Санкт-Петербургский государственный университет
Математическое обеспечение и администрирование информационных систем
Системное программирование

Бабанов Пётр Андреевич

Многопоточное исполнение MAK-обфусцированных программ

Бакалаврская работа

Научный руководитель:
ст. преподаватель
Баклановский М. В.
Рецензент:
инженер-программист ООО "ИнтеллиДжей Лабс"
Мордвинов Д. А.

Санкт-Петербург
2018

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Babanov Petr Andreevich

Multuthreaded execution of MAC-obfuscated programs

Graduation Thesis

Scientific supervisor:

Senior Lecturer

Maxim Baklanovsky

Reviewer:

IntelliJ Labs software engineer Dmitry Mordvinov

Saint-Petersburg

2018

Введение

Практически любое современное приложение является многопоточным. Без использования многопоточности нельзя создать даже простое “оконное” приложение, в котором бы интерфейс не блокировался на время обработки данных. Без многопоточности не обойтись при создании простого приложения-чата для общения двух пользователей: один поток необходим для отправки сообщений, второй для приема. Кроме того, во многих случаях многопоточность позволяет ускорить работу приложения. Подробно такие примеры рассмотрены в [1].

Важной особенностью многопоточных приложений, отличающей их от классических однопоточных (последовательных) приложений, является наличие разделяемых ресурсов. Разделяемый ресурс может представлять из себя как переменные, структуры, так и системные ресурсы, файлы, события и т.п.

В основе любого приложения лежит алгоритм, который, зачастую, является интеллектуальной собственностью разработчика (отдельного человека или целой компании). В то же время, предоставляя приложение пользователям, разработчик передает все алгоритмы, связанные с ним в формате исполняемого файла. В связи с этим остро встает вопрос защиты этой собственности от взлома с целью модификации или кражи, в том числе посредством реверс-инжиниринга – дизассемблирования, статического и динамического анализа кода приложения.

Для защиты от реверс-инжиниринга применяются различные приемы обфускации. Обфускация – это запутывание кода (как

непосредственно на уровне машинных команд, так и на высоком уровне языков программирования), организация исполняемого кода таким образом, что его анализ становится крайне затруднительным. Подробный обзор методов обфускации приводится в статье [2].

В течении нескольких последних лет на кафедре системного программирования СПбГУ разрабатывается компилятор МАК для языка программирования Си. Одной из его отличительных особенностей является простота применения обфускации на уровне машинных команд. В данный момент компилятор МАК поддерживает работу с однопоточными приложениями с обфускацией и с многопоточными приложениями без обфускации

Главным отличием компилятора МАК является разбиение кода приложения на линейные участки – фрагменты кода, все инструкции внутри которых выполняются последовательно.

Такой подход позволяет использовать множество приемов обфускации на уровне ассемблера, в частности, изменять код во время выполнения программы, модифицируя линейные участки, которые в данный момент не используются. Под модификацией можно понимать как собственно изменение фрагмента (XOR с каким-то известным участком или другая модификация), так и копирование из другого участка памяти, либо преобразования в обратную сторону.

Очевидно, что это накладывает серьезные ограничения на многопоточные программы: в случае, если модификация некоторого линейного участка находится внутри многопоточной части программы, то преобразуемый участок будет модифицирован несколько раз, и это во многих случаях нарушает корректность работы программы. Например, двойное применение XOR с некоторым ключом даст

исходное состояние модифицируемого линейного участка. Кроме того, без применения методов синхронизации для линейных участков возможна ситуация, когда один поток начнет модифицировать линейный участок, выполняемый другим потоком. Все это делает распараллеливание без использования дополнительных механизмов синхронизации невозможным.

Для того, чтобы использовать обфускацию в многопоточных приложениях, необходим модуль, отвечающий за синхронизацию доступа к линейному участку и его модификации.

Постановка задачи

Целью данной работы является создание прототипа модуля синхронизации для компилятора МАК. Для достижения данной цели можно выделить следующие задачи.

- Изучить способы низкоуровневой синхронизации многопоточных приложений.
- Разработать шаблоны реализации модуля синхронизации линейных участков.
- Выполнить прототипную реализацию данных шаблонов в инфраструктуре компилятора МАК для оценки их воздействия на время выполнения приложений.
- Провести тестирование и апробацию полученных прототипов.

