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

**Программа** – данные, предназначенные для управления конкретными компонентами системы обработки информации (СОИ) в целях реализации определенного алгоритма.

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

Необходимые **свойства ПО**: необходимость документирования, эффективность, надежность, возможность сопровождения.

**Системная программа** – программа, предназначенная для поддержания работы СОИ или повышения эффективности её использования. Например: ОС, файловые системы, драйверы, утилиты, системы программирования.

**Прикладная программа** – программа, предназначенная для решения задачи или класса задач в определенной области применения СОИ. Например: текстовые редакторы, табличные редакторы, графические редакторы.

**Современное деление ПО** предусматривает следующие градации:

* Прикладное ПО – совокупность программ, осуществляющих управление программными ресурсами, порожденными программами и ориентированными на решение широкого спектра задач (СУБД, модули управления языком интерфейса ИС, программы сбора и предварительной обработки информации);
* Промежуточное (связующее) ПО – комплекс технологического ПО для обеспечения взаимодействия между различными приложениями, системами, компонентами (веб-сервер, сервер приложений, сервисная шина, система управления контентом);
* Системное ПО.

К **функциям системного ПО** принято относить:

1. Создание операционной среды функционирования для программ
2. Автоматизация разработки новых программ
3. Обеспечение надежной и эффективной работы компьютера и компьютерной сети
4. Проведение диагностики и профилактики аппаратуры компьютера и компьютерных сетей
5. Выполнение вспомогательных технологических процессов (копирование, архивирование, восстановление после сбоев и т.д.)

Существуют следующие **группы системного ПО**:

* Операционные системы
* Интерфейсные оболочки (ОС)
* Системы управления файлами
* Системы программирования
* Утилиты
* Драйверы
* Средства сетевого доступа

**Классификация системного ПО:**

1. Управляющее ПО – системные программы, реализующие набор функций, который включает в себя управление ресурсами и взаимодействие с внешней средой СОИ, восстановление работы системы после проявления неисправностей в технических средствах.
2. Обслуживающее ПО (утилиты) – программы, предназначенные для оказания услуг общего характера пользователям и обслуживающему персоналу СОИ.
3. Базовое системное ПО – минимальный набор программных средств, обеспечивающий работу компьютера и компьютерной сети.
4. Сервисное системное ПО – программы и программные комплексы, которые расширяют возможности базового ПО и организуют удобную среду для работы других программ и пользователя.

## **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 системного ПО, программ сложной структуры.

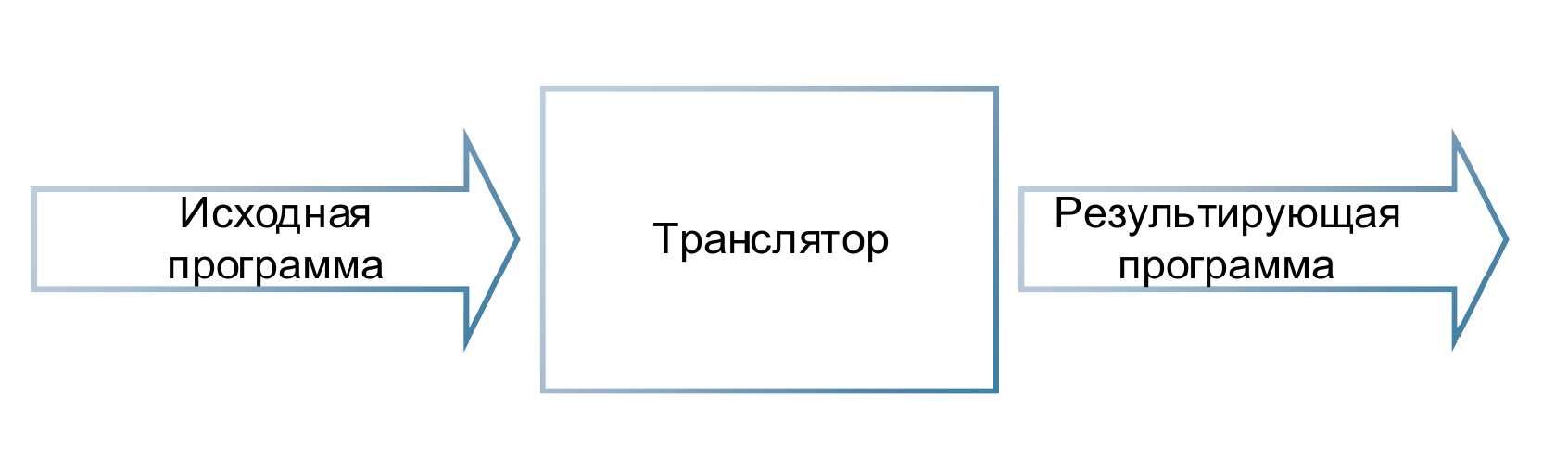
Разработка системного ПО предполагает знание и использование ассемблера для создания модулей и ассемблерных вставок.

**Система программирования** – набор специализированных программ, которые выступают инструментальными средствами разработчика для полной поддержки процессов совместной разработки, доступа к коду, проектирования, разработки, отладки и тестирования программ, их развертывания.

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.



**Программный модуль** - программа или функционально завершенный фрагмент программы, предназначенный для хранения, трансляции, объединения с другими программными модулями и загрузки в оперативную память.

**Требования к программным модулям:** функциональность, несвязность, специфицируемость.

**Этапы подготовки программы:**

**1. Редактор текста.** Исходный модуль – программный модуль на исходном языке, обрабатываемый транслятором и представляемый для него как целое, достаточное для проведения трансляции.

**2. Компиляция.** Результатом работы является объектный модуль – программный модуль, получаемый в результате трансляции исходного модуля. Содержимое объектного модуля не содержит признаков на каком языке был написан исходный модуль

1) Предварительная обработка кода: присоединение исходных файлов, работа макропроцессоров

2) Анализ: лексический, синтаксический, семантический

3) Синтез: распределение памяти, генерация и оптимизация машинно-независимого кода

**3. Компоновка.** Поскольку транслятор обрабатывает только один конкретный модуль, он не может должным образом обработать те части этого модуля, в которых запрограммированы обращения к данным или процедурам, определенным в другом модуле. Компоновщик (редактор связей) соединяет вместе все объектные модули, входящие в программу. Загрузочный модуль – программный модуль, представленный в форме, пригодной для загрузки в оперативную память для выполнения.

## **3. Введение. Что такое операционная система (далее – ОС)? Для чего предназначена ОС? Что такое POSIX? Какие абстракции над железом даёт ОС разработчику ПО? Что такое ядро ОС? Каковы задачи ядра ОС? Что такое пользовательский режим и режим ядра?**

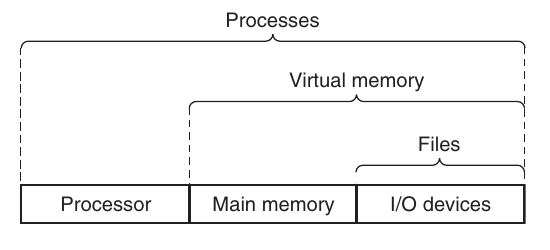
**ОС** – комплекс системного программного обеспечения, который предоставляет полезные абстракции базовых аппаратных средств.

**Типичные аппаратные средства, для которых ОС предоставляет абстракции**: процессор, RAM (она же первичная или физическая память), диски (какой-либо вид вторичной памяти), сетевые интерфейсы, дисплеи, клавиатуры, мыши.

ОС имеет **два основных предназначения**:

* Защита аппаратного обеспечения от неправильного использования неконтролируемыми приложениями
* Предоставление приложениям простого и единообразного механизма для управления сложными и зачастую широко разнообразными низкоуровневыми аппаратными устройствами

Достижение данных двух целей достигается путём введения **следующих абстракций**:



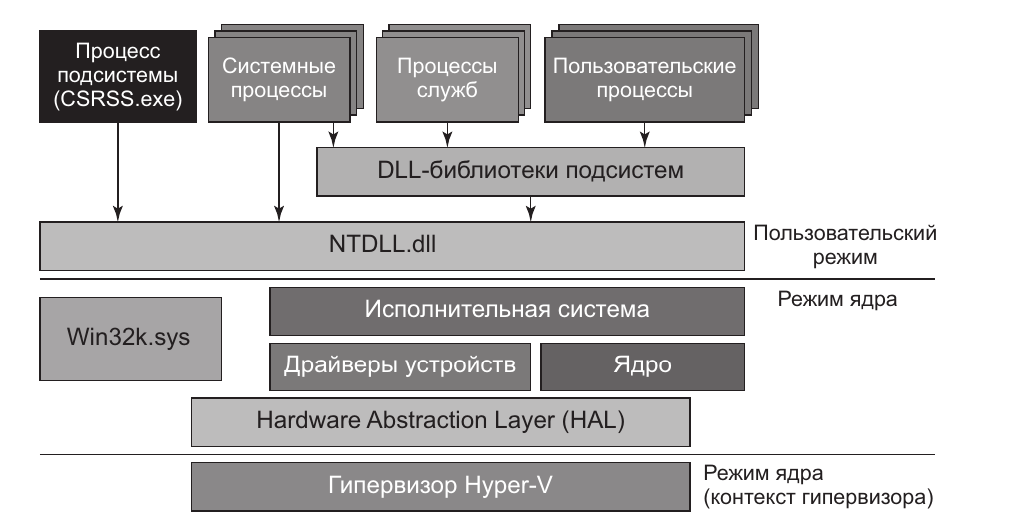
API ОС Windows **(WinAPI)** предоставляет следующий функционал: UI, среда Windows (Shell), пользовательский ввод и сообщения, доступ к данным и хранение, диагностика, графика и мультимедиа, взаимодействие с устройствами, системные сервисы, безопасность и идентификация, установка и обслуживание приложений, администрирование и управление системой, сетевые и интернет-функции.

В Linux системах как такового не существует полностью единого API ОС, однако выделяют некоторый базовый набор функций, который предназначен для работы с ресурсами системы. Данный набор функций основан на стандартах POSIX. **POSIX** – набор стандартов, описывающих интерфейсы между ОС и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов.

**Ядро ОС** состоит из набора функций, предоставляющих фундаментальные механизмы системы. Сюда относятся сервисные функции планирования потоков и синхронизации, используемые исполнительными компонентами, и низкоуровневая поддержка, зависящая от аппаратной архитектуры, - диспетчеризация прерываний и исключений, зависящая от архитектуры процессора.

**Задачи ядра ОС**: диспетчеризация процессов, управление памятью, предоставление файловой системы, создание и завершение процессов, доступ к устройствам, работа в сети, предоставление интерфейса прикладного программирования (API) системных вызовов.

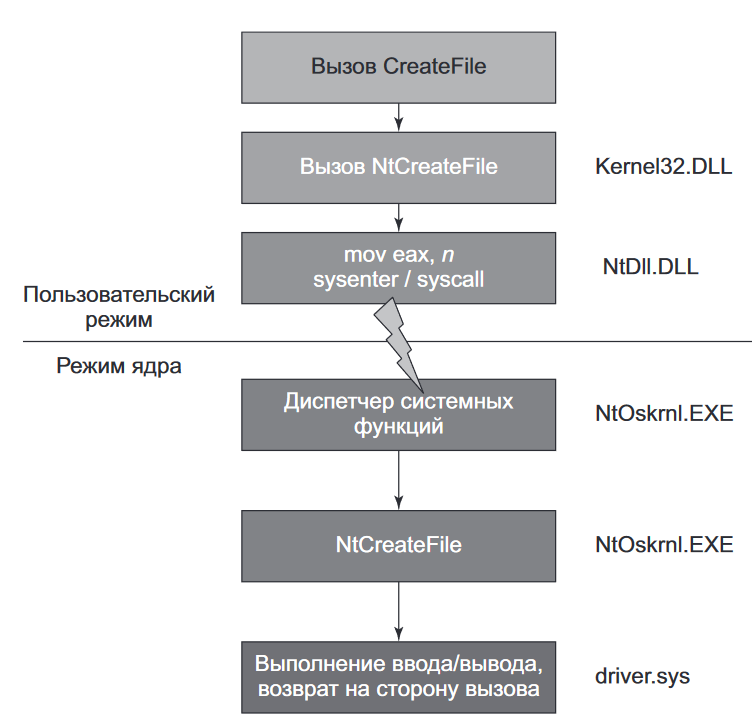
Чтобы пользовательские приложения не могли прочитать критические данные операционной системы и/или изменить их, в Windows/Linux предусмотрены **два режима доступа к процессору**: пользовательский режим (user mode) и режим ядра (kernel mode).



## **4. Введение. Что такое операционная система (далее – ОС)? Что такое системный вызов? Что такое ловушки? Что такое прерывания? Что такое объекты ядра ОС? Какие объекты вы знаете? Что такое дескриптор?**

**ОС** – комплекс системного программного обеспечения, который предоставляет полезные абстракции базовых аппаратных средств. **Ядро ОС** состоит из набора функций, предоставляющих фундаментальные механизмы системы.

**Системный вызов** – представляет собой управляемую точку входа в ядро, позволяющую процессу запрашивать у ядра осуществления некоторых действий в интересах процесса. Системный вызов – вызов функции ядра ОС прикладной программой.



**Ловушки** представляет собой неуправляемую точку входа в ядро, например запросы вызванные ошибкой деления на ноль и т.п. Обработка ловушек происходит в рамках программы, вызвавшей такое поведение.

**Прерывания** представляют собой запросы к ядру ОС от внешних аппаратных устройств. Обрабатываются независимо от каких-либо программ пользователя.

Системный вызов изменяет состояние процессора. Набор системных вызовов не изменяется. Каждый системный вызов идентифицируется по уникальному номеру. Обычно программам эта система нумерации неизвестна, они идентифицируют системные вызовы по именам. У каждого системного вызова может быть набор аргументов, определяющих информацию, которая должна быть передана из пользовательского пространства (то есть из виртуального адресного пространства процесса) в пространство ядра и наоборот.

**Объектом** в Windows называется структура данных, которая представляет системный ресурс.

**Объектами ядра являются**: маркеры доступа, файлы, проекции файлов, порты завершения ввода-вывода, задания, почтовые ящики, мьютексы, каналы, процессы, семафоры, потоки и ожидаемые таймеры.

Каждый объект ядра – на самом деле просто блок памяти, выделенный ядром и доступный только ему. Поскольку структуры объектов ядра доступны только ядру, приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое.

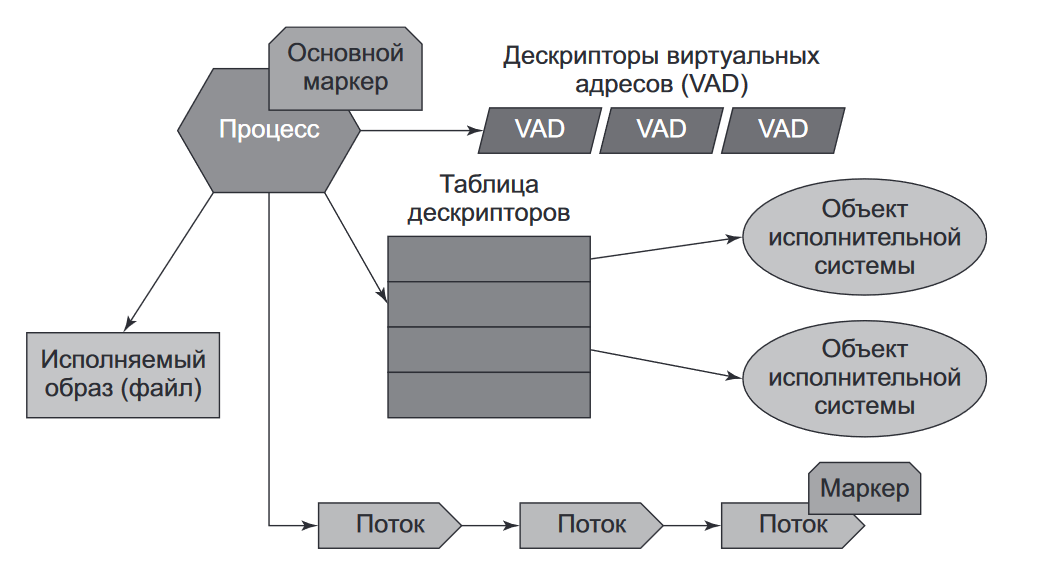
Чтобы получить доступ к объектам ядра требуется использовать функции предоставляемые API ОС. Когда вы вызываете функцию, создающую объект ядра, она возвращает описатель (HANDLE), идентифицирующий созданный объект.

**Дескриптор** — абстракция, используемая в ОС для управления доступом к объектам. Дескриптор представляет собой уникальный идентификатор, который используется приложениями и ядром ОС для взаимодействия с объектами, такими как файлы, устройства и другие ресурсы. Например, при открытии файла ОС возвращает дескриптор, который можно использовать для дальнейших операций с этим файлом (чтение, запись, закрытие и т.д.).

## **5. Процессы. Что такое процесс? Какие ресурсы процесса вам известны? Контекст процесса: определение, назначение и состав. Понятие дочернего и родительского процессов. Что такое процессная многозадачность? В каких случаях лучше всего прибегать к ней?**

**Процесс** – управляющий объект, который обеспечивает изоляцию адресных пространств и представляет работающий экземпляр программы. Процесс – исполняемое на устройстве приложение весте со всеми ресурсами, которые требуются для его исполнения.

**К типичным ресурсам процесса относятся**: образ исполняемого файла; память; список дескрипторов объектов, выделенных процессу; атрибуты безопасности (маркеры); контекст процесса.



**Контекст процесса** – минимальный набор данных, используемый процессом, который должен быть сохранен, чтобы выполнение процесса могло быть прервано и в последующем возобновлено с той же точки.

**Контекст процесса состоит из**: Program Counter (PC) или Instruction Pointer (IP), значений регистров, стека, кучи, глобальных переменных.

Каждому процессу соответствует своя структура данных в ядре ОС, которая представляет его и содержит важную информацию о состоянии, атрибутах и ресурсах процесса. Кроме ранее названных ресурсов в такой структуре также содержатся: идентификатор процесса (PID), идентификатор родительского процесса (PPID), состояние процесса, флаги процесса, информация о диспетчеризации.

За создание процесса отвечает ядро ОС. Чтобы запросить у ядра создание нового процесса необходимо совершить системный вызов: в Windows – NtCreateUserProcess, в Linux (POSIX) – fork, vfork.

**Дочерний процесс** – процесс создаваемый в результате системного вызова

**Родительский процесс** – процесс инициировавший системный вызов

**Характеристики процесса**:

* Инструмент обеспечения многозадачности и параллелизма
  + Процессная многозадачность
  + Поточная многозадачность
* Защита ресурсов необходимых отдельному приложению
* Модульность и абстракция
* Управление процессом обеспечивает ОС
* Обеспечение стабильности и надёжности работы системы

**Процессная многозадачность** — способность ОС одновременно выполнять несколько процессов. Это достигается путем многократного переключения контекста между процессами, что создает иллюзию одновременного выполнения.

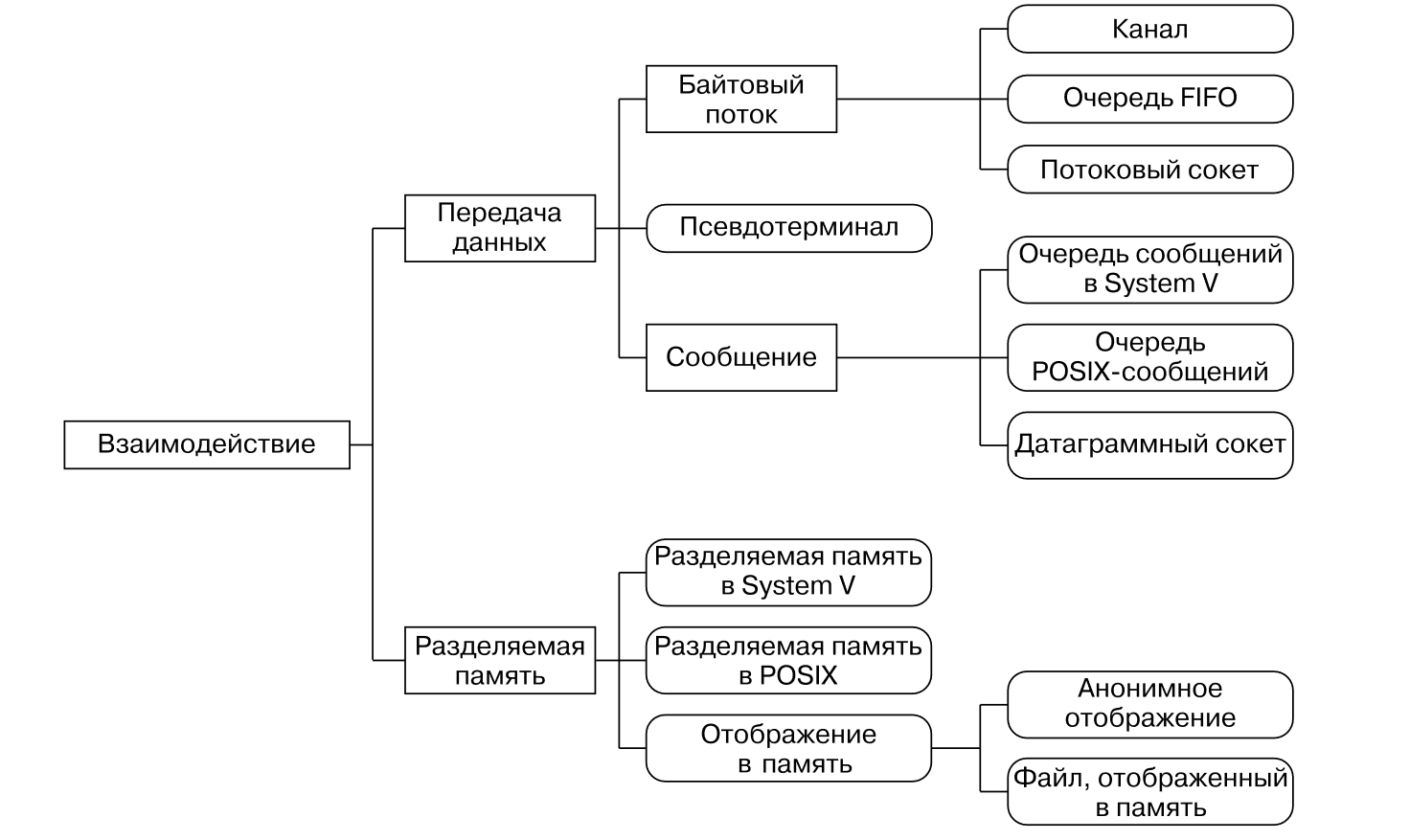
**Аргументы в пользу использования процессной многозадачности**:

* Задачи являются независимыми
* Безопасность и изоляция отдельных задач
* Поддержка языками и библиотеками
* Ресурсоёмкие задачи
* Переносимость и масштабирование
* Устойчивость к ошибкам
* Задачи, связанные с вводом/выводом и сетью
* Длительные и сложные задачи

## **6. Процессы. Межпроцессное взаимодействие (IPC): определение, классификация механизмов. Общие концепции передачи данных. В каких ситуациях предпочтительнее использовать те или иные IPC механизмы взаимодействия?**

**IPC** – механизм, позволяющий процессам обмениваться данными и синхронизировать свои действия. Это фундаментальная концепция системного программирования, поскольку она позволяет процессам работать вместе для достижения общей цели.

Категории IPC-механизмов: взаимодействие, синхронизация, сигналы.



