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

- **Программа** – это данные, предназначенные для управления конкретными компонентами системы обработки информации (СОИ) в целях реализации определенного алгоритма. Один из основных принципов машины фон Неймана гласит, что программы и данные хранятся в одной и той же памяти. Сохраняемая в памяти программа представляет собой коды, которые могут рассматриваться как данные. С точки зрения программиста программа – активный компонент, выполняющий действия, но с точки зрения процессора команды программы – это данные, которые процессор читает и интерпретирует.
- **Программное обеспечение (ПО)** – это совокупность программ СОИ и программных документов, необходимых для их эксплуатации. Существенно, что ПО – это программы, предназначенные для многократного использования и применения разными пользователями. В связи с этим ПО должно обладать рядом свойств:
    - **Необходимость документирования**: программы становятся ПО только при наличии документации. Конечный пользователь не может работать без документации, которая также делает возможным тиражирование и продажу ПО без его разработчика. Ошибкой в ПО считается ситуация, когда программное изделие функционирует не в соответствии со своим описанием, следовательно, ошибка в документации также является ошибкой в программном изделии.
    - **Эффективность**: ПО, рассчитанное на многократное использование (например, ОС, текстовый редактор), пишется и отлаживается один раз, а выполняется многократно, что делает выгодным перенос затрат на этап производства ПО и освобождение от затрат на этапе выполнения.
    - **Надежность**: включает тестирование программы при всех допустимых спецификациях входных данных, защиту от неправильных действий пользователя, защиту от взлома (пользователи должны взаимодействовать с ПО только через легальные интерфейсы). Появление ошибок любого уровня не должно приводить к краху системы; ошибки должны вылавливаться, диагностироваться и, если невозможно исправить, превращаться в корректные отказы. Системные структуры данных должны сохраняться безусловно, а целостность пользовательских данных желательна.
    - **Возможность сопровождения**: возможные цели сопровождения – адаптация ПО к конкретным условиям применения, устранение ошибок, модификация. Во всех случаях требуется тщательное структурирование ПО, а носителем информации о структуре должна быть программная документация. Адаптация часто может быть передоверена пользователю при тщательной отработке и описании сценариев инсталляции и настройки. Исправление ошибок требует развитой сервисной службы, собирающей информацию об ошибках и формирующей исправляющие пакеты. Модификация предполагает изменение спецификаций на ПО, при этом, как правило, должны поддерживаться и старые спецификации, а эволюционное развитие ПО экономит вложения пользователей.

- **Классификация ПО**: Подразделение ПО на системное и прикладное до некоторой степени устарело. Сегодняшнее деление предусматривает по меньшей мере три градации ПО:
    - **Системное ПО**.
    - **Промежуточное (связующее) ПО**: совокупность программ, осуществляющих управление вторичными (конструируемыми самим ПО) ресурсами, ориентированными на решение определенного (широкого) класса задач. Примеры: СУБД, модули управления языком интерфейса ИС, программы сбора и предварительной обработки информации. Также это комплекс технологического ПО для обеспечения взаимодействия между различными приложениями, системами, компонентами. Примеры: Веб-сервер, сервер приложений, сервисная шина, система управления контентом. С точки зрения инструментальных средств разработки промежуточное ПО ближе к прикладному, так как не работает напрямую с первичными ресурсами, а использует сервисы системного ПО. С точки зрения алгоритмов и технологий разработки промежуточное ПО ближе к системному, так как является сложным программным изделием многократного и многоцелевого использования. Современные тенденции развития ПО состоят в снижении объема как системного, так и прикладного программирования, и основная часть работы программистов выполняется в промежуточном ПО.
    - **Прикладное ПО**.
- **Системная программа** – это программа, предназначенная для поддержания работы СОИ или повышения эффективности ее использования. Например: операционные системы, файловые системы, драйверы, утилиты, системы программирования.
- **Классификация и функции системного ПО**:
    - **Функции системного ПО** включают:
        - Создание операционной среды функционирования для программ.
        - Автоматизация разработки новых программ.
        - Обеспечение надежной и эффективной работы компьютера и компьютерной сети.
        - Проведение диагностики и профилактики аппаратуры компьютера и компьютерных сетей.
        - Выполнение вспомогательных технологических процессов (копирование, архивирование, восстановление после сбоев и т.д.).
    - **Группы системного ПО**:
        - Операционные системы.
        - Интерфейсные оболочки (ОС).
        - Системы управления файлами.
        - Системы программирования.
        - Утилиты.
        - Драйверы.
        - Средства сетевого доступа.
    - **Классификация системного ПО** по назначению:
        - **Управляющие программы** – системные программы, реализующие набор функций, который включает управление ресурсами и взаимодействие с внешней средой СОИ, восстановление работы системы после неисправностей в технических средствах.
        - **Обслуживающие программы (утилиты)** – программы, предназначенные для оказания услуг общего характера пользователям и обслуживающему персоналу СОИ.

- **Классификация системного ПО** по минимальной необходимости:
  - **Базовое системное ПО** – минимальный набор программных средств, обеспечивающий работу компьютера и компьютерной сети. К нему относятся: Операционные системы, Интерфейсные оболочки (ОС), Системы управления файлами, Драйверы (также могут быть сервисными), Средства сетевого доступа.
  - **Сервисное системное ПО** – программы и программные комплексы, которые расширяют возможности базового ПО и организуют удобную среду для работы других программ и пользователя. К нему относятся: Системы программирования, Утилиты, Драйверы (также могут быть базовыми).

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 процесс разработки системных программ (в том числе, управляющих и обслуживающих). Также системное программирование – это разработка программ сложной структуры, поскольку система – это единое целое, состоящее из множества компонентов и связей между ними. Эти два определения не противоречат друг другу, так как разработка программ сложной структуры ведется именно для обеспечения работоспособности или повышения эффективности СОИ.
- **Система программирования** – это 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. В лабораторных работах будет использоваться система программирования на основе 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
  - Компилятор
  - Интерпретатор
  - Эмулятор
  - Перекодировщик
  - Макропроцессор
- **Этапы подготовки программы** и результаты работы:
  - **Исходный модуль** – программный модуль на исходном языке, обрабатываемый транслятором и представляемый для него как целое, достаточное для проведения трансляции.

- **Шаг первый – Предварительная обработка кода**: включает присоединение исходных файлов и работу макропроцессоров (например, через clang).
- **Шаг второй – Анализ**:
  - Лексический анализ (через clang).
  - Синтаксический анализ (через clang).
  - Семантический анализ (через clang).
- **Шаг третий – Синтез**:
  - Генерация машинно-независимого кода (через clang).
  - Оптимизация машинно-независимого кода (через clang).
  - Распределение памяти (через clang).
  - Генерация машинного кода (через clang).
  - Оптимизация машинного кода (через clang).
- **Результатом работы компилятора является объектный модуль**.
  - **Объектный модуль** – программный модуль, получаемый в результате трансляции исходного модуля. Его содержимое не содержит признаков, на каком языке был написан исходный модуль. Объектный модуль представляет собой программу на машинном языке. Поскольку транслятор обрабатывает только один конкретный модуль, он не может должным образом обработать части этого модуля, в которых запрограммированы обращения к данным или процедурам, определенным в другом модуле. Такие обращения называются **внешними ссылками**, и в объектном модуле они транслируются в промежуточную форму.
- **Разрешение внешних ссылок** выполняется на следующем этапе – **компоновки**, который обеспечивается **Редактором Связей (Компоновщиком)**. Он соединяет 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.

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

- **Clang** – это не компилятор в прямом смысле слова, а **«фронтенд»** для языка программирования C (для C++ – clang++). Под «фронтендом» в инфраструктуре **LLVM** понимается транслятор из некоторого языка программирования в промежуточный язык (LLVM IR), то есть непосредственная генерация объектного кода перекладывается на LLVM.
- **Преимущества использования Clang**:
  - Широко используемый «компилятор» в разработке ПО.
  - Поддерживается «большой тройкой» операционных систем и не только.
  - Большая инфраструктура, поддерживаемая open-source сообществом (компиляторы, отладчик, анализаторы кода, линтеры).
  - Язык LLVM IR является модульным, что позволяет более гибко разрабатывать ПО.

    - Поддержка «фронтендов» для множества популярных языков программирования.
    - Хорошая система вывода информации об ошибках в понятном для разработчика виде (за счёт особой внутренней реализации).
    - Производительность (предоставляет очень хорошую скорость работы конечного приложения).
    - Хорошая автоматическая оптимизация кода.
- **Система сборки** – это инструмент или набор инструментов, используемых для автоматизации процесса преобразования исходного кода программы в исполняемый файл или библиотеку. Основная задача системы сборки – управлять различными этапами этого процесса, такими как компиляция, компоновка, препроцессинг и другие задачи, связанные с подготовкой ПО к выполнению. Системы сборки позволяют описать процесс сборки на некотором внутреннем языке, команды которого вызывают соответствующие утилиты с заданными параметрами. Они играют важную роль, обеспечивая автоматизацию, надежность и повторяемость сборки, что особенно важно для крупных и сложных проектов. Также одна из важнейших причин их использования – поддержка проекта на нескольких платформах одновременно без необходимости вникать в тонкости сборки под каждую из них.
- **CMake** не является традиционной системой сборки; это **система мета-сборки**. **Система мета-сборки** – это инструмент, который генерирует файлы конфигурации для других систем сборки. В отличие от традиционных систем сборки, мета-сборки создают промежуточные файлы, которые затем используются другими инструментами для выполнения фактической сборки. **Основные особенности систем мета-сборки**:
    - **Генерация конфигураций**: создают файлы конфигурации (например, Makefile, Ninja файлы), которые затем используются другими инструментами для сборки проекта.
    - **Кроссплатформенность**: обеспечивают возможность генерации конфигураций для различных платформ и компиляторов, что делает их особенно полезными для кроссплатформенных проектов.
    - **Абстракция**: предоставляют более высокий уровень абстракции, позволяя разработчикам описывать процесс сборки в более общем виде, не привязываясь к конкретным инструментам сборки.
    - **Гибкость**: поддерживают различные сценарии сборки и позволяют легко изменять конфигурацию сборки.
- **Генератор** – это та система сборки, для которой будут сгенерированы файлы сборки. Генератор указывается при первичной инициализации проекта CMake через флаг `-G`.
- **Виды сборок в CMake**:
    - **In-source** – выходные файлы проекта будут располагаться в каталоге с исходными «source» файлами.
    - **Out-of-source** – выходные файлы проекта будут располагаться в отдельном от каталога с исходными «source» файлами месте (что делается, например, с помощью флага `-B` при инициализации CMake).
- **Мультиконфигурация** – это возможность настроить и генерировать файлы под несколько конфигураций (например, Debug и Release). Таких генераторов в настоящий момент только два: Visual Studio и XCode.
- **CMakeLists.txt** – это конфигурационный файл, который необходимо добавить в корневую папку проекта для генерации файлов сборки.
- **Структура CMakeLists для базового проекта** (на примере "Hello, World"):

- Пример файла `CMakeLists.txt` для проекта с одним исходным файлом `src/Source.c`:
- `cmake_minimum_required(VERSION 3.30)`
- `project(HelloWorld C)`
- `add_executable(Hello src/Source.c)`
- **`cmake_minimum_required(VERSION 3.30)`**: обозначает минимальную необходимую версию CMake для работы с проектом.
- **`project(HelloWorld C)`**: устанавливает имя проекта и указывает используемый язык (C).
- **`add_executable(Hello src/Source.c)`**: добавляет исполняемый файл `Hello` в проект, используя указанные исходные файлы (`src/Source.c`).
- Для указания используемого компилятора для языка C можно использовать параметр `-DCMAKE_C_COMPILER=<название компилятора>` (однако этот параметр может не работать со всеми генераторами, например, с Visual Studio, где может потребоваться флаг `-T` для указания инструментов сборки).

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 файлов в память (memory-mapped files)** – это механизм в современных ОС, который позволяет резервировать регион адресного пространства и передавать ему физическую память. Отличие от виртуальной памяти в том, что физическая память берется из файла, уже находящегося на диске, а не выделяется из страничного файла. Как только файл спроецирован в память, к нему можно обращаться так, будто он целиком в нее загружен.
- **Применение отображения файлов в память в ОС**:
  - **Загрузка и выполнение исполняемых файлов и библиотек**: позволяет существенно экономить как на размере страничного файла, так и на времени, необходимом для подготовки приложения к выполнению.
  - **Доступ к файлу данных, размещенному на диске**: позволяет обойтись без операций файлового ввода-вывода и буферизации его содержимого.
  - **Разделение данных между несколькими процессами**, выполняемыми на одной машине: в Windows другие методы совместного доступа к данным реализованы на основе проецируемых в память файлов.
- **Как устроен данный механизм в Windows**: Примитивы, на основе которых диспетчер памяти реализует общую память, называются **объектами секций**. В Windows API они представлены объектами отображенных файлов (объект «проекция файла»). Это важнейший примитив диспетчера памяти, используемый для отображения виртуальных адресов в основную память, в страничный файл или в любой другой файл, к содержимому которого приложение захочет обращаться как к находящемуся в памяти.
- **Классификация объектов секций в Windows**:
  - **Секции на базе файлов** – объект секции связан с открытым файлом на диске, и этот файл называется спроецированным (отображенным).
  - **Секции на базе страничных файлов** – объект секции связан с подтвержденной памятью (для реализации общей памяти); страницы

записываются в страничный файл (вместо отображенного файла), если возникнет необходимость в физической памяти. Общие подтвержденные страницы всегда заполняются нулями при первом обращении для предотвращения утечки конфиденциальных данных.

- **Представление секции (section view)** – это часть секции, которая фактически видна процессу. Процесс создания представления для секции называется отображением (проецированием) представления секции. Каждый процесс, манипулирующий содержимым секции, имеет свое собственное представление; процесс также может иметь несколько представлений. Отображение представлений позволяет процессам экономить адресное пространство, потому что в память отображаются только необходимые в данный момент представления объекта секции.
- **Согласованность доступа к данным в нескольких секциях**:
  - Система позволяет проецировать сразу несколько представлений одних и тех же файловых данных (например, первые 10 Кб файла в одно представление, а первые 4 Кб того же файла в другое).
  - **Если проецируется один и тот же объект**, система **гарантирует согласованность (coherence)** отображаемых данных. Изменение содержимого файла в одном представлении приводит к обновлению данных и в другом, так как система хранит данные на единственной странице оперативной памяти. Если представления одного и того же файла данных создаются несколькими процессами, данные по-прежнему сохраняют согласованность, будучи сопоставленными только с одним экземпляром каждой страницы в оперативной памяти.
  - **Однако Windows позволяет создавать несколько объектов «проекция файла», связанных с одним и тем же файлом данных**. В этом случае **не будет гарантий**, что содержимое представлений этих объектов согласованно. Такую гарантию Windows дает только для нескольких представлений *одного* объекта «проекция файла».
- **Алгоритм взаимодействия с файлами, отображенными в память, с использованием WinAPI**:
  1. **Создать или открыть объект ядра «файл»**, идентифицирующий дисковый файл, который вы хотите использовать как проецируемый в память. Для этого применяется функция **`CreateFile`**. Важные параметры: `pszFileName` (имя файла), `dwDesiredAccess` (способ доступа: `GENERIC_READ` или `GENERIC_READ | GENERIC_WRITE`), `dwShareMode` (тип совместного доступа).
  2. **Создать объект ядра «проекция файла»**, чтобы сообщить системе размер файла и способ доступа к нему. Для этого вызывается функция **`CreateFileMapping`**.
     - `hFile`: дескриптор файла, полученный от `CreateFile`. Для секций на базе страничных файлов устанавливается в `INVALID_HANDLE_VALUE`.
     - `fdwProtect`: желаемые атрибуты защиты (например, `PAGE_READWRITE`). Режимы доступа должны соответствовать режимам доступа к файлу. Существуют также флаги `PAGE_EXECUTE_*` для памяти с атрибутом исполнения кода.
     - `dwMaximumSizeHigh` и `dwMaximumSizeLow`: максимальный размер файла в байтах. Для текущего размера файла передайте нули. Если файл окажется меньше заданного при `PAGE_READWRITE`, `CreateFileMapping` увеличит его размер.
     - `pszName`: имя объекта «проекция файла» (NULL, если совместное использование не требуется).

3. **Указать системе, как отобразить в адресное пространство вашего процесса объект «проекция файла»** – целиком или частично. Это делается функцией **`MapViewOfFile`**.
    - `hFileMappingObject`: дескриптор объекта «проекция файла».
    - `dwDesiredAccess`: вид доступа к данным (например, `FILE_MAP_READ`, `FILE_MAP_WRITE`). Повторно указывается, как вы хотите обращаться к файловым данным.
    - `dwFileOffsetHigh`, `dwFileOffsetLow`: смещение в файле, откуда начинать отображение. Должно быть кратно гранулярности выделения памяти (64 КБ).
    - `dwNumberOfBytesToMap`: размер представления в байтах.
4. **Когда необходимость в данных файла отпадет, освободить регион** вызовом **`UnmapViewOfFile`**. Ее параметр `pvBaseAddress` указывает базовый адрес возвращаемого региона. Вызов этой функции обязателен, иначе регион не освободится до завершения процесса.
5. (Опционально) **Заставить систему записать измененные данные** в дисковый образ файла функцией **`FlushViewOfFile`**. Параметры указывают адрес и количество байтов для записи. Данные будут обновлены и без этого вызова во время свопинга страниц.
6. **Закрыть объекты «проекция файла» и «файл»**. Для этого дважды вызывается функция **`CloseHandle`**. При работе с файлами, отображенными в память, можно закрыть описатели «файл» и «проекция файла» сразу после вызова `MapViewOfFile`, так как система увеличивает счетчики числа пользователей этих объектов.

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 файлов в память (memory-mapped files)** – это механизм в современных ОС, который позволяет резервировать регион адресного пространства и передавать ему физическую память, взятую из файла на диске. К отображенному файлу можно обращаться так, будто он целиком загружен в память. **Применение в ОС**: используется для загрузки и выполнения исполняемых файлов и библиотек, доступа к файлам данных без операций ввода/вывода, а также для разделения данных между процессами.
- **Как устроен данный механизм согласно POSIX (Linux)**:
    - В ОС семейства Linux нет дополнительной сложности в виде объектов секций; объект с диска отображается в память ядром напрямую.
    - Отображения могут производиться для двух типов объектов:
        - **Обычный файл в файловой системе Linux**: секция файла разделена на фрагменты размером со страницу, каждый из которых содержит исходное содержимое виртуальной страницы. Страницы фактически не загружаются в физическую память до первого обращения (подкачка по требованию). Если область памяти больше, чем секция файла, часть, выходящая за рамки размеров файла, заполняется нулями (файл при этом не расширяется).
        - **Анонимный файл**: область памяти сопоставляется с анонимным файлом, созданным ядром и заполненным нулями. При первом

обращении ядро находит страницу-жертву в физической памяти, выгружает её (если `dirty`), перезаписывает нулями и обновляет таблицу страниц. Данные не передаются между диском и памятью (demand-zero pages).

