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

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

* Системное ПО
* **Промежуточное (связующее) ПО**
* Прикладное ПО

**Промежуточное (связующее) ПО** – совокупность программ, осуществляющих управление программными ресурсами, порожденными программами и ориентированными на решение широкого спектра задач

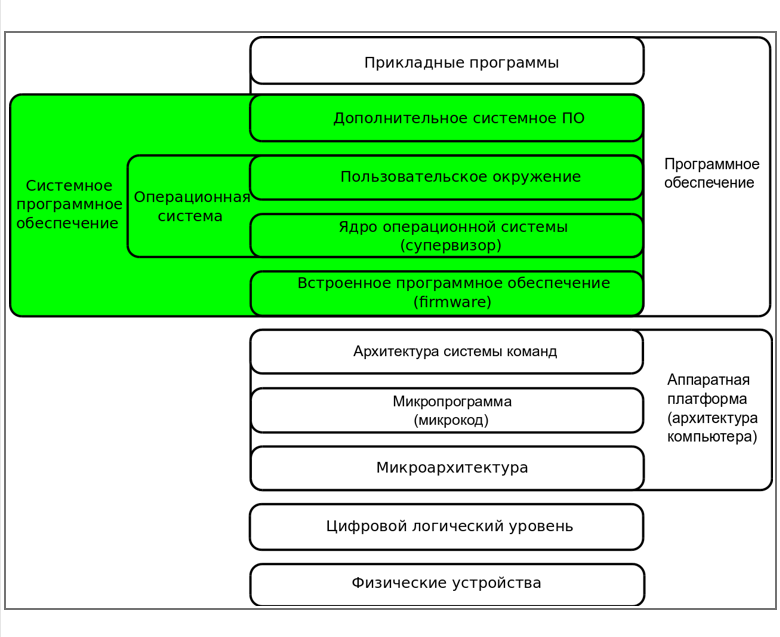
* Например: СУБД, модули управления языком интерфейса ИС, программы сбора и предварительной обработки информации

**Промежуточное (связующее) ПО** – комплекс технологического ПО для обеспечения взаимодействия между различными приложениями, системами, компонентами

* Например: Веб-сервер, сервер приложений, сервисная шина, система управления контентом

**Системное ПО**

Системное программное обеспечение — это программное обеспечение, которое управляет аппаратными компонентами компьютера и предоставляет платформу для выполнения прикладных программ. Оно обеспечивает взаимодействие между аппаратным обеспечением и прикладными программами.

\

К функциям системного ПО принято относить:

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

Существуют следующие группы Системного ПО:

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

**Классификация системного ПО**

**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

**Обслуживающее ПО (утилиты)** – программы, предназначенные для оказания услуг общего характера пользователям и обслуживающему персоналу СОИ

**Базовое системное ПО** – минимальный набор программных средств, обеспечивающий работу компьютера и компьютерной сети

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

Группы системного ПО

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

1. 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

**Системное программирование** – это процесс разработки программ сложной структуры

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

**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**

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

**Состав.** Системы программирования включают в себя следующие средства:

* Редактор текста
* Транслятор
* Компоновщик
* Отладчик
* Библиотеки подпрограмм

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

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



Виды трансляторов:

* Ассемблер
* Компилятор
* Интерпретатор
* Эмулятор
* Перекодировщик
* Макропроцессор

**Этапы подготовки программы:**

**КОМПИЛЯТОР**

**Шаг первый – Предварительная обработка кода**:

* Присоединение исходных файлов
* Работа макропроцессоров

**Шаг второй – Анализ**:

* Лексический анализ
* Синтаксический анализ
* Семантический анализ

**Шаг третий – Синтез**:

* Генерация машинно-независимого кода
* Оптимизация машинно-независимого кода
* Распределение памяти
* Генерация машинного кода
* Оптимизация машинного кода

Результатом работы компилятора является объектный модуль

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

Содержимое объектного модуля не содержит признаков на каком языке был написан исходный модуль

**КОМПОНОВЩИК:**

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

**В ВОПРОСЕ ТАКОГО НЕТ**

**Архитектурная модель ПО** – принципиальная организация ПО, воплощенная в его элементах, их взаимоотношениях друг с другом и со средой, а также принципы, направляющие проектирование и эволюцию ПО

Классификация ПО по количеству звеньев (уровней):

* Одноуровневая
* Двухуровневая
* Трёхуровневая

**Виды организации межпрограммного взаимодействия**:

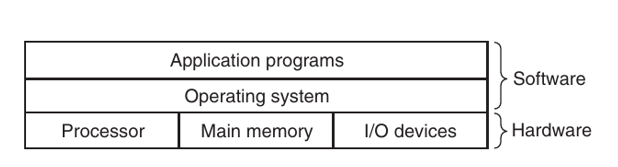
* На уровне библиотек
* На уровне сервисов, осуществляющих обмен данными по различным протоколам
* На уровне данных
* Через различные технологии интеграции (ESB, CORBA, COM, DCOM, ActiveX и др.)

1. 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?

**Операционная система** – комплекс системного программного обеспечения, который предоставляет полезные абстракции базовых аппаратных средств

Типичные аппаратные средства для которых операционная система предоставляет абстракции:

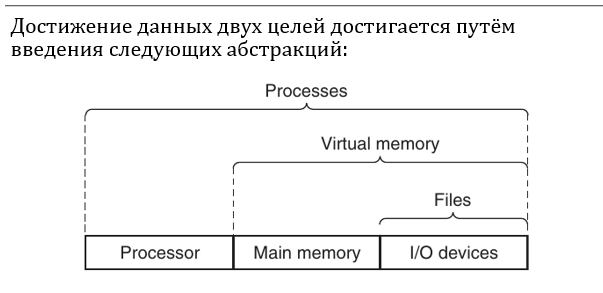
* Процессор
* RAM (она же первичная или физическая память)
* Диски (какой-либо вид вторичной памяти)
* Сетевые интерфейсы
* Дисплеи
* Клавиатуры
* Мыши



Операционная система 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

АБСТРАКЦИЯ



**Processes (Процессы):**

* Процессы представляют собой активные программы, выполняющиеся в системе. Каждому процессу выделяется память и ресурсы, позволяющие ему выполнять свои задачи

**Virtual Memory (Виртуальная память):**

* Виртуальная память позволяет системе использовать часть дискового пространства как дополнительную оперативную память. Это дает возможность запускать большие приложения, чем физическая память, и обеспечивает изоляцию процессов

**Files (Файлы):**

* Файлы представляют собой абстракцию для хранения данных на устройствах.

**Main Memory (Основная память):**

* Основная память (ОП) используется для временного хранения данных и инструкций, необходимых для выполнения процессов.

**I/O Devices (Устройства ввода-вывода):**

* Устройства ввода-вывода (например, клавиатуры, мыши, принтеры) позволяют пользователям взаимодействовать с компьютером.

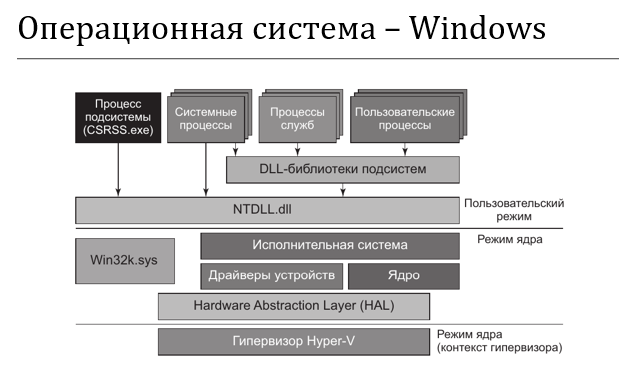
POSIX (Portable Operating System Interface) — это стандарт, разработанный для обеспечения совместимости и переносимости операционных систем. Он определяет интерфейсы, которые операционные системы должны поддерживать, включая системные вызовы, утилиты командной строки и библиотеки.

**POSIX** – набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов

**Ядро ОС** состоит из набора функций, предоставляющих фундаментальные механизмы системы. К их числу 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?

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

**Системный вызов (system call)** – 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 (traps)** – 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 (interrupts)** – 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

В частности в системном программировании в первую очередь изучаются **объекты ядра** операционной системы

В частности в Windows **объектами ядра** являются:

* маркеры доступа (access token objects),
* файлы (file objects),
* проекции файлов (file-mapping objects),
* порты завершения ввода-вывода (I/O completion port objects),
* задания (job objects),
* почтовые ящики (mailslot objects),
* мьютексы (mutex objects),
* каналы (pipe objects),
* процессы (process objects),
* семафоры (semaphore objects),
* потоки (thread objects)
* ожидаемые таймеры (waitable timer objects)

Дисписитор (или **дескриптор**) в контексте операционных систем и программирования — это специальный идентификатор, который используется для ссылки на объекты, управляемые ядром операционной системы.

**Дескриптор** — это специальный идентификатор, который операционная система использует для управления доступом к различным объектам ядра, таким как процессы, потоки, файлы и события.

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

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 процессной многозадачности?

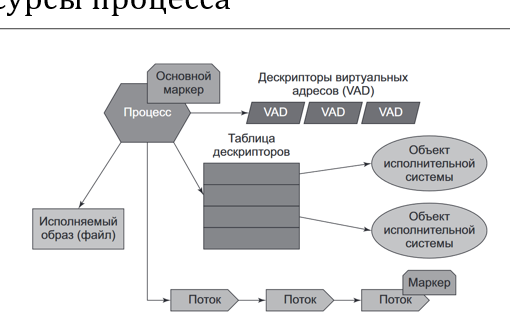
**Процесс** – управляющий объект, который обеспечивает изоляцию адресных пространств и представляет работающий экземпляр программы

**Процесс** – исполняемое на устройстве приложение весте со всеми ресурсами, которые требуются для его исполнения

* 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

К типичным ресурсам процесса относятся:

* Образ исполняемого файла
* Память
* Список дескрипторов объектов выделенных процессу (файлы, потоки, объекты синхронизации и т.д.)
* 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
* и т. д.

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 связанная с диспетчеризацией
* И т. д.

Чтобы запросить у ядра создание нового процесса необходимо совершить системный вызов

Системный вызов Windows – **NtCreateUserProcess**

Системные вызовы Linux (POSIX) – **fork**, **vfork**

Функция **fork** – предназначена для создания нового дочернего процесса, который будет являться полной (насколько это возможно) копией родительского процесса

Функция **exec** – предназначена для замены образа исполняемого файла в рамках существующего процесса

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

**Родительский процесс** –процесс инициировавший системный вызов

**СОСТАВ ПРОЦЕССА:**

* Процессу соответствует исполняемый программный файл
* У процесса есть PID
* У процесса есть Parent PID
* Идентификатор объекта процесса: в Windows - HANDLE и в Linux - pid\_t
* В OS есть процесс инициализации (родитель для всех)
* Запуск и управление (создать, остановить,…) процессом осуществляется с помощью системных вызовов
* Процессы изолированы друг от друга
* Процессу выделяется линейное адресное пространство (размер зависит от разрядности), сегменты: code, static, data, heap, stack
* Контекст процесса – данные, которые сохраняются при переключении процессов и предназначенные для продолжения работы
* Процессу автоматически доступны три потока данных: ввода (0), вывода (1), вывода ошибок (2)
* В составе ОS есть таблица, содержащая объекты ядра процессов (состояние, приоритет, указатели на другие объекты); есть средства ОС позволяющие ее просматривать
* Процесс – единица работы ОС

Процессная многозадачность — это способность операционной системы выполнять несколько процессов одновременно, обеспечивая каждому из них выделенное время для выполнения. В рамках этой концепции операционная система управляет процессами, обеспечивая их изоляцию и взаимодействие.

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 (далее – IPC): определение, классификация механизмов. Общие концепции передачи данных. В каких ситуациях предпочтительнее использовать те или иные IPC механизмы взаимодействия?

**IPC** (Inter-process communication, рус., **межпроцессное взаимодействие**) – механизм, позволяющий процессам обмениваться данными и синхронизировать свои действия

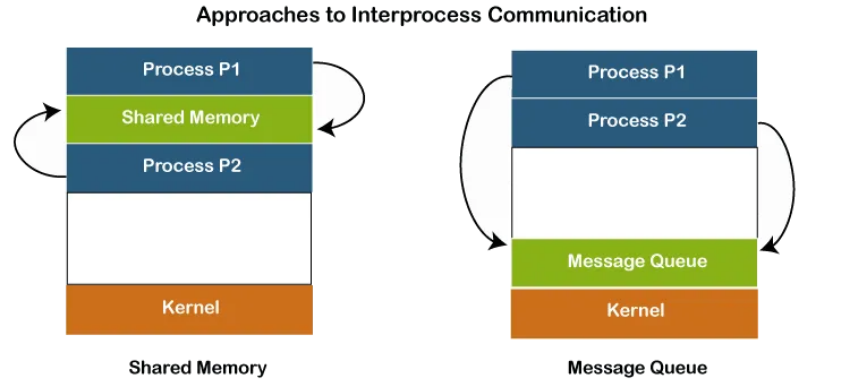
Это фундаментальная концепция системного программирования, поскольку она позволяет процессам работать вместе для достижения общей цели

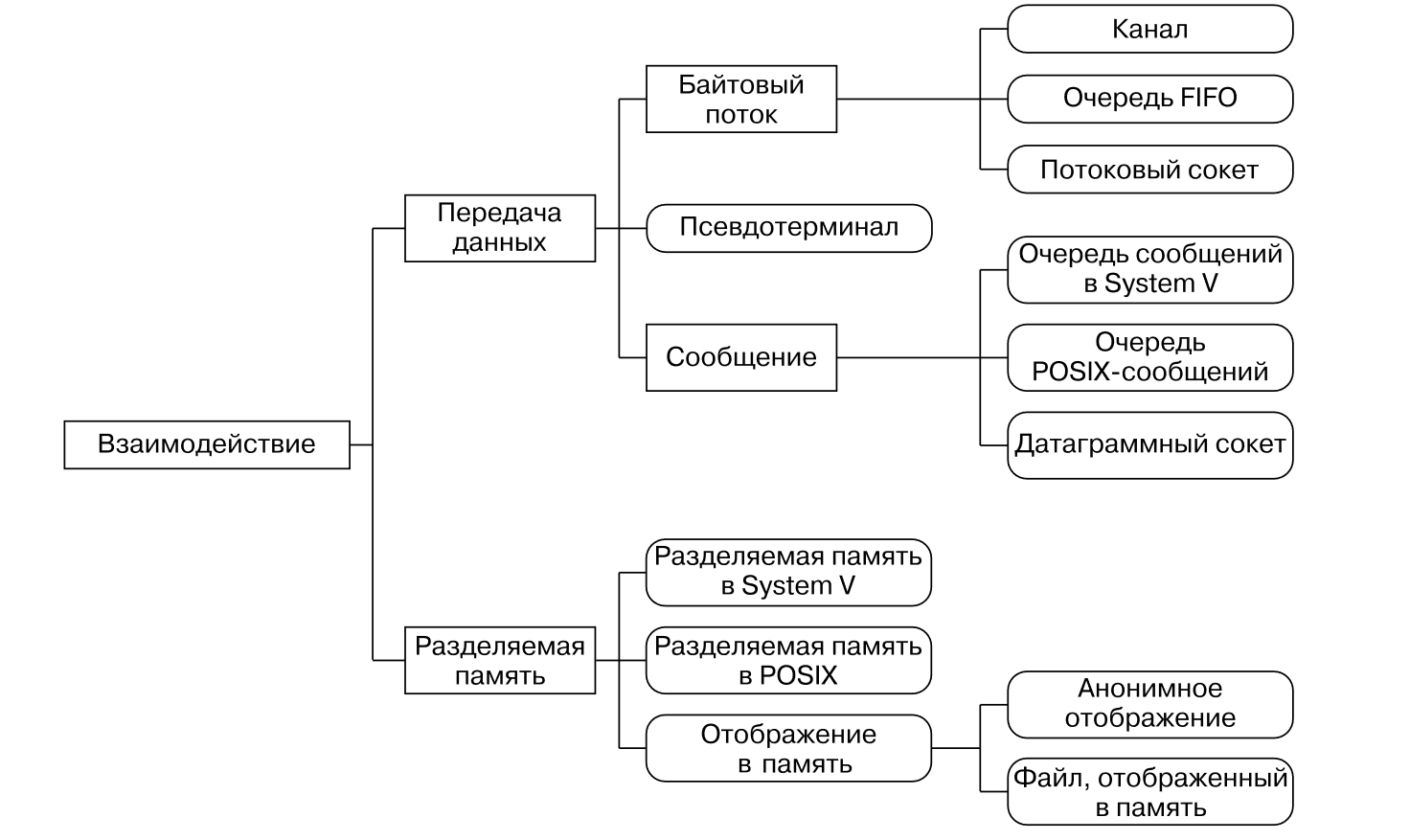
Категории IPC-механизмов:

* Взаимодействие
* Синхронизация
* Сигналы

Средства взаимодействия можно разделить на две категории:

* **Средства передачи данных**
  + Байтовый поток
  + Сообщение
  + Псевдотерминалы
* **Разделяемая память**





**Взаимодействие**

1. **Передача данных:**
   * **Байтовый поток:** Позволяет обмениваться данными между процессами через поток байтов.
   * **Очередь FIFO:** Реализует очередь, в которой данные обрабатываются в порядке их поступления.
   * **Псевдотерминал:** Эмулирует терминал, позволяя взаимодействовать между процессами как через консоль.
   * **Потоковый сокет:** Используется для сетевого взаимодействия между процессами, позволяя обмениваться данными через TCP/IP.
2. **Сообщения:**
   * **Очередь в System V:** Позволяет процессам обмениваться сообщениями через очередь, обеспечивая надежную передачу.
   * **Очередь POSIX-сообщений:** Современный аналог очередей сообщений, поддерживающий стандарты POSIX.
   * **Датаграммный сокет:** Используется для передачи сообщений без установления соединения, что позволяет отправлять данные в сеть.
3. **Разделяемая память:**
   * **Разделяемая память в System V:** Позволяет нескольким процессам совместно использовать область памяти для быстрого обмена данными.
   * **Разделяемая память в POSIX:** Аналогичный механизм, но с поддержкой стандартов POSIX.
4. **Анонимное отображение в память:**
   * Позволяет отображать файл или область памяти в адресное пространство процесса, что упрощает доступ к данным.
5. **Файл, отображаемый в память:**
   * Позволяет процессам взаимодействовать с файлами через их отображение в память, что ускоряет доступ к данным.