**Общие концепции передачи данных:**

1. Процесс, который посылает данные другому потоку, называется отправителем
2. Процесс, который получает данные от другого потока, называется адресатом или получателем
3. С точки зрения направления передачи данных различают следующие виды связей:

* Симплексная – передача только, в одном направлении
* Полудуплексная – передача в обе стороны, но одновременно только в одну сторону
* Дуплексная – передача в двух направлениях одновременно

**Разделяемая память** – это тип механизма IPC, который позволяет нескольким процессам делиться общей областью памяти, обеспечивая быструю и эффективную передачу данных между процессами.

**Свойства разделяемой памяти**: дуплексный обмен данными, синхронный обмен данными, возможность моделирования любой топологии связей.

**Алгоритм работы с разделяемой памятью:**

1. Создание объекта разделяемой памяти
2. Соединение с областью памяти
3. Обмен данными
4. Закрытие объекта общей памяти

**Сравнение передачи данных и разделяемой памяти:**

* Передача данных: процессы слабосвязанные; процессы – часть децентрализованной системы; требования к масштабированию и повышенной устойчивости к сбоям; упрощенная синхронизация.
* Разделяемая память: требования к повышенной производительности и гибкости при работе с данными; обмен большими данными; частная сериализация/десериализация данных.

**Синхронизация процессов** – достижение некоторого фиксированного соотношения (порядка) между сигналами, которыми обмениваются эти процессы.

**Сигнал** – это оповещение процесса о том, что произошло некое событие. Иногда сигналы также описываются как программные прерывания. Сигналы аналогичны аппаратным прерываниям в том смысле, что они останавливают нормальное выполнение программы. В большинстве случаев невозможно предсказать, когда именно будет доставлен тот или иной сигнал.

**Предпочтительные сценарии использования механизмов взаимодействия**:

1. Каналы:

* Передача данных между связанными процессами (например, родительскими и дочерними).
* Данные передаются в одном направлении.
* Простота реализации важнее, чем производительность.

2. Байтовый поток:

* Нужно передать последовательность байтов между процессами.
* Необходима простота и легкость в использовании.

3. Потоковый сокет:

* Необходимо обеспечить связь между процессами, работающими на разных машинах.
* Требуется надежная передача данных с гарантией доставки.

4. Псевдотерминал:

* Эмуляция поведения терминала для взаимодействия с пользовательскими приложениями.
* Требуется ввод и вывод данных, как в терминале.

5. Очереди сообщений:

* Необходимо асинхронное взаимодействие между процессами.
* Важно обрабатывать сообщения в порядке их поступления или по приоритету.

6. Почтовый ящик:

* Передача сообщений от процессов-клиентов к процессам-серверам, выполняющимся на компьютерах в пределах локальной сети.

## **7. Процессы. Что такое действие? Контекст действия? Что такое атомарное действие? Что такое синхронизация процессов? Какие механизмы синхронизации вам известны? В каких ситуациях предпочтительнее использовать те или иные механизмы синхронизации?**

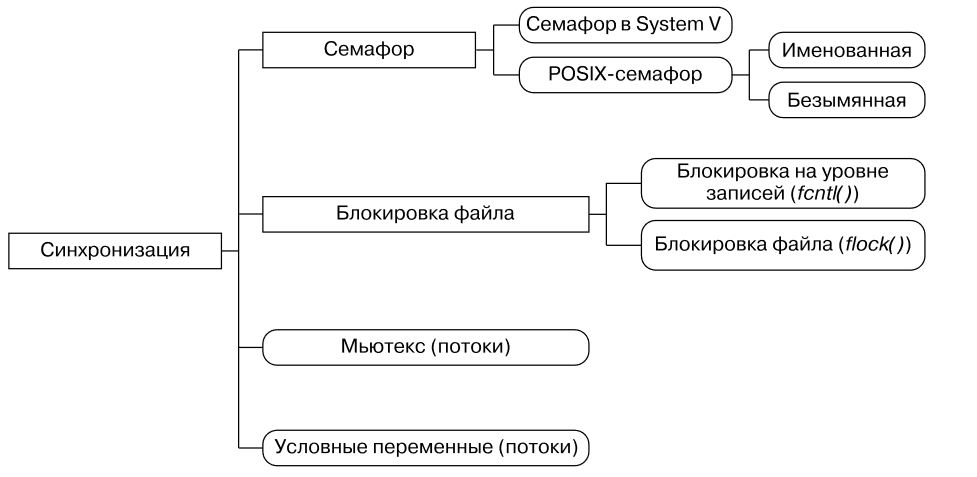
**Действием** называется изменение контекста потока или, другими словами, действием можно назвать любую последовательность команд, которая изменяет контекст потока.

Под **контекстом действия** понимается только та часть контекста потока, которая используется этим действием.

Действие называется **атомарным**, если оно удовлетворяет следующим двум требованиям:

1. Не прерывается во время своего исполнения: действие может быть прервано только сигналом прерывания, который устанавливает соответствующий флаг микропроцессор, поэтому необходимо запретить обработку сигнала прерывания от внешних устройств во время выполнения этого действия.
2. Контекст действия изменяется только самим действием: обеспечивает непрерывность действия на мультипроцессорных системах или, другими словами, запрещает действию, исполняемому одним процессором, изменять контекст действия, исполняемого другим процессором.

Если рассматривать параллельные процессы абстрактно, то **синхронизация процессов** – это есть достижение некоторого фиксированного соотношения (порядка) между сигналами, которыми обмениваются эти процессы. Под **синхронизацией потоков** понимается исполнение этими потоками условных непрерывных действий.



**Критическая секция** – это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Обеспечивает одновременный доступ к нескольким ресурсам.

**«Мьютексы»** гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Поскольку «мьютексы» являются объектами ядра, то возможна синхронизация потоков между различными процессами.

Объекты ядра «**семафоры**» гарантируют потокам взаимоисключающий доступ ко множеству ресурсов.

Блокировка файлов используется, если необходимо синхронизировать доступ к файлам, особенно в случаях, когда несколько процессов могут одновременно читать и записывать в файл, чтобы предотвратить состояние гонки при работе с файлами.

Кроме этого, существуют следующие механизмы синхронизации: события, барьеры, спинлок, RW-лок, условные переменные, атомарные операции.

## **8. Потоки. Что такое поток выполнения? Контекст потока: определение, назначение и состав. Что такое потокобезопасная функция? Какая функция называется реентерабельной? Что такое Thread Local Storage? В каких случаях лучше всего прибегать к потоковой многозадачности?**

Последовательность выполнения инструкций программы называется **потоком управления** внутри программы.

**Классификации программ** в зависимости от количества определяемых ими параллельных потоков управления:

1. Многопоточная, если в ней может одновременно существовать несколько потоков;
2. Однопоточная, если в программе одновременно может существовать только один поток.

**Поток выполнения** – структура данных, которая используется для реализации указателей на данные уровня потока (ДОП). Это наименьшая последовательность выполнения операций в программе. Потоки позволяют разделить выполнение задач на независимые части, которые могут выполняться параллельно. Каждый поток имеет свой собственный стек, регистры и контекст выполнения, но может совместно использовать память с другими потоками в пределах одного процесса.

**Контекст потока** – в общем случае, это содержимое памяти, к которой поток имеет доступ во время своего исполнения. Контекст потока позволяет системе сохранять и восстанавливать состояние потока при его приостановке и возобновлении, что критично для многозадачных систем.

Каждому потоку в Windows принадлежат **следующие ресурсы**: код исполняемой функции; набор регистров процессора; стек для работы приложения; стек для работы ОС; маркер доступа, который содержит информацию для системы безопасности.

**Потокобезопасная** функция – функция, которая может быть вызвана одновременно из нескольких потоков без риска возникновения ошибок или непредсказуемого поведения.

В общем случае функция называется повторно **входимой** или **реентерабельной** если она удовлетворяет следующим требованиям:

* не использует глобальные переменные, значения которых изменяются параллельно исполняемыми потоками
* не использует статические переменные, определенные внутри функции
* не возвращает указатель на статические данные, определенные внутри функции

**Thread Local Storage** – это механизм, который позволяет каждому потоку хранить свои собственные данные в отдельном пространстве, которое не доступно другим потокам. Это позволяет каждому потоку иметь свои собственные значения переменных, которые не будут перезаписаны другими потоками. Задействование данных, относящихся к отдельному потоку, позволяет сделать функцию потокобезопасной, не изменяя при этом ее интерфейс.

**Когда лучше использовать поточную многозадачность:**

* Ограниченность системных ресурсов
* Частое использование данных между подзадачами
* Частая синхронизация подзадач
* Поддержка языками и библиотеками
* Задачи с упором на вычисления (так называемые CPU-bound tasks)
* Написание GUI
* Задачи с требованием к масштабированию

## **9. Память. Что такое физическая, логическая и виртуальная память? Как они связаны между собой? Опишите страничную организацию памяти. Что такое рабочее множество страниц процесса? Какие виды памяти вам известны? В каких ситуациях предпочтительнее работать с виртуальной памятью напрямую?**

Интегральные схемы, предназначенные для хранения программ и данных, называются **физической памятью**. Обычно под физической памятью мы понимаем память, к которой процессор может обращаться, используя адресную шину и шину данных, а внутренняя память самого процессора представляется регистрами.

Каждый байт физической памяти имеет свой номер или индекс, который называется **физическим адресом**. При обращении к физической памяти процессор должен выставить на адресную шину физический адрес памяти, к которой он хочет получить доступ.

Под **логической памятью** процесса понимается массив байтов, к которым может обратиться процесс. Индекс каждого элемента этого массива называется **логическим адресом**. Так как логическая память процесса представляется линейным массивом байтов, то логический адрес процесса обычно называют линейным адресом.

Так как в действительности процесс может работать только с данными в физической памяти, то во время работы процесса необходимо отображать логическую память процесса в физическую память компьютера. Обычно, прямое отображение невозможно по той простой причине, что объем логической памяти процесса превышает объем физической памяти компьютера.

Для решения этой задачи физическую память компьютера дополняют памятью на дисках. Полученную расширенную память называют **виртуальной памятью**, адрес элемента этой памяти – **виртуальным адресом**.

Преобразование линейного адреса процесса в виртуальный адрес выполняется ОС посредством настройки регистров процессора. Обычно линейный адрес процесса отличается от виртуального адреса только интерпретацией бит в этом адресе. Преобразование виртуального адреса в физический адрес выполняется аппаратным образом, а именно – процессором.

Виртуальную память разбивают на блоки одинаковой длины, обычно равной 4 Кбайт, которые называют **страницами**. В этом случае файлы, в которых хранятся страницы виртуальной памяти, называются файлами страниц или файлами подкачки. При обращении процесса по адресу в виртуальной странице, если необходимо, то происходит загрузка этой страницы в реальную память компьютера и настройка адресного пространства процесса на работу с этой страницей. Эта организация вирт. памяти называется **страничной**.

Эмпирически было определено, что при работе многих программ наблюдается свойство локальности. То есть выполняемый в какой-то интервал времени код программы и используемая программой память расположены локально, а не разбросаны по всей программе.

Для эффективной работы программы необходимо, чтобы какое-то множество часто используемых на данном интервале времени виртуальных страниц находилось в реальной памяти. Это множество виртуальных страниц называется **рабочим множеством** страниц процесса.

**Виды памяти:**

1. Куча (уровень пользователя): по умолчанию + общая куча, используемая для передачи больших аргументов экземпляру сеанса Csrss.exe процесса + созданная библиотекой времени выполнения языка С.

2. Стек (уровень пользователя и ядра)

3. Свободная виртуальная память (уровень пользователя)

4. Пул памяти (уровень ядра) – выгружаемый и невыгружаемый

5. Резервные списки (уровень ядра)

**Когда работать с виртуальной памятью напрямую:**

* При разработке системного программного обеспечения
* Когда в приложении требуется максимальный контроль над выделением памяти (кучи не могут похвастаться таким)
* Когда планируется большое количество операций с большими массивами объектов или структур
* Когда планируется реализовать свой собственный алгоритм выделения памяти пользовательского уровня

## **10. Память. Перечислите и поясните назначение секций адресного пространства процесса? Куча: определение, принцип работы. Стек: определение, принцип работы. Советы при работе с кучей и стеком. В каких ситуациях предпочтительнее работать с кучами вместо виртуальной памяти?**

**Основные секции включают:**

1. Текстовая секция (Text Segment) хранит исполняемый код программы. Обычно только для чтения, чтобы предотвратить случайное изменение кода.

2. Секция данных (Data Segment) содержит глобальные и статические переменные программы, которые инициализированы значениями. Подразделы: инициализированные данные (data; переменные, которые имеют заданные значения при запуске программы) и неинициализированные данные (bss; хранит глобальные и статические переменные, которые не были инициализированы явно. Эти переменные изначально имеют значение ноль).

3. Секция стека (Stack Segment) используется для хранения локальных переменных и параметров функций, а также для управления вызовами функций (адрес возврата).

4. Секция кучи (Heap Segment) используется для динамического выделения памяти во время выполнения программы (например, с помощью malloc в C или new в C++).

5. Секция памяти, отображаемой в файл (Memory-Mapped Segment), используется для отображения файлов и других объектов в адресное пространство процесса, что позволяет работать с файлами как с массивами в памяти.

**Куча** – регион зарезервированного адресного пространства. Первоначально большей его части физическая память не передается. По мере того, как программа занимает эту область под данные, специальный диспетчер, управляющий кучами, постранично передает ей физическую память (из страничного файла). А при освобождении блоков в куче диспетчер возвращает системе соответствующие страницы физической памяти.

**Динамическая память, или куча** – область, из которой память (для переменных) может динамически выделяться в ходе выполнения программы. Верхний конец кучи называют крайней точкой программы.

Программный поток должен иметь доступ к временной области памяти для хранения параметров функций, локальных переменных и адреса возврата после вызова функции. Эта часть памяти называется **стеком**. Стек является довольно статичной частью адресного пространства, в том смысле, что практически не существует инструментов взаимодействия с ним на пользовательском уровне.

**Советы по работе со стеком и кучей:**

* Избегайте использования стека для динамических структур. Используйте кучу, а в стек записывайте указатель на структуру в куче
* Внимательно работайте с большими локальными переменными. Такие переменные быстро заполнят стек и приведут к ошибке переполнения стека
* Избегайте рекурсий с большими структурами данными (по тем же причинами, что и в прошлом пункте)
* Если требуется увеличить размер стека, делайте это рассудительно, так как выделение большего количества памяти, чем требуется может привести к неоправданному потреблению памяти.

**Когда стоит использовать кучи:**

1. Наиболее подходящие для работы с множеством малых объектов
2. Когда в приложении не требуется полный контроль над выделением памяти
3. Когда требуется точное выделение памяти (память в кучах не подвержена гранулярности, но при создании самих куч правило гранулярности работает)

**Когда стоит создавать отдельные кучи (WinAPI):**

1. Защита компонентов
2. Более эффективное управление памятью (по сравнению с использованием общей кучи)
3. Локальный доступ
4. Исключение издержек, связанных с синхронизацией потоков
5. Быстрое освобождение всей памяти в куче

## **11. Память. Виды памяти уровня пользователя. Виды памяти уровня ядра. Два системных пула памяти. Что такое тегирование пула и зачем оно необходимо? Что такое резервные списки? Общие советы при работе с памятью.**

**Виды памяти:**

1. Куча (уровень пользователя): по умолчанию + общая куча, используемая для передачи больших аргументов экземпляру сеанса Csrss.exe процесса + созданная библиотекой времени выполнения языка С.

2. Стек (уровень пользователя и ядра)

3. Свободная виртуальная память (уровень пользователя)

4. Пул памяти (уровень ядра) – выгружаемый и невыгружаемый

5. Резервные списки (уровень ядра)

На стадии инициализации системы диспетчер памяти создает два пула памяти, или кучи, с динамически изменяемым размером; они используются большинством компонентов режима ядра для выделения системной памяти.

**Невыгружаемый пул** – состоит из диапазонов системных виртуальных адресов, которые заведомо будут постоянно находиться в физической памяти. Таким образом, к этим адресам можно обратиться в любой момент без возникновения ошибки страницы.

**Выгружаемый пул** – область виртуальной памяти в системном пространстве, содержимое которой может подгружаться в систему и выгружаться из нее.

**Тегирование пулов** – это механизм, который позволяет присвоить уникальный идентификатор, называемый "тегом", пулу памяти или ресурсов ввода/вывода. Этот тег можно использовать для отслеживания и управления ресурсами, связанными с конкретным процессом, потоком или компонентом.

Тегирование пулов может помочь оптимизировать производительность системы, позволяя определить и управлять ресурсами, которые вызывают задержки или проблемы с производительностью. Тегирование пулов может помочь в устранении неполадок, предоставляя способ определить источник утечек ресурсов или других проблем.

Windows также предоставляет быстрый механизм выделения памяти – так называемые **резервные списки**. Основное различие между пулами и резервными списками заключается в том, что из пула могут выделяться блоки памяти переменного размера, тогда как резервные списки содержат только блоки фиксированного размера. И хотя обобщенные пулы памяти более гибки в отношении того, что они могут поддерживать, резервные списки работают быстрее, потому что они не используют спин-блокировки.

**Советы при работе с памятью:**

1. Когда возможно используйте стандартные функции выделения памяти (malloc, calloc, free, …), вместо работы с виртуальной памятью напрямую
2. Всегда проверяйте возвращаемое значение функций выделения памяти, чтобы убедиться, что выделение прошло успешно
3. Всегда освобождайте выделенную память, когда она больше не нужна, чтобы предотвратить утечки памяти
4. Используйте память стека (например, локальные переменные) вместо памяти кучи (например, динамически выделенной памяти), когда это возможно
5. Избегайте использования глобальных переменных, так как они могут привести к утечкам памяти и сделать код более сложным
6. Избегайте использования сырых указателей (например, int\*), когда это возможно, так как они могут привести к утечкам памяти и висячим указателям

## **12. Файлы. Хранение данных на магнитном диске (далее – МД): основные понятия. Что такое доступ к данным? Как получить доступ к данным на МД? Виды доступа к данным. Поясните понятия сектора и кластера. Из чего они состоят? Каковы их приблизительные аналоги на твердотельных накопителях? Что такое файл на физическом уровне?**

Основными элементами накопителя на жестких магнитных дисках являются круглые алюминиевые или некристаллические стекловидные пластины. Эти пластины нельзя согнуть и поэтому они называются **жесткими дисками**. Жесткие диски покрыты слоем ферромагнитного материала, который позволяет хранить информацию, используя направление магнитного поля. Жесткие диски также называются жесткими магнитными дисками.

Если магнитная головка не перемещается, то она описывает на дисковой поверхности окружность, которая называется **дорожкой**. Дорожки нумеруются от 0 до n, где дорожка с индексом 0 имеет наибольший радиус. Группа дорожек, находящихся под всеми магнитными головками в каком-то конкретном положении стержня с магнитными головками, называется **цилиндром**.

**Доступ к данным** – под ним будем понимать операции записи данных на магнитные диски и чтения данных с магнитных дисков.

Чтобы **получить доступ к данным** на магнитном диске, необходимо выполнить следующие операции:

* установить магнитные головки на соответствующий цилиндр
* дождаться, пока под магнитной головкой окажется точка на вращающемся магнитном диске, с которой начинаются данные
* прочитать или записать данные на магнитный диск во время его вращения

**Доступ к данным бывает следующих видов:**

1. Последовательный. Информация в файле обрабатывается по порядку, одна запись за другой. Этот способ доступа на сегодняшний день является наиболее распространенным; например, редакторы и компиляторы обычно получают доступ к файлам таким образом
2. Произвольный. В данном случае файл состоит из логических записей фиксированной длины, которые позволяют программам быстро считывать и записывать записи в произвольном порядке

**Сектором** называется наименьшая область (дуга) одной дорожки магнитного диска, которая может быть записана или считана магнитной головкой диска за его один полный поворот. Размер сектора обычно равен 512 байт. В начале каждого сектора хранится заголовок или префикс, который определяет начало и номер сектора. В конце каждого сектора хранится заключение или суффикс, который содержит контрольную сумму, необходимую для проверки целостности данных.

**Кластером** называется наименьшая область магнитного диска, которая может быть записана или прочитана операционной системой на диск. Обычно кластер состоит из нескольких секторов, имеющих последовательные номера.

**Соотношения понятий для HDD и SSD** (соотношения не являются прямыми в следствие разницы их устройства):

Сектор → Страница

Кластер → Блок

Дорожка → Сегмент

**Логической записью** или структурой называется упорядоченное множество данных разных типов. Порядок следования этих данных называется **структурой записи**.

На физическом уровне **файл** представляет собой поименованное множество секторов или кластеров, хранящихся на диске.

## **13. Файлы. Что такое файловая система (далее – ФС)? Что такое файл на прикладном уровне? Что такое указатель файла? Перечислите основные функции ФС над файлами. Что такое каталог? Корневой каталог? Текущий каталог? Перечислите основные функции ФС над каталогами.**

На уровне прикладной программы **файл** представляет собой множество логических записей.

Часть ОС, которая обеспечивает доступ к файлам и выполняет связывание между логическими записями файла и их физическим представлением, называется системой управления файлами или **файловой системой**.

Для того чтобы выполнять операции доступа к логическим записям файла, с каждым файлом связывают **указатель файла**, который указывает на текущую логическую запись файла. После каждой операции записи или чтения логической записи файловая система передвигает указатель на следующую логическую запись.

Для обеспечения доступа к файлам система управления файлами должна выполнять, по крайней мере, следующие функции:

* создание файла
* удаление файла
* открытие доступа к существующему файлу
* закрытие доступа к существующему файлу
* запись данных в файл
* чтение данных из файла
* установка указателя файла на нужную запись

**Каталогом** называется файл, который содержит имена и местонахождение других файлов. Каталоги имеют древовидную структуру, в которой каждая вершина указывает на каталог, а каждый лист – на файл. Каталог, который находится в вершине этого дерева называется **корневым**. Каталог, с которым в данный момент работает приложение называется **текущим**.

В Windows корневой каталог (\) определяется относительно какого-либо логического диска (D:, C:, …), в то время как в Linux работа с файловой системой ведётся от единственного корневого каталога (/).

В Windows **«абсолютный» корневой каталог** существует, однако он не доступен для использования на уровне файловой системы и прикладных программ напрямую (используется менеджером объектов ядра ОС).

Файловая система обеспечивает следующие функции для работы с каталогами:

* создание каталога
* удаление каталога
* включение подкаталога в каталог
* исключение подкаталога из каталога
* включение файла в каталог
* исключение файла из каталога

## **14. Файлы. Что такое буферы ввода-вывода? Для чего они предназначены? Что такое кэширование? Что подразумевает кэширование ввода данных?**

**Буфером ввода-вывода** называется область оперативной памяти, предназначенная для временного хранения записей файла. Обычно длина буфера выбирается кратной длине кластера.

Буферы ввода-вывода **предназначены для решения двух задач**:

* устранение несоответствия между размером логической записи файла, определяемым в приложении, и размером кластера, который записывается на диск
* снижение влияния внешних устройств на скорость работы процессора, которая значительно превышает скорость работы внешних устройств

**Кэширование** ввода данных подразумевает, что система выполняет упреждающее чтение данных с магнитного диска без ожидания следующей команды на чтение данных из приложения. Это сокращает время на чтение записей файла, если они читаются приложением последовательно.

**Кэширование** – процесс записи данных в память с более быстрым доступом.

**Принципы кэширования:**

1. Хранение копий: кэширование предполагает создание копий данных, которые часто используются, чтобы избежать повторного доступа к более медленным источникам данных (например, диску или сети).

2. Принцип локальности: кэширование основывается на принципе локальности, который гласит, что если данные были недавно использованы, то они с высокой вероятностью будут использоваться снова в ближайшее время. Это может быть как временная локальность (частое использование одних и тех же данных), так и пространственная локальность (использование данных, находящихся рядом в памяти).

