**Экзаменационные вопросы**

**ИСиТ-3, лето 2024**

**Системное программирование**

1. Системное программирование: 2 определения, назначение, применение, особенности разработки приложений.

**Определения**:

1. СП – Использование api OS **в целях разработки**. СП – программирование на уровне api OS. Т.е. используем фреймворк ОС
2. Разработка **общего программного обеспечения**. Когда разрабатываем какой-то код, который будет использоваться другими программистами. Разработка не для конечного пользователя, а для программиста (часто называемых «проблемными».).
3. Можно также выдвинуть и третий признак **Третий признак**: системное программное обеспечение напрямую использует системные вызовы операционной системы.

Пример такой программы общего назначения COM от Microsoft.

Эти два определения не противоречат друг другу. Потому что когда сис программисты пишут общий код, его основное назначение – спрятать сложности использования api OS за слоем ПО (библиотека, которая доступная на ОС: на C#, Python, …)

Разработка сис программиста привязана к **конкретной** ОС или ее версии. Потому что разрабатываемый низкоуровенымй код часто изменяется в зависимости от сис вызовов разных версий ОС.

Если залезем в .h файл то увидим что вызов ф-й заменяется на макросы. Получается, мы вызываем старую версию, а на самом деле вызывается макрос, который подставляет новую.

Если будем разрабатывать, **не гарантируется** что будет работать наш код в следующей версии ОС. Проблема совместимости. Не всегда достигается. Это и есть особенность разработки.

– *в каких случаях привлекаются системные программисты? – если надо разработать что-то зависимое от конкретной ОС. Они пишут прослойку между фреймворком ОС и системой программирования.*

- *какие приложения разрабатывают системные программисты? – драйверы (если есть устройство, которое подключается к ПК через порт, можем написать драйвер который скрывает сложности работы с ним. Или как плата, которая вставляется внутрь такого ус-ва (печатная плата (PCB, Printed Circuit Board) или PCI) или программы общего пользования.*

*Раньше можно было самому вызывать программные прерывания – команду int, теперь на уровне компилятора не позволяют это сделать. Пользуемся api, int не выдают. Смелов писал учебную лабу ­– обработчика клавиш клавы, которые реагируют на аппаратные прерывания (не програм), обрабатывают отжатие/нажатие.*

1. Системный вызов: определение, назначение, применение, реализация в операционных системах, механизм исполнения.

**Системный вызов** – Вызов функций ядра ОС – оно переключает процесс (каждое программное прерывание переключает режим процессора). Есть различные ринги в каждом свой набор инструкций процессора. Когда делаем системный вызов происходит переключение в режим ядра (незащищенный режим, нулевой ринг) (сис вызов каждый раз сопровождается программным прерыванием (в каждой архитектуре своя. Если говорить про 88.6 - int, другие – syscall, setevent( вроде он сказал SYSENTER/SYSEXIT, чат говорит так)

– команды (системный вызов) от 86 процессора (32-bit) **сохранились** у самых новых процессоров, потому что инструкции процессора **не выбрасываются**, они прибавляются (**наращиваются**), а не заменяются)). Если С-комилятором, который приходит с этой ОС, откомпелировать программу на ассемблере для 86 процессора, то она будет работать и здесь.

Рассказать как происходит сис. Вызов. Эта часть чатом

1. Аргументы системного вызова помещаются в определенные регистры или на стек.
2. Команда int (для x86. Syscall для x86\_64) используется для вызова системного прерывания.
3. Ядро получает управление и использует номер системного вызова, чтобы определить, какую функцию выполнить.
4. Ядро выполняет необходимую функцию, используя переданные аргументы.
5. После выполнения системного вызова управление возвращается в пользовательское пространство.
6. Возвращаемое значение системного вызова помещается в регистр eax.

Надо рассказать вектор прерываний (обычно находится на 0 (нулевых) адресах (в первых 1024 байтах памяти, начиная с адреса 0), если 32 разрядная машина – каждый элемент вектора это 4-х байтный адрес, 64 бит – 8 байтный. Адреса обработчика прерываний).

Эта часть чатом

Вектор прерываний — это таблица, содержащая адреса обработчиков прерываний. Когда процессор получает прерывание, он использует вектор прерываний для определения адреса обработчика, который будет выполнять соответствующий код. Эта таблица обычно располагается в памяти по фиксированному адресу (почитайте пример нижу, сразу станет все понятно).

* Когда происходит прерывание, процессор умножает номер прерывания на размер записи (8 байт) и использует полученный адрес для извлечения 64-разрядного адреса обработчика прерывания из вектора.
* Затем процессор переходит к этому адресу и начинает выполнение обработчика прерывания.

Пример.

1. Предположим, что прерывание 14 (страничный сбой) произошло. Вектор прерываний на 32-разрядной системе будет выглядеть так:
2. Номер прерывания = 14
3. Адрес в векторе = 14 \* 4 байта = 56
4. Предположим, что вектор прерываний в памяти имеет следующий вид
5. [56] -> 0x0010ABCD (адрес обработчика прерывания 14)
6. Процессор переходит к адресу 0x0010ABCD и начинает выполнение обработчика прерывания.

Делается так. Есть вектор прерываний. Умножается на 4 или 8. Находится смещение в векторе прерываний. Находится этот адрес и передается управление соответствующему оборудованию.

**Можно сделать ОС без программных прерываний**, но **без аппартаных** – **нет** (потому что не будешь постоянно опрашивать все ус-ва об их состоянии, ус-во должно само сообщать о своем состоянии через аппаратное прерывание). На спутнике делал смелов без программных прерываний. Сам пишешь код, фактически сам пишешь ОС, загружаешь код и не нужно программные прерывания.

Два подхода в ОС: **разделяют** функции между **ядром** и **приложением**. Есть ОС где много чего заталикавается (функций. Около 2000 сис. вызовов) в ядро ОС (пр., **Windows**). А есть **Linux**, который большинство своих функций заволок в фреймворк ОС без сис. Вызовов.

**Механизм сис вызова** – когда происходит системный вызов (специальная команда (номер)) – происходит переключение ядра в другой режим (ядра), по вектору вычисляется адрес обраб-ка прерывания – обработчник прерываний работает – в конце происходит переключает в обратную сторону с помощью спец команды завершает работу в (вроде сказал sysref), переключает ядро обратно в режим в пользовательский режим.

Для чего нужны сис вызовы – чтобы что-то **скрыть** от прикладной программы, чтобы возложить на ядро (потому что перключается ядро и скрывается то что на самом деле происходит) чтобы потом можно было переделать. Сис вызов будет такой же, но реализация другая.

– *можно обойтись без системных вызовов? –* ***можно****.*

- *зачем нужны системные вызовы? – скрыть функционал от прикладной программы*

**Системный вызов (System Call)** в операционных системах представляет собой ***интерфейс между пользовательским приложением и ядром операционной системы***. Этот механизм позволяет приложениям взаимодействовать с ядром для выполнения привилегированных операций, таких как чтение/запись в файлы, создание процессов, управление памятью и другие. Примеры системных вызовов могут включать: Открытие файла Чтение/запись данных в файл и создание процессов.

- *что нужно сделать, чтобы создать собственный системный вызов?*

*Без перекомпиляции нельзя. Нужно в ядро засовывать свою функцию, переписать вектор прерываний - добавить свой номер туда.*

*IBM 370 – установка ОС заключалась в перекомпиляции ядра. Есть часть в объектном, часть в исходном. Процесс установки заключается в том что при компиляции ядра, мы получаем новое. И там можно было дописать свой код для прерываний. Тоже самое можно сделать и в Linux и Win, Linux код доступен.*

1. Фреймворк операционной системы: определение, назначение, применение, состав. Стандарт POSIX. Фреймворки систем программирования.

**Фреймворк** (перевод, каркас) – **набор библиотек и инструментов** к их применению, ***которые являются посредниками между разрабатываемым ПО и ядром ОС***. Пошло понятие от POSIX в Linux, применили для ОС впервые. Примеры фреймворков ОС: .Net, .NetCore. в рамках этих фреймворков может исполняться наш код. всегда наш код выполняется в рамках одного из фреймворков. В одной ОС может быть несколько фреймов. У каждого свои правила написания кода.

[**Назначение**: Фреймворк операционной системы предназначен для упрощения процесса разработки приложений, обеспечивая общие функции, которые большинство приложений могут использовать, вместо того чтобы разработчики создавали эти функции с нуля для каждого приложения](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA).

[**Состав**: Фреймворк операционной системы может включать в себя API, библиотеки для работы с файлами и сетями, графические и пользовательские интерфейсы, системы управления памятью и другие компоненты](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA).

От библиотеки отличает то, что ФВ диктует правила построения архитектуры приложения, когда как библиотека это набор подпрограмм похожей функциональности, не влияя на архитектуру программного продукта и не накладывая на неё никаких ограничений

**Фреймворк Системы Программирования** – Entity Framework, Identity FW – отдельный ФВ которые испо-ся в разных СП. Термин ФВ перекочевал из ОС в СП. Поэтмоу Исп-ся и там и там.

Стандарт POSIX – описывает сис. Вызовы, api OS, Стандарт api ОС, ФВ ОС. Стандарты консольных команд cmd, bash.

IEEE – поддерживает POSIX.

[**Цель создания этого стандарта** - обеспечение совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода](https://ru.wikipedia.org/wiki/POSIX).

– *где дано определение понятие «фреймворк»? –* Пошло понятие от POSIX в Linux, применили для ОС впервые.

*- какая организация поддерживает стандарт POSIX? – IEEE (****Institute of Electrical and Electronics Engineers****)*

*- что написано в POSIX?*

*Описывает api ОС (стандарт), стандартых консольных команд (команды cmd, bash).*

*Api OS, набор команд*

*- перечислите известные фреймворки.*

***React, Angular, Express.js*** *(Node.js),* ***Django*** *(Python),* ***ASP.NET Core*** *(C#),* ***Laravel*** *(PHP), Unity (C#)*

1. Применение процессов в ОС Windows, API для работы с процессами.

**Процесс** с одной стороны — это **программа**, которая запущена и выполняется, а с другой – **Объект ядра** ОС, который соответствует запущенной программы.

Для чего нужен? Ресурс какие выделяются? **!!!Не выд-ся машинное время**, оно выд-ся потоку.

Выделяется: **Память, адресное пространство**, Три потока **input**, **output**, **error**, закрепляются **Дескрипторы файлов** (которые создаются при открытии файлов). Все ресурсы в рамках ОС закрепляются за процессом **кроме процессорного (машинного) времени**. Закрепляется – означает: есть блок – объект ядра ОС и к нему списком перечислены ресурсы, которые исп-ся данным процессом: **адреса, адреса памяти, вирт адреса, heap**. Процесс = объект ядра ОС + все ресурсы (вирт. Адреса, heap, …).

* процессу выделяется линейное адресное пространство (размер зависит от разрядности), сегменты: code, static(?), data, heap, stack;

(Сегмент кода (Code): Этот сегмент содержит исполняемый код программы.

Сегмент данных (Data): В этом сегменте хранятся глобальные переменные.

Куча (Heap): Это область памяти, которую программа может использовать для динамического выделения памяти во время выполнения.

Стек (Stack): Этот сегмент используется для хранения локальных переменных и информации о вызовах функций.)

* **контекст процесса** – данные, которые сохраняются при переключении процессов и предназначенные для продолжения работы;
* **Модель 7 состояний:**



– *в каких случаях вы будете создавать дочерний процесс?*

*Когда требуется* ***изоляция****, чтобы работающие куски нашей программы не мешали друг другу, они запускаются в отдельных процессах. Основное назначение процесса:* ***изоляция от других процессов****.*

*- как создать в операционной системе процесс автоматически?*

***Сервисы, демоны*** *(linux) и второй способ* ***автозапуск*** *(в любой ОС) и еще каждому пользователю можно прикрепить* ***скрипт*** *при создании пользователя – там записать автоматические приложения*

*- особенности создания процессов в linux (fork);*

***Копия исходного процесса****. Особенность, после fork* ***Нужно проверять*** *где мы находимся в дочернем или исходном.*

*- особенности исполнения процесса в docker-окружении.*

*У любого процесса закрепляются какие-то ресурсы. есть окружение которое у него есть в ОС – переменные окружения, файлы которые он открывает (это все создается в докер-окружении) – создает среду исполнения для процесса, хотя сам выполняется на ядре хост машины.*

1. Методы межпроцессного взаимодействия в ОС Windows: обмен данными, синхронизация.

Обмен данными - это процесс передачи информации между двумя или более процессами. [В Windows для этого используются механизмы IPC (Inter-Process Communication), такие как буфер обмена, сопоставление файлов, сокеты и т.д1](https://learn.microsoft.com/ru-ru/windows/win32/ipc/interprocess-communications)[2](https://bing.com/search?q=%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D1%8B+%D0%BC%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B3%D0%BE+%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D1%8F+%D0%B2+%D0%9E%D0%A1+Windows+%D0%BE%D0%B1%D0%BC%D0%B5%D0%BD+%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%BC%D0%B8+%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F). [Синхронизация - это процесс управления доступом к ресурсам, который позволяет избежать конфликтов при одновременном доступе нескольких процессов к одному ресурсу 3](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5). [В Windows для этого используются механизмы синхронизации, такие как мьютексы, семафоры](https://learn.microsoft.com/ru-ru/windows/win32/ipc/interprocess-communications).

