1. Введение. Введение в системное программирование. Что такое программа? Что такое ПО? Классификация ПО. Что такое системное ПО? Классификация и функции системного ПО

**Программа** – данные, предназначенные для управления конкретными компонентами системы обработки информации (СОИ). Программы и данные хранятся в одной и той же памяти.

**Программное обеспечение** (**ПО**) – совокупность программ СОИ и программных документов, необходимых для их эксплуатации.

Свойства ПО: необходимость документирования, эффективность, надежность, возможность сопровождения

**Системная программа** – программа, предназначенная для поддержания работы СОИ или повышения эффективности её использования

**Прикладная программа** – программа, предназначенная для решения задачи в области применения СОИ. Например: Текстовые редакторы, табличные редакторы, графические редакторы

Промежуточное (связующее ПО) – совокупность программ, осуществляющих управление вторичными (сделанными самим ПО) ресурсами и взаимодействующими между различными приложениями, системами, компонентами

Например: СУБД, модули управления языком интерфейса ИС, программы сбора и предварительной обработки информации; Веб-сервер, сервер приложений, сервисная шина, система управления контентом

Существуют следующие группы системного ПО: ОС, интерфейсные оболочки (ОС), системы управления файлами, системы программирования, утилиты, драйверы, средства сетевого доступа

Классификация системного ПО:

**Управляющие программы** – системные программы, реализующие набор функций, который включает в себя управление ресурсами и взаимодействие с внешней средой СОИ, восстановление работы системы после проявления неисправностей в технических средствах

**Обслуживающие программы (утилиты)** – программы, предназначенные для оказания услуг пользователям

**Базовое системное ПО** – минимальный набор программных средств, обеспечивающий работу компа

**Сервисное системное ПО** – программы и программные комплексы, которые расширяют возможности базового ПО и организуют удобную среду для работы других программ и пользователя

К функциям системного ПО принято относить:

* Создание операционной среды функционирования для программ
* Автоматизация разработки новых программ
* Обеспечение надежной и эффективной работы компа
* Проведение диагностики и профилактики аппаратуры
* Выполнение вспомогательных технологических процессов (копирование, архивирование, восстановление после сбоев и т.д.)

2. Введение. Что такое системное программирование? Системы программирования: определение и состав. Что такое транслятор? Какие существуют виды трансляторов? Назовите и опишите этапы подготовки программы. Назовите и опишите результат работы каждого из этапов.

С**истемное программирование** – процесс разработки системных программ.

**Система программирования** – система, образуемая языком программирования, компилятором или интерпретатором программ, представленных на этом языке, соответствующей документацией, а также вспомогательными средствами для подготовки программ к форме, пригодной для выполнения (Clang + LLVM и Cmake)

**Программный модуль** –программа, предназначенная для хранения, трансляции, объединения с другими программными модулями и загрузки в оперативную память.

Системы программирования включают в себя следующие средства: редактор текста, транслятор, компоновщик, отладчик, библиотеки подпрограмм

**Транслятор** – системная программа, преобразующая исходную программу на одном языке программирования в программу на другом языке

Виды: ассемблер, компилятор, интерпретатор, эмулятор, перекодировщик, макропроцессор

Подготовка программы включает несколько этапов:

1. Редактор текста: Описание этапа: На этом этапе создается исходный код программы. Результат работы: **Исходный модуль** – программный модуль на исходном языке, обрабатываемый транслятором.

2.Транслятор (Компилятор): Описание этапа: Трансляция исходного модуля в объектный код. Этот этап делится на три шага:

Шаг первый – Предварительная обработка кода: Включает присоединение исходных файлов и работу макропроцессоров (например, через clang в Windows и Linux).

Шаг второй – Анализ: Включает лексический анализ, синтаксический анализ и семантический анализ (например, через clang в Windows и Linux).

Шаг третий – Синтез:

Включает генерацию машинно-независимого кода, оптимизацию машинно-независимого кода, распределение памяти, генерацию машинного кода и оптимизацию машинного кода (например, через clang в Windows и Linux).

Результат работы: **Объектный модуль** – программный модуль, получаемый в результате трансляции исходного модуля. Содержимое объектного модуля не содержит признаков на каком языке был написан исходный модуль. Те места в объектном модуле, где содержатся внешние ссылки, транслируются в некоторую промежуточную форму, подлежащую дальнейшей обработке. Говорят, что объектный модуль представляет собой программу на машинном языке с неразрешенными внешними ссылками.

3. Компоновщик (Редактор Связей):

Описание этапа: Этот этап следует за трансляцией и обеспечивает разрешение внешних ссылок, соединяя вместе все объектные модули, входящие в программу. Результат работы: **Загрузочный модуль** – программный модуль, представленный в форме, пригодной для загрузки в оперативную память для выполнения.

1 Препроцессинг (Preprocessing): #include, #define, #ifdef, #endif.

clang -E source.c -o source.i

Результат: Чистый C-код без директив препроцессора (файл .i).

2. Компиляция (Анализ + Синтез)

(а) Лексический анализ

Разбиение кода на токены (идентификаторы, ключевые слова, операторы).

clang -fsyntax-only -Xclang -dump-tokens source.c

(б) Синтаксический анализ

Построение абстрактного синтаксического дерева (AST).

clang -fsyntax-only -Xclang -ast-dump source.c

(в) Семантический анализ

Проверка типов, областей видимости, других семантических правил.

(г) Генерация промежуточного кода (LLVM IR)

clang -S -emit-llvm source.c -o source.ll

(д) Оптимизация (машинно-независимая)

clang -O3 -S -emit-llvm source.c -o source\_opt.ll

(е) Генерация ассемблерного кода

clang -S source.c -o source.s

Результат: Ассемблерный код (файл .s).

3. Ассемблирование (Assembling) Действие: Преобразование ассемблерного кода в машинный (объектный файл).

clang -c source.s -o source.o

Результат: Объектный файл (.o или .obj).

4. Компоновка Объединение всех .o и библиотек, разрешение внешних символов Файл: исполняемый .exe / .out clang source.o -o program

3. clang и CMake. Что такое clang? LLVM? Преимущества использования clang? Что такое система сборки и зачем она нужна? Что такое CMake и в чём особенность таких систем? Что такое генератор? Какие бывают сборки в CMake? Что такое мультиконфигурация? Что такое CMakeLists? Опишите структуру CMakeLists для базового проекта.

Clang – не компилятор, а фронтенд для С (транслятор из некоторого языка в промежуточный LLVM IR). Часть LLVM.

**LLVM –** низкоуровневая виртуальная машина с набором модульных и переиспользуемых компонентов для компиляции, оптимизации и генерации кода

Преимущества: широко используемый в ПО, поддерживается 3 ОС…, поддержка фронтендов для множества языков, хорошая система вывода об ошибках, производительность, хорошая автоматическая оптимизация кода

**Система сборки** – набор инструментов, используемых для автоматизации процесса преобразования исходного кода программы в исполняемый файл или библиотеку. Задача: управлять этапами этого процесса, такими как компиляция, компоновка, препроцессинг и др.

**СMake – это система мета-сборки -** инструмент, который генерирует файлы конфигурации для других систем сборки. В отличие от традиционных систем сборки, которые напрямую управляют процессом компиляции и компоновки, мета-сборки создают промежуточные файлы, которые затем используются другими инструментами для выполнения фактической сборки.

Особенности: генерация конфигураций (Makefile, который используется другими инструментами для сборки проекта), кроссплатформенность (платформы и компиляторы), абстракция (описание процесса не привязываясь к конкретным инструментам сборки), гибкость (различные сценарии сборки и параметры можно изменять)

**Генератор** – система сборки, для которой будут сгенерированы файлы сборки

Сборки бывают двух основных видов:

* In-source – вид сборки при котором выходные файлы проекта будут располагаться в каталоге с исходными «source» файлами
* Out-of-source – вид сборки при котором выходные файлы проекта будут располагаться в отдельном от каталога с исходными «source» файлами месте

Чтобы собрать проект для начала опишем простейший файл конфигурации CMakeLists.txt:

* [cmake\_minimum\_required](https://cmake.org/cmake/help/latest/command/cmake_minimum_required.html)(VERSION 3.30) – данная команда обозначает минимальную необходимую версию CMake для работы с проектом
* [project](https://cmake.org/cmake/help/latest/command/project.html)(HelloWorld C) – устанавливает имя нашего проекта
* [add\_executable](https://cmake.org/cmake/help/latest/command/add_executable.html)(Hello src/Source.c) - добавление исполняемого файла в проект, используя указанные исходные файлы

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Основная настройка проекта перекладывается на конфигурационный файл верхнего уровня:



* [add\_subdirectory](https://cmake.org/cmake/help/latest/command/add_subdirectory.html)() – добавляет в сборку другой каталог
* [include](https://cmake.org/cmake/help/latest/command/include.html)() – команда которая подключает модули CMake к текущему проекту
* [check\_language](https://cmake.org/cmake/help/latest/module/CheckLanguage.html)() – команда которая проверяет доступность компиляторов для того или иного языка
* [if](https://cmake.org/cmake/help/latest/command/if.html)(), else(), endif() – команды ветвления
* [enable\_language](https://cmake.org/cmake/help/latest/command/enable_language.html)() – включает поддержку указанного языка в проекте
* [message](https://cmake.org/cmake/help/latest/command/message.html)() – позволяет вывести сообщение на консоль
* [set](https://cmake.org/cmake/help/latest/command/set.html)() – устанавливает значение какой-либо переменной



Генератор флаг –G

Через флаг –B каталог в котором будут размещены файлы сборки проекта (что по сути делает сборку «out-of-source»)

Флаг –T инструмент для сборки

Для указания используемого компилятора для языка C необходимо использовать параметр -DCMAKE\_C\_COMPILER=<название компилятора>

После создания файлов сборки необходимо собственно собрать проект. Для этого необходимо вызвать тот сборщик, генератор которого был указан в параметре –G

Но чтобы не задумываться о всём многообразии сборщиков и как их вызывать, CMake позволяет вызывать их в общем виде



Некоторые генераторы также поддерживают мультиконфигурацию, т.е. позволяют настроить и генерировать файлы под несколько конфигурации (например, Debug и Release). Таких в настоящий момент только 2: Visual Studio и XCode

4. Файлы, отображенные в память. Что такое отображение файла в память? Для чего они применяются в ОС? Как устроен данный механизм в Windows? Классификация объектов секций в Windows? Что такое представление секции? Согласован ли доступ к данным в нескольких секциях? Алгоритм взаимодействия с файлами, отображенными в память с использованием WinAPI.  
**Файл, отображённый в память (memory-mapped file, MMF),** — механизм, позволяющий ассоциировать содержимое файла с областью виртуальной памяти процесса. Это даёт возможность работать с файлом так же, как с массивом в памяти, без необходимости явного чтения или записи через системные вызовы.  
Для чего? Загрузка и выполнение исполняемых файлов и динамических библиотек (DLL), Доступ к файлам данных на диске без ReadFile и WriteFile, Разделение данных между несколькими процессами

В Windows существует два вида объектов секций:

1. **Секции на базе файлов**: Это основной тип, используемый для отображения файлов. Объект секции напрямую связан с открытым файлом на диске. Изменения в памяти могут быть записаны обратно в этот файл.

2. **Секции на базе страничных файлов**: Этот тип не связан с конкретным файлом на диске. Память для такой секции выделяется из общего системного ресурса — страничного файла. Они используются для создания разделяемой памяти между процессами, когда не нужно сохранять данные в файле после завершения работы. Такие страницы всегда инициализируются нулями для предотвращения утечки данных.

**Представление секции** — часть проекции файла, которая фактически отображена в виртуальное адресное пространство процесса и доступна ему для работы.

Процесс может создать объект проекции для очень большого файла (например, несколько гигабайт), но отобразить в память лишь небольшую его часть (представление), например, несколько килобайт, с которыми он работает в данный момент. Это позволяет эффективно работать с огромными файлами, не занимая все адресное пространство процесса. Один процесс может создавать несколько представлений для одного и того же объекта секции, отображая разные его части.

**Согласованность ГАРАНТИРУЕТСЯ**, если несколько процессов (или один процесс) создают несколько **представлений** из **ОДНОГО** объекта "проекция файла". Все изменения, сделанные в одном представлении, будут немедленно видны в других, так как все они ссылаются на одни и те же страницы в физической памяти.

**Согласованность НЕ ГАРАНТИРУЕТСЯ**, если создать **НЕСКОЛЬКО** объектов "проекция файла" для **ОДНОГО И ТОГО ЖЕ** файла на диске. В этом случае Windows не обеспечивает когерентность данных между представлениями, созданными от разных объектов проекции.

**«проекции файла»** (file mapping object) — объект ядра, который связывает файл с областью виртуальной памяти процесса.

**Представление отображения** (mapped view) — область виртуальной памяти, связанная с объектом проекции файла. Процесс может использовать представление для чтения и записи данных в файл, работая с ним как с обычным массивом.

**CreateFile** — создание или открытие объекта ядра файл.  
**CreateFileMapping** — создаёт объект проекции файла (возвращает дескриптор объекта). Сообщает какой объем физ памяти нужен для проекции файла. Для секций на базе страничного файла вместо дескриптора передается `INVALID\_HANDLE\_VALUE`.  
**MapViewOfFile** — создаёт представление отображения. Возвращает указатель на представление файла. Проецирование файловых данных на адресное пространство процесса.  
**UnmapViewOfFile** — удаляет представление. Отключение файла данных от адресного пространства процесса.  
**FlushViewOfFile** — записывает на диск диапазон байтов в пределах отображенного представления файла.  
**CloseHandle** — закрывает объект проекции файла и файл.

5. Файлы, отображенные в память. Что такое отображение файла в память? Для чего они применяются в ОС? Как устроен данный механизм согласно POSIX? Классификация отображений согласно POSIX? Поясните каждый тип. Что такое Copy on Write? Алгоритм взаимодействия с файлами, отображенными в память с использованием POSIX API.

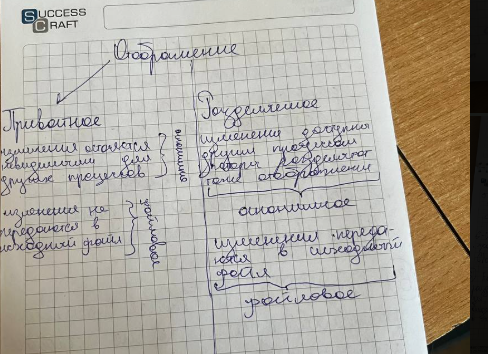
**Файл, отображённый в память (memory-mapped file, MMF),** — механизм, позволяющий ассоциировать содержимое файла с областью виртуальной памяти процесса. Это даёт возможность работать с файлом так же, как с массивом в памяти, без необходимости явного чтения или записи через системные вызовы.  
Для чего? Загрузка и выполнение исполняемых файлов и динамических библиотек (DLL), Доступ к файлам данных на диске без ReadFile и WriteFile, Разделение данных между несколькими процессами.

В POSIX-совместимых системах не присуща дополнительная сложность в виде объектов секций. Ядро ОС отображает объект напрямую в память процесса.

Механизм работает следующим образом:

**1. Источник отображения:** Отображение может быть создано на основе одного из двух типов объектов:

**Файловые** (File-backed mapping) — данные хранятся в файле, а изменения могут быть синхронизированы с диском.  
**Анонимные** (Anonymous mapping) — не привязаны к файлу и существуют только в оперативной памяти (например, для выделения больших массивов или разделяемой памяти).