Общие концепции ПЕРЕДАЧИ ДАННЫХ:

Процесс, который посылает данные другому потоку, называется **отправителем**

Процесс, который получает данные от другого потока, называется **адресатом** или **получателем**

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

Концептуально обмен данными между процессами выполняется при помощи двух функций:

***send*** – послать данные

***receive*** – получить данные

При передаче данных может использоваться **прямая** или **косвенная** адресация процессов

При прямой адресации процессов в функциях send и receive явно указываются процессы отправитель и адресат

При косвенной адресации в функциях send и receive указываются не адреса, а имя связи, по которой передаются данные

Адресация процессов может быть **симметричной** и **асимметричной**. Если при обмене данными между процессами используется только прямая или только косвенная адресация, то такая адресация процессов называется симметричной. Если же при обмене данными между процессами используется как прямая, так и косвенная адресация, то такая адресация процессов называется асимметричной

При передаче данных различают **синхронный** и **асинхронный** обмен данными

Синхронная отправка – отправитель при отправке данных, блокируется до получения этих данных адресатом

Синхронное получение – адресат вызывая ***receive*** блокируется до тех пор, пока не получит данные

Асинхронные варианты данных операций не приводят к блокировке отправителя и адресата соответственно

(А)синхронный обмен = (А)синхронная отправка + (А)синхронное получение

В целом в системах может использоваться **смешанный** обмен данными

**Буфером** называется вместимость связи между процессами, т. е. количество данных, которые могут одновременно пересылаться по этой связи. Различаются три типа буферизации:

* **Нулевая вместимость связи** (нет буфера) – в этом случае возможен только синхронный обмен данными между процессами
* **Ограниченная вместимость связи** (ограниченный буфер) – в этом случае, если буфер полон, то отправитель данных должен ждать очистки буфера хотя бы от части данных
* **Неограниченная вместимость связи** (неограниченный буфер) – в этом случае отправитель никогда не ждет при отправке сообщения

**Канал** (pipe) – это однонаправленный канал связи, который позволяет передавать данные между двумя связанными процессами

Свойства присущие любым каналам:

* Полудуплексные
* Передача данных потоком
* Синхронный обмен данными
* Возможность моделирования любой топологии связей

**Каналы** бывают **анонимные** и **именованные**

Анонимные каналы применяются только в рамках родительски-дочерних отношений между процессами, откуда следует, что данные каналы могут применяться только в рамках одного устройства

Именованные каналы могут применяться между независимыми процессами, а также могут использоваться между процессами на разных устройствах (по сети)

Именованные каналы обладают дополнительными свойствами:

* Могут быть дуплексными
* Передача данных может быть и поточная, и сообщениями
* Есть возможность асинхронного обмена данными
* Имеют имя формата: «\\.\pipe\pipe\_name**» (**[**Windows**](https://learn.microsoft.com/en-us/windows/win32/ipc/pipes)**)**, в Linux имя не имеет строгого формата



Почтовый ящик ([**Mailslot**](https://learn.microsoft.com/en-us/windows/win32/ipc/mailslots)) – объект ядра операционной системы Windows, который обеспечивает передачу сообщений от процессов-клиентов к процессам-серверам, выполняющимся на компьютерах в пределах локальной сети

Почтовые ящики обладают следующими свойствами:

* Передача данных осуществляется сообщениями
* Направление передачи данных от клиента к серверу
* обмен данными может быть как синхронным, так и асинхронным
* Имеют имя формата: «\\.\mailslot\mailslot\_name**»**
* При размере сообщения до 425 Байт отправка широковещательная
* При размере сообщения от 426 Байт до 64 Кбайт – отправка от одного клиента одному серверу
* Создание почтового ящика сервером – [**CreateMailslot**](https://learn.microsoft.com/en-us/windows/desktop/api/Winbase/nf-winbase-createmailslota)
* Соединение клиента с почтовым ящиком – [**CreateFile**](https://learn.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-createfilea)
* Обмен данными через почтовый ящик – [**ReadFile**](https://learn.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-readfile) **+** [**WriteFile**](https://learn.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-writefile)
* Закрытие почтового ящика клиентом и сервером – [**CloseHandle**](https://learn.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle)

**Очередь сообщений** – IPC механизм использующийся для передачи информации между процессами. Считывающий и записывающий процессы обмениваются блоками (сообщениями) с четкими границами (в отличие от каналов, которые предоставляют сплошной байтовый поток)

Стандарт POSIX позволяет назначать каждому сообщению отдельный приоритет; сообщения с более высоким приоритетом передаются раньше остальных

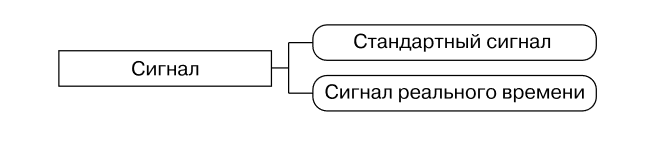
* Создание почтового ящика сервером – [**mq\_open**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/mq_open.html)
* Соединение клиента с почтовым ящиком – [**mq\_open**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/mq_open.html)
* Обмен данными через почтовый ящик – [**mq\_send**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/mq_send.html)+ [**mq\_receive**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/mq_receive.html)
* Закрытие почтового ящика клиентом и сервером – [**mq\_close**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/mq_close.html)+ [**mq\_unlink**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/mq_unlink.html)
* Оповещение о появлении сообщения – [**mq\_notify**](https://pubs.opengroup.org/onlinepubs/9799919799/functions/mq_notify.html)

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

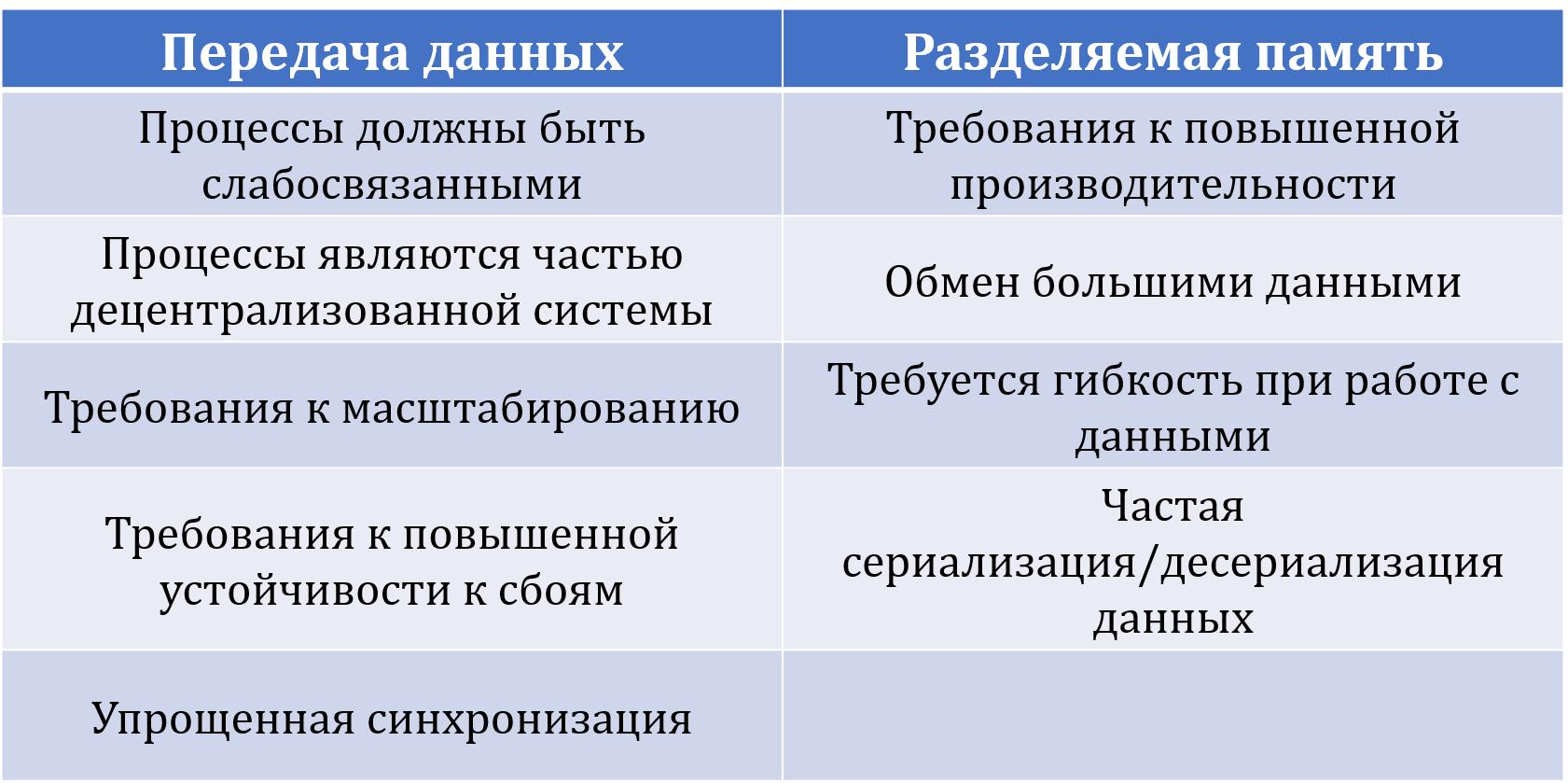
Свойства разделяемой памяти:

* Дуплексный обмен данными
* Синхронный обмен данными
* Возможность моделирования любой топологии связей

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



ГДЕ ЛУЧШЕ, ЧТО ИСПОЛЬЗОВАТЬ



|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

 Используйте **каналы** для простого синхронного обмена между связанными процессами.

 **Именованные каналы** подходят, когда процессы независимы и могут находиться на разных устройствах.

 **Почтовые ящики** хороши для передачи сообщений в локальной сети с различными размерами сообщений.

 **Очереди сообщений** удобны, когда нужно управлять приоритетами при обмене данными.

 **Разделяемая память** обеспечит высокую скорость обмена данными между процессами.

 **Сигналы** полезны для получения уведомлений о событиях, но не для передачи данных.

1. В какой ситуации предпочтительнее использовать каналы и очереди? Разделяемую память?

Для обработки данных в конвейерной структуре, когда один процесс передает вывод другому процессу, предпочтительнее выбрать каналы.

Если требуется обработка последующих запросов или важен приоритет сообщений, то очереди.

Разделяемую память предпочтительнее использовать в случае частого обмена большими объёмами данных

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:

* не прерывается во время своего исполнения
* контекст действия изменяется только самим действием

**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

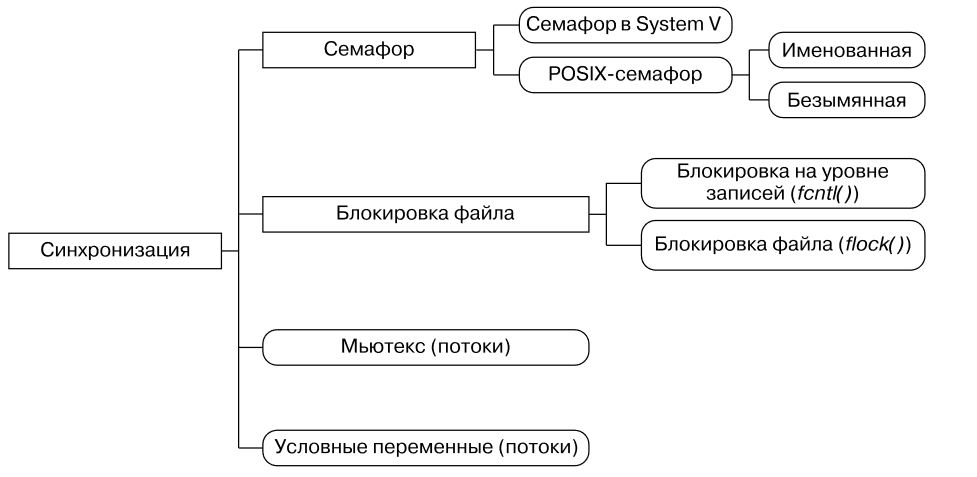
**Контекст действия изменяется только самим действием**:

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

1. Какие механизмы синхронизации вам известны?

* Критическая секция
* Семафор
* Мьютекс
* Условные переменные
* События ([WinAPI](https://learn.microsoft.com/en-us/windows/win32/sync/synchronization-functions), WinAPI [UseCase](https://learn.microsoft.com/en-us/windows/win32/sync/using-event-objects), [POSIX API](https://pubs.opengroup.org/onlinepubs/9799919799/functions/pthread_cond_init.html))
* Барьеры (WinAPI [about](https://learn.microsoft.com/en-us/windows/win32/sync/synchronization-barriers), [POSIX API](https://pubs.opengroup.org/onlinepubs/9799919799/functions/pthread_barrier_destroy.html))
* Блокировка файлов
* Спинлок ()
* RW-лок ()
* Условные переменные ()
* Атомарные операции ()



**Критическая секция** – это небольшой участок кода, требующий монопольного доступа к каким-то общим данным

Механизм для работы с критическими секциями в рамках одного процесса на подобии из Windows API не представлен в POSIX

В POSIX работа с критическими секциями базируется на объектах ядра в следствие специфичности понятия поток

Объекты ядра «**мьютексы**» гарантируют потокам взаимоисключающий доступ к единственному ресурсу

Поскольку «**мьютексы**» являются объектами ядра, то возможна синхронизация потоков между различными процессами

Для мьютексов определены следующие правила:

* если его идентификатор потока равен 0 (у самого потока не может быть такой идентификатор), мьютекс не захвачен ни одним из потоков и находится в свободном состоянии
* если его идентификатор потока не равен 0, мьютекс захвачен одним из потоков и находится в занятом состоянии

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

Для семафоров определены следующие правила:

* когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние
* если этот счетчик равен 0, семафор занят
* система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов
* счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов

Выбор конкретного механизма зависит от конкретных требований и сценариев использования. Например, если требуется простая и быстрая защита небольших участков кода, критические секции или мьютексы будут оптимальным выбором. Если же необходимо управлять доступом к ресурсу с ограниченной емкостью, лучше использовать семафоры

8. Потоки. Что такое поток выполнения? Контекст потока: определение, назначение и состав. Что такое потокобезопасная функция? Какая функция будет называться 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:

* Будем говорить, что программа является **многопоточной**, если в ней может одновременно существовать несколько потоков
* Если в программе одновременно может существовать только один поток, то такая программа называется **однопоточной**

**Контекст потока** – в общем случае, это содержимое памяти, к которой поток имеет доступ во время своего исполнения

**Назначение:**

Контекст потока используется для:

* **Сохранения состояния:** Позволяет сохранять текущее состояние потока, чтобы его можно было восстановить позже.
* **Управления выполнением:** Обеспечивает возможность переключения между потоками, что позволяет реализовать многозадачность.
* **Изоляции ресурсов:** Предоставляет отдельные ресурсы каждому потоку, что повышает безопасность и стабильность.

**Состав контекста потока:**

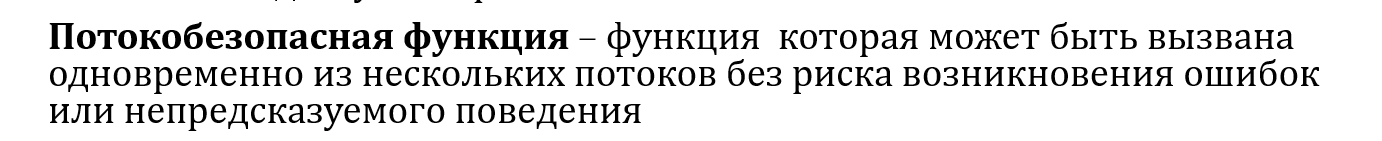
Контекст потока включает в себя следующие компоненты:

1. **Регистры процессора:**
   * Состояние всех регистров, включая указатели на стек и программный счетчик.
2. **Состояние стека:**
   * Информация о текущем стеке, включая адреса возврата и локальные переменные.
3. **Приоритет потока:**
   * Уровень приоритета, определяющий порядок выполнения потоков.
4. **Идентификатор потока:**
   * Уникальный идентификатор, позволяющий различать потоки.
5. **Контекст памяти:**
   * Ссылка на область памяти, используемую потоком, включая данные и код.
6. **Состояние ввода-вывода:**
   * Информация о любых операциях ввода-вывода, связанных с потоком.

**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

В каких случаях лучше использовать многозадачность на основе процессов, а в каких на основе потоков?

Многозадачность на основе процессов лучше использовать, когда важна безопасность и изоляция, так как процессы имеют собственные адресные пространства и не влияют друг на друга. Это также удобно для приложений, использующих разные языки или библиотеки, поскольку процессы могут работать независимо. Контроль ресурсов также проще реализовать на уровне процессов. С другой стороны, многозадачность на основе потоков предпочтительна, когда требуется высокая производительность и частое взаимодействие между компонентами. Потоки легче создавать и обеспечивают общий доступ к памяти, что упрощает обмен данными. Это особенно полезно для приложений, которые должны оставаться отзывчивыми, например, в графических интерфейсах, и для задач, которые могут быть эффективно распараллелены на многоядерных системах.

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

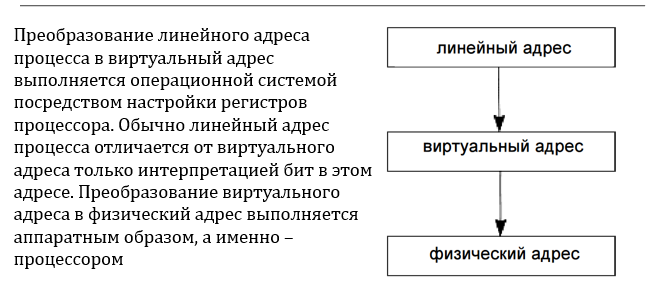
Под **логической памятью** процесса понимается массив байтов, к которым может обратиться процесс

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**

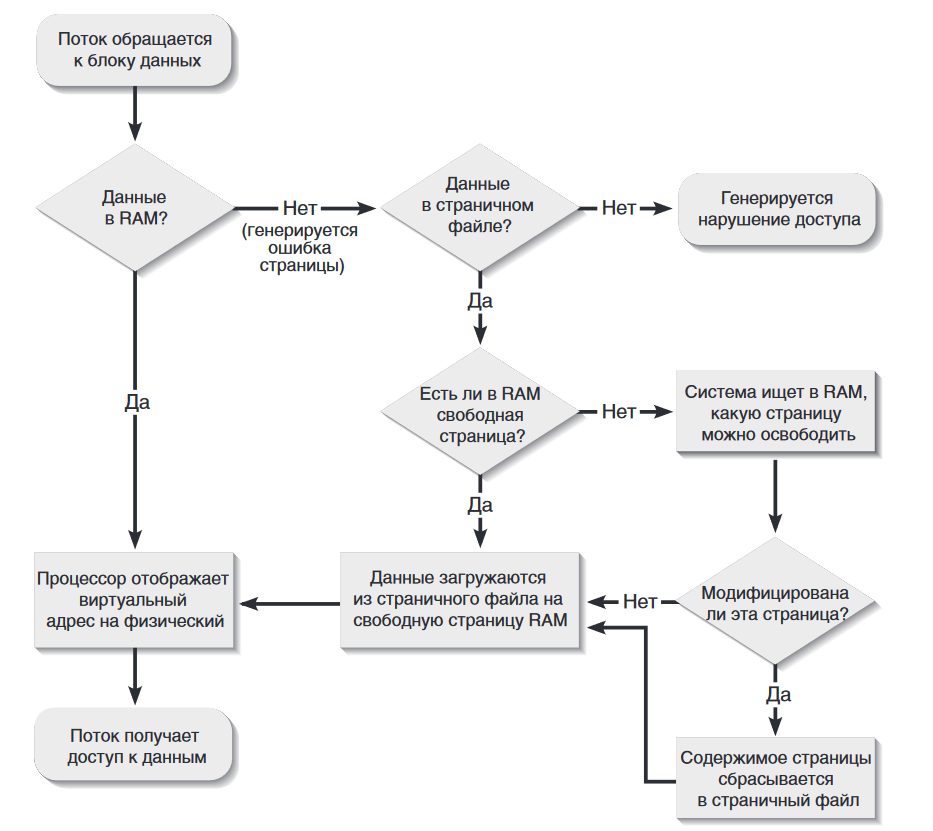
Так как в действительности процесс может работать только с данными в физической памяти, то во время работы процесса необходимо отображать логическую память процесса в физическую память компьютера. Обычно, прямое отображение невозможно по той простой причине, что объем логической памяти процесса превышает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

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



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. Такая организация виртуальной памяти называется **страничной**



