Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

СРАВНИТЕЛЬНЫЙ АНАЛИЗ МЕХАНИЗМОВ ВЗАИМНОГО ИСКЛЮЧЕНИЯ В МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЯХ

Автор Тараканов Денис Сергеевич _____
(Фамилия, Имя, Отчество) (Подпись)

Направление подготовки
(специальность) 09.04.04 «Программная инженерия»

Квалификация магистр

Руководитель Косяков М.С., доцент, к.т.н. _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

К защите допустить

Зав. кафедрой Алиев Т.И., профессор, д.т.н. _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

«___ »_____ 20 ____ г.

Санкт-Петербург, 20 18 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Объектом исследования являются механизмы взаимного исключения, используемые в многопоточных приложениях.

Предметом исследования является влияние различных механизмов взаимного исключения на производительность многопоточных приложений.

Цель исследования состоит в повышении производительности многопоточных приложений, использующих механизмы взаимного исключения.

Для достижения цели исследования решаются следующие **задачи**:

1. Провести обзор различных механизмов взаимного исключения, которые используются для синхронизации работы потоков в многопоточном приложении.
2. Провести подробный анализ спин-блокировок и шаблона проектирования «Активный Объект», а также вариантов их реализации для решения задачи взаимного исключения.
3. Подготовить библиотеку рассмотренных вариантов реализаций спин-блокировок и шаблона проектирования «Активный Объект»
4. Разработать тестовые многопоточные программы для проверки эффективности реализаций, представленных в написанной библиотеке.
5. Проанализировать полученные результаты тестов.

Методами исследования, применяемыми в работе, являются анализ (рассмотрение возможных реализаций механизмов взаимного исключения), эксперимент (проверка корректности тестовой библиотеки и сравнение представленных в ней реализаций) и методы математической статистики (для обработки полученных результатов). **Средства** исследования: для разработки программ использовался языки программирования высокого уровня C (стандарт C99) и C++ (стандарт C++11), для реализации многопоточности – библиотека потоков POSIX (pthread), в качестве компилятора - GCC 5.4.0 и clang-3.8.0, также

применялись отладчик GDB 7.11.1, GNU make 4.1 и операционная система Ubuntu 16.0.4 LTS.

Актуальность темы. В современном мире все более остро встает вопрос повышения производительности приложений. В то время как рост вычислительных возможностей ограничен законом Мура, а возможности распараллеливания в сложных многоядерных и многопроцессорных системах законом Амдала, одним из важных аспектов становится более эффективное использование процессорного времени каждого ядра. Также это относится и к возможному простою ядер процессора при ожидании доступа к критической секции. Тем не менее, многие вопросы не находят широкого освещения.

Так, например, существует ряд исследований посвященных сравнению эффективности спин-блокировок на многопроцессорных системах с раздельной кэш-памятью, при этом нет точных данных о наличие подобных проблем на многоядерных процессорах.

Большинство решений задачи взаимного исключения связаны с прямой или косвенной блокировкой и ожиданием освобождения критической секции, тогда как существуют альтернативные, полностью неблокирующие поток подходы, например, шаблон проектирования «Активный Объект». Существующие исследования не рассматривают возможность его применения к задаче взаимного исключения.

Следовательно, возникает необходимость в более подробном исследовании проблем спин-блокировок, а также данного шаблона проектирования и его конкурентоспособности в сравнении с классическими алгоритмами.

Апробация результатов исследования. Основные положения и исследования работы обсуждались на XLVI и XLVII Научной и учебно-методической конференции Университета ИТМО. Также была подготовлена к публикации статья «Сравнительный анализ реализаций спин-блокировок», принятая к публикации журналом «Программные продукты и системы».

Новизна работы состоит в рассмотрении различных вариантов решения задачи взаимного исключения и подробном разборе преимуществ и недостатков применения шаблона проектирования «Активный Объект» для алгоритмов взаимного исключения в многопоточном приложении, которые не имеют широкого освещения в существующих исследованиях.

ГЛАВА 1. СПОСОБЫ РЕШЕНИЯ ЗАДАЧИ ВЗАИМНОГО ИСКЛЮЧЕНИЯ В МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЯХ

1.1 Многопоточность и взаимное исключение в современных операционных системах

Постепенно развитие операционных систем привело их к необходимости эффективно использовать ресурсы процессора и избегать возможных простоев. Изначально эта проблема решалась с помощью многозадачности – возможность псевдопараллельно выполнять несколько процессов.

Современные операционные системы предлагают дополнительный механизм, являющийся развитием многозадачности. Этим механизмом является многопоточность, которая позволяет в рамках одного процесса выполнять несколько потоков.

Поток обладает различными характеристиками, такими как состояния его выполнения, контекст, стек, статическая память. Но также все потоки одного процесса имеют общее адресное пространство и файловые дескрипторы [1]. Это определяет одно из наиболее важных преимуществ использования многопоточность над многопроцессностью – не требуется никаких дополнительных механизмов (таких как общая память, каналы и другие) для взаимодействия потоков между собой.

В то же время, если несколько потоков предпримут попытку обращения к общему ресурсу – это может привести к неопределенному поведению, т.е. нельзя будет точно предсказать возможный результат выполнения такой программы.

Таким образом возникает необходимость предоставления потокам эксклюзивного доступа, к некоторым общим ресурсам. Код, в котором осуществляется этот доступ, называется критической секцией, а решение данной задачи – алгоритмом взаимного исключения. Общая структура кода с критической секцией представлена на листинге 1 [2].

```
void P1 (int R){
        while (true) {
        /* предшествующий кол */
        request_cs(R);
        /* критическая секция */
        release_cs(R);
        /* последующий код */
        }
}
```

Листинг 1 – Общая структура кода с критической секцией

Стоит отметить основные свойства, которым должен удовлетворять алгоритм взаимного исключения:

1) безопасность – гарантируется, что в критической секции не может находиться больше одного потока в каждый момент времени
2) живучесть – каждый поток должен рано или поздно попасть в критическую секцию.

Дополнительно можно выделить более строгий вариант свойства живучести – выполнение условия справедливости, при котором потоки поочередно могут получить доступ к критической секции. Свойство справедливости не является обязательным для алгоритма взаимного исключения, но тем не менее наличие у алгоритма этого свойства делает его более выгодным по отношению к другим реализациям.

1.2 Блокирующие подходы к решению задачи взаимного исключения

Существует множество алгоритмов, решающих задачу взаимного исключения. Среди них можно выделить классические программные решения (алгоритмы Петерсона и Деккера) и аппаратные, основанные на поддержке

процессорами атомарных инструкций (например, инструкций Test-and-Set и Compare-and-Swap). Рассмотрим подробнее некоторые из них.

Первым известным корректным решением задачи взаимного исключения является алгоритм Деккера.

Выполнение свойства безопасности в данном алгоритме обеспечивается с помощью двух переменных-флагов и специальной переменной «вход». Для сообщения о намерении войти в критическую секцию поток устанавливает соответствующую ему переменную-флаг. Если флаг другого потока не установлен – то можно безопасно войти в критическую секцию. В ином случае поток сбрасывает свой флаг и ожидает, пока переменная «вход» не укажет, что наступила его очередь войти в критическую секцию [3]

Ключевым недостатком алгоритма Деккера является возможность синхронизации только двух потоков между собой. Обобщением для большего числа потоков можно считать алгоритм Петерсона.

Для него вводится тот же набор переменных. Желающий войти в критическую секцию поток должен установить свой флаг и записать в переменную «вход» номер второго потока. Затем поток в бесконечном цикле ожидает, пока другой поток не сбросит флаг.

Основным преимуществом данных алгоритмов является то, что они не требуют наличия у процессора специальных атомарных команд и могут работать вне зависимости от архитектуры процессора и операционной системы. Тем не менее, алгоритм Петерсона приводит к ожиданию потоком в так называемом холостом цикле – постоянной проверке значения переменной, пока она не начнет удовлетворять некоторому условию, что является нежелательным с точки зрения использования ресурсов процессора.

Аппаратные решения задачи взаимного исключения также можно разделить на две группы: блокирующие алгоритмы (такие примитивы синхронизации, как

семафоры, мьютекс, различные виды спин-блокировок) и неблокирующие алгоритмы. Рассмотрим наиболее важные из них.

Определение понятия семафор впервые дано Дейкстрой. Представляет собой объект, который ограничивает количество одновременно находящихся в критической секции потоков. Семафоры управляются двумя операциям, которые принято называть V и P. Операция V увеличивает счетчик семафора на единицу. Это сигнализирует об освобождение места в критической секции для другого потока. Операция P уменьшает счетчик семафора и позволяет потоку войти в критическую секцию, если значение счетчика было положительным, иначе поток ожидает освобождения семафора [4].

Примитив синхронизации мьютекс является аналогом бинарного семафора. Ключевая его особенность – в случае провала захвата критической секции поток должен перейти в состояние сна. Для этого требуется наличие у операционной системы специального системного вызова.

Подробнее рассмотрим, как реализован мьютекс в библиотеке pthreads для операционной системы Linux. Ключевым механизмом для создания мьютексов в ней служит фьютекс (от английского FUTEX – Fast Userspace muTEX). Фьютекс является выровненной в памяти целочисленной переменной, с которой связана очередь ожидания в пространстве ядра [5].

Переменная хранит состояние самого фьютекса, ее изменение проводится за одну атомарную инструкцию (обычно используется инструкция вида Compare-and-Swap). Если оказывается, что в критической секции уже есть другой поток, и фьютекс захвачен им, то используется специальный системный вызов с параметром FUTEX_WAIT. Действие потока приостанавливается (процесс захвата переходит в «медленный» вариант), до тех пор, пока фьютекс не будет разблокирован (примет необходимое для этого значение), и этот поток будет иметь в очереди наивысший приоритет.

Процесс разблокировки критической секции происходит в 2 этапа. Сначала выходящий из нее поток модифицирует значение переменной фьютекса с помощью атомарной инструкции. По предыдущему значению переменной он может определить о существовании потоков в очереди ожидания. Если такие потоки существует, то используется специальный системный вызов с параметром FUTEX_WAKE, который пробуждает следующий в очереди поток [6]. Подобный двухэтапный подход приводит к тому, что мьютекс перестает гарантировать выполнение условия справедливости. Это связано с тем, что описанные этапы происходят не атомарно, таким образом в момент между изменением значения переменной фьютекса и реальным началом выполнения «спящего» потока, какой-либо другой поток может успеть захватить критическую секцию. Тем не менее, такая ситуация может серьезно повлиять на работу примитива синхронизации только при небольших объемах критической секции [7]

Можно отметить, что мьютекс, представленный в библиотеки pthreads, обладает дополнительными возможностями. Так, он может быть рекурсивным (предоставлять одному и тому же потоку заблокировать его несколько раз) или запоминать идентификатор заблокировавшего его потока (для проверки того, верный ли поток пытается его освободить). Также мьютекс используется для реализации более сложного примитива синхронизации – условной переменной.

Схожим с мьютексом примитивом синхронизация является спин-блокировка. Главное ее отличии от мьютекса – использование холостого цикла для проверки переменной спин-блокировки в процессе ожидания освобождения критической секции. По внутреннему механизму можно выделить несколько реализаций спин-блокировок.

В библиотеке потоков POSIX спин-блокировка представлена целочисленной переменной, работу с которой компилятору запрещено оптимизировать (например, для избегания ситуации, когда такая переменной попадет в регистр ядра процессора, будет изменяться только в нем, и таким

образом другие потоки не узнают о ее модификации). При инициализации этой переменной присваивается значение 1. Поток при попытке захвата такой спин-блокировки применит к ней операцию атомарного декремента, после чего проверит флаг процессора Zero Flag (ZF), устанавливающийся в случае, если результат выполненной инструкции равен нулю. Если ZF равен 1, то поток может войти в критическую секцию. В противном случае, по аналогии с алгоритмом Петерсона будет использовать холостой цикл, пока переменная спин-блокировки не станет больше нуля, после чего весь алгоритм повторится.

Стоит отметить возникающий проблему при использовании такого холостого цикла. В таком цикле поток продолжает выполняться на процессоре, не совершая при этом никакой полезной работы. Дополнительно, холостой цикл может приводить к проблемам с температурой процессора, а также постоянным ошибкам предсказателя ветвления. Решением этой проблемы на архитектуре x86-64 является ассемблерная инструкция pause, которая предупреждает процессор о том, что выполняющийся поток находится в холостом цикле. В отличие от инструкции nop (которую процессор пытается выполнить за минимальное количество тактов) выполнение инструкции pause растягивается, например, на процессорах Intel с микроархитектурой Skylake (6 и 7 поколения процессоров Intel Core) ее выполнение может достигать 140 циклов [8]. Такая оптимизация также используется и в других версиях спин-блокировок.

К достоинствам этой реализации спин-блокировки можно отнести ее простоту и отсутствие необходимости в поддержки процессором атомарных инструкций вида «Test-and-Set». В то же время, очевидно, что отсутствие какой-либо внутренней очереди приводит к тому, что данная спин-блокировка не выполняет условие справедливости – доступ потоков к критической секции не распределяется равномерно. Дополнительно можно поставить под сомнение и наличие свойства живучести, поскольку возможно ситуация, когда один из потоков будет длительное время «голодать», в то время как другие потоки будут получать доступ к критической секции.

Данную проблему решает билетная спин-блокировка, которую можно увидеть, например, в ядре операционной системы Linux. Эта спин-блокировка является структурой данный, состоящей из двух целочисленных переменных – «следующего билета» и «обслуживаемого билета» [9].

При попытке захвата критической секции поток использует атомарную инструкцию «Increment-and-Swap» для переменной, хранящей номер «следующего билета». Затем он ожидает, пока номер полученного им «билета» не окажется обслуживаемым. При выходе из критической секции поток увеличивает номер «обслуживаемого билета».

К преимуществам данного вида спин-блокировки следует отнести фактическое наличие внутренней очереди – потоки могут получить доступ к критической секции только в том порядке, в каком они попытались ее захватить; к недостаткам – возможность переполнения переменных и проблемы с масштабируемостью.

Как показывает ряд исследований при увеличении количества потоков, пытающихся захватить критическую секцию, защищённую такой спин-блокировкой, на многопроцессорных и многоядерных системах возникает проблема необходимости синхронизации значения переменной холостого цикла между различными процессорами или ядрами. В ходе работы цикла она попадает в кэш-память процессора и, когда поток выходит из критической секции и изменяет значение этой переменной, то происходит обновление записей кэш-памяти каждого процессора, что негативно сказывается на эффективности всего алгоритма взаимного исключения [10-11].