**2. Подкачка по требованию** (Demand Paging): Страницы файла не загружаются в физическую память сразу в момент создания отображения. Загрузка происходит только тогда, когда процесс впервые обращается к виртуальному адресу, попадающему в эту страницу.  


**Copy-on-Write (CoW, копирование при записи)** — это метод оптимизации, который используется ядром для приватных отображений (`**MAP\_PRIVATE**`)

Суть метода: когда несколько процессов разделяют одну и ту же страницу памяти (например, после `fork()` или при отображении одного файла), эта страница изначально помечается как "только для чтения". Пока процессы только читают данные, они все используют одну и ту же физическую страницу.

Как только один из процессов пытается записать что-либо в эту страницу, происходит следующее: 1. Выполнение процесса прерывается. 2. Ядро создает приватную копию этой страницы специально для этого процесса. 3. Таблица страниц этого процесса обновляется так, чтобы она указывала на новую, скопированную страницу. 4. Процессу разрешается выполнить операцию записи уже в свою собственную копию.

Таким образом, изменения остаются локальными для процесса, который их произвел, и не влияют на другие процессы.

**open** – получение файлового дескриптора. Для анонимных пропускается этот шаг.

**mmap** — создаёт отображение в виртуальном адресном пространстве процесса. С возвращ. адрес отображения. `flags` (тип отображения: `MAP\_SHARED` или `MAP\_PRIVATE`, а также `MAP\_ANONYMOUS` для анонимных).

**mprotect** — изменяет права доступа к отображённой области.  
**msync** — синхронизирует разделяемое отображение с отображенным файлов. (изменения становятся видимыми для MAP\_SHARED)  
**munmap** — удаляет отображение из виртуального адресного пространства.

**close** – закрыть файл.

6. Библиотеки. Что такое библиотека? Какова причина возникновения библиотек? Какие бывают библиотеки? Что такое связывание? Какие виды связывания существуют? Как они соотносятся с типами библиотек? Поясните каждый из видов связывания.

**1. Проблемы больших исходных файлов:**

**- Долгое время компиляции:** Любое незначительное изменение в большом файле требует его полной перекомпиляции.

**- Сложность командной работы:** Когда над одним файлом работает много людей, отслеживать изменения становится почти невозможно.

**- Сложность навигации и отладки:** Поиск ошибок и ориентирование в большом объёме кода требуют значительных усилий. Для решения этих проблем код стали разбивать на функционально законченные модули (исходные файлы).

**2. Проблемы управления множеством объектных файлов:**

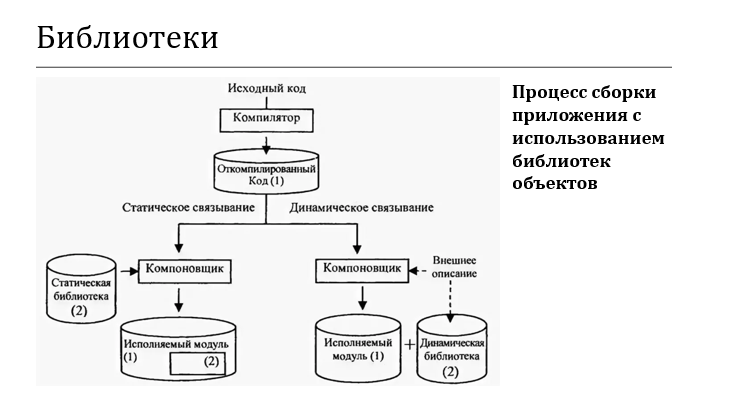
**- Повторное использование кода:** Некоторые модули могут использоваться в нескольких программах, и чтобы не компилировать их каждый раз, их компилируют в объектные файлы.

**- Сложность компоновки:** На этапе компоновки необходимо указывать длинный список всех объектных файлов.

**- Беспорядок в проекте:** Большое количество объектных файлов создаёт неразбериху в каталоге проекта.

**Библиотека объектных файлов** – это файл, содержащий несколько obj файлов, которые будут использоваться вместе на стадии сборки (**связывания**, линковки) программы. Нормальная библиотека содержит **символьный индекс**, состоящий из названий функций, переменных и т. д., которые содержатся в библиотеке. Это позволяет ускорить процесс сборки программы

Библиотеки объектов бывают двух видов: **статические и динамические (разделяемые)**

**Связывание (или компоновка, линковка)** — это процесс сборки программы из множества отдельных объектных модулей. Поскольку каждый модуль компилируется независимо, он не содержит реальных адресов функций и данных, расположенных в других модулях. Задача связывания — разрешить эти внешние ссылки, то есть вставить правильные адреса в места обращений к коду и данным из других модулей, создавая единый исполняемый файл.

**Виды связывания** - раннее(во время трансляции или во время сборки) и позднее(при загрузке или отложенное[итеративное(явная загрузка), декларативное(неявная загрузка)]).

**Раннее связывание - СТАТИЧЕСКАЯ**

**1. Во время сборки (link-time):** Это основной вид связывания для статических библиотек. Компоновщик находит в библиотеке необходимые объектные модули и полностью копирует их код и данные в конечный исполняемый файл. В результате получается самодостаточный файл, не зависящий от внешних файлов библиотек во время выполнения.

**2. Во время трансляции:** Это особый случай раннего связывания. Например, компилятор MSVC с помощью директивы препроцессора `#pragma comment(lib, ...)` может инициировать связывание с библиотекой непосредственно в процессе компиляции (трансляции) исходного кода.

**Позднее связывание - ДИНАМИЧЕСКАЯ**

**1. Отложенное декларативное (неявная загрузка).**

**Как работает:** При компоновке приложения компоновщик не копирует код из библиотеки, а лишь записывает в исполняемый файл информацию о том, какие DLL/SO-файлы ему нужны и какие функции из них он использует (в специальную **таблицу импорта**).

**Когда происходит:** Когда пользователь запускает программу, загрузчик ОС анализирует таблицу импорта, находит нужные динамические библиотеки на диске, загружает их в адресное пространство процесса и настраивает все адреса вызовов. Это происходит до того, как начнёт выполняться основной код программы.

**2. Отложенное императивное (явная загрузка).**

**Как работает:** Приложение само, в процессе своего выполнения, решает, какую библиотеку и когда ему нужно загрузить. Программист в коде явно вызывает функции системного API (например, `LoadLibrary` и `GetProcAddress` в Windows или `dlopen` и `dlsym` в Linux) для загрузки библиотеки в память и получения адреса нужной функции.

**Когда происходит:** Связывание происходит в любой момент во время выполнения (run-time) по команде из кода программы. Это позволяет загружать функциональность "по требованию".

7. Библиотеки. Что такое статическая библиотека? Какое связывание лежит в основе статических библиотек? Как создать статическую библиотеку используя clang напрямую? Используя CMake? Как собрать приложение с использованием статических библиотек используя clang напрямую? Используя CMake? Преимущества и недостатки статических библиотек.

**Статическая библиотека** - обычный файл, содержащим копии всех помещенных в него obj файлов. Это архив, там хранятся атрибуты для каждого obj файла: права доступа, id пользователя и группы и время последнего изменения. В Linux название - **libname.a, в win пофик. Статическая библиотека содержит lib файл (obj)**

+ набор часто используемых obj файлов можно поместить в единую библиотеку, при этом не нужно будет перекомпилировать ориг код.  
+ упрощение команды для компоновки. не перечисляем список obj файлов, а указываем только имя библиотеки. Компоновщик знает, как выполнять поиск по имени и извлекать объекты, необходимые для создания исполняемого файла.   
может быть несколько копий одних и тех же obj модулей, поэтому минусы:   
-дисковое пространство хранит одни и те же obj модули  
-виртуальная память потребляется много, если несколько программ одноврем.  
-если obj модуль требует изменение, то заново компонуем все исполняемые файлы, где используется модуль

**Раннее (статическое) связывание.** На этапе компоновки или трансляции obj-код содержащийся в библиотеке внедряется в будущий исполняемый файл.

У нас использовалось на этапе трансляции. Есть еще во время сборки - используется pragma.

**Создание статической библиотеки с использованием clang напрямую:**

1. **Скомпилируйте исходные файлы библиотеки в объектные файлы (.o).**

clang -target i386-pc-win32 -I.\include\ -c -o ..\bin\bsearch\_i.o .\src\bsearch\_i.c   
clang -target i386-pc-win32 -I.\include\ -c -o ..\bin\bsearch\_r.o .\src\bsearch\_r.c   
clang -target i386-pc-win32 -I.\include\ -c -o ..\bin\barray.o .\src\barray.c

LINUX

clang -I./include/ -c -o ../bin/bsearch\_i.o ./src/bsearch\_i.c  
clang -I./include/ -c -o ../bin/bsearch\_r.o ./src/bsearch\_r.c  
clang -I./include/ -c -o ../bin/barray.o ./src/barray.c

Используйте утилиту **llvm-ar** для создания архива из объектных файлов.

**ar <operation>[modifiers] <archive> [files]**

○ Операция r (replace - «заменить») вставляет объектный файл в архив, заменяя существующий с тем же именем. С модификатором c (create) заставляет создавать библиотеку, если ее нет.

llvm-ar cr ..\bin\ZSSLib.lib ..\bin\b\*.o

LINUX

llvm-ar cr ../bin/ZSSLib.a ../bin/b\*.o  
3. (Опционально). Используйте утилиту **llvm-ranlib** для добавления символьного индекса к библиотеке. llvm-ar tv ..\bin\ZSSLib.lib

LINUX

llvm-ranlib ../bin/ZSSLib.a

Аргумент **operation** состоит из буквы, которая является кодом операции, а **modifiers** - модификаторами, влияющим на то, как эта операция будет выполняться, которые записываются как набор букв слитно с буквой операции

Аргумент **archive** представляет собой имя создаваемой библиотеки, а **files** – obj файлы, из которых она будет состоять.

Среди часто используемых операций и модификаторов можно выделить:

* **r**.Вставляет obj файл в архив, заменяя им любой существующий файл с тем же именем. Это стандартный способ создания и обновления архивов. Используя с модификатором **c,** заставляет создавать библиотеку, если ее нет.

**Сборка приложения с использованием статических библиотек с использованием clang напрямую:**

Используйте драйвер компилятора (например, clang) для компоновки объектных файлов приложения с файлом статической библиотеки.

○ Windows: clang -target i386-pc-win32 -I..\ZSSLib\include\ -o ..\bin\Lab-03a.exe .\src\Lab-03a.c ..\bin\ZSSLib.lib  
 ○ На Linux очень важен порядок входных файлов:

clang -I../ZSSLib/include/ -o ../bin/Lab-03a ./src/Lab-03a.c ../bin/ZSSLib.a

**СНАЧАЛА ФАЙЛЫ ПРИЛОЖЕНИЯ, ЗАТЕМ БИБЛИОТЕКИ.** Это правило исходит из необходимости сначала передать код, в котором используется функция, а только потом код, в котором она объявлена.

Изображение выглядит как текст, программное обеспечение, Мультимедийное программное обеспечение, Значок на компьютере

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как текст, Шрифт, программное обеспечение, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

[**include\_directories**](https://cmake.org/cmake/help/latest/command/include_directories.html) – указывает на каталоги, которые будут использоваться для поиска заголовочных файлов (include)

[**add\_library**](https://cmake.org/cmake/help/latest/command/add_library.html) – необходимо собрать библиотеку из указанных файлов (в данном случае статическую поскольку указан параметр STATIC) (.c)

[**target\_link\_libraries**](https://cmake.org/cmake/help/latest/command/target_link_libraries.html) – для связывания приложения с библиотекой. Также через неё можно передавать и другие параметры компоновщику

Cборка примера на Linux без CMake отличия: название libtest.**a** вместо libtest.**lib**, значение опции target **x86\_64-pc-linux** вместо **x86\_64-pc-win32**

Также стоит отметить, что очень важен порядок, в котором передаются входные файлы драйверу компилятора (clang не компилятор, а программа, которая вызывает всё что необходимо для сборки приложения):

В примере выше раннее связывание **во время сборки (link-time)**. Компилятор MSVC поддерживает связывание во время трансляции через директиву препроцессора [**pragma**](https://learn.microsoft.com/en-us/cpp/preprocessor/comment-c-cpp?view=msvc-170):

Изображение выглядит как Шрифт, снимок экрана, текст, Графика

Контент, сгенерированный ИИ, может содержать ошибки.

В случае использования данной директивы рассмотренный ранее пример может быть собран следующей командой: (здесь не нужна .lib, он используется в pragma)



8. Библиотеки. Что такое разделяемая (динамическая) библиотека? В чем ключевая идея таких библиотек? Какой механизм ОС лежит в основе работы разделяемых библиотек? Какие способы подключения разделяемых библиотек существуют? Как создать разделяемую библиотеку используя clang напрямую? Используя CMake? Преимущества и недостатки разделяемых библиотек.

**Разделяемые** (**динамические**) **библиотеки (в Win Dynamic Link Library, в Linux разделяемые объекты shared objects)**: одна копия obj модуля разделяется между всеми программами, задействующими его. Obj модули **не копируются** в компонуемый исполняемый файл; вместо этого **единая копия библиотеки** загружается в память при запуске первой программы, которой требуются ее obj модули. Если позже будут запущены другие программы, использующие эту разделяемую библиотеку, они обращаются к копии, уже загруженной в память.

+можно загружать в адресное пространство динамически, что позволяет приложению подгружать нужный код.  
+ использование разных ЯП (приложение c#, логика с++)  
+ проще управлять проектом  
+ экономия памяти (исполняемые файлы требуют меньше места на диске и в виртуальной памяти (при выполнении))  
+ разделение ресурсов

+ упрощение локализации

- сложность поиска библиотеки

- нужен файл библиотеки

DLL-файл представляет собой файл в формате **Portable Executable** ([**PE**](https://learn.microsoft.com/en-us/windows/win32/debug/pe-format))

SO-файл представляет собой файл в формате **Executable and Linkable Format** ([**ELF**](https://gist.github.com/x0nu11byt3/bcb35c3de461e5fb66173071a2379779))

Чтобы приложение (или другая DLL) могло вызывать функции, содержащиеся в DLL, образ ее файла нужно сначала спроецировать на адресное пространство вызывающего процесса. Это достигается либо за счет неявного связывания при загрузке, либо за счет явного – в период выполнения.

Как только DLL спроецирована на адресное пространство вызывающего процесса, ее функции доступны всем потокам этого процесса. Фактически библиотеки при этом теряют почти всю индивидуальность: для потоков код и данные DLL – просто доп код и данные, оказавшиеся в адресном пространстве процесса. Когда поток вызывает из DLL какую-то функцию, та считывает свои параметры из стека потока и размещает в этом стеке собственные локальные переменные. Кроме того, любые созданные кодом DLL объекты принадлежат вызывающему потоку или процессу – DLL ничем не владеет!

в конечный DLL:  
Библиотека импорта - LIB-файл, нужен для любого exe-модуля, содержит идентификаторы.   
Раздел экспорта - таблица экспортируемых идентификаторов.  
Относительный виртуальный адрес каждого идентификатора.

