Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РЕАЛИЗАЦИЯ ПРИМИТИВОВ СИНХРОНИЗАЦИИ ДЛЯ ЛЕГКИХ ПОТОКОВ В JAVA НА NUMA-АРХИТЕКТУРЕ

Автор: Коробейников Николай Андреевич _____

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Аксенов В.Е., PhD, науки _____

Санкт-Петербург, 2023 г.

Обучающийся Коробейников Николай Андреевич
Группа М34341 Факультет ИТиП

Направленность (профиль), специализация
Информатика и программирование

Консультанты:

а) Малахов А.А., специалист, ООО Техкомпания Хуавэй, ведущий инженер ключевых проектов _____

б) Чурбанов А.В., специалист, ООО Техкомпания Хуавэй, главный инженер проектов _____

ВКР принята «____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «____» _____ 20___ г.

Секретарь ГЭК Штумпф С.А. _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 6
1. Обзор предметной области 8
   1.1. Тяжелые и легкие потоки 8
   1.2. Java API для потоков 9
   1.3. Архитектуры памяти 10
   1.4. Примитивы синхронизации в Java 12
   1.5. Целевые процессоры 13
   1.6. Обзор решений и цели работы 14
   Выводы по главе 1 15
2. Реализация инструмента для измерения эффективности примитивов синхронизации 16
   2.1. Конфигурация инструмента 16
   2.2. Подготовка ОС к измерению 16
   2.3. Реализация инструмента 17
   2.4. Измерение на различных уровнях конкуренции 18
   Выводы по главе 2 19
3. Реализация примитивов синхронизации для легких потоков на NUMA архитектуре 20
   3.1. Реализации эффективной блокировки 20
   3.2. Реализация вспомогательных утилит для NUMA эффективной блокировки 23
   3.3. Кэширование номера NUMA узла для легкого потока 23
   3.4. Решение проблемы false sharing 25
   Выводы по главе 3 26
4. Измерение эффективности блокировок для легких потоков на NUMA системе 27
   4.1. Описание среды для запуска бенчмарков 27
   4.2. Тестирование при высокой конкуренции потоков 27
   4.3. Тестирование при низкой конкуренции потоков 28
   4.4. Тестирование на задаче текстовой статистики 29
   Выводы по главе 4 30

ЗАКЛЮЧЕНИЕ 32
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 33
ПРИЛОЖЕНИЕ А. Пример конфигурационного файла 36

ВВЕДЕНИЕ

С появлением многоядерных систем перед программистами была поставлена задача создания эффективных и масштабируемых структур данных. Основным решением для распараллеливания программ являются потоки, которые делятся на 2 типа: тяжелые (платформенные) потоки и легкие (виртуальные) потоки.

Использование потоков может привести к проблемам, которые не возникают в однопоточных реализациях. Одной из таких проблем является состояние гонки, которое может привести к недетерминированному результату, нарушению констистентности данных и даже к уязвимостям [6]. Состояние гонки опасно тем, что его трудно обнаружить и воспроизвести.

Чтобы избежать состояния гонки, программисты используют примитивы синхронизации, например блокировки. Код, находящийся между взятием блокировки и освобождением блокировки, называют критической секцией. Корректная блокировка не допускает нахождение двух потоков внутри критической секции одновременно. Таким образом, если обращение к объекту происходит только внутри критической секции, то состояние гонки не может возникнуть.

Помимо корректности от многопоточной программы требуется эффективность. Однако использование неэффективных примитивов синхронизации может сильно замедлить многопоточную программу. Реализация эффективной блокировки зависит от особенности платформы, на которой запускается программа. Например, примитив синхронизации может показывать хорошие результаты на процессорах архитектуры ARM, но при этом медленно работать на процессорах архитектуры x86. Большую роль в выборе блокировки играет архитектура памяти.

В последние десятилетия большую популярность для высоконагруженных серверов набирает архитектура памяти с неоднородным доступом, или NUMA архитектура. Современные процессоры работают быстрее памяти, поэтому производители пытаются сократить потери на обращение в нее и делают рядом с процессором локальную память с быстрым доступом. Для эффективного использования NUMA архитектуры памяти требуется использовать специальные паттерны для структур данных.

Наряду с архитектурой памяти, выбор блокировки зависит от вида потоков для которых она будет использоваться. В Java до недавнего времени существовали только тяжелые потоки. В качестве экспериментальной опции в Java 19 версию были добавлены виртуальные или легкие потоки [10]. Тем не менее, их использование тоже может привести к состоянию гонки.

Для Java не были найдены реализации эффективных блокировок для виртуальных потоков на NUMA архитектуре. Такие примитивы сихнронизации могут быть полезны для приложений, которые будут работать на NUMA системах и использовать легкие потоки. Такими приложениями могут быть Apache Spark, Kafka, Apache Hive. В связи с этим, был инициирован проект по разработке примитивов синхронизации для легких потоков в Java на NUMA архитектуре совместно с компанией Huawei. Для достижения цели выпускной квалификационной работы были поставлены и решены следующие задачи:

а) создание инструмента измерения эффективности блокировок;
б) реализация эффективной блокировки, учитывающей особенность устройства целевых процессоров;
в) измерение эффективности примитивов синхронизации, сравнение с решениями из стандартной библиотеки.

В главе 1 представлено описание предметной области.

В главе 2 описано устройство инструмента измерения эффективности блокировок.

В главе 3 описана реализация блокировки на языке Java.

В главе 4 представлены результаты измерения эффективности примитивов синхронизации и их сравнение.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Выбор эффективной реализации примитива синхронизации зависит от особенности платформы и используемой реализацией потоков. Поэтому прежде чем перейти непосредственно к написанию блокировки, необходимо разобраться в отличиях и особенностях тяжелых и легких потоков. Нужно понять: кто ими управляет, в каком количестве их может поддержать система. Затем необходимо описать, какое API предоставляет Java для использования легких и тяжелых потоков. Так как примитивы синхронизации должны быть реализованы для NUMA архитектуры, то необходимо разобраться, чем она отличается от UMA. Далее нужно понять: какие узкие места у нее имеются, в чем ее достоинства и недостатки. Затем нужно описать основный принцип эффективных программ, работающих на NUMA. Реализованные примитивы синхронизации будут сравниваться с реализациями из стандартной библиотеки, поэтому нужно описать их принцип работы. Далее нужно описать целевые процессоры, на которых будет происходить тестирование блокировок. В конечном итоге, надо понять, что уже сделано и что нужно сделать в рамках этой работы. Речь о всех этих вопросах пойдет в этой главе.

1.1. Тяжелые и легкие потоки

Основной причиной появления потоков является то, что в приложении может происходить несколько событий одновременно, и мы хотим параллельно их обрабатывать. Концепция потоков в целом похожа на процессы, за исключением того, что потоки делят общее адресное пространство и ресурсы операционной системы (файлы, сетевые соединения). В некоторых приложениях обработка параллельных событий требует какую-то общую память, поэтому недостаточно использовать просто процессы. Также потоки обходятся дешевле чем процессы, они быстрее создаются и разрушаются. Поток обладает локальным состоянием: указатель на следующую исполняемую инструкцию (англ. program counter), регистры (состояние локальных переменных) и текущий стек исполнения. [7]

Тяжелые потоки или потоки операционный системы (ОС) — это потоки управляемые планировщиком операционной системы. Он контролирует время исполнения потоков, место исполнения потока и решает: готов ли поток к исполнению, нужно ли передать ресурсы процессора кому-то другому, нужно ли перенести поток на другое ядро и так далее. Планировщик может разрешить

исполняться другому потоку, когда текущий, например, заблокировался. При этом происходит смена контекста процессора: данные старого потока загружаются в память, а состояние нового выгружается из памяти. Это называется переключение контекста и является дорогой операцией. Кроме того, следует учесть тот факт, что данные кэша, использованные предыдущим потоком, могут быть совершенно бесполезны для нового потока [3, 18].

Легкие (англ. userlevel) потоки — это потоки управляемые на пользовательском уровне, например библиотекой или виртуальной машиной. Обычно они работают поверх заранее созданных потоков уровня ОС и эффективно их переиспользуют. Тем не менее, ядро операционной системы ничего не знает о существовании легких потоков. За время своей жизни легкий поток может исполниться на нескольких тяжелых потоках. Так как они не привязаны к какому-то определенному потоку уровня операционной системы, то система может поддерживать их в большом количестве. Их создание и разрушение обходится гораздо дешевле подобных операций с тяжелыми аналогами. Смена контекста для легких потоков происходит гораздо быстрее, так как это происходит при помощи вызова локальных функций, без обращений к ядру ОС. Легкие потоки могут использоваться для того, чтобы не блокировать поток ОС при исполнении блокирующих операций (например, обращение по сети). Более того, библиотека, предоставляющая виртуальными потоками, может реализовывать свой алгоритм для планировщика, отличающийся от алгоритма планирования операционной системы. Планировщик может учитывать особенности языка (например то, что существует сборщик мусора) или особенности задач.