Глава 1. Обзор

Обфускация

Подробный обзор методов обфускации и их классификация приведены в статье [2]. Для рассмотрения в рамках работы были выбраны приемы, использующие модификацию исполняемого кода в момент выполнения программы. Именно эти приемы делает невозможным многопоточное выполнение приложений без использования дополнительных методов синхронизации.

Общий принцип работы таких приемов обфускации можно описать следующим образом: до тех пор, пока фрагмент кода не исполняется, на его месте находится различный “мусор”, набор байтов, не несущий никакой смысловой нагрузки. При этом код, который должен там находиться, хранится в другом месте – сегменте данных или буфере внутри сегмента кода. Как только возникает необходимость передать управление на участок, заполненный “мусором” – происходит копирование на это место нужного кода. А по окончании выполнения этот участок снова замещается “мусором”. Вместо замещения выполняемого кода возможны и другие методы модификации, например, шифрование с использованием ключа.

Многие современные алгоритмы шифрования используют схему с несколькими раундами шифрования. Раундом шифрования в этом случае называют некоторую последовательность преобразований, которая выполняется последовательно и которую нельзя прервать. При этом по окончании раунда не обязательно получается расшифрованный или окончательно зашифрованный фрагмент.

При использовании шифрования с несколькими раундами возможно сокращение времени подготовки шифруемого фрагмента - начальные раунды можно выполнять заранее, а окончательные - только если переход на этот фрагмент действительно необходим.

Компилятор МАК

Данная работа выполнена в контексте проекта компилятора МАК. Особенностью этого компилятора является возможность применения низкоуровневых механизмов обфускации, однако, как говорилось выше, не все методы обфускации применимы в многопоточных приложениях без дополнительных механизмов синхронизации.

В процессе работы компилятор МАК создает ассемблерный код, разбитый на линейные участки. Такое промежуточное состояние удобно для применения различных методов обфускации и встраивания различных дополнительных модулей. На этом этапе и должны добавляться механизмы синхронизации и модификаторы линейных участков. Каждый линейный участок на этом этапе имеет свой номер, в явном виде указываются линейные участки, на которые будет осуществлен переход после выполнения текущего участка, а также команда, с помощью которой будет осуществлен переход. Таким образом возможно изменять порядок выполнения линейных участков.

Многопоточные приложения в ОС Windows

Работа многопоточных приложений в операционной системе Windows подробно описана в большом количестве литературы. Например, в [3] детально рассмотрена система потоков с точки зрения

программиста, создающего пользовательские приложения: функции работы с потоками на высоком уровне с использованием библиотек Win32 API. Там же рассматриваются стандартные методы работы с разделяемыми ресурсами, в частности: с глобальными переменными, объектами ядра (файлы, семафоры, мьютексы, события и др.). В книге [4] рассматривается работа многопоточных приложений с точки зрения операционной системы: рассматриваются системные структуры, описывающие потоки, механизмы переключения потоков.

Стандартные примитивы синхронизации

При работе с разделяемыми ресурсами выделяют критические секции – фрагменты программы, выполнение которых невозможно одновременно несколькими потоками. Для ограничения количества потоков, которые выполняют критическую секцию, используются примитивы синхронизации, например, атомарные операции – операции, которые либо выполняется полностью, либо не выполняется совсем. Например, сложение двух чисел разбивается на три этапа: загрузка из памяти в регистры, сложение, выгрузка в память. В случае неатомарного выполнения возможно прерывание потока между этими этапами и, как следствие – возникновение некорректных результатов работы. Эта проблема получила название «ABA problem» и описывается в документации Intel [5]. Подробно атомарные операции рассматриваются в статье [6]. На базе атомарных операций строятся спинлоки и более высокоуровневые примитивы синхронизации – мьютексы, семафоры.

Все упомянутые примитивы синхронизации не подходят для решения поставленной задачи, так как важной особенностью

механизма, необходимого для синхронизации доступа и модификации линейных участков, является контроль состояния разделяемого ресурса (в данном случае - линейного участка). Однако их можно использовать в качестве частей механизма для ограничения доступа, когда это необходимо.

