

**Я не хочу этот пиздец учить. ПОМОГИТЕ  
Я тоже, брат**

**ЕБАНАЯ ХУЙНЯ Работаем!!!!!!!!!!!!**

**<https://www.tiktok.com/@santeluca/video/7508071555511962913>**

**Афіцыйны саўндтрэк дадзенага дока**

**<https://open.spotify.com/track/0zwAuvxUyR3ycjWjmg9TDj?si=d12e7089fe944603>**

**Нихуя, вот настоящий саунд:**

**<https://open.spotify.com/track/1OtNHTBjEubG8cU2BUypAa?si=b219122ca1954215>**

**Не, мужики, вот:**

**<https://open.spotify.com/track/3uTKAhaBZSORtCokwDTMDq?si=20d2f9acf1dc4346>**

***Ave, Caesar, morituri te salutant***

**Падрыхтована**

**Павел Буйко**

**Павел Іваноў**

**Яраслаў Янкевіч**

**БДТУ 2025**



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

## **Что такое программа?**

**Программа** — это последовательность инструкций, написанных на языке программирования, предназначенная для выполнения определённых задач компьютером. Программа обрабатывает данные и управляет ресурсами системы.

## **Что такое программное обеспечение (ПО)?**

**Программное обеспечение (ПО)** — совокупность программ и данных, предназначенных для работы на компьютере. Оно делится на разные уровни в зависимости от назначения и близости к оборудованию.

### **Системное ПО**

- Управляет аппаратными средствами компьютера.
- Обеспечивает среду для запуска прикладных программ.

### **Прикладное ПО**

- Предназначено для выполнения конкретных пользовательских задач (например, текстовые редакторы, браузеры, игры).

### **Инструментальное (сервисное) ПО**

- Средства разработки (компиляторы, отладчики).
- Утилиты для обслуживания и настройки системы.

**Системное ПО** — это программное обеспечение, обеспечивающее работу аппаратной части компьютера и предоставляющее базовую функциональность для выполнения прикладных программ.

### **Классификация системного ПО**

#### **Операционные системы (ОС)**

- Управление процессами, памятью, файлами, устройствами.
- Примеры: Windows, Linux, macOS.

#### **Драйверы устройств**

- Обеспечивают взаимодействие между ОС и аппаратными компонентами.

### **Загрузчики (bootloaders)**

- Загружают ОС при включении компьютера.

### **Утилиты и системные службы**

- Поддержка дисков, резервное копирование, настройка параметров.

### **Системы виртуализации**

- Создание виртуальных машин, управление ими.

***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.***

**Системное программирование** — это область программирования, связанная с разработкой программного обеспечения, которое взаимодействует напрямую с аппаратным обеспечением компьютера и/или предоставляет базовые функции для других программ.

**Определения снизу говна лопата, не удивлюсь, если они с лекции. Сверху нормальное**

**системное программирование** – это процесс разработки системных программ (в том числе, управляющих и обслуживающих)

И еще сверху

**системное программирование** – это разработка программ сложной структуры

Эти два определения не противоречат друг другу, так как разработка программ сложной структуры ведется именно для обеспечения работоспособности или повышения эффективности СОО(Системы обработки информации).

**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 (У нас это Clang + LLVM и Cmake).

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 трансляторов:

**Ассемблер** — преобразует текст программы на языке ассемблера в машинный код конкретного процессора.

**Компилятор** — переводит исходный код на высокоуровневом языке в машинный код или промежуточный байткод, создавая исполняемый файл.

**Интерпретатор** — выполняет программу построчно, сразу интерпретируя и исполняя команды исходного кода.

**Эмулятор** — имитирует работу одного аппаратного устройства или архитектуры на другом, позволяя запускать программы без изменения их кода.

**Перекодировщик** — преобразует код из одного формата или языка программирования в другой без обязательной привязки к аппаратной платформе.

**Макропроцессор** — обрабатывает макросы, расширяя их в исходном коде до финального текстового представления перед компиляцией.

Я НАПИСАЛ АДЭКВАТНА АЛЕ ДАКІНУЇ У ДЗІК ПІК КАБ БЫЛО БОЛЬШ ІНФЫ

**Этапы подготовки программы**

**Вот Это из Лекции**

**Назовите и опишите этапы подготовки программы. Назовите и опишите результат работы каждого из этапов.**

Подготовка программы включает несколько этапов:

1. Редактор текста:

Описание этапа: На этом этапе создается исходный код программы.

Результат работы: Исходный модуль – это программный модуль на исходном языке, который обрабатывается транслятором и представляется для него как целое, достаточное для проведения трансляции.

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 – это программный модуль, полученный в результате трансляции исходного модуля. Объектный модуль не содержит признаков того, на каком языке был написан исходный модуль. Он представляет собой программу на машинном языке с неразрешенными внешними ссылками к данным или процедурам, определенным в других модулях, которые транслируются в промежуточную форму для дальнейшей обработки.

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 – это программный модуль, представленный в форме, пригодной для загрузки в оперативную память для выполнения.

## 1 Препроцессинг (Preprocessing)

Действия:

Подключение заголовочных файлов (`#include`).

Раскрытие макросов (`#define`).

Условная компиляция (`#ifdef`, `#endif`).

Команда для Clang:

```
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).

Команда для просмотра токенов:

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

### (б) Синтаксический анализ

Построение абстрактного синтаксического дерева (AST).

Команда для просмотра 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)

Команда:

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

### (д) Оптимизация (машинно-независимая)

Команда:

```
clang -O3 -S -emit-llvm source.c -o source_opt.ll
```

(е) Генерация ассемблерного кода

Команда:

```
clang -S source.c -o source.s
```

Результат:

Ассемблерный код (файл .s).

### 3. Ассемблирование (Assembling)

Действие:

Преобразование ассемблерного кода в машинный (объектный файл).

Команда:

```
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
```

Разрешение внешних ссылок выполняется на следующем этапе подготовки, который обеспечивается **Редактором Связей (Компоновщиком)**

Он соединяет вместе все объектные модули, входящие в программу. Результатом работы Редактора Связей является загрузочный модуль

**Загрузочный модуль** – программный модуль, представленный в форме, пригодной для загрузки в оперативную память для выполнения

**Объектный модуль** – программный модуль, получаемый в результате трансляции исходного модуля

***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 мультikonфигурация? Что такое CMakeLists? Опишете структуру CMakeLists для базового проекта.***

**Clang** — это компилятор (точнее, фронтенд(што блять) блять а што тут не зразумела, у цябе ёсць кланг які да пэўнага этапа (асэблiраванiя) за ручку праводзiць за ручку твой код а пасля ўсё аддае LLVM, таму й фронтэнд (шизофрения, не мажай богоподобный фронт с этой парашей(ЛАЙК))), предназначенный для языков программирования C, C++, Objective-C и

других. Он является частью проекта **LLVM** и переводит исходный код в промежуточное представление LLVM IR, которое затем обрабатывается для генерации машинного кода

**LLVM (Low Level Virtual Machine)** — это инфраструктура для разработки компиляторов и сопутствующих инструментов. Она предоставляет набор модульных и переиспользуемых компонентов для компиляции, оптимизации и генерации кода

### Преимущества использования Clang:

- **Высокая скорость компиляции:** Clang обеспечивает быструю компиляцию по сравнению с некоторыми другими компиляторами.
- **Понятные сообщения об ошибках:** Clang предоставляет подробные и читаемые сообщения об ошибках, что облегчает отладку кода.
- **Модульная архитектура:** Позволяет легко интегрировать Clang в различные инструменты и среды разработки.
- **Поддержка современных стандартов:** Clang активно поддерживает новейшие стандарты языков C и C++, включая C++20 и частично C++23.
- **Кроссплатформенность:** Работает на различных операционных системах, включая Windows, macOS и Linux

### Что такое система сборки и зачем она нужна?

**Система сборки** — это инструмент, который автоматизирует процесс преобразования исходного кода в исполняемые программы или библиотеки. Она управляет компиляцией, связыванием и другими этапами сборки, обеспечивая правильную последовательность и учет зависимостей между файлами.

### Зачем нужна система сборки:

- **Автоматизация:** Упрощает и ускоряет процесс сборки, особенно в больших проектах.
- **Управление зависимостями:** Гарантирует, что изменения в исходных файлах приводят к пересборке только необходимых компонентов.
- **Кроссплатформенность:** Позволяет собирать проект на разных операционных системах с минимальными изменениями в настройках.

**CMake** — это кроссплатформенная система автоматизации сборки, которая генерирует файлы сборки для различных систем, таких как Make, Ninja, Visual Studio и другие. Она не выполняет сборку напрямую, а создает конфигурационные файлы, которые затем используются выбранной системой сборки

## Особенности CMake:

- **Кроссплатформенность:** Позволяет использовать один и тот же файл конфигурации на разных операционных системах.
- **Гибкость:** Поддерживает различные компиляторы и системы сборки.
- **Модульность:** Позволяет легко управлять большими проектами с множеством компонентов и зависимостей.
- **Интеграция с IDE:** Может генерировать проекты для популярных сред разработки, таких как Visual Studio и Xcode

**Генератор** в CMake определяет, какой тип файлов сборки будет создан. Например, для Unix-систем это могут быть Makefiles, для Windows — проекты Visual Studio, для кроссплатформенной сборки — файлы для Ninja.

## Примеры генераторов:

- **Unix Makefiles:** Генерирует Makefiles для использования с утилитой make.
- **Ninja:** Генерирует файлы для системы сборки Ninja, которая обеспечивает быструю и параллельную сборку.
- **Visual Studio:** Создает проекты и решения для различных версий Visual Studio.
- **Xcode:** Генерирует проекты для среды разработки Xcode на macOS.

**Мультиконфигурация** означает возможность поддерживать несколько конфигураций сборки (например, Debug и Release) в рамках одного проекта без необходимости пересоздавать файлы сборки для каждой конфигурации.

## Основные типы сборок:

- **Debug:** Включает отладочную информацию, отключает оптимизации, облегчает отладку программы.
- **Release:** Включает оптимизации, отключает отладочную информацию, предназначена для финальной версии продукта.
- **RelWithDebInfo:** Сочетает оптимизации и отладочную информацию, полезна для профилирования.
- **MinSizeRel:** Оптимизирует размер исполняемого файла, может быть полезна для ограниченных по ресурсам систем.



**CMakeLists.txt** — это основной файл конфигурации проекта в CMake. Он содержит инструкции и команды, определяющие, как должен быть собран проект: какие файлы исходного кода использовать, какие библиотеки подключать, какие флаги компиляции применять и т.д.

Каждый каталог в структуре проекта может содержать свой файл CMakeLists.txt, что позволяет организовать сборку модульно и иерархически.

Пример CMakeLists.txt

```
# Указываем минимально необходимую версию CMake
cmake_minimum_required(VERSION 3.14)

# Название проекта и его версия
project(MyProject VERSION 1.0 LANGUAGES CXX)

# Устанавливаем стандарт C++ (например, C++17) и запрещаем его понижение
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Добавляем исполняемый файл, который будет собран из main.cpp
add_executable(MyProject main.cpp)

# (Опционально) Указываем дополнительные директории с заголовками
# target_include_directories(MyProject PRIVATE include)

# (Опционально) Подключаем внешние библиотеки
# find_package(SomeLibrary REQUIRED)
# target_link_libraries(MyProject PRIVATE SomeLibrary::SomeLibrary)

# (Опционально) Устанавливаем флаги компиляции для платформ или режимов
# if(MSVC)
#     target_compile_options(MyProject PRIVATE /W4) # /W4 — высокий уровень предупреждений для MS
# else()
#     target_compile_options(MyProject PRIVATE -Wall -Wextra -pedantic)
# endif()
```

**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.***

**Отображение файла в память (memory-mapping)** — это механизм ОС, при котором содержимое файла **связывается с областью виртуальной памяти** процесса.

После того как файл отображен, процесс может обращаться к его содержимому через указатели так, как будто весь файл уже загружен в оперативную память в виде массива. Операционная система сама управляет подкачкой необходимых страниц файла с диска в физическую память по мере обращения к ним (on-demand paging).

### **Для чего они применяются в ОС?**

1. **Загрузка и выполнение исполняемых файлов и динамических библиотек (DLL).** Это самый распространенный пример. ОС отображает код и данные программы в память, что позволяет экономить время на запуске и место в страничном файле, так как страницы могут подгружаться по мере необходимости прямо из исполняемого файла.
2. **Доступ к файлам данных на диске.** Это позволяет работать с содержимым файла напрямую в памяти, избегая сложных операций файлового ввода-вывода (таких как `ReadFile`, `WriteFile`) и ручной буферизации. Вся работа по кэшированию и записи изменений на диск ложится на плечи ОС.
3. **Разделение данных между несколькими процессами** - Если несколько процессов отображают один и тот же объект "проекция файла", они получают доступ к одной и той же области физической памяти. Это один из самых быстрых способов организации общей памяти, и в Windows многие другие механизмы IPC реализованы на его основе.

### **Как устроено в Windows:**

Windows использует **трёхшаговую модель**:

1. **Открытие файла**  
`CreateFile(...)` → `hFile`
2. **Создание объекта "проекция файла"**  
`CreateFileMapping(hFile, ..., PAGE_READWRITE, ..., name)`  
→ создаёт **section object**, который ОС ассоциирует с файлом.
3. **Отображение представления файла в память**  
`MapViewOfFile(hMapping, ..., offset, size)`  
→ возвращает указатель на память, связанную с файлом.

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** — это часть объекта секции (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.

Здесь есть важный нюанс:

**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.

## 1. Шаг 1: Создать или открыть файл.

Используется функция `CreateFile`, чтобы получить дескриптор (handle) файла. Необходимо указать права доступа (`GENERIC_READ` или `GENERIC_READ | GENERIC_WRITE`).

```
HANDLE hFile = CreateFile(pszFileName, dwDesiredAccess, ...);
```

## 2. Шаг 2: Создать объект "проекция файла".

Используется функция `CreateFileMapping`. В нее передается дескриптор файла из шага 1, атрибуты защиты страниц (`PAGE_READONLY`, `PAGE_READWRITE` и т.д., которые должны быть совместимы с правами доступа к файлу) и максимальный размер, до которого файл может вырасти.

Для секций на базе страничного файла вместо дескриптора передается `INVALID_HANDLE_VALUE`.

```
HANDLE hMapping = CreateFileMapping(hFile, NULL, fdwProtect, dwMaximumSizeHigh, dwMaximumSizeLow, pszName);
```

### 3. Шаг 3: Отобразить представление файла в адресное пространство.

Используется функция `MapViewOfFile`. Она принимает дескриптор объекта проекции из шага 2, желаемый тип доступа к представлению (`FILE_MAP_READ`, `FILE_MAP_WRITE` и т.д.), смещение внутри файла, с которого нужно начать отображение, и количество байт для отображения.

Функция возвращает указатель на начало отображенной области памяти.

```
LPVOID pView = MapViewOfFile(hMapping, dwDesiredAccess, dwFileOffsetHigh, dwFileOffsetLow, dwNumberOfBytesToMap);
```

После этого с указателем `pView` можно работать как с обычным массивом в памяти.

### 1. Шаг 4: Отменить отображение представления.

Когда работа с представлением закончена, его необходимо освободить с помощью `UnmapViewOfFile`.

```
UnmapViewOfFile(pView);
```

### 2. Шаг 5: Закрыть объект "проекция файла".

Объект, созданный `CreateFileMapping`, является объектом ядра и должен быть закрыт с помощью `CloseHandle`.

```
CloseHandle(hMapping);
```

### 3. Шаг 6: Закрыть файл.

Файл, открытый `CreateFile`, также должен быть закрыт с помощью `CloseHandle`.

```
CloseHandle(hFile);
```

**Важное замечание (оптимизация):** Поскольку вызов `MapViewOfFile` увеличивает счетчики ссылок на объекты файла и проекции, их дескрипторы можно безопасно закрыть (`CloseHandle`) сразу после успешного вызова `MapViewOfFile`. Ресурсы будут автоматически освобождены системой, когда будет отменено отображение последнего представления (`UnmapViewOfFile`) и процесс завершится. Это помогает избежать утечек ресурсов, если в коде возникнет ошибка.

**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.**

Отображение файла в память — это механизм операционной системы, который позволяет связать содержимое файла на диске с регионом виртуального адресного пространства процесса. Вместо того чтобы считывать файл в буфер с помощью операций ввода-вывода, процесс получает прямой доступ к содержимому файла через указатели в памяти. Операционная система берет на себя задачу подгрузки страниц файла с диска в физическую память по мере необходимости (по требованию), когда процесс обращается к соответствующим адресам.

**Для чего они применяются в ОС?**

1. **Загрузка и выполнение исполняемых файлов и библиотек.** Это позволяет ОС эффективно загружать код и данные программы в память, экономя время и ресурсы.
2. **Доступ к файлам данных на диске.** Это упрощает работу с файлами, так как устраняет необходимость в ручной буферизации и явных вызовах функций чтения/записи.
3. **Разделение данных между несколькими процессами.** Это один из самых быстрых способов организации межпроцессного взаимодействия (IPC).

В POSIX-совместимых системах, таких как Linux, механизм устроен более прямолинейно, чем в Windows. Как указано в лекции, здесь **не присуща дополнительная сложность в виде объектов секций**". Ядро операционной системы отображает объект напрямую в память процесса.

Механизм работает следующим образом:

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

**Обычный файл:** Содержимое региона памяти инициализируется данными из файла на диске.

**Анонимный файл:** Ядро создает специальный "файл", заполненный нулями, который не связан ни с каким файлом в файловой системе. Эта область памяти используется как временное хранилище или для выделения памяти процессу (аналогично секциям на базе страничного файла в Windows).

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.

В POSIX отображения классифицируются по двум независимым атрибутам: **тип источника** (файловое или анонимное) и **тип доступа** (приватное или разделяемое). Это создает четыре комбинации:

1. **Приватное файловое отображение ('MAP\_PRIVATE' + файл)**

Пояснение: Содержимое памяти инициализируется данными из файла. Несколько процессов могут отображать один и тот же файл и изначально будут разделять одни и те же страницы памяти. Однако благодаря механизму **копирования при записи (Copy-on-Write)**, как только один из процессов пытается изменить страницу, для него создается приватная копия этой страницы.

Результат: Изменения, сделанные одним процессом, не видны другим процессам и **не записываются** обратно в исходный файл.

Применение: Инициализация сегментов кода и данных процесса при загрузке исполняемых файлов и библиотек.

## 2. Разделяемое файловое отображение ('MAP\_SHARED' + файл)

Пояснение: Все процессы, отображающие один и тот же участок одного файла, разделяют одни и те же страницы физической памяти.

Результат: Любые изменения, сделанные одним процессом в отображенной памяти, **видны всем остальным** процессам и **записываются** обратно в исходный файл на диске.

Применение: Ввод/вывод, отображенный в память: Альтернатива системным вызовам `'read'/'write'`.

Быстрое межпроцессное взаимодействие (IPC) между неродственными процессами.

## 3. Приватное анонимное отображение ('MAP\_PRIVATE' + анонимный)

Пояснение: Создается область памяти, заполненная нулями. Если процесс создает потомка с помощью `'fork()'`, тот наследует отображение, но из-за **копирования при записи** изменения родителя и потомка остаются изолированными друг от друга.

Результат: Процесс получает свой собственный, "чистый" блок памяти.

Применение: Выделение больших блоков памяти для процесса (например, используется функцией `'malloc'`).

## 4. Разделяемое анонимное отображение ('MAP\_SHARED' + анонимный)

Пояснение: Создается область памяти, заполненная нулями. При создании потомка через `'fork()'` механизм копирования при записи **не применяется**.

Результат: Родительский и дочерний процессы разделяют одну и ту же область физической памяти, и изменения, сделанные одним, видны другому.

Применение: Организация межпроцессного взаимодействия **между родственными процессами** (созданными через `'fork'`).

**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.

Алгоритм состоит из следующих шагов:

### 1. Шаг 1: Открыть файл.

Используется системный вызов ``open()``, чтобы получить файловый дескриптор (``fd``) для файла, который нужно отобразить. Для анонимных отображений этот шаг пропускается.

### 2. Шаг 2: Создать отображение.

Используется системный вызов ``mmap()``. Ему передаются:

``addr`` (обычно ``NULL``, чтобы ядро само выбрало адрес),

``length`` (размер отображаемой области),

``prot`` (права доступа к памяти: ``PROT_READ``, ``PROT_WRITE``),

``flags`` (тип отображения: ``MAP_SHARED`` или ``MAP_PRIVATE``, а также ``MAP_ANONYMOUS`` для анонимных),

``fd`` (файловый дескриптор из шага 1, для анонимных -1),

``offset`` (смещение от начала файла, с которого начинается отображение).

В случае успеха ``mmap`` возвращает указатель на начало отображённой области памяти.

### 3. Шаг 3: Работать с отображённой памятью.

Использовать возвращённый указатель для чтения и/или записи данных, как если бы это был обычный массив в памяти.

### 4. Шаг 3.5: Принудительная синхронизация.

Для разделяемых файловых отображений (``MAP_SHARED``) можно вызвать ``msync()``, чтобы принудительно записать изменения из памяти обратно в файл на диске, не дожидаясь, пока система сделает это сама.

### 5. Шаг 4: Удалить отображение.

Когда работа с областью памяти завершена, необходимо освободить ресурсы, вызвав ``munmap()`. Ему передаётся указатель, полученный от `mmap`, и размер отображения.`

## 6. Шаг 5: Закрывать файл.

Вызвать функцию ``close()`. с файловым дескриптором, чтобы закрыть файл, открытый на шаге 1.`

## ***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.***

**Библиотека** (или библиотека объектов) — это файл, который содержит один или несколько объектных файлов, сгруппированных в единую сущность. Она предназначена для использования на стадии сборки (связывания, линковки) программы.

Качественная библиотека также содержит **символьный индекс** — список имён функций, переменных и других символов, находящихся в ней. Этот индекс позволяет компоновщику быстрее находить нужные элементы, ускоряя процесс сборки приложения.

Причины возникновения библиотек вытекают из проблем, связанных с разработкой больших программ:

### 1. Проблемы больших исходных файлов:

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

Чтобы решить эти проблемы управления, набор связанных объектных файлов стали группировать в **библиотеки**.

Библиотеки бывают двух видов:

- **Статические**

- **Динамические** (также называемые разделяемыми)

**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.

- **Раннее (compile-time)**

- Во время трансляции

- Во время сборки (link-time)

- **Позднее (run-time)**

- При загрузке

- Отложенное (декларативное)

- Отложенное (on-demand)

- Отложенное (императивное)

- **Статические библиотеки** используют **раннее (статическое) связывание**. Код из библиотеки копируется в исполняемый файл на этапе сборки (link-time) или, в особых случаях, на этапе трансляции.

- **Динамические (разделяемые) библиотеки** используют **позднее (динамическое) связывание**. Код не копируется в исполняемый файл, а загружается в память во время выполнения программы.

- **Отложенное декларативное связывание** (при загрузке) соответствует **неявной загрузке** DLL/SO.

- **Отложенное императивное связывание** (on-demand) соответствует **явной загрузке** DLL/SO.

**Раннее связывание**

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**

**1. Отложенное декларативное (неявная загрузка):** Этот вид связывания используется с динамическими библиотеками.

**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 операционной системы 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):** Этот вид связывания также используется с динамическими библиотеками.

**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" (on-demand), например, для реализации плагинов.

***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** — это обычный файл, содержащий копии всех помещённых в него объектных файлов. Библиотека объектных файлов (общий термин, включающий статические и динамические библиотеки) — это файл, содержащий несколько объектных файлов, которые будут использоваться вместе на стадии сборки (связывания, линковки) программы. Статическая библиотека также хранит различные атрибуты для каждого объектного файла и символьный индекс, состоящий из названий функций, переменных и т.д., содержащихся в библиотеке, что

ускоряет процесс сборки. В Unix-подобных системах статические библиотеки обычно имеют имена вида `libname.a`, тогда как в Windows общепринятых правил именования нет.

В основе статических библиотек лежит **раннее (статическое) связывание**. В результате статического связывания **весь объектный код, содержащийся в библиотеке, внедряется в будущий исполняемый файл на этапе компоновки или трансляции**. Функция связывания состоит в компоновке программы из множества объектных модулей, разрешая обращения к данным или процедурам, определенным в других модулях, путем вставки правильных адресов. Транслятор обрабатывает только один модуль, оставляя внешние ссылки в промежуточной форме, которые разрешаются компоновщиком, соединяющим все объектные модули. Результатом работы компоновщика является загрузочный модуль, пригодный для выполнения.

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

1. Скомпилируйте исходные файлы библиотеки в объектные файлы (.o). Например, с помощью clang: `clang -c src/TestA.c -o obj/TestA.o` и `clang -c src/TestW.c -o obj/TestW.o`.
2. Используйте утилиту `ar` или `llvm-ar` для создания архива из объектных файлов. Форма команды: `ar <operation>[modifiers] <archive> [files]`.
  - Операция `r` (replace - «заменить») вставляет объектный файл в архив, заменяя существующий с тем же именем. С модификатором `s` (create) заставляет создавать библиотеку, если ее нет.
  - Например: `llvm-ar rcs libtest.lib obj/TestA.o obj/TestW.o`.
3. (Опционально, но рекомендуется для гарантии индексации) Используйте утилиту `ranlib` или `llvm-ranlib` для добавления символьного индекса к библиотеке. Например: `llvm-ranlib libtest.lib`.

```
# 1. Компиляция исходных файлов библиотеки в объектные файлы (.o)
clang -c src/TestA.c -o obj/TestA.o # Компилирует TestA.c без линковки, сохраняет в obj/TestA.o
clang -c src/TestW.c -o obj/TestW.o # Аналогично, для TestW.c

# 2. Создание статической библиотеки с помощью llvm-ar
llvm-ar rcs libtest.lib obj/TestA.o obj/TestW.o
# r - заменить существующие объектные файлы
# s - создать архив, если он не существует
# s - создать индекс (опционально, но часто полезен)
# В результате получится статическая библиотека libtest.lib


# 3. (Опционально) Генерация символьного индекса для совместимости с компоновщиками
llvm-ranlib libtest.lib
# Создаёт/обновляет символьный индекс для библиотеки libtest.lib
# Это может ускорить линковку и улучшить совместимость с некоторыми системами
```

#### Создание статической библиотеки с использованием CMake:

1. В корневой папке проекта добавьте конфигурационный файл `CMakeLists.txt`.
2. В файле `CMakeLists.txt` используйте команду `add_library` с указанием типа `STATIC` и списка исходных файлов библиотеки.
  - Пример: `add_library(test STATIC TestA.c TestW.c)`.

- Используйте команду `include_directories` для указания каталогов, используемых для поиска заголовочных файлов.

cmake

 Копировать  Редактировать

```
# Указываем минимально необходимую версию CMake
cmake_minimum_required(VERSION 3.14)

# Объявляем проект и язык программирования
project(TestLibProject LANGUAGES C)

# Указываем директорию, где находятся заголовочные файлы
include_directories(include)

# Создаём статическую библиотеку с именем 'test' из исходников TestA.c и TestW.c
add_library(test STATIC
    src/TestA.c
    src/TestW.c
)

# (Опционально) Устанавливаем стандарт языка C, если нужно
set_property(TARGET test PROPERTY C_STANDARD 11)
```

```
mkdir build && cd build      # создаём отдельную директорию для сборки
cmake ..                    # генерируем Makefile или другой build-файл
cmake --build .              # компилируем библиотеку
```

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

1. Скомпилируйте исходные файлы приложения в объектные файлы (.o). Например: `clang -c src/Test-client.c -o obj/Test-client.o`.
2. Используйте драйвер компилятора (например, clang) для компоновки объектных файлов приложения с файлом статической библиотеки.
  - Пример для Windows: `clang obj/Test-client.o libtest.lib -o bin/test.exe`.
  - **Важно:** На ОС Linux очень важен порядок входных файлов: **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.