**Применение кэширования ввода данных:**

1. Кэширование результатов запросов: когда пользователь запрашивает данные (например, из базы данных или веб-сервиса), результаты этого запроса могут быть сохранены в кэше. Повторные запросы на те же данные могут быть обслужены из кэша, что значительно ускоряет время ответа.

2. Кэширование файлов: когда данные загружаются из файлов (например, изображений или текстовых документов), они могут быть кэшированы в оперативной памяти. При последующих запросах к этим файлам система может быстро предоставить данные из кэша без необходимости повторного чтения с диска.

3. Кэширование пользовательского ввода: в приложениях, где пользователи вводят данные (например, формы на веб-сайтах), кэширование может помочь сохранить последние введенные значения, что позволяет быстро заполнять поля формы при повторных заходах или ошибках.

**Преимущества кэширования ввода данных:**

1. Ускорение доступа: Кэширование позволяет быстро получать доступ к часто используемым данным, что улучшает отзывчивость системы.

2. Снижение нагрузки на источники данных: Уменьшение количества обращений к медленным хранилищам данных (например, базам данных и дискам) снижает нагрузку на систему.

3. Экономия ресурсов: Кэширование позволяет более эффективно использовать ресурсы системы, так как данные могут быть извлечены быстрее и с меньшими затратами.

## **15. Файлы. Что такое отображение файла в память? Алгоритм действий при работе с файлами, отображенными в память? Как устроен данный механизм в Windows? В Linux? Когда следует использовать данный механизм? Когда не следует?**

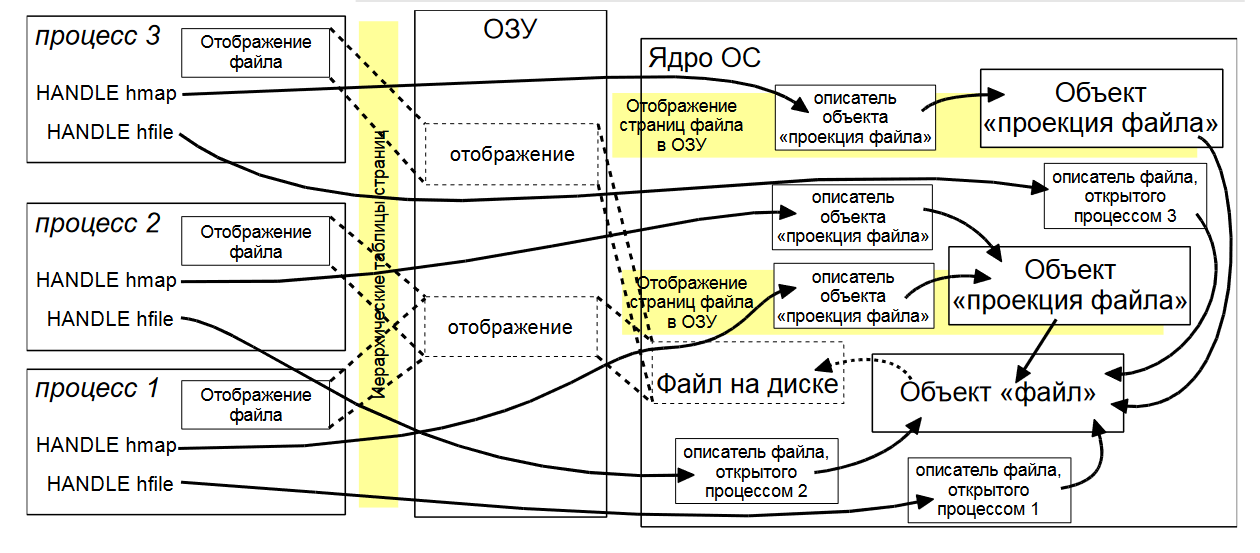
В ОС Windows и Linux реализован механизм, который позволяет отображать в адресное пространство процесса не только содержимое файлов подкачки, но и содержимое обычных файлов. То есть в этом случае файл или его часть рассматривается как набор виртуальных страниц процесса, которые имеют последовательные логические адреса. Файл, отображенный в адресное пространство процесса, называется представлением или видом файла. После отображения файла в адресное пространство процесса доступ к виду может осуществляться с помощью указателя, как к обычным данным в адресном пространстве процесса. Этот механизм называется **отображением содержимого файла** (file mapping) в виртуальную память процесса.

**Общая последовательность действий**, которые необходимо выполнить для работы с отображаемым в память файлом. Эти действия могут быть разбиты на следующие шаги:

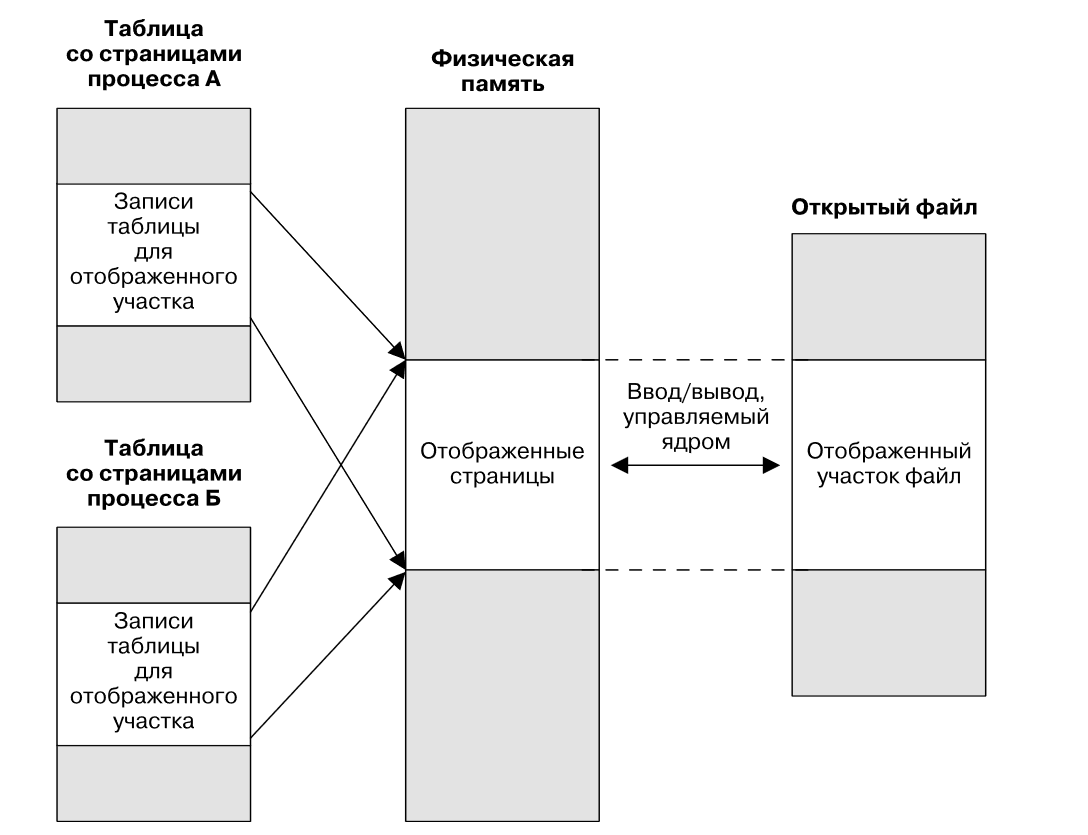
1) открыть файл, который будет отображаться в память; 2) создать объект ядра, который выполняет отображение файла; 3) отобразить файл или его часть в адресное пространство процесса; 4) выполнить необходимую работу с видом файла; 5) отменить отображение файла; 6) закрыть объект ядра для отображения файла; 7) закрыть файл, который отображался в память

В случае с Linux некоторые из этих шагов не выполняются в виду особенностей реализации механизма.

**Механизм отображения в Windows:**



**Механизм отображения в Linux:**



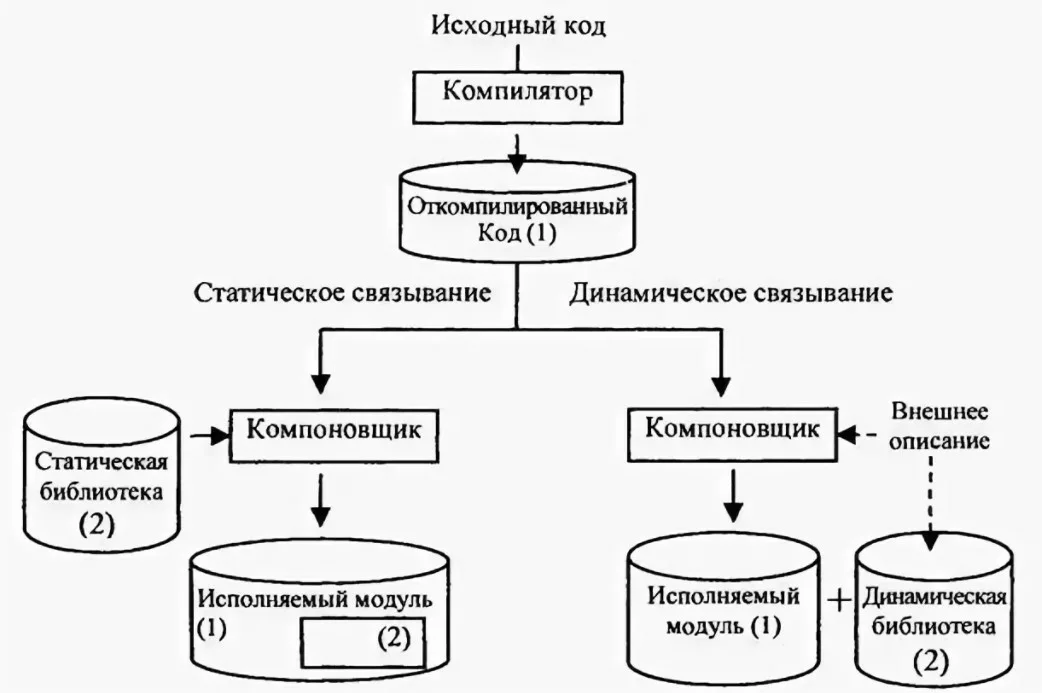
Когда использовать File Mapping: при работе с большими файлами, когда требуется произвольный доступ, когда производительность критична, когда память ограничена.

Когда не использовать File Mapping: при работе с маленькими файлами, когда требуется последовательный доступ, частые изменения файла.

## **16. 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.**

Один из способов сборки приложения заключается в компиляции его исходных файлов в объектные с последующей их компоновкой в итоговую исполняемую программу. Однако часто бывает так, что некоторые из исходных файлов можно было бы использовать в нескольких программах. Первым делом, чтобы не заниматься лишней работой, эти файлы можно скомпилировать только один раз и затем уже по необходимости компоновать их с разными исполняемыми файлами. И хотя такой подход уменьшает время компиляции, он все равно не избавляет от необходимости каждый раз указывать все объектные файлы на этапе компоновки. Более того, с увеличением количества таких файлов можно создать неразбериху в каталоге проекта.

Чтобы обойти эти проблемы, можно сгруппировать набор объектных файлов в единую сущность – **библиотеку объектов** (или объектную библиотеку). Библиотеки объектов бывают двух видов: **статические** и **разделяемые** (динамические).



**Связывание** — это процесс, при котором компилятор или линковщик объединяет объектные файлы, создаваемые компилятором, в исполняемый файл или библиотеку. Связывание может происходить на этапе компиляции или во время выполнения программы.



**Виды связывания:**

* Статическое, оно же – раннее. При нём адреса могут быть вычислены статически. Происходит на этапе компиляции, когда статические библиотеки объединяются с исполняемым файлом. Программа содержит все необходимые функции, и нет необходимости в дополнительных загрузках во время выполнения.
* Динамическое, оно же – позднее. Адреса определяются на этапе исполнения в момент связывания. Происходит во время выполнения программы, когда динамические библиотеки загружаются в память.

## **17. Библиотеки. Что такое статическая библиотека? Какое связывание лежит в основе статических библиотек? Как собрать приложение с использованием статических библиотек? Преимущества и недостатки статических библиотек.**

**Статическая библиотека** является обычным файлом, содержащим копии всех помещенных в него объектных файлов. В архиве также хранятся различные атрибуты для каждого объектного файла, включая права доступа, числовые идентификаторы пользователя и группы и время последнего изменения.

В Unix-подобных системах статическим библиотекам принято давать имена вида libname.a

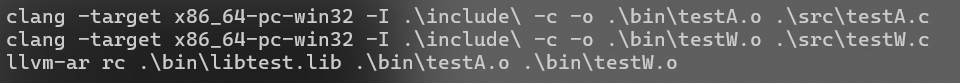
В Windows каких-либо общепринятых правил наименования статических библиотек нет

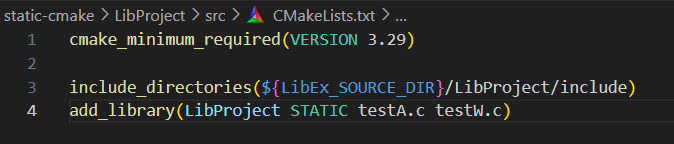
Статические библиотеки имеют следующие **положительные стороны**:

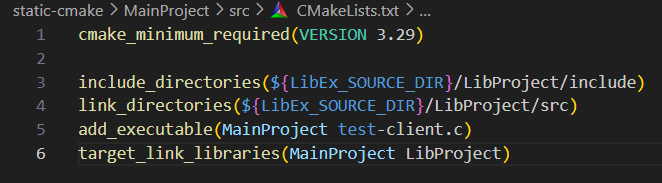
* можно поместить набор часто используемых объектных файлов в единую библиотеку, которую потом можно будет применять для сборки разных программ; при этом не нужно будет перекомпилировать оригинальные исходные тексты при компоновке каждой программы
* упрощаются команды для компоновки. Вместо перечисления длинного списка объектных файлов можно указать всего лишь имя статической библиотеки. Компоновщик знает, как выполнять поиск по ней и извлекать объекты, необходимые для создания исполняемого файла

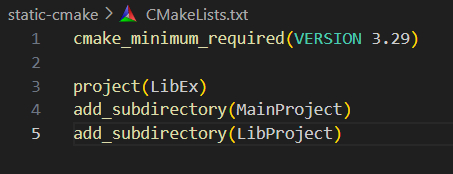
Как понятно из названия при использовании статической библиотеки речь идёт о **раннем (статическом) связывании**. В результате такого связывания весь объектный код, содержащийся в библиотеке, внедряется в будущий исполняемый файл на этапе компоновки.

**Использование библиотеки:**









Когда программа компонуется со статической библиотекой (или вовсе без использования библиотек), итоговый исполняемый файл содержит копии всех объектных модулей, скомпонованных с программой.

Таким образом, несколько разных программ могут содержать в себе копии одних и тех же объектных модулей. Подобная избыточность несет в себе **несколько недостатков**:

* дисковое пространство уходит на хранение нескольких копий одних и тех же объектных модулей. Такие потери могут быть значительными
* если несколько программ, применяющих одни и те же модули, выполняются одновременно, каждая из них будет хранить в виртуальной памяти свою отдельную копию этих модулей, увеличивая тем самым потребление виртуальной памяти в системе
* если объектный модуль статической библиотеки требует каких-либо изменений (возможно, нужно закрыть дыру в безопасности или исправить ошибку), придется заново компоновать все исполняемые файлы, в которых этот модуль используется. Данный недостаток усугубляется тем фактом, что системному администратору необходимо знать, с какими приложениями скомпонована библиотека.

## **18. 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? Преимущества и недостатки динамических библиотек.**

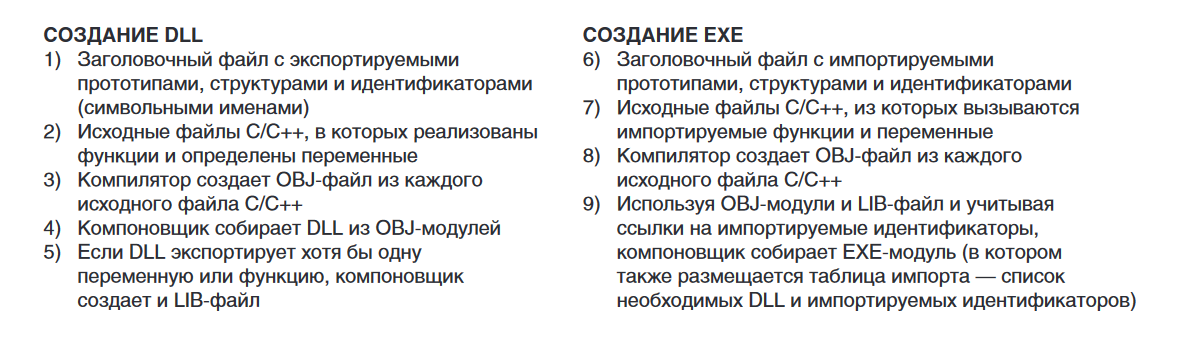
Их **ключевая идея** состоит в том, что одна копия объектного модуля разделяется между всеми программами, задействующими его. Объектные модули не копируются в компонуемый исполняемый файл; вместо этого единая копия библиотеки загружается в память при запуске первой программы, которой требуются ее объектные модули. Если позже будут запущены другие программы, использующие эту разделяемую библиотеку, они обращаются к копии, уже загруженной в память. Благодаря применению разделяемых библиотек исполняемые файлы требуют меньше места на диске и в виртуальной памяти.

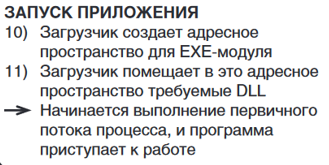
В ОС Windows подобные библиотеки называются Dynamic Link Library (DLL). В ОС семейства Linux подобные библиотеки называются разделяемые объекты (shared objects). DLL-файл представляет собой файл в формате Portable Executable. SO-файл представляет собой файл в формате Executable and Linkable Format.

Некоторые из **причин использования DLL** в ОС Windows: 1) расширение функциональности приложения; 2) возможность использования разных языков программирования; 3) более простое управление проектом; 4) экономия памяти; 5) разделение ресурсов; 6) упрощение локализации; 7) решение проблем, связанных с особенностями различных платформ; 8) реализация специфических возможностей

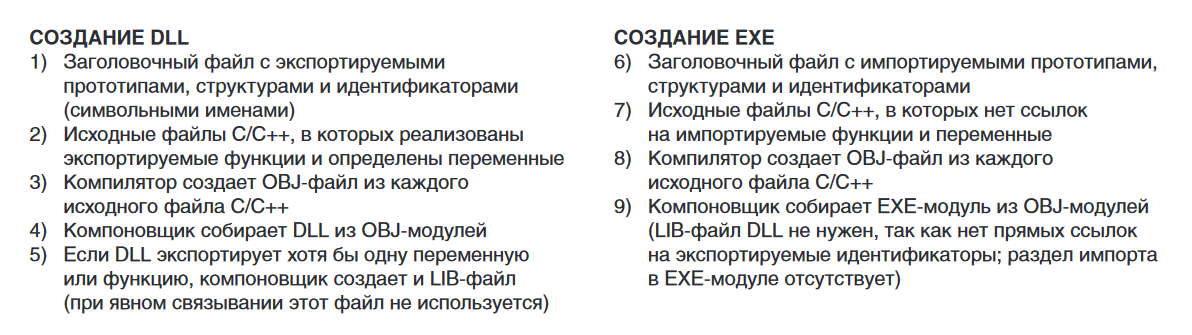
**Недостатки динамических библиотек**: 1) задержки при загрузке; зависимости от внешних файлов; 2) проблемы совместимости; 3) безопасность; 4) увеличение сложности развертывания; 5) увеличение потребления памяти.

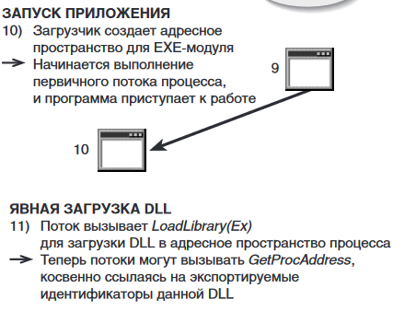
**Общая картина работы с DLL (неявный способ)**





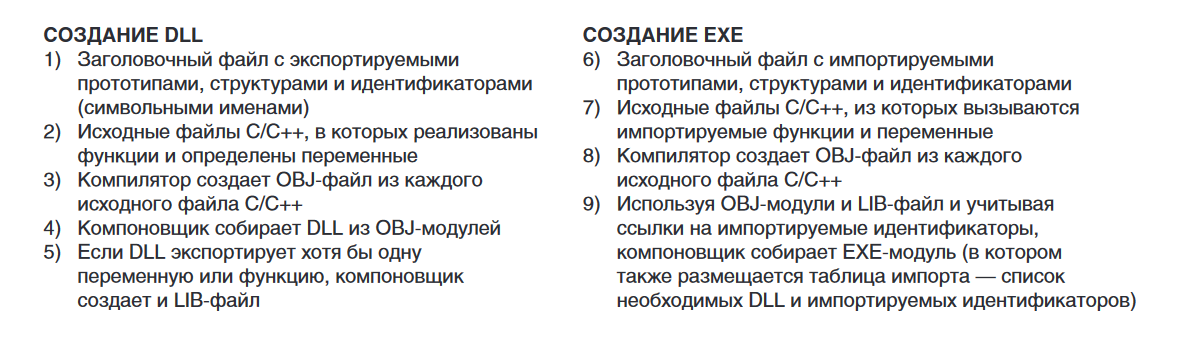
**Общая картина работы с DLL (явный способ)**

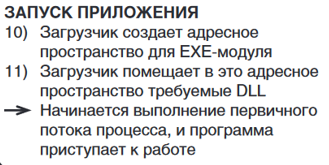




## **19. Библиотеки. Что такое неявный способ подключения разделяемой (динамической) библиотеки? Опишите алгоритм работы. Какое связывание лежит в основе неявного подключения? Что такое библиотека импорта? Что такое раздел экспорта? Какие способы экспорта функций существуют? Как их реализовать?**

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

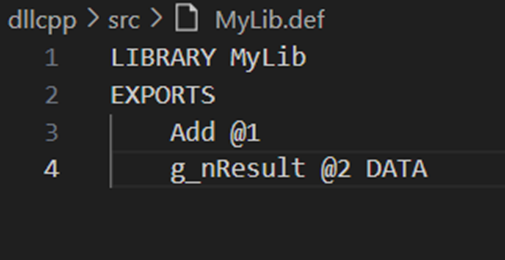




Если модификатор \_\_declspec(dllexport) указан перед переменной, прототипом функции или C++-классом, компилятор Microsoft С/С++ встраивает в конечный OBJ-файл дополнительную информацию. Она понадобится компоновщику при сборке DLL из OBJ-файлов. Обнаружив такую информацию, компоновщик создает LIB-файл со списком идентификаторов, экспортируемых из DLL. Этот LIB-файл нужен при сборке любого EXE-модуля, ссылающегося на такие идентификаторы, и называется **библиотекой импорта**.

Компоновщик также вставляет в конечный DLL-файл таблицу экспортируемых идентификаторов – **раздел экспорта**, в котором содержится список (в алфавитном порядке) идентификаторов экспортируемых функций, переменных и классов. Туда же помещается относительный виртуальный адрес каждого идентификатора внутри DLL-модуля.