  - Участок памяти в отображении одного процесса можно разделять с отображением другого процесса (записи таблицы страниц указывают на одни и те же страницы физической памяти). Это достигается, когда два процесса отображают один и тот же участок файла, или когда дочерний процесс, созданный `fork`, наследует копии родительских отображений.
  - Изменения, вносимые в разделяемые страницы, видны другим процессам, но это зависит от того, является ли отображение приватным или разделяемым.
- **Что такое Copy on Write (Копирование при записи)**: В ситуации с `fork` страницы приватного отображения изначально являются разделяемыми. Однако, когда процесс пытается изменить содержимое страницы, ядро создает для него ее копию (и корректирует таблицу страниц процесса). Это гарантирует, что изменения одного процесса остаются невидимыми для других, и изменения не передаются в исходный файл. Поэтому отображение `MAP_PRIVATE` иногда называют приватным, копируемым при записи.
- **Классификация отображений согласно POSIX**:
  - **Приватное файловое отображение (`MAP_PRIVATE` с файлом)**: Содержимое инициализируется участком файла. Несколько процессов, отображающих один файл, изначально разделяют одни и те же страницы памяти. Благодаря копированию при записи, изменения, вносимые процессом, не видны остальным и не передаются в исходный файл. Основное применение – инициализация участка памяти содержимым файла, например, для сегментов кода и данных исполняемого файла или разделяемой библиотеки.
  - **Приватное анонимное отображение (`MAP_PRIVATE` с `MAP_ANONYMOUS`)**: Каждый вызов `mmap` создает новое отображение с отдельными страницами памяти. Хотя дочерний процесс наследует отображения родителя, процедура копирования при записи гарантирует, что после `fork` родитель и потомок не будут видеть изменения друг друга. Основное применение – выделение (и заполнение нулями) памяти для процесса, например, при выделении больших блоков памяти `malloc` использует `mmap`.
  - **Разделяемое файловое отображение (`MAP_SHARED` с файлом)**: Все процессы, отображающие один участок того же файла, разделяют одни и те же страницы физической памяти. Изменения, вносимые в отображение, передаются обратно в файл. Используется для отображения ввода/вывода в память (изменения в памяти автоматически записываются в файл, альтернатива `read`/`write`) и для межпроцессного взаимодействия неродственных процессов.
  - **Разделяемое анонимное отображение (`MAP_SHARED` с `MAP_ANONYMOUS`)**: Каждый вызов `mmap` создает новое отображение со своими отдельными страницами памяти. К страницам отображения не применяется процедура копирования при записи. После `fork` родитель и дочерний процессы разделяют одни и те же страницы, и изменения, вносимые одним, видны другому. Позволяет организовать межпроцессное взаимодействие между родственными процессами.
- **Алгоритм взаимодействия с файлами, отображенными в память, с использованием POSIX API**:

1. **Получить дескриптор файла** (обычно с помощью вызова **`open`**). Параметры: `pathname` (путь к файлу) и `flags` (битовая маска режима доступа, например, `O_RDONLY`, `O_RDWR`).
2. **Передать этот файловый дескриптор вызову `mmap`** в виде аргумента `fd`. Системный вызов **`mmap`** создает новое отображение в виртуальном адресном пространстве вызывающего процесса.
   - `addr`: виртуальный адрес, по которому будет находиться отображение (предпочтительно `NULL` для выбора ядром).
   - `length`: размер отображения в байтах (округляется до ближайшего кратного размеру страницы).
   - `prot`: битовая маска защиты отображения (`PROT_NONE`, `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`).
   - `flags`: битовая маска, управляющая аспектами работы отображения, обязательно включает `MAP_PRIVATE` или `MAP_SHARED`. Также может быть `MAP_ANONYMOUS`.
   - `fd`: файловый дескриптор файла.
   - `offset`: начальная позиция отображения в файле (должен быть кратен размеру страницы).
3. **Отменить отображение файла** функцией **`munmap`**. Аргументы: `addr` (начальный адрес участка памяти) и `length` (размер участка).
4. (Опционально) **Синхронизировать разделяемое отображение** с отображенным файлом функцией **`msync`**. Ядро автоматически возвращает изменения, но `msync` дает приложению контроль над синхронизацией.
5. **Закрыть объект ядра «файл»** функцией **`close`**.

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 объектов (или объектная библиотека)** – это файл, содержащий несколько объектных файлов, которые будут использоваться вместе на стадии сборки (связывания, линковки) программы. Нормальная библиотека содержит символьный индекс, состоящий из названий функций, переменных и т.д., что позволяет ускорить процесс сборки программы.
- **Причина возникновения библиотек**:
  - Простые программы состоят из одного исходного файла, но большие программы сталкиваются с проблемами: большой файл увеличивает время компиляции, малейшие изменения требуют перекомпиляции.
  - Сложно отслеживать изменения, если над программой работает много людей.
  - Процесс правки и ориентирование в большом исходном тексте становится сложным, поиск ошибок требует «изучения» кода заново.
  - Часто одни и те же исходные файлы можно было бы использовать в нескольких программах, и их однократная компиляция экономит время, но все равно требует указания всех объектных файлов на этапе компоновки, что может вызвать неразбериху.
  - Для решения этих проблем набор объектных файлов группируется в единую сущность – библиотеку.
- **Какие бывают библиотеки**:

    - **Статические библиотеки**.
    - **Динамические (разделяемые) библиотеки**.
- **Связывание (компоновка)** – это функция, состоящая в компоновке программы из многих объектных модулей. Поскольку каждый объектный модуль получается в результате отдельной трансляции, которая работает только с этим модулем, обращения к процедурам и данным в других модулях не содержат актуальных адресов. Загрузчик же «видит» все объектные модули и может вставить в обращения к внешним точкам правильные адреса.
- **Виды связывания**:
    - **Раннее (статическое) связывание (`compile-time`, `link-time`)**:
        - **Во время трансляции**: особый случай, поддерживается, например, компилятором MSVC через директиву препроцессора `pragma`.
        - **Во время сборки (`link-time`)**: весь объектный код, содержащийся в статической библиотеке, внедряется в будущий исполняемый файл на этапе компоновки.
    - **Позднее (динамическое) связывание (`run-time`)**:
        - **При загрузке (неявное)**: образ файла DLL проецируется на адресное пространство вызывающего процесса при его запуске. Система ищет DLL в определенной последовательности каталогов (каталог EXE-файла, текущий каталог, системный каталог Windows, основной каталог Windows, PATH).
        - **Отложенное (императивное/явное) связывание (`on-demand`)**: поток явно загружает DLL в адресное пространство процесса во время выполнения приложения, получает виртуальный адрес необходимой DLL-функции и вызывает ее по этому адресу. Это позволяет загружать нужный код динамически, когда он требуется.
- **Соотношение видов связывания с типами библиотек**:
    - **Статическая библиотека** использует **раннее (статическое) связывание**.
    - **Динамические (разделяемые) библиотеки** (DLL в Windows, shared objects в Linux) используют **позднее (динамическое) связывание**.

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 помещенных в него объектных файлов. В архиве также хранятся различные атрибуты для каждого объектного файла. В Unix-подобных системах им принято давать имена вида `libname.a`, в Windows каких-либо общепринятых правил наименования нет.
- В основе статических библиотек лежит **раннее (статическое) связывание**. В результате такого связывания **весь объектный код, содержащийся в библиотеке, внедряется в будущий исполняемый файл на этапе компоновки или трансляции**.
- **Создание статической библиотеки используя clang напрямую** (на примере Windows, идентично для Linux с `llvm-ar` или `ar`):

1. Скомпилировать исходные файлы (`TestA.c`, `TestW.c`) в объектные файлы (`TestA.obj`, `TestW.obj`):
2. `clang -c TestA.c TestW.c`
3. Использовать утилиту `llvm-ar` (или `ar` в Linux) для создания статической библиотеки из объектных файлов:
4. `llvm-ar r TestLib.lib TestA.obj TestW.obj`
    - `ar <operation>[modifiers] <archive> [files].`
    - `r` (replace) – вставляет объектный файл в архив, заменяя существующий.
    - `c` (create) – используется с `r` для создания библиотеки, если ее нет.
    - `t` (table of contents) – выводит оглавление архива (с `v` для подробного вывода).
    - `d` (delete) – удаляет модуль из архива.
5. (Необязательно, но рекомендуется для гарантии) Использовать утилиту `llvm-ranlib` (или `ranlib`) для добавления индекса символов в библиотеку, хотя `llvm-ar` обычно делает это по умолчанию.
6. `llvm-ranlib TestLib.lib`

- **Создание статической библиотеки используя CMake**:
    - В файле `CMakeLists.txt` для проекта с библиотекой (`LibProject/src/CMakeLists.txt`):
    - `add_library(TestLib STATIC TestA.c TestW.c)`
        - `add_library` – команда для указания, что необходимо собрать библиотеку из указанных файлов (в данном случае статическую, поскольку указан параметр `STATIC`).
    - В корневом `CMakeLists.txt` проекта:
    - `add_subdirectory(LibProject)`
        - `add_subdirectory()` – добавляет другой каталог в сборку.
- **Как собрать приложение с использованием статических библиотек используя clang напрямую**:
- `clang Test-client.c TestLib.lib -o Test-client.exe`
    - **Очень важен порядок**, в котором передаются входные файлы драйверу компилятора (clang): **СНАЧАЛА ФАЙЛЫ ПРИЛОЖЕНИЯ, ЗАТЕМ БИБЛИОТЕКИ!**. Это правило исходит из необходимости сначала передать код, в котором используется функция, а только потом тот, в котором она объявлена.
- **Как собрать приложение с использованием статических библиотек используя CMake**:
    - В файле `CMakeLists.txt` для проекта с приложением (`MainProject/src/CMakeLists.txt`):
    - `add_executable(TestClient Test-client.c)`
    - `target_link_libraries(TestClient TestLib)`
        - `target_link_libraries` – команда для связывания приложения с библиотекой (чаще всего используется для библиотек, которые собираются в этом же проекте, но можно и для внешних). Также через нее можно передавать и другие параметры компоновщику.
- **Преимущества статических библиотек**:
    - Можно поместить набор часто используемых объектных файлов в единую библиотеку, которую потом можно применять для сборки разных программ, не перекомпилируя оригинальные исходные тексты.
    - Упрощаются команды для компоновки: вместо перечисления длинного списка объектных файлов можно указать имя статической библиотеки, и компоновщик сам найдет и извлечет нужные объекты.

- **Недостатки статических библиотек**:
    - Итоговый исполняемый файл содержит копии всех скомпонованных объектных модулей, что приводит к избыточности и значительным потерям дискового пространства.
    - Если несколько программ, использующих одни и те же модули, выполняются одновременно, каждая из них хранит в виртуальной памяти свою отдельную копию этих модулей, увеличивая потребление виртуальной памяти.
    - Если объектный модуль статической библиотеки требует изменений (например, исправление ошибки или дыры в безопасности), придется заново компоновать все исполняемые файлы, в которых этот модуль используется. Системному администратору необходимо знать, с какими приложениями скомпонована библиотека.

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.

Что такое статическая библиотека? Статическая библиотека – это обычный файл, содержащий копии всех помещенных в него объектных файлов. Она также хранит различные атрибуты для каждого объектного файла, включая права доступа, числовые идентификаторы пользователя и группы, а также время последнего изменения. В Unix-подобных системах статическим библиотекам принято давать имена вида `libname.a`, тогда как в Windows каких-либо общепринятых правил наименования статических библиотек нет. **Библиотека объектных файлов** – это файл, содержащий несколько объектных файлов, которые будут использоваться вместе на стадии сборки (связывания, линковки) программы. Нормальная библиотека содержит символьный индекс, состоящий из названий функций, переменных и т.д., которые содержатся в библиотеке, что ускоряет процесс сборки программы.

Какое связывание лежит в основе статических библиотек? В основе статических библиотек лежит **раннее (статическое) связывание**, которое происходит на этапе компоновки или трансляции. В результате такого связывания весь объектный код, содержащийся в библиотеке, внедряется в будущий исполняемый файл.

Как создать статическую библиотеку используя clang напрямую? Для создания и редактирования статических библиотек используется утилита `ar` или `llvm-ar`. Форма использования: `ar <operation>[modifiers] <archive> [files]`. Часто используемые операции и модификаторы:

- **`r`** (replace) – вставляет объектный файл в архив, заменяя существующий файл с тем же именем. Используется с модификатором `c` для создания библиотеки, если её нет.
- **`t`** (table of contents) – выводит оглавление архива. С модификатором `v` (verbose) показывает все атрибуты каждого файла в архиве.
- **`d`** (delete) – удаляет заданный модуль из архива. Утилита `llvm-ranlib` (или `ranlib`) позволяет добавить к библиотеке индекс символов, т.е. список вложенных функций

и переменных. `llvm-ar` (и `ar`) обычно индексирует по умолчанию, но рекомендуется явно вызывать `llvm-ranlib` для гарантии индексации.

Как создать статическую библиотеку используя CMake? Для создания статической библиотеки с помощью CMake используется команда `add_library()` с параметром `STATIC`. Например, `add_library(Hello STATIC src/Source.c)`.

Как собрать приложение с использованием статических библиотек используя clang напрямую? При сборке приложения с использованием статических библиотек с помощью драйвера компилятора (например, `clang`), очень важен **порядок передачи входных файлов**: сначала файлы приложения, затем библиотеки. Это связано с необходимостью сначала передать код, в котором используется функция, а затем тот, в котором она объявлена.

Как собрать приложение с использованием статических библиотек используя CMake? Для связывания приложения со статической библиотекой с помощью CMake используется команда `target_link_libraries()`. Например, `target_link_libraries(Hello mylib)`. Эта команда чаще всего используется для библиотек, собираемых в том же проекте, но может применяться и для внешних библиотек.

Преимущества и недостатки статических библиотек. Преимущества:

- Можно поместить набор часто используемых объектных файлов в единую библиотеку, которую затем можно применять для сборки разных программ, не перекомпилируя оригинальные исходные тексты.
- Упрощаются команды для компоновки: вместо перечисления длинного списка объектных файлов можно указать только имя статической библиотеки. Компоновщик знает, как выполнять поиск по ней и извлекать необходимые объекты.

Недостатки:

- **Избыточность дискового пространства:** несколько разных программ могут содержать копии одних и тех же объектных модулей, что приводит к значительным потерям дискового пространства.
- **Избыточность использования виртуальной памяти:** если несколько программ, использующих одни и те же модули, выполняются одновременно, каждая из них будет хранить в виртуальной памяти свою отдельную копию этих модулей, увеличивая потребление виртуальной памяти.
- **Сложность обновлений:** если объектный модуль статической библиотеки требует изменений (например, исправление ошибки или дыры в безопасности), придется заново компоновать все исполняемые файлы, в которых этот модуль используется. 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.

Что такое разделяемая (динамическая) библиотека? Разделяемые (динамические) библиотеки – это библиотеки, разработанные для устранения недостатков статических библиотек, таких как избыточность дискового пространства и виртуальной памяти. В операционной системе Windows такие библиотеки называются Dynamic Link Library (DLL), а в операционных системах семейства Linux – разделяемые объекты (shared objects, SO). DLL-файл представляет собой файл в формате Portable Executable (PE), а SO-файл – в формате Executable and Linkable Format (ELF).

В чем ключевая идея таких библиотек? Ключевая идея разделяемых библиотек состоит в том, что **одна копия объектного модуля разделяется между всеми программами, задействующими его**. Объектные модули не копируются в компонуемый исполняемый файл. Вместо этого единая копия библиотеки загружается в память при запуске первой программы, которой требуются её объектные модули. Если позже будут запущены другие программы, использующие эту библиотеку, они обращаются к уже загруженной в память копии.

Какой механизм ОС лежит в основе работы разделяемых библиотек? В основе работы разделяемых библиотек лежит механизм **проецирования (отображения) файлов в память**. Чтобы приложение (или другая DLL) могло вызывать функции, содержащиеся в DLL, образ её файла нужно сначала спроецировать на адресное пространство вызывающего процесса. Это достигается либо за счет неявного связывания при загрузке, либо за счет явного – в период выполнения.

Какие способы подключения разделяемых библиотек существуют? Существует два основных способа подключения разделяемых библиотек:

- **Неявное связывание при загрузке** (отложенное декларативное связывание).
- **Явное связывание в период выполнения** (отложенное императивное связывание).

Как создать разделяемую библиотеку используя clang напрямую? Источники предоставляют примеры команд для создания DLL и SO файлов напрямую:

- Для Windows DLL: `clang -shared TestA.c TestW.c -o mylib.dll`.
- Для Linux SO: `clang -shared mod1.c mod2.c mod3.c -o libmod.so`.

Как создать разделяемую библиотеку используя CMake? Для создания разделяемой библиотеки с помощью CMake используется команда `add_library()` с параметром `SHARED`. Например, `add_library(mylib SHARED ${SOURCES})`.

Преимущества и недостатки разделяемых библиотек. Преимущества (на примере DLL в Windows):

- **Расширение функциональности приложения:** DLL можно загружать динамически, позволяя приложению подгружать нужный код.
- **Возможность использования разных языков программирования:** Приложение на одном языке (например, C#) может загружать DLL, написанные на других языках (C++, Кобол, Фортран и др.).

- **Более простое управление проектом:** При разработке программного продукта отдельными группами с использованием DLL управлять проектом проще.
- **Экономия памяти:** Если одну и ту же DLL использует несколько приложений, в оперативной памяти может храниться только один её экземпляр.
- **Разделение ресурсов:** DLL могут содержать общие ресурсы, такие как шаблоны диалоговых окон, строки, значки, доступные любым программам.
- **Решение проблем, связанных с особенностями различных платформ:** Позволяет запускать приложения на более ранних версиях Windows, даже если новые функции находятся в отдельной DLL.
- **Упрощение локализации:** DLL часто применяются для локализации приложений, содержа только компоненты пользовательского интерфейса.
- **Реализация специфических возможностей:** Определенная функциональность в Windows доступна только при использовании DLL, например, отдельные виды ловушек или создание COM-объектов.

Недостатки (неявно следуют из преимуществ статических библиотек, которые были перечислены в вопросе 7):

- Необходимость наличия самой DLL для запуска приложения.
- Возможные проблемы с версиями DLL (DLL Hell), когда разные приложения требуют разные версии одной и той же библиотеки.
- Задержка при запуске приложения из-за необходимости динамической загрузки библиотеки.

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?

Что такое неявный способ подключения разделяемой (динамической) библиотеки?
Неявный способ подключения DLL, также известный как **отложенное декларативное связывание**, означает, что приложение подключается к DLL при загрузке. Для этого компоновщик, при сборке исполняемого файла, использует специальный файл, называемый библиотекой импорта, которая содержит ссылки на функции и переменные, экспортируемые из DLL.

Опишите алгоритмы неявного связывания с использованием clang и CMake.
Алгоритм неявного связывания (общая картина):

1. **Компиляция исходных файлов библиотеки** в объектные файлы.
2. **Сборка DLL/SO файла** из объектных файлов. При этом создается **библиотека импорта** (для Windows DLL), содержащая ссылки на экспортируемые имена.
3. **Компиляция исходных файлов приложения** в объектные файлы.
4. **Компоновка исполняемого файла приложения** с объектными файлами приложения и библиотекой импорта DLL (или напрямую с SO для Linux), в результате чего в исполняемом файле создается **таблица импорта**.
5. **Запуск приложения.** Динамический загрузчик ОС использует таблицу импорта для определения необходимых DLL/SO и их загрузки в адресное пространство процесса, а также для разрешения ссылок на экспортируемые функции.

Использование clang напрямую (пример Windows):

- Создание DLL: `clang -shared TestA.c TestW.c -o mylib.dll -Wl,--out-implib,mylib.lib` (ключ `-Wl,--out-implib` используется для генерации библиотеки импорта `mylib.lib`).
- Создание EXE: `clang test.c mylib.lib -o test.exe` (связывание с библиотекой импорта).
- Для Linux SO: `clang -shared mod1.c mod2.c mod3.c -o libmod.so`.
- Сборка программы с SO: `clang prog.c libmod.so -o prog`.

Использование CMake:

- В `LibProject/src/CMakeLists.txt`: `add_library(mylib SHARED ${SOURCES})`.
- В `MainProject/src/CMakeLists.txt`: `target_link_libraries(myapp PRIVATE mylib)`. CMake автоматически генерирует необходимые библиотеки импорта и управляет процессом связывания.

Какое связывание лежит в основе неявного подключения? В основе неявного подключения лежит **раннее (compile-time) связывание** или **отложенное декларативное связывание**.

Что такое библиотека импорта? Библиотека импорта (LIB-файл в Windows) – это файл, создаваемый компоновщиком при сборке DLL. Он содержит список идентификаторов, экспортируемых из DLL. Этот LIB-файл необходим при сборке любого EXE-модуля, ссылающегося на такие идентификаторы. В библиотеке импорта содержатся ссылки на все экспортируемые из динамической библиотеки имена.