```
# 1. Компилируем исходный файл приложения в объектный файл (.o)
clang -c src/Test-client.c -o obj/Test-client.o
# -c: компиляция без линковки
# -o: указание выходного файла

# 2. Компоуем объектный файл приложения с ранее созданной статической библиотекой
clang obj/Test-client.o libtest.lib -o bin/test.exe
# ВАЖНО: obj/Test-client.o идёт ПЕРВЫМ, затем библиотека
# Иначе линковщик не найдёт нужные символы (особенно на Linux)

# (На Linux библиотека обычно называется libtest.a, пример:)
# clang obj/Test-client.o libtest.a -o bin/test

# (Опционально) Добавьте флаг -v для вывода подробностей компоновки
# clang -v obj/Test-client.o libtest.a -o bin/test
```

## Сборка приложения с использованием статических библиотек с использованием CMake:

1. В файле CMakeLists.txt приложения (или верхнего уровня) используйте команду `target_link_libraries` для связывания исполняемого файла приложения с целевой статической библиотекой.
  - Пример: `target_link_libraries(test_client test)`. Здесь `test_client` - цель, созданная командой `add_executable`, а `test` - цель, созданная командой `add_library`.

```
# Создаём исполняемый файл 'test_client' из исходников
add_executable(test_client
    src/Test-client.c
)

# Связываем исполняемый файл с нашей статической библиотекой
target_link_libraries(test_client test)
```

### Преимущества статических библиотек:

- Позволяют сгруппировать набор часто используемых объектных файлов в единую библиотеку для использования в разных программах без необходимости перекомпиляции оригинальных исходных текстов.
- Упрощают команды для компоновки, так как вместо длинного списка объектных файлов достаточно указать только имя статической библиотеки. Компоновщик знает, как искать в ней и извлекать нужные объекты.

### Недостатки статических библиотек:

- **Дисковое пространство** тратится на хранение нескольких копий одних и тех же объектных модулей в разных исполняемых файлах, что может быть значительным.
- Если несколько программ, использующих одни и те же модули, выполняются одновременно, каждая хранит в виртуальной памяти свою отдельную копию этих модулей, **увеличивая потребление виртуальной памяти** в системе.
- Если объектный модуль статической библиотеки требует изменений (исправление ошибки, дыра в безопасности), необходимо **заново компоновать все исполняемые файлы**, использующие этот модуль. Это усугубляется тем, что системный администратор должен знать, с какими приложениями скомпонована библиотека.

## ***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.***

Разделяемая (динамическая) библиотека — это сущность, которая позволяет использовать одну копию объектного модуля несколькими программами. В операционной системе Windows такие библиотеки называются **Dynamic Link Library (DLL)** и представляют собой файлы в формате Portable Executable (PE). В операционных системах семейства Linux подобные библиотеки называются **разделяемыми объектами (shared objects)** и представляют собой файлы в формате Executable and Linkable Format (ELF).

Ключевая идея разделяемых библиотек заключается в том, что **единая копия объектного модуля разделяется между всеми программами, задействующими его**. Объектные модули из разделяемой библиотеки не копируются в компокуемый исполняемый файл. Вместо этого единая копия библиотеки загружается в память при запуске первой программы, которой требуются ее объектные модули. Если позже запускаются другие программы, использующие эту же разделяемую библиотеку, они обращаются к уже загруженной в память копии.

Для того чтобы приложение (или другая DLL) могло вызывать функции, содержащиеся в DLL, образ ее файла нужно сначала **спроецировать на адресное пространство вызывающего процесса**. Этот механизм проекции файлов в память (memory-mapped files) позволяет

резервировать область адресного пространства и передавать ему физическую память, взятую из файла на диске. Проецируемые в память файлы применяются для загрузки и выполнения исполняемых файлов и библиотек.

Если представления одного и того же файла данных создаются сразу несколькими процессами, данные сохраняют согласованность, так как они сопоставлены с единственным экземпляром каждой страницы в оперативной памяти, что равносильно проекции страниц оперативной памяти на адресные пространства нескольких процессов одновременно.

Таким образом, в основе работы разделяемых библиотек лежит механизм **отображения файлов в память (memory-mapped files)**, который позволяет нескольким процессам использовать одну и ту же физическую копию библиотеки в оперативной памяти.

### Какие способы подключения разделяемых библиотек существуют?

Существует два основных способа подключения разделяемых библиотек к приложению:

1. **Неявное связывание (implicit linking)** или отложенное декларативное связывание при загрузке. При этом компоновщик добавляет в исполняемый файл информацию о зависимостях от DLL (например, в таблицу импорта). Система при загрузке приложения автоматически загружает требуемые DLL в адресное пространство процесса.
2. **Явное связывание (explicit linking)** или отложенное императивное связывание в период выполнения. При этом способе поток приложения явно загружает DLL в свое адресное пространство во время выполнения, получает виртуальный адрес необходимой функции DLL и вызывает ее по этому адресу. Все это происходит в уже выполняющемся приложении.

### Как создать разделяемую библиотеку используя clang напрямую?

Для создания разделяемой библиотеки (DLL на Windows или SO на Linux) с использованием clang (который выступает в качестве драйвера компилятора, компоновщика и других утилит), используется флаг `-shared`.

Пример для Linux SO файла (из источника):

```
clang -shared mod1.c mod2.c mod3.c -o libtest.so
```

Пример для Windows DLL файла (из источника):

```
clang mylib.c -shared -o mylib.dll
```

Для успешной сборки проекта использующего функции из библиотеки Advapi32.dll (WinAPI для работы с сервисами), необходимо подключить эту библиотеку, например, флагом `-ladvapi32`.

При сборке приложения, использующего разделяемую библиотеку, важно соблюдать порядок входных файлов: сначала файлы приложения, затем библиотеки.

### Как создать разделяемую библиотеку используя CMake?

CMake является системой мета-сборки, которая генерирует файлы конфигурации для других систем сборки. Для создания разделяемой библиотеки с использованием CMake в файле CMakeLists.txt используется команда `add_library` с параметром `SHARED`.

Пример из источника (LibProject/src/CMakeLists.txt):

```
add_library(test SHARED Source.c)
```

(Предполагается, что Source.c содержит код библиотеки).

Затем, в файле CMakeLists.txt клиентского проекта (MainProject/src/CMakeLists.txt) или корневого проекта ({root}/CMakeLists.txt), используется команда `target_link_libraries` для связывания исполняемого файла с созданной библиотекой.

Пример из источника (MainProject/src/CMakeLists.txt):

```
add_executable(test test.c)
```

```
target_link_libraries(test PRIVATE test) # PRIVATE указывает, что зависимость нужна только для сборки этого таргета
```

Здесь `test` в `add_executable` - имя исполняемого файла, `test.c` - исходный файл приложения, `test` в `target_link_libraries` - имя библиотеки, созданной командой `add_library` в другом CMakeLists.txt. Команда `add_subdirectory()` используется для включения каталогов с подпроектами в сборку верхнего уровня.

## Преимущества и недостатки разделяемых библиотек.

Преимущества разделяемых библиотек по сравнению со статическими:

- **Экономия дискового пространства:** Объектные модули не копируются в каждый исполняемый файл, использующий библиотеку.
- **Экономия оперативной памяти:** При одновременном выполнении нескольких программ, использующих одну и ту же разделяемую библиотеку, в оперативной памяти хранится только один ее экземпляр.
- **Упрощение обновлений:** При необходимости внесения изменений в библиотеку достаточно заменить только файл самой библиотеки. Нет необходимости заново компоновать все исполняемые файлы, которые ее используют.
- **Расширение функциональности приложения:** DLL могут загружаться динамически, позволяя приложению подгружать необходимый код в зависимости от требуемых действий.
- **Возможность использования разных языков программирования:** Различные части приложения могут быть написаны на разных языках программирования, а затем скомпонованы с использованием DLL/SO.
- **Более простое управление проектом:** При разделении разработки на модули, каждый из которых реализуется в отдельной DLL, управление проектом становится проще.
- **Разделение ресурсов:** DLL могут содержать общие ресурсы (шаблоны диалоговых окон, строки, значки), которые доступны любым программам.
- **Упрощение локализации:** DLL часто применяются для локализации приложений, храня локализованные компоненты интерфейса.



- **Решение проблем, связанных с особенностями различных платформ/версий ОС:** Функциональность, доступная только в определенных версиях ОС, может быть помещена в отдельные DLL, чтобы приложение могло запускаться и в более ранних версиях (хотя без доступа к специфичной функциональности).
- **Реализация специфических возможностей:** Определенная функциональность в Windows доступна только при использовании DLL (например, некоторые виды ловушек, расширение оболочки Windows через COM).

Аспекты, связанные с управлением зависимостями (ака недостатки):

- **Необходимость наличия файла библиотеки:** Приложение не запустится, если файл разделяемой библиотеки отсутствует или не найден системой.
- **Сложность поиска библиотеки:** Система ищет библиотеки в определенной последовательности каталогов. На Linux для указания нестандартного места расположения может использоваться переменная среды `LD_LIBRARY_PATH`. На Windows также есть определенная последовательность поиска.

Понятие "DLL Hell", связанное с конфликтами версий библиотек, в предоставленных источниках явно не описано как недостаток.

## ***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?***

Неявное подключение разделяемой библиотеки, также известное как **отложенное декларативное связывание** или загрузка во время загрузки (load-time linking), представляет собой механизм, при котором зависимости исполняемого файла от динамических библиотек разрешаются операционной системой автоматически при запуске программы.

Вместо того чтобы копировать код объектных модулей из библиотеки непосредственно в исполняемый файл (как при статическом связывании), при неявном подключении исполняемый файл содержит только ссылки на нужные функции и данные, расположенные в разделяемой библиотеке. Операционная система при загрузке программы находит и загружает требуемые разделяемые библиотеки в память. Если другая программа уже использует ту же библиотеку, операционная система может просто предоставить ей доступ к уже загруженной копии, что позволяет более эффективно использовать дисковое пространство и оперативную память.

### **Алгоритмы неявного связывания с использованием clang и CMake**

Процесс неявного связывания при использовании инструментов разработки, таких как clang (или, точнее, драйвер компилятора clang, который вызывает необходимые утилиты, включая компоновщик) и системы мета-сборки CMake, обычно выглядит следующим образом:

1. **Создание разделяемой библиотеки:** Вы пишете исходный код библиотеки, определяя функции и данные, которые вы хотите сделать доступными для других программ.
2. **Сборка библиотеки:** Используя компилятор (например, clang) и компоновщик, вы компилируете исходные файлы библиотеки в объектные модули, а затем компоуете их для создания разделяемой библиотеки (на Windows это DLL-файл, на Linux — SO-файл). На этом этапе также генерируется информация, необходимая для связывания с библиотекой (например, таблица экспорта).

```
clang -target x86_64-pc-win32 -shared -I .\include\ -o .\bin\mylib.dll .\src\mylin.c
```

```
clang -target x86_64-pc-linux -fPIC -c ./source.c — Здесь мы получаем онли объектный файл
```

```
clang -target x86_64-pc-linux -shared -o some.so ./source.o
```

fPIC это то же самое, что и shared на Винде. Надо для динамической библиотеки

3. **Создание исполняемого файла:** Вы пишете исходный код программы, которая будет использовать функции из вашей разделяемой библиотеки. В этом коде вы объявляете (но не определяете) функции или данные, которые импортируются из библиотеки.
4. **Сборка исполняемого файла:** Компилятор обрабатывает исходные файлы программы. Затем компоновщик связывает объектные модули программы с разделяемой библиотекой.

```
clang -target x86_64-pc-win32 -shared -I .\include\ -L .\bin -lmylib -o .\bin\my.exe .\src\my.c
```