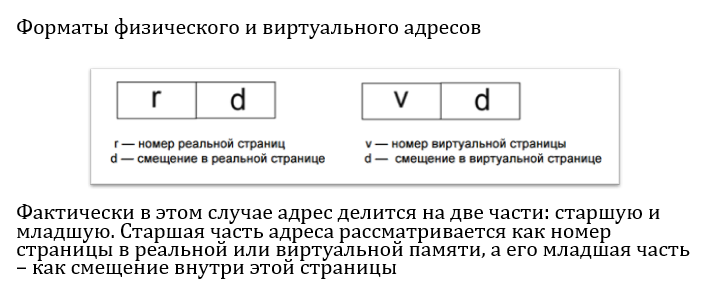
УПРАВЛЕНИЕ ПАМЯТЬЮ

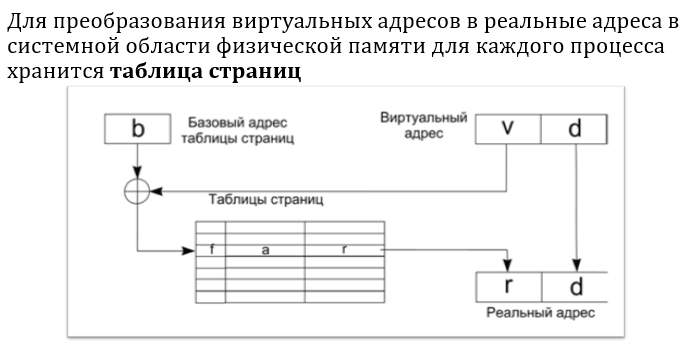
Можно подумать будто при каждом запуске приложения система резервирует регионы адресного пространства для кода и данных процесса, передает им физическую память, а затем копирует код и данные из файла программы (расположенного на жестком диске) в физическую память, переданную из страничного файла

На самом деле происходит вот что: при запуске приложения система открывает его исполняемый файл и определяет объем кода и данных. Затем резервирует регион адресного пространства и помечает, что физическая память, связанная с этим регионом, – сам EXE-файл

Образ исполняемого файла (т. е. EXE- или DLL-файл), размещенный на жестком диске и применяемый как физическая память для того или иного региона адресного пространства, называется **проецируемым в память файлом** (**memory-mapped file**)

При загрузке EXE или DLL система автоматически резервирует регион адресного пространства и проецирует на него образ файла. Помимо этого, система позволяет (с помощью набора функций) проецировать на регион адресного пространства еще и файлы данных. (О проецируемых в память файлах мы поговорим в следующей лекции)

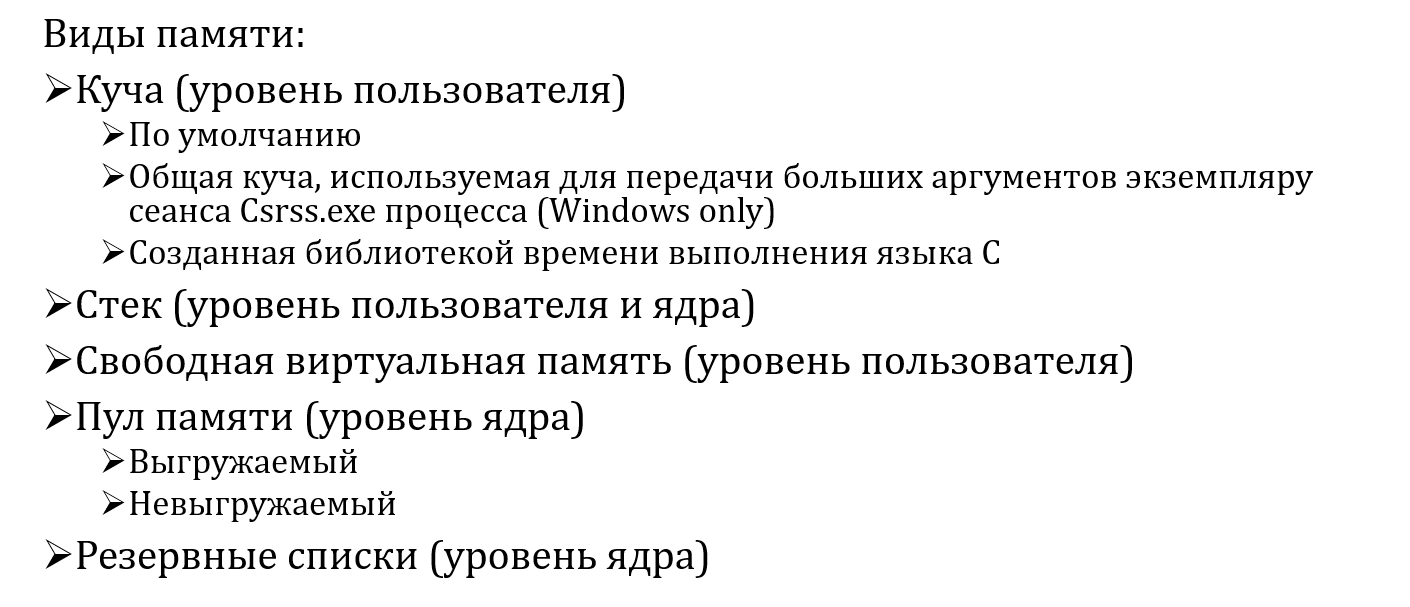




Эмпирически было определено, что при работе многих программ наблюдается свойство локальности. То есть выполняемый в какой-то интервал времени код программы и используемая программой память расположены локально, а не разбросаны по всей программе

Обычно, программисты пишут свои программы, также следуя этому правилу. Поэтому 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**

Существует несколько основных видов памяти, используемых в компьютерных системах.



Вот основные из них:

**1. Оперативная память (RAM)**

* **Описание:** Временная память, используемая для хранения данных и программ, которые активно используются процессором.
* **Типы:**
  + **DRAM (Dynamic RAM):** Использует транзисторы и ёмкости для хранения данных, требует периодического обновления.
  + **SRAM (Static RAM):** Быстрее DRAM, но дороже, используется для кэш-памяти.

**2. Постоянная память (ROM)**

* **Описание:** Неперезаписываемая память, содержащая основные инструкции для загрузки системы.
* **Типы:**
  + **PROM (Programmable ROM):** Может быть записана один раз.
  + **EPROM (Erasable Programmable ROM):** Может быть стерта и перепрограммирована с помощью ультрафиолетового света.
  + **EEPROM (Electrically Erasable Programmable ROM):** Может быть стерта и перепрограммирована электрически.

**3. Виртуальная память**

* **Описание:** Технология, которая позволяет расширять доступную память за счет использования дискового пространства.
* **Назначение:** Обеспечивает иллюзию большего объема RAM и позволяет запускать более крупные приложения.

**4. Кэш-память**

* **Описание:** Быстрая память, расположенная близко к процессору, используемая для хранения часто запрашиваемых данных и инструкций.
* **Уровни:**
  + **L1:** Встроенная в процессор, самая быстрая.
  + **L2:** Внешняя к процессору, немного медленнее.
  + **L3:** Общая кэш-память для нескольких ядер, медленнее L1 и L2.

**5. Флеш-память**

* **Описание:** Ненадежная память, используемая в устройствах хранения данных, таких как SSD, USB-накопители.
* **Типы:**
  + **NAND:** Более распространенный тип, используется в SSD.
  + **NOR:** Используется в некоторых микроконтроллерах.

**6. Жесткий диск и твердотельные накопители (SSD)**

* **Описание:** Устройства долгосрочного хранения данных.
* **Жесткие диски:** Используют магнитные пластины для хранения данных.
* **SSD:** Используют флеш-память для хранения данных, быстрее жестких дисков.

**7. Системная память**

* **Описание:** Общая память, доступная для всех компонентов системы, включая GPU, CPU и другие устройства.

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?

Адресное пространство процесса делится на несколько секций, каждая из которых имеет свое назначение. Вот основные секции и их функции:

**1. Текстовая секция**

* **Назначение:** Содержит исполняемый код программы.
* **Пояснение:** Эта секция является обычно только для чтения и не изменяется во время выполнения.

**2. Секция данных**

* **Назначение:** Хранит глобальные и статические переменные, которые инициализированы.
* **Пояснение:** Делится на две части: инициализированные и неинициализированные данные.

**3. Секция BSS (Block Started by Symbol)**

* **Назначение:** Содержит неинициализированные глобальные и статические переменные.
* **Пояснение:** При загрузке программы эта секция инициализируется нулями.

**4. Стек**

* **Назначение:** Используется для хранения локальных переменных, параметров функций и адресов возврата.
* **Пояснение:** Работает по принципу "последний пришел — первый вышел" (LIFO).

**5. Куча**

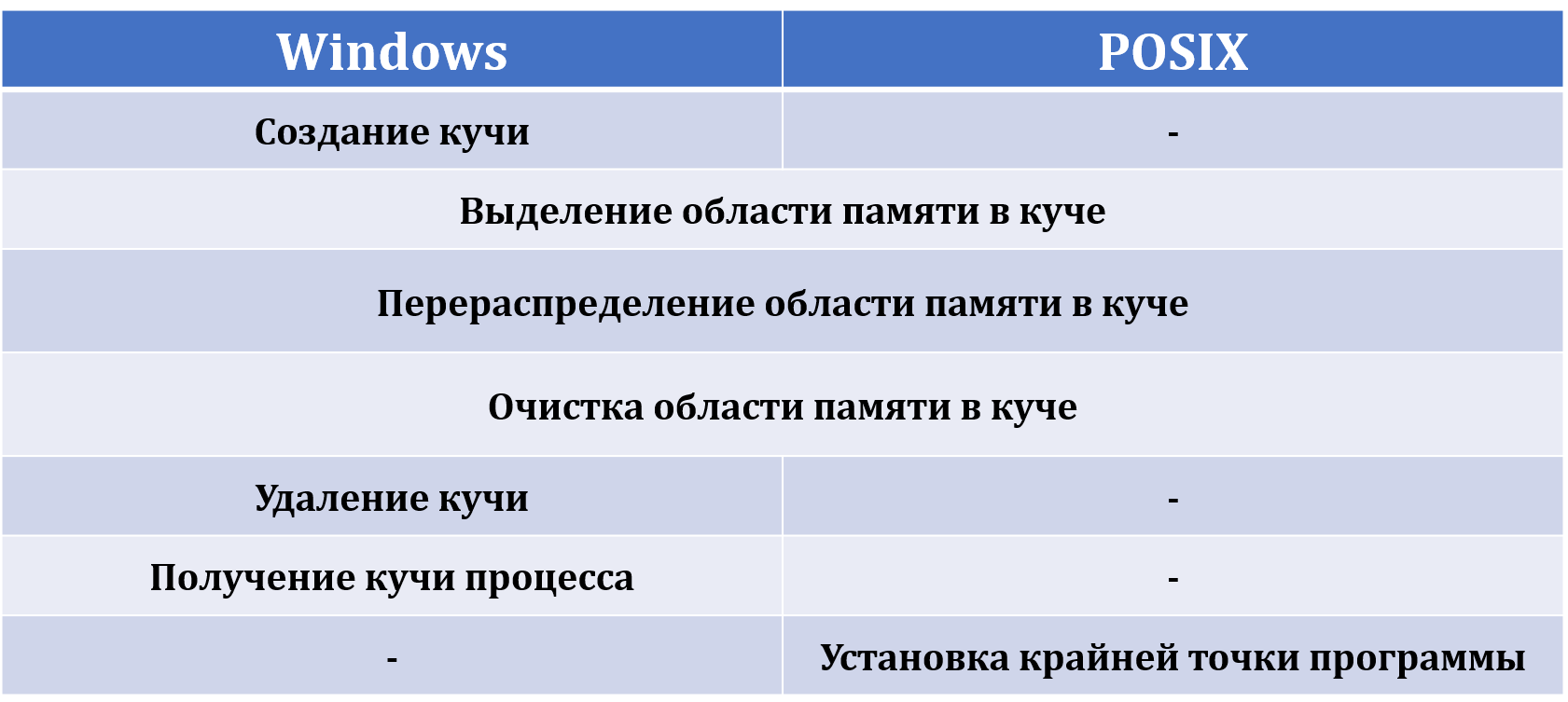
* **Назначение:** Используется для динамического выделения памяти во время выполнения программы.
* **Пояснение:** Позволяет запрашивать и освобождать память по мере необходимости.

**6. Секция окружения**

* **Назначение:** Хранит информацию о среде выполнения, включая аргументы командной строки и переменные окружения.
* **Пояснение:** Позволяет программе получать информацию о своей среде.

**Куча** – это 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 (heap manager), 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**



Когда стоит использовать кучи:

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

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** (**stack**)

Стек является довольно статичной частью адресного пространства, в том смысле, что практически не существует инструментов взаимодействия с ним на пользовательском уровне

Примечание: существует функция **alloca** для выделения памяти в стеке. Однако её использование является **крайне не рекомендованным!**

Советы по работе со стеком:

* 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 и ошибкам «Out-of-memory»

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:

* Куча (уровень пользователя)
  + По умолчанию
  + Общая куча, используемая для передачи больших аргументов экземпляру сеанса Csrss.exe процесса (Windows only)
  + Созданная библиотекой времени выполнения языка С
* Стек (уровень пользователя и ядра)
* Свободная виртуальная память (уровень пользователя)
* Пул памяти (уровень ядра)
  + Выгружаемый
  + Невыгружаемый
* Резервные списки (уровень ядра)

На стадии инициализации системы диспетчер памяти создает два пула памяти, или кучи (heap), 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** (**look-aside lists**). 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:

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

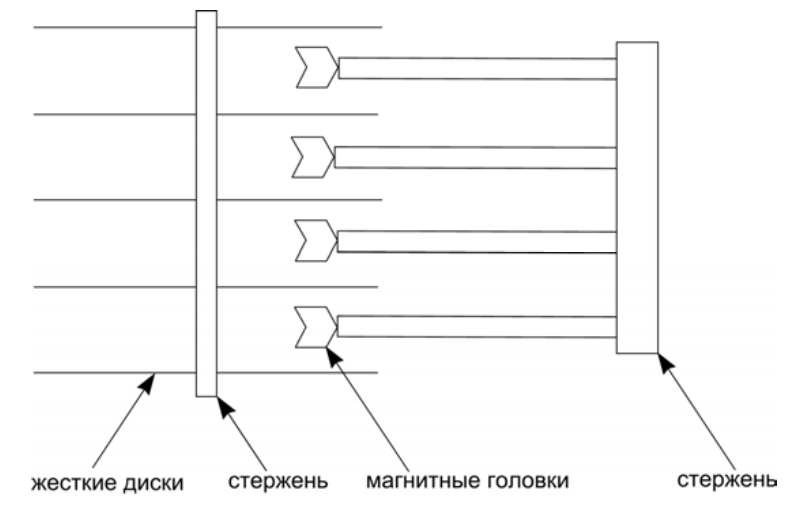
12. Файлы. Хранение данных на магнитном диске (далее – МД): основные понятия. Что такое доступ к данным? Как получить доступ к данным на МД? Виды доступа к данным. Поясните понятия сектора и кластера. Из чего они состоят? Каковы их приблизительные аналоги на твердотельных накопителях? Что такое файл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

Файл — это структурированный набор данных, хранящийся на носителе информации (например, жестком диске). Он используется для организации и хранения различных типов информации, таких как текст, изображения или аудио.

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:

* 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
* 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 (ТВЕРДОТЕЛЬНЫЙ НАКОПИТЕЛЬ) 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 **уровне** **прикладной** программы файл представляет собой множество **логических записей**. 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.

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

Файл на прикладном уровне — это логическая единица хранения данных, используемая в программном обеспечении для организации, обработки и хранения информации. Файлы представляют собой структурированные наборы данных, которые могут использоваться различными приложениями.

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 файла 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 приложений 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** – процесс записи данных в память с более быстрым доступом

**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

Рассматриваемые файловые системы: NTFS для Windows и ext4 для Linux

Примеры ФС: FAT16, FAT32, FAT64, NTFS, ext, ext2, ext3, ext4, BtrFS (Oracle), ISO 9660 (CD, DVD)

**Filesystem Hierarchy Standard (FHS) –** стандарт иерархии файловой системы для ОС семейства Unix ([сайт](https://refspecs.linuxfoundation.org/fhs.shtml))

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 операционных системах 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 (**file view**). 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:

* открыть файл, который будет отображаться в память
* создать объект ядра, который выполняет отображение файла
* отобразить файл или его часть в адресное пространство процесса
* выполнить необходимую работу с представлением файла
* отменить отображение файла
* закрыть объект ядра для отображения файла
* закрыть файл, который отображался в память

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

**Механизм в Windows:**

1. **Создание файла:**
   * Файл создается с помощью CreateFile, который возвращает дескриптор файла.
2. **Создание отображения:**
   * CreateFileMapping создает объект отображения, который связывает файл с адресным пространством процесса. Можно указать параметры, такие как размер отображения и права доступа.
3. **Отображение в память:**
   * MapViewOfFile отображает часть или весь файл в память, возвращая указатель на начало области памяти.
4. **Доступ к данным:**
   * Данные файла доступны через полученный указатель, что позволяет выполнять операции чтения и записи.
5. **Синхронизация и удаление:**
   * Изменения могут быть синхронизированы с файлом на диске при помощи различных функций, таких как FlushViewOfFile. После завершения работы с файлом отображение удаляется с помощью UnmapViewOfFile.