**Как создать разделяемую библиотеку используя clang напрямую?**

clang -target i386-pc-win32 -shared -I .\include\ -o ..\bin\ZSSDLib.dll .\src\ZSSDLib.c .\src\bsearch\_i.c .\src\bsearch\_r.c .\src\barray.c  
clang -target i386-pc-win32 -shared -I .\include\ '-Wl,/def:.\src\ZSSDLib.def' -o ..\bin\ZSSDLib.dll .\src\ZSSDLib.c .\src\bsearch\_i.c .\src\bsearch\_r.c .\src\barray.c

LINUX

clang -target x86\_64-pc-linux -I./include/ -o ../bin/LibZSSd.o -fPIC -c ./src/LibZSSd.c  
clang -target x86\_64-pc-linux -shared -o ../bin/[LibZSSd.so](https://libzssd.so/) ./src/LibZSSd.c

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, Мультимедийное программное обеспечение, программное обеспечение, Графическое программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

**Как создать разделяемую библиотеку используя CMake?**

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

clang -target i386-pc-win32 -I .\include\ -L..\bin -o ..\bin\Lab-03b-ex.exe .\src\Lab-03b-ex.c -lZSSDLib -lZSSDLib++ -luser32

LINUX

clang -o ../bin/Lab-03c ./src/Lab-03c.c -ldl

Стоит обратить внимание на тот факт, что в примере файл mylib.dll располагается в том же каталоге, что и исполняемый файл. Это связано с тем, где система будет искать библиотеку. Если для используемой библиотеки используется только её имя (нет полного пути), то поиск DLL осуществляется в следующей последовательности:

* Каталог с EXE
* Текущий каталог процесса
* Системный каталог Windows
* Основной каталог Windows
* Каталоги PATH

9. Библиотеки.. Что такое неявный способ подключения разделяемой (динамической) библиотеки? Опишите алгоритмы неявного связывания с использованием clang и CMake. Какое связывание лежит в основе неявного подключения? Что такое библиотека импорта? Что такое раздел экспорта? Какие способы экспорта функций существуют? Как их реализовать?

Как можно заметить в коде и заголовочном файле библиотеки применяются модификаторы:

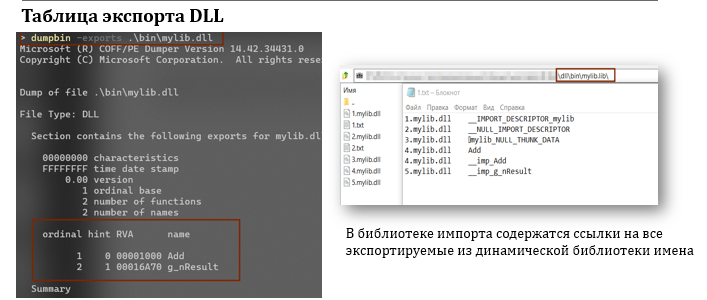
* [**\_\_declspec(dllimport)**](https://learn.microsoft.com/en-us/cpp/cpp/dllexport-dllimport?view=msvc-170)– такой модификатор означает, что данная переменная или функция импортируются из DLL (указывается с исполняемых модулях или библиотеках, которые зависят от других библиотек)
* [**\_\_declspec(dllexport)**](https://learn.microsoft.com/en-us/cpp/cpp/dllexport-dllimport?view=msvc-170)– такой модификатор означает, что данная переменная или функция экспортируется из DLL (указывается в самой библиотеке)

**\_\_declspec(dllexport)**

Если он указан перед переменной, прототипом функции или C++-классом, компилятор С/С++ встраивает в конечный OBJ-файл дополнительную информацию. Она понадобится компоновщику при сборке DLL из OBJ-файлов.

Обнаружив такую информацию, компоновщик создает LIB-файл со списком идентификаторов, экспортируемых из DLL. Этот LIB-файл нужен при сборке любого EXE-модуля, ссылающегося на такие идентификаторы и называется **библиотекой импорта**

Компоновщик также вставляет в конечный DLL-файл **таблицу** экспортируемых идентификаторов – [**раздел экспорта**](https://learn.microsoft.com/en-us/windows/win32/debug/pe-format), в котором содержится список (в алфавитном порядке) идентификаторов экспортируемых функций, переменных и классов. Туда же помещается [**относительный виртуальный адрес**](https://learn.microsoft.com/en-us/windows/win32/debug/pe-format)(**relative virtual address, RVA**) каждого идентификатора внутри DLL-модуля



Как можно заметить из таблицы экспорта, у каждой функции есть **порядковый номер** ([**ordinal**](https://learn.microsoft.com/en-us/cpp/build/exporting-functions-from-a-dll-by-ordinal-rather-than-by-name?view=msvc-170)) и **имя** (**name**)

Соответственно существуют два способа экспорта функций по номеру и по имени

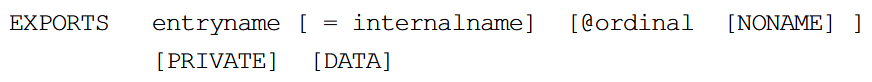
Чтобы экспортировать по имени достаточно применить модификатор **\_\_declspec(dllexport)**

Изображение выглядит как текст, Шрифт, снимок экрана, число

Контент, сгенерированный ИИ, может содержать ошибки.



def-файл (файл определений)  
+ настраивать экспорт по порядковому номеру  
+ указать конечное имя для функции  
+ область видимости экспортируемых функций  
+ можно и без имен, чисто по порядковым номерам



Параметр ***entryname (один обязательный)*** - имя экспортируемой функции или переменной. Если это имя отличается от внутреннего имени функции или переменной в DLL, то это внутреннее имя задается параметром ***internalname***.

Параметр **@*ordinal*** задает порядковый номер экспортируемого имени

Атрибут ***DATA*** - экспортируемое имя является именем переменной

Ключевое слово ***PRIVATE*** запрещает размещение экспортируемого имени в библиотеке импорта.

Атрибут ***NONAME*** указывает на то, что в таблице экспорта не будет храниться данное экспортируемое имя. В этом случае импорт адреса возможен только по его порядковому номеру

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки. Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

Эти файлы удобно использовать тогда, когда DLL содержит большое число экспортируемых функций и переменных. В этом случае поиск экспортируемых имен занимает очень много времени. Поэтому для более быстрой работы приложения функции и переменные импортируются по их порядковым номерам. Полезен при неявном связывании, когда весь импорт системой производится по номерам и в именах нету смысла, а их отсутствие может сэкономить ресурсы.

Если мы говорим о создании библиотек на языках отличных от С, то скорее всего при создании символов для таблицы экспорта/импорта компилятор преобразует имена из кода в понятные для себя имена. Данный процесс называется **name decoration** или **name mangling** (компилятор C такого не делает, он сохраняет имя функций)

.def-файл позволяет оставить имена функций в том виде, который понятен человеку

Альтернативным способом сохранения имени функции будет модификатор **extern “C”**

  
Изображение выглядит как текст, снимок экрана, меню, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

**\_\_declspec(dllimport)**

Импортируя идентификатор, необязательно прибегать к \_\_declspec(dllimport) – можно использовать стандартное ключевое слово extern языка C. Но компилятор создаст чуть более эффективный код, если ему будет заранее известно, что идентификатор, на который мы ссылаемся, импортируется из LIB-файла DLL-модуля

Разрешая ссылки на импортируемые идентификаторы, компоновщик создает в конечном EXE-модуле [**раздел или таблицу импорта**](https://learn.microsoft.com/en-us/windows/win32/debug/pe-format) (**imports section**). В нем перечисляются DLL, необходимые этому модулю, и идентификаторы, на которые есть ссылки из всех используемых DLL

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

**Общая картина работы с DLL (неявный способ)**

Изображение выглядит как текст, Шрифт, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.  
Изображение выглядит как текст, снимок экрана, диаграмма, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Попытка запуска провалилась, хоть и файл разделяемой библиотеки находится в той же папке. Связано это с тем, где Linux ищет файлы библиотек, а именно: динамический компоновщик анализирует список динамических зависимостей программы и находит соответствующие библиотечные файлы, используя набор заранее заданных правил. Часть этих правил основывается на списке стандартных каталогов, в которых обычно хранятся разделяемые библиотеки (к примеру, /lib и /usr/lib). Причина сообщения об ошибке, приведенного ранее, заключается в том, что библиотека находится в текущем каталоге, которая не учитывается динамическим компоновщиком при выполнении поиска.

Для оповещения динамического компоновщика о том, что разделяемая библиотека находится в нестандартном месте, можно воспользоваться переменной среды **LD\_LIBRARY\_PATH**, указав соответствующий каталог в качестве одного из элементов списка, разделенного двоеточиями. После указания текущего каталога приложение успешно запустилось.

**Таблица импорта программы**

Изображение выглядит как текст, Шрифт, Мультимедийное программное обеспечение, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Как можно заметить в отличие от прошлого варианта изменилось следующее:

* Подключать заголовочный файл не требуется
* Появились вызовы функций API для работы с DLL
* При сборке приложения больше не требуется указывать использование DLL

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

10. Библиотеки.. Что такое явный способ подключения разделяемой (динамической) библиотеки? Опишите алгоритмы явного связывания с использованием clang и CMake. Какое связывание лежит в основе явного подключения? Что такое раздел импорта? Что такое name mangling? Как его избежать?

**2 СПОСОБ** (ЯВНАЯ ЗАГРУЗКА DLL в период выполнения приложения))

Поток приложения явно загружает DLL в адресное пространство процесса, получает виртуальный адрес необходимой DLL-функции и вызывает ее по этому адресу.

+ подключать .h не требуется

- появились вызовы функций API для работы с DLL  
+ при сборке приложения не требуется указывать DLL

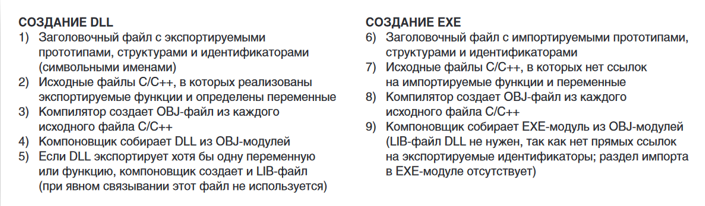
Изображение выглядит как текст, снимок экрана, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.



Таблица импорта. Если вызвать утилиту dumpbin, то для текущего примера мы не найдём в таблице импорта нашей библиотеки, ей не откуда там взяться, компоновщик ничего не знает о библиотеке в данном случае.

**Общая картина работы с DLL (явный способ)**



Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

**Явная загрузка SO-файла**

* Появились вызовы функций API для работы с SO
* При сборке приложения больше не требуется указывать использование библиотеки

test.c

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.



**Функции WinAPI для работы с DLL**

В любой момент поток может спроецировать DLL на адресное пространство процесса, вызвав одну из двух функций:

Изображение выглядит как текст, Шрифт, снимок экрана, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Обе функции ищут образ DLL-файла и пытаются спроецировать его на адресное пространство вызывающего процесса.

Если необходимость в DLL отпадает, ее можно выгрузить из адресного пространства процесса, вызвав функцию:



Поток получает адрес экспортируемого идентификатора из явно загруженной DLL вызовом **GetProcAddress**:

Изображение выглядит как текст, Шрифт, белый, типография

Контент, сгенерированный ИИ, может содержать ошибки.

Заметьте: тип параметра ***pszSymbolName*** – PCSTR, а не PCTSTR. Это значит, что функция **GetProcAddress** принимает только ANSI-строки – ей нельзя передать Unicode-строку. А причина в том, что идентификаторы функций и переменных в разделе экспорта DLL всегда хранятся как ANSI-строки

Вторая форма параметра ***pszSymbolName*** позволяет указывать порядковый номер нужной функции:



Здесь подразумевается, что Вам известен порядковый номер (2) искомого идентификатора, присвоенный ему автором данной DLL. Microsoft настоятельно не рекомендует пользоваться порядковыми номерами

**Функции POSIX для работы с SO**

Функция [**dlopen**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/dlopen.html) загружает в виртуальное адресное пространство вызывающего процесса разделяемую библиотеку с именем libfilename и инкрементирует счетчик открытых ссылок на нее :

Изображение выглядит как текст, Шрифт, линия, алгебра

Контент, сгенерированный ИИ, может содержать ошибки.

Если разделяемая библиотека, указанная с помощью аргумента libfilename, зависит от других библиотек, то **dlopen** загрузит их автоматически. При необходимости эта процедура выполняется рекурсивно. Мы будем называть набор загруженных таким образом библиотек **деревом зависимостей.**

Функция [**dlclose**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/dlclose.html) закрывает библиотеку:

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Функция [**dlsym**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/dlsym.html) ищет именованный символ (symbol – функцию или переменную) в библиотеке, на которую указывает дескриптор (handle), и в ее дереве зависимостей.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

При получении ошибки из **dlopen** или другой функции, входящей в программный интерфейс **dlopen**, можно попытаться узнать ее причину, получив указатель на соответствующую строку с текстом ошибки, используя вызов [**dlerror**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/dlerror.html)**:**

Изображение выглядит как текст, Шрифт, алгебра, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

11. Библиотеки. Общий алгоритм загрузки и очистки разделяемой (динамической) библиотеки в/из памяти. Функции жизненного цикла разделяемых библиотек в Windows и Linux. Что такое DLL Injection? Алгоритм внедрения DLL в Windows с помощью удаленных потоков. Алгоритм внедрения SO в Linux.

**Общий алгоритм загрузки и отчистки библиотеки из памяти**

Функцию **dlopen**/**LoadLibrary** можно вызвать несколько раз для одной и той же библиотеки. При этом загрузка будет выполнена лишь при первом вызове, а во всех последующих случаях станет возвращаться одно и то же значение HANDLE

Однако программный интерфейс **dlopen**/**LoadLibrary** хранит счетчик ссылок для каждого дескриптора

С каждым вызовом **dlopen**/**LoadLibrary** он инкрементируется, а декрементация происходит при вызове **dlclose**/**FreeLibrary**; последний выгружает библиотеку из памяти только в том случае, если счетчик равен 0

В DLL может быть указана функция входа/выхода (одна). Система вызывает ее в некоторых ситуациях сугубо в информационных целях, и обычно она используется DLL для инициализации и очистки ресурсов в конкретных процессах или потоках. Если Вашей DLL подобные уведомления не нужны, Вы не обязаны реализовывать эту функцию. Пример – DLL, содержащая только ресурсы

В Linux в SO такой функции быть не может, однако вместо этого можно определить одну или несколько функций, которые будут автоматически вызываться при загрузке и выгрузке разделяемой библиотеки

Изображение выглядит как текст, программное обеспечение, Мультимедийное программное обеспечение, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

**DLL injection** – это техника используемая для запуска кода в адресном пространстве другого процесса путем принуждения загрузки в него DLL

Для Windows рассмотрим пример на основе внедрения DLL с помощью удаленных потоков

Для Linux рассмотрим пример на основе внедрения с помощью LD\_PRELOAD

Внедрение DLL с помощью удаленных потоков предполагает вызов функции **LoadLibrary** потоком целевого процесса для загрузки нужной DLL. Так как управление потоками чужого процесса сильно затруднено, Вы должны создать в нем свой поток. К счастью, Windows-функция **CreateRemoteThread** делает эту задачу несложной.