1.2. Java API для потоков

С версии JDK 1.0 в стандартной библиотеке Java присутствует класс `java.lang.Thread`, который до Java 19 предоставлял разработчикам интерфейс только для работы с потоками операционной системы. Тяжелые потоки в Java принято называть платформенными. Они являются обертками над потоками ОС и соотносятся с ними по схеме 1-1. Таким образом, создание платформенного потока в Java это дорогая операция, и система может поддерживать только ограниченной количество.

С Java 19 класс Thread стал также поддерживать создание легких потоков, которые в языке принято называть виртуальными. История легких потоков берет свое начало в 2017 году, когда стартовала работа над Project

Loom [21], который вводит в Java новое понятие — виртуальные потоки (англ. VirtualThread). Планировщик виртуальных потоков владеет некоторым числом заранее созданных платформенных потоков. Их число регулируется через системные настройки. Виртуальные потоки сопоставляются с платформенными потоками по схеме многие-ко-многим. Тяжелый Java поток, к которому привязан виртуальный, называют несущим (англ. carrier thread). Схема соотношения виртуальных и платформенных потоков представлена на рисунке 1. Легкий поток исполняется до тех пор, пока самостоятельно не вызовет переключение (например `Thread.yield()`). После переключения, когда легкий продолжит исполнение, его несущий тяжелый поток может смениться. Большинство блокирующих вызовов не блокируют несущий поток, однако существует ряд методов, которые по тем или иным причинам блокирует их обоих (например некоторые операции с файловой системой) [16].

Таким образом, легкие потоки дают разработчикам возможность создавать поток под каждую задачу. Задача эффективного использования лёгких потоков полностью ложится на планировщик.

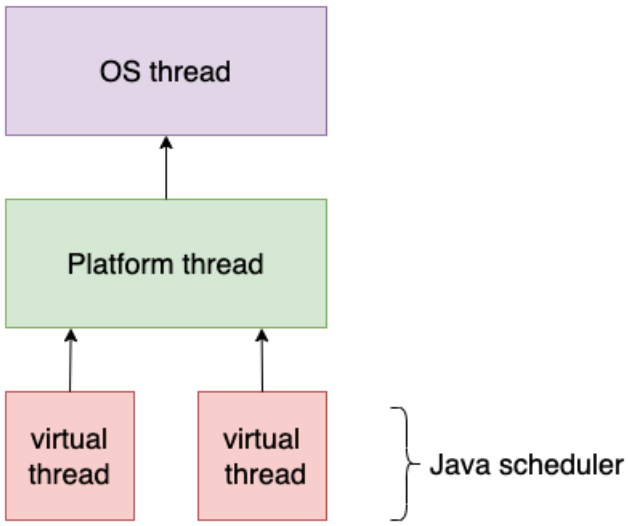

Рисунок 1 – Соотношение виртуальных и платформенных потоков в Java

1.3. Архитектуры памяти

При использовании примитива синхронизации, потоки неизбежно будут обращаться в память. Они могут обращаться как в быструю кэш память, так и в долгую оперативную память. Однако на различных системах память может быть устроена по-разному. Все архитектуры памяти делят на две основные

группы — это UMA (Uniform memory access) и NUMA (Non Uniform Memory access) [23]. У каждой из них есть свои узкие места, поэтому алгоритм, эффективно работающий на одной архитектуре, может плохо показывать себя на другой. Таким образом, необходимо разобраться в особенности устройства данных архитектур памяти, перед тем как реализовывать примитивы синхронизации.

UMA или однородный доступ к памяти — это архитектура памяти, в которой все вычислительные ядра имеют одинаковую скорость доступа к памяти. Такая архитектура имеет простое устройство, поэтому стоит относительно дешево. Процессоры и память соединены шиной передачи данных. Чтобы не нагружать шину, у процессоров есть небольшая, быстрая, локальная память, которая называется кэш памятью. Несмотря на это, при добавлении процессоров нагрузка на шину возрастает, и она не будет успевать передавать данные. Проблемы могут также возникнуть, когда много ядер начнут одновременно читать и писать в один участок памяти. Итого, шина передачи данных является узким местом для UMA. Тем не менее данная архитектура распространена и отлично подходит для систем с одним сокетом и домашних компьютеров

NUMA или неоднородный доступ к памяти — это архитектура памяти, в которой процессоры имеют различное время доступа к памяти. У каждого процессора или группы процессоров есть локальная память, обращение к которой происходит быстрее, чем обращение к удаленной. Скорость обращения процессора к какой-то части памяти называют NUMA расстоянием. Группу процессоров, имеющих одинаковые NUMA расстояния, называют NUMA узлом или NUMA кластером. Работа с локальной памятью не создает нагрузку на внешнюю шину передачи данных, поэтому данная архитектура легко расширяется при добавлении процессоров. Для эффективного использования NUMA нужно использовать специально написанные программы. Такие программы стараются минимизировать обращение к удаленной памяти. NUMA отлично подходит для многопроцессорных систем, например для высоконагруженных серверов. Недостатком такой архитектуры является ее дороговизна и требование использовать специально написанные программы для эффективного исполнения. Схема NUMA архитектуры представлена на рисунке 2.

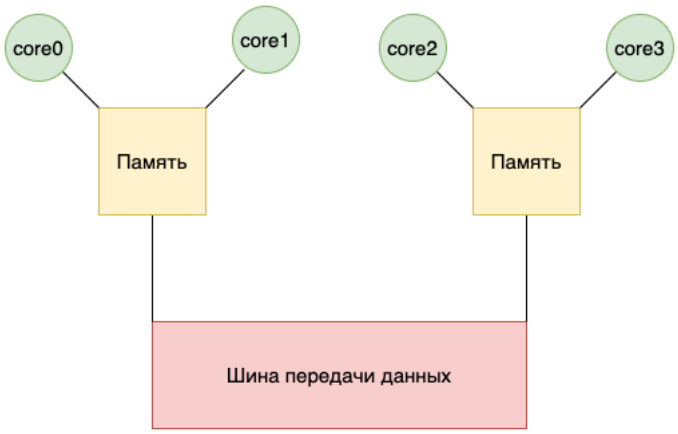

Рисунок 2 – Схема NUMA архитектуры

1.4. Примитивы синхронизации в Java

Стандартная библиотека Java предоставляет несколько способов синхронизации потоков. Реализованная блокировка будет сравниваться со стандартными аналогами, поэтому необходимо представить их описание.

Одним из способов синхронизации потоков является ключевое слово `synchronized`. Его можно указывать перед объявлением метода класса, и тогда оно приводит к следующим результатам. Во-первых, в любой момент времени у объекта может исполнятся только один синхронизированный метод. Во-вторых, выход из `synchronized` метода находится в отношении «произошло до» с любым последующим вызовом синхронизированного метода. Конструкторы класса не могут быть помечены ключевым словом syncronized.

Второй способ использования ключевого слова `synchronized` — это синхронизированные блоки кода (англ. synchronized statements). Чтобы воспользоваться этим способом, необходимо указать объект, на котором будет происходить синхронизация. Синхронизированный блок кода может быть полезен для взятия блокировки над объектом при вызове стороннего метода.

Реализация ключевого слова `synchronized` использует мониторы. Монитор — это сущность, которой обладает любой объект Java. Когда поток входит в какую-либо `synchronized` секцию, он в первую очередь должен захватить монитор объекта. Если захватить монитор не удается, то поток блокируется и ожидает своей очереди. Когда поток покидает синхронизированную секцию, он освобождает монитор, и дает возможность исполнить синхронизированную секцию кому-то другому [9].

Синхронизация — это простой и удобный инструмент, чтобы избежать состояния гонки, который часто часто используется совместно с платформенными потоками. Однако использование ключевого слова `synchronized` с легкими потоками приводит к нежелательным эффектам. Виртуальный поток блокирует несущий поток, когда выполняет `synchronized` секцию [16]. Это приводит к снижению пропускной способности, так как тяжелый поток простаивает и не выполняет работу.

