Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра Вычислительных систем

Допустить к защите

Зав.каф. _____Мамойленко С.Н.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Оптимизация синхронизации параллельных программ для вычислительных систем с общей памятью

Пояснительная записка

Студент _____ / Гайдай А.В. /

Факультет ИВТ Группа ИВ-222

Руководитель _____ / Кулагин И.И. /

Новосибирск 2016 г.

Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

КАФЕДРА

Вычислительных систем

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА

СТУДЕНТА Гайдая А.В. ГРУППЫ ИВ-222

УТВЕРЖДАЮ

«___» _____ 2016 г.

Зав. кафедрой

_____ *Мамойленко С.Н.*

Новосибирск 2016 г.

1. Тема выпускной квалификационной работы бакалавра
Оптимизация синхронизации параллельных программ для вычислительных систем с общей памятью

утверждена приказом СибГУТИ от «6» июня 2016 г. № 4/515о-16

2. Срок сдачи студентом законченной работы «10» июня 2016 г.

3. Исходные данные к работе

Бовет. Д., Чезати М. Ядро Linux, 3-е изд.: Пер. с англ. // Спб. БХВ-Петербург, 2007. – 1104 с.: ил.

Лав Р. Ядро Linux: описание процесса разработки, 3-е изд. – М.: Вильямс // 2015. – 496 с.

4. Содержание пояснительной записки (перечень подлежащих разработке вопросов)	**Сроки выполнения по разделам**
Введение	15.05 – 17.05
Процессы и потоки	18.05 – 21.05
Аппаратный кэш	22.05 – 25.05
Алгоритм оптимизации синхронизации	26.05 – 30.05
Программный пакет «mutex-optimizer»	31.05 – 3.06
Результаты экспериментов	4.06 – 6.06
Заключение	6.06 – 6.06

Дата выдачи задания «___» _____ 2016 г.
Руководитель _____
подпись

Задание принял к исполнению «___» _____ 2016 г.
Студент _____
подпись

АННОТАЦИЯ

Выпускная квалификационная работа Гайдая А.В. по теме «Оптимизация синхронизации параллельных программ для вычислительных систем с общей памятьюОптимизация синхронизации параллельных программ для вычислительных систем с общей памятью»

Объём работы - *36* страниц, на которых размещены *19* рисунков и *11* таблиц. При написании работы использовалось *6* источников.

Целью бакалаврской работы была реализация алгоритма, уменьшающего время операции захвата мьютекса при работе параллельной программы в вычислительной системе с общей памятью за счёт предварительного профилирования данной программы и анализа возникающих конфликтных ситуаций при работе с используемыми в ней мьютексами.

Мьютекс – это примитив синхронизации, позволяющий создавать в коде программы критические секции, выполнение которых возможно только одним потоком в каждый момент времени. Мьютексы применяются повсеместно и актуальность в их исследовании довольно высока.

В рамках дипломного проекта был разработан программный пакет «mutex-optimizer» состоящий из двух модулей: профилирования и оптимизации. Проведено экспериментальное исследование эффективности алгоритма, с помощью синтетических тестов (microbenchmark).

По результатам проведённых экспериментов выявлено увеличение производительности оптимизированной версии параллельной программы и установлена зависимость времени её выполнения от количества и интенсивности возникновения конфликтных ситуаций при конкурентном доступе к мьютексам.

Ключевые слова: мьютекс, конфликт, профилирование, оптимизация, многопоточность.

Работа выполнена на кафедре вычислительных систем СибГУТИ.

Ожидаемые результаты: увеличение производительности оптимизированной версии параллельной программы.

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Форма утверждена научно-методическим советом университета
протокол № 3 от 19.02.2015 г.

ОТЗЫВ

на выпускную квалификационную работу студента Гайдая А.В. по теме «Оптимизация синхронизации параллельных программ для вычислительных систем с общей памятью»

В выпускной квалификационной работе Гайдай А.В. разработал программный пакет для повышения эффективности использования примитивов синхронизации – мьютексов. Его особенностью является учет динамически изменяющихся характеристик критических секций. Программный пакет так же содержит модуль профилирования мьютексов в многопоточных программах.

Считаю, что выпускная квалификационная работа Гайдая А.В. заслуживает оценки «ОТЛИЧНО», а Гайдай А.В. присвоения степени бакалавр по направлению 09.03.01 «Информатика и вычислительная техника».

Оценка уровней сформированности общекультурных и профессиональных компетенций обучающегося:

Компетенции		Уровень сформированности компетенции		
		Высокий	Средний	Низкий
Профессиональные	ПК-11			

Работа имеет практическую ценность		Тема предложена предприятием	
Работа внедрена		Тема предложена студентом	
Рекомендую работу к внедрению		Тема является фундаментальной	
Рекомендую работу к опубликованию		Рекомендую студента в магистратуру	
Работа выполнена с применением ЭВМ		Рекомендую студента в аспирантуру	

Ст. преп. Кафедры ВС Кулагин И.И. _____

Содержание

1 ВВЕДЕНИЕ 6
2 ПРОЦЕССЫ И ПОТОКИ 7
2.1 Состояния процесса 7
2.2 Взаимоотношения между процессами 8
2.3 Многопоточность 9
3 АППАРАТНЫЙ КЭШ 12
3.1 Методы отображения 12
3.2 Работа с кэшем 15
3.3 Протокол когерентности MESI 16
3.4 Мьютекс как переменная 19
4 АЛГОРИТМ ОПТИМИЗАЦИИ СИНХРОНИЗАЦИИ 20
4.1 Этап профилирования 20
4.2 Этап оптимизации 21
5 ПРОГРАММНЫЙ ПАКЕТ «MUTEX-OPTIMIZER» 22
5.1 Структура модуля профилирования 22
5.2 Структура модуля оптимизации 24
6 РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ 26
7 ЗАКЛЮЧЕНИЕ 33
ПРИЛОЖЕНИЕ А 34
ПРИЛОЖЕНИЕ Б 35

1 ВВЕДЕНИЕ

При разработке параллельных программ для обеспечения их корректной работы необходимо избегать возникновения ситуации гонки за данными (data race). Для этой цели используются механизмы взаимного исключения – мьютексы.

Мьютекс – примитив синхронизации, позволяющий создавать в коде программы критические секции, выполнение которых возможно только одним потоком в каждый момент времени. Мьютексы могут находиться в одном из двух состояний – открытом или закрытом. При выполнении потоком операции захвата (mutex lock), мьютекс переводится в закрытое состояние – становится недоступен для захвата другими потоками. Когда поток освобождает мьютекс (mutex unlock), его состояние становится открытым – доступным для захвата.

Однако, при попытке захвата мьютекса может возникать ситуация, называемая конфликтом (contention) [1], при которой несколько потоков пытаются одновременно захватить мьютекс. Операция захвата мьютекса в случае возникновения конфликта требует больше времени, чем операция захвата без возникновения конфликтной ситуации за счёт негативных явлений процессорного кэша. Промышленным стандартом реализации мьютексов при разработке параллельных программ является динамическая библиотека pthread в GNU libc, однако, реализация в текущей версии (glibc 2.23) не учитывает динамически изменяющееся состояние конфликта при захвате мьютекса.

В данной бакалаврской работе (БР) реализован алгоритм синхронизации оптимизации параллельных программ для вычислительных систем с общей памятью. Алгоритм учитывает динамически изменяющиеся характеристики критических секций.

2 ПРОЦЕССЫ И ПОТОКИ

Процесс – это фундаментальная абстракция. Существует множество трактовок определения процесса: экземпляр выполняемой программы, контекст выполнения работающей программы и др. Между программой и процессом имеется важное отличие: несколько процессов могут параллельно выполнять одну программу, и в то же время один процесс может последовательно выполнять несколько программ [2][3].

Для взаимодействия операционной системы с процессами, каждому из них соответствует некоторая структура, называемая дескриптором процесса (рисунок 2.1). Дескриптор описывает состояние процесса, его приоритет, адресное пространство, права доступа и т. д.

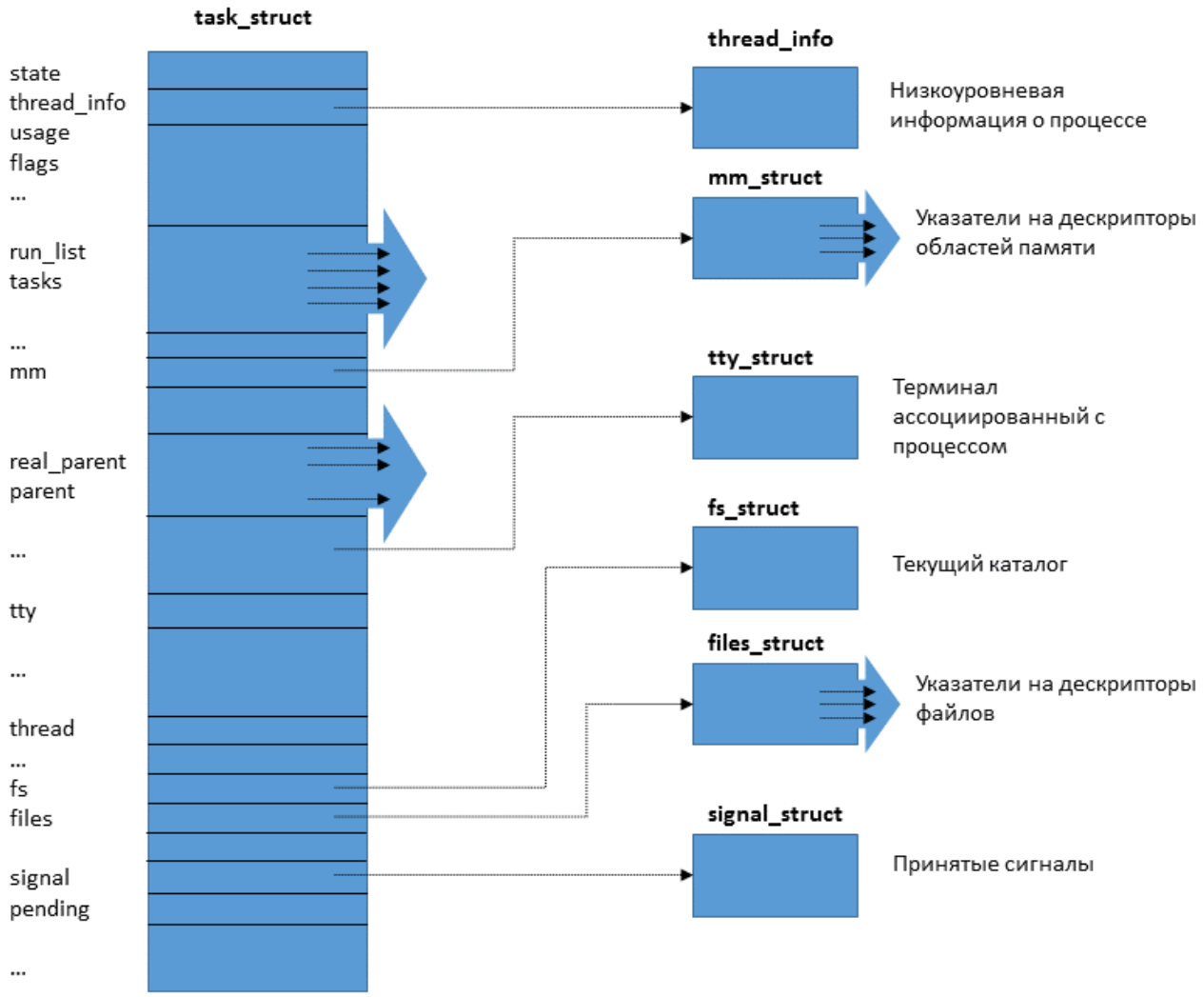


Рисунок 2.1 – Дескриптор процесса в Linux

2.1 Состояния процесса

В Linux процесс может находится в следующих состояниях (рисунок 2.2):

- TASK_RUNNING – процесс либо выполняется или ждёт своего выполнения;

- TASK_INTERRUPTIBLE – процесс приостановлен до удовлетворения некоторого условия. Доставка сигнала ожидающему процессу изменяет его состояния;
- TASK_UNINTERRUPTIBLE - процесс приостановлен до удовлетворения некоторого условия. Доставка сигнала ожидающему процессу не изменяет его состояния;
- TASK_STOPPED – выполнение процесса остановлено;
- TASK_TRACED – выполнение процесса остановлено отладчиком;
- EXIT_ZOMBIE – выполнение процесса завершено. Процесс-родитель ещё не сделал системный вызов wait4() или waitpid();
- EXIT_DEAD – выполнение процесса завершено. Процесс-родитель сделал системный вызов wait4() или waitpid().