**Методы IPC**: socket, named pipe, pipe, mailslot, общая память (загрузка файла в оперативную память File Mapping), синхронизация

[**Обмен данными между процессами в ОС Windows:**](https://learn.microsoft.com/ru-ru/windows/win32/ipc/named-pipes)

[**1. NamedPipe (Именованный канал)**](https://learn.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication)[1](https://learn.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication)[2](https://learn.microsoft.com/en-us/windows/win32/ipc/named-pipes)[3](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%B5%D0%BD%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB)[: Именованный канал - это один из методов межпроцессного взаимодействия, который обеспечивает однонаправленную или двунаправленную связь между сервером канала и одним или несколькими клиентами канала1](https://learn.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication). [Все экземпляры именованного канала используют одно и то же имя канала, но каждый экземпляр имеет свои собственные буферы и дескрипторы и предоставляет отдельный канал для связи клиент/сервер2](https://learn.microsoft.com/en-us/windows/win32/ipc/named-pipes). [Именованные каналы поддерживают полнодуплексную связь по сети и несколько экземпляров сервера, обмен сообщениями и имитацию клиента, что позволяет подключающимся процессам использовать свой собственный набор разрешений на удаленных серверах](https://learn.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication)

[**3. Mailslot**](https://learn.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication)[8](https://en.wikipedia.org/wiki/MailSlot)[9](https://ru.wikipedia.org/wiki/Mailslot)[10](https://learn.microsoft.com/en-us/windows/win32/ipc/mailslots)[11](https://stackoverflow.com/questions/7186876/whats-the-difference-between-named-pipe-and-mailslot-mailbox): Mailslot - это механизм для однонаправленного межпроцессного взаимодействия (IPC). Приложения могут хранить сообщения в Mailslot. [Владелец Mailslot может извлекать сообщения, которые там хранятся8](https://en.wikipedia.org/wiki/MailSlot). [Эти сообщения обычно передаются по сети либо на указанный компьютер, либо на все компьютеры в указанном домене8](https://en.wikipedia.org/wiki/MailSlot). Mailslot представляет собой серверно-клиентский интерфейс. [Сервер может создать Mailslot, а клиент может записать в него по имени9](https://ru.wikipedia.org/wiki/Mailslot). [Только сервер может читать Mailslot, поэтому Mailslot представляет собой механизм одностороннего взаимодействия9](https://ru.wikipedia.org/wiki/Mailslot).

**4.** **Межпроцессные Очереди Сообщений (Message Queues):**

Обмен данными: Очереди сообщений предоставляют механизм для отправки и приема сообщений между процессами.

Синхронизация: Очереди сообщений могут использоваться для управления доступом к данным, обеспечивая синхронизацию.

**5.Разделяемая Память (Shared Memory):**

Обмен данными: Разделяемая память позволяет процессам разделять общий участок памяти, в котором они могут записывать и читать данные.

[**Синхронизация процессов в ОС Windows:** Синхронизация процессов в Windows обеспечивает целостность какого-либо ресурса (файл, данные в памяти), когда он используется несколькими процессами или потоками в случайном порядке14](https://bing.com/search?q=). [Для синхронизации процессов и потоков используются семафоры, мьютексы и критические секции15](http://novostynauki.com/e-ntsiklopediya/lektsii/sistemnoe-programmnoe-obespechenie/sredstva-sinhronizatsii-potokov-v-os-windows-kriticheskie-sektsii-m-yuteksy-semafory-soby-tiya/).

[Синхронизация процессов в операционной системе Windows обеспечивает целостность какого-либо ресурса (файл, данные в памяти), когда он используется несколькими процессами или потоками в случайном порядке1](https://bing.com/search?q=). [Для синхронизации процессов и потоков используются различные механизмы1](https://bing.com/search?q=):

1. **Семафоры**: Семафоры используются для контроля доступа к общим ресурсам. [Они представляют собой переменную, которая показывает, сколько ресурсов доступно для использования1](https://bing.com/search?q=).
2. **Мьютексы**: Мьютексы, или взаимные исключения, используются для предотвращения одновременного доступа к общему ресурсу. [Если один процесс использует ресурс, другие процессы должны ждать, пока первый процесс не освободит ресурс1](https://bing.com/search?q=).
3. [**События**: События используются для уведомления одного или нескольких процессов о том, что произошло определенное событие1](https://bing.com/search?q=).
4. **RPC (Remote Procedure Call):**

Обмен данными: RPC позволяет вызывать функции или процедуры в удаленных процессах, обеспечивая передачу аргументов и результатов.

Синхронизация: RPC может включать в себя механизмы синхронизации, такие как удаленные мьютексы.

**Сокеты**: Обеспечивают обмен данными между процессами на одном компьютере или в сети через стандартные сетевые протоколы TCP/IP или UDP.

**COM и DCOM**: Компонентный объектный модель (COM) позволяет объектам, созданным в одном процессе, взаимодействовать с объектами в других процессах на том же компьютере или через сеть с использованием DCOM.

**Сообщения Windows (Windows Messaging)**: Позволяют обмениваться сообщениями между окнами, которые могут принадлежать разным процессам.

– *в каких случаях вы будете использовать IPC? Ответ чатом*

* *Когда необходимо координировать или синхронизировать действия между несколькими процессами.*
* *Для обмена большими объемами данных между приложениями, работающими на одном или разных устройствах.*
* *В распределенных системах для обеспечения совместной работы компонентов, разбросанных по разным узлам сети.*

*- можно ли организовать IPC c помощью DLL?*

*Можно но сложно. Есть статическая память общая для нескольких процессов. Нужно синхронизировать.*

*- как организовать обмен данными через память, но без файла?*

*MapView работает в двух режимах – 1й режим работает с файлом, 2 режим* ***INVALID\_HANDLE\_VALUE*** *– создаст область памяти без файла (будет думать что он файл подкачки)*

*Чат:*

*Вместо того чтобы отображать реальный файл, можно создать разделяемую память, используя функцию* ***CreateFileMapping*** *с дескриптором файла* ***INVALID\_HANDLE\_VALUE*** *и затем отображать эту память в адресное пространство процесса с помощью* ***MapViewOfFile****. Это создаст блок памяти, который может быть доступен из любого процесса, который знает имя или дескриптор этого блока.*

1. Применение потоков в ОС Windows, API для работы с потоками, API для синхронизации потоков.(37:47)

– *чем отличается асинхронность от многопоточности?*

*- поясните понятие «потокобезопасный объект»*

*- зачем придумали потоки?*

*- особенности реализации потоков в unix/linux;*

Что такое поток, для чего придумали, для чего нужен.

В Linux поток и процесс одно и тоже для ядра

Он различает только 2 вида процессов: процессы у которых есть общая память и у которых нету общей памяти.

Потоки нужны, когда запуск процесса тяжелый и как-то надо делить процессорное время между приложениями -- для диспетчеризации процессов.

Если один процессор реального распараллеливания нет у никакого они все по очереди будет выполняться. Что позволяет это сделать- программы которые мелкие быстрее проскакивают и кажется что все быстрее работает, но на деле все выполняется оп порядку.

*- поясните понятие «потокобезопасный объект»*

Это такой объект который позволяет его использовать не используя синхронизацию. Позволяет использовать объект без синхронизации. Менять в нескольких потоках без синхронизации. Внутри Api встроен объект синхронизации.

В Heap одновременно с несколькими потоками, тк внутри уже встроен Апи с поток безопасностью

Для чего Heap.

Обьект относящийся к процессу, когда процес создается длина Heap динамически меняется и нач значение укзывается в Линкере(параметр который при запуске линкера указывается) У Стека длина тоже задается в линкере.

Когда создается загрузочный код там и указана длина стека, а записывается она в спец клочок в PE блок(что это?) и в нем находится длина стека и хипа и когда загрузчик запускает он считывает жто PE блок и выделяет указанные параметры для стека и хипа

*- особенности реализации потоков в unix/linux;*

Ядро не разлечает потоки. Процессы легковесные как потоки в виндовс.  
 Как они достигли изоляции – все вытащили за предлелы ядра = вынесли все в фреймворк.   
есть отдельная библиотека с помощью которых сделать потоки

*Что такое fiber* –собственная реализация потоков делающаяся не через ОС, а мы сами делаем, фактически мы сами делаем поток и разделение между кодом **Квантуем** код но внутри одного потока. Переходы делаем сами. Собственное квантование, квантуется в ОС и квантуется сами

*- зачем придумали потоки?*

**Поток** - объект ядра операционной системы, которому OS выделяет процессорное время. -Наименьшая единица работы ядра OS.

Зачем потоки?

・Потоки позволяют процессу выполнять несколько задач одновременно, что улучшает общую производительность и эффективность использования процессора.

・Экономия ресурсов, тк создать новый поток проще и менее ресурс затратно, чем новый процесс.

・Более эффективно распределяют процессорное время между задачами (**Диспетчеризация процессорного времени**) (процессорное время- это временной интервал, в течение которого центральный процессор выполняет задачу или обрабатывает инструкции программы (секунды, миллисекунды, микросекунды и тд))

*- особенности реализации потоков в unix/linux*

В **Linux** поток и процесс одно и тоже для ядра. Они оба являются задачами (tasks) для ядра.

Linux различает только 2 вида процессов: процессы у которых есть общая память(делят еще и ресурсы) и у которых нету общей памяти(потоки которые изолированы друг от друга).

**Fiber (волокна)** – собственная реализация потоков делающаяся не через ОС, а через приложения (сами делаем), а не операционной системой. Они могут использоваться в различных операционных системах, включая Linux, Windows и другие.

**Квантование в контексте Fiber** – это процесс управления временем выполнения различных Fiberвнутри одного потока. В отличие от потоков, которые могут быть принудительно прерваны операционной системой, Fiberсами определяют моменты переключения контекста.

– *чем отличается асинхронность от многопоточности?*

**Многопоточность** – модель (парадигма) программирования, которая заключается в представлении процесса в виде набора одновременно выполняемых потоков.

• созданием и управлением несколькими потоками, каждый из которых может выполняться параллельно.

• Требует дополнительных ресурсов для создания и управления потоками

• Идеальна для параллельных вычислений и задач, которые могут быть разбиты на независимые подзадачи.

**Асинхронность** – это способ выполнения задач, при котором задачи запускаются и выполняются в фоновом режиме, позволяя основной программе продолжать работу без ожидания завершения этих задач.

• использованием обратных вызовов

• Эффективно использует ресурсы, так как не требует дополнительных потоков, снижая накладные расходы.

• Идеальна для ввода-вывода и других операций, которые могут блокировать основной поток.

*- поясните понятие «потокобезопасный объект»*

Это такой объект, который, позволяет его **использовать не используя синхронизацию**. Т.е объект, который может безопасно использоваться несколькими потоками одновременно без необходимости внешней синхронизации. Внутри таких объектов уже встроены механизмы синхронизации.

Пример: Встроенные в API объекты, такие как критические секции или мьютексы, могут быть потокобезопасными, позволяя нескольким потокам обращаться к ним одновременно.

Heap (куча) – это область памяти, используемая процессом для динамического выделения памяти. Она может одновременно использоваться несколькими потоками благодаря встроенной поддержке потокобезопасности в API.

Когда создается процесс, начальные размеры кучи и стека задаются в линкере. Эта информация записывается в PE-блок исполняемого файла. Загрузчик операционной системы считывает PE-блок при запуске процесса и выделяет указанные размеры памяти для стека и кучи.

**PE (Portable Executable) блок** – это структура данных, используемая в Windows для определения исполняемых файлов и DLL. Она содержит информацию о размере стека и кучи, которая используется загрузчиком для выделения соответствующих ресурсов при запуске программы*.*

API для работы с потоками, API для синхронизации потоков.

**Механизмы синхронизации потоков в Windows:**

*- Critical section(только если находятся в 1 процессе);*

*- Mutex;*

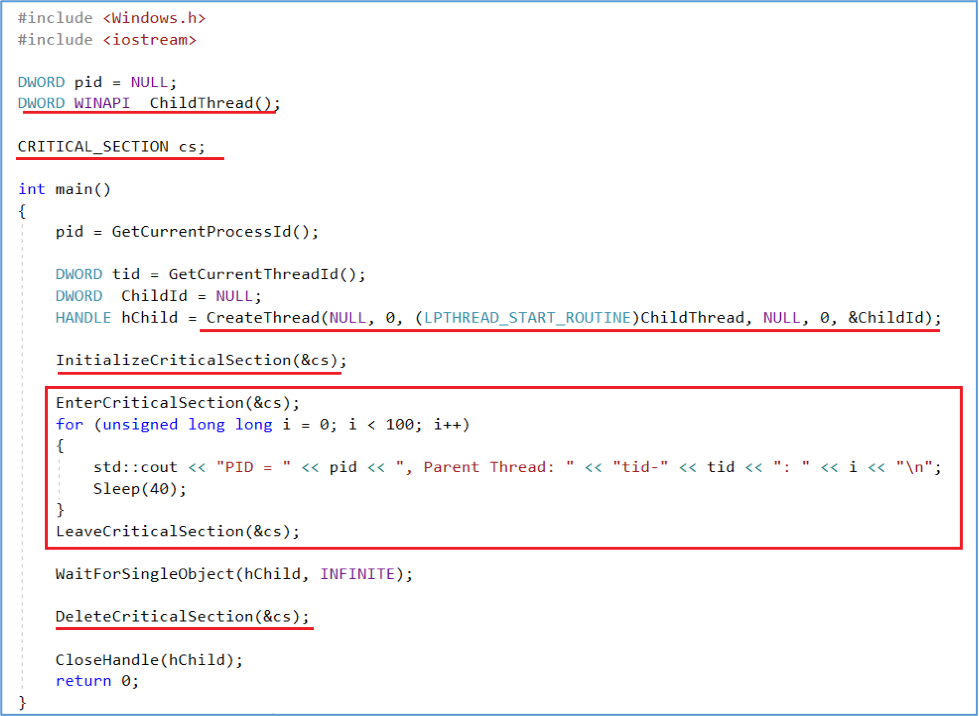
*- Semaphore;*

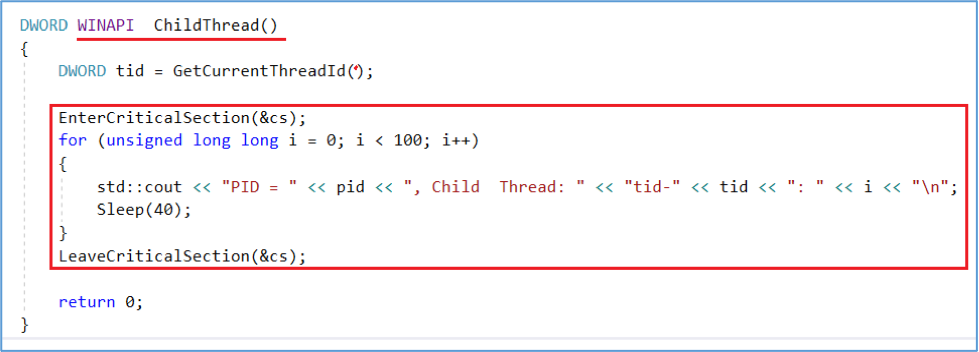
*- Atomic operation (interlocking function)*

*- Event;*

*- Writable timer.*

* 1. **Критическая секция(*Critical section*)** – механизм, синхронизации нескольких потоков одного процесса и не требует системного вызова=> не является объектом ядра ОС.