Спин-блокировкой, обеспечивающей высокую масштабируемость, является MCS-блокировка (в названии используется аббревиатура из фамилий авторов идеи). Эта блокировка также использует очередь в пространстве пользователя. Но здесь она реализуется с помощью связного списка, при этом каждый узел этого списка хранит целочисленную переменную. При попытке захвата такой спин-

блокировки поток добавляет себя в очередь. Если она пуста, он заходит в критическую секцию. Иначе поток устанавливает значение своей переменной равным 1 и в холостом цикле ожидает ее изменения. При выходе из критической секции поток удаляет себя из очереди, если он единственный в ней, либо же воспользовавшись тем, что это связанный список, изменяет переменную холостого цикла в следующем в очереди потоке, позволяя ему войти в критическую секцию [12].

Использование связанного списка и разделение между потоками переменных холостого цикла позволяет гарантировать этой спин-блокировке высокую масштабируемость. С другой стороны, это приводит к усложнению ее реализации и дополнительным накладным расходам.

Стоит отметить, что существуют дальнейшие развития механизма MCS-блокировки, например, CLH-блокировка [13] или K-42-блокировка [14], но в данной работе они рассматриваться не будут, так как вносят в механизм сугубо оптимизационные улучшения.

Также существует примитив синхронизации под названием «адаптивный мьютекс», который является гибридным решением. Его принцип работы можно разделить на две фазы. В первой фазе он функционирует как обычная спин-блокировка. Если в течение определенного количества циклов потока так и не удалось войти в критическую секцию, то происходит переход во вторую фазу – примитив синхронизации начинает работать как вышеописанный мьютекс и использует системный вызов для ожидания освобождения критической секции. Количество попыток во время работы в первой фазе может как определяться пользователем, быть фиксированным, или же вычисляться на основе статистики, накопленной разными потоками при работе с конкретным адаптивным мьютексом.

Данный примитив синхронизации представлен в библиотеки pthreads для Linux (как частный случай обычного мьютекс), так и в операционной системе Windows (как примитив CRITICAT_SECTION).

Другим подходом к решению задачи взаимного исключение является использование неблокирующих алгоритмов [15]. Все эти алгоритмы основаны на использовании специальных атомарных операций. В неблокирующей синхронизации выделяют три уровня: без ожиданий (код критической секции может быть выполнен потоком за конечное число шагов вне зависимости от посторонних факторов), без блокировок (достаточно гарантировать продвижение всей системы вперед, а не конкретного потока), без препятствий (поток может продвинутся вперед только в отсутствии конкуренции) [16].

Существует большое количество реализаций неблокирующих алгоритмов. Чаще всего они представлены как часть алгоритмов для конкурентной работы со структурами данных [17] или простыми атомарным операциями (такими как инкремент, Compare-and-Swap).

Стоит отметить высокую сложность реализации подобных алгоритмов и наличие специфичных проблем (зацикливание, проблема ABA и другие).

1.3 Шаблон проектирования «Активный Объект»

В прошлом разделе были представлены различные примитивы синхронизации, решающие задачу взаимного исключения. Тем не менее, можно выделить общий минус для всех этих алгоритмов – доступ к критической секции потоки получают синхронно. Таким образом, поток (а иногда вместе с ним и ядро процессора) вынужден простаивать в ожидании критической секции вместо того, чтобы заниматься полезной работой. Дополнительно, неточные реализации сложных механизмы взаимного исключения могут приводить к зацикливаниям или взаимоблокировкам.

Задачу взаимного исключения можно решить с помощью асинхронного подхода. Для этого требуется выделить отдельный поток и дать ему

эксклюзивный доступ к критической секции. Остальные потоки должны иметь механизм быстрого и удобного взаимодействия с выделенным потоком, в ходе которого они смогут передавать ему задачи, требующий выполнения в критической секции, а затем забирать результат, либо иметь возможность получить уведомление о завершении операции.

Такого способ организации алгоритма, в котором поток выполнения задачи отделяется от потока, которому требуется результат выполнения задачи, принято называть шаблоном проектирования «Активный Объект».

К основным преимуществам данного подхода можно отнести не только асинхронное выполнение операция с критической секцией, но и возможности по гибкому управлению над очередью задач, более высокую масштабируемость, независимость от аппаратных особенностей процессоров и механизмов операционной системы. При этом критическая секции внутри «Активного Объекта» может быть представлена как простой задачей (какие-либо вычисления), эксклюзивным доступом к структуре данных или устройству, так и более сложными алгоритмами. В общем случае, потоки могут передавать в «Активный Объект» код, который должен быть для них выполнен в данной критической секции.

Явными минусами является необходимость на каждую критическую секцию создавать отдельный поток. Также классические примитивы синхронизации не требуют копировать данные, которые должны быть обработаны в критической секции (что может быть ощутимо при больших обрабатываемых объемах). Потенциально возрастают накладные расхода и из-за формирования некоторого объекта «задания» для выполнения «Активным Объектом».

Более подробно внутреннее устройство шаблона проектирования «Активный Объект» и его применение к решению задачи взаимного исключения будет рассмотрено в главе 3.

1.4 Выводы по главе 1

В главе 1 было дано развернутое пояснение термина многопоточность, раскрыта необходимость решение задачи взаимного исключения, сформулированы основные условия, которые необходимо выполнять примитивам синхронизации.

Были описаны различные способы решения этой задачи, такие как блокирующие примитивы синхронизации, неблокирующие алгоритмы, возможность использования шаблона проектирования «Активный Объект».

Блокирующие подходы можно разделить на несколько видов.

Мьютекс позволяет отправлять потоки в сон, вместо активного ожидания доступа к критической секции. В то же время не которые реализации мьютексов позволяют нарушить условие справедливость.

Спин-блокировки напротив, используют холостой цикл в ожидании критической секции. Они могут иметь различное внутреннее устройство. POSIX-блокировка, например, в отличие от «билетной» спин-блокировки и MCS-блокировки не использует внутреннюю очередь.

Адаптивный мьютекс является гибридным решением, позволяющим в случае длительного холостого цикла уйти потоку в сон.

Неблокирующие алгоритмы разделяют на алгоритмы без ожидания, без блокировок и без препятствий.

Альтернативным подходом к решению задачи взаимного исключения является использование шаблона проектирования «Активный Объект». Он позволяет потокам передавать выполнение некоторых задач в отдельный поток. Таким образом, выделенный поток можно безопасно работать с критической секцией.

Тем не менее реализация шаблона «Активный Объект» вносит серьезные накладные расходы, которые могут негативно сказываться на эффективности

приложений, использующих его. Другой минус «Активного Объекта» - необходимость выделять отдельный поток на каждую критическую секцию.

ГЛАВА 2. ИСПОЛЬЗОВАНИЕ СПИН-БЛОКИРОВОК ДЛЯ РЕШЕНИЯ ЗАДАЧИ ВЗАИМНОГО ИСКЛЮЧЕНИЯ

2.1 Современные реализации спин-блокировок

Спин-блокировка является одним из основных используемых примитивом синхронизации, который может использоваться для работы с недлительными критическими секциями, защищающими различные структуры данных или действия над ними. В противовес спин-блокировкам другие примитивы синхронизации имеют более конкретные области применения (например, мьютекс в большей степени предназначен для длительных блокировок, а неблокирующий алгоритмы – только для работы со специальными структурами данных).

Тем не менее, как было указано ранее, существуют различные виды спин-блокировок, отличающиеся внутренним устройством. Различия в реализации приводят к изменениям свойств спин-блокировок.

Ключевыми свойствами спин-блокировки можно считать требования к выполнению условия справедливости и минимизацию накладных расходов от работы примитива синхронизации.

Особенно ярко проблема накладных расходов может проявляться при увеличении количества потоков, взаимодействующих с примитивом синхронизации. Существует ряд исследований, показывающих наличие проблем с масштабируемостью на многопроцессорных системах [10-11].

Причины данных проблем могут быть специфичны для определенных систем. Например, слабая масштабируемость спин-блокировок на многопроцессорных системах в первую очередь связана с синхронизацией кэш-памяти процессоров при освобождении критической секции. Тем не менее, при использовании одного многоядерного процессора общим для разных ядер является только кэш последнего уровня, а значит, если возникнет необходимость доступа к одной и той же области памяти у разных ядер, то будет происходить

синхронизация кэш-памяти между ядрами. Это является одним из узких мест при использовании кэш-памяти [8]. Влияние данной проблемы синхронизации на масштабируемость спин-блокировок не установлено.

Для проверки обозначенных выше проблем спин-блокировок была разработана библиотека, включающая такие примитивы синхронизации, как POSIX-блокировка, «билетная» спин-блокировка и MCS-блокировка.

При написании библиотеки использовался язык программирования высокого уровня C (стандарт C99) с использованием GNU-расширений (например, GNU Extended Asm), поддерживаемых компилятором GCC (версия 5.4.0). Дополнительно разработанная библиотека предоставляет общий интерфейс custom_lock для работы со всеми спин-блокировками (выбор блокировки можно осуществлять на этапе компиляции с помощью директив препроцессора).

Проверка корректности реализованных спин-блокировок была проведена с помощью модульных тестов. Полученная библиотека была выложена в свободный доступ [18].

2.2 Реализация спин-блокировок

2.2.1 Реализация POSIX-блокировки

Тип данных спин-блокировки является целочисленной переменной spinlock, хранящей значение единицы, когда спин-блокировка свободна. Если значение этой переменной меньше единицы, то другой поток уже находится в критической секции.

Захват спин-блокировки можно разделить на два этапа. На первом этапе с блокировкой шины декрементируется переменная spinlock, после чего проверяется флаг процессора ZF. Если он установлен, значит перед попыткой захвата спин-блокировки она была свободной – поток может войти в критическую секцию.

Иначе поток уходит в бесконечный холостой цикл, состоящий из 2 ассемблерных инструкций: сравнения переменной spinlock с нулем и инструкции pause, назначение которой было описано ранее.

Для выхода из критической секции и освобождения спин-блокировки поток должен занести в переменную spinlock значение 1 (см. приложение А).

2.2.2 Реализация «билетной» спин-блокировки

Данная разновидность спин-блокировки описывается структурой данных, состоящей из двух полей: __next – хранит значение следующего доступного «билета», и __serv – используется для указания обслуживаемого потока. Они инициализируются нулем.

При попытке захвата используется инструкция “lock; xadd”, позволяющее увеличить значение поля __next (увеличить номер следующего билета), на значение, хранящееся в регистре eax (1), а затем сохранить в регистре eax предыдущее значение __next (получить «билет»). Если номер «билета» совпадает со значение поля __serv – спин-блокировка успешно захвачена.

В противном случае используется холостой цикл, схожий с обычной спин-блокировкой – состоящий из инструкции pause и сравнения «билета» с полем __serv.

Чтобы освободить такую спин-блокировку достаточно увеличить на 1 поле __serv, таким образом, открыв доступ в критическую секцию потоку со следующим по номеру «билетом» (см. приложение А).

2.2.3 Реализация MCS-блокировки

Примитив синхронизации представлен типом MCS_lock_t, который скрывает в себе указатель на структуру MCS_node *L, по которому будет храниться последний элемент в очереди на захват спинлока. Сама структура MCS_node также содержит два поля: указатель на структуру того же типа *next и поле locked, устанавливаемое в значение 1, если поток ожидает входа в

критическую секцию. Также, каждый поток в начале своей работы должен создать собственную структуру MCS_node I, которую он будет использовать для представления себя в очереди ожидания и передавать в функцию блокировки через указатель.

Процесс захвата критической секции осуществляется следующим образом. Поле I->next устанавливается равным NULL (т.е. данный поток последний в очереди). Затем сохраняется текущее значение L, и на его место записывается I. Таким образом, следующий поток при попытке захвата спин-блокировки получит по указателю доступ к переменной I текущего потока.

Если сохраненное значение L равно NULL, значит, в очереди и в критической секции нет других поток – спин-блокировка захвачена. Иначе, поток устанавливает для поля I->locked значение 1, и через сохраненный указатель L записывает в поле next указатель на I, таким образом, сообщая потоку, который выше в очереди, информацию о себе. После чего начинается холостой цикл в ожидании изменения I->locked (аналогично прошлым примерам используются инструкции cmp и pause).

При разблокировке поток проверяет значение поля I->next. Если какой-то поток встал в очередь, то оно равно не NULL, а указателю на структуру I следующего потока. Чтобы позволить войти ему в критическую секцию, достаточно сменить значение I->next->locked на 0.

Если же I->next равно NULL, значит, потоку нужно изъять из L информацию о себе. Но, существует вероятность, что в этот момент другой поток попытается войти в критическую секцию. Поэтому для изменения L используется инструкция “lock; cmpxchg”. Если текущее значение L и I совпадают, значит, в L записывается NULL, спин-блокировка освобождена.

В противном случае поток сталкивается с ситуацией, когда захватывающий поток установил в L, принадлежащую ему I, но не успел сообщить предыдущему в очереди потоку о себе. Поэтому поток должен ожидать, пока значение I->next не

изменится с NULL на указатель, после чего он сможет сменить значение I->next->locked на 0, и позволить следующему потоку войти в критическую секцию (см. приложение А).

2.3 Параметры проведения эксперимента

Для проведения эксперимента по сравнению спин-блокировок использовалось следующее тестовое оборудование: компьютер с процессором Intel Core i7-2630QM 2,0 ГГц и отключенной технологией Turbo Boost (использование данной технологии может приводить к невозможности сравнения результатов между собой ввиду сложного алгоритма изменения тактовой частоты процессора, используемого производителем процессора). В данном процессоре установлены 4 ядра с поддержкой технологии Hyper-threading (позволяет одновременно выполнять до 8 потоков). Кэш первого уровня – 256 Кб, второго уровня – 1 Мб, третьего уровня – 6 Мб.

