Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Архитектура вычислительных систем (АВС)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту на тему

«Эмулятор ARM»

Выполнила: студент группы 353501,

Ярош Г.И.

Руководитель: ассистент кафедры информатики, Плехова Т. В.

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#__RefHeading___Toc1589_988211203)

[1. Анализ предметной области 4](#__RefHeading___Toc1591_988211203)

[1.1 Архитектура ARM 4](#__RefHeading___Toc1593_988211203)

[1.2 RISC процессоры 4](#__RefHeading___Toc1617_988211203)

[1.3 Сопроцессоры ARM 6](#__RefHeading___Toc1595_988211203)

[1.4 Регистры ARM процессоров 7](#__RefHeading___Toc1619_988211203)

[1.5 Обзор аналогов 7](#__RefHeading___Toc1621_988211203)

[1.6 Постановка задачи 8](#__RefHeading___Toc1623_988211203)

[2. Используемые технологии 9](#__RefHeading___Toc1597_988211203)

[2.1 Python 9](#__RefHeading___Toc1599_988211203)

[Возможности: 9](#__RefHeading___Toc1601_988211203)

[3. Программная реализация 11](#__RefHeading___Toc1603_988211203)

[1.1 Возможности платформы .NET 11](#__RefHeading___Toc1605_988211203)

[2Описание применения 15](#__RefHeading___Toc1607_988211203)

[2.1 Главное окно системы 15](#__RefHeading___Toc1609_988211203)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 21](#__RefHeading___Toc1611_988211203)

[ПРИЛОЖЕНИЯ 22](#__RefHeading___Toc1613_988211203)

[Приложение А Класс получения информации о текущем процессе (использование возможностей платформы .NET) 22](#__RefHeading___Toc1615_988211203)

ВВЕДЕНИЕ

При программировании на различных мобильных платформах программист часто сталкивается с трудностями при решении поставленных. Это связано с ограниченными средствами отладки программ на большинстве мобильных устройств. С такой проблемой можно столкнуться программируя и устройства построенные на архитектуре ARM.

Все ограничения можно в один момент устранить, если предоставить возможность программисту запускать и отлаживать код программ на ПК. Однако, так как большинство компьютеров построены на архитектуре отличной от ARM, приходиться прибегать к виртуализации и эмуляции ARM-устройств.

Для ряда специфических задач приходится эмулировать работу конкретного устройства со всеми нужными модулями. Но для большинства общих проблем достаточно только базовая эмуляция архитектуры с возможностью выполнения минимального набора инструкций. Для этого создаются небольшие приложения, которые созданы для эмулирования работы процессора и выполнения.

Таким образом, целью данной курсовой работы являлось: создать приложение, эмулирующего работу реального процессора построенного на архитектуре ARM и позволяющего выполнять базовые инструкции ARM-процессоров.

1. Анализ предметной области

1.1 Архитектура ARM

Архитектура ARM (Advanced RISC Machine, Acorn RISC Machine, усовершенствованная RISC-машина) — семейство лицензируемых 32-битных и 64-битных микропроцессорных ядер разработки компании ARM Limited.

Уже давно существует справочное руководство по архитектуре ARM, которое разграничивает все типы интерфейсов, которые поддерживает ARM, так как детали реализации каждого типа процессора могут отличаться. Архитектура развивалась с течением времени, и начиная с ARMv7 были определены 3 профиля:

* A(application) — для устройств, требующих высокой производительности (смартфоны, планшеты)
* R(real time) — для приложений, работающих в реальном времени,
* M(microcontroller) — для микроконтроллеров и недорогих встраиваемых устройств.

Профили могут поддерживать меньшее количество команд (команды определенного типа). Процессор может находиться в одном из следующих операционных режимов:

* User mode — обычный режим выполнения программ. В этом режиме выполняется большинство программ.
* Fast Interrupt (FIQ) — режим быстрого прерывания (меньшее время срабатывания).
* Interrupt (IRQ) — основной режим прерывания.
* System mode — защищённый режим для использования операционной системой.
* Abort mode — режим, в который процессор переходит при возникновении ошибки доступа к памяти (доступ к данным или к инструкции на этапе prefetch конвейера).
* Supervisor mode — привилегированный пользовательский режим.
* Undefined mode — режим, в который процессор входит при попытке выполнить неизвестную ему инструкцию.

Переключение режима процессора происходит при возникновении соответствующего исключения, или же модификацией регистра статуса.

1.2 RISC процессоры

Архитектура ARM обладает следующими особенностями RISC:

* Архитектура загрузки/хранения
* Нет поддержки нелинейного (не выровненного по словам) доступа к памяти (теперь поддерживается в процессорах ARMv6, за некоторыми исключениями, и полностью в ARMv7)
* Равномерный 16х32-битный регистровый файл
* Фиксированная длина команд (32 бит) для упрощения декодирования за счет снижения плотности кода. Позднее режим Thumb повысил плотность кода.
* Одноцикловое исполнение

Чтобы компенсировать простой дизайн, в сравнении с современными процессорами вроде Intel 80286 или Motorola 68020 были использованы некоторые особенности дизайна:

* Арифметические инструкции заменяют условные коды, только когда это необходимо
* 32-битное многорегистровое циклическое сдвиговое устройство, которое может быть использовано без потерь производительности в большинстве арифметических инструкций и адресных расчетов.
* Мощные индексированные адресные режимы
* Регистр ссылок для быстрого вызова функций листьев
* Простые, но быстрые, с двумя уровнями приоритетов подсистемы прерываний с включенными банками регистров.

Одним из существенных отличий архитектуры ARM (изначальная архитектура) от других архитектур ЦПУ является так называемая *предикация* — возможность условного исполнения команд. Под «условным исполнением» здесь понимается то, что команда будет выполнена или проигнорирована в зависимости от текущего состояния флагов состояния процессора. В Thumb и Arm 64 предикация не используется — в первом режиме для неё нет места в команде (всего 16 бит), а во втором предикация бессмысленна и сложна для реализации на суперскалярных архитектурах.

В то время как для других архитектур таким свойством, как правило, обладают только команды условных переходов, в архитектуру ARM была заложена возможность условного исполнения практически любой команды. Это было достигнуто добавлением в коды их инструкций особого 4-битового поля (предиката). Одно из его значений зарезервировано на то, что инструкция должна быть выполнена безусловно, а остальные кодируют то или иное сочетание условий (флагов). С одной стороны, с учётом ограниченности общей длины инструкции, это сократило число битов, доступных для кодирования смещения в командах обращения к памяти, но с другой — позволило избавляться от инструкций ветвления при генерации кода для небольших if-блоков.

ARM7 и более ранние версии имеют трехступенчатый конвейер. Это ступени переноса, декодирования и исполнения. Более производительные архитектуры, типа ARM9, имеют более сложные конвейеры. Cortex-a8 имеет 13-ступенчатый конвейер.

1.3 Сопроцессоры ARM

Архитектура предоставляет способ расширения набора команд, используя сопроцессоры, которые могут быть адресованы, используя MCR, MRC, MRRC, MCRR и похожие команды. Пространство сопроцессора логически разбито на 16 сопроцессоров с номерами от 0 до 15, причем 15-й зарезервирован для некоторых типичных функций управления, типа управления кэш-памятью и операции блока управления памятью (на процессорах, в которых они есть).

В машинах на основе ARM периферийные устройства обычно подсоединяются к процессору путем сопоставления их физических регистров в памяти ARM или в памяти сопроцессора, или путем присоединения к шинам, которые, в свою очередь, подсоединяются к процессору. Доступ к сопроцессорам имеет большее время ожидания, поэтому некоторые периферийные устройства проектируются для доступа в обоих направлениях. В остальных случаях разработчики чипов лишь пользуются механизмом интеграции сопроцессора. Например, движок обработки изображений должен состоять из малого ядра ARM7TDMI, совмещенного с сопроцессором, который поддерживает примитивные операции по обработке элементарных кодировок HDTV.

Расширение усовершенствованного SIMD, также называемое технологией NEON — это комбинированный 64- и 128-битный набор команд SIMD (single instruction multiple data), который обеспечивает стандартизованное ускорение для медиаприложений и приложений обработки сигнала. NEON может выполнять декодирование аудиоформата mp3 на частоте процессора в 10 МГц, и может работать с речевым кодеком GSM AMR (adaptive multi-rate) на частоте более 13МГц. Он обладает внушительным набором команд, отдельными регистровыми файлами, и независимой системой исполнения на аппаратном уровне. NEON поддерживает 8-, 16-, 32-, 64-битную информацию целого типа, одинарной точности и с плавающей запятой, и работает в операциях SIMD по обработке аудио и видео (графика и игры). В NEON SIMD поддерживает до 16 операций единовременно.

Технология VFP (Vector Floating Point, вектора чисел с плавающей запятой) — расширение сопроцессора в архитектуре ARM. Она производит низкозатратные вычисления над числами с плавающей запятой одинарной/двойной точности, в полной мере соответствующие стандарту ANSI/IEEE Std 754—1985 Standard for Binary Floating-Point Arithmetic. VFP производит вычисления с плавающей запятой, подходящие для широкого спектра приложений — например, для КПК, смартфонов, сжатие звука, трёхмерной графики и цифрового звука, а также принтеров и телеприставок. Архитектура VFP также поддерживает исполнение коротких векторных команд. Но, поскольку процессор выполняет операции последовательно над каждым элементом вектора, то VFP нельзя назвать истинным SIMD-набором инструкций. Этот режим может быть полезен в графике и приложениях обработки сигнала, так как он позволяет уменьшить размер кода и выработку команд.

Другие сопроцессоры с плавающей запятой и/или SIMD, находящиеся в ARM-процессорах, включают в себя FPA, FPE, iwMMXt. Они обеспечивают ту же функциональность, что и VFP, но не совместимы с ним на уровне опкодов.

1.4 Регистры ARM процессоров

ARM предоставляет 31 регистр общего назначения разрядностью 32 бит. В зависимости от режима и состояния процессора пользователь имеет доступ только к строго определённому набору регистров. В ARM state разработчику постоянно доступны 17 регистров:

* 13 регистров общего назначения (r0..r12).
* Stack Pointer (r13) — содержит указатель стека выполняемой программы.
* Link register (r14) — содержит адрес возврата в инструкциях ветвления.
* Program Counter (r15) — биты [31:1] содержат адрес выполняемой инструкции.
* Current Program Status Register (CPSR) — содержит флаги, описывающие текущее состояние процессора. Модифицируется при выполнении многих инструкций: логических, арифметических, и др.