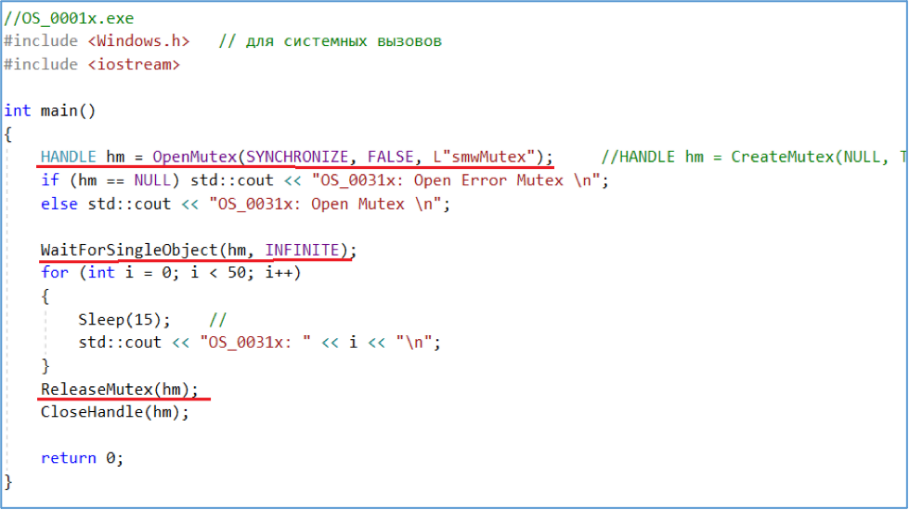


* 1. **Мьютекс** (англ. mutex, от **mut**ual **ex**clusion — «взаимное исключение») — это базовый механизм синхронизации. Он предназначен для организации взаимоисключающего доступа к общим данным для нескольких потоков с использованием барьеров памяти (для простоты можно считать мьютекс дверью, ведущей к общим данным). Является объектом ядра ОС.

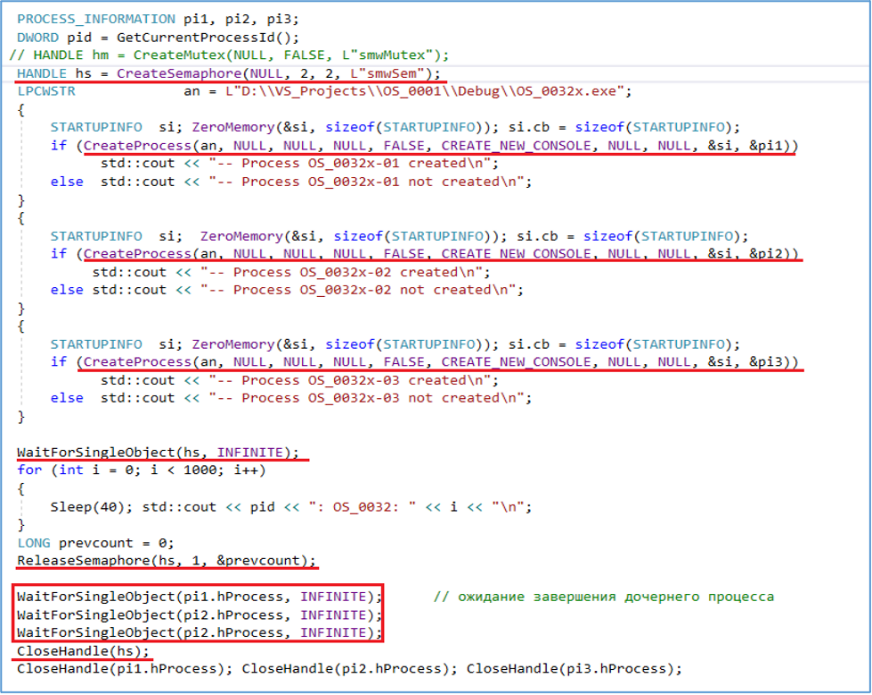
Как только хотя бы один процесс запрашивает владение мьютексом, он переходит в несигнальное состояние и остается таким до тех пор, пока не будет освобожден владельцем. Такое поведение позволяет использовать мьютексы для синхронизации совместного доступа нескольких процессов к разделяемому ресурсу.

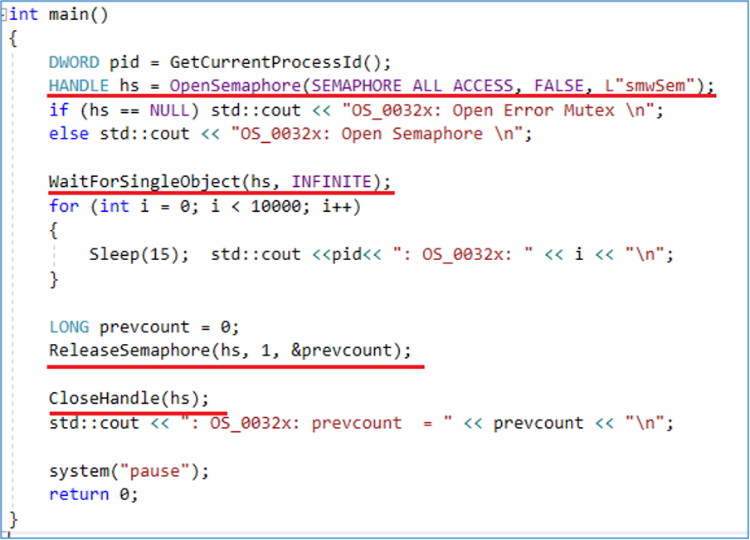
В ОС семейства [Windows](https://ru.wikipedia.org/wiki/Microsoft_Windows) разница между мьютексом и критической секцией в том, что мьютекс является объектом ядра и может быть использован несколькими процессами одновременно (**т.е. может выполнять синхронизацию процессов),** критическая секция же принадлежит процессу и служит для **синхронизации только его потоков**.





* 1. **Семафор (Semaphore)** представляет собой счетчик, содержащий целое число в диапазоне от 0 до максимальной величины, заданной при его создании. Когда поток (или процесс) захватывает ресурс, счетчик семафора уменьшается. При достижении семафором значения 0 он переходит в несигнальное состояние, при любых других значениях счетчика его состояние — сигнальное. Такое поведение позволяет использовать семафор в качестве ограничителя доступа к ресурсу, поддерживающему заранее заданное количество подключений.





* 1. **Событие (Event) -** объекты-события используются для уведомления ожидающих нитей (потоков, процессов) о наступлении какого-либо события. Различают два вида событий - с ручным и автоматическим сбросом.

- Ручная событийная блокировка требует явного вызова функций SetEvent и ResetEvent для установки и сброса состояния события, в то время как автоматическая событийная блокировка автоматически устанавливает состояние события при захвате и сбрасывает его при освобождении.

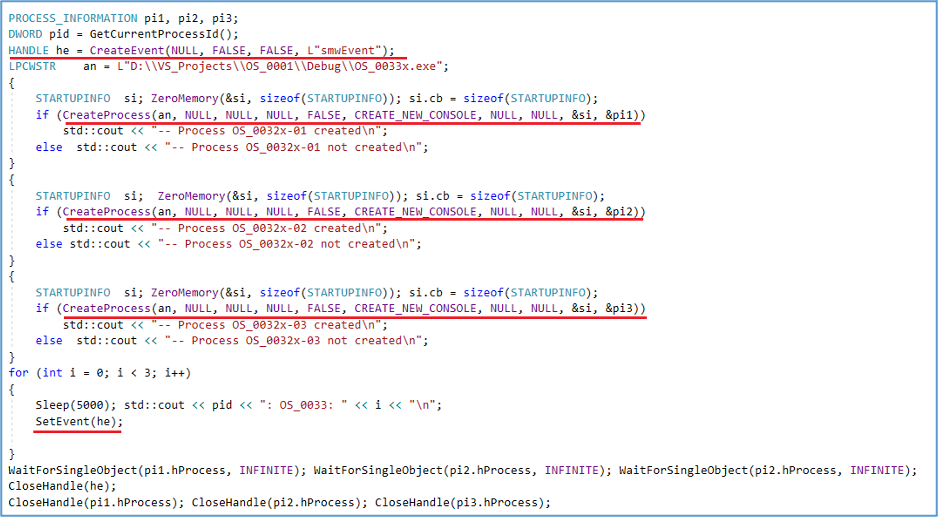
- Ручная событийная блокировка имеет два состояния (сигнальное и незавершенное), в то время как автоматическая событийная блокировка имеет только одно состояние (сигнальное).

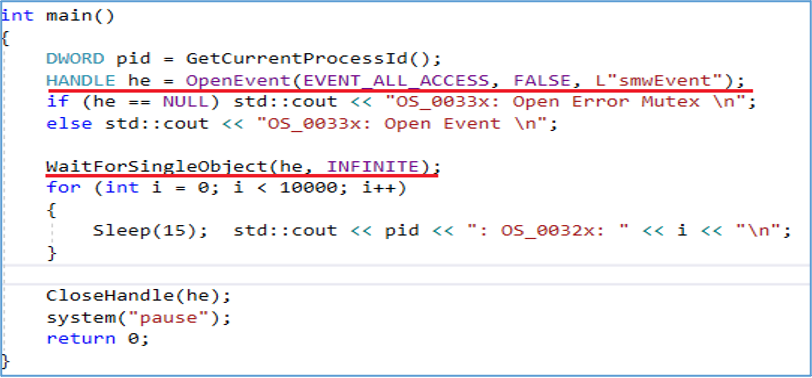
- Ручная событийная блокировка может использоваться для синхронизации между несколькими потоками или процессами, в то время как автоматическая событийная блокировка обычно используется для синхронизации внутри одного потока.

Ручной сброс осуществляется функцией ResetEvent. События с ручным сбросом используются для уведомления сразу нескольких нитей.

автосбросом уведомление получит и продолжит свое выполнение только одна ожидающая нить, остальные будут ожидать дальше.

С помощью **Event** можно осуществить барьер, который будет блокировать другие потоки, пока не произойдёт какое-то событие. Барьер преодолеет только 1 поток. Чтобы прошёл следующий поток, необходимо снова сделать setEvent. Есть специальная функция pulse, которая пропустит все потоки через барьер, которые подошли к этому барьеру в данный момент.





* 1. **Waitable timer (Ожидающий таймер)** - объекты ядра, которые предназначены для отсчета промежутков времени. Окончание временного интервала определяется по переходу таймера в сигнальное состояние. Момент перехода таймера в сигнальное состояние определяется одной из ожидающих функций. Таймер ожидания переходит в сигнальное состояние по завершении заданного интервала времени.

Таймер ожидания может быть либо синхронизирующим, либо сбрасываемым вручную уведомляющим (manual-reset notification timer) таймером. Синхронизирующий таймер связывается с функцией вызова, тогда как по сбрасываемому вручную уведомляющему таймеру используется функция ожидания.

Для его создания используется функция CreateWaitableTimer.

После получения идентификатора таймера поток может задать время его срабатывания функцией SetWaitableTimer.

**Атомарная операция** (*Atomic operation*) операция, которая либо выполняется целиком, либо не выполняется вовсе; операция, которая не может быть частично выполнена и частично не выполнена. Атомарные операции в Windows — это группа особых функций, названия которых начинаются с префикса Interlocked. Суть их в том, что каждая из них позволяет выполнить пару простых операций, но так, чтоб они выполняются атомарно, то есть как бы «одним махом», так что их выполнение не может быть прервано другим потоком. Проиллюстрирую их использование на конкретном примере.

Каждый, кто работает с объектом, вызывает в начале AddRef, увеличивая счётчик, а по окончании работы – Release, уменьшая его. Если счётчик при очередном вызове Release стал равен нулю, значит объект никому больше не нужен и Release удаляет его.