С версии 1.5 в языке представлен интерфейс `java.util.concurrent.locks.Lock`, который предоставляет более обширные возможности для синхронизации потоков. Реализации этого интерфейса могут быть использованы, например, для вложенных блокировок. `Lock` предоставляет разработчику больше контроля над критическими секциями, но при этом накладывает ответственность за взятие и отпускание блокировки.

Одной из реализаций интерфейса `Lock` является класс `ReentrantLock`, который появился в стандартной библиотеке языка Java с версии 1.5. Семантика `ReentrantLock` похожа на семантику ключевого слова `synchronized`. `ReentrantLock` позволяет брать и освобождать блокировку при помощи методов `lock()` и `unlock()`. Помимо этого, поток, обладающий блокировкой, может взять ее еще раз. Также конструктор класса `ReentrantLock` принимает параметр честности. Этот параметр никак не влияет на планировщик операционной системы, однако влияет на порядок, в котором блокировка будет передаваться от одного потока к другому. Честный `ReentrantLock` старается не допустить голодания потоков, поэтому передает блокировку самому долго ожидающему. Нечестный не дает каких-либо гарантий по поводу порядка взятия блокировки.

В данной работе под стандартными блокировками будем подразумевать честный и нечестный `ReentrantLock`. Именно с ними будем сравнивать реализованные примитивы синхронизации. Ключевое слово `synchronized` рассматривать не будем, так как его использование с виртуальными потоками не рекомендуется.

1.5. Целевые процессоры

В разработке примитивов синхронизации для легких потоков на NUMA архитектуре заинтересована компания Huawei, поэтому для тестирования бло-

кировок был предоставлен доступ к компьютерам с процессорами Kunpeng. Этот процессор построен на архитектуре ARM. Измерения проводились на серверах, обладающих различным количество ядер. Такие компьютеры обладают иерархической NUMA архитектурой. Ядра делятся на группы размера четыре, которые называют CCL (полн. CPU Cluster). CCL является самой нижней ступенью иерархии памяти в данных компьютерах. Между ядрами, находящимися в одной четверке, происходит быстрая смена владельца кэш линии. На следующей ступени иерархии несколько (в зависимости от числа ядер) CCL объединяются в один NUMA узел. Пример двухуровневой NUMA архитектуры приведен на рисунке 3. В системах с большим числом ядер выделяют еще одну ступень иерархии памяти. Этот уровень называют SUPERNUMA узлом или NUMA пакетом. Архитектуры, обладающие всеми описанными уровнями, будем называть трехуровневыми. Визуализация такой архитектуры представлена на рисунке 4.

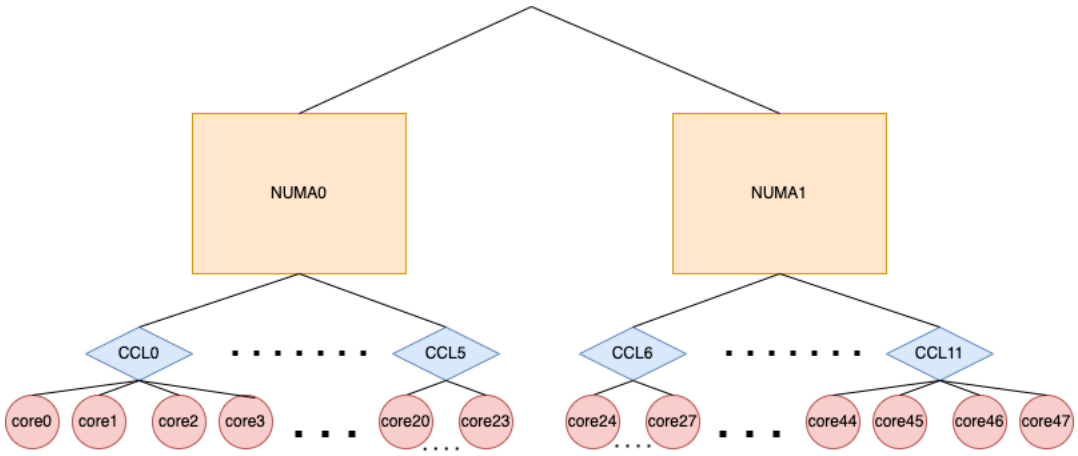

Рисунок 3 – Графическая визуализация архитектуры памяти 48 ядерной системы с Kunpeng 920

1.6. Обзор решений и цели работы

Существует много научных статей, которые описывают реализацию NUMA эффективных примитивов синхронизации для тяжелых потоков. К сожалению, они не подходят для лёгких потоков. Во-первых, они не учитывают их особенности. Во-вторых, в Java отсутствуют реализации эффективных блокировок для легких потоков для NUMA архитектуры. Необходимо реализовать эффективную блокировку для легких потоков в Java для NUMA-архитектуры, реализовать бенчмарк, протестировать и сравнить со стандартными реализациями.

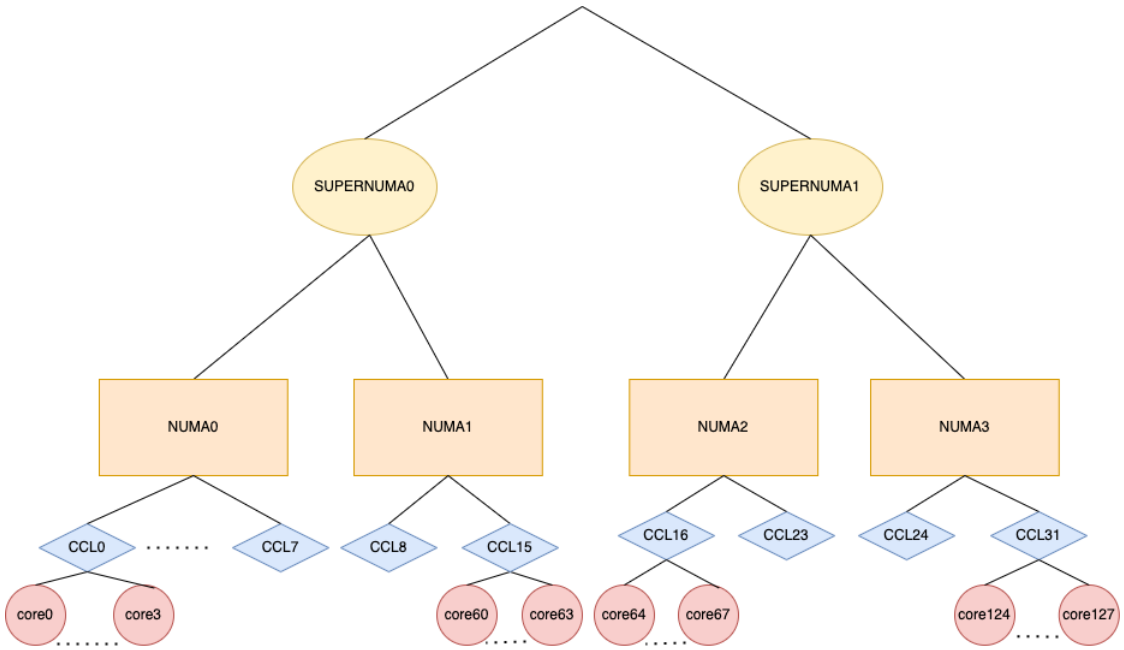

Рисунок 4 – Графическая визуализация архитектуры памяти 128 ядерной системы с Kunpeng 920

Выводы по главе 1

В секции 1.1 было разобрано, чем отличаются тяжелые потоки от легких. Затем в разделе 1.2 представлено описание API, которое предоставляет язык Java для работы с потоками. В секции 1.3 разобраны NUMA и UMA архитектуры памяти и их различия. Затем в разделе 1.4 описаны примитивы синхронизации, которые предоставляет Java. Выяснилось, что в языке отсутствуют блокировки, созданные для эффективной работы на NUMA архитектуре с легкими потоками. В секции 1.5 дано описание целевых процессоров, на которых будет проходить тестирование. В последнем разделе 1.6 рассказано о том, что уже сделано и что нужно сделать в рамках данной работы.

ГЛАВА 2. РЕАЛИЗАЦИЯ ИНСТРУМЕНТА ДЛЯ ИЗМЕРЕНИЯ ЭФФЕКТИВНОСТИ ПРИМИТИВОВ СИНХРОНИЗАЦИИ

Из предыдущей главы мы поняли, что в Java отсутствуют специальные примитивы синхронизации для легких потоков на NUMA архитектуре. Перед созданием таких блокировок, необходимо научиться правильно измерять их эффективность. Поэтому на первом этапе необходимо создать программу, которая позволит оценивать время работы примитива синхронизации. При помощи такого инструмента можно сравнивать реализованные примитивы синхронизации со стандартными. Будем называть такую программу бенчмарком (от англ. benchmark). В этой главе рассмотрим детали реализации этого инструмента, а также ответим на вопросы: как нужно подготовить ОС для тестирования и какие метрики можно использовать для измерения.