Что такое раздел экспорта? Раздел экспорта – это таблица, которую компоновщик вставляет в конечный DLL-файл. В нём содержится список (в алфавитном порядке) идентификаторов экспортируемых функций, переменных и классов. Туда же помещается относительный виртуальный адрес (RVA) каждого идентификатора внутри DLL-модуля. У каждой функции в таблице экспорта есть порядковый номер (ordinal) и имя (name).

Какие способы экспорта функций существуют? Как их реализовать? Существуют два способа экспорта функций:

1. **По имени:** Достаточно применить модификатор `__declspec(dllexport)` перед переменной, прототипом функции или C++-классом. Компилятор C/C++ встраивает в OBJ-файл дополнительную информацию, которая понадобится компоновщику для создания LIB-файла и таблицы экспорта.
2. **По порядковому номеру:** Для этого нужно использовать **.DEF-файл (файл определений)**. Хотя порядковые номера присваиваются всем экспортируемым символам и без DEF-файла, он позволяет управлять экспортом по порядковому номеру.
    - **Преимущества .DEF-файла:**
        - Позволяет настраивать экспорт по порядковому номеру.
        - Позволяет указать конечное имя для функции в таблице экспорта (полезно для избежания `name mangling` для языков, отличных от C).
        - Позволяет настраивать область видимости экспортируемых функций.
        - Позволяет реализовать экспорт только с порядковыми номерами (без имён), что может сэкономить ресурсы.

- **Структура записи в .DEF-файле:** `entryname [=internalname] [@ordinal [NONAME]] [DATA] [PRIVATE].`

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?

Что такое явный способ подключения разделяемой (динамической) библиотеки? Явный способ подключения DLL (также известный как **отложенное императивное связывание**) — это метод, при котором поток приложения явно загружает DLL в адресное пространство процесса в период выполнения, получает виртуальный адрес необходимой DLL-функции и вызывает её по этому адресу. Это позволяет всему происходить в уже выполняющемся приложении.

Опишите алгоритмы явного связывания с использованием clang и CMake. Алгоритм явного связывания:

1. **Компиляция исходных файлов библиотеки** в DLL/SO файл. При этом **не создается библиотека импорта**.
2. **Компиляция исходных файлов приложения** в исполняемый файл **без указания библиотеки импорта или разделяемой библиотеки**.
3. **Загрузка DLL/SO в адресное пространство процесса** во время выполнения с помощью функций API ОС (например, `LoadLibrary` в Windows или `dlopen` в Linux).
4. **Получение адреса конкретной функции** из загруженной библиотеки с помощью функций API ОС (например, `GetProcAddress` в Windows или `dlsym` в Linux).
5. **Вызов функции** по полученному адресу.
6. **Выгрузка DLL/SO** из адресного пространства процесса после завершения работы с ней (например, `FreeLibrary` в Windows или `dlclose` в Linux).

Использование clang напрямую:

- Создание DLL: `clang -shared MyLib.c -o MyLib.dll`.
- Создание EXE: `clang test.c -o test.exe` (без связывания с `MyLib.lib`).
- Для Linux SO: `clang -shared mod1.c mod2.c mod3.c -o libmod.so`.
- Сборка программы с SO: `clang test.c -o test` (без явного указания `libmod.so`).

Использование CMake: CMake не используется для прямого управления явным связыванием в том же смысле, как для неявного. CMake управляет процессом сборки (компиляцией и компоновкой) приложения и библиотеки, но **вызовы API для явной загрузки и выгрузки библиотеки осуществляются непосредственно в коде приложения**.

Какое связывание лежит в основе явного подключения? В основе явного подключения лежит **позднее (run-time) связывание** или **отложенное императивное связывание**.

Что такое раздел импорта? При явной сборке, в отличие от неявной, **в таблице импорта исполняемого файла не будет информации о динамически загружаемой библиотеке**,

так как компоновщик ничего не знает о ней на этапе сборки. Раздел импорта (imports section) создаётся в EXE-модуле при неявном связывании и содержит список DLL, необходимых этому модулю, и идентификаторы, на которые есть ссылки.

Что такое name mangling? Как его избежать? 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, особенно для языков, отличных от C (например, C++). Компилятор C обычно не делает этого, сохраняя имена функций. **Как избежать name mangling:**

- Использовать **.DEF-файл (файл определений)**, который позволяет оставить имена функций в том виде, который понятен человеку.
- Использовать модификатор **`extern "C"`** перед объявлением функций, чтобы указать компилятору, что функция должна иметь "C" связывание, предотвращая name mangling.

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 очистки разделяемой (динамической) библиотеки в/из памяти. Функции `dlopen`/`LoadLibrary` могут быть вызваны несколько раз для одной и той же библиотеки. Загрузка будет выполнена лишь при первом вызове, а во всех последующих случаях будет возвращаться одно и то же значение HANDLE/дескриптор. Программный интерфейс `dlopen`/`LoadLibrary` хранит **счётчик ссылок** для каждого дескриптора. С каждым вызовом `dlopen`/`LoadLibrary` счётчик инкрементируется, а декрементация происходит при вызове `dlclose`/`FreeLibrary`. Библиотека выгружается из памяти только в том случае, если счётчик равен 0.

Функции жизненного цикла разделяемых библиотек в Windows и Linux.

- **Windows:**
  - **`LoadLibrary`** или **`LoadLibraryEx`**: загружают DLL в адресное пространство вызывающего процесса.
  - **`FreeLibrary`**: выгружает DLL из адресного пространства процесса.
  - **`GetProcAddress`**: получает адрес экспортируемого идентификатора (функции или переменной) из явно загруженной DLL.
  - **Функция входа/выхода (DllMain)**: может быть указана в DLL и вызывается системой в некоторых ситуациях (например, при загрузке в процесс, завершении процесса, создании/завершении потока) сугубо в информационных целях, обычно для инициализации и очистки ресурсов.
- **Linux (POSIX):**
  - **`dlopen`**: загружает разделяемую библиотеку (SO) в виртуальное адресное пространство вызывающего процесса и инкрементирует счётчик открытых ссылок. Если библиотека зависит от других, `dlopen` загрузит их автоматически.
  - **`dlclose`**: закрывает библиотеку.

- **`dlsym`**: ищет именованный символ (функцию или переменную) в библиотеке и её дереве зависимостей.
- **`dlerror`**: позволяет узнать причину ошибки при получении ошибки из `dlopen` или других функций `dlopen` API.
- **Функции-конструкторы и функции-деструкторы:** В Linux в SO не может быть функции DllMain, но можно определить одну или несколько функций, которые будут автоматически вызываться при загрузке и выгрузке разделяемой библиотеки.

Что такое DLL Injection? DLL Injection – это техника, используемая для запуска кода в адресном пространстве другого процесса путём принуждения загрузки в него DLL.

Алгоритм внедрения DLL в Windows с помощью удаленных потоков. Внедрение DLL с помощью удаленных потоков предполагает вызов функции `LoadLibrary` потоком целевого процесса для загрузки нужной DLL. Последовательность операций:

1. **Выделить блок памяти** в адресном пространстве удаленного процесса с помощью `VirtualAllocEx`.
2. **Скопировать строку с полным именем файла DLL** в этот выделенный блок памяти, используя `WriteProcessMemory`.
3. **Получить истинный адрес функции `LoadLibraryA` или `LoadLibraryW`** внутри `Kernel32.dll` с помощью `GetProcAddress`.
4. **Создать поток в удаленном процессе** с помощью `CreateRemoteThread`, который вызовет соответствующую функцию `LoadLibrary`, передав ей адрес блока памяти с именем DLL. На этом этапе DLL внедрена, и её функция `DllMain` получает уведомление `DLL_PROCESS_ATTACH`.
5. (Для очистки после завершения удаленного потока): **Освободить блок памяти**, выделенный в п. 1, вызовом `VirtualFreeEx`.
6. (Для очистки): **Определить истинный адрес функции `FreeLibrary`** внутри `Kernel32.dll` с помощью `GetProcAddress`.
7. (Для очистки): **Создать в удаленном процессе поток** с помощью `CreateRemoteThread`, который вызовет `FreeLibrary` с передачей адреса внедренной DLL.

Алгоритм внедрения SO в Linux. В Linux внедрение кода может быть реализовано с помощью переменной среды **`LD_PRELOAD`**. Ей можно присвоить строку с именами разделяемых библиотек, которые следует загрузить раньше других. Поскольку эти библиотеки загружаются в первую очередь, их функции, запрашиваемые программой, будут использоваться автоматически, переопределяя любые одноимённые символы. По соображениям безопасности программы, устанавливающие пользовательские и групповые идентификаторы, игнорируют переменную `LD_PRELOAD`.

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.

Что такое реестр Windows? 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 её приложениями.

Каковы причины его возникновения? До появления Windows NT приложения и операционная система Windows 3.x и DOS использовали множество разрозненных файлов инициализации (INI-файлы, Config.SYS, Autoexec.BAT) для хранения параметров конфигурации и пользовательских настроек. Эти файлы записывались в ASCII-формате, были подвержены повреждениям, случайному удалению, и даже невинные изменения могли привести к нарушению работы всей системы, что создавало "вечную головную боль программистов". С появлением Windows NT реестр пришел на смену этим файлам, хотя некоторые приложения до сих пор используют INI-файлы.

В каких случаях стоит использовать реестр?

- **Рекомендуется не использовать Реестр для хранения данных, связанных с приложениями или пользователями.** Вместо этого приложения должны хранить информацию в файловой системе (например, в форматах INI, XML, JSON, YAML).
- **Для системной разработки под ОС Windows** рано или поздно придётся столкнуться с Реестром.
- Иногда бывает удобно хранить **небольшую информацию** в Реестре, например, путь к файлу, тогда как основная часть информации хранится в самом файле.

Каких видов бывают данные в реестре? Данные в реестре бывают двух видов:

- **Энергозависимые (volatile)** – создаются во время работы системы и удаляются из Реестра при её завершении.
- **Энергонезависимые (non-volatile)** – данные, которые сохраняются в файлах на постоянном накопителе (HDD, SSD).

Опишите структуру реестра. Реестр разделён на **ульи (hives)** или **кусты**, в каждом из которых содержится определённая информация. Внутри ульев доступ к реестру осуществляется через **разделы, или ключи, реестра (registry keys)**, играющие роль каталогов в файловой системе. Раздел может содержать подразделы, параметры или параметр по умолчанию. Каждый **параметр** имеет три части:

1. **Имя**.
2. **Тип**.
3. **Значение**.

Какие типы данных поддерживаются в реестре? Записи реестра, называемые параметрами, могут содержать данные различных 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 предыдущему.

- `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_RESOURCE_LIST`, `REG_FULL_RESOURCE_DESCRIPTOR`, `REG_RESOURCE_REQUIREMENTS_LIST` – полезны только в режиме ядра.
- `REG_QWORD` – 64-битное значение (Little Endian).

Каковы ограничения?

- **Имя раздела (подраздела)**: максимальная длина 255 символов (полный путь, начиная от названия улья).
- **Имя параметра**: 16 383 Unicode символа.
- **Значение параметра**: 1 МБ в стандартном варианте, в последних версиях Windows – может быть использована вся доступная память.
- **Глубина дерева разделов**: до 512 уровней, при возможности создать 32 уровня за один API вызов.
- При хранении путей к файлам в реестре, все символы `\` должны быть экранированы, т.е. записываться как `\\`.
- Длинные значения (более 2048 байт) должны храниться в файле, а местоположение файла должно быть сохранено в реестре.

Назовите пять основных ульев и опишите их назначение. Хотя RegEdit показывает 5 ульев, "реальные" из них только два: `HKEY_USERS` и `HKEY_LOCAL_MACHINE`. Остальные состоят из комбинации данных в этих двух "реальных" ульях.

1. **HKEY_LOCAL_MACHINE (HKLM)**: Хранит информацию обо всем компьютере, которая не относится к какому-либо конкретному пользователю. Большая часть данных важна для правильного запуска системы. Только пользователи уровня администратора могут вносить изменения. Содержит важные подразделы, такие как `SOFTWARE` (для приложений) и `SYSTEM` (для системных параметров, служб, драйверов).
2. **HKEY_USERS**: Хранит всю информацию о каждом пользователе, который когда-либо входил в локальную систему. Каждый пользователь представлен своим SID'ом.
3. **HKEY_CURRENT_USER (HKCU)**: Является ссылкой на информацию из улья `HKEY_USERS` для текущего пользователя, который запустил RegEdit. Данные из этого улья сохраняются в скрытом файле `NtUser.dat`.
4. **HKEY_CLASSES_ROOT (HKCR)**: Состоит из комбинации `HKEY_LOCAL_MACHINE\Software\Classes` и `HKEY_CURRENT_USER\Software\Classes`. Содержит данные оболочки Explorer (типы файлов и ассоциации, расширения оболочки) и информацию, связанную с объектной моделью компонента (COM). Настройки `HKEY_CURRENT_USER` переопределяют `HKEY_LOCAL_MACHINE` в случае конфликта.
5. **HKEY_CURRENT_CONFIG (HKCC)**: (Не упомянут явно в этом списке в источниках, но является одним из пяти ульев, видимых в RegEdit). Обычно содержит информацию о текущем аппаратном профиле.

Опишите API для работы с реестром. API для работы с реестром Windows предоставляет функции для CRUD (Create, Read, Update, Delete) операций. Windows работает с реестром через аналог дескрипторов, называемый **HKEY**, который по сути

является дескриптором раздела Реестра. При работе с HKEY применяются те же правила, что и с HANDLE: открывать разделы только при необходимости и закрывать дескрипторы, когда они больше не нужны.

Основные функции API:

- **Создание или открытие раздела:**
  - `RegOpenKeyEx`: Открывает существующий раздел. Если его нет, функция завершится с ошибкой.
  - `RegCreateKeyEx`: Создает новый раздел или открывает существующий.
    - Параметры включают базовый раздел (`hKey`), имя подраздела (`lpSubKey`), маску доступа (`samDesired`), возвращаемый дескриптор (`phkResult`), а также опции для типа создаваемого раздела (энергозависимый/независимый).
- **Чтение параметров:**
  - `RegQueryValueEx`: Читает значение параметра.
    - Параметры: имя параметра (`lpValueName`), тип возвращаемых данных (`lpType`), буфер для данных (`lpData`), размер буфера (`lpcbData`).
  - `RegGetValue`: Аналогична `RegQueryValueEx`, но добавляет опцию для ограничения типа(ов) возвращаемых значений.
- **Запись параметров:**
  - `RegSetValueEx`: Записывает данные в параметр раздела.
  - `RegSetKeyValue`: Практически идентична `RegSetValueEx`, но позволяет указать подраздел относительно `hKey`.
    - Параметры аналогичны функциям чтения, но буфер содержит записываемые данные, а последний параметр – их размер.
- **Удаление разделов/параметров:**
  - `RegDeleteKey` или `RegDeleteKeyEx`: Удаляют подраздел реестра. Могут удалять только раздел без подразделов.
  - `RegDeleteValueA`: Удаляет параметры раздела.
  - `RegDeleteTree`: Удаляет раздел со всеми его подразделами.
- **Закрытие дескрипторов:**
  - `RegCloseKey`: Освобождает дескриптор раздела реестра. Не обязательно записывает данные на диск немедленно.
- **Явная запись данных на диск:**
  - `RegFlushKey`: Записывает данные реестра на жесткий диск. Использует много системных ресурсов и должен вызываться только в случае крайней необходимости.

Важно: Все описанные функции требуют соответствующих прав доступа. Следует следить за их выдачей при открытии или создании разделов Реестра.

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)** — это объектная модель компоненты фирмы Microsoft, представляющая собой **платформенно-независимую, распределенную, объектно-ориентированную систему для создания бинарных программных**

компонентов, которые могут взаимодействовать между собой. COM является стандартом, а не объектно-ориентированным языком программирования. Она определяет объектную модель и требования к программированию, позволяющие COM-объектам взаимодействовать с другими объектами.

- **Два свойства, лежащие в основе COM**:
  - **Двоичный стандарт (или независимость от языка программирования)**: COM предоставляет двоичный стандарт для программных компонентов, позволяя объектам и компонентам, разработанным на разных языках программирования и работающим в различных операционных системах, взаимодействовать без изменений в двоичном (исполняемом) коде. Это означает, что разработчики могут создавать компоненты, используя любой удобный для них язык и средства разработки, не заботясь о языке, используемом другими разработчиками.
  - **Независимость от местоположения (Location Transparency)**: Клиент COM-компонента не обязан знать фактическое расположение компонента. 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.
- **Почему COM называют двоичным стандартом?** COM называется двоичным стандартом, потому что он **применяется после того, как программа переведена в двоичный машинный код**, позволяя компонентам, написанным на разных языках, взаимодействовать.
- **COM-компонент** — это программный модуль, который можно сравнить с объектом в понимании C++ или Java. COM-объект представляет собой сущность, имеющую состояние и методы доступа, позволяющие изменять это состояние. Он не может быть уничтожен напрямую; вместо этого используется механизм **самоуничтожения, основанный на подсчете ссылок**.
- **COM-интерфейс** — это **контракт, состоящий из списка связанных прототипов функций**, чье назначение определено, а реализация — нет. Эти прототипы функций эквивалентны абстрактным базовым классам C++, имеющим только виртуальные методы.
- **Два типа COM-интерфейсов**: стандартные и произвольные.
- **COM-интерфейс характеризуется** тем, что является контрактом, состоящим из списка связанных прототипов функций, чье назначение определено, а реализация — нет. Напрямую с интерфейсом не ассоциировано никакой реализации. Реализация интерфейса — это код, который программист создает для выполнения действий, оговоренных в определении интерфейса.
- **Два стандартных COM-интерфейса**: **IUnknown** (важнейший, от которого наследуют все остальные интерфейсы) и **IClassFactory**.
- **GUID (Globally Unique Identifier)** — это глобально уникальный идентификатор размером **128 бит**, уникальный в пространстве и времени. Его уникальность достигается путем внедрения в него информации об уникальных частях компьютера, на котором он был создан, и времени создания.
- **CLSID (Class Identifier)** — это **GUID**, используемый для точного указания, какой именно COM-класс (CoClass) требуется.
- **IID (Interface Identifier)** — это **тип данных GUID**, применяемый 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?

- **Типы COM-контейнеров**: **DLL-файл** и **EXE-файл**.
- **COM-сервер** — это контейнер, в котором расположены COM-компоненты.
- **COM-клиент** — это приложение, которое использует COM-компоненты (вызывает функции интерфейсов, реализованных COM-компонентами).
- **Типы COM-серверов** в зависимости от типа контейнера и его расположения:
  - **INPROC** (DLL, локальный).
  - **LOCAL** (EXE, локальный).
  - **REMOTE** (EXE, удаленный).
  - Также существуют in-process и out-of-process серверы, где in-process — это DLL, а out-of-process — EXE.
- **«Однокомпонентные» COM-серверы** содержат только один компонент в контейнере. **«Многокомпонентные» COM-серверы** содержат два и более компонента.
- Для использования COM-объекта **COM-клиент должен «знать»**:
  - **GUID-идентификатор** этого компонента (CLSID).
  - **GUID-идентификаторы** (IID) интерфейсов.
  - **Тип сервера**.
  - **Структуры (сигнатуры соответствующих методов) произвольных интерфейсов** компонента, которые он предполагает применять.
- **Алгоритм создания «однокомпонентного» COM-сервера**:
  1. **Описать COM-интерфейс**, который обязан наследоваться хотя бы от IUnknown. Это можно сделать с помощью языка IDL и MIDL-компилятора, или напрямую на языке C/C++.
  2. **Создать COM-компонент** путем написания класса, который реализует описанный COM-интерфейс.
  3. **Реализовать фабрику классов** путем реализации стандартного COM-интерфейса IClassFactory для удобного управления жизненным циклом COM-компонентов.
  4. **Реализовать набор из 5 обязательных функций DLL**, которые обеспечивают работу COM-компонентов и обязательно экспортируются из DLL. Эти функции включают: `DllCanUnloadNow`, `DllGetClassObject`, `DllInstall`, `DllRegisterServer` (источник явно перечисляет 4 из 5 функций с описаниями).
  5. **Скомпилировать COM-сервер**.
  6. **Зарегистрировать COM-сервер** с использованием утилиты `regsvr32`.
  7. **Разработать COM-клиент**, который включает: инициализацию библиотеки OLE32 (вызов `CoInitializeEx`), создание экземпляра компонента (`CoCreateInstance`), получение указателя на другие интерфейсы (`QueryInterface`), и освобождение 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. IDL-файлы содержат описание COM-компонентов и COM-интерфейсов независимым 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.