```
clang -target x86_64-pc-linux -o exec -L. -lsome ./main.c
```

- **На Windows:** Компоновщик связывает исполняемый файл не напрямую с DLL, а с **библиотекой импорта (.lib)**, которая была создана при сборке DLL. Библиотека импорта содержит информацию о функциях и данных, экспортируемых из DLL. Компоновщик использует эту информацию для создания **таблицы импорта** в исполняемом файле.
  - **На Linux:** Компоновщик связывает исполняемый файл с разделяемой библиотекой (.so). Исполняемый файл содержит ссылки на символы в разделяемой библиотеке.
  - Источник отмечает, что при сборке очень важен порядок передачи файлов драйверу компилятора: сначала файлы приложения, затем библиотеки.
5. **Запуск программы:** Когда пользователь запускает исполняемый файл, загрузчик операционной системы читает его и находит ссылки на символы в разделяемых библиотеках, указанные в таблице импорта (на Windows) или в списке динамических зависимостей (на Linux).
  6. **Поиск и загрузка библиотеки:** Загрузчик ищет файлы необходимых разделяемых библиотек в предопределенных системных каталогах или каталогах, указанных в переменных окружения (например, PATH на Windows или LD\_LIBRARY\_PATH на Linux).
  7. **Разрешение зависимостей:** Найденные библиотеки загружаются в адресное пространство процесса, и загрузчик разрешает все ссылки в исполняемом файле, указывая на фактические адреса функций и данных в загруженных библиотеках. Если нужная библиотека не найдена или не может быть загружена, загрузчик выдает ошибку, и программа не запускается.

## Использование CMake для неявного связывания:

CMake упрощает этот процесс, абстрагируя специфику платформы. В файлах CMakeLists.txt вы используете команды:

- `add_library(<имя_библиотеки> SHARED <список_исходных_файлов>)` для определения сборки разделяемой библиотеки.
- `add_executable(<имя_исполняемого_файла> <список_исходных_файлов>)` для определения сборки исполняемого файла.
- `target_link_libraries(<имя_исполняемого_файла> <имя_библиотеки>)` для указания зависимости исполняемого файла от разделяемой библиотеки. CMake автоматически определит, как правильно связать программу с библиотекой на текущей платформе.
- При необходимости могут использоваться команды `include_directories` для указания путей к заголовочным файлам.

В основе неявного подключения лежит **позднее связывание (run-time linking)**, в частности, **связывание при загрузке (load-time linking)**. Это отличается от раннего связывания, которое происходит во время компиляции или сборки (link-time).

**Библиотека импорта (Import Library)** — это специальный файл (обычно с расширением .lib на Windows), который создается компоновщиком при сборке динамически подключаемой библиотеки (DLL). Этот файл содержит информацию, необходимую для связывания исполняемых модулей (EXE или других DLL) с экспортируемыми функциями и данными из соответствующей DLL. Библиотека импорта содержит ссылки на все экспортируемые из динамической библиотеки имена. При сборке исполняемого файла компоновщик использует эту библиотеку импорта для создания таблицы импорта в исполняемом файле.

**Раздел экспорта (Export Section)**, или таблица экспортируемых идентификаторов (Export Table), — это часть структуры файла динамически подключаемой библиотеки (DLL на Windows, SO на Linux). Этот раздел содержит список всех функций, переменных и классов, которые библиотека делает доступными для использования другими программами. Для каждого экспортируемого элемента в таблице экспорта хранится информация, такая как его имя и/или порядковый номер, а также его относительный виртуальный адрес (RVA) внутри модуля DLL.

Существуют два основных способа экспорта функций из динамической библиотеки: **по имени (by name)** и **по порядковому номеру (by ordinal)**.

### 1. Экспорт по имени:

- **На Windows:** Самый распространенный способ — использовать модификатор `__declspec(dllexport)` перед объявлением функции или переменной в исходном коде библиотеки. Компилятор и компоновщик обрабатывают этот модификатор и включают соответствующую информацию в таблицу экспорта DLL и библиотеку импорта. Если библиотека написана на языке, который выполняет преобразование имен (name mangling, например, C++), а вы хотите сохранить исходное имя функции, понятное человеку (или для совместимости с C), вы можете использовать модификатор `extern "C"` вместе с `__declspec(dllexport)`.

- **На Linux:** Символы (функции, переменные) по умолчанию экспортируются из разделяемых библиотек, если они не были объявлены как статические.
2. **Экспорт по порядковому номеру:**
- **На Windows:** Этот способ требует использования **.DEF файла (файла определений модуля)**. В .def файле вы можете явно указать имена экспортируемых элементов и присвоить им порядковые номера с помощью атрибута @ordinal. .DEF файл дает полный контроль над таблицей экспорта, позволяя, например, экспортировать только по порядковому номеру без сохранения имен (с атрибутом NONAME). Однако Microsoft настоятельно не рекомендует использовать порядковые номера, за исключением особых случаев.

## ***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?***

Явная загрузка, также известная как отложенное императивное связывание, — это способ подключения требуемой DLL (в Windows) или SO-файла (в Linux) в период выполнения приложения. При этом способе поток приложения явно загружает библиотеку в свое адресное пространство, получает виртуальный адрес необходимой функции библиотеки и вызывает ее по этому адресу. Изящество такого подхода заключается в том, что все происходит в уже выполняющемся приложении.

Общий алгоритм загрузки и очистки библиотеки из памяти включает вызов функции загрузки (dlopen в POSIX/Linux, LoadLibrary/LoadLibraryEx в WinAPI/Windows), получение адреса нужного символа (dlsym в POSIX/Linux, GetProcAddress в WinAPI/Windows), вызов этой функции по полученному адресу, и, наконец, выгрузку библиотеки (dlclose в POSIX/Linux, FreeLibrary в WinAPI/Windows). Функции dlopen/LoadLibrary могут быть вызваны несколько раз для одной и той же библиотеки; загрузка выполняется только при первом вызове, а последующие вызовы возвращают одно и то же значение. API хранит счетчик ссылок для каждого дескриптора (HANDLE в Windows, дескриптор из dlopen в POSIX). Счетчик инкрементируется с каждым вызовом dlopen/LoadLibrary и декрементируется при вызове dlclose/FreeLibrary. Библиотека выгружается из памяти только тогда, когда счетчик ссылок становится равен 0.

В отличие от неявного подключения, при явном связывании:

- Не требуется подключать заголовочный файл библиотеки.
- Появляются вызовы функций API для работы с библиотеками (LoadLibrary/GetProcAddress/FreeLibrary для Windows, dlopen/dlsym/dlclose для Linux).
- При сборке приложения больше не требуется указывать использование библиотеки.

Вот тут **бернацкий долбоёб**, так как при явном способе отличий в clang/cmake от обычной сборки нет никаких, всё подключение идёт через api (loadlibrary, getprocaddress, нувыпоняли). Но я оставляю это

Явное связывание НЕ требует линковки с `.so` / `.dll` на этапе компиляции.

### Сборка с Clang (Windows/Linux)

`clang -o myprog main.c -ldl` # на Linux `-ldl` нужно для `dlopen`

- На Windows флаг `-ldl` не нужен.
- Упрощённо: просто собираешь обычный исполняемый файл, как будто нет никаких библиотек.

### Сборка с CMake

`add_executable(myprog main.c)`

`target_link_libraries(myprog PRIVATE dl)` # только для Linux

- На Windows можно вообще **не подключать никаких библиотек**
- Функции `LoadLibrary` и `GetProcAddress` идут из `kernel32.dll`, которая линкуется по умолчанию

В основе явного подключения лежит отложенное императивное связывание. Это один из видов позднего (run-time) связывания.

При явной сборке (с использованием `LoadLibrary/dlopen`) таблица импорта приложения не содержит информации о явно загружаемых библиотеках, так как компоновщик ничего о них не знает на этапе сборки.

Name Mangling (или name decoration) — это процесс, при котором компилятор преобразует имена функций и переменных из исходного кода в понятные для себя имена при создании символов для таблицы экспорта/импорта. Компилятор C++ часто выполняет name mangling, тогда как компилятор C обычно этого не делает, сохраняя имена функций в исходном виде. Этот процесс может затруднить понимание имен в таблице экспорта.

Чтобы сохранить имена функций в понятном для человека виде (без mangling), можно использовать `.def`-файл (файл определений модуля). Этот файл позволяет указать конечное имя для функции в таблице экспорта. Альтернативным способом сохранения имени функции является использование модификатора `extern "C"`.

Явная загрузка позволяет приложению, определив, какие действия от него требуются, подгружать нужный код динамически. Это может быть использовано для расширения функциональности приложения за счет DLL от других компаний, а также для решения проблем, связанных с особенностями различных платформ, позволяя запускать программу даже в более ранних версиях Windows, если специфичные функции вынесены в DLL и загружаются явно (хотя воспользоваться ими все равно не удастся).

## ***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 выгрузки разделяемой библиотеки в память включает использование специальных функций, которые управляют счетчиком ссылок на библиотеку.

Функция **dlopen** (для POSIX/Linux) или **LoadLibrary** (для Windows) может быть вызвана несколько раз для одной и той же библиотеки. Загрузка самой библиотеки будет выполнена только при первом вызове, а все последующие вызовы будут возвращать одно и то же значение дескриптора (HANDLE). Программный интерфейс хранит счетчик ссылок для каждого дескриптора: с каждым вызовом dlopen/LoadLibrary он инкрементируется.

Функция **dlclose** (для POSIX/Linux) или **FreeLibrary** (для Windows) декрементирует счетчик ссылок. Библиотека выгружается из памяти только в том случае, **если счетчик ссылок становится равен 0**.

В **Windows DLL** может содержать одну **функцию входа/выхода**. Система вызывает ее в определенных ситуациях (например, при загрузке DLL в адресное пространство процесса или при выгрузке) исключительно в информационных целях. Обычно она используется DLL для инициализации и очистки ресурсов в конкретных процессах или потоках. Если DLL не нуждается в таких уведомлениях, эту функцию можно не реализовывать (например, для DLL, содержащей только ресурсы).

В **Linux SO** такой функции входа/выхода (аналогичной DllMain) быть не может. Однако вместо этого можно определить одну или несколько функций, которые будут автоматически вызываться **при загрузке и выгрузке** разделяемой библиотеки.

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

Перехват API функций (API hooking), при котором вызовы функций из библиотеки API перенаправляются на пользовательские функции, позволяет модифицировать поведение программы без изменения ее исходного кода. **DLL injection** является одним из основных методов перехвата API функций, позволяющим обойти многие ограничения системы и изменять ее поведение.

Добросовестное использование API hooking, в том числе посредством DLL injection, включает отладку, мониторинг, программное обеспечение для обеспечения безопасности (антивирусы), мониторинг производительности и расширение функциональных возможностей.

Недобросовестное использование может включать руткиты, кражу данных, несанкционированный доступ и распространение вредоносного ПО. Использование таких техник требует соблюдения законодательства, информированного согласия (где применимо), прозрачности и следования рекомендациям по безопасности.

Внедрение DLL в Windows с помощью **удаленных потоков** предполагает вызов функции LoadLibrary потоком целевого процесса для загрузки нужной DLL. Поскольку управление потоками чужого процесса затруднено, необходимо создать в нем свой поток. Функция Windows CreateRemoteThread облегчает эту задачу.

Последовательность операций для внедрения DLL с помощью удаленных потоков:

1. **Выделить блок памяти** в адресном пространстве удаленного процесса с помощью функции VirtualAllocEx.
2. **Скопировать строку с полным именем файла DLL** в выделенный блок памяти в удаленном процессе, вызвав WriteProcessMemory.
3. **Получить истинный адрес функции LoadLibraryA или LoadLibraryW** внутри Kernel32.dll в текущем процессе, используя GetProcAddress. Адрес LoadLibrary одинаков во всех процессах, использующих одну и ту же версию Kernel32.dll.
4. **Создать поток в удаленном процессе** с помощью CreateRemoteThread, который вызовет соответствующую функцию LoadLibrary, передав ей адрес блока памяти, выделенного на шаге 1. На этом этапе DLL будет внедрена в удаленный процесс, и ее функция DllMain получит уведомление DLL\_PROCESS\_ATTACH и может начать выполнять нужный код.
5. (Продолжение шага 4) Когда DllMain вернет управление, удаленный поток выйдет из LoadLibrary и вернется в функцию BaseThreadStart, которая, в свою очередь, вызовет ExitThread и завершит этот поток.
6. **Освободить блок памяти**, выделенный на шаге 1, вызвав VirtualFreeEx.
7. **Определить истинный адрес функции FreeLibrary** внутри Kernel32.dll с помощью GetProcAddress.
8. **Создать в удаленном процессе еще один поток** с использованием CreateRemoteThread, который вызовет FreeLibrary с передачей HINSTANCE внедренной DLL для ее выгрузки из памяти.

Для внедрения разделяемой библиотеки (SO) в Linux можно использовать переменную среды **LD\_PRELOAD**. Эта переменная позволяет присвоить строку с именами разделяемых библиотек, которые следует загрузить раньше других библиотек (имена разделяются двоеточиями).

Поскольку библиотеки, указанные в LD\_PRELOAD, загружаются первыми, их функции, запрашиваемые программой, будут использоваться автоматически, **переопределяя любые одноименные символы**, которые в противном случае пришлось бы искать динамическим компоновщиком.

Из соображений безопасности программы, устанавливающие пользовательские и групповые идентификаторы (suid/sgid), игнорируют переменную LD\_PRELOAD.

Пример использования LD\_PRELOAD из источника: LD\_LIBRARY\_PATH=. ./prog (это пример использования LD\_LIBRARY\_PATH для указания нестандартного пути к библиотеке, а не LD\_PRELOAD для переопределения функций, но принцип использования переменной среды для влияния на динамический компоновщик аналогичен). Пример для LD\_PRELOAD в источнике дан так: LD\_PRELOAD="libmod2.so libmod1.so" перед вызовом исполняемого файла.

**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.**

Системный реестр Windows — это централизованная иерархическая база данных, хранящая информацию о параметрах конфигурации операционной системы и установленных приложений. Он обеспечивает единообразное централизованное хранение всей конфигурационной информации, связанной с системой и ее приложениями.

До появления семейства ОС Windows NT, приложения, работавшие в среде Windows 3.x и DOS, обычно использовали множество разрозненных файлов инициализации (INI-файлов), таких как Win.INI, Reg.DAT, System.INI и другие. Эти файлы содержали данные о конфигурации системы, связях между приложениями и расширениями файлов, аппаратной конфигурации, а также параметры отдельных приложений.

Основной причиной появления реестра стало желание заменить множество разрозненных файлов инициализации единой системной базой данных. 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.

Таким образом, 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 носителе** (HDD, SSD). Тип создаваемого раздела по умолчанию — энергонезависимый (REG\_OPTION\_NON\_VOLATILE).

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.

Но бернацкий долбоёб, так что вот все:

**HKEY\_USERS** - Содержит параметры всех пользователей, у которых есть профиль на компьютере. Каждая запись — SID пользователя

**HKEY\_LOCAL\_MACHINE** - Глобальные настройки для всей системы, вне зависимости от пользователя.

**HKEY\_CURRENT\_USER** - Настройки текущего пользователя

**HKEY\_CURRENT\_CONFIG** - Текущая конфигурация оборудования (например, видеоадаптер, принтеры)

**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 следующих 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 также типы данных, полезные только в режиме ядра, такие как REG\_RESOURCE\_LIST, REG\_FULL\_RESOURCE\_DESCRIPTOR, REG\_RESOURCE\_REQUIREMENTS\_LIST.

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.

**Назовите пять основных ульев и опишите их назначение.**

Пять основных ульев, отображаемых RegEdit:

1. **HKEY\_LOCAL\_MACHINE (HKLM)** – этот улей хранит **информацию обо всем компьютере, которая не относится к какому-либо конкретному пользователю**. 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).
2. **HKEY\_USERS** – этот улей хранит **всю информацию о каждом пользователе**, который когда-либо входил в локальную систему. Каждый пользователь представлен своим SID (идентификатором безопасности). Подразделы SID содержат различные настройки для каждого пользователя (рабочий стол, консоль, переменные среды и т.д.).
3. **HKEY\_CURRENT\_USER (HKCU)** – этот улей **является ссылкой на информацию из улья HKEY\_USERS для текущего пользователя**, который запустил RegEdit. Данные из этого улья сохраняются в скрытом файле NtUser.dat в домашнем каталоге пользователя.
4. **HKEY\_CLASSES\_ROOT (HKCR)** – этот улей состоит из **комбинации разделов HKEY\_LOCAL\_MACHINE\Software\Classes и HKEY\_CURRENT\_USER\Software\Classes**. Настройки HKEY\_CURRENT\_USER переопределяют настройки HKEY\_LOCAL\_MACHINE в случае конфликта. **HKEY\_CLASSES\_ROOT** содержит информацию о:

- **Данных оболочки Explorer:** типах файлов и ассоциациях, расширениях оболочки.
  - **Информации, связанной с объектной моделью компонента (COM).** Здесь регистрируются COM-компоненты.
5. **HKEY\_CURRENT\_CONFIG** – этот улей содержит информацию о текущей конфигурации оборудования. (Этот улей не описан подробно в предоставленных источниках, но является одним из стандартных пяти, отображаемых RegEdit, и его назначение хорошо известно. Учитывая инструкции использовать только предоставленные источники, я не могу предоставить детали извне источников).

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.

- RegSetValue: 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**.
  - RegDeleteValue: 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 (COM), разработанная фирмой Microsoft, является **моделью для проектирования и создания компонентных объектов**. Это **независимая от платформы, распределенная, объектно-ориентированная система для создания бинарных программных компонентов, которые могут взаимодействовать между собой**. Модель определяет набор технических приемов, которые используются разработчиком для создания **независимых от языка программных модулей**, соблюдающих определенный двоичный стандарт. COM поддерживается корпорацией Microsoft во всех ее средах Windows, а также может поддерживаться в других операционных средах, таких как Macintosh и UNIX.

Два важных свойства COM:

- **Двоичный стандарт (или независимость от ЯП).** Это свойство обеспечивает средства для взаимодействия объектов и компонентов, разработанных на разных языках программирования и работающих в различных операционных системах, без необходимости изменения двоичного кода. Единственное языковое требование COM – это то, что код генерируется на языке, который может создавать структуры указателей и вызывать функции через указатели.
- **Независимость от местоположения (Location Transparency).** Это свойство означает, что клиент (пользователь компонента) не обязательно должен знать, где физически находится компонент. Клиентское приложение использует одинаковые сервисы COM для создания экземпляра и использования компонента, независимо от того, находится ли он в адресном пространстве клиента (DLL-файл), в пространстве другой задачи на том же компьютере (EXE-файл), или на удаленном компьютере (распределенный объект).

COM называют двоичным стандартом, потому что он определяет стандарт, который применяется после того, как программа переведена в **двоичный машинный код**. Это позволяет создавать программные компоненты, которые могут применяться **без использования специфических языков, средств и систем программирования**, а только с помощью двоичных компонентов (например, DLL- или EXE- файлов). Это означает, что компоненты могут быть написаны на разных языках программирования, но при взаимодействии они следуют единому двоичному контракту.

COM-компонент (COM-объект) является **программным модулем**, представляющим собой некоторую **сущность, имеющую состояние и методы доступа, позволяющие изменять это состояние**. COM-объект можно создать вызовом специальных функций, но его нельзя уничтожить напрямую; вместо этого используется **механизм самоуничтожения, основанный на подсчете ссылок**.

В понимании COM, интерфейс – это **контракт, состоящий из списка связанных прототипов функций**, чье назначение определено, но реализация отсутствует. Эти прототипы функций, называемые методами, эквивалентны абстрактным базовым классам C++. Определение интерфейса описывает методы, типы их возвращаемого значения, число и типы параметров, а также их назначение. **Напрямую с интерфейсом не ассоциировано никакой реализации.**

Интерфейсы бывают двух типов:

**Стандартные:** Имеют предопределенные GUID-идентификаторы. Важнейшим является IUnknown.

**Произвольные (Custom):** Определяются разработчиками. Все произвольные интерфейсы должны прямо или косвенно наследоваться от IUnknown.

**COM-интерфейс** — это договор (контракт) между объектом и его клиентом. Он определяет набор функций (методов), которые объект предоставляет внешнему миру.

**Каждый** COM-интерфейс: наследуется от базового интерфейса **'IUnknown'**, определяет только методы, никаких данных, представлен как таблица функций (**vtable**)

Важнейшим стандартным интерфейсом является **интерфейс IUnknown**. Все остальные интерфейсы являются производными от IUnknown. Еще один стандартный интерфейс – **IClassFactory**, который отвечает за управление жизненным циклом компонентов путем реализации паттерна "фабрика классов".

**Что такое GUID? CLSID? IID? АААААААА ИДИТЕ НАХУЙ ВСЕ**

- **GUID (Globally Unique Identifier)** – это **глобально уникальный идентификатор**. Он имеет размер **128 бит** и уникален в пространстве и времени.
- **CLSID (Class Identifier)** – это **бинарный идентификатор** для CoClass (класса в COM). Это тип данных GUID, который позволяет точно указать, какой именно объект требуется.
- **IID (Interface Identifier)** – это **идентификатор интерфейса**. Это также тип данных GUID, используемый для идентификации COM-интерфейсов.

***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» COM-сервера. Что такое IDL?***

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

Контейнеры с расположенными в них компонентами называют **COM-серверами**. Серверы in-process реализуются в динамической библиотеке (DLL), а серверы out-of-process реализуются в исполняемом файле (EXE).

Приложения, использующие COM-компоненты (вызывающие функции интерфейсов, реализованных COM-компонентами), называют **COM-клиентами**.

**Назовите типы COM-серверов.** В зависимости от типа контейнера (DLL-файл или EXE-файл) и места его расположения (локальное или удаленное) различают несколько типов серверов:

1. **INPROC** (DLL, локальный). Серверы in-process реализуются в динамической библиотеке (DLL).
2. **LOCAL** (EXE, локальный). Серверы out-of-process реализуются в исполняемом файле (EXE).
3. **REMOTE** (EXE, удаленный). Серверы out-of-process могут размещаться либо на локальном компьютере, либо на удаленном компьютере. COM предоставляет механизм, который позволяет серверу in-process (DLL) запускаться в

суррогатном процессе EXE, чтобы получить преимущество выполнения процесса на удаленном компьютере.

COM-серверы в зависимости от количества реализуемых ими компонентов подразделяются на «однокомпонентные» и «многокомпонентные».

- 1) Если в контейнере расположен только один компонент, то сервер – **«однокомпонентный»**.
- 2) Если два и более – **«многокомпонентный»**.

При работе с COM-компонентом клиент должен «знать» только:

- 1) **GUID-идентификатор** этого компонента (**CLSID**).
- 2) **GUID идентификаторы (IID)** интерфейсов.
- 3) Тип сервера (например, INPROC, LOCAL, REMOTE).
- 4) Структуры (сигнатуры соответствующих методов) произвольных интерфейсов компонента, которые он предполагает применять.

**Алгоритм создания «однокомпонентного» COM-сервера** включает следующие шаги:

1. **Описать COM-интерфейс.** Этот интерфейс обязан наследоваться хотя бы от IUnknown8. Есть два пути: с помощью языка IDL и компилятора MIDL, либо напрямую на языке C/C++8.

2. **Создать COM-компонент.** Это делается путем написания класса, который реализует ранее описанный COM-интерфейс.

3. **Реализовать фабрику классов.** Это осуществляется путем реализации стандартного COM-интерфейса IClassFactory. Фабрика классов предназначена для удобного управления жизненным циклом COM-компонентов.

4. **Реализовать набор из 5 обязательных функций DLL.** Эти функции обеспечивают работу одного или нескольких COM-компонентов и обязательно экспортируются из DLL. К ним относятся:

- **DllCanUnloadNow:** вызывается OLE32.DLL для определения возможности выгрузки COM-сервера.
- **DllGetClassObject:** первая функция компонента, вызываемая OLE32.DLL; проверяет идентификатор компонента, создает фабрику классов и возвращает указатель на IClassFactory.
- **DllInstall:** вызывается утилитой regsvr32 для выполнения дополнительных действий при регистрации/удалении компонентов.
- **DllRegisterServer:** вызывается утилитой regsvr32 для регистрации компонентов сервера в реестре.
- **DllUnregisterServer:** вызывается утилитой regsvr32 для удаления информации о компонентах сервера из реестра.

5. **Скомпилировать COM-сервер.**

**6.Зарегистрировать COM-сервер** с использованием утилиты regsvr32. Эта утилита просто вызывает некоторые экспортируемые функции из DLL (например, DllRegisterServer). Регистрация необходима для реализации свойства «Независимость от местоположения», так как информация о каждом COM-компоненте должна быть зарегистрирована в Windows-реестре.

**7.Разработать COM-клиент.** Работа клиента должна начинаться с инициализации библиотеки OLE32 (CoInitializeEx), затем для создания экземпляра компонента вызывается CoCreateInstance (или CoGetObject), для получения указателей на другие интерфейсы используется QueryInterface, а работа завершается освобождением библиотеки OLE32 (CoUninitialize)

**IDL (MS Interface Definition Language)** – это **язык описания интерфейсов**, созданный Microsoft, который позволяет программе или объекту, написанному на одном языке, взаимодействовать с другой программой, написанной на неизвестном ему языке. При создании COM-объектов на C++ использование IDL является стандартной практикой, но не обязательной. IDL файлы содержат описание COM-компонентов и COM-интерфейсов не зависящим от языка способом.

***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.***

Component Object Model (COM) — объектная модель компонентов от Microsoft, предназначенная для проектирования и создания модулей, независимых от языка программирования. COM реализована во всех версиях Windows, а в других системах (например, UNIX или macOS) может поддерживаться сторонними средствами. Модель развивалась как продолжение подходов к модульной разработке на C++.

Одна из ключевых особенностей COM — это предоставление общего двоичного стандарта для компонентов. Благодаря ему модули, созданные на разных языках программирования и под разные операционные системы, могут взаимодействовать без перекомпиляции или изменения кода. Это особенно важно для повторного использования программного обеспечения в разных средах.

Компоненты можно распространять в виде двоичных файлов (например, DLL или EXE), и они будут работать независимо от языка или системы разработки.

Другое важное свойство COM известно под названием независимости от местоположения. Независимость от местоположения означает, что пользователь компонента, клиент, не обязательно должен знать, где находится определенный компонент



Компонент может находиться непосредственно в адресном пространстве задачи клиента (DLL-файл), в пространстве другой задачи на том же компьютере (EXE-файл) или на компьютере, расположенном за сотни миль (распределенный объект).

Чтобы понять COM (и, следовательно, все технологии, основанные на COM), важно понимать, что это не объектно-ориентированный язык, а стандарт

Ключевая идея – разделение интерфейса и реализации. Клиент работает только с интерфейсами, не зная деталей реализации объекта.

Единственным языковым требованием COM является то, что код генерируется на языке, который может создавать структуры указателей и, явно или неявно, вызывать функции с помощью указателей

### **Что такое COM-интерфейс?**

COM-интерфейс — это договор (контракт) между объектом и его клиентом. Он определяет набор функций (методов), которые объект предоставляет внешнему миру.

Каждый COM-интерфейс: наследуется от базового интерфейса 'IUnknown', определяет только методы, никаких данных, представлен как таблица функций (vtable)

### **Какие бывают интерфейсы?**

Стандартные: Имеют предопределенные GUID-идентификаторы. Важнейшим является IUnknown.

Произвольные (Custom): Определяются разработчиками. Все произвольные интерфейсы должны прямо или косвенно наследоваться от IUnknown.

### **Поясните функции интерфейса IUnknown.**

**IUnknown** – это базовый интерфейс, от которого должны наследоваться все COM-интерфейсы. Он предоставляет три основные функции (метода) для управления жизненным циклом объекта и навигации по интерфейсам:

**QueryInterface**(REFIID iid, void ppvObject): Позволяет запросить у объекта указатель на другой интерфейс, поддерживаемый этим объектом, по его IID. Если интерфейс поддерживается, метод увеличивает счетчик ссылок и возвращает указатель.

**AddRef**(): Увеличивает внутренний счетчик ссылок на интерфейс на 1. Вызывается каждый раз, когда клиент получает новый указатель на интерфейс.

**Release**(): Уменьшает счетчик ссылок на интерфейс на 1. Вызывается, когда клиент прекращает использование указателя на интерфейс. Когда счетчик достигает нуля, объект может быть уничтожен.

### **Объясните «счётчик ссылок на интерфейс». Когда увеличивается/уменьшается?**

«Счётчик ссылок на интерфейс» – это внутренний счетчик, который ведет каждый COM-объект для отслеживания того, сколько клиентов в данный момент используют указатели на его интерфейсы. Это основной механизм управления временем жизни объекта в COM.

Счетчик увеличивается (на 1) при вызове метода `AddRef()`. Это происходит, когда клиент успешно получает указатель на интерфейс (например, через `CoCreateInstance` или `QueryInterface`).

Счетчик уменьшается (на 1) при вызове метода `Release()`. Это происходит, когда клиент больше не нуждается в указателе на интерфейс и освобождает его.

Когда счетчик достигает нуля, объект понимает, что на него больше нет ссылок, и может безопасно самоуничтожиться.

### **Объясните «счётчик ссылок на компонент». Когда увеличивается/уменьшается?**

Этот счетчик отслеживает общее количество активных экземпляров всех компонентов, предоставляемых данным сервером.

Он увеличивается, когда создается новый экземпляр компонента (обычно в конструкторе компонента, который вызывается через `IClassFactory::CreateInstance`).

Он уменьшается, когда экземпляр компонента уничтожается (обычно в деструкторе компонента).

Этот счетчик используется сервером (особенно DLL) для определения, можно ли его выгрузить из памяти (например, в функции `DllCanUnloadNow`). Он отличен от «счетчика ссылок на интерфейс», который ведется каждым объектом индивидуально.

### **Какие соглашения о вызове и возврате должны соблюдаться для COM-функций?**

Соглашение о вызове: Все методы COM-интерфейсов должны использовать соглашение `stdcall`.

Тип возвращаемого значения: Почти все методы должны возвращать значение типа `HRESULT` для индикации успеха или кода ошибки.

**Исключениями** являются методы `IUnknown::AddRef` и `IUnknown::Release`, которые возвращают `ULONG` (текущее значение счетчика ссылок).

### **Опишите структуру HRESULT.**

**HRESULT** – это 32-битное целое число, используемое для возврата кодов состояния в COM. Его структура:

Бит 31 (**Severity**): 0 для успеха (`SUCCEEDED`), 1 для ошибки (`FAILED`). (Слайд 50 уточняет: биты 31-30 кодируют степень серьезности: 00=Success, 01=Informational, 10=Warning, 11=Error).

Бит 29 (**Customer**): 0 для кодов Microsoft/системных, 1 для пользовательских кодов.

Бит 28 (**Reserved**): Зарезервирован.

Биты 16-27 (**Facility**): Код, идентифицирующий источник ошибки (например, FACILITY\_WIN32, FACILITY\_STORAGE, FACILITY\_RPC, FACILITY\_ITF и т.д.).

Биты 0-15 (**Code**): Конкретный код состояния или ошибки в рамках указанного Facility.

Для проверки HRESULT используются макросы **SUCCEEDED()** и **FAILED()**.

### **Что должен знать COM-клиент чтобы взаимодействовать с COM-сервером?**

CLSID компонента, экземпляр которого он хочет создать.

IID интерфейсов, которые он собирается использовать.

Тип сервера (in-proc, local, remote - это влияет на контекст создания).

Структуры (сигнатуры) методов используемых интерфейсов (т.е., какие параметры они принимают и что возвращают).