Проще всего вместо ++ и -- использовать атомарные функции InterlockedIncrement и InterlockedDecrement. Они гарантируют, что между чтением величины счётчика из памяти и записью туда нового значения поток не будет прерван.

С методом Release дело обстоит немного сложнее. Он должен не только поправить значение счётчика, но проверить, не обнулился ли он. К счастью, функции IntrlockedDecrement и IntrlockedIncrement позволяют сделать и это. Если в результате уменьшения или увеличения счётчика он станет нулём, возвращаемое значение тоже будет нулём.

Таким образом, IntrlockedIncrement и IntrlockedDecrement выполняет атомарно сразу две операции: изменяют счётчик на единичку и сравнивают результат с нулём. Это общее свойство всех Interlocked функций: все они позволяют атомарно выполнить не одну, а две или даже три простых операции.

1. Применение механизмов синхронизации в ОС Windows, API для синхронизации.

* *основа любой синхронизации;*
* *как написать собственную синхронизацию потоков в рамках одного процесса (типа критической секции)?*
* *как реализовать синхронизацию процессов в сети.*

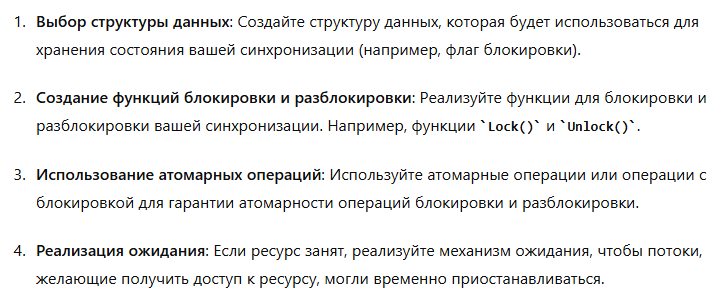
*- основа любой синхронизации;*

**Синхронизация** - это процесс управления доступом к ресурсам, который позволяет избежать конфликтов и обеспечить целостность какого либо ресурса при одновременном доступе нескольких процессов к нему.

**Основа любой синхронизации** – это контроль доступа к общим ресурсам с целью обеспечения корректного выполнения программ и не допущения повреждения ресурсов и появления конфликтов.

* *как написать собственную синхронизацию потоков в рамках одного процесса (типа критической секции)?*

Есть ассемблерное семейство команд BTC (Bit Test and Complement)(они предоставляют механизм для проверки битов и одновременного выполнения операции инверсии (переключения битов) в одной инструкции). Которые одновременно проверяет и переключает, что как раз и делает крит секция. Поэтому нужно писать такую команду, которая будет проверять переключилась или не приключилась

  
 *-как реализовать синхронизацию процессов в сети.*

Нужно брать посредника. Нужно делать сервер и 2 процесса подключается, а синхронизация внутри ОС, т.е фактически синхронизируем подключение. Почти все так серверные ОС так и делают.

1. Файловая система: логическая и физическая организация данных, определение файловой системы, отличие файловых систем, оглавление файловой системы, файлы, каталоги, основные функции файловой системы, буферы ввода/вывода, кеширование ввода/вывода, основные функции API файловой системы, маркер файла, текущая позиция файла, блокировка файлов, наблюдение за изменением в каталоге, особенности устройства файловой системы в Linux.

- *что значит «монтировать» файловую систему?*

- *для чего используется файловая система proc в linux*?

- *чем отличаются файловые системы?*

*логическая и физическая организация данных*

**Логическая** **организация** данных определяет, как файлы и директории представлены и организованы для пользователей и приложений

• Файлы - логические единицы данных, которые могут содержать текст, изображения, звуки и другие типы информации. Директории служат для организации файлов в иерархической структуре.

• Разрешения и доступ к данным: Логическая организация определяет, какие пользователи или приложения имеют доступ к файлам и директориям, а также какие операции с данными разрешены (чтение, запись, удаление)

**Физическая** **организация** данных отвечает за способ хранения данных на физическом носителе информации, таком как жесткий диск

• Блоки данных: Данные на носителе хранятся в виде блоков определенного размера, который является минимальной единицей чтения и записи.

• Секторы диска: Диск разделен на физические секторы, которые являются минимальными адресуемыми единицами для чтения и записи данных на диск.

*определение файловой системы*

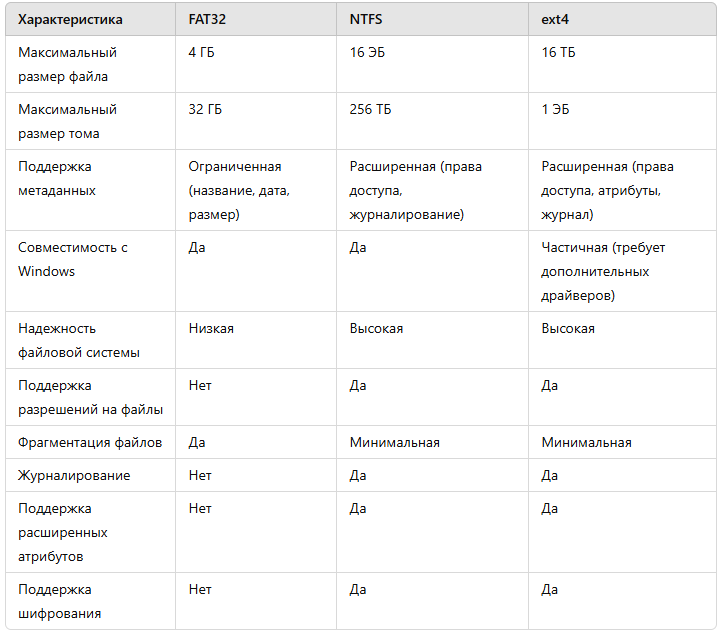
**Файловая система** - это способ организации данных на носителе информации, обеспечивающий удобный доступ к файлам и каталогам.

*отличие файловых систем*

**FAT32 (File Allocation Table 32)** ⎯ это файловая система, которая использует таблицу размещения файлов (File Allocation Table) для организации и управления данными на диске. Она была разработана компанией Microsoft

**NTFS (New Technology File System)** ⎯ это более современная и продвинутая файловая система, разработанная компанией Microsoft для операционных систем семейства Windows. Она предоставляет расширенные функции, такие как поддержка разрешений на файлы, журналирование, шифрование и сжатие данных.

**ext4 (Fourth Extended File System)** ⎯ это файловая система, используемая в операционных системах на базе Linux. Она является последней версией серии Extended File System (ext), предоставляющей поддержку расширенных атрибутов, журналирования, улучшенной производительности и надежности



**Журналируемые** — данный тип ФС сохраняет историю действий пользователя, а также план проверки системы в специальном файле. Особенности: устойчивость к сбоям и сохранение целостности информации.

**Не журналируемые** — не предусматривают хранение логов. Особенности: работают быстрее, но не гарантируют сохранность данных

*оглавление файловой системы*

**Оглавление файловой системы** - это структура данных, которая содержит информацию о расположении файлов и каталогов на диске. Она позволяет ОС быстро находить нужные файлы. Представляет собой таблицу или индекс, где каждая запись соответствует файлу или каталогу на диске. Каждая запись обычно содержит информацию, такую как имя файла, путь, размер, дата создания, разрешения доступа и т. д.

*файлы, каталоги*

**Файл** – набор логических записей.

**Каталог** – файл, содержащий информацию о месте расположения других файлов.

*основные функции файловой системы*

Основные **функции** файловой системы:

• создание/удаление каталогов,

• включение/исключение подкаталогов,

• включение/исключение файла в каталог,

• создание/удаление файла,

• открытие/закрытие доступа к файлу,

*буферы ввода/вывода*

**Буферы ввода/вывода (I/O buffers)** – это области памяти, используемые операционной системой для временного хранения данных, которые перемещаются между устройствами ввода/вывода.

– это области памяти для хранения физически считанных данных;

Необходимы для устранения несоответствия между физическим и логическим чтением/записью.

Сначала заполняется буфер кластерами (физически считанными данными), а затем осуществляется чтение логических файлов © Смелов

**Кластер** – это минимальная единица распределения данных на диске. Физически считанные данные обычно считываются кластерами и сохраняются в буфере ввода/вывода. Это уменьшает количество операций ввода/вывода и повышает производительность системы.

***кеширование ввода/вывода***

**Это техника**, при которой операционная система осуществляет перемещение в быстродействующую память, наиболее часто используемых данных (обычно упреждающее чтение (предварительно читают данные в кэш, основываясь на предположении, что они будут запрошены в ближайшем будущем.)).

В состав ядра OS входит специальная программа обеспечивающая кэширование данных – **менеджер кэша файловой системы**. Этот компонент отвечает за управление кэшированием данных файловой системы. В Linux, например, этот компонент известен как VFS Cache (Virtual File System Cache).

Кроме того кэширование осуществляет контроллер дисковода(Контроллер дисковода - это устройство, ответственное за управление доступом к жесткому диску. Он может иметь собственный кэш, в который данные записываются перед тем, как они физически записываются на диск)

*основные функции API файловой системы*

* **Delete file**
* **WriteFile**
* **ReadFile**
* **CopyFile**
* **MoveFile**
* **ReplaceFile** (заменяет файл в другом месте, сохраняя его атрибуты и текущую дату)
* **LockFile**(блокировка на определённый диапазон байтов в файле)
* **UnlockFile**
* **LockFileEx (с указанием типа блокировки: монопольная/общая)**

*маркер файла*

**Маркер файла (или файловый дескриптор)** – это уникальный идентификатор, который операционная система использует для идентификации открытого файла в процессе.

• Дескриптор файла (File Handle) Это числовое значение, которое представляет собой ссылку на открытый файл в рамках процесса. (для чтения, записи и получения доступа к файлу.

*текущая позиция файла*

**Текущая позиция файла (или указатель файла)** – это специальный указатель, который отслеживает текущую позицию в файле, который был открыт для чтения или записи. Он показывает, где следует начинать чтение или запись данных в файл.

Может быть изменен с помощью функций установки указателя позиции файла, таких как fseek() в C.

*блокировка файлов*

**Блокировка файла** – это механизм, используемый для ограничения доступа к файлу другим процессам во избежание конфликтов доступа и повреждения данных.

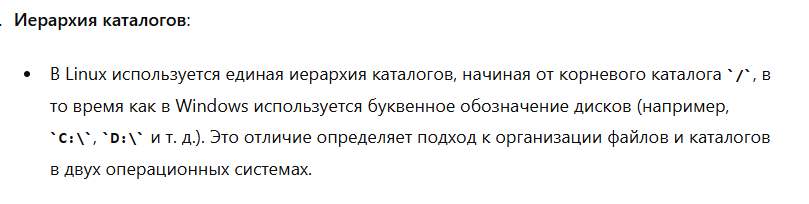
*наблюдение за изменением в каталоге*

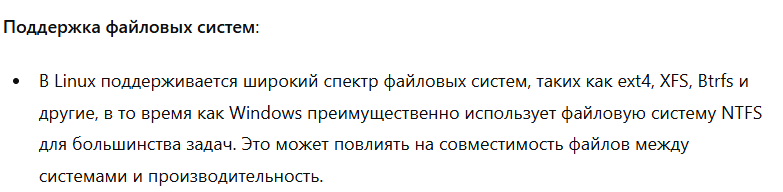
**Наблюдение за изменениями в каталоге** – это механизм, который позволяет отслеживать изменения в содержимом каталога операционной системы. В ОС Windows функция *ReadDirectoryChangesW*, позволяет получать уведомления об изменениях в каталоге, таких как создание, удаление или изменение файлов.

*особенности устройства файловой системы в Linux*

**FileSystem Hierarchy Standard (FHS)** – стандарт иерархии файловой системы. © Смелов

**FHS (Filesystem Hierarchy Standard)** - это стандарт, определяющий структуру каталогов в UNIX-подобных операционных системах, в том числе Linux. Он определяет расположение файлов, каталогов и установленных пакетов в системе, чтобы улучшить переносимость и совместимость различных дистрибутивов Linux.





*- что значит «монтировать» файловую систему?*

Если вы делаете это в Windows то связано с подключением сетевого диска.

В Linux надо взять ветку из внутри и подсунуть другую ветку, т.е происходит монтировка: какая то ветвь есть из дерева файловой системы, можно взять узел и перетащить другой узел. Можно делать в сети и это как раз и монтировка. В любой файловой системе такое есть.(Смелов)

**Монтирование файловой системы** означает подключение (или сопоставление) определенной области файловой системы к определенному месту в иерархии каталогов операционной системы. Это процесс, при котором файловая система становится доступной для чтения и записи через указанное место в файловой системе.

В Windows, монтирование часто связано с подключением сетевых дисков или удаленных ресурсов. Это позволяет пользователям получить доступ к файлам и папкам на удаленном сервере, как если бы они находились на локальном компьютере.

В Linux, монтирование может использоваться для подключения различных устройств и файловых систем, включая жесткие диски, USB-накопители, CD/DVD-диски и сетевые ресурсы.