- **Методы интерфейса IUnknown**:
  - **AddRef**: увеличивает счетчик ссылок на интерфейс на 1.
  - **QueryInterface**: получение указателя на интерфейс по его IID.
  - **Release**: уменьшает счетчик ссылок на интерфейс на 1.
- **«Счетчик ссылок на интерфейсы»** — это механизм, необходимый для отслеживания момента, когда экземпляр COM-компонента больше не требуется и может быть удален. Он является частью механизма самоуничтожения COM-объектов.
- Этот счетчик **увеличивается** при вызове метода **AddRef** и **уменьшается** при вызове метода **Release**. Экземпляр COM-компоненты не может быть выгружен, пока счетчик ссылок на его интерфейсы не равен нулю.
- **Соглашение о вызове и возврате методов COM-объекта**: Все методы COM-интерфейса должны поддерживать соглашение о вызовах **stdcall** и возвращать значение типа **HRESULT**.
- **Исключения**: Методы **AddRef** и **Release** являются исключениями, так как они возвращают текущее значение счетчика ссылок на интерфейс, а не HRESULT.
- **HRESULT** — это тип данных, который служит для **отображения результата выполнения функции COM-компонента**, указывая на успех или ошибку.
- **Структура HRESULT**:
  - **30-31 биты**: отображают успешность выполнения функции (успех или неудача).
  - **29 бит**: отображает, кем был определен данный статус-код (пользователем или системой).
  - **28 бит**: (не поясняется).
  - **16-27 биты**: отображают, к какой технологии относится статус-код.
  - **0-15 биты**: отображают точный результат в рамках заданной технологии и серьезности.
  - Для определения успешности или неудачи используются макросы `SUCCEEDED()` и `FAILED()`.

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.

- **«Фабрика классов» (IClassFactory)** — это интерфейс, отвечающий за **управление жизненным циклом компонентов путем реализации паттерна «фабрика классов»**. COM требует, чтобы каждый класс имел собственную фабрику классов для создания экземпляров, но многие классы могут использовать одну и ту же реализацию. Фабрика классов упрощает отслеживание созданных объектов и определение момента, когда их можно выгружать из памяти.
- **Методы интерфейса IClassFactory**:
  - **CreateInstance**: метод, предназначенный для создания экземпляра COM-компонента.

    - **LockServer**: увеличивает счетчик блокировки COM-сервера. Блокировка COM-сервера гарантирует, что он не будет закрыт раньше времени (DLL не будет выгружена).
- **Назначение «счетчика экземпляров компонент»**: Этот счетчик помогает отслеживать, сколько COM-компонентов используется, что является важной частью работы COM-сервера для определения возможности освобождения его ресурсов.
- **Увеличение и уменьшение счетчика экземпляров компонент**: Увеличение счетчика происходит в **конструкторе COM-компонента** (он вызывается методом CreateInstance). Уменьшение происходит, когда компонент больше не используется (источник не описывает, когда он уменьшается, но подразумевает, что он достигает нуля для выгрузки сервера).
- **Условие, при котором объект компонента удаляется**: Экземпляр COM-компоненты не может быть выгружен, пока **счетчик ссылок на интерфейсы не равен нулю**.
- **Жизненный цикл COM-сервера**:
    - Сервер **не может быть выгружен**, пока **счетчик экземпляров компонент не равен нулю** (экземпляр COM-компоненты обычно уникален в рамках одного процесса, т.е. по сути Singleton на уровне процесса).
    - Экземпляр COM-компоненты **не может быть выгружен**, пока **счетчик ссылок на интерфейсы не равен нулю**.
    - Сервер **не может быть выгружен**, пока **счетчик блокировок (LockServer) не равен нулю**.

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

- **Процесс регистрации COM-объекта** заключается в том, что для реализации свойства «Независимости от местоположения» в Windows каждый COM-компонент **должен быть зарегистрирован в Windows-реестре**. Это позволяет операционной системе определять место расположения контейнера компонента по его CLSID.
- **Назначение утилиты regsvr32 и принцип ее работы**: `regsvr32` — это специальная утилита, применяемая для **регистрации COM-компонентов**. По сути, она **просто вызывает некоторые экспортируемые функции из DLL** (например, `DllInstall`, `DllRegisterServer`).
- **Пять функций, экспортируемых COM/DLL-контейнером** (источник явно описывает 4 из 5):
    - **DllCanUnloadNow**: Функция, которая автоматически вызывается `OLE32.DLL` перед попыткой клиентом выгрузить COM-сервер. В зависимости от результата работы этой функции `OLE32.DLL` выгружает или не выгружает COM-сервер.
    - **DllGetClassObject**: Первая функция компонента, вызываемая `OLE32.DLL` при работе с клиентом. Функция проверяет идентификатор компонента, создает фабрику классов компонента и через параметры возвращает `OLE32.DLL` указатель на стандартный интерфейс `IClassFactory`.
    - **DllInstall**: Функция, вызываемая утилитой `regsvr32` (при наличии соответствующего параметра), применяется для выполнения

дополнительных действий при регистрации и удалении регистрации компонентов.
    - **DllRegisterServer**: Функция, вызываемая утилитой `regsvr32` (при наличии соответствующего параметра), применяется для регистрации.
    - (Пятая функция, `DllUnregisterServer`, обычно дополняет `DllRegisterServer`, но не описана явно в источнике, хотя упоминается необходимость 5 обязательных функций.)
- **Работа с памятью в COM и почему она такая**: COM определяет свои собственные функции для выделения и освобождения памяти в куче: **CoTaskMemAlloc** (выделяет блок памяти) и **CoTaskMemFree** (освобождает блок памяти).
    - Это сделано для обеспечения **уровня абстракции** и **единого подхода** к управлению памятью. Без этого различные методы могли бы вызывать `malloc` или `new`, требуя от программы отслеживать, когда вызывать `free` или `delete`.
    - Еще одна причина заключается в том, что COM является **двоичным стандартом** и не привязан к определенному языку программирования, следовательно, он не может полагаться на какую-либо специфичную для языка форму распределения памяти.

18. Сервисы

- **Что такое сервис?**
    - **Сервис** или **служба** — это процесс, который выполняет служебные функции. Они являются аналогами резидентных программ, использовавшихся в операционных системах, предшествующих Windows NT.
    - Сервис запускается при загрузке операционной системы или в процессе ее работы по специальной команде и заканчивает свою работу при завершении работы операционной системы или по специальной команде. Однако не каждая программа, запускаемая со стартом операционной системы, является сервисом.
    - Примерами сервисов могут служить фоновые процессы, обеспечивающие доступ к базе данных (называемые **серверами**), программы, обеспечивающие доступ к внешним устройствам (называемые **драйверами**), или процессы, отслеживающие работу других приложений (называемые **мониторами**).
- **Виды сервисов:**
    - Сервисы могут быть реализованы как **самостоятельные процессы** (`SERVICE_WIN32_OWN_PROCESS`, `SERVICE_USER_OWN_PROCESS`) или **разделять процесс с другими сервисами** (`SERVICE_WIN32_SHARE_PROCESS`, `SERVICE_USER_SHARE_PROCESS`).
    - Также существуют сервисы, являющиеся **драйверами устройств** (`SERVICE_KERNEL_DRIVER`) или **драйверами файловой системы** (`SERVICE_FILE_SYSTEM_DRIVER`).

    - Некоторые сервисы могут **взаимодействовать с рабочим столом** (`SERVICE_INTERACTIVE_PROCESS`).
- **Характеристики сервисов:**
    - Работают только в **фоновом режиме**.
    - **Не имеют собственного управляющего интерфейса** (ни GUI, ни TUI), взаимодействуют с пользователем только через рабочий стол или через окно сообщений.
    - Управляются специальной программой ОС – **менеджером служб** (Service Control Manager, SCM).
    - Запускаются или 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 непосредственно пользователям.
- **Что такое SCM? Для чего он предназначен?**
    - **Менеджер сервисов (Service Control Manager, SCM)** – это специальная программа операционной системы Windows, которая управляет работой сервисов.
    - Функции SCM включают:
        - Поддержку базы данных установленных сервисов.
        - Запуск сервисов при загрузке операционной системы или по запросу.
        - Перечисление установленных сервисов.
        - Поддержку информации о состоянии работающих сервисов.
        - Передачу управляющих запросов работающим сервисам.
- **Опишите структуру сервиса.**
    - Так как сервисы работают под управлением SCM, они должны удовлетворять определенным соглашениям, определяющим интерфейс сервиса. Обычно сервисы оформляются как консольные приложения.
    - Каждый сервис должен содержать **две функции обратного вызова**: одна определяет **точку входа** сервиса, а вторая должна **реагировать на управляющие сигналы** от операционной системы.
    - Главная задача функции `main` приложения сервиса — запуск **диспетчера сервиса** (потока), который управляет этим сервисом. Диспетчер сервиса получает управляющие сигналы от SCM по именованному каналу и передает эти запросы функции обработки управляющих запросов (например, `ServiceCtrlHandler`).
    - Если в приложении несколько сервисов, то для каждого сервиса запускается свой диспетчер и для каждого диспетчера определяется своя функция обработки управляющих запросов.
- **Какова особенность точки входа сервиса?**
    - Функция, определяющая точку входа сервиса, обычно называется `ServiceMain` (хотя могут быть и другие имена, особенно если сервисов несколько).
    - Главная задача функции `main` — запуск диспетчера сервиса с помощью функции `StartServiceCtrlDispatcher`, которая **должна быть вызвана в течение 30 секунд** с момента запуска программы `main`. Если это условие не будет выполнено, последующий вызов `StartServiceCtrlDispatcher` завершится неудачей.
    - Всю необходимую инициализацию сервиса следует выполнять **в самой функции `ServiceMain`**.
    - `ServiceMain` должна выполнить следующую последовательность действий:

1. Немедленно запустить обработчик управляющих команд, вызвав `RegisterServiceCtrlHandler`.
2. Установить стартующее состояние сервиса (`SERVICE_START_PENDING`) с помощью `SetServiceStatus`.
3. Провести локальную инициализацию сервиса.
4. Установить рабочее состояние сервиса (`SERVICE_RUNNING`) с помощью `SetServiceStatus`.
5. Выполнять работу сервиса, учитывая изменения состояния от SCM.
6. После перехода в состояние останова (`SERVICE_STOPPED`) освободить захваченные ресурсы и завершить работу.

- **Что такое функция обратного вызова?**
  - **Функция обратного вызова** — это функция, которая передаётся другой функции в качестве аргумента. В контексте сервисов, это функции, которые вызываются операционной системой (например, точка входа сервиса и обработчик управляющих сигналов).
- **Где и какая хранится информация о сервисах Windows?**
  - SCM поддерживает базу данных установленных сервисов в **реестре Windows**.
  - База данных находится под ключом **`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`**.
  - Этот раздел содержит подраздел для каждого установленного сервиса и сервиса драйвера, причем название подраздела является названием службы.
  - Информация о сервисе, хранящаяся в реестре, включает такие параметры как:
    - `DependOnGroup` (группы порядка загрузки, от которых зависит сервис).
    - `DependOnService` (сервисы, от которых зависит данный сервис).
    - `Description` (описание).
    - `DisplayName` (отображаемое имя).
    - `ErrorControl` (уровень управления ошибками).
    - `FailureActions` (действия при сбоях).
    - `Group` (группа порядка загрузки, в которой состоит сервис).
    - `ImagePath` (путь к исполняемому файлу).
    - `ObjectName` (учетная запись, от имени которой запускается сервис).
    - `Start` (тип запуска).
    - `Tag` (уникальный тег для сервиса в рамках группы).
- **Что такое группа порядка загрузки?**
  - **Группы порядка загрузки** — это логически объединенный набор сервисов, который определяет порядок их загрузки относительно остальных групп или сервисов.
  - Порядок загрузки сервисов определяется следующим образом:
    1. Порядок групп в списке групп порядка загрузки, хранящийся в `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ServiceGroupOrder`.
    2. Порядок загрузки сервисов в рамках своей группы, хранящийся в `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\GroupOrderList`.

19. Сервисы (Linux)

- **Что такое демон?**

    - **Демон (daemon)** — это процесс, обладающий следующими 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**. Это 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`.
- **Опишите и поясните алгоритм создания процесса-демона вручную.**
    - Для того чтобы стать демоном, программа должна выполнить следующие шаги:
        1. **Вызов `fork` и завершение родителя**: После вызова `fork`, родительский процесс завершается, а дочерний продолжает работу, становясь потомком процесса `init`. Это позволяет командной оболочке вывести новое приглашение и позволяет потомку выполняться в фоновом режиме. Также это гарантирует, что потомок не станет лидером группы процессов, что необходимо для следующего шага.
        2. **Вызов `setsid` дочерним процессом**: Дочерний процесс вызывает `setsid`, чтобы начать новую сессию и разорвать любые связи с контролирующим терминалом.
        3. **Предотвращение повторного соединения с контролирующим терминалом**: Чтобы демон не восстановил соединение с контролирующим терминалом, можно либо указывать флаг `O_NOCTTY` для любых вызовов `open`, либо, что проще, после `setsid` **снова вызвать `fork`**, позволив родителю завершиться. Это гарантирует, что правнук не станет лидером сессии.
        4. **Очистка атрибута `umask` процесса**: Это необходимо, чтобы файлы и каталоги, созданные демоном, имели запрашиваемые права доступа.
        5. **Смена текущего рабочего каталога процесса**: Обычно на корневой (`/`), чтобы файловая система, на которой находится демон, не могла быть отключена. В качестве рабочего каталога демон может использовать место выполнения своей работы или значение из конфигурационного файла.
        6. **Закрытие всех открытых файловых дескрипторов**: Демон должен закрыть дескрипторы, унаследованные от родителя, особенно 0, 1 и 2, если они ссылаются на терминал. Это освобождает ресурсы и предотвращает блокировку файловых систем.
        7. **Перенаправление дескрипторов 0, 1 и 2 в `/dev/null`**: После закрытия, демон обычно перенаправляет их в `/dev/null` с помощью `dup2` (или похожего вызова). Это предотвращает ошибки при вызове библиотечных функций, выполняющих I/O с этими дескрипторами, и исключает возможность повторного открытия файлов с помощью 1 или 2, так как библиотечные функции ожидают, что они указывают на stdout/stderr.
- **Рекомендации при создании демонов.**
    - **Завершение работы**: Обычно демон завершается во время выключения системы. Для многих стандартных демонов предусмотрены специальные скрипты. Остальные получают сигнал `SIGTERM`, по умолчанию приводящий к завершению. Если демону необходимо освободить ресурсы, он должен

делать это в обработчике `SIGTERM`, так как через 5 секунд `init` отправляет `SIGKILL`.
    - **Утечки ресурсов**: Из-за длинного жизненного цикла, необходимо тщательно следить не только за потенциальными утечками памяти, но и за файловыми дескрипторами.
    - **Один экземпляр**: Часто демону необходимо убедиться, что только один его экземпляр активен в любой заданный момент времени. Это достигается созданием файла в стандартном каталоге и применением к нему блокировки для записи. Если другой экземпляр не сможет получить блокировку, он завершится.
- **Что такое systemd и init?**
    - **`init`** (инициализация) и **`systemd`** — это менеджеры сервисов в Linux (аналоги SCM в Windows). `init` считается устаревшим, а `systemd` является более новым и распространённым.
    - Эти менеджеры самостоятельно выполняют алгоритм "демонизации" приложения, поэтому нет необходимости в коде будущего демона совершать описанные манипуляции. Они также позволяют настроить автозапуск сервисов вместе с запуском операционной системы.
- **Опишите процесс создания сервиса на примере systemd или init.**
    - **При использовании менеджера `init`**:
        1. Скомпилировать приложение будущего демона.
        2. Расположить получившийся бинарный файл в одной из папок `/sbin` или `/usr/sbin`.
        3. Расположить скрипт запуска (`<имя демона>`) в каталоге `/etc/init.d/` и изменить ему права через `chmod` на 755.
        4. (Опционально) Создать каталог `/etc/<имя демона>`, в котором будут располагаться конфигурационные файлы сервиса. Файлы логов обычно располагают в каталоге `/var/log`.
        5. Проверить с помощью команды `service`, что система распознала скрипт инициализации сервиса.
        6. Если скрипт написан корректно, будет доступно управление сервисом через команду `service` (например, `service <имя демона> start/stop/restart/status`).
        7. Для автозапуска вместе с системой необходимо перейти в каталог `/etc/init.d` и выполнить команду `update-rc.d <имя демона> defaults`.
    - **При использовании менеджера `systemd`**:
        1. Скомпилировать приложение будущего сервиса.
        2. Расположить получившийся бинарный файл в одной из папок `/sbin` или `/usr/sbin`.
        3. Расположить файл описания (`<имя демона>.service`) в каталоге `/etc/systemd/system`.
        4. (Опционально) Создать каталог `/etc/<имя демона>`, в котором будут располагаться конфигурационные файлы демона. Файлы логов обычно располагают в каталоге `/var/log`.
        5. Проверить с помощью команды `systemctl`, что система распознала описание сервиса.
        6. Если файл с описанием написан корректно, будет доступно управление сервисом через команду `systemctl` (например, `systemctl start/stop/status <имя демона>`).

7. Для автозапуска сервиса вместе с системой `systemctl` имеет команду `systemctl enable <имя демона>`.

20. Драйверы

- **Что такое драйвер?**
  - **Драйвер** — это часть кода операционной системы, отвечающая за **взаимодействие с аппаратурой**. Под аппаратурой 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**.
  - Обычно разрабатывается производителем устройства и знает, как взаимодействовать с его аппаратным обеспечением для получения или отправки данных.
- **Какое место занимает драйвер в структуре ОС?**
  - Драйвер является **посредником** между операционной системой и аппаратными устройствами. Он интерпретирует высокоуровневые команды от ОС (например, чтение или запись) и выполняет низкоуровневые команды, связанные с устройством (например, запись в регистр управления).
  - Сформировался как **отдельный и довольно независимый модуль**, легко заменяемая часть операционной системы, хотя многие драйверы остаются практически неотделимыми от ОС.
  - Большинство драйверов работают в **режиме ядра** операционной системы, что накладывает определенные особенности на их разработку.
- **Основные концепции драйверов.**
  - Способ работы с драйверами как с файлами, используя лексически идентичные функции, такие как `open`, `close`, `read`, `write`.
  - Драйвер как легко заменяемая часть ОС.
  - Существование **режима ядра**.
  - Использование механизма **IOCTL (Input/Output Control Code)** для запросов.
- **Что такое подсистема ввода/вывода?**
  - В Windows **подсистема ввода/вывода** состоит из набора компонентов исполнительной системы, которые совместно управляют устройствами и предоставляют приложениям и системе интерфейсы к этим устройствам.
  - Она проектировалась как **абстрактный интерфейс** приложений для аппаратных (физических) и программных (виртуальных, или логических) устройств.
- **Какие функциональные возможности она предоставляет?**
  - **Унифицированные средства безопасности и именования устройств** для защиты общих ресурсов.
  - **Высокопроизводительный асинхронный пакетный ввод/вывод** для поддержки масштабируемых приложений.
  - 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.