Что такое regsvr32? Принцип работы?

regsvr32.exe – это стандартная утилита Windows для регистрации и отмены регистрации COM-серверов (в основном, in-process DLL) в системном реестре.

Принцип работы:

При вызове regsvr32 <имя\_dll>: утилита загружает указанную DLL в память и вызывает ее экспортируемую функцию DllRegisterServer(). Эта функция должна создать в реестре все необходимые записи (CLSID, ProgID, путь к серверу, модель потоков и т.д.).

При вызове regsvr32 /u <имя\_dll>: утилита загружает DLL и вызывает ее экспортируемую функцию DllUnregisterServer(). Эта функция должна удалить из реестра все записи, созданные DllRegisterServer().

Регистрация необходима для реализации свойства «независимости от местоположения», так как она позволяет системной библиотеке COM (OLE32.dll) находить и загружать сервер по его CLSID.

Где найти информацию о компоненте в реестре?

Информация о зарегистрированных COM-компонентах хранится в системном реестре Windows, в основном, в разделе HKEY\_CLASSES\_ROOT\CLSID.

Каждый компонент имеет свой подраздел, имя которого соответствует его CLSID в формате {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}.

Внутри этого подраздела находятся другие подразделы, описывающие компонент, например:

InprocServer32: Для DLL-серверов, содержит путь к файлу DLL и модель потоков (ThreadingModel).

LocalServer32: Для EXE-серверов, содержит путь к файлу EXE.

ProgID: Человекочитаемый идентификатор.

***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.***

В COM присутствует понятие класса, который носит название CoClass. COM требует, чтобы каждый класс имел собственную фабрику классов. Фабрика классов — это сущность, предназначенная для управления жизненным циклом компонентов путем реализации паттерна «фабрика классов». Использование фабрики классов упрощает отслеживание того, какие объекты были созданы и когда их можно выгружать из памяти. Фабрика классов не является обязательной к реализации для разрабатываемых компонентов, но с ней становится проще управлять их жизненным циклом.

Интерфейс IClassFactory предоставляет следующие методы:

- CreateInstance – метод, предназначенный для создания экземпляра COM-компонента. Увеличение счетчика экземпляров компонента происходит в конструкторе COM-компонента, который вызывается этим методом.
- LockServer – метод для увеличения счетчика блокировки COM-сервера. Блокировка COM-сервера предназначена для гарантии того, что он не будет закрыт раньше времени (DLL не будет выгружена).

Фабрика классов помогает отслеживать жизненный цикл COM-компонентов. Для этого на сервере существует понятие «счетчик экземпляров компонент». Этот счетчик 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.

Увеличение счетчика экземпляров компонента происходит в конструкторе COM-компонента (который вызывается методом CreateInstance интерфейса IClassFactory), а уменьшается в деструкторе.

Экземпляр COM-компоненты не может быть выгружен, пока счетчик ссылок на интерфейсы не равен нулю. Механизм самоуничтожения, основанный на подсчете ссылок (AddRef, Release интерфейса IUnknown), используется для уничтожения COM-объектов, так как напрямую их уничтожить невозможно.

Жизненный цикл COM-сервера описывается следующими условиями:

- COM-сервер не может быть выгружен, пока счетчик экземпляров компонент не равен нулю. Экземпляр COM-компоненты обычно уникален в рамках одного процесса (по сути, является Singleton на уровне процесса).

- Экземпляр COM-компоненты **не может быть выгружен, пока счетчик ссылок на интерфейсы не равен нулю.**
- COM-сервер **не может быть выгружен, пока счетчик блокировок (используемый методом LockServer интерфейса IClassFactory) не равен нулю.**

Таким образом, COM-сервер остается загруженным до тех пор, пока существует хотя бы один активный экземпляр компонента, хотя бы одна ссылка на интерфейс этого компонента или пока сервер явно заблокирован через LockServer.

### **Объясните «счётчик ссылок на компонент». Когда увеличивается/уменьшается?**

Этот счетчик отслеживает общее количество активных экземпляров всех компонентов, предоставляемых данным сервером.

Он увеличивается, когда создается новый экземпляр компонента (обычно в конструкторе компонента, который вызывается через IClassFactory::CreateInstance).

Он уменьшается, когда экземпляр компонента уничтожается (обычно в деструкторе компонента).

Этот счетчик используется сервером (особенно DLL) для определения, можно ли его выгрузить из памяти (например, в функции DllCanUnloadNow). Он отличен от «счетчика ссылок на интерфейс», который ведется каждым объектом индивидуально.

### **Перечислите и объясните 5 COM-функций экспортируемых из DLL.**

COM DLL-сервер должен экспортировать несколько стандартных функций для взаимодействия с COM-инфраструктурой:

DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID ppv): Вызывается OLE32.DLL для получения объекта фабрики классов для указанного rclsid. Возвращает указатель на запрошенный интерфейс фабрики (riid, обычно IID\_IClassFactory). Это основная точка входа для создания компонентов из DLL.

DllCanUnloadNow(): Вызывается OLE32.DLL, чтобы спросить у DLL, можно ли ее безопасно выгрузить из памяти. DLL должна вернуть S\_OK (можно), только если нет активных объектов и блокировок сервера (счетчики экземпляров и блокировок равны нулю), иначе S\_FALSE (нельзя).

DllRegisterServer(): Вызывается утилитой regsvr32 (без /u). Функция должна записать в реестр всю информацию о компонентах, реализованных в DLL.

DllUnregisterServer(): Вызывается утилитой regsvr32 /u. Функция должна удалить из реестра всю информацию о компонентах DLL.

DllInstall(BOOL bInstall, PCWSTR pszCmdLine): (Менее обязательная, но часто присутствует) Вызывается regsvr32 /i для выполнения дополнительных действий по установке/удалению, не связанных напрямую со стандартной COM-регистрацией.

Замечание: DllMain тоже экспортируется, но это стандартная точка входа для любой Windows DLL, а не специфичная функция COM-инфраструктуры.

### **Опишите жизненный цикл COM-сервера в целом.**

Жизненный цикл COM-сервера управляется операционной системой Windows через библиотеку OLE32.DLL. Сервер представляет собой контейнер для одного или нескольких COM-компонентов и может быть реализован как динамически подключаемая библиотека (DLL) или исполняемый файл (EXE).

Основные моменты жизненного цикла COM-сервера:

**1. Загрузка и инициализация:** Клиентское приложение инициализирует COM-библиотеку, вызывая функции типа CoInitializeEx. Когда клиент запрашивает создание экземпляра COM-компонента (например, через CoCreateInstance или CoGetClassObject), OLE32.DLL определяет местоположение сервера по его CLSID с помощью реестра и загружает его в память процесса (клиента или отдельного серверного процесса). При загрузке DLL-сервера система вызывает его функцию входа/выхода (если она реализована), которая может использоваться для инициализации ресурсов сервера.

**2. Получение фабрики классов:** OLE32.DLL вызывает первую обязательную экспортируемую функцию сервера DllGetClassObject, передавая CLSID запрошенного компонента. В этой функции сервер создает экземпляр соответствующей фабрики классов и возвращает клиенту через OLE32.DLL указатель на интерфейс IClassFactory.

**3. Создание экземпляров компонентов:** Клиент использует полученный указатель на IClassFactory и вызывает метод CreateInstance для создания экземпляров нужных COM-компонентов. В процессе создания экземпляра компонента его конструктор увеличивает **счетчик экземпляров компонент** на сервере.

**4. Взаимодействие с компонентами:** Клиент получает указатели на интерфейсы компонентов (начиная с IUnknown) и вызывает их методы. Получение указателя на интерфейс через IUnknown::QueryInterface или создание компонента через CoCreateInstance (которая сама вызывает QueryInterface) увеличивает **счетчик ссылок** на этот интерфейс.

**5. Блокировка сервера:** Клиент или сам сервер может вызвать метод IClassFactory::LockServer для увеличения **счетчика блокировки сервера**, явно предотвращая его выгрузку, даже если нет активных экземпляров компонентов.

**6. Освобождение интерфейсов и экземпляров:** Когда клиент завершает работу с интерфейсом, он вызывает IUnknown::Release, уменьшая счетчик ссылок на этот интерфейс. Когда счетчик ссылок для экземпляра компонента достигает нуля, компонент может быть уничтожен (вызывается его деструктор). Деструктор компонента уменьшает **счетчик экземпляров компонент** на сервере.

**7. Попытка выгрузки сервера:** Периодически (или по запросу) OLE32.DLL вызывает обязательную экспортируемую функцию сервера DllCanUnloadNow. Эта функция должна проверить, можно ли выгрузить сервер. Сервер может быть выгружен только в том случае, если **счетчик экземпляров компонент равен нулю И счетчик блокировок сервера равен нулю**.

Если эти условия выполняются, DllCanUnloadNow возвращает S\_OK, и OLE32.DLL выгружает сервер. В противном случае возвращается S\_FALSE или код ошибки, и сервер остается загруженным.

**8. Завершение работы:** Клиентское приложение освобождает библиотеку COM, вызывая CoUninitialize. Если сервер является EXE-приложением, он может завершить свою работу после выгрузки всех компонентов и снятия всех блокировок. Если сервер является DLL, он выгружается из адресного пространства процесса, вызвавшей его, после успешного выполнения DllCanUnloadNow.

***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?***

Процесс регистрации COM-объекта заключается в занесении необходимой информации о компоненте в реестр Windows. Эта регистрация необходима для реализации свойства «Независимости от местоположения» (Location Transparency). Независимость от местоположения означает, что клиент компонента, пользователь, не обязан знать, где фактически расположен компонент. Клиентское приложение использует одинаковые сервисы COM для создания экземпляра компонента и взаимодействия с ним, независимо от того, находится ли компонент в адресном пространстве клиента (DLL-файл), в пространстве другой задачи на том же компьютере (EXE-файл) или на удаленном компьютере. Именно по идентификатору CLSID через реестр операционной системы библиотека OLE32.DLL определяет местоположение контейнера компонента, загружает и инициализирует его.

Утилита **regsvr32** применяется для регистрации COM-компонентов. По сути, данная утилита просто вызывает определенные экспортируемые функции из DLL-контейнера компонента для выполнения процесса регистрации или отмены регистрации.

COM/DLL-контейнер, обеспечивающий работу одного или нескольких COM-компонентов, обязательно должен экспортировать набор из пяти обязательных функций:

- **DllCanUnloadNow:** Эта функция автоматически вызывается OLE32.DLL перед попыткой клиента выгрузить COM-сервер. В зависимости от результата работы функции OLE32.DLL принимает решение о выгрузке COM-сервера или отказе от нее.
- **DllGetClassObject:** Первая функция компонента, которую вызывает OLE32.DLL при взаимодействии с клиентом. Функция проверяет идентификатор компонента (CLSID), создает фабрику классов компонента (объект, реализующий интерфейс IClassFactory) и

через параметры возвращает OLE32.DLL указатель на стандартный интерфейс IClassFactory.

- **DllInstall**: Эта функция вызывается утилитой regsvr32 при наличии соответствующего параметра и применяется для выполнения дополнительных действий при регистрации и отмене регистрации компонентов.
- **DllRegisterServer**: Эта функция вызывается утилитой regsvr32 при наличии соответствующего параметра и применяется для регистрации информации о компонентах сервера в реестре операционной системы.
- **DllUnregisterServer**: Эта функция вызывается утилитой regsvr32 при наличии соответствующего параметра и применяется для удаления информации о компонентах сервера из реестра.

При работе с памятью в COM-приложениях COM определяет свои собственные функции для выделения и освобождения памяти в куче. Это функции:

- **CoTaskMemAlloc**: выделяет блок памяти.
- **CoTaskMemFree**: освобождает блок памяти, выделенный с помощью CoTaskMemAlloc.

COM определяет свои функции управления памятью по нескольким причинам:

- **Уровень абстракции**: Эти функции предоставляют уровень абстракции над конкретным распределителем кучи. Без них различные методы могли бы использовать malloc или new, требуя от клиента вызывать free или delete соответственно, что быстро стало бы сложным отслеживать. Функции выделения памяти COM создают единый подход.
- **Независимость от языка программирования**: COM является двоичным стандартом и не привязан к определенному языку программирования. Следовательно, COM не может полагаться на специфические для языка формы распределения памяти. Использование общих функций CoTaskMemAlloc и CoTaskMemFree обеспечивает совместимость при взаимодействии компонентов, написанных на разных языках.

Таким образом, COM предоставляет стандартизированный механизм управления памятью для обеспечения взаимодействия бинарных компонентов, написанных на различных языках, и упрощения разработки, предоставляя единый набор функций для работы с памятью, используемой COM-объектами и их интерфейсами.

***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 или служба** – это процесс, который выполняет служебные функции. Сервисы являются аналогами резидентных программ, которые использовались в операционных системах, предшествующих Windows NT. По сути, сервис — это программа, которая запускается при загрузке операционной системы или в процессе ее работы по специальной команде и завершает



свою работу при завершении работы ОС или по специальной команде. Однако, не каждая программа, запускаемая со стартом операционной системы, является сервисом. Обычно сервисы выполняют определенные служебные функции, необходимые для работы приложений или какого-то конкретного приложения.

#### **Виды сервисов:**

- **По выполняемым функциям:**
  - **Серверы** – например, фоновый процесс, обеспечивающий доступ к базе данных.
  - **Драйверы** – программы, обеспечивающие доступ к внешним устройствам.
  - **Мониторы** – процесс, отслеживающий работу некоторого приложения.
- **По способу выполнения (в структуре SERVICE\_STATUS):**
  - **SERVICE\_WIN32\_OWN\_PROCESS** – сервис является самостоятельным процессом.
  - **SERVICE\_WIN32\_SHARE\_PROCESS** – сервис разделяет процесс с другими сервисами.
  - **SERVICE\_KERNEL\_DRIVER** – сервис является драйвером устройства.
  - **SERVICE\_FILE\_SYSTEM\_DRIVER** – сервис является драйвером файловой системы.
  - **SERVICE\_USER\_OWN\_PROCESS** – сервис является самостоятельным процессом, запускаемым для определенного пользователя.
  - **SERVICE\_USER\_SHARE\_PROCESS** – сервис разделяет процесс с другими сервисами, запускаемыми для определенного пользователя.
  - Флаги **SERVICE\_WIN32\_OWN\_PROCESS** и **SERVICE\_WIN32\_SHARE\_PROCESS** могут быть установлены совместно с флагом **SERVICE\_INTERACTIVE\_PROCESS** – сервис может взаимодействовать с рабочим столом.

#### **Основные характеристики сервисов:**

- Работают только в **фоновом режиме**.
- **Не имеют собственного управляющего интерфейса** (ни GUI, ни TUI).
- **Управляются специальной программой ОС – менеджером служб** (Service Control Manager, SCM).
- **Запускаются/останавливаются** со стартом/выключением ОС, со входом/выходом пользователя или по команде (от менеджера служб).
- Предназначены для **предоставления услуг другим программам или ОС**, а не пользователям.

SCM (Service Control Manager, менеджер сервисов) – это **специальная программа операционной системы, которая управляет работой сервисов**.

#### **Функции, которые выполняет менеджер сервисов:**

- **Поддержка базы данных установленных сервисов.**
- **Запуск сервисов** при загрузке операционной системы или по запросу.
- **Перечисление установленных сервисов.**
- **Поддержка информации о состоянии работающих сервисов.**

- **Передача управляющих запросов** работающим сервисам.

Структура Windows-сервиса включает:

- Главную функцию приложения (main для консольного приложения), задача которой – запустить **диспетчер сервиса**. Это выполняется вызовом функции **StartServiceCtrlDispatcher**, которая должна быть вызвана в течение 30 секунд с момента запуска программы.
- **Функцию обратного вызова**, определяющую **точку входа сервиса**. Эта функция (обычно ServiceMain, но может иметь другое имя) запускается диспетчером сервиса и содержит основную логику сервиса.
- **Функцию обратного вызова** (например, ServiceCtrlHandler), которая **реагирует на управляющие сигналы от операционной системы**. Эта функция регистрируется в начале работы сервиса вызовом **RegisterServiceCtrlHandler** и выполняется в контексте соответствующего диспетчера сервисов.

В теле функции точки входа сервиса (ServiceMain) выполняется следующая последовательность действий:

1. Немедленно запускается обработчик управляющих команд (RegisterServiceCtrlHandler).
2. Устанавливается стартовое состояние сервиса (SERVICE\_START\_PENDING) с помощью **SetServiceStatus**.
3. Проводится локальная инициализация сервиса.
4. Устанавливается рабочее состояние сервиса (SERVICE\_RUNNING) с помощью **SetServiceStatus**.
5. Выполняется работа сервиса, учитывая возможные изменения состояния.
6. После перехода в состояние останова (SERVICE\_STOPPED) освобождаются ресурсы и работа завершается.

Особенность точки входа сервиса (функция, обычно ServiceMain) заключается в том, что она **запускается не напрямую операционной системой, а потоком-диспетчером сервиса**.

Главная задача основной функции приложения (main) – вызвать функцию **StartServiceCtrlDispatcher**, которая запустит этот диспетчер. Вызов **StartServiceCtrlDispatcher** должен произойти в течение 30 секунд после запуска приложения. Вся необходимая инициализация самого сервиса должна выполняться в теле функции точки входа (ServiceMain).

Функция обратного вызова – это функция, которая **передаётся другой функции в качестве аргумента**. В контексте сервисов, это функции, которые вы реализуете в коде сервиса, а операционная система (через SCM и диспетчер сервиса) вызывает их в ответ на определенные события (запуск сервиса, управляющие сигналы).

SCM поддерживает базу данных установленных сервисов в **реестре Windows**. Эта база данных находится по ключу **HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services**. Под этим ключом находится подраздел для каждого установленного сервиса и сервиса драйвера, имя подраздела совпадает с названием службы.

Информация, хранящаяся в реестре для каждого сервиса, включает:

- **DependOnGroup** – группы порядка загрузки, от которых зависит сервис.

- **DependOnService** – сервисы, от которых зависит данный сервис.
- **Description** – описание сервиса.
- **DisplayName** – отображаемое имя сервиса.
- **ErrorControl** – уровень управления ошибками.
- **FailureActions** – действия при возникновении ошибки.
- **Group** – группа порядка загрузки, в которой состоит сервис.
- **ImagePath** – путь к исполняемому файлу сервиса.
- **ObjectName** – учетная запись, от имени которой запускается сервис.
- **Start** – тип запуска сервиса.
- **Tag** – уникальный тег сервиса в рамках группы порядка загрузки.
- **Type** – тип сервиса.

**Группы порядка загрузки** – это логически объединенный набор сервисов, который **определяет порядок загрузки сервисов**, входящих в него, относительно остальных групп или сервисов. Порядок загрузки сервисов в Windows определяется следующим образом:

1. Сначала определяется порядок групп в списке групп порядка загрузки (хранится в реестре по пути `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ServiceGroupOrder`).
2. Затем определяется порядок загрузки сервисов в рамках их собственной группы (хранится в реестре по пути `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\GroupOrderList`).

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

**Демон** (англ. daemon) — это процесс, который обладает двумя основными свойствами:

- **Имеет длинный жизненный цикл.** Часто демоны создаются во время загрузки системы и работают до момента ее выключения.
- **Выполняется в фоновом режиме и не имеет контролирующего терминала.** Эта особенность гарантирует, что ядро не сможет генерировать для такого процесса никаких сигналов, связанных с терминалом или управлением заданиями (таких как SIGINT, SIGTSTP и SIGHUP).

Названия демонов принято заканчивать буквой **d**, хотя это не является обязательным правилом.

Чтобы стать демоном, программа должна выполнить следующую последовательность шагов:

1. **Вызов `fork`.** Родительский процесс завершается, а потомок продолжает работать. В результате этого демон становится потомком процесса `init`. Этот шаг выполняется по двум причинам:
  - Если демон был запущен из командной строки, завершение родителя будет обнаружено командной оболочкой, которая выведет новое приглашение и позволит потомку выполняться в фоновом режиме.

- Потомок гарантированно не станет лидером группы процессов, поскольку он наследует PGID от своего родителя и получает свой уникальный идентификатор, который отличается от унаследованного PGID. Это необходимо для успешного выполнения следующего шага.
- 2. **Дочерний процесс вызывает `setsid`.** Это действие начинает новую сессию и разрывает любые связи с контролирующим терминалом.
- 3. **Обеспечение отсутствия контролирующего терминала.** Если после вызова `setsid` демон больше не открывает никаких терминальных устройств, можно не беспокоиться о восстановлении соединения с контролирующим терминалом. В противном случае, чтобы терминальное устройство не стало контролирующим, можно поступить двумя способами:
  - Указывать флаг `O_NOCTTY` для любых вызовов `open`, которые могут открыть терминальное устройство.
  - Более простой вариант: после `setsid` можно еще раз сделать вызов `fork`, снова позволив родителю завершиться, а потомку (правнуку) — продолжить работу. Это гарантирует, что потомок не станет лидером сессии.
- 4. **Очистить атрибут `umask` процесса.** Это делается для того, чтобы файлы и каталоги, созданные демоном, имели запрашиваемые права доступа.
- 5. **Поменять текущий рабочий каталог процесса (обычно на корневой — `/`).** Это необходимо, поскольку демон обычно выполняется до выключения системы. Если его текущий рабочий каталог находится на файловой системе, отличной от корневой, она не может быть отключена. В качестве варианта демон может задействовать место, где он выполняет свою работу, или использовать значение из конфигурационного файла; главное, чтобы файловая система, в которой находится этот каталог, никогда не отключалась во время работы демона.
- 6. **Закрывать все открытые файловые дескрипторы,** которые демон унаследовал от своего родителя (возможно, некоторые из них необходимо оставить открытыми, поэтому данный шаг является необязательным и может быть откорректирован). Это делается по ряду причин:
  - Поскольку демон потерял свой контролирующий терминал и работает в фоновом режиме, ему больше не нужно хранить дескрипторы с номерами 0, 1 и 2 (если они ссылаются на терминал).
  - Нельзя отключить файловую систему, на которой долгоживущий демон удерживает открытыми какие-либо файлы.
  - Следуя обычным правилам, следует закрывать неиспользуемые дескрипторы.
- 7. **Перенаправить дескрипторы с номерами 0, 1 и 2 (после их закрытия)** в предварительно открытый файл `/dev/null`, используя вызов `dup2` (или похожий). Это делается по двум причинам:
  - Позволяет избежать ошибки при вызове библиотечных функций, которые выполняют операции ввода/вывода с этими дескрипторами.
  - Исключает возможность повторного открытия демоном файлов с помощью дескрипторов 1 или 2, так как библиотечные функции, которые записывают в них данные, ожидают, что эти дескрипторы указывают на потоки `stdout` (стандартный вывод) и `stderr` (стандартный вывод ошибок).

При разработке демонов следует учитывать следующие рекомендации:

- **Завершение работы:** Демон обычно завершается во время выключения системы. Для стандартных демонов предусмотрены специальные скрипты завершения. Остальные демоны получают сигнал SIGTERM от процесса init при выключении компьютера. Демон должен выполнять освобождение ресурсов в обработчике этого сигнала. Процедуру очистки следует выполнять как можно быстрее, так как через 5 секунд после SIGTERM процесс init отправляет сигнал SIGKILL, который приводит к завершению процесса.
- **Утечки ресурсов:** Поскольку демоны имеют длинный жизненный цикл, необходимо тщательно следить за потенциальными утечками памяти и файловых дескрипторов. При утечках файловых дескрипторов демон приходится перезапускать.
- **Один экземпляр:** Часто демону нужно гарантировать, что в системе активен только один его экземпляр. Обычно это достигается путем создания файла в стандартном каталоге и применения к нему блокировки на запись на протяжении всего существования демона. Если другой экземпляр попытается запуститься, он не сможет получить блокировку и завершится.

**init** (Initd) и **systemd** являются **менеджерами сервисов** в Linux. Они отвечают за управление демонами. Стоит заметить, что ранее описанный алгоритм "демонизации" приложения выполняется менеджерами самостоятельно. Нет необходимости в коде вашего будущего демона совершать описанные манипуляции. Они также позволяют настроить автозапуск сервисов вместе с запуском операционной системы. В подсистеме Linux для Windows (WSL) можно встретить оба варианта: Debian использует init, а Ubuntu — systemd.

Добавление сервиса на уровне системы с помощью Initd (менеджера сервисов):

1. **Скомпилируйте приложение** будущего демона.
2. **Разместите получившийся бинарный файл** в одной из папок `/sbin` или `/usr/sbin`.
3. **Разместите скрипт запуска** `<имя демона>` в каталоге `/etc/init.d/`. Измените права доступа к нему с помощью `chmod` на `755`.
4. (Опционально) **Создайте каталог** `/etc/<имя демона>` для размещения конфигурационных файлов, которые будут использоваться самим сервисом.
5. (Справочно) **Файлы логов** обычно располагают в каталоге `/var/log`.
6. **Проверьте с помощью команды `service`**, что система распознала скрипт инициализации сервиса.
7. Если скрипт написан корректно, **управление сервисом** будет доступно через команду `service` (например, `service <имя демона> start, stop, status` и т.д.).
8. **Для автозапуска** вместе с системой перейдите в каталог `/etc/init.d` и выполните команду `update-rc.d <имя демона> defaults`.

В системах с менеджером **systemd**, работающим в режиме совместимости с **initd**, данная последовательность действий также является полностью рабочей.

**Кратко о systemd:** При использовании менеджера **systemd** вместо скрипта в `/etc/init.d` создается файл описания сервиса с расширением `.service` в каталоге

/etc/systemd/system/. Управление сервисами осуществляется с помощью команды `systemctl` (например, `systemctl start <имя демона>.service, enable <имя демона>.service` для автозапуска).

## ***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 устройства** — это **системный программный компонент**, обеспечивающий связь между операционной системой и конкретным аппаратным устройством.

Проще: **драйвер знает, как "разговаривать" с железом** и предоставляет унифицированный интерфейс для ядра ОС и приложений.

Драйвер является частью кода операционной системы. Он действует как посредник между операционной системой/приложениями и аппаратным обеспечением устройства. Драйверы находятся в режиме ядра (Kernel-Mode Drivers), хотя существуют и драйверы пользовательского режима (User-Mode Drivers). В операционной системе Windows программные потоки выполняют операции ввода/вывода с виртуальными файлами. Диспетчер ввода/вывода ни с чем другим работать не умеет, и именно драйвер отвечает за преобразование файловых команд (открытие, закрытие, чтение, запись) в команды для конкретного устройства.

Драйвер может быть легко заменяемой частью операционной системы. Как отдельный и довольно независимый модуль, драйвер сформировался не сразу. Существование режима ядра является одной из общих концепций драйверов в Windows- и UNIX-системах.

### **Место драйвера в структуре ОС**

Драйвер — часть **ядра** или уровня, близкого к ядру

- Работает в **режиме ядра (kernel mode)**
- Обеспечивает **аппаратную абстракцию**: ОС не должна знать особенности каждого принтера/клавиатуры/диска
- В Windows драйверы — это **.sys** файлы, загружаемые диспетчером устройств (`ntoskrnl.exe / I/O Manager`)
- В Linux — модули ядра (**.ko**), подключаемые через **modprobe** или встроенные

Среди основных общих концепций драйверов в Windows- и UNIX-системах можно выделить:

- **Способ работы с драйверами как файлами.** Функции, используемые при взаимодействии с файлами (`open`, `close`, `read`, `write`), практически идентичны таковым при взаимодействии с драйверами.
- **Драйвер, как легко заменяемая часть ОС.**

- **Существование режима ядра.** Разработка программ в режиме пользователя и ядра имеет существенные различия, например, необработанные исключения в режиме ядра "уронят" систему, а не только процесс.
- **Механизм IOCTL (Input/Output Control Code).**
- **Многоуровневая модель и расширяемость.** Драйверы могут быть одно- и многоуровневыми. При многоуровневой архитектуре обработка запросов ввода/вывода распределяется между несколькими драйверами, каждый из которых выполняет свою часть работы. Между этими драйверами можно «поставить» любое количество фильтр-драйверов. Данные идут от вышестоящих драйверов (higher-level) к нижестоящим (lower-level), а при возврате результатов – наоборот.
- **Стек драйверов** – набор драйверов, которые необходимо вызывать для получения конечного результата.
- **Динамическая загрузка и выгрузка драйверов устройств** позволяют выполнять данные процедуры по запросу, экономя системные ресурсы.
- **Технология Plug and Play (PnP)** – обеспечивает обнаружение и установку драйверов для нового оборудования и выделение им нужных аппаратных ресурсов.

Также в контексте ввода/вывода в ядре Windows важны две концепции:

- **Уровни запросов прерываний, или IRQ (Interrupt Request Level).** IRQ может означать приоритет, назначаемый источнику прерываний от физического устройства, или текущий уровень IRQ у каждого центрального процессора. Код с более низким IRQ не может вмешиваться в работу кода с более высоким IRQ. Важные уровни IRQ для ввода/вывода включают Passive (0), Dispatch/DPC (2) и Device IRQ (DIRQL). (Уявіце што гэта як прыярытэты патокаў і адразу ўсё больш-менш зразумела стане)
- **Отложенные вызовы процедур, или DPC (Deferred Procedures Calls).** DPC – это объект, инкапсулирующий вызов функции на уровне IRQ DPC\_LEVEL (2). Объекты DPC существуют прежде всего для выполнения действий после прерывания. Отложенность означает, что DPC не выполняется немедленно, а запускается, когда уровень IRQ процессора падает до 2.

В Windows NT существуют типы драйверов пользовательского режима и режима ядра. WDM-драйверы являются драйверами устройств, соответствующими модели WDM, и делятся на драйверы шины, функциональные драйверы и фильтрующие драйверы. Современные драйверы для Windows создаются с применением модели Windows Driver Foundation (WDF), которая включает KMDF и UMDF. WDF-драйвер является WDM-драйвером, но не наоборот.

Подсистема ввода/вывода в Windows состоит из набора компонентов исполнительной системы, которые совместно управляют устройствами и предоставляют приложениям и системе интерфейсы к этим устройствам. Подсистема ввода/вывода в Windows проектировалась как абстрактный интерфейс приложений для аппаратных (физических) и программных (виртуальных, или логических) устройств.

Функциональные возможности подсистемы ввода/вывода включают:

- Унифицированные средства безопасности и именования устройств для защиты общих ресурсов.
- Высокопроизводительный асинхронный пакетный ввод/вывод.

- Специальные службы, позволяющие писать драйверы устройств на высокоуровневом языке и упрощающие их перенос на машины с другой архитектурой.
- Многоуровневая модель и расширяемость, обеспечивающие возможность добавлять драйверы, меняющие поведение других драйверов или устройств без необходимости модификации последних.
- Динамическая загрузка и выгрузка драйверов устройств.
- Поддержка технологии Plug and Play, обеспечивающая обнаружение и установку драйверов для нового оборудования и выделение им нужных аппаратных ресурсов, давая приложениям возможность находить и задействовать интерфейсы устройств.
- Подсистема управления электропитанием, позволяющая системе или отдельным устройствам переходить в состояния с низким энергопотреблением.

Для реализации своей функциональности подсистема ввода/вывода Windows включает ряд компонентов исполнительной подсистемы и драйверов устройств. Центральное место занимает **диспетчер ввода/вывода (I/O manager)**.

Другие компоненты и концепции, связанные с подсистемой ввода/вывода:

- **Драйверы устройств.**
- **PnP-диспетчер** (работает совместно с диспетчером ввода/вывода и драйвером шины).
- **Диспетчер электропитания** (тесно связан с диспетчером ввода/вывода и PnP-диспетчером).
- **Реестр** (база данных с описанием подключенных устройств, параметров инициализации драйверов и конфигурации).
- **INF-файлы** (управляют установкой драйверов).