Для оценки эффективности спин-блокировок между собой использовались следующие характеристики. Первая характеристика - оперативность приложения τ – время (в процессорных циклах), необходимое приложению для выполнения заданного количества итераций count. Другая характеристика – выполнение условия справедливости. Это условие для спин-блокировки подразумевает одинаковое число захватов критической секции каждым из конкурирующих потоков (например, ожидается, что при выполнении четырьмя потоками 1000 итераций, каждый из потоков выполнит по $x_1 = x_2 = x_3 = x_4 = x_{fair} = 250$ итераций). В качестве количественной характеристики выполнения условия справедливости в работе рассматривалось среднее отклонение от xfair, вычисляемое по формуле $\delta = \frac{1}{N}\sum_{k=1}^{N}\frac{|x_k - x_{fair}|}{x_{fair}} * 100\%$, где k ∈ 1..N, N – число конкурирующих потоков. Данные характеристики были выбраны как наиболее информативные на основе уже существующих исследований в области определения производительности многопоточных приложений [9; 10; 12].

Для проведения эксперимента было разработано тестовое приложение, в котором каждый поток выполнял функцию, общий вид которой представлен в листинге 2.

```
1.  void thread_function(){
2.      while(true){
3.          spin_lock();
4.          factorial(i);
5.          count--;
6.          spin_unlock();
7.          factorial(j);
8.          if (count <= 0) return;
9.      }
10. }
```

Листинг 2. Общий вид функции из тестового приложения на языке C

В качестве полезной нагрузки, выполнявшейся в критической секции и в промежутках между попытками ее захвата, использовалось вычисление различных значений факториала (factorial(i) и factorial(j), где i, j – параметры теста). Таким образом, в зависимости от параметров i и j варьировалось соотношение T времени, проводимого потоком в критической секции (T_{cs}), и вне ее (T_{out}). Оно вычислялось по формуле $\delta T = \frac{T_{cs}}{T_{cs}+T_{out}} * 100\%$. Также варьировалось количество потоков N, используемых приложением.

Стоит подробнее остановиться о способе измерения оперативности. Для работы с процессорными циклами использовались высокоточные TSC-счетчики (от английского Time Stamp Counter), позволяющие получить максимально точное время при поддержании постоянной частоты работы процессора [19-20]. С этим и

связано отключение технологии Turbo Boost и использование драйвера частоты процессора acpi-cpufrreq вместо intel-pstate.

Для автоматического проведения тестирования дополнительно использовался Makefile, проводивший компиляцию необходимого набора приложений (по одному на способ реализации примитива), а также скрипт для командной оболочки

2.4 Анализ полученных результатов эксперимента

По результатам тестирования были получены следующие результаты (см. Таблицу Б.1 в приложении Б). На рисунках отмечен разброс результатов измерения времени с доверительной вероятностью 95%. Стоит отметить, что для MCS-блокировки и билетной спин-блокировки он составил доли процентов от измеренных результатов и не виден в масштабе рисунка.

Для приложения, в котором потоки выполняют более длительную задачу в критической секции, а не вне ее (85% времени выполнения приходилось на критическую секцию), при большом количестве потоков лучшую эффективность продемонстрировала MCS-блокировка (Рисунок 1). Схожую эффективность имела и билетная спин-блокировка.

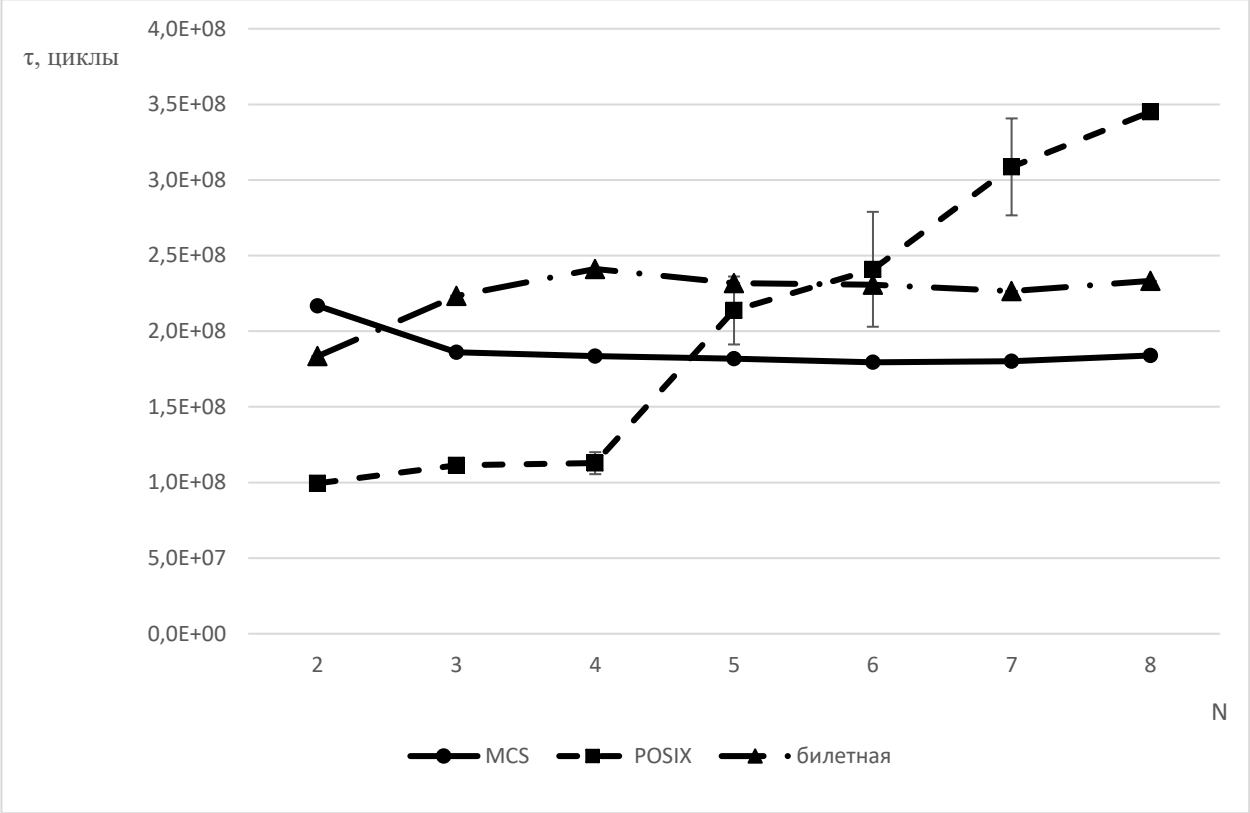

Рисунок 1 – Зависимость времени τ от количества потоков N при δT = 85%

Стоит отметить лучшее время работы POSIX-блокировки при небольшом количестве потоков. Но как можно увидеть в Таблице 1, это связано с тем, что она не гарантировала доступность критической секции для всех потоков, то есть некоторые потоки ни разу не смогли получить доступ к критической секции. Незначительные отклонения от 0% (для билетной спин-блокировки и MCS-блокировки) связаны с неравномерным стартом и прекращением работы потоков.

Дополнительно это приводит к большому разбросу результатов измерений для POSIX-блокировки – в разных тестовых запусках порядок доступа потоков к критической секции мог сильно отличаться, что влияло на необходимость в синхронизации кэш-памяти между ядрами процессора.

Таблица 1 – Отклонение количества захватов критической секции δ при δT = 85%

N	2	3	4	5	6	7	8
MCS	0,04%	0,01%	0,09%	0,1%	4,1%	1%	1,7%

POSIX	100%	120%	120%	102%	62%	30%	25%
билетная	0,03%	0,4%	0,07%	0,08%	0,2%	0,03%	0,1%

Для приложения с δT = 8% наблюдается схожее поведение (Рисунок 2). Стоит отметить понижение эффективности для билетной спин-блокировки и MCS-блокировки при трех потоках. Это связано с возникающей конкуренцией за критическую секцию и очередью на вход в нее.

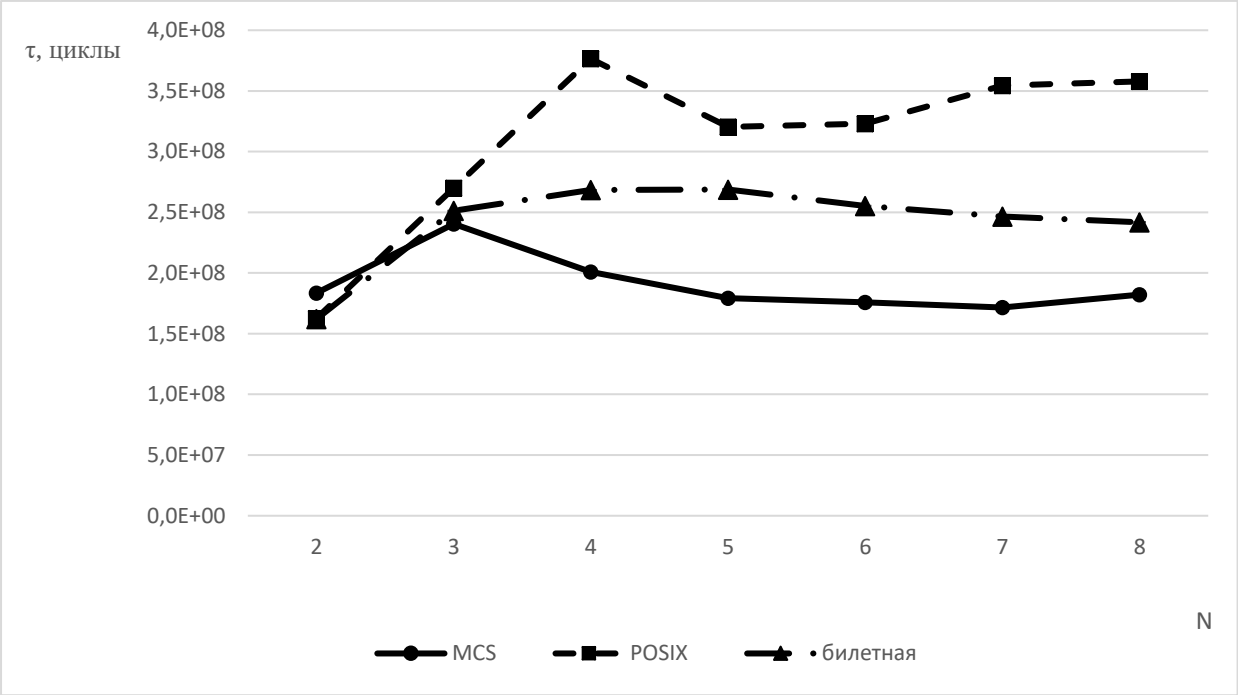

Рисунок 2 – Зависимость времени τ от количества потоков N при δT = 8%

При уменьшении конкуренции потоков за критическую секцию все потоки имеют возможность доступа к критической секции. Это можно увидеть в Таблице 2 для POSIX-блокировки. Тем не менее с увеличением количества потоков (следовательно, и с увеличением конкуренции), POSIX-блокировка перестает быть справедливой.

Таблица 2 – Отклонение количества захватов критической секции δ при δT = 8%

N	2	3	4	5	6	7	8
MCS	0,01%	0,2%	0,02%	0,8%	2,3%	0,9%	0,4%
POSIX	0%	0,1%	8%	28%	17%	16%	12%

билетная	0%	0,01%	0,04%	0,3%	0,2%	0,05%	0,03%

POSIX-блокировка показывает плохую масштабируемость, причиной которой является использование декремента переменной холостого цикла при каждой проверке доступности критической секции. Это приводит к необходимости постоянной синхронизации кэш-памяти между ядрами.

Использование билетной спин-блокировки также связано с общей переменной для холостого цикла. Ее изменение при выходе из критической секции заставляет все ядра процессора сразу синхронизировать кэш-память, что приводит к более низкой эффективности по сравнению с MCS-блокировкой, которая лишена этих недостатков. Тем не менее в некоторых ситуациях (например, при небольшом количестве потоков) накладные расходы от ее использования могут негативно сказаться на эффективности.

Также стоит отметить, что для $\delta T = 0{,}5\%$ наблюдалось повышение эффективности для всех спин-блокировок, что связано с отсутствием очереди при ожидании критической секции. При таком характере работы с критической секцией вид спин-блокировки не влияет на доступ потоков к ней (Рисунок 3 и Таблица 3).

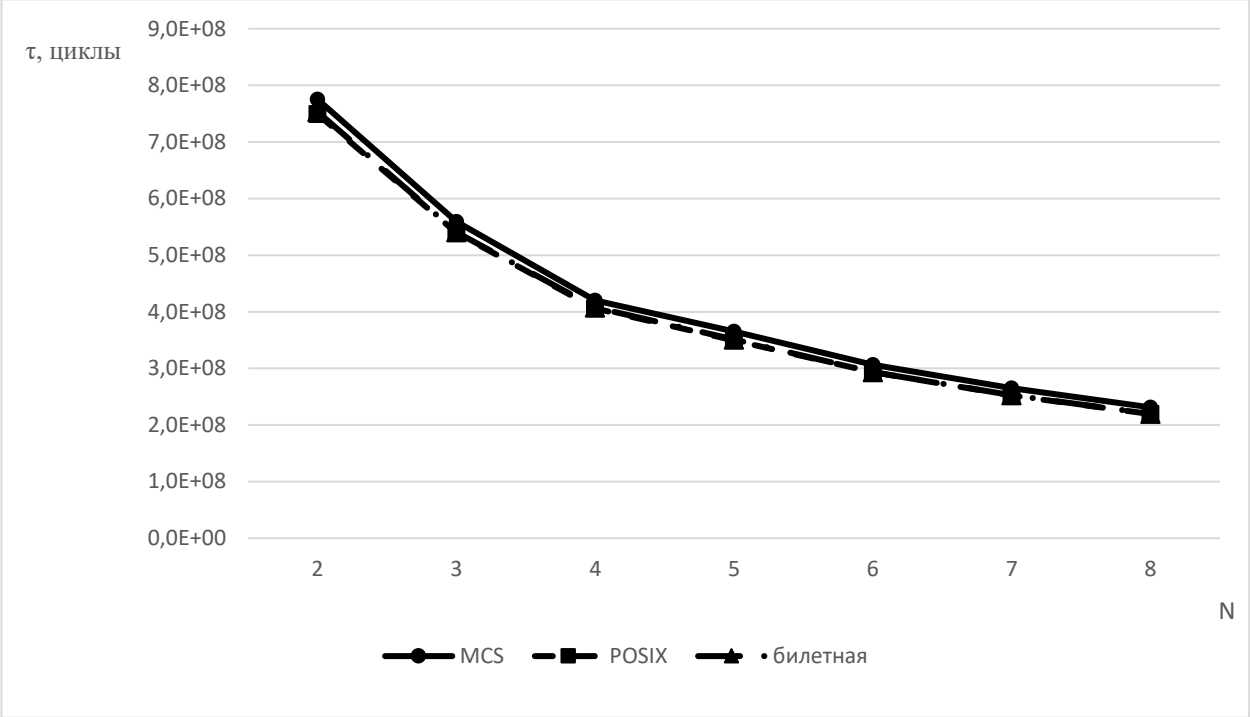

Рисунок 3 – Зависимость времени τ от количества потоков N при δT = 0,5%

Таблица 3 – Отклонение количества захватов критической секции δ при δT = 0,5%

N	2	3	4	5	6	7	8
MCS	0%	0%	0,1%	0,6%	3%	0,3%	0,6%
POSIX	0,01%	0,02%	0,04%	0,5%	0,07%	0,03%	0,05%
билетная	0,02%	0%	0,02%	2,4%	1,6%	0,6%	0,03%

2.5 Выводы по главе 2

В данной главе были подробно рассмотрены различные реализации спин-блокировок, такие как POSIX-блокировка, «билетная» спин-блокировка и MCS-блокировка.