2.1. Конфигурация инструмента

Для тестирования блокировок пользователю необходимо задавать параметры запуска. Было принято решение реализовать конфигурацию через текстовый файл в формате JSON [12]. Во-первых, такой подход позволяет удобно редактировать параметры запуска. Во-вторых, такая конфигурация легко расширяется, когда в программу будут добавляться разные режимы измерения. Парсинг конфигурации происходит при помощи библиотеки Jackson. Файл конфигурации содержит различные настройки для тестирования примитивов синхронизации: список блокировок, число потоков, спецификация работы, число запусков, число итераций разогрева JVM, используемые профилировщики, тип бенчмарка. Если пользователь не укажет список потоков, то инструмент сгенерирует его автоматически. Пример конфигурационного файла в приложении А.

2.2. Подготовка ОС к измерению

Платформенные потоки необходимо подготовить для измерения. Если этого не сделать, то результат может быть недетерминированным. Рассмотрим такой пример: бенчмарк запускает виртуальные потоки, которые фиксированное число раз берут блокировку, и замеряет время исполнения. Пусть число виртуальных потоков равно числу ядер на одном NUMA узле. К тому же число тяжелых потоков, выделенных планировщику легких потоков, тоже равно числу ядер на одном NUMA узле. Тогда тяжелые потоки на различных запусках

могут исполняться на различных NUMA узлах, что приведет к неожиданным результатам. Чтобы избежать такого неопределенного поведения, необходимо привязать тяжелые потоки к конкретным ядрами. Java не предоставляет инструмент, чтобы привязать потоки к фиксированный ядрам. Для этого можно использовать системный вызов Linux pthread_setaffinity_np [22]. Реализована утилита на языке C, которая привязывает тяжелый поток к ядру при помощи данного системного вызова. Код этой утилиты представлен в листинге 1. Для вызова этой утилиты из Java используется Java Native Access [15].

Листинг 1 – Утилита на языке C, которая привязывает тяжелый поток к ядру

```
int pinToCore(int cpuId) {
  pthread_t current_thread = pthread_self();
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(cpuId, &cpuset);
  return pthread_setaffinity_np(current_thread, sizeof(cpu_set_t),
      &cpuset);
}
```

2.3. Реализация инструмента

Для реализации инструмента решено использовать фреймворк Java Microbenchmark Harness (сокр. JMH), который позволяет измерять время выполнения Java программы [19]. Также JMH предоставляет утилиты, которые можно использовать, чтобы избежать оптимизаций компилятора во время тестирования. В файле конфигурации бенчмарка можно задать параметр, отвечающий за общее количество работы, называемый $actionsCount$. Работа делится равномерно между потоками. Количество работы, исполняемой одним потоком, будем называть $localWork$. Поток итерируется в цикле $localWork$ раз и на каждом шаге: выполняет работу вне критической секции, вызывает переключение (`Thread.yield()`), берет блокировку, выполняет внутри критической секции какую-то работу, снова вызывает переключение, отпускает блокировку. Перед тем как начать работу, легкие потоки привязывают несущие потоки к ядрам. Для этого потоки ждут друг друга на барьере, не сотрудничающим с планировщиком легких потоков. Затем потоки по очереди привязывают несущие к ядрам (при помощи системного вызова). Далее потоки ожидают друг друга на CyclicBarrier. Затем инструмент дожидается, когда все потоки закончат исполнение, и при помощи JMH получает время исполнения. Псевдокод

алгоритма, выполняемого каждым потоком представлен в листинге 2. Итоговым результатом является медиана времени выполнения всех запусков.

Для оценки эффективности потоков принято решение использовать пропускную способность. Для измерения пропускной способности необходимо общее количество работы разделить на время выполнения работы параллельно. Чем больше данная величина, тем лучше себя показывает блокировка.

Результаты замеров записываются в файл формата CSV. Для удобства сравнения полученных результатов, была реализована программа визуализации на языке Python. Программа строит графики при помощи библиотеки Matploblib и объединяет их в файл формата PDF.

Листинг 2 – Псевдокод алгоритма выполняемого каждым потоком

```
localWork = actionsCount / threadCnt
VThread {
      for i in 0..localWork {
            externalWork()
            yield()
            lock.lock()
            internalWork()
            yield()
            lock.unlock()
      }
}
```

Для измерения эффективности блокировки необходимо выбрать работу, которая будет выполняться потоком. Нужно помнить об оптимизациях, которые могут изменить фактическое количество работы, что может привести к непредсказуемым результатам. Для этого в JMH существует функция `Blackhole.consumeCPU()`, принимающая некоторое число токенов. Этот метод потребляет процессорное время, почти линейно пропорциональное числу токенов [17]. Коэффициент пропорции зависит от системы. В качестве работы решено использовать эту функцию.

2.4. Измерение на различных уровнях конкуренции

Потоки в системе могут обращаться за блокировкой без конкуренции или одновременно с большой конкуренцией. Некоторые блокировки в зависимости от уровня конкуренции по-разному себя ведут. При низкой конкуренции потоки берут потоки по быстрому пути, а при высокой конкуренции по долгому пути (то есть поток становится в очередь и через какое-то время засыпает).

Поэтому было интересно измерить эффективность блокировок на различных уровнях конкуренции.

При помощи размера внешней и внутренней работы можно контролировать уровень конкуренции потоков. Если работа внутри критической секции меньше либо равна работе внутри критической секции, то такая конфигурация соответствует высокой конкуренции потоков. Графическое объяснение представлено на рисунке 5. Вход в критическую секцию ждут всегда несколько потоков, что и создает высокую конкуренцию.

Конфигурация, когда количество работы вне критической секции минимум в число ядер больше количества работы внутри критической секции, соответствует низкой конкуренции потоков. Графическое объяснение представлено на рисунке 6.

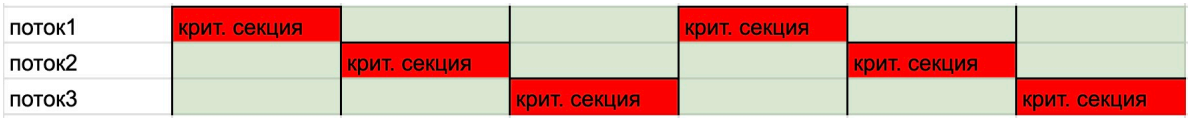

Рисунок 5 – Создание высокой конкуренции, при помощи работы внутри критической секции и отсутствия работы вне

Рисунок 6 – Создание низкой конкуренции, при помощи большой работы вне критической секции

Выводы по главе 2

В этой главе была описана реализация инструмента для измерения эффективности примитивов синхронизации. С его помощью можно измерить пропускную способность блокировки. В разделе 2.1 описано, как реализована конфигурация данного инструмента. В секции 2.2 объяснено, как и зачем нужно подготавливать ОС для измерения блокировок. Далее в разделе 2.3 подробно описано, как реализована программа измерения эффективности блокировок. Там же даны ответы на вопросы: какие измерения производятся, как считаются метрики, какую работу выполняют потоки. В секции 2.4 описано, как реализовано измерение эффективности на различных уровнях конкуренции. При помощи реализованного бенчмарка можно сравнивать эффективность блокировок между собой.

ГЛАВА 3. РЕАЛИЗАЦИЯ ПРИМИТИВОВ СИНХРОНИЗАЦИИ ДЛЯ ЛЕГКИХ ПОТОКОВ НА NUMA АРХИТЕКТУРЕ

В предыдущей главе была описана реализация инструмента измерения эффективности примитивов синхронизации. С его помощью можно оценить пропускную способность блокировок. Следующим этапом является реализация эффективных примитивов синхронизации для легких потоков на NUMA архитектуре. В этой главе рассмотрим особенности реализованной блокировки на языке Java.