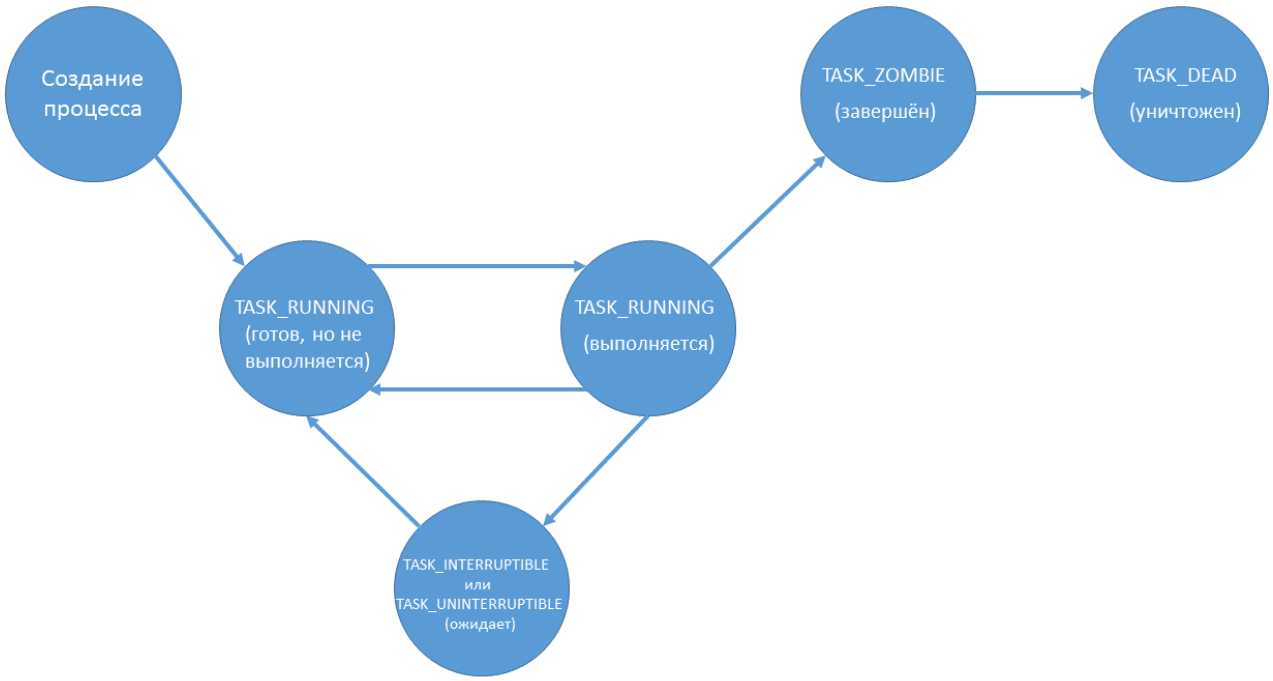


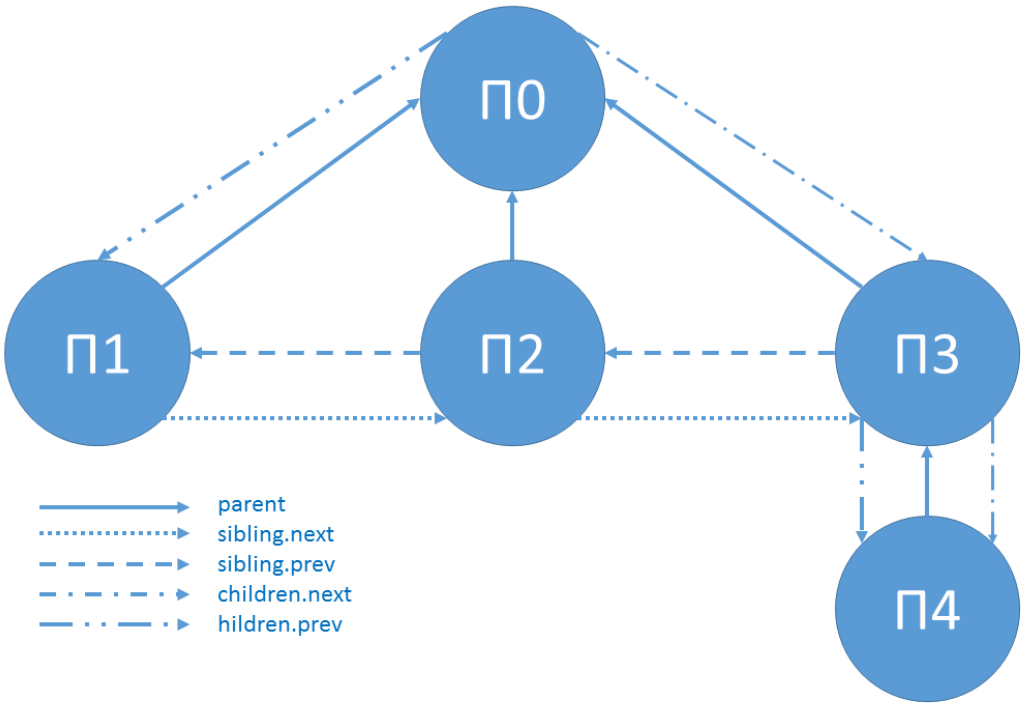
Рисунок 2.2 – Граф состояний процесса

2.2 Взаимоотношения между процессами

Выделяют два типа процессов: ядерные и пользовательские. Как можно понять из названия, первые выполняются в пространстве ядра (kernel space), вторые – в пользовательском пространстве (user space). В данной БР рассматриваются пользовательские процессы.

В общем случае, процесс выполняет одну последовательность инструкций в собственном адресном пространстве. Однако, в современных операционных системах процессы могут состоять из нескольких параллельных ветвей, выполняемых в том же адресном пространстве, что и процесс-родитель. В сфере информационных технологий параллельные ветви процесса называют потоками (thread). У каждого потока имеется свой набор значений регистров и собственный стек. Программы, включающие в себя несколько параллельно выполняющихся ветвей, называются многопоточными.

Для примера взаимоотношения между процессами и потоками на рисунке 2.3 приведён граф отношений группы процессов.



Процесс П0 последовательно создаёт процессы П1, П2 и П3. Процесс П3, в свою очередь, создаёт процесс П4.

Рисунок 2.3 – Отношения в группе из пяти процессов

Таким образом, программа, которая состоит из нескольких потоков, называется многопоточной.

2.3 Многопоточность

Многопоточность (multi-threading) – одна из наиболее интересных и актуальных тем в области информационных технологий. Актуальность этой темы особенно велика, в связи с широким распространением многоядерных процессоров и стремительном развитии вычислительных систем с общей памятью. Многие современные отрасли не смогли бы существовать без многопоточных программ.

При разработке параллельных программ, для обеспечения корректной работы, следует избегать возникновения ситуаций гонки за данными.

2.3.1 Состояние гонки

Состояние гонки за данными возникает в случае обращения нескольких потоков к одним и тем же адресам памяти. В случае обращения для чтения, конфликта нет, однако, если речь идёт о записи, появляется неопределённость результата.

Пусть поток П0 изменяет переменную VAR общую с потоком П1. В это же время поток П1 также модифицирует эту переменную. Итогом таких действий будет неопределённый результат. Происходит потеря данных – программа работает некорректно. На рисунке 2.4 представлена иллюстрация описанной ситуации.

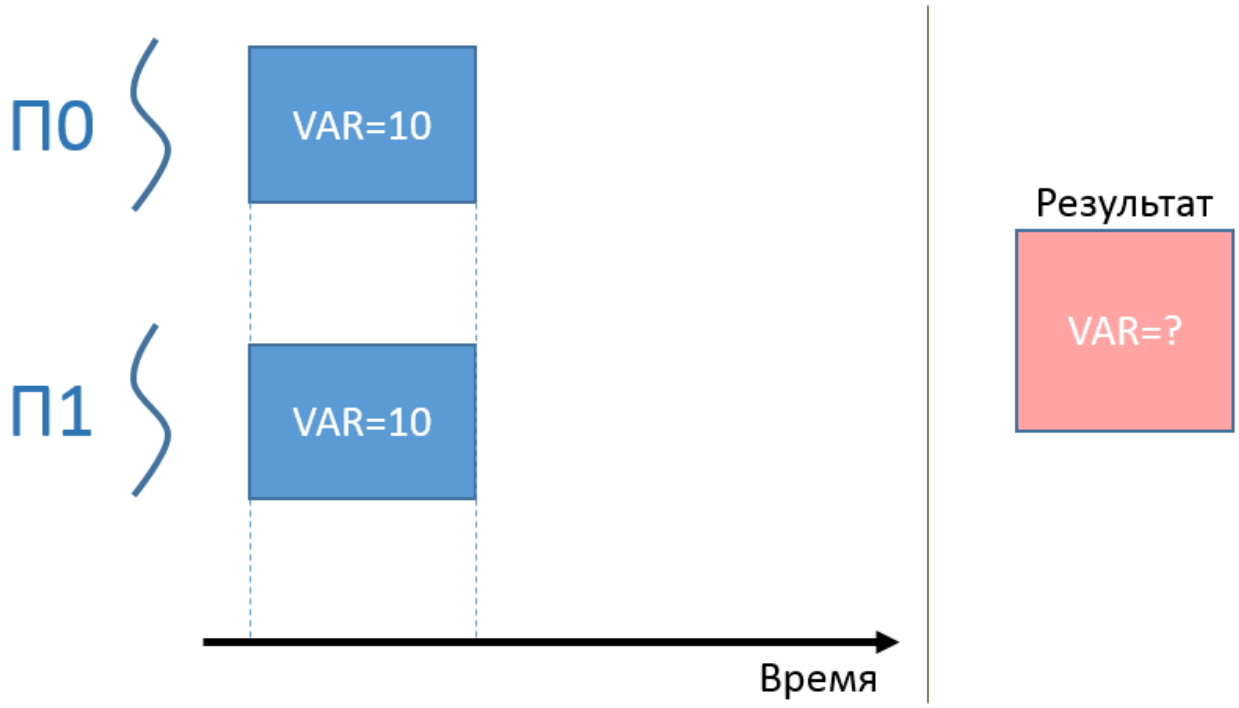


Рисунок 2.4 – Пример возникновения состояния гонки

2.3.2 Синхронизация доступа к общим данным

Для исключения возможности возникновения ситуации гонки при создании многопоточных приложений используются примитивы синхронизации, позволяющие создавать в коде программы критические секции, выполнение которых возможно ограниченным количеством потоков в каждый момент времени.

В данной БР используются мьютексы (mutex, mutual exclusion – взаимное исключение) – примитив синхронизации гарантирующий выполнение критической секции только одним потоком в каждый момент времени. Как отмечалось ранее, мьютексы могут находится в одном из двух состояний – открытом или закрытом. Также было сказано о возможности возникновения конфликтных ситуаций при попытке захвата мьютекса, если имеется конкурентный доступ нескольких потоков к одному мьютексу.