POSIX-блокировка является наиболее простой. В ней используется инструкция атомарного декремента для захвата критической секции. Данная реализация не является справедливой.

«Билетная» спин-блокировка использует внутреннюю очередь для контроля над выполнением условия справедливости. Ей также необходимы атомарные операции.

MCS-блокировка имеет сложную реализацию. Каждый поток хранит свой узел внутренней очереди и проводит холостой цикл в ожидании изменения собственной переменной.

Для сравнения эффективности этих спин-блокировок при использовании многоядерного процессора было разработано тестовое приложение и определены оцениваемые характеристики: оперативность приложения и выполнение условия справедливости, которое количественно оценивалось с помощью среднего отклонения от x_fair.

На основе полученных данных можно сделать выводы о том, что POSIX-блокировка показывает недостаточную масштабируемость в приложениях с большим количеством потоков даже на многоядерных системах с общим кэшем последнего уровня. В тоже время билетная спин-блокировка и MCS-блокировка являются более масштабируемыми, но при этом MCS-блокировка может быть неэффективна при малом количестве потоков. Основные проблемы масштабируемости связаны с синхронизацией кэш-памяти между ядрами процессора.

Общей проблемой всех спин-блокировок можно назвать большое время простоя ядер процессора в холостом цикле.

ГЛАВА 3. ИСПОЛЬЗОВАНИЕ ШАБЛОНА ПРОЕКТИРОВАНИЯ «АКТИВНЫЙ ОБЪЕКТ» ДЛЯ РЕШЕНИЯ ЗАДАЧИ ВЗАИМНОГО ИСКЛЮЧЕНИЯ

3.1 Общие черты реализаций шаблона проектирования Активный Объект

Рассматриваемый шаблон проектирования при реализации подразумевает формирование сложного объекта, состоящего из нескольких частей. Его общую организацию можно увидеть на рисунке 4.

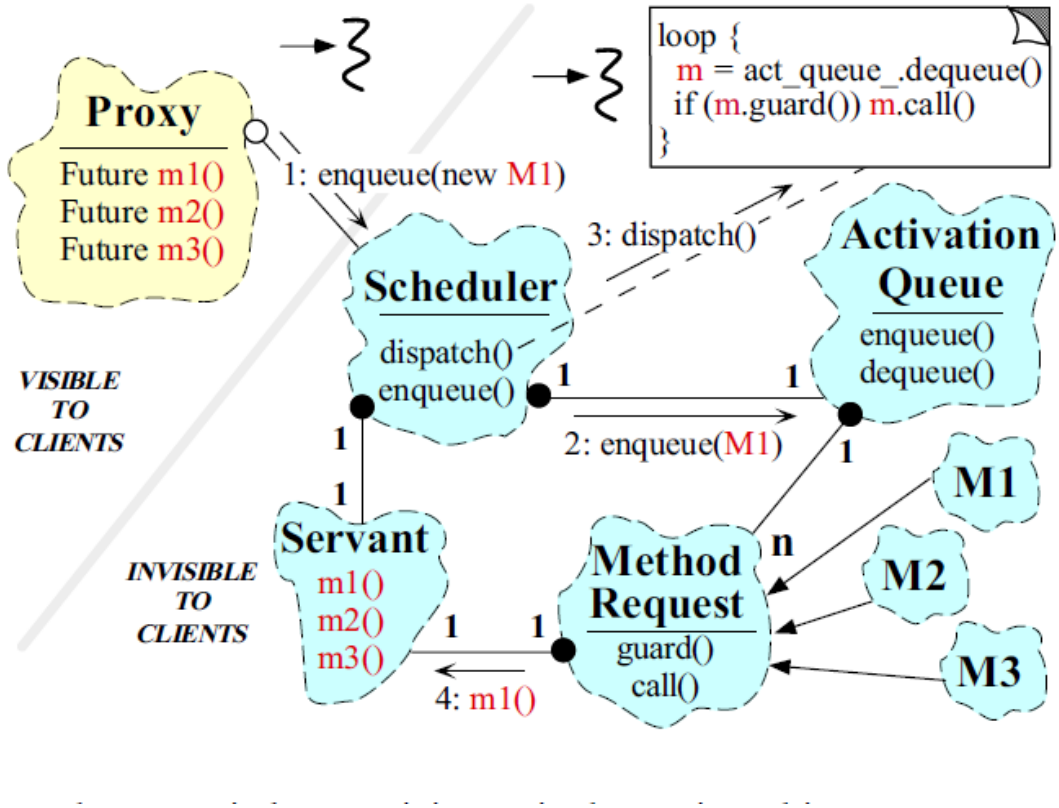

Рисунок 4 – Общая структура «Активного Объекта» [21]

В «Активном Объекте» выделяют шесть основных частей [21]:

1. Заместитель (от англ. Proxy)
2. Запрашиваемый метод (от англ. Method Request)
3. Очередь активации (от англ. Activation Queue)
4. Планировщик (от англ. Scheduler)

5. Обслуживающий поток (от англ. Servant)
6. Будущий результат (от англ. Future)

Заместитель используется для предоставления клиентам интерфейса, который позволит использовать стандартизированный способ для добавления задач, выполняемых «Активным Объектом».

Запрашиваемый метод является абстракцией для представления задачи, передаваемой через Заместителя. В таком объекте обычно передается код, который должен быть выполнен в «Активном Объекте» и параметры, необходимые для этого.

Очередь активации хранит в себе все Запрашиваемые методы в ожидании их выполнения. Тесно с ней взаимодействует Планировщик. Он по определенному принципу выбирает, как из задач в Очереди активации будет выполнена следующей. Дополнительно, планировщик может проверять, готова ли задача к выполнению (например, освобождено место в структуре, выделены ресурсы и прочее), если функция для проверки была определена в Запрашиваемом методе.

Слуга как правило является одним или несколькими потоками, которые выполняют код Запрашиваемого метода. Интерфейс Будущего результата является опциональным, в случае если клиенту требуется узнать результат выполнений Запрашиваемого метода.

Все взаимодействия при использовании «Активного Объекта» можно разделить на три фаза, изображенные на диаграмме последовательности на рисунке 5.

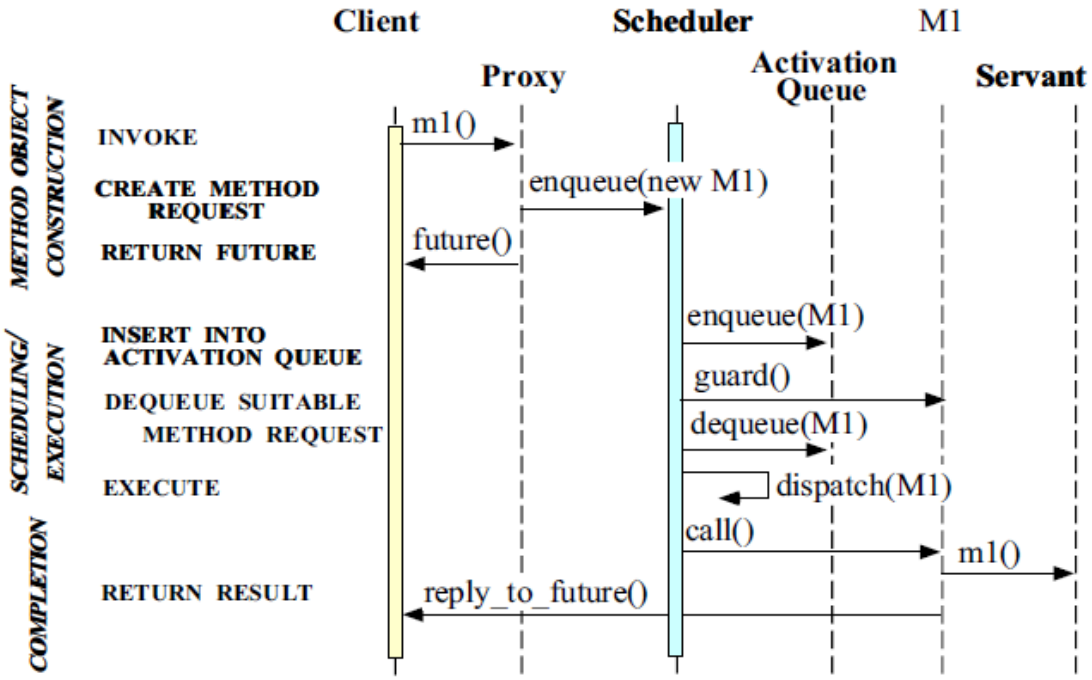

Рисунок 5 – Диаграмма последовательности для «Активного Объекта» [21]

Первая фаза начинается с создания клиентом Запрашиваемого метода. После этого созданный метод через Заместителя передается Планировщику, который помещает его в Активационную очередь. Если клиент хочет узнать результат выполнения своей задачи, то ему следует вернуть объект Будущего результата.

На второй фазе часть Планировщика, выполняемая потоком внутри «Активного Объекта», выбирает из Активационной Очереди Запрашиваемые методы, которые готовы к выполнению. Если такой метод был обнаружен, то он удаляется из очереди и передается Слуге для выполнения.

На последней фазе в объект Будущего результата помещается возвращаемое значение, после чего клиент может получить к нему доступ. В дальнейшем, ресурсы выделенные на Запрашиваемый метод и Будущий результат могут быть освобождены.

Стоит отметить, что в существующих реализациях некоторые компоненты «Активного Объекта» претерпевают серьезные изменения. Так, например, иногда происходит распределение функций Планировщика между Заместителем и самой Активационной очередью. Другой пример – за формирование Запрашиваемого метода может отвечать Заместитель, но в некоторых случаях предоставляется отдельный интерфейс, который используется клиентами.

3.2. Варианты механизма получения результата при использовании шаблона проектирования «Активный Объект»

Рассмотрим отдельно способы получения результата выполнения Запрашиваемого метода в ситуациях, когда он интересует поток-клиент.

По способу ожидания механизмы получения результата можно разделить на следующие:

1. Синхронный
2. Асинхронный

Первый способ приводит к тому, что поток после добавления задачи переходит в бесконечное (или же ограниченное некоторым таймером) ожидание возвращения результата. Данный способ не даёт какой-либо принципиальной выгоды по сравнению с блокирующими примитивами синхронизации, тем не менее может использоваться в определенных ситуациях, когда у потока закончилась полезная работа.

Альтернативным подходом является асинхронное взаимодействие. В этом случае поток не ожидает завершения вычислений в «Активном Объекте», а продолжает выполнять полезную работу, имея объект Будущего Результата. В дальнейшем, если полезная работа закончится, или потоку потребуется результат операции, выполняемой в «Активном Объекте», поток обратится к объекту Будущего Результата. Если задача уже выполнена – он получит результат, в противном случае произойдет переход к синхронному механизму.

Как можно увидеть из сказанного выше, при любом способе ожидания результата требуется иметь некоторый механизм синхронизации, который не позволит потоку прочитать результат до того, как он будет получен в «Активном Объекте». Можно выделить два основных варианта блокировок, которые можно использовать:

1. Блокировка на бинарном семафоре
2. Блокировка в ожидании условия

Для блокировки в первой варианте реализации шаблона «Активный Объект» может использоваться обычная спин-блокировка или мьютекс – он блокируется, когда создаётся задача, и освобождается, когда задача выполнена. Поток, ожидающий результат, попробует захватить примитив синхронизации: в случае успеха он сможет забрать результат, иначе будет ожидать в холостом цикле, или же воспользуется системным вызовом.

Во втором случае будет использоваться такой примитив синхронизации как условная переменная. Серьезных отличий во взаимодействии от ранее описанного вариант наблюдаться не будет. Можно отметить лишь более удобный синтаксис.

Также для получения результата можно использовать высокоуровневый примитивы, которые принято называть Future/Promise. Их реализации присутствуют в различных библиотеках для языка C++, а также в стандартной библиотеке языка C++, начиная со стандарта C++11.

3.3 Реализация шаблона проектирования «Активный Объект»

В данной работе будут рассмотрены три варианта реализации шаблона проектирования «Активный Объект»: реализация в фреймворке ACE (The ADAPTIVE Communication Environment), реализация в библиотеке GCD (Grand Central Dispatch / libdispatch) и собственная реализация данного шаблона.

3.3.1 Реализация шаблона «Активный Объект» в библиотеке ACE

ACE является программным фреймворком с открытым исходным кодом, используемым в сетевом взаимодействии. Во многих случаях обработка приходящих по сети пакетов может быть связана с необходимостью распараллелить работу приложения. Для этих целей в ACE существуют различные средства, в том числе и своя реализация шаблона проектирования «Активный Объект» [22]. Фреймворк полностью написан на языке C++.

Основным классом для формирования «Активного Объекта» в ACE является ACE::ACE_Task, который инкапсулируют в себя ACE::ACE_Message_Queue.

Экземпляр класса ACE::ACE_Task представляет из себя объект, которые выступает в роли Слуги, выполняющего переданные задачи. Стоит отметить, что при запуске такого объекта (осуществляется с помощью метода ACE::ACE_Task::activate()), можно указать количество потоков, которое будет выполнять задачи внутри «Активного Объекта». Тем не менее, в данной работе рассматриваются только «Активные Объекты» с одним потоком внутри (в силу требования безопасности при решении задачи взаимного исключения).