3.1. Реализации эффективной блокировки

Реализованная блокировка представляет из себя модификацию MCS. Далее будем называть ее NUMA_MCS. Блокировка инициализирует несколько очередей, каждая из которых является очередью MCS [5] на одном из NUMA узлов. Таким образом, число очередей равно числу NUMA узлов. Для каждой очереди хранится ее хвост в переменной типа AtomicReference. Блокировка самостоятельно определяет количество NUMA узлов в системе, что позволяет пользователю не конфигурировать блокировку. Помимо локальной очереди, блокировка поддерживает переменную типа boolean, операции с которой будут производиться атомарно. Эта переменная является флагом, показывающим свободна блокировка или нет. В локальную очередь поток кладет экземпляр класса Node. Происходит это при помощи атомарной операции CAS на хвосте соответствующей локальной очереди. Псевдокод класса Node представлен в листинге 3. Поле `spin` предназначено для того, чтобы известить поток о том, что он стал лидером локальной очереди. Поле next используется для того, чтобы следующий поток в очереди мог известить предыдущий о своем существовании. При помощи поля `thread` предыдущий поток может разбудить следующий в очереди. Основная идея блокировки заключается в том, чтобы снизить число обращений потока в удаленную память.

Рассмотрим детальнее алгоритм взятия блокировки. Виртуальный поток получает текущий номер NUMA узла при помощи утилиты (особенности ее реализации представлены в разделе 3.2). Затем поток пытается взять блокировку при помощи CAS операции на флаге. Если потоку это удается — он получает блокировку. Если потоку удалось сразу получить блокировку, то будем называть такое поведение быстрым путем получения блокировки. Быстрый путь

позволяет эффективно брать блокировку при отсутствии конкуренции потоков.

Листинг 3 – Псевдокод класса узла для локальной NUMA очереди

```
class Node {
    Thread thread = Thread.currentThread();
    volatile boolean spin = true;
    AtomicRef<Node> next = new AtomicRef<>();
}
```

Если пройти по быстрому пути не удалось, то поток берет блокировку по долгому пути. Сначала он инициализирует экземпляр класса Node. По текущему номеру NUMA узла поток встает в соответствующую локальную очередь. Поток либо становится лидером очереди (если он первый в очереди), либо обозначает себя для предыдущего потока и в цикле ожидает когда ему передадут лидерство. Поток получает лидерство, когда поле `spin` в его узле эквивалентно значению false. Далее поток лидер пытается обновить флаг при помощи CAS. Визуализации алгоритма взятия блокировки представлена на рисунке 8. Поток VT1 сделал неуспешный CAS на флаге взятия блокировки и встал в очередь на своем NUMA узле. Находясь в очереди, поток сделает небольшое количество итераций ожидания, а затем засыпает при помощи вызова `LockSupport.park()`. Это позволяет не потреблять впустую процессорное время и дать исполниться другим легким потокам. Когда предыдущий в очереди поток будет отпускать блокировку, он разбудит следующий поток. После этого текущий поток ждет успешного CAS на флаге взятия блокировки. Псевдокод метода взятия блокировки представлен в листинге 4.

Листинг 4 – Псевдокод метода взятия блокировки NUMA_MCS

```
if cas(flag, 0, 1) {
    return;
}
var cur = Node()
prev = localQueues[getNumaId()].getAndSet(cur);
while (prev != null && cur.spin) {}
while !cas(flag, 0, 1) {}
```

Далее рассмотрим детальнее алгоритм отпускания блокировки. Когда поток отпускает блокировку, он атомарно пишет в флаг взятия блокировки

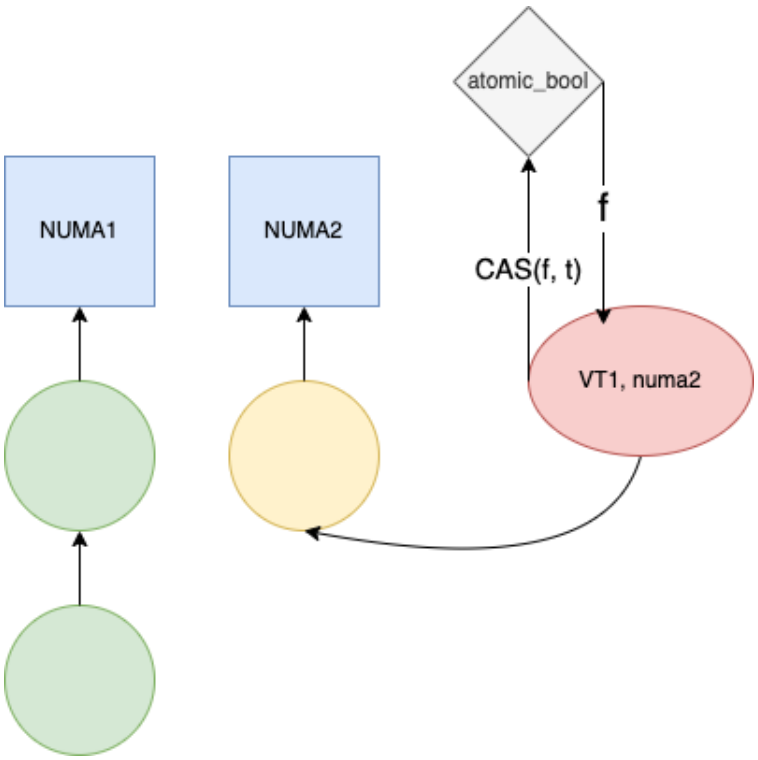

Рисунок 7 – NUMA_MCS, взятие блокировки

значение false (то есть блокировка свободна) и будит следующий за ним в локальной очереди (если такой имеется). Пробуждение происходит при помощи вызова `LockSupport.unpark()`. Визуализация алгоритма отпускания блокировки представлена на рисунке 8. Поток VT1 атомарно записывает значение в флаг и будит следующий поток VT2.

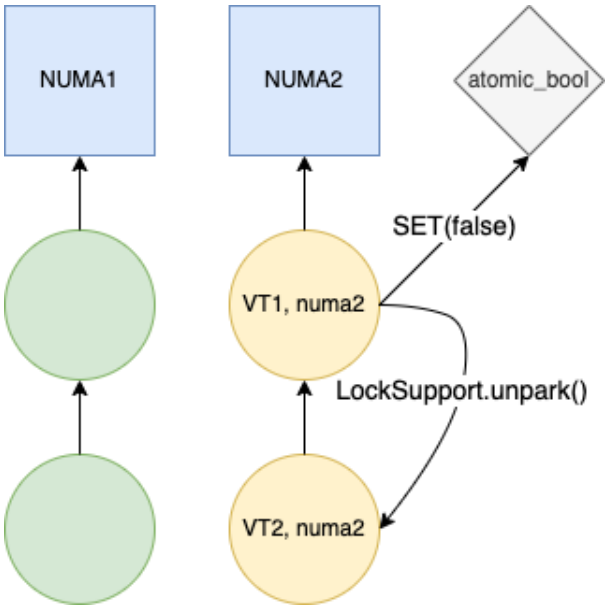

Рисунок 8 – NUMA_MCS, отпускание блокировки

Интересным замечанием является то, что виртуальный поток может встать в очередь на каком-то NUMA узле, затем уснуть, а когда придет время брать блокировку, то проснуться уже на другом NUMA узле. Ожидается, что хотя бы иногда поток будет просыпаться на той же NUMA, при этом в худшем случае пропускная способность будет не хуже ReentrantLock.

3.2. Реализация вспомогательных утилит для NUMA эффективной блокировки

Для реализации эффективной NUMA блокировки необходимо реализовать утилиту для получения номера NUMA узла и процессора. Некоторые блокировки для тяжелых потоков, такие как CNA [2], HCLH [4], HMCS [1] используют эти значения для эффективной работы. Стандартная библиотека Java не имеет готового решения, поэтому было принято реализовать утилиту при помощи системного вызова.

Системный вызов Linux `getcpu` позволяет определить номер процессора и NUMA узла, на котором исполняется поток [11]. Для исполнения системного вызова в Java был использован Java Native Interface (сокр. JNI) [13]. Код, реализующий эту утилиту, представлен в листинге 5. При помощи небольших изменений была также реализована утилита для получения номера текущего процессора. Эта утилита поддерживает архитектуры процессора x86 и ARM. Важным замечанием является то, что вызов утилиты не прерывает исполнение легкого потока [16], то есть поток останется исполняться на том же тяжелом потоке.

3.3. Кэширование номера NUMA узла для легкого потока

В MCS_NUMA легкий поток узнает текущий номер NUMA при каждом взятии блокировки. Для этого можно на каждом взятии вызывать утилиту, реализованную в разделе 3.2, однако это может плохо повлиять на производительность, так как системный вызов — это дорогая операция. Поэтому можно закэшировать номер NUMA узла на несколько взятий блокировок. И раз в некоторое время обновлять это значение. Для этого возможно использовать класс ThreadLocal, который поддерживает в том числе виртуальные потоки. Однако использование ThreadLocal вместе с виртуальными потоками может привести к большому потреблению памяти, так как приложение может создать огромное количество легких потоков.