Мьютекс является обычной переменной, которая кэшируется, используясь на разных ядрах. Таким образом, согласно протоколу согласования кэшей MESI, при одновременном обращении нескольких потоков к одному мьютексу, фактически происходит одновременный доступ к одной и той же области памяти и, как следствие, используемые данные становятся невалидными для других ядер. Повышение производительности операции захвата мьютекса достигается за счёт уменьшения накладных расходов при работе с критическими секциями.

С помощью мьютексов пример описанный в разделе 2.3.1 можно модифицировать так, что неопределённость результата пропадёт (рисунок 2.5). Для этого обозначим участок кода, изменяющий общую переменную VAR, как критическую секцию. Это позволит избежать одновременной модификации данных.

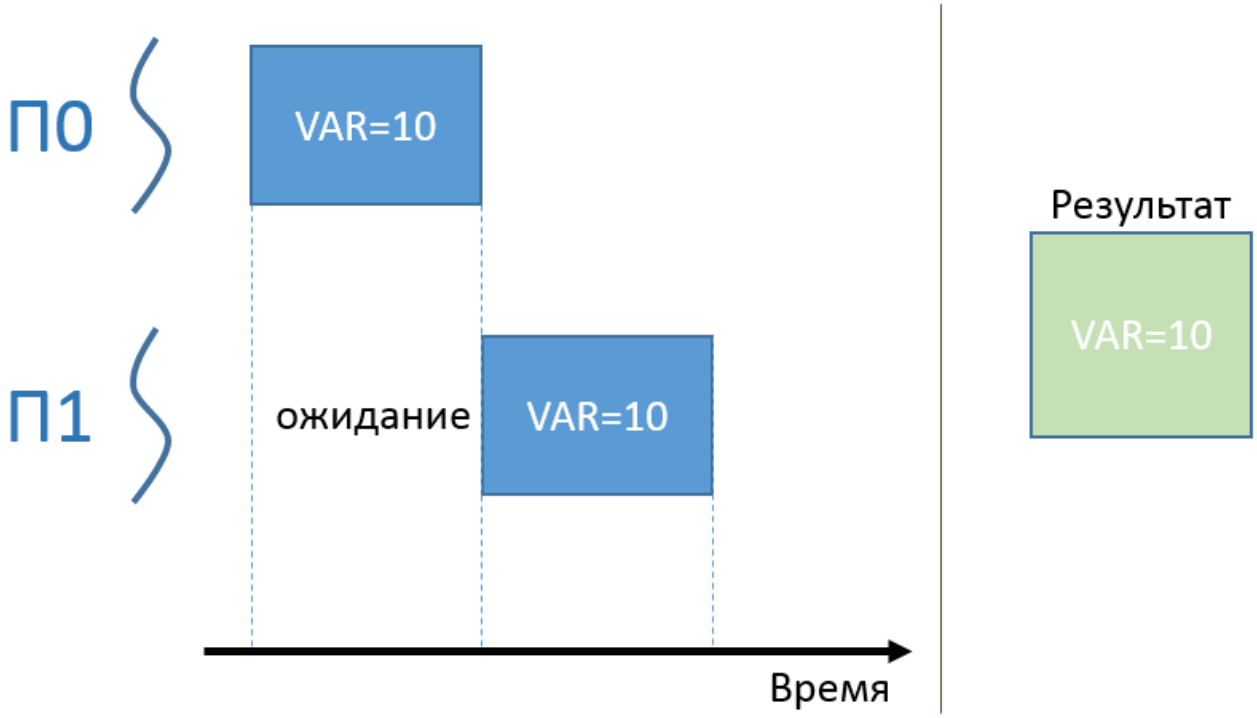


Рисунок 2.5 – Пример синхронизации доступа к общим данным

3 АППАРАТНЫЙ КЭШ

В современных микропроцессорах тактовая частота измеряется в гигагерцах. Время доступа чипов динамической оперативной памяти находится в пределах сотен тактов. В следствие этого, процессор работает с задержкой при выполнении инструкций, которые требуют выборки операндов из памяти или сохранения результатов в память.

С целью сглаживания разности в скорости работы между процессором и оперативной памятью были созданы аппаратные кэши. В их основе лежит принцип локальности ссылок: в силу циклической структуры большинства программ и размещения их данных в линейных массивах, адреса, смежные с адресами, использованными недавно, имеют высокую вероятность выбора при дальнейшей работе программы. По этой причине имеет смысл использовать небольшую, но более быструю память для хранения недавно используемых данных.

На рисунке 3.1 показано как размещается кэш. Он состоит из статической оперативной памяти и контроллера кэша, который хранит информацию о состояниях строк кэша.

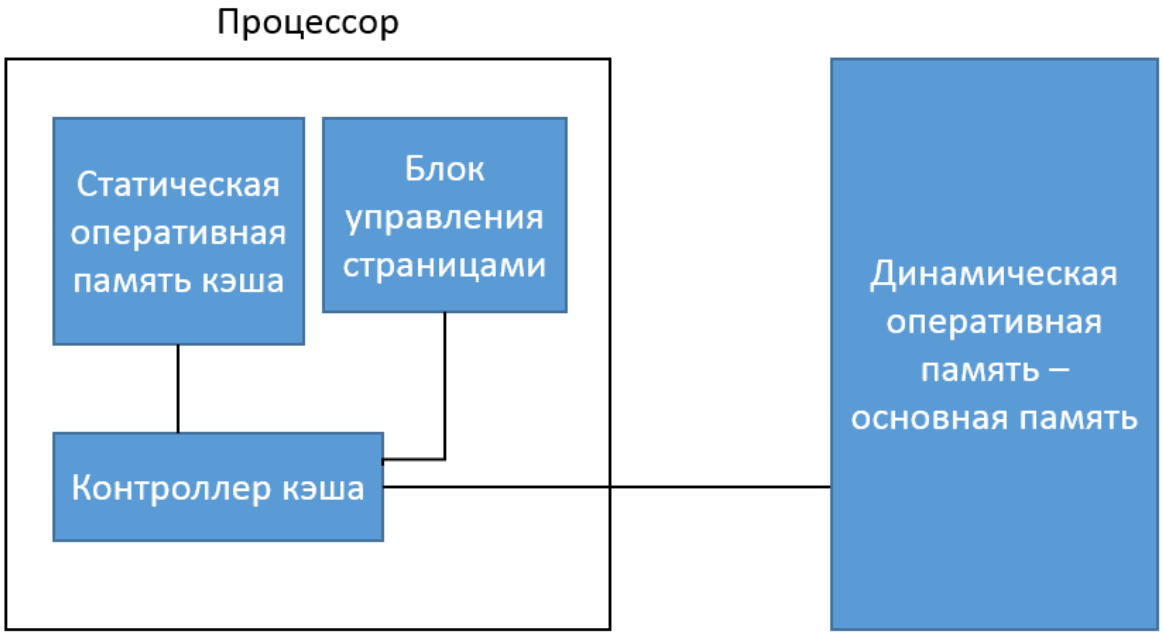


Рисунок 3.1 – Аппаратный кэш процессора

3.1 Методы отображения

Предположим, что основная память имеет ёмкость $M = 2^N$, где N – число двоичных разрядов адреса. Ёмкость кэша равна m, при этом $n = \log_2 m$ – число разрядов, необходимых для адресации m строк кэша. Физический адрес обычно разбивают на три секции:

- Старшие биты – тег ($T = N - S - b$);
- Средние биты – индекс подмножества (S);

- Младшие биты – смещение внутри строки (b).

Выделяют три типа отображения в кэш:

1) Прямое: строка основной памяти всегда хранится в одном и том же месте кэша (рисунок 3.2).
   $S = n; T = 0; b = 4.$
2) Ассоциативное: любая строка основной памяти может храниться в любом участке кэша (рисунок 3.3).
   $S = 0; T = n; b = 4.$
3) Множественно-ассоциативное с n каналами: любая строка основной памяти может быть сохранена в любой из n строк кэша (рисунок 3.4).
   $0 < S < n, S \in Z; T = N - S - b; b = 4.$