Последовательность операций, которые Вам надо будет выполнить:

1. Выделите блок памяти в адресном пространстве удаленного процесса через **VirtualAllocEx**

2. Вызвав **WriteProcessMemory**, скопируйте строку с полным именем файла DLL в блок памяти, выделенный в п. 1

3. Используя **GetProcAddress**, получите истинный адрес функции **LoadLibraryA** или **LoadLibraryW** внутри Kernel32.dll

4. Вызвав **CreateRemoteThread**, создайте поток в удаленном процессе, который вызовет соответствующую функцию **LoadLibrary**, передав ей адрес блока памяти, выделенного в п. 1. На этом этапе DLL внедрена в удаленный процесс, а ее функция DllMain получила уведомление DLL\_PROCESS\_ATTACH и может приступить к выполнению нужного кода. Когда DllMain вернет управление, удаленный поток выйдет из **LoadLibrary** и вернется в функцию **BaseThreadStart**, которая в свою очередь вызовет **ExitThread** и завершит этот поток. Теперь в удаленном процессе имеется блок памяти, выделенный в п. 1, и DLL, все еще «сидящая» в его адресном пространстве. Для очистки после завершения удаленного потока потребуется несколько дополнительных операций

5. Вызовом **VirtualFreeEx** освободите блок памяти, выделенный в п. 1

6. С помощью **GetProcAddress** определите истинный адрес функции **FreeLibrary** внутри Kernel32.dll

7. Используя **CreateRemoteThread**, создайте в удаленном процессе поток, который вызовет **FreeLibrary** с передачей HINSTANCE внедренной DLL



Во время тестирования иногда может понадобиться переопределить функции (и другие символы), которые в обычных условиях были бы найдены динамическим компоновщиком. Для этого переменной среды LD\_PRELOAD можно присвоить строку с именами разделяемых библиотек, которые следует загрузить раньше других (имена разделяются двоеточиями)

Поскольку данные библиотеки загружаются в первую очередь, их функции, запрашиваемые программой, будут использоваться автоматически, переопределяя любые одноименные символы, которые в противном случае пришлось бы искать динамическому компоновщику

Из соображений безопасности программы, устанавливающие пользовательские и групповые идентификаторы, игнорируют переменную LD\_PRELOAD





неявное -   
.h .c .o .dll - экспорт хотя бы одной функцию и компоновщик создает lib файл  
.h .c .o - учитывает ссылки и lib создает exe

При неявном подключении (implicit linking) линкеру передается библиотека импорта (расширение lib), содержащая список переменных и функций DLL, которые могут использовать приложения. Обнаружив, что программа обращается хотя бы к одной из них, линкер добавляет в целевой exe-файл таблицу импорта. Таблица импорта содержит список всех DLL, которые использует программа, с указанием конкретных переменных и функций, к которым она обращается. Позже, когда exe-файл будет запущен, загрузчик проецирует все DLL, перечисленные в таблице импорта, на адресное пространство процесса;

В случае неявного подключения все библиотеки, используемые приложением, загружаются в момент его запуска и остаются в памяти до его завершения (даже если другие запущенные приложения их не используют). Это может привести к нерациональному расходу памяти, а также заметно увеличить время загрузки приложения, если оно использует очень много различных библиотек. Кроме того, если хотя бы одна из неявно подключаемых библиотек отсутствует, работа приложения будет немедленно завершена. Явный метод лишен этих недостатков, но делает программирование более неудобным, поскольку требуется следить за своевременными вызовами LoadLibrary и соответствующими им вызовами FreeLibrary, а также получать адрес каждой функции через вызов GetProcAddress.

Явное связывание, требует, чтобы в программе содержались конкретные указания относительно того, когда именно необходимо загрузить или освободить библиотеку DLL. Далее программа получает адрес запрошенной точки входа и использует этот адрес в качестве указателя при вызове функции. В вызывающей программе функция не объявляется; вместо этого в качестве указателя на функцию объявляется переменная. Поэтому во время компоновки программы присутствие библиотеки не является обязательным. Для выполнения необходимых операций требуются три функции: LoadLibrary (или LoadLibraryEx), GetProcAddress и FreeLibrary.

Если используемая вами библиотека является статической, компоновщик скопирует объектный код этих функций непосредственно из библиотеки и вставит его в исполняемый файл.  
Если библиотека является динамической, компоновщик не будет вставлять объектный код, а вставит заглушку, которая по сути сообщает, что эта функция находится в этой DLL в этом месте.

Неявное связывание — ОС загружает библиотеку DLL в тот момент, когда она используется EXE. EXE вызывает экспортированные функции DLL так же, как статически скомпонованные и включенные в состав самого EXE функции. Явное связывание — ОС загружает DLL по запросу во время выполнения. EXE должен явно загружать и выгружать ее. Кроме того, в нем должен быть настроен указатель функции для доступа к каждой используемой функции из библиотеки DLL. В отличие от вызовов функций в статически скомпонованной или неявно связанной библиотеке DLL, при работе с явно связанной DLL EXE должен вызывать экспортированные функции с помощью указателей функций.

12. Registry.. Что такое реестр Windows? Каковы причины его возникновения? В каких случаях стоит использовать реестр? Каких видов бывают данные в реестре? Опишите структуру реестра. Какие типы данных поддерживаются в реестре? Каковы ограничения? Назовите пять основных ульев и опишите их назначение. Опишите API для работы с реестром.

**Реестр Windows** – централизованная иерархическая бд, хранящая информацию о параметрах конфигурации ОС и установленных приложений.

Основной причиной появления реестра стало желание заменить множество разрозненных файлов инициализации единой системной бд. Файлы инициализации имели ряд недостатков:

● Они записывались в формате ASCII и могли быть доступны и случайно изменены любым текстовым редактором.

● Часто портились или случайно удалялись.

● Даже самые невинные изменения могли привести к необратимым последствиям, нарушая работу всей системы.

Реестр Windows был введен для решения проблем, связанных с хрупкостью и децентрализованным характером INI-файлов, предоставляя более структурированный и устойчивый способ хранения конфигурационных данных.

С точки зрения прикладной разработки, рекомендуется не использовать Реестр для хранения данных, связанных непосредственно с приложениями или пользователями. Вместо этого, приложения должны хранить такую информацию в файловой системе, используя форматы вроде INI, XML, JSON или YAML.

Однако, в случае системной разработки под ОС Windows, вам придется столкнуться с Реестром. Даже для прикладной разработки, иногда бывает удобно хранить небольшую информацию в Реестре. Один из распространенных методов — сохранение в значении реестра только пути к файлу, где хранится основная часть информации. Это помогает реестру работать более эффективно, поскольку длинные значения (более 2048 байт) должны храниться в файле.

С точки зрения прикладного разработчика, важнее знать, как приложения используют реестр для записи и чтения системной информации.

**Данные в реестре бывают двух видов:**

● **Энергозависимые** (volatile) – создаются во время работы системы и удаляются из Реестра при ее завершении.

● **Энергонезависимые** (non-volatile) – данные, которые сохраняются в файлах на постоянном накопителе (HDD, SSD).

Реестр разделен на ульи (hives) или кусты. Каждый улей содержит определенную информацию. Хотя стандартный инструмент просмотра RegEdit показывает 5 ульев, "реальными" из них являются только два: **HKEY\_USERS** и **HKEY\_LOCAL\_MACHINE**. Все остальные являются комбинацией данных из этих двух.

**HKEY\_USERS** - Содержит параметры всех пользователей. SID пользователя. Подразделы SID содержат различные настройки для каждого пользователя (рабочий стол, консоль, переменные среды и т.д.).

**HKEY\_LOCAL\_MACHINE** - Глобальные настройки для всей системы. Большая часть данных в нем очень важна для правильного запуска системы. По умолчанию только пользователи уровня администратора могут вносить изменения

Важные подразделы включают:

○ SOFTWARE – здесь установленные приложения обычно хранят свою не относящуюся к пользователям информацию.

○ SYSTEM – здесь хранится большинство системных параметров и информация, считываемая системными компонентами при запуске. Интересные для разработчиков подразделы SYSTEM включают Services (службы и драйверы), Enum (физические устройства), Control (системные параметры), BCD00000000 (Boot Configuration Data), SECURITY (локальные политики безопасности).

**HKEY\_CURRENT\_USER** - Настройки текущего пользователя (ссылка на инфу из HKEY\_USERS)

**HKEY\_CURRENT\_CONFIG** - Текущая конфигурация оборудования

**HKEY\_CLASSES\_ROOT** - Информация о ассоциациях файлов, OLE, COM-объектах

Внутри ульев доступ к реестру осуществляется через разделы или ключи реестра (registry keys), которые играют роль каталогов в файловой системе. Раздел может содержать подразделы и параметры. Каждый параметр имеет имя, тип и значение.

Записи реестра, называемые параметрами, могут содержать данные типов:

● REG\_NONE – пустой тип записи.

● REG\_SZ – Нуль-терминированная Unicode-строка.

● REG\_EXPAND\_SZ - Нуль-терминированная Unicode-строка (может содержать неразвернутые переменные окружения в %%).

● REG\_BINARY – двоичные (любые) данные.

● REG\_DWORD – 32-битное значение (Little-Endian).

● REG\_DWORD\_LITTLE\_ENDIAN – аналогично прошлому.

● REG\_DWORD\_BIG\_ENDIAN - 32-битное значение (Big-Endian).

● REG\_LINK – символьная ссылка (Unicode).

● REG\_MULTI\_SZ – несколько Unicode-строк, разделенных NULL-символом, два

NULL-символа обозначают конец значения.

● REG\_QWORD – 64-битное значение (Little-Endian).

● REG\_QWORD\_LITTLE\_ENDIAN – аналогично прошлому

**Существуют некоторые ограничения на значения, хранимые в Реестре:**

● Имя раздела (подраздела): максимальная длина **255 символов** (полный путь начиная от названия улья).

● Имя параметра: **16,383 Unicode символа**.

● Значение параметра**: 1МБ в стандартном варианте**, в последних версиях Windows может быть использована вся доступная память.

● Дерево разделов может быть в глубину **до 512 уровней**, при возможности создать 32 уровня за один API вызов.

● При хранении путей к файлам в реестре, все символы "" должны быть **экранированы** ("\").

**● Длинные значения (более 2048 байт) должны храниться в файле, а местоположение файла сохранено в реестре**.

Для работы с Реестром из программ используется API реестра Windows, который содержит множество функций, включая операции CRUD (Create, Read, Update, Delete). Приложение всегда обращается к разделам относительно уже открытого раздела. **Существуют предопределенные открытые разделы (ульи):** HKEY\_LOCAL\_MACHINE, HKEY\_CLASSES\_ROOT, HKEY\_USERS и HKEY\_CURRENT\_USER.

Windows работает с Реестром через **аналог дескрипторов, называемый HKEY**, который является дескриптором раздела Реестра. При работе с HKEY действуют те же правила, что и с HANDLE: открывать разделы только при необходимости и закрывать дескрипторы, когда они больше не нужны.

Основные функции API для работы с Реестром:

1. Открытие или создание раздела:

○ RegOpenKeyEx: открывает существующий раздел. Если раздел не существует, функция завершится с ошибкой.

○ RegCreateKeyEx: создает раздел. Если раздел уже существует, функция открывает его.

2. Обе функции принимают базовый дескриптор (hKey), имя подраздела (lpSubKey), опции (ulOptions), требуемую маску доступа (samDesired), атрибуты безопасности (lpSecurityAttributes), а также возвращают дескриптор открытого/созданного раздела (phkResult). Маски доступа вроде KEY\_READ и KEY\_WRITE используются для определения прав. Параметр dwOptions в RegCreateKeyEx определяет, является ли раздел энергозависимым или энергонезависимым.

3. Чтение параметров (значений):

○ RegQueryValueEx: читает значение параметра. Принимает дескриптор раздела, имя параметра (lpValueName), указатель на тип данных (lpType), буфер для данных (lpData), и указатель на размер буфера/данных (lpcbData).

○ RegGetValue: аналогична RegQueryValueEx, но позволяет ограничивать типы возвращаемых значений (через dwFlags) и предпочтительна для строковых параметров, так как гарантирует нуль-терминацию.

4. Запись параметров (значений):

○ RegSetValueEx: записывает значение параметра. Принимает дескриптор раздела, имя параметра, тип данных, буфер с данными и размер данных.

○ RegSetKeyValue: практически идентична RegSetValueEx, но позволяет указать подраздел (lpSubKey) относительно базового hKey для установки параметра без необходимости явно открывать подраздел.

5. Удаление разделов или параметров: ○ RegDeleteKey или RegDeleteKeyEx: удаляют подраздел. Могут удалить раздел только в том случае, если у него нет дополнительных подразделов.

○ RegDeleteTree: удаляет раздел со всеми его подразделами.

○ RegDeleteValueA: удаляет параметр из раздела.

6. Удаленный раздел помечается для удаления и фактически удаляется только после закрытия всех открытых дескрипторов для этого раздела.

7. Закрытие дескриптора раздела:

○ RegCloseKey: закрывает дескриптор раздела (HKEY). Принимает дескриптор раздела для освобождения. Следует всегда закрывать дескрипторы, когда они больше не нужны, чтобы избежать утечки ресурсов.

8. Принудительная запись данных на диск:

○ RegFlushKey: явно записывает измененные данные Реестра на жесткий диск. Использует много системных ресурсов и должна вызываться только в случае крайней необходимости. RegCloseKey не обязательно записывает данные на диск перед возвратом.

Важно помнить, что все описанные функции требуют соответствующих прав доступа, которые указываются при открытии или создании разделов.

13. COM. Что такое Component Object Model (далее – COM)? Два свойства лежащих в основе COM? Почему COM называют двоичным стандартом? Что такое COM-компонент? Что такое COM-интерфейс? Два типа COM-интерфейсов. Чем характеризуется COM-интерфейс? Назовите два стандартных COM-интерфейса. Что такое GUID? CLSID? IID?

**Component Object Model** – объектная модель компоненты (объект) (**двоичный стандарт**, тк компоненты могут быть написаны на разных языках) фирмы Microsoft является *моделью* для создания компонентных объектов. **Независимость от языка** программных модулей, в которых соблюдается определенный двоичный стандарт. **Независимость от местоположения** означает, что клиент не обязательно должен знать, где находится определенный компонент.

**COM-компонент (объект)** представляет собой программный модуль (сущность, имеющая состояние и методы доступа, чтобы изменить это состояние).

**СОМ интерфейс** – контракт, состоящий из прототипов функций без реализации.

Экземпляр «реализации интерфейса» на самом деле является указателем на массив указателей на методы (таблицу функций, ссылающуюся на реализации всех методов, определенных в интерфейсе, также называемую виртуальной таблицей). Интерфейс в С++ ссылается на все функции, поддерживаемые классом. COM-интерфейс же ссылается на предварительно оговоренную группу связанных функций, реализуемых COM-классом, но **не обязательно на ВСЕ функции**, поддерживаемые классом.

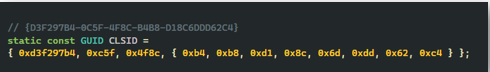
Интерфейсы бывают двух типов: **стандартные** и **произвольные**

За стандартными интерфейсами закреплены предопределенные GUID-идентификаторы. **IUnknown**. Все остальные интерфейсы являются производными (наследуют все методы) от **IUnknown**. Каждый компонент должен поддерживать как минимум стандартный интерфейс **IUnknown**

Среди стандартных интерфейсов также стоит выделить **IClassFactory**. Отвечает за управление жц компонентов путём реализации паттерна «фабрика классов». Не обязательный, но с ним становится проще следить за тем какие объекты были созданы и когда можно их выгружать из памяти.