Листинг 5 – Получение текущего NUMA узла для потока при помощи системного вызова JNI

```
public static int getClusterID() {
    int res;
    var numaNode = new IntByReference();
    var cpu = new IntByReference();
    if (Platform.isARM()) {
        res = CLibrary.INSTANCE.syscall(GET_CPU_ARM_SYSCALL, cpu,
            numaNode, null);
    } else {
        res = CLibrary.INSTANCE.syscall(GET_CPU_x86_SYSCALL, cpu,
            numaNode, null);
    }
    if (res < 0) {
        throw new IllegalStateException();
    }
    return numaNode.getValue();
}
```

Чтобы избежать большого потребления памяти, принято решение хранить номер текущего кластера на стороне несущего (англ. carrier) потока. Число несущих потоков ограничено, поэтому ThreadLocal выделит небольшое количество памяти. Виртуальному потоку необходимо получить информацию о своем несущем потоке, чтобы прочитать его закэшированное значение. Стандартная библиотека Java не предоставляет публичного метода, позволяющего получить несущий поток для виртуального. Был исследован исходный код OpenJDK 19 и найден приватный метод, позволяющий получить информацию о несущем потоке. Часть исходного кода представлена в листинге 6

Листинг 6 – Часть исходного кода класса Thread

```
/**
    * Returns the Thread object for the current platform thread.
      If the
    * current thread is a virtual thread then this method returns
      the carrier.
*/
@IntrinsicCandidate
static native Thread currentCarrierThread();
```

Чтобы вызвать этот метод, можно воспользоваться технологией reflection [20]. Однако вызов метода с использованием рефлексии гораздо медленнее прямого. Рефлексию нужно избегать в частях программы, которые

часто исполняются и требуют высокой производительности. Вызов метода получения несущего потока происходил бы при каждом взятии блокировки, поэтому использование рефлексии снизило бы производительность. MethodHandle API позволяет делать вызовы приватных методов быстрее, чем если бы это делалось через reflection. Использование static final переменной типа MethodHandle позволяет достичь скорости вызов метода соизмеримую с прямым вызовом [14]. Для реализации блокировки использован именно этот подход.

После получения несущего потока, нужно обратиться в его ThreadLocal переменную, в которой хранится информация о NUMA узле. Класс ThreadLocal не предоставляет публичного интерфейса, чтобы получить значение не для текущего потока. Был исследован исходный код класса ThreadLocal и найден способ получить локальную переменную для потока не являющимся текущим. Часть исходного кода класса ThreadLocal представлена в листинге 7. При помощи MethodHandle функция получения значения по потоку может быть вызвана, и виртуальный поток получит информацию о NUMA узле или процессоре несущего потока.

Листинг 7 – Часть исходного кода класса ThreadLocal

```
public T get() {
    return get(Thread.currentThread());
}
```

3.4. Решение проблемы false sharing

Эффективность блокировки зависит в том числе от расположения данных в памяти. Необходимо избежать ситуации, когда различные потоки меняют данные, попавшие на одну кэш линию. Такая ситуация может возникнуть, когда данные двух экземпляров NUMA_MCS попадут на одну кэш линию. К тому же несколько узлов для локальной очереди могут попасть на одну кэш линию. Для решения этой проблемы необходимо выровнять данные под размер кэщ линии. В случае целевой системы Kunpeng — это 128 байт. Этот эффект был достигнут при помощи аннотации @Contended [8], которая активно используется внутри JDK. С ее помощью можно избежать попадания полей двух классов на одну кэш линию. Класс узла локальной очереди `Node` и класс блокировки NUMA_MCS помечены этой аннотацией. Раз-

мер выравнивания конфигурируется при запуске JVM при помощи параметра `ContendedPaddingWidth`.

Выводы по главе 3

В этой главе описаны особенности реализации примитива синхронизации для легких потоков на NUMA архитектуре. В разделе 3.1 подробно описана реализация блокировки NUMA_MCS. В разделе 3.2 описана реализация вспомогательных утилит для данной блокировки. Далее в секции 3.3 дано объяснение, как и зачем нужно кэшировать номер NUMA узла. Затем в разделе 3.4 объяснено, как избегалась проблема false sharing.

ГЛАВА 4. ИЗМЕРЕНИЕ ЭФФЕКТИВНОСТИ БЛОКИРОВОК ДЛЯ ЛЕГКИХ ПОТОКОВ НА NUMA СИСТЕМЕ

В предыдущей главе были описаны особенности реализованной блокировки для легких потоков на NUMA архитектуре. Далее необходимо сравнить NUMA_MCS с примитивами синхронизации из стандартной библиотеки (честный и нечестный ReentrantLock). Для измерения эффективности, необходимо запустить бенчмарки на системах с NUMA архитектурой. В данной главе речь пойдет об анализе полученных результатов. На основе этого будут сделаны выводы об эффективности блокировок.

При этом протестировать блокировки в реальных приложениях на момент написания ВКР не представляется возможным. Виртуальные потоки находятся в JDK лишь в `preview` режиме, и какие-либо большие проекты еще не используют нововведение.

4.1. Описание среды для запуска бенчмарков

Тестирование проводилось на серверах с различным число ядер, с процессорами линейки Kunpeng-920. Процессоры данной линейки обладают архитектурой aarch64.

Первая система имеет 48 ядер и два NUMA узла. Ядра 0-23 относятся к NUMA-0, ядра 24-47 относятся к NUMA-1.

Вторая система имеет 128 ядер и четыре NUMA узла, по 32 ядра на каждый NUMA узел. NUMA узлы попарно объединяются в SUPERNUMA узел.

Параметры JVM, использованные для запуска тестов, представлены в листинге 8. Тестирование проводилось на OpenJdk версии 19.0.2.

Для каждого теста создавалось четыре экземпляра JVM, и на каждом экземпляре проводилось десять прогревочных итераций и десять итераций замера.

Листинг 8 – Параметры JVM для запуска бенчмарков

```
java -XX:+UseParallelGC -XX:+UseNUMA -XX:-RestrictContended --add-
    opens java.base/java.lang=ALL-UNNAMED --enable-preview -jar
    NUMA-aware-locks.jar
```

4.2. Тестирование при высокой конкуренции потоков

Было проведено тестирование на легких потоках при высокой конкуренции. В главе 2 подробно описано, как эмулируется высокая конкуренция. В

данных запусках отсутствует внешняя работа, а работа внутри критической секции занимает порядка 3000 наносекунд. Результаты для 48 ядерной системы представлены на рисунке 9. На рисунке видно, что у NUMA_MCS наибольшая пропускная способность. Ее пропускная способность в среднем на 20% больше по сравнению с лучшей стандартной (UNFAIR_REENTRANT).

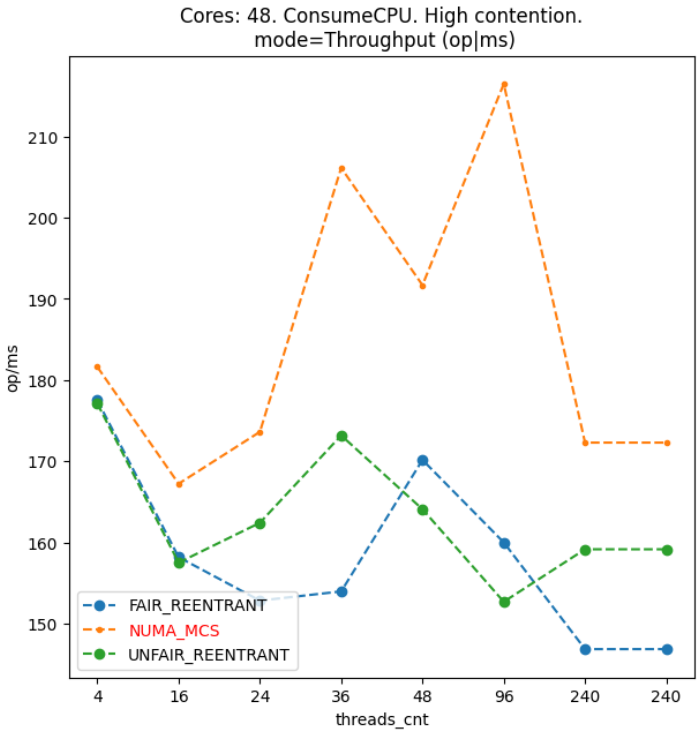

Рисунок 9 – Пропускная способность на 48 ядерной системе, микробенчмарк, высокая конкуренция

Тестирование на системе со 128 ядрами показало похожие результаты, которые представлены на рисунке 10. Реализованная блокировка NUMA_MCS имеет наибольшую пропускную способность на высокой конкуренции и опережает стандартные блокировки Java.