Во всех режимах, кроме User mode и System mode, доступен также Saved Program Status Register (SPSR). После возникновения исключения регистр CPSR сохраняется в SPSR. Тем самым фиксируется состояние процессора (режим, состояние; флаги арифметических, логических операций, разрешения прерываний) на момент непосредственно перед прерыванием

**1.5 Обзор аналогов**

В настоящее время существует огромное множество инструментов для эмуляции процессоров построенных на разнообразных архитектурах. Для архитектуры ARM широко используются следующие эмуляторы: ARMware, QEMU, SkyEye.

Эти приложения предоставляют возможность выполнять образ программы на не ARM-платформах. Все имеют графический интерфейс, возможность запуска в режиме отладки, дизассеблинга исходного кода.

ARMware — эмулятор ARM-архитектуры работающий на платформе x86. Может эмулировать процессоры StrongARM SA-110 и Intel Xscale.

QEMU — свободная программа с открытым исходным кодом для эмуляции аппаратного обеспечения различных платформ. Эмулятор для архитектур множества архитектур, включая x86, ARM, SPARC and PowerPC. Поддерживает работу на множестве платформ. Имеет удобных графический интерфейс.

SkyEye — быстрый и легкий эмулятор платформ ARM, Coldfine, Mips, PowerPC, Sparc, x86. Позволяет эмулировать многоядерные и многокомпьютерные системы

Общим недостатком явсляется медленная скорость выполнения кода, сложный процесс запуска образов программ,

**1.6 Постановка задачи**

Приведенные аналоги эмуляторов архитектуры излишне сложны в использовании, настройке. Они обладают сложным интерфейсом, излишним функционалом для решения простых задач по отладке ARM-кода

В данной курсовой работе будет реализовано приложение для эмулирования архитектуры процессора ARM с возможностью выполнения, отладки, дизассемблинга кода.

Основными задачами программного обеспечения являются:

* Эмуляция работы ARM процессора и его основных элементов
* Эмуляция работы и взаимосвязи памяти и процессора.
* Реализация возможности выполнения программ для ARM-платформ с помощью эмулируемого процессора и памяти
* Реализация возможности отладки и дизассемблинга кода

Разрабатываемый продукт будет являться простым средством для работы с ARM кодом на других платформах, дополнит уже существубшие аналоги

2. Используемые технологии

2.1 Python

Python — высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой объём полезных функций.

Python поддерживает несколько парадигм программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное. Основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных. Код в Python организовывается в функции и классы, которые могут объединяться в модули (они в свою очередь могут быть объединены в пакеты).

Эталонной реализацией Python является интерпретатор CPython, поддерживающий большинство активно используемых платформ. Он распространяется под свободной лицензией Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях, включая проприетарные. Есть реализации интерпретаторов для JVM (с возможностью компиляции), MSIL (с возможностью компиляции), LLVM и других. Проект PyPy предлагает реализацию Python на самом Python, что уменьшает затраты на изменения языка и постановку экспериментов над новыми возможностями.

Python — активно развивающийся язык программирования, новые версии (с добавлением/изменением языковых свойств) выходят примерно раз в два с половиной года. Вследствие этого и некоторых других причин на Python отсутствуют стандарт ANSI, ISO или другие официальные стандарты, их роль выполняет CPython.

2.1. JetBrains PyCharm

PyCharm — интегрированная среда разработки для языка программирования Python. Предоставляет средства для анализа кода, графический отладчик, инструмент для запуска юнит-тестов и поддерживает веб-разработку на Django. PyCharm разработана компанией JetBrains[3] на основе IntelliJ IDEA.

Возможности:

* Статический анализ кода, подсветка синтаксиса и ошибок.
* Навигация по проекту и исходному коду: отображение файловой структуры проекта, быстрый переход между файлами, классами, методами и использованиями методов.
* Рефакторинг: переименование, извлечение метода, введение переменной, введение константы, подъём и спуск метода и т. д.
* Инструменты для веб-разработки с использованием фреймворка Django
* Встроенный отладчик для Python
* Встроенные инструменты для юнит-тестирования
* Разработка с использованием Google App Engine
* Поддержка систем контроля версий: общий пользовательский интерфейс для Mercurial, Git, Subversion, Perforce и CVS с поддержкой списков изменений и слияния

3. Программная реализация

В данном разделе будет рассмотрена архитектура приложения, а также будут приведены примеры некоторых пользовательских функций и хранимых процедур.

3.1 Обработка инструкций

Инструкции архитектуры ARM имеют фиксированую длинну — 4 байта. Это очень положительно сказывается, на удобстве работы с образом программы, т. к. воозможно однозначно прочитать и выполнить все инструкции программы без необхоидмости сложной их обработки.

В загрузки образа программы и отдельных инструкций из файла на диске были написаны функции read\_instructions\_from\_file и read\_instructions. В первую функцию аргументом указывается имя файла с образом программы и она возаращает прочитанный образ, находящийся в памяти. Вторая разбивает образ на куски по 4 байта и кажлый декодирует как инструкцию.

Также в программу добавлены средства для компиляции исходного ассемблерного кода с помощью компилятора fasmarm. В модуле emu.utils.compile содержаться функции для компиляции отдельного файла и для компиляции ARM-кода из строки напрямую.

**3.2. Эмуляция оперативной памяти**

Классы RandomAccessMemory и RandomAccessMemoryManager реализованы для создания эмулируемой оперативной памяти, поведение которой максимально подобно на настоящую.

Класс RandomAccecssMemory и представляет собой оперативную память. Хранение данных реализовано на основе массива, в котором по адресу записывается либо читаются значения.

Класс RandomAccessMemoryManager нужен для удобного управдения памятью изнутри приложения. Он содержит в себе операции по записи, чтению из базы данных по произвольному адресу, чтение блока данных по произвольному адресу, позволяет загружать образы программ в эмулируемую память

3.3 Эмуляция процессора

Для эмуляции процессора был создан класс Procssor, который является корневым и агрегирует все остальные сущности. Методы load\_image и run\_image служат для загруски и начала выполнения образа программы соответственно.

При создании объекта процессора вместе с ним создаются объекты классов RegisterManager, Pipeline, RandomAccessMemoryManager, StateManager

Класс RegisterManager служит для удобного управления регистрами процессора. При его создании можно указать количество регистров общего доступа, которые будут доступны инструкциям. Дополнительно создаются три регистра: PC — счетчик команд(указывает на текущую выполняемую команду), SP — указатель стека, LC — содержит адресс возврата в функциях с ветвлением.

StateManager — класс, который занимается обновлением флагов состояния процессора. Поддерживаются четыре флага: C — флаг переноса, Z — флаг нуля, N — флаг отрицательноси, V — флаг переполнения. Запрещено прямое изменение флагов на любое значение. Это сследует делать через специально реализованные функции get\_flag(), set\_flag(), clear\_flag()

Класс Pipeline является программной реализацией трехступенчатого так конвеера. Каждая инструкция обрабаотывается конвеером в три этапа: получение инструкции, екодирование инструкции, выполнение инструкции. На этапе получения инструкции из оперативной памяти запрашиваются 4 байта по адресу, который храниться в регистре PC процессора. Затем команда проходит этап декодирования, где определяется тип инструкции, операнды, дополнительная информация. После инструкция выполняется и регистр PC увеличивается на 4 — становиться указателем на следующую команду.

При реализации системы по сбору информации о процессах/потоках были использованы возможности платформы .NET.

Статический класс Process из пространства имен System.Diagnostic предоставляет доступ к локальным и удаленным процессам и позволяет запускать и останавливать локальные системные процессы. В работе использовались методы GetProcesses(), GetProcessById() для получения процесса, Kill() – для завершения работы процесса. Экземплярные свойства ProseccName, Handle, SessionId, WorkingSet и т.д. предоставляют возможность получить полную информацию о процессе.

Класс DriveInfo из пространства имен System.IO предоставляет доступ к сведениям на диске. Для получения коллекции всех дисков используется статический метод GetDrives(). Для получения подробной информации о каждом диске используются свойства DriveType, RootDirectory, TotalSize, DriveFormat, TotalFreeSpace.

В обширной библиотеке .NET поставляются классы Registry и RegistryKey из пространства имен Microsoft Win32, которые позволяют получать доступ к реестру. Registry предоставляет собой класс, который позволяет получить эксклюзивный доступ к ключам реестра для выполнения простых операций. Другим предназначением класса Registry является предоставление экземпляров RegistryKey, представляющих ключи наивысшего уровня, т.е. разделы, которые позволяют осуществлять навигацию по реестру. Предоставляются эти экземпляры через семь статических свойств со следующими именами: ClassesRoot, CurrentConfig, CurrentUser, DynData, LocalMachine, PerformanceData и Users.

Экземпляр RegistryKey представляет ключ реестра. В этом классе есть методы для просмотра дочерних ключей, для создания новых ключей, а также для чтения и изменения значений в существующих ключах, т.е. можно выполнять все, что обычно требуется делать с ключами реестра, в том числе устанавливать для них уровни безопасности.

Для получения доступа к каждому сервису в системе используется запрос вида Registry.LocalMachine.OpenSubKey(

"SYSTEM\\CurrentControlSet\\services\\" + serv.ServiceName), гду serv – список всех сервисов. Для получения информации о сервисе используются его свойства DisplayName, Status, а также метод GetValue() для чтения соответствующей информации у ключа реестра.

Для предоставления информации об установленном в системе программном обеспечении также используются возможности платформы .NET для работы с реестром. Для этого происходит открытие ключа по пути "SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall".

В системе по сбору информации предоставлена возможность работы с реестром: поиск в реестре по регулярному выражению, чтение значения по ключу, запись значения по ключу, удаление ключа и поддерева ключей, подсчет количества подключей у текущего ключа и значений. Это реализовано с использованием классов Registry и RegistryKey.