Как можно заметить из таблицы экспорта, у каждой функции есть порядковый номер и имя. Соответственно существуют **два способа экспорта функций**: по номеру и по имени. Чтобы экспортировать по имени достаточно применить модификатор \_\_declspec(dllexport). Чтобы экспортировать по порядковому номеру придётся применить альтернативный способ – .DEF-файл



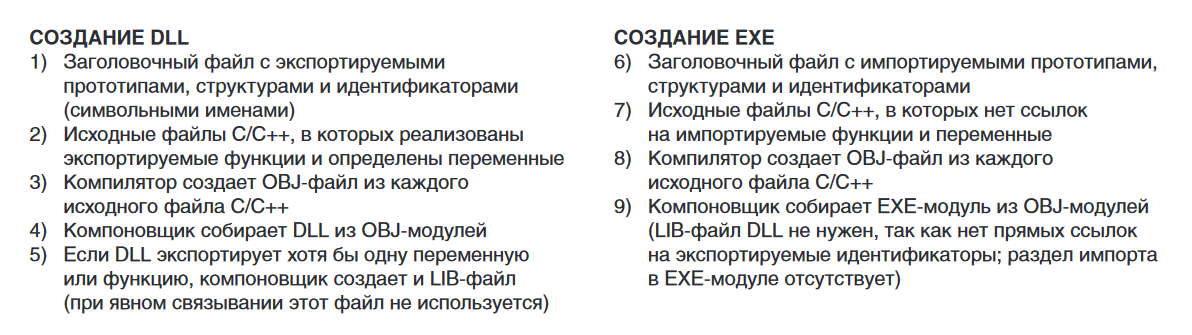
У **.def-файла** есть некоторые преимущества:

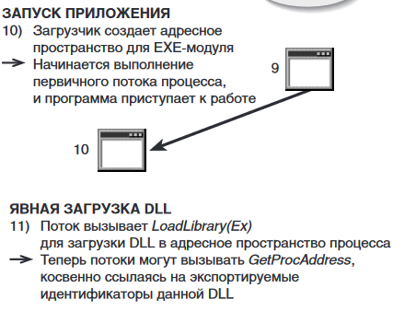
* Он позволяет использовать экспорт по порядковому номеру
* Он позволяет указать конечное имя для функции в таблице экспорта
* Позволяет настраивать область видимости экспортируемых функций

**\_\_declspec(dllexport)** – такой модификатор означает, что данная переменная или функция экспортируется из DLL (указывается в самой библиотеке).

## **20. Библиотеки. Что такое явный способ подключения разделяемой (динамической) библиотеки? Опишите алгоритм работы. Какое связывание лежит в основе явного подключения? Что такое раздел импорта? Что такое name mangling? Как его избежать?**

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





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

Разрешая ссылки на импортируемые идентификаторы, компоновщик создает в конечном EXE-модуле **раздел или таблицу импорта**. В нем перечисляются DLL, необходимые этому модулю, и идентификаторы, на которые есть ссылки из всех используемых DLL.

При создании символов для таблицы экспорта/импорта компилятор преобразует имена из кода в понятные для себя имена. Данный процесс называется **name mangling** (компилятор C такого не делает, он сохраняет имя функций)

Чтобы **избежать name mangling**, можно использовать следующие методы:

1. Использование extern "C" (в C++): оборачивайте объявления функций, которые должны быть доступны из других языков (например, C), в блок extern "C":

extern "C" {

void myFunction();

}

2. Использование C: поскольку язык C не поддерживает перегрузку функций, функции, написанные на C, не подвержены name mangling, и их имена остаются неизменными.

3. Избегайте перегрузки функций: если возможно, избегайте перегрузки функций или используйте уникальные имена для функций, чтобы минимизировать необходимость в mangling.

## **21. 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.**

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

Функцию dlopen/LoadLibrary можно вызвать несколько раз для одной и той же библиотеки. При этом загрузка будет выполнена лишь при первом вызове, а во всех последующих случаях станет возвращаться одно и то же значение handle. Однако программный интерфейс dlopen/LoadLibrary хранит счетчик ссылок для каждого дескриптора. С каждым вызовом dlopen/LoadLibrary он инкрементируется, а декрементация происходит при вызове dlclose/FreeLibrary; последний выгружает библиотеку из памяти только в том случае, если счетчик равен 0.

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

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

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

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

**Последовательность операций, которые Вам надо будет выполнить:**

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

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

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

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

Для **очистки после завершения удаленного потока** потребуется несколько дополнительных операций

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

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

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

Во время тестирования иногда может понадобиться переопределить функции (и другие символы), которые в обычных условиях были бы найдены динамическим компоновщиком. Для этого переменной среды LD\_PRELOAD можно присвоить строку с именами разделяемых библиотек, которые следует загрузить раньше других (имена разделяются двоеточиями). Поскольку данные библиотеки загружаются в первую очередь, их функции, запрашиваемые программой, будут использоваться автоматически, переопределяя любые одноименные символы, которые в противном случае пришлось бы искать динамическому компоновщику.

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

## **22. СOM. Что такое Component Object Model (далее – COM)? Два свойства лежащих в основе COM? Что такое COM-компонент? Что такое COM-интерфейс? Два типа COM-интерфейсов. Чем характеризуется COM-интерфейс? Назовите два стандартных COM-интерфейса. Что такое GUID? CLSID? IID?**

**Component Object Model** – объектная модель компоненты фирмы Microsoft является, как следует из её названия, моделью для проектирования и создания компонентных объектов. Модель определяет множество технических приемов, которые могут быть использованы разработчиком при создании независимых от языка программных модулей, в которых соблюдается определенный двоичный стандарт. Корпорация Microsoft обеспечивает реализацию модели СОМ во всех своих Windows-средах. В других операционных средах, таких как MacOS и UNIX, технология СОМ также поддерживается, но не обязательно средствами фирмы Microsoft. Данная модель была разработана как развитие техник разработки модульных приложений на языке C++.

**Свойства:**

1. *Двоичный стандарт (или независимость от языка программирования)*. Одной из наиболее важных черт СОМ является ее способность предоставлять двоичный стандарт для программных компонентов. Этот двоичный стандарт обеспечивает средства, с помощью которых объекты и компоненты, разработанные на разных языках программирования разными поставщиками и работающие в различных операционных системах, могут взаимодействовать без каких-либо изменений в двоичном (исполняемом) коде. Это является основным достижением создателей СОМ и отвечает насущным потребностям сообщества разработчиков программ.

2. *Независимость от местоположения*. Другое важное свойство СОМ известно под названием независимости от местоположения (Location Transparency). Независимость от местоположения означает, что пользователь компонента, клиент, не обязательно должен знать, где находится определенный компонент. Клиентское приложение использует одинаковые сервисы СОМ для создания экземпляра и использования компонента независимо от его фактического расположения. Компонент может находиться непосредственно в адресном пространстве задачи клиента (DLL-файл), в пространстве другой задачи на том же компьютере (ЕХЕ-файл) или на компьютере, расположенном за сотни миль (распределенный объект)

Первым основным понятием, которым оперирует стандарт СОМ, является **COM-компонент** (COM-объект), представляющий собой программный модуль. Каждый компонент имеет свой уникальный 128-битный идентификатор в формате **GUID** (Global Unique Identifier – глобальный уникальный идентификатор).

**GUID** – это всего лишь тип данных применяемый для идентификаторов COM-объектов, которые обычно принято называть **CLSID** (Class ID).

Вторым основным понятием стандарта является **COM-интерфейс**. Интерфейс представляет собой набор абстрактных функций, имеющий аналогично COM-компонентам свой GUID-идентификатор.

В данном случае такие идентификаторы принято называть **IID** (Interface ID).

**Интерфейсы бывают двух типов**: стандартные и произвольные. За стандартными интерфейсами закреплены предопределенные GUID-идентификаторы. Важнейшим среди стандартных интерфейсов является интерфейс **IUnknown**. Все остальные интерфейсы являются производными (наследуют все методы) от IUnknown. Каждый компонент должен поддерживать (часто говорят «реализовывать») как минимум стандартный интерфейс IUnknown. **IDispatch** — это интерфейс, который позволяет динамически вызывать методы объектов. Он используется в сценариях, где необходимо взаимодействие с объектами без предварительного определения всех методов на этапе компиляции. **IClassFactory** — для удобного управления жизненным циклом COM-компонентов.

## **23. СOM. Какие типы COM-контейнеров бывают? Что такое COM-сервер? Что такое COM-клиент? Назовите типы COM-серверов. Что такое «одно-компонентные» и «многокомпонентные» COM-сервера? Что должен «знать» COM-клиент, чтобы использовать COM-объект?**

**Component Object Model** – объектная модель компоненты фирмы Microsoft является, как следует из её названия, моделью для проектирования и создания компонентных объектов. Модель определяет множество технических приемов, которые могут быть использованы разработчиком при создании независимых от языка программных модулей, в которых соблюдается определенный двоичный стандарт. Корпорация Microsoft обеспечивает реализацию модели СОМ во всех своих Windows-средах. В других операционных средах, таких как MacOS и UNIX, технология СОМ также поддерживается, но не обязательно средствами фирмы Microsoft. Данная модель была разработана как развитие техник разработки модульных приложений на языке C++.

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

Приложения, использующие COM-компоненты (вызывающие функции интерфейсов, реализованных COM-компонентами), называют **COM-клиентами**, а контейнеры с расположенными в них компонентами – **COM-серверами**.

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

COM-серверы в зависимости от количества реализуемых ими компонентов **подразделяются** на «однокомпонентные» и «многокомпонентные». Соответственно, если в контейнере расположен только один компонент, то сервер - «однокомпонентный», если два и более – «многокомпонентный».

При этом COM-сервер сам может выступать в виде клиента, если он вызывает методы интерфейсов, реализованные другими компонентами.

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

При этом при работе с COM-компонентом **клиент должен «знать»** только GUID-идентификатор этого компонента (CLSID), GUID идентификаторы (IID) и структуры (сигнатуры соответствующих методов) произвольных интерфейсов компонента, которые он предполагает применять.

## **24. СOM. 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 и структуру HRESULT.**

**Component Object Model** – объектная модель компоненты фирмы Microsoft является, как следует из её названия, моделью для проектирования и создания компонентных объектов. Модель определяет множество технических приемов, которые могут быть использованы разработчиком при создании независимых от языка программных модулей, в которых соблюдается определенный двоичный стандарт.

**COM-интерфейс** представляет собой набор абстрактных функций, имеющий аналогично COM-компонентам свой GUID-идентификатор.

Важнейшим среди стандартных интерфейсов является интерфейс **IUnknown**. Все остальные интерфейсы являются производными (наследуют все методы) от IUnknown. Каждый компонент должен поддерживать (часто говорят «реализовывать») как минимум стандартный интерфейс IUnknown.

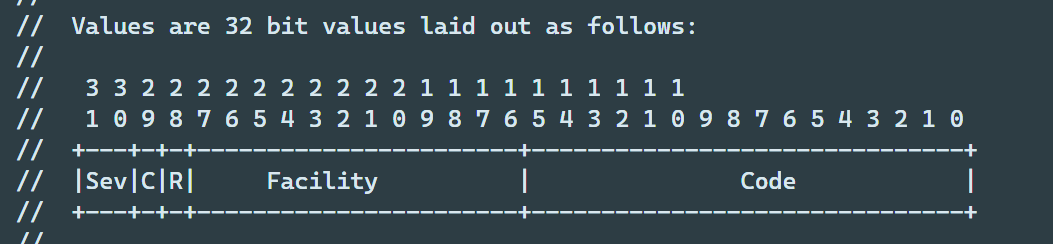
Интерфейс компонента обязательно будет содержать следующие **методы жизненного цикла**:

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

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

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

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



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

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

28 бит HRESULT является зарезервированным.

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

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

## **25. СOM. 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.**

**Component Object Model** – объектная модель компоненты фирмы Microsoft является, как следует из её названия, моделью для проектирования и создания компонентных объектов. Модель определяет множество технических приемов, которые могут быть использованы разработчиком при создании независимых от языка программных модулей, в которых соблюдается определенный двоичный стандарт.

**COM-интерфейс** представляет собой набор абстрактных функций, имеющий аналогично COM-компонентам свой GUID-идентификатор.

**IClassFactory** — для удобного управления жизненным циклом COM-компонентов. Он представляет собой так называемую **фабрику классов**, которая позволяет клиентам запрашивать новые экземпляры COM-объектов. COM требует, чтобы каждый класс имел собственную фабрику классов для создания экземпляров, но многие классы фактически могут использовать одну и ту же реализацию фабрики классов.

IClassFactory предоставляет **следующие методы**:

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

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

Фабрика классов также помогает удобно следить за жизненным циклом COM-компонент. Это является важной частью работы COM-сервера, так как при попытке освободить его ресурсы, вывод о том можно это сделать или нет, основывается на том факте используются ли хоть какие-то его COM-компоненты или нет.

Для этого на сервере существует такое понятие как **«счётчик экземпляров компонент»**. Увеличение этого счётчика происходит в конструкторе COM-компонента (он вызывается методом CreateInstance), а уменьшается в деструкторе.

**Жизненный цикл COM-сервера:**

1. Не может быть выгружен пока счётчик экземпляров компонент не равен нулю (экземпляр COM-компоненты обычно уникален в рамках одного процесса, т.е. по сути Singleton на уровне процесса)
2. Экземпляр COM-компоненты не может быть выгружен пока счётчик ссылок на интерфейсы не равен нулю
3. Не может быть выгружен пока счётчик блокировок (LockServer) не равен нулю

## **26. СOM. 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.**

**Component Object Model** – объектная модель компоненты фирмы Microsoft является, как следует из её названия, моделью для проектирования и создания компонентных объектов. Модель определяет множество технических приемов, которые могут быть использованы разработчиком при создании независимых от языка программных модулей, в которых соблюдается определенный двоичный стандарт.

Для реализации свойства «Независимости от местоположения» каждый СОМ-компонент должен быть зарегистрирован в Windows-реестре. Для регистрации компонента применяется специальная утилита **regsvr32**. Данная утилита просто вызывает некоторые экспортируемые функции из DLL.

При этом при работе с COM-компонентом клиент должен «знать» только GUID-идентификатор этого компонента (CLSID), GUID идентификаторы (IID) и структуры (сигнатуры соответствующих методов) произвольных интерфейсов компонента, которые он предполагает применять.

**Функции:**

1. DllCanUnloadNow: автоматически вызывается OLE32.DLL перед попыткой клиентом выгрузить СОМ-сервер. В зависимости от результата работы функции OLE32.DLL выгружает или не выгружает СОМ-сервер.
2. 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.
3. DllInstall: вызывается утилитой regsvr32 при наличии соответствующего параметра, применяется для выполнения дополнительных действий при регистрации и удаления регистрации компонентов.
4. DllRegisterServer: вызывается утилитой regsvr32 при наличии со ответствующего параметра, применяется для регистрации компонентов сервера в реестре ОС.
5. DllUnregisterServer: вызывается утилитой regsvr32 при наличии соответствующего параметра, применяется для удаления информации о компонентах сервера из реестра ОС.

## **27. Сервисы. Что такое сервис? Виды сервисов. Характеристики сервисов. Что такое SCM? Для чего он предназначен? Опишите структуру сервиса. Что такое функция обратного вызова? Где хранится информация о сервисах Windows?**

**Сервис или служба** – это процесс, который выполняет служебные функции. Сервисы являются аналогами резидентных программ, которые использовались в операционных системах, предшествующих операционной системе Windows NT.

То есть сервис — это такая программа, которая 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.

НО! Не каждая программа, запускаемая со стартом ОС, будет сервисом.

Обычно сервисы выполняют определенные служебные функции, необходимые для работы приложений или какого-то конкретного приложения. Примером сервиса может служить фоновый процесс, который обеспечивает доступ к базе данных – такие сервисы также называются **серверами**.

Другой тип сервисов – это программы, обеспечивающие доступ к внешним устройствам, такие сервисы называются **драйверами**.

Как сервис также может быть реализован процесс, отслеживающий работу некоторого приложения, такие сервисы также называются **мониторами**.

**Характеристики сервисов**:

1. Работают только в фоновом режиме
2. Не имеют собственного управляющего интерфейса (ни GUI, ни TUI)
3. Управляются специальной программой ОС – менеджером служб
4. Запускаются(останавливаются) со стартом (выключением) ОС, со входом (выходом) пользователя или по команде (от менеджера служб)
5. Предназначены для предоставления услуг другим программам или ОС, а не пользователям

Управляет работой сервисов специальная программа операционной системы, которая называется **менеджер сервисов (Service Control Manager, SCM)**.

**Функции, которые выполняет менеджер сервисов**:

1. Поддержка базы данных установленных сервисов
2. Запуск сервисов при загрузке операционной системы
3. Поддержка информации о состоянии работающих сервисов
4. Передача управляющих запросов работающим сервисам
5. Блокировка и разблокирование базы данных сервисов

Сам сервис может быть как консольным приложением, так и приложением с графическим интерфейсом. Это не имеет значения, т. к. сервисы могут взаимодействовать с пользователем только через рабочий стол или через окно сообщений

**Структура сервиса**:

int main(int argc, char \*argv) { … } // *главная функция консольного приложения*

VOID WINAPI ServiceMain(DWORD dwArgc, LPTSTR \*lpszArgv) { … } // *точка входа сервиса*

VOID WINAPI ServiceCtrlHandler (DWORD dwControl) { … } // *обработчик запросов*

Каждый сервис должен содержать две функции обратного вызова, которые вызываются ОС.

**Функция обратного вызова** – это функция которая передаётся другой функции в качестве аргумента.

Одна из этих функций определяет точку входа сервиса, т. е., собственно, и является сервисом, а вторая – должна реагировать на управляющие сигналы от операционной системы.

Чтобы менеджер сервисов знал о существовании определенного сервиса, его нужно установить.

**Информация обо всех установленных сервисах** хранится в реестре ОС Windows под ключом HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services

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

**Демон** – это процесс, обладающий следующими свойствами:

1. Имеет длинный жизненный цикл. Часто демоны создаются во время загрузки системы и работают до момента ее выключения
2. Выполняется в фоновом режиме и не имеет контролирующего терминала

Последняя особенность гарантирует, что ядро не сможет генерировать для такого процесса никаких сигналов, связанных с терминалом или управлением заданиями (таких как SIGINT, SIGTSTP и SIGHUP).

Названия демонов принято заканчивать буквой d (хотя это не является обязательным правилом).

Для того **чтобы стать демоном**, программа должна выполнить следующие шаги:

1. Сделать вызов **fork**, после которого родитель завершается, а потомок продолжает работать (в результате этого демон становится потомком процесса init)

2. Дочерний процесс вызывает **setsid**, чтобы начать новую сессию и разорвать любые связи с контролирующим терминалом

3. Если после этого демон больше не открывает никаких терминальных устройств, мы можем не волноваться о том, что он восстановит соединение с контролирующим терминалом. В противном случае нам необходимо сделать так, чтобы терминальное устройство не стало контролирующим

4. Очистить атрибут **umask** процесса, чтобы файлы и каталоги, созданные демоном, имели запрашиваемые права доступа

5. Поменять текущий рабочий каталог процесса (обычно на корневой – /)

6. Закрыть все открытые файловые дескрипторы, которые демон унаследовал от своего родителя (возможно, некоторые из них необходимо оставить открытыми, поэтому данный шаг является необязательным и может быть откорректирован)

7. Закрыв дескрипторы с номерами 0, 1 и 2, демон обычно перенаправляет их в предварительно открытый файл /dev/null, используя вызов dup2 (или похожий)

**Рекомендации:**

Для многих стандартных демонов предусмотрены специальные скрипты, которые выполняются, когда система завершает работу. Остальные демоны просто получают сигнал SIGTERM, который при выключении компьютера отправляется процессом init всем своим потомкам. По умолчанию этот сигнал приводит к завершению процесса. Если демону перед этим необходимо освободить какие-либо ресурсы, он должен делать это в обработчике данного сигнала.

Эту процедуру следует выполнять как можно быстрее, поскольку через 5 секунд после SIGTERM процесс init отправляет сигнал SIGKILL (это вовсе не означает, что у демона есть 5 секунд процессорного времени на освобождение ресурсов; init шлет эти сигналы всем процессам в системе одновременно, поэтому процедуру очистки в этот момент может выполнять каждый из них).

Так как демоны имеют длинный жизненный цикл, следует особенно тщательно следить не только за потенциальными утечками памяти, но и за файловыми дескрипторами (когда приложению не удается закрыть все файловые дескрипторы, которые оно открыло). Для временного исправления подобных ошибок демон приходится перезапускать заново.

Часто демону необходимо убедиться в том, что только один его экземпляр активен в любой заданный момент времени. Обычно это достигается следующим образом: демон создает файл в стандартном каталоге и применяет к нему блокировку для записи. Он удерживает ее на протяжении всего своего существования и удаляет прямо перед завершением.

Для работы с демонами в Linux также как и в Windows существует **менеджер сервисов**: init (считается устаревшим) или systemd (является более новым). Они позволяют настроить автозапуск демонов вместе с запуском ОС. В рамках подсистемы Linux для Windows используется более старый вариант – init.

**При использовании менеджера init**:

* Каталог с конфигурационными файлами сервиса должен располагаться по пути /etc/<имя демона>
* Каталог приложения расположить по пути /sbin/<имя демона>
* Файлы со значениями по умолчанию для скрипта
* Скрипт с настройками управления демоном должны располагаться в каталоге /etc/init.d/<имя демона>
* Файлы журналов должны находиться в каталоге /var/log

Первый и последний пункты не являются требованиями, а скорее общепринятыми практиками при создании системных служб.

INIT скрипт может иметь следующие состояния:

* start – запуск службы
* stop – остановка службы
* restart – перезапуск службы (остановка и запуск)
* reload – перезагрузка службы (перечитывание конфигурации без перезапуска или остановки)
* force-reload – перезагрузка конфигурации, если поддерживается, иначе – перезапуск службы
* status – покажет состояние службы

**Добавление демона на уровне системы с помощью Initd:**

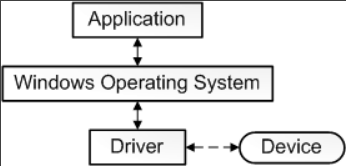
1. Компилируем приложение будущего демона
2. Располагаем получившийся бинарный файл в одной из папок /sbin или /usr/sbin
3. Располагаем скрипт запуска <имя демона> в каталоге /etc/init.d/ (меняем права через chmod на 755)
4. (Опционально) Создаём каталог /etc/<имя демона> в котором располагаем конфигурационные файлы которые будут использоваться самим демоном.
5. Проверяем с помощью команды service, что система распознала скрипт инициализации демона
6. Если скрипт написан корректно, то будет доступно управление демоном через команду service
7. Для автозапуска вместе с системой надо перейти в каталог /etc/init.d и выполнить следующую команду: sudo update-rc.d <имя демона> defaults

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

**Драйвер** – это часть кода ОС, отвечающая за взаимодействие с аппаратурой. В данном контексте слово "аппаратура" имеет самый широкий смысл. Под этим словом можно подразумевать как реальные физические устройства, так и виртуальные или логические.

**Драйвер** – это ПО, которое предоставляет другому ПО API для работы с аппаратными устройствами.

Когда приложению требуется считать данные с устройства, оно вызывает функцию, реализованную ОС. Затем ОС вызывает функцию, реализованную драйвером. Драйвер, обычно разрабатываемый производителем устройства, знает, как взаимодействовать с аппаратным обеспечением устройства для получения данных. Как только драйвер получает данные, он возвращает их обратно в ОС, которая затем возвращает их обратно в приложение.