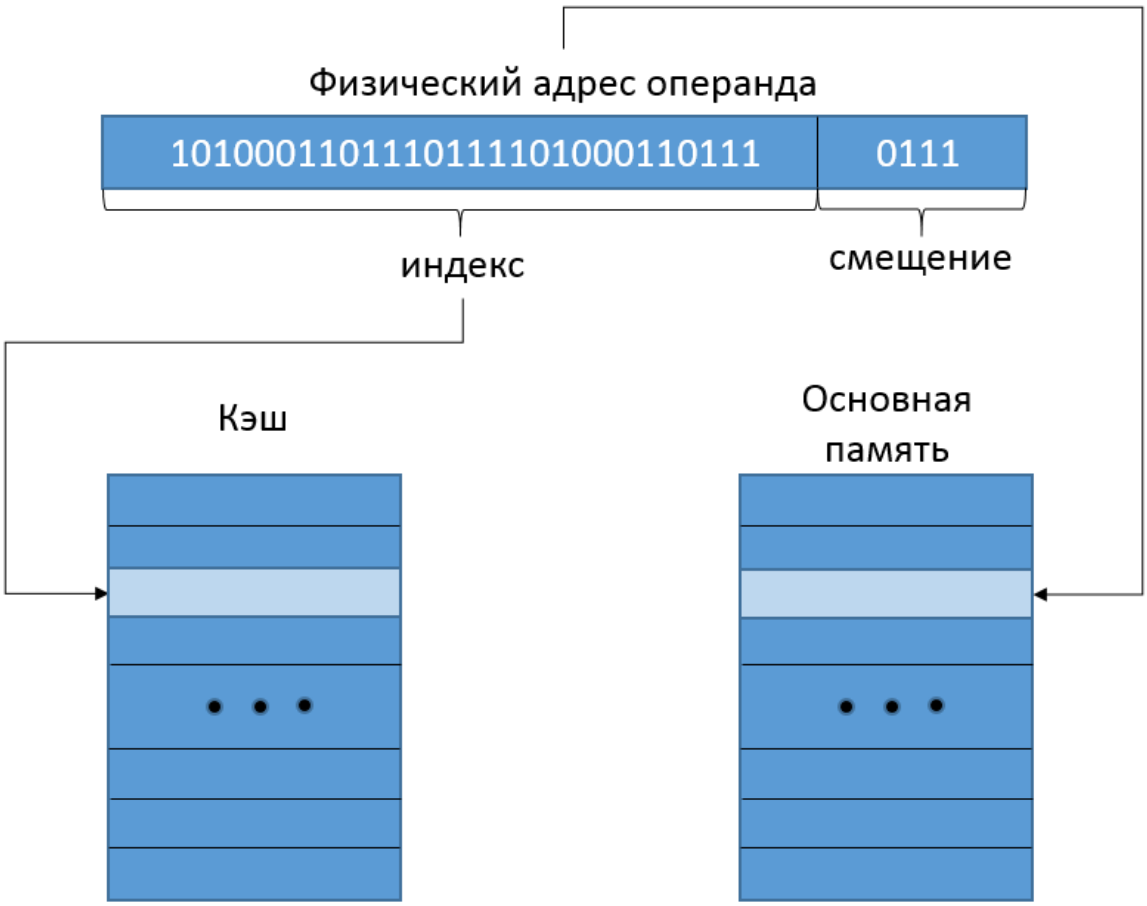


Рисунок 3.2 – Метод прямого отображения в кэш

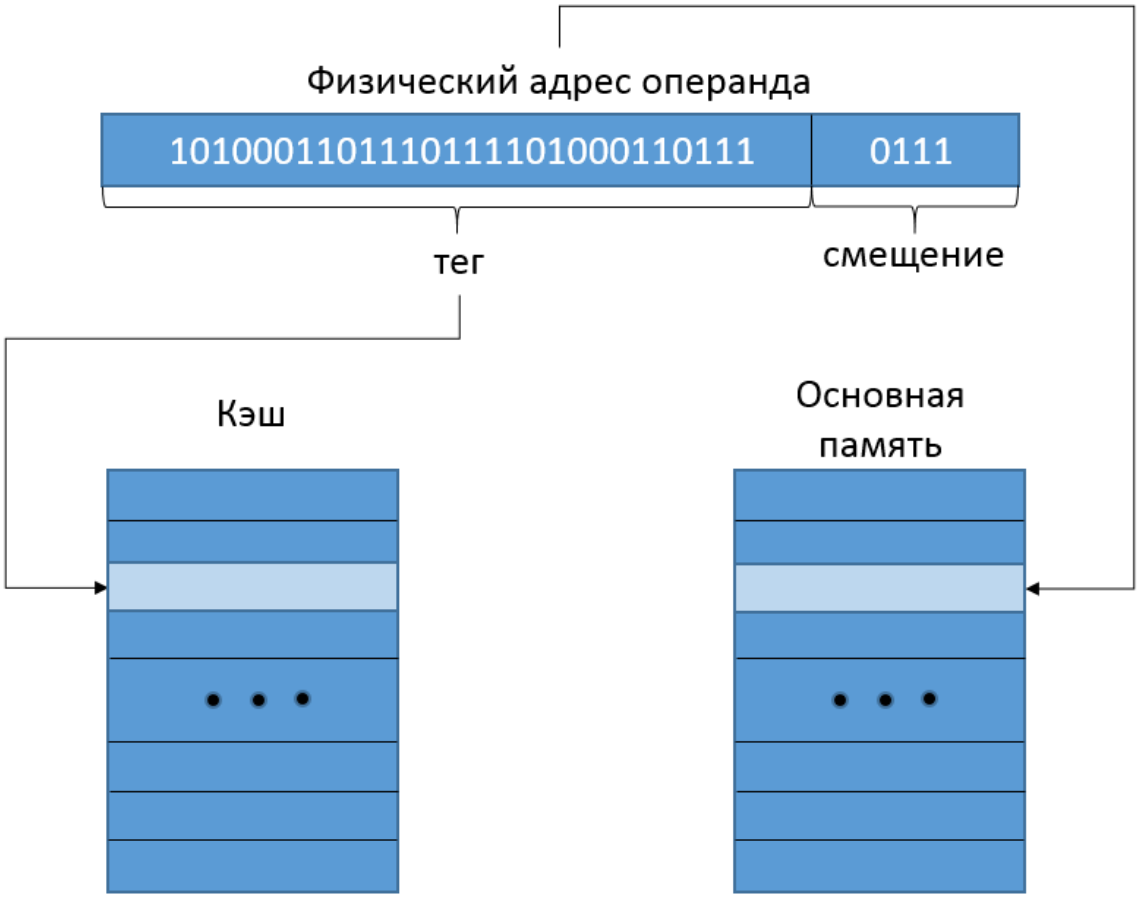
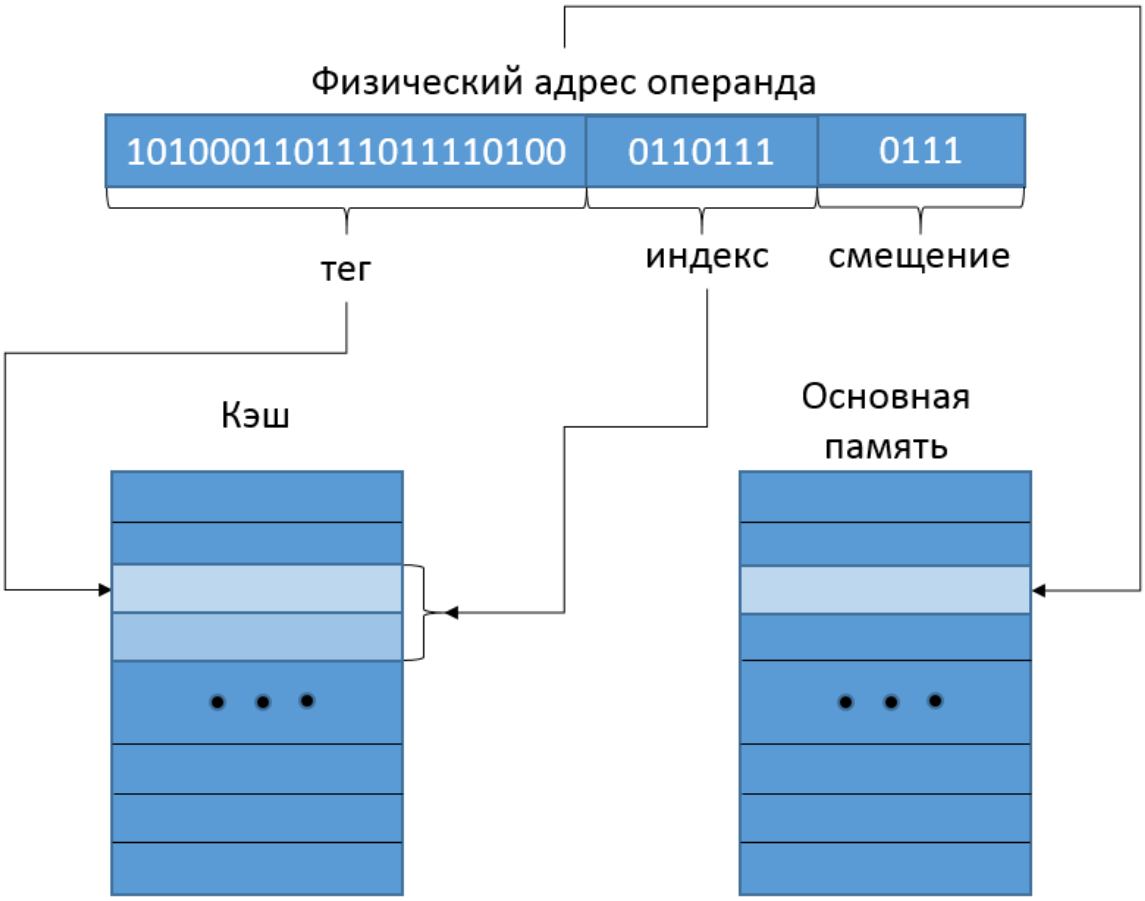


Рисунок 3.3 – Метод ассоциативного отображения в кэш



Двухкональный множественно-ассоциативный кэш

Рисунок 3.4 – Метод множественно-ассоциативного отображения в кэш

3.2 Работа с кэшем

В большинстве случаев в вычислительных системах с общей памятью используется метод множественно-ассоциативного отображения в кэш-память. При обращении к ячейке оперативной памяти процессор извлекает из физического адреса индекс и сравнивает теги всех строк в найденном подмножестве со старшими битами физического адреса. Если для одной из строк теги совпали, говорят, что процессор попал в кэш (Cache Hit); иначе произошёл промах (Cache Miss).

В случае попадания в кэш, контроллер кэша ведёт себя по-разному в зависимости от типа доступа. Если выполнялась операция чтения, то процессор берёт необходимые данные из кэша, не обращаясь к оперативной памяти, и, как следствие, экономится время процессора. В случае выполнения операции записи, контроллер может прибегнуть к одной из базовых стратегий: сквозная запись (Write-through) – данные записываются как в кэш, так и в оперативную память, фактически выключая кэш для записи; обратная запись (Write-back) – данные записываются только в кэш. При стратегии обратной записи оперативная память, в конце концов, тоже должна быть обновлена. Это происходит только в случае выполнения процессором инструкции, требующей сброса записей в кэше, или, когда приходит сигнал FLUSH (обычно после ситуации промаха мимо кэша).

Если процессор промахнулся мимо кэша, то некоторая строка кэша, в случае необходимости, записывается в основную память, а искомая информация наоборот – добавляются в кэш из основной памяти. Существуют различные алгоритмы замещения записей в кэш-памяти [4]: алгоритм L. Belady, LRU (Least Recently Used), MRU (Most Recently Used), RR (Random Replacement) и др.

3.3 Протокол когерентности MESI

В многопроцессорных/многоядерных системах (рисунок 3.5) у каждого процессора/ядра имеется свой аппаратный кэш. В таких системах необходим контроль согласованности кэшей. Связанно это с тем, что при модификации данных из кэша одного процессора/ядра, нужно проверять наличие соответствующих данных в кэшах других процессоров/ядер системы и, в случае необходимости, обновлять информацию, делая её актуальной. Такая деятельность выполняется на аппаратном уровне с помощью специальной электронной схемы. Отслеживание аппаратных кэшей осуществляется согласно протоколу когерентности – MESI (Modify Exclusive Shared Invalid) [5].

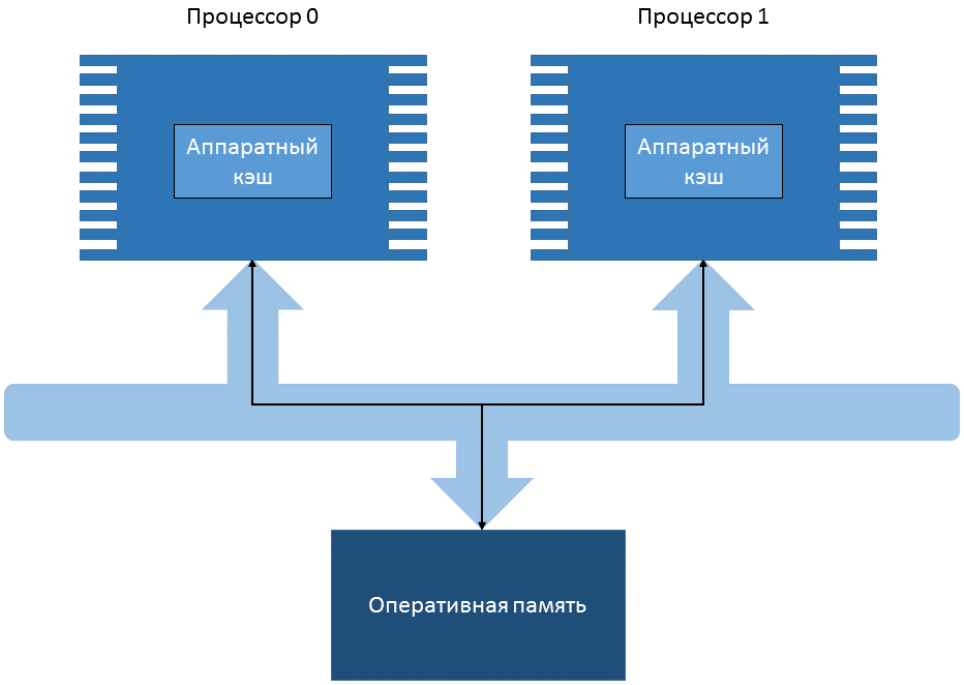


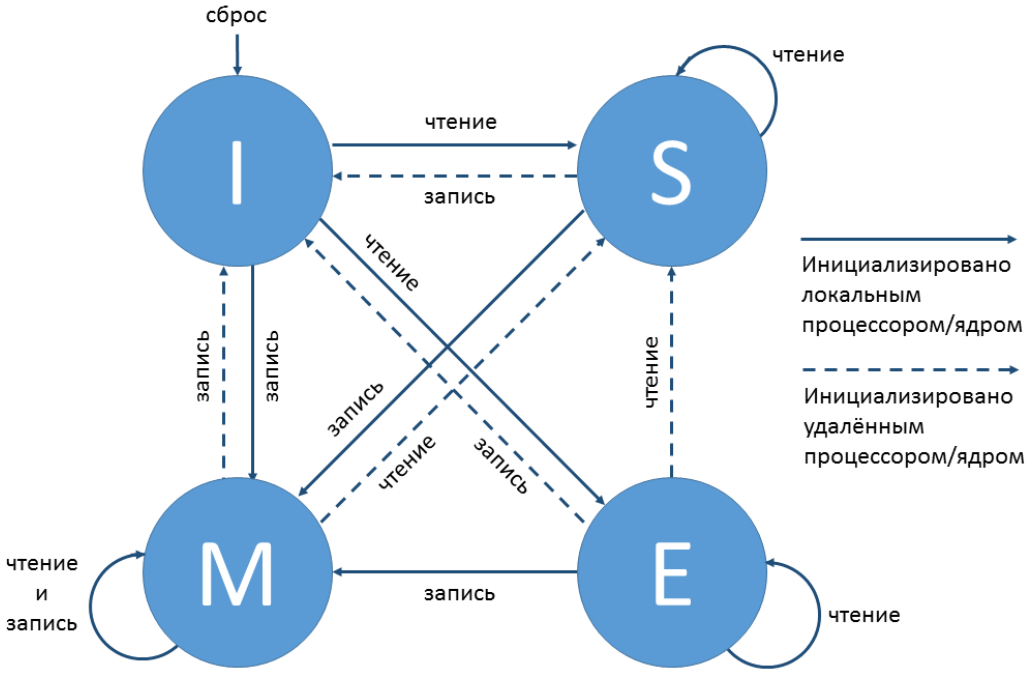
Рисунок 3.5 – Кэши в двухпроцессорной системе

Протокол отложенной записи MESI реализован на аппаратном уровне и его основная задача осуществлять согласование данных, хранящихся в кэшах системы. В соответствии с этим протоколом данные каждой строки кэша могут находится в одном из четырёх состояний:

- Модифицированные (Modify) – данные в кэше являются действительными, а в основной памяти недействительными. Копий не существует;
- Эксклюзивные (Exclusive) – данные хранятся только в одном кэше. Память обновлена;
- Разделяемые (Shared) – данные могут храниться в нескольких кэшах. Память обновлена;

- Недействительные (Invalid) – данные являются невалидными.