Когда использовать File Mapping:

* Когда вы работаете с большими файлами
* Когда требуется произвольный доступ
* Когда производительность критична
* Когда память ограничена

Когда не использовать File Mapping:

* Когда вы работаете 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)**

**Статическая библиотека**, по сути, 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

Рассматриваемый далее пример будет собираться идентичным образом и на ОС Linux (утилита [**llvm-ar**](https://llvm.org/docs/CommandGuide/llvm-ar.html) может отсутствовать, что решается либо установкой через **sudo apt install llvm**, либо использованием утилиты [**ar**](https://pubs.opengroup.org/onlinepubs/9799919799/utilities/ar.html)), отличия будут приведены в конце примера

Также рассматриваемый пример не будет корректно работать на Linux, в силу особенностей вывода char и wchar\_t друг за другом

**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

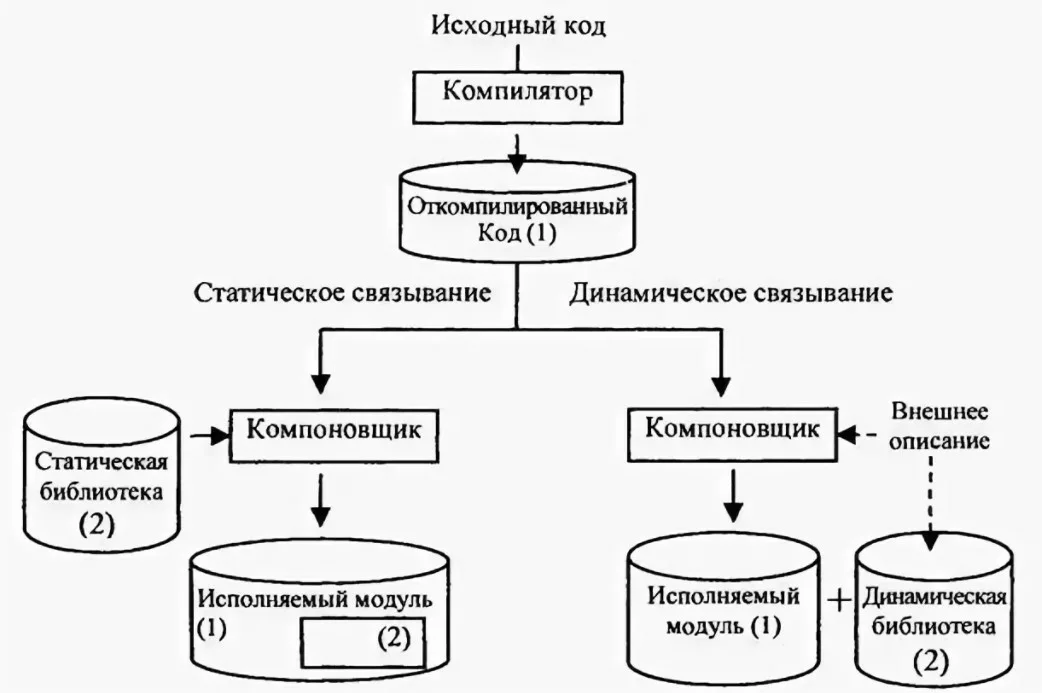
Для устранения представленных недочетов были придуманы **разделяемые** (**динамические**) **библиотеки**

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 (при выполнении)

**Некоторые из причин использования DLL** в ОС Windows:

* Расширение функциональности приложения
* Возможность использования разных языков программирования
* Более простое управление проектом
* Экономия памяти
* Разделение ресурсов
* Упрощение локализации
* Решение проблем, связанных с особенностями различных платформ
* Реализация специфических возможностей

**Процесс сборки приложения с использованием библиотек объектов**



**Виды связывания**

Статическое, оно же – раннее. При нём адреса могут быть вычислены статически

Динамическое, оно же – позднее. Адреса определяются на этапе исполнения в момент связывания



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

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

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**, по сути, 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

Рассматриваемый далее пример будет собираться идентичным образом и на ОС Linux (утилита [**llvm-ar**](https://llvm.org/docs/CommandGuide/llvm-ar.html) может отсутствовать, что решается либо установкой через **sudo apt install llvm**, либо использованием утилиты [**ar**](https://pubs.opengroup.org/onlinepubs/9799919799/utilities/ar.html)), отличия будут приведены в конце примера

Также рассматриваемый пример не будет корректно работать на Linux, в силу особенностей вывода char и wchar\_t друг за другом

**Сборка приложения с использованием статических библиотек с CMake и Clang**

**Шаги для Windows и Linux**

1. **Создание структуры проекта:**

my\_project/

├── CMakeLists.txt

├── main.c

└── mylib.c

1. **Создание CMakeLists.txt:**

cmake\_minimum\_required(VERSION 3.10)

project(MyApp)

# Создание статической библиотеки

add\_library(mylib STATIC mylib.c)

# Создание исполняемого приложения

add\_executable(myapp main.c)

# Связывание библиотеки с приложением

target\_link\_libraries(myapp mylib)

1. **Компиляция с использованием CMake и Clang:**
   * **Windows:**

mkdir build

cd build

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

cmake --build .

* + **Linux:**

mkdir build

cd build

cmake -DCMAKE\_C\_COMPILER=clang ..

cmake --build .

1. **Запуск приложения:**
   * **Windows:**

.\myapp.exe

* + **Linux:**

./myapp

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.

Для устранения представленных недочетов были придуманы **разделяемые** (**динамические**) **библиотеки**

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 (при выполнении)

В операционной системе Windows подобные библиотеки называются **Dynamic Link Library (DLL)**

В операционных системах семейства Linux подобные библиотеки называются **разделяемые объекты** (**shared objects**)

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

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

Некоторые из причин использования DLL в ОС Windows:

* Расширение функциональности приложения
* Возможность использования разных языков программирования
* Более простое управление проектом
* Экономия памяти
* Разделение ресурсов
* Упрощение локализации
* Решение проблем, связанных с особенностями различных платформ
* Реализация специфических возможностей

ХЗ НУЖНО ЛИ:

**Расширение функциональности приложения**

DLL можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код. Поэтому одна компания, создав какое-то приложение, может предусмотреть расширение его функциональности за счет DLL от других компаний

**Возможность использования разных языков программирования**

У Вас есть выбор, на каком языке писать ту или иную часть приложения. Так, пользовательский интерфейс приложения Вы скорее всего будете создавать на C#, но прикладную логику лучше всего реализовать на C++. Программа на C# может загружать DLL, написанные на C++, Коболе, Фортране и др.

**Более простое управление проектом**

Если в процессе разработки программного продукта отдельные его модули создаются разными группами, то при использовании DLL таким проектом управлять гораздо проще. Однако конечная версия приложения должна включать как можно меньше файлов

**Экономия памяти**

Если одну и ту же DLL использует несколько приложений, в оперативной памяти может храниться только один ее экземпляр, доступный этим приложениям.

**Разделение ресурсов**

DLL могут содержать такие ресурсы, как шаблоны диалоговых окон, строки, значки и битовые карты (растровые изображения). Эти ресурсы доступны любым программам

**Решение проблем, связанных с особенностями различных платформ**

В разных версиях Windows содержатся разные наборы функций. Зачастую разработчикам нужны новые функции, существующие в той версии системы, которой они пользуются. Если Ваша версия Windows не поддерживает эти функции, Вам не удастся запустить такое приложение: загрузчик попросту откажется его запускать. Но если эти функции будут находиться в отдельной DLL, Вы загрузите программу даже в более ранних версиях Windows, хотя воспользоваться ими Вы все равно не сможете

**Упрощение локализации**

DLL нередко применяются для локализации приложений. Например, приложение, содержащее только код без всяких компонентов пользовательского интерфейса, может загружать DLL с компонентами локализованного интерфейса

**Реализация специфических возможностей**

Определенная функциональность в Windows доступна только при использовании DLL. Например, отдельные виды ловушек (устанавливаемых вызовом SetWindowsHookEx и SetWinEventHook) можно задействовать при том условии, что функция уведомления ловушки размещена в DLL. Кроме того, расширение функциональности оболочки Windows возможно лишь за счет создания COM-объектов, существование которых допустимо только в DLL.

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

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

**Шаги для Windows и Linux**

1. **Создание структуры проекта:**

my\_project/

├── CMakeLists.txt

├── main.c

└── mylib.c

1. **Создание CMakeLists.txt:**

cmake

cmake\_minimum\_required(VERSION 3.10)

project(MyApp)

# Создание динамической библиотеки

add\_library(mylib SHARED mylib.c)

# Создание исполняемого приложения

add\_executable(myapp main.c)

# Связывание библиотеки с приложением

target\_link\_libraries(myapp mylib)

1. **Компиляция с использованием CMake и Clang:**
   * **Windows:**

mkdir build

cd build

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

cmake --build .

* + **Linux:**

mkdir build

cd build

cmake -DCMAKE\_C\_COMPILER=clang ..

cmake --build .

1. **Запуск приложения:**
   * **Windows:**

.\myapp.exe

* + **Linux:**

./myapp

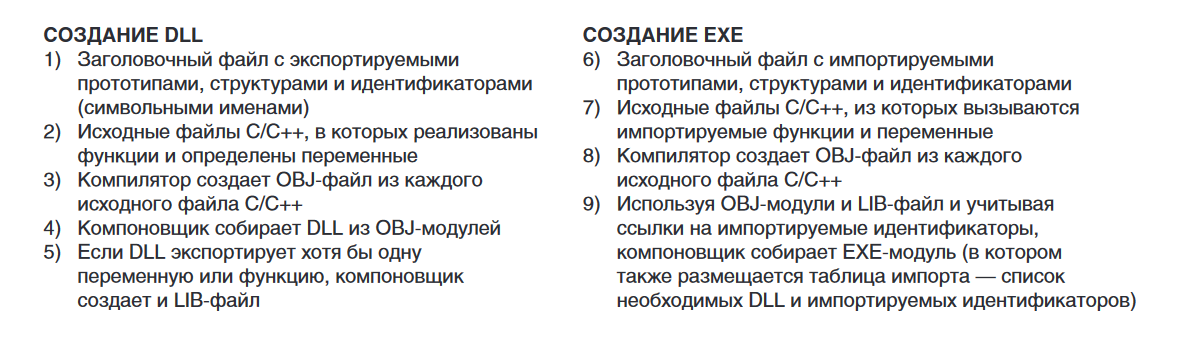
**Механизм ОС для работы с динамическими библиотеками**

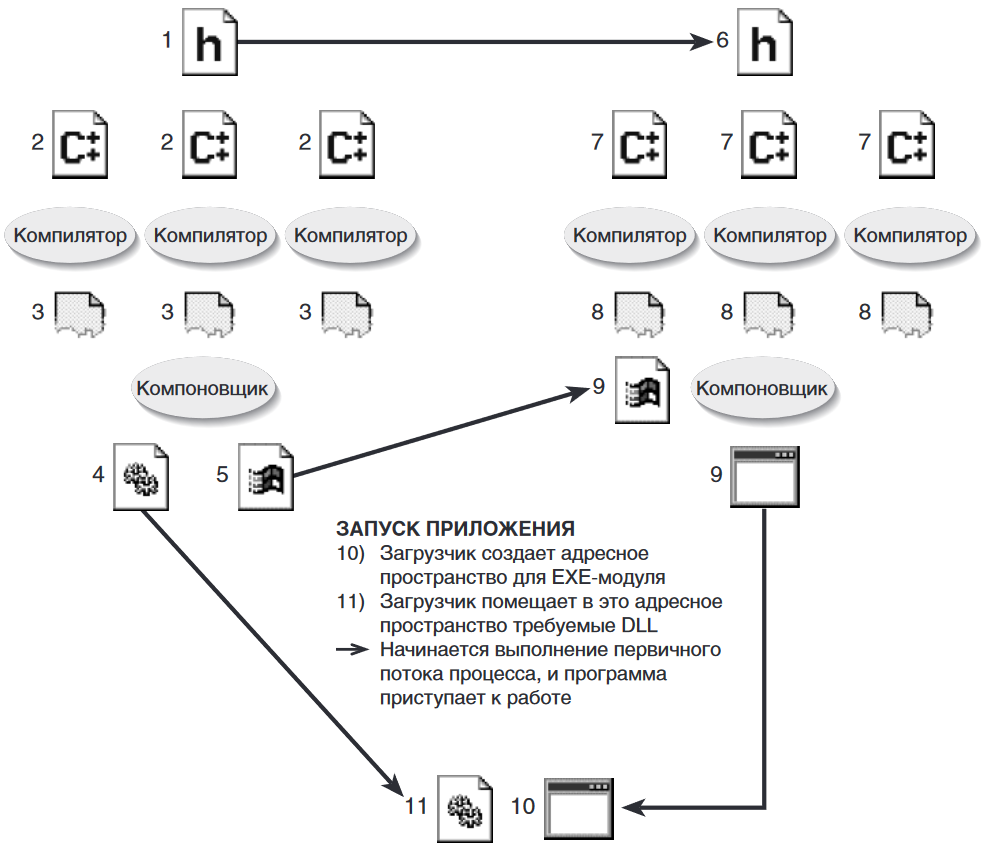
* **Windows:**
  + Использует механизм Dynamic Link Library (DLL). При загрузке приложения операционная система загружает необходимые DLL в память и создает таблицу импорта для управления вызовами функций.
* **Linux:**
  + Использует механизм Shared Object (SO). При запуске программы динамическая библиотека загружается в память, и система использует динамическую линковку для связывания функций во время выполнения.

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)**

Рассмотренный способ подключения DLL можно подытожить следующим способом(алгоритм работы):





**Определение:**  
Неявный способ подключения подразумевает, что динамическая библиотека (DLL) связывается с исполняемым файлом во время компиляции, но фактическая загрузка происходит во время выполнения.

**Связывание, лежащее в основе неявного подключения**

Неявное связывание основано на **статическом связывании**. Во время компиляции создается файл импорта, который содержит информацию о функциях DLL, что позволяет компилятору знать, какие функции будут использоваться.

Отложенное декларативное связывание

**\_\_declspec(dllexport)**

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

**Способы экспорта функций и их реализация**

1. **С помощью директивы \_\_declspec(dllexport) (Windows):**
   * В исходном коде DLL функции, которые нужно экспортировать, помечаются следующим образом:

\_\_declspec(dllexport) void MyFunction();

1. **С использованием файла определения (DEF-файла):**
   * Создается DEF-файл, в котором перечислены все экспортируемые функции. Например:

LIBRARY MyLibrary

EXPORTS

MyFunction

AnotherFunction

* + При компиляции DLL этот файл используется для генерации раздела экспорта.

1. **С помощью \_\_declspec(dllimport) (Windows):**
   * Для использования экспортируемых функций в других модулях применяют \_\_declspec(dllimport):

\_\_declspec(dllimport) void MyFunction();

* [**\_\_declspec(dllimport)**](https://learn.microsoft.com/en-us/cpp/cpp/dllexport-dllimport?view=msvc-170)– такой модификатор означает, что данная переменная или функция импортируются из DLL (указывается с исполняемых модулях или библиотеках, которые зависят от других библиотек)
* [**\_\_declspec(dllexport)**](https://learn.microsoft.com/en-us/cpp/cpp/dllexport-dllimport?view=msvc-170)– такой модификатор означает, что данная переменная или функция экспортируется из DLL (указывается в самой библиотеке)
* **\_\_declspec(dllexport)**
* Если он указан перед переменной, прототипом функции или C++-классом, компилятор Microsoft С/С++ встраивает в конечный OBJ-файл дополнительную информацию. Она понадобится компоновщику при сборке DLL из OBJ-файлов.
* Как можно заметить из таблицы экспорта, у каждой функции есть **порядковый номер** ([**ordinal**](https://learn.microsoft.com/en-us/cpp/build/exporting-functions-from-a-dll-by-ordinal-rather-than-by-name?view=msvc-170)) и **имя** (**name**)
* Соответственно существуют два способа экспорта функций по номеру и по имени
* Чтобы экспортировать по имени достаточно применить модификатор **\_\_declspec(dllexport)**
* 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

Что значит «Он позволяет указать конечное имя для функции в таблице экспорта»?

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

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

Как уже говорилось ранее, рассмотренные примеры использовали **неявную загрузку**, оно же **отложенное декларативное связывание.**

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?

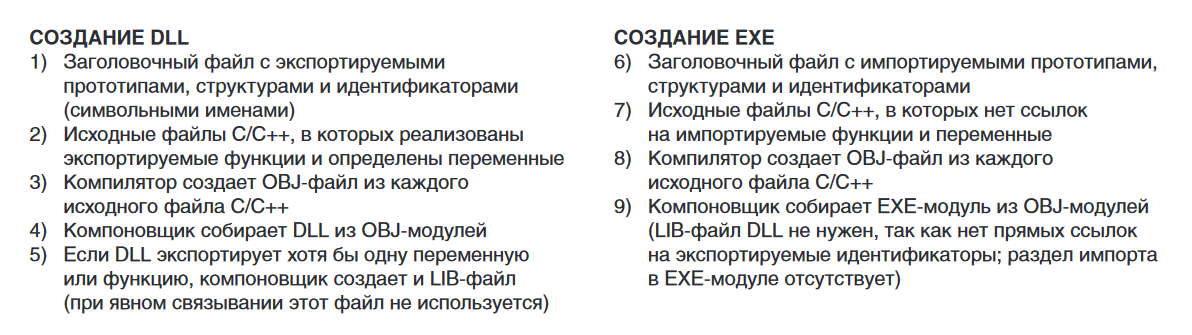
Второй способ который стоит рассмотреть – **явная загрузка** или **отложенное императивное связывание** требуемой DLL в период выполнения приложения. Иначе говоря, его поток явно загружает DLL в адресное пространство процесса, получает виртуальный адрес необходимой DLL-функции и вызывает ее по этому адресу. Изящество такого подхода в том, что все происходит в уже выполняемом приложении

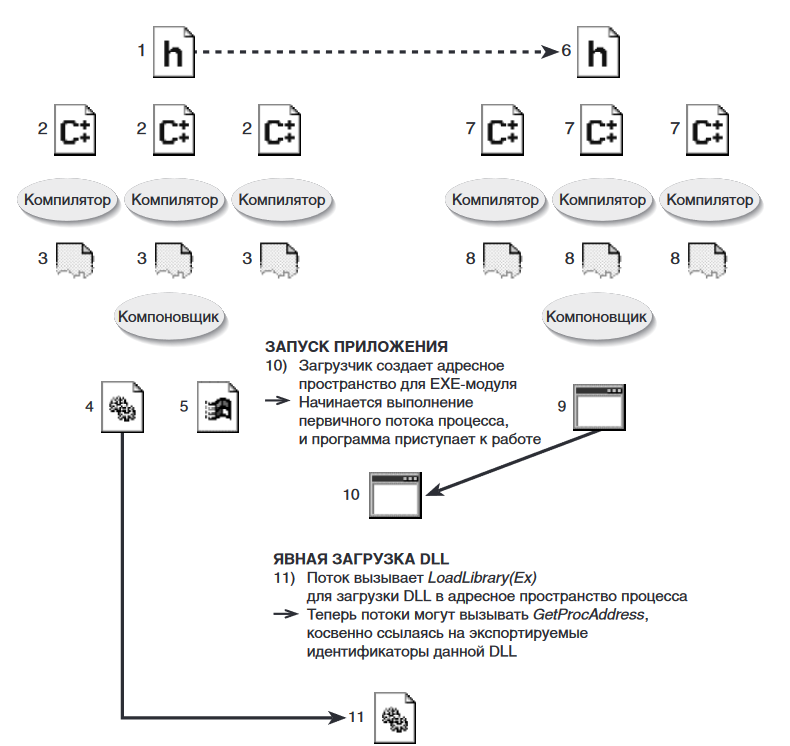
Как можно заметить в отличие от прошлого варианта изменилось следующее:

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

Как можно заметить в отличие от прошлого варианта изменилось следующее:

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





Что значит «Он позволяет указать конечное имя для функции в таблице экспорта»?

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

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

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

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

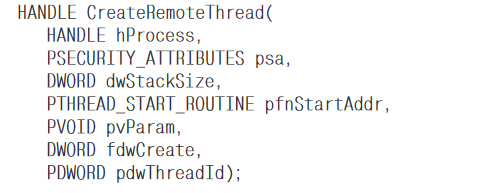
В Linux в SO такой функции быть не может, однако вместо этого можно определить одну или несколько функций, которые будут автоматически вызываться при загрузке и выгрузке разделяемой библиотеки

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

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

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

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



Последовательность операций, которые Вам надо будет выполнить:

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

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

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

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

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

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

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

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?

При разработке повторно используемого программного обеспечения, системный программист берет уже существующую или предлагает новую систему соглашений, которой оно должно соответствовать. Соглашения могут быть оформлены в виде спецификаций или корпоративных стандартов. В этих документах, как правило, оговариваются принципы именования объектов (имена функций, параметров, переменных), структуры и типы используемых данных, система интерфейсов (группы функций, классифицированных по каким-то признакам) и т. д.

Примером такой спецификации может служить COM (Component Object Model – объектная модель компоненты) компании Microsoft.