**Список основных общих концепций драйверов** в Windows- и UNIX-системах выглядит так:

1. способ работы с драйверами как файлами (функции, используемые при взаимодействии с файлами, практически идентичны таковым при взаимодействии с драйверами)
2. драйвер, как легко заменяемая часть ОС
3. существование режима ядра

**Подсистема ввода/вывода** в Windows 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) устройств, обладающий определенными **функциональными возможностями**:

* Унифицированные средства безопасности и именования устройств для защиты общих ресурсов
* Высокопроизводительный асинхронный пакетный ввод/вывод для поддержки масштабируемых приложений
* Многоуровневая модель и расширяемость обеспечивают возможность добавлять драйверы, меняющие поведение других драйверов или устройств без необходимости модификации последних
* Динамические загрузка и выгрузка драйверов устройств позволяют выполнять данные процедуры по запросу, экономя системные ресурсы
* Поддержка технологии Plug and Play обеспечивает обнаружение и установку драйверов для нового оборудования и выделение им нужных аппаратных ресурсов, давая приложениям возможность находить и задействовать интерфейсы устройств
* Подсистема управления электропитанием позволяет системе или отдельным устройствам переходить в состояния с низким энергопотреблением

**Состав подсистемы ввода/вывода**: 1) диспетчер ввода/вывода; 2) драйвер устройства; 3) PnP-диспетчер; 4) диспетчер электропитания; 5) реестр; 6) INF-файлы; 7) уровень аппаратных абстракций (HAL).

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

**Драйвер** – это часть кода ОС, отвечающая за взаимодействие с аппаратурой. В данном контексте слово "аппаратура" имеет самый широкий смысл. Под этим словом можно подразумевать как реальные физические устройства, так и виртуальные или логические.

Центральное место в подсистеме ввод/вывода занимает **диспетчер ввода/вывода**; он соединяет приложения и системные компоненты с виртуальными, логическими и физическими устройствами, создаёт поддерживающую драйверы устройств инфраструктуру.

**Драйвер устройства**, как правило, предоставляет интерфейс ввода/вывода к конкретному типу устройства. Он представляет собой программный модуль, интерпретирующий высокоуровневые команды (такие, как команды чтения или записи) и выполняющий 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. Данный диспетчер часто используется драйверами устройств для пересылки команд ввода/вывода другим драйверам, задействованным в реализации интерфейса того же устройства и участвующим в управлении им.

**PnP-диспетчер** работает совместно с диспетчером ввода/вывода и такой разновидностью драйверов устройств, как драйвер шины. Он управляет выделением аппаратных ресурсов, а также распознает устройства и реагирует на их подключение или отключение. Именно PnP-диспетчер и драйверы шин обеспечивают загрузку соответствующего драйвера при обнаружении нового устройства. Если нужный драйвер устройства отсутствует, компоненты исполнительной системы, отвечающие за поддержку технологии PnP, вызывают сервисные функции установки устройств PnP-диспетчера в пользовательском режиме.

**Диспетчер электропитания** также тесно связан с диспетчером ввода/вывода и PnP-диспетчером. Он управляет переходами в различные состояния энергопотребления как самой системы, так и отдельных драйверов устройств.

**Реестр** представляет собой базу данных с описанием основных подключенных к подсистеме устройств, а также параметров инициализации драйверов и конфигурации.

**INF-файлы**, которые можно узнать по расширению .inf, управляют установкой драйверов. Они связывают аппаратные устройства с драйверами, управляющими этими устройствами. Содержимое такого файла состоит из инструкций (напоминающих инструкции языков сценариев), которые описывают собственно устройство, исходное и целевое положение файлов драйвера, вносимые в реестр при установке драйвера изменения и сведения о зависимостях драйвера. Удостоверяющие файлы драйверов цифровые подписи, проверенные лабораторией WHQL (Microsoft Windows Hardware Quality Lab), хранятся в файлах с расширением .cat. Цифровые подписи также применяются для предотвращения взлома драйвера или его INF-файла.

**Уровень аппаратных абстракций (HAL)** изолирует драйверы от специфических особенностей конкретных процессоров и контроллеров прерываний, поддерживая прикладные программные интерфейсы, скрывающие межплатформенные различия. В сущности, HAL является драйвером шины для устройств на материнской плате компьютера, которые не управляются другими драйверами.

## **31. Драйверы. Что такое пакет запроса на ввод/вывод (далее – IRP)? Опишите «жизненный цикл» IRP. Что такое виртуальные файлы? Что такое уровни запросов прерываний? Что такое отложенные вызовы процедур? Поясните эти две концепции на примере.**

**Драйвер** – это часть кода ОС, отвечающая за взаимодействие с аппаратурой. В данном контексте слово "аппаратура" имеет самый широкий смысл. Под этим словом можно подразумевать как реальные физические устройства, так и виртуальные или логические.

Центральным элементом подсистемы ввода/вывода является диспетчер ввода/вывода (I/O manager), задающий инфраструктуру (или модель) для доставки драйверам устройств запросов на ввод и вывод. Данная подсистема имеет пакетное управление. Большинство запросов представлены именно **пакетами запросов на ввод/вывод (IRP)**, передаваемыми от одного компонента системы к другому. Подобное проектное решение позволяет отдельному программному потоку приложения одновременно управлять целым набором запросов на ввод и вывод. Такая структура данных, как IRP-пакет, содержит информацию, полностью описывающую запрос на ввод и вывод.

Диспетчер ввода/вывода представляет операции ввода и ввода 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 с целью передачи пакета другому драйверу для дальнейшей обработки. В дополнение к созданию и уничтожению IRP-пакетов диспетчер ввода/вывода предоставляет различным драйверам общий код, который они используют при обработке ввода/вывода.

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

ОС абстрагирует все запросы ввода/вывода как операции с виртуальными файлами, так как диспетчер ввода/вывода ни с чем другим работать не умеет. При этом за преобразование файловых команд (открытие, закрытие, чтение, запись) в команды для конкретного устройства отвечает драйвер.

Две концепции ядра Windows: **уровни запросов прерываний (IRQL)** и **отложенные вызовы процедур (DPC)**.

**IRQL** – приоритет, назначаемый источнику прерываний от физического устройства. Число задается HAL (при содействии контроллера прерываний, к которому подключаются устройства, требующие обслуживания прерываний). У каждого центрального процессора имеется собственный уровень IRQL.

Предположим, что у вас есть драйвер сетевой карты, который получает данные. Когда данные приходят, драйвер может вызвать прерывание на уровне IRQL 2. Если в этот момент происходит прерывание от диска на уровне IRQL 1, оно не сможет прервать обработку сетевого прерывания, так как сетевое прерывание имеет более высокий приоритет.

**DPC** – 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» в названии означает, что DPC не выполняется немедленно – да и не может, потому что текущий уровень IRQL выше 2. Но когда ISR вернет управление, при отсутствии ожидающих обработки прерываний уровень IRQL процессора падает до 2, и он выполняет накопившиеся вызовы.

## **32. Драйверы. Что такое драйвер? Какие бывают драйверы? Что такое WDM-драйверы и какие они бывают? Что такое стек драйверов? Какие бывают многоуровневые WDM-драйверы? Последовательность вызова функционала, реализованного многоуровневым драйвером.**

**Драйвер** – это часть кода ОС, отвечающая за взаимодействие с аппаратурой. В данном контексте слово "аппаратура" имеет самый широкий смысл. Под этим словом можно подразумевать как реальные физические устройства, так и виртуальные или логические.

**Классификация типов драйверов** для ОС Windows NT:

* драйверы пользовательского режима (User-Mode Drivers):
* Драйверы, являющиеся компонентами среды UMDF
* Драйверы принтеров подсистемы Windows
* драйверы режима ядра (Kernel-Mode Drivers):
* Драйверы файловой системы (File system drivers) – принимают запросы 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 и объеди няются с диспетчером электропитания и PnP -диспетчером. В эту категорию входят драйверы запоминающих устройств, видеоадаптеров, устройств ввода и сетевых адаптеров
* Драйверы без поддержки Plug and Play (Non-Plug and Play drivers) включают в себя также расширения ядра и делают систему более функциональной. Как правило, они не интегрированы с PnP-диспетчером или с диспетчером электропитания, так как не связаны с физическими аппаратными устройствами. К этой категории относятся драйверы протоколов и сетевого API

**WDM-драйверы** являются драйверами устройств, соответствующими модели WDM (Windows Driver Model). WDM поддерживает управление электропитанием, технологию Plug and Play и инструментарий управления Windows. Драйверы данной категории делятся на три типа:

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

Стоит отметить, что драйверы бывают одно- и многоуровневыми. Если драйвер является **многоуровневым**, то обработка запросов ввода/вывода распределяется между несколькими драйверами, каждый из которых выполняет свою часть работы. Между этими драйверами можно "поставить" любое количество фильтр-драйверов. **Одноуровневый** драйвер является противоположностью многоуровневому.

**Стек драйверов** – набор драйверов, которые необходимо вызывать для получения конечного результата

Что касается многоуровневых драйверов, то кроме WDM-драйверов шины, функциональных и фильтрующих драйверов, поддержка аппаратного обеспечения может обеспечиваться еще и другими компонентами:

* Драйверы классов отвечают за обработку ввода/вывода для устройств конкретного класса
* Драйверы мини-классов реализуют обработку ввода/вывода, заданную производителем для определенного класса устройств
* Драйверы портов обрабатывают запросы на ввод и вывод в соответствии с типом порта ввода/вывода
* Драйверы мини-портов преобразуют обобщенный запрос ввода/вывода о типе порта в запрос о типе адаптера.

Запуском драйверов устройств занимается подсистема ввода/вывода. Драйверы состоят из **набора процедур**, вызываемых для обработки различных этапов запроса на ввод или вывод.

1) Процедура инициализации; 2) Процедура добавления устройства; 3) Процедуры диспетчеризации; 4) Процедура начала ввода/вывода; 5) Процедура обработки прерываний; 6) Процедура DPC.

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

**Драйвер** – это часть кода ОС, отвечающая за взаимодействие с аппаратурой. В данном контексте слово "аппаратура" имеет самый широкий смысл. Под этим словом можно подразумевать как реальные физические устройства, так и виртуальные или логические.

**Запуском драйверов** устройств занимается подсистема ввода/вывода. Драйверы состоят из набора процедур, вызываемых для обработки различных этапов запроса на ввод или вывод.

1) **Процедура инициализации** (регистрирует остальные процедуры драйвера в диспетчере ввода/вывода и при необходимости выполняет всю глобальную инициализацию драйвера); 2) **Процедура добавления устройства** (драйвер, создает объект, представляющий устройство); 3) **Процедуры диспетчеризации** (основные точки входа для драйвера устройства: открытие, закрытие, чтение, запись, операции РnР); 4) **Процедура начала ввода/вывода** (инициирует передачу данных на устройство или с него); 5) **Процедура обработки прерываний** (когда устройство приостанавливает свою работу, диспетчер прерываний ядра передает управление ей); 6) **Процедура DPC** (выполняет основную часть обработки прерывания, оставшейся после ISR-процедуры).

Многие драйверы устройств обладают **дополнительными процедурами**:

* Процедуры завершения ввода/вывода
* Процедуры отмены ввода/вывода
* Процедуры быстрой диспетчеризации
* Процедура выгрузки
* Процедура уведомления о завершении работы системы
* Процедуры регистрации ошибок

При открытии программным потоком дескриптора файлового объекта диспетчер ввода/вывода должен определить по имени этого объекта, к какому драйверу следует обратиться для обработки запроса. Более того, диспетчер ввода/вывода должен быть в состоянии найти данную информацию, когда программный поток в следующий раз воспользуется тем же самым дескриптором. Это достигается с помощью следующих объектов: объект драйвера и объект устройства.

**Объект драйвера** представляет отдельный драйвер в системе (структура DRIVER\_OBJECT). Именно он дает диспетчеру ввода/вывода адрес процедур диспетчеризации (точек входа) всех драйверов.

**Объект устройства** представляет физическое или логическое устройство в системе и описывает его характеристики (структура DEVICE\_OBJECT) – например, границы выравнивания буферов и адреса очередей входящих IRP-пакетов. Именно он является точкой назначения для всех операций ввода/вывода, так как именно с ним взаимодействует дескриптор.

**Файловый объект** – структура данных режима ядра, представляющая дескриптор устройства

Файловые объекты точно 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, кроме того, они поддерживают синхронизацию.

Операции с ресурсами совместного использования в подсистеме ввода/вывода, как и в других компонентах исполнительной подсистемы Windows, осуществляются в виде объектов

Файловые объекты обеспечивают представление ресурсов в памяти, которое напоминает интерфейс, ориентированный на ввод/вывод и реализующий чтение или запись

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

Центральным элементом подсистемы ввода/вывода является диспетчер ввода/вывода, задающий инфраструктуру (или модель) для доставки драйверам устройств запросов на ввод и вывод. Данная подсистема имеет пакетное управление. Большинство запросов представлены именно **пакетами запросов на ввод/вывод (IRP)**, передаваемыми от одного компонента системы к другому. Подобное проектное решение позволяет отдельному программному потоку приложения одновременно управлять целым набором запросов на ввод и вывод. Такая структура данных, как IRP-пакет, содержит информацию, полностью описывающую запрос на ввод и вывод.

Существуют различные типы запросов ввода/вывода. Большинство операций ввода/вывода, запрашиваемые приложениями, являются **синхронными** (по умолчанию); т. е. программный поток ждет, когда устройство выполнит операцию с данными и по завершении ввода или вывода вернет код состояния. После этого программа может продолжить работу и немедленно воспользоваться переданными ей данными. При **асинхронном** вводе/выводе приложение может выдать несколько запросов ввода/вывода и продолжить свою работу, пока устройство выполняет операции ввода/вывода. Это повышает эффективность приложения, позволяя его программному потоку решать другие задачи параллельно с операцией ввода/вывода.

Независимо от типа запроса на ввод или вывод, внутренние операции ввода/вывода, инициированные драйвером на стороне приложения, выполняются асинхронно; т. е. после выдачи запроса на ввод или вывод драйвер устройства должен как можно быстрее вернуть управление подсистеме ввода/вывода

**Технология Plug and Play** (в условном переводе – "подключи и работай") – технология, состоящая как из программной, так и из аппаратной поддержки механизма, позволяющего подключать/отключать, настраивать и т. д. применительно к системе все устройства, подключаемые к ней. В идеале весь этот процесс осуществляет только механизм Plug and Play, и какие-то действия со стороны пользователя вообще не требуются. Для каких-то устройств это так и происходит, для других – проблем, к сожалению, может быть гораздо больше. Кроме того, для успешной работы Plug and Play необходима не только поддержка этой технологии со стороны устройств, но также, конечно, со стороны драйверов и системного ПО.

**Возможности** системного ПО (вместе с драйверами), поддерживающего технологию Plug and Play:

* автоматическое распознание подключенных к системе устройств
* распределение и перераспределение ресурсов между запросившими их устройствами
* загрузка необходимых драйверов
* предоставление драйверам необходимого интерфейса для взаимодействия с технологией PnP
* реализация механизма, позволяющего драйверам и приложениям получать информацию касаемо изменений в наборе устройств, подключенных к системе, и совершить необходимые действия

**Система PnP** состоит из двух компонентов, находящихся соответственно в пользовательском режиме и режиме ядра. **Менеджер PnP режима ядра** работает с ОС и драйверами для конфигурирования, управления и обслуживания устройств. **Менеджер PnP пользовательского режима** же взаимодействует с установочными компонентами пользовательского режима для конфигурирования и установки устройств. Также, при необходимости, менеджер Plug and Play взаимодействует с приложениями.

PnP может успешно работать со **следующими типами устройств**: физические, виртуальные и логические устройства.

**Какие условия драйвер должен выполнить** для осуществления полной поддержки PnP?

* наличие функций DriverEntry, AddDevice, DispatchPnp, DispatchPower и Unload
* наличие cat-файла (файла каталога), содержащего сигнатуру WHQL
* наличие inf-файла для установки драйвера

## **35. Перехват API. Поясните как происходит выполнение кода программы в ОС. Что такое функция? Что происходит при вызове функции? Что такое стек вызовов и каков принцип его работы? Что такое стековый кадр? Что такое соглашение о вызовах? Перечислите и кратко опишите какие существуют соглашения о вызовах.**

Для компьютера любая программа является не больше, чем последовательным набором инструкций, который зачастую объединены в блоки, называемые функциями, процедурами или подпрограммами.

Программа – некоторый набор инструкций и вызовов подпрограмм в порядке необходимом для достижения некоторого результата. Если представить, что выполняемая программа является простейшей, т. е. состоит только из инструкций известных процессору, то всё происходит по очень простому сценарию, а именно:

1. Загрузка из памяти следующей инструкции
2. Декодирование полученной инструкции
3. Расчёт эффективных адресов для данных
4. Выполнение инструкции

Функции являются наиболее фундаментальной языковой возможностью для абстрагирования и повторного использования кода. Они позволяют нам ссылаться на некоторый фрагмент кода по имени.

Все, что нужно знать для использования этой функции – сколько аргументов требуется, какого типа аргументы и что возвращает функция, а также что функция выполняет.

Что происходит при вызове функции:

1. Когда выполняется вызов функции, аргументы должны быть преобразованы в значения
2. Затем поток управления переходит к телу функции, и код начинает выполняться там
3. Как только встречаем оператор return, завершаем работу с функцией и возвращаемся обратно к вызову функции

**Стек вызовов** – стек, хранящий информацию для возврата управления из подпрограмм (процедур, функций) в программу (или подпрограмму, при вложенных или рекурсивных вызовах) и/или для возврата в программу из обработчика прерывания (в том числе при переключении задач в многозадачной среде).

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 и т. д. При возврате из подпрограммы или обработчика прерывания, адрес возврата снимается со стека и управление передается на следующую инструкцию приостановленной (под-)программы.

**Стековые кадры** – это машинно-зависимые структуры данных, содержащие информацию о состоянии подпрограммы. Каждый кадр стека соответствует вызову подпрограммы, кот. еще не завершился возвратом.

**Соглашение о вызовах** определяет, как функция вызывается, как функция управляет стеком и стековым кадром, как аргументы передаются в функцию, как функция возвращает значения

**Существует множество соглашений о вызовах, таких как**:

1. stdcall. Аргументы функций передаются через стек, справа налево. Очистку стека производит вызываемая подпрограмма. Перед возвратом значений из функции вызываемая подпрограмма восстанавливает значения сегментных регистров, регистров указателя стека и стекового кадра. Сохранением-восстановлением остальных регистров занимается вызывающая программа.
2. cdecl. 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.
3. fastcall. Параметры передаются через регистры. Если для сохранения всех параметров и промежуточных результатов – регистров недостаточно, то дополнительно используется стек.
4. pascal. 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. Параметр Result создаётся неявно и является первым аргументом функции

## **36. Перехват API. Что такое перехват API-функций? Перечислите основные методы перехвата. Разделите их по критерию режима выполнения. Расскажите всё о перехвате API-вызовов путём модификации исходного кода. Расскажите всё о перехвате API-вызовов путём модификации таблиц импорта. Для чего может использоваться перехват API-функций?**

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

**Основными методами перехвата являются**:

* Подмена адреса настоящей функции (модификация IAT таблиц, модификация SSDT/IDT таблиц)
* Непосредственное изменение ф-ии (сплайсинг, перехват в режиме ядра с модификацией тела ф-ии)
* Непосредственная подмена всего компонента приложения/системы (библиотеки с целевой функцией)

Методы можно также разделить **по критерию режима выполнения**:

* Пользовательские методы (модификация IAT таблиц, сплайсинг). Их особенность в том, что невозможно что-либо изменить в поведении ядра ОС и его расширений.
* Режима ядра (модификация SSDT/IDT таблиц, перехват в режиме ядра с модификацией тела функции). Позволяет модифицировать структуры данных и код любой части ОС и приложений.

**Перехват API-вызовов путём модификации исходного кода:**

**Сплайсинг** – метод перехвата API функций путём изменения кода целевой функции. Обычно изменяются первые 5 байт функции. Вместо них вставляется переход на функцию, которую определяет программист.

Чтобы обеспечить корректность выполнения операции, приложение, которое перехватывает функцию, обязано дать возможность выполниться коду, который был изменён в результате сплайсинга. Для этого приложение сохраняет заменяемый участок памяти у себя, а после отработки функции перехвата восстанавливает изменённый участок функции и дает полностью выполниться настоящей функции.

Классический способ реализации API-хуков осуществляется с помощью трамплинов. **Трамплин** – шеллкод, который используется для изменения пути выполнения кода путем перехода на другой конкретный адрес в адресном пространстве процесса. Шеллкод трамплина вставляется в начало функции, в результате чего функция становится "подцепленной". Когда вызывается подцепленная функция, вместо нее активируется шеллкод трамплина, и поток выполнения передается и изменяется на другой адрес, что приводит к выполнению другой функции.

**Шеллкод** – небольшой фрагмент машинного кода, который обычно используется в эксплуатации уязвимостей для выполнения произвольного кода на целевой системе. **Встраиваемый хук** – это альтернативный метод выполнения API-хуков, который работает аналогично хуку на основе трамплина. Разница заключается в том, что встраиваемые хуки возвращают выполнение законной функции, позволяя нормальному выполнению продолжаться. Несмотря на то, что они сложнее в реализации и потенциально труднее в обслуживании, встраиваемые хуки более эффективны.

**Перехват API-вызовов путём модификации таблиц импорта:**

В разделе импорта содержится список DLL, необходимых модулю для нормальной работы, и все идентификаторы, которые модуль импортирует из каждой DLL. Вызывая импортируемую функцию, поток получает ее адрес фактически из раздела импорта. Поэтому, чтобы перехватить определенную функцию, надо лишь изменить ее адрес в разделе импорта.

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

**Добросовестное использование перехвата функций**: отладка и мониторинг, ПО для обеспечения безопасности, мониторинг производительности, расширение функциональных возможностей.

## **37. Перехват API. Что такое перехват API-функций? Расскажите всё о перехвате API-вызовов путём модификации системных таблиц. Расскажите всё об использовании драйверов-фильтров для перехвата. Сложности в перехвате в Windows. Для чего может использоваться перехват API-функций?**

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

**Перехват API-вызовов путём модификации системных таблиц:**

Перехват API-вызовов путем 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, и сопоставляет системные вызовы с адресами функций ядра. Эти функции вызываются приложениями и драйверами для выполнения различных операций, таких как управление процессами, файлами, памятью и т.д.

Когда системный вызов выполняется приложением пользовательского пространства, он содержит служебный индекс в качестве параметра, указывающего, какой системный вызов вызывается. Затем SSDT используется для определения адреса соответствующей функции внутри ntoskrnl.exe.

**Основные шаги для перехвата API-вызовов путем модификации SSDT:**

1. Получение адреса SSDT: адрес SSDT можно получить, используя системные структуры и функции Windows. Обычно это делается с помощью недокументированных структур и функций ядра.
2. Поиск целевого системного вызова: в SSDT каждому системному вызову соответствует индекс. Необходимо определить индекс системного вызова, который нужно перехватить.
3. Сохранение оригинального адреса: перед заменой адреса системного вызова необходимо сохранить 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 адрес системного вызова в SSDT, чтобы избежать постоянного перехвата.

**Использование драйверов-фильтров:**

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. Теперь любой запрос, поступающий к устройству под управлением перехваченного драйвера, будет вызывать диспетчерские процедуры перехватывающего драйвера. При этом не создаются дополнительные объекты устройств и не происходит никакого присоединения.

Чтобы подключить драйвер, нам нужно найти указатель на объект драйвера (DRIVER\_OBJECT), и для этого мы можем использовать недокументированную, но экспортируемую функцию, которая может найти любой объект по его имени. Подключающий драйвер после этого может заменить указатели основных функций, процедуру выгрузки, процедуру добавления устройства и т.д. При любой такой замене всегда следует сохранять предыдущие функциональные указатели для отключения при необходимости и для отправки запроса реальному драйверу.

**Добросовестное использование перехвата функций**: отладка и мониторинг, ПО для обеспечения безопасности, мониторинг производительности, расширение функциональных возможностей.