*- для чего используется файловая система proc в linux?*

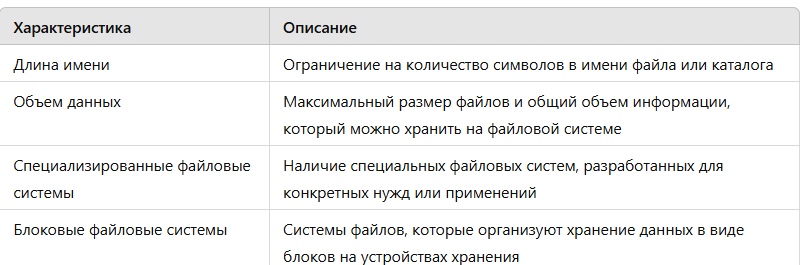
**Файловая система /proc в Linux** представляет собой виртуальную файловую систему, которая обеспечивает доступ к информации о текущем состоянии системы и ее процессах.

от обычных файловых систем, */proc не хранит данные на диске, а создается динамически ядром ОС* при ее загрузке.

В ней содержатся файлы и каталоги, представляющие информацию о процессах, загруженных модулях ядра, параметрах ядра и другие системные ресурсы. Она используется для мониторинга и управления процессами, настройки параметров ядра и отладки системы.

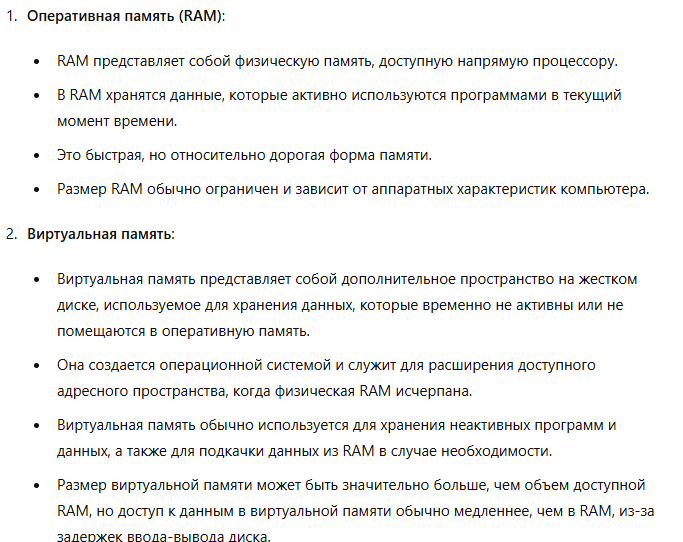
*- чем отличаются файловые системы?*

Длина имени, кол во инф которую можно записать, есть спец ф системы которые могут находится в ОС, блоковые файловые системы в линуксе.

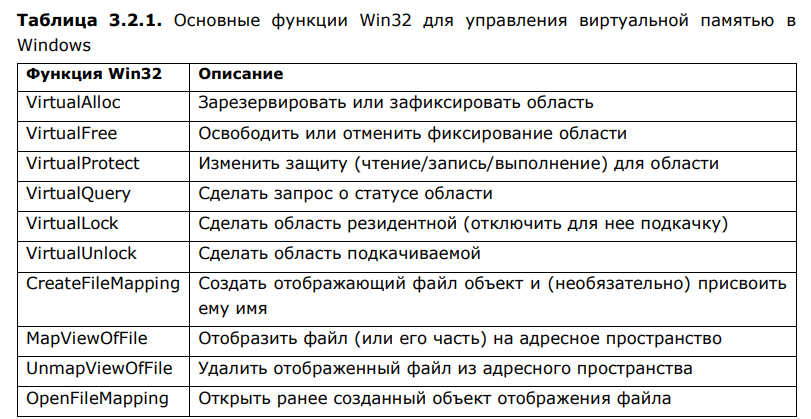


1. Работа с оперативной памятью в ОС Windows: API для работы с виртуальной памятью, API для работы с Heap.

**Оперативная память** - это физическая память, которая состоит из микросхем, подключенных к материнской плате. Она используется для хранения программ и данных, которые в данный момент выполняются или используются компьютером.

**Виртуальная память** - это логическая память, которая использует часть жесткого диска для имитации оперативной памяти. Она используется для хранения программ и данных, которые не помещаются в оперативную память или не нужны в данный момент 

**Функции API OS для работы с virtual memory**:



**Функции API OS для работы с heap**:

Каждый процесс имеет кучу по умолчанию, предоставляемую системой. Приложения, которые часто выделяют ресурсы из кучи, могут повысить производительность с помощью частных кучи.

Windows API (WinAPI):

Функции, связанные с кучей в Windows API, обычно начинаются с префикса Heap.

Примеры функций:

**HeapAlloc**: Выделяет блок памяти на куче.

**HeapFree**: Освобождает блок памяти, выделенный на куче.

**HeapCreate**: Создает новую кучу.

**HeapDestroy**: Уничтожает указанную кучу.

C Standard Library (stdlib.h) в языке C:

Для работы с **динамической памятью** в языке C можно использовать функции, такие как **malloc**, **calloc**, **realloc** и **free**.

Примеры функций:

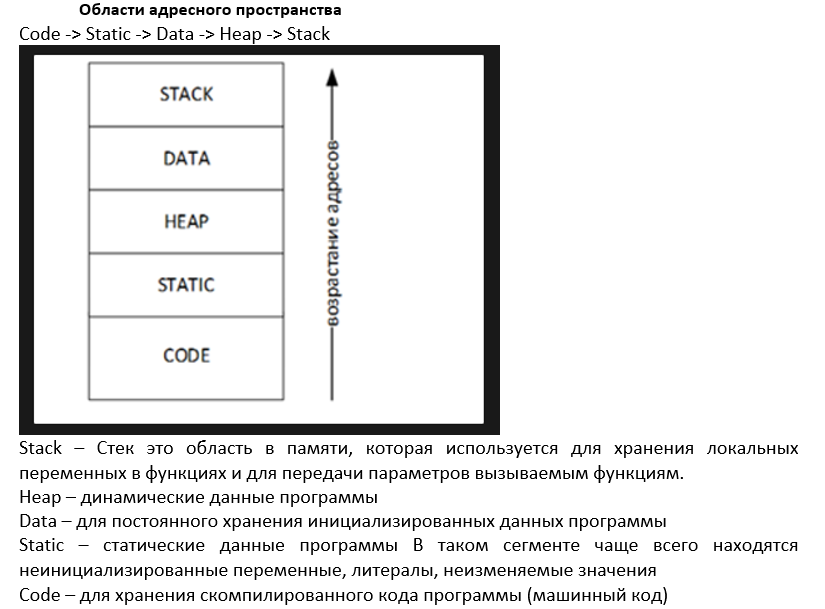
malloc: Выделяет блок памяти.

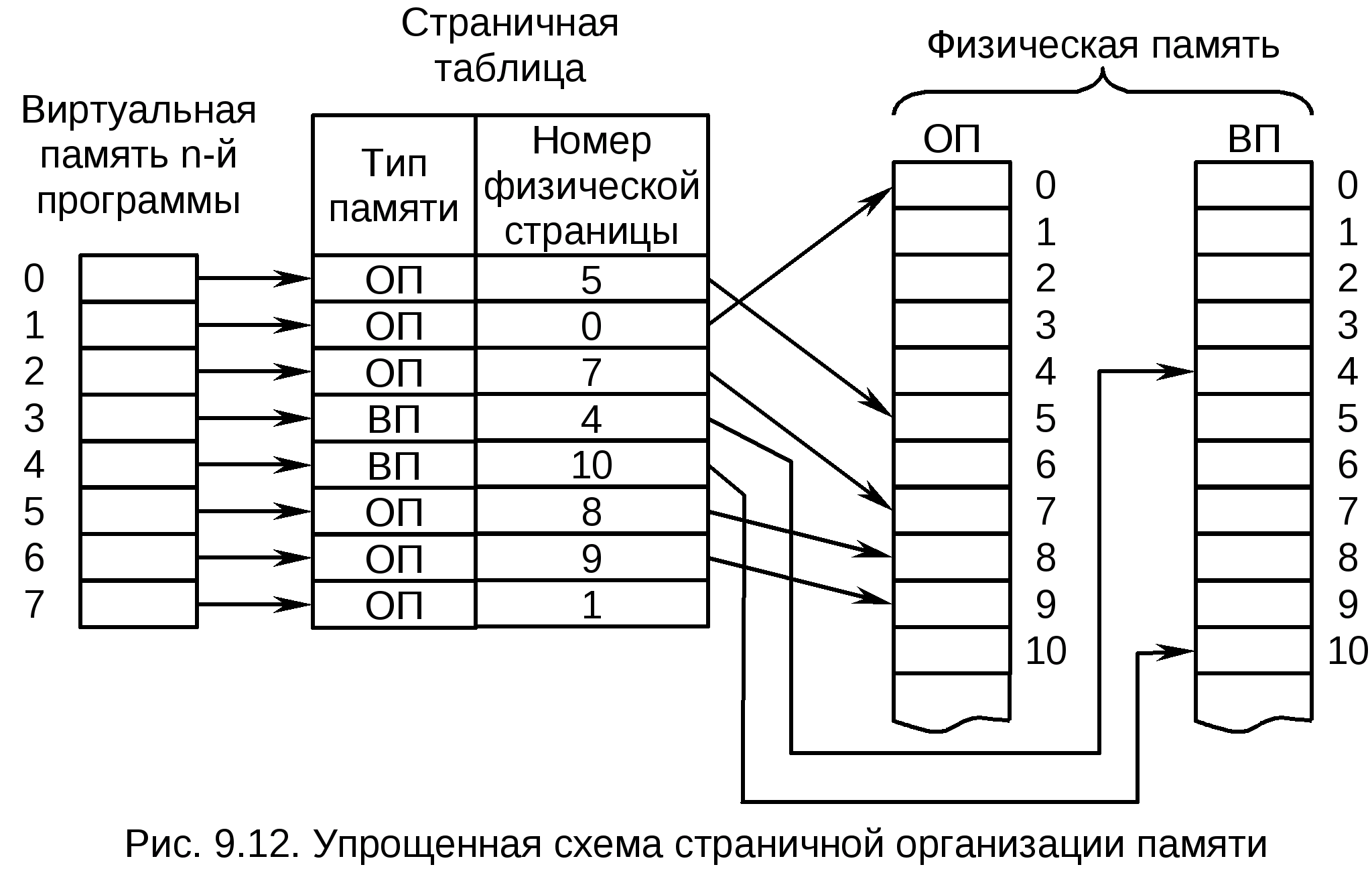
calloc: Выделяет и обнуляет блок памяти.

realloc: Изменяет размер ранее выделенного блока.

free: Освобождает выделенный блок памяти

**Куча** - это область оперативной памяти компьютера, в которой возможно в динамическом режиме выделять память для различных программных нужд. Куча представляет собой непрерывную область памяти, поделённую на занятые и свободные области (блоки) различного размера.





**Фрагментация в куче (heap**) — это состояние, при котором свободное пространство в куче разбито на множество небольших фрагментов, которые в сумме могут быть достаточно большими для выделения нужного объема памяти, но недоступны для выделения, так как они распределены по всей куче.

- *что такое «утечка памяти»?*

**Утечка памяти (memory leak)** - это ситуация, когда программа неправильно управляет выделенной ей оперативной памятью и не освобождает ее после использования. Выделенная память не освобождается, и с течением времени объем доступной памяти уменьшается, что может привести к нехватке памяти и снижению производительности всей системы.

"**упаковка**" и "**распаковка**" обычно используются в отношении сжатия или оптимизации использования памяти

Упаковка (Packing):

- Определение: Упаковка относится к процессу компактного распределения данных в памяти, чтобы уменьшить фрагментацию и максимизировать использование доступного пространства.

- Пример: Если в куче есть несколько блоков памяти, которые могли бы быть скомбинированы в один больший блок, это может быть выполнено операцией упаковки.

Распаковка (Unpacking):

- Определение: Распаковка — это процесс освобождения памяти, который может включать в себя перемещение данных, чтобы освободить фрагментированное пространство.

- Пример: Если в результате освобождения блока памяти создается свободное пространство, оставшееся после удаления, то эта область может быть распакована и объединена с другими свободными областями.

- *зачем придумали Heap?*

**Гибкость выделения памяти**: Heap позволяет выделять память в маленьких фрагментах, не привязываясь к размеру страниц виртуальной памяти. Прим от Смелова: если надо юзать 20 байт все равно выделится страница, а эт не рационально. А heap как раз позволяет выделять маленькие фрагменты меньше страницы и это рационально.

**Динамическое управление памятью**: Память, выделенная в heap, может быть освобождена и переиспользована в любое время во время выполнения программы. Это позволяет оптимально использовать ресурсы системы и предотвращать утечки памяти.

Heap представляет двухсвязный список из 2 частей:

**1. Занятая область памяти**: часть списка, в которой расположены уже выделенные блоки памяти, используемые программой.

**2. Свободная область памяти**: Эта часть списка содержит блоки памяти, которые еще не были выделены программой и могут быть использованы для выделения новых блоков.

Heap легко использовать, но надо быть осторожным тк при работе с указателями часто появляются сообщения, что сломался Heap, это из-за того что чет не записали и указатели потерялись.

1. Механизм отображение файлов в памяти:

последовательность системных вызовов Windows для создания образа файла в оперативной памяти,

**File Mapping** ⎯ механизм, позволяющий отобразить файл или его фрагмент в оперативной памяти. Этот образ можно использовать далее в нескольких процессах и для межпроцессного обмена данными.

В windows OC поддерживают механизм работы с файлами через файловые проекции, механиз подразумевает что файл проецируется в виртуальное адресное пространство процесса.