- **Удостоверяющие файлы драйверов (.cat)** (хранят цифровые подписи).
- **Уровень аппаратных абстракций (Hardware Abstraction Layer, HAL)** (изолирует драйверы от специфических особенностей процессоров и контроллеров прерываний).
- **Пакеты запросов на ввод/вывод (I/O Request Packets, IRP)** (представляют запросы на ввод и вывод в памяти).
- **Виртуальные файлы** (любой источник или приемник запроса на ввод/вывод, который рассматривается как файл).
- **Уровни запросов прерываний (IRQL).**
- **Отложенные вызовы процедур (DPC).**
- **Объект драйвера (DRIVER\_OBJECT)** (представляет отдельный драйвер).
- **Объект устройства (DEVICE\_OBJECT)** (представляет физическое или логическое устройство).
- **Файловый объект** (структура данных режима ядра, представляющая дескриптор устройства).

**21. Драйверы. Что такое драйвер устройства? Что такое диспетчер ввода/вывода? Каково 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?**



# Что такое драйвер устройства?

**Драйвер устройства** — это модуль ядра, обеспечивающий взаимодействие между операционной системой и конкретным аппаратным устройством.

Он:

- управляет конкретным железом (диск, клавиатура, сетевая карта...)
- реализует стандартный интерфейс (например, `IRP_MJ_READ`, `IRP_MJ_WRITE`)
- скрывает от ОС детали работы устройства

Драйвер делает устройство «понятным» для операционной системы и приложений.

Драйвер — это часть кода операционной системы, отвечающая за взаимодействие с аппаратурой. Под аппаратурой в данном контексте подразумеваются как реальные физические устройства, так и виртуальные или логические. **Драйвер — это программное обеспечение, которое предоставляет другому программному обеспечению API для работы с аппаратными устройствами.** Например, когда приложению требуется считать данные с устройства, оно вызывает функцию операционной системы, которая затем вызывает функцию, реализованную драйвером. Драйвер, обычно разрабатываемый производителем устройства, знает, как взаимодействовать с аппаратным обеспечением устройства для получения данных, и возвращает их обратно в операционную систему, а затем в приложение.

Подсистема ввода/вывода в Windows состоит из набора компонентов исполнительной системы, которые совместно управляют устройствами и предоставляют приложениям и системе интерфейсы к этим устройствам. **Центральное место в подсистеме ввода/вывода занимает диспетчер ввода/вывода (I/O manager).**

## Что такое диспетчер ввода/вывода (I/O Manager)?

Это ключевой компонент ядра Windows, отвечающий за обработку **всех операций ввода-вывода** (файлы, устройства, сети и т.д.).

### Назначение:

- Создаёт и отправляет **I/O Request Packets (IRP)** драйверам
- Вызывает соответствующие функции драйвера
- Поддерживает **стек драйверов** (например, фильтрующий → файловая система → диск)
- Обеспечивает интерфейс системных вызовов типа `ReadFile`, `WriteFile`

Он — как диспетчер маршрутов: получает запрос и передаёт его нужному драйверу.

## Что такое PnP-диспетчер (Plug and Play Manager)?

**PnP-диспетчер** — это компонент ОС, отвечающий за **динамическое управление устройствами**.

### Назначение:

- Автоматически **обнаруживает** устройства (вставка USB, запуск Wi-Fi и т.д.)
- Назначает **ресурсы** (IRQ, I/O-порты, память)
- **Загружает драйверы** устройств через INF-файлы и реестр
- Удаляет устройства при отключении

Он делает возможной "горячую" замену устройств и минимальное участие пользователя в установке драйверов.

## Что такое диспетчер электропитания (Power Manager)?

Это подсистема ядра Windows, которая управляет **режимами сна, пробуждения, энергосбережением** для всех устройств.

### Назначение:

- Координирует перевод устройств в состояния D0–D3 (работа – выключено)
- Реагирует на события вроде закрытия крышки ноутбука, простоя
- Взаимодействует с драйверами через IRP IRP\_MJ\_POWER

Помогает ОС экономить энергию и управлять питанием как на уровне системы, так и на уровне отдельных устройств.

## Как используется реестр в работе драйверов?

Драйверы читают конфигурацию из **реестра Windows** при загрузке.

### Используется для:

- Хранения **параметров и настроек драйвера**

- Указания пути к двоичному файлу драйвера
- Хранения статуса устройства
- Настройки автозагрузки

Раздел:

HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\{DriverName}

**INF-файлы** — это файлы с расширением `.inf`, которые **управляют установкой драйверов**. Они связывают аппаратные устройства с драйверами, управляющими этими устройствами. Содержимое такого файла состоит из инструкций, которые описывают устройство, исходное и целевое положение файлов драйвера, вносимые в реестр при установке драйвера изменения, и сведения о зависимостях драйвера.

**Уровень аппаратных абстракций (Hardware Abstraction Layer, HAL)** - это компонент ядра, который скрывает различия аппаратных платформ от остального кода ОС, он поддерживает прикладные программные интерфейсы, скрывающие межплатформенные различия:

- Обеспечивает **единый интерфейс** к CPU, таймерам, прерываниям
- Позволяет ОС работать на **разных архитектурах** без изменения остального кода ядра
- HAL «переводит» системные вызовы в инструкции, специфичные для платформы (например, Intel x86 vs ARM)

По факту, он делает ОС **переносимой и независимой от конкретного железа**. Ну и назначает приоритет IRQL источникам прерывания, но Ярик еблан это стер.

**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.**

## Драйверы

**Драйвер** – это часть кода операционной системы (ОС), которая **отвечает за взаимодействие с аппаратурой**. Аппаратура в этом контексте может означать как реальные физические устройства, так и виртуальные или логические. По сути, драйвер – это программное обеспечение, которое **предоставляет другому программному обеспечению API (интерфейс прикладного программирования) для работы с аппаратными устройствами**.

Например, когда приложению нужно считать данные с устройства, оно вызывает функцию, реализованную операционной системой. Затем ОС вызывает функцию, реализованную

драйвером. Драйвер, который обычно разрабатывается производителем устройства, обладает знаниями о том, как взаимодействовать с аппаратным обеспечением для получения данных. Получив данные, драйвер возвращает их обратно в операционную систему, а она, в свою очередь, передает их обратно приложению.

Драйверы постоянно эволюционировали, и этот процесс не завершен. Одним из ключевых моментов эволюции является становление концепции драйвера как **легко заменяемой части операционной системы**. Изначально они не были независимыми модулями, и до сих пор многие драйверы тесно интегрированы с ОС, что иногда требует переустановки системы (в Windows) или пересборки ядра (в UNIX-системах) при их изменении.

Общие концепции работы с драйверами в Windows- и UNIX-системах включают:

- Способ работы с драйверами как с файлами, что означает использование идентичных функций (например, `open`, `close`, `read`).
- Драйвер как легко заменяемая часть ОС.
- Существование режима ядра.
- Идентичность механизма IOCTL (Input/Output Control Code).

В Windows, подсистема ввода/вывода (I/O) спроектирована как абстрактный интерфейс для аппаратных и программных устройств, обеспечивая унифицированные средства безопасности и именования, высокопроизводительный асинхронный пакетный ввод/вывод, возможность написания драйверов на высокоуровневом языке, многоуровневую модель и расширяемость, динамическую загрузку/выгрузку, а также поддержку Plug and Play.

### «Жизненный цикл» IRP (I/O Request Packet)

**Пакет запроса ввода/вывода (IRP)** – это структура данных, которая содержит полную информацию о запросе ввода/вывода и **представляет операции ввода/вывода в памяти**. **Диспетчер ввода/вывода (I/O manager)** является центральным элементом подсистемы ввода/вывода и отвечает за создание инфраструктуры для доставки IRP драйверам устройств.

«Жизненный цикл» IRP выглядит следующим образом:

1. Когда приложение или системный компонент запрашивает операцию ввода/вывода, **диспетчер ввода/вывода генерирует IRP-пакет**.
2. **Указатель на созданный IRP-пакет передается соответствующему драйверу** через одну из процедур диспетчеризации (например, `open`, `close`, `read`, `write`, PnP-операции).
3. **Драйвер, получивший IRP, выполняет указанную в пакете операцию**. Если драйвер является частью многоуровневой архитектуры (например, WDM-драйвер), он может обработать часть запроса и **переслать IRP другим драйверам** для дальнейшей обработки (от вышестоящих к нижестоящим).
4. После завершения операции или передачи пакета, драйвер **возвращает пакет диспетчеру ввода/вывода**, сигнализируя о завершении операции.
5. **По завершении операции пакет удаляется**.

Эта пакетная обработка позволяет отдельному программному потоку одновременно управлять набором запросов на ввод/вывод. Драйверы могут использовать процедуру начала ввода/вывода

для инициации передачи данных, при этом диспетчер ввода/вывода может сериализовать IRP-пакеты, если устройство не поддерживает параллельную работу с несколькими запросами.

## Виртуальные файлы

В операционной системе Windows **программные потоки выполняют операции ввода/вывода с виртуальными файлами**. Термин «виртуальный файл» относится к **любому источнику или приемнику запроса на ввод/вывод, который рассматривается как файл**. Это может быть устройство, файл, папка, канал или почтовая ячейка.

Операционная система **абстрагирует все запросы ввода/вывода как операции с виртуальными файлами**, поскольку диспетчер ввода/вывода умеет работать только с этим концептом. При этом **за преобразование файловых команд** (таких как открытие, закрытие, чтение, запись) **в команды для конкретного устройства отвечает драйвер**.

Файловые объекты, представляющие дескрипторы устройств, являются структурами данных режима ядра. Они функционируют как системные ресурсы, доступные нескольким процессам в пользовательском режиме, могут иметь имена, защищены моделью безопасности объектов и поддерживают синхронизацию.

## Особенности программирования на уровне ядра

Разработка программ в режиме пользователя и режиме ядра имеет существенные различия:

- **Необработанные исключения:** В режиме пользователя они «уронят» процесс, тогда как в режиме ядра – приведут к падению всей системы.
- **Завершение работы:** В режиме пользователя все приватные ресурсы процесса освобождаются автоматически. **В режиме ядра, если драйвер выгружен без очистки выделенных ресурсов, утечка ресурсов устраняется только при следующем перезапуске системы.**
- **Возвращаемые значения:** В режиме пользователя коды (значения) ошибок часто игнорируются, тогда как **в режиме ядра это практически недопустимо.**
- **Уровни запросов прерываний (IRQL):** В режиме пользователя значение IRQL всегда равно 0, а **в режиме ядра оно может быть разным, что требует знания текущего уровня, на котором работает функция.**
- **Плохо написанный код:** В режиме пользователя эффект от такого кода обычно не выходит за рамки процесса, тогда как **в режиме ядра эффект распространяется на всю систему.**
- **Тестирование и отладка:** В режиме пользователя производится на устройстве разработчика. **В режиме ядра для этого требуется отдельное устройство.**
- **Использование библиотек:** В режиме пользователя доступны практически все библиотеки C/C++, тогда как **в режиме ядра большинство стандартных библиотек недоступны.**
- **Обработка исключений:** В режиме пользователя доступны исключения C++ и SEH. **В режиме ядра – только SEH.**
- **Использование C++:** В режиме пользователя доступна полная среда выполнения C++. **В режиме ядра – нет среды выполнения C++.**

## Уровни запросов прерываний (IRQL)

Уровни запросов прерываний (IRQL) – это важная концепция ядра Windows, имеющая два значения:

1. **Приоритет, назначаемый источнику прерываний от физического устройства.** Это число задается HAL (Hardware Abstraction Layer) при содействии контроллера прерываний.
2. **Уровень IRQL, на котором работает центральный процессор.**

Фундаментальное правило IRQL гласит, что код с более низким IRQL не может вмешиваться в работу кода с более высоким IRQL, и наоборот – код с более высоким IRQL не может вытеснять код, работающий с более низким IRQL. Обычно уровень IRQL процессора равен 0 (Passive level), что означает, что в системе не происходит «ничего особенного», и планировщик ядра работает нормально. В пользовательском режиме значение IRQL всегда равно 0, и его невозможно повысить.

Наиболее важные уровни IRQL в контексте ввода/вывода:

- **Passive (0):** Нормальный уровень IRQL, при котором работает планировщик ядра. Определяется макросом `PASSIVE_LEVEL`.
- **Dispatch/DPC (2):** Уровень IRQL, на котором работает планировщик ядра. Когда поток поднимает текущий уровень IRQL до 2 или выше, он фактически **получает бесконечный квант времени и не может быть вытеснен другим потоком**. Планировщик не может активизироваться на текущем процессоре, пока уровень IRQL не упадет ниже 2. Определяется макросом `DISPATCH_LEVEL`.
- **Device IRQL (DIRQL):** Диапазон уровней (3-26 на x86; 3-12 на x64 и ARM), закрепленных за аппаратными преобразованиями. При поступлении прерывания диспетчер вызывает соответствующую **функцию обслуживания прерывания (ISR, Interrupt Service Routine)** и повышает ее IRQL до DIRQL.

## Отложенные вызовы процедур (DPC)

Отложенный вызов процедуры (DPC) – это объект, **инкапсулирующий вызов функции на уровне IRQL `DPC_LEVEL` (2)**. DPC-объекты существуют прежде всего для **выполнения действий после прерывания**, так как выполнение на уровне DIRQL маскирует (а следовательно, задерживает) другие прерывания, ожидающие обработки.

Термин «отложенный» в названии означает, что DPC **не выполняется немедленно** – это невозможно, потому что текущий уровень IRQL выше 2. **Когда ISR возвращает управление, при отсутствии ожидающих обработки прерываний, уровень IRQL процессора падает до 2, и он выполняет накопившиеся вызовы DPC.** DPC-процедура выполняет основную часть обработки прерывания, оставшуюся после ISR-процедуры. Типичная DPC-процедура инициирует завершение одной операции ввода/вывода и начало следующей из очереди устройства.

## Пример взаимодействия IRQL и DPC

Рассмотрим последовательность событий при обработке прерывания:

1. **Исходное состояние:** Некий код пользовательского режима или режима ядра выполняется, когда процессор находится на уровне **IRQL 0 (Passive\_level)**. Большая часть времени процессор проводит именно на этом уровне.
2. **Поступление первого прерывания:** Поступает аппаратное прерывание, например, на уровне **IRQL 5**. Так как 5 больше 0, **состояние процессора сохраняется, IRQL повышается до 5, и вызывается обработчик ISR, связанный с прерыванием**. При этом не происходит переключения контекста; работает тот же поток, который теперь выполняет код ISR. Если поток был в пользовательском режиме, он переключается в режим ядра.
3. **Выполнение ISR 1:** **ISR 1 начинает выполняться на уровне IRQL 5**. В этот момент любые прерывания с IRQL 5 и ниже не могут вмешаться в обработку.
4. **Поступление второго (более высокого) прерывания:** Предположим, поступает новое прерывание с **IRQL 8**. Если система решает, что оно должно быть обработано тем же процессором, и так как  $8 > 5$ , **выполнение ISR 1 прерывается, состояние процессора сохраняется, IRQL повышается до 8, и процессор переходит к ISR 2**. Опять же, выполнение происходит в том же потоке, и переключение контекста невозможно, так как планировщик потоков не может активизироваться при таком высоком IRQL.
5. **Выполнение ISR 2 и постановка DPC в очередь:** Выполняется обработчик ISR 2. До его завершения ISR 2 может захотеть выполнить дополнительную обработку на более низком уровне IRQL (чтобы не блокировать прерывания с IRQL менее 8). Поэтому **ISR 2 вставляет правильно инициализированный объект DPC со ссылкой на функцию драйвера в очередь DPC**, вызвав, например, KeInsertQueueDpc. Затем ISR 2 возвращает управление, и восстанавливается состояние процессора, сохраненное перед входом в ISR 2.
6. **Возвращение к ISR 1:** На этой стадии **IRQL падает до предыдущего уровня (5), и процессор продолжает выполнение обработчика ISR 1**, который был прерван ранее.
7. **Завершение ISR 1 и постановка DPC в очередь:** Непосредственно перед завершением **ISR 1 также ставит в очередь собственный объект DPC** для выполнения своей последующей обработки. Затем ISR 1 возвращает управление, восстанавливая состояние процессора, сохраненное перед началом выполнения ISR 1.
8. **Обработка DPC:** В этот момент **уровень IRQL должен был бы упасть до старого нулевого значения**. Но ядро замечает наличие необработанных DPC, поэтому **IRQL уменьшается до уровня 2 (DPC\_LEVEL), и запускается цикл обработки DPC**, который перебирает накопившиеся DPC и последовательно вызывает каждую процедуру DPC. Когда очередь DPC опустеет, обработка DPC завершается.
9. **Возвращение к исходному коду:** Наконец, **IRQL уменьшается до 0, состояние процессора снова восстанавливается, и возобновляется выполнение изначально прерванного исходного кода пользовательского режима или режима ядра**.

Важно отметить, что вся описанная обработка происходит в одном потоке. Это означает, что ISR и процедуры DPC не должны зависеть от конкретного потока (и, следовательно, части конкретного процесса) для выполнения своего кода, так как поток может быть любым.

**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.***

### **Что такое драйвер?**

**Драйвер** — это часть кода операционной системы, которая **отвечает за взаимодействие с аппаратурой**. Он представляет собой программное обеспечение, предоставляющее другому программному обеспечению API (интерфейс прикладного программирования) для работы с аппаратными устройствами. Аппаратура в данном контексте понимается в самом широком смысле, включая как реальные физические устройства, так и виртуальные или логические.

Когда приложению требуется считать данные с устройства, оно вызывает функцию, реализованную операционной системой. Затем операционная система вызывает функцию, реализованную драйвером. Драйвер, обычно разрабатываемый производителем устройства, обладает знаниями о том, как взаимодействовать с аппаратным обеспечением для получения данных. Получив данные, драйвер возвращает их операционной системе, которая, в свою очередь, передает их обратно приложению.

Драйверы непрерывно эволюционировали с момента своего появления, и этот процесс продолжается. Одним из ключевых моментов эволюции является формирование концепции драйвера как **легко заменяемой части операционной системы**. Как отдельный и достаточно независимый модуль, драйвер сформировался не сразу, и многие из них до сих пор тесно интегрированы с ОС, что иногда требует переустановки системы (в Windows) или пересборки ядра (в UNIX-системах). В Windows- и UNIX-системах работа с драйверами как с файлами схожа, используя функции типа `open`, `close`, `read`.

### **Какие бывают драйверы?**

В операционных системах семейства Windows NT драйверы условно классифицируются на следующие типы:

- **Драйверы пользовательского режима (User-Mode Drivers):**
  - Драйверы, являющиеся компонентами среды UMDF.
  - Драйверы принтеров подсистемы Windows.
- **Драйверы режима ядра (Kernel-Mode Drivers):**
  - **Драйверы файловой системы (File system drivers):** принимают запросы ввода/вывода к файлам и на их основе выдают более конкретные запросы к драйверам запоминающих или сетевых устройств.
  - **Драйверы PnP (Plug and Play drivers):** работают с оборудованием и интегрированы с диспетчером электропитания и PnP-диспетчером. К этой



категории относятся драйверы запоминающих устройств, видеоадаптеров, устройств ввода и сетевых адаптеров.

- **Драйверы без поддержки Plug and Play (Non-Plug and Play drivers):** включают расширения ядра и делают систему более функциональной. Они, как правило, не интегрированы с PnP-диспетчером или диспетчером электропитания, поскольку не связаны с физическими аппаратными устройствами.

Современные драйверы для ОС семейства Windows создаются с применением **модели Windows Driver Foundation (WDF-драйверы)**, которая включает в себя две части: KMDF (Kernel-Mode Driver Framework) и UMDF (User-Mode Driver Framework). Любой WDF-драйвер является WDM-драйвером, но не наоборот. WDF значительно упрощает написание драйверов, скрывая многие нюансы, не относящиеся к логике самого драйвера, и совместим с более ранними WDM-драйверами.

## Что такое WDM-драйверы и какие они бывают?

**WDM-драйверы** — это драйверы устройств, соответствующие модели Windows Driver Model (WDM). WDM поддерживает управление электропитанием, технологию Plug and Play и инструментарий управления Windows.

Драйверы данной категории делятся на три основных типа:

- **Драйверы шины (bus drivers):** управляют логической или физической шиной.
- **Функциональные драйверы (function drivers):** управляют устройствами конкретного типа.
- **Фильтрующие драйверы (filter drivers):** могут располагаться как выше, так и ниже функционального и шинного драйверов. Они дополняют или меняют поведение устройства или другого драйвера.

## Что такое стек драйверов?

Такая многоуровневая архитектура приводит к появлению понятия **стека драйверов** — это набор драйверов, которые необходимо вызывать для получения конечного результата.

## Какие бывают многоуровневые WDM-драйверы?

Помимо WDM-драйверов шины, функциональных и фильтрующих драйверов, поддержка аппаратного обеспечения в многоуровневой архитектуре может обеспечиваться следующими компонентами:

- **Драйверы классов (class drivers):** отвечают за обработку ввода/вывода для устройств конкретного класса, таких как жесткий диск, клавиатура или компакт-диск.
- **Драйверы мини-классов (miniclass drivers):** реализуют обработку ввода/вывода, заданную производителем для определенного класса устройств. По сути, это DLL уровня ядра с определенным API, а не полноценные драйверы устройств.

- **Драйверы портов (port drivers):** обрабатывают запросы на ввод и вывод в соответствии с типом порта ввода/вывода, например SATA. Они реализуются как библиотеки функций режима ядра.
- **Драйверы мини-портов (miniport drivers):** преобразуют обобщенный запрос ввода/вывода о типе порта в запрос о типе адаптера. Они являются истинными драйверами устройств.

Драйверы могут быть одно- или многоуровневыми. В случае **многоуровневых драйверов**, обработка запросов ввода/вывода распределяется между несколькими драйверами, каждый из которых выполняет свою часть работы. Между этими драйверами может быть "поставлено" любое количество фильтр-драйверов. При этом различают **вышестоящие (higher-level)** и **нижестоящие (lower-level) драйверы**.

### **Последовательность вызова функционала, реализованного многоуровневым драйвером**

Система ввода/вывода, в центре которой находится **диспетчер ввода/вывода (I/O manager)**, задает инфраструктуру для доставки запросов на ввод/вывод драйверам устройств. Большинство запросов представлены **пакетами запросов на ввод/вывод (I/O Request Packets, IRP)**, которые передаются от одного компонента системы к другому.

Последовательность обработки запроса в многоуровневой модели выглядит следующим образом:

1. **Приложение или системный компонент** инициирует операцию ввода/вывода, которая абстрагируется операционной системой как операция с виртуальным файлом (устройством, файлом, папкой и т.д.).
2. **Диспетчер ввода/вывода** создает IRP-пакет, который содержит всю необходимую информацию об операции.
3. Этот **IRP-пакет передается вышестоящему драйверу** в стеке драйверов.
4. **Вышестоящий драйвер** обрабатывает свою часть запроса. Он может изменить IRP-пакет, добавить информацию или, в случае фильтрующего драйвера, изменить поведение запроса.
5. После обработки, вышестоящий драйвер, как правило, **пересылает IRP-пакет нижестоящему драйверу** для дальнейшей обработки. Этот процесс продолжается вниз по стеку драйверов.
6. В конечном итоге, **самый нижний драйвер** (часто функциональный драйвер или драйвер мини-порта) взаимодействует непосредственно с аппаратным обеспечением, преобразуя высокоуровневые команды в низкоуровневые, такие как запись в регистр управления устройством.
7. После завершения операции устройством, драйвер уведомляет диспетчер ввода/вывода. Результаты операции **возвращаются обратно по стеку драйверов**, от нижестоящих драйверов к вышестоящим.
8. **Драйверы обрабатывают IRP-пакет** (например, через процедуру DPC), сигнализируя о завершении операции или передавая пакет другому драйверу для дальнейшей обработки.
9. Наконец, **диспетчер ввода/вывода** удаляет IRP-пакет, и результат операции возвращается изначально вызвавшему его приложению или системному компоненту.

# ***БЛЯЦЬ ВІТАЕМ У ПЕКЛЕ, НАСТУПНЫЯ 2 БІЛЕТЫ - ПОЎНЫ ПІЗДЗЕЦ Я НАВАТ ВУЧЫЦЬ ГЭТА НЕ БУДУ, ГЭТА ІНСТА ПЕРАЗДАЧА***

***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?***

Драйвер — это часть кода операционной системы, отвечающая за взаимодействие с аппаратурой. Он представляет собой программное обеспечение, которое предоставляет другому программному обеспечению API для работы с аппаратными устройствами. Драйверы непрерывно эволюционируют, формируясь как легко заменяемая часть операционной системы. Обычно драйвер разрабатывается производителем устройства. Он знает, как взаимодействовать с аппаратным обеспечением устройства, чтобы получить данные и вернуть их операционной системе, которая затем передает их приложению.

## **Кто занимается запуском драйвера?**

Запуском драйвера занимается:

- Диспетчер устройств (Device Manager) через
- Plug and Play Manager и
- Service Control Manager (SCM) — системный компонент, управляющий службами (драйверы в Windows — тоже службы уровня ядра)

Вручную это делает `ntoskrnl.exe` при загрузке, либо `rundll32, sc.exe, devcon.exe` по команде пользователя.

## **Что требуется для запуска драйвера (и зачем)?**

1. **INF-файл**
  - Установка драйвера
  - Указывает имя `.sys` файла, параметры, секции реестра, hardware ID
2. **Запись в реестре**
  - `HKLM\SYSTEM\CurrentControlSet\Services\{DriverName}`
  - Хранит параметры запуска, тип (`boot/start/system`), путь к файлу и т.д.
  -

### 3. **.sys файл драйвера**

- Содержит исполняемый код драйвера (формат PE, как .exe)
- Загружается в kernel mode при старте устройства или системы

### 4. **Подпись (цифровая)**

- Начиная с Windows 10 — обязательна для драйверов в kernel mode
- Обеспечивает защиту от поддельных и вредоносных драйверов

## **Какие дополнительные возможности может включать драйвер?**

- **Фильтрация** (например, антивирус фильтрует доступ к файлам)
- **Обработка событий Plug and Play**
- **Обработка энергосбережения (Power IRP)**
- **Работа с буферами I/O** (Direct I/O, Buffered I/O, Neither)
- **Поддержка асинхронных операций (IRP в очереди)**
- **Создание виртуальных устройств**
- **Поддержка сетевых протоколов (NDIS)**
- **Интерфейсы для user-mode взаимодействия (IOCTL)**

## **Что такое объекты драйвера и файла?**

ОС Windows — объектно-ориентированная на уровне ядра.

Она использует **объекты ядра** (DRIVER\_OBJECT, DEVICE\_OBJECT, FILE\_OBJECT) для управления драйверами и I/O.

### **DRIVER\_OBJECT**

- Представляет **драйвер как сущность**
- Содержит указатели на функции (IRP\_MJ\_\*)
- Создаётся системой при загрузке драйвера
- Связан с одним или несколькими DEVICE\_OBJECT

### **DEVICE\_OBJECT**

- Представляет **одно логическое устройство**, которым управляет драйвер
- Содержит:

- имя устройства (\Device\MyDisk)
- очередь IRP
- ссылку на DRIVER\_OBJECT
- Может быть создан вручную драйвером (например, виртуальные устройства)

## FILE\_OBJECT

- Представляет **отдельный открытый дескриптор** к DEVICE\_OBJECT
- Создаётся, когда процесс вызывает CreateFile("\\.\MyDevice", ...)
- Содержит:
  - позицию чтения
  - режим доступа (read/write/share)
  - ссылку на DEVICE\_OBJECT

Аналогия: DEVICE\_OBJECT — это как "файл", а FILE\_OBJECT — это конкретное "открытие" этого файла/устройства.

***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?***

### Что такое IRP (I/O Request Packet)?

**IRP (Пакет запроса на ввод/вывод)** — это структура ядра Windows, с помощью которой I/O Manager передаёт запрос драйверам.

IRP инкапсулирует:

- Тип операции (IRP\_MJ\_READ, IRP\_MJ\_WRITE, IRP\_MJ\_DEVICE\_CONTROL и т.д.)
- Буферы данных
- Ссылку на файловый/устройственный объект

- Параметры I/O
- Указатель на стек вызовов (стек драйверов)

IRP позволяет маршрутизировать один запрос через **стек драйверов**, например: фильтр → файловая система → драйвер диска.

#### Виды IRP (по основным Major Function Codes):

Код	Назначение
IRP_MJ_CREATE	Открытие устройства или файла (CreateFile)
IRP_MJ_CLOSE	Закрытие дескриптора
IRP_MJ_READ	Чтение из устройства
IRP_MJ_WRITE	Запись в устройство
IRP_MJ_DEVICE_CONTROL	Управляющие команды (IOCTL)
IRP_MJ_PNP	PnP-уведомления: подключение/отключение устройств
IRP_MJ_POWER	Управление питанием
IRP_MJ_SYSTEM_CONTROL	Используется WMI и инструментами мониторинга
IRP_MJ_INTERNAL_DEVICE_CONTROL	Междрайверные команды
IRP_MJ_CLEANUP	Завершение всех операций по дескриптору

Каждый IRP передаётся драйверу, у которого в DRIVER\_OBJECT->MajorFunction[N] зарегистрирована функция-обработчик для данного типа.

## Что такое Plug and Play (PnP)?

**PnP** — это технология ОС, позволяющая автоматически **обнаруживать, настраивать, и использовать устройства** без участия пользователя.

### Возможности, предоставляемые PnP-поддержкой:

- **Автоматическое обнаружение** новых устройств
- **Автоматическая установка драйвера** (через INF и реестр)
- **Назначение ресурсов** (IRQ, DMA, I/O-порты)
- **Горячее подключение и извлечение** (USB, PCI, Bluetooth и т.д.)
- **Обновление или удаление драйверов**
- **Динамическая смена конфигурации** оборудования без перезагрузки

### Из чего состоит система PnP (в Windows):

Компонент	Роль
<b>PnP Manager</b>	Координирует все операции Plug and Play
<b>Device Manager</b>	GUI-интерфейс управления устройствами
<b>INF Installer</b>	Обрабатывает .inf при установке
<b>Service Control Manager (SCM)</b>	Загружает драйверы как службы
<b>Класс-драйверы и miniport-драйверы</b> Специализированные драйверы устройств	
<b>Реестр</b>	Хранит параметры устройств и драйверов
<b>IRP_MJ_PNP</b>	Механизм передачи PnP-команд драйверам

**С чем работает PnP:**

- Железо: USB, PCI, Bluetooth, ACPI, HDD, GPU и т.д.
- Виртуальные устройства
- Фильтрующие драйверы
- Модули управления электропитанием

**Условия, которые должен выполнить драйвер для полной поддержки PnP:**

1. **Обрабатывать IRP\_MJ\_PNP**
  - Поддержка MajorFunction для PnP-запросов (IRP\_MN\_START\_DEVICE, IRP\_MN\_REMOVE\_DEVICE, IRP\_MN\_QUERY\_STOP\_DEVICE и т.д.)
2. **Создавать и регистрировать DEVICE\_OBJECT**
  - Поддержка AddDevice() (вызывается PnP Manager при обнаружении устройства)
3. **Реагировать на события управления устройством**
  - Корректно освобождать ресурсы (память, порты, IRQ) при IRP\_MN\_REMOVE\_DEVICE
  - Принимать/отклонять запросы QUERY\_STOP / SURPRISE\_REMOVE
4. **Уметь обрабатывать отложенные старты (delayed starts)**
  - Применимо для устройств, которые требуют времени для инициализации
5. **Поддерживать энергосберегающие состояния**
  - Обрабатывать IRP\_MJ\_POWER и работать совместно с Power Manager
6. **Работать с реестром**
  - Использовать IoOpenDeviceRegistryKey() для чтения/записи конфигурации

**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?**



## Что такое исключение?

**Исключение** – это событие, возникающее из-за выполнения определённой команды, которая вызвала ошибку процессора. Это событие, которого вы не ожидали. В хорошо написанной программе не предполагается попыток обращения по неверному адресу или деления на нуль, и всё же такие ошибки случаются.

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)**.

## Сравните их с прерываниями.

Исключения в некотором роде **похожи на прерывания**. Однако, **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**.

## Что такое Structured Exception Handling (SEH)?

**Structured Exception Handling (SEH)** – это механизм, который предоставляет операционная система Windows для обработки исключений. Когда возникает исключение, ядро перехватывает его и позволяет коду обработать исключение, если это возможно. Этот механизм доступен как для кода пользовательского режима, так и для кода режима ядра. Стоит отметить, что SEH является частью исключительно операционной системы Windows, а полная поддержка SEH присутствует только в компиляторе MSVC.

Хотя всю работу по отлову исключений берёт на себя операционная система, **основная нагрузка по поддержке SEH ложится на компилятор**, а не на операционную систему. Компилятор генерирует специальный код на входах и выходах блоков исключений, создаёт таблицы вспомогательных структур данных для поддержки SEH и предоставляет функции обратного вызова, к которым система могла бы обращаться для прохода по блокам исключений. Компилятор также отвечает за формирование стековых фреймов и другой внутренней информации, используемой операционной системой.

## Что такое блок исключения?

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

## Какие основные возможности предоставляет SEH?

SEH предоставляет две **основные возможности**:

- **Обработка завершения (termination handling).**