Порядок перехода строки кэша из одного состояния в другое зависит от: текущего состояния строки, типа выполняемой операции (чтение/запись), попадания или промаха процессора при обращении к кэш-памяти. На рисунке 3.6 приведён граф переходов состояний без учёта однократной записи.



M – «Modify»; E – «Exclusive»; S – «Shared»; I – «Invalid»

Рисунок 3.6 – Граф переходов состояний без учёта однократной записи

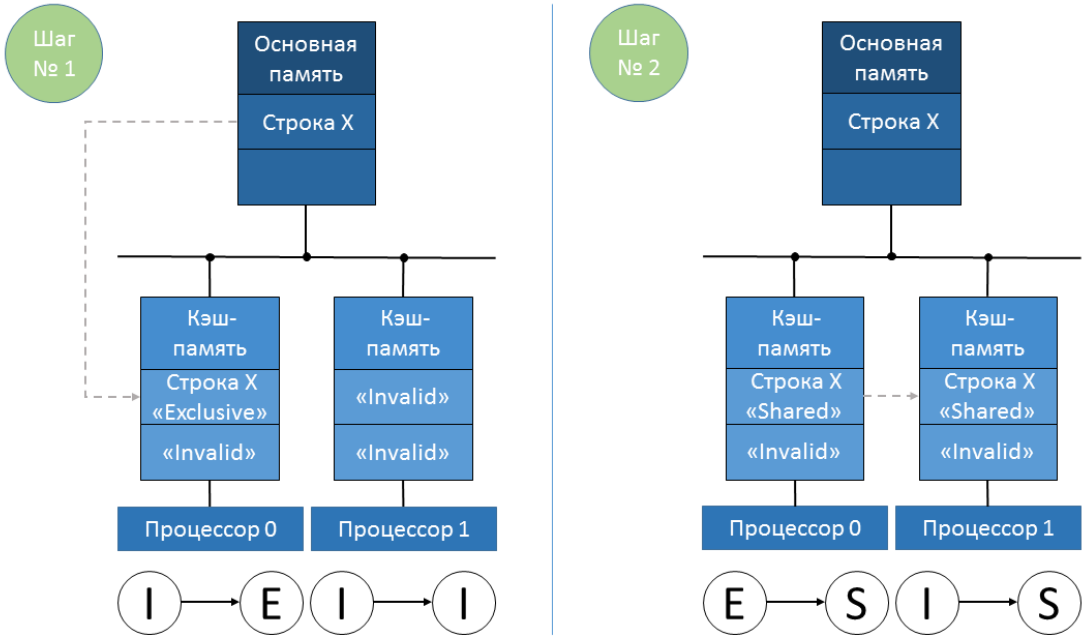
Пусть одним из процессоров/ядер делается запрос на чтение данных из кэша, которых там нет (промах мимо кэша при чтении). В таком случае запрос будет широковещательно разослан по шине всем другим процессорам/ядрам. Если не в одном из всех кэшей системы нет необходимой копии строки, то строка будет считана в кэш запросившего процессора/ядра из оперативной памяти, копия перейдёт в состояние «Exclusive». Если искомая копия обнаруживается в одном из кэшей, то соответствующий контроллер сообщит о необходимости перевести все имеющиеся в системе копии данной строки в состояние «Shared».

Если процессором/ядром был сделан запрос на запись в строку, которой нет в его кэш-памяти (промах мимо кэша при записи), то перед загрузкой необходимых данных в кэш из оперативной памяти, следует удостоверится в их актуальности, то есть узнать не имеется ли в кэшах системы модифицированной копии. В данном случае формируется последовательность операций «чтение с намеренной модификацией» (RWITM, Read With Intent To Modify). При наличии в одном из кэшей копии нужной строки, находящийся в состоянии «Modify», генерируется прерывание RWITM-последовательности и происходит обновление основной памяти, после чего состояние строки в найденном кэше меняется на «Invalid». Затем RWITM-последовательность продолжается и происходит считывание данных из оперативной памяти. В конце концов в кэше записывающего процессора/ядра появляются достоверные данные имеющие

состояние «Modify». В других кэшах системы и в основной памяти информация становится недостоверной.

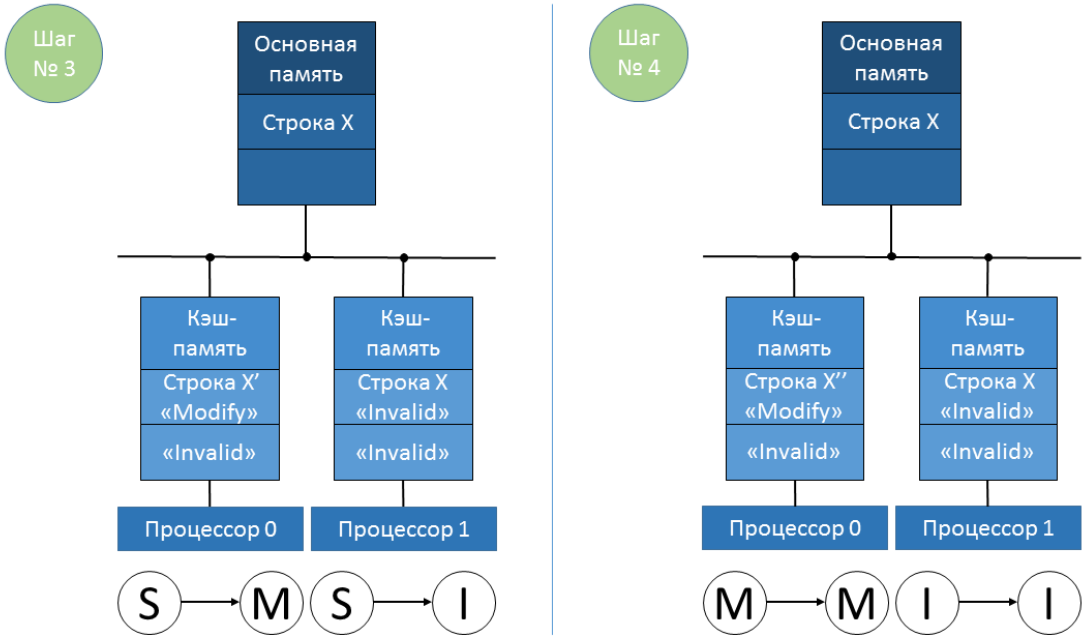
При попадании в кэш операция чтения не изменяет состояние читаемой строки. Если производится запись в строку с состоянием «Shared», то формируется широковещательный запрос с требованием перевести копии изменяемой строки в других кэшах системы в состояние «Invalid». Если состояние изменяемой строки изначально было «Exclusive», то эта строка переходит в состояние «Modify», других действий не требуется.

На рисунках 3.7 – 3.8 проиллюстрированы действия системы из двух процессоров, запрашивающих данные из строки X.



E – «Exclusive»; S – «Shared»; I – «Invalid»
Шаг № 1) Процессор 0 считывает строку X
Шаг № 2) Процессор 1 считывает строку X

Рисунок 3.7 – Последовательность смены состояний в протоколе MESI



M – «Modify»; S – «Shared»; I – «Invalid»
Шаг № 3) Процессор 0 первый раз модифицирует строку X
Шаг № 4) Процессор 0 второй раз модифицирует строку X

Рисунок 3.8– Последовательность смены состояний в протоколе MESI

3.4 Мьютекс как переменная

С точки зрения операционной системы мьютекс является обычной переменно, которая также, как и другие переменные, кэшируется при выполнении кода программы. В таком случае операцию захвата мьютекса можно рассматривать как попытку модификации данных. Следовательно, при конкуренции нескольких потоков параллельной программы за доступ к общему мьютексу появляется возможность возникновения накладных расходов: если потоки выполняются на разных процессорах/ядрах, то при изменении данных, находящихся в одном из кэшей системы, одним из потоков, соответствующие копии данных, хранящиеся в остальных кэшах системы, будут переходить в состояние «Invalid». Другими словами, при конкурентном доступе к мьютексу потоки, выполняемые на одном процессоре/ядре системы, будут провоцировать сброс соответствующих данных в кэшах остальных процессоров/ядер системы.

Ситуация усугубляется в случае, когда в кэш-строку вместе с мьютексом попадают данные из критической секции, которую он защищает.

Исходя из изложенного, очевидно, что, уменьшив количество возникающих конфликтных ситуаций при доступе к мьютексу, можно сократить количество накладных расходов параллельной программы и, как следствие, добиться оптимизации.

4 АЛГОРИТМ ОПТИМИЗАЦИИ СИНХРОНИЗАЦИИ

При конкуренции потоков в конечном итоге только один из них сможет захватить мьютекс, а остальные будут вынуждены либо продолжать пытаться захватить мьютекс в цикле (spinlock блокировка), либо, не тратить процессорное время, а освободить вычислительные ресурсы для выполнения другого потока и перейти в состояние ожидания. Обычно каждый конфликтующий поток сначала пытается захватить мьютекс с помощью spinlock'а и только в случае неудачи перестаёт расходовать временные ресурсы процессора (или ядра) погружаясь в «сон» на определённое время. Очевидно, что чем больше количество конфликтующих потоков, тем менее эффективно будет расходоваться процессорное время. Однако, если сократить количество попыток захвата мьютекса, перед тем как отправить тот или иной поток «спать», общее время простоя также сократиться. Таким образом для оптимизации процесса использования мьютексов необходима некоторая статистика возникновения возможных конфликтных ситуаций при работе с ним. Данная информация – это результат предварительного профилирования.

Реализованный алгоритм оптимизации синхронизации параллельных программ можно разделить на два этапа:

1) Профилирование.
2) Оптимизация.

4.1 Этап профилирования

На этапе профилирования производится подсчёт конфликтных ситуаций, возникающих при захвате мьютекса. Вся информация записывается в хэш-таблицу, где ключом является адрес конкретного мьютекса, а значением – частота возникновения искомой ситуации для него.