**CoClass** – класс, поддерживающий набор методов и свойств, с помощью которых можно взаимодействовать с объектами этого класса. Набор методов и свойств - **COM-интерфейс**

Каждый CoClass имеет два идентификатора – текстовый **ProgID** и предназначен для человека, и бинарный **CLSID -** глобально уникальным идентификатором (**GUID**).С помощью CLSID можно точно указать, какой именно объект требуется. Тип данных GUID применяется и для идентификации COM-интерфейсов – **IID.**



14. COM. Какие типы COM-контейнеров бывают? Что такое COM-сервер? Что такое COM-клиент? Назовите типы COM-серверов. Что такое «однокомпонентные» и «многокомпонентные» COM-сервера? Что должен «знать» COM-клиент, чтобы использовать COM-объект? Алгоритм создания «однокомпонентного» COMсервера. Что такое IDL?

Для размещения компонентов в Windows могут быть применены **два вида контейнеров**: DLL-файл и EXE-файл.

Приложения, использующие COM-компоненты, называют **COM-клиентами**, а контейнеры с расположенными в них компонентами – **COM-серверами.**

В зависимости от типа контейнера и места его расположения (локальное или удаленное) различают несколько типов серверов: **INPROC** (DLL, локальный), **LOCAL** (EXE, локальный), **REMOTE** (EXE, удаленный)

Существует два основных типа серверов: **in-process** (в процессе) и **out-of-process** (вне процесса). Серверы **in-process** реализуются в динамической библиотеке (DLL), а серверы **out-of-process** реализуются в исполняемом файле (EXE). Серверы **out-of-process** могут размещаться либо на локальном компьютере, либо на удаленном компьютере. Кроме того, COM предоставляет механизм, который позволяет серверу **in-process** (DLL) запускаться в суррогатном процессе EXE, чтобы получить преимущество выполнения процесса на удаленном компьютере.

Построение **in-process** и **out-of-process** серверов ничем не отличается с точки зрения структуры, однако при работе с **out-of-process** серверами возникает некоторая сложность, а именно: как получить указатель на функцию или объект которые располагается в другом процессе

В таком случае между клиентом и сервером появляется прослойка в виде прокси-объекта. Для создания таких объектов необходимо будет применять MIDL

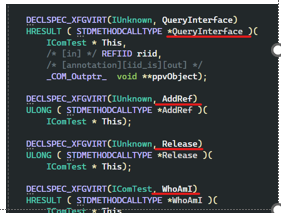
Соответственно, если в контейнере расположен только один компонент, то сервер – «однокомпонентный», если два и более – «многокомпонентный». При этом COM-сервер сам может выступать в виде клиента, если он вызывает методы интерфейсов, реализованные другими компонентами.

При работе с COM-компонентом **клиент** **должен** «**знать**» только **CLSID**, **IID**, тип сервера, **сигнатуры методов** произвольных интерфейсов компонента.

Рассмотрим создание однокомпонентного COM-сервера:

1. Описать COM-интерфейс который обязан наследоваться хотя бы от **IUnknown.** Тут есть два пути - с помощью языка IDL и скомпилировать его используя MIDL или на языке С/С++ напрямую.

В итоге использования MIDL один из получившихся файлов будет являться заголовочным и содержать описание вашего интерфейса примерно следующего вида:

**MIDL** (**MS Interface Definition Language**) – язык описания интерфейсов, который позволяет программе или объекту, написанному на одном языке, взаимодействовать с другой программой, написанной на неизвестном ему языке.

Midl ..\IComTest.idl

IDL файлы содержат описание COM-компонентов и COM-интерфейсов не зависящим от языка способом.

Инструкция IDL **import** используется для ввода заголовочного файла

Атрибут **object** идентифицирует интерфейс как объектный и указывает компилятору MIDL генерировать код прокси/заглушки вместо клиентских и серверных заглушек RPC. Методы объектного интерфейса должны возвращать значение типа HRESULT

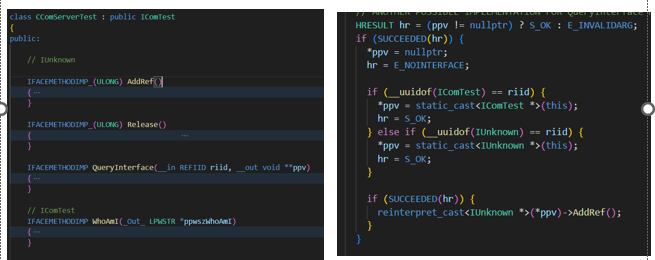
Атрибут **uuid** определяет идентификатор интерфейса (IID).

Атрибут **version** определяет конкретную версию среди нескольких версий COM-интерфейса.

Атрибут **helpstring** определяет символьную строку, которая используется для описания элемента, к которому он применяется

Ключевое слово **interface** определяет имя интерфейса. Все интерфейсы объектов должны быть производными, прямо или косвенно, от Unknown.

2. Создать COM-компонент путём написания класса который реализует ранее полученный COM-интерфейс



3. Реализовать фабрику классов.COM требует, чтобы каждый класс имел собственную фабрику классов для создания экземпляров, но многие классы фактически могут использовать одну и ту же реализацию фабрики классов.

4. Реализовать набор из 5 обязательных функций DLL которые обеспечивают работу одного или нескольких COM-компонентов и которые обязательно экспортируются из DLL

5. Скомпилировать COM-сервер

6. Зарегистрировать COM-сервер с использованием утилиты **regsvr32** (по сути данная утилита просто вызывает некоторые экспортируемые функции из DLL)

Для реализации свойства «Независимости от местоположения» в Windows каждый СОМ-компонент должен быть зарегистрирован в Windows-реестре. Для регистрации компонента и применяется специальная утилита **regsvr32**

7. Разработать COM-клиент:

* Работа клиента должна начинаться с инициализации библиотеки OLE32 (вызов функции **CoInitializeEx**)
* Для создания экземпляра компонента необходимо вызвать функцию **CoCreanteInstance**
* Для получения указателя на другие интерфейсы можно применить метод **QueryInterface**
* Работа клиента должна завершаться освобождением библиотеки OLE32 (вызов функции **CoUninitialize**)

15. COM. Интерфейс IUnknown. Перечислите методы интерфейса IUnknown и поясните их назначение. Что такое «счетчик ссылок на интерфейсы»? Для чего он нужен? Каким образом и когда этот счетчик увеличивается и уменьшается? Какое соглашение о вызове и возврате должен обеспечивать метод COM-объекта? Какие методы являются исключением? Поясните назначение типа HRESULT и его структуру.

**IUnknown** методы жизненного цикла:

* **AddRef** – увеличивает счётчик ссылок на интерфейс на 1
* **QueryInterface** – получение указателя на интерфейс по IID
* **Release** - уменьшает счётчик ссылок на интерфейс на 1

Данный «**счётчик ссылок на интерфейс**» необходим для отслеживания момента, когда экземпляр COM-компонента больше не требуется и может быть удалён

Все методы COM-интерфейса должны поддерживать соглашение о вызовах **stdcall**, а также возвращать **HRESULT** за исключением **AddRef** и **Release** – они возвращают текущее значение счётчика ссылок на интерфейс.

За небольшим исключением все функции компонента должны возвращать результат в виде значения **HRESULT**,которое имеет следующую структуру:

30-31 биты HRESULT отображают успешность выполнения функции COM-компонента

29 бит HRESULT отображает кем был определен данный статус код: пользователем или системой

28 бит HRESULT является зарезервированным

16-27 биты HRESULT отображают к какой технологии относится статус код

0-15 биты HRESULT отображают точный результат в рамках заданной технологии и серьезности

Чтобы определить, был ли вызов успешен или произошла ошибка, можно воспользоваться макросами: **SUCCEEDED**() – успех и **FAILED**() – неудача



16. COM. Интерфейс IClassFactory. Что такое «фабрика классов» и для чего она нужна? Перечислите методы интерфейса IClassFactory и поясните их назначение. Поясните назначение «счетчика экземпляров компонент». Где этот счетчик увеличивается и где уменьшается? Назовите условие, при котором объект компонента удаляется. Опишите жизненный цикл COM-сервера в целом.

**IClassFactory** методы:

* **CreateInstance** – создания экземпляра COM-компонента
* **LockServer** – увеличение счётчика блокировки COM-сервера

Блокировка COM-сервера предназначена для гарантии того, что он не будет закрыт раньше времени (DLL не будет выгружена).

Фабрика классов также помогает удобно следить за жц COM-компонент. Это является важной частью работы COM-сервера, так как при попытке освободить его ресурсы, вывод о том можно это сделать или нет, основывается на том факте используются ли хоть какие-то его COM-компоненты или нет. Для этого на сервере существует такое понятие как «**счётчик экземпляров компонент**». Увеличение этого счётчика происходит в конструкторе COM-компонента (он вызывается методом **CreateInstance**), а уменьшается в деструкторе.

Жизненный цикл COM-сервера:

* Не может быть выгружен пока счётчик экземпляров компонент не равен нулю
* Экземпляр COM-компоненты не может быть выгружен пока счётчик ссылок на интерфейсы не равен нулю
* Не может быть выгружен пока счётчик блокировок (**LockServer**) не равен нулю

17. COM. Объясните в чем заключается процесс регистрации COM-объекта? Поясните назначение утилиты regsvr32 и принцип ее работы. Перечислите пять функций, которые экспортируются COM/DLL-контейнером. Поясните назначение этих функций. Работа с памятью в COM и почему она такая?

Реестр Windows – это централизованная иерархическая база данных, хранящая информацию о параметрах конфигурации операционной системы и установленных приложений по сути данная утилита просто вызывает некоторые экспортируемые функции из DLL Что это и зачем? Для реализации свойства «Независимости от местоположения» в Windows каждый СОМкомпонент должен быть зарегистрирован в Windows-реестре. Для регистрации компонента и применяется специальная утилита regsvr32 regsvr32.exe ./ComSampleServer.dll

**Принципы взаимодействия клиента и сервера**

Поддержка программ соответствующих COM-модели в операционной системе Windows обеспечивается с помощью динамически подключаемой библиотеки **OLE32.DLL** и соответствующей ей библиотеки экспорта функций **OLE32.LIB**

Именно OLE32.DLL по идентификатору CLSID через реестр ОС определяет место расположения контейнера компонента, загружает и инициализирует его.

|  |  |
| --- | --- |
| **Название функции** | **Описание функции** |
| **DllCanUnloadNow** | автоматически вызывается OLE32.DLL перед попыткой клиентом выгрузить СОМ-сервер. выгружает или не выгружает СОМ-сервер |
| **DllGetClassObject** | Первая функция компонента, вызываемая OLE32.DLL. Функция проверяет id компонента, создает фабрику классов компонента и возвращает указатель на стандартный интерфейс IClassFactory |
| **DllInstal** | вызывается утилитой regsvr32 при наличии соответствующего параметра, применяется для выполнения дополнительных действий при регистрации и удаления регистрации компонентов |
| **DllRegisterServer** | для регистрации компонентов сервера в реестре ОС |
| **DllUnregisterServer** | для удаления информации о компонентах сервера из реестра ОС |

COM определяет пару функций для выделения и освобождения памяти в куче:

* Функция [CoTaskMemAlloc](https://learn.microsoft.com/en-us/windows/desktop/api/combaseapi/nf-combaseapi-cotaskmemalloc) выделяет блок памяти
* Функция [CoTaskMemFree](https://learn.microsoft.com/en-us/windows/desktop/api/combaseapi/nf-combaseapi-cotaskmemfree) освобождает блок памяти, который был выделен с помощью [CoTaskMemAlloc](https://learn.microsoft.com/en-us/windows/desktop/api/combaseapi/nf-combaseapi-cotaskmemalloc)

Почему COM определяет свои собственные функции выделения памяти? Одна из причин заключается в том, чтобы обеспечить уровень абстракции над распределителем кучи. В противном случае некоторые методы могли бы вызывать malloc, а другие - new. Тогда вашей программе пришлось бы вызывать free в одних случаях и delete в других, и отслеживать все это быстро стало бы невозможно. Еще одним соображением является тот факт, что COM – это двоичный стандарт, поэтому он не привязан к определенному языку программирования.

18. Сервисы. Что такое сервис? Виды сервисов. Характеристики сервисов. Что такое SCM? Для чего он предназначен? Опишите структуру сервиса. Какова особенность точки входа сервиса? Что такое функция обратного вызова? Где и какая хранится информация о сервисах Windows? Что такое группа порядка загрузки?

**Сервис** (**служба)** – процесс, который выполняет служебные функции.

То есть запускается или по команде, или при загрузке ОС, заканчивает работу по команде или при завершении работы ОС.

Не каждая программа запускаемая со стартом ОС является сервисом.

**Примеры:**

Фоновый процесс, который обеспечивает доступ к бд – **серверы**

Программы, обеспечивающие доступ к внешним устройствам, – **драйверы**

Сервис как процесс, отслеживающий работу приложения, – **мониторы**

**Характеристики сервисов:**

* Только фоновый режим
* Не имеют собственного управляющего интерфейса (ни GUI, ни TUI)
* Управляются специальной программой ОС – менеджером служб
* Запуск/стоп по команде, входом/выходом пользователя, стартом/выключением ОС.
* Предоставление услуг другим программам или ОС, а не пользователям

**SCM** – это Service Control Manager (Менеджер Управления Службами). Это специальная программа операционной системы Windows, которая управляет работой сервисов. Её функции включают поддержку базы данных установленных сервисов, запуск/остановку сервисов, передачу им управляющих запросов и предоставление информации об их состоянии.

Должен содержать две функции обратного вызова (функция в аргументах другой функции), которые вызываются ОС.   
Одна функция - точка входа сервиса (сам сервис), вторая - реагирует на управляющие сигналы от ОС. Функции ServiceMain, ServiceCtrlHandler.

Отсюда следует, что если консольное приложение определяет один сервис, то оно должно иметь следующую структуру:

Изображение выглядит как текст, чек, Шрифт, алгебра

Содержимое, созданное искусственным интеллектом, может быть неверным.

main - запускает диспетчер сервиса, который управляет сервисом и является потоком.

Если определяется только 1 сервис, то функция называется [**ServiceMain**](https://learn.microsoft.com/en-us/windows/win32/services/service-servicemain-function). Возможны другие. Если же в приложении определяется несколько сервисов, то каждый имеет свое имя. Эта функция содержит два параметра, которые аналогичны параметрам **main**.  
  
Запуск диспетчера - StartServiceCtrlDispatcher - должна быть вызвана из main в течение 30 секунд после запуска программы, если нет - неудача. Чтобы не было этого, инициализировать сервис можно в теле ServiceMain, а инициализация глобальных переменных в отдельном потоке функции main.  
Диспетчер сервиса передает запросы ServiceCtrlHandler от менеджера сервисов, которая обрабатывает все запросы.  
Несколько сервисов - несколько диспетчеров – свои функции обработки управляющих запросов.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.  
  