Глава 2. Обзор средств низкоуровневой синхронизации в архитектуре x86

Важной особенностью архитектуры x86 является то, что даже одна команда на языке ассемблера может быть прервана для выполнения другой операции. Для решения этой задачи в наборе команд x86 существует префикс `LOCK` (opcode 0xF0). Но не все команды могут быть использованы с этим префиксом. Так, например, команды `CMPXCHG`, `CMPXCHG8B`, `CMPXCHG16B` (только для X64) выполняются атомарно, кроме того, атомарно могут быть выполнены операции такого рода, как `SUB, ADD, INC, DEC, AND, OR` и др.

Операция `CMPXCHG` используется для организации спинлоков и более высокоуровневых механизмов синхронизации. С помощью этой команды, однако, невозможно организовать механизм, который бы изменял один фрагмент памяти (переменную) в зависимости от состояния другой. Только в случае, если они расположены последовательно, этого можно добиться с помощью команды `CMPXCHG8B`, которая проводит операцию сравнения с заменой для 8 байт сразу.

Но эти команды не дают возможности организации семафоров и подсчета числа потоков, исполняющих критическую секцию, и изменения флагов без их проверки. Для этого необходимо использование префикса `LOCK` перед командами `SUB, ADD, INC, DEC.`

Также возможно использование префикса `LOCK` с командами логических операций `XOR, AND, OR, NOT` и т.д.

С помощью этих команд можно реализовать более высокоуровневые примитивы синхронизации, такие как мьютексы, спинлоки, семафоры и другие.

Важной особенностью является то, что нет атомарной операции, которая бы изменяла один участок памяти в зависимости от значения другого. Однако в некоторых случаях возможно реализовать такое действие с помощью команды `CMPXCHG8B`. В случае если в сумме операнды имеют размер не более 8 байт, размещаются в памяти подряд, и заданы состояния обоих участков, при которых происходит замена значения одного из них.

Глава 3. Изучение возможных вариантов архитектуры

В начале рассмотрим общий принцип работы требуемого модуля (рис. 1)

Рис. 1

Схема работы требуемого модуля

Механизмы, работающие внутри блоков, могут быть различными. Рассмотрим возможные варианты:

Для того чтобы избежать коллизий во время работы различных потоков над одним линейным участком можно выделить для каждого потока свой буфер, в который он будет копировать зашифрованный линейный участок, после чего расшифровывать, выполнять его, и, по окончании работы, очищать и освобождать буфер. Такой метод отличается своей простотой. При этом он требует большого количества памяти. Кроме того, не всегда известно точное число потоков, которые будут работать с шифруемым линейным участком, а значит – не всегда получится заранее подготовить нужное количество буферов, а это

повлечет простаивание одних потоков в ожидании пока другие освободят буфера.

Другие принципы основываются на совместном использовании одного участка памяти разными потоками. Рассмотрим такие механизмы подробнее.

Архитектура с флагами состояния и блокировки

Одним из очевидных решений является реализация прототипа на основе флагов, отвечающих за состояние, блокировку доступа и счетчика, контролирующего количество потоков, выполняющих в данный момент линейный участок. Рассмотрим данную архитектуру подробнее. На рис. 2 изображена схема модуля синхронизации перед началом выполнения линейного участка.