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

**Безопасное программирование** – подход к разработке ПО, который направлен на предотвращение, обнаружение и реагирование на угрозы безопасности. **Основная цель** безопасного программирования – защитить данные, системы и пользователей от несанкционированного доступа, модификации или уничтожения. В контексте системного программирования безопасное программирование включает в себя разработку и управление низкоуровневыми компонентами, такими как ОС, драйверы устройств и компиляторы, с учетом принципов безопасности.

**Уязвимость** – недостаток программы, который может быть использован для реализации угроз безопасности информации. **Недостаток программы** – любая ошибка, допущенная в ходе проектирования или реализации программы, которая в случае ее неисправления может являться причиной уязвимости программы.

Уязвимость программы может быть результатом ее разработки без учета требований по обеспечению безопасности информации или результатом наличия ошибок проектирования или реализации. Обычно уязвимость позволяет атакующему «обмануть» приложение – выполнить непредусмотренные создателем действия или заставить приложение совершить действие, на которое у того не должно быть прав.

Существует несколько популярных **классификаторов уязвимостей**: CVE (Common Vulnerabilities and Exposures), CWE (Common Weakness Enumeration), SecurityFocus BID, OSVDB (Open Sourced Vulnerability Database), Secunia, IBM ISS X-Force.

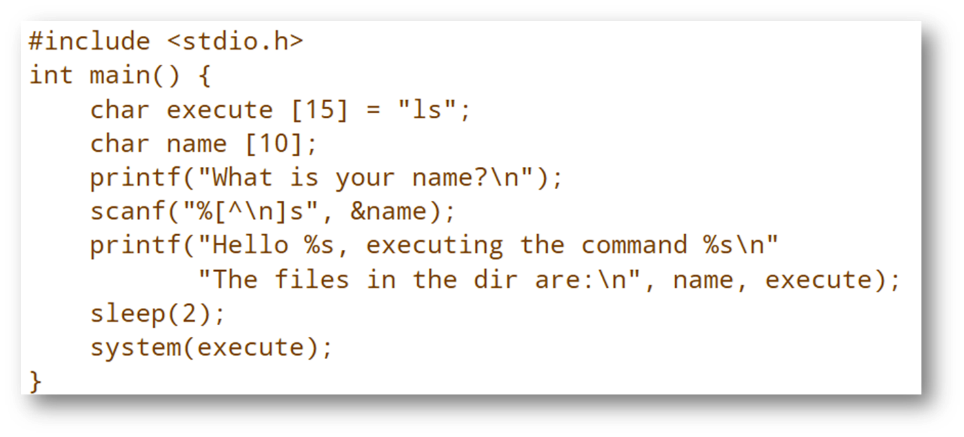
Существуют **две основные категории ошибок** ПО, которые вызваны:

1) проблемами проектирования: программист не продумал, какой тип аутентификации требуется

2) проблемы с реализацией: программист случайно ввел ошибку, используя небезопасный библиотечный метод, или попытался сохранить слишком много данных в переменной.

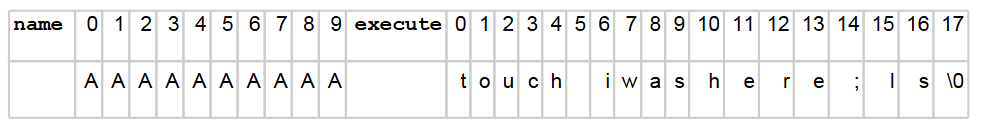
**Список распространённых ошибок**, ставящих под угрозу безопасность современных программ:

1. внедрение SQL-кода (SQL injection)
2. уязвимости, связанные с web-серверами (XSS, XSRF, расщепление HTTP запроса)
3. уязвимости web-клиентов (DOM XSS)
4. переполнение буфера (англ. Buffer Overflow)
5. дефекты форматных строк (Uncontrolled format string)
6. целочисленные переполнения (Integer overflow)
7. некорректная обработка исключений и ошибок
8. внедрение команд (Command injection)
9. утечка информации (Information Exposure)
10. ситуация гонки (Race condition)
11. слабое юзабилити (Insufficient Psychological Acceptability)
12. выполнение кода с завышенными привилегиями (Execution with Unnecessary Privileges)
13. хранение незащищенных данных (Protection Mechanism Failure)
14. проблемы мобильного кода (Mobile Code Issues)
15. слабые пароли
16. слабые случайные числа
17. неудачный выбор криптографических алгоритмов
18. использование небезопасных криптографических решений
19. незащищенный сетевой трафик (Cleartext Transmission of Sensitive Information)
20. неправильное использование PKI (Improper Certificate Validation)
21. доверие к механизму разрешения сетевых имен (Reliance on Reverse DNS Resolution)



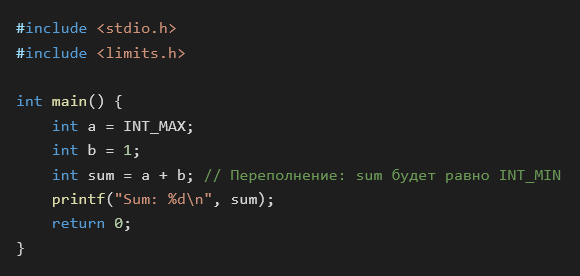
В нашем примере ввод 10 букв “A”, за которыми следует “touch iwashere;ls”, приводит к простейшей форме переполнения буфера.

**Переполнение буфера** – это когда буфер переполняется в другую память. В этом случае один буфер переполняется в соседнюю переменную. Соответственно в результате память принимает следующий вид:



Можно безопасно использовать scanf(), указав длину буфера в строке формата. Например, “%31[^\n]” будет содержать до 31 символа. Рекомендуется использовать функцию fgets(), которая принимает длину текста для чтения в качестве параметра.

**Целочисленное переполнение** – 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 программы, включая сбои, неправильные вычисления и уязвимости безопасности.



Целочисленное переполнение может привести к различным проблемам, включая как функциональные ошибки, так и серьезные уязвимости безопасности. Вот **некоторые из основных проблем**, к которым может привести целочисленное переполнение: неправильные вычисления, сбои программы, уязвимости безопасности, утечка информации, проблемы с производительностью.

**Методы предотвращения целочисленного переполнения:**

1. Использование безопасных функций
2. Проверка границ (не превышают ли результаты арифметических операций допустимые пределы типа данных)
3. Использование типов данных с большей емкостью
4. Статический и динамический анализ кода
5. Обучение разработчиков методам безопасного программирования и повышение осведомленности о рисках целочисленного переполнения

## **39. Безопасное программирование. Поясните ошибку форматирования строк и как её можно избежать. Что такое валидация и очистка? На чём основывается достижение безопасности ОС? Какие ещё есть способы повышения безопасности? Что такое ASLR? Что такое DEP? Что такое PoLP и какие принципы лежат в его основе?**

**Безопасное программирование** – подход к разработке ПО, который направлен на предотвращение, обнаружение и реагирование на угрозы безопасности.

**Ошибки форматирования строк:**

Некоторые функции, такие как printf(), получают строку формата, за которой следует несколько переменных для отображения, как указано в строке формата.

Например: printf("Hello %s, you entered %d", string, number);

Когда выполняется функция printf(), она считывает строку формата, затем просматривает в стеке данные для замены значений строки формата. “%s” заменяется строкой из стека (входные данные), а “%d” заменяется целым числом (number).

Корректным способом вывода единственного строкового значения будет: printf("%s", string);

Решение заключается в том, что все данные, поступающие из ненадежного источника, должны быть проверены (validated) и обработаны (sanitized) перед использованием. **Валидация** включает в себя проверку того, что данные представлены в том формате, который вы ожидаете. **Очистка** включает в себя удаление любого потенциально опасного форматирования/содержимого из переменной. Это может "исправить" вводимые данные, чтобы сделать их безопасными для использования.

Безопасность ОС основывается **на достижении триады CIA**: конфиденциальности (неавторизованные пользователи не могут получить доступ к данным), целостности (неавторизованные пользователи не могут изменять данные) и доступности (гарантируя, что система остается доступной для авторизованных пользователей даже в случае атаки типа "отказ в обслуживании").

Как и в случае с другими компьютерными системами, изолирование доменов безопасности – в случае ОС, ядра, процессов, и виртуальные машины – это ключ к достижению безопасности.

**Другие способы повышения безопасности** включают простоту, позволяющую свести к минимуму поверхность атаки, блокировку доступа к ресурсам по умолчанию, проверку всех запросов на авторизацию, принцип наименьших полномочий (предоставление минимальных привилегий, необходимых для выполнения задачи), chains of trust, разделение привилегий и сокращение общие данные

Безопасность ОС осложняется их растущей сложностью и, как следствие, неизбежностью появления ошибок. Поскольку формальная проверка ОС может оказаться невыполнимой, разработчики используют повышение надежности ОС для уменьшения уязвимостей, например: рандомизацию расположения адресного пространства, целостность потока управления, ограничения доступа, и другие методы.

**ASLR** («рандомизация размещения адресного пространства») – технология, применяемая в ОС, при использовании которой случайным образом изменяется расположение в адресном пространстве процесса важных структур данных, а именно образов исполняемого файла, подгружаемых библиотек, кучи и стека. Она создана для усложнения эксплуатации нескольких типов уязвимостей.

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

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

**Основные аспекты принципа наименьших полномочий**: минимизация прав доступа, разделение обязанностей, контроль доступа на основе ролей, минимизация времени доступа.

## **40. SEH. Что такое исключение? Сравните исключения и прерывания. Что такое Structured Exception Handling (далее – SEH)? Что такое блок исключения? Какие основные возможности предоставляет SEH? Что такое защищённый блок? Поясните принципы работы обработчика завершения. Что такое локальная раскрутка? Причины, по которым следует применять обработчики завершения?**

**Исключение** – это событие, возникающее из-за выполнения определенной команды, которая вызвала ошибку процессора. Скорее всего в результате такого события нормальное выполнение программы становится невозможным!

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

Примеры исключений: деление на ноль, точку останова, ошибку страницы, переполнение стека.

Если возникает исключение, ядро перехватывает его и позволяет коду обработать исключение, если это возможно. Этот механизм называется **Structured Exception Handling (SEH)** и доступен как для кода пользовательского режима, так и для кода режима ядра. SEH является частью исключительно ОС Windows! Также стоит отметить, что полная поддержка SEH присутствует только в компиляторе MSVC!

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

**Блок исключения** — это код, который содержит инструкции для обработки исключений. В SEH блоки исключений определяются с помощью директив \_\_try и \_\_except. Код в блоке \_\_try выполняется, и если возникает исключение, управление передается в блок \_\_except.

SEH предоставляет **две основные возможности**: обработку завершения и обработку исключений.

**Защищенный или охраняемый блок кода** – блок кода, ограниченный фигурными скобками оператора \_\_try. Предполагается, что в этом блоке может возникнуть исключение, которое следует обработать.

Собственно, **обработчик завершения** (\_\_finally) гарантирует, что блок кода (собственно обработчик) выполнится независимо от того, как происходит выход из другого блока кода – защищенного участка программы. Применив его, мы не допустили преждевременного выполнения оператора return.

После выполнения блока \_\_finally функция фактически завершает работу. Любой код за блоком \_\_finally не выполняется, поскольку возврат из функции происходит внутри блока \_\_try

Просматривая исходный текст, компилятор видит, что мы вставили return внутрь блока \_\_try. Тогда он генерирует код, который сохраняет возвращаемое значение в созданной им же временной переменной. Затем создает код для выполнения инструкций, содержащихся внутри блока \_\_finally, – это называется **локальной раскруткой**. Локальная раскрутка происходит, когда система выполняет блок \_\_finally из-за преждевременного выхода из блока \_\_try. Значение временной переменной, сгенерированной компилятором, возвращается из функции после выполнения инструкций в блоке \_\_finally.

**Причины, по которым следует применять обработчики завершения**: устойчивость программы, управление ресурсами, улучшение отладки.

## **41. 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?**

**Исключение** – это событие, возникающее из-за выполнения определенной команды, которая вызвала ошибку процессора. Скорее всего в результате такого события нормальное выполнение программы становится невозможным!

В хорошо написанной программе не предполагается попыток обращения по неверному адресу или деления на нуль. И все же такие ошибки случаются. За перехват попыток обращения по неверному адресу и деления на нуль отвечает центральный процессор, возбуждающий исключения в ответ на эти ошибки

Исключение, возбужденное процессором, называется аппаратным (hardware exception)

Также операционная система и прикладные программы способны возбуждать собственные исключения – программные (software exceptions)

**Защищенный или охраняемый блок кода** – блок кода, ограниченный фигурными скобками оператора \_\_try. Предполагается, что в этом блоке может возникнуть исключение, которое следует обработать.

**Блок исключения** — это код, который содержит инструкции для обработки исключений. В SEH блоки исключений определяются с помощью директив \_\_try и \_\_except. Код в блоке \_\_try выполняется, и если возникает исключение, управление передается в блок \_\_except.

За блоком \_\_try всегда должен следовать либо блок \_\_finally, либо блок \_\_except. Для данного блока \_\_try нельзя указать одновременно и блок \_\_finally, и блок \_\_except; к тому же за \_\_try не может следовать несколько блоков \_\_finally или \_\_except. Однако try-finally можно вложить в try-except, и наоборот.

В отличие от обработчиков завершения, фильтры и обработчики исключений выполняются непосредственно ОС – нагрузка на компилятор при этом минимальна.

**Принципы работы обработчика исключений**: 1) когда возникает исключение, система определяет, какой обработчик должен быть вызван; 2) управление передается в блок \_\_except, где определяется, как обрабатывать это исключение; 3) код в блоке \_\_except выполняется, и обработчик может решить, как реагировать на исключение.

**Фильтры** — это специальные функции, используемые для определения, как обрабатывать конкретные исключения. Фильтры можно использовать в блоках \_\_except для определения, будет ли исключение обработано, и если да, то каким образом.

**Фильтры исключений:**

EXCEPTION\_EXECUTE\_HANDLER –сообщает системе: «Я вижу это исключение; так и знал, что оно где-нибудь произойдет; у меня есть код для его обработки, и я хочу его сейчас выполнить».

EXCEPTION\_CONTINUE\_SEARCH – указывает системе перейти к предыдущему блоку \_\_try, которому соответствует блок \_\_except, и обработать его фильтр.

EXCEPTION\_CONTINUE\_EXECUTION – обнаружив такое значение выражения в фильтре, система возвращается к инструкции, вызвавшей исключение, и пытается выполнить ее снова.

**Глобальная раскрутка** — процесс, при котором происходит раскрутка всего стека вызовов, включая вызовы функций, находящихся во всех модулях. Это необходимо для корректной обработки исключений, которые могут возникнуть в различных частях программы.

Альтернативный подход заключается в том, что при неудачном вызове функции **возбуждают исключения**. Тогда написание и сопровождение кода становится гораздо проще, а программы работают намного быстрее. Последнее связано с тем, что та часть кода, которая отвечает за контроль ошибок, вступает в действие лишь при сбоях, т. е. в исключительных ситуациях.

Возбудить программное исключение несложно – достаточно вызвать функцию **RaiseException**. Ее первый параметр, **dwExceptionCode**, – значение, которое идентифицирует генерируемое исключение. Второй параметр функции – **dwExceptionFlags** – должен быть либо 0, либо EXCEPTION\_NONCONTINUABLE. В принципе этот флаг указывает, может ли фильтр исключений вернуть EXCEPTION\_CONTINUE\_EXECUTION в ответ на данное исключение. Третий и четвертый параметры (**nNumberOfArguments** и **pArguments**) функции RaiseException позволяют передать дополнительные данные о генерируемом исключении.

## **42. Виртуализация. Что такое BPF? Каковы преимущества использования BPF? Что такое трассировка? Что такое инструменты выборки? Что такое наблюдаемость? Виды программ в eBPF. Что такое CRIU? Что такое checkpoint dumping? Зачем он нужен?**

**BPF** расшифровывается как Berkeley Packet Filter (пакетный фильтр Беркли). BPF — это гибкая и эффективная технология, состоящая из набора инструкций, хранимых объектов и вспомогательных функций. Она определяет набор виртуальных инструкций, поэтому ее можно считать виртуальной машиной. Эти инструкции исполняются средой выполнения BPF в ядре Linux, которая включает интерпретатор и JIT-компилятор, преобразующие инструкции BPF в машинные инструкции.

**Трассировка** - технология фиксации (записи) происходящих событий, основанная на использовании соответствующих инструментов BPF. Возможно, вам уже приходилось иметь дело с некоторыми специализированными инструментами трассировки. Инструмент strace, например, фиксируети выводит события обращения к системным вызовам. Есть и инструменты, которые не трассируют, а измеряют события, используя фиксированные статистические счетчики, а затем выводят их, как, например, top. Отличительная черта трассировщика — это способность фиксировать исходные события и их метаданные.

**Инструменты выборки** выполняют некоторые измерения, чтобы составить общую картину цели. Их также называют инструментами создания профиля, или профилирования. Есть BPF-инструмент под названием profile, который берет выполняемый код по таймеру. Например, он может производить выборку каждые 10 миллисекунд, или, иначе говоря, 100 раз в секунду (на каждом процессоре). Преимущество инструментов выборки состоит в том, что у них обычно более низкий уровень оверхеда, чем у трассировщиков, потому что они оценивают только одно из большого числа событий. С другой стороны, выборка дает только приблизительную картину и может пропускать некоторые важные события.

Под **наблюдаемостью** понимается исследование системы через наблюдение с использованием специальных инструментов. К инструментам наблюдаемости относятся инструменты трассировки и выборки, a также инструменты, основанные на фиксированных счетчиках. К ним не относятся инструменты бенчмаркинга, которые изменяют состояние системы, экспериментируя с рабочей нагрузкой.

**Первая категория программ** – программы трассировки. Многие написанные вами программы помогут лучше понять, что происходит в вашей системе. Они обеспечивают информацию о поведении системы и оборудовании, на котором она работает. Могут получать доступ к областям памяти, используемым конкретными программами, и трассировать выполнение запущенных процессов. Они также дают прямой доступ к ресурсам, выделенным для каждого конкретного процесса, от применения файловых дескрипторов до использования процессора и памяти.

**Вторая категория программ** – программы для работы в сети. Они позволяют вам контролировать сетевой трафик в своей системе, фильтровать пакеты, поступающие от сетевого интерфейса, и даже полностью отбрасывать их. Разные типы программ могут быть связаны с различными этапами сетевой обработки в ядре. В этом есть и преимущества, и недостатки. Например, вы можете привязать программы BPF к сетевым событиям, как только ваш сетевой драйвер получает пакет, но такая программа будет иметь доступ к меньшему количеству сведений о пакете, поскольку ядро еще не обладает достаточной информацией. В то же время можно привязать программы BPF к сетевым событиям непосредственно перед их передачей в пространство пользователя. В этом случае у вас будет гораздо больше информации о пакете, что поможет принимать более обоснованные решения, однако придется затратить ресурсы для полной обработки пакета.

**Checkpoint dumping** - мощный инструмент для Linux, позволяющий создавать моментальные снимки процессов и восстанавливать их позже. **CRIU** – библиотека с открытым исходным кодом, которая упрощает эту задачу.

**Checkpoint dumping** – процесс создания «моментального снимка» состояния работающего процесса в Linux. Это не просто копирование файла; это гораздо более сложная операция, охватывающая все аспекты процесса, необходимые для его возобновления. Сюда входит память процесса (его содержимое и структура данных), открытые файлы и их указатели, сетевые соединения (включая сокеты и буферы), а также другие важные ресурсы, используемые процессом. Созданный снимок – это застывшее во времени состояние процесса, которое можно сохранить на диск для последующего восстановления.

## **43. Виртуализация. Что такое виртуализация? Перечислите и опишите типы виртуализации. Из каких компонент состоит виртуализация? Опишите гипервизоры первого и второго типов. Что такое контейнеризация? Принципы работы и основные компоненты контейнеризации. Namespaces и cgroups в Linux.**

**Виртуализация** позволяет создавать виртуальные машины (ВМ) на физическом сервере. Каждая ВМ получает свою собственную ОС и ресурсы, изолированные от других ВМ. Включает эмуляцию полноценной аппаратной инфраструктуры.

**Типы виртуализации**:

a. Полная виртуализация (полная эмуляция оборудования, гостевая ОС не требует модификации)

b. Паравиртуализация (частичная эмуляция, требует модификации гостевой ОС)

c. Аппаратная виртуализация (поддержка на уровне процессора, технологии Intel VT-х и AMD-V)

**Компоненты**: гипервизор (VMM - Virtual Machine Monitor), виртуальные машины, виртуальные ресурсы (CPU, RAM, storage).

**Гипервизор типа 1**, также известный как "голый металл", работает непосредственно на аппаратном обеспечении компьютера, без посредничества ОС хоста. Это обеспечивает высокую производительность и безопасность, так как гипервизор имеет прямой доступ к ресурсам системы. Архитектура гипервизора типа 1 обычно включает в себя модули для управления виртуальными машинами, распределения ресурсов и обеспечения безопасности. Они нужны для работы с различными аппаратными платформами и предоставлять широкий спектр функций, включая виртуализацию памяти, процессора и ввода-вывода.

**Гипервизор типа 2** работает как обычное приложение поверх ОС хоста. Это упрощает установку и настройку, так как не требует глубокого знания аппаратного обеспечения. Однако, наличие промежуточного слоя ОС может снизить производительность по сравнению с гипервизором типа 1. Архитектура гипервизора типа 2 включает в себя взаимодействие с ОС хоста для доступа к ресурсам. Они часто предоставляют более простой интерфейс пользователя, что делает их более доступными для начинающих пользователей.

**Контейнеризация** – технология, которая позволяет изолировать приложения и их зависимости в "контейнерах". Контейнеры делят ресурсы хост-системы, но приложения в них работают независимо. Изоляция процессов без эмуляции оборудования.

**Контейнер** — изолированное окружение для запуска приложений. Образ контейнера содержит все необходимые зависимости для запуска приложения. Реестр образов хранит и предоставляет доступ к образам. Движок контейнеризации управляет жизненным циклом контейнеров.

**Ключевые технологии**:

a. Пространства имен (Namespaces): PID namespace, Network namespace, Mount namespace

b. Control Groups (cgroups): управление ресурсами, ограничение использования CPU/RAM

**Основные компоненты**: контейнерный движок (Docker, containerd), образы контейнеров, Runtime (runC).

**Namespaces** в Linux — механизм, который изолирует и виртуализирует системные ресурсы для групп процессов. По сути, это как если бы каждый процесс жил в своём собственном мире, не подозревая о существовании других.

Основные виды namespaces: 1) PID Namespace (изолирует идентификаторы процессов); 2) Mount Namespace (изолирует точки монтирования файловой системы); 3) UTS Namespace (позволяет изолировать имена узлов и доменов, т.е. hostname и domainname); 4) Network Namespace (изолирует сетевые интерфейсы, маршруты, таблицы ARP); 5) IPC Namespace (изолирует межпроцессное взаимодействие); 6) User Namespace (изолирует идентификаторы пользователей и групп); 7) Cgroup Namespace (изолирует видимость cgroups); 8) Time Namespace (позволяет изолировать системное время).

**Cgroups** — это механизм ядра Linux, который позволяет ограничивать, учитывать и изолировать использование системных ресурсов для групп процессов. Допустим, вы на афтепати и хотите, чтобы каждый гость съел не больше определённого количества пиццы. Cgroups делают то же самое, только вместо пиццы — CPU, память, диск и другие ресурсы.

Основные виды Cgroups: 1) CPU (контролирует использование процессорного времени); 2) Memory (ограничивает использование оперативной памяти и swap); 3) IO (контролирует ввод-вывод на блочные устройства); 4) Devices (управляет доступом к устройствам); PIDs (ограничивает количество процессов в группе).

