\_node\_ptr p( alloc\_node());

f( p->m\_value );

if ( base\_class::enqueue( \*p )) {

p.release();

return true;

}

return false;

}

/// Enqueues data of type \ref value\_type constructed from <tt>std::forward<Args>(args)...</tt>

template <typename... Args>

bool emplace( Args&&... args )

{

scoped\_node\_ptr p( alloc\_node\_move( std::forward<Args>( args )... ));

if ( base\_class::enqueue( \*p )) {

p.release();

return true;

}

return false;

}

/// Synonym for \p enqueue() function

bool push( value\_type const& val )

{

return enqueue( val );

}

/// Synonym for \p enqueue() function

bool push( value\_type&& val )

{

return enqueue( std::move( val ));

}

/// Synonym for \p enqueue\_with() function

template <typename Func>

bool push\_with( Func f )

{

return enqueue\_with( f );

}

/// Dequeues a value from the queue

/\*\*

If queue is not empty, the function returns \p true, \p dest contains copy of

dequeued value. The assignment operator for type \ref value\_type is invoked.

If queue is empty, the function returns \p false, \p dest is unchanged.

\*/

bool dequeue( value\_type& dest )

{

return dequeue\_with( [&dest]( value\_type& src ) {

// TSan finds a race between this read of \p src and node\_type constructor

// I think, it is wrong

CDS\_TSAN\_ANNOTATE\_IGNORE\_READS\_BEGIN;

dest = std::move( src );

CDS\_TSAN\_ANNOTATE\_IGNORE\_READS\_END;

});

}

/// Dequeues a value using a functor

/\*\*

\p Func is a functor called to copy dequeued value.

The functor takes one argument - a reference to removed node:

\code

cds:container::MSQueue< cds::gc::HP, Foo > myQueue;

Bar bar;

myQueue.dequeue\_with( [&bar]( Foo& src ) { bar = std::move( src );});

\endcode

The functor is called only if the queue is not empty.

\*/

template <typename Func>

bool dequeue\_with( Func f )

{

typename base\_class::dequeue\_result res;

if ( base\_class::do\_dequeue( res )) {

f( node\_traits::to\_value\_ptr( \*res.pNext )->m\_value );

base\_class::dispose\_result( res );

return true;

}

return false;

}

/// Synonym for \p dequeue() function

bool pop( value\_type& dest )

{

return dequeue( dest );

}

/// Synonym for \p dequeue\_with() function

template <typename Func>

bool pop\_with( Func f )

{

return dequeue\_with( f );

}

/// Clear the queue

/\*\*

The function repeatedly calls \ref dequeue until it returns \p nullptr.

\*/

void clear()

{

base\_class::clear();

}

/// Checks if the queue is empty

bool empty() const

{

return base\_class::empty();

}

/// Returns queue's item count (see \ref intrusive::MSQueue::size for explanation)

/\*\* \copydetails cds::intrusive::MSQueue::size()

\*/

size\_t size() const

{

return base\_class::size();

}

/// Returns reference to internal statistics

const stat& statistics() const

{

return base\_class::statistics();

}

};

}}

template <typename GC, typename T, typename Traits = optimistic\_queue::traits >

class OptimisticQueue:

#ifdef CDS\_DOXYGEN\_INVOKED

private intrusive::OptimisticQueue< GC, cds::intrusive::optimistic\_queue::node< T >, Traits >

#else

private details::make\_optimistic\_queue< GC, T, Traits >::type

#endif

{

//@cond

typedef details::make\_optimistic\_queue< GC, T, Traits > maker;

typedef typename maker::type base\_class;

//@endcond

public:

/// Rebind template arguments

template <typename GC2, typename T2, typename Traits2>

struct rebind {

typedef OptimisticQueue< GC2, T2, Traits2 > other ; ///< Rebinding result

};

public:

typedef GC gc; ///< Garbage collector

typedef T value\_type; ///< Value type to be stored in the queue

typedef Traits traits; ///< Queue traits

typedef typename base\_class::back\_off back\_off; ///< Back-off strategy used

typedef typename maker::allocator\_type allocator\_type; ///< Allocator type used for allocate/deallocate the nodes

typedef typename base\_class::item\_counter item\_counter; ///< Item counting policy used

typedef typename base\_class::stat stat; ///< Internal statistics policy used

typedef typename base\_class::memory\_model memory\_model; ///< Memory ordering. See \p cds::opt::memory\_model option

static constexpr const size\_t c\_nHazardPtrCount = base\_class::c\_nHazardPtrCount; ///< Count of hazard pointer required for the algorithm

protected:

//@cond

typedef typename maker::node\_type node\_type; ///< queue node type (derived from intrusive::optimistic\_queue::node)

typedef typename maker::cxx\_allocator cxx\_allocator;

typedef typename maker::node\_deallocator node\_deallocator; // deallocate node

typedef typename base\_class::node\_traits node\_traits;