Объект класса ACE::ACE_Message_Queue выполняется функции Очереди Активации и Планировщика. В частности, очередь может быть настроена на различные системы приоритетов. Данная очередь имеет два варианта использования – с помощью параметра ACE_SYNCH_DECL эта очередь может быть как синхронизируемой (используется мьютекс для доступа к ней), так и потоконебезопасной.

Роль Заместителя выполняют такие методы класса ACE::ACE_Task, как putq() или put_next(). В качестве Запрашиваемого метода используется класс ACE::ACE_Message_Block. Данный класс представляет из себя некоторый блок «сырых» данных, который может быть помещен в ACE::ACE_Message_Queue. Проблемой использования данной реализации является необходимость

конвертации запрос в ACE::ACE_Message_Block, а затем обратно внутри «Активного Объекта» для выполнения задачи.

Альтернативным подходом является работа с классом ACE::ACE_Task_base, который является базовым по отношению к ACE::ACE_Task. Он предоставляет те же функции и методы, за исключением того, что в него еще не инкапсулирована никакая очередь. В таком случае можно воспользоваться ACE::ACE_Activation_Queue. Данная очередь строится на синхронизируемой версии ACE::ACE_Message_Queue и позволяет работать с наследниками класса ACE::ACE_Method_Request.

ACE::ACE_Method_Request инкапсулирует в себе все необходимые методы для описания Запрашиваемого метода. Очередь ACE::ACE_Activation_Queue имеет все необходимые методы для конвертации ACE::ACE_Method_Request в ACE::ACE_Message_Block.

Также стоит добавить, что может быть описан и произвольный планировщик для работы с очередью.

Получение результатов выполнения операции в «Активном Объекте» можно реализовать с помощью объекта класса ACE::ACE_Future. Тем не менее, возможным является и вариант написания произвольного способа доступа к результату, путем добавления методов к реализации класса ACE::ACE_Method_Request.

3.3.2 Реализации шаблона «Активный Объект» в библиотеке GCD

Библиотека libdispatch разработана компанией Apple и распространяется по лицензии Apache. Эта библиотека реализует параллелизм задач, позволяя разработчику не заниматься решением задачи менеджмента потоков. Данная библиотека реализована на языке C и имеет соответствующий программный интерфейс [23].

Одним из механизмов, представляемых этой библиотекой, являются очереди. Всего можно выделить три вида очередей:

1. Основная очередь отправки
2. Конкурирующие очереди (от английского concurrent queue)
3. Последовательные очереди (от английского serial queue)

Основная очередь отправки обычно связывается с циклом выполнения в основном потоке приложения и не будет рассматриваться в данной работе.

Конкурирующие очереди отличаются от последовательных количеством потоков, параллельно выполняющих задачи, которые были добавлены в очередь. Для конкурирующих очередей количество потоков является произвольным (оно зависит от различных параметров, таких как количество доступных ядер, количество других активных очередей и другие). В последовательной очереди задачи выполняет всегда один поток.

Для создания последовательной очереди используется функция dispatch_queue_create().

Для добавления задач в очередь существует 4 функции, которые можно квалифицировать по двумя признакам. Во-первых, по способу ожидания результата:

1. Синхронно (dispatch_sync() и dispatch_sync_f())
2. Асинхронно (dispatch_async() и dispatch_async_f())

Различие данных способов описано в предыдущем разделе. Стоит отметить, что рекомендуемым способом добавления задач в последовательную очередь является асинхронный, так как синхронные вызовы прекращают выполнение текущего потока до момента выполнения задачи. Также попытка вызвать dispatch_sync() из задачи, которая уже выполняется синхронно, может привести к взаимоблокировке.

Другой признак – способ представления выполняемой задачи.

1. Функция (передается указатель на функцию, которая должна выполняться внутри «Активного Объекта»)
2. Блок (нестандартное расширение языка C)

Блок по сути является так называемой анонимной функцией (или лямбда-функцией). Данный синтаксис не поддерживается стандартом языка C, поэтому в данной работе рассматриваться не будет. Тем не менее, использование данной библиотекой блоков приводит к тому, что при необходимости сборки библиотеки libdispatch из исходного кода под целевую платформу приходится использовать компилятор clang (так как компилятор gcc не поддерживает указанное расширение).

Функции dispatch_sync_f() и dispatch_async_f() принимают три аргумента – очередь, в которую будет добавлена задача; контекст, который будет передан как аргумент в задачу; задача, представленная в виде указателя на функцию, которая будет выполняться в «Активном Объекте».

Последовательные очереди в библиотеки libdispatch не имеют никакого механизма приоритезации задач в очереди, как и механизма возврата результата выполненной в «Активной Объекте» операции.

3.3.3 Внутреннее устройство реализации шаблона «Активный Объект» на основе MPSC-очереди

Для собственной реализации шаблона «Активный Объект» будут использоваться следующие компоненты (см. приложение В).

Функции Заместителя будет выполнять класс ActiveObject, предоставляющий функции для взаимодействия внешнего мира с «Активным Объектом».

Запрашиваемый метод будет реализован с помощью класса Task. Объекты этого класса будут храниться в специальной очереди. Формировать каждый объект с заданием потоки будут самостоятельно, вызывая его конструктор.

Очередь активации и Планировщик будут объединены в структуре mpscq_t. Данная структура является реализацией MPSC-очереди. Рассмотрим ее устройство подробнее [24].

Название данной очереди расшифровывается как Множество-Поставщиков-Один-Обработчик (Multiple-Producers-Single-Consumer)

Структура очереди хранит два указателя: на голову очереди (данный указатель запрещено оптимизировать компилятору) и на ее хвост.

Данная очередь является связанным списком, поэтому каждый элемент хранит указатель на следующий элемент в очереди (next) и указатель на значение, связанное с данным узлом (state).

При создании этой очереди в нее всегда добавляется специальный узел-заглушка.

Для очереди определены две операции:

1. Добавление объекта в начало очереди (функция mpscq_push())
2. Получение объекта из конца очереди (функция mpscq_pop())

Первая операция реализуется с помощью атомарной операции XCHG – требуется заменить указатель на голову очереди на указатель на добавляемый объект. После требуется восстановить связанный объект, установив связь между старым объектом в голове очереди и новым, через указатель в поле next.

Атомарная операция XCHG реализована с помощью специальной встроенной атомарной функции компилятора GCC __sync_lock_test_and_set() [25]. Данная операция должна быть атомарной, так как сразу несколько потоков могут пытаться добавить узел в очередь.

Сложнее реализуется операция извлечения элемента. Поскольку только один поток может забирать элементы из очереди, то это можно сделать без использования блокировок или каких-либо атомарных операций.

В переменные сохраняется указатель на хвост очереди (tail), который всегда указывает на узел-заглушку, и указатель на следующий после него элемент (next). Если next равен нулю, значит в очереди нет других элементов кроме заглушки, и можно считать ее пустой.

Если же такой элемент есть, то хранимый в нем указатель на связанное с узлом значение копируется в узел-заглушку, после чего возвращается указатель на узел-заглушку. Также восстанавливается связанный список, путем изменения указателя на хвост очереди – он заменяет на указатель next, а узел по его адресу становится новым узлом-заглушкой.

Можно выделить некоторые преимущества данной очереди:

1. Является неблокирующей очередью без ожидания
2. Операция извлечения элемента всегда выполняется за O(1) и не требует блокировок или атомарных операций

К минусам можно отнести отсутствии у данной реализации свойства линеаризуемости и невозможность модификации алгоритма для работы с приоритетами.

Тем не менее данная очередь подходит для работы внутри простой реализации «Активного Объекта».

Следующий компонент шаблона проектирования «Активный Объект» – Обслуживающий поток представлен также классом Active_Object. Он выполняет метод Active_Object::call_next(), который в бесконечном цикле пытается прочитать значение из внутренней MPSC-очереди. В случае успеха происходит обработка задачи, в случае получения специальной терминирующей задачи – выход из цикла и завершение работы потока. Если очередь оказалась пуста, то по аналогии с описанным в первой главе холостым циклом, происходит вызов инструкции pause (через макрос __mm_pause()).

Последним компонентом является Будущее значение. В данной реализации шаблона проектирования «Активный Объект» все задачи выполняются асинхронно. В случае необходимости получить результат можно обратиться к методу result() класса Task. После чего поток или получить результат выполнения задачи, или будет заблокирован на спин-блокировке (на основе сказанного ранее была выбрана билетная спин-блокировка) в ожидании получения результата.

3.4 Описание разработанных классов и структур для представления «Активного Объекта»

Для представления шаблона проектирования «Активный Объект» были написаны классы-обертки и структуры при использовании различных реализаций.

При работе с фреймворком ACE были описаны следующие классы:

1. класс Job
2. класс Task

Класс Job наследует от класса ACE::ACE_Method_Request, содержит в себе реализацию его методов. С точки зрения шаблона проектирования «Активный Объект» он выполняет функцию Запрашиваемого метода.

Класс Task наследует от класса ACE::ACE_Task_Base, добавляя в него ACE::ACE_Activation_Queue и методы для работы с ней. Эта связка реализует Очередь Активации, Планировщик и частично Заместителя. Создание потока внутри «Активного Объекта» выполняется средствами самого фреймворка ACE с помощью метода Task::activate(), унаследованного от базового класса.

Ключевым элементом «Активного Объекта» является метод Task::svc(), который представляет собой бесконечный цикл получения элементов из очереди и выполнения функции, указатель на которую был передан (также передаются и аргументы). В случае если указатель равен nullptr, то считается, что было отправлено специальное терминирующее задание, предписывающее «Активному Объекту» завершить свое выполнение.

Поскольку библиотека libdispatch имеет программный интерфейс на языке C, то и все необходимые структуры для взаимодействия описаны на том же языке без использования объектно-ориентированной модели.

«Активный Объект» создаётся с помощью вызова функции dispatch_queue_create(). Для работы с полученной очередью (представляет Активационную очередь и Планировщик в терминах шаблона проектирования) описана структура task (реализует Запрашиваемый метод). Она имеет поля для хранения указателя на функцию, аргументов, а также для реализации механизма возвращения результата, которым будет подробнее описан в дальнейшем. Дополнительно описана функция queue_func(). Эта функция будет выполняться каждый раз, когда какое-либо задание будет извлечено с вершины очереди. Внутри функции queue_func() происходит интерпретация полей структуры task с последующим вызовом целевой функции.

После завершения выполнения всех операций внутри «Активного Объекта» вызывается функция dispatch_release(). Данная функция уменьшает количество существующих ссылок в системе на указанную очередь. Если число ссылок достигнет нуля, то все ресурсы, связанные с очередью, будут освобождены. Стоит отметить, что если выполнить dispatch_release() в тот момент, когда внутри «Активного Объекта» продолжает асинхронно выполняться некоторая задача, то это приводит к неопределенному поведению – очередь может прекратить свое существование до того, как задача будет полностью выполнена.

Обратное действие по отношению к функции dispatch_release() выполняет функция dispatch_retain().

Для собственной реализации шаблона проектирования «Активный Объект» в качестве Активационной очереди использовалась ранее описанная MPSC-очередь, представленная структурами mpscq_t для описания самой очереди и mpscq_node_t для описания ее узлов.

Также используются два класса: Active_Object и Task. В данном случае класс Task служит для представления Запрашиваемого метода, инкапсулирует в себе все необходимые поля и реализует наиболее важные методы.

Все остальные роли выполняет класс Active_Object, который инкапсулирует в себе структуру типа mpscq_t для представления очереди.

Для завершения работы «Активного Объекта» в очередь добавляется специальная задача, в которой вместо указателя на функцию, которую требуется выполнить, передается nullptr.

Также для всех реализаций был описан класс (структура для GCD) Spinlock, инкапсулирующий операции над билетной спин-блокировкой. Данный класс используется для синхронизации при получении Будущего результата.

Получение Будущего результата реализовано во всех реализациях приблизительно одинаково. Объект задачи имеет два дополнительных поля – переменную, представляющую результат выполнения, и переменную спин-блокировки для защиты результата.

В момент формирования задачи спин-блокировка переводится в состояние захваченной. После добавления задачи в очередь поток может выполнять другую полезную работу. Как только ему потребуется непосредственный результат, он сможет обратиться к методу result() класса, представляющего Запрашиваемый метод. Внутри этого метода поток попытается захватить спин-блокировку. Если это действие будет успешно, то поток сможет забрать результат. В противном случае поток ожидает в холостом цикле.

Освобождение этой спин-блокировки осуществляется внутренним потоком «Активного Объекта» после того, как он выполнил функцию, переданную через Запрашиваемый метод.

3.5 Архитектура тестового приложения

Для сравнения различных механизмов решения задачи взаимного исключения (обычных блокирующих примитивов синхронизации и реализованных на основе шаблона проектирования «Активный Объект») были подготовлены и оптимизированы их различные реализации.

Со стороны блокирующих примитивов синхронизации была выбрана билетная спин-блокировка. Это связано с несколькими ее достоинствами, полученными на основе информации, полученной в прошлом эксперименте (см. Глава 2):

1. Простота реализации
2. Выполнение условия справедливости
3. Достаточная масштабируемость при небольшом числе потоков
4. Отсутствие высоких накладных расходов при выполнении быстрых задач в критической секции.

Как было рассмотрено ранее, шаблон проектирования «Активный Объект» имеет большое количество преимуществ над обычными блокирующими примитивами. Однако, использование «Активного Объекта» приводит к нежелательным накладным расходам, которые в определенной ситуации могут негативно сказаться на работе приложения.

Для оценки эффективности использовалась следующая характеристика: оперативность приложения при фиксированном количестве потоков (по аналогии с экспериментом на спин-блокировках). Проверка выполнения условия справедливости в данном случае не является целесообразной, так как и «билетная» спин-блокировка, и примитивы синхронизации на основе «Активного Объекта» выполняют это условие.