Подключение обработчика к сервису- RegisterServiceCtrlHandler  
Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Точка входа - ServiceMain  
Должна выполнять:  
1. запустить RegisterServiceCtrlHandler (обработчик управляющих команд от менеджера сервисов) немедленно  
2.вызвать функцию SetServiceStatus - состояние SERVICE\_START\_PENDING  
3. локальная инициализация сервиса  
4. установить в рабочее состояние SetServiceStatus SERVICE\_RUNNING  
5. выполнять работу, реагируя на команды  
6. после SERVICE\_STOPPED освободить ресурсы и закончить работу

Изображение выглядит как текст, снимок экрана, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

SCM поддерживает бд установленных сервисов в реестре.

**HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services**

содержит подраздел(название службы) для каждого сервиса

Информация о сервисе:

* DependOnGroup – группы порядка загрузки сервисов, от которых зависит данный сервис
* DependOnService – сервисы, от которых зависит данный сервис
* Description – описание
* DisplayName – отображаемое имя
* ErrorControl – уровень управления ошибками
* FailureActions – действия выполняемые при возникновении ошибки
* Group – группа порядка загрузки сервисов, в которой состоит данный сервис
* ImagePath – путь к исполняемому файлу
* ObjectName – учетка от имени которой запускаем
* Start – тип запуска
* Tag – уникальный тег для данного сервиса в рамках группы порядка загрузки в которой он состоит
* Type – тип сервиса

**Группы порядка загрузки** – логически объеденный набор сервисов, который определяет порядок загрузки сервисов входящих в него относительно остальных групп или сервисов

В целом порядок загрузки сервисом определяется следующим образом:

1. Порядок групп в списке групп порядка загрузки  
   **HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Control\ServiceGroupOrder**
2. Порядок загрузки сервисов в рамках своей группы  
   **HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Control\GroupOrderList**

Остальные зависимости каждого из сервиса

19. Сервисы. Что такое демон? Опишите и поясните алгоритм создания процессадемона вручную. Рекомендации при создании демонов. Что такое systemd и init? Опишите процесс создания сервиса на примере systemd или init.

**Демон - процесс со свойствами: длинный жц, фоновый режим без контролирующего терминала.**

* **Особенности**:
  + Запускаются при старте системы, завершаются при выключении.
  + Не имеют терминала (не получают сигналы SIGINT, SIGHUP).
  + Название обычно заканчивается на d (например, httpd).

Чтобы стать демоном:  
1. fork, родитель рип, потомок продолжает (становится потомком init).  
2. setsid, новая сессия (отвязка от терминала)  
3. fork, чтобы не стать лидером сессии  
4. Очистить атрибут unmask чтобы были права на файлы и каталоги созданные демоном  
5. смена рабочего каталога процесса на корневой (система не сможет быть отключена, если будет не корневой)  
6. закрыть файловые дескрипторы, которые демон унаследовал от родителя  
(Демон закрывает дескрипторы 0, 1 и 2, чтобы освободить ресурсы, так как они больше не необходимы в фоновом режиме и могут ссылаться на терминал, который уже потерян.)

1. Закрывая дескрипторы 0, 1 и 2 и перенаправляя их в /dev/null, демон предотвращает ошибки ввода/вывода и исключает возможность повторного открытия этих дескрипторов для записи в стандартные потоки.

**Обработка сигнала SIGTERM**: Демон должен корректно обрабатывать сигнал SIGTERM, освободив все ресурсы перед завершением, чтобы избежать утечек памяти и других проблем. При выключении компьютера отправляется процессом **init** всем своим потомкам. По умолчанию этот сигнал приводит к завершению процесса. Если демону перед этим необходимо освободить какие-либо ресурсы, он должен делать это в обработчике данного сигнала.

Демон должен быстро освобождать ресурсы, так как через 5 секунд после SIGTERM будет отправлен сигнал SIGKILL, который принудительно завершит процесс.

**Закрытие файловых дескрипторов**: Демон должен внимательно следить за закрытием всех файловых дескрипторов, чтобы избежать утечек и ошибок, особенно с длинным жизненным циклом.

**Единственный экземпляр**: Для предотвращения запуска нескольких экземпляров демона, демон должен использовать блокировку файла, удостоверяясь, что только один его экземпляр работает в системе.

Init (SysV Init) и Systemd это менеджеры сервисов (системы инициализации) в Linux, которые отвечают за запуск, остановку и управление демонами (службами).

**Init**: Более старая система, использует скрипты в /etc/init.d/. Управляется в основном командой service. Считается устаревшей.

**Systemd**: Более новая и современная система, использует конфигурационные файлы (unit-файлы, например, .service) обычно в /etc/systemd/system/. Управляется командой systemctl.

Обе системы могут выполнять процесс «демонизации» (отсоединение от терминала и т.д.) для запускаемых программ и позволяют настраивать автозапуск служб при загрузке ОС.

1. Скомпилировать демон.
2. Поместить бинарный файл /usr/sbin/<demoserviced>.
3. Создать файл описания < demoserviced >.service в /etc/systemd/system. **Изображение выглядит как текст, снимок экрана, Шрифт

   Содержимое, созданное искусственным интеллектом, может быть неверным.**
4. Создать /etc/demoserviced/< demoserviced >.config для конфигураций. (необязательно)
5. Логи хранить в /var/log.
6. Проверить распознавание: systemctl status < demoserviced >.
7. Управлять: systemctl start/stop/restart < demoserviced >.
8. Для автозапуска: systemctl enable < demoserviced >

20. Драйверы. Что такое драйвер? Какое место занимает драйвер в структуре ОС? Основные концепции драйверов. Что такое подсистема ввода/вывода? Какие функциональные возможности она предоставляет? Перечислите из чего состоит подсистема ввода/вывода?

Драйвер — часть ПО ОС, которая обеспечивает взаимодействие с аппаратными устройствами. Он предоставляет API для взаимодействия с аппаратным обеспечением, позволяя ОС и приложениям управлять физическими и виртуальными устройствами.

Драйверы занимают важное место в ядре ОС, обеспечивая взаимодействие между ОС и аппаратными устройствами. Они позволяют ОС и приложениям взаимодействовать с устройствами через подсистему ввода/вывода.

**Основные концепции драйверов:**

1. **Легко заменяемые модули ОС:** Драйверы могут быть добавлены, удалены или обновлены без изменений в ядре ОС.
2. **Работа с драйверами как с файлами:** Драйверы взаимодействуют с системой через стандартные файловые операции (например, open, close, read)
3. **Использование IOCTL (Input/Output Control Code):** Этот механизм позволяет контролировать операции ввода/вывода на низком уровне.
4. **Типы драйверов:** Драйверы могут быть файловыми, сетевыми, для устройств Plug and Play и другие

Подсистема ввода/вывода — это набор компонентов, которые управляют устройствами и предоставляют интерфейсы для приложений. Она включает средства безопасности, поддержку асинхронных операций, динамическую загрузку драйверов, поддержку Plug and Play и управление энергопотреблением.

**Из чего состоит подсистема ввода/вывода?**

1. **Диспетчер ввода/вывода** — основной компонент, который соединяет приложения и системные компоненты с устройствами.
2. **Драйверы устройств** — обеспечивают интерфейсы для работы с конкретными типами устройств.
3. **Поддержка Plug and Play** — для автоматического обнаружения и настройки устройств.
4. **Подсистема управления электропитанием** — для управления энергопотреблением устройств.

21. Драйверы. Что такое драйвер устройства? Что такое диспетчер ввода/вывода? Какого его назначение? Что такое PnP-диспетчер и каково его назначение? Что такое  
диспетчер электропитания? Для чего используется реестр в случае с драйверами  
и что такое INF-файлы? Что такое HAL?

Драйвер устройства — это программный модуль, который управляет конкретным типом устройства. Он выполняет команды на низком уровне, такие как запись в регистры управления, для выполнения операций с устройством. Драйвер принимает высокоуровневые команды от диспетчера ввода/вывода и преобразует их в низкоуровневые операции, которые могут быть выполнены на конкретном устройстве

Диспетчер ввода/вывода (I/O Manager) — это центральный компонент подсистемы ввода/вывода операционной системы, который управляет взаимодействием между приложениями и устройствами через драйверы. Он передает запросы на ввод/вывод, управляет IRP-пакетами и предоставляет необходимые интерфейсы для драйверов.

PnP-диспетчер (Plug and Play Manager) работает в режиме ядра и совместно с диспетчером ввода/вывода, а также драйверами устройств, чтобы автоматически распознавать и управлять подключением и отключением устройств. Он также управляет выделением аппаратных ресурсов для новых устройств и координирует установку необходимых драйверов для их работы.

Диспетчер электропитания (Power Manager) управляет переходами системы и отдельных устройств в режимы с низким энергопотреблением. Это помогает экономить электроэнергию, позволяя системе и устройствам переходить в режимы ожидания или спящего режима, когда они не используются. Он тесно связан с PnP-диспетчером и диспетчером ввода/вывода для эффективного управления энергопотреблением устройств.

Реестр используется для хранения параметров инициализации драйверов и конфигурации устройств. INF-файлы управляют установкой драйверов, связывая устройства с драйверами, и содержат инструкции для их установки.

HAL (Hardware Abstraction Layer) — это слой абстракции аппаратуры, который изолирует драйверы от специфики процессоров и контроллеров прерываний, скрывая различия между платформами и аппаратными средствами, обеспечивая единообразие работы драйверов на разных архитектурах.

22. Драйверы. Что такое драйвер? Опишите «жизненный цикл» IRP. Что такое виртуальные файлы? Особенности программирования на уровне ядра. Что такое уровни запросов прерываний? Что такое отложенные вызовы процедур? Поясните эти две концепции на примере.

IRP (Input/Output Request Packet) — это структура данных, которая описывает запрос на ввод/вывод. IRP проходит через несколько этапов: от диспетчера ввода/вывода (создание IRP пакета) к драйверам, обработка запроса и возврат результатов в диспетчер ввода/вывода, потом в приложение. Когда операция ввода/вывода завершена, IRP удаляется, и результат возвращается вызывающему процессу.

Виртуальные файлы — абстракции для различных устройств, с которыми ОС выполняет операции ввода/вывода. Это могут быть устройства (например, жесткий диск, принтер), файлы, каналы или даже почтовые ячейки. ОС абстрагирует все такие операции, предоставляя единую модель работы с ними через виртуальные файлы.

Программирование на уровне ядра требует точности, так как ошибки могут повлиять на всю систему, а не только на процесс. Также необходимо учитывать отсутствие стандартных библиотек и поддержку только SEH (Structured Exception Handling) для обработки ошибок.

IRQL (Interrupt Request Level) — приоритет, назначаемый процессору для обслуживания прерываний. Уровни IRQL определяют, насколько приоритетен запрос на прерывание по сравнению с другими. Например:

* **IRQL 0 (Passive Level):** Обычный уровень, на котором работает планировщик ядра. Код, работающий на этом уровне, может быть прерван.
* **IRQL 2 (Dispatch Level):** На этом уровне код получает бесконечный квант времени, и планировщик не может прерывать выполнение потока.
* **IRQL выше 2:** Коды, работающие на более высоких уровнях IRQL, имеют приоритет и могут блокировать обработку прерываний с более низкими уровнями

DPC — это механизмы, которые позволяют отложить выполнение задач, требующих низкого уровня IRQL, после завершения обработки прерываний, чтобы не блокировать другие прерывания.

**Пример использования DPC:**

1. Когда происходит аппаратное прерывание (например, с устройства), на уровне IRQL, соответствующем прерыванию (DIRQL), выполняется обработка прерывания (ISR).
2. Если в ходе обработки нужно выполнить дополнительные операции, они ставятся в очередь как DPC.
3. После завершения работы ISR уровень IRQL понижается до 2 (Dispatch Level), и система начинает выполнять DPC в порядке очереди, не блокируя другие прерывания

23. Драйверы. Что такое драйвер? Какие бывают драйверы? Что такое WDM-драйверы  
и какие они бывают? Что такое стек драйверов? Какие бывают многоуровневые  
WDM-драйверы? Опишите последовательность вызова функционала, реализованного многоуровневым драйвером.

Драйверы можно классифицировать по нескольким критериям, включая:

* **Драйверы пользовательского режима (User-Mode Drivers):** Эти драйверы работают в пользовательском режиме и взаимодействуют с операционной системой через стандартные интерфейсы. Примером могут служить драйверы для принтеров или графических карт.
* **Драйверы ядра (Kernel-Mode Drivers):** Эти драйверы работают в режиме ядра и имеют доступ ко всем ресурсам системы. Они более мощные и могут напрямую взаимодействовать с аппаратным обеспечением. Примером таких драйверов являются драйверы файловой системы или драйверы сетевых устройств.
* **Драйверы устройств (Device Drivers):** Эти драйверы управляют конкретными устройствами, такими как жесткий диск, видеокарта, или клавиатура.
* **Фильтрующие драйверы (Filter Drivers):** Эти драйверы изменяют или дополняют поведение других драйверов, добавляя новые функциональности или изменяя их работу.
* **Драйверы шины (Bus Drivers):** Эти драйверы управляют шинами, которые передают данные между различными устройствами и системами, например, шина PCI или USB.
* **Драйверы для устройств Plug and Play (PnP):** Эти драйверы автоматизируют процесс обнаружения и установки устройств, поддерживающих технологию Plug and Play, без необходимости вручную настраивать параметры.

WDM (Windows Driver Model) — это модель драйвера для операционных систем Windows, которая поддерживает управление электропитанием, технологию Plug and Play и другие инструменты управления. WDM-драйверы могут быть:

* **Драйверами шины (Bus drivers)** — управляют логическими или физическими шинами, передающими данные.
* **Функциональными драйверами (Function drivers)** — управляют конкретными типами устройств.
* **Фильтрующими драйверами (Filter drivers)** — изменяют поведение других драйверов или устройств.

**Стек драйверов** — это набор драйверов, которые взаимодействуют между собой для выполнения операций с устройствами. Стек драйверов состоит из нескольких уровней драйверов, где каждый драйвер выполняет свою часть работы. Запросы на ввод/вывод проходят от вышестоящих драйверов к нижестоящим. Например, когда устройство посылает запрос на чтение, драйвер устройства может передать этот запрос функциональному драйверу, который затем передаст его драйверу шины, если это необходимо. В некоторых случаях между этими драйверами могут располагаться фильтрующие драйверы, которые изменяют или модифицируют поведение запросов.

Многоуровневые драйверы включают несколько типов драйверов (сразу 3 вместе).

Последовательность вызова функционала в **многоуровневом драйвере** может быть следующей:

1. **Запрос на ввод/вывод (IRP) создается** диспетчером ввода/вывода и передается верхнему драйверу (например, фильтрующему драйверу).
2. **Фильтрующий драйвер** может изменить или дополнить запрос, добавив дополнительные операции или проверку данных.
3. **Функциональный драйвер** обрабатывает запрос, управляя устройством конкретного типа (например, жестким диском или клавиатурой). Он выполняет основные операции ввода/вывода, такие как чтение или запись.
4. **Драйвер шины** может быть задействован, если запрос требует взаимодействия с аппаратной шиной (например, для передачи данных через PCI или USB).
5. После выполнения операций в нижнем уровне драйвера, запрос передается обратно через стек драйверов, возвращая результат в приложение, которое инициировало запрос.

24. Драйверы. Что такое драйвер? Кто занимается запуском драйвера? Что для этого  
требуется: перечислите и поясните назначение. Какие дополнительные возможности может включать в себя драйвер? Что такое объекты драйвера и файла и зачем  
они нужны? Что такое файловый объект?