  - **Динамические загрузка и выгрузка** драйверов устройств по запросу, экономящие системные ресурсы.
  - Поддержка технологии **Plug and Play (PnP)**, обеспечивающая обнаружение и установку драйверов для нового оборудования, выделение им ресурсов и возможность приложениям находить и задействовать интерфейсы устройств.
  - Подсистема **управления электропитанием**, позволяющая системе или отдельным устройствам переходить в состояния с низким энергопотреблением.
- **Перечислите из чего состоит подсистема ввода/вывода?**
  - Ряд компонентов исполнительной подсистемы и драйверов устройств.
  - **Диспетчер ввода/вывода (I/O manager)**: Центральное место, соединяет приложения и системные компоненты с устройствами, создает инфраструктуру для драйверов.
  - **Драйвер устройства**: Предоставляет интерфейс ввода/вывода к конкретному типу устройства.
  - **PnP-диспетчер**: Работает совместно с диспетчером ввода/вывода и драйверами шины, управляет выделением ресурсов, распознает устройства.
  - **Диспетчер электропитания**: Тесно связан с диспетчером ввода/вывода и PnP-диспетчером, управляет состояниями энергопотребления.
  - **Реестр**: База данных с описанием подключенных устройств и параметров инициализации драйверов.
  - **INF-файлы**: Управляют установкой драйверов, связывают устройства с драйверами.
  - **Удостоверяющие файлы драйверов (.cat)**: Хранят цифровые подписи WHQL.
  - **Уровень аппаратных абстракций (HAL)**: Изолирует драйверы от специфических особенностей процессоров, скрывает межплатформенные различия.

21. Драйверы (продолжение)

- **Что такое драйвер устройства?**
  - **Драйвер устройства** — это программный модуль, который предоставляет интерфейс ввода/вывода к конкретному типу устройства.
  - Он интерпретирует высокоуровневые команды (такие как команды чтения или записи) и выполняет низкоуровневые команды, связанные с устройством (например, запись в регистр управления).
  - Драйверы устройств принимают от диспетчера ввода/вывода команды, предназначенные для управляемых ими устройств, и уведомляют диспетчер о выполнении этих команд.
- **Что такое диспетчер ввода/вывода? Какого его назначение?**
  - **Диспетчер ввода/вывода (I/O manager)** является центральным элементом подсистемы ввода/вывода в Windows.
  - Его назначение:
    - Соединять приложения и системные компоненты с виртуальными, логическими и физическими устройствами.
    - Создавать инфраструктуру для поддержки драйверов устройств.
    - Задавать инфраструктуру для доставки драйверам устройств запросов на ввод и вывод.

        - Представлять операции ввода/вывода в памяти в виде **IRP-пакетов (I/O Request Packets)**.
        - Передавать IRP-пакеты нужному драйверу и удалять их после завершения операции.
        - Предоставлять различным драйверам общий код, который они используют при обработке ввода/вывода.
- **Что такое PnP-диспетчер и каково его назначение?**
    - **PnP-диспетчер (Plug and Play manager)** работает совместно с диспетчером ввода/вывода и драйверами шины. Он состоит из двух компонентов: PnP-менеджера пользовательского режима и PnP-менеджера ядерного режима.
    - Его назначение:
        - Управлять **выделением аппаратных ресурсов**.
        - **Распознавать устройства** и реагировать на их подключение или отключение.
        - Обеспечивать **загрузку соответствующего драйвера** при обнаружении нового устройства.
        - Вызывать сервисные функции установки устройств PnP-диспетчера в пользовательском режиме, если нужный драйвер отсутствует.
- **Что такое диспетчер электропитания?**
    - **Диспетчер электропитания** тесно связан с диспетчером ввода/вывода и PnP-диспетчером.
    - Он управляет **переходами в различные состояния энергопотребления** как самой системы, так и отдельных драйверов устройств.
- **Для чего используется реестр в случае с драйверами и что такое INF-файлы?**
    - **Реестр** представляет собой базу данных с описанием основных подключенных к подсистеме устройств, а также **параметров инициализации драйверов и конфигурации**.
    - **INF-файлы** (.inf) управляют **установкой драйверов**. Они связывают аппаратные устройства с драйверами, описывают устройство, исходное и целевое положение файлов драйвера, вносимые в реестр изменения при установке драйвера и сведения о зависимостях драйвера.
- **Что такое HAL?**
    - **Уровень аппаратных абстракций (Hardware Abstraction Layer, HAL)** — это компонент операционной системы, который **изолирует драйверы от специфических особенностей** конкретных процессоров и контроллеров прерываний.
    - Он поддерживает прикладные программные интерфейсы, скрывающие межплатформенные различия, и по сути является драйвером шины для устройств на материнской плате компьютера, которые не управляются другими драйверами.

22. Драйверы (продолжение)

- **Что такое драйвер?**
    - **Драйвер** — это часть кода операционной системы, отвечающая за взаимодействие с аппаратурой, предоставляющая другому программному обеспечению API для работы с аппаратными устройствами.
- **Опишите «жизненный цикл» IRP.**
    - **IRP (I/O Request Packet)** — это структура данных, инкапсулирующая информацию, полностью описывающую запрос на ввод/вывод. Диспетчер ввода/вывода представляет все операции ввода/вывода в виде таких пакетов.

- Жизненный цикл IRP:
    1. Диспетчер ввода/вывода **создает IRP-пакет** в ответ на запрос приложения.
    2. Указатель на IRP **передается нужному драйверу**.
    3. Драйвер **выполняет** указанную в пакете операцию.
    4. После завершения операции драйвер **возвращает пакет диспетчеру** ввода/вывода (либо сигнализируя о завершении, либо с целью передачи пакета другому драйверу для дальнейшей обработки).
    5. Диспетчер ввода/вывода **удаляет** пакет.

- **Что такое виртуальные файлы?**
    - **Виртуальный файл** — это термин, относящийся 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, так как диспетчер ввода/вывода ни с чем другим работать не умеет. Драйверы отвечают за преобразование файловых команд (открытие, закрытие, чтение, запись) в команды для конкретного устройства.
- **Особенности программирования на уровне ядра.**
    - **Необработанные исключения**: В режиме пользователя "уронят" процесс, в режиме ядра – **систему**.
    - **Завершение работы**: В режиме пользователя все приватные ресурсы процесса освобождаются автоматически. В режиме ядра, если драйвер выгружен без очистки, утечка ресурсов устраняется только при следующем перезапуске системы.
    - **Возвращаемые значения**: В режиме пользователя коды ошибок часто игнорируются, в режиме ядра – это **практически недопустимо**.
    - **Уровни запросов прерываний (IRQL)**: В режиме пользователя значение всегда равно 0. В режиме ядра может быть разным, что требует знания уровня, на котором работает соответствующая функция.
    - **Плохо написанный код**: В режиме пользователя эффект от такого кода обычно не выходит за рамки процесса, в режиме ядра – эффект **распространяется на всю систему**.
    - **Тестирование и отладка**: В режиме пользователя производится на устройстве разработчика, в режиме ядра – **требуется отдельное устройство**.
    - **Использование библиотек**: В режиме пользователя доступны практически все библиотеки C/C++, 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 – **нету среды выполнения C++**.
- **Что такое уровни запросов прерываний?**
    - **Уровни запросов прерываний, или IRQL (Interrupt Request Level)** — это важная концепция ядра Windows.
    - Термин IRQL имеет два значения:
        - **Приоритет**, назначаемый источнику прерываний от физического устройства.
        - **Собственный уровень IRQL** у каждого центрального процессора.
    - **Фундаментальное правило IRQL**: 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)**. В пользовательском режиме IRQL может быть равен только 0.
    - Важные уровни IRQL в контексте ввода/вывода:
        - **Passive (0)**: Нормальный уровень, при котором планировщик ядра работает нормально.
        - **Dispatch/DPC (2)**: Уровень IRQL, на котором работает планировщик ядра. Если поток поднимает IRQL до 2 или выше, он получает бесконечный квант и не может быть вытеснен другим потоком.
        - **Device IRQL (DIRQL, 3-26 на x86)**: Закрепляются за аппаратными преобразованиями. При поступлении прерывания диспетчер вызывает соответствующую функцию обслуживания прерывания (ISR) и повышает ее IRQL до уровня прерывания.
- **Что такое отложенные вызовы процедур?**
    - **Отложенный вызов процедуры, или DPC (Deferred Procedure Call)** — это объект, инкапсулирующий вызов функции на уровне IRQL DPC_LEVEL (2).
    - Объекты DPC существуют прежде всего для выполнения действий **после прерывания**, так как выполнение на уровне DIRQL маскирует (а следовательно, задерживает) другие прерывания, ожидающие обработки.
    - Термин «отложенный» означает, что DPC не выполняется немедленно, потому что текущий уровень IRQL выше 2. Выполнение происходит, когда ISR вернет управление и уровень IRQL опустится до 2.
    - Обычно ISR-процедура ставит DPC в очередь для выполнения на более низком IRQL-уровне (на уровне DPC/dispatch). DPC выполняет основную часть обработки прерывания, оставшейся после ISR-процедуры, и инициирует завершение одной операции ввода/вывода и начало следующей.
- **Поясните эти две концепции на примере.**
    - Пример последовательности событий, иллюстрирующей работу IRQL и DPC:
        1. Некий код пользовательского или ядерного режима выполняется, когда процессор находится на уровне **IRQL 0** (Passive Level).
        2. Поступает **аппаратное прерывание** (например, **IRQL 5**). Состояние процессора сохраняется, IRQL повышается до 5, и вызывается функция обслуживания прерывания **ISR 1**. Контекстное переключение при этом не происходит.
        3. **ISR 1** начинает выполняться на **IRQL 5**. Прерывания с IRQL 5 и ниже не могут вмешиваться.
        4. Поступает **новое прерывание** (например, **IRQL 8**). Если система решает обработать его тем же процессором, выполнение прерывается, состояние сохраняется, IRQL повышается до 8, и процессор переходит к **ISR 2**. Контекстное переключение невозможно, так как планировщик не может активизироваться при IRQL уровня 2 и выше.
        5. Выполняется **ISR 2**. Он желает выполнить дополнительную обработку на более низком IRQL, чтобы другие прерывания могли быть обработаны.
        6. **ISR 2 вставляет объект DPC** со ссылкой на функцию драйвера для последующей обработки (вызовом `KeInsertQueueDpc`). Затем ISR

возвращает управление, восстанавливается состояние процессора, сохраненное перед входом в ISR 2.

7. IRQL падает до предыдущего уровня (**IRQL 5**), и процессор продолжает выполнение обработчика **ISR 1**, прерванного ранее.
8. Непосредственно перед завершением **ISR 1 ставит в очередь собственный объект DPC** для выполнения своей последующей обработки. ISR 1 возвращает управление, восстанавливая состояние процессора.
9. На этом этапе IRQL должен был бы упасть до 0, но ядро замечает наличие необработанных DPC. Поэтому **IRQL уменьшается до уровня 2 (DPC_LEVEL)**, и запускается цикл обработки DPC, который перебирает накопленные DPC и последовательно вызывает каждую процедуру DPC. Когда очередь DPC опустеет, обработка DPC завершается.
10. Наконец, **IRQL уменьшается до 0**, состояние процессора восстанавливается, и возобновляется выполнение изначально прерванного исходного кода пользовательского или ядерного режима. Вся описанная обработка происходит в одном потоке, и ISR и DPC не должны зависеть от конкретного потока.

23. Драйверы (классификация и архитектура)

- **Что такое драйвер?**
  - **Драйвер** — это часть кода операционной системы, отвечающая за взаимодействие с аппаратурой, предоставляющая другому программному обеспечению API для работы с аппаратными устройствами.
- **Какие бывают драйверы?**
  - Драйверы для ОС Windows NT условно классифицируются на:
    - **Драйверы пользовательского режима (User-Mode Drivers)**: Включают драйверы, являющиеся компонентами среды UMDF, и драйверы принтеров подсистемы Windows.
    - **Драйверы режима ядра (Kernel-Mode Drivers)**:
      - **Драйверы файловой системы**: Принимают запросы к файлам на ввод/вывод и на их основе выдают более конкретные запросы к драйверам запоминающих или сетевых устройств.
      - **Драйверы PnP (Plug and Play drivers)**: Работают с оборудованием, интегрируются с диспетчером электропитания и PnP-диспетчером (например, драйверы запоминающих устройств, видеоадаптеров, устройств ввода, сетевых адаптеров).
      - **Драйверы без поддержки Plug and Play (Non-Plug and Play drivers)**: Включают расширения ядра, не интегрированы с PnP-диспетчером или диспетчером электропитания, так как не связаны с физическими аппаратными устройствами.
- **Что такое WDM-драйверы и какие они бывают?**
  - **WDM-драйверы** — это драйверы устройств, соответствующие модели **WDM (Windows Driver Model)**. WDM поддерживает управление электропитанием, технологию Plug and Play и инструментарий управления Windows.
  - WDM-драйверы делятся на три 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?**
    - Многоуровневая архитектура драйверов приводит к появлению понятия **стека драйверов**.
    - По сути, это **набор драйверов, которые необходимо вызывать для получения конечного результата**.
    - При обработке запроса данные идут от **вышестоящих (higher-level)** драйверов к **нижестоящим (lower-level)**, а при возврате результатов — наоборот.
- **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 (например, жесткий диск, клавиатура, компакт-диск).
        - **Драйверы мини-классов (miniclass drivers)**: Реализуют обработку ввода/вывода, заданную производителем для определенного класса устройств (по сути, являются DLL уровня ядра с API).
        - **Драйверы портов (port drivers)**: Обрабатывают запросы на ввод и вывод в соответствии с типом порта ввода/вывода (например, SATA). Реализуются как библиотеки функций режима ядра.
        - **Драйверы мини-портов (miniport drivers)**: Преобразуют обобщенный запрос ввода/вывода о типе порта в запрос о типе адаптера. По сути, являются истинными драйверами устройств и импортируют функции, предоставляемые драйвером порта.
- **Опишите последовательность вызова функционала, реализованного многоуровневым драйвером.**
    - Последовательность вызова функционала в многоуровневом драйвере основывается на концепции **стека драйверов**.
    - Когда приложение инициирует операцию ввода/вывода, диспетчер ввода/вывода создает **IRP-пакет**.
    - Этот IRP-пакет передается **самому верхнему (вышестоящему)** драйверу в стеке.
    - Каждый драйвер в стеке обрабатывает часть запроса, выполняя необходимые действия и/или модифицируя IRP-пакет. Затем он **передает IRP-пакет следующему нижестоящему драйверу**.
    - Этот процесс продолжается до тех пор, пока IRP-пакет не достигнет **самого нижнего драйвера**, который непосредственно взаимодействует с аппаратным обеспечением устройства.
    - После того как низкоуровневый драйвер завершает операцию, результаты или статус выполнения **возвращаются вверх по стеку** драйверов, от нижестоящих к вышестоящим, пока не достигнут диспетчера ввода/вывода и, в конечном итоге, запрашивающего приложения.

24. Драйверы (запуск и объекты)

- **Что такое драйвер?**
  - **Драйвер** — это часть кода операционной системы, отвечающая за взаимодействие с аппаратурой, предоставляющая другому программному обеспечению API для работы с аппаратными устройствами.
- **Кто занимается запуском драйвера?**
  - **Подсистема ввода/вывода** занимается запуском драйверов устройств.
- **Что для этого требуется: перечислите и поясните назначение.**
  - Драйверы состоят из набора процедур, вызываемых для обработки различных этапов запроса на ввод или вывод. Ключевые процедуры включают:
    - **Процедура инициализации (`DriverEntry`)**: Выполняется диспетчером ввода/вывода при загрузке драйвера в операционную систему. Разработчик драйвера реализует `DriverEntry`, которая регистрирует остальные процедуры драйвера в диспетчере ввода/вывода и выполняет всю глобальную инициализацию драйвера.
    - **Процедура добавления устройства**: Реализуется PnP-драйверами. PnP-диспетчер посылает уведомление драйверу при обнаружении устройства, за которое он отвечает. В рамках этой процедуры драйвер, как правило, создает объект, представляющий устройство.
    - **Процедуры диспетчеризации**: Основные точки входа для драйвера устройства (открытие, закрытие, чтение, запись, PnP-операции и т. д.). Диспетчер ввода/вывода генерирует IRP-пакет и через одну из этих процедур вызывает драйвер.
    - **Процедура начала ввода/вывода**: Используется драйвером для инициации передачи данных на устройство или с него. Она определена только в драйверах, ставящих входящие запросы на ввод/вывод в очередь через диспетчер ввода/вывода.
    - **Процедура обработки прерываний (ISR - Interrupt Service Routine)**: Когда устройство приостанавливает свою работу, диспетчер прерываний ядра передает управление ISR. ISR работают на уровне DIRQL, выполняют минимум действий и обычно ставят DPC в очередь для выполнения на более низком IRQL.
    - **Процедура DPC (Deferred Procedure Call)**: Выполняет основную часть обработки прерывания, оставшейся после ISR-процедуры. Работает на уровне IRQL 2 и инициирует завершение одной операции ввода/вывода и начало следующей.
- **Какие дополнительные возможности может включать в себя драйвер?**
  - Многие драйверы устройств обладают дополнительными процедурами, такими как:
    - Процедуры завершения ввода/вывода.
    - Процедуры отмены ввода/вывода.
    - Процедуры быстрой диспетчеризации.
    - Процедура выгрузки.
    - Процедура уведомления о завершении работы системы.
    - Процедуры регистрации ошибок.
- **Что такое объекты драйвера и устройства и зачем они нужны?**
  - Для того чтобы диспетчер ввода/вывода мог определить, к какому драйверу следует обратиться для обработки запроса, и находить эту информацию при

последующем использовании дескриптора, используются следующие объекты:

- **Объект драйвера (`DRIVER_OBJECT`)**: Представляет отдельный драйвер в системе. Он предоставляет диспетчеру ввода/вывода адрес процедур диспетчеризации (точек входа) всех драйверов.
- **Объект устройства (`DEVICE_OBJECT`)**: Представляет физическое или логическое устройство в системе и описывает его характеристики (например, границы выравнивания буферов и адреса очередей входящих IRP-пакетов). Он является точкой назначения для всех операций ввода/вывода, так как именно с ним взаимодействует дескриптор.

- **Что такое файловый объект?**
  - **Файловый объект (`FILE_OBJECT`)** — это структура данных режима ядра, представляющая дескриптор устройства.
  - Файловые объекты соответствуют определению объектов в Windows: это системные ресурсы, доступные двум и более процессам в пользовательском режиме; у них могут быть имена, их безопасность обеспечивается моделью защиты объектов, и они поддерживают синхронизацию.
  - Они обеспечивают представление ресурсов в памяти, напоминающее интерфейс, ориентированный на ввод/вывод и реализующий чтение или запись.

25. Драйверы

- **Что такое пакет запроса на ввод/вывод (IRP)?** Пакет запроса на ввод/вывод (IRP) – это **структура данных, которая полностью описывает запрос на ввод и вывод**. Диспетчер ввода/вывода (I/O manager) представляет операции ввода и вывода в памяти в виде IRP-пакетов. **Указатель на IRP передается нужному драйверу**, а после завершения операции пакет удаляется. Драйвер, получивший IRP, выполняет указанную в пакете операцию и возвращает пакет диспетчеру ввода/вывода, либо сигнализируя о завершении операции, либо для передачи пакета другому драйверу для дальнейшей обработки. Подобное проектное решение позволяет отдельному программному потоку приложения одновременно управлять целым набором запросов на ввод и вывод.
- **Какие бывают IRP? Опишите их.** Источники не описывают различные типы IRP напрямую, но указывают, что запросы на ввод/вывод могут быть:
  - **Синхронные**: Большинство операций ввода/вывода, запрашиваемых приложениями, являются синхронными. Программный поток **ждет, когда устройство выполнит операцию** с данными и вернет код состояния, после чего программа может продолжить работу и немедленно воспользоваться переданными ей данными.
  - **Асинхронные**: Приложение может **выдать несколько запросов ввода/вывода и продолжить свою работу**, пока устройство выполняет операции, что повышает эффективность приложения.
  - Внутренние операции ввода/вывода, инициированные драйвером, выполняются асинхронно; драйвер должен как можно быстрее вернуть управление подсистеме ввода/вывода после выдачи запроса.
- **Что такое Plug and Play (PnP)? Plug and Play (PnP)** – это технология, включающая программную и аппаратную поддержку механизма, который позволяет **подключать/отключать, настраивать и т. д. все устройства**, подключаемые к системе, при условии, что они поддерживают PnP. В идеале, весь

процесс осуществляется только механизмом PnP без участия пользователя. Для успешной работы PnP требуется поддержка со стороны устройств, драйверов и системного программного обеспечения.