* 1. **Windows Management Instrumentation**

Windows Management Instrumentation (WMI) - это одна из базовых технологий для централизованного управления и слежения за работой различных частей компьютерной инфраструктуры под управлением платформы Windows.

Технология WMI — это расширенная и адаптированная под Windows реализация стандарта WBEM, принятого многими компаниями, в основе которого лежит идея создания универсального интерфейса мониторинга и управления различными системами и компонентами распределенной информационной среды предприятия с использованием объектно-ориентированных идеологий и протоколов HTML и XML.

В основе структуры данных в WBEM лежит Common Information Model (CIM), реализующая объектно-ориентированный подход к представлению компонентов системы. CIM является расширяемой моделью, что позволяет программам, системам и драйверам добавлять в неё свои классы, объекты, методы и свойства.

WMI, основанный на CIM, также является открытой унифицированной системой интерфейсов доступа к любым параметрам операционной системы, устройствам и приложениям, которые функционируют в ней.

Важной особенностью WMI является то, что хранящиеся в нём объекты соответствуют динамическим ресурсам, то есть параметры этих ресурсов постоянно меняются, поэтому параметры таких объектов не хранятся постоянно, а создаются по запросу потребителя данных. Хранилище свойств объектов WMI называется репозиторием и расположено в системной папке операционной системы Windows: %SystemRoot%\System32\WBEM\Repository.

Так как WMI построен по объектно-ориентированному принципу, то все данные операционной системы представлены в виде объектов и их свойств и методов.

Все классы группируются в пространства имен, которые иерархически упорядочены и логически связаны друг с другом по определенной технологии или области управления. В WMI имеется одно корневое пространство имен Root, которое в свою очередь имеет 4 подпространства: CIMv2, Default, Security и WMI.

Классы имеют свойства и методы и находятся в иерархической зависимости друг от друга, то есть классы-потомки могут наследовать или переопределять свойства классов-родителей, а также добавлять свои свойства.

Свойства классов используются для однозначной идентификации экземпляра класса и для описания состояния используемого ресурса. Обычно все свойства классов доступны только для чтения, хотя некоторые из них можно модифицировать определенным методом. Методы классов позволяют выполнить действия над управляемым ресурсом.

Для обращения к объектам WMI используется специфический язык запросов WMI Query Language (WQL), который является одной из разновидностей SQL. Основное его отличие от ANSI SQL — это невозможность изменения данных, то есть с помощью WQL возможна лишь выборка данных с помощью команды SELECT. Помимо ограничений на работу с объектами, WQL не поддерживает такие операторы как DISTINCT, JOIN, ORDER, GROUP, математические функции. Конструкции IS и NOT IS применяются только в сочетании с константой NULL.

В работе для получения информации о ресурсах системы использовались следующие WMI queries:

* жесткий диск

var query = new ManagementObjectSearcher("SELECT \* FROM Win32\_DiskDrive WHERE Model = '" + model + "'");

* операционная системы

var query = new ManagementObjectSearcher("SELECT \* FROM Win32\_OperatingSystem");

* процессор

var query = new ManagementObjectSearcher("SELECT \* FROM Win32\_Processor");

* оперативная память

var query = new ManagementObjectSearcher("SELECT \* FROM Win32\_PhysicalMemory");

* видеокарта

var query = new ManagementObjectSearcher("SELECT \* FROM Win32\_VideoController").

* 1. **Log Parser**

Log Parser мощный, универсальный инструмент, который предоставляет универсальный доступ в виде запросов к лог, XML и CSV файлам, а также к ключевым источникам данным операционной системы Windows таким как журнал событий, реестр, файловая система и т.д.

С использованием возможностей Log Parser в программе реализована поиск в реестре по значению (см. приложение).

* 1. **Lazy initialize**

Отложенная инициализацияобъекта означает, что его создание откладывается до первого использования. Отложенная инициализация в основном используется, чтобы повысить быстродействие, избежать ресурсоемких вычислений и уменьшить требования к памяти программы.

В языке C# поддержку отложенной инициализации обеспечивает класс Lazy<T>. После создания объекта Lazy, экземпляр **типа** не создается до первого доступа к свойству Value переменной Lazy. При первом доступе заключенный в оболочку тип создается, возвращается и сохраняется для любого использования в будущем.

В проекте возможности отложенной инициализации использованы для получения списка всех сервисов в системе.

private Lazy<ServiceController[]> services = new Lazy<ServiceController[]>(() => { return ServiceController.GetServices(); }).

* 1. **Double Checked Locking**

Блокировка с двойной проверкой (Double checked locking) — параллельный шаблон проектирования, предназначающийся для уменьшения накладных расходов, связанных с получением блокировки. Сначала проверяется условие блокировки без какой-либо синхронизации; поток делает попытку получить блокировку, только если результат проверки говорит о том, что ни один другой поток не владеет блокировкой.

В проекте данный паттерн используется при обновлении информации р загрузке процессора каждую секунду (см. приложение).

1. Описание применения

Приложение содержит ряд окон для предоставления возможности предоставления информации о системе. Рассмотрим работу с каждым из них.

* 1. Главное окно системы

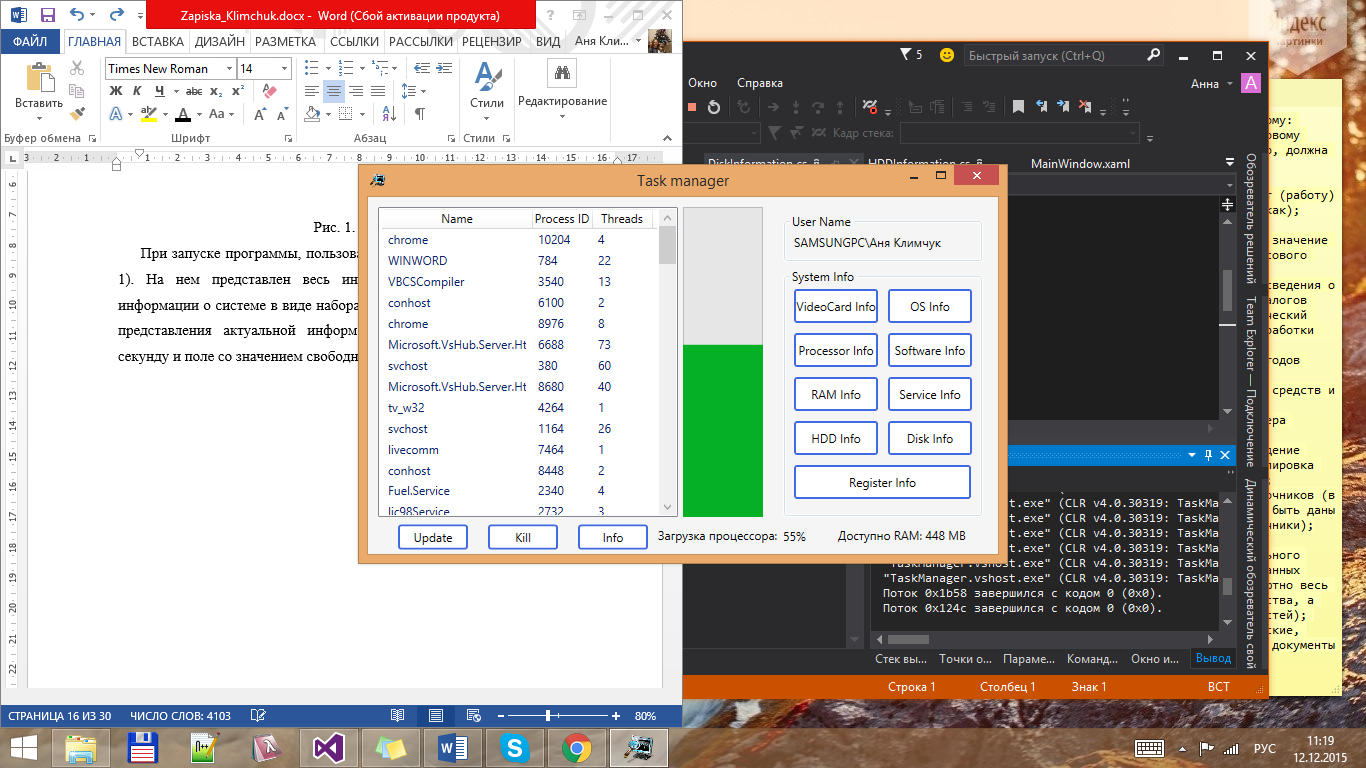


Рис. 1. Главное окно

При запуске программы, пользователя представляется главное окно (рис. 1). На нем представлен весь интерфейс для дальнейшего получения информации о системе в виде набора кнопок. Также имеется ProgressBar для представления актуальной информации о загрузке процессора каждую секунду и поле со значением свободной оперативной памяти.

* 1. **Окна с информацией о системе**

Для обработки запросов пользователя в виде нажатия на кнопки для получения информации о соответствующем ресурсе представлен ряд окон.