- **Обработка исключений (exception handling).**

Не следует путать SEH с обработкой исключений в C++, которая представляет собой другую форму обработки исключений, построенную на применении ключевых слов языка C++ `catch` и `throw`. Однако Microsoft Visual C++ использует преимущества поддержки SEH, уже обеспеченной компилятором и операционными системами Windows. Компилятор MSVC реализует обработку исключений C++ на основе SEH операционной системы: C++-блок `try` становится SEH-блоком `__try`, C++-блок `catch` становится SEH-фильтром исключений, а код блока `catch` – кодом SEH-блока `__except`. При обработке C++-оператора `throw` компилятор генерирует вызов Windows-функции `RaiseException`.

## Что такое защищённый блок?

**Защищённый или охраняемый блок кода** – это блок кода, ограниченный фигурными скобками оператора `__try`. Предполагается, что в этом блоке может возникнуть исключение.

## Поясните принципы работы обработчика завершения.

**Обработчик завершения (`__finally`)** – это ключевое слово, используемое в MSVC для работы с SEH. Он **гарантирует, что блок кода (собственно обработчик) будет выполнен независимо от того, как происходит выход из другого блока кода – защищённого участка программы**. Совместные действия операционной системы и компилятора гарантируют, что код блока `__finally` будет выполнен независимо от того, как произойдёт выход из защищённого блока `__try`, даже если в нём размещены операторы `return` или `goto`.

Принцип работы заключается в следующем:

- Если в защищённом блоке `__try` встречается оператор `return` (или `goto`, `continue`, `break`), который пытается осуществить преждевременный выход из блока, компилятор проверяет, чтобы сначала был выполнен код в блоке `__finally`.
- **Компилятор генерирует дополнительный код**, который сохраняет возвращаемое значение (если есть) во временной переменной, а затем выполняет инструкции из блока `__finally`. Это называется **локальной раскруткой**.
- После выполнения блока `__finally`, функция фактически завершает работу.
- Например, если в `__try` есть `ReleaseSemaphore` и функция пытается выйти через `return`, обработчик завершения в `__finally` гарантирует, что семафор будет освобождён, предотвращая блокировку других потоков. Это делает код более надёжным, так как очистка ресурсов происходит гарантированно.

Для более изящного и оптимизированного выхода из блока `__try` и перехода к `__finally` можно использовать ключевое слово `__leave`. `__leave` вызывает переход в конец блока `__try`, после чего выход из `__try` и вход в `__finally` происходит естественным образом, не вызывая локальной раскрутки. При этом может потребоваться дополнительная булева переменная для отслеживания успешности завершения функции.

При разработке функций с обработчиками завершения рекомендуется инициализировать все дескрипторы ресурсов недопустимыми значениями перед входом в блок `__try`. Затем в блоке

`__finally` можно проверять, какие ресурсы были успешно выделены, и освобождать только их. Альтернативный метод — установка флага при успешном выделении ресурса, который затем проверяется в `__finally`.

Существует три сценария, которые приводят к выполнению блока `__finally`:

1. **Нормальная передача управления** от блока `__try` блоку `__finally`.
2. **Локальная раскрутка** — преждевременный выход из блока `__try` (из-за операторов `goto`, `continue`, `break`, `return` и т.д.), вызывающий принудительную передачу управления блоку `__finally`.
3. **Глобальная раскрутка (global unwind)** — происходит, когда исключение возникает в функции, вызванной из блока `__try`, и приводит к выполнению блока `__finally`.

Для определения, какой из этих сценариев вызвал выполнение блока `__finally`, можно использовать встраиваемую функцию **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.

## Что такое локальная раскрутка?

**Локальная раскрутка (local unwind)** — это процесс, который происходит, когда система выполняет блок `__finally` из-за **преждевременного выхода из блока `__try`**. По сути, это процесс освобождения локальных объектов. При этом компилятору приходится генерировать дополнительный код, а системе — выполнять дополнительную работу.

## 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` выполняется 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?

Применять обработчики завершения следует по следующим причинам:

- 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`.
- Удаётся 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?***

Структурная обработка исключений (SEH) — это механизм операционной системы Windows, который позволяет обрабатывать сбои, возникающие в программах. Он доступен как для кода пользовательского режима, так и для кода режима ядра.

- **Что такое исключение? Исключение** — это событие, которое возникает из-за выполнения определенной команды, вызвавшей ошибку процессора. Чаще всего в результате такого события нормальное выполнение программы становится невозможным. Исключения в некотором роде похожи на прерывания, но 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. Примеры исключений включают деление на ноль, точку останова или ошибку доступа к памяти.
- **Что такое аппаратное и программное исключения?**
  - **Аппаратное исключение (hardware exception)** — это исключение, возбужденное центральным процессором в ответ на ошибки, такие как попытки обращения по неверному адресу или деления на ноль.
  - **Программное исключение (software exception)** — это исключение, которое может быть возбуждено операционной системой или прикладными программами. Приложения могут генерировать собственные программные исключения по ряду причин, например, для отправки информационных сообщений в системный журнал событий или для уведомления о внутренних фатальных ошибках.

- **Что такое защищённый блок? Защищенный (или охраняемый) блок кода** — это блок кода, ограниченный фигурными скобками оператора `__try`. Предполагается, что в этом блоке может возникнуть исключение.
- **Поясните принципы работы обработчика исключений.** При возникновении аппаратного или программного исключения операционная система предоставляет приложению возможность определить его тип и самостоятельно обработать. Синтаксис обработчика исключений включает ключевое слово `__except`, которое должно следовать за блоком `__try`. 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.
- **Что такое фильтры? Фильтр исключений** — это выражение, указанное в блоке `__except`, которое система проверяет при возникновении исключения. Фильтр отвечает за анализ ситуации и возврат одного из трех стандартных идентификаторов, определенных в заголовочном Windows-файле `Except.h`. 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.
- **Какие есть стандартные фильтры и как они работают?** Существует три стандартных фильтра исключений:
  - **EXCEPTION\_EXECUTE\_HANDLER:** Это значение сообщает системе, что исключение обработано, и система должна передать управление коду внутри блока `__except` (коду обработчика исключений). После выполнения кода в блоке `__except` система считает исключение обработанным и разрешает программе продолжить работу с инструкции, следующей за блоком `__except`. При возврате этого значения система проводит **глобальную раскрутку**.
  - **EXCEPTION\_CONTINUE\_EXECUTION:** Обнаружив такое значение, система возвращается к инструкции, вызвавшей исключение, и пытается выполнить ее снова.
  - **EXCEPTION\_CONTINUE\_SEARCH:** Этот идентификатор указывает системе перейти к предыдущему блоку `__try`, которому соответствует блок `__except`, и обработать его фильтр. Это означает, что система пропускает при просмотре цепочки блоков любые блоки `__try`, которым соответствуют блоки `__finally`, так как в них нет фильтров исключений.

- **Что такое глобальная раскрутка? Глобальная раскрутка (global unwind)** — это сценарий выполнения блока `__finally`, который происходит, когда исключение возникает в функции, вызванной из блока `__try`. Например, если в функции `Funcinator`, вызванной из блока `__try` в `Funcfurther1`, возникает нарушение доступа к памяти, глобальная раскрутка приводит к выполнению блока `__finally` в `Funcfurther1`. Глобальная раскрутка также происходит, когда фильтр исключений возвращает `EXCEPTION_EXECUTE_HANDLER`. Она приводит к продолжению обработки всех незавершенных блоков `try-finally`, выполнение которых началось вслед за блоком `try-except`, обрабатывающим данное исключение. Выполнение глобальной раскрутки может быть остановлено, если в блок `__finally` включить оператор `return`.
- **Как ВОЗБУДИТЬ исключения в SEH?** Вы можете сами генерировать программные исключения, вызывая функцию **`RaiseException`**.
  - Ее первый параметр, `dwExceptionCode`, — это значение, которое идентифицирует генерируемое исключение.
  - Второй параметр, `dwExceptionFlags`, должен быть либо 0, либо `EXCEPTION_NONCONTINUABLE`, указывая, может ли фильтр исключений вернуть `EXCEPTION_CONTINUE_EXECUTION` в ответ на данное исключение.
  - Третий и четвертый параметры (`nNumberOfArguments` и `pArguments`) позволяют передать дополнительные данные об исключении. Вызов `RaiseException` позволяет функциям сообщать о неудаче вызвавшему ее коду, что упрощает написание и сопровождение кода, а также повышает производительность, так как код контроля ошибок активируется только в исключительных ситуациях.
- **Что такое необработанное исключение? 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 операционной системой, `UnhandledExceptionFilter`, которая выводит окно, указывающее на то, что поток в процессе вызвал необрабатываемое исключение, и предлагает либо закрыть процесс, либо начать его отладку. Стандартное поведение `UnhandledExceptionFilter` может быть изменено вызовом функции `SetUnhandledExceptionFilter`, которая позволяет установить собственный фильтр для необработанных исключений.

**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.**

Безопасное программирование – это **подход к разработке программного обеспечения, направленный на предотвращение, обнаружение и реагирование на угрозы безопасности.** В контексте системного программирования, оно включает разработку и управление низкоуровневыми компонентами, такими как операционные системы, драйверы устройств и компиляторы, с учетом принципов безопасности.

**Основная цель безопасного программирования – защитить данные, системы и пользователей** от несанкционированного доступа, модификации или уничтожения. Системное программирование критически важно для безопасности, так как уязвимости в низкоуровневых компонентах (например, операционных системах и драйверах устройств) могут быть использованы для атак, влияя на всю компьютерную систему.

**Уязвимость – это недостаток программы, который может быть использован для реализации угроз безопасности информации.** Уязвимость может быть результатом разработки программы без учета требований безопасности или наличия ошибок проектирования/реализации. Обычно она позволяет атакующему «обмануть» приложение, заставив его выполнить непредусмотренные создателем действия.

**Недостаток программы – это любая ошибка, допущенная в ходе проектирования или реализации программы,** которая, если ее не исправить, может стать причиной уязвимости программы.

## **Классификация уязвимостей ПО**

Классификация уязвимостей помогает разработчикам и специалистам по кибербезопасности понимать и управлять рисками. Существует несколько популярных классификаторов уязвимостей программного обеспечения:

- CVE (Common Vulnerabilities and Exposures)
- CWE (Common Weakness Enumeration)
- SecurityFocus BID
- OSVDB (Open Sourced Vulnerability Database)
- Secunia
- IBM ISS X-Force

## Классификация уязвимостей:

**Архитектурные** — ошибки в логике или дизайне (отсутствие аутентификации, неправильное распределение прав)

**Имплементационные (кодовые)** — ошибки в реализации (переполнение буфера, use-after-free, неверная арифметика)

**Конфигурационные** — неправильные параметры системы (лишние порты, слабые пароли, небезопасные политики)

## Категории ошибок ПО

Написать безопасный код очень сложно, поскольку небольшие ошибки могут привести к серьезным уязвимостям. Существуют две основные категории ошибок программного обеспечения:

- **Проблемы проектирования:** например, когда программист не продумал необходимый тип аутентификации.
- **Проблемы с реализацией:** например, случайное использование небезопасного библиотечного метода или попытка сохранить слишком много данных в переменной.

## Список распространенных ошибок ПО

Часто встречающиеся уязвимости, которые легко допустить, но последствия которых могут быть катастрофическими, включают:

- Внедрение SQL-кода (SQL injection)
- Уязвимости, связанные с web-серверами (XSS, XSRF, расщепление HTTP запроса)
- Уязвимости web-клиентов (DOM XSS)
- Переполнение буфера (Buffer Overflow)
- Дефекты форматных строк (Uncontrolled format string)
- Целочисленные переполнения (Integer overflow)
- Некорректная обработка исключений и ошибок
- Внедрение команд (Command injection)
- Утечка информации (Information Exposure)
- Ситуация гонки (Race condition)
- Слабое юзабилити (Insufficient Psychological Acceptability)
- Выполнение кода с завышенными привилегиями (Execution with Unnecessary Privileges)
- Хранение незащищенных данных (Protection Mechanism Failure)
- Слабые пароли
- Слабые случайные числа
- Неудачный выбор криптографических алгоритмов
- Использование небезопасных криптографических решений
- Незащищенный сетевой трафик (Cleartext Transmission of Sensitive Information)
- Неправильное использование PKI (Improper Certificate Validation)
- Доверие к механизму разрешения сетевых имен (Reliance on Reverse DNS Resolution)





- **Статический и динамический анализ кода:** Использование инструментов, таких как Clang Static Analyzer, Valgrind и другие, которые могут обнаруживать потенциальные переполнения.
- **Обучение и осведомленность:** Обучение разработчиков методам безопасного программирования и повышение осведомленности о рисках, связанных с целочисленными переполнениями.

**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?**

Безопасное программирование – это подход к разработке программного обеспечения, направленный на предотвращение, обнаружение и реагирование на угрозы безопасности. Его основная цель – защитить данные, системы и пользователей от несанкционированного доступа, модификации или уничтожения. В контексте системного программирования, безопасное программирование включает в себя разработку и управление низкоуровневыми компонентами, такими как операционные системы, драйверы устройств и компиляторы, с учетом принципов безопасности. Уязвимости в этих низкоуровневых компонентах могут быть использованы для атак, что делает безопасное программирование особенно важным.

### **Ошибка форматирования строк и как её избежать**

**Ошибка форматирования строк** возникает, когда функции, подобные `printf()`, получают строку формата, за которой следуют несколько переменных для отображения. Например, `printf("Hello %s, you entered %d", string, number);` корректно подставляет значения из стека в соответствии с форматными спецификаторами.

**Уязвимость** возникает, когда ленивый или неопытный программист передает пользовательский ввод напрямую в качестве строки формата, например, `printf(string);` вместо `printf("%s", string);`. В этом случае злоумышленник может использовать хитроумные уловки (например, вводя спецификаторы формата `%x` или `%d`) для **просмотра содержимого стека или даже записи в память**, что может привести к серьезным уязвимостям в системе безопасности. Примером является ввод строки "AAAA %x %x ... %x" (где `%x` повторяется 31 раз), которая заставляет `printf()` проходить вниз по стеку, считывая и печатая значения в шестнадцатеричном виде, включая такие данные, как `secret_number`.

**Чтобы избежать ошибки форматирования строк**, все данные, поступающие из ненадежного источника, должны быть **канонизированы (Canonicalization)**, **проверены (Validation)** и **обработаны (Sanitization)** перед использованием.

**Что такое канонизация, валидация и очистка?**

Эти три понятия часто используются вместе, хотя они имеют разные значения:

- **Канонизация (Canonicalization)** – это процесс преобразования данных, имеющих более одного возможного представления, в «стандартную», «нормальную» или **каноническую форму**. Например, приведение разных форматов даты ("30 января 2018", "30-1-2018", "2018/01/30") к единому формату (например, ММ-ДД-ГГГГ).
- **Валидация (Validation)** – это проверка того, что данные представлены в том формате, который вы ожидаете, и что они являются **допустимыми входными данными**. Валидация включает в себя проверку успешности канонизации и нахождение окончательной даты в пределах ожидаемого диапазона. Обычно валидация является **более безопасным подходом** по сравнению с очисткой, так как она отклоняет недопустимые данные, а не пытается их "исправить".
- **Очистка (Sanitization)** – это процесс удаления любого потенциально опасного форматирования или содержимого из переменной, чтобы "исправить" вводимые данные и сделать их безопасными. Например, распознавание и удаление `<script>alert(1)</script>` из поля даты.

## Что такое триада CIA?

**Триада CIA** (Confidentiality, Integrity, Availability) – это основа безопасности операционных систем и компьютерных систем в целом. Она включает:

- **Конфиденциальность (Confidentiality)**: неавторизованные пользователи не могут получить доступ к данным.
- **Целостность (Integrity)**: неавторизованные пользователи не могут изменять данные.
- **Доступность (Availability)**: система остается доступной для авторизованных пользователей даже в случае атаки типа "отказ в обслуживании".

## Какие ещё есть способы повышения безопасности в рамках ОС?

Помимо достижения триады CIA, существуют другие способы повышения безопасности операционных систем:

- **Простота**: минимизация поверхности атаки.
- **Блокировка доступа к ресурсам по умолчанию**.
- **Проверка всех запросов на авторизацию**.
- **Принцип наименьших полномочий (Principle of Least Privilege)**.
- **Цепочки доверия (chains of trust)**.
- **Разделение привилегий**.
- **Сокращение общих данных**.
- **Повышение надежности операционной системы** для уменьшения уязвимостей, например, с использованием:
  - **Рандомизации расположения адресного пространства (ASLR)**.
  - **Целостности потока управления (Control Flow Integrity)**.
  - **Ограничений доступа**.

## Что такое ASLR?

**ASLR (address space layout randomization – «рандомизация размещения адресного пространства»)** – это технология, применяемая в операционных системах, при использовании которой **случайным образом изменяется расположение в адресном пространстве процесса важных структур данных**, таких как образы исполняемого файла, подгружаемые библиотеки, куча и стек.

Технология ASLR создана для **усложнения эксплуатации нескольких типов уязвимостей**. Например, если атакующий получает возможность передать управление по произвольному адресу (при помощи переполнения буфера или другим методом), ему нужно будет угадать, по какому именно адресу расположен стек, куча или другие структуры данных, куда можно поместить шелл-код.

## Что такое DEP?

**DEP (Data Execution Prevention – Предотвращение выполнения данных)** – это функция безопасности, встроенная в различные операционные системы, которая **не позволяет приложению исполнять код из области памяти, помеченной как «только для данных»**. Она помогает предотвратить некоторые атаки, которые, например, сохраняют код в такой области с помощью переполнения буфера.

## Что такое PoLP и какие аспекты лежат в его основе?

**Принцип наименьших полномочий (Principle of Least Privilege, PoLP)** – это фундаментальный принцип информационной безопасности, который гласит, что **каждый субъект (пользователь, процесс, система) должен иметь только те минимальные права и доступы, которые необходимы для выполнения его задач**. Этот принцип помогает минимизировать риски, связанные с несанкционированным доступом, и уменьшить потенциальный ущерб в случае компрометации.

**Основные аспекты принципа наименьших полномочий** включают:

- **Минимизация прав доступа:** Субъект имеет только необходимые права для выполнения задач.
- **Разделение обязанностей:** Разделение задач и обязанностей между разными субъектами для предотвращения концентрации полномочий.
- **Контроль доступа на основе ролей (RBAC):** Назначение прав доступа на основе ролей пользователей, а не индивидуальных учетных записей.
- **Минимизация времени доступа:** Предоставление доступа только на время, необходимое для выполнения задачи.

## Какие есть лучшие практики в области безопасного программирования?

Лучшие практики при разработке программного обеспечения включают:

- **Информируйте себя:**
  - Следите за обсуждениями уязвимостей на открытых форумах, изучайте примеры в исходном коде.

- Читайте книги и статьи по методам безопасного программирования и анализу недостатков ПО.
- Изучайте программное обеспечение с открытым исходным кодом, но будьте осторожны, так как там могут быть и примеры уязвимостей.
- **Обращайтесь с данными с осторожностью:**
  - **Очистка данных:** проверяйте предлагаемые входные данные на наличие злого умысла.
  - **Выполняйте проверку границ:** убедитесь, что предоставленные данные могут поместиться в отведенное пространство; проверяйте индексы массива.
  - **Проверяйте конфигурационные файлы:** всегда предполагайте, что данные могли быть изменены злоумышленником.
  - **Проверяйте параметры командной строки:** они легко поддаются обману.
  - **Проверяйте переменные среды:** злоумышленники могут использовать их для изменения поведения программы.
  - **Проверяйте другие источники данных:** будьте особенно осторожны с информацией, которая не была явно упомянута.
  - **Будьте осторожны с косвенными ссылками на файлы:** злоумышленник может обманом заставить программу прочитать или записать файл, к которому она не должна иметь доступа.
- **Повторно используйте «хороший код»,** когда это возможно.
- **Не пишите код, который использует относительные имена файлов:** Ссылки на имена файлов должны быть «полными» для предотвращения несанкционированного раскрытия или изменения информации.
- **Не ссылайтесь на файл дважды в одной и той же программе по его имени:** Откройте файл один раз и используйте дескриптор файла; злоумышленник может изменить файл между двумя ссылками.
- **Не вызывайте ненадежные программы из надежных:** Если ваше ПО работает в привилегированном режиме, лучше выполнить работу самостоятельно, чем делегировать её другой программе, так как вы не можете быть уверены в её действиях.
- **Не думайте, что ваши пользователи не являются злоумышленниками:** Всегда перепроверяйте каждую часть внешней информации, предполагайте, что она содержит вредоносные намерения, пока не будет проверена.
- **Не рассчитывайте на успех:** Всегда проверяйте условия завершения системного вызова (например, открытие/чтение файла, извлечение переменной среды) и корректно обрабатывайте неудачи.
- **Не вызывайте оболочку или командную строку.**
- **Не выполняйте проверку подлинности по ненадежным критериям:** Программисты часто делают ошибочные предположения о личности пользователя или процесса.
- **Не используйте хранилище, доступное для глобальной записи, даже временно:** Если это абсолютно необходимо, предполагайте, что информация может быть подделана, изменена или уничтожена.
- **Не доверяйте хранилищу, доступному для записи пользователем, чтобы избежать несанкционированного доступа:** Никогда не доверяйте записываемым пользователем данным.

**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?**

Управление доступом является ключевым элементом систем информационной безопасности.

- **Контроль доступа к ресурсам** — это механизм, который разрешает доступ к ресурсу только авторизованным пользователям этого ресурса, то есть тем, кому разрешен доступ. Часто этот механизм также включает **аутентификацию пользователя**, что означает установление подлинности пользователя.
- В системах информационной безопасности все ресурсы делятся на две категории: **объекты и субъекты**.
  - **Объекты** представляют собой пассивные ресурсы, которые подлежат защите. Примеры объектов включают файлы, каналы передачи данных, оперативную память, принтеры и другие внешние устройства для хранения и отображения информации. Для каждого объекта определяется множество операций, которые можно над ним выполнить.
  - **Субъекты** представляют собой активные ресурсы, которые выполняют операции над объектами и представляются процессами, исполняемыми от имени пользователей информационной системы. Выполнение субъектами операций над объектами называется **доступом субъектов к объектам**. Для контроля доступа субъектов к объектам для каждой пары (субъект, объект) определяется множество операций, которые данный субъект может выполнять над данным объектом.
- **Политика безопасности и менеджер безопасности:**
  - **Политика безопасности** — это набор требований, выполнение которых обеспечивает безопасную работу информационной системы. Это общее понятие, используемое не только в информационных системах, но и в системах безопасности любых организаций.
  - **Менеджер или монитор безопасности** — это субъект в системе информационной безопасности, который контролирует доступ других субъектов к объектам, руководствуясь при этом определенными правилами (политикой безопасности). Предполагается, что менеджер безопасности всегда находится в активном состоянии. Важно **отличать менеджера безопасности** (программу) **от администратора системы информационной безопасности** (человека). Администратор занимается регистрацией пользователей, наделением их правами и привилегиями, а также отслеживанием работы системы безопасности.
- **Разница между правами и привилегиями:**
  - **Право** — это возможность субъекта выполнять некоторые операции над объектами. Например, пользователь имеет право читать определенный файл.
  - **Привилегия** — это более общее понятие, которое позволяет пользователю выполнять действия в отношении других объектов и субъектов системы

информационной безопасности. Как правило, привилегия выделяет пользователя из числа обычных пользователей системы, например, дает возможность блокировать доступ к системе другим пользователям.

- **Порядок разработки политики безопасности** обычно разбивается на два этапа:
  - **Анализ угроз для системы информационной безопасности** (также называемый **анализом рисков**). На этом этапе подробно изучаются следующие вопросы:
    - Определяются информационные ресурсы, которые должны быть защищены.
    - Определяются возможные угрозы этим информационным ресурсам, рассматриваемые как с точки зрения внутренних угроз (от программного обеспечения), так и внешних угроз (от пользователей).
  - **Разработка средств защиты от возможных угроз безопасности** (также называемая **управлением рисками**). Средства защиты должны предусматривать работу в трех режимах:
    - Обработка рутинных задач информационной безопасности.
    - Обработка исключительных ситуаций, например, обнаружение атаки вируса.
    - Обработка аварийных ситуаций, таких как обнаружение вируса в системе.
- **Модель безопасности** — это формальное описание разработанной политики безопасности. Часто понятия политики и модели безопасности не различают. Модель безопасности рассматривается как система, включающая следующие компоненты:
  - Пассивные ресурсы (объекты).
  - Наборы операций, которые можно выполнять над каждым объектом.
  - Активные ресурсы, выполняющие операции над объектами (субъекты).
  - Атрибуты защиты объектов, описывающие права доступа.
- **Состояние системы безопасности и её общая задача:**
  - **Состояние системы безопасности** определяется состоянием объектов, наборов операций, субъектов и атрибутов защиты объектов. Состояние системы безопасности называется **безопасным**, если в этом состоянии нет несанкционированного доступа.
  - **Общая задача системы безопасности** состоит в том, чтобы обеспечить переход из безопасного состояния в безопасное состояние. Это включает следующие принципы:
    - Начальное состояние системы должно быть безопасным.
    - Правила управления доступом должны обеспечивать переход из безопасного состояния системы в безопасное состояние системы.
    - Любое состояние системы, достижимое из ее начального состояния, должно быть безопасным.

***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.***

Управление доступом является фундаментальной частью систем информационной безопасности.

## **Дискреционная политика безопасности и её суть**

Дискреционная политика безопасности основывается на следующих принципах:

- Для каждого объекта определяется набор операций, которые можно над ним выполнять.
- Субъект может выполнить операцию над объектом при условии, если он имеет право на выполнение этой операции.
- Субъект, который имеет права на выполнение некоторых операций над объектом, может передать эти права другому субъекту.

Суть дискреционной политики безопасности заключается в том, что **права доступа субъекта к объекту определяются другим субъектом, который имеет эти права**. Иными словами, доступ субъектов к объектам разрешается другими субъектами, или это "оставлено на усмотрение другим субъектам, которые обладают этими правами". Менеджер безопасности контролирует доступ субъекта к объекту, проверяя, есть ли у субъекта разрешение на выполнение затребованной операции; если есть, то операция разрешается, в противном случае — запрещается.

Существует проблема в первоначальном варианте дискреционной политики безопасности: субъект может передать права на доступ к объекту другому субъекту без контроля менеджера безопасности. Такая политика называется **либеральной дискреционной политикой безопасности** и является самой ненадежной. Чтобы решить эту проблему, принцип передачи прав изменяют: субъект может передать права на выполнение операций над объектом другому субъекту **только при условии, если он наделен такими полномочиями**. Полученная политика безопасности называется **строгой дискреционной политикой безопасности**.

## **Порядок построения дискреционной модели безопасности**

Для построения дискреционной модели безопасности поступают следующим образом:

- **Идентифицируются все объекты и все субъекты системы.**
- Для каждого объекта **определяются операции**, которые субъекты могут выполнять над этим объектом.
- После этого **строится матрица**, где каждая строка соответствует одному субъекту, а каждый столбец — одному объекту.
- В клетки полученной матрицы **записывают права**, которые субъект (соответствующий строке) имеет по отношению к объекту (соответствующему столбцу).

## **Матрица управления доступом**



**Матрица управления доступом** — это структура, используемая в дискреционной модели безопасности, где каждая строка соответствует субъекту, а каждый столбец — объекту. Клетка на пересечении строки субъекта и столбца объекта содержит информацию о правах, которые этот субъект имеет по отношению к данному объекту. Менеджер безопасности использует эту матрицу для проверки запросов субъектов на доступ к объектам. На практике матрица управления доступами часто бывает разреженной и имеет большой размер, поэтому она редко хранится непосредственно в памяти компьютера.

## Режимы доступа к объекту

**Режимы доступа к объекту** (или операции, которые разрешается выполнять над объектами) — это права, которые субъект может иметь по отношению к объекту. В контексте объектов, подобных файлам, к таким режимам относятся:

- **READ (R)**: Разрешается чтение содержимого объекта, а также, как правило, копирование файла.
- **WRITE (W)**: Разрешается любая модификация файла.
- **WRITE\_APPEND (WA)**: Разрешается только добавление данных в объект.
- **WRITE\_CHANGE (WC)**: Разрешается только изменять или удалять данные из объекта, но не разрешается добавлять данные.
- **WRITE\_UPDATE (WU)**: Разрешается только изменять данные из объекта, но не разрешается добавлять или удалять данные.
- **DELETE (D)**: Разрешается удаление объекта.
- **EXECUTE (E)**: Разрешается исполнение объекта.
- **NULL (N)**: Нет доступа к объекту.

## Режимы управления объектами

**Режимы управления объектом** определяют субъектов, которые могут изменять права доступа других субъектов к объекту или передавать права управления этим объектом другим субъектам. Существуют два основных режима управления:

- **CONTROL (C)**: Разрешается устанавливать режимы доступа к объекту для субъектов, но не разрешается передавать режим управления другому субъекту.
- **CONTROL WITH PASSING ABILITY (CP)**: Разрешается устанавливать режимы доступа к объекту для субъектов и передавать режимы управления объектом другому субъекту.

## Модели управления в дискреционной модели безопасности

Набор правил, которым подчиняются субъекты при передаче прав управления, определяет **модель управления** в дискреционной модели безопасности. Существует четыре основные модели управления:

- **Иерархическое управление (hierarchical control)**:
  - Субъекты, управляющие правами доступа, упорядочиваются иерархически (древовидная структура).

- На вершине находится администратор системы безопасности, который может управлять правами доступа ко всем объектам и наделять ими любого субъекта.