- **Какие возможности предоставляет ПО с поддержкой PnP?** Системное ПО (вместе с драйверами), поддерживающее технологию PnP, предоставляет следующие возможности:
  - **Автоматическое распознавание** подключенных к системе устройств.
  - **Распределение и перераспределение ресурсов** (например, портов ввода/вывода и участков памяти) между запросившими их устройствами.
  - **Загрузка необходимых драйверов**.
  - **Предоставление драйверам необходимого интерфейса** для взаимодействия с технологией PnP.
  - **Реализация механизма**, позволяющего драйверам и приложениям получать информацию об изменениях в наборе устройств, подключенных к системе, и совершать необходимые действия.
- **Из чего состоит система PnP?** Система PnP состоит из двух компонентов:
  - **Менеджер PnP пользовательского режима**: взаимодействует с установочными компонентами пользовательского режима для конфигурирования и установки устройств, а также при необходимости взаимодействует с приложениями.
  - **Менеджер PnP "ядерного" режима**: работает с операционной системой и драйверами для конфигурирования, управления и обслуживания устройств.
- **С чем может работать PnP?** PnP может успешно работать со следующими типами устройств:
  - Физические устройства
  - Виртуальные устройства
  - Логические устройства
- **Какие условия драйвер должен выполнить для осуществления полной поддержки PnP?** Для полной поддержки PnP драйвер должен выполнить следующие условия:
  - Наличие функции `DriverEntry`.
  - Наличие функции `AddDevice`.
  - Наличие функции `DispatchPnp`.
  - Наличие функции `DispatchPower`.
  - Наличие функции `Unload`.
  - Наличие **cat-файла** (файла каталога), содержащего сигнатуру WHQL.
  - Наличие **inf-файла** для установки драйвера.

26. SEH. Что такое исключение? Сравните их с прерываниями. Что такое Structured Exception Handling (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 определенной команды, которая вызвала ошибку процессора**. В результате такого события нормальное выполнение программы, скорее всего, становится невозможным. Примеры исключений включают деление на ноль, точку останова, ошибку страницы, переполнение стека и недопустимую инструкцию.

- **Сравните их с прерываниями.** Исключения в некотором роде похожи на прерывания. Основное отличие заключается в том, что **исключение является синхронным** и технически воспроизводимым при тех же условиях, в то время как **прерывание является асинхронным** и может произойти в любой момент.
- **Что такое Structured Exception Handling (SEH)? Structured Exception Handling (SEH)** – это механизм, при котором, когда возникает исключение, ядро перехватывает его и позволяет коду обработать исключение, если это возможно. SEH доступен как для кода пользовательского режима, так и для кода режима ядра, и **является частью исключительно операционной системы Windows**. Полная поддержка SEH присутствует только в компиляторе MSVC. Основная нагрузка по поддержке SEH ложится на компилятор, который генерирует специальный код на входах и выходах блоков исключений, создает таблицы вспомогательных структур данных и предоставляет функции обратного вызова.
- **Что такое блок исключения?** Термин "блок исключения" (exception blocks) относится к **блокам кода, определенным ключевыми словами `__try`, `__except` и `__finally`**. Компилятор генерирует специальный код на входах и выходах этих блоков.
- **Какие основные возможности предоставляет SEH?** SEH предоставляет две основные возможности:
  - **Обработка завершения (termination handling)**.
  - **Обработка исключений (exception handling)**.
- **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, которое следует обработать.
- **Поясните принципы работы обработчика завершения.** Обработчик завершения (`__finally`) **гарантирует, что блок кода внутри `__finally` будет выполнен независимо от того, как происходит выход из защищенного участка программы**, то есть из блока `__try`. Это означает, что даже если в защищенном блоке присутствуют операторы `return` или `goto`, код в `__finally` будет выполнен. Компилятор генерирует дополнительный код, а система выполняет дополнительную работу для обеспечения выполнения `__finally` при преждевременном выходе из `__try`.
- **Что такое локальная раскрутка? Локальная раскрутка (local unwind)** происходит, когда система выполняет блок `__finally` **из-за преждевременного выхода из блока `__try`**. Это может быть вызвано операторами `goto`, `continue`, `break`, `return` и т.д.. Компилятор генерирует код, который сохраняет возвращаемое значение во временной переменной, затем выполняет инструкции в блоке `__finally`, и только после этого возвращает сохраненное значение. Локальная раскрутка может отрицательно сказаться на быстродействии программы, так как требует генерации и выполнения большого количества дополнительных машинных команд.
- **Как избежать локальной раскрутки?** Чтобы избежать локальной раскрутки, рекомендуется **избегать любых операторов, способных вызвать преждевременный выход из блока `__try`** обработчика завершения, таких как `return`, `continue`, `break`, `goto`. В идеале, эти операторы следует удалить как из блоков `__try`, так и из блоков `__finally`. Использование ключевого слова `__leave` в блоке `__try` также помогает, поскольку оно вызывает переход в конец этого блока, что рассматривается как естественный выход, избегая при этом проблем, связанных с локальной раскруткой.

- **Причины, по которым следует применять обработчики завершения?** Применение обработчиков завершения рекомендуется по следующим причинам:
    - **Упрощается обработка ошибок** – очистка гарантируется и проводится в одном месте.
    - **Улучшается восприятие текста программ**.
    - **Облегчается сопровождение кода**.
    - **Удается добиться минимальных издержек** по скорости и размеру кода — при условии правильного применения обработчиков.

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?

- **Что такое исключение? Исключение – это событие, возникающее из-за выполнения определенной команды, которая вызвала ошибку процессора**. Такие ошибки, как попытки обращения по неверному адресу или деления на нуль, могут случиться, и операционная система предоставляет приложению возможность определить тип исключения и самостоятельно его обработать.
- **Что такое аппаратное и программное исключения?**
    - **Аппаратное исключение (hardware exception)**: Это исключение, возбужденное самим процессором в ответ на ошибки, такие как деление на нуль или неверный доступ к памяти.
    - **Программное исключение (software exception)**: Это исключения, которые операционная система и прикладные программы способны генерировать самостоятельно.
- **Что такое защищённый блок? Защищенный или охраняемый блок кода – это блок кода, ограниченный фигурными скобками оператора `__try`**. В этом блоке предполагается возникновение исключений, которые могут быть обработаны.
- **Поясните принципы работы обработчика исключений.** Синтаксис обработчика исключений включает блок `__try` и следующий за ним блок `__except`. За блоком `__try` обязательно должен следовать либо `__finally`, либо `__except`, но не оба одновременно, и не несколько блоков одного типа. Однако `try-finally` можно вложить в `try-except` и наоборот. В отличие от обработчиков завершения, **фильтры и обработчики исключений выполняются непосредственно операционной системой**, с минимальной нагрузкой на компилятор. Если исключение не произошло, код в блоке `__except` не выполняется. Если же исключение произошло, операционная система ищет начало блока `__except` и проверяет выражение, указанное в качестве фильтра исключений.
- **Что такое фильтры? Фильтр исключений** – это выражение, указанное в скобках после `__except`, которое должно возвращать один из трех предопределенных идентификаторов. Фильтр анализирует ситуацию (например, тип исключения) и определяет, какое значение вернуть. Встраиваемая функция `GetExceptionCode` возвращает идентификатор типа исключения, а `GetExceptionInformation` возвращает указатель на структуру `EXCEPTION_POINTERS` с детальной информацией об исключении; обе функции могут быть вызваны только в фильтре исключений или обработчике исключений (для `GetExceptionCode`).
- **Какие есть стандартные фильтры и как они работают?** Стандартные фильтры исключений включают:

    - **`EXCEPTION_EXECUTE_HANDLER`**: Это значение сообщает системе, что исключение распознано, и есть код для его обработки. Система выполняет глобальную раскрутку стека, затем управление передается коду внутри блока `__except` (коду обработчика исключений). После выполнения обработчика система считает исключение обработанным, и программа продолжает работу с инструкции, следующей за блоком `__except`.
    - **`EXCEPTION_CONTINUE_EXECUTION`**: Система возвращается к инструкции, вызвавшей исключение, и пытается выполнить ее снова.
    - **`EXCEPTION_CONTINUE_SEARCH`**: Данный идентификатор указывает системе продолжить поиск обработчика исключений, переходя к предыдущему блоку `__try` (которому соответствует `__except`), игнорируя блоки `__finally` на этом пути.
- **Что такое глобальная раскрутка? Глобальная раскрутка (global unwind)** происходит, когда исключение возникает в функции, вызванной из блока `__try`, и фильтр исключений возвращает `EXCEPTION_EXECUTE_HANDLER`. В ходе глобальной раскрутки система продолжает обработку всех незавершенных блоков `try-finally`, выполнение которых началось после блока `try-except`, обрабатывающего исключение. Глобальную раскрутку можно остановить, если в блок `__finally` включить оператор `return`.
- **Как возбудить исключения в SEH?** Можно самостоятельно генерировать программные исключения, вызвав функцию **`RaiseException`**. Параметры функции включают код исключения (`dwExceptionCode`), флаги (`dwExceptionFlags`, например, `EXCEPTION_NONCONTINUABLE`), количество дополнительных аргументов (`nNumberOfArguments`) и массив этих аргументов (`lpArguments`), которые могут содержать дополнительную информацию об исключении. Это позволяет функциям сообщать о неудаче вызывающему коду, посылать информационные сообщения в системный журнал событий или уведомлять о внутренних фатальных ошибках.
- **Что такое необработанное исключение? Необработанное исключение (unhandled exception)** возникает, если все фильтры исключений (`__except` блоки) возвращают `EXCEPTION_CONTINUE_SEARCH`, и система не находит обработчика. В таких случаях может быть вызвана особая функция фильтра, предоставляемая операционной системой: `UnhandledExceptionFilter`. Эта функция выводит окно, информирующее о необрабатываемом исключении, и предлагает закрыть процесс или начать его отладку. Поведение `UnhandledExceptionFilter` можно изменить, установив пользовательский фильтр с помощью функции `SetUnhandledExceptionFilter`.

28. Безопасное программирование

Безопасное программирование — это подход к разработке программного обеспечения, цель которого заключается в предотвращении, обнаружении и реагировании на угрозы безопасности. Его **основная цель** – защитить данные, системы и пользователей от несанкционированного доступа, модификации или уничтожения. В контексте системного программирования, оно включает разработку и управление низкоуровневыми компонентами, такими как операционные системы, драйверы устройств и компиляторы, с учетом принципов безопасности.

Уязвимость – это недостаток программы, который может быть использован для реализации угроз безопасности информации. **Недостаток программы** – это любая ошибка, допущенная в ходе проектирования или реализации программы, которая, в случае её неисправления, может стать причиной уязвимости. Обычно уязвимость позволяет

атакующему «обмануть» приложение, заставив его выполнить непредусмотренные действия или совершить действия, на которые у него не должно быть прав.

Классификация уязвимостей ПО является важным аспектом безопасности, помогающим разработчикам и специалистам по кибербезопасности понимать и управлять рисками. Существуют несколько популярных классификаторов, таких как:

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

Категории ошибок ПО, приводящих к уязвимостям, делятся на:

- **Проблемы проектирования**: например, программист не продумал требуемый тип аутентификации.
- **Проблемы с реализацией**: например, программист случайно допустил ошибку, используя небезопасный библиотечный метод, или попытался сохранить слишком много данных в переменной.

Список распространённых ошибок, ставящих под угрозу безопасность современных программ, включает:

- 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)
- Проблемы мобильного кода (Mobile Code Issues)
- 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

Ошибка переполнения буфера (Buffer Overflow): Это происходит, когда буфер переполняется в другую область памяти, например, в соседнюю переменную. В примере, когда функция `scanf` считывает входные данные без ограничения длины, ввод очень

длинной строки может привести к перезаписи данных в соседних переменных в стеке. Это может изменить значение других переменных, приводя к непредсказуемому поведению программы или даже к выполнению произвольного кода. **Как избежать:**

- Можно безопасно использовать `scanf()`, указав длину буфера в строке формата (например, `%31[^\n]`).
- Рекомендуется использовать функцию `fgets()`, которая принимает длину буфера для чтения в качестве параметра, предотвращая переполнение.

Ошибка целочисленного переполнения (Integer overflow): Это ситуация, когда результат арифметической операции над целыми числами **превышает максимальное значение**, которое может быть представлено данным типом данных. Это может привести к непредсказуемому поведению, включая сбои, неправильные вычисления и уязвимости безопасности. Проблемы, к которым может привести целочисленное переполнение, включают неправильные вычисления, сбои программы, уязвимости безопасности, утечку информации и проблемы с производительностью. **Как избежать:**

- **Использование безопасных функций**: Использование функций, которые проверяют возможность переполнения перед выполнением операции.
- **Проверка границ**: Всегда проверять, не превышают ли результаты арифметических операций допустимые пределы типа данных.
- **Использование типов данных с большей емкостью**: Использование типов данных, таких как `long long` вместо `int`, если это возможно.
- **Статический и динамический анализ кода**: Использование инструментов для статического и динамического анализа кода (например, Clang Static Analyzer, Valgrind) для обнаружения потенциальных переполнений.
- **Обучение и осведомленность**: Обучение разработчиков методам безопасного программирования и повышение осведомленности о рисках целочисленного переполнения.

29. Безопасное программирование (продолжение)

Ошибка форматирования строк: Некоторые функции, такие как `printf()`, получают строку формата, за которой следуют переменные для отображения. Если пользовательский ввод передается напрямую в качестве строки формата (например, `printf(string);`), злоумышленник может использовать спецификаторы формата (например, `%x`, `%d`) для чтения содержимого стека или даже записи в память. Например, ввод `%d` может заставить `printf` прочитать число из стека, а ввод `AAAA %x %x %x ...` может отобразить содержимое стека в шестнадцатеричном виде, раскрывая конфиденциальные данные. **Как избежать:**

- Все данные, поступающие из ненадежного источника, должны быть **канонизированы**, **проверены** (валидация) и **обработаны** (очистка) перед использованием.
- Корректным способом вывода строкового значения является `printf("%s", string);`.

Канонизация, валидация и очистка:

- **Канонизация (Canonicalization)**: Процесс преобразования данных, имеющих несколько возможных представлений, в «стандартную» или «нормальную» форму для избежания путаницы.
- **Валидация (Validation)**: Проверка того, что данные представлены в ожидаемом формате. Это более безопасный подход, чем очистка, и обычно включает проверку успешности канонизации и соответствие данных ожидаемому диапазону.
- **Очистка (Sanitization)**: Удаление любого потенциально опасного форматирования или содержимого из переменной, чтобы сделать вводимые данные безопасными для использования. Например, удаление `<script>alert(1)</script>` из поля даты.

Триада CIA (конфиденциальность, целостность, доступность) — это основа безопасности операционных систем:

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

Другие способы повышения безопасности в рамках ОС включают:

- Изолирование доменов безопасности (ядра, процессов, виртуальных машин).
- Простота, позволяющая минимизировать поверхность атаки.
- Блокировка доступа к ресурсам по умолчанию.
- Проверка всех запросов на авторизацию.
- **Принцип наименьших полномочий (PoLP)**.
- Chains of trust.
- Разделение привилегий.
- Сокращение общих данных.
- Повышение надежности операционной системы для уменьшения уязвимостей, например, **рандомизация расположения адресного пространства (ASLR)**, целостность потока управления, **предотвращение выполнения данных (DEP)** и другие методы.

ASLR (Address Space Layout Randomization – «рандомизация размещения адресного пространства») – это технология, при которой **случайным образом изменяется расположение в адресном пространстве процесса важных структур данных**, таких как образы исполняемого файла, подгружаемые библиотеки, куча и стек. Это усложняет эксплуатацию уязвимостей, таких как переполнение буфера, так как атакующему становится сложнее угадать адреса, куда можно поместить шелл-код.

DEP (Data Execution Prevention – Предотвращение выполнения данных) – это функция безопасности, встроенная в различные ОС, которая **не позволяет приложению исполнять код из области памяти, помеченной как «только для данных»**. Она предотвращает атаки, которые пытаются сохранить и исполнить код в такой области, например, с помощью переполнения буфера.

PoLP (Principle of Least Privilege – Принцип наименьших полномочий) – 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. **Аспекты PoLP**:

- **Минимизация прав доступа**: Предоставление только необходимых прав.
- **Разделение обязанностей**: Разделение задач между субъектами для предотвращения концентрации полномочий.
- **Контроль доступа на основе ролей (RBAC)**: Назначение прав доступа на основе ролей, а не индивидуальных учетных записей.
- **Минимизация времени доступа**: Предоставление доступа только на необходимое время.

Лучшие практики в области безопасного программирования:

- **Информируйте себя**: Следите за обсуждениями уязвимостей на открытых форумах, читайте книги и статьи, изучайте программное обеспечение с открытым исходным кодом, но будьте осторожны.
- **Обращайтесь с данными с осторожностью**:
    - **Очистка данных**: Проверяйте входные данные на наличие злого умысла.
    - **Проверка границ**: Убедитесь, что предоставленные данные помещаются в отведенное пространство (например, индексы массива).
    - **Проверяйте конфигурационные файлы**: Относитесь к данным из них как к потенциально измененным злоумышленником.
    - **Проверяйте параметры командной строки**: Пользователь может ввести их напрямую, пытаясь обмануть программу.
    - **Проверяйте переменные среды**: Злоумышленники могут использовать их для изменения поведения программ.
    - **Проверяйте другие источники данных**: Будьте осторожны со всеми источниками ввода.
    - **Будьте осторожны с косвенными ссылками на файлы**: Злоумышленник может обманом заставить программу читать или записывать непредназначенные файлы.
- **Повторно используйте «хороший код»**, когда это возможно.
- **Не пишите код, который использует относительные имена файлов**: Ссылки на имена файлов должны быть «полными».
- **Не ссылайтесь на файл дважды в одной и той же программе по его имени**: Откройте файл один раз по имени и используйте дескриптор файла.
- **Не вызывайте ненадежные программы из надежных**.
- **Не думайте, что ваши пользователи не являются злоумышленниками**: Всегда перепроверяйте внешнюю информацию.
- **Не рассчитывайте на успех системных вызовов**: Всегда проверяйте условия завершения системного вызова и обрабатывайте ошибки.
- **Не вызывайте оболочку или командную строку**.
- **Не выполняйте проверку подлинности по ненадежным критериям**.
- **Не используйте хранилище, доступное для глобальной записи**, даже временно.
- **Не доверяйте хранилищу, доступному для записи пользователем**, чтобы избежать несанкционированного доступа.

30. Управление доступом

Контроль доступа к ресурсам — это механизм, который **разрешает доступ к ресурсу только авторизованным пользователям** этого ресурса, то есть только тем пользователям, которым разрешен доступ. Часто механизм авторизации пользователя включает также и его аутентификацию, то есть установление подлинности пользователя.

В системах информационной безопасности все ресурсы разбиваются на две категории:

- **Объекты**: Пассивные ресурсы, которые требуется защитить, например, файлы, каналы передачи данных, оперативная память, принтеры. Для каждого объекта определяется множество операций, которые можно выполнить над ним.
- **Субъекты**: Активные ресурсы, которые выполняют операции над объектами и представляются процессами, исполняемыми от имени пользователей информационной системы.

Политика безопасности — это набор требований, выполнение которых обеспечивает безопасную работу информационной системы. **Менеджер безопасности** (или монитор безопасности) — это субъект (программа), который контролирует доступ других субъектов к объектам, руководствуясь при этом некоторыми правилами (политикой безопасности). Он всегда находится в активном состоянии.