Ниже приведён код простой многопоточной программы:

Листинг 4.1 – Пример многопоточной программы

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

int shared_tmp;

void *foo (void *arg)
{
    for (int i = 0; i < 500; ++i) {
        pthread_mutex_lock (&mut);
        shared_tmp = i;
        pthread_mutex_unlock (&mut);
    }
    pthread_exit (NULL);
}

int main (int argc, char *argv[])
{
```

```
    pthread_t thr[2];

    pthread_create (&thr[0], NULL, foo, NULL);
    pthread_create (&thr[1], NULL, foo, NULL);
    pthread_join (thr[0], NULL);
    pthread_join (thr[1], NULL);

    return 0;
}
```

Для данного кода этап профилирования схематично будет выглядеть так (рисунок 4.1):

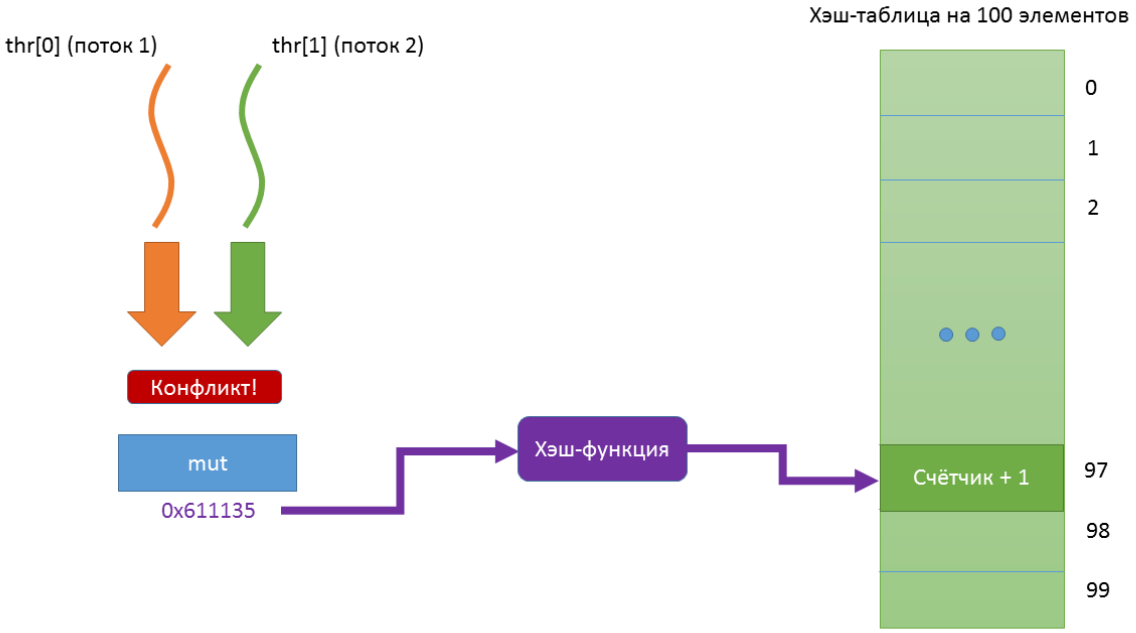


Рисунок 4.1 – Схематичное представление этапа профилирования

4.2 Этап оптимизации

На данном этапе производится оптимизация «проблемных мьютексов», то есть таких, значение в хэш-таблице для которых больше среднего. Иначе говоря, для мьютексов, за которые часто одновременно конкурируют множество потоков, будет отрегулирован показатель, отвечающий за количество попыток захвата, перед тем как отправить тот или иной поток «спать». В glibc 2.23 этот показатель является статическим и не регулируется до начала работы многопоточной программы.

5 ПРОГРАММНЫЙ ПАКЕТ «MUTEX-OPTIMIZER»

Для профилирования приложения используется методика подмены стандартных функций библиотеки pthread. Таким образом «mutex-optimizer» включает в себя две динамические библиотеки: profiler.so — позволяет строить статистику возникновения конфликтных ситуаций; optimizer.so — позволяет оптимизировать процесс захвата мьютекса по результатам профилирования. Каждая библиотека используется в определённом режиме функционирования: профилирования или оптимизации.

Выбор режима выполняется при помощи переменной среды окружения LD_PRELOAD:
[name@host] $ LD_PRELOAD=profiler.so ./prog.out — режим профилирования;
[name@host] $ LD_PRELOAD=optimizer.so ./prog.out — режим оптимизации.

Программный пакет «mutex-optimizer» условно можно разделить на две части:

- Модуль профилирования;
- Модуль оптимизации.

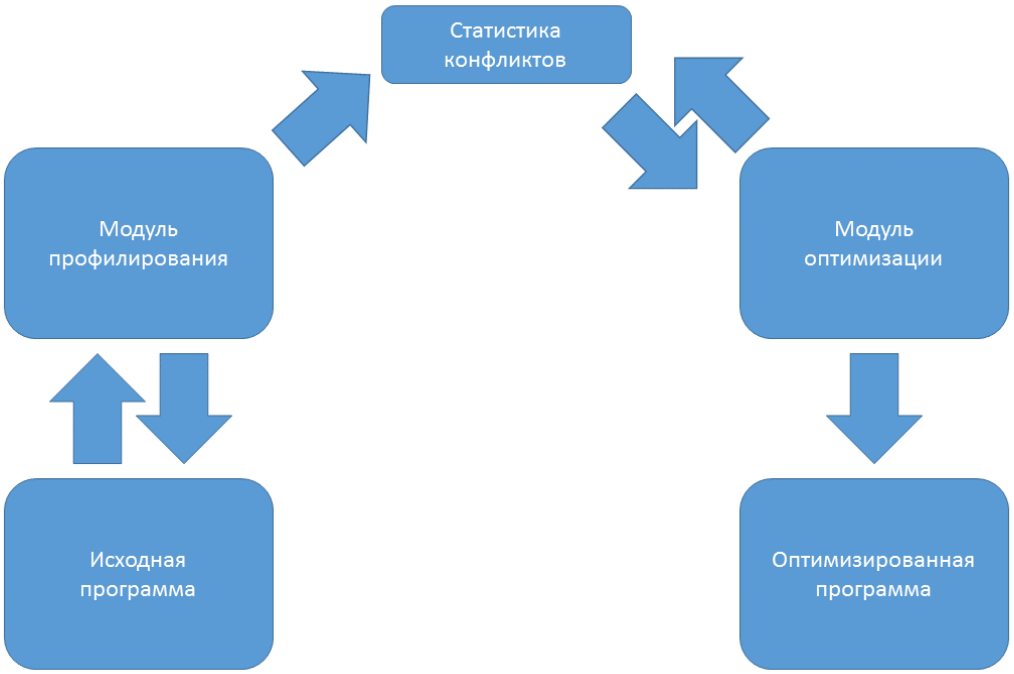


Рисунок 5.1 - Структура функционирования программного пакета «mutex-optimizer»

5.1 Структура модуля профилирования

Модуль профилирования состоит из конструктора, деструктора и функции захвата мьютекса.

Конструктор определяет адреса всех оригинальных функций, которые в дальнейшем планируется перехватывать. Также в конструкторе создаётся и инициализируется хэш-таблица, хранящая статистику частоты возникновения конфликтов при работе с мьютексами. Хэш-таблица имеет фиксированный размер (100 элементов). При коллизии выбирается следующий свободный элемент

таблицы. Хэш-функция принимает на вход адрес мьютекса и преобразовывает его в индекс от 0 до 99. В качестве результата работы хэш-функции берётся остаток от деления адреса мьютекса в десятичной системе счисления на размер хэш-таблицы (100). Каждый элемент содержит счётчик конфликтных ситуаций. API разработанной хэш-таблицы приведено в листинге 5.1.

Листинг 5.1 – API разработанной хэш-таблицы

```
// Структура элемента
struct hash_elem {
    long              ptr;   // Адрес мьютекса в 10 с.с.
    long              cont;  // Счётчик конфликтов
};

// Создание хэш-таблицы
struct hash_elem *hash_table_create (void);
// Удаление хэш-таблицы
int hash_table_free (struct hash_elem *);
// Добавление элемента в хэш-таблицу
int hash_elem_add (void *, struct hash_elem *);
// Поиск элемента в хэш-таблице
int find_hash_elem (void *, struct hash_elem *);
//   Увеличение   счётчика   конфликтов   для   конкретного
элемента
int inc_cont_elem (void *, struct hash_elem *);
//   Уменьшение   счётчика   конфликтов   для   конкретного
элемента
int dec_cont_elem (void *, struct hash_elem *);
// Печать хэш-таблицы
int hash_table_print (struct hash_elem *);
// Печать информации о мьютексах
int info_print (struct hash_elem *, long int);
// Сохранение хэш-таблицы в файл
int write_result (char *, struct hash_elem *, long int);
// Чтение из файла в хэш-таблицу
int read_result (char *, struct hash_elem *, long int *);
```

Функция захвата мьютекса – pthread_mutex_lock() (листинг 5.2) подменяет оригинальную функцию. Если мьютекс используется впервые — информация о нём добавляется в таблицу; если мьютекс свободен и используется не в первый раз — выполняется оригинал функции; если мьютекс занят и, следовательно, используется не в первый раз — увеличивается счётчик конфликтных ситуаций для данного мьютекса.

Деструктор записывает статистику во внешний файл и очищает область память, выделенную под хэш-таблицу.

Листинг 5.2– Функция сбора статистики

```
/*
 * Hooking function "pthread_mutex_lock"
 */
int pthread_mutex_lock (pthread_mutex_t *__mutex)
```

```
{
    int stat;

   /*
    * Failure lock nutex
    */
    if((stat = pthread_mutex_trylock (__mutex)) == EBUSY) {
        // Call original mutex lock
        stat = orig_pthread_mutex_lock (__mutex);
        // Increase contention counter
        inc_cont_elem ((void *) __mutex, table);
        ++avg_contention;

        return stat;
   /*
    * Successful lock mutex
    */
    }else if(find_hash_elem((void*)__mutex, table) == -1) {
       /*
        * If mutex using first time
        * Add info about mutex to hash table
        */
        hash_elem_add ((void *) __mutex, table);
        printf ("Mutex %p to created\n", __mutex);
        ++mutex_count;

        return stat;
    }

    return stat;
}
```

5.2 Структура модуля оптимизации

Модуль оптимизации также состоит из конструктора, деструктора и функции захвата мьютекса.

Конструктор определяет адреса всех оригинальных функций, которые в дальнейшем планируется перехватывать. Помимо этого, в конструкторе происходит чтение результатов предварительного профилирования в заранее созданную хэш-таблицу.

Функция захвата мьютекса – pthread_mutex_lock() (листинг 5.3) подменяет оригинальную функцию. В этой функции происходит сравнение показателей частоты возникновения конфликтов при захвате мьютекса для частного случая и для общего (средней показатель). Если показатель частного случая больше среднего, то уменьшается количество попыток захвата перед тем как отправить работающий поток «спать». Уменьшение количества попыток захвата достигается за счёт установки некоторой постоянной задержки. Деструктор очищает область память выделенную под хэш-таблицу.

Листинг 3.4 – Функция оптимизации операции захвата мьютекса

```
/*
 * Hooking function "pthread_mutex_lock"
```

```
 */
int pthread_mutex_lock (pthread_mutex_t *__mutex)
{
    int stat;

    if((stat = find_hash_elem((void*)__mutex, table))!=-1){
        if (table[stat].cont > avg_contention) {
            while((stat=pthread_mutex_trylock(__mutex))
                   == EBUSY) {
                usleep (10);
            }
            return stat;
        } else {
            return orig_pthread_mutex_lock (__mutex);
        }
    } else {
        return orig_pthread_mutex_lock (__mutex);
    }
}
```

6 РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

Эффективность разработанного пакета исследовалось на синтетических тестах (microbenchmark) (листинг 6.1).

Листинг 6.1 – Синтетический тест для разработанного программного пакета