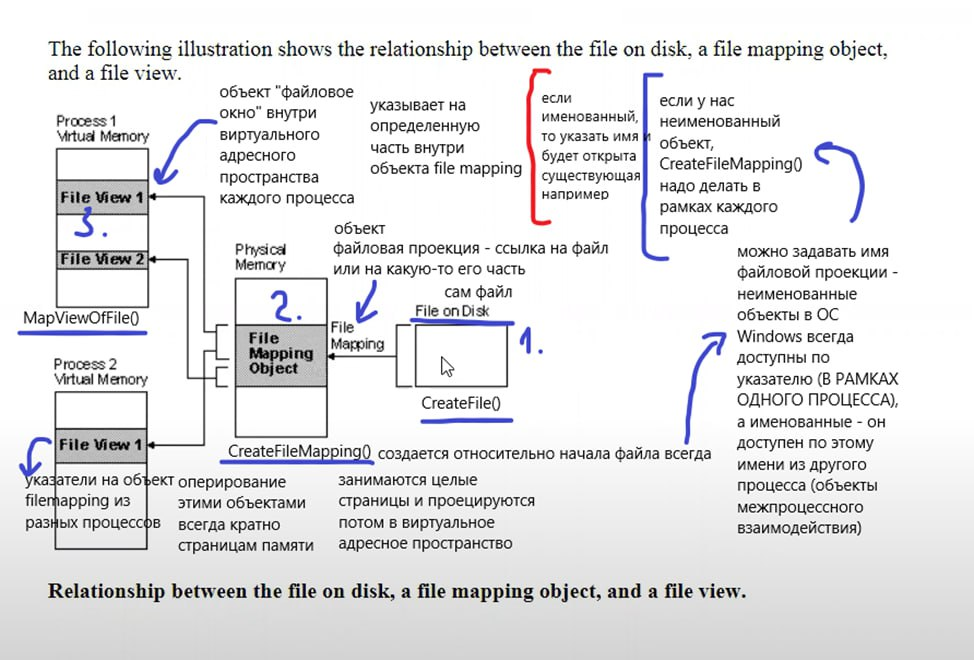
То есть работа с файлом в конечном счете будет осуществляться через указатель на память(lpvoid,int\* и тд).

**Файловая проекция** – это ссылка на весь файл или на некоторую часть, которая проецируется в физическую память. То есть вот эта часть данных будет находится в физической памяти.

Далее для того чтобы получить доступ к File Mapping объекту (он может быть довольно таки большим 50-100мб), необходимо создавать **«файловое окно» File View** (файловое представление).

Это уже объект внутри каждого виртуального адресного пространства процесса, который является указателем на определенную часть File Mapping объекта

Оперирование этими объектами (file maping, file view) кратно страницам памяти. Потому что занимаются целые страницы и проецируются в виртуальное адресно пространство процесса.



После того как мы получили HANDLE File mapping, у нас становится доступным создание файловых окон которые будут указывать на эту файловую проекцию из разных процессов.

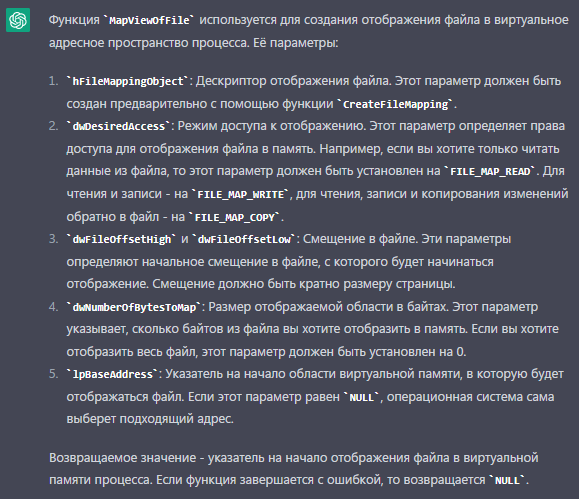
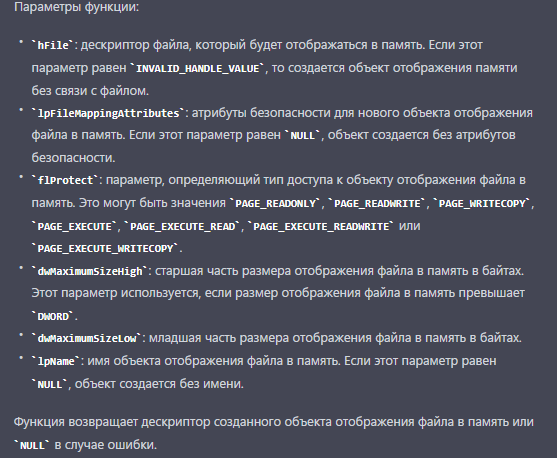
Так как у нас есть HANDLE , то это означает, что создать файловую проекцию нужно будет В КАЖДОМ ПРОЦЕССЕ ОТДЕЛЬНО, если при создании мы укажем имя существующей проекции, она будет просто открыта, тк она уже существует и новая создаваться не будет.

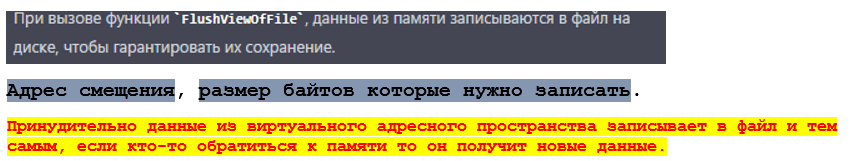
Функция CreateFileMapping относится к каждому процессу отдельно. Если в одном процесса создали, то в другом мы тоже должны ее вызывать, если хотим получить доступ к той же файловой проекции, НО С ПЕРЕДАЧЕЙ УЖЕ СУЩЕСТВУЮЩЕГО ИМЕНИ (если именована, если нет то создаем новую).

После того как файловая проекция создана одним из процессов у нас появляется возможность получить к ней доступ по имени. Мы можем создавать файловые окна.

После того как **мы получили файловое окно**, нам возвращает LPVOID указатель, возвращает УКАЗАТЕЛЬ НА ПАМЯТЬ, то есть у нас этот файл через окно спроецирован в виртуальное адресное пространство, это обычный указатель на память (в адресном пространстве),

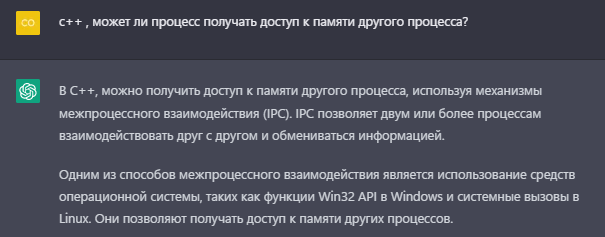
**CreateFileMapping()**





использование образа файла, как средства межпроцессного взаимодействия.

Это может использоваться как ipc процессов (процесс создавший проекцию может записывать, а другой читать). Может использоваться как канал взаимодействия между процессами (для обмена инфы).

-Процесс создает образ файла, используя функцию создания файловой проекции (например, в Windows - *CreateFileMapping*, в Linux - *mmap*). Затем другие процессы могут открыть тот же образ файла и работать с данными, проецированными в общее адресное пространство.

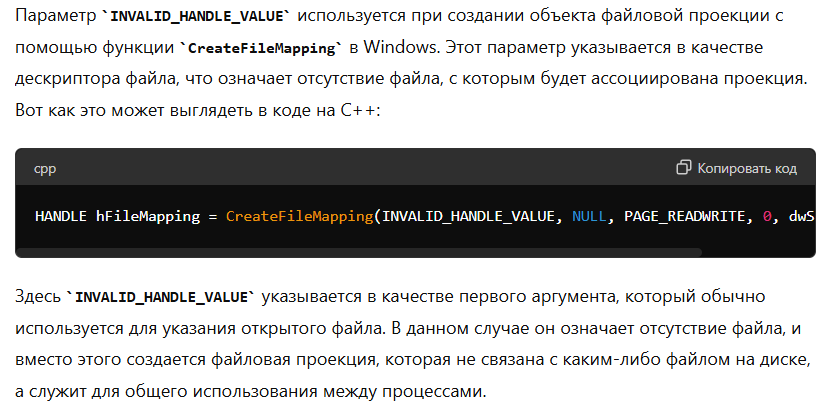
Объекты ОС которые могут быть *именованы* и *не* *именованы*:

***Неименованные*** объекты в ОС всегда доступны только по указателю (только в рамках одного процесса), если объект ***именованный*** он может быть доступен по имени из другого процесса (как правило это объекты для межпроцессного взаимодействия).

Если мы будем осуществлять межпроцессное взаимодействие, то указываем имя файловой проекции !! если не нужно то просто передаем NULL

- *как можно сделать общую память для процессов без файла?*

Если указывается Invalid\_Handle\_Value и у вас создастся общая область памяти без файла.

Этот метод создает область памяти, которая не связана с конкретным файлом, и может быть использована как общая память между различными процессами. Эта область памяти будет отображена в виртуальное адресное пространство каждого процесса, который откроет эту проекцию, что позволяет процессам обмениваться данными.

1. Динамически вызываемые библиотеки: структура DLL-библиотеки, экспорт функций, загрузка динамической библиотеки, динамический вызов функций динамической библиотеки, создание и применение библиотеки импорта.

* *что такое «позднее связывание»?*
* *как организовать общую память для двух DLL?*
* *зачем нужны библиотеки импорта?*

*Как устроена? Что такое dll main. В каких случаях вызывается dll main (4 случая). Что такое позднее связывание и раннее. Как организовать общую память для 2 dll (в одной dll когда запускаете есть ли map view) (ну или через статическую память). Зачем нужна библиотека импорта (чтобы упростить работу, чтобы не писать loadlibrary и getprocaddr (внутри нее выполняется)). Внутри нее это выполняется. Библ импорта это делает внутри себя*

DLL (Dynamic Link Library) - это динамически подключаемая библиотека, которая содержит код и данные, которые могут использоваться несколькими приложениями одновременно. Она загружается в память при запуске приложения, и вызовы к функциям из библиотеки выполняются через указатели на функции.

**Структура DLL-библиотеки:**

- необязательную часть кода, которая отвечает за инициализацию и очистку библиотеки;

- набор подпрограмм библиотеки;

- явное указание, какие подпрограммы должны экспортироваться из библиотеки.

**Загрузка динамической библиотеки:**

* + **Явная** (через LoadLibrary, а в таблицу экспорта будут заносится только те функции, переменные и т.д., которые мы в программе загрузим явным образом)
  + **Неявная** (через свойства проекта, в таблицу экспорта будут занесены ВСЕ функции из DLL)

*Определение смелова:* DLL– программный модуль, который может быть загружен динамически и содержать функции и данные. Механизм проецирования – один и тот же экземпляр DLL используется несколькими процессами (код – общий, данные по отдельности).

*DLL библиотека содержит в себе функции, обеспечивает позднее связывание (связывания на этапе выполнения). Параллельно с dll создается статическая библиотека (он имеет ввиду библиотека импорта). Статическая библиотека обеспечивает раннее связывание, считается с помощью линтера*

*В ОС фрэймворк представлен в виде динамических библиотек (фреймворк ос представлен библиотеками и функциями, которые предоставляются программисту). И у каждой библиотеки есть своя библиотека импорта.*

*В любое приложение которое работает на ОС, в него прикомпоновываются библиотеки с пом которых мы будем осущ обращения к ядру ОС библиотеки фрэймворка ОС.*

*Особенности dll:*

1. *загружается 1 раз (LoadLibrary загрузка библиотеки), все остальные случаи исп мэппинг библиотеки на адресное пространство процесса*

*Любое приложение состоит из секций (стека, данных и т.д.). Некоторые секции остаются общими, а некоторые секции создаются для каждого процесса.*

*Библиотека содержит точку входа – функция dllMain. Принимает параметры: хэндл – число, которое является номером объекта ядра ОС (когда мы создаем файл, процесс, поток возвращается хэндл). Это то число которое означает номер объекта внутри ядра ОС. И когда мы вызываем системный вызов и указываем хэндл, ядро понимает какой объект мы имеем ввиду. Так вот, длл когда загружается она тоже создает объект ядра ос.*

*Все что может использоваться между разными процессами, создается объект ядра ОС. Второй параметр это причина вызова (4 штуки).* **DLL\_PROCESS\_ATTACH -** когда DLL загружается в адресное пространство процесса

**DLL\_PROCESS\_DETACH -** когда DLL выгружается из адресного пространства процесса

**DLL\_THREAD\_ATTACH -** когда создается новый поток в процессе, к которому уже подключена DLL

**DLL\_THREAD\_DETACH-** когда поток завершает работу в процессе, к которому подключена DLL

*Понятное дело, что dllMain у каждого процесса свой.*

*LoadLibrary и FreeLibrary.*

*Функции нужно экспортировать.*

*2 способа экспорта функции:*

1. *extern “C” \_\_declspec(dllexport) int Sum(int x, int y) {return x + y}*

*extern “C” – указатель компилятору чтобы он не искажал имя функции Sum. Имя функции включал идентификатор и через спец знаки типы параметров.*

*Эта херня сделана чтобы в си плас плас была перегрузка функций, но нам будет неудобно вызывать функцию с таким идентификатором, поэтому мы говорим компилятору что нам нужен просто идентификатор Sum (т.к. в си перегрузки нет и имена не искажаются)*

*2)через .def файл. Вижла считывает этот файл, подает на вход компоновщика, и он указанные функции экспортирует*

*GetProcAddress(хэндл loadlibrary, “Имя функции”) – получение адреса функции*

*ДРУГОГО СПОСОБА ПОЛУЧИТЬ АДРЕС ФУНКЦИИ ИЗ ДЛЛ НЕТ. ТОЛЬКО ЧЕРЕЗ LOADLIBRARY И GETPROCADDRESS. И ЭТА ФУНКЦИЯ ДОЛЖНА БЫТЬ ЭКСПОРТИРУЕМА*

*Библитека импорта – статическая. Компонуется на этапе компоновки*

*#pragma comment(lib, “путь к .lib”)*

*Extern “C” int Sum(int,int);*

*В библиотеке импорта происходит loadLibrary, FreeLibrary, GetProcAddress. Просто все это обернуто в функцию (Sum). Т.е. внутри функции Sum есть getProcAddress и loadlibrary, извлекает эту функцию. Т.е. внутри статической библиотеки выполняется getProcAddress и loadlibrary. Просто все это скрыто.*

1. Спецификация COM: понятие позднего связывания программных модулей, COM-интерфейс, стандартные COM-интерфейсы, структура COM-клиента, структура COM/DLL-сервера, экспортируемые стандартные функции, регистрация COM/DLL-сервера.