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

**Оптимизация кода** – процесс преобразования части кода в другую функционально эквивалентную часть для улучшения одной или более характеристик кода. **Две самые важные характеристики** – скорость работы и размер кода. К **другим характеристикам** относятся энергопотребление, необходимое для выполнения кода, время компиляции кода и длительность JIT-компиляции.

**Нужно ли заниматься оптимизацией кода вручную**? Однозначного ответа нет, но если коротко – всё зависит от ситуации. Компиляторы постоянно совершенствуются в отношении методов, применяемых ими для оптимизации кода. Но они не идеальны. Тем не менее, вместо траты на ручную оптимизацию программы гораздо продуктивнее использовать специфические средства компилятора и дать ему возможность оптимизировать код.

Оптимизация должна проводиться строго при необходимости и проводиться с осторожностью (поэтому каждый шаг оптимизации должен быть тщательно отлажен и протестирован). «Преждевременная оптимизация – это корень всех бед». Как разработчик вы не должны при разработке думать об оптимальности, а должны думать о целостности и завершенности своего кода, т.е. нет смысла писать оптимальный код, который не выполняет своего предназначения.

**Оптимизация кода и системное программирование**:

1) Производительность:

* Оптимизация кода: улучшение времени выполнения, уменьшение использования памяти.
* Системное программирование: часто требует высокой производительности.

2) Эффективное использование ресурсов:

* Оптимизация кода: минимизация использования процессорного времени и памяти.
* Системное программирование: управление памятью и планирование задач.

3) Надежность и стабильность:

* Оптимизация кода: улучшение структуры кода и повышение стабильности программы.
* Системное программирование: требует высокой надежности.

4) Совместимость и переносимость:

* Оптимизация кода: адаптация кода для работы на различных платформах и архитектурах.
* Системное программирование: требует создания кода, работающего на различных платформах.

5) Безопасность:

* Оптимизация кода: улучшение безопасности кода, устранение уязвимостей и защиту от атак
* Системное программирование: требует высокого уровня безопасности.

6) Отладка и тестирование

* Оптимизация кода: процессы отладки и тестирования для выявления и устранения проблем в коде
* Системное программирование: требует тщательного тестирования и отладки.

**Возможно несколько уровней оптимизации:**

1. На самом абстрактном уровне вы можете оптимизировать программу, выбрав для нее лучший алгоритм. Этот метод не зависит от компилятора и языка программирования.
2. Снижая уровень абстракции, следующим шагом будет ручная оптимизация вашего кода на основе используемого вами HLL, при этом оптимизация не должна завесить от конкретной реализации этого языка. Хотя такие оптимизации могут быть неприменимы к другим языкам, они должны быть применимы в разных компиляторах для одного и того же языка.
3. Спустившись еще на один уровень, вы можете начать думать о структурировании кода таким образом, чтобы оптимизация была применима только к определенному поставщику или, возможно, только к определенной версии компилятора.
4. Наконец, на самом низком уровне вы можете учитывать машинный код, который выдаёт компилятор, и корректировать инструкции, написанные так, чтобы заставить компилятор генерировать некоторую определённую оптимизированную последовательность машинных инструкций.

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

**Оптимизация кода** – процесс преобразования части кода в другую функционально эквивалентную часть для улучшения одной или более характеристик кода. **Две самые важные характеристики** – скорость работы и размер кода. К **другим характеристикам** относятся энергопотребление, необходимое для выполнения кода, время компиляции кода и длительность JIT-компиляции.

Хотя описанный процесс разработки может быть немного завышен в требованиях, одно можно сказать наверняка: программисты, которые его используют, создадут наилучший машинный код с помощью компилятора. Это тот тип кода, который сопоставим с тем, что создают приличные программисты на языке ассемблера, и тот тип выходных данных компилятора, на который любят ссылаться программисты HLL, утверждая, что компиляторы создают код, сравнимый с рукописным ассемблером. Тщательно написанный HLL-код может быть почти таким же эффективным, как и приличный ассемблерный код.

Понимание того, как компилятор организует промежуточный код (для получения более качественного машинного кода на более поздних этапах), очень важно, если вы хотите помочь оптимизатору выполнять свою работу более эффективно.

По мере прохождения управления по программе оптимизатор отслеживает значения переменных в процессе, известном как **анализ потока данных (DFA)**. После тщательного анализа компилятор может определить, где переменная не инициализирована, когда переменная содержит определенные значения, когда программа больше не использует переменную и (что не менее важно) когда компилятор просто ничего не знает о значении переменной.

Чтобы проанализировать поток данных, компиляторы разбивают исходный код на последовательности, известные как **базовые блоки** – машинные инструкции, от которых нет ответвлений, кроме как в начале и конце. 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, на основе их начальных значений и вычислений для них в базовом блоке

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.

Любая программа, состоящая только из структурированных управляющих команд (if, while, repeat..until и т.д.) и избегающая операторов goto, будет **упрощаемой**.

**Основные типы оптимизаций компилятора**:

1. Свёртка констант
2. Распространение констант
3. Удаление мёртвого кода
4. Удаление общих подвыражений
5. Снижение стоимости операций
6. Анализ индуктивных переменных
7. Анализ инвариантов цикла

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

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

**Основные типы оптимизаций компилятора**:

1. Свёртка констант (вычисление значений константных выражений или подвыражений во время компиляции, а не во время выполнения)
2. Распространение констант (замена переменных постоянными значениями, если компилятор определяет, что программа присвоила эту константу переменной ранее в коде)
3. 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)
4. Удаление общих подвыражений (оптимизация компилятора, которая ищет экземпляры одинаковых выражений и анализирует возможность замены их на одну переменную с вычисленным значением)
5. Снижение стоимости операций (замена медленных операций, например, умножения и деления, на более быстрые, такие как сложение, вычитание, сдвиг)
6. Анализ индуктивных переменных (во многих выражениях, особенно в тех, которые появляются в цикле, значение одной переменной в выражении полностью зависит от какой-либо другой переменной. Часто компилятор может исключить вычисление нового значения или объединить два вычисления в одно на время этого цикла)
7. Анализ инвариантов цикла (Инвариант цикла – это выражение, которое не меняется на каждой итерации некоторого цикла. Оптимизатор может вычислить результат такого вычисления только один раз, вне цикла, а затем использовать вычисленное значение в теле цикла)

**Вычисления по короткой схеме** – 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.

**Скорость выполнения операций:**

*Самые быстрые* – целочисленное сложение, вычитание, отрицание, логические операции AND, OR, XOR и сравнения, логические сдвиги, логические вращения. *Умеренная скорость* – умножение. *Медленнее* – деление. *Ещё медленнее* – сравнения с плавающей точкой и отрицание, сложение и вычитание с плавающей точкой, умножение с плавающей точкой. *Самые медленные* – деление с плавающей точкой.

Чем выше уровень оптимизации, тем более радикальные изменения компилятор вносит в программу. 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. Мы их называем **блокировщиками оптимизации**.

**Два типа блокировщиков оптимизации**:

Одним из них являются **указатели**. Компилятор не может точно знать, будут ли два указателя указывать на одну и ту же область памяти, и поэтому не выполняет некоторые оптимизации. Другой блокировщик оптимизации – **вызов функций**. Они влекут накладные расходы, и нам нужно стараться их избегать.

Данные имеют важное свойство, которое мы называем **локальностью**. Когда мы работаем над данными, желательно чтобы они находились в памяти рядом. Обход матрицы по столбцам имеет плохую локальность, потому что матрица хранится в памяти построчно.

Локальность бывает двух типов. Когда мы обращаемся к одному и тому же месту в памяти много раз, это **временнáя** локальность. Когда мы обращаемся к данным, а потом обращаемся к другим данным, которые расположены в памяти рядом с первоначальными, это **пространственная** локальность.

## **1. Опишите алгоритм сборки программы с использованием CMake и Clang.**

Шаг 1: Установка необходимых инструментов

Шаг 2: Создание структуры проекта

MyProject/

├── CMakeLists.txt

└── MyProject/

└── main.c

├── CMakeLists.txt

Шаг 3: Написание CMakeLists.txt

cmake\_minimum\_required(VERSION 3.18)

project(MyProject C)

add\_subdirectory("MyProject")

add\_executable(MyProject MyProject/main.c)

if (CMAKE\_VERSION VERSION\_GREATER 3.12)

set\_property(TARGET MyProject PROPERTY C\_STANDARD 11)

endif()

Шаг 4: Написание кода

#include <iostream>

int main() {

std::cout << "Hello, CMake with Clang!" << std::endl;

return 0;

}

Шаг 5: Генерация сборки

cd path/to/MyProject

mkdir build

cd build

cmake -G Ninja -DCMAKE\_C\_COMPILER=clang ..

Шаг 6: Сборка проекта

ninja

Шаг 7: Запуск программы

MyProject

./MyProject

## **2. Опишите алгоритм запуска программы через дочерний процесс в Windows.**

Шаг 1: Подготовка необходимых заголовков

#include <windows.h>

#include <iostream>

using namespace std;

Шаг 2: Определение функции для запуска дочернего процесса

void LaunchProcess(const char\* applicationPath) {

// Структура для информации о процессе

STARTUPINFO si;

PROCESS\_INFORMATION pi;

// Инициализация структуры

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

ZeroMemory(&pi, sizeof(pi));

// Запуск процесса

if (!CreateProcess(

applicationPath, // Путь к исполняемому файлу

NULL, // Командная строка

NULL, // Процесс-родитель

NULL, // Поток-родитель

FALSE, // Нельзя наследовать дескрипторы

0, // Флаги создания

NULL, // Указывать переменные окружения

NULL, // Текущий каталог

&si, // Указатель на структуру STARTUPINFO

&pi // Указатель на структуру PROCESS\_INFORMATION

)

) {

cerr << "Не удалось запустить процесс. Ошибка: " << GetLastError() << endl;

return;

}

WaitForSingleObject(pi.hProcess, INFINITE); // Ждем завершения доч. процесса

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

}

Шаг 3: Вызов функции

int main() {

const char\* pathToExecutable = "C:\\Path\\To\\Your\\Executable.exe";

LaunchProcess(pathToExecutable);

return 0;

}

## **3. Опишите алгоритм запуска программы через дочерний процесс в Linux.**

Шаг 1: Подготовка необходимых заголовков

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

Шаг 2: Определение функции для запуска дочернего процесса

void LaunchProcess(const char\* programPath) {

pid\_t pid = fork(); // Создаем новый процесс

if (pid < 0) {

// Ошибка при создании процесса

perror("Ошибка fork");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

// Это дочерний процесс

execl(programPath, programPath, NULL); // Запускаем программу

// Если execl вернется, значит произошла ошибка

perror("Ошибка exec");

exit(EXIT\_FAILURE);

} else {

// Это родительский процесс

int status;

waitpid(pid, &status, 0); // Ждем завершения дочернего процесса

if (WIFEXITED(status)) {

printf("Доч. процесс завершился с кодом: %d\n", WEXITSTATUS(status));

}

}

}

Шаг 3: Вызов функции

int main() {

const char\* pathToExecutable = "/path/to/your/executable";

LaunchProcess(pathToExecutable);

return 0;

}

Этот алгоритм описывает, как запустить программу через дочерний процесс в Linux с использованием системных вызовов `fork()` и `exec()`. Функция `fork()` создает новый процесс, а `execl()` заменяет его образ на исполняемый файл. Родительский процесс ждет завершения дочернего с помощью `waitpid()`.

## **4. Опишите алгоритм установки связи между дочерним и родительским процессами посредством анонимного канала в Windows.**

void CreatePipeAndLaunchProcess() {

HANDLE hReadPipe, hWritePipe;

SECURITY\_ATTRIBUTES sa;

…

// Создаем анонимный канал

if (!CreatePipe(&hReadPipe, &hWritePipe, &sa, 0)) {

fprintf(stderr, "Ошибка при создании канала: %d\n", GetLastError());

exit(EXIT\_FAILURE);

}

// Создаем дочерний процесс

STARTUPINFO si;

PROCESS\_INFORMATION pi;

…

si.hStdOutput = hWritePipe; // Подключаем вывод дочернего процесса к каналу

si.hStdError = hWritePipe; // Подключаем ошибочный вывод доч. процесса к каналу

…

// Запускаем дочерний процесс

if (!CreateProcess(

"C:\\Path\\To\\Your\\Executable.exe",

NULL, // Командная строка

NULL, // Процесс-родитель

NULL, // Поток-родитель

TRUE, // Наследование дескрипторов

0, // Флаги создания

NULL, // Переменные окружения

NULL, // Текущий каталог

&si, // Указатель на структуру STARTUPINFO

&pi // Указатель на структуру PROCESS\_INFORMATION

)) {

fprintf(stderr, "Ошибка при запуске процесса: %d\n", GetLastError());

exit(EXIT\_FAILURE);

}

// Закрываем дескрипторы записи в канале в родительском процессе

CloseHandle(hWritePipe);

// Чтение данных из канала

char buffer[128];

DWORD bytesRead;

while (ReadFile(hReadPipe, buffer, sizeof(buffer) - 1, &bytesRead, NULL) &&

bytesRead > 0) {

buffer[bytesRead] = '\0'; // Завершаем строку

printf("Получено: %s", buffer); // Выводим данные

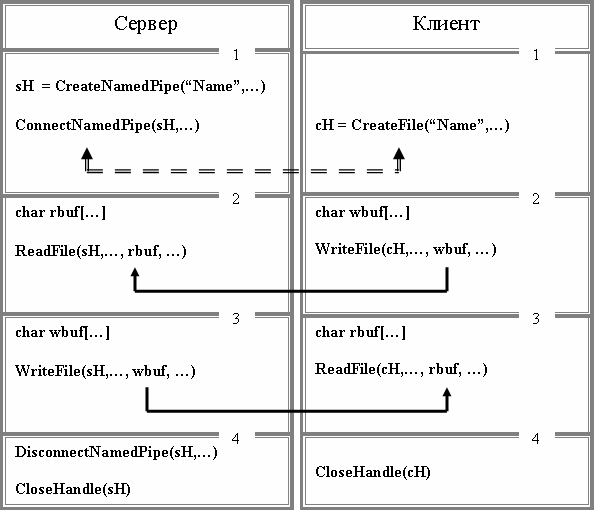
}

// Закрываем дескрипторы

…

}

## **5. Опишите алгоритм установки связи между двумя процессами посредством именованного канала в Windows.**



## **6. Опишите алгоритм установки связи между двумя процессами посредством fifo-очереди в Linux.**

#### Серверная функция

void ServerFunction() {

const char \*fifoPath = "/tmp/myfifo";

char buffer[128];

mkfifo(fifoPath, 0666); // Создаем FIFO-очередь

int fd = open(fifoPath, O\_RDONLY); // Ожидаем подключения

…

read(fd, buffer, sizeof(buffer)); // Чтение данных из FIFO

printf("Получено сообщение от клиента: %s\n", buffer);

close(fd); // Закрываем дескриптор

}

#### Клиентская функция

void ClientFunction() {

const char \*fifoPath = "/tmp/myfifo";

const char \*message = "Привет от клиента!";

sleep(1); // Ожидание, пока FIFO не будет создана

int fd = open(fifoPath, O\_WRONLY); // Запись в FIFO

…

write(fd, message, strlen(message) + 1); // +1 для '\0'

close(fd); // Закрываем дескриптор

}

## **7. Опишите алгоритм работы с TLS в потоках средствами API в Windows.**

Шаг 1: Определение TLS-ключа

**DWORD tlsIndex;**

Шаг 2: Инициализация TLS-ключа

**tlsIndex = TlsAlloc();**

**if (tlsIndex == TLS\_OUT\_OF\_INDEXES) {**

**printf("Ошибка при выделении TLS-индекса: %lu\n", GetLastError());**

**return 1;**

**}**

Шаг 3: Создание потоков

**DWORD WINAPI ThreadFunc(LPVOID lpParam) {**

**// Установка значения TLS для потока**

**int\* threadData = (int\*)malloc(sizeof(int));**

**\*threadData = (int)(uintptr\_t)lpParam; // Пример хранения уникального значения**

**TlsSetValue(tlsIndex, threadData);**

**// Получение значения TLS**

**int\* value = (int\*)TlsGetValue(tlsIndex);**

**printf("Поток %d: TLS = %d\n", \*value, \*value);**

**// Освобождение ресурсов**

**free(threadData);**

**return 0;**

**}**

Шаг 4: Запуск потоков

**#define NUM\_THREADS 5**

**int main() {**

**HANDLE threads[NUM\_THREADS];**

**for (int i = 0; i < NUM\_THREADS; i++) {**

**threads[i] = CreateThread(NULL, 0, ThreadFunc, (LPVOID)(uintptr\_t)(i + 1), 0, NULL);**

**}**

**// Ожидание завершения всех потоков**

**WaitForMultipleObjects(NUM\_THREADS, threads, TRUE, INFINITE);**

**TlsFree(tlsIndex); // Освобождение TLS-ключа**

**return 0;**

**}**

Шаг 5: Завершение работы

После завершения работы с потоками не забудьте освободить выделенный TLS-ключ с помощью `TlsFree`.

## **8. Опишите алгоритм работы с TLS в потоках средствами POSIX API в Linux.**

Шаг 1: Включение необходимых заголовков

Шаг 2: Определение TLS-переменной

**pthread\_key\_t tlsKey;**

Шаг 3: Создание и инициализация TLS-ключа

**void cleanup(void\* ptr) {**

**free(ptr); // Освобождение памяти**

**}**

**int main() {**

**if (pthread\_key\_create(&tlsKey, cleanup) != 0) {**

**perror("Ошибка при создании TLS-ключа");**

**return 1;**

**}**

**}**

Шаг 4: Создание потоков

**void\* threadFunc(void\* arg) {**

**int\* threadData = malloc(sizeof(int)); // Выделение памяти для данных потока**

**\*threadData = (int)(uintptr\_t)arg; // Уникальное значение для потока**

**pthread\_setspecific(tlsKey, threadData); // Установка значения TLS**

**int\* value = (int\*)pthread\_getspecific(tlsKey); // Получение значения TLS**

**printf("Поток %d: TLS = %d\n", \*value, \*value);**

**return NULL;**

**}**

Шаг 5: Запуск потоков

**#define NUM\_THREADS 5**

**int main() {**

**// Создание TLS-ключа**

**if (pthread\_key\_create(&tlsKey, cleanup) != 0) {**

**perror("Ошибка при создании TLS-ключа");**

**return 1;**

**}**

**pthread\_t threads[NUM\_THREADS];**

**for (int i = 0; i < NUM\_THREADS; i++) {**

**if (pthread\_create(&threads[i], NULL, threadFunc, (void\*)(uintptr\_t)(i + 1)) != 0) {**

**perror("Ошибка при создании потока");**

**return 1;**

**}**

**}**

**// Ожидание завершения всех потоков**

**for (int i = 0; i < NUM\_THREADS; i++) {**

**pthread\_join(threads[i], NULL);**

**}**

**// Освобождение TLS-ключа**

**pthread\_key\_delete(tlsKey);**

**return 0;**

**}**

## **9. Опишите алгоритм работы с виртуальной памятью в Windows при условии, что сначала память резервируется и только потом появляется необходимость её использования.**

Шаг 1: Подключение необходимых заголовков

Шаг 2: Резервирование виртуальной памяти

**void\* ReserveVirtualMemory(SIZE\_T size) {**

**void\* reservedMemory = VirtualAlloc(NULL, size, MEM\_RESERVE, PAGE\_NOACCESS);**

**if (reservedMemory == NULL) {**

**fprintf(stderr, "Ошибка при резервировании памяти: %lu\n", GetLastError());**

**exit(EXIT\_FAILURE);**

**}**

**return reservedMemory;**

**}**

Шаг 3: Выделение и инициализация памяти по мере необходимости

**void\* CommitVirtualMemory(void\* reservedMemory, SIZE\_T size) {**

**void\* committedMemory = VirtualAlloc(reservedMemory, size, MEM\_COMMIT, PAGE\_READWRITE);**

**if (committedMemory == NULL) {**

**fprintf(stderr, "Ошибка при выделении памяти: %lu\n", GetLastError());**

**exit(EXIT\_FAILURE);**

**}**

**return committedMemory;**

**}**

Шаг 4: Освобождение памяти

**void FreeVirtualMemory(void\* memory) {**

**if (!VirtualFree(memory, 0, MEM\_RELEASE)) {**

**fprintf(stderr, "Ошибка при освобождении памяти: %lu\n", GetLastError());**

**}**

**}**

Шаг 5: Пример использования в `main`

**int main() {**

**SIZE\_T size = 1024 \* 1024; // 1 МБ**

**// Резервирование виртуальной памяти**

**void\* reservedMemory = ReserveVirtualMemory(size);**

**// Выделение и инициализация памяти**

**void\* committedMemory = CommitVirtualMemory(reservedMemory, size);**

**// Использование памяти**

**memset(committedMemory, 0, size); // Инициализация памяти нулями**

**// Освобождение памяти**

**FreeVirtualMemory(committedMemory);**

**FreeVirtualMemory(reservedMemory);**

**return 0;**

**}**

## **10. Опишите алгоритм работы с виртуальной памятью в Linux при условии, что сначала память резервируется и только потом появляется необходимость её использования.**