Было разработано тестовое приложение, схожее с представленным во второй главе. Общий вид функции, выполняемой рабочими потоками представлен в листинге 3.

```
void thread_function (args args *arg arg){
Active_Object *ao = arg arg->ao ;
double res = 0.0;
while (arg->thr_count){
Task temp(arg->f, arg->a);
ao ->add_task (&temp);
res = temp.result();
arg->thr_count-- ;
}
}
```

Листинг 3 – Общий вид функции, выполняемой потоками

Стоит отметить основные отличия данной функция. Каждый поток получает информацию о требуемом количестве итераций и указатель на «Активный Объект». Затем используется асинхронный выполнение задачи в «Активном Объекте» с помощью метода add_task(). Тем не менее, в данном тесте не имеет значения, выполняет ли поток полезную нагрузку вне критической секции, поэтому потом сразу переходит к синхронному ожиданию результата посредством вызова метода result().

С помощью параметров arg->f и arg->a передаются соответственно указатель на функцию и на ее аргументы. В данном тесте также использовалась функция факториала. Поскольку возникла ситуация, когда могут использоваться различные компиляторы, то функция вычисления факториала была вынесена из тестов и скомпилирована в отдельную статическую библиотеку, чтобы избежать влияния оптимизаций на различных тестах.

Также важно отметить, что для точности эксперимента для всех реализаций использовались POSIX-потоки, представленные в библиотеки pthreads, хотя, например, фреймворк ACE предоставляет собственную реализацию.

Для ожидания возвращаемого результата во всех реализациях использовалась билетная спин-блокировка.

Аналогично предыдущему эксперименту использовался компьютер с процессором Intel Core i7-2630QM 2,0 ГГц и отключенной технологией Turbo Boost. Дополнительно, у него был отключен Hyper-threading ввиду отсутствия необходимости работы с большим количеством потоков.

Замеры времени осуществлялись с помощью TSC-таймеров (ассемблерная инструкция rdtsc).

Для тестов не использовались большие значения факториала, поскольку ожидается, что «Активный Объект» в большей степени рассчитан на работу со структурами в критической секции, либо выполнение небольшого участка кода.

3.6 Анализ полученных результатов тестирования

По результатам исследования были получены результаты (см. Таблицу Г.1 в приложении Г), представленные на рисунке 6. Также на нем отмечен разброс результатов измерения с доверительной вероятностью 95% (только там, где позволяет масштаб рисунка).

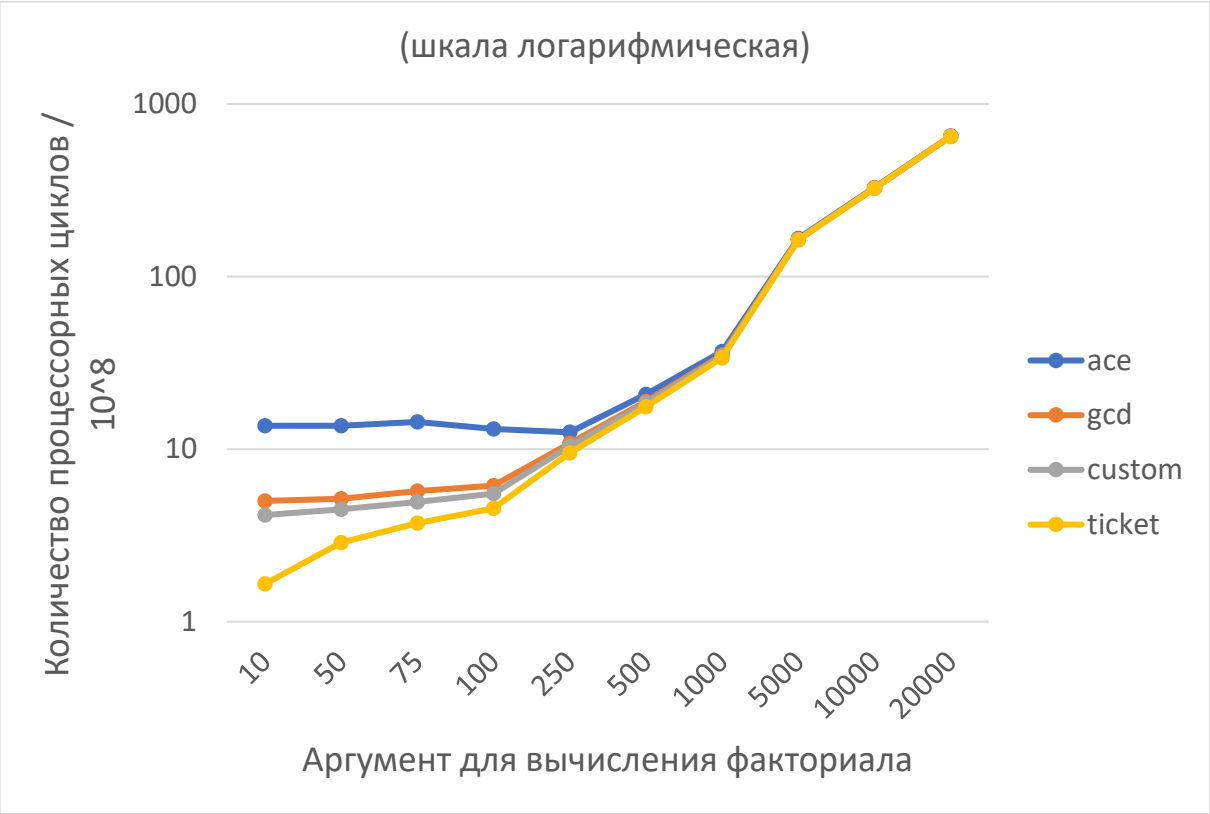

Рисунок 6 – Результаты эксперимента с «Активным Объектом»

На основе полученных результатов можно сделать выводы о том, что для малых критических секций, вычисление в которых проходит очень быстро, «билетная» спин-блокировка демонстрирует несколько лучшую производительность, чем реализации шаблона проектирования «Активный Объект». Тем не менее, в определенных ситуациях, такие накладные расходы могут быть допустимы.

В тоже время, с определенного момента (значение факториала 500, N процессорных циклов на вычисление) накладные расходы от использования

«Активного Объекта» становятся не существенны. Следовательно, шаблон «Активный Объект» позволяет эффективно решать задачу взаимного исключения.

Стоит выделить отдельно низкую эффективность реализации «Активного Объекта» в ACE. Это связано с блокирующей природой его Активационной Очереди – она не способна справляться с большим количеством быстрых задач.

3.7 Выводы по главе 3

В главе 3 было подробно разобрано внутреннее устройство шаблона проектирования «Активный Объект».

Его основные компоненты: Заместитель, Активационная Очередь, Запрашиваемый метод, Возвращаемое значение.

Также были разобраны наиболее известные реализации шаблона проектирования «Активный Объект».

В фреймворке ACE «Активный Объект» представляется с помощью классов ACE::Task_Base и ACE::Activation_Queue. Стоит отметить, что Активационная Очередь в ACE представлена классической очередью, операции с которой не являются потокобезопасными, поэтому данная очередь защищается с помощью мьютекса.

Другая реализация относится к библиотеке GCD. Данная библиотека позволяет создавать специальные очереди, с которыми можно взаимодействовать как синхронно, так и асинхронно. В библиотеке активно используются нестандартные расширения языка C (например, блоки). Это приводит к потенциальным проблемам с компиляцией приложений, использующих эту библиотеку.

Дополнительно была разработана собственная реализация шаблона проектирования «Активный Объект», которая использует специально оптимизированную MPSC-очередь для быстрого доступа потока-слуга к задачам.

Минус такой реализации – отсутствие возможности ввести систему приоритетов задач.

Для сравнения эффективности различных реализаций шаблона «Активный Объект» и оценки влияния накладных расходов на работу приложения было разработано тестовое приложение.

По результатам выполнения тестов было показано, что реализация из фреймворка ACE может иметь невысокую эффективность при малых критических секциях. В тоже время при критических секциях, выполняющихся дольше приблизительно N процессорных циклов, накладные расходы перестают влиять на общее время работы приложения.

Таким образом, можно сделать вывод о применимости шаблона проектирования «Активный Объект» для реализации механизма взаимного исключения.

ЗАКЛЮЧЕНИЕ

В результате проделанной работы были решены следующие задачи:

1. Проведен обзор различных механизмов взаимного исключения, таких как мьютексы, различные варианты спин-блокировок, пояснено как может быть использован шаблон проектирования «Активный Объект» для решения этой задачи
2. Проведен подробный анализ различных спин-блокировок, подробно описано их внутреннее устройство и детали реализации; разобраны реализации шаблона проектирования «Активный Объект», указаны его основные компоненты, отражены отличия его реализаций в наиболее известных библиотеках и фреймворках.
3. Разработана статическая библиотека спин-блокировок; также разработаны оболочки для использования шаблона «Активный Объект» и представлена собственная реализация на основе оптимизированной MPSC-очереди.
4. Разработаны тестовые приложения, позволяющие без перекомпиляции проводить различные тесты с использованием написанной библиотеки и оболочек (можно варьировать потоки, длительность выполнения задач, вид примитива синхронизации). Дополнительно разработаны скрипты для командной оболочки, управляющие выполнением тестов.
5. Проведен анализ полученных в ходе тестирований результатов, определены ситуации, в которых использование одних примитивов синхронизации, может быть выгоднее, чем других.

В результате можно сделать следующие выводы:

1. При использовании спин-блокировок следует учитывать, что наиболее быстрые реализации могут не гарантировать выполнение условия справедливости (POSIX-блокировка) или плохо масштабироваться при

увеличении количества потоков, взаимодействующих с критической секцией («билетная» спин-блокировка).

2. MCS-блокировка является хорошо масштабируемой спин-блокировкой, тем не менее она имеет большие накладные расходы на обслуживание внутренней очереди.
3. «Активный Объект» имеет ряд преимуществ над блокирующими примитивами синхронизации (например, позволяет выполнять работу в критической секции асинхронно, гарантирует справедливость и масштабируемость, имеет существующие реализации в популярных фреймворках и библиотеках).
4. Ключевые недостатки «Активного Объекта» - необходимость на каждую критическую секцию создавать отдельный поток и возможные накладные расходы на реализацию его компонентов.
5. Шаблон проектирования «Активный Объект» может эффективно использоваться для решения задачи взаимного исключения, накладные расходы от его использования могут быть приемлемы в большинстве задач.
6. Различные реализации «Активного Объекта» влияют на его эффективность – оптимизация Активационной Очереди как правило не позволяет реализовать механизмы приоритета и планирования.

Дальнейшее развитие темы возможно по следующим направлениям:

1. Сравнение реализация шаблона «Активный Объект» с другими примитивами синхронизации, такими как мьютекс, адаптивный мьютекс или неблокирующими алгоритмами.
2. Исследование применимости шаблона проектирования «Активный Объект» к реализации блокировок чтения-записи.
3. Исследование возможности использования механизмов планирования в шаблоне проектирования «Активный Объект» для работы со структурами данных.

4. Дополнительные тестирования с использованием более сложных приложений, подразумевающих использование нескольких критических секций одновременно.

СПИСОК ЛИТЕРАТУРЫ

1. Столлингс В. Операционные системы. М.: Вильямс, 2013. 848 с.
2. Lamport, L. A new solution of Dijkstra's concurrent programming problem // Communications of the ACM, 1974. V. 17 N. 8. P. 453–455. doi:10.1145/361082.361093
3. Dijkstra E.W. Co-operating Sequential Processes // Programming Languages / Ed F. Genuys. New York: Academic Press, 1968. P. 43-112.
4. Dijkstra E.W. Solution of a problem in concurrent programming control // Communications of the ACM. 1983. V. 26 N. 1. P. 21-22. doi: 10.1145/357980.357989
5. Franke H., Russell R., and Kirkwood M., Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux // Ottawa Linux Summit, 2002.
6. Drepper U. Futexes Are Tricky [Электронный ресурс] // Компания RedHat: [сайт], 2011. URL: http://people.redhat.com/drepper/futex.pdf (дата обращения 22.04.2018).
7. Тараканов Д. С. Сравнение примитивов синхронизации в многопоточных приложениях // Сборник трудов VIII научно-практической конференции молодых ученых «Вычислительные системы и сети (Майоровские чтения)». 2017. С. 47—49.
8. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2017.
9. Scott M. L., Scherer W. N. Scalable queue-based spin locks with timeout // SIGPLAN Notices (ACM Special Interest Group on Programming Languages). 2001. V. 36. N. 7. P. 44-52.
10. Boyd-Wickizer S., Kaashoek M.F., Morris R., Zeldovich N. Non-scalable locks are dangerous // Proceedings of the Linux Symposium. 2012. P. 119-130.

11. Anderson T.E. The performance of spin lock alternatives for shared-money multiprocessors // IEEE Transactions on Parallel and Distributed Systems. 1990. V. 1. N. 1. P. 6-16. doi: 10.1109/71.80120
12. Mellor-Crummey J.M., Scott M.L. Algorithms for scalable synchronization on shared-memory multiprocessors // ACM Transactions on Computer Systems (TOCS). 1991. V. 9. N. 1. P. 21-65. doi: 10.1145/103727.103729
13. Luchangco V., Nussbaum D., Shavit N. A hierarchical CLH queue lock // Lecture Notes in Computer Science. 2006. V. 4128. P. 801-810.
14. J. Appavoo J., Auslander M., Butrico M., da Silva D.M., Krieger O., Mergen M.F., Ostrowski M., Rosenburg B., Wisniewski R.W., Xenidis J. Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel // IBM Systems Journal. 2005. V. 44. N. 2. P. 427-440.
15. Michael M.M., Scott M.L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms // Proceedings of the Annual ACM Symposium on Principles of Distributed Computing. 1996. P. 267-275. doi: 10.1145/248052.248106
16. Вьюков Д. Lockfree, Waitfree, Obstruction-free, Atomic-free Algorithms [Электронный ресурс] // 1024cores [сайт], 2016. URL: http://www.1024cores.net/home/in-russian/lock--wait--obstruction--atomic-free-algorithms (дата обращения 22.04.2018)
17. Хижинский М. Lock-free структуры данных. 1 — Начало [Электронный ресурс] // Хабр [сайт], 2013. URL: https://habr.com/post/195770/ (дата обращение 22.04.2018)
18. Liblock [Электронный ресурс] // Github: [сайт], 2017. URL: https://github.com/NaglijLamer/liblock (дата обращения: 22.04.2018).
19. Intel 64 and IA-32 Architectures Software Developer’s Manual, 2017
20. Dawson B. Rdtsc in the Age of Sandybridge [Электронный ресурс] // Random ASCII [сайт], 2011. URL:

https://randomascii.wordpress.com/2011/07/29/rdtsc-in-the-age-of-sandybridge/ (дата обращения: 22.04.2018)

21. Lavender R.G., Schmidt D.C. Active Object: an Object Behavioral Pattern for Concurrent Programming, in Pattern Languages of Program Design (J.O. Coplien, J. Vlissides, and N.Kerth, eds.), Reading // MA:Addison-Wesley, 1996
22. ACE Documentation [Электронный ресурс] // Distributed Object Computing (DOC) Group for Distributed Real-time and Embedded (DRE) Systems [сайт], 2018. URL: http://www.dre.vanderbilt.edu/Doxygen/6.4.8/html/libace-doc/index.html (дата обращения 22.04.2018)
23. Dispatch Queues [Электронный ресурс] // Apple Developer Support [сайт], 2015. URL: https://developer.apple.com/library/archive/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html (дата обращения 22.04.2018)
24. Вьюков Д. Non-intrusive MPSC node-based queue [Электронный ресурс] // 1024cores [сайт], 2016. URL: http://www.1024cores.net/home/lock-free-algorithms/queues/non-intrusive-mpsc-node-based-queue (дата обращения 22.04.2018)
25. Built-in functions for atomic memory access [Электронный ресурс] // GCC online documentation [сайт], 2016. URL: https://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Atomic-Builtins.html (дата обращения 22.04.2018)

ПРИЛОЖЕНИЕ А

Примеры исходного кода разработанной библиотеки спин-блокировок.