**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++

2 свойства

**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

**Двоичный стандарт (или независимость от языка программирования)**

Многоразовое использование программного обеспечения является одной из первоочередных задач при его разработке и обеспечивается составляющими его модулями, которые должны работать в разнообразных средах. Обычно программное обеспечение разрабатывается с использованием определенного языка программирования, например C++, и может эффективно применяться только в том случае, если другие разработчики компонентов также применяют C++

Применение двоичного кода позволяет разработчику создавать программные компоненты, которые могут применяться без использования языков, средств и систем программирования, а только с помощью ***двоичных компонентов*** (например, DLL- или ЕХЕ- файлов)

**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)

**Спецификация COM** – это технологический стандарт компании Microsoft, первая версия которого разработана ещё в 1993 году. Стандарт предназначен для создания программного обеспечения на основе взаимодействующих компонентов. Последняя версия данного стандарта была опубликована в 2021 году

**COM-программирование** –разработка программного обеспечения, имеющего модель согласно спецификации COM

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**

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

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

**Принципы взаимодействия клиента и сервера**

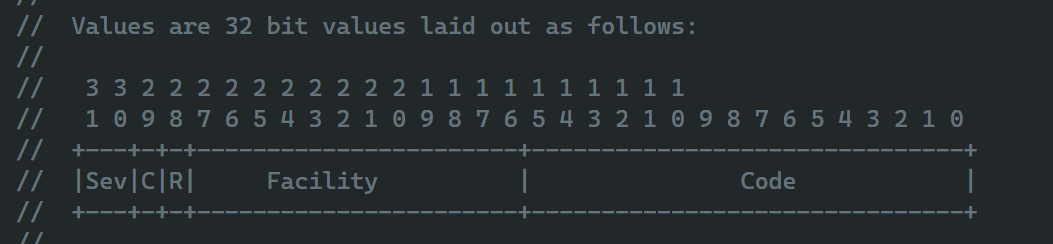
Поддержка программ соответствующих COM-модели в операционной системе Windows обеспечивается с помощью динамически подключаемой библиотеки **OLE32.DLL** и соответствующей ей библиотеки экспорта функций **OLE32.LIB**

**Принципы взаимодействия клиента и сервера**

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

**Принципы взаимодействия клиента и сервера**

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

Существует два основных типа серверов: **in-process (в процессе) и out-of-process (вне процесса)**

Серверы in-process реализуются в динамической библиотеке (DLL), а серверы out-of-process реализуются в исполняем файле (EXE)

Серверы out-of-process могут размещаться либо на локальном компьютере, либо на удаленном компьютере

Кроме того, COM предоставляет механизм, который позволяет серверу in-process (DLL) запускаться в суррогатном процессе EXE, чтобы получить преимущество выполнения процесса на удаленном компьютере

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

В таком случае между клиентом и сервером появляется прослойка в виде прокси-объекта

Для создания таких объектов зачастую применяется Microsoft Interface Definition Language (MIDL) – компилятор IDL файлов

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

Рассмотрим создание однокомпонентного COM-сервера

1. Описать COM-интерфейс который обязан наследоваться хотя бы от **IUnknown** с помощью языка IDL и скомпилировать его используя MIDL

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

**COM (Component Object Model)** — это архитектура, разработанная Microsoft для создания программного обеспечения, позволяющая различным компонентам взаимодействовать друг с другом, независимо от языка программирования.

**Интерфейс IUnknown**

Интерфейс IUnknown является базовым для всех COM-объектов. Он определяет три ключевых метода:

Кроме ваших методов, интерфейс будущего 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

AddRef

Назначение: Увеличивает счетчик ссылок на объект.

Использование: Каждый раз, когда создается ссылка на объект, вызывается этот метод, чтобы указать, что объект активен и не должен быть уничтожен.

Release

Назначение: Уменьшает счетчик ссылок на объект.

Использование: Когда ссылка на объект больше не нужна, вызывается этот метод, чтобы уменьшить счетчик. Если счетчик становится равным нулю, объект уничтожается.

QueryInterface

Назначение: Позволяет запрашивать другие интерфейсы, реализуемые объектом.

Использование: Принимает идентификатор интерфейса и возвращает указатель на него, если объект поддерживает этот интерфейс.

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

**Соглашение о вызове и возврате**

**Соглашение о вызове (Calling Convention):**

* COM-методы должны использовать стандартное соглашение о вызове stdcall. Это соглашение определяет порядок передачи аргументов и место возврата значения.

**Исключения:**

* Методы, которые не следуют этому соглашению, должны быть явно обозначены и использовать другие соглашения, если это необходимо.

**HRESULT**

**Определение:** HRESULT — это 32-битный код, который указывает результат выполнения метода COM. Он используется для передачи информации о статусе выполнения, включая ошибки.

**Структура HRESULT:**

* **Бит 31:** Указывает на успешное или неуспешное выполнение (0 — успех, 1 — ошибка).
* **Биты 30-16:** Зарезервированы для кодов состояния.
* **Биты 15-0:** Уникальный код ошибки или состояние.

**Назначение:**

* Позволяет разработчикам легко определять, было ли выполнение метода успешным или нет, а также получать информацию об ошибках.

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

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

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.

3. Реализовать фабрику классов путём реализации стандартного COM-интерфейса IClassFactory для удобного управления жизненным циклом COM-компонентов

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

COM-интерфейс 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

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

. Скомпилировать COM-сервер

**Фабрика классов** — это специальный объект, который отвечает за создание экземпляров других компонентов (объектов). Она используется в COM для инкапсуляции процесса создания объектов, что позволяет клиентам запрашивать экземпляры без необходимости знать детали их реализации.

**Назначение фабрики классов**

* Позволяет динамически создавать объекты на основе их классов.
* Обеспечивает управление временем жизни создаваемых объектов.
* Упрощает использование объектов, предоставляя единую точку для их создания.

**Интерфейс IClassFactory**

Интерфейс IClassFactory определяет два основных метода:

1. **CreateInstance**
   * **Назначение:** Создает новый экземпляр COM-объекта.
   * **Параметры:** Принимает указатель на интерфейс, который клиент хочет получить, и возвращает указатель на этот интерфейс, если объект успешно создан.
   * **Использование:** Клиент вызывает этот метод, чтобы получить новый экземпляр запрашиваемого объекта.
2. **LockServer**
   * **Назначение:** Увеличивает или уменьшает счетчик экземпляров компонента.
   * **Параметры:** Принимает логическое значение, указывающее, нужно ли блокировать сервер (true) или разблокировать (false).
   * **Использование:** Этот метод используется для управления временем жизни сервера компонентов, предотвращая его уничтожение, пока существуют активные экземпляры.

**Счетчик экземпляров компонентов**

**Определение:** Счетчик экземпляров — это целое число, отражающее количество активных экземпляров объектов, созданных через фабрику классов.

**Назначение:**

* Позволяет управлять временем жизни COM-сервера. Когда счетчик увеличивается, сервер не может быть уничтожен, пока существуют активные экземпляры.

**Увеличение и уменьшение счетчика:**

* **Увеличение:** Счетчик экземпляров увеличивается при каждом вызове метода CreateInstance, который успешно создает новый объект.
* **Уменьшение:** Счетчик уменьшается, когда вызывается метод Release для созданного объекта. Если счетчик экземпляров становится равным нулю, это может привести к уничтожению сервера.

**Условие, при котором объект компонента удаляется**

Объект компонента удаляется, когда:

* Счетчик ссылок на объект (от метода Release интерфейса IUnknown) становится равным нулю.
* Это означает, что на объект больше нет активных ссылок, и он может быть уничтожен, освобождая занятые ресурсы.

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.

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

6. Разработать COM-клиент:

* Работа клиента должна начинаться с инициализации библиотеки OLE32.DLL (вызов функции CoInitialize)
* Для создания экземпляра компонента необходимо вызвать функцию CoCreanteInstance
* Для получения указателя на другие интерфейсы можно применить метод QueryInterface стандартного интерфейса IUnknown
* Перед своим завершением COM-клиент обязан корректно закончить работу с библиотекой OLE32.DLL. Для этого он должен выполнить функцию CoFreeUnusedLibrary
* Работа клиента должна завершаться освобождением библиотеки OLE32.DLL (вызов функции CoUninitialize)

Жизненный цикл COM-сервера:

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

**Процесс регистрации COM-объекта**

Процесс регистрации COM-объекта включает в себя создание записей в системном реестре Windows, которые позволяют системе и клиентским приложениям находить и использовать этот объект. Регистрация необходима для того, чтобы клиент мог запросить экземпляр COM-объекта и использовать его методы.

**Этапы регистрации:**

1. **Создание записи в реестре:**
   * При регистрации COM-объекта создаются записи, указывающие на его класс, CLSID (идентификатор класса), интерфейсы и местоположение файла библиотеки (DLL или EXE), содержащей объект.
2. **Настройка информации о типе:**
   * Указываются типы интерфейсов, которые поддерживает объект, и информация о том, как создавать экземпляры.
3. **Настройка прав доступа:**
   * Устанавливаются разрешения на доступ к COM-объекту, что позволяет или ограничивает доступ для различных пользователей или приложений.

**Утилита regsvr32**

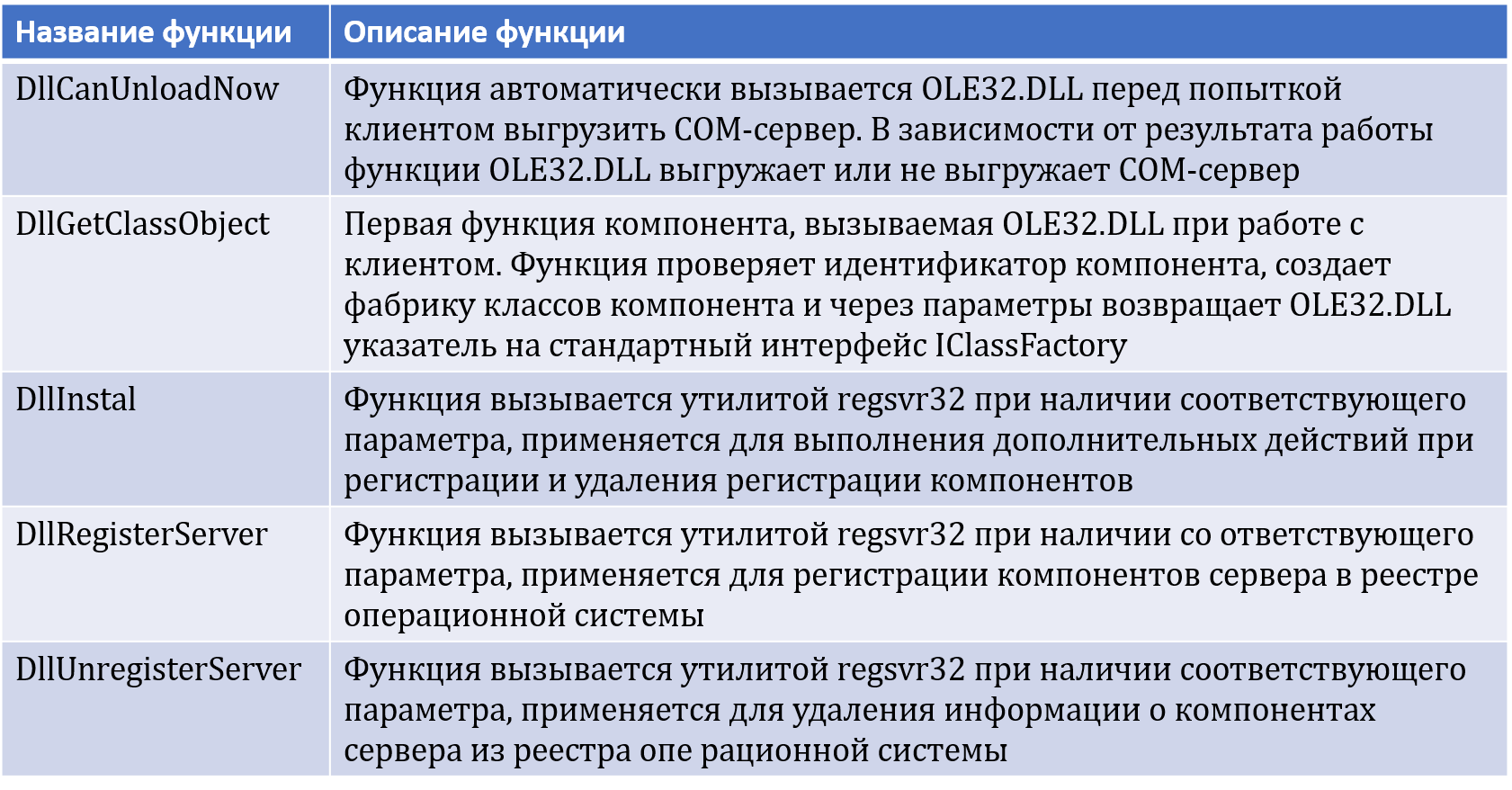
**Назначение:**  
regsvr32 — это утилита командной строки Windows, предназначенная для регистрации и отмены регистрации COM-библиотек (DLL) и ActiveX-контролей.

**Принцип работы:**

* Когда вызывается regsvr32, она загружает указанную DLL и вызывает функцию DllRegisterServer, которая отвечает за создание необходимых записей в реестре.
* Аналогично, для отмены регистрации вызывается функция DllUnregisterServer, которая удаляет соответствующие записи.

**Функции, экспортируемые COM/DLL-контейнером**

Вот пять основных функций, которые экспортируются COM/DLL-контейнером:



1. **DllMain**
   * **Назначение:** Основная точка входа для DLL, вызываемая системой при загрузке или выгрузке библиотеки.
   * **Использование:** Позволяет выполнять инициализацию и очистку при загрузке или выгрузке библиотеки.
2. **DllRegisterServer**
   * **Назначение:** Регистрация COM-объекта в системе.
   * **Использование:** Создает записи в реестре для CLSID, интерфейсов и местоположения DLL.
3. **DllUnregisterServer**
   * **Назначение:** Отмена регистрации COM-объекта.
   * **Использование:** Удаляет записи из реестра, связанные с COM-объектом.
4. **DllGetClassObject**
   * **Назначение:** Получение указателя на фабрику классов для создания экземпляров объектов.
   * **Использование:** Позволяет клиентам запрашивать фабрику классов, чтобы создать экземпляр COM-объекта, используя его CLSID.
5. **DllCanUnloadNow**
   * **Назначение:** Определяет, может ли DLL быть выгружена из памяти.
   * **Использование:** Проверяет, есть ли активные ссылки на объекты, и если их нет, DLL может быть выгружена для освобождения ресурсов.

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. 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 операционной системы будет сервисом

Виды сервисов: 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:**

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

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

Функции, которые выполняет менеджер сервисов:

* Поддержка базы данных установленных сервисов
* Запуск сервисов при загрузке операционной системы
* Поддержка информации о состоянии работающих сервисов
* Передача управляющих запросов работающим сервисам

Блокировка и разблокирование базы данных сервисов

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

**Да, в его голове – это структура сервиса:**

Кроме того, каждый сервис должен содержать **две функции обратного вызова**, которые вызываются операционной системой.

Одна из этих функций определяет точку входа сервиса, т. е., собственно, и является сервисом (**void WINAPI ServiceMain**), а вторая – должна реагировать на управляющие сигналы от операционной системы (**void WINAPI ServiceCtrlHandler**).

Диспетчер сервиса получает управляющие сигналы от менеджера сервисов по именованному каналу и передает эти запросы функции **ServiceCtrlHandler**, которая обрабатывает эти управляющие запросы

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

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 (англ. daemon) – это процесс, обладающий следующими свойствами:

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

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)

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)

Этот шаг делается по двум следующим причинам:

* Исходя из того, что демон был запущен в командной строке, завершение родителя будет обнаружено командной оболочкой, которая вслед за этим выведет новое приглашение и позволит потомку выполняться в фоновом режиме
* Потомок гарантированно не станет лидером группы процессов, поскольку он наследует PGID от своего родителя и получает свой уникальный идентификатор, который отличается от унаследованного PGID. Это необходимо для успешного выполнения следующего шага.
* 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
* Это можно сделать двумя способами:
* Указывать флаг O\_NOCTTY для любых вызовов **open**, которые могут открыть терминальное устройство.
* Есть более простой вариант: после **setsid** можно еще раз сделать вызов **fork**, опять позволив родителю завершиться, а потомку (правнуку) – продолжить работу.
* 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)

Рекомендации:

* Как уже отмечалось выше, процесс-демон обычно завершается во время выключения системы. 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, нам 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. Если попытаться запустить другой экземпляр того же демона, то он не сможет получить блокировку для соответствующего файла и автоматически завершится, понимая: один его экземпляр уже выполняется в системе.

Конфигурацию принято хранить в каталоге **/etc** или в одном из его подкаталогов, а журнал часто находится в **/var/log**

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 операционной системы

В рамках подсистемы Linux для Windows используется более старый вариант – **init.**

Добавление демона на уровне системы с помощью Initd (auto\_daemon.c в примерах):

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 ему права через **chmod** на **755**)
4. (Опционально) Создаём каталог **/etc/<имя демона>** в котором располагаем конфигурационные файлы которые будут использоваться самим демоном
5. (Справочно) Файлы логов обычно располагают в каталоге **/var/log**
6. Проверяем с помощью команды **service**, что система распознала скрипт инициализации демона
7. Если скрипт написан корректно, то будет доступно управление демоном через команду **service**
8. Для автозапуска вместе с системой надо перейти в каталог **/etc/init.d** и выполнить следующую команду:



Добавление демона на уровне системы с помощью Systemd:

1. Компилируем приложение будущего демона
2. Располагаем получившийся бинарный файл в одной из папок **/sbin** или **/usr/sbin**
3. Располагаем файл описания **<имя демона>.service** в каталоге **/etc/systemd/system**
4. (Опционально) Создаём каталог **/etc/<имя демона>** в котором располагаем конфигурационные файлы которые будут использоваться самим демоном
5. (Справочно) Файлы логов обычно располагают в каталоге **/var/log**
6. Проверяем с помощью команды **systemctl**, что система распознала описание демона
7. Если файл с описанием написан корректно, то будет доступно управление демоном через команду **systemctl**
8. Для автозапуска демона вместе с системой systemctl имеет следующую команду



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 операционной системы, 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 программное обеспечение которое предоставляет другому программному обеспечению API для работы с аппаратными устройствами.

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

* способ работы с драйверами как файлами
* драйвер, как легко заменяемая часть ОС
* существование режима ядра

Способ работы с драйверами как файлами означает, что функции, используемые при взаимодействии с файлами, практически идентичны таковым при взаимодействии с драйверами (имеется в виду лексически): open, close, read и т. д

Подсистема ввода/вывода в Windows состоит из набора компонентов исполнительной системы, которые совместно управляют устройствами и предоставляют приложениям и системе интерфейсы к этим устройства.

Подсистема ввода/вывода в Windows проектировалась как абстрактный интерфейс приложений для аппаратных (физических) и программных (виртуальных, или логических) устройств, обладающий определенными функциональными возможностями:

* Унифицированные средства безопасности и именования устройств для защиты общих ресурсов
* Высокопроизводительный асинхронный пакетный ввод/вывод служит для поддержки масштабируемых приложений
* Специальные службы позволяют писать драйверы устройств на высокоуровневом языке и упрощают их перенос на машины с другой архитектурой
* 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

Центральное место в подсистеме ввод/вывода занимает

**Диспетчер ввода/вывода** (I/O Manager): Основной компонент, который управляет взаимодействием между приложениями и устройствами.

- **Драйверы устройств** (Device Drivers): Программные модули, которые обеспечивают интерфейс для конкретных типов устройств.

- **Объекты устройств** (DEVICE\_OBJECT): Структуры данных, представляющие устройства в системе.

- **Объекты драйвера** (DRIVER\_OBJECT): Структуры, которые содержат информацию о драйверах и их процедурах.

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?

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.

**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.

**Уровень аппаратных абстракций** (Hardware Abstraction Layer, 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.

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** (I/O Request Packets, 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.

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

**Уровни запросов прерываний**, или **IRQL.**

Термин «IRQL» имеет два разных значения, которые сходятся в некоторых ситуациях:

* 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

Фундаментальное правило IRQL гласит, что код с более низким IRQL не может вмешиваться в работу кода с более высоким IRQL, и наоборот – код с более высоким IRQL не может вытеснять код, работающий с более низким IRQL.