- Остальные субъекты могут управлять правами доступа только субъектов, находящихся ниже их в иерархии.
- **Управление правами доступа владельцем объекта (concept of ownership):**
  - Управление правами доступа к объекту для всех субъектов выполняется именно **владельцем этого объекта**.
  - Владелец объекта, как правило, считается субъект, который его создал.
  - Администратор системы безопасности также присутствует и может ограничить права владельца на передачу прав доступа.
  - Фактически эта модель представляет собой двухуровневую иерархическую модель управления.
- **Либеральное управление (laissez-faire):**
  - Передача прав доступа к объекту от субъекта, обладающего этими правами, к другому субъекту **никак не контролируется**.
  - Это самая ненадежная модель управления.
  - "Laissez faire, laissez passer" переводится как "Пусть все идет своим чередом".
- **Централизованное управление (centralized control):**
  - **Правами управления доступом к объектам обладает только один субъект**, как правило, администратор системы.
  - Все вопросы по разрешению доступа субъектов к объектам системы решаются только администратором.
  - Эта модель является **наиболее надежной** из рассмотренных.

## Два подхода к хранению матрицы управления

Поскольку матрица управления доступом велика и разрежена, её редко хранят целиком. Для хранения матрицы управления доступом используют один из двух подходов:

1. **Построчный подход (capabilities):**
  - Матрица управления доступами рассматривается по строкам. Каждая строка описывает **все объекты, к которым данный субъект имеет доступ, и режимы доступа к ним**. Эта информация называется **возможностями (capabilities) субъекта**.
  - Для каждого пользователя администратор задает его возможности. Менеджер безопасности разрешает доступ к ресурсам только в соответствии с этими возможностями.
  - Возможности каждого пользователя хранятся в виде списка, называемого **профилем (profile) пользователя**. Каждый элемент списка содержит имена объектов и режимы доступа к ним. Субъект может открыть заданный режим доступа к объекту только в том случае, если этот объект и режим присутствуют в его профиле.
2. **Столбчатый подход (Access Control List, ACL):**
  - Матрица управления доступами рассматривается по столбцам. Каждый столбец описывает **всех субъектов, которые имеют доступ к данному объекту, и режимы доступа каждого субъекта**.
  - В этом случае столбец матрицы управления доступами хранится в виде списка, который называется **списком управления доступами (Access Control List,**

**ACL).** Элементы этих списков сокращенно называются **ACE (access control elements).**

- Каждый элемент списка управления доступами содержит имя пользователя и разрешенные ему режимы доступа к объекту.
- Список управления доступами объекта обычно создается владельцем объекта или администратором информационной системы.
- Менеджер безопасности открывает субъекту требуемый доступ к объекту только в том случае, если его имя и требуемый режим доступа находятся в списке управления доступами для этого объекта.
- В операционных системах Windows NT реализована дискреционная модель безопасности, где процессы и потоки являются активными субъектами. Каждый охраняемый объект в Windows имеет дескриптор безопасности, который содержит информацию для защиты, включая **список управления дискреционным доступом (Discretionary Access-Control List, DACL)** для хранения информации о разрешенном или запрещенном доступе пользователей и групп. Контроль доступа к объекту в Windows зависит от порядка расположения элементов в списке DACL. Важно различать **отсутствие списка DACL (Null DACL)**, когда доступ не ограничен, и **пустой список DACL (Empty DACL)**, когда доступ запрещен для всех.

***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?***

В операционных системах Windows NT в качестве активных субъектов модели безопасности рассматриваются процессы и потоки, каждый из которых работает от имени некоторого пользователя. Когда пользователь регистрируется и входит в систему, для него создается **маркер доступа (access token)**. Маркер доступа **идентифицирует этого пользователя и содержит его привилегии**. Каждый процесс, исполняемый от имени пользователя, имеет маркер доступа этого пользователя. Маркер доступа используется **для контроля доступа процесса к объектам**, которые называются в Windows охраняемыми объектами. Фактически доступ к объектам осуществляет поток, который выполняется в контексте этого процесса. Каждый поток также имеет свой маркер доступа, который может быть первичным (совпадать с маркером процесса) или заимствованным у другого процесса в случае подмены контекста безопасности (impersonation). При контроле доступа исполняемый поток представляется маркером доступа этого потока.

**Охраняемые объекты (securable objects)** — это объекты Windows, доступ к которым контролируется системой управления безопасностью. К охраняемым объектам относятся **все объекты Windows, которые могут иметь имя**. Кроме того, к охраняемым объектам относятся

также **потоки и процессы**. При контроле доступа охраняемый объект представляется **дескриптором безопасности этого объекта**.

**Дескриптор безопасности (security descriptor)** создается вместе с охраняемым объектом. Он содержит **информацию, необходимую для защиты объекта от несанкционированного доступа**.

В дескрипторе безопасности **идентифицируется владелец объекта**. Также в нем **определяются пользователи и группы пользователей, которым разрешен или запрещен доступ к охраняемому объекту**. Кроме того, он содержит **информацию для аудита доступа к объекту**.

Для хранения информации о пользователях, которым разрешен или запрещен доступ к охраняемым объектам, каждый дескриптор безопасности содержит **список управления дискреционным доступом (Discretionary Access-Control List, DACL)**. Для управления аудитом доступа к объекту в дескрипторе безопасности хранится **список управления системным доступом (System Access-Control List, SACL)**. Общее название для этих списков – **списки управления доступом (Access-Control Lists) или сокращенно ACL**.

При открытии доступа к охраняемому объекту система управления безопасностью просматривает список DACL этого объекта для поиска элемента, в котором хранится идентификатор безопасности субъекта. Если элемент не найден, доступ отклоняется. Если элемент найден и имеет тип ACCESS\_ALLOWED\_ACE, система проверяет запрошенные права; если они установлены, доступ разрешается, иначе - отклоняется. Если найденный элемент имеет тип ACCESS\_DENIED\_ACE, и хотя бы один из запрошенных флагов прав доступа установлен, доступ отклоняется; в противном случае просмотр списка DACL продолжается. Отсутствие списка DACL у объекта (Null DACL) означает, что доступ к объекту не ограничен и разрешен для любого потока. Пустой список DACL (Empty DACL) означает, что доступ к объекту запрещен для всех потоков.

Прежде чем пользователь сможет работать в среде операционных систем Windows NT, он должен быть зарегистрирован администратором системы. При регистрации пользователя администратором создается **учетная запись пользователя (user account)**. Учетная запись пользователя **хранится в базе данных менеджера учетных записей (Security Account Manager, SAM)**. В учетной записи пользователя хранится различная информация, включая имя пользователя для входа в систему, пароль, полное имя, допустимое время работы в системе, допустимые компьютеры для входа, дата окончания срока действия, рабочий каталог, сценарий входа, профиль пользователя и тип учетной записи.

Существуют четыре типа учетных записей:

- Учетная запись пользователя
- Учетная запись группы пользователей
- Учетная запись компьютера
- Учетная запись домена

По умолчанию Windows NT создает три учетные записи: Administrator (администратор), Guest (гость) и System (система). Учетная запись администратора предназначена для управления локальной системой и по умолчанию создается без пароля (рекомендуется изменить имя и

установить пароль). Учетную запись администратора системы нельзя заблокировать, и администратор может создавать новые учетные записи. Учетная запись гостя предназначена для обычного пользователя без административных полномочий, также создается без пароля, не может быть удалена, но может быть переименована. Учетная запись системы используется самой операционной системой для задач, требующих аутентификации, и имеет административные привилегии.

**Группа (group)** – это набор учетных записей пользователей, которые объединены по какому-либо признаку, например, пользователи одного отдела. Группы пользователей были введены для того, чтобы упростить управление безопасностью объектов. Одна учетная запись пользователя может входить более чем в одну группу. Каждая группа имеет свою учетную запись и наделена своими правами и полномочиями, которые передаются каждому члену группы. Администратор системы может изъять некоторые права и полномочия у отдельных членов группы. Максимальное количество групп, в которые может входить пользователь, равно 1000.

На платформе Windows NT различают три типа групп:

- Глобальные группы
- Локальные группы
- Специальные группы

Локальные и глобальные группы создаются администратором системы. Глобальная группа используется для организации пользователей с целью упорядочения их доступа к ресурсам вне домена, где создана группа. Локальная группа используется для организации доступа пользователей к ограниченному множеству ресурсов внутри домена. Специальные группы создаются системой по умолчанию для управления доступом к ресурсам, их членство предопределено и не может быть изменено.

Для каждой учетной записи операционная система создает **идентификатор безопасности (Security Identifier, SID)**. SID хранится в базе данных SAM. Идентификатор безопасности является **бинарным представлением учетной записи** и используется системой безопасности при своей работе для **идентификации учетных записей**. Фактически идентификатор безопасности идентифицирует пользователя на уровне системы безопасности. Использование SID ускоряет работу системы безопасности, так как она работает с числовыми данными.

Символически структура идентификатора безопасности может быть описана следующим образом: **S – R – I – SA0 – SA1 – SA2 – SA3 – SA4 ...**.

- **S** – представляет символ S, обозначающий, что дальнейшее числовое значение является идентификатором безопасности.
- **R** – представляет версию (Revision Level) формата идентификатора безопасности. Начиная с Windows NT версии 3.1, формат не изменялся, и значение R всегда равно 1.
- **I** – представляет 48-битное число, обозначающее уровень авторизации учетной записи (Top-level Authority или Identifier Authority). Это значение также называется идентификатором авторизации учетной записи. Предопределенные значения уровней авторизации учетных записей приведены в источнике.

- SA – представляет 32-битное число, уточняющее уровень авторизации учетной записи (Subauthority). Это число также называется относительным идентификатором учетной записи (Relative Identifier, RID).

В общем случае количество битовых полей типа SA в идентификаторе безопасности может быть произвольным, но не превышать 15. Для пользователей и групп идентификаторы безопасности имеют структуру S – R – I – SA0 – SA1 – SA2 – SA3 – SA4. В этом случае поле SA0 уточняет авторизацию учетной записи. Поля SA1, SA2 и SA3 представляют уникальный 96-битовый идентификатор компьютера. Поле SA4 нумерует идентификаторы безопасности, создаваемые внутри системы. Номера от 0 до 999 зарезервированы для системы, а номера, начиная с 1000, присваиваются новым идентификаторам безопасности пользователей или групп, при этом значение SA4 увеличивается на 1 при создании каждого нового SID. Идентификаторы безопасности предопределенных групп обычно имеют структуру S – R – I – SA0.

***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.***

**Перехват API-функций (API hooking)** — это программная техника, при которой вызовы функций из библиотеки API перенаправляются на пользовательские функции. Перехват API функций позволяет **модифицировать поведение программы**, добавлять новые функции или изменять параметры вызовов без необходимости изменения исходного кода программы.

Для компьютера любая программа представляет собой последовательный набор инструкций, часто объединенных в блоки, называемые функциями, процедурами или подпрограммами. Таким образом, программа — это набор инструкций и вызовов подпрограмм, упорядоченных для достижения определенного результата. Если программа простейшая, состоящая только из инструкций, известных процессору, то выполнение происходит по простому сценарию: загрузка следующей инструкции из памяти, декодирование, расчет эффективных адресов для данных и выполнение инструкции.

**Что такое функция?** Функции являются фундаментальной языковой возможностью для абстрагирования и повторного использования кода. Они позволяют ссылаться на некоторый фрагмент кода по имени. Для использования функции необходимо знать, сколько аргументов требуется, какого типа аргументы и что возвращает функция, а также что функция выполняет.

**Что происходит при вызове функции?** При вызове функции происходят следующие шаги:

- Аргументы должны быть преобразованы в значения (по крайней мере, для языков программирования, подобных C).
- Поток управления переходит к телу функции, и код начинает выполняться там.

- При встрече оператора `return`, работа с функцией завершается, и происходит возврат обратно к месту вызова функции.
- *Примечание: В вашем запросе было "вызове 4 функции". Если это не опечатка и имелось в виду что-то конкретное про четвертую функцию, уточните, пожалуйста. В контексте источников, это скорее всего относится к общему процессу вызова любой функции.*

**Что такое стек вызовов и каков принцип его работы?** Современные системы программирования организуют исполнение кода программ по модели распределения памяти на основе стека. Для хранения параметров (аргументов) процедур и функций, их локальных переменных, а также адреса возврата в программе выделяется специальная область памяти, организованная в виде стека. Этот стек выделяется отдельно для каждого потока и называется стеком вызовов. **Стек вызовов (call stack)** — это стек, хранящий информацию для возврата управления из подпрограмм (процедур, функций) в вызывающую программу или подпрограмму (при вложенных или рекурсивных вызовах), а также для возврата в программу из обработчика прерывания. При вызове подпрограммы или возникновении прерывания, **адрес возврата** (адрес следующей инструкции приостановленной программы) заносится в стек, и управление передается подпрограмме или обработчику. При последующих вложенных или рекурсивных вызовах адрес возврата снова заносится в стек. При возврате из подпрограммы адрес возврата снимается со стека, и управление передается на следующую инструкцию по этому адресу.

**Что такое стековый кадр? Стековые кадры (stack frames)** — это машинно- и ABI-зависимые структуры данных, содержащие информацию о состоянии подпрограммы. Каждый кадр стека соответствует вызову подпрограммы, который еще не завершился возвратом. Стековый фрейм на вершине стека предназначен для выполняющейся в данный момент процедуры. Он обычно содержит, по крайней мере, следующие элементы: аргументы (значения параметров), передаваемые подпрограмме; адрес, возвращаемый вызывающей программой; пространство для локальных переменных подпрограммы. Процедура может получать доступ к информации внутри своего фрейма (параметрам, локальным переменным) в любом порядке.

**Что такое соглашение о вызовах и почему они важны при перехвате? Соглашение о вызовах** определяет, как функция вызывается, как функция управляет стеком и стековым кадром, как аргументы передаются в функцию, и как функция возвращает значения. Соглашения о вызовах важны при перехвате API, поскольку они определяют механику передачи управления и данных между вызывающей и вызываемой функциями. Чтобы успешно перехватить функцию и вставить свой код (функцию-перехватчик), необходимо соблюдать то же соглашение о вызовах, что и оригинальная функция. Несоблюдение соглашения о вызовах может привести к некорректной работе программы, сбоям или ошибкам при передаче аргументов и возврате значений.

Соглашения о вызовах:

- **stdcall (Standard Calling Convention)**
  - Применяется в ОС Windows для вызова функций WinAPI.
  - Аргументы функций передаются через стек, справа налево.
  - Очистку стека производит вызываемая подпрограмма.

- Перед возвратом вызываемая подпрограмма обязана восстановить значения сегментных регистров, регистров указателя стека и стекового кадра.
- Сохранением-восстановлением остальных регистров занимается вызывающая программа.
- **cdecl (C calling convention)**
  - Используется компиляторами для языка Си.
  - Аргументы функций передаются через стек, справа налево.
  - Аргументы размером менее 4 байт расширяются до 4 байт.
  - За сохранение регистров EAX, ECX, EDX и стека сопроцессора отвечает вызывающая программа, за остальные – вызываемая функция.
  - Очистку стека производит вызывающая программа.
  - Перед вызовом функции вставляется код, называемый прологом, который сохраняет значения регистров и записывает аргументы в стек.
  - После вызова функции вставляется код, называемый эпилогом, который восстанавливает значения регистров и очищает стек от локальных переменных функции.
- **fastcall (Fast calling convention)**
  - Общее название соглашений, передающих параметры через регистры.
  - Если регистров недостаточно, дополнительно используется стек.
  - Не стандартизировано, используется только для вызова процедур и функций, не экспортируемых или импортируемых извне.
  - В 32-разрядной версии компилятора Microsoft первые два параметра передаются слева направо в регистрах, остальные – справа налево в стеке.
  - Очистку стека производит вызываемая подпрограмма.
- **pascal (Pascal calling convention)**
  - Используется компиляторами для языка Паскаль.
  - Аргументы процедур и функций передаются через стек, слева направо.
  - Указатель на вершину стека возвращает на исходную позицию вызываемая подпрограмма.
  - Изменяемые параметры передаются только по ссылке.
  - Возвращаемое значение передаётся через изменяемый параметр Result, который создаётся неявно и является первым аргументом функции.

### **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 API-функций?***

Перехват API функций (API hooking) – это техника программирования, при которой вызовы функций из библиотеки API перенаправляются на пользовательские функции. Перехват API функций позволяет модифицировать поведение программы, добавлять новые функции или изменять параметры вызовов **без необходимости изменения исходного кода**



**программы.** В операционной системе Windows NT, построенной на системе динамически загружаемых библиотек (DLL), перехват API функций позволяет обойти многие ограничения системы и делать с ней практически что угодно, поскольку система предоставляет приложениям сервисные API функции для взаимодействия.

Основными методами перехвата являются:

- **Подмена адреса настоящей функции:**
  - Модификация IAT таблиц.
  - Модификация SSDT/IDT таблиц.
- **Непосредственное изменение функции:**
  - Сплайсинг (splicing).
  - Перехват в режиме ядра с модификацией тела функции.
- **Непосредственная подмена всего компонента приложения/системы.**

Методы перехвата можно также разделить по критерию режима выполнения:

- **Пользовательские методы:** модификация IAT таблиц, сплайсинг. Их особенность в том, что невозможно что-либо изменить в поведении ядра операционной системы и его расширений.
- **Методы режима ядра:** модификация SSDT/IDT таблиц, перехват в режиме ядра с модификацией тела функции. Эти методы позволяют модифицировать структуры данных и код любой части операционной системы и приложений.

**Перехват API-вызовов путём модификации исходного кода: Сплайсинг (splicing)** – это метод перехвата API функций путём изменения кода целевой функции. Обычно изменяются первые 5 байт функции. Вместо них вставляется переход на функцию, которую определяет программист. Чтобы обеспечить корректность выполнения операции, приложение, которое перехватывает функцию, обязано дать возможность выполниться коду, который был изменён в результате сплайсинга. Для этого приложение сохраняет заменяемый участок памяти у себя, а после отработки функции перехвата восстанавливает изменённый участок функции и даёт ему полностью выполниться. Все функции стандартных DLL Windows поддерживают **hot-patch point**. При использовании этой технологии перед началом функции располагаются пять неиспользуемых однобайтовых операций `nop`, сама же функция начинается с двухбайтовой инструкции `mov edi, edi`. Места, занимаемого пятью `nop`, достаточно, чтобы разместить команду перехода на функцию-перехватчик.

Классический способ реализации API-хуков осуществляется с помощью **трамплинов**.

**Трамплин** – это шеллкод, который используется для изменения пути выполнения кода путём перехода на другой конкретный адрес в адресном пространстве процесса. Шеллкод трамплина вставляется в начало функции, в результате чего функция становится "подцепленной". Когда вызывается подцепленная функция, вместо неё активируется шеллкод трамплина, и поток выполнения передаётся и изменяется на другой адрес, что приводит к выполнению другой функции. **Шеллкод (shellcode)** - это небольшой фрагмент машинного кода, который обычно используется в эксплуатации уязвимостей для выполнения произвольного кода на целевой системе.

**Встраиваемый хук (inline hook)** – это альтернативный метод выполнения API-хуков, который работает аналогично хуку на основе трамплина. Разница заключается в том, что встраиваемые

хуки возвращают выполнение законной функции, позволяя нормальному выполнению продолжаться. Несмотря на то что они сложнее в реализации и потенциально труднее в обслуживании, встраиваемые хуки более эффективны.

Существует много способов реализации API-хука, один из способов – через открытые библиотеки, такие как библиотека Detours или Minhook. Ещё один, более ограниченный способ, – использование API Windows, предназначенных для выполнения API-хука (хотя с ограниченными возможностями).

**Локальный перехват может быть реализован в WinNT посредством подмены адреса перехватываемой функции в таблице импорта.** В разделе импорта исполняемого модуля содержится список DLL, необходимых модулю для нормальной работы, и перечисляются все идентификаторы, которые модуль импортирует из каждой DLL. **Вызывая импортируемую функцию, поток получает её адрес фактически из раздела импорта.** Поэтому, чтобы перехватить определённую функцию, надо изменить этот адрес в таблице импорта.

Этот метод выглядит так: определяется точка входа перехватываемой функции. Составляется список модулей, в настоящий момент загруженных в контекст требуемого процесса. Затем перебираются дескрипторы импорта этих модулей в поиске адресов перехватываемой функции. В случае совпадения этот адрес изменяется на адрес нашего обработчика. К достоинствам данного метода можно отнести то, что код перехватываемой функции не изменяется, что обеспечивает корректную работу в многопоточном приложении. Недостаток этого метода в том, что приложения могут сохранить адрес функции до перехвата и затем вызывать её минуя обработчик. Также можно получить адрес функции используя GetProcAddress из Kernel32.dll.

Перехват API-вызовов имеет двойственную природу и может использоваться как в добросовестных, так и в недобросовестных целях.

#### **Добросовестное использование перехвата функций:**

- **Отладка и мониторинг:** Перехват может использоваться для перехвата и регистрации вызовов функций в целях отладки, помогая разработчикам понять и диагностировать проблемы в их программном обеспечении.
- **Программное обеспечение для обеспечения безопасности:** Антивирусное программное обеспечение часто использует перехват для отслеживания и перехвата потенциально вредоносных действий.
- **Мониторинг производительности:** Подключение может использоваться для сбора показателей производительности и оптимизации программного обеспечения.
- **Расширение функциональных возможностей:** Подключение может использоваться для добавления или изменения функциональных возможностей в существующее программное обеспечение без изменения исходного кода.

#### **Недобросовестное использование перехвата функций:**

- **Руткиты:** Вредоносное программное обеспечение может использовать перехват, чтобы скрыть своё присутствие в системе, что затрудняет его обнаружение и удаление.
- **Кража данных:** Перехват может быть использован для перехвата конфиденциальных данных, таких как пароли или ключи шифрования.

- **Несанкционированный доступ:** Перехват может использоваться для обхода механизмов безопасности и получения несанкционированного доступа к системам или данным.
- **Распространение вредоносного ПО:** Перехват может использоваться для распространения вредоносного ПО путём перехвата и изменения сетевого трафика или файловых операций.

### ***35. Перехват API. Что такое перехват API-функций?***

***Расскажите всё о перехвате API-вызовов путём модификации системных таблиц: суть, алгоритм, что такое SSDT.***

***Расскажите всё об использовании драйверов-фильтров для перехвата. Сложности перехвата в Windows. Для чего может использоваться перехват API-функций?***

**Перехват API-функций (API hooking)** – это техника программирования, при которой вызовы функций из библиотеки API перенаправляются на пользовательские функции. Эта техника позволяет модифицировать поведение программы, добавлять новые функции или изменять параметры вызовов без необходимости изменения исходного кода программы.

Одним из основных методов перехвата является **подмена адреса настоящей функции**, которая может выполняться путем модификации системных таблиц, таких как IAT (Import Address Table) или SSDT (System Service Dispatch Table). Перехват можно разделить по критерию режима выполнения: пользовательские методы (модификация IAT, сплайсинг) и методы режима ядра (модификация SSDT/IDT, перехват с модификацией тела функции в режиме ядра). Методы режима ядра позволяют модифицировать структуры данных и код любой части операционной системы и приложений.

**Перехват API-вызовов путем модификации системных таблиц SSDT** Суть этой техники заключается в изменении поведения операционной системы Windows. **SSDT (System Service Dispatch Table)** содержит адреса функций, которые реализуют системные вызовы ядра. Эти функции вызываются приложениями и драйверами для выполнения различных операций, таких как управление процессами, файлами, памятью и т.д.. SSDT сопоставляет системные вызовы с адресами функций ядра. Когда приложение пользовательского пространства выполняет системный вызов, оно содержит служебный индекс в качестве параметра, указывающего, какой системный вызов вызывается. Затем SSDT используется для определения адреса соответствующей функции внутри ntoskrnl.exe.

**Алгоритм перехвата API-вызовов путем модификации SSDT** включает следующие основные шаги:

1. **Получение адреса SSDT:** Адрес SSDT можно получить, используя системные структуры и функции Windows, что обычно делается с помощью недокументированных структур и функций ядра.
2. **Поиск целевого системного вызова:** В SSDT каждому системному вызову соответствует индекс. Необходимо определить индекс системного вызова, который нужно перехватить.

3. **Сохранение оригинального адреса:** Перед заменой адреса системного вызова необходимо сохранить оригинальный адрес, чтобы можно было вызвать оригинальную функцию из пользовательской функции.
4. **Замена адреса на пользовательскую функцию:** Адрес системного вызова в SSDT заменяется на адрес пользовательской функции, которая будет выполнять необходимые действия перед или после вызова оригинальной функции.
5. **Восстановление оригинального адреса (опционально):** После выполнения необходимых действий можно восстановить оригинальный адрес системного вызова в SSDT.

**Использование драйверов-фильтров для перехвата** Эта техника перехвата основана на идее замены указателей на процедуры диспетчеризации работающих драйверов. Это автоматически обеспечивает "фильтрацию" для всех устройств, управляемых этим драйвером.

Перехватывающий драйвер сохраняет старые указатели на функции, а затем заменяет основной массив функций в объекте драйвера на свои собственные функции. Теперь любой запрос, поступающий к устройству под управлением перехваченного драйвера, будет вызывать диспетчерские процедуры перехватывающего драйвера. При этом не создаются дополнительные объекты устройств и не происходит никакого перемещения IRP-пакетов. Чтобы подключить драйвер, необходимо найти указатель на объект драйвера (DRIVER\_OBJECT), для чего можно использовать недокументированную, но экспортируемую функцию, которая может найти любой объект по его имени. Подключающий драйвер после этого может заменить указатели основных функций, процедуру выгрузки, процедуру добавления устройства и т.д.. При любой такой замене всегда следует сохранять предыдущие функциональные указатели для отключения при необходимости и для отправки запроса реальному драйверу.

**Сложности перехвата в Windows** Стоит отметить, что перехват API-вызовов путем модификации системных таблиц и драйверами-фильтрами являются трудно реализуемыми в немалой степени по причине существования такой технологии как **PatchGuard**. PatchGuard, также известный как Kernel Patch Protection (KPP), – это технология защиты, встроенная в операционные системы Windows. Основная цель PatchGuard – предотвратить модификацию критических структур данных ядра и таблиц системных вызовов, таких как System Service Dispatch Table (SSDT), Interrupt Descriptor Table (IDT) и Global Descriptor Table (GDT), а также защитить от других попыток изменения поведения ядра. Например, если злоумышленник попытается изменить SSDT для перехвата системных вызовов, PatchGuard обнаружит это изменение и вызовет BSOD, чтобы предотвратить дальнейшее выполнение.

**Для чего может использоваться перехват API-функций?** Учитывая двойную природу методов перехвата, он может использоваться как в добросовестных, так и в недобросовестных целях.

**Добросовестное использование перехвата функций:**

- **Отладка и мониторинг:** Перехват может использоваться для перехвата и регистрации вызовов функций в целях отладки, помогая разработчикам понять и диагностировать проблемы в их программном обеспечении.

- **Программное обеспечение для обеспечения безопасности:** Антивирусное программное обеспечение часто использует перехват для отслеживания и перехвата потенциально вредоносных действий.
- **Мониторинг производительности:** Подключение может использоваться для сбора показателей производительности и оптимизации программного обеспечения.
- **Расширение функциональных возможностей:** Подключение может использоваться для добавления или изменения функциональных возможностей в существующее программное обеспечение без изменения исходного кода.

#### **Недобросовестное использование перехвата функций:**

- **Руткиты:** Вредоносное программное обеспечение может использовать перехват, чтобы скрыть свое присутствие в системе, что затрудняет его обнаружение и удаление.
- **Кража данных:** Перехват может быть использован для перехвата конфиденциальных данных, таких как пароли или ключи шифрования.
- **Несанкционированный доступ:** Перехват может использоваться для обхода механизмов безопасности и получения несанкционированного доступа к системам или данным.
- **Распространение вредоносного ПО:** Перехват может использоваться для распространения вредоносного ПО путем перехвата и изменения сетевого трафика или файловых операций.

Важно подходить к этой теме с учетом этических соображений, соблюдая законодательство, получая информированное согласие при использовании в законных целях, обеспечивая прозрачность и применяя рекомендации по обеспечению безопасности для защиты от злонамеренного использования перехвата.

### **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 — это процесс преобразования части кода в другую, функционально эквивалентную часть, с целью улучшения одной или нескольких характеристик кода.

**Основные характеристики, которые могут быть оптимизированы, включают:**

- **Скорость работы.**
- **Размер кода.**
- Энергопотребление, необходимое для выполнения кода.
- Время компиляции кода.
- Длительность JIT-компиляции, если конечный код требует ее.

На вопрос, **стоит ли заниматься оптимизацией кода вручную**, нет однозначного ответа; все зависит от ситуации. Компиляторы постоянно совершенствуются в методах оптимизации, но они не идеальны. Вместо ручной оптимизации программы часто продуктивнее использовать специфические средства компилятора и дать ему возможность оптимизировать код.

Оптимизирующий компилятор может создавать код, который почти так же хорош, как и оптимизированный вручную ассемблерный код, но для достижения этого код на языке высокого уровня (HLL) должен быть написан соответствующим образом. Это требует четкого понимания того, как работают компьютеры и исполняется программное обеспечение. Программисты, которые тщательно пишут HLL-код, могут создавать наилучший машинный код с помощью компилятора.