//@endcond

protected:

///@cond

static node\_type \* alloc\_node()

{

return cxx\_allocator().New();

}

static node\_type \* alloc\_node( const value\_type& val )

{

return cxx\_allocator().New( val );

}

template <typename... Args>

static node\_type \* alloc\_node\_move( Args&&... args )

{

return cxx\_allocator().MoveNew( std::forward<Args>( args )... );

}

static void free\_node( node\_type \* p )

{

node\_deallocator()( p );

}

struct node\_disposer {

void operator()( node\_type \* pNode )

{

free\_node( pNode );

}

};

typedef std::unique\_ptr< node\_type, node\_disposer > scoped\_node\_ptr;

//@endcond

public:

/// Initializes empty queue

OptimisticQueue()

{}

/// Destructor clears the queue

~OptimisticQueue()

{}

/// Enqueues \p val value into the queue.

/\*\*

The function makes queue node in dynamic memory calling copy constructor for \p val

and then it calls \p intrusive::OptimisticQueue::enqueue.

Returns \p true if success, \p false otherwise.

\*/

bool enqueue( const value\_type& val )

{

scoped\_node\_ptr p( alloc\_node(val));

if ( base\_class::enqueue( \*p )) {

p.release();

return true;

}

return false;

}

/// Enqueues \p val value into the queue, move semntics

bool enqueue( value\_type&& val )

{

scoped\_node\_ptr p( alloc\_node\_move( std::move( val )));

if ( base\_class::enqueue( \*p )) {

p.release();

return true;

}

return false;

}

/// Enqueues \p data to queue using a functor

/\*\*

\p Func is a functor called to create node.

The functor \p f takes one argument - a reference to a new node of type \ref value\_type :

\code

cds::container::OptimisticQueue< cds::gc::HP, Foo > myQueue;

Bar bar;

myQueue.enqueue\_with( [&bar]( Foo& dest ) { dest = bar; } );

\endcode

\*/

template <typename Func>

bool enqueue\_with( Func f )

{

scoped\_node\_ptr p( alloc\_node());

f( p->m\_value );

if ( base\_class::enqueue( \*p )) {

p.release();

return true;

}

return false;

}

/// Enqueues data of type \ref value\_type constructed with <tt>std::forward<Args>(args)...</tt>

template <typename... Args>

bool emplace( Args&&... args )

{

scoped\_node\_ptr p( alloc\_node\_move( std::forward<Args>(args)... ));

if ( base\_class::enqueue( \*p )) {

p.release();

return true;

}

return false;

}

/// Synonym for \p enqueue( const value\_type& ) function

bool push( const value\_type& val )

{

return enqueue( val );

}

/// Synonym for \p enqueue( value\_type&& ) function

bool push( value\_type&& val )

{

return enqueue( std::move( val ));

}

/// Synonym for \p enqueue\_with() function

template <typename Func>

bool push\_with( Func f )

{

return enqueue\_with( f );

}

/// Dequeues a value from the queue

/\*\*

If queue is not empty, the function returns \p true, \p dest contains copy of

dequeued value. The assignment operator for type \p value\_type is invoked.

If queue is empty, the function returns \p false, \p dest is unchanged.

\*/

bool dequeue( value\_type& dest )

{

return dequeue\_with( [&dest]( value\_type& src ) { dest = std::move( src ); });

}

/// Dequeues a value using a functor

/\*\*

\p Func is a functor called to copy dequeued value.

The functor takes one argument - a reference to removed node:

\code

cds:container::OptimisticQueue< cds::gc::HP, Foo > myQueue;

Bar bar;

myQueue.dequeue\_with( [&bar]( Foo& src ) { bar = std::move( src );});

\endcode

The functor is called only if the queue is not empty.

\*/

template <typename Func>

bool dequeue\_with( Func f )

{

typename base\_class::dequeue\_result res;

if ( base\_class::do\_dequeue( res )) {

f( node\_traits::to\_value\_ptr( \*res.pNext )->m\_value );

base\_class::dispose\_result( res );

return true;

}

return false;

}

/// Synonym for \ref dequeue() function

bool pop( value\_type& dest )

{

return dequeue( dest );

}

/// Synonym for template version of \p dequeue\_with() function

template <typename Func>

bool pop\_with( Func f )

{

return dequeue\_with( f );

}

/// Checks if the queue is empty

bool empty() const

{

return base\_class::empty();

}

/// Clear the queue

/\*\*

The function repeatedly calls \ref dequeue until it returns \p nullptr.

\*/

void clear()

{

base\_class::clear();

}

/// Returns queue's item count

/\*\* \copydetails cds::intrusive::OptimisticQueue::size()

\*/

size\_t size() const

{

return base\_class::size();

}

/// Returns reference to internal statistics

const stat& statistics() const

{

return base\_class::statistics();

}

};

}} // namespace cds::container