Самые важные уровни IRQL в контексте ввода/вывода:

* **Passive** (**0**). Определяется макросом PASSIVE\_LEVEL в заголовочном файле WDK wdm.h. Это нормальный уровень IRQL, при котором планировщик ядра работает нормально
* **Dispatch/DPC** (**2**) (DISPATCH\_LEVEL). Уровень IRQL, на котором работает планировщик ядра. Это означает, что если поток поднимает текущий уровень IRQL до 2 (или выше), поток фактически получает бесконечный квант и не может быть вытеснен другим потоком. По сути, планировщик не может активизироваться на текущем процессоре, пока уровень IRQL не упадет ниже 2
* **Device IRQL** (3-26 на х86; 3-12 на х64 и ARM) (DIRQL). Эти уровни закрепляются за аппаратными преобразованиями. При поступлении прерывания диспетчер вызывает соответствующую функцию обслуживания прерывания (ISR, Interrupt Service Routine) и повышает ее уровень IRQL до уровня соответствующего прерывания.

**Отложенный вызов процедуры**, или **DPC** (Deferred Procedure Call), – 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.

Краткая (да ну нахуй краткая) сводка последовательности событий:

1. Некий код пользовательского режима или режима ядра выполняется тогда, когда процессор находится на уровне 0 (на этом уровне проходит большая часть времени)
2. Поступает аппаратное прерывание на уровне IRQL 5. Так как 5 больше 0 (текущий уровень IRQL) состояние процессора сохраняется, IRQL повышает до 5, и вызывается обработчик ISR, связанный с прерыванием

Обратите внимание: переключение контекста при этом не происходит; работает тот же поток, который теперь выполняет код ISR. (Если поток находился в пользовательском режиме, он переключается в режим ядра при поступлении прерывания

1. ISR 1 начинает выполняться, когда процессор работает на уровне IRQL 5. На этот момент любые прерывания с IRQL 5 и ниже вмешиваться в обработку не могут
2. Предположим, поступает новое прерывание с IRQL 8. Предположим, система решает, что оно должно быть обработано тем же процессором. Так как 8>5, выполнение снова прерывается, состояние процессора сохраняется, IRQL повышается до 8, и процессор переходит к ISR 2. Заметьте также, что выполнение происходит в том же потоке. Никакое переключение контекста при этом невозможно, потому что планировщик потоков не может активизироваться при IRQL уровня 2 и выше
3. Выполняется обработчик ISR 2. До его завершения ISR 2 хотелось бы выполнить дополнительную обработку на более низком уровне IRQL, чтобы прерывания с IRQL менее 8 тоже могли быть обработаны
4. Напоследок ISR 2 вставляет правильно инициализированный объект DPC со ссылкой на функцию драйвера, которая выполняет всю последующую обработку после закрытия прерывания, вызовом функции KelnsertQueueDpc. Затем ISR возвращает управление, и восстанавливается состояние процессора, сохраненное перед входом в ISR 2
5. На этой стадии IRQL падает до предыдущего уровня (5), и процессор продолжает выполнение обработчика ISR 1, прерванного ранее
6. Непосредственно перед завершением ISR 1 ставит в очередь собственный объект DPC для выполнения своей последующей обработки. Объекты DPC ставятся в очередь DPC. ISR 1 возвращает управление, восстанавливая состояние процессора, сохраненное перед началом выполнения ISR 1
7. В этот момент уровень IRQL должен был бы упасть до старого нулевого значения, чтобы начать обработку прерываний. Но ядро замечает наличие необработанных DPC, поэтому IRQL уменьшается до уровня 2 (DPC\_LEVEL), и запускается цикл обработки DPC, который перебирает накопленные DPC и последовательно вызывает каждую процедуру DPC. Когда очередь DPC опустеет, обработка DPC завершается
8. Наконец, IRQL уменьшается до 0, состояние процессора снова восстанавливается, и возобновляется выполнение изначально прерванного исходного кода пользовательского режима или режима ядра. И снова следует напомнить, что вся описанная обработка происходит в одном потоке. Этот факт подразумевает, что ISR и процедуры DPC не должны зависеть от конкретного потока (а следовательно, части конкретного процесса) для выполнения своего кода. Поток может быть любым.

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? Опишите последовательность вызова функционала, реализованного многоуровневым драйвером.

**Драйвер –** это часть кода операционной системы, 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 программное обеспечение которое предоставляет другому программному обеспечению API для работы с аппаратными устройствами.

Теперь рассмотрим классификацию типов драйверов (довольно условную) 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

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 (filter-drivers)

Многоуровневая архитектура приводит к появлению такого понятия как **стек драйверов** – по сути это набор драйверов которые необходимо вызывать для получения конечного результата

Также необходимо запомнить два термина – **вышестоящие** (**higher-level**) и **нижестоящие** (**lower-level**) драйверы. При обработке запроса данные идут от вышестоящих драйверов к нижестоящим, а при возврате результатов – наоборот

Ну и, понятно, **одноуровневый** (**monolithic**) драйвер просто является противоположностью многоуровневому

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

Многоуровневые WDM-драйверы могут включать:

- Драйверы классов (Class Drivers): Для обработки ввода/вывода для устройств определенного класса.

- Драйверы мини-классов (MiniClass Drivers): Реализуют обработку, заданную производителем для конкретного класса.

- Драйверы портов (Port Drivers): Обрабатывают запросы в соответствии с типом порта.

- Драйверы мини-портов (Miniport Drivers): Преобразуют обобщенные запросы в специфичные для адаптера.

Процесс вызова функционала многоуровневого драйвера может выглядеть следующим образом:

1. \*\*Инициализация\*\*: Диспетчер ввода/вывода вызывает процедуру инициализации драйвера при загрузке.

2. \*\*Добавление устройства\*\*: При обнаружении устройства PnP-диспетчер уведомляет драйвер, который создает объект устройства.

3. \*\*Диспетчеризация\*\*: Диспетчер ввода/вывода генерирует IRP-пакет и вызывает соответствующую диспетчеризирующую процедуру драйвера.

4. \*\*Начало ввода/вывода\*\*: Драйвер инициирует передачу данных, обрабатывая запросы на ввод/вывод.

5. \*\*Обработка прерываний\*\*: Диспетчер прерываний передает управление процедуре обработки прерываний (ISR).

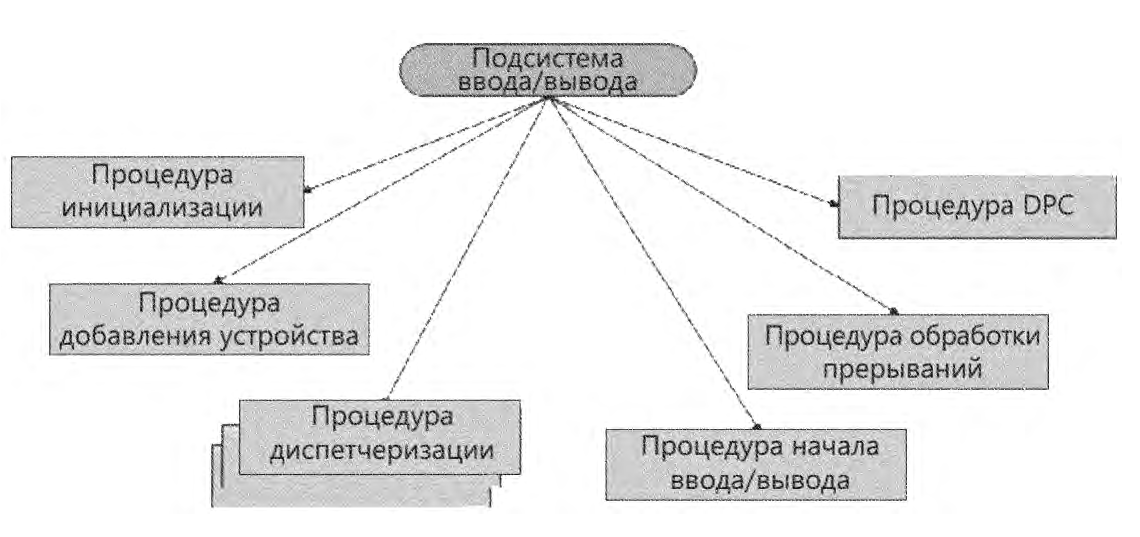
6. \*\*DPC\*\*: ISR ставит в очередь DPC для выполнения на более низком IRQL, где происходит основная обработка ввода/вывода.

7. \*\*Завершение\*\*: DPC завершает обработку и возвращает управление диспетчеру ввода/вывода.

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 операционной системы, 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 программное обеспечение которое предоставляет другому программному обеспечению API для работы с аппаратными устройствами.

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



Для успешного запуска драйвера требуются следующие компоненты:

- \*\*Процедура инициализации (DriverEntry)\*\*: Точка входа для драйвера, где выполняется регистрация других процедур и глобальная инициализация.

- \*\*Объект драйвера (DRIVER\_OBJECT)\*\*: Представляет сам драйвер в системе и предоставляет адреса процедур диспетчеризации.

- \*\*Объект устройства (DEVICE\_OBJECT)\*\*: Описывает физическое или логическое устройство, с которым работает драйвер, включая его характеристики.

- \*\*INF-файл\*\*: Файл конфигурации, который управляет установкой драйвера и связывает устройства с соответствующими драйверами.

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

**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** (FILE\_OBJECT) — это структура данных в режиме ядра, представляющая дескриптор устройства. Он используется для управления доступом к ресурсам, обеспечивая возможность синхронизации и защиту данных. Файловые объекты могут иметь имена и обеспечивают интерфейс для выполнения операций ввода/вывода, таких как чтение и запись. Они позволяют нескольким процессам в пользовательском режиме взаимодействовать с одним и тем же ресурсом.

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 (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** (I/O Request Packets, 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.

Вот ответы на ваши вопросы, основанные на материалах лекции:

IRP можно классифицировать по различным критериям, включая типы операций:

- \*\*IRP\_MJ\_READ\*\*: Запрос на чтение данных из устройства.

- \*\*IRP\_MJ\_WRITE\*\*: Запрос на запись данных на устройство.

- \*\*IRP\_MJ\_DEVICE\_CONTROL\*\*: Запрос на выполнение операции управления устройством.

- \*\*IRP\_MJ\_CLEANUP\*\*: Запрос на завершение работы с файловым объектом.

- \*\*IRP\_MJ\_CLOSE\*\*: Запрос на закрытие файлового объекта.

Каждый тип IRP обрабатывается соответствующими диспетчеризующими процедурами драйвера.

Plug and Play (PnP) — это технология, позволяющая операционной системе автоматически обнаруживать и настраивать устройства, подключаемые к компьютеру, без необходимости ручной настройки со стороны пользователя. PnP обеспечивает удобство подключения и использование аппаратуры.

Программное обеспечение с поддержкой PnP предоставляет следующие возможности:

- \*\*Автоматическое обнаружение устройств\*\*: Операционная система автоматически определяет новое подключенное устройство.

- \*\*Автоматическая установка драйверов\*\*: При подключении нового устройства операционная система автоматически загружает необходимые драйверы.

- \*\*Выделение ресурсов\*\*: PnP управляет распределением ресурсов, таких как прерывания и адреса памяти, для новых устройств.

- \*\*Динамическое управление устройствами\*\*: Возможность добавления и удаления устройств без перезагрузки системы.

Система PnP состоит из следующих компонентов:

- \*\*PnP-диспетчер\*\*: Управляет процессом обнаружения и установки устройств.

- \*\*Драйверы устройств\*\*: Обрабатывают запросы от PnP-диспетчера и обеспечивают взаимодействие с аппаратным обеспечением.

- \*\*Реестр\*\*: Хранит информацию о подключенных устройствах и их конфигурации.

- \*\*INF-файлы\*\*: Определяют, как устанавливать драйверы и настраивать устройства. PnP может работать с различными аппаратными устройствами, такими как:

- Жесткие диски

- Принтеры

- Видеокарты

- Сетевые адаптеры

- Устройства ввода (клавиатуры, мыши и т.д.)

Чтобы обеспечить полную поддержку PnP, драйвер должен:

- Реализовать процедуры добавления и удаления устройства.

- Обрабатывать запросы от PnP-диспетчера на инициализацию и завершение работы устройства.

- Поддерживать информацию о состоянии устройства и управлять его ресурсами.

- Реагировать на события подключения и отключения устройств.

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.

Как происходит выполнение кода программы в операционной системе

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

Соответственно будем считать, что программа есть 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:

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

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

Что происходит при вызове функции:

* Когда выполняется вызов функции, аргументы должны быть преобразованы в значения (по крайней мере, для языков программирования, подобных C)
* Затем поток управления переходит к телу функции, и код начинает выполняться там
* Как только мы встречаем оператор return, мы завершаем работу с функцией и возвращаемся обратно к вызову функции

Современные системы программирования в вопросе организации исполнения кода программ работают по модели **распределения памяти на основе стека**. Данная модель основана на том, что для хранения параметров (аргументов) процедур и функций, их локальных переменных, а также адреса возврата в программе выделяется специальная область памяти организованная в виде стека

Этот стек для каждого потока выделяется отдельно и называется **стеком вызовов.**

**Стек вызовов** (call stack) – 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 (stack frames)** – это машинно-зависимые и ABI-зависимые структуры данных, содержащие информацию о состоянии подпрограммы

Каждый кадр стека   
соответствует вызову   
подпрограммы,   
который еще не завершился   
возвратом. Стековый фрейм в верхней части стека предназначен для выполняющейся в данный момент процедуры, которая может получать доступ к информации внутри своего фрейма (например, к параметрам или локальным переменным) в любом порядке.

Стековый фрейм обычно содержит, по крайней мере, следующие элементы:

* аргументы (значения параметров), передаваемые подпрограмме (если таковые имеются)
* адрес, возвращаемый вызывающей программе
* пространство для локальных переменных подпрограммы (если таковые имеются)

**Соглашение о вызовах** определяют 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:

* stdcall (Standart Calling Convention)
* cdecl (C calling convention)
* fastcall (Fast calling convention)
* pascal (Pascal calling convention)

stdcall (Standart Calling Convention) – соглашение о вызовах, применяемое в ОС Windows для вызова функций WinAPI

* Аргументы функций передаются через стек, справа налево
* Очистку стека производит вызываемая подпрограмма
* Перед возвратом значений из функции **вызываемая** подпрограмма обязана восстановить значения сегментных регистров, регистров указателя стека и стекового кадра
* Сохранением-восстановлением остальных регистров занимается **вызывающая** программа

cdecl (C calling convention) – соглашение о вызовах, используемое компиляторами для языка Си

* Аргументы функций передаются через стек, справа налево
* Аргументы, размер которых меньше 4 байт, расширяются до 4 байт
* За сохранение регистров EAX, ECX, EDX и стека сопроцессора отвечает **вызывающая** программа, за остальные – **вызываемая** функция
* Очистку стека производит **вызывающая** программа

cdecl (C calling convention) – соглашение о вызовах, используемое компиляторами для языка Си

Перед вызовом функции вставляется код, называемый **прологом** (англ. prolog) и выполняющий следующие действия:

* сохранение значений регистров, используемых внутри функции
* запись в стек аргументов функции.

После вызова функции вставляется код, называемый **эпилогом** (англ. epilog) и выполняющий следующие действия:

* восстановление значений регистров, сохранённых кодом пролога
* очистка стека (от локальных переменных функции)

fastcall (Fast calling convention) - общее название соглашений, передающих параметры через регистры. Если для сохранения всех параметров и промежуточных результатов – регистров недостаточно, то дополнительно используется стек

* Данное соглашение о вызовах не стандартизировано, поэтому используется только для вызова процедур и функций, не экспортируемых из исполняемого модуля и не импортируемых извне
* В 32-разрядной версии компилятора фирмы Microsoft, соглашение определяет передачу первых двух параметров слева направо в регистрах, а остальные параметры передаются справа налево в стеке. Очистку стека производит **вызываемая** подпрограмма.

pascal (Pascal calling convention) – соглашение о вызовах, используемое компиляторами для языка Паскаль

* Аргументы процедур и функций передаются через стек, слева направо
* Указатель на вершину стека на исходную позицию возвращает **вызываемая** подпрограмма
* Изменяемые параметры передаются только по **ссылке**
* Возвращаемое значение передаётся через изменяемый параметр Result. 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 (API hooking)** – это техника программирования, при которой вызовы функций из библиотеки API перенаправляются на пользовательские функции

Перехват API функций позволяет модифицировать поведение программы, добавлять новые функции или изменять параметры вызовов без необходимости изменения исходного кода программы

Основными методами перехвата являются:

* Подмена адреса настоящей функции (модификация IAT таблиц, модификация SSDT/IDT таблиц)
* Непосредственное изменение функции (сплайсинг, перехват в режиме ядра с модификацией тела функции)
* Непосредственная подмена всего компонента приложения/системы (например 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 операционной системы 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 операционной системы и приложений

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

**Сплайсинг** (**splicing**) – метод перехвата API функций путём изменения кода целевой функции. Обычно изменяются первые 5 байт функции. Вместо них вставляется переход на функцию, которую определяет программист. Чтобы обеспечить корректность выполнения операции, приложение, которое перехватывает функцию, обязано 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.

Все функции стандартных DLL Windows поддерживают hot-patch point. При использовании этой технологии перед началом функции располагаются пять неиспользуемых однобайтовых операций nop, сама же функция начинается с двухбайтовой инструкции mov edi, edi.

Места, занимаемого пятью nop,  
достаточно, чтобы разместить  
команду перехода на   
функцию-перехватчик.

Классический способ реализации API-хуков осуществляется с помощью трамплинов.

**Трамплин** – это шеллкод, который используется для изменения пути выполнения кода 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. Проясним! Шеллкод (shellcode) - это 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. Существует много способов реализации API-хука, один из способов – через открытые библиотеки, такие как библиотека Detours или Minhook

Еще один, более ограниченный способ, – использование API Windows, предназначенных для выполнения API-хука (хотя с ограниченными возможностями)

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

Локальный перехват может быть реализован в WinNT посредством подмены адреса перехватываемой функции в таблице импорта

Как вам уже известно, в разделе импорта содержится список DLL, необходимых модулю для нормальной работы. Кроме того, в нем перечислены 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. К достоинствам данного метода можно отнести то, что код перехватываемой функции не изменяется, что обеспечивает корректную работу в многопоточном приложении. Недостаток этого метода в том, что приложения могут сохранить адрес функции до перехвата, и затем вызывать её минуя обработчик. Также можно получить адрес функции используя GetProcAddress из Kernel32.dll

Добросовестное использование перехвата функций:

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

Недобросовестное использование перехвата функций:

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

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

**Перехват API функций (API hooking)** – это техника программирования, при которой вызовы функций из библиотеки API перенаправляются на пользовательские функции

Перехват API функций позволяет модифицировать поведение программы, добавлять новые функции или изменять параметры вызовов без необходимости изменения исходного кода программы.