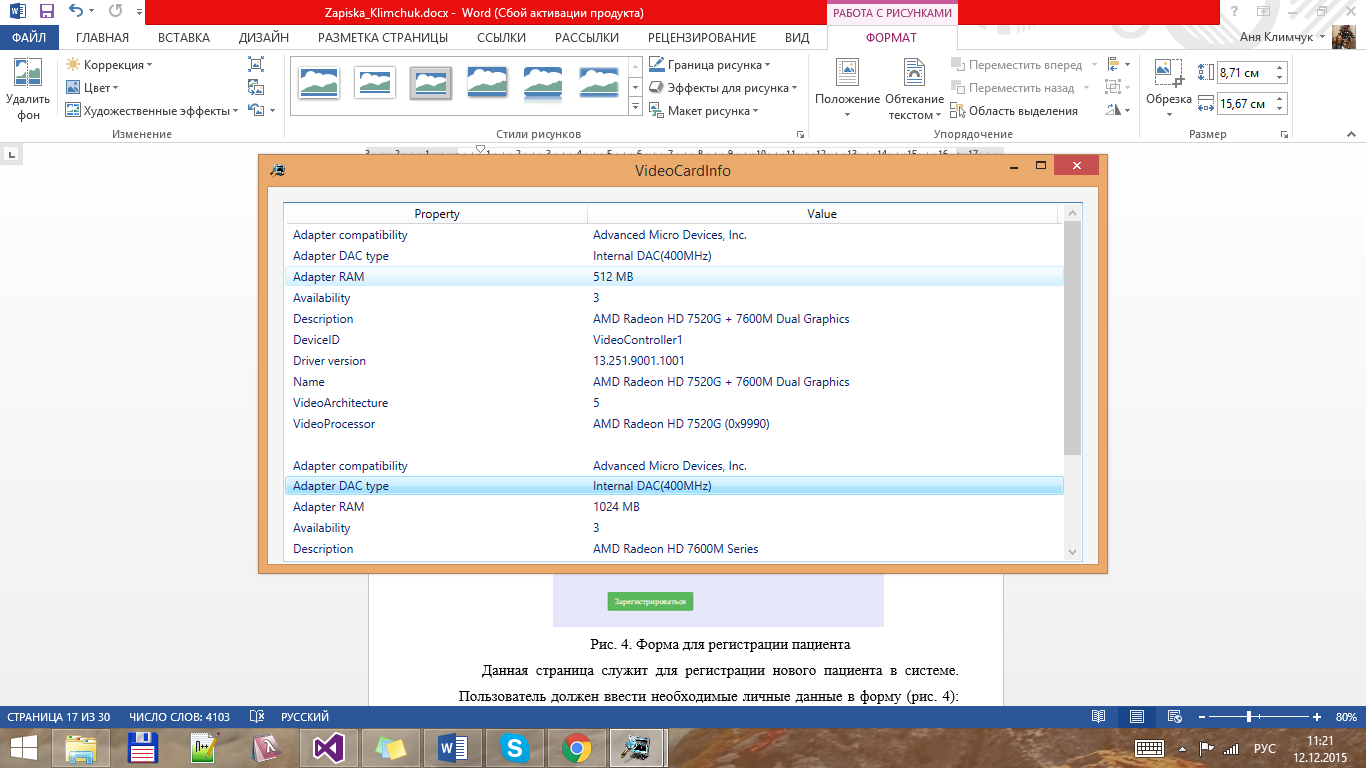


Рис. 3. Окно с информацией о видеокарте

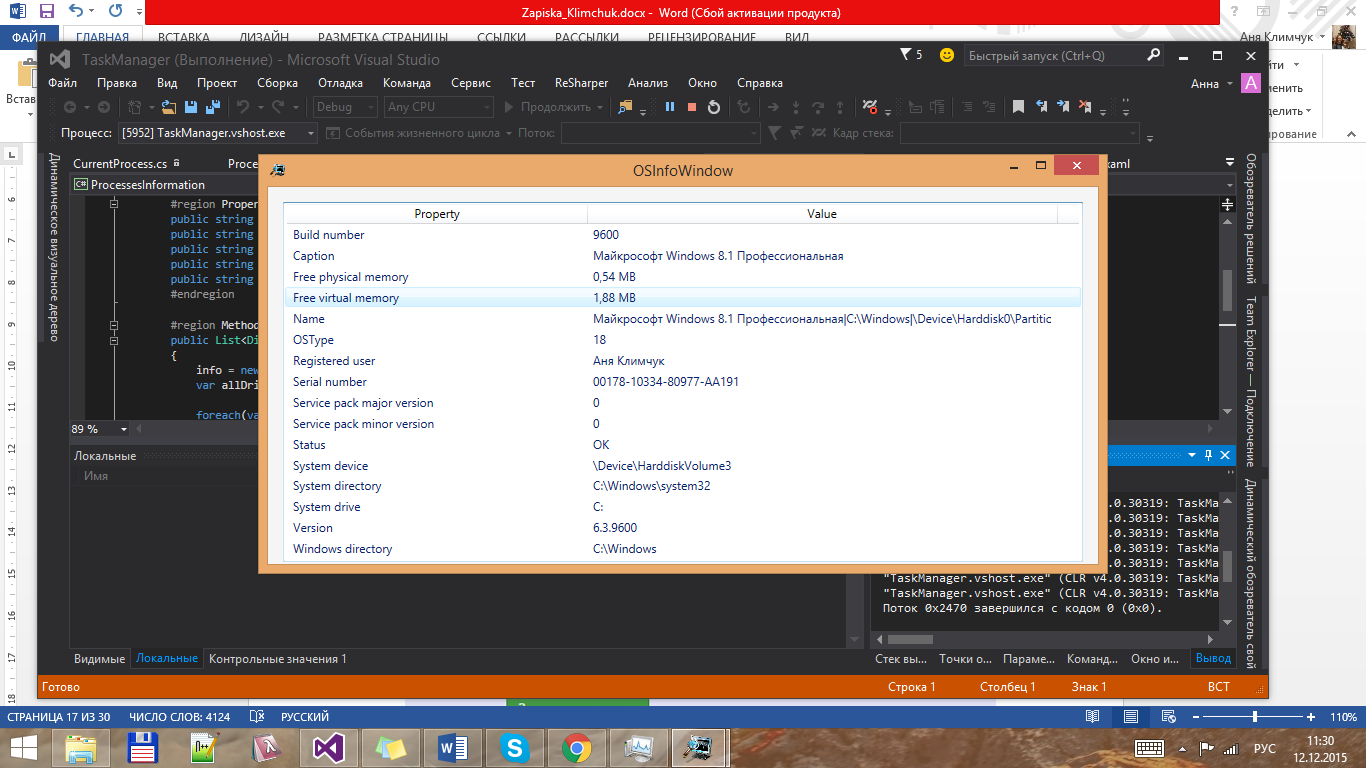


Рис. 4. Окно с информацией об операционной системе

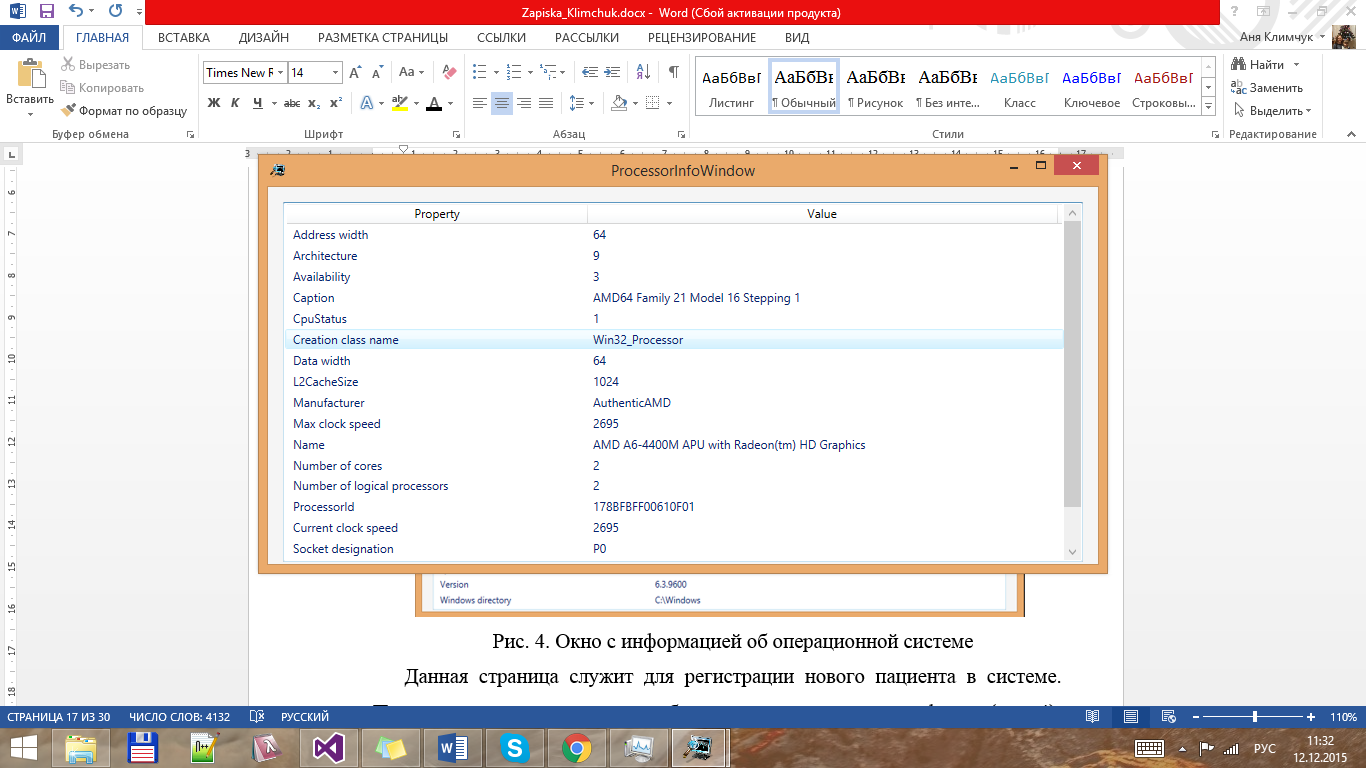


Рис. 5. Окно с информацией о процессоре

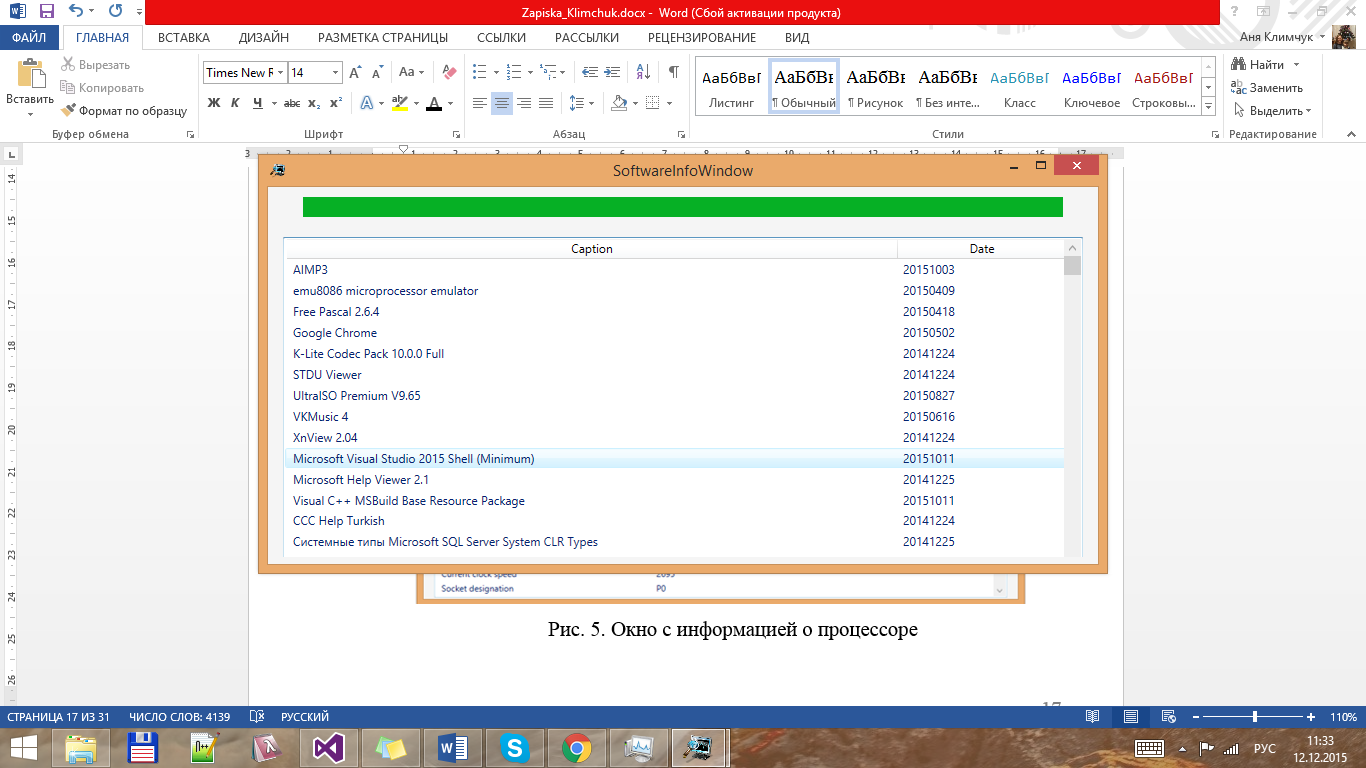