Запуском драйвера занимается **подсистема ввода/вывода** ОС. Для этого требуется:

* **Драйверный объект** — содержит точку входа в драйвер и его функции.
* **Объект устройства** — описывает устройство и его характеристики.
* **INF-файл** — используется для установки драйвера и связывания устройства с драйвером.
* **Реестр** — используется для хранения настроек драйвера и параметров устройства.

Драйвер может поддерживать:

* **Поддержку Plug and Play (PnP)** — автоматическое подключение и настройка устройства.
* **Поддержку управления электропитанием** — переход в низкоэнергетические режимы.
* **Поддержку обработки прерываний** — реагирование на аппаратные прерывания.
* **Фильтрацию и модификацию данных** — через фильтрующие драйверы.

**Объект драйвера** — это структура данных, представляющая сам драйвер в системе. Он содержит все точки входа (функции), которые драйвер предоставляет для взаимодействия с системой.

**Объект файла** — это структура данных, представляющая дескриптор файла в системе. Он используется для работы с виртуальными файлами и позволяет управлять операциями ввода/вывода.

**Файловый объект** — это структура данных, представляющая дескриптор устройства или другого ресурса, с которым система выполняет операции ввода/вывода. Этот объект необходим для работы с виртуальными файлами и управления операциями ввода/вывода, такими как открытие, чтение, запись и закрытие файлов.

25. Драйверы. Что такое пакет запроса на ввод/вывод (далее – IRP)? Какие бывают IRP? Опишите их. Что такое Plug and Play (далее – PnP)? Какие возможности  
предоставляет ПО с поддержкой PnP? Из чего состоит система PnP? С чем может работать PnP? Какие условия драйвер должен выполнить для осуществления  
полной поддержки PnP?

IRP (Input/Output Request Packet) — это структура данных, которая описывает запрос на операцию ввода/вывода. IRP может быть разных типов в зависимости от операции:

* **Синхронный IRP** — запрос, при котором приложение ждет завершения операции.
* **Асинхронный IRP** — запрос, при котором приложение продолжает выполнение, пока операция не завершится.

Plug and Play (PnP) — это технология, которая позволяет автоматически обнаруживать, настраивать и управлять подключенными устройствами без необходимости вмешательства пользователя.

ПО с поддержкой PnP предоставляет:

* Автоматическое распознавание подключенных устройств.
* Автоматическое распределение ресурсов (например, порты ввода/вывода).
* Загрузку нужных драйверов.
* Реализацию механизма для взаимодействия драйверов с новым оборудованием.

Система PnP состоит из двух частей:

* **Менеджер PnP в режиме ядра** — взаимодействует с ОС и драйверами для конфигурирования устройств.
* **Менеджер PnP в пользовательском режиме** — взаимодействует с установочными компонентами и приложениями для настройки устройств.

PnP может работать с физическими (клава, принтер, видеокарта, сетевые адаптеры), виртуальными (виртуальные диски или сетевые интерфейсы) и логическими (логические тома) устройствами.

Для полной поддержки PnP драйвер должен:

* Иметь функцию **DriverEntry** для инициализации.
* Иметь функцию **AddDevice** для добавления устройств.
* Реализовывать функции **DispatchPnp (обрабатывает запросы PnP, связанные с подключением/отключением устройств)**, **DispatchPower (изменение состояния электропитания)**, и **Unload (выгрузка драйвера и очистка ресурсов)**.
* Обеспечивать наличие **INF-файла** для установки и **cat-файла** для цифровых подписей.

26. SEH. Что такое исключение? Сравните их с прерываниями. Что такое Structured Exception Handling (далее – SEH)? Что такое блок исключения? Какие основные возможности предоставляет SEH? Что такое защищённый блок? Поясните принципы работы обработчика завершения. Что такое локальная раскрутка? Как избежать локальной раскрутки? Причины, по которым следует применять обработчики завершения?

**Исключение** – это событие, возникающее из-за выполнения определенной команды, которая вызвала ошибку процессора

Исключения в некотором роде похожи на прерывания, основное отличие заключается в том, что исключение является **синхронным** и **технически воспроизводимым** при тех же условиях, в то время как прерывание является асинхронным и может произойти в любой момент

**SEH** (Structured Exception Handling) – механизм Windows для обработки исключений на уровне системы. Он позволяет:

- Перехватывать аппаратные исключения (например, доступ к `NULL`-указателю).

- Обрабатывать программные ошибки.

- Выполнять очистку ресурсов при аварийном завершении.

Если возникает исключение, ядро перехватывает его и позволяет коду обработать исключение, если это возможно.

Хотя всю работу по отлову исключений берёт на себя операционная система, основная нагрузка по поддержке SEH ложится на компилятор, а не на операционную систему.

SEH предоставляет две основные возможности: **обработку завершения** (termination handling) и **обработку исключений** (exception handling)

Основные ключевые слова:

- `\_\_try` – начало защищённого блока.

- `\_\_except` – блок обработки исключения (фильтр + обработчик).

- `\_\_finally` – блок завершения (выполняется всегда).

**Блок исключения** – специальный код, который генерируется компилятором на входах и выходах, а также вспомогательные структуры данных для поддержки SEH.

**Защищённый блок** – блок, в котором возможны исключения, и для которого определены обработчики.

**Обработка завершений** (termination handling) — блок finally, который обрабатывается всегда при возникновении исключения.

**Локальная раскрутка** происходит, когда управление передается в блок *finally из-за преждевременного выхода из блока* try (например, с использованием return). По сути, это процесс освобождения локальных объектов. При этом компилятору приходится генерировать дополнительный код, а системе – выполнять дополнительную работу. **Глобальная раскрутка** происходит, когда ошибка вызывает выход из нескольких уровней вложенных блоков *try, и выполнение кода возвращается в блоки* finally.

Ключевое слово ***leave*** *используется для выхода из блока* try, переходя к выполнению блока \_\_finally. Это позволяет корректно завершить выполнение функции без преждевременного выхода.

AbnormalTermination. Её можно вызвать только из блока \_\_finally, и она возвращает булево значение: FALSE, если управление было передано естественным образом, и TRUE, если выход был преждевременным.

**Как избежать локальной раскрутки?**

Чтобы избежать локальной раскрутки и связанных с ней издержек, лучше не писать код, вызывающий преждевременный выход из блока \_\_try обработчика завершения. Это означает, что следует избегать операторов return, continue, break, goto (и им подобные) как из блоков \_\_try, так и из блоков \_\_finally.

Если поток управления выходит из блока \_\_try естественным образом, издержки от вызова блока \_\_finally минимальны, так как для входа в \_\_finally при нормальном выходе из \_\_try исполняется всего одна машинная команда. Преждевременный выход из \_\_try (например, с помощью return или goto) может значительно снизить быстродействие программы в зависимости от типа процессора, так как компилятору приходится генерировать дополнительный код для локальной раскрутки.

При этом стоит отметить, что использование операторов return, goto, continue и break в блоке \_\_try, связанном с блоком \_\_except, не приводит к снижению быстродействия или увеличению размера кода и не вызывает локальной раскрутки.

**Причины, по которым следует применять обработчики завершения?**

● Упрощается обработка ошибок – очистка гарантируется и проводится в одном месте. ● Улучшается восприятие текста программ.

● Облегчается сопровождение кода – при необходимости включения чего-либо в функцию для очистки достаточно добавить одну строку в блок \_\_finally.

● Удается добиться минимальных издержек по скорости и размеру кода — при условии правильного применения обработчиков (т.е. избегая преждевременных выходов из \_\_try).

27. SEH. Что такое исключение? Что такое аппаратное и программное исключения? Что такое защищённый блок? Поясните принципы работы обработчика исключений. Что такое фильтры? Какие есть стандартные фильтры и как они работают? Что такое глобальная раскрутка? Как возбудить исключения в SEH? Что такое необработанное исключение?

Примеры исключений: деление на ноль, точка останова, ошибка. Исключение, возбуждённое процессором, называется **аппаратным** (hardware exception). Операционная система и прикладные программы также способны возбуждать собственные исключения, которые называются **программными** (software exceptions).

**Обработка исключений** – это механизм перехвата и реакции на исключительные ситуации. В SEH обработка исключений осуществляется через блоки *try и* except, а фильтры проверяют тип исключения и выполняют соответствующие действия.

**Фильтр исключений** — это выражение, указанное в блоке \_\_except, которое система проверяет при возникновении исключения.

Встраиваемую функцию GetExceptionCode можно вызвать только из фильтра исключений или из обработчика исключений, чтобы получить идентификатор типа исключения. Встраиваемая функция GetExceptionInformation возвращает указатель на структуру EXCEPTION\_POINTERS, которая содержит информацию об исключении (структуры EXCEPTION\_RECORD и CONTEXT) и может быть вызвана только в фильтре исключений, так как эти структуры существуют только во время обработки фильтра.

- `EXCEPTION\_EXECUTE\_HANDLER` – выполнить обработчик.

- `EXCEPTION\_CONTINUE\_SEARCH` – передать исключение выше (найти фильтр).

- `EXCEPTION\_CONTINUE\_EXECUTION` – продолжить выполнение (если будет исключение оно будет проигнорировано).

В отличие от обработчиков завершения (\_\_finally), фильтры и обработчики исключений выполняются непосредственно операционной системой, при этом нагрузка на компилятор минимальна. Если код в блоке \_\_try не приводит к исключению, код в блоке \_\_except никогда не выполняется. При использовании \_\_except, применение операторов return, goto, continue и break в блоке \_\_try не приводит к снижению быстродействия или увеличению размера кода, а также не вызывает локальной раскрутки.

Вы можете сами генерировать программные исключения, вызывая функцию **RaiseException**.

**Необработанное исключение** (unhandled exception) возникает, если все фильтры исключений (продвигаясь вверх по дереву вызовов) возвращают EXCEPTION\_CONTINUE\_SEARCH. В таких случаях может быть вызвана особая функция фильтра, предоставляемая ОС, UnhandledExceptionFilter, которая выводит окно, указывающее на то, что поток в процессе вызвал необрабатываемое исключение, и предлагает либо закрыть процесс, либо начать его отладку.

28. Безопасное программирование. Что такое безопасное программирование? Какова его цель? Что такое уязвимость? Что такое недостаток программы? Классификация уязвимостей. Категории ошибок ПО. Список распространенных ошибок ПО. Поясните ошибку переполнения буфера и как её можно избежать. Поясните ошибку целочисленного переполнения и как её можно избежать.

Что такое безопасное программирование?  
Подход к разработке ПО, направленный на предотвращение, обнаружение и реагирование на угрозы безопасности.  
  
Цель: Защита данных, систем и пользователей от несанкционированного доступа, изменения или уничтожения.  
  
Что такое уязвимость? Недостаток программы, который может быть использован для нарушения её безопасности.  
  
Что такое недостаток программы? Любая ошибка в проектировании или реализации, которая может привести к уязвимости.  
  
Классификация уязвимостей: CVE, CWE, OSVDB, SecurityFocus BID, Secunia, IBM ISS X-Force  
  
Категории ошибок ПО: Ошибки проектирования Ошибки реализации  
  
Распространённые ошибки: SQL injection, XSS, CSRF Переполнение буфера Format string bugs Integer overflow Утечка информации, command injection Неправильное использование криптографии и т. д.  
  
Ошибка переполнения буфера:  
Возникает, когда в переменную записываются данные сверх её выделенной памяти.  
Избежание: использовать fgets() вместо scanf(), указывать длину буфера, проверка границ.  
  
Ошибка целочисленного переполнения:  
Результат арифметики выходит за пределы типа данных.  
Избежание: использовать безопасные функции, проверку границ, типы с большей ёмкостью, статический/динамический анализ.

29. Безопасное программирование. Поясните ошибку форматирования строк и как  
её можно избежать. Что такое канонизация, валидация и очистка? Что такое триада  
CIA? Какие ещё есть способы повышения безопасности в рамках ОС? Что такое  
ASLR? Что такое DEP? Что такое PoLP и какие аспекты лежат в его основе? Какие  
есть лучшие практики в области безопасного программирования?

Ошибка форматирования строк:  
Небезопасное использование printf(string); вместо printf("%s", string);.  
Избежание: Никогда не передавать пользовательские данные как строку формата напрямую.  
Канонизация, валидация, очистка:  
Канонизация: приведение данных к стандартному виду  
Валидация: проверка формата данных  
Очистка: удаление потенциально опасного содержимого  
Триада CIA: Confidentiality – конфиденциальность Integrity – целостность Availability – доступность  
Способы повышения безопасности ОС:  
Простота (минимизация поверхности атаки) Проверка авторизации Изоляция процессов и виртуальных машин  
  
ASLR, DEP, PoLP  
ASLR: рандомизация адресного пространства, усложняет эксплуатацию уязвимостей.  
  
DEP: запрет выполнения кода из области памяти, предназначенной только для данных.  
  
PoLP (Принцип наименьших полномочий):

Минимальные права  
Разделение обязанностей  
Контроль доступа по ролям  
Ограничение по времени доступа  
  
Лучшие практики:  
Проверка всех входных данных (вкл. конфигурации, переменные среды и т. д.)  
Не использовать относительные пути и ненадёжные программы  
Не рассчитывать на успешность операций – всегда проверять  
Не доверять данным, доступным для записи пользователем  
Использовать безопасные библиотеки, следить за уязвимостями и обучаться  
30. Управление доступом. Что такое контроль доступа к ресурсам? Что такое объекты и субъекты? Понятия политики безопасности и менеджера безопасности и  
связь между ними. Разница между правами и привилегиями? Порядок разработки политики безопасности. Что такое модель безопасности? Что такое состояние  
системы безопасности и какова общая задача системы безопасности?

Контроль доступа к ресурсам – механизм, разрешающий доступ к ресурсам только авторизованным пользователям.  
Объекты – пассивные ресурсы (файлы, память, принтеры).  
Субъекты – активные процессы, выполняющие действия над объектами.  
Политика безопасности – совокупность правил доступа.  
Менеджер безопасности – субъект, реализующий политику и контролирующий доступ.  
Права – конкретные разрешения (например, читать файл).  
Привилегии – более широкие полномочия (например, блокировать пользователей).  
Порядок разработки политики безопасности:  
Анализ угроз (внутренних и внешних).  
Разработка механизмов защиты (для рутинных, исключительных и аварийных ситуаций).  
Модель безопасности – формальное описание политики: объекты, операции, субъекты и атрибуты защиты.  
Состояние системы безопасности – совокупность объектов, операций, субъектов и атрибутов.  
Безопасное состояние – нет несанкционированного доступа.  
Цель – переходы только между безопасными состояниями

31. Управление доступом. Что такое дискреционная политика безопасности и в чём  
её суть? Алгоритм её построения. Что такое матрица управления доступом? Что  
такое режимы доступа к объекту? Режимы управления объектами? Опишите модели управления в дискреционной модели безопасности. Опишите два подхода к  
хранению матрицы управления.

Суть: субъект может передавать права другим субъектам по своему усмотрению (если обладает такими полномочиями).  
Алгоритм построения:  
Идентификация объектов и субъектов.  
Определение допустимых операций.  
Построение матрицы управления доступом (субъекты – строки, объекты – столбцы).  
  