4.3. Тестирование при низкой конкуренции потоков

Было проведено тестирование на легких потоках при низкой конкуренции. В главе 2 подробно описано, как эмулируется низкая конкуренция. Работа внутри критической секции занимает порядка 3000 наносекунд, а внешняя работа в 128 раз больше. Это позволяет создавать условие низкой конкуренции на всех целевых системах. Результаты для 48 ядерной системы представлены на рисунке 11. На рисунке видно, что пропускная способность реализованной блокировки больше пропускной способности лучшей стандартной блокировки

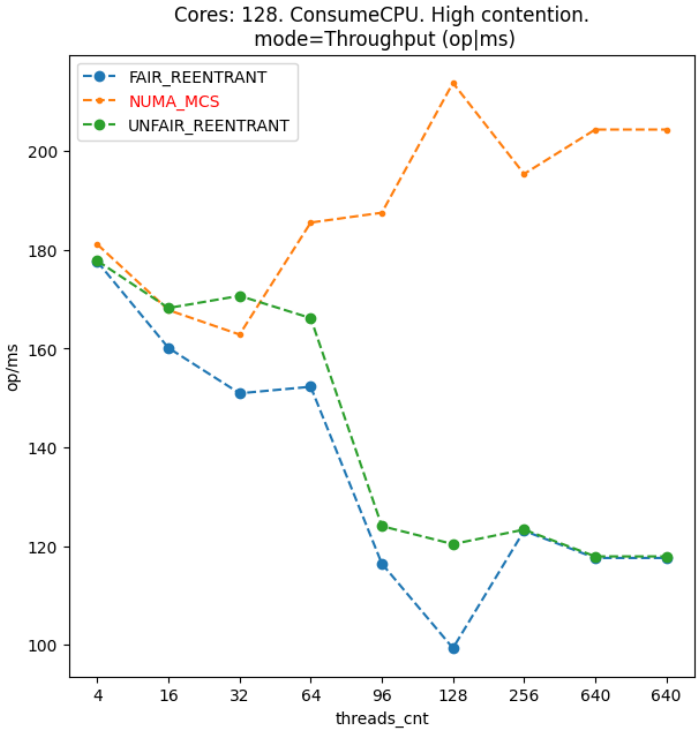

Рисунок 10 – Пропускная способность на 128 ядерной системе, микробенчмарк, высокая конкуренция

UNFAIR_REENTRANT в среднем на 2%. Более хорошие результаты наблюдаются на 128 ядерной системе и представлены на рисунке 12. На этой системе пропускная способность NUMA_MCS на 40% больше лучшей стандартной.

4.4. Тестирование на задаче текстовой статистики

Чтобы показать эффективность реализованной блокировки, принято решение провести тестирование на задаче поиска текстовой статистики.

Задача ставится так: дан массив из 100000 строк, а каждую строку нужно разбить на слова и посчитать число появлений каждого слова. Для этого заведем несколько HashMap и столько же блокировок. Работа с HashMap происходит только внутри критической секции. Слово кладется в тот или иной ассоциативный массив в зависимости от хэша слова (подобное разделение реализовано в классе ConcurrentHashMap). Строки текста делятся между потоками равномерно. Произведены замеры времени выполнения данной задачи при использовании различных блокировок. Результаты для 48 ядерной системы представлены на рисунке 13. На рисунке видно, что при использовании блокировки NUMA_MCS время исполнения самое низкое.

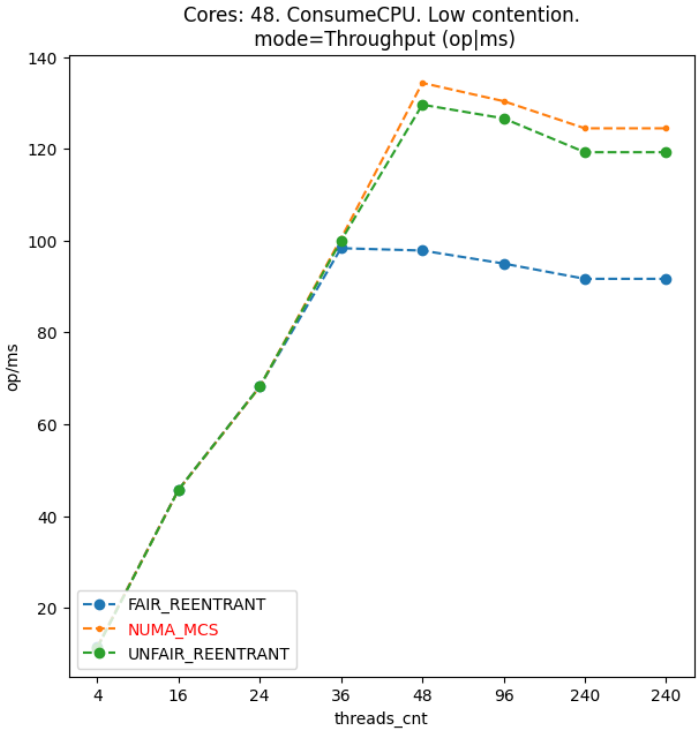

Рисунок 11 – Пропускная способность на 48 ядерной системе, низкая конкуренция

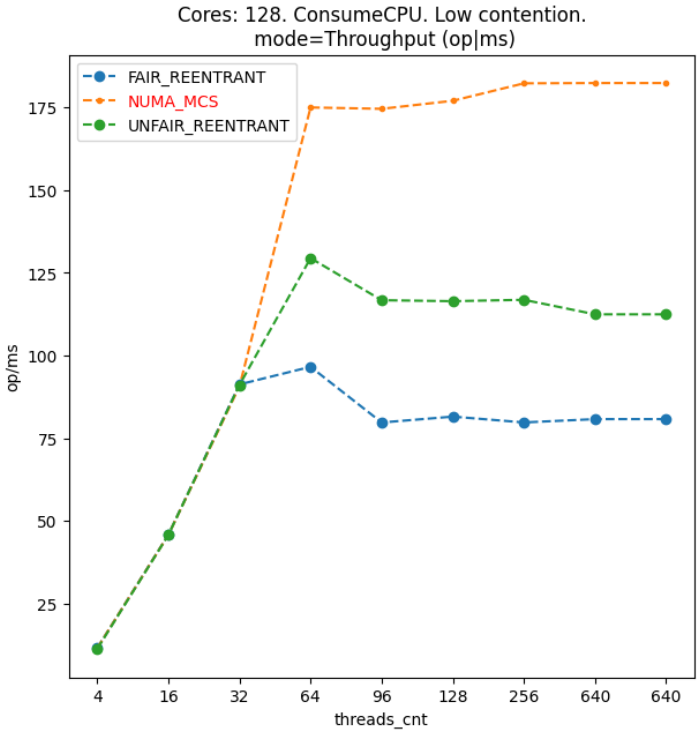

Рисунок 12 – Пропускная способность на 128 ядерной системе, низкая конкуренция

Выводы по главе 4

В этой главе предоставлены измерения эффективности реализованной и стандартных блокировок, а также проведен анализ полученных результатов.

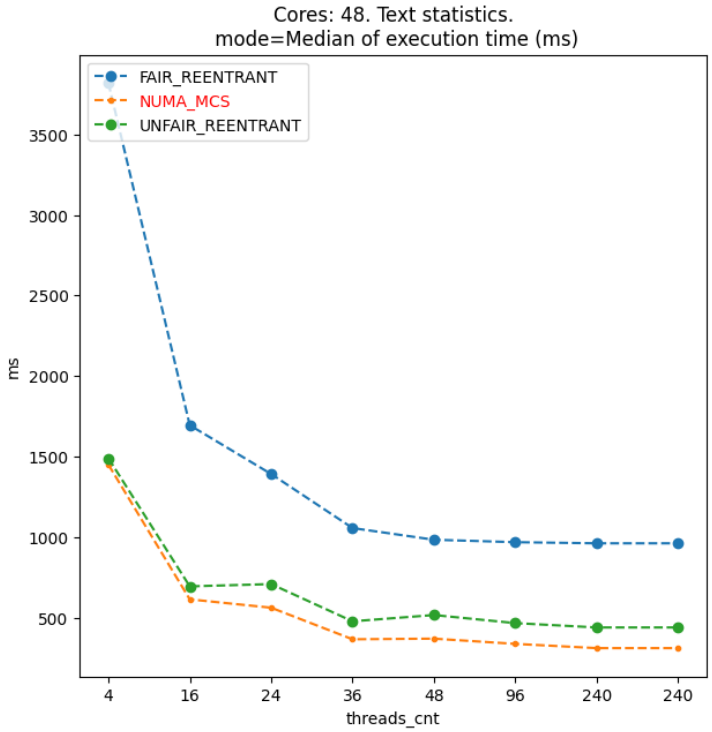

Рисунок 13 – Время выполнения задачи поиска статистики по тексту на 48 ядерной системе

В разделе описаны технические характеристики систем, на которых проводилось тестирование. Также дано описание параметров, передаваемых в JVM для проведения замеров.