Рис. 6. Окно с информацией об установленном ПО

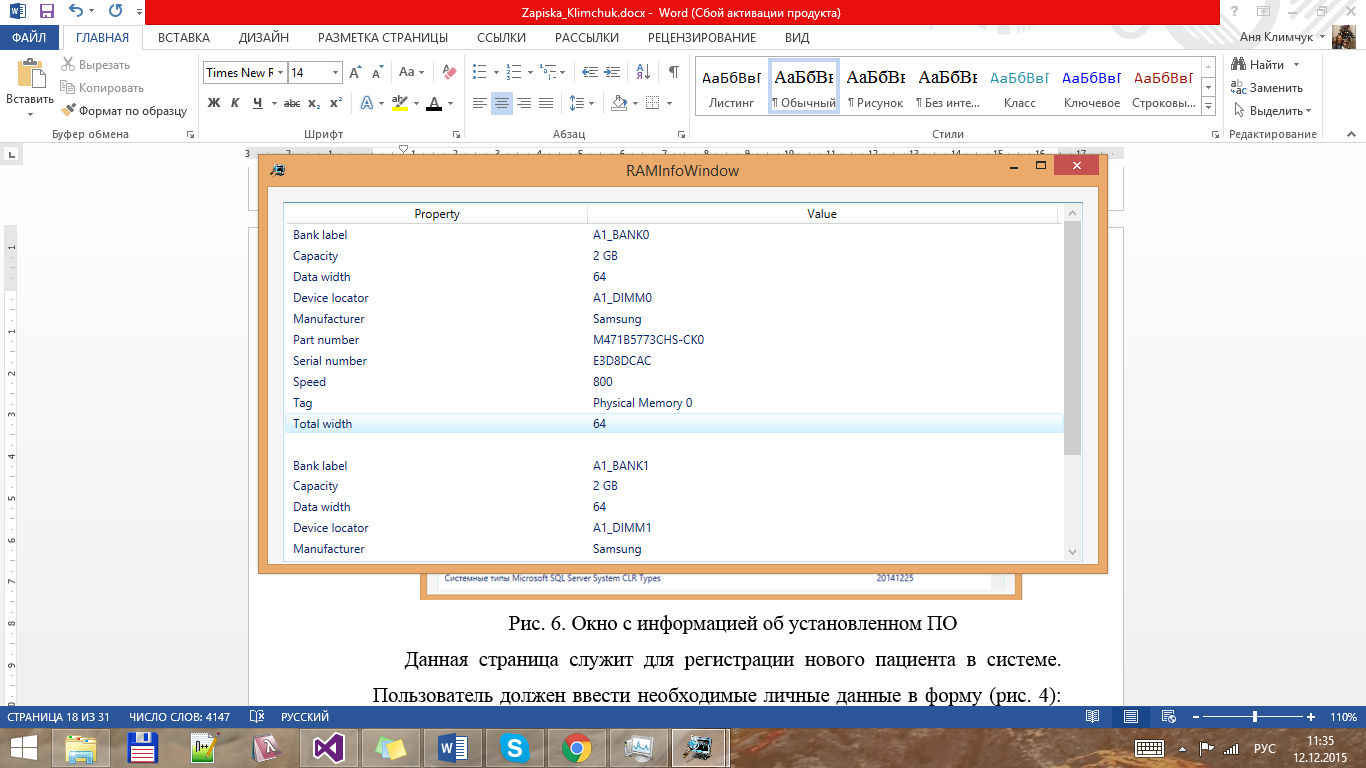


Рис. 7. Окно с информацией об RAM

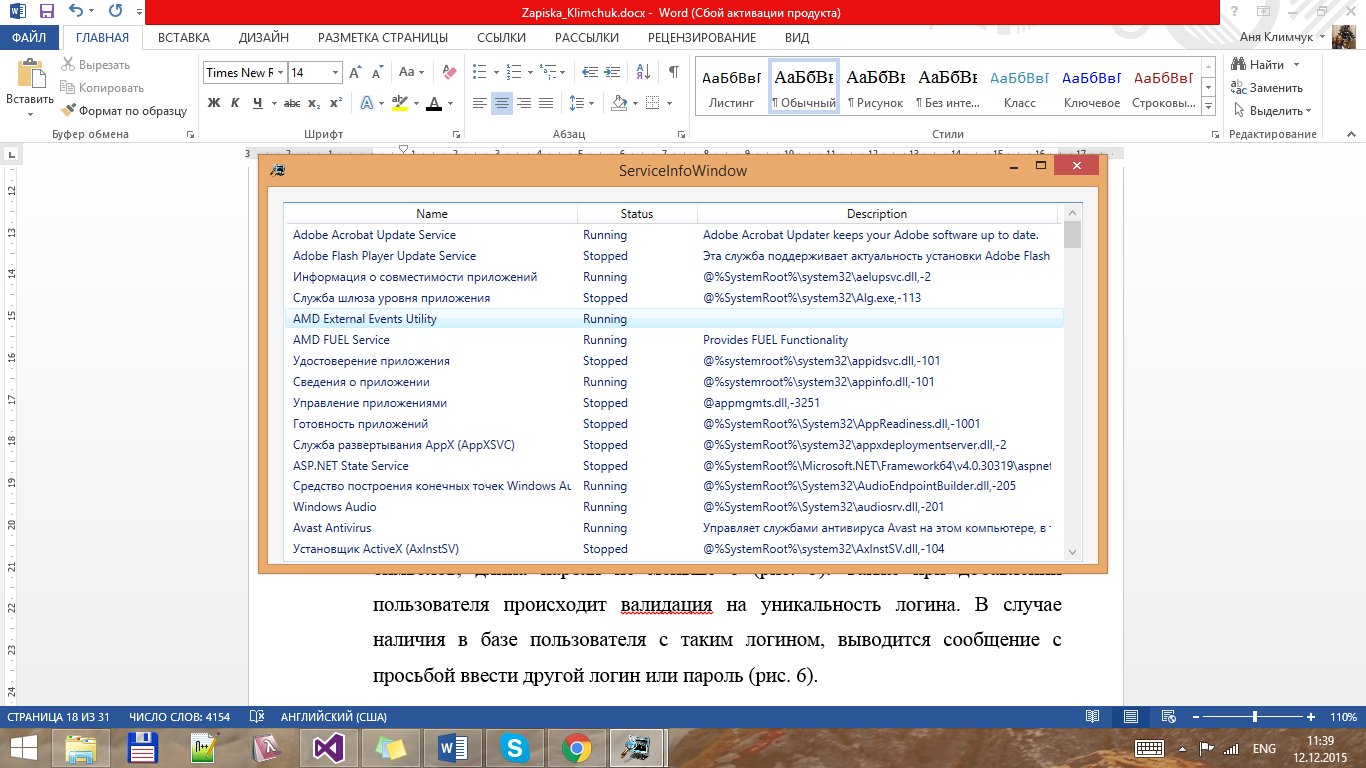


Рис. 8. Окно с информацией о сервисах

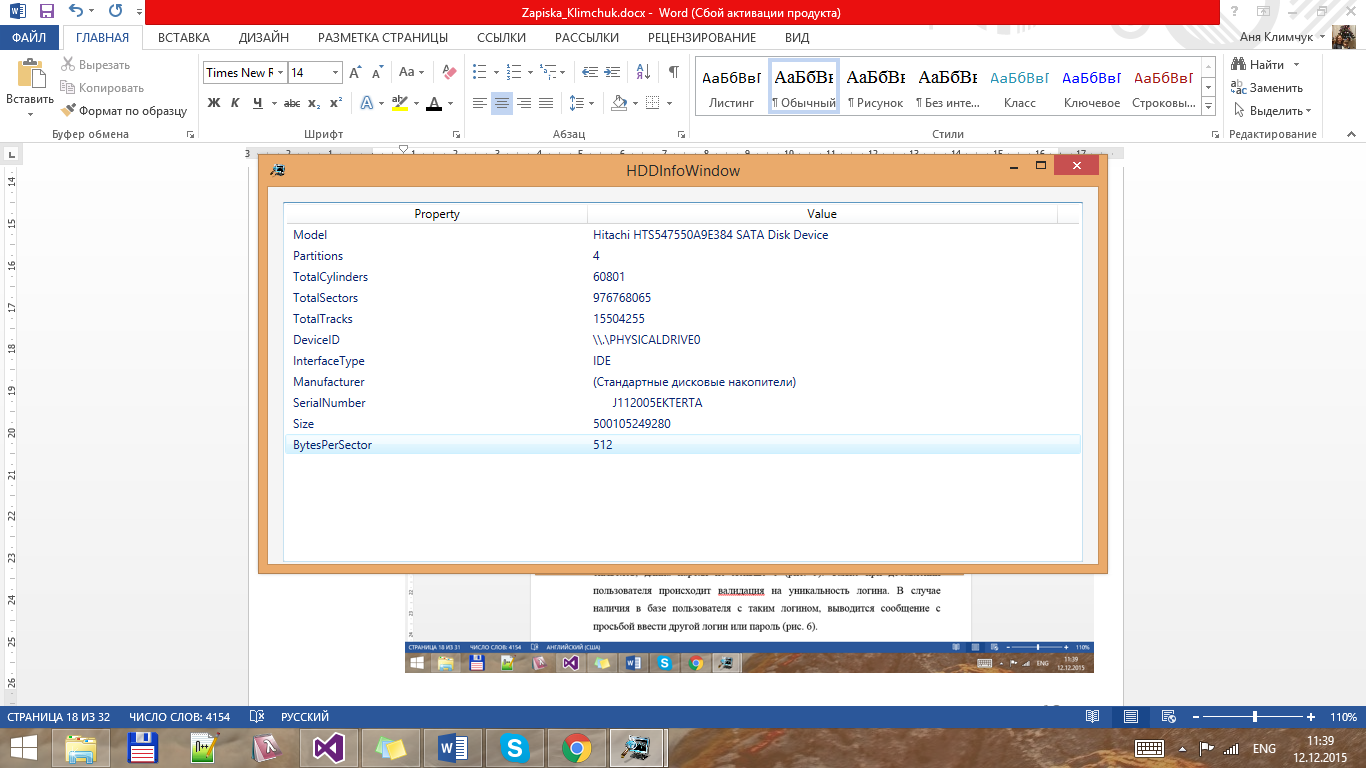


Рис. 9. Окно с информацией о жестком диске