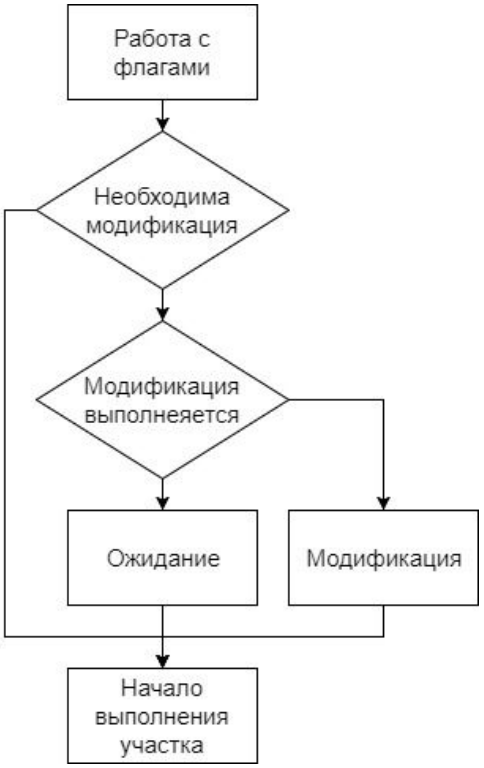

Рис.2

Алгоритм синхронизации перед началом выполнения участка

При этом блок “Модификация” может быть не один (в случае если используется несколько раундов шифрования).

Схема модуля синхронизации по окончании представлена на рис. 3

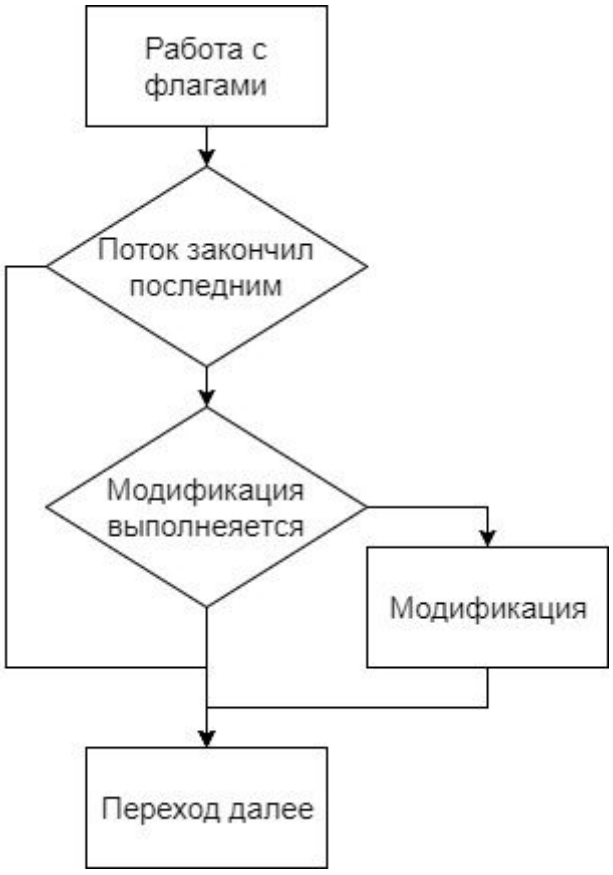

Рис. 3

Алгоритм синхронизации по окончании выполнения участка

Архитектура с изменяемым указателем

Так как компилятор МАК генерирует программу, состоящую из отдельных линейных участков, то можно аналогично разбить механизм синхронизации на отдельные линейные участки и, используя переход по указателю, хранящемуся в памяти, организовать выполнение модификаций и ожидание потоками подготовки линейного участка к выполнению. При этом все равно необходим счетчик, отвечающий за

количество потоков, выполняющих модифицируемый участок, для того чтобы предотвратить модификацию в момент исполнения другим потоком. Первая часть представлена на рис. 4

Рис. 4

Алгоритм синхронизации перед началом выполнения участка

Тут важно отметить, что в случае работы с несколькими раундами шифрования возможно простое масштабирование такого модуля для работы с большим числом состояний участка. Вторая часть модуля изображена на рис. 5

Рис. 5

Алгоритм синхронизации по окончании выполнения участка

Архитектура с внешним управлением состояния потока

В основе данного решения лежит идея вывести всю работу с модификацией линейных участков в отдельный поток. При этом вся работа этого потока будет состоять в проверки каждого из флагов и проверки соответствия состояния линейного участка этим флагам.

Таким образом взаимодействие между потоками реализуется просто на атомарных увеличениях и уменьшениях одного флага для каждого участка, а всю работу по подготовке линейного участка к выполнению берет на себя новый вспомогательный поток.

Архитектура с использованием неблокирующей синхронизации

Все архитектуры, рассмотренные выше, допускали шифрование и дешифрование только одним потоком. Это является их узким местом: пока один поток работает над линейным участком, остальные простаивают в ожидании. Для того чтобы избавиться от него, можно

разрешить нескольким потокам одновременно заниматься модификацией одного линейного участка.