В разделе 4.2 описаны результаты, полученные при тестировании на высоком уровне конкуренции потоков. Лучший результат показала реализованная блокировка NUMA_MCS, которая в среднем имеет пропускную способность на 20% больше по сравнению со стандартными.

В разделе 4.3 описаны результаты, полученные при тестировании на низком уровне конкуренции потоков. Реализованная блокировка NUMA_MCS опережает по пропускной способности лучшую стандартную блокировку на 48 ядерной системе в среднем на 2% процента. На 128 ядерной системе NUMA_MCS опережает лучшую стандартную в среднем на 20%.

В разделе 4.4 представлены результаты для задачи поиска статистики текста. Решение с разработанной блокировкой NUMA_MCS опережает по времени исполнения решения со стандартными блокировками.

ЗАКЛЮЧЕНИЕ

В этой работе подробно описаны особенности реализации эффективной блокировки для легких потоков на NUMA архитектуре. Были изучены научные статьи о NUMA-aware блокировках для тяжелых потоков и на их основе реализован примитив синхронизации для легких потоков в Java. Реализованная блокировка учитывает особенности целевой NUMA архитектуры.

Был реализован инструмент для тестирования эффективности блокировок. С его помощью можно измерять пропускную способность реализованных примитивов синхронизации. Бенчмарк позволяет тестировать блокировки при различных уровнях конкуренции. Измерение проводилось для высокой и низкой конкуренции потоков. Также проводилось тестирование на задаче поиска текстовой статистики.

На высокой конкуренции реализованная блокировка MCS_NUMA имеет пропускную способность в среднем на 20% больше по сравнению с лучшей стандартной.

На низкой конкуренции реализованная блокировка MCS_NUMA имеет пропускную способность в среднем на 2% больше по сравнению с лучшей стандартной на 48 ядерной системе. На 128 ядерной системе MCS_NUMA имеет пропускную способность на 20% больше по сравнению с лучшей стандартной на низкой конкуренции.

Реализованная блокировка может использоваться на целевых процессорах при работе с легкими потоками на высокой и низкой конкуренции.

В дальнейшем планируется провести тестирование реализованной блокировки в больших проектах, использующих многопоточность и блокировки. Однако на данный момент виртуальные потоки находятся лишь в режиме preview и не используются в больших проектах.

Дополнительными результатами стали использование материалов данной работы для выступления с докладом на Конгрессе молодых ученых в Университете ИТМО.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Chabbi M.*, *Fagan M.*, *Mellor-Crummey J.* High Performance Locks for Multi-Level NUMA Systems // SIGPLAN Not. — New York, NY, USA, 2015. — Янв. — Т. 50, № 8. — С. 215–226. — ISSN 0362-1340. — DOI: 10.1145/2858788.2688503. — URL: https://doi.org/10.1145/2858788.2688503.

2 *Dice D.*, *Kogan A.* Compact NUMA-Aware Locks // Proceedings of the Fourteenth EuroSys Conference 2019. — Dresden, Germany : Association for Computing Machinery, 2019. — (EuroSys '19). — DOI: 10.1145/3302424.3303984. — URL: https://doi.org/10.1145/3302424.3303984.

3 *Kavi K.* Multithreading Implementations. — 1998. — Сент.

4 *Luchangco V.*, *Nussbaum D.*, *Shavit N.* A hierarchical CLH queue lock //. Т. 4128. — 11.2006. — С. 801–810. — ISBN 978-3-540-37783-2. — DOI: 10.1007/11823285_84.

5 *Mellor-Crummey J. M.*, *Scott M. L.* Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors // ACM Trans. Comput. Syst. — New York, NY, USA, 1991. — Февр. — С. 21–65. — ISSN 0734-2071. — DOI: 10.1145/103727.103729. — URL: https://doi.org/10.1145/103727.103729.

6 Multithreading Implementations / T. Farah [и др.] // IISCI. — 2018. — Т. 16. — С. 22–25. — URL: https://www.iiis.org/CDs2017/CD2017Summer/papers/SA025BU.pdf.

7 *Tanenbaum A.*, *Bos H.* Modern Operating Systems, 4th Edition". — Pearson Higher Education, 2015. — ISBN 978-0133591620.

8 A Guide to False Sharing and @Contended [Электронный ресурс]. — URL: https://www.baeldung.com/java-false-sharing-contended (visited on 04/07/2023).

9 *Beqir Hamidi L. H.* Synchronization Possibilities and Features in Java // IJIS. — 2015. — Apr. — Vol. 1. — P. 75–84. — URL: https://revistia.org/files/articles/ejis_v1_i1_15/Beqir_Hamidi.pdf.

10 Coming to Java 19: Virtual threads and platform threads [Электронный ресурс]. — URL: https://blogs.oracle.com/javamagazine/post/java-loom-virtual-threads-platform-threads (visited on 02/11/2022).

11 getcpu(2) — Linux manual page [Электронный ресурс]. — URL: https://man7.org/linux/man-pages/man2/getcpu.2.html (visited on 02/16/2022).

12 Introducing JSON [Электронный ресурс]. — URL: https://www.json.org/json-en.html (visited on 02/15/2022).

13 Java Native Interface Overview [Электронный ресурс]. — URL: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html (visited on 02/16/2022).

14 Java Reflection, but much faster [Электронный ресурс]. — URL: https://www.optaplanner.org/blog/2018/01/09/JavaReflectionButMuchFaster.html (visited on 04/07/2023).

15 Java-native-access github page [Электронный ресурс]. — URL: https://github.com/java-native-access/jna (visited on 04/24/2023).

16 JEP 425: Virtual Threads [Электронный ресурс]. — 2023. — URL: https://openjdk.org/jeps/425 (visited on 02/11/2022).

17 JMH Core 1.23 javadoc [Электронный ресурс]. — URL: https://javadoc.io/static/org.openjdk.jmh/jmh-core/1.23/org/openjdk/jmh/infra/Blackhole.html (visited on 04/24/2023).

18 Microsoft Learn [Электронный ресурс]. — URL: https://learn.microsoft.com/en-us/dotnet/standard/threading/threads-and-threading (visited on 02/11/2023).

19 OpenJDK: JMH [Электронный ресурс]. — URL: https://openjdk.org/projects/code-tools/jmh/ (visited on 02/15/2022).

20 Oracle official site: Using Java Reflection [Электронный ресурс]. — URL: https://www.oracle.com/technical-resources/articles/java/javareflection.html (visited on 02/16/2022).

21 Project Loom [Электронный ресурс]. — URL: https://openjdk.org/projects/loom/ (visited on 02/11/2022).

22 pthread_setaffinity_np(3) — Linux manual page [Электронный ресурс]. — URL: https://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html (visited on 04/07/2023).

23 *Vanya Rajput Sanjay Kumar V.* Performance Analysis of UMA and NUMA Models // IJCSET. — 2012. — Oct. — Vol. 2. — P. 1457–1458. — URL: http://www.ijcset.net/docs/Volumes/volume2issue10/ijcset2012021006.pdf.

ПРИЛОЖЕНИЕ А. ПРИМЕР КОНФИГУРАЦИОННОГО ФАЙЛА

Листинг А.1 – Конфигурационный файл бенчмарка

```
{
  "benches": [
    {
      "name": "consumeCpu",
      "payload": {
        "actionsCount": 64000,
        "beforeCpuTokens": 0,
        "inCpuTokens": 1000,
        "warmupIterations": 7,
        "measurementIterations": 7,
        "forks": 3,
        "yieldsBefore": 1,
        "yieldInCrit": true,
        "title": "ConsumeCPU. High contention.",
        "skip": true,
        "locks": [
          {
            "name": "NUMA_MCS"
          },
          {
            "name": "UNFAIR_REENTRANT"
          },
          {
            "name": "FAIR_REENTRANT"
          }
        ]
      }
    }
  ]
}
```