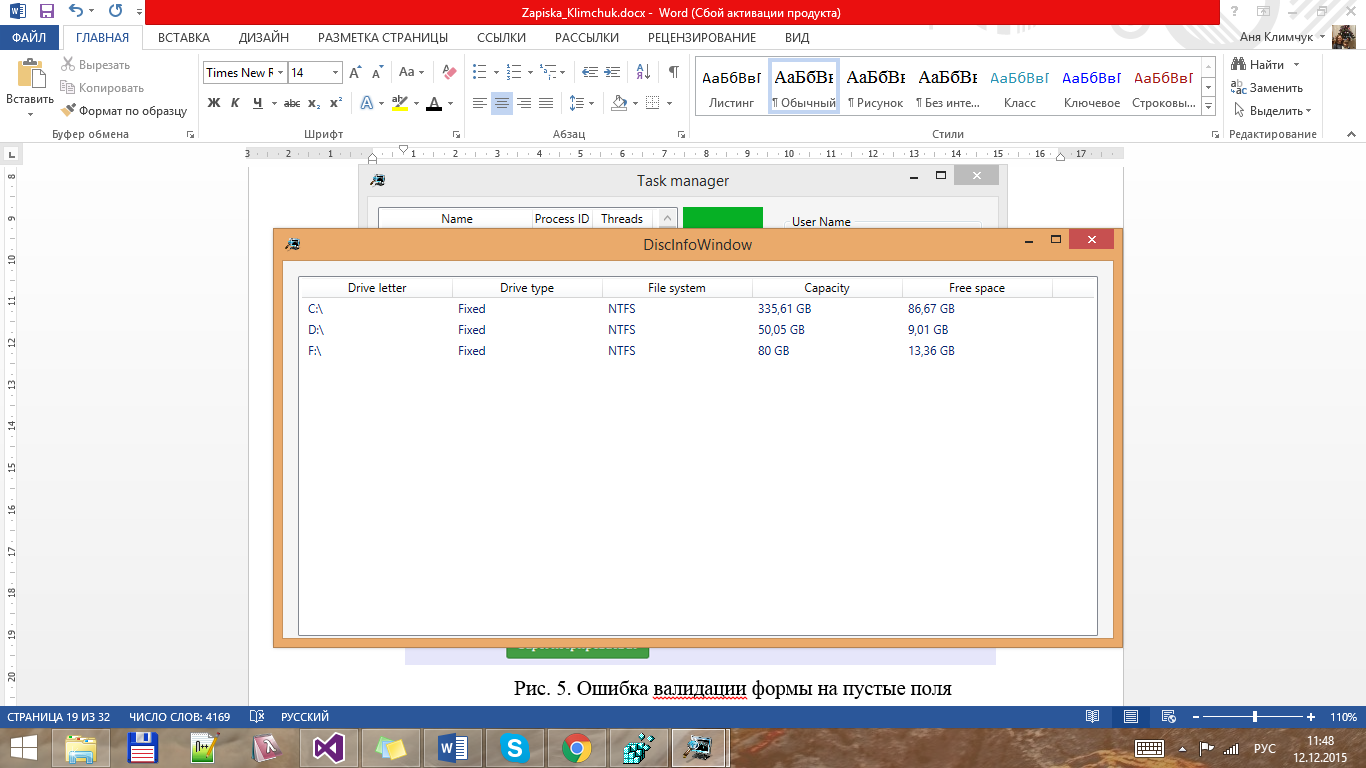


Рис. 10. Окно с информацией о дисках

О каждом процессе в системе можно получить информацию (рис. 11).

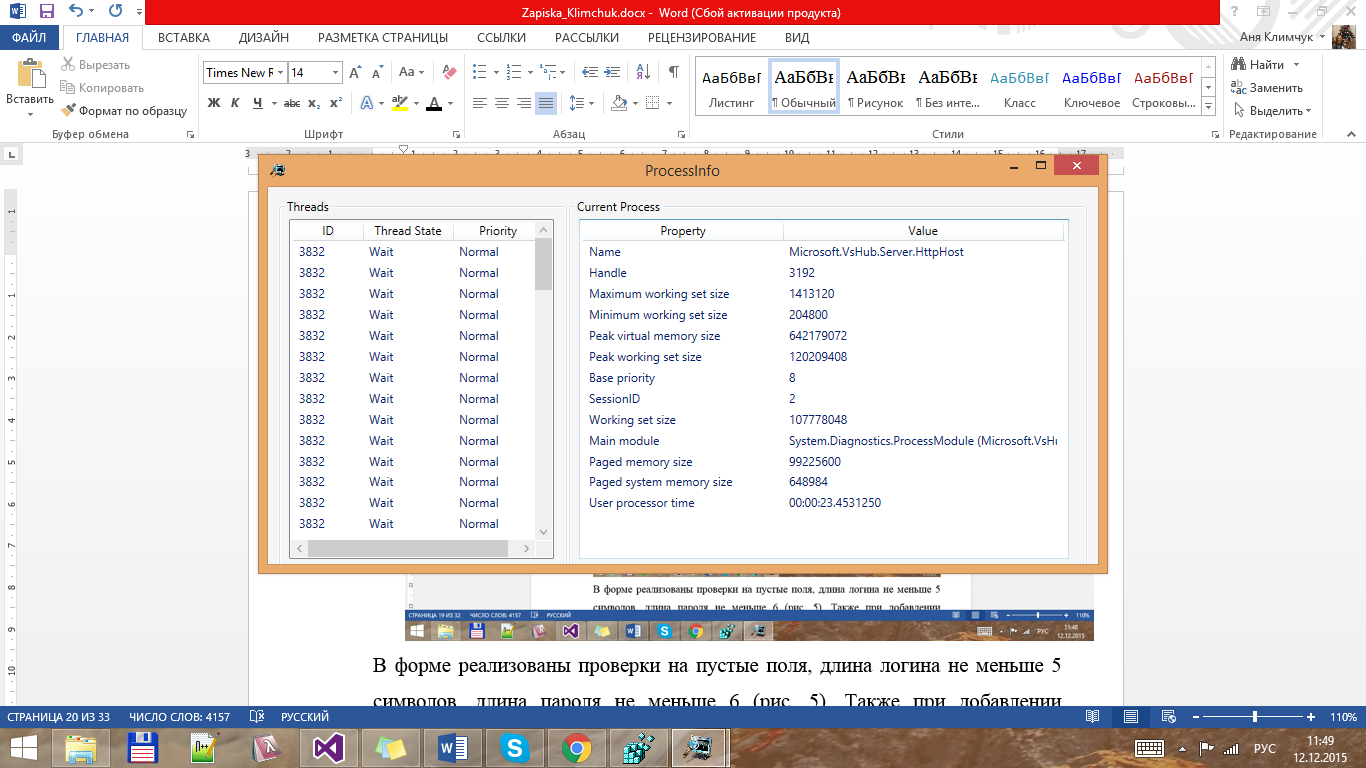


Рис. 11. Окно с информацией о конкретном процессе

В случае запрещенного доступа будет получено окно с предупреждением.