Разница между правами и привилегиями:

- **Право** — это возможность субъекта выполнять некоторые операции над объектами (например, читать файл).
- **Привилегия** — это более общее понятие, позволяющее пользователю выполнять действия в отношении других объектов и субъектов системы информационной безопасности. Как правило, привилегия выделяет пользователя из числа обычных пользователей системы (например, блокировать доступ).

Порядок разработки политики безопасности разбивается на два этапа:

1. **Первый этап**: Анализ угроз для системы информационной безопасности.
    - Определение информационных ресурсов, которые должны быть защищены.
    - Определение возможных угроз этим ресурсам (внутренние от ПО и внешние от пользователей).
2. **Второй этап**: Разработка средств защиты от возможных угроз безопасности.
    - Средства защиты должны предусматривать работу в трех режимах: обработка рутинных задач, обработка исключительных ситуаций (обнаружение атаки вируса), обработка аварийных ситуаций (обнаружение вируса).

Модель безопасности — это формальное описание разработанной политики безопасности. Она рассматривается как система, включающая следующие компоненты: пассивные ресурсы (объекты), наборы операций над каждым объектом, активные ресурсы (субъекты), атрибуты защиты объектов (описывают права доступа субъектов к объектам).

Состояние системы безопасности — это состояние объектов, наборов операций, субъектов и атрибутов защиты объектов. Состояние системы безопасности называется **безопасным**, если в этом состоянии нет несанкционированного доступа субъекта к объекту. **Общая задача системы безопасности** формулируется следующим образом:

- Начальное состояние системы безопасное.
- Правила управления доступом обеспечивают переход из безопасного состояния системы в безопасное состояние системы.
- Любое состояние системы, достижимое из ее начального состояния, является безопасным состоянием.

31. Управление доступом (продолжение)

Дискреционная политика безопасности основывается на принципах, где права доступа субъекта к объекту определяются другим субъектом, который обладает этими правами. Суть в том, что доступ субъектов к объектам разрешается другими субъектами или оставлен на их усмотрение. Менеджер безопасности проверяет наличие разрешения у субъекта на выполнение затребованной операции над объектом. **Основные принципы**:

- Для каждого объекта определяется набор операций, которые можно выполнять над ним.
- Субъект может выполнить операцию над объектом при условии, если он имеет право на выполнение этой операции.
- Субъект, имеющий права, может передать эти права другому субъекту (в строгой дискреционной политике – только при наличии таких полномочий).

Алгоритм построения дискреционной модели безопасности:

1. Идентифицируются все объекты и все субъекты системы.
2. Для каждого объекта определяются операции, которые субъекты могут выполнять над ним.
3. Строится **матрица управления доступом**, где каждая строка соответствует субъекту, а каждый столбец – объекту.
4. В клетки матрицы записываются права, которые субъект имеет по отношению к соответствующему объекту.

Матрица управления доступом — это матрица, каждая строка которой соответствует одному субъекту, а каждый столбец — одному объекту системы. Клетки матрицы содержат права доступа, которые субъект, соответствующий строке, имеет по отношению к объекту, соответствующему столбцу.

Режимы доступа к объекту (операции, которые разрешается выполнять над объектами) включают:

- **READ (R)**: Разрешается чтение содержимого объекта, часто также 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, но не добавлять.
- **WRITE_UPDATE (WU)**: Разрешается только изменять данные, но не добавлять или удалять.
- **DELETE (D)**: Разрешается удаление объекта.
- **EXECUTE (E)**: Разрешается исполнение объекта.
- **NULL (N)**: Нет доступа к объекту.

Режимы управления объектом определяют субъектов, которые могут изменять права доступа других субъектов к объекту или передавать права управления этим объектом:

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

Модели управления в дискреционной модели безопасности определяют правила передачи прав управления объектами:

- **Иерархическое управление (hierarchical control)**: Субъекты, управляющие правами доступа, упорядочиваются иерархически. Администратор системы безопасности находится на вершине, может управлять всеми правами и наделять ими любого субъекта. Остальные субъекты управляют правами доступа только тех субъектов, которые находятся ниже их в иерархии.
- **Управление правами доступа владельцем объекта (concept of ownership)**: Управление правами доступа к объекту для всех субъектов выполняется его владельцем (обычно тем, кто создал объект). Администратор системы безопасности может ограничить права владельца на передачу прав.
- **Либеральное управление (laissez-fair)**: Передача прав доступа к объекту от субъекта, обладающего этими правами, другому субъекту никак не контролируется. Считается самой ненадежной моделью.
- **Централизованное управление (centralized control)**: Правами управления доступом к объектам обладает только один субъект, как правило, администратор системы. Все вопросы по разрешению доступа разрешаются только администратором. Является наиболее надежной.

Два подхода к хранению матрицы управления доступом (так как она часто разреженная и большого размера):

1. **Построчный подход (capabilities)**: Матрица рассматривается по строкам, каждая строка описывает все объекты, к которым субъект имеет доступ, и режимы доступа к ним. Информация хранится в виде **профиля (profile)** пользователя – списка, где каждый элемент содержит имена объектов и режимы доступа. Менеджер безопасности разрешает доступ только в соответствии с возможностями из профиля.
2. **Столбчатый подход (access control list - ACL)**: Матрица рассматривается по столбцам, каждый столбец описывает всех субъектов, имеющих доступ к данному объекту, и их режимы доступа. Хранится в виде **списка управления доступом (ACL)**, элементы которого называются ACE (access control elements), содержащие имя пользователя и разрешенные режимы доступа. Менеджер безопасности разрешает доступ, если имя субъекта и требуемый режим доступа находятся в ACL объекта.

32. Управление доступом (продолжение)

В операционных системах Windows NT реализована дискреционная модель безопасности.

Маркер доступа (access token) — это объект, который создается для пользователя при регистрации и входе в систему. Он **идентифицирует 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.

Охраняемые объекты (securable objects) — это все объекты Windows, которые могут иметь имя, а также потоки и процессы.

Дескриптор безопасности (security descriptor) — это объект, создаваемый вместе с охраняемым объектом. Он **содержит информацию, необходимую для защиты объекта от несанкционированного доступа**. В дескрипторе безопасности идентифицируется владелец объекта, определяются пользователи и группы пользователей, которым разрешен или запрещен доступ, а также информация для аудита доступа. **Состоит из**:

- **Список управления дискреционным доступом (Discretionary Access-Control List, DACL)**: Для хранения информации о пользователях, которым разрешен или запрещен доступ.
- **Список управления системным доступом (System Access-Control List, SACL)**: Для управления аудитом доступа к объекту.
- Общее название для этих списков – **списки управления доступом (Access-Control Lists)** или сокращенно **ACL**.

Учётная запись пользователя (user account) — это информация о пользователе, которая создается администратором системы при регистрации пользователя и хранится в базе данных менеджера учетных записей (Security Account Manager, SAM). **В учетной записи хранится следующая информация**:

- Имя пользователя для входа в систему (username)
- Пароль (password)
- Полное имя пользователя (full name)
- Допустимое время работы в системе (logon hours)
- Допустимые компьютеры для входа в систему (logon workstations)
- Дата окончания срока действия учетной записи (expiration date)
- Рабочий каталог пользователя (home directory)
- Действия, выполняемые при загрузке (logon script)
- Профиль пользователя (profile)
- Тип учетной записи (account type) **Существуют четыре типа учетных записей**:
- Учетная запись пользователя
- Учетная запись группы пользователей
- Учетная запись компьютера
- Учетная запись домена По умолчанию Windows NT создает три учетных записи: **Administrator**, **Guest**, **System**.

Группа пользователей (group) — это набор учетных записей пользователей, объединенных по какому-либо признаку. Одна учетная запись пользователя может входить в более чем одну группу. Каждая группа имеет свою учетную запись и наделена правами и полномочиями, которые передаются каждому члену группы. **Различают три типа групп** на платформе Windows NT:

- **Глобальные группы**: Используются для организации пользователей с целью упорядочения их доступа к ресурсам вне домена, в котором создана группа.

- **Локальные группы**: Используются для организации доступа пользователей к ограниченному множеству ресурсов внутри домена.
- **Специальные группы**: Создаются системой по умолчанию для управления доступом к ресурсам. Членство в них предопределено и не может быть изменено.

SID (Security Identifier – Идентификатор безопасности) — это бинарное представление учетной записи, которое операционная система создает для каждой учетной записи и хранит в базе данных SAM. Он **используется системой безопасности для идентификации учетных записей** и ускоряет работу системы безопасности.
Символическая структура SID: `S – R – I – SA0 – SA1 – SA2 –SA3 – SA4 …`:

- **S**: Символ `S`, обозначающий, что дальнейшее числовое значение является идентификатором безопасности.
- **R**: **Версия формата** идентификатора безопасности (всегда `1` начиная с Windows NT 3.1).
- **I**: **Уровень авторизации учетной записи** (48-битное число, также называемое Identifier Authority). Предопределенные значения уровней авторизации учетных записей, такие как `SECURITY_NULL_SID_AUTHORITY`, `SECURITY_WORLD_SID_AUTHORITY`, `SECURITY_LOCAL_SID_AUTHORITY`, `SECURITY_NT_AUTHORITY` и другие.
- **SA** (Subauthority): 32-битное число, уточняющее уровень авторизации учетной записи (также называемое Relative Identifier, RID). Количество полей SA может быть произвольным, но не превышать 15.
    - Для SID пользователей и групп, поля SA имеют следующий смысл:
        - `SA0` уточняет авторизацию учетной записи.
        - `SA1`, `SA2`, `SA3` представляют уникальный 96-битовый идентификатор компьютера.
        - `SA4` нумерует идентификаторы безопасности, создаваемые внутри системы (номера от 0 до 999 зарезервированы для системы, с 1000 — для новых идентификаторов).
    - Предопределенные группы имеют структуру `S – R – I – SA0`.
    - Таблица предоставляет символьные обозначения для RID, используемых с `SECURITY_NT_AUTHORITY` (S-1-5) для создания SID, известных на платформе Windows (например, `DOMAIN_ALIAS_RID_ADMINS`, `SECURITY_BUILTIN_DOMAIN_RID`).

33. Перехват API

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.

Выполнение кода программы в ОС: Для компьютера любая программа представляет собой **последовательный набор инструкций**, которые зачастую объединены в блоки, называемые функциями, процедурами или подпрограммами. Таким образом, программа — это некоторый набор инструкций и вызовов подпрограмм, расположенных в порядке, необходимом для достижения определённого результата. Если выполняемая программа состоит только из инструкций, известных процессору, то процесс её выполнения происходит по простому сценарию:

- Загрузка из памяти следующей инструкции.
- Декодирование полученной инструкции.
- Расчёт эффективных адресов для данных.
- Выполнение инструкции.

Функция является наиболее фундаментальной языковой возможностью для **абстрагирования и повторного использования кода**. Она позволяет ссылаться на некоторый фрагмент кода по имени. Для использования функции достаточно знать, сколько аргументов требуется, какого типа аргументы, что возвращает функция и что она выполняет.

При вызове функции происходит следующее:

- Аргументы должны быть преобразованы в значения (по крайней мере, для языков программирования, подобных C).
- Затем поток управления переходит к телу функции, и код начинает выполняться там.
- Как только встречается оператор `return`, работа с функцией завершается, и управление возвращается обратно к месту вызова функции.

Стек вызовов (call stack) — это стек, хранящий информацию для возврата управления из подпрограмм (процедур, функций) в программу (или подпрограмму, при вложенных или рекурсивных вызовах), а также для возврата в программу из обработчика прерывания (в том числе при переключении задач в многозадачной среде). **Принцип работы стека вызовов:**

- При вызове подпрограммы или возникновении прерывания, в стек заносится **адрес возврата** — адрес в памяти следующей инструкции приостановленной программы, и управление передается подпрограмме или подпрограмме-обработчику.
- При последующем вложенном или рекурсивном вызове, прерывании подпрограммы или обработчика прерывания, в стек заносится очередной адрес возврата.
- При возврате из подпрограммы или обработчика прерывания, адрес возврата снимается со стека, и управление передается на соответствующую инструкцию. Современные системы программирования в вопросе организации исполнения кода программ работают по модели распределения памяти на основе стека, где для хранения параметров (аргументов) процедур и функций, их локальных переменных, а также адреса возврата выделяется специальная область памяти, организованная в виде стека, называемая стеком вызовов. Этот стек для каждого потока выделяется отдельно.

Стековые кадры (stack frames) — это машинно-зависимые и ABI-зависимые структуры данных, содержащие информацию о состоянии подпрограммы. Каждый кадр стека соответствует вызову подпрограммы, который еще не завершился возвратом. Стековый фрейм обычно содержит, по крайней мере, следующие элементы:

- аргументы (значения параметров), передаваемые подпрограмме.
- адрес, возвращаемый вызывающей программе.
- пространство для локальных переменных подпрограммы.

Соглашения о вызовах определяют, как функция вызывается, как функция управляет стеком и стековым кадром, как аргументы передаются в функцию, как функция возвращает значения. Они важны при перехвате, так как для успешного перенаправления вызова функции и её корректной обработки необходимо точно знать, как ожидается передача аргументов, кто отвечает за очистку стека и какие регистры должны быть сохранены/восстановлены.

Существующие 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 значений из функции 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, размер которых меньше 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 (пролог), сохраняющий значения используемых регистров и записывающий аргументы в стек.
  - После вызова функции вставляется код (эпилог), восстанавливающий значения регистров и очищающий стек от локальных переменных.
- **fastcall (Fast calling convention)**:
  - Общее название соглашений, передающих параметры через **регистры**.
  - Если регистров недостаточно, дополнительно используется стек.
  - Не стандартизировано.
  - В 32-разрядной версии компилятора Microsoft определяет передачу первых двух параметров 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 на исходную позицию возвращает **вызываемая подпрограмма**.
  - Изменяемые параметры передаются только по ссылке.
  - Возвращаемое значение передаётся через изменяемый параметр `Result`.

34. Перехват API

Перехват API-функций (API hooking) — это техника программирования, при которой вызовы функций из библиотеки API перенаправляются на пользовательские функции.

Основные методы перехвата: Методы перехвата можно разделить по критерию режима выполнения:

- **Пользовательские методы (User-Mode)**:
  - **Модификация IAT таблиц** (таблиц импорта).
  - **Сплайсинг** (непосредственное изменение функции).
  - Особенность этих методов в том, что **невозможно что-либо изменить в поведении ядра операционной системы и его расширений**.
- **Методы режима ядра (Kernel-Mode)**:
  - **Модификация SSDT/IDT таблиц** (системных 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: В ОС Windows NT система построена на динамически загружаемых библиотеках (DLL), которые предоставляют приложениям сервисные API функции для взаимодействия с системой. Перехват API функций позволяет обойти многие ограничения системы.

- **Сплайсинг (splicing)**:
  - 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 сохранить заменяемый участок памяти у себя, а после отработки функции перехвата восстанавливает изменённый участок и даёт полностью выполниться настоящей функции.
  - Все функции стандартных DLL Windows поддерживают **hot-patch point**, где перед началом функции располагаются пять неиспользуемых однобайтовых операций `nop`, а сама функция начинается с двухбайтовой инструкции `mov edi, edi`. Места, занимаемого пятью `nop`, достаточно, чтобы разместить команду перехода на функцию-перехватчик.
- **Трамплин**:
  - 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, делая её "подцепленной". При вызове подцепленной функции активируется шеллкод трамплина, и поток выполнения передается 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)**:
  - Альтернативный метод выполнения API-хуков, работающий аналогично хуку на основе трамплина.
  - Разница в том, что встраиваемые хуки **возвращают выполнение законной функции**, позволяя нормальному выполнению продолжаться.

- Они сложнее в реализации и потенциально труднее в обслуживании, но более эффективны.
- Существуют открытые библиотеки, такие как Detours или Minhook, для реализации API-хуков, а также API Windows с ограниченными возможностями.

Перехват API-вызовов путём модификации таблиц импорта:

- **Локальный перехват** может быть реализован в WinNT посредством подмены адреса перехватываемой функции в **таблице импорта (IAT)**.
- В разделе импорта содержится список DLL, необходимых модулю для нормальной работы, а также все идентификаторы, которые модуль импортирует из каждой DLL.
- Вызывая импортируемую функцию, поток получает её адрес фактически из раздела импорта.
- Чтобы перехватить определённую функцию, необходимо **найти её адрес в таблице импорта и изменить его на адрес нашего обработчика**.
- **Достоинства** данного метода: код перехватываемой функции не изменяется, что обеспечивает корректную работу в многопоточном приложении.
- **Недостатки**: приложения могут сохранить адрес функции до перехвата и затем вызывать её минуя обработчик.

Для чего может использоваться перехват API-функций:

- **Добросовестное использование**:
  - **Отладка и мониторинг**: перехват может использоваться для перехвата и регистрации вызовов функций в целях отладки, помогая разработчикам понять и диагностировать проблемы.
  - **Программное обеспечение для обеспечения безопасности**: антивирусное программное обеспечение часто использует перехват для отслеживания и перехвата потенциально вредоносных действий.
  - **Мониторинг производительности**: для сбора метрик производительности и оптимизации ПО.
  - **Расширение функциональных возможностей**: для добавления или изменения функциональных возможностей в существующее ПО без изменения исходного кода.
- **Недобросовестное использование**:
  - **Руткиты**: вредоносное ПО может использовать перехват, чтобы скрыть своё присутствие в системе, затрудняя его обнаружение и удаление.
  - **Кража данных**: перехват может быть использован для перехвата конфиденциальных данных (пароли, ключи шифрования).
  - **Несанкционированный доступ**: для обхода механизмов безопасности и получения несанкционированного доступа к системам или данным.
  - **Распространение вредоносного ПО**: путём перехвата и изменения сетевого трафика или файловых операций.
- **Этические соображения** при использовании методов перехвата включают:
  - **Соблюдение законодательства**: убедиться, что использование соответствует законам.
  - **Информированное согласие**: получить согласие пользователей при законном использовании.

- **Прозрачность**: быть откровенным в отношении использования методов перехвата.
- **Рекомендации по обеспечению безопасности**: применять безопасные методы кодирования, регулярные проверки безопасности и мониторинг несанкционированных действий.

35. Перехват API

Перехват API-функций (API hooking) — это техника программирования, при которой вызовы функций из библиотеки API перенаправляются на пользовательские функции.

Перехват API-вызовов путём модификации системных таблиц:

- **Суть**: Перехват API-вызовов путём модификации системных таблиц **SSDT (System Service Dispatch Table)** — это техника, используемая для изменения поведения операционной системы Windows. SSDT содержит адреса функций, которые реализуют системные вызовы ядра. Эти функции вызываются приложениями и драйверами для выполнения различных операций (управление процессами, файлами, памятью и т.д.).
- **Что такое SSDT**: SSDT сопоставляет системные вызовы с адресами функций ядра. Когда системный вызов выполняется приложением пользовательского пространства, он содержит служебный индекс, указывающий, какой системный вызов вызывается. Затем SSDT используется для определения адреса соответствующей функции внутри `ntoskrnl.exe`.
- **Алгоритм основных шагов для перехвата API-вызовов путём модификации SSDT**:
    1. **Получение адреса SSDT**: Адрес SSDT можно получить, используя системные структуры и функции Windows (обычно недокументированные структуры и функции 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, чтобы избежать постоянного перехвата.

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 сохранит старые указатели на функции, а затем заменит основной массив функций в объекте драйвера на свои собственные функции.
- Любой запрос, поступающий к устройству под управлением перехваченного драйвера, будет вызывать диспетчерские процедуры перехватывающего драйвера.
- Чтобы подключить драйвер, нужно найти указатель на объект драйвера (`DRIVER_OBJECT`), используя 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:

- Перехват API-вызовов путём модификации системных таблиц и драйверами-фильтрами являются трудно реализуемыми, в немалой степени по причине существования такой технологии как **PatchGuard (Kernel Patch Protection, KPP)**.
- Основная цель PatchGuard – **предотвратить модификацию критических структур данных ядра и таблиц системных вызовов**, таких как SSDT, IDT (Interrupt Descriptor Table) и GDT (Global Descriptor Table), 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** (синий экран смерти), чтобы предотвратить дальнейшее выполнение вредоносного кода.