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

Когда 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:

**Получение адреса SSDT**: Адрес SSDT можно получить, используя системные структуры и функции Windows. Обычно это делается с помощью недокументированных структур и функций ядра

**Поиск целевого системного вызова**: В SSDT каждому системному вызову соответствует индекс. Необходимо определить индекс системного вызова, который нужно перехватить

**Сохранение оригинального адреса**: Перед заменой адреса системного вызова необходимо сохранить оригинальный адрес, чтобы можно было вызвать оригинальную функцию из пользовательской функции

**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

**Восстановление оригинального адреса (опционально)**: После выполнения необходимых действий можно восстановить оригинальный 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: Перехват может использоваться для перехвата и регистрации вызовов функций в целях отладки, помогая разработчикам понять и диагностировать проблемы в их программном обеспечении
* Программное обеспечение для обеспечения безопасности: Антивирусное программное обеспечение часто использует перехват для отслеживания и перехвата потенциально вредоносных действий
* Мониторинг производительности: Подключение может использоваться для сбора показателей производительности и оптимизации программного обеспечения
* Расширение функциональных возможностей: Подключение может использоваться для добавления или изменения функциональных возможностей в существующее программное обеспечение без изменения исходного кода.

Недобросовестное использование перехвата функций:

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

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** – это подход к разработке программного обеспечения, 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.

**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 не исправления 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.

Классификация уязвимостей ПО важным аспектом безопасности, так как она помогает разработчикам и специалистам по кибербезопасности понимать и управлять рисками

В настоящее время существует несколько популярных классификаторов:

* 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 программного обеспечения, которые вызваны:

* **проблемами проектирования**: например, программист не продумал, какой тип аутентификации требуется
* **проблемы с реализацией**: например, 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:

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

**Перечислить все известные уязвимости невозможно, учитывая, что каждый день появляются новые**

**Ошибка переполнения буфера**

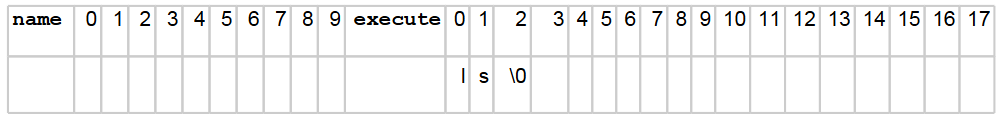
****

Строка "scanf("%[^\n]s”, &name);" считывает любые входные данные, принимая все, кроме новой строки. “[^\n]” - это регулярное выражение, которое представляет любой символ, кроме \n (новой строки). Как указано, scanf() сохраняет данные в переменной “name”. Переменная execute – изменилась.

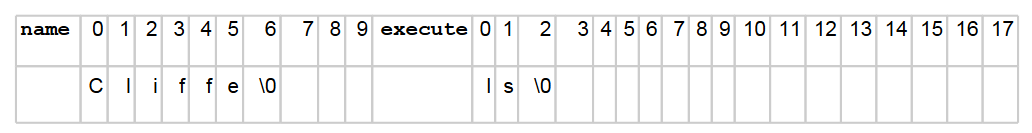
Когда мы ввели длинные данные, эта программа начинает работать некорректно.

**Что же произошло?**

Две переменные располагаются в стеке вместе, одна рядом с другой. Компилятор сам решает, в каком порядке они будут располагаться в стеке, но во время тестов они были расположены следующим образом



Если пользователь поступает так, как ожидалось, и просто вводит короткое имя, наши переменные содержат такие значения, как



Однако наш код не указывал scanf, сколько символов следует считывать от пользователя, поэтому scanf() услужливо считает столько символов, сколько вводит пользователь, и записывает их в буфер имени

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.

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

char \*fgets(char \*s, int size, FILE \*stream);

**Целочисленное переполнение** (integer overflow) – это ситуация, когда результат арифметической операции над целыми числами превышает максимальное значение, которое может быть представлено данным типом данных

Это может привести к непредсказуемому поведению программы, включая сбои, неправильные вычисления и уязвимости безопасности

Методы предотвращения целочисленного переполнения

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

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);

**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 означает защиту пользователей от других пользователей того же компьютера, а также от тех, кто ищет удаленный доступ к нему по сети

Безопасность операционных систем 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").

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. Поскольку формальная проверка операционных систем может оказаться невыполнимой, разработчики используют повышение надежности операционной системы для уменьшения уязвимостей, например: **рандомизацию расположения адресного пространства**, **целостность потока управления**, **ограничения** **доступа**, и другие методы.

**ASLR** (address space layout randomization – «рандомизации размещения адресного пространства») – технология, применяемая в операционных системах, при использовании которой случайным образом изменяется расположение в адресном пространстве процесса важных структур данных, а именно образов исполняемого файла, подгружаемых библиотек, кучи и стека. Технология ASLR создана для усложнения эксплуатации нескольких типов уязвимостей. Например, если при помощи переполнения буфера или другим методом атакующий получит возможность передать управление по произвольному адресу, ему нужно будет угадать, по какому именно адресу расположен стек, куча или другие структуры данных, в которые можно поместить шелл-код.

Также для уменьшения поверхности атаки некоторые ОС предоставляют механизм DEP

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

Основные аспекты принципа наименьших полномочий:

* **Минимизация прав доступа**: Каждый субъект должен иметь только те права, которые необходимы для выполнения его задач
* **Разделение обязанностей:** Разделение задач и обязанностей между различными субъектами для предотвращения концентрации полномочий
* **Контроль доступа на основе ролей (RBAC):** Назначение прав доступа на основе ролей пользователей, а не на основе индивидуальных учетных записей
* **Минимизация времени доступа:** Предоставление доступа только на то время, которое необходимо для выполнения задачи

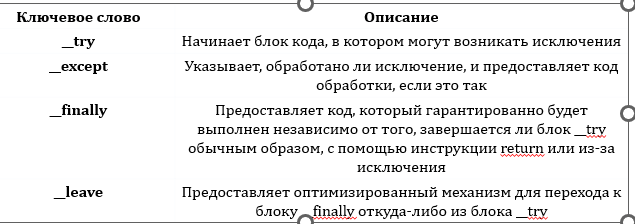
40. SEH. Что такое исключение? Сравните исключения и прерывания. Что такое Structured Exception Handling (далее – SEH)? Что такое блок исключения? Какие основные возможности предоставляет SEH? Что такое защищённый блок? Поясните принципы работы обработчика завершения. Что такое локальная раскрутка? Причины,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

**Исключение** – это событие, возникающее из-за выполнения определенной команды, которая вызвала ошибку процессораСкорее всего в результате такого события нормальное выполнение программы становится **не возможным**! 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 и недопустимую инструкцию.

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

Хотя всю работу по отлову исключений берёт на себя операционная система, однако основная нагрузка по поддержке SEH ложится на компилятор, а не на операционную систему. Он генерирует специальный код на входах и выходах **блоков исключений** (**exception blocks**), создает таблицы вспомогательных структур данных для поддержки SEH и предоставляет функции обратного вызова, к которым система могла бы обращаться для прохода по блокам исключений.

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



Собственно, **обработчик завершения (\_\_finally)** гарантирует, что блок кода (собственно обработчик) будет выполнен независимо от того, как происходит выход из другого блока кода – защищенного участка программы.

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

Принципы работы обработчика завершения.

Обработчик завершения (`\_\_finally`) гарантирует выполнение определенного кода независимо от того, как происходит выход из защищенного блока. Это может быть как нормальный выход, так и выход из-за исключения или инструкции `return`. Код в блоке `\_\_finally` выполняется в любом случае, обеспечивая корректную очистку ресурсов.

Локальная раскрутка происходит, когда управление передается к блоку `\_\_finally` из-за преждевременного выхода из блока `\_\_try`. Это происходит, например, при выполнении операторов `return`, `goto` и других, которые могут нарушить нормальный поток выполнения.

Причины, по которым следует применять обработчики завершения:

- Упрощение обработки ошибок: очистка ресурсов гарантируется и проводится в одном месте.

- Улучшение восприятия кода и его сопровождения.

- Минимизация издержек по скорости и размеру кода при правильном применении обработчиков.

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 процессораСкорее всего в результате такого события нормальное выполнение программы становится **не возможным**!

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

Принцип работы обработчика исключений:

Обработчик исключений в SEH работает с использованием конструкции `\_\_try` и `\_\_except`. Когда в блоке `\_\_try` возникает исключение, управление передается в блок `\_\_except`, где осуществляется проверка на наличие и тип исключения. Если исключение обрабатывается, программа продолжает выполнение с инструкции, следующей за блоком `\_\_except`. Важно, что блок `\_\_except` позволяет выполнять код, который обрабатывает различные типы исключений.

Фильтры — это выражения, которые определяют, какое поведение должно быть выполнено при возникновении исключения. Они используются в конструкции `\_\_except` и позволяют системе понять, как обрабатывать данное исключение.

Фильтры исключений:

- \*\*EXCEPTION\_EXECUTE\_HANDLER\*\*: Указывает, что исключение обнаружено, и у вас есть код для его обработки. Управление передается в блок `\_\_except`.

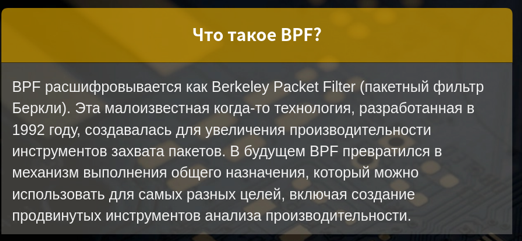
- \*\*EXCEPTION\_CONTINUE\_SEARCH\*\*: Указывает системе продолжить поиск предыдущих блоков `\_\_try`, соответствующих текущему блоку `\_\_except`. Если такой блок найден, будет проверен его фильтр.

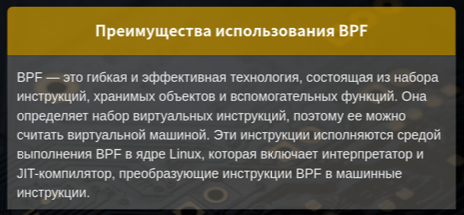
- \*\*EXCEPTION\_CONTINUE\_EXECUTION\*\*: Позволяет системе вернуться к инструкции, вызвавшей исключение, и попытаться выполнить её снова.

Глобальная раскрутка происходит, когда возникает серьезное исключение, которое требует выполнения всех соответствующих блоков `\_\_finally` в стеке вызовов, прежде чем управление передастся в блок `\_\_except`. Это означает, что все незавершенные блоки `try-finally` будут выполнены, чтобы гарантировать, что все ресурсы будут корректно освобождены.

Исключения в SEH можно возбудить с помощью функции `RaiseException`. В этой функции первый параметр — это код исключения, который идентифицирует тип генерируемого исключения. Вы также можете указать дополнительные параметры, которые могут содержать дополнительные данные о генерируемом исключении. Это позволяет программам сообщать о своих ошибках и обрабатывать их на более высоком уровне.

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 (Berkeley Packet Filter) — это мощный механизм фильтрации и мониторинга сетевых пакетов в операционных системах, таких как Linux. Изначально он был разработан для фильтрации сетевых пакетов, но со временем его возможности значительно расширились, что привело к созданию eBPF (extended BPF), который позволяет выполнять произвольный код в ядре операционной системы.

Преимущества использования BPF:

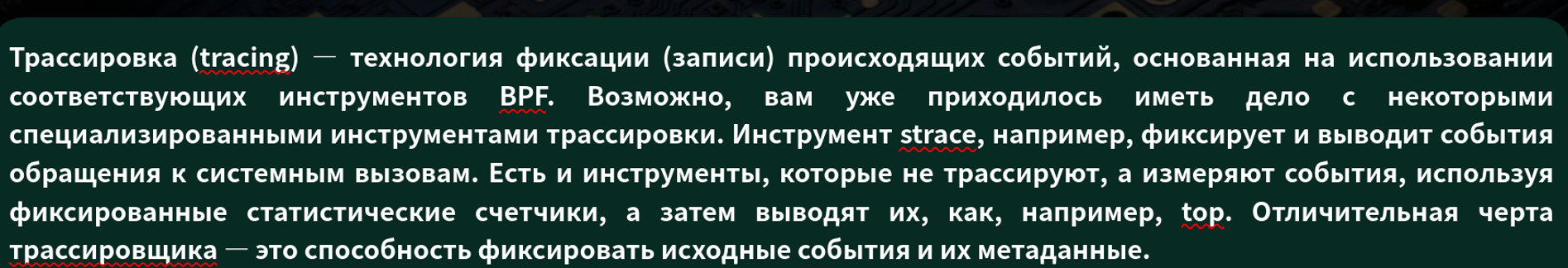
1. \*\*Высокая производительность\*\*: BPF работает в контексте ядра, что минимизирует накладные расходы на обработку пакетов и позволяет обрабатывать их с высокой скоростью.

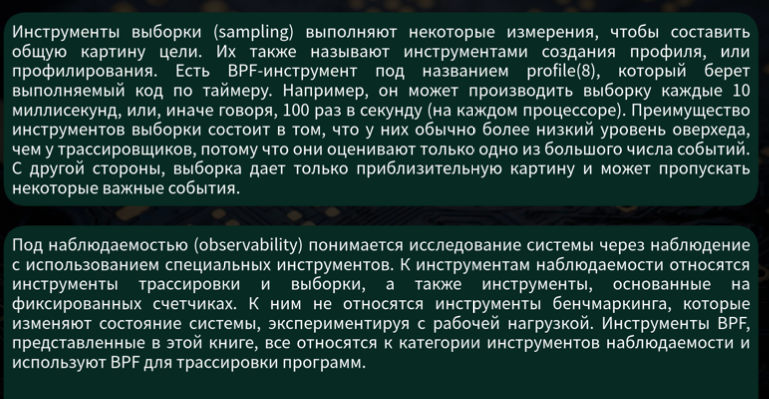
2. \*\*Гибкость\*\*: Позволяет разработчикам писать собственные программы для фильтрации, мониторинга и анализа данных, что подходит для различных задач.

3. \*\*Безопасность\*\*: eBPF имеет встроенные механизмы проверки безопасности, которые гарантируют, что загружаемый код не может повредить систему или вызвать сбой.

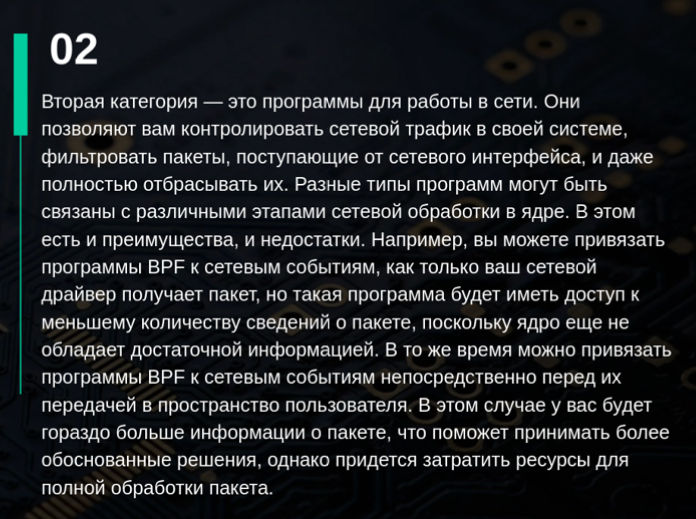
4. \*\*Минимизация нагрузки на систему\*\*: Позволяет обрабатывать пакеты на уровне ядра, уменьшая количество данных, которые нужно передавать в пользовательское пространство.

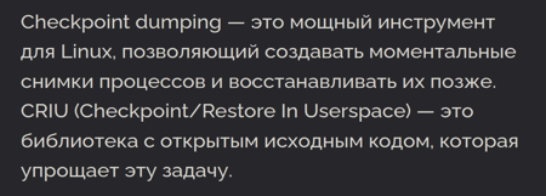
5. \*\*Расширяемость\*\*: Поддерживает множество различных приложений, включая сетевой мониторинг, сбор метрик, безопасность и отладку.











CRIU (Checkpoint/Restore In Userspace) — это инструмент для сохранения и восстановления состояния процессов в Linux. Он позволяет "замораживать" (checkpoint) состояние работающего процесса, включая его память, файловые дескрипторы, состояние сети и другие ресурсы, а затем восстанавливать (restore) этот процесс позже.

### Основные функции CRIU:

1. \*\*Сохранение состояния процессов\*\*:

- CRIU позволяет сохранять текущее состояние одного или нескольких процессов в файл, что может быть полезно для резервного копирования или миграции.

2. \*\*Восстановление процессов\*\*:

- Позволяет восстановить ранее сохраненные процессы, продолжая их выполнение с того места, где они были остановлены.

3. \*\*Поддержка контейнеров\*\*:

- CRIU часто используется в контейнерных технологиях, таких как Docker и LXC, для миграции контейнеров между узлами.

4. \*\*Работа с сетевыми соединениями\*\*:

- CRIU сохраняет информацию о сетевых соединениях, что позволяет восстановить их при восстановлении процесса.

5. \*\*Гибкость\*\*:

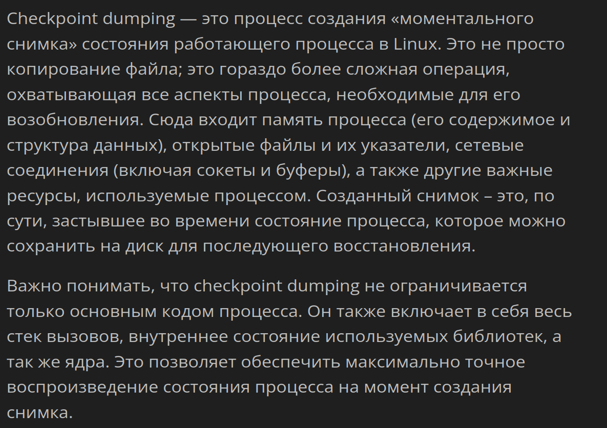
- Поддерживает множество различных конфигураций и позволяет настраивать процесс сохранения и восстановления.

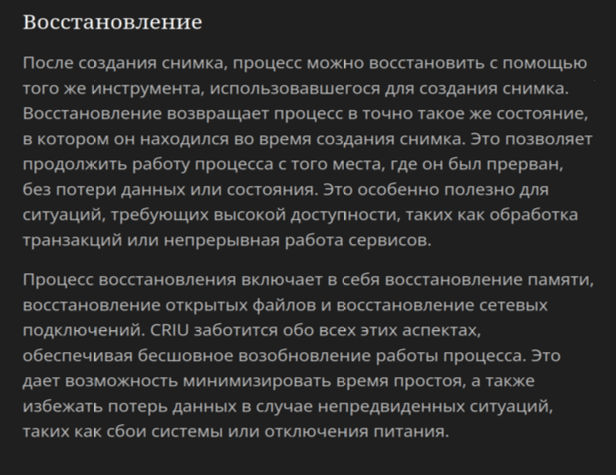
### Применение CRIU:

- \*\*Миграция\*\*: Перемещение процессов между различными серверами без потери состояния.

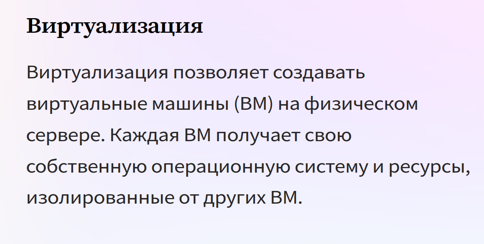
- \*\*Резервное копирование\*\*: Создание резервных копий работающих приложений для предотвращения потери данных.