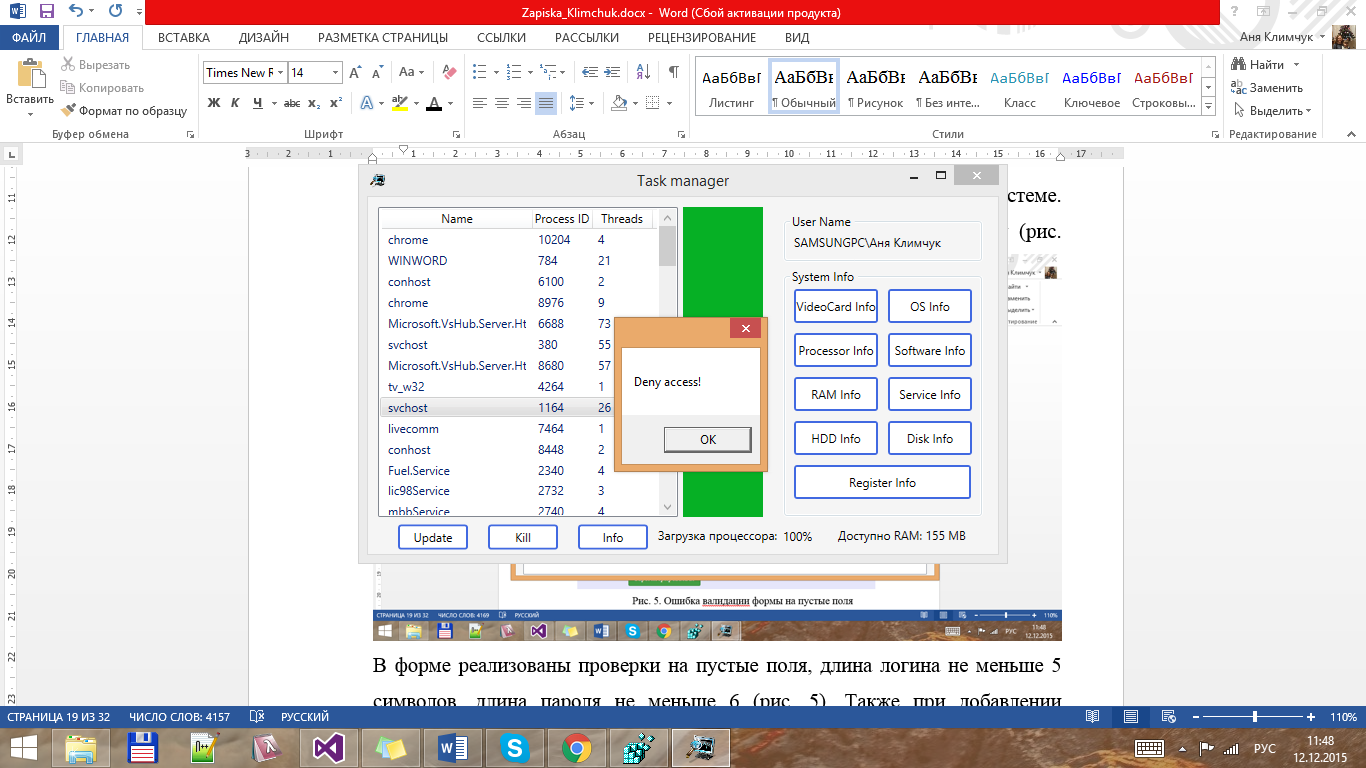


Рис. 12. Окно с предупреждением об отказе в доступе

В системе предоставлена возможность работы с реестром: удаление поддерева ключей, поиск по значению ключа, по регулярному выражению, чтение, удаление и запись значения ключа.

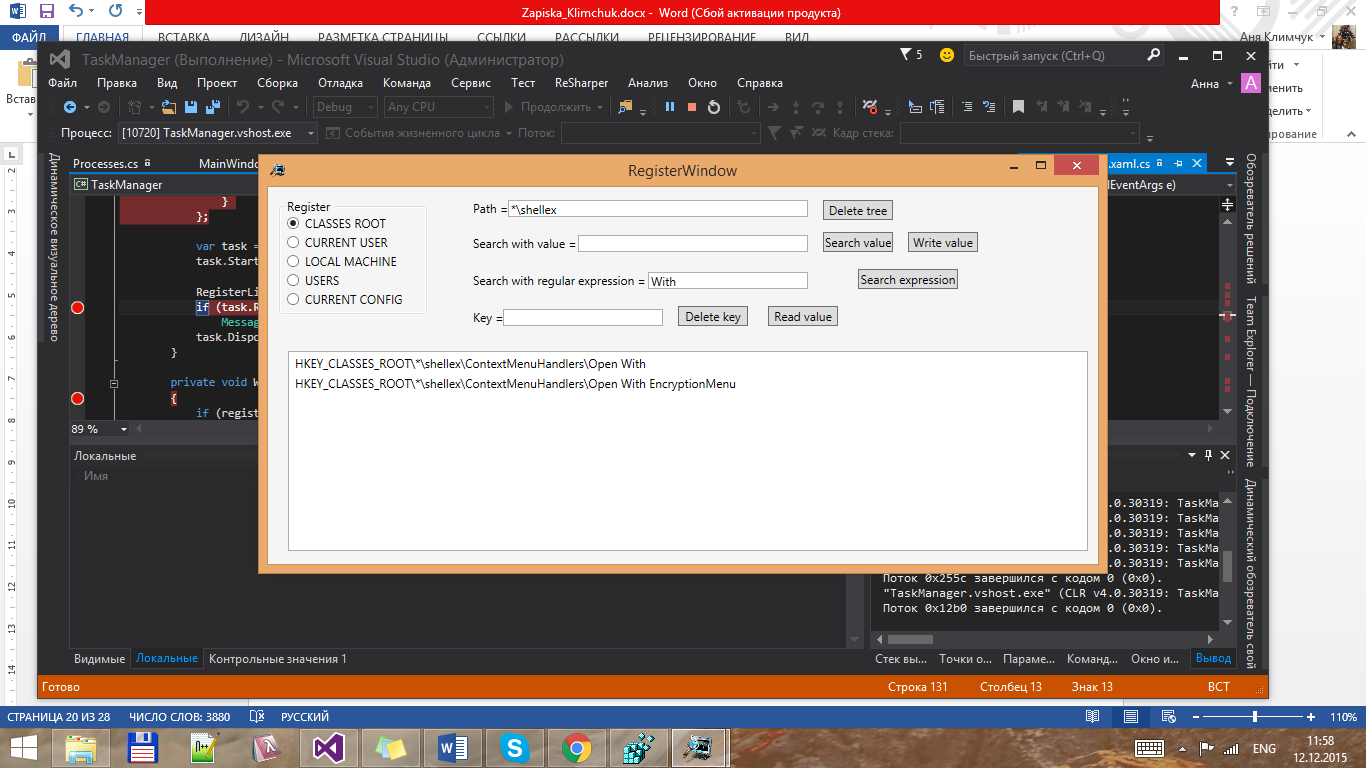


Рис. 13. Окно работы с реестром

**ЗАКЛЮЧЕНИЕ**

В результате работы над курсовым проектом был разработан программный продукт для сбора информации о ресурсах системы, таких как видеокарта, процессор, операционная система, диски, сервисы, оперативная память, установленное ПО. Система позволяет просмотреть все запущенные процессы, полную информацию и предоставляет возможность их завершения.

В процессе написания работы были закреплены навыки работы с технологией WPF, многопоточностью. При создании приложения были применены паттерны проектирования, способы синхронизации потоков, отложенная инициализация, WMI queries, Log Parser.

Разработанное программное средство представляет собой законченный программный продукт, готовый к использованию. Но при желании приложение можно доработать: расширить функциональность программного продукта, предоставить больше возможностей для управления ресурсами системы и т.д.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. М.Лутц. Изучаем Python, 4-ое издание / М. Лутц. - Символ-Плюс, 2010, – 1280 с.
2. ARM documentation [Электронный ресурс]. – Электронные данные. – Режим доступа : <http://infocenter.arm.com/help/index.jsp>.
3. Реестр в C# [Электронный ресурс]. – Электронные данные. – Режим доступа : <https://docs.python.org/>.

ПРИЛОЖЕНИЯ

Приложение А Класс получения информации о текущем процессе (использование возможностей платформы .NET)

public class CurrentProcess

{

#region Fields

private Process process;

private List<CurrentProcess> info;

#endregion

#region Properties

public string Property { get; private set; }

public string Value { get; private set; }

#endregion

#region Methods

public List<CurrentProcess> GetCurrentProcessInformationById(int id)

{

info = new List<CurrentProcess>();

process = Process.GetProcessById(id);

info.Add(new CurrentProcess() { Property = "Name", Value = process.ProcessName.ToString() });

info.Add(new CurrentProcess() { Property = "Handle", Value = process.Handle.ToString() });

info.Add(new CurrentProcess() { Property = "Maximum working set size", Value = process.MaxWorkingSet.ToString() });

info.Add(new CurrentProcess() { Property = "Minimum working set size", Value = process.MinWorkingSet.ToString() });

info.Add(new CurrentProcess() { Property = "Peak virtual memory size", Value = process.PeakVirtualMemorySize.ToString() });

info.Add(new CurrentProcess() { Property = "Peak working set size", Value = process.PeakWorkingSet.ToString() });

info.Add(new CurrentProcess() { Property = "Base priority", Value = process.BasePriority.ToString() });

info.Add(new CurrentProcess() { Property = "SessionID", Value = process.SessionId.ToString() });

info.Add(new CurrentProcess() { Property = "Working set size", Value = process.WorkingSet.ToString() });

info.Add(new CurrentProcess() { Property = "Main module", Value = process.MainModule.ToString() });

info.Add(new CurrentProcess() { Property = "Paged memory size", Value = process.PagedMemorySize.ToString() });

info.Add(new CurrentProcess() { Property = "Paged system memory size", Value = process.PagedSystemMemorySize.ToString() });

info.Add(new CurrentProcess() { Property = "User processor time", Value = process.UserProcessorTime.ToString() });

return info;

}

#endregion

}

**Приложение Б Класс получения информации о процессоре (использование WMI)**

public class ProcessorInformation

{

private List<ProcessorInformation> info;

public string NameProperty { get; private set; }

public string Value { get; private set; }

public List<ProcessorInformation> GetProcessorInfo()

{

info = new List<ProcessorInformation>();

var query = new ManagementObjectSearcher("SELECT \* FROM Win32\_Processor");

foreach (ManagementObject obj in query.Get())