*- для чего нужен COM? Какие есть функции которые экспортируются. Для чего каждая нужна. Для чего фабрика классов. Как устроен ком компонент. Где создается и когда удаляется. Какие счетчики поддерживаются. Как регистрация осуществляется (в dll есть код который регистрирует). Ком – правило написания кода. Правило написания приложений которое было принято в майкрасофт.*

*Для чего нужен ком?*

*ЛК: COM – спецификация разработки приложения в виндоус. ОС виндоус придумали стандарт разработки приложений. Обеспечивает позднее связывание. Раньше технология называлась оле32. COM – модель ПО. Впринципе может применяться во всех ОС. Задача ком программирования – разработка специального ПО которое соответсвутет модели КОМ.*

*КОМ программирование – компонентное программирование.*

*У каждого компонента есть идентификатор (CLSID). Идентификатор построен по алгоритму GUID (128 битное число которое с очень низкой вероятностью может повториться).*

*В результате у нас есть ком объекты и ком экземпляры.*

*КОМ компонент и КОМ объект это если что одно и то же, это что-то вроде класса в ООП. А вот КОМ экземпляр это уже как объект у класса в ООП. Может быть выбрано 2 типа компонентов, либо будет находиться внутри длл, либо внутри exe. Т.е. в качестве капсулы для наших компонентов.*

*Многокомпонентная длл -–длл с несколькими ком компонентами. Ну и эту длл с компонентами называют ком сервером*

все константы и прототипы функций необходимые для поддержки COM-программирования становятся доступными с помощью **#include <objbase.h>**

3 типа серверов: **CLSCTX\_INPROC\_SERVER (длл внутри процесса), CLSCTX\_LOCAL\_SERVER(**EXE-сервер за границами процесса, но та том же компьютере**), СLSCTX\_REMOTE\_SERVER (**EXE-сервер на удаленном компьютере**)**

**КОМ клиент – программный модуль который создает ком экземпляр и использует его. Он обращается к ком серверу, в рамках его создает экземпляр .** КОМ объекты могут быть клиентами, т.е. ком объект обращается к другому ком объекту**.**

1. При создании объекта посредником между COM-клиентом и COM-сервером выступает библиотека **OLE32.DLL** (библиотека импорта **OLE32.LIB**).

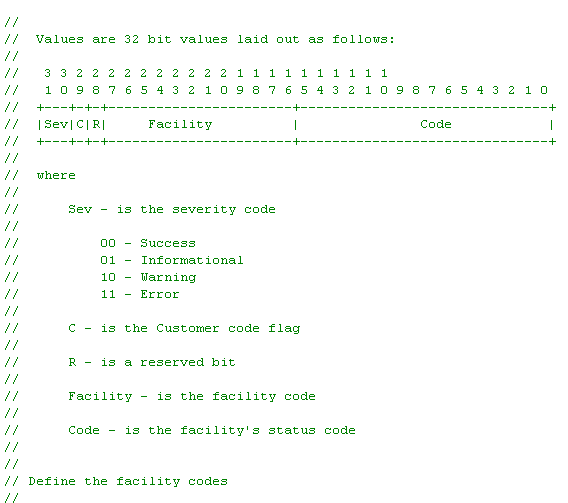


Все функции оле32 длл должны возвращать структуру HRESULT (HRESULT размер 32 бит

первый бит указывает на успешность выполнения функции

следующие 15 бит хранят информацию о типе ошибке

и последние 16 бит хранят специфическую информацию об ошибке) (кроме 2 Addref и Release)

****

*Любой ком объект характеризуется набором интерфейсов и clsid. Чтобы мы могли добраться к функциям ком объекта, нам нужно это делать через интерфейс. Ком компонент содержит 1 или несколько интерфейсов. Каждый интерфейс имеет свой GUID.*

*Каждый интерфейс характеризуется CLSID и типом контейнера в котором он находится*

*Среди всех интерфейса есть 2 стандартных:* **IUnknown и IClassFactory. Объекты ком знают их идентификаторы, они в качестеве константы записаны в define. А еще все знают какие там методы вызываются, какие параметры передаются и что возвращается**

**Любой** *ком компонент обязан поддерживать* **IUnknown. Более того, каждый интерфейс должен быть унаследован от IUnknown**

1. **IUnknown:** имеет три метода: **QueryInterface** – получить адрес интерфейса по его идентификатору(первый параметр метода **QueryInterface это id, а второй это адрес, куда мы поместим результат функции**); **AddRef** (увеличить на 1 счетчик ссылок на интерфейс); **Release**(уменьшить счетчик ссылок на интерфейс).

*Когда мы вытаскиваем интерфейс, мы должны отметить что мы его забрали. А когда он нам больше не нужен мы должны сообщить об этом. Внутри экземпляра компонента хранится счетчик количество выбранных интерфейсов. Т.е. если счетчик равен 0 то можно выгрузить экземпляр.*

*Все методы интерфейса должны поддерживать* **stdcall соглашение (**аргументы передаются через стек, справа налево, очистку стека производит **вызываемая** подпрограмма**)**

**COM-клиент**: для создания COM-объекта и работы с ним должен знать: 1)**CLSID объекта**; 2) **тип DLL-сервера** (контейнера); 3) **ID** **интерфейсов** объекта



CoInitialize – инициализация библиотеки OLE32

*CoCreateInstance(CSLID, NULL, SERVER\_TYPE, ID\_INTERFACE, (void\*\*)&ID\_INTERFACE) – создание экземпляра объекта ком*

*SUCCEEDDED – марок что все ок*

*Через полученный интерфейс мы можем вызывать методы интерфейса*

*CoFreeUnusedLibraries() – завершение работы с OLE32*



Соответствие между clsid и dll находится в реестре

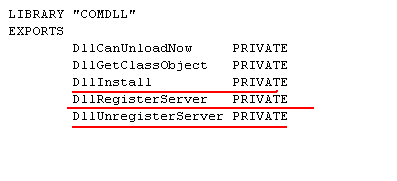
OLE32 как раз таки и берет информацию о clsid из реестра

Regsvr32 – регает/анргает com dll

Regsvr32 просто ищет функцию register/unregister в вашей dll и вызывает ее

Install – что-то еще попутно сделать с регистрацией/анрегистрацией

Dll com должен экспортировать след функции:



Писал чат:

HRESULT RegisterServer(

HMODULE hModule, // Дескриптор модуля DLL, используемый для получения пути к DLL

const CLSID& clsid, // Идентификатор класса (CLSID) компонента

const WCHAR\* szFriendlyName, // Читаемое имя компонента

const WCHAR\* szVerIndProgID, // Программный идентификатор версии (Version Independent ProgID)

const WCHAR\* szProgID // Программный идентификатор (ProgID)

);

* **hModule**: Дескриптор модуля DLL. Обычно это значение используется для получения пути к DLL, которая содержит COM компонент. Значение передается через функцию **LoadLibrary**.
* **clsid**: clsid
* **szFriendlyName**: Удобочитаемое имя компонента. Это строка, которая описывает компонент в более понятной форме, чем идентификатор CLSID.(чтобы в реестре было удобнее искать)
* **szVerIndProgID**: Программный идентификатор, независимый от версии (Version Independent ProgID). Это строка, которая используется для идентификации компонента без учета версии. Примером может быть **MyComponent**.
* **szProgID**: Программный идентификатор (ProgID). Это строка, которая включает версию компонента, например, **MyComponent.1**. Используется для регистрации конкретной версии компонента.

HRESULT UnregisterServer(

const CLSID& clsid, // Идентификатор класса (CLSID) компонента

const WCHAR\* szVerIndProgID, // Программный идентификатор версии (Version Independent ProgID)

const WCHAR\* szProgID // Программный идентификатор (ProgID)

);

* **clsid**: Идентификатор класса (CLSID) компонента. Это глобально уникальный идентификатор, используемый для идентификации COM компонента.
* **szVerIndProgID**: Программный идентификатор, независимый от версии (Version Independent ProgID). Используется для удаления всех записей, связанных с данным идентификатором.
* **szProgID**: Программный идентификатор (ProgID). Используется для удаления всех записей, связанных с конкретной версией компонента.

***DllCanUnloadNow*** *– можно ли выгрузить com сервер*

*Если счетчик ком экземпляров больше 0 или если сервер заблокирован (lockServer) то тоже нельзя выгрузить*

*Т.е. есть счетчик к-ва ком экземпляров, и счетчик ссылок на интерфейс*

***Счетчики:***

*Счетчик ссылок на интерфейс – можно ли выгрузить экземпляр*

*Счетчик к-ва ком экземпляров – можно ли выгрузить dll*

*Ну и счетчик lockServer*

*Любой экземпляр может заблокировать*

***DllGetClassObject*** *– вызывается через OLE32, вызывается по createInstance. Именно она создает экземпляр и через нее потом отдается интерфейс*

*DllGetClassObject(clsid, id\_iclassfactory)*

*Короче мы сюда передаем clsid объекта и iclassfactory и как бы говорим OLE32 “Верни мне указатель на фабрику классов, которая может создавать экземпляры объекта, с указанным clsid”*

***Фабрика классов*** *– экземпляр, который реализует интерфейс IClassFactory*

*Фабрика классов создает экземпляры на основе объектов*

*ДЛЯ КАЖДОГО ОБЪЕКТА ЕСТЬ СВОЯ ФАБРИКА КЛАССОВ*

*У IClassFactory есть методы: createInstance, lockServer*

*IClassFactory реализует интерфейс IUnknown*

*У CFactory тоже есть AddRef и Release*

1. Управление пользователями и группами пользователей в Windows: понятие дискреционной системы безопасности, типы Windows- пользователей, группы пользователей, возможности API управления пользователями и группами.

- ***как можно использовать API*** *Windows для авторизации в приложении?*

*Все что можно делать с апи и как это можно использовать(можно не писать никаких баз данных, ничего не придумывать а просто использовать то что уже есть в ОС). Можно создать свои собственные группы, можно делать аутентификацию, используя возможности ОС (если что в sql эта windows аутентификация она делается как раз через этот прикол)*

**Дискреционная система безопасности (Discretionary Access Control, DAC)** - это модель контроля доступа, которая определяет, какие пользователи или субъекты имеют право доступа к ресурсам системы.

**Дискреционное разграничение доступа к объектам** (Discretionary Access Control — DAC) характеризуется следующим набором свойств:

* все субъекты и объекты компьютерной системы должны быть однозначно идентифицированы;
* для любого объекта компьютерной системы определен пользователь-владелец;
* владелец объекта обладает правом определения прав доступа к объекту со стороны любых субъектов компьютерной системы;
* в компьютерной системе существует привилегированный пользователь, обладающий правом полного доступа к любому объекту (или правом становиться владельцем любого объекта).

Дискреционная система безопасности реализуется обычно в виде матрицы доступа, строки которой соответствуют субъектам компьютерной системы, а столбцы — ее объектам.

Элементы матрицы доступа определяют права доступа субъектов к объектам. В целях сокращения затрат памяти матрица доступа может задаваться в виде списков прав субъектов или в виде списков контроля доступа.

К достоинствам дискреционной системы безопасности относятся относительно простая реализация и хорошая изученность.

Недостатки. Прежде всего, к ним относится статичность разграничения доступа — права доступа к уже открытому субъектом объекту в дальнейшем не изменяются независимо от изменения состояния компьютерной системы.

При использовании дискреционной системы безопасности не существует возможности проверки, не приведет ли разрешение доступа к объекту для некоторого субъекта к нарушению безопасности информации в компьютерной системе.

Наконец, к недостаткам еще относится автоматическое назначение прав доступа субъектам.

В дискреционных системах есть понятия принципал. От лица которого исполняются все действия.

Юзеры могут быть принципалами. Всё, что выполняется, происходит от лица какого-то принципала.

API для справочника юзеров.

типы Windows-пользователей

Все учетные записи три типа:

администратор компьютера,

ограниченная учетная запись

гостевая учетная запись.

Пользователь с ограниченной учетной записью может выполнять операции со своим паролем (создание, изменение, удаление), изменять рисунок своей учетной записи, параметры настройки рабочего стола, а также просматривать файлы.

Учетная запись администратора позволяет выполнять следующие операции:

• Создание, удаление и редактирование учетных записей пользователей и своей;

• Операции со своим паролем;

• Установка и удаление программ и оборудования, редактирование их параметров и свойств;

• Чтение всех общих файлов;

• Внесение изменений в конфигурацию на уровне системы.