```
pthread_mutex_t m[NUM_MUTEX];
pthread_barrier_t barrier;
int tmp;

void *foo (void *arg)
{
    pthread_barrier_wait (&barrier);
    for (int j = 0; j < NUM_MUTEX; ++j) {
        for (int i = 0; i < N; ++i) {
            pthread_mutex_lock (&m[j]);
            tmp = i;
            pthread_mutex_unlock (&m[j]);
        }
    }

    pthread_exit (NULL);
}
```

Тесты запускались на персональном компьютере фирмы ASUS модель K53S под управлением операционной системы Linux (Fedora 22), компилятор GCC версии 5.3.1. Необходимые характеристики данного компьютера можно увидеть в таблице 6.1.

Таблица 6.1 – Характеристики ноутбука фирмы ASUS модель K53S

Процессор	Intel® Core™ i5 2450M (2.5 ГГц, 35 Вт)
Количество ядер	2
Оперативная память	4 Гб SO-DIMM DDR3 1333 МГц

При тестировании использовался один мьютекс, количество параллельных ветвей – 10, количество итераций с входом в критическую секцию варьировалось: 10 000 000, 50 000 000 и 100 000 000.

Результаты измерений для указанного набора входных данных приведены в таблицах 6.2 и 6.3.

Таблица 6.2 – Результаты измерений на персональном компьютере (мьютекс – 1; потоков – 10; версия – неоптимизированная)

В секундах

№ изм. / Кол. итераций	1	2	3	4	5
10 млн.	11,279045	11,153182	11,281347	11,247811	11,440520
50 млн.	56,181755	55,755458	56,257449	55,940098	55,835272
100 млн.	111,335737	111,038542	110,287768	111,831761	111,277769

Таблица 6.3 – Результаты измерений на персональном компьютере (мьютекс – 1; потоков – 10; версия – оптимизированная)

В секундах

№ изм. / Кол. итераций	1	2	3	4	5
10 млн.	4,735136	4,668115	4,590554	4,667527	4,626296
50 млн.	23,134244	22,690051	22,920487	23,057434	22,693934
100 млн.	46,025612	45,078907	45,818411	45,716328	45,780343

На рисунке 6.1 представлен график, построенный согласно таблиц 6.2 и 6.3.

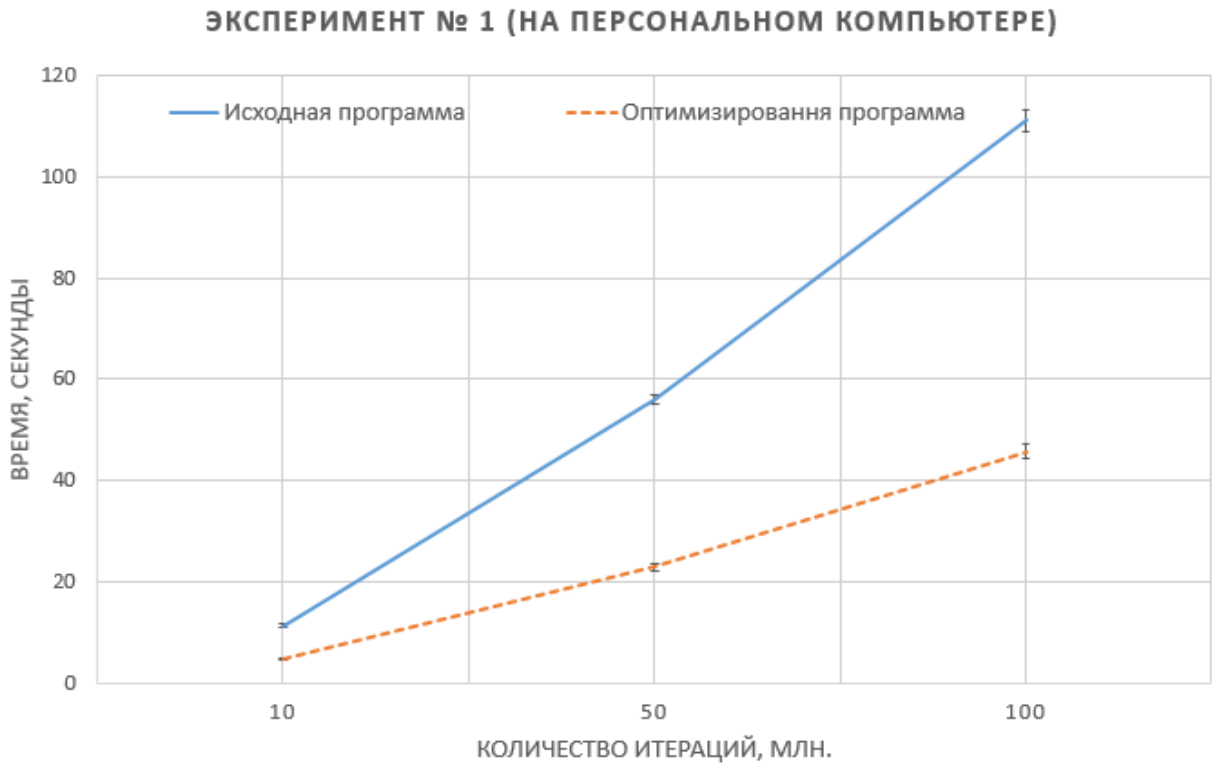


Рисунок 6.1 – График зависимости времени выполнения тестовой параллельной программы от количества итераций со входом в критическую секцию

Как видно из графика, при разных входных данных оптимизированная версия программы показывает лучшее время. Максимальное ускорение составило 2.74, среднее – 2.44, минимальное – 2.29.

Этот же тест запускался при других входных данных: мютекс – 1, количество итераций со входом в критическую секцию – 10 000 000, количество параллельных ветвей варьировалось от 10 до 30 с шагом 10.

Результаты измерений для указанного набора входных данных приведены в таблицах 6.4 и 6.5.

Таблица 6.4 – Результаты измерений на персональном компьютере (мьютекс – 1; итераций 10 млн.; версия – неоптимизированная)

В секундах

№ изм. / Кол. потоков	1	2	3	4	5
10	11,279045	11,153182	11,281347	11,247811	11,440520
20	22,534934	22,631339	22,658104	22,556417	22,422885
30	33,581070	33,576202	33,795028	33,663258	33,755966

Таблица 6.5 – Результаты измерений на персональном компьютере (мьютекс – 1; итераций – 10 млн.; версия – оптимизированная)

В секундах

№ изм. / Кол. потоков	1	2	3	4	5
10	4,735136	4,668115	4,590554	4,667527	4,626296
20	9,352720	9,204148	9,299738	9,211307	9,284673
30	14,247778	14,336100	14,407313	14,450508	14,256084

На рисунке 6.2 представлен график, построенный согласно таблиц 6.4 и 6.5.

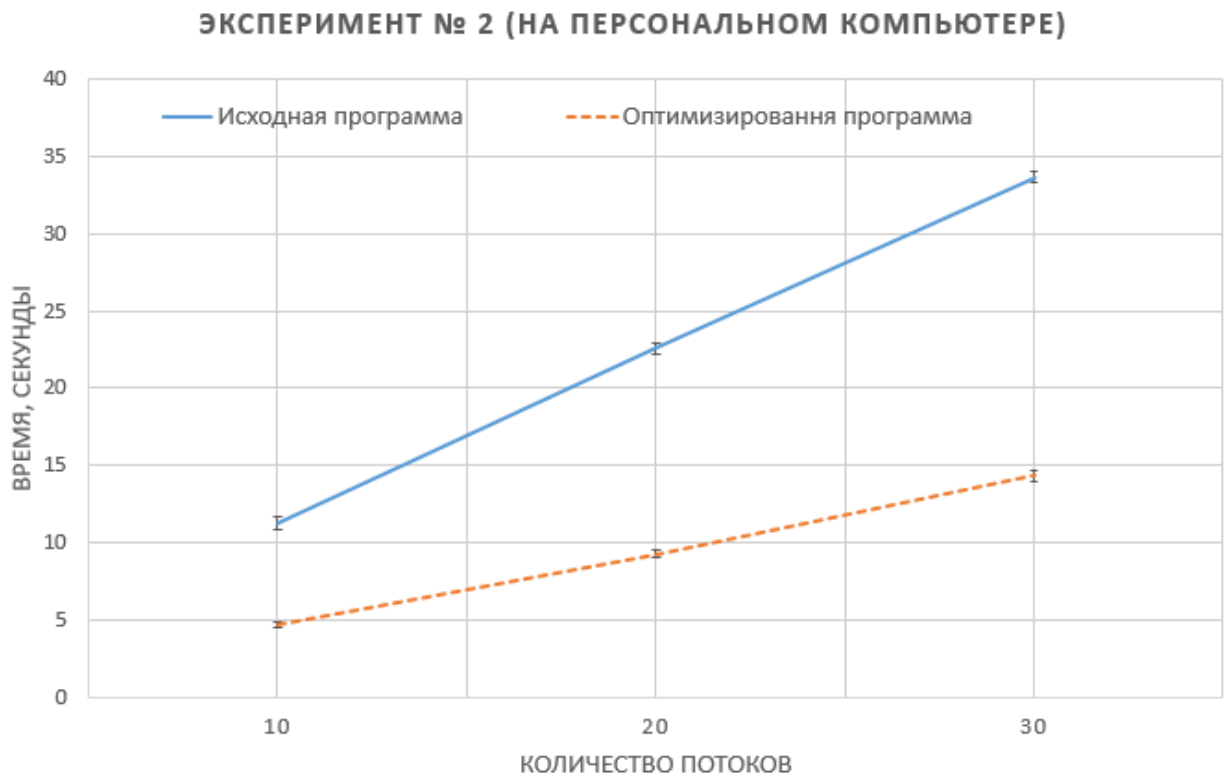


Рисунок 6.2 – График зависимости времени выполнения тестовой параллельной программы от количества параллельных ветвей в ней

Как видно из графика, при разных входных данных оптимизированная версия программы показывает лучшее время. Максимальное ускорение составило 2.74, среднее – 2.37, минимальное – 2.13.

Тесты также запускались на кластере Jet, компилятор GCC версии 4.8.3, операционная система Linux (Fedora 20). Необходимые характеристики данного кластера можно увидеть в таблицах 6.6 и 6.7. Кластер Jet укомплектован 18 вычислительными узлами, управляющим узлом, вычислительной и сервисной сетями связи, а также системой бесперебойного электропитания.

Таблица 6.6 – Конфигурация вычислительного узла кластера Jet

Процессор	Intel® Xeon® CPU E5420 (2.50 ГГц) x2
Количество ядер	8 x 2 = 16
Оперативная память	8 GB (4 x 2GB PC-5300)
Жесткий диск	SATAII 500GB (Seagate 500Gb Barracuda)

Таблица 6.7 – Конфигурация управляющего узла кластера Jet

Процессор	Intel® Xeon® CPU E5420 (2.50 ГГц) x2
Количество ядер	8 x 2 = 16
Оперативная память	16 GB (8 x 2GB PC-5300)
Жесткий диск	3 x SATAII 500GB (Seagate 500Gb Barracuda)

При первом эксперименте входные данные были следующие: мьютекс – 1, количество итераций со входом в критическую секцию – 10 000 000, количество параллельных ветвей – 1, 2, 3, 4, 5, 6, 7, 8.

Результаты измерений для указанного набора входных данных приведены в таблицах 6.8 и 6.9.

Таблица 6.8 – Результаты измерений на кластере Jet (мьютекс – 1; итераций 10 млн.; версия – неоптимизированная)

В секундах

№ изм. / Кол. потоков	1	2	3	4	5
1	0,203873	0,199321	0,200230	0,213083	0,198243
2	2,292837	2,343769	2,303261	2,319740	2,421753
3	5,002762	4,987394	4,899672	5,017580	5,026781
4	9,887333	9,932049	9,900329	9,802344	9,852180
5	7,443875	7,892232	7,545700	7,567202	7,400987
6	10,993763	10,973394	11,097433	11,113211	10,875644
7	10,013762	10,247494	10,334764	10,232323	10,348788
8	14,000340	13,999324	13,921077	13,893420	13,702083