Таким образом невозможно на данном этапе однозначно выделить наилучший вариант архитектуры. Поэтому в рамках данной работы были реализованы все вышеперечисленные варианты и на практике оценены их положительные и отрицательные стороны.

Глава 4. Реализация прототипов

Архитектура с использованием флагов

Для реализации прототипа с использованием архитектуры, основанной на флагах состояния используются следующие поля.

Count – поле-счетчик, отображающее количество потоков, исполняющих в данный момент модифицируемый участок, либо ожидающих окончания модификации участка для его выполнения.

Spin – флаг, используемый при организации спинлока для предотвращения многократной модификации линейного участка.

Stat – флаг, необходимый для отслеживания текущего состояния линейного участка внутри критической секции.

Схема работы прототипов представлены на схемах на рис. 6 и 7

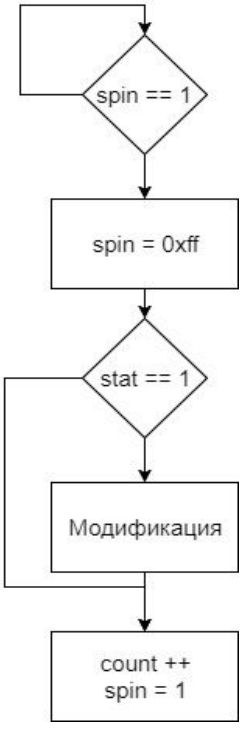

Рис. 6

Алгоритм синхронизации перед началом выполнения линейного участка

Алгоритм работы по окончании аналогичен с точностью до изменения флага stat и модификации.

При реализации прототипа данной архитектуры была обнаружена следующая особенность: данное решение плохо масштабируется при использовании шифрования с несколькими раундами.

Архитектура с использованием изменяющегося указателя

Введем следующие обозначения:

Count – поле, необходимое для подсчета количества потоков, выполняющих модифицируемый линейный участок, либо ожидающих окончания модификации.

Ptr - поле-указатель, по нему осуществляется переход при синхронизации перед началом выполнения линейного участка. Обозначим его возможные значения как lu, s, d – собственно переход на сам линейный участок, спинлок и модификация перед выполнением соответственно. Важной деталью является то, что поля count и ptr располагаются подряд друг за другом.

Синхронизация перед началом линейного участка основывается на четырехбайтовом поле ptr. Кроме этого в состав механизма синхронизации входит линейный участок, реализующий спинлок. Количество возможных значений ptr зависит от количества применяемых модификаторов.

Начальное состояние ptr зависит от того, в каком состоянии изначально находится шифруемый линейный участок. Алгоритм работы представлен на рисунке 7.

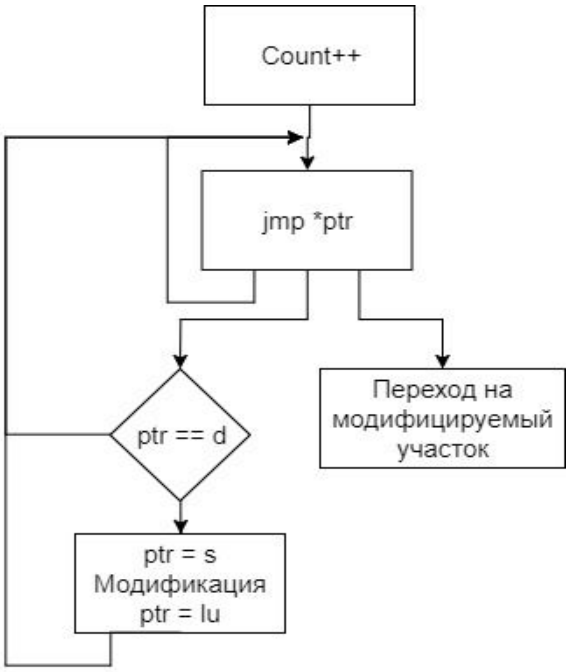

Рис. 7

Алгоритм работы перед началом выполнения участка

В случае применения нескольких раундов шифрования добавляются модули, аналогичные применяемым с одним раундом.