Файл ticket_spin.h

```
#ifndef _TICKET_SPINLOCK_H
#define _TICKET_SPINLOCK_H
#include "custom_lock.h"
#include "global_metric.h"
#include "metric_function.h"
typedef custom_lock ticket_spinlock_t;
int ticket_spin_init(ticket_spinlock_t *lock, int locked);
int ticket_spin_lock(ticket_spinlock_t *lock);
int ticket_spin_unlock(ticket_spinlock_t *lock);
#endif
```

Файл ticket_spin.c

```
#include "ticket_spin.h"
int ticket_spin_lock(ticket_spinlock_t *lock){
	int ign;
	__asm __volatile(
	"lock; xaddl %0, %1\n\t"
	"jmp 2f\n\t"
	"1:\tpause\n\t"
	"2:\tcmpl %0, 8%1\n\t"
	"jne 1b\n\t"
	: "=a" (ign), "=m" (*lock)
	: "0" (1), "m" (*lock));
	return 0;
}
int ticket_spin_unlock(ticket_spinlock_t *lock){
	lock->serv++;
	return 0;
}
int ticket_spin_init(ticket_spinlock_t *lock, int locked){
	lock->serv = 0;
	lock->next = locked > 0? 1 : 0;
	return 0;
}
```

Файл MCS_spin.h

```
#ifndef _MCS_SPIN_H
#define _MCS_SPIN_H
#include <pthread.h>
#include "global_metric.h"
#include "metric_function.h"
#include "custom_lock.h"
struct MCS_node{
	MCS_node *next;
```

```
	int locked;
};
typedef custom_lock MCS_lock_t;
int MCS_spin_lock(MCS_lock_t *lock);
int MCS_spin_unlock(MCS_lock_t *lock);
int MCS_spin_lock2(MCS_lock_t *lock, MCS_node *node);
int MCS_spin_unlock2(MCS_lock_t *lock, MCS_node *node);
int MCS_spin_init(MCS_lock_t *lock, int locked);
#endif
```

Файл MCS_spin.c

```
#include "MCS_spin.h"
#include <malloc.h>
int MCS_spin_lock2(MCS_lock_t *lock, MCS_node *I){
	MCS_node *pred;
	long long ign0;
	I->locked = 0;
	__asm __volatile(
		"movq %q4, (%1)\n\t"
		"movq %1, %3\n\t"
		"lock xchgq %3, 8(%0)\n\t"
		"testq %3, %3\n\t"
		"jz 3f\n\t"
		"movq $1, 8(%1)\n\t"
		"movq %1, (%3)\n\t"
		"jmp 2f\n\t"
		"1:\tpause\n\t"
		"2:\tcmpl %4, 8(%1)\n\t"
		"jnz 1b\n\t"
		"3:"
	: "=D" (lock), "=S" (I), "=d" (ign0), "=c" (pred)
	: "2" (0), "S" (I), "D" (lock));
	return (0);
}
int MCS_spin_unlock2(MCS_lock_t *lock, MCS_node *I){
	long long ign0;
	__asm __volatile(
		"cmpq %q3, (%1)\n\t"
		"jne 1f\n\t"
		"movq %1, %%rax\n\t"
		"lock; cmpxchgq %q3, 8(%0)\n\t"
		"jz 2f\n\t"
		"3:\tpause\n\t"
		"cmpq %q3, (%1)\n\t"
		"je 3b\n\t"
		"1:\tmovq (%1), %%rax\n\t"
		"movq %q3, 8(%%rax)\n\t"
		"2:"
	: "=D" (lock), "=S" (I), "=d" (ign0)
	: "2" (0), "S" (I), "D" (lock)
	: "rax");
```

```
	return 0;
}
int MCS_spin_init(MCS_lock_t *lock, __attribute__ ((unused)) int locked){
	lock->L = NULL;
	return 0;
}
```

Файл pthread_spin.h

```
#ifndef _SPIN_H
#define _SPIN_H
#include "custom_lock.h"
#include "global_metric.h"
#include "metric_function.h"
typedef custom_lock pthread_spinlock_m_t;
int pthread_spin_init_m (pthread_spinlock_m_t *lock, int ignore);
int pthread_spin_lock_m (pthread_spinlock_m_t *lock);
int pthread_spin_unlock_m (pthread_spinlock_m_t *lock);
#endif
```

Файл pthread_spin.c

```
#include "pthread_spin.h"
#include <pthread.h>
int pthread_spin_lock_m(pthread_spinlock_m_t *lock){
	int res = pthread_spin_lock(lock->spinlock_for_posix);
	return res;
}
int pthread_spin_unlock_m(pthread_spinlock_m_t *lock){
	return pthread_spin_unlock(lock->spinlock_for_posix);
}
int pthread_spin_init_m(pthread_spinlock_m_t *lock, int ignore){
	lock->spinlock_for_posix = (pthread_spinlock_t*)malloc(sizeof(pthread_spinlock_t));
	return pthread_spin_init(lock->spinlock_for_posix, ignore);
}
```

Файл gloval_metric.h

```
#if !defined(_SPINLIBRARY) && (defined(_METRIC) || defined(_GLOBAL_TIMER_LOCK))
#ifndef _GLOBAL_METRIC_H
#define _GLOBAL_METRIC_H
#include <time.h>
#include "common_defs.h"
#include <stdio.h>
#ifdef _SHORT_METRIC
#define OUTPUTGL "%f\n%lu\n"
#else
#define OUTPUTGL "Program execution time: %f\nCount of cycles: %lu\n"
#endif
struct timespec __start_time;
unsigned cycles_high, cycles_low, cycles_high_f, cycles_low_f;
int start_global_timer() __attribute__((constructor(102)));
int finish_global_timer() __attribute__((destructor(102)));
int start_global_timer(){
	clock_gettime(CLOCK_MONOTONIC, &__start_time);
```

```
	RDTSC_START(cycles_high, cycles_low);
	return 0;
}
int finish_global_timer(){
	struct timespec finish_time;
	RDTSC_FINISH(cycles_high_f, cycles_low_f);
	clock_gettime(CLOCK_MONOTONIC, &finish_time);
	uint64_t cycles = CALC_CYCLES(cycles_high_f, cycles_low_f) -
CALC_CYCLES(cycles_high, cycles_low);
	double time = TIMESPEC_TO_DOUBLE(finish_time) -
TIMESPEC_TO_DOUBLE(__start_time);
	fprintf(stderr, OUTPUTGL, time, cycles);
	return 0;
}
#endif
#endif
```

Файл common_defs.h

```
#ifndef _COMMON_DEFS_H
#define _COMMON_DEFS_H
#include <inttypes.h>
#define TIMESPEC_TO_DOUBLE(x) (1000.0 * x.tv_sec + 1e-6 * x.tv_nsec)
#define RDTSC_START(cycles_high, cycles_low) \
	__asm __volatile(                    \
	"cpuid\n\t"                          \
	"rdtsc\n\t"                          \
	"mov %%edx, %0\n\t"                  \
	"mov %%eax, %1\n\t"                  \
	: "=r" (cycles_high), "=r" (cycles_low) \
	:                                    \
	: "%rax", "%rbx", "%rcx", "rdx"      \
	)
#define RDTSC_FINISH(cycles_high, cycles_low) \
	__asm __volatile(                    \
	"rdtscp\n\t"                         \
	"mov %%edx, %0\n\t"                  \
	"mov %%eax, %1\n\t"                  \
	"cpuid\n\t"                          \
	: "=r" (cycles_high), "=r" (cycles_low) \
	:                                    \
	: "%rax", "%rbx", "%rcx", "rdx"      \
	)
#define CALC_CYCLES(cycles_high, cycles_low) (((uint64_t)(cycles_high) << 32) | (cycles_low))
#endif
```

Файл test_m.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <ctype.h>
```

```
#include <pthread.h>
#include <errno.h>
#include "custom_lock.h"
#define USAGE "Usage: app_name amount_of_threads total_locks iteractions_in_section
iteractions_between_sections"
#define FALSE 0
#define TRUE 1
#define SUCCESS 0
#define ERROR -1
#define SLEEP_TIME 1000000
#define error(msg) do{\
        perror(msg);\
        exit(EXIT_FAILURE);\
    }while(FALSE)\
custom_lock lock;
int count;
int critical_section_iterations;
int after_section_iterations;
void *thread_function(void *args){
    volatile double res;
    double m;
    #ifdef _MCS_SPIN
        MCS_node node;
    #endif
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET((long)args, &cpuset);
    if (sched_setaffinity(0, sizeof(cpuset), &cpuset) != 0){error("n"); return (void*)0;}
    while (TRUE){
    #ifdef _MCS_SPIN
        custom_lock_lock(&lock, &node);
    #else
        custom_lock_lock(&lock);
    #endif
        m = 1.0;
        for (int i = 1; i < critical_section_iterations; i++)
            m *= i;
        res = m;
        count--;
    #ifdef _MCS_SPIN
        custom_lock_unlock(&lock, &node);
    #else
        custom_lock_unlock(&lock);
    #endif
        if (count <= 0) {printf("%f\n", res); return NULL;}
        m = 1.0;
        for (int i = 1; i < after_section_iterations; i++)
            m *= i;
        res = m;
    }
}
```