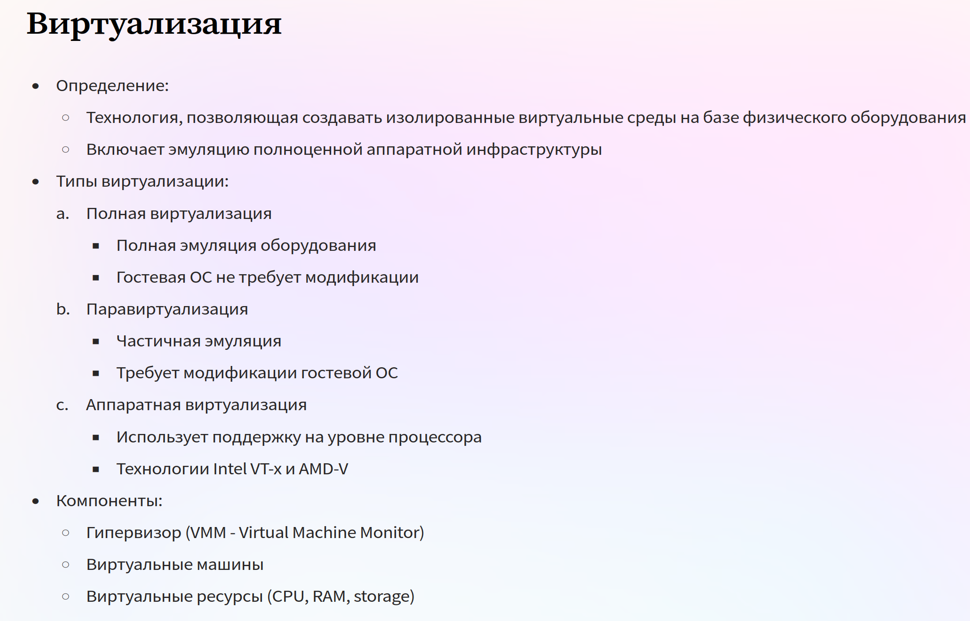
- \*\*Обновление систем\*\*: Позволяет обновлять или заменять программное обеспечение без остановки работы процессов.

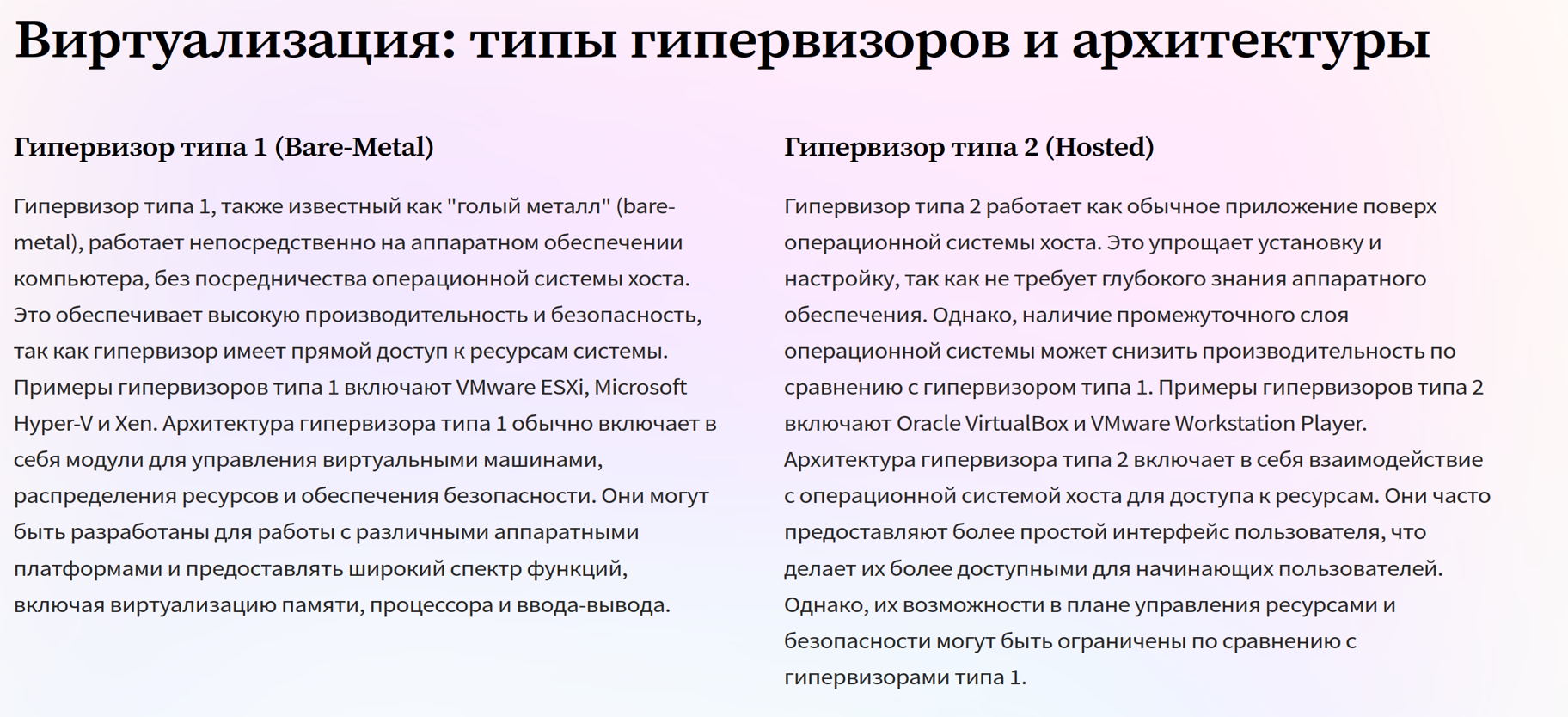


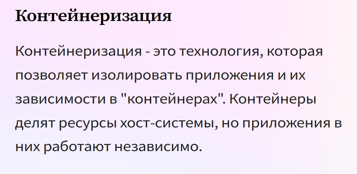


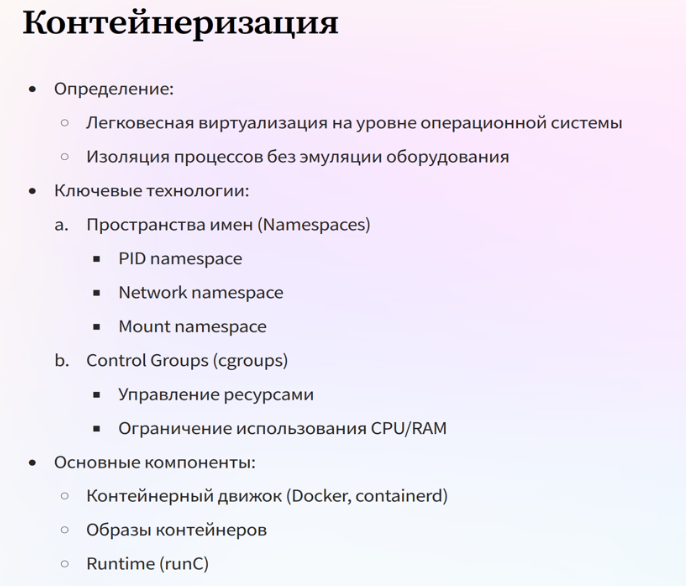
43. Виртуализация. Что такое виртуализация? Перечислите и опишите типы виртуализации. Из каких компонент состоит виртуализация? Опишите гипервизоры первого и второго типов. Что такое контейнеризация? Принципы работы и основные компоненты контейнеризации. Namespaces и cgroups в Linux

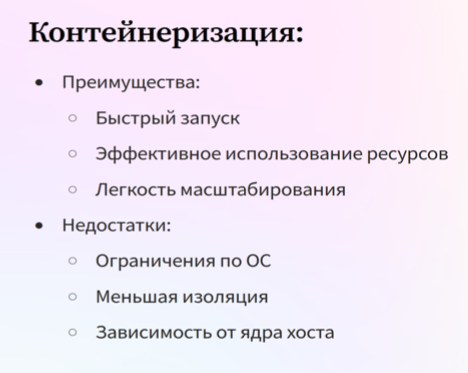




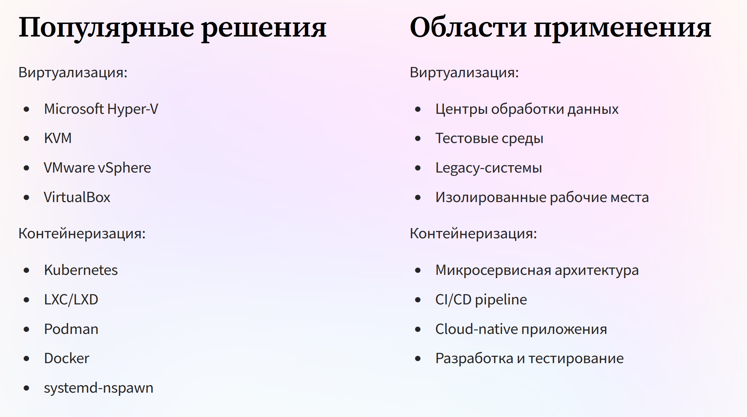


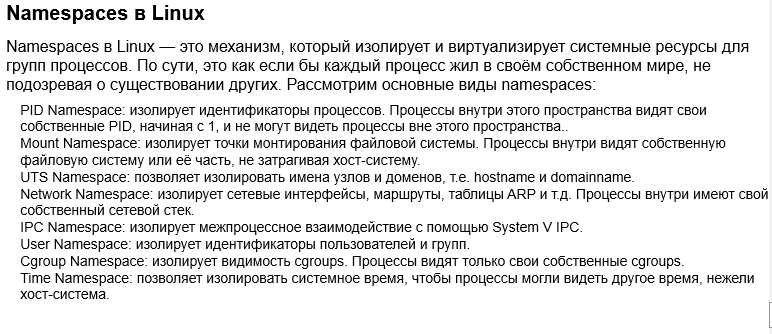


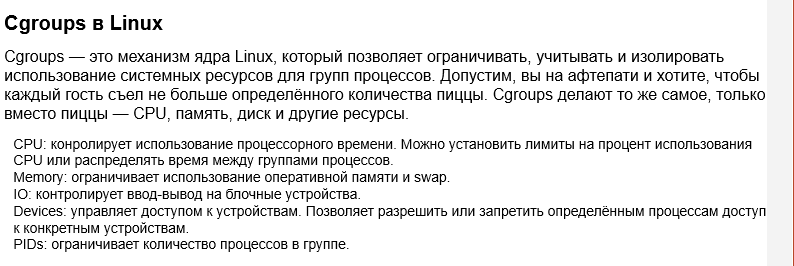












44.Оптимизация кода. Что такое оптимизация кода? Какие характеристики могут быть оптимизированы? Стоит ли оптимизировать код вручную? Основные принципы проведения оптимизации. Ключевые аспекты связи оптимизации и системного программирования. Какие уровни оптимизации существуют?

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

Две самые важные характеристики – это **скорость работы** и **размер кода.**

К другим характеристикам относятся **энергопотребление**, необходимое для выполнения кода, **время компиляции кода** и – в случае, если конечный код требует JIT-компиляции, – **длительность JIT-компиляции.**

Нужно ли заниматься оптимизацией кода вручную?

Однозначного ответа нет, но если коротко – всё зависит от ситуации

Компиляторы не идеальны. Тем не менее, вместо траты на ручную оптимизацию программы гораздо продуктивнее использовать специфические средства компилятора и дать ему возможность оптимизировать код. При определенных условиях оптимизирующий компилятор может создавать код, который почти так же хорош, как и язык ассемблера, оптимизированный вручную

Однако для достижения этих уровней производительности код HLL\* должен быть написан соответствующим образом. Написание HLL-кода таким образом требует четкого понимания того, как работают компьютеры и исполняется программное обеспечение

Поэтому даже если вам не требуется проводить оптимизацию вручную, важно понимать как её проводит компилятор и что в вашем исходном коде может препятствовать эффективной работе оптимизирующего компилятора. Перед тем, как переходить к обсуждению оптимизации стоит отметить, что **она должна проводиться строго при необходимости и проводиться с осторожностью. Нет смысла писать оптимальный код, который не выполняет своего предназначения.**

Оптимизация кода и системное программирование ещё более тесно связаны, особенно в контексте создания эффективных и надежных систем

Вот несколько ключевых аспектов, которые объясняют эту связь:

* Производительность
* Эффективное использование ресурсов
* Надежность и стабильность
* Совместимость и переносимость
* Безопасность
* Отладка и тестирование
* **Производительность:**
* ***Оптимизация кода***: Включает в себя улучшение времени выполнения программы, уменьшение использования памяти и других ресурсов
* ***Системное программирование***: Часто требует высокой производительности, особенно в критически важных системах, таких как операционные системы, драйверы устройств и встроенные системы.
* **Эффективное использование ресурсов:**
* ***Оптимизация кода***: Направлена на минимизацию использования процессорного времени, памяти и других ресурсов
* ***Системное программирование***: Включает управление ресурсами на низком уровне, такими как управление памятью, планирование задач и взаимодействие с аппаратным обеспечением.
* **Надежность и стабильность:**
* ***Оптимизация кода***: Может включать улучшение структуры кода, устранение узких мест и повышение общей стабильности программы
* ***Системное программирование***: Требует высокой надежности, так как ошибки могут привести к серьезным сбоям в системе.
* **Совместимость и переносимость:**
* ***Оптимизация кода***: Может включать адаптацию кода для работы на различных платформах и архитектурах
* ***Системное программирование***: Часто требует создания кода, который может работать на различных аппаратных и программных платформах.
* **Безопасность:**
* ***Оптимизация кода***: Может включать улучшение безопасности кода, устранение уязвимостей и защиту от атак
* ***Системное программирование***: Требует высокого уровня безопасности, особенно в критически важных системах.
* **Отладка и тестирование:**
* ***Оптимизация кода***: Включает процессы отладки и тестирования для выявления и устранения проблем в коде
* ***Системное программирование***: Требует тщательного тестирования и отладки, так как ошибки могут иметь серьезные последствия.

**Возможно несколько уровней оптимизации:**

На самом абстрактном уровне вы можете оптимизировать программу, выбрав для нее лучший алгоритм. Этот метод не зависит от компилятора и языка программирования

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.

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

Наконец, возможно, на самом низком уровне вы можете учитывать машинный код, который выдаёт компилятор, и корректировать инструкции написанные на HLL 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** – это процесс преобразования части кода в другую функционально эквивалентную часть для улучшения одной или более характеристик кода.

Перед тем, как переходить к обсуждению оптимизации стоит отметить, что **она должна проводиться строго при необходимости и проводиться с осторожностью. Нет смысла писать оптимальный код, который не выполняет своего предназначения.**

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. Самый простой способ определить, где начинаются и заканчиваются **базовые блоки**, – это рассмотреть ассемблерный код, который сгенерирует компилятор. Везде, где есть условная ветвь/переход, безусловный переход или команда вызова, базовый блок будет завершаться.

Базовые блоки позволяют компилятору легко отслеживать, что происходит с переменными и другими программными объектами.

Хорошие программы создают **упрощаемые графики потоков** – наглядные изображения пути потока управления.

Как уже упоминалось, потоковые графики в хорошо структурированных программах **поддаются** **упрощению**

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

Краткий обзор основных типов оптимизаций компилятора:

* Свёртка констант
* Распространение констант
* Удаление мёртвого кода
* Удаление общих подвыражений
* Снижение стоимости операций
* Анализ индуктивных переменных
* Анализ инвариантов цикла
* И много другое

**Свёртка констант** – вычисление значений константных выражений или подвыражений во время компиляции, а не во время выполнения.

**Распространение констант** – замена переменных постоянными значениями, если компилятор определяет, что программа присвоила эту константу переменной ранее в коде

**Удаление мёртвого кода** – удаление объектного кода, связанного с определенным оператором исходного кода, когда программа никогда не будет использовать результат этого оператора или когда условный блок никогда не будет истинным

**Удаление общих подвыражений (Часто часть выражения появляется в другом месте текущей функции; это называется подвыражением)** – оптимизация компилятора, которая ищет экземпляры одинаковых выражений и анализирует возможность замены их на одну переменную, содержащую вычисленное значение

**Снижение стоимости операций** – замена медленных операций, например, умножения и деления, на более быстрые, такие как сложение, вычитание, сдвиг

**Анализ индуктивных переменных** – в выражениях, которые появляются в цикле, значение одной переменной в выражении полностью зависит от какой-либо другой переменной. Компилятор может исключить вычисление нового значения или объединить два вычисления в одно на время этого цикла

**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.

**Раскрутка циклов –** техника оптимизации компьютерных программ, состоящая в искусственном увеличении количества инструкций, исполняемых в течение одной итерации 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** – это процесс преобразования части кода в другую функционально эквивалентную часть для улучшения одной или более характеристик кода.

Краткий обзор основных типов оптимизаций компилятора:

* Свёртка констант
* Распространение констант
* Удаление мёртвого кода
* Удаление общих подвыражений
* Снижение стоимости операций
* Анализ индуктивных переменных
* Анализ инвариантов цикла
* И много другое

**Свёртка констант** – вычисление значений константных выражений или подвыражений во время компиляции, а не во время выполнения.

**Распространение констант** – замена переменных постоянными значениями, если компилятор определяет, что программа присвоила эту константу переменной ранее в коде

**Удаление мёртвого кода** – удаление объектного кода, связанного с определенным оператором исходного кода, когда программа никогда не будет использовать результат этого оператора или когда условный блок никогда не будет истинным

**Удаление общих подвыражений (Часто часть выражения появляется в другом месте текущей функции; это называется подвыражением)** – оптимизация компилятора, которая ищет экземпляры одинаковых выражений и анализирует возможность замены их на одну переменную, содержащую вычисленное значение

**Снижение стоимости операций** – замена медленных операций, например, умножения и деления, на более быстрые, такие как сложение, вычитание, сдвиг

**Анализ индуктивных переменных** – в выражениях, которые появляются в цикле, значение одной переменной в выражении полностью зависит от какой-либо другой переменной. Компилятор может исключить вычисление нового значения или объединить два вычисления в одно на время этого цикла

**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.

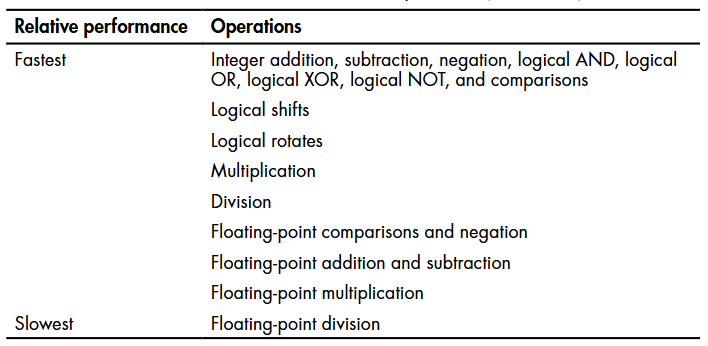
**Раскрутка циклов –** техника оптимизации компьютерных программ, состоящая в искусственном увеличении количества инструкций, исполняемых в течение одной итерации цикла.

**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

Таким образом, после того, как результат выражения становится очевидным, его вычисление прекращается

Основные примеры таких вычислений – логические операторы AND и OR.

К сожалению, мы не можем создать таблицу операторов, в которой указаны их относительные скорости. Производительность отдельного арифметического оператора зависит от конкретного процессора. Однако можно дать несколько общих рекомендаций



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 (*в некоторых случаях, плохой код ведёт к нарушению этого правила*).

**Блокировщики оптимизации –** определённые характеристики кода, которые не позволят компилятору совершить оптимизацию.

Рассмотрим два типа блокировщиков оптимизации

Одним из них являются **указатели, вторым - вызов функций.**

**Указатель:** Компилятор не может точно знать, будут ли два указателя указывать на одну и ту же область памяти, и поэтому не выполняет некоторые оптимизации.

**Вызов функций:** Компилятору тяжело определить,  
имеет вызов функции побочные эффекты или  
нет. Когда он не может этого сделать, он   
рассчитывает на худшее и не выполняет   
оптимизацию.

Данные имеют важное свойство, которое мы называем **локальностью**. Когда мы работаем над данными, желательно чтобы они находились в памяти рядом.

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.**