Порядок работы по окончании выполнения линейного участка представлен на рисунке 8.

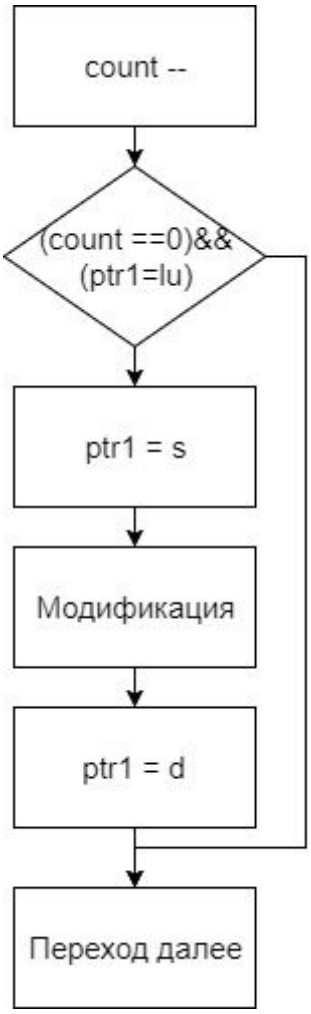

Рис. 8

Алгоритм работы по окончании выполнения участка

В случае применения нескольких раундов шифрования для каждого из них добавляется по два модуля: первый перед началом очередного раунда проверяет состояние счетчика count, и если он нулевой то выполняется модификация. Если он не нулевой - то алгоритм открывает спинлок и переходит к дальнейшему выполнению программы. Эти модули должны быть выполнены между участками “Модификация” и “Ptr = d” на рис. 7.

Архитектура с использованием диспетчера

Для определенности будем называть потоки, создаваемые внутри программы в соответствии с ее логикой работы и выполняющие код самой программы – основными потоками, а поток, добавленный для управления состояниями модифицируемых участков – вспомогательным

Для каждого модифицируемого линейного участка заводится 4 байтовое поле flag. Основные потоки перед началом выполнения модифицируемого участка (непосредственно или заранее) атомарно инкрементируют флаг соответствующего участка. По окончании выполнения либо после того, как становится очевидно, что участок выполняться не будет – основной поток должен атомарно декрементировать флаг участка.

Для защиты от выполнения модифицированного участка до того, как он будет приведен в состояние готовности, перед его началом добавляется спинлок, управляемый дополнительным потоком.

Архитектура с использованием неблокирующей синхронизации

Неблокирующая синхронизация в данном случае подразумевает возможность модификации отдельных фрагментов линейного участка. В связи с этим необходимо отслеживать состояние каждого отдельного минимального фрагмента, который можно модифицировать отдельно. Для этого будем использовать массив, размер которого равен количеству минимальных фрагментов, которые можно модифицировать по отдельности. Кроме этого, для того чтобы управлять процессом модификации каждого отдельного фрагмента

потребуется еще один массив. В нем будем держать указатель на необходимую в данный момент следующую модификацию. Также, для того чтобы избежать начала исполнения неподготовленного участка добавим спинлок перед переходом на него, а также счетчик фрагментов, готовых к исполнению. Это необходимо для того чтобы каждый поток имел информацию о том, готов ли участок полностью.

В начале работы алгоритма в регистр dl устанавливается значение 0 или 1 которое говорит о том, шифрование или расшифровка производится данным потоком. Общая схема работы алгоритма представлена на рисунке 9.

Рис. 9

Общая схема работы алгоритма с неблокирующей синхронизацией

Также важно отметить что алгоритмы работы при шифровании и дешифровка будут схожи, и отличаются только тем, как зависит исполнение модификации от состояния регистра dl.