Шаг 1: Подключение необходимых заголовков

Шаг 2: Резервирование виртуальной памяти

**void\* ReserveVirtualMemory(size\_t size) {**

**void\* reservedMemory = mmap(NULL, size, PROT\_NONE, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);**

**if (reservedMemory == MAP\_FAILED) {**

**perror("Ошибка при резервировании памяти");**

**exit(EXIT\_FAILURE);**

**}**

**return reservedMemory;**

**}**

Шаг 3: Выделение и инициализация памяти по мере необходимости

**void\* CommitVirtualMemory(void\* reservedMemory, size\_t size) {**

**if (mprotect(reservedMemory, size, PROT\_READ | PROT\_WRITE) == -1) {**

**perror("Ошибка при выделении памяти");**

**exit(EXIT\_FAILURE);**

**}**

**return reservedMemory;**

**}**

Шаг 4: Освобождение памяти

**void FreeVirtualMemory(void\* memory, size\_t size) {**

**if (munmap(memory, size) == -1) {**

**perror("Ошибка при освобождении памяти");**

**}**

**}**

Шаг 5: Пример использования в `main`

**int main() {**

**size\_t size = 1024 \* 1024; // 1 МБ**

**// Резервирование виртуальной памяти**

**void\* reservedMemory = ReserveVirtualMemory(size);**

**// Выделение и инициализация памяти**

**CommitVirtualMemory(reservedMemory, size);**

**// Использование памяти**

**memset(reservedMemory, 0, size); // Инициализация памяти нулями**

**// Освобождение памяти**

**FreeVirtualMemory(reservedMemory, size);**

**return 0;**

**}**

## **11. Опишите алгоритм работы с пользовательской кучей в Windows.**

Шаг 1: Подключение необходимых заголовков

Шаг 2: Создание пользовательской кучи

**HANDLE CreateCustomHeap() {**

**HANDLE hHeap = HeapCreate(HEAP\_GENERATE\_EXCEPTIONS, 0, 0);**

**if (hHeap == NULL) {**

**fprintf(stderr, "Ошибка при создании кучи: %lu\n", GetLastError());**

**exit(EXIT\_FAILURE);**

**}**

**return hHeap;**

**}**

Шаг 3: Выделение памяти из кучи

**void\* AllocateMemoryFromHeap(HANDLE hHeap, SIZE\_T size) {**

**void\* ptr = HeapAlloc(hHeap, HEAP\_GENERATE\_EXCEPTIONS, size);**

**if (ptr == NULL) {**

**fprintf(stderr, "Ошибка при выделении памяти: %lu\n", GetLastError());**

**exit(EXIT\_FAILURE);**

**}**

**return ptr;**

**}**

Шаг 4: Освобождение памяти

**void FreeMemoryFromHeap(HANDLE hHeap, void\* ptr) {**

**if (!HeapFree(hHeap, 0, ptr)) {**

**fprintf(stderr, "Ошибка при освобождении памяти: %lu\n", GetLastError());**

**}**

**}**

Шаг 5: Удаление кучи

**void DestroyCustomHeap(HANDLE hHeap) {**

**if (!HeapDestroy(hHeap)) {**

**fprintf(stderr, "Ошибка при уничтожении кучи: %lu\n", GetLastError());**

**}**

**}**

Шаг 6: Пример использования в `main`

**int main() {**

**// Создание пользовательской кучи**

**HANDLE hHeap = CreateCustomHeap();**

**// Выделение памяти из кучи**

**SIZE\_T size = 1024; // 1 КБ**

**void\* ptr = AllocateMemoryFromHeap(hHeap, size);**

**// Использование памяти (например, инициализация)**

**memset(ptr, 0, size);**

**// Освобождение выделенной памяти**

**FreeMemoryFromHeap(hHeap, ptr);**

**// Удаление пользовательской кучи**

**DestroyCustomHeap(hHeap);**

**return 0;**

**}**

## **12. Опишите алгоритм получения информации о файле в Windows.**

**void GetFileInformation(const char\* filePath) {**

**WIN32\_FILE\_ATTRIBUTE\_DATA fileInfo;**

// Получение атрибутов файла

**if (GetFileAttributesEx(filePath, GetFileExInfoBasic, &fileInfo)) {**

// Получение информации о размере файла

**LARGE\_INTEGER fileSize;**

**fileSize.LowPart = fileInfo.nFileSizeLow;**

**fileSize.HighPart = fileInfo.nFileSizeHigh;**

// Получение времени создания, последнего доступа и изменения

**FILETIME creationTime = fileInfo.ftCreationTime;**

**FILETIME lastAccessTime = fileInfo.ftLastAccessTime;**

**FILETIME lastWriteTime = fileInfo.ftLastWriteTime;**

// Конвертация времени в формат удобный для вывода

**char creationTimeStr[100], lastAccessTimeStr[100], lastWriteTimeStr[100];**

**SYSTEMTIME st;**

**FileTimeToSystemTime(&creationTime, &st);**

**sprintf(creationTimeStr, "%02d/%02d/%d %02d:%02d:%02d", st.wMonth, st.wDay, st.wYear, st.wHour, st.wMinute, st.wSecond);**

**FileTimeToSystemTime(&lastAccessTime, &st);**

**sprintf(lastAccessTimeStr, "%02d/%02d/%d %02d:%02d:%02d", st.wMonth, st.wDay, st.wYear, st.wHour, st.wMinute, st.wSecond);**

**FileTimeToSystemTime(&lastWriteTime, &st);**

**sprintf(lastWriteTimeStr, "%02d/%02d/%d %02d:%02d:%02d", st.wMonth, st.wDay, st.wYear, st.wHour, st.wMinute, st.wSecond);**

// Вывод информации о файле

**printf("Размер файла: %lld байт\n", fileSize.QuadPart);**

**printf("Дата и время создания: %s\n", creationTimeStr);**

**printf("Дата и время последнего доступа: %s\n", lastAccessTimeStr);**

**printf("Дата и время последнего изменения: %s\n", lastWriteTimeStr);**

**} else {**

**fprintf(stderr, "Ошибка получения информации о файле: %lu\n", GetLastError());**

**}**

**}**

## **13. Опишите алгоритм добавления строки в файл через WinAPI.**

Шаг 1: Подключение необходимых заголовков

Шаг 2: Определение функции для добавления строки в файл

**void AppendStringToFile(const char\* filePath, const char\* stringToAppend) {**

**// Открытие файла для добавления**

**HANDLE hFile = CreateFile(**

**filePath,**

**FILE\_APPEND\_DATA, // Открытие для добавления данных**

**FILE\_SHARE\_READ, // Разрешение на чтение другими процессами**

**NULL, // Без безопасности**

**OPEN\_ALWAYS, // Открыть, если существует, или создать новый**

**FILE\_ATTRIBUTE\_NORMAL, // Обычные атрибуты**

**NULL // Без шаблона файла**

**);**

**…**

**// Запись строки в файл**

**DWORD bytesWritten;**

**BOOL success = WriteFile(**

**hFile,**

**stringToAppend,**

**(DWORD)strlen(stringToAppend), // Количество байт для записи**

**&bytesWritten,**

**NULL // Без асинхронного ввода-вывода**

**);**

**if (!success) {**

**fprintf(stderr, "Ошибка при записи в файл: %lu\n", GetLastError());**

**} else {**

**printf("Записано %lu байт в файл.\n", bytesWritten);**

**}**

**CloseHandle(hFile); // Закрытие файла**

**}**

Шаг 3: Пример использования в `main`

**int main() {**

**const char\* filePath = "example.txt"; // Укажите путь к вашему файлу**

**const char\* stringToAppend = "Hello, World!\n"; // Строка для добавления**

**AppendStringToFile(filePath, stringToAppend);**

**return 0;**

**}**

## **14. Опишите алгоритм удаления строки в файл через WinAPI.**

void RemoveStringFromFile(const char\* filePath, const char\* stringToRemove) {

HANDLE hFile = CreateFile( // Открытие файла для чтения

filePath,

GENERIC\_READ, // Открытие для чтения

FILE\_SHARE\_READ, // Разрешение на чтение другими процессами

NULL, // Без безопасности

OPEN\_EXISTING, // Открыть, если существует

FILE\_ATTRIBUTE\_NORMAL, // Обычные атрибуты

NULL // Без шаблона файла

);

…

DWORD fileSize = GetFileSize(hFile, NULL); // Получение размера файла

…

char\* buffer = (char\*)malloc(fileSize + 1); // Чтение содержимого файла

…

DWORD bytesRead;

BOOL success = ReadFile(hFile, buffer, fileSize, &bytesRead, NULL);

…

buffer[bytesRead] = '\0'; // Завершение строки

CloseHandle(hFile); // Закрытие файла после чтения

// Удаление строки

char\* pos = strstr(buffer, stringToRemove);

if (pos != NULL) {

// Сдвинуть оставшуюся часть строки

size\_t lenToRemove = strlen(stringToRemove);

memmove(pos, pos + lenToRemove, bytesRead - (pos - buffer) - lenToRemove + 1); // +1 для 0-терминатора

bytesRead -= lenToRemove; // Обновление размера

}

hFile = CreateFile( // Открытие файла для записи

filePath,

GENERIC\_WRITE, // Открытие для записи

0, // Не разрешать совместный доступ

NULL, // Без безопасности

TRUNCATE\_EXISTING, // Удалить содержимое

FILE\_ATTRIBUTE\_NORMAL, // Обычные атрибуты

NULL // Без шаблона файла

);

…

// Запись измененного содержимого в файл

DWORD bytesWritten;

success = WriteFile(hFile, buffer, bytesRead, &bytesWritten, NULL);

if (!success) {

fprintf(stderr, "Ошибка при записи в файл: %lu\n", GetLastError());

} else {

printf("Удалено %lu байт из файла.\n", bytesWritten);

}

CloseHandle(hFile); // Освобождение ресурсов

free(buffer); // Освобождение ресурсов

}

## **15. Опишите алгоритм добавления строки в файл через POSIX API.**

Шаг 1: Подключение необходимых заголовков

Шаг 2: Определение функции для добавления строки в файл

**void AppendStringToFile(const char\* filePath, const char\* stringToAppend) {**

**// Открытие файла для добавления**

**int fd = open(filePath, O\_WRONLY | O\_APPEND | O\_CREAT, S\_IRUSR | S\_IWUSR);**

**…**

**// Запись строки в файл**

**ssize\_t bytesWritten = write(fd, stringToAppend, strlen(stringToAppend));**

**if (bytesWritten == -1) {**

**perror("Ошибка при записи в файл");**

**} else {**

**printf("Записано %zd байт в файл.\n", bytesWritten);**

**}**

**close(fd); // Закрытие файла**

**}**

Шаг 3: Пример использования в `main`

**int main() {**

**const char\* filePath = "example.txt"; // Укажите путь к вашему файлу**

**const char\* stringToAppend = "Hello, World!\n"; // Строка для добавления**

**AppendStringToFile(filePath, stringToAppend);**

**return 0;**

**}**

## **16. Опишите алгоритм удаления строки в файл через POSIX API.**

Шаг 1: Подключение необходимых заголовков

Шаг 2: Определение функции для удаления строки из файла

**void RemoveStringFromFile(const char\* filePath, const char\* stringToRemove) {**

**int fd = open(filePath, O\_RDONLY); // Открытие файла для чтения**

**…**

**off\_t fileSize = lseek(fd, 0, SEEK\_END); // Получение размера файла**

**lseek(fd, 0, SEEK\_SET); // Сброс указателя на начало файла**

**char\* buffer = (char\*)malloc(fileSize + 1); // Чтение содержимого файла**

**…**

**ssize\_t bytesRead = read(fd, buffer, fileSize);**

**…**

**buffer[bytesRead] = '\0'; // Завершение строки**

**close(fd); // Закрытие файла после чтения**

**// Удаление строки**

**char\* pos = strstr(buffer, stringToRemove);**

**if (pos != NULL) {**

**// Сдвинуть оставшуюся часть строки**

**size\_t lenToRemove = strlen(stringToRemove);**

**memmove(pos, pos + lenToRemove, bytesRead - (pos - buffer) - lenToRemove + 1);**

**bytesRead -= lenToRemove; // Обновление размера**

**}**

**// Открытие файла для записи (с обнулением)**

**fd = open(filePath, O\_WRONLY | O\_TRUNC);**

**…**

**// Запись измененного содержимого в файл**

**ssize\_t bytesWritten = write(fd, buffer, bytesRead);**

**if (bytesWritten == -1) {**

**perror("Ошибка при записи в файл");**

**} else {**

**printf("Удалено %zd байт из файла.\n", bytesWritten);**

**}**

**close(fd);**

**free(buffer);**

**}**

Шаг 3: Пример использования в `main`

**int main() {**

**const char\* filePath = "example.txt"; // Укажите путь к вашему файлу**

**const char\* stringToRemove = "Hello, World!\n"; // Строка для удаления**

**RemoveStringFromFile(filePath, stringToRemove);**

**return 0;**

**}**

## **17. Опишите алгоритм чтения строки файла, отображенного в память через WinAPI.**

void ReadStringFromFile(const char\* filePath, int lineNumber) {

// Открытие файла

HANDLE hFile = CreateFile(

filePath,

GENERIC\_READ, // Открытие для чтения

FILE\_SHARE\_READ, // Разрешение на чтение другими процессами

NULL, // Без безопасности

OPEN\_EXISTING, // Открыть, если существует

FILE\_ATTRIBUTE\_NORMAL, // Обычные атрибуты

NULL // Без шаблона файла

);

…

DWORD fileSize = GetFileSize(hFile, NULL); // Получение размера файла

…

HANDLE hMapping = CreateFileMapping( // Создание объекта отображения файла

hFile, NULL,

PAGE\_READONLY, // Чтение

0, 0, NULL

);

…

LPVOID pView = MapViewOfFile( // Отображение файла в память

hMapping, FILE\_MAP\_READ, 0, 0, 0);

// Чтение строки

char\* buffer = (char\*)pView;

int currentLine = 0;

char\* lineStart = buffer;

char\* lineEnd;

while (currentLine < lineNumber) {

lineEnd = strchr(lineStart, '\n');

if (lineEnd == NULL) {

break; // Достигнут конец файла

}

lineStart = lineEnd + 1; // Переход к следующей строке

currentLine++;

}

if (currentLine == lineNumber) {

// Завершение строки

\*lineEnd = '\0';

printf("Строка %d: %s\n", lineNumber + 1, lineStart);

} else {

printf("Строка %d не найдена.\n", lineNumber + 1);

}

UnmapViewOfFile(pView);

CloseHandle(hMapping);

CloseHandle(hFile);

}

## **18. Опишите алгоритм чтения строки файла, отображенного в память через POSIX API.**

void ReadStringFromFile(const char\* filePath, int lineNumber) {

int fd = open(filePath, O\_RDONLY); // Открытие файла

…

off\_t fileSize = lseek(fd, 0, SEEK\_END); // Получение размера файла

lseek(fd, 0, SEEK\_SET); // Сброс указателя на начало файла

// Отображение файла в память

char\* mapped = (char\*)mmap(NULL, fileSize, PROT\_READ, MAP\_PRIVATE, fd, 0);

…

// Чтение строки

char\* lineStart = mapped;

char\* lineEnd;

int currentLine = 0;

while (currentLine < lineNumber) {

lineEnd = strchr(lineStart, '\n');

if (lineEnd == NULL) {

break; // Достигнут конец файла

}

lineStart = lineEnd + 1; // Переход к следующей строке

currentLine++;

}

if (currentLine == lineNumber) {

// Завершение строки

lineEnd = strchr(lineStart, '\n'); // Найти конец строки

if (lineEnd != NULL) {

\*lineEnd = '\0'; // Завершение строки

}

printf("Строка %d: %s\n", lineNumber + 1, lineStart);

} else {

printf("Строка %d не найдена.\n", lineNumber + 1);

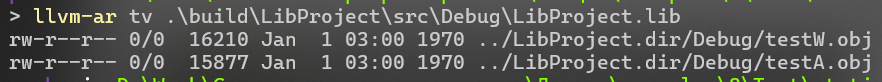
}

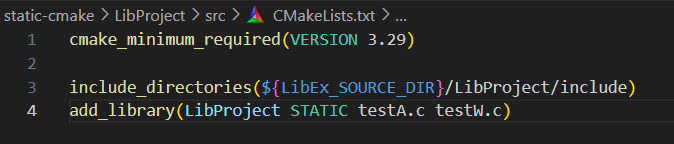
munmap(mapped, fileSize);

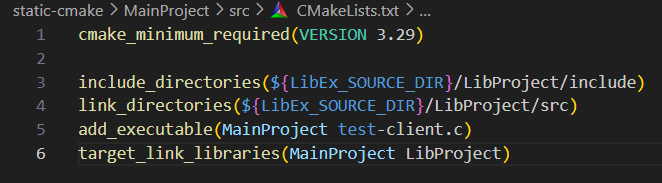
close(fd);

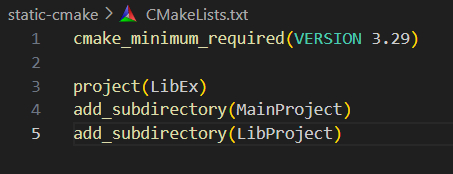
}

## **19. Опишите алгоритм сборки программы, использующей статическую библиотеку с использованием CMake и Clang.**

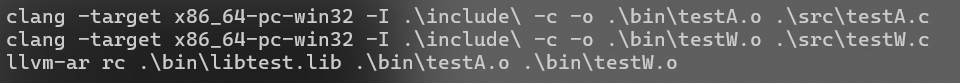


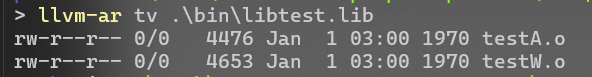




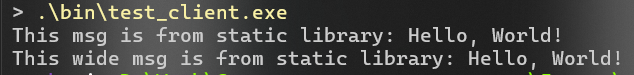


## **20. Опишите алгоритм сборки программы, использующей статическую библиотеку с использованием только Clang.**





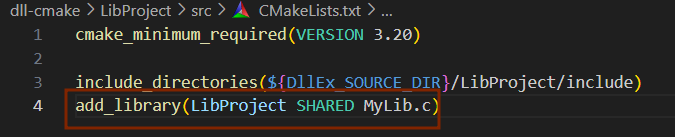


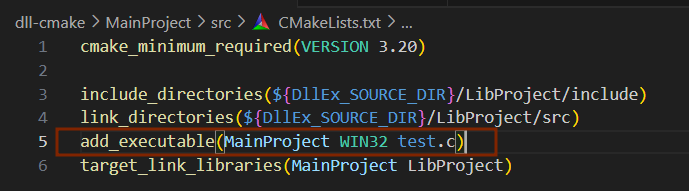


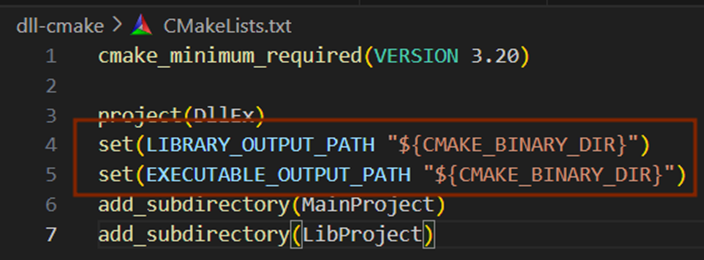
## **21. Опишите алгоритм сборки программы, неявно использующей динамическую библиотеку с использованием CMake и Clang.**

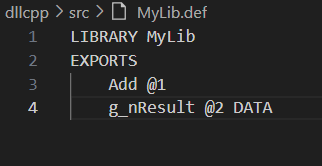




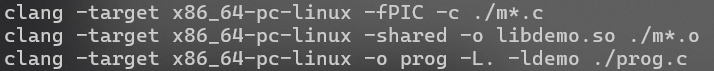














## **22. Опишите алгоритм сборки программы, неявно использующей динамическую библиотеку с использованием только Clang.**





## **23. Опишите алгоритм написания программы, явно использующей динамическую библиотеку средствами WinAPI.**

Шаг 1: Создание динамической библиотеки (DLL)

// my\_library.c

**extern "C" \_\_declspec(dllexport) void hello() { // Экспортируемая функция**

**std::cout << "Hello from MyLibrary!" << std::endl;**

**}**

**extern "C" \_\_declspec(dllexport) void onLoad() { // для указания, что DLL загружена**

**std::cout << "MyLibrary loaded!" << std::endl;**

**}**

**extern "C" \_\_declspec(dllexport) void onUnload() { // Указываем, что DLL выгружается**

**std::cout << "MyLibrary unloaded!" << std::endl;**

**}**

// my\_library.h

**#ifndef MY\_LIBRARY\_H**

**#define MY\_LIBRARY\_H**

**extern "C" \_\_declspec(dllexport) void hello();**

**extern "C" \_\_declspec(dllexport) void onLoad();**

**extern "C" \_\_declspec(dllexport) void onUnload();**

**#endif // MY\_LIBRARY\_H**

Шаг 2: Написание основной программы

**typedef void (\*HelloFunc)();**

**typedef void (\*OnLoadFunc)();**

**typedef void (\*OnUnloadFunc)();**

**int main() {**

**// Загрузка DLL**

**HMODULE hModule = LoadLibrary(TEXT("my\_library.dll"));**

**if (hModule == NULL) {**

**std::cerr << "Ошибка при загрузке DLL!" << std::endl;**

**return 1;**

**}**

**// Получение адресов функций**

**OnLoadFunc onLoad = (OnLoadFunc)GetProcAddress(hModule, "onLoad");**

**HelloFunc hello = (HelloFunc)GetProcAddress(hModule, "hello");**

**OnUnloadFunc onUnload = (OnUnloadFunc)GetProcAddress(hModule, "onUnload");**

**if (onLoad && hello && onUnload) {**

**// Вызов функций**

**onLoad();**

**hello();**

**onUnload();**

**} else {**

**std::cerr << "Ошибка при получении адресов функций!" << std::endl;**

**}**

**FreeLibrary(hModule); // Освобождение DLL**

**return 0;**

**}**

## **24. Опишите алгоритм написания программы, явно использующей динамическую библиотеку средствами POSIX API.**

Шаг 1: Создание динамической библиотеки

// my\_library.c

**#include <stdio.h>**

**void hello() { printf("Hello from MyLibrary!\n"); }**

// my\_library.h

**#ifndef MY\_LIBRARY\_H**

**#define MY\_LIBRARY\_H**

**void hello();**

**#endif // MY\_LIBRARY\_H**

Шаг 2: Написание основной программы

**#include <stdio.h>**

**#include "my\_library.h"**

**int main() {**

**hello();**

**return 0;**

**}**

Шаг 3: Запуск программы

export LD\_LIBRARY\_PATH=.:$LD\_LIBRARY\_PATH

./my\_program

Hello from MyLibrary!

## **25. Опишите алгоритм создания Windows-сервиса средствами WinAPI (без SCM).**

1) Регистрация сервиса:

SC\_HANDLE schService = CreateService( schSCManager, // дескриптор менеджера сервисов

L"MyService", // имя сервиса

L"MyService", // отображаемое имя сервиса

SERVICE\_ALL\_ACCESS, // тип доступа

SERVICE\_WIN32\_OWN\_PROCESS, // тип сервиса

SERVICE\_DEMAND\_START, // тип запуска

SERVICE\_ERROR\_NORMAL, // уровень ошибки

L"C:\\Path\\To\\MyService.exe", // путь к исполняемому файлу

nullptr,

nullptr,

nullptr,

nullptr,

nullptr);

2) Запуск и управление сервисом:

StartService(schService, 0, nullptr);

SERVICE\_STATUS status;

ControlService(schService, SERVICE\_CONTROL\_STOP, &status);

DeleteService(schService);

3) Освобождение ресурсов: CloseServiceHandle(…);

## **26. Опишите алгоритм создания Windows-сервиса средствами SCM.**

1) Подготовка исполняемого файла сервиса: создайте исполняемый файл, реализующий логику сервиса. Сервис должен:

• Иметь функцию ServiceMain, которая реализует логику работы сервиса.

• Иметь обработчик управляющих команд, который регистрируется через RegisterServiceCtrlHandler.

2) Регистрация сервиса в SCM:

SC\_HANDLE schSCManager = OpenSCManager(nullptr, nullptr, SC\_MANAGER\_CREATE\_SERVICE);

3) Создание записи о сервис:

SC\_HANDLE schService = CreateService( schSCManager, // Дескриптор SCM L"MyService", // Имя сервиса

L"My Display Name", // Отображаемое имя

SERVICE\_ALL\_ACCESS, // Доступ

SERVICE\_WIN32\_OWN\_PROCESS, // Тип сервиса

SERVICE\_AUTO\_START, // Тип запуска

SERVICE\_ERROR\_NORMAL, // Уровень ошибок

L"C:\\Path\\To\\MyService.exe", // Путь к исполняемому файлу

nullptr,

nullptr,

nullptr,

nullptr,

nullptr);

4) Закрытие дескрипторов:

CloseServiceHandle(schService);

CloseServiceHandle(schSCManager);

## **27. Опишите алгоритм создания Linux-сервиса средствами init.**

1) Создайте исполняемый файл демона:

2) Создайте скрипт init.d: Создайте скрипт в директории /etc/init.d/, который будет запускать ваш демон. Скрипт должен поддерживать команды start, stop, restart, status;

3) Сделайте скрипт исполняемым: sudo chmod +x /etc/init.d/mydaemon

4) Управление сервисом:

sudo /etc/init.d/mydaemon start

sudo /etc/init.d/mydaemon stop

## **28. Опишите алгоритм создания Linux-сервиса средствами systemd.**

1) Создайте исполняемый файл демона

2) Создайте файл unit для systemd:

3) Создайте файл unit в директории /etc/systemd/system/, например, mydaemon.service.

4) Перезагрузите systemd: После создания файла unit перезагрузите systemd, чтобы он распознал новый сервис: sudo systemctl daemon-reload

5) Запуск и управление сервисом:

sudo systemctl start mydaemon.service

sudo systemctl stop mydaemon.service