```
int main (int argc, char *argv[]){
	int amount_thr;
	pthread_t *threads;
	if (argc != 5) error("test_m:wrong amount of args");
	if ((amount_thr = atoi(argv[1])) <= 0) error("test_m:wrong first arg");
	if ((count = atoi(argv[2])) < 0) error("test_m:wrong second arg");
	if ((critical_section_iterations = atoi(argv[3])) < 0) error("test_m:wrong third arg");
	if ((after_section_iterations = atoi(argv[4])) < 0) error("test_m:wrong fourth arg");
count = 840000;
	if (custom_lock_init(&lock, 0) != SUCCESS)
		error("test_m:custom_lock_init");
	threads = (pthread_t*)malloc(sizeof(pthread_t) * amount_thr);
	for (long i = 0; i < amount_thr; i++)
		if (pthread_create(&(threads[i]), NULL, &thread_function, (void*)i) != SUCCESS)
			error("test_m:pthread_create");
	printf("%s/n", "create them all");
	fflush(stdout);
	for (int i = 0; i < amount_thr; i++)
		pthread_join(threads[i], NULL);
	return EXIT_SUCCESS;
}
```

ПРИЛОЖЕНИЕ Б

Таблица Б.1 – Результаты сравнения спин-блокировок

			Количество потоков						
В крит	Вне крит	Тип	2	3	4	5	6	7	8
1	1	MCS	2,27E+08	1,77E+08	1,65E+08	1,61E+08	1,56E+08	1,55E+08	1,55E+08
		POSIX	1,5E+08	1,66E+08	2,05E+08	2,66E+08	2,87E+08	2,79E+08	3,16E+08
		билетная	1,87E+08	2,16E+08	2,31E+08	2,22E+08	2,18E+08	2,2E+08	2,19E+08
	10	MCS	2,15E+08	1,76E+08	1,68E+08	1,6E+08	1,57E+08	1,56E+08	1,58E+08
		POSIX	97126248	1,05E+08	1,13E+08	1,51E+08	2,39E+08	2,65E+08	3,03E+08
		билетная	1,87E+08	2,25E+08	2,4E+08	2,15E+08	2,1E+08	2,11E+08	2,07E+08
	100	MCS	1,87E+08	2,4E+08	1,94E+08	1,66E+08	1,57E+08	1,51E+08	1,46E+08
		POSIX	1,48E+08	2,51E+08	3,61E+08	3,11E+08	3,12E+08	3,4E+08	3,43E+08
		билетная	1,54E+08	2,37E+08	2,55E+08	2,53E+08	2,34E+08	2,27E+08	2,23E+08
	500	MCS	7,75E+08	5,59E+08	4,2E+08	3,65E+08	3,07E+08	2,65E+08	2,31E+08
		POSIX	7,49E+08	5,39E+08	4,06E+08	3,5E+08	2,93E+08	2,52E+08	2,19E+08
		билетная	7,53E+08	5,41E+08	4,08E+08	3,51E+08	2,93E+08	2,53E+08	2,2E+08
10	1	MCS	2,17E+08	1,86E+08	1,84E+08	1,82E+08	1,79E+08	1,8E+08	1,84E+08
		POSIX	99439571	1,11E+08	1,13E+08	2,14E+08	2,41E+08	3,09E+08	3,45E+08
		билетная	1,83E+08	2,23E+08	2,41E+08	2,32E+08	2,31E+08	2,27E+08	2,33E+08
	10	MCS	2,16E+08	1,83E+08	1,78E+08	1,81E+08	1,79E+08	1,84E+08	1,87E+08
		POSIX	1,29E+08	1,62E+08	1,43E+08	2,18E+08	2,87E+08	3,16E+08	3,51E+08
		билетная	1,91E+08	2,27E+08	2,51E+08	2,3E+08	2,26E+08	2,31E+08	2,32E+08
	100	MCS	1,83E+08	2,4E+08	2,01E+08	1,79E+08	1,76E+08	1,72E+08	1,82E+08
		POSIX	1,63E+08	2,7E+08	3,76E+08	3,2E+08	3,23E+08	3,54E+08	3,58E+08
		билетная	1,62E+08	2,51E+08	2,68E+08	2,69E+08	2,55E+08	2,46E+08	2,42E+08
	500	MCS	7,79E+08	5,62E+08	4,22E+08	3,73E+08	3,14E+08	2,72E+08	2,45E+08
		POSIX	7,64E+08	5,49E+08	4,13E+08	3,55E+08	2,99E+08	2,57E+08	2,23E+08
		билетная	7,62E+08	5,49E+08	4,14E+08	3,68E+08	3,06E+08	2,64E+08	2,28E+08
100	1	MCS	4,41E+08	4,32E+08	4,43E+08	4,55E+08	4,65E+08	4,77E+08	4,85E+08

Продолжение таблицы Б.1

			Количество потоков						
В крит	Вне крит	Тип	2	3	4	5	6	7	8
100		POSIX	3,15E+08	3,36E+08	3,45E+08	3,68E+08	3,72E+08	3,75E+08	3,83E+08
		билетная	4,35E+08	4,81E+08	4,93E+08	4,97E+08	5,08E+08	5,16E+08	5,19E+08
	10	MCS	4,49E+08	4,38E+08	4,43E+08	4,54E+08	4,66E+08	4,79E+08	4,84E+08
		POSIX	3,19E+08	3,74E+08	3,68E+08	3,72E+08	3,83E+08	3,72E+08	3,89E+08
		билетная	4,34E+08	4,83E+08	5,02E+08	5,04E+08	5,11E+08	5,19E+08	5,27E+08
	100	MCS	4,18E+08	4,27E+08	4,42E+08	4,52E+08	4,65E+08	4,74E+08	4,81E+08
		POSIX	4,47E+08	5,33E+08	5,7E+08	4,96E+08	5,05E+08	5,14E+08	5,35E+08
		билетная	4,35E+08	4,71E+08	4,8E+08	4,85E+08	4,93E+08	5,02E+08	5,1E+08
	500	MCS	9,1E+08	6,47E+08	4,9E+08	4,68E+08	4,62E+08	4,69E+08	4,81E+08
		POSIX	8,76E+08	6,29E+08	4,74E+08	5,44E+08	5,62E+08	5,73E+08	5,75E+08
		билетная	8,74E+08	6,31E+08	4,74E+08	4,88E+08	5,14E+08	5,2E+08	5,31E+08
500	1	MCS	1,63E+09	1,74E+09	1,73E+09	1,75E+09	1,76E+09	1,77E+09	1,78E+09
		POSIX	1,52E+09	1,64E+09	1,65E+09	1,65E+09	1,66E+09	1,67E+09	1,68E+09
		билетная	1,64E+09	1,78E+09	1,78E+09	1,79E+09	1,8E+09	1,81E+09	1,82E+09
	10	MCS	1,63E+09	1,74E+09	1,73E+09	1,75E+09	1,76E+09	1,77E+09	1,78E+09
		POSIX	1,51E+09	1,68E+09	1,68E+09	1,69E+09	1,68E+09	1,68E+09	1,68E+09
		билетная	1,65E+09	1,79E+09	1,79E+09	1,8E+09	1,81E+09	1,81E+09	1,82E+09
	100	MCS	1,64E+09	1,74E+09	1,73E+09	1,75E+09	1,76E+09	1,77E+09	1,78E+09
		POSIX	1,65E+09	1,84E+09	1,86E+09	1,81E+09	1,81E+09	1,81E+09	1,8E+09
		билетная	1,65E+09	1,77E+09	1,77E+09	1,79E+09	1,8E+09	1,8E+09	1,8E+09
	500	MCS	1,62E+09	1,74E+09	1,73E+09	1,75E+09	1,76E+09	1,77E+09	1,78E+09
		POSIX	1,65E+09	1,82E+09	1,85E+09	1,78E+09	1,78E+09	1,78E+09	1,78E+09
		билетная	1,64E+09	1,77E+09	1,78E+09	1,78E+09	1,79E+09	1,79E+09	1,8E+09

ПРИЛОЖЕНИЕ В

Примеры исходного кода библиотеки реализаций шаблона проектирования «Активный Объект»

Файл ACE/Job,hpp

```
#ifndef _JOB_HPP
#define _JOB_HPP
#include "ace/Method_Request.h"
#include "Spinlock.hpp"
class Job : public ACE_Method_Request{
public:
	Job(void);
	Job(double (*foo)(int), int arg);
	virtual int call(void);
	double result();
private:
	double (*_foo)(int);
	int _arg;
	double _result;
	Spinlock _result_lock;
};
#endif /*_JOB_HPP*/
```

Файл ACE/Job.cpp

```
#include "Job.hpp"
Job::Job(void){
	this->_foo = nullptr;
}
Job::Job(double (*foo)(int), int arg){
	this->_foo = foo;
	this->_arg = arg;
	this->_result_lock.lock();
}
int Job::call(void){
	if (this->_foo == nullptr)
		return (-1);
	this->_result = this->_foo(this->_arg);
	this->_result_lock.unlock();
	return (0);
}
double Job::result(){
	this->_result_lock.lock();
	double res = (this->_result);
	return (res);
}
```

Файл ACE/Task.cpp

```
#include "Task.hpp"
#include "Job.hpp"
Task::Task(void) : Task(1)
{}
Task::Task(int threads) {
	if (this->activate(THR_NEW_LWP, threads) == -1) {
		ACE_ERROR((LM_ERROR, "%p\n", "smth wrong"));
	}
}
Task::~Task(void){
}
int Task::svc(void){
	while(1){
		ACE_Method_Request *request = this->_queue.dequeue();
		if (request == nullptr)
			return (-1);
		int result = request->call();
		if (result != 0) break;
	}
	return (0);
}
int Task::put(Job *job, ACE_Time_Value *tv){
	return (this->_queue.enqueue(job));
}
```

Файл ACE/Task.hpp

```
#ifndef _TASK_HPP
#define _TASK_HPP
#include "ace/Task.h"
#include "ace/Activation_Queue.h"
#include "Job.hpp"
class Task : public ACE_Task_Base {
public:
	Task(void);
	Task(int threads);
	~Task(void);
	virtual int svc(void);
	int put(Job *job, ACE_Time_Value *tv = 0);
private:
	ACE_Activation_Queue _queue;
};
#endif /*_TASK_HPP*/
```

Файл GCD/mainGCD.c

```
#include <stdio.h>
#include <pthread.h>
#include "spinlock.h"
#include <stdlib.h>
```

```
#include <dispatch/dispatch.h>
#include "global_metric.h"
#include "factorial.h"
typedef struct {
	double result;
	spinlock_t spin;
	int arg;
	double (*foo)(int);
} task;
typedef struct {
	dispatch_queue_t *queue;
	int thr_count;
	int arg;
	double (*foo)(int);
} args;
void queue_func(void *arg){
	task *t = (task*)arg;
	t->result = t->foo(t->arg);
	spin_unlock(&(t->spin));
}
void *thread_func(void *a){
	args *arg = (args*)a;
	int thr_count = arg->thr_count;
	int fact =arg->arg;
	dispatch_queue_t *queue = arg->queue;
	double res = 0.0;
	while (thr_count){
		task t;
		t.arg = fact;
		t.foo = arg->foo;
		spin_init(&(t.spin), 0);
		spin_lock(&(t.spin));
		dispatch_async_f(*queue, &t, queue_func);
		spin_lock(&(t.spin));
		res = t.result;
#ifdef _DEBUG
		printf("%f\n", res);
#endif /*_DEBUG*/
		thr_count--;
	}
	return (0);
}
int main(int argc, char *argv[]){
	dispatch_queue_t queue;
	queue = dispatch_queue_create("some_queue", NULL);
	args arg;
	int amount_thr = atoi(argv[1]);
	arg.thr_count = atoi(argv[2]) / amount_thr;
	arg.arg = atoi(argv[3]);
	arg.foo = factorial;
	arg.queue = &queue;
```

```
	pthread_t *threads;
	threads = malloc(sizeof(pthread_t) * amount_thr);
	for (int i = 0; i < amount_thr; i++)
		pthread_create(&(threads[i]), NULL, &thread_func, (void*)&arg);
	for (int i = 0; i < amount_thr; i++)
		pthread_join(threads[i], NULL);
	dispatch_release(queue);
	return (0);
}
```

Файл custom/ActiveObject.hpp

```
#ifndef _ACTIVE_OBJECT_HPP
#define _ACTIVE_OBJECT_HPP
#include "queue.hpp"
#include "Task.hpp"
#include <pthread.h>
class Active_Object{
public:
	Active_Object();
	~Active_Object();
	pthread_t activate();
	void add_task(Task *t);
private:
	mpscq_t queue;
	static void *call_next(void *ao);
};
#endif /*_ACTIVE_OBJECT_HPP*/
```

Файл custom/ActiveObject.cpp

```
#include <xmmintrin.h>
#include <functional>
#include "Active_Object.hpp"
#include <stdio.h>
Active_Object::Active_Object(){
	mpscq_node_t *stub = new mpscq_node_t();
	mpscq_create(&(this->queue), stub);
}
Active_Object::~Active_Object(){
	mpscq_node_t *node;
	while ((node = mpscq_pop(&(this->queue))) != 0)
		delete node;
	delete this->queue.tail;
}
pthread_t Active_Object::activate(){
	pthread_t thread;
	pthread_create(&thread, NULL, Active_Object::call_next, (void*)this);
	return (thread);
}
void *Active_Object::call_next(void *ao){
```

```
        Active_Object *th = (Active_Object*)ao;
        mpscq_node_t *node;
        while(1){
                node = mpscq_pop(&(th->queue));
                if (node != NULL){
                        int res = ((Task*)(node->state))->call();
                        delete node;
                        if (res == -1) break;
                }
                _mm_pause();
                continue;
        }
        return (NULL);
}
void Active_Object::add_task(Task *t){
        mpscq_node_t *next = new mpscq_node_t();
        next->state = (void*)t;
        mpscq_push(&(this->queue), next);
}
```

Файл custom/queue.cpp

```
#include "queue.hpp"
void mpscq_create(mpscq_t *self, mpscq_node_t *stub){
   stub->next = 0;
   self->head = stub;
   self->tail = stub;
}
void mpscq_push(mpscq_t *self, mpscq_node_t *n){
   n->next = 0;
      mpscq_node_t *prev = __sync_lock_test_and_set(&(self->head), n);
   prev->next = n;
}
mpscq_node_t *mpscq_pop(mpscq_t *self){
   mpscq_node_t *tail = self->tail;
   mpscq_node_t *next = tail->next;
   if (next){
      self->tail = next;
      tail->state = next->state;
      return (tail);
   }
   return (0);
}
```

Файл custom/queue.hpp

```
#ifndef _QUEUE_HPP
#define _QUEUE_HP
struct mpscq_node_t {
        mpscq_node_t* volatile next;
        void *state;
};
```

```
struct mpscq_t {
	mpscq_node_t* volatile head;
	mpscq_node_t *tail;
};
void mpscq_create(mpscq_t *self, mpscq_node_t *stub);
void mpscq_push(mpscq_t *self, mpscq_node_t *n);
mpscq_node_t *mpscq_pop(mpscq_t *self);
#endif /*_QUEUE_HPP*/
```

Файл custom/Task.cpp

```
#include "Task.hpp"
Task::Task(void){
	this->_foo = nullptr;
}
Task::Task(double (*foo)(int), int arg){
	this->_foo = foo;
	this->_arg = arg;
	this->_result_lock.lock();
}
int Task::call(void){
	if (this->_foo == nullptr)
		return (-1);
	this->_result = this->_foo(this->_arg);
	this->_result_lock.unlock();
	return (0);
}
double Task::result(){
	this->_result_lock.lock();
	double res = (this->_result);
	return (res);
}
```

Файл custom/Task.hpp

```
#ifndef _TASK_HPP
#define _TASK_HPP
#include "Spinlock.hpp"
class Task {
public:
	Task(void);
	Task(double (*foo)(int), int arg);
	int call(void);
	double result();
private:
	double (*_foo)(int);
	int _arg;
	double _result;
	Spinlock _result_lock;
};
#endif /*_TASK_HPP*/
```

ПРИЛОЖЕНИЕ Г

Таблица Г.1 – Результаты сравнения реализаций шаблона проектирования «Активный Объект»

Тип	Значение аргумента	Среднее время в циклах	Среднее квадратичное отклонение
ACE	10	1366085681	43762103,28
	50	1367104920	23100689,98
	75	1434786104	26331013,52
	100	1307712613	31353861,21
	250	1251348410	6265901,83
	500	2074223016	8844215,096
	1000	3676294732	5922559,103
	5000	16603529628	5217146,102
	10000	32764421900	3938196,792
	20000	65100809200	6358507,598
GCD	10	500953422,3	11843654,65
	50	515873305,1	8854553,271
	75	570909178,5	13994850,77
	100	613103451,3	9570507,847
	250	1079703433	6511635,403
	500	1885784135	5531243,49
	1000	3508189664	5174902,386
	5000	16412477855	1553958,227
	10000	32574618861	1101235,346
	20000	64900445601	905748,06
Собственная реализация	10	414971156,3	1109920,661
	50	448516958,6	952480,8282
	75	493392179,2	1027242,223
	100	552751970,7	1003148,684
	250	1038573222	754006,4742
	500	1846183555	974343,4865
	1000	3461318755	1003779,263
	5000	16391460950	950023,4983
	10000	32556869383	828156,735
	20000	64883634257	1021320,188
"Билетная" спин-блокировка	10	165215584,8	2281803,541
	50	286723148,3	712020,9806
	75	371937381,7	776696,062
	100	452586880,7	730796,6189

Продолжение таблицы Г.1

Тип	Значение аргумента	Среднее время в циклах	Среднее квадратичное отклонение
	250	950703559,1	560513,3011
	500	1758995593	630193,2743
	1000	3374191215	804408,2527
	5000	16307027802	866055,6412
	10000	32469492762	1007133,842
	20000	64796267777	772139,3345