Режимы доступа к объекту:  
READ, WRITE, WRITE\_APPEND, DELETE, EXECUTE, NULL и др.  
  
Режимы управления объектом:  
CONTROL – управление доступом без передачи.  
CONTROL WITH PASSING – разрешена передача полномочий.  
  
Модели управления:  
Иерархическая – древовидная структура с админом в вершине.  
Владельческая – доступ управляется владельцем объекта.  
Либеральная – передача без контроля.  
Централизованная – управление только администратором.  
  
Хранение матрицы управления:  
Построчное (по субъектам) – используется профиль пользователя.  
Столбцовое (по объектам) – используется ACL (списки управления доступом).

32. Управление доступом. Что такое маркер доступа? Что такое охраняемые объекты? Что такое дескриптор безопасности? Из чего он состоит? Поясните понятия  
DACL, SACL. Что такое учётная запись пользователя? Какие бывают? Что такое  
группа пользователей? Какие бывают? Что такое SID и какова его структура?

**Маркер доступа** – структура, ассоциированная с процессом/потоком, содержащая SID, права, привилегии.

**Охраняемые объекты** – любые ресурсы Windows (файлы, процессы, потоки), доступ к которым регулируется.

**Дескриптор безопасности** – структура, содержащая информацию об управлении доступом к объекту.

**Состоит из:**

* Владелец
* **DACL** (список управления доступом) – кто имеет или не имеет доступ.
* **SACL** (список аудита доступа) – кто и как доступался для аудита.

**Учётная запись пользователя** – запись в SAM, хранящая данные о пользователе (имя, пароль, профиль и т.д.).

**Типы учётных записей:**

* Пользователь, группа, компьютер, домен.

**Группы пользователей:**

* **Глобальные**, **локальные**, **специальные**.

**SID (Security Identifier)** – бинарный идентификатор учётной записи:

* Структура: S-R-I-SA0-SA1-…
* Используется для идентификации в системе безопасности.

33. Перехват API. Что такое перехват API-функций? Поясните как происходит выполнение кода программы в ОС. Что такое функция? Что происходит при вызове  
функции? Что такое стек вызовов и каков принцип его работы? Что такое стековый кадр? Что такое соглашение о вызовах и почему они важны при перехвате?  
Перечислите и кратко опишите какие существуют соглашения о вызовах.

 **Что такое перехват API-функций?**  
Это техника, при которой вызовы стандартных функций (например, из WinAPI) перенаправляются на пользовательские. Используется для модификации поведения программ.

 **Как происходит выполнение кода:**  
Процессор:

1. Загружает инструкцию,
2. Декодирует её,
3. Вычисляет адреса,
4. Выполняет.

 **Функция** – блок кода, выполняющий определённую задачу, вызывается по имени.

 **При вызове функции:**  
Аргументы передаются, управление переходит к функции, после return – возвращается к месту вызова.

 **Стек вызовов** – структура для хранения адресов возврата, локальных переменных, аргументов.

 **Стековый кадр** – область в стеке, выделенная под один вызов функции.

 **Соглашение о вызовах** – определяет, как передаются параметры, кто очищает стек, какие регистры сохраняются.

 **Типы соглашений:**

* stdcall – стек, очистка вызываемой функцией.
* cdecl – стек, очистка вызывающей функцией.
* fastcall – первые параметры в регистрах, остальные в стеке.
* pascal – параметры передаются слева направо, результат — в Result

34. Перехват API. Что такое перехват API-функций? Перечислите основные методы перехвата. Разделите их по критерию режима выполнения. Расскажите всё  
о перехвате API-вызовов путём модификации исходного кода: сплайсинг, трамплин, шелл-код, встраиваемый хук. Расскажите всё о перехвате API-вызовов путём модификации таблиц импорта. Для чего может использоваться перехват APIфункций?

 **Основные методы перехвата:**

* **Модификация кода (user mode):**
  + Сплайсинг (замена первых байт функции).
  + Трамплин (shellcode для перенаправления).
  + Встраиваемый хук (перехват с возвратом).
* **Модификация таблиц импорта (user mode):**
  + IAT (Import Address Table) – изменение адресов в таблице импорта.
* **Модификация системных таблиц (kernel mode):**
  + SSDT, IDT – изменения в ядре.
* **Драйверы-фильтры (kernel mode)**

 **Подробнее о методах модификации кода:**

* **Сплайсинг:** перезапись первых 5 байт на jmp, сохранение оригинального кода.
* **Трамплин:** shellcode изменяет поток выполнения.
* **Shellcode:** машинный код, вставляемый в исполняемый поток.
* **Встраиваемый хук:** позволяет вернуть выполнение оригинальной функции после перехвата.

 **Модификация IAT:**

* Изменяется адрес нужной функции в разделе импорта.
* Преимущество – не меняется код функции.
* Недостаток – можно обойти через сохранённый до этого адрес или GetProcAddress().

 **Назначение перехвата:**

* Отладка, безопасность, мониторинг, расширение функциональности.
* Злоупотребления: руткиты, кража данных, обход защиты.

35. Перехват API. Что такое перехват API-функций? Расскажите всё о перехвате APIвызовов путём модификации системных таблиц: суть, алгоритм, что такое SSDT.  
Расскажите всё об использование драйверов-фильтров для перехвата. Сложности  
перехвата в Windows. Для чего может использоваться перехват API-функций?

 **Модификация системных таблиц (SSDT):**

* **SSDT** – таблица диспетчеризации системных вызовов ядра.
* Алгоритм:
  1. Получение адреса SSDT.
  2. Нахождение нужного индекса.
  3. Сохранение оригинального адреса.
  4. Подмена на пользовательскую функцию.
  5. (Опционально) восстановление.

 **Драйверы-фильтры:**

* Замена указателей функций в объекте драйвера.
* Обрабатываются все обращения к устройствам.
* Используется DRIVER\_OBJECT, сохраняются старые указатели.

 **Сложности в Windows:**

* **PatchGuard (KPP):** защита ядра от модификаций SSDT, IDT и GDT.
* При вмешательстве может вызывать BSOD.

 **Назначение перехвата API:**

* **Добросовестное:** отладка, мониторинг, безопасность.
* **Недобросовестное:** маскировка, обход защиты, вредоносная активность.

36. Оптимизация кода. Что такое оптимизация кода? Какие характеристики могут  
быть оптимизированы? Стоит ли оптимизировать код вручную? Основные принципы проведения оптимизации. Ключевые аспекты связи оптимизации и системного программирования. Какие уровни оптимизации существуют?

* **Оптимизация кода** — преобразование кода в функционально эквивалентный, но более эффективный вариант.
* **Оптимизируемые характеристики:**
  + Скорость выполнения
  + Размер кода
  + Энергопотребление
  + Время компиляции и JIT-компиляции
* **Стоит ли оптимизировать вручную?**
  + Иногда да, но чаще лучше использовать возможности компилятора.
  + Преждевременная оптимизация — вредна (Кнут).
* **Основные принципы:**
  + Оптимизировать **при необходимости**.
  + **Понимание работы компилятора** помогает писать оптимизируемый код.
* **Связь с системным программированием:**
  + Критична для производительности, надежности и эффективности ресурсов.
* **Уровни оптимизации:**
  + Алгоритмы (независимы от языка/компилятора)
  + Структура HLL-кода
  + Учет конкретного компилятора
  + Учет машинного кода (низкоуровневые приёмы)

37. Оптимизация кода. Что такое оптимизация кода? Принципы оптимизации кода  
компилятором. Что такое анализ потока данных? Что такое базовые блоки? Зачем они нужны? Что такое упрощаемые графики потоков? Какая программа будет  
упрощаемой? Перечислите основные типы оптимизаций компилятора.

* **Принципы:**
  + Преобразование промежуточного кода без изменения поведения.
  + Компромисс между производительностью и размером.
* **Анализ потока данных (DFA):**
  + Отслеживает значения переменных: где инициализируются, используются и забываются.
* **Базовые блоки:**
  + Последовательности команд без ветвлений (кроме начала/конца).
  + Упрощают анализ и оптимизацию.
* **Упрощаемые графики потока:**
  + Программы, использующие if, while, for, без goto, поддаются лучшей оптимизации.
* **Основные типы оптимизаций:**
  + Свёртка и распространение констант
  + Удаление мёртвого кода
  + Удаление общих подвыражений
  + Упрощение операций
  + Анализ индуктивных переменных и инвариантов цикла
  + Раскрутка циклов

38. Оптимизация кода. Перечислите и поясните основные типы оптимизаций компилятора. Что такое вычисления по короткой схеме? Примерная иерархия скорости  
выполнения операторов процессором. Что такое безопасные оптимизации? Что  
такое блокировщики оптимизации? Какие существуют блокировщики? Что такое  
локальность данных? Какая бывает локальность?

* **Вычисления по короткой схеме:**
  + Логические операции && и || не вычисляют вторую часть, если результат уже очевиден.
* **Иерархия скорости операторов:**
  + Зависит от процессора. В целом: сложение < умножение < деление < операции с плавающей точкой.
* **Безопасные оптимизации:**
  + Компилятор применяет только те, что **не меняют поведение программы** при любых входных данных.
* **Блокировщики оптимизации:**
  + **Указатели**: компилятор не может гарантировать, что они не перекрываются.
  + **Вызовы функций**: могут иметь побочные эффекты — мешают оптимизации.
* **Локальность данных:**
  + **Временная** — повторное обращение к тем же данным.
  + **Пространственная** — обращение к данным, находящимся рядом в памяти.
  + **Хорошая локальность** ускоряет выполнение за счёт использования кэша.

39. Виртуализация. Что такое виртуализация? Понятие монитора виртуальных машин. Пример «виртуализации»ресурсов в рамках ОС. Что такое виртуальная машина? Что такое гостевая и хост системы? В каких направлениях должен развиваться гипервизор? Какие бывают гипервизоры и как они работают? Что является центральной концепцией в любом виде виртуализации? Какие виды виртуализации вы знаете?

**Виртуализация** — это концепция, при которой некоторая программа, называемая «монитор виртуальных машин», создает иллюзию присутствия нескольких (виртуальных) машин на одном и том же физическом оборудовании. В контексте системного программирования виртуализация решает задачи управления ресурсами, оптимизации и безопасности.

Некоторая программа - «**монитор виртуальных машин**» (гипервизор) создаёт иллюзию присутствия нескольких (виртуальных) машин на одном и том же физическом оборудовании.

Стоит заметить, что нечто подобное виртуализации вам уже должно быть знакомо: ПРОЦЕССЫ, ВИРТУАЛЬНАЯ ПАМЯТЬ, ФАЙЛЫ. Объекты/механизмы ОС позволяют программе использовать физические ресурсы компьютера, а также «верить» в то, что они принадлежат только ей. ОС здесь некоторый «гипервизор», задача которого управлять доступом к ресурсам всеми «виртуальными машинами» - процессами.

В**иртуальная машина** - изолированная среда, имитирующая физический компьютер.

Система/компьютер, на котором установлен гипервизор, называется **хост-системой**. ОС, установленная в виртуальной машине, называется **гостевой ОС**.

Гипервизоры должны хорошо проявлять себя в следующих направлениях:

●Безопасность — у гипервизора должно быть полное управление виртуализированными ресурсами.

● Эквивалентность — поведение программы на виртуальной машине должно быть идентичным поведению этой же программы, запущенной на реальном оборудовании. ● Эффективность — основная часть кода в виртуальной машине должна выполняться без вмешательства гипервизора.

2 основных типа гипервизоров:

● Гипервизоры первого типа (bare-metal). Этот тип гипервизора технически похож на ОС, поскольку это единственная программа, запущенная в самом привилегированном режиме. Его работа заключается в поддержке нескольких копий имеющегося оборудования, - виртуальными машинами. Microsoft Hyper-V, KVM, Xen (в режиме HVM).

● Гипервизоры второго типа (hosted). Программа, которая опирается на основную ОС при распределении и планировании использования ресурсов и очень похожа на обычный процесс. Притворяется полноценным компьютером с центральным процессором и различными устройствами. Многие его функциональные возможности зависят от основной ОС. Oracle VirtualBox.

Центральная концепция виртуализации - **абстрагирование физических ресурсов**

На глобальном уровне можно выделить отметить три основных типа виртуализации:

* Клиентская виртуализация (возможность виртуализации, размещенных на клиенте).
* Серверная виртуализация (к проблемам эффективного использования серверов в центрах обработки данных)
* Виртуализация хранилищ

40. Виртуализация. Что такое виртуализация? Какие виды виртуализации на глобальном уровне вы можете выделить? На какие типы делится клиентская виртуализация? Серверная? Поясните принципы контейнеризации, полной виртуализации и паравиртуализации? Виртуализация хранилищ данных.

На глобальном уровне можно выделить отметить три основных типа виртуализации:

* Клиентская виртуализация
* Серверная виртуализация
* Виртуализация хранилищ

Типы клиентской виртуализации:

* Упаковка приложений (изолирование приложения, работающее на клиентском компе, от ОС, на которой будут выполняться. Приложение не сможет изменять основные важные ресурсы ОС-> - вирусы)
  + Symantec Workspace Virtualization, VMware ThinApp
* Потоковая передача приложений (хранит приложения на серверах и загружает их на лету, обеспечивая актуальность ПО без полной установки)
  + Microsoft App-V, Citrix Virtual Apps
* Эмуляция аппаратного обеспечения. Полная виртуализация. (Программное обеспечение для виртуализации загружается на клиентский компьютер и создает контейнер для гостевой ОС — виртуальную машину)
  + Hyper-V, Oracle VirtualBox

**Существуют три основных типа серверной виртуализации:**

* **Виртуализация уровня ОС (Docker, OpenVZ, Virtuozzo).** Контейнеризация. (Позволяет запускать изолированные контейнеры внутри одной ОС, используя общее ядро хост-системы. Контейнеры изолированы друг от друга, но не требуют эмуляции аппаратуры. Это легковесный тип, не требующий запуска полноценной ОС для каждого контейнера.)
* **Полная виртуализация (Hyper-V, VirtualBox).** (На серверах ничем не отличается от таковой на клиенте. В рамках виртуальных машин может быть установлена любая ОС, так как она не знает о существовании гипервизора. Приложения запускаются в по-настоящему изолированной среде.)
* **Паравиртуализация (Xen).** (Представляет машиноподобный программный интерфейс, явно раскрывающий факт наличия виртуальной среды. В отличие от полной виртуализации, требует модификации гостевой ОС для взаимодействия с интерфейсом для корректной работы и лучшей производительности. Разработчики CPU внедрили дополнительные наборы команд (AMD-V, VT-x) для работы с виртуализацией, что позволяет запускать не модифицированные экземпляры ОС с помощью правиртуализации.)

**Виртуализация хранилищ** - технология, которая объединяет разные физические устройства хранения данных в единое логическое хранилище для централизованного управления. Пользователи и приложения работают с данными как с одним ресурсом, не зная о физической структуре. 3 уровня: физические устройства, слой виртуализации (управление распределением данных) и виртуальные диски.

Виртуализация широко используется для **тестирования и отладки ПО** (создание изолированных сред, snapshot'ы), **изоляции приложений** (песочницы, контейнеры) и **в облачных вычислениях** (масштабирование сервисов). Песочницы и контейнеры обеспечивают безопасность, выполняя недоверенный код в изолированной среде и ограничивая доступ к ресурсам.