#### **Основные принципы проведения оптимизации:**

- Оптимизация должна проводиться строго при необходимости и с осторожностью. Каждый шаг оптимизации должен быть тщательно отлажен и протестирован.
- «Преждевременная оптимизация — это корень всех бед». Разработчик не должен при разработке думать об оптимальности, а должен думать о целостности и завершенности кода.
- Понимание того, как компилятор организует промежуточный код, очень важно, чтобы помочь оптимизатору выполнять свою работу более эффективно.
- Хорошие программы создают упрощаемые графы потоков. Любая программа, состоящая только из структурированных управляющих команд (if, while, repeat..until и т.д.) и избегающая операторов goto, будет упрощаемой. Оптимизаторы компиляторов, как правило, гораздо лучше справляются с работой над упрощаемыми программами.
- Существуют определенные характеристики кода, которые могут препятствовать оптимизации компилятором. К таким «блокировщикам оптимизации» относятся указатели, когда компилятор не может точно знать, будут ли два указателя указывать на одну и ту же область памяти, и вызовы функций, которые могут иметь побочные эффекты и изменять глобальное состояние.
- Память имеет значение; программы с хорошей локальностью выполняются быстрее, чем программы с плохой локальностью. Локальность бывает временной (обращение к одному и тому же месту в памяти много раз) и пространственной (обращение к данным, а затем к данным, расположенным рядом в памяти). Следует стремиться максимизировать пространственную и временную локальность.

#### **Советы по разработке программ для оптимизации включают:**

- **Высокоуровневый дизайн:** Выбирать эффективные алгоритмы и структуры данных.
- **Базовые принципы кодирования:** Избегать блокировщиков оптимизации (указателей, ненужных вызовов функций, ненужных запросов к памяти). Выносить вычисления за пределы цикла, если возможно. Вводить временные переменные для хранения промежуточных результатов.
- **Низкоуровневая оптимизация:** Применять раскрутку циклов, задействовать параллелизм на уровне инструкций, писать инструкции if-else в функциональном стиле.
- Концентрировать внимание на внутренних циклах.
- Максимизировать пространственную и временную локальность.

#### **Ключевые аспекты связи оптимизации и системного программирования:**

- **Производительность:** Системное программирование часто требует высокой производительности, и оптимизация кода помогает достичь необходимых уровней.

- **Эффективное использование ресурсов:** Системное программирование включает управление низкоуровневыми ресурсами, а оптимизация помогает эффективно их использовать.
- **Надежность и стабильность:** Оптимизация может улучшить структуру кода и стабильность программы, что важно для системного программирования, где ошибки могут привести к серьезным сбоям.
- **Совместимость и переносимость:** Системное программирование часто требует создания кода для разных платформ, и оптимизация помогает обеспечить совместимость.
- **Безопасность:** Системное программирование требует высокого уровня безопасности, а оптимизация может помочь улучшить безопасность кода и защитить систему.
- **Отладка и тестирование:** Оптимизация включает процессы отладки и тестирования, что крайне важно в системном программировании из-за серьезных последствий ошибок.

Существует **несколько уровней оптимизации:**

- На самом абстрактном уровне можно оптимизировать программу, выбрав **лучший алгоритм**. Этот метод не зависит от компилятора и языка программирования.
- На следующем уровне можно проводить **ручную оптимизацию кода на основе используемого HLL**, не зависящую от конкретной реализации этого языка.
- Еще на один уровень ниже можно **структурировать код таким образом, чтобы оптимизация была применима только к определенному поставщику или версии компилятора**.
- На самом низком уровне можно **учитывать машинный код, выдаваемый компилятором, и корректировать инструкции на HLL**, чтобы заставить компилятор генерировать определенную оптимизированную последовательность машинных инструкций.

Оптимизация компилятором сводится к преобразованию промежуточного кода «в более эффективную форму». Определение эффективности связано с минимизацией использования системных ресурсов, обычно памяти или циклов процессора. Оптимизация для одной цели (например, повышение производительности) может конфликтовать с другой целью (например, сокращение использования памяти), поэтому процесс оптимизации обычно является компромиссом. Компиляторы применяют только безопасные оптимизации, которые не должны менять поведение программы для всех входных данных.

***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.***

**Что такое оптимизация кода?**

**Оптимизация кода** – это процесс преобразования одной части кода в другую, функционально эквивалентную часть, с целью улучшения одной или нескольких характеристик кода. Две наиболее важные характеристики – это **скорость работы** и **размер кода**. К другим характеристикам относятся энергопотребление, необходимое для выполнения кода, время компиляции кода, а также длительность JIT-компиляции, если конечный код требует ее. В контексте системного программирования оптимизация кода тесно связана с созданием эффективных и надежных систем, улучшая производительность, эффективность использования ресурсов, надежность, стабильность, совместимость, переносимость и безопасность.

## Принципы оптимизации кода компилятором.

Оптимизирующие компиляторы постоянно совершенствуются в методах оптимизации кода. Они могут создавать код, который почти так же хорош, как и вручную оптимизированный ассемблерный код, при определенных условиях. Однако для достижения таких уровней производительности код на языке высокого уровня (HLL) должен быть написан соответствующим образом, что требует понимания того, как работают компьютеры и исполняется программное обеспечение. Важно понимать, как компилятор проводит оптимизацию и что в вашем исходном коде может препятствовать его эффективной работе.

Основные принципы и аспекты оптимизации компилятором:

- **Безопасные оптимизации:** Компиляторы применяют только "безопасные" оптимизации, то есть они изменяют программу лишь таким образом, чтобы это не меняло её поведение для всех входных данных.
- **Компромиссы:** Процесс оптимизации обычно представляет собой **компромисс** между различными целями, например, увеличение производительности может конфликтовать с сокращением использования памяти. Для достижения приемлемого результата часто приходится жертвовать определенными подцелями.
- **Понимание промежуточного кода:** Понимание того, как компилятор организует промежуточный код, очень важно для помощи оптимизатору.
- **Отслеживание значений переменных:** Оптимизатор отслеживает значения переменных в процессе, известном как **анализ потока данных (DFA)**.
- **Блокировщики оптимизации:** Существуют определенные характеристики кода, которые могут препятствовать эффективной оптимизации компилятором. К ним относятся:
  - **Указатели:** Компилятор не может точно знать, будут ли два указателя указывать на одну и ту же область памяти, что препятствует некоторым оптимизациям.
  - **Вызовы функций:** Вызовы функций влекут накладные расходы, и компилятору трудно определить, имеют ли они побочные эффекты (изменение глобального состояния). В таких случаях компилятор исходит из худшего сценария и не выполняет оптимизацию.
- **Важность локальности данных:** Эффективность программы сильно зависит от того, как данные расположены в памяти и как к ним обращаются. Программы с хорошей локальностью данных (пространственной и временной) выполняются быстрее, так как данные остаются в кеше процессора.

## Что такое анализ потока данных?



**Анализ потока данных (DFA)** – это процесс, в котором оптимизатор компилятора отслеживает значения переменных по мере прохождения управления по программе. После тщательного анализа компилятор может определить, когда переменная не инициализирована, когда она содержит определенные значения, когда программа больше не использует переменную, и когда компилятор просто ничего не знает о значении переменной.

### **Что такое базовые блоки? Зачем они нужны?**

**Базовые блоки** – это последовательности машинных инструкций, от которых нет ответвлений, кроме как в начале и конце. Компиляторы разбивают исходный код на такие блоки для анализа потока данных.

- **Определение:** Базовый блок заканчивается там, где происходит переход к последовательности инструкций или из неё, а также при условной или безусловной ветви, или команде вызова.
- **Назначение:** Базовые блоки позволяют компилятору легко отслеживать, что происходит с переменными и другими программными объектами. Обработывая каждую инструкцию внутри базового блока, компилятор может символически отслеживать значения, которые будут храниться в переменной, на основе их начальных значений и вычислений.

### **Что такое упрощаемые графики потоков?**

**Упрощаемые графики потоков** – это наглядные изображения пути потока управления в хорошо структурированных программах. Плохо структурированные программы, напротив, могут создавать пути потока управления, которые сбивают с толку компилятор, уменьшая возможности оптимизации.

### **Какая программа будет упрощаемой?**

**Упрощаемой** будет любая программа, состоящая только из структурированных управляющих команд (таких как `if`, `while`, `repeat...until` и т.д.) и избегающая использования операторов `goto`. Это важно, потому что оптимизаторы компиляторов гораздо лучше справляются с работой над упрощаемыми программами, в то время как программы, не являющиеся упрощаемыми, как правило, их замедляют. Работа с сокращаемыми программами облегчает работу оптимизаторам, поскольку их базовые блоки могут быть схематично свернуты, а вложенные блоки наследуют свойства (например, какие переменные изменяет блок) от вложенных блоков.

### **Перечислите основные типы оптимизаций компилятора.**

К основным типам оптимизаций компилятора относятся:

- **Свёртка констант:** Вычисление значений константных выражений или подвыражений во время компиляции, а не во время выполнения.
- **Распространение констант:** Замена переменных постоянными значениями, если компилятор определяет, что эта константа была присвоена переменной ранее в коде.

Это часто приводит к более эффективному коду, так как манипулирование непосредственными константами может быть более эффективным, чем с переменными.

- **Удаление мёртвого кода:** Удаление объектного кода, связанного с определенным оператором исходного кода, когда результат этого оператора никогда не будет использоваться программой, или когда условный блок никогда не будет истинным.
- **Удаление общих подвыражений:** Оптимизация, при которой компилятор ищет экземпляры одинаковых выражений и заменяет их одной переменной, содержащей вычисленное значение, если значения переменных в подвыражении не изменились.
- **Снижение стоимости операций:** Замена медленных операций (например, умножения и деления) на более быстрые (например, сложение, вычитание, сдвиг). Однако ручное снижение стоимости рискованно и непереносимо.
- **Анализ индуктивных переменных:** В выражениях, особенно в циклах, где значение одной переменной полностью зависит от другой, компилятор может исключить вычисление нового значения или объединить два вычисления в одно на время цикла.
- **Анализ инвариантов цикла:** Вычисление выражений, которые не меняются на каждой итерации цикла, только один раз вне цикла, а затем использование вычисленного значения в теле цикла. Многие оптимизаторы достаточно умны, чтобы обнаружить такие вычисления.
- **Раскрутка циклов:** Техника, увеличивающая количество инструкций, исполняемых за одну итерацию цикла, для увеличения параллелизма и более интенсивного использования регистров, кеша и исполнительных устройств процессора.
- **Вычисления по короткой схеме:** Стратегия, при которой второй логический оператор в логическом выражении выполняется или вычисляется только в том случае, если первого оператора недостаточно для определения значения выражения. Это означает, что вычисление прекращается, как только результат становится очевидным.

***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?***

Оптимизация кода – это процесс преобразования части кода в другую функционально эквивалентную часть для **улучшения одной или более характеристик кода**, таких как **скорость работы и размер кода**. К другим характеристикам относятся энергопотребление, время компиляции и длительность JIT-компиляции.

Ниже представлены основные аспекты оптимизации кода:

- **Основные типы оптимизаций компилятора:**

- **Свёртка констант** – это вычисление значений константных выражений или подвыражений во время компиляции, а не во время выполнения. Например, компилятор не будет генерировать инструкции для умножения, а просто заменит выражение на заранее высчитанное значение.
- **Распространение констант** – это замена переменных постоянными значениями, если компилятор определяет, что программа присвоила эту константу переменной ранее в коде. Это часто приводит к лучшему коду, так как манипулирование непосредственными константами часто более эффективно, чем манипулирование переменными. В некоторых случаях это также позволяет компилятору полностью исключить определенные переменные и инструкции.
- **Удаление мёртвого кода** – это удаление объектного кода, связанного с определенным оператором исходного кода, когда программа никогда не будет использовать результат этого оператора или когда условный блок никогда не будет истинным.
- **Удаление общих подвыражений** – это оптимизация, которая ищет экземпляры одинаковых выражений и анализирует возможность замены их на одну переменную, содержащую вычисленное значение. Если значения переменных в подвыражении не изменились, программе не нужно пересчитывать их везде, где оно появляется; программа может просто сохранить значение подвыражения при первом вычислении, а затем использовать его при каждом последующем появлении.
- **Снижение стоимости операций** – это замена медленных операций (например, умножения и деления) на более быстрые (такие как сложение, вычитание, сдвиг). Важно быть осторожным при попытке снизить стоимость вручную, так как многие такие оптимизации не переносимы на другие процессоры, и эту работу лучше доверить компилятору.
- **Анализ индуктивных переменных** – во многих выражениях, особенно в цикле, значение одной переменной полностью зависит от другой. Компилятор часто может исключить вычисление нового значения или объединить два вычисления в одно на время цикла.
- **Анализ инвариантов цикла** – инвариант цикла — это выражение, которое не меняется на каждой итерации цикла. Оптимизатор может вычислить результат такого выражения только один раз, вне цикла, а затем использовать вычисленное значение в теле цикла. Многие оптимизаторы способны обнаруживать такие вычисления и перемещать их за пределы цикла.
- **Раскрутка циклов** – техника увеличения количества инструкций, выполняемых за одну итерацию цикла, что позволяет увеличить количество параллельно выполняемых блоков инструкций и более интенсивно использовать регистры процессора, кэш данных и исполнительные устройства.
- **Вычисления по короткой схеме (short-circuit evaluation):**
  - Это стратегия вычисления в некоторых языках программирования, при которой **второй логический оператор выполняется или вычисляется только в том случае, если первого логического оператора недостаточно для определения значения выражения**. Таким образом, после того, как результат выражения становится очевидным, его вычисление прекращается.

- Основные примеры таких вычислений – **логические операторы AND (&&) и OR (| |)**. Например, в выражении  $P1 \ \&\& \ P2$ , если  $P1$  ложно,  $P2$  не будет вычисляться, так как весь результат уже известен (ложно). Аналогично, в  $P1 \ || \ P2$ , если  $P1$  истинно,  $P2$  не будет вычисляться. Это свойство важно, и корректная работа многих алгоритмов зависит от него.
- **Примерная иерархия скорости выполнения операторов процессором:**
  - Источники отмечают, что **невозможно создать универсальную таблицу скоростей операторов**, поскольку производительность отдельного арифметического оператора зависит от конкретного процессора.
  - Однако существуют общие рекомендации: **простое сложение целых чисел часто выполняется намного быстрее, чем умножение целых чисел**. Аналогично, **операции с целыми числами обычно выполняются намного быстрее, чем соответствующие операции с плавающей запятой**. Общая методология анализа алгоритмов часто упрощает это, предполагая, что все операции занимают одинаковое количество времени, но это редко верно.
- **Безопасные оптимизации:**
  - Компиляторы могут применять только **безопасные оптимизации**. Это означает, что **компилятор может изменять программу только таким образом, чтобы это не изменило её поведение для всех возможных входных данных**. В некоторых случаях, плохо написанный код может привести к нарушению этого правила.
- **Блокировщики оптимизации:**
  - Это **определенные характеристики кода, которые не позволят компилятору совершить оптимизацию**, даже если он обычно способен её выполнить. Компилятору тяжело определить, имеет ли вызов функции побочные эффекты, и когда он не может этого сделать, он предполагает худшее и не выполняет оптимизацию.
- **Какие существуют блокировщики?**
  - **Указатели:** Компилятор не может точно знать, будут ли два указателя указывать на одну и ту же область памяти. Из-за этой неопределенности он не выполняет некоторые оптимизации. Например, если две функции семантически выглядят одинаково, но при определенных условиях (когда указатели указывают на одну и ту же ячейку памяти) ведут себя по-разному, компилятор не осмелится трансформировать менее эффективную функцию в более эффективную.
  - **Вызовы функций:** Вызовы функций влекут за собой накладные расходы, и их следует избегать, если это возможно. Компилятору сложно определить, имеет ли вызов функции побочные эффекты (например, изменение глобального состояния). Если компилятор не может быть уверен, что функция не имеет побочных эффектов, он не будет выполнять оптимизацию, которая могла бы изменить порядок или количество вызовов функции.
- **Локальность данных:**
  - Это свойство данных, при котором **желательно, чтобы данные находились в памяти рядом, когда мы над ними работаем**. Программы с хорошей

локальностью обычно выполняются быстрее, чем программы с плохой локальностью. Например, обход матрицы по столбцам может иметь плохую пространственную локальность, если матрица хранится в памяти построчно.

- **Какая бывает локальность?**

- **Временная локальность:** Возникает, когда мы **обращаемся к одному и тому же месту в памяти много раз**. Например, переменные `sum` и `i` в цикле имеют хорошую временную локальность, так как к ним обращаются на каждой итерации, и они могут располагаться в быстрых регистрах процессора. Программе лучше работать с меньшим количеством переменных и обращаться к ним чаще.
- **Пространственная локальность:** Возникает, когда мы **обращаемся к данным, а затем обращаемся к другим данным, которые расположены в памяти рядом с первоначальными**. Элементы массива, к которым обращаются последовательно, демонстрируют хорошую пространственную локальность. Когда процессор читает данные из памяти, он копирует их в свой кэш блоками определенного размера (например, 64 байта), и если вы читаете байты, а затем байты рядом с ними, они, скорее всего, уже будут в кэше, обеспечивая быстрый доступ.

***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. Виртуализация является основой современных технологий, например, позволяя облачным сервисам обеспечивать изолированные среды для миллионов пользователей. В контексте системного программирования виртуализация решает задачи управления ресурсами, оптимизации и безопасности. Источники упоминают ранние эксперименты с виртуальными машинами еще в 1960-х годах на IBM System/360, паузу в 1990-х из-за дороговизны оборудования, возрождение в 2000-х с развитием процессоров и ее современную роль как сердца облаков, DevOps и безопасных сред.

**Монитор виртуальных машин** (большинству разработчиков известный под названием **гипервизор**) — это программа, которая создает иллюзию существования нескольких виртуальных машин на одном физическом оборудовании. Гипервизор позволяет одному компьютеру стать базой для нескольких виртуальных машин, на каждой из которых потенциально может быть запущена совершенно другая операционная система. По сути, задача гипервизора — управлять доступом к ресурсам и предоставлять иллюзию функционирования виртуальных машин с достаточной эффективностью.

Вам уже могут быть знакомы концепции, подобные виртуализации, из области операционных систем:

- **Процессы.**
- **Виртуальная память.**
- **Файлы.** По сути, эти объекты/механизмы операционной системы позволяют программе использовать физические ресурсы компьютера, а также «верить» в то, что они принадлежат только ей. Операционная система в данном случае выступает как некоторый «гипервизор», управляющий доступом к ресурсам.

Собственно, под **виртуальной машиной (ВМ)** понимается некоторая изолированная среда, имитирующая физический компьютер. Виртуальные машины работают так же, как и реальные, им нужна возможность начальной загрузки и установки произвольных операционных систем.

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

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.

Существуют два основных типа гипервизоров:

- **Гипервизоры первого типа (bare-metal).** Этот тип гипервизора технически похож на операционную систему, поскольку это единственная программа, запущенная в самом привилегированном режиме. Его работа заключается в поддержке нескольких копий имеющегося оборудования, которое называется виртуальными машинами. Поскольку такой гипервизор работает непосредственно с аппаратным обеспечением, его называют «bare-metal». Примерами гипервизоров первого типа являются VMware ESXi, Microsoft Hyper-V, KVM, Xen (в режиме HVM).
- **Гипервизоры второго типа (hosted).** В отличие от первого типа, это программа, которая опирается на основную операционную систему, например, Windows или Linux, при распределении и планировании использования ресурсов и очень похожа на обычный процесс. Гипервизор второго типа притворяется полноценным компьютером с центральным процессором и различными устройствами. Многие его функциональные возможности зависят от основной операционной системы. Примерами гипервизоров второго типа являются VMware Workstation, Oracle VirtualBox, Parallels Desktop.

Оба типа гипервизоров должны выполнять набор машинных инструкций безопасным образом.

**Центральная концепция** в любом типе виртуализации — это **абстрагирование физических ресурсов**.

На глобальном уровне можно выделить три основных **типа виртуализации**:

1. **Клиентская виртуализация.** Относится к возможностям виртуализации, размещенным на клиенте. Возникла из потребности организаций в обслуживании множества персональных устройств. Основные проблемы, на разрешение которых нацелен этот тип виртуализации: защита клиентской ОС от не одобренных приложений и поддержание одобренных приложений и ОС в актуальном состоянии. Типы клиентской виртуализации включают:
  - **Упаковка приложений** (например, Symantec Workspace Virtualization, VMware ThinApp). Изолирует приложение от операционной системы, снижая вероятность компрометации ОС.
  - **Потоковая передача приложений** (например, Microsoft App-V, Citrix Virtual Apps). Хранит приложения на серверах и загружает их на лету, обеспечивая актуальность ПО без полной установки.
  - **Эмуляция аппаратного обеспечения** (также называется **полной виртуализацией**) (например, Hyper-V, Oracle VirtualBox). Программное обеспечение для виртуализации загружается на клиентский компьютер и создает контейнер для гостевой ОС — виртуальную машину.
2. **Серверная виртуализация.** Направление виртуализации, относящееся к проблемам эффективного использования серверов в центрах обработки данных. Существуют три основных типа:
  - **Виртуализация уровня ОС** (Контейнеризация) (например, Docker, OpenVZ, Virtuozzo). Позволяет запускать изолированные контейнеры внутри одной операционной системы, 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** (например, Hyper-V, VirtualBox). На серверах ничем не отличается от таковой на клиенте. В рамках виртуальных машин может быть установлена любая операционная система, так как она не знает о существовании гипервизора. Приложения запускаются в по-настоящему изолированной среде.
  - **Паравиртуализация** (например, Xen). Представляет машиноподобный программный интерфейс, явно раскрывающий факт наличия виртуальной среды. В отличие от полной виртуализации, требует модификации гостевой ОС для взаимодействия с предоставляемым 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 паравиртуализации. Источники также приводят сравнительную таблицу этих трех подходов по критериям поддержки гостевых ОС, производительности, изоляции, управления, масштабируемости и сценариев использования.
3. **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. Эта виртуализация включает 3 уровня: физические устройства, слой виртуализации 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.

**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.**

**Виртуализация** — это концепция, при которой специальная программа, называемая **монитор виртуальных машин**, создает иллюзию присутствия нескольких (виртуальных) машин на одном и том же физическом оборудовании. Монитор виртуальных машин также известен как **гипервизор (hypervisor)**. Виртуализация позволяет одному компьютеру стать основой для нескольких виртуальных машин, на каждой из которых потенциально может быть запущена совершенно другая операционная система. Система или компьютер, на котором установлен гипервизор, называется **хост-системой**.

Под **виртуальной машиной (ВМ)** понимается некоторая изолированная среда, имитирующая физический компьютер. Так как виртуальная машина по сути является отдельным компьютером, для ее функционирования необходима операционная система, которая называется **гостевой операционной системой**.

Гипервизоры должны хорошо проявлять себя в трех направлениях: безопасность, эквивалентность и эффективность.

- **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.

На глобальном уровне можно выделить три основных **типа виртуализации**:

- Клиентская виртуализация
- Серверная виртуализация
- Виртуализация хранилищ

**Клиентская виртуализация** относится к возможностям виртуализации, размещенным на клиенте. Этот тип виртуализации возник из потребности организаций в обслуживании множества персональных устройств. Основные проблемы, на разрешение которых нацелен этот тип виртуализации, включают защиту клиентской ОС от не одобренных приложений и поддержание одобренных приложений и самой ОС в актуальном состоянии.

Типы клиентской виртуализации:



- **Упаковка приложений (Application Packaging).** Этот метод изолирует приложение, работающее на клиентском компьютере, от операционной системы, на которой оно выполняется. Изолируя приложение, оно не может изменять основные критически важные ресурсы ОС, что снижает вероятность компрометации вредоносными программами. Этого можно достичь, выполняя приложение "поверх" ПО, которое предоставляет каждому процессу свой виртуальный набор ресурсов, или включив ПО для виртуализации в конечный бинарный файл.
- **Потоковая передача приложений (Application Streaming).** Этот подход решает проблему поддержания актуального программного обеспечения, избегая его полной установки. Приложения хранятся на серверах в центре обработки данных и загружаются на лету на компьютер конечного пользователя, когда он хочет их использовать. Обновленное приложение доставляется автоматически без физической установки на клиент, что экономит место на диске. Этот подход имеет ограниченный спектр применения, в основном для статичных рабочих сред.
- **Эмуляция аппаратного обеспечения (Hardware Emulation),** также называемая **полной виртуализацией.** При этой форме виртуализации ПО для виртуализации загружается на клиентский компьютер с уже установленной базовой ОС. Это ПО создает контейнер для гостевой ОС - виртуальную машину. Установка гостевой ОС происходит через интерфейс ПО виртуализации, и виртуальной машиной можно управлять (запускать, останавливать, приостанавливать, уничтожать) через панель управления. Взаимодействие с гостевой ОС ВМ аналогично работе с ней как с единственной ОС на компьютере.

**Серверная виртуализация** относится к проблемам эффективного использования серверов в центрах обработки данных.

Существуют три основных типа серверной виртуализации:

- **Виртуализация уровня ОС (OS-level virtualization).**
- **Полная виртуализация (Full virtualization).**
- **Паравиртуализация (Paravirtualization).**

Поясним принципы этих трех типов:

- **Виртуализация уровня ОС (Контейнеризация):** Эта технология позволяет запускать изолированные контейнеры ("песочницы") внутри одной операционной системы. Эти контейнеры используют **общее ядро (kernel)** хост-системы, но изолируют процессы, память, сеть и другие ресурсы. Такой тип виртуализации называют **контейнеризацией**. В рамках каждого контейнера вместо полноценной гостевой ОС используется виртуальная ОС. Контейнеры изолированы друг от друга, но не требуют эмуляции аппаратуры. Они легковесны, не требуют запуска полноценной ОС для каждого контейнера, используют меньше ресурсов и быстрее запускаются. Контейнеризация обеспечивает слабую изоляцию, поскольку все контейнеры делят ядро хост-ОС. Управление простое, а масштабируемость максимальная (тысячи контейнеров могут работать на одном сервере). Сценарии использования включают микросервисные архитектуры, DevOps и CI/CD.

- **Полная виртуализация:** На серверах этот тип виртуализации аналогичен эмуляции аппаратного обеспечения на клиенте. В рамках виртуальных машин может быть установлена любая операционная система, поскольку гостевая ОС не знает о существовании гипервизора. Приложения запускаются в по-настоящему изолированной среде. Полная виртуализация обеспечивает высокую поддержку гостевых ОС (любые ОС) и полную (аппаратную) изоляцию. Производительность ниже из-за аппаратной эмуляции. Управление сложное, а масштабируемость средняя. Сценарии использования включают запуск разных ОС на одном сервере, тестирование ОС и облачные IaaS. Полная виртуализация может использовать как гипервизоры первого типа (bare-metal), так и второго типа (hosted).
- **Паравиртуализация:** Форма виртуализации, при которой виртуальная машина не пытается выглядеть как настоящее основное оборудование. Вместо этого она представляет машиноподобный программный интерфейс, который явно раскрывает факт наличия виртуальной среды. Паравиртуализация требует **модификации гостевой ОС** для взаимодействия с предоставляемым интерфейсом, что необходимо для корректной работы и лучшей производительности. Поскольку не во все ОС можно внести такие изменения, работоспособность решений может быть под вопросом. Разработчики CPU внедрили дополнительные наборы команд (AMD-V и VT-x), направленные на работу с виртуализацией, что позволяет запускать не модифицированные экземпляры ОС с помощью паравиртуализации. Паравиртуализация обеспечивает ограниченную поддержку гостевых ОС (требуется модификация) и частичную (программную) изоляцию, так как гостевая ОС "знает" о виртуализации. Производительность выше, чем у полной виртуализации. Управление среднее, а масштабируемость высокая. Сценарии использования включают высокопроизводительные вычисления и серверные среды с поддерживаемыми ОС. Паравиртуализация может использовать гипервизоры первого типа, но возможен и второй тип.

**Виртуализация хранилищ** – это технология, которая объединяет разные физические устройства хранения данных в единое логическое хранилище, позволяя централизованно управлять ими. Пользователи и приложения работают с данными как с одним ресурсом, не зная о физической структуре. Виртуализация хранилищ включает три уровня: физические устройства, слой виртуализации (управляет распределением данных) и виртуальные диски.

**КАЖДОГО КТО ДОЧИТАЛ ДО СЮДА, Я  
БЛАГОСЛАВЛЯЮ ПОРЧЕЙ НА ПОНОС, КОТОРАЯ БУДЕТ  
АКТИВИРОВАНА НЕМЕДЛЕННО И НАЧНЁТ СВОЁ  
ДЕЙСТВИЕ ЗАВТРА, В 8:20! СПАСЕНИЯ НЕТ, МЫ ВСЕ  
УМРЁМ**

**You told me once  
That you'll never leave me alone  
But I fucked it up  
I miss you so fucking much**

**I feel so sad  
Now I want to fucking die  
I'm all alone  
I want you by my side**

**Do you forgive me?  
Let's start over again  
Do you still love me?  
Won't let you down again**

**Do you forgive me?  
Let's start over again  
Do you still love me?  
Won't let you down again**

**You told me once  
That you'll never leave me alone  
But I fucked it up  
I miss you so fucking much**

**I feel so sad  
Now I want to fucking die  
I'm all alone  
I want you by my side**

**Do you forgive me?  
Let's start over again  
Do you still love me?  
Won't let you down again**

**Do you forgive me?  
Let's start over again  
Do you still love me?  
Won't let you down again**