{

info.Add(new ProcessorInformation() { NameProperty = "Address width", Value = obj["AddressWidth"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Architecture", Value = obj["Architecture"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Availability", Value = obj["Availability"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Caption", Value = obj["Caption"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "CpuStatus", Value = obj["CpuStatus"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Creation class name", Value = obj["CreationClassName"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Data width", Value = obj["DataWidth"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "L2CacheSize", Value = obj["L2CacheSize"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Manufacturer", Value = obj["Manufacturer"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Max clock speed", Value = obj["MaxClockSpeed"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Name", Value = obj["Name"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Number of cores", Value = obj["NumberOfCores"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Number of logical processors", Value = obj["NumberOfLogicalProcessors"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "ProcessorId", Value = obj["ProcessorId"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Current clock speed", Value = obj["CurrentClockSpeed"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Socket designation", Value = obj["SocketDesignation"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Current voltage", Value = obj["CurrentVoltage"].ToString() });

info.Add(new ProcessorInformation() { NameProperty = "Status", Value = obj["Status"].ToString() });

}

return info;

}

}

**Приложение В Класс для получения актуальной информации о загрузке процессора и состоянии RAM (Double checked locking)**

public class ProcessorUsage

{

const float sampleFrequencyMillis = 1000;

private object syncLock = new object();

private PerformanceCounter counter;

private PerformanceCounter ramCounter;

private float result;

private DateTime lastSampleTime;

public ProcessorUsage()

{

this.counter = new PerformanceCounter("Processor", "% Processor Time", "\_Total", true);

this.ramCounter = new PerformanceCounter("Memory", "Available MBytes");

}

public float GetCurrentValue()

{

if ((DateTime.UtcNow - lastSampleTime).TotalMilliseconds > sampleFrequencyMillis)

{

lock (syncLock)

{

if ((DateTime.UtcNow - lastSampleTime).TotalMilliseconds > sampleFrequencyMillis)

{

result = counter.NextValue();

lastSampleTime = DateTime.UtcNow;

}

}

}

return result;

}

public float GetAvailableRAM()

{

return ramCounter.NextValue();

}

}

**Приложение Г Метод поиска в реестре с использованием классов Registry и RegistryKey**

public List<string> GetKeyPath(string searchKey)

{

info = new List<string>();

string path = string.Empty;

try

{

foreach (string keyname in baseRegistryKey.OpenSubKey(subKey).GetSubKeyNames())

{

try

{

using (RegistryKey key = baseRegistryKey.OpenSubKey(subKey).OpenSubKey(keyname, true))

{

foreach (var name in key.GetSubKeyNames())

{

MatchCollection coll = Regex.Matches(name.ToString(), searchKey);

if (coll != null)

{

foreach (var item in coll)

info.Add(baseRegistryKey.ToString() + "\\" + subKey + "\\" + keyname + "\\" + name);

}

}

}

}

catch (System.Security.SecurityException)

{

continue;

}

}

}

catch (NullReferenceException ex)

{

throw ex;

}

return info;

}

**Приложение Д Метод поиска в реестре с использованием Log Parser**

public List<string> Search(string value)

{

info = new List<string>();

RegRecordSet rs = null;

try

{

LogQuery qry = new LogQuery();

RegistryInputFormat registryFormat = new RegistryInputFormat();

string query = "SELECT Path from \\" + baseRegistryKey + "\\" + subKey + " where Value=\'" + value + "\'";

//string query = @"SELECT Path from \HKLM\SOFTWARE\Microsoft where Value='VisualStudio'";

rs = qry.Execute(query, registryFormat);

for (; !rs.atEnd(); rs.moveNext())

info.Add(rs.getRecord().toNativeString("\n"));

}

catch (System.IO.FileNotFoundException ex)

{

throw ex;

}

finally

{

if (rs != null)

rs.close();

}

return info;

}

**Приложение Е Класс главного окна MainWindow.sc**

public partial class MainWindow : Window

{

private SystemProcess systemProcess;

private ThreadsInfo threadsInfo;

private CurrentProcess currentProcess;

private VideoCardInformation videoCard;

private OSInformation os;

private ProcessorInformation processor;

private RAMInformation ram;

private DiskInformation disc;

private HDDInformation hdd;

private ProcessorUsage usage;

private System.Timers.Timer timer;

private ServiceInformation service;

private SoftwareInformation software;

private SoftwareInfoWindow softInfoWindow;

private bool isReady = false;

private List<SystemProcess> processes;

private List<ThreadsInfo> threads;

private List<CurrentProcess> currentProcessInfo;

private List<VideoCardInformation> videoCardInfo;

private List<RAMInformation> ramInfo;

private List<HDDInformation> hddInfo;

private List<SoftwareInformation> softInfo;

public MainWindow()

{

InitializeComponent();

systemProcess = new SystemProcess();

threadsInfo = new ThreadsInfo();

currentProcess = new CurrentProcess();

videoCard = new VideoCardInformation();

os = new OSInformation();

processor = new ProcessorInformation();

ram = new RAMInformation();

disc = new DiskInformation();

hdd = new HDDInformation();

usage = new ProcessorUsage();

service = new ServiceInformation();

software = new SoftwareInformation();

processes = new List<SystemProcess>();

currentProcessInfo = new List<CurrentProcess>();

InformationAboutProcesses();

UserNameLabel.Content = UserName.GetUserName().ToString();

timer = new System.Timers.Timer(500);

timer.Elapsed += new ElapsedEventHandler(UpdateProcessorUsage);

timer.Start();

}

private void UpdateProcessorUsage(object sender, ElapsedEventArgs e)

{

this.Dispatcher.Invoke(System.Windows.Threading.DispatcherPriority.Normal, (Action)(() =>

{

CPUProgressBar.Value = usage.GetCurrentValue();

ProcessorCount.Text = ((int)usage.GetCurrentValue()).ToString() + "%";

RAMCount.Text = ((int)usage.GetAvailableRAM()).ToString() + " MB";

}));

}

private void InformationAboutProcesses()

{

ProcessesList.ItemsSource = null;

ProcessesList.Items.Clear();

ProcessesList.ItemsSource = systemProcess.GetProcesses();

}

private void VideoCard\_Click(object sender, RoutedEventArgs e)

{

videoCardInfo = new List<VideoCardInformation>();

videoCardInfo.AddRange(videoCard.GetVideoCardInfo()[0]);

videoCardInfo.AddRange(videoCard.GetVideoCardInfo()[1]);

var videoCardInfoWindow = new VideoCardInfo();

videoCardInfoWindow.Show();

videoCardInfoWindow.VideoCardInfoList.ItemsSource = videoCardInfo;

}

private void Update\_OnClick(object sender, RoutedEventArgs e)

{

InformationAboutProcesses();

}

private void Kill\_Click(object sender, RoutedEventArgs e)

{

try

{

var item = ProcessesList.SelectedItem;

var process = item as SystemProcess;

systemProcess.KillProcessById(process.ProcessID);

}

catch (NullReferenceException ex)

{

MessageBox.Show("Process to kill is not selected!");

}

catch

{

MessageBox.Show("Deny access!");

}

Thread.Sleep(500);

InformationAboutProcesses();

}

private void Info\_Click(object sender, RoutedEventArgs e)

{

try

{

var item = ProcessesList.SelectedItem;

var process = item as SystemProcess;

threads = threadsInfo.GetThreadsInformationById(process.ProcessID);

currentProcessInfo = currentProcess.GetCurrentProcessInformationById(process.ProcessID);

}

catch (NullReferenceException ex)

{

MessageBox.Show("Process to info is not selected!");

return;

}

catch (Exception ex)

{

MessageBox.Show("Deny access!");

return;

}

var processInfoWindow = new ProcessInfoWindow();

processInfoWindow.Show();

processInfoWindow.ThreadsList.ItemsSource = threads;

processInfoWindow.CurrentProcessList.ItemsSource = currentProcessInfo;

}

private void OS\_Click(object sender, RoutedEventArgs e)

{

var osInfoWindow = new OSInfoWindow();

osInfoWindow.Show();

osInfoWindow.OSInfoList.ItemsSource = os.GetOSInfo();

}

private void Processor\_Click(object sender, RoutedEventArgs e)

{

var processorInfoWindow = new ProcessorInfoWindow();

processorInfoWindow.Show();

processorInfoWindow.ProcessorInfoList.ItemsSource = processor.GetProcessorInfo();

}

private void RAM\_Click(object sender, RoutedEventArgs e)

{

ramInfo = new List<RAMInformation>();

ramInfo.AddRange(ram.GetRAMInfo()[0]);

ramInfo.AddRange(ram.GetRAMInfo()[1]);

var ramInfoWindow = new RAMInfoWindow();

ramInfoWindow.Show();

ramInfoWindow.RAMInfoList.ItemsSource = ramInfo;

}

private void Disk\_Click(object sender, RoutedEventArgs e)

{

var discInfoWindow = new DiscInfoWindow();

discInfoWindow.DiscInfoList.ItemsSource = disc.GetDrivesInfo();

discInfoWindow.Show();

}

private void HDD\_Click(object sender, RoutedEventArgs e)

{

var hddInfoWindow = new HDDInfoWindow();

hddInfoWindow.Show();

hddInfoWindow.HDDInfoList.ItemsSource = hdd.GetHDDInfo();

}

private void Register\_Click(object sender, RoutedEventArgs e)

{

var regysterWindow = new RegisterWindow();

regysterWindow.Show();

}

private void Service\_Click(object sender, RoutedEventArgs e)

{

try

{

var serviceInfoWindow = new ServiceInfoWindow();

serviceInfoWindow.Show();

serviceInfoWindow.ServiceInfoList.ItemsSource = service.GetServiceInfo();

}

catch

{

MessageBox.Show("Can't download all services!");

}

}

private void Software\_Click(object sender, RoutedEventArgs e)

{

softInfoWindow = new SoftwareInfoWindow();

softInfoWindow.Show();

isReady = false;

softInfo = software.GetSoftwareInfo(out isReady);

timer = new System.Timers.Timer(1000);

timer.Elapsed += new ElapsedEventHandler(UpdateProgressBar);

timer.Start();

}

private void UpdateProgressBar(object sender, ElapsedEventArgs e)

{

this.Dispatcher.Invoke(System.Windows.Threading.DispatcherPriority.Normal, (Action)(() =>

{

if (isReady)

{

softInfoWindow.SoftProgressBar.Value = 100;

softInfoWindow.SoftwareInfoList.ItemsSource = softInfo;

}

}));

}

}