Для чего может использоваться перехват API-функций: (См. раздел "Для чего может использоваться перехват API-функций" в ответе на вопрос 34).

36. Оптимизация кода

Оптимизация кода — это процесс преобразования части кода в другую функционально эквивалентную часть для улучшения одной или более характеристик кода.

Характеристики, которые могут быть оптимизированы:

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

Стоит ли оптимизировать код вручную? Однозначного ответа нет, всё зависит от ситуации. Компиляторы постоянно совершенствуются в отношении методов оптимизации кода, но они не идеальны. Часто продуктивнее использовать специфические средства компилятора и дать ему возможность оптимизировать код, чем тратить время на ручную оптимизацию. Тем не менее, для достижения высоких уровней производительности код HLL (High Level Language) должен быть написан соответствующим образом, что требует чёткого понимания того, как работают компьютеры и исполняется программное обеспечение. Поэтому даже если не требуется ручная оптимизация, важно понимать, как

её проводит компилятор и что в исходном коде может препятствовать эффективной работе оптимизирующего компилятора. Как говорил Дональд Кнут – **«Преждевременная оптимизация – это корень всех бед»**. Разработчик не должен при разработке думать об оптимальности, а должен думать о целостности и завершенности своего кода; нет смысла писать оптимальный код, который не выполняет своего предназначения.

Основные принципы проведения оптимизации: Оптимизация должна проводиться **строго при необходимости и с осторожностью**. Каждый шаг оптимизации должен быть тщательно отлажен и протестирован.

Ключевые аспекты связи оптимизации и системного программирования: Системное программирование играет критическую роль в обеспечении стабильности, производительности и безопасности всей компьютерной системы. Оптимизация кода и системное программирование тесно связаны по следующим аспектам:

- **Производительность**: Системное программирование часто требует высокой производительности (ОС, драйверы, встроенные системы), и оптимизация кода помогает достичь необходимых уровней.
- **Эффективное использование ресурсов**: Оптимизация направлена на минимизацию использования процессорного времени, памяти и других ресурсов, что важно для управления ресурсами на низком уровне в системном программировании.
- **Надежность и стабильность**: Оптимизация может улучшить структуру кода, устранить узкие места и повысить общую стабильность программы, что критично для системного программирования, где ошибки могут привести к серьёзным сбоям.
- **Совместимость и переносимость**: Оптимизация может адаптировать код для работы на различных платформах и архитектурах, обеспечивая совместимость и переносимость в системном программировании.
- **Безопасность**: Оптимизация может улучшить безопасность кода, устранить уязвимости и защитить от атак, что требует высокого уровня безопасности в критически важных системах.
- **Отладка и тестирование**: Оптимизация включает процессы отладки и тестирования для выявления и устранения проблем, что крайне важно для системного программирования из-за возможных серьёзных последствий ошибок.

Уровни оптимизации: Возможно несколько уровней оптимизации:

- **На самом абстрактном уровне**: **выбор лучшего алгоритма**. Этот метод не зависит от компилятора и языка программирования.
- **Снижая уровень абстракции**: **ручная оптимизация кода на основе используемого HLL**, не зависящая от конкретной реализации языка. Такие оптимизации применимы к разным компиляторам для одного и того же языка.
- **Ещё ниже**: **структурирование кода таким образом, чтобы оптимизация была применима только к определённому поставщику или версии компилятора**.
- **На самом низком уровне**: **учитывать машинный код, который выдаёт компилятор, и корректировать инструкции, написанные на HLL, так, чтобы заставить компилятор генерировать определённую оптимизированную последовательность машинных инструкций**.

37. Оптимизация кода

Оптимизация кода — это процесс преобразования части кода в другую функционально эквивалентную часть для улучшения одной или более характеристик кода.

Принципы оптимизации кода компилятором:

- Преобразование промежуточного кода «в более эффективную форму» не является четко определённым процессом, так как то, что делает одну форму программы более эффективной, чем другую, может варьироваться.
- Основное определение эффективности заключается в том, что программа сводит к минимуму использование некоторых системных ресурсов, обычно **памяти (пространства) или циклов процессора (скорости)**.
- Оптимизация для достижения одной цели (например, повышения производительности) может конфликтовать с другой целью (например, сокращением использования памяти), поэтому процесс оптимизации обычно представляет собой **компромисс**.
- Компиляторы могут применять только **безопасные оптимизации**. Это значит, что компилятор может изменять программу только так, чтобы это **не изменило её поведение для всех входных данных**.
- Существуют определённые характеристики кода, которые не позволят компилятору совершить оптимизацию, называемые **блокировщиками оптимизации**. К ним относятся:
    - **Указатели**: Компилятор не может точно знать, будут ли два указателя указывать на одну и ту же область памяти, и поэтому не выполняет некоторые оптимизации. Например, функции с указателями, которые могут указывать на одну и ту же область памяти, могут иметь разное поведение при оптимизации, что препятствует преобразованию кода.
    - **Вызов функций**: Вызовы функций влекут накладные расходы, и компилятору тяжело определить, имеет ли вызов функции побочные эффекты (изменение глобального состояния). Если он не может этого сделать, он предполагает худшее и не выполняет оптимизацию.

Анализ потока данных (DFA):

- Это процесс, в ходе которого оптимизатор отслеживает значения переменных по мере прохождения управления по программе.
- После тщательного анализа компилятор может определить, где переменная не инициализирована, когда переменная содержит определённые значения, когда программа больше не использует переменную, и когда компилятор просто ничего не знает о значении переменной.

Базовые блоки:

- Для анализа потока данных компиляторы разбивают исходный код на последовательности, известные как **базовые блоки**.
- **Базовые блоки** — это последовательности машинных инструкций, от которых нет ответвлений, кроме как в начале и конце.
- **Зачем они нужны?** Базовые блоки позволяют компилятору легко отслеживать, что происходит с переменными и другими программными объектами. Когда

компилятор обрабатывает каждую инструкцию, он может (символически) отслеживать значения, которые будут храниться в переменной.

Упрощаемые графики потоков:

- Плохо структурированные программы могут создавать пути потока управления, которые сбивают с толку компилятор, уменьшая возможности оптимизации.
- **Хорошие программы создают упрощаемые графики потоков** — наглядные изображения пути потока управления.
- **Какая программа будет упрощаемой?** Любая программа, состоящая только из структурированных управляющих команд (`if`, `while`, `repeat..until` и т.д.) и избегающая операторов `goto`, будет упрощаемой.
- Это важно, потому что оптимизаторы компиляторов, как правило, гораздо лучше справляются с работой над упрощаемыми программами. Их базовые блоки могут быть схематично свёрнуты, а заключённые в них блоки наследуют свойства от вложенных блоков. Такой иерархический подход к оптимизации более эффективен и позволяет оптимизатору сохранять больше информации о состоянии программы.

Основные типы оптимизаций компилятора:

- **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 программа присвоила эту константу переменной ранее в коде. Часто приводит к более эффективному коду, так как манипулирование непосредственными константами эффективнее, чем переменными.
- **Удаление мёртвого кода**: удаление объектного кода, связанного с определённым оператором исходного кода, когда программа никогда не будет использовать результат этого оператора или когда условный блок никогда не будет истинным.
- **Удаление общих подвыражений**: поиск экземпляров одинаковых выражений и анализ возможности замены их на одну переменную, содержащую вычисленное значение. Если значения переменных в подвыражении не изменились, его не нужно пересчитывать везде.
- **Снижение стоимости операций**: замена медленных операций (умножение, деление) на более быстрые (сложение, вычитание, сдвиг).
- **Анализ индуктивных переменных**: в выражениях, особенно в циклах, значение одной переменной полностью зависит от другой; компилятор может исключить или объединить вычисления.
- **Анализ инвариантов цикла**: вычисление результата выражения, которое не меняется на каждой итерации цикла, только один раз вне цикла, а затем использование вычисленного значения в теле цикла.
- **Раскрутка циклов**: искусственное увеличение количества инструкций, исполняемых в течение одной итерации цикла. Позволяет увеличить количество параллельно исполняемых блоков инструкций и более интенсивно использовать регистры процессора, кэш данных и исполнительных устройств.
- **Вычисления по короткой схеме (Short-circuit evaluation)**: стратегия вычисления, при которой второй логический оператор выполняется или вычисляется только в том случае, если первого логического оператора недостаточно для определения значения выражения. Это позволяет прекратить вычисление, как только результат становится очевидным (например, для `AND` и `OR`).

38. Оптимизация кода

Оптимизация кода — это процесс преобразования части кода в другую, функционально эквивалентную часть, с целью **улучшения одной или более характеристик кода**. Две самые важные характеристики — это **скорость работы и размер кода**. К другим характеристикам относятся энергопотребление, необходимое для выполнения кода, время компиляции кода и длительность JIT-компиляции. Оптимизация кода должна проводиться строго при необходимости и с осторожностью, поскольку «преждевременная оптимизация — это корень всех бед».

Существует несколько **уровней оптимизации**:

- На самом абстрактном уровне можно **оптимизировать программу, выбрав для нее лучший алгоритм**. Этот метод не зависит от компилятора и языка программирования.
- На следующем уровне — **ручная оптимизация кода на основе используемого языка высокого уровня (HLL)**, при этом оптимизация не должна зависеть от конкретной реализации этого языка. Такие оптимизации должны быть применимы в разных компиляторах для одного и того же языка.
- Еще на один уровень ниже — **структурирование кода 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.

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 заранее вычисленным значением.
- **Распространение констант** – замена переменных постоянными значениями, если компилятор определяет, что программа присвоила эту константу переменной ранее в коде. Манипулирование непосредственными константами часто более эффективно, чем манипулирование переменными, что приводит к лучшему коду. В некоторых случаях это также позволяет компилятору полностью исключить определенные переменные и инструкции.
- **Удаление мёртвого кода** – 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** – 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.

- **Снижение стоимости операций** – замена медленных операций (например, умножения и деления) на более быстрые (такие как сложение, вычитание, сдвиг).
- **Анализ индуктивных переменных** – во многих выражениях, особенно в цикле, значение одной переменной полностью зависит от другой. Компилятор может исключить вычисление нового значения или объединить два вычисления в одно.
- **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.
- **Раскрутка циклов** – техника, искусственно увеличивающая количество инструкций, исполняемых за одну итерацию цикла, что позволяет увеличить параллелизм и интенсивнее использовать регистры и кэш процессора.

Вычисления по короткой схеме – 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` (||).

Примерная иерархия скорости выполнения операторов процессором: Невозможно создать универсальную таблицу скоростей операций, так как производительность зависит от конкретного процессора. Однако **некоторые арифметические операции выполняются медленнее, чем другие**; например, сложение целых чисел часто выполняется намного быстрее, чем умножение целых чисел. Операции с целыми числами обычно выполняются намного быстрее, чем соответствующие операции с плавающей запятой.

Безопасные оптимизации: Компиляторы применяют только **безопасные оптимизации**, что означает, что компилятор может изменять программу только так, чтобы это **не изменило ее поведение для всех входных данных**.

Блокировщики оптимизации: Это определенные характеристики кода, которые **не позволят компилятору совершить оптимизацию**. Существуют два основных типа блокировщиков оптимизации:

- **Указатели**: Компилятор не может точно знать, будут ли два указателя указывать на одну и ту же область памяти. Например, если `xp` и `yp` указывают на одну и ту же ячейку памяти, функции, семантически выглядящие одинаково, могут иметь разное поведение. По этой причине компилятор не трансформирует менее эффективную функцию в более эффективную, если есть риск изменения поведения.
- **Вызовы функций**: Вызовы функций влекут накладные расходы, и компилятору тяжело определить, имеет ли вызов функции побочные эффекты (например, изменение глобального состояния). Когда он не может этого сделать, он рассчитывает на худшее и не выполняет оптимизацию.

Локальность данных: Это важное свойство данных, означающее, что **при работе с данными желательно, чтобы они находились в памяти рядом**. Программы с хорошей локальностью обычно выполняются быстрее, чем программы с плохой локальностью.

Виды локальности:

- **Временна́я локальность**: Когда мы обращаемся к одному и тому же месту в памяти много раз. Если к каким-то данным обращаться часто, они имеют больше шансов оставаться в кэше процессора.
- **Пространственная локальность**: Когда мы обращаемся к данным, а потом обращаемся к другим данным, которые расположены в памяти рядом с первоначальными. При перемещении данных между уровнями иерархии памяти (например, в кэш), они перемещаются блоками, поэтому если вы читаете какие-то байты, а потом байты рядом с ними, они, вероятно, уже будут в кэше.

39. Виртуализация (общие понятия)

Виртуализация — это концепция, при которой некоторая программа, называемая «монитор виртуальных машин», **создаёт иллюзию присутствия нескольких (виртуальных) машин на одном и том же физическом оборудовании**. Это основа современных технологий, позволяющая, например, облачным сервисам обеспечивать изолированные среды для миллионов пользователей. Для системных программистов виртуализация важна тем, что она позволяет **управлять ресурсами, оптимизировать их использование и повышать безопасность**.

Понятие монитора виртуальных машин: Монитор виртуальных машин большинству разработчиков известен под названием **гипервизор (hypervisor)**. Он создает иллюзию нескольких виртуальных машин на одном физическом оборудовании.

Пример «виртуализации» ресурсов в рамках ОС: Нечто подобное виртуализации уже знакомо вам из работы операционной системы:

- **Процессы**
- **Виртуальная память**
- **Файлы** По сути, эти объекты/механизмы операционной системы позволяют программе использовать физические ресурсы компьютера и «верить» в то, что они принадлежат только ей. Операционная система в данном случае является своего рода «гипервизором», задача которого — управлять доступом к ресурсам всех «виртуальных машин»-процессов.

Что такое виртуальная машина? Под **виртуальной машиной (ВМ)** понимается **некоторая изолированная среда, имитирующая физический компьютер**.

Что такое гостевая и хост системы? Система/компьютер, в рамках которой установлен гипервизор, называется **хост-системой**. Операционная система, функционирующая внутри виртуальной машины, называется **гостевой операционной системой**.

В каких направлениях должен развиваться гипервизор? Гипервизоры должны хорошо проявлять себя в следующих направлениях:

- **Безопасность**: Гипервизор должен иметь 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.

Какие бывают гипервизоры и как они работают? Существуют два основных типа гипервизоров:

- **Гипервизоры первого типа (bare-metal)**: Технически похожи на операционную систему, поскольку это единственная программа, запущенная в самом привилегированном режиме. Их работа заключается в поддержке нескольких копий оборудования (виртуальных машин). Такой гипервизор работает непосредственно с аппаратным обеспечением. Примеры: VMware ESXi, Microsoft Hyper-V, KVM, Xen (в режиме HVM).
- **Гипервизоры второго типа (hosted)**: Это программы, которые при распределении и планировании использования ресурсов опираются на существующую операционную систему (например, Windows или Linux) и очень похожи на обычный процесс. Многие функциональные возможности гипервизоров второго типа зависят от основной операционной системы. Примеры: VMware Workstation, Oracle VirtualBox, Parallels Desktop.

Что является центральной концепцией в любом виде виртуализации? Центральной концепцией, вокруг которой построены все типы виртуализации, является **абстрагирование физических ресурсов**.

Какие виды виртуализации вы знаете? На глобальном уровне можно выделить три основных типа виртуализации:

- Клиентская виртуализация
- Серверная виртуализация
- Виртуализация хранилищ

40. Виртуализация (виды и хранилища)

Виртуализация — это концепция создания иллюзии нескольких виртуальных машин на одном физическом оборудовании, управляемой гипервизором. Её цель — оптимизация ресурсов, изоляция сред и повышение безопасности.

На глобальном уровне выделяют три основных вида виртуализации:

- **Клиентская виртуализация**
- **Серверная виртуализация**
- **Виртуализация хранилищ**

Клиентская виртуализация относится к возможностям виртуализации, размещенным на стороне клиента, и направлена на решение проблем обслуживания множества персональных устройств. Она делится на следующие типы:

- **Упаковка приложений (Application Packaging)**: Изолирует приложение от операционной системы, на которой оно будет выполняться, предотвращая изменение критически важных ресурсов ОС. Это снижает вероятность компрометации ОС вредоносными программами.
- **Потоковая передача приложений (Application Streaming)**: Хранит приложения на серверах в центре обработки данных, и они загружаются на лету на компьютер конечного пользователя при необходимости, без полной установки. Это позволяет избежать полной установки и обеспечивает автоматическое обновление.

- **Эмуляция аппаратного обеспечения (Полная виртуализация)**: Программное обеспечение для виртуализации загружается на клиентский компьютер, на котором уже установлена базовая ОС, и создает контейнер для гостевой ОС — виртуальную машину. Внутри ВМ можно установить любую операционную систему, так как гостевая ОС не знает о существовании гипервизора.

Серверная виртуализация относится к проблемам эффективного использования серверов в центрах обработки данных. Существуют три основных типа серверной виртуализации:

- **Виртуализация уровня ОС (Контейнеризация)**: Технология, которая позволяет запускать изолированные контейнеры («песочницы») внутри одной операционной системы.
    - **Принципы контейнеризации**:
        - **Общее ядро**: Все контейнеры делят ядро основной системы, но работают как независимые среды.
        - **Изоляция**: Контейнеры изолированы друг от друга (нет доступа к файлам/процессам соседей), но не требуют эмуляции аппаратуры.
        - **Легковесность**: Не нужно запускать полноценную ОС для каждого контейнера, что приводит к меньшему потреблению ресурсов и более быстрому запуску.
    - Примеры: Docker, OpenVZ, Virtuozzo.
- **Полная виртуализация**: 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 полностью изолированной среде. Примеры: Hyper-V, VirtualBox.
- **Паравиртуализация**: Форма виртуализации, при которой **явно раскрывается факт наличия виртуальной среды**. Она требует **модификации гостевой ОС** для взаимодействия с предоставляемым интерфейсом для корректной работы и лучшей производительности. Однако разработчики CPU внедрили в свои продукты дополнительные наборы команд (AMD-V и VT-x), которые позволяют запускать не модифицированные экземпляры ОС с помощью паравиртуализации. Примеры: Xen.

Сравнение типов виртуализации:

Критерий	**Полная виртуализация**	**Паравиртуализация**	**Контейнеризация**
Поддержка гостевых ОС	**Высокая (любые ОС)**	Ограниченная (требуется модификация гостевой ОС)	Низкая (работает только с ОС, совместимым с ядром хост-системы)
Производительность	Ниже (из-за аппаратной эмуляции)	Выше (прямое взаимодействие с гипервизором)	**Максимальная (не требует эмуляции)**
Изоляция	**Полная (аппаратная)**	Частичная (программная, гостевая ОС «знает» о виртуализации)	Слабая (все контейнеры делят ядро хост-ОС)

Управление	Сложное (требует настройки гипервизора и управления множеством ВМ)	Среднее (требует поддержки гостевой ОС и гипервизора)	**Простое**
Масштабируемость	Средняя (ограниченная количеством ресурсов)	Высокая (меньше накладных расходов, чем у полной виртуализации)	**Максимальная (тысячи контейнеров могут работать на одном сервере)**
Сценарии использования	Запуск разных ОС на одном сервере	Высокопроизводительные вычисления, Серверные (в оригинале не указано, но логично из контекста)	Микросервисные архитектуры

Виртуализация хранилищ данных: Это технология, которая **объединяет разные физические устройства хранения данных в единое логическое хранилище**, что позволяет управлять ими централизованно. Пользователи и приложения работают с данными как с одним ресурсом, не зная о физической структуре. Виртуализация хранилищ состоит из трех уровней:

1. Физические устройства.
2. Слой виртуализации – управляет распределением данных.
3. Виртуальные диски.