Глава 5. Тестирование и сравнение

Для проверки корректности работы механизмов синхронизации необходимо выбрать такой алгоритм, который был бы, с одной стороны, достаточно прост, а с другой стороны – легко масштабировался для разного числа потоков. Как соответствующий этим требованиям был выбран алгоритм сортировки слияниями. С одной стороны -- он легко выполняется параллельно, с другой стороны – его можно модифицировать так, что, изменяя всего лишь одно значение, можно изменять число потоков, которое будет выполнять данную сортировку. В качестве этой величины была выбрана глубина рекурсии, на которой заканчивается распараллеливание, и количество потоков, на которых проверялась корректность было 2^n где n=1,...,6. Для получения заметной разницы во времени выполнения размер массива был установлен в 16000 элементов типа int. Для того, чтобы максимально снизить влияния фоновых приложений, запуск производился 300 раз и находилось среднее время выполнения.

Поскольку необходимо было проверить и корректность работы синхронизирующего механизма, требовалось предусмотреть, чтобы при переходе в неподготовленный фрагмент происходил вызов исключения. Для этого в качестве модификации использовалось замещение модифицируемого участка байтами 0xCC, при переходе на которые произойдет исключение и программа завершит работу.

Первое тестирование представляло собой шифрование линейного участка длиной 12 байт с одним раундом шифрования, результаты представлены в таблице 1.

Количество потоков	1	2		3		4		5		6	
	M	M	D	M	D	M	D	M	D	M	D
2	15908	15946	93	16026	197	16031	78	15913	59	16063	82
4	7941	8032	30	8272	415	8082	61	7970	84	8246	171
8	3985	4036	13	4108	111	4128	58	4005	39	4323	181
16	2006	2049	15	2198	158	2232	109	2020	40	2457	512
32	1016	1063	19	1214	78	1365	75	1024	6	1795	724
64	516	566	5	1004	104	1103	61	528	22	1363	765

Таблица 1

1– Без шифрования, 2 – Разделенный буфер 3 – Неблокирующая синхронизация, 4 – С использованием флагов, 5 – Изменяемый указатель, 6 – с использованием диспетчера. M – Среднее время выполнения, D – Среднеквадратичное отклонение

Из таблицы видно, что механизм с внешним управлением показывает очень большое среднеквадратичное отклонение и оказывает большое влияние на время выполнения программы.

Второе тестирование представляло из себя шифрование с помощью трех раундов линейного участка длинной 111 байт. Результаты приведены в таблице 2.

Количество потоков	Раздельный буфер		Изменяемый указатель		Неблокирующая синхронизация	
	M	D	M	D	M	D
2	30871	603	30296	201	15867	83
4	30714	354	30055	258	7973	7
8	30588	467	30143	318	3996	3
16	30967	395	30277	196	2006	11
32	30846	223	30188	161	1017	31
64	30233	320	29893	373	530	2

Таблица 2

M – Среднее время выполнения, D – Среднеквадратичное отклонение

Из таблицы видно что все методы, которые используют синхронизацию с блокировками крайне плохо себя показывают в данной ситуации. В то время как использование неблокирующей синхронизации практически не влияет на время работы приложения не зависимо от размеров шифруемого участка.

Заключение

В рамках данной работы были выполнены следующие задачи.

- Изучены механизмы синхронизации на уровне ассемблера в архитектуре x86.
- Рассмотрены пять архитектур модуля синхронизации.
- Реализованы прототипы для каждого из рассмотренных шаблонов на языке ассемблера x86.
- Проведено тестирование и сравнение полученных прототипов, которое показало преимущества использования неблокирующей синхронизации.

Список литературы

[1] К. Ю. Богачев Основы параллельного программирования Изд. Бином

[3] Ю. Лифшиц. Обфускация (запутывание) программ. Обзор. URL: https://logic.pdmi.ras.ru/~yura/of/survey1.pdf

[3] Дж. Рихтер. Windows для профессионалов. Создание эффективных WIN32-приложений с учетом специфики 64-разрядной версии Windows. Изд-ва: Питер, Русская Редакция, 2001

[4] М. Руссинович, Д. Соломон. Внутреннее устройство Microsoft Windows. В 2х частях. Изд-во Питер.

[5] URL: https://software.intel.com/en-us/node/506090.

[6] Paul E. McKenney Is Parallel Programming Hard, And, If So, What Can You Do About It? January 2, 2017

[7] Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2 Instruction Set Reference. September 2016

[8] Камерон Хьюз, Трейси Хьюз. Параллельное и распределенное программирование с использованием C++. Изд-во Вильямс, 2004