Таблица 6.9 – Результаты измерений на кластере Jet (мьютекс – 1; итераций – 10 млн.; версия – оптимизированная)

В секундах

№ изм. / Кол. потоков	1	2	3	4	5
1	0,221021	0,223301	0,230111	0,221488	0,231085
2	1,873540	1,987600	1,800872	1,937210	1,701198
3	3,001451	2,87209	2,962308	2,832077	2,739998
4	4,009821	3,976531	3,987871	3,889919	4,000131
5	4,989888	4,807629	4,913844	5,091870	4,843399
6	5,953399	6,001761	5,973333	5,890091	6,011901

7	7,000011	6,876093	6,980001	6,990001	6,798987
8	7,79219	7,777172	7,811092	8,102309	7,900182

На рисунке 6.3 представлен график, построенный согласно таблиц 6.8 и 6.9.

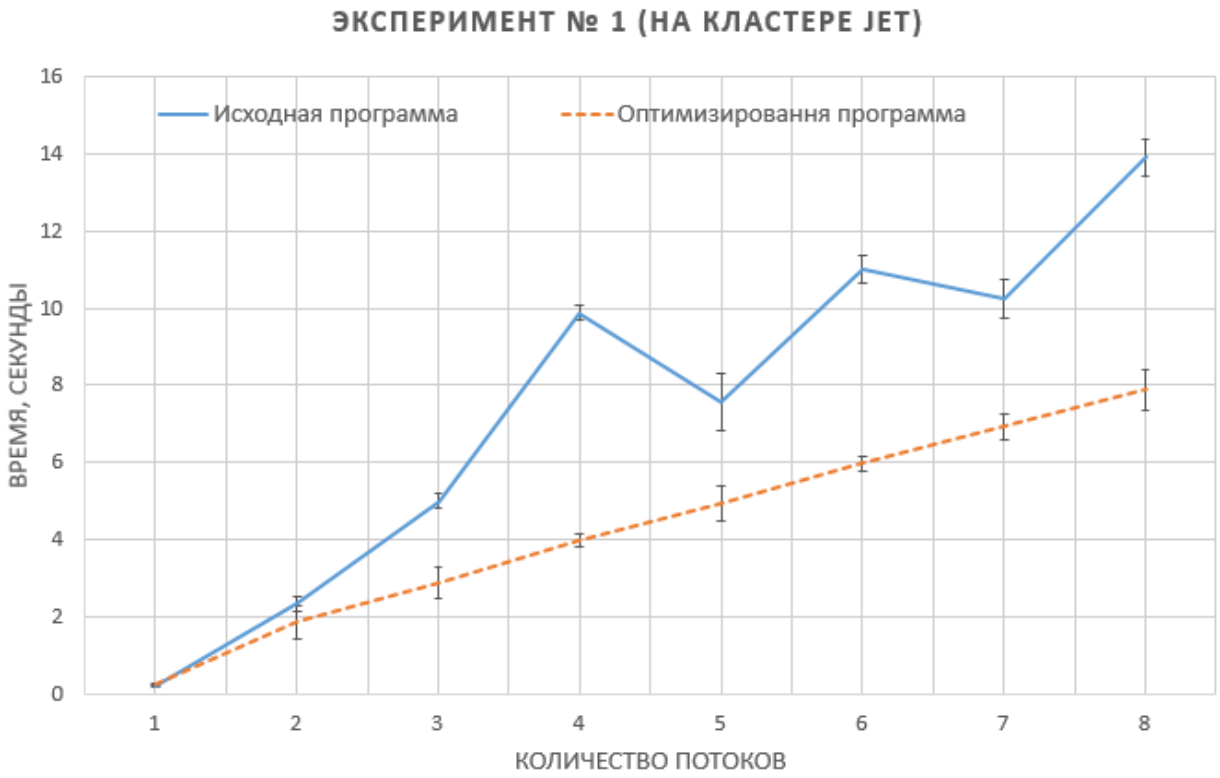


Рисунок 6.3 – График зависимости времени выполнения тестовой параллельной программы от количества параллельных ветвей в ней

Как видно из графика, при разных входных данных оптимизированная версия программы показывает лучшее время, за исключением работы при одном потоке, в этом случае конфликтов за доступ к мьютексу нет и, следовательно, возникают накладные расходы на инициализацию хэш-таблицы и освобождение памяти, выделенной под неё. Максимальное ускорение составило 2.57, среднее – 1.78, минимальное – 0.92. Нелинейный характер кривой, показывающий время выполнения неоптимизированной версии тестовой программы, объясняется особенностями работы планировщика.

При втором эксперименте входные данные были следующие: мьютекс – 1, количество итераций со входом в критическую секцию – 10 000 000, количество параллельных ветвей – 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

Результаты измерений для указанного набора входных данных приведены в таблицах 6.10 и 6.11.

Таблица 6.10 – Результаты измерений на кластере Jet (мьютекс – 1; итераций 10 млн.; версия – неоптимизированная)

В секундах

№ изм. / Кол. потоков	1	2	3	4	5
10	18,977321	18,873401	19,003003	18,881012	19,108730
20	39,323409	39,400092	39,109821	39,509811	38,999821
30	53,763122	54,097234	53,987222	54,108921	54,012863
40	69,131581	70,012987	69,562111	69,876300	69,421999
50	85,623709	85,143601	84,991091	85,338761	85,721301
60	103,132098	102,973033	103,455532	104,937653	102,723075
70	136,235099	133,765006	135,999760	134,008754	133,331311
80	139,326000	138,231288	140,109232	138,008887	137,121516
90	151,321965	149,777652	149,108923	150,150672	149,223586
100	178,555265	180,008346	177,121677	178,340098	178,476320

Таблица 6.11 – Результаты измерений на кластере Jet (мьютекс – 1; итераций – 10 млн.; версия – оптимизированная)

В секундах

№ изм. / Кол. потоков	1	2	3	4	5
10	10,102333	9,832991	10,001222	10,193120	9,983701
20	20,721256	19,863453	20,008764	20,654442	20,346521
30	31,123084	31,771589	30,700064	30,013586	31,000080
40	41,875002	41,171615	39,846762	42,239899	43,008634
50	58,776210	57,998347	59,008462	58,364803	58,874763
60	68,234812	68,895623	69,777489	71,437611	70,823418
70	80,234394	81,342897	82,378102	79,989967	81,823623
80	98,762389	99,092351	98,762319	97,347891	100,008760
90	111,211311	110,234712	109,326834	110,347892	112,340912
100	124,329784	126,234789	125,891232	123,734760	124,000327

На рисунке 6.4 представлен график, построенный согласно таблиц 6.10 и 6.11.

Как видно из графика, при разных входных данных оптимизированная версия программы показывает лучшее время. Максимальное ускорение составило 1.84, среднее – 1.34, минимальное – 1.50.

Для измерений были построены доверительные интервалы [6], которые присутствуют на всех графиках, построенных в данной БР. Коэффициент Стьюдента равен 8.610.

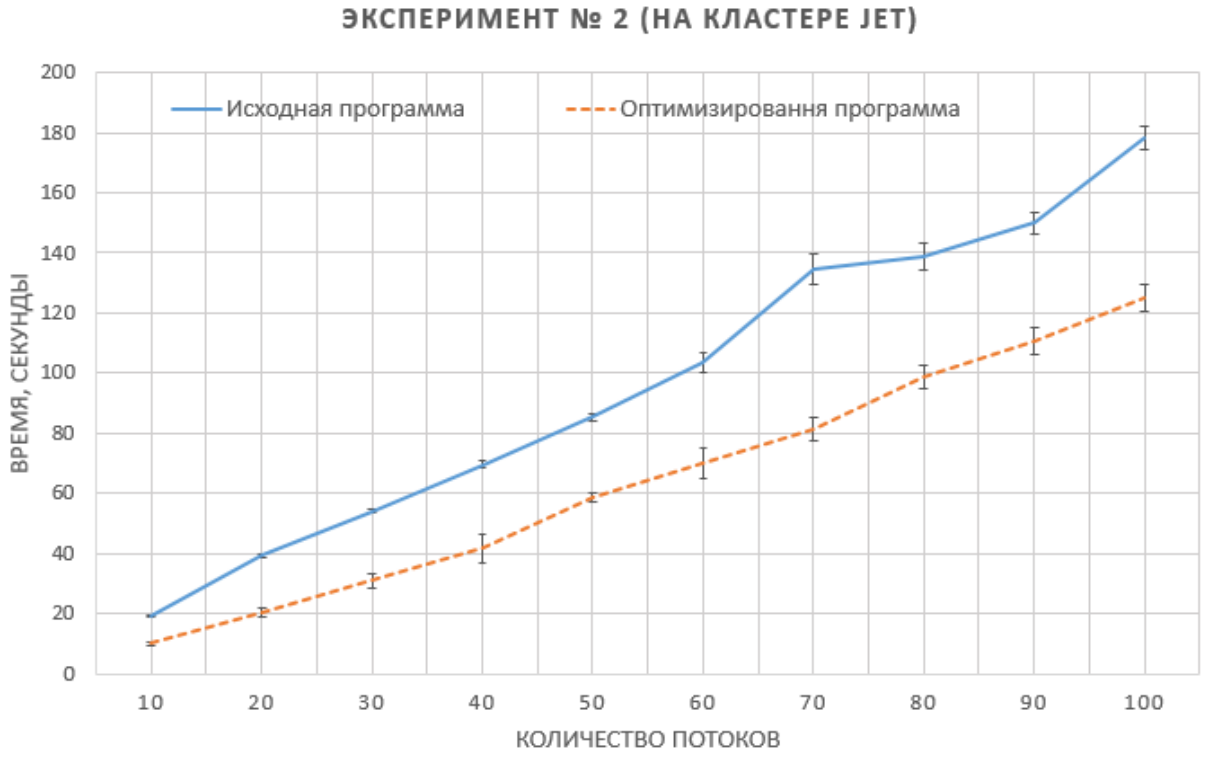


Рисунок 6.4 – График зависимости времени выполнения тестовой параллельной программы от количества параллельных ветвей в ней

7 ЗАКЛЮЧЕНИЕ

В ходе БР поставленные цели были достигнуты, задачи – выполнены. Экспериментально подтверждена эффективность разработанного программного пакета «mutex-optimizer», проведён анализ характера получившихся графиков, рассчитано минимальное, среднее и максимальное ускорение для разных конфигураций системы и входных данных. Изучены основы оптимизации синхронизации параллельных программ для вычислительных систем с общей памятью. Освоена методика профилирования работы мьютексов в пользовательском пространстве операционной системы.

ПРИЛОЖЕНИЕ А

(справочное)
Библиография

1 Herlihy M., Shavit N. The Art of Multiprocessor Programming, Revised Reprint // Elsevier – 2012.
2 Бовет. Д., Чезати М. Ядро Linux, 3-е изд.: Пер. с англ. // Спб. БХВ-Петербург, 2007. – 1104 с.: ил.
3 Лав Р. Ядро Linux: описание процесса разработки, 3-е изд. – М.: Вильямс // 2015. – 496 с.
4 John L. Hennessy; David A. Patterson (16 September 2011). Computer Architecture: A Quantitative Approach. Elsevier. // Retrieved 25 March 2012.
5 Drepper U. Memory part 2: CPU caches. // Retrieved from the internet on 10 November 2010. – С. 1-53.
6 Курносов М.Г., Пазников А.А. Основы теории функционирования распределенных вычислительных систем. // Новосибирск: Автограф, 2015. – 52 с.

ПРИЛОЖЕНИЕ Б

(рекомендуемое)
Наиболее употребляемые текстовые сокращения

ВС – вычислительная система
БР – бакалаврская работа
RWITM – Read With Intent To Modify
MRU – Most Recently
MESI – Modify Exclusive Shared Invalid
LRU – Least Recently Used
RR – Random Replacement