Что касается гостевой записи, то она формируется автоматически в процессе установки системы, и предназначена для сторонних пользователей, не имеющих на данном компьютере собственных учетных записей. Под учетной записью гостя нет доступа к файлам, папкам, параметрам и приложениям, которые защищены паролем. С помощью соответствующих ссылок можно включать/отключать учетную запись гостя.

**Группа ­–** совокупность пользователей с определёнными правами или **–** это набор учетных записей пользователей, которые объединены по какому-либо признаку. При этом отметим, что одна учетная запись пользователя может входить более чем в одну группу. Каждая группа имеет свою учетную запись и наделена своими правами и полномочиями. Эти права и полномочия передаются каждому члену группы.

**Группы пользователей**

* Администраторы. Неограниченный доступ.
* Операторы архива. Члены данной группы имеют права создания резервной копии даже тех объектов, к которым не имеют доступа.
* Опытные пользователи. Группа включена только для совместимости с предыдущими версиями
* Пользователи системного монитора. С помощью *Системного монитора* можно отследить использование различных ресурсов компьютером. А группа дает доступ к данному инструменту.
* Операторы настройки сети. Члены группы могут изменять параметры TCP/IP.
* Пользователи удаленного рабочего стола. Смогут входить в систему через удал. рабочий стол.
* Пользователи журналов производительности. Более полные права к *Системному монитору*.
* Криптографические операторы. Члены данной группы могут выполнять криптографические операции.
* Читатели журнала событий.

Привилегии можно назначать напрямую пользователю либо включать его в группу и назначать **группе привилегии**, тогда он будет владеть всеми этими привилегиями.

При авторизации каких-то операций, можно проверить принадлежность группы.

Встроенные пользователи: Администратор и др. (их нельзя удалить)

Встроенные группы: Все, Администраторы и др.

Пользователь наследует все свойства группы пользователей.

**Идентификация**: Это первый шаг, когда информационная система определяет, существует ли конкретный пользователь или нет. [Это делается с помощью идентификатора, который может быть логином, электронной почтой, номером телефона или другим признаком, который есть только у одного пользователя](https://skillbox.ru/media/code/identifikatsiya-autentifikatsiya-avtorizatsiya-chem-oni-razlichayutsya/)

1. [**Аутентификация**: Это второй шаг, когда пользователь вводит ключ (например, пароль или пин-код), подтверждая своё право на доступ к той или иной учётной записи и хранящейся в ней информации1](https://skillbox.ru/media/code/identifikatsiya-autentifikatsiya-avtorizatsiya-chem-oni-razlichayutsya/).
2. [**Авторизация**: Это последний шаг, который определяет, какие действия пользователь может выполнять после успешной аутентификации1](https://skillbox.ru/media/code/identifikatsiya-autentifikatsiya-avtorizatsiya-chem-oni-razlichayutsya/).

Субъекты безопасности – активные, те, что что-то делают (процессы, потоки)

Объекты – те, над кем выполняются действия (файлы и тд)

Возможности API управления пользователями и группами

**1. Имя текущего пользователя**

BOOL GetUserName

(

LPTSTR lpBuffer, // указатель на буфер

LPDWORD nSize // получает размер возвращает количество символов

);

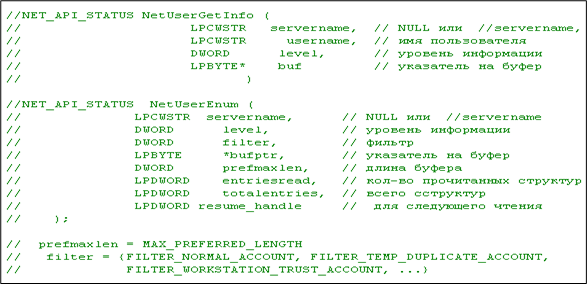
**2. Информация о пользователей**

****

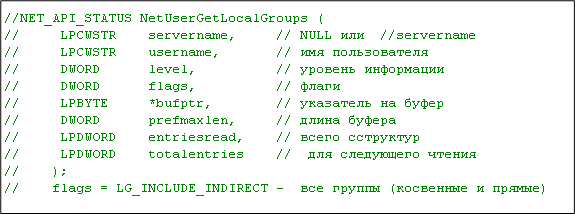
**3. Освободить память**

****

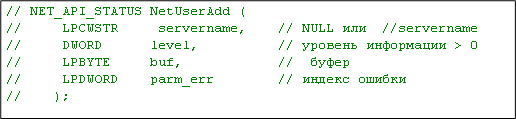
**4. Получить коллекцию пользователей**

****

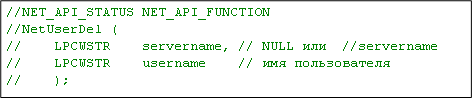
**5. Группы пользователя**

****

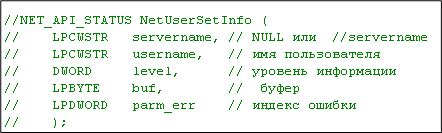
**6. Добавить пользователя**

****

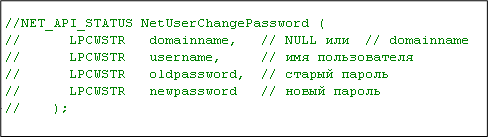
**7. Удалить пользователя**

****

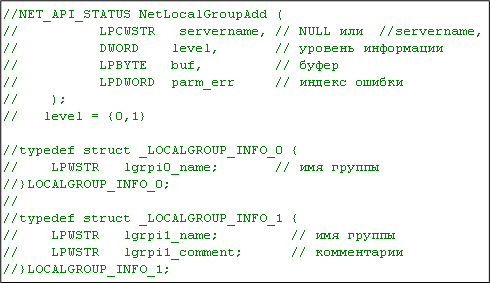
**8. Изменить информацию о пользователе**

****

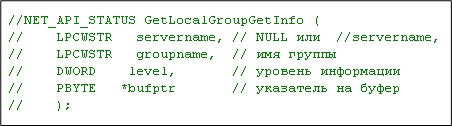
**9. Изменить пароль**

****

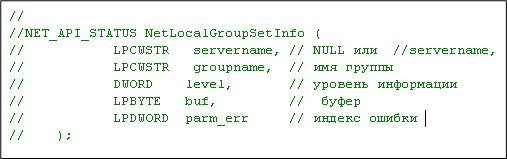
**10. Добавление группы**

****

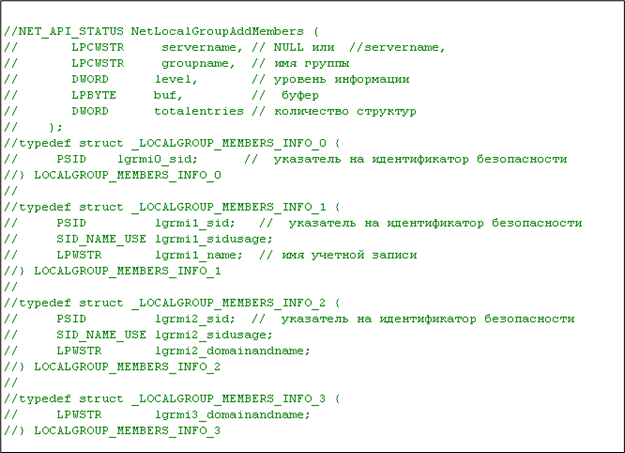
**11. Получить информацию о группе**

****

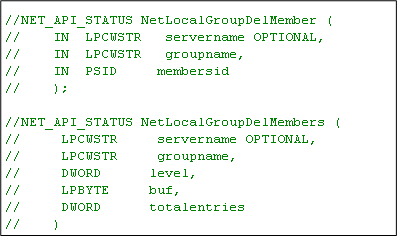
**12. Изменить информацию о группе**

****

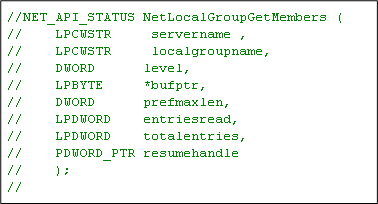
**13. Добавление членов локальной группы**

****

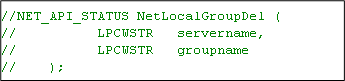
**14. Удаление членов группы**

****

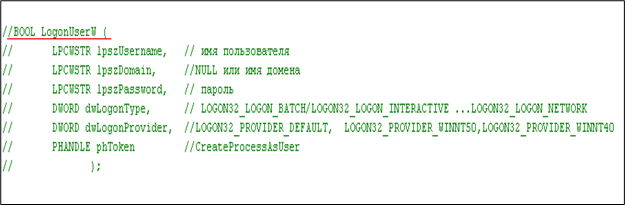
**15. Получить коллекцию членов группы**

****

**16. Удалить группу**

****

**17. Подключение пользователя**

****

*Дальше тоже чат:*

**Методы авторизации с использованием Windows API**

1. **Windows Integrated Authentication (WIA)**:
   * Это метод аутентификации, который использует учетные записи и группы Windows для управления доступом к ресурсам.
   * Полезен для приложений, работающих в доменной среде.
2. **API для управления учетными записями и группами пользователей**:
   * Использование функций Windows API для управления учетными записями пользователей и группами, проверки их прав и привилегий.
3. **Security Support Provider Interface (SSPI)**:
   * Это API Windows для реализации различных схем аутентификации, включая Kerberos, NTLM и другие.
4. **Token-based Authorization**:
   * Работа с токенами доступа, предоставляемыми системой после аутентификации пользователя.

### Пример использования Security Support Provider Interface (SSPI)

SSPI позволяет использовать различные протоколы аутентификации. Рассмотрим простой пример аутентификации с использованием NTLM.

#include <windows.h>

#include <sspi.h>

#include <stdio.h>

void AuthenticateWithSSPI()

{

CredHandle hCred;

TimeStamp tsExpiry;

SECURITY\_STATUS ss;

// Получение учетных данных по умолчанию

ss = AcquireCredentialsHandle(

NULL, // Имя пользователя

L"NTLM", // Поставщик безопасности

SECPKG\_CRED\_OUTBOUND, // Учетные данные для исходящих соединений

NULL, // Зарезервировано

NULL, // Данные для аутентификации

NULL, // Указатель на функцию

NULL, // Данные для функции

&hCred, // Дескриптор учетных данных

&tsExpiry); // Срок действия учетных данных

if (ss != SEC\_E\_OK)

{

printf("AcquireCredentialsHandle Error %u\n", ss);

return;

}

printf("Credentials acquired successfully\n");

// Здесь можно продолжить с вызовом InitializeSecurityContext и других функций SSPI для завершения аутентификации.

// Освобождение учетных данных

FreeCredentialHandle(&hCred);

}

int main()

{

AuthenticateWithSSPI();

return 0;

}

1. Структурная обработка ошибок в Windows: программное исключение, программные конструкции для обработки ошибок в Windows, фильтры, возможности API для структурной обработки ошибок, генерация ошибок, финальная обработка исключений.

* *зачем нужен SEH?*
* *когда надо применять SHE*

*Для чего нужен сех. Почему есть трай кэтч, а есть сех. Потому что трай кэтч он привязан к системе программирования (языку). Он может не знать о некоторых исключениях которые происходят в операционной системе. Это языки платформонезависимые. И си ++ тоже старается делать платформонезависимые. А есть какие то особенности которые все же нужно обрабатывать в си ++ вот они и запихнули. Каким образом идет распространение? (есть спец код который приделывается к каждому приложению) (на самом деле main вызывается не напрямую а вызывается через спец код который в себе выдает такой трай, и внутри трая вызывается main функция)*

Обработка ошибок – механизм, кот. встраивается в ЯП, чтобы обозначить некоторые особенности ОС.

**Исключение** – событие в программе, произошедшее во время ее выполнения, в результате которого нормальное выполнение программы становится невозможным. Для дальнейшей работы приложения требуется либо восстановить ее рабочее состояние, либо аварийно ее завершить с очищением всех ресурсов этой программы. Для этого применяется механизм SEH.

**SEH (structured exception handling) –** является низкоуровневым механизмом операционной системой Windows в том смысле, что все ошибки (аппаратные и программные сбои, ошибки исполнения программы), возникающие при выполнении программ в Windows, обрабатываются именно по этой схеме. Все остальные способы обработки ошибок, предоставляемые языками программирования, в конце концов, сводятся к SEH.

SEH может отлавливать не только программные, но и аппаратные ошибки

SEH: Фрейм – блок кода, в котором может произойти исключение(внутри try). Код называется охраняемым кодом.

SEH: Обработчик исключения – блок программного кода, который обрабатывает исключение.

1. **Контекст выполнения**: Набор информации, которая позволяет восстановить состояние программы в момент возникновения исключения.

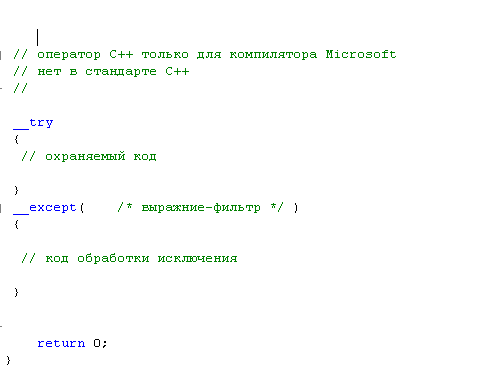
Исключения могут быть основаны на оборудовании или ПО. Структурированная обработка исключений полезна даже в том случае, когда приложения не могут полностью восстанавливаться после исключений. SEH позволяет отображать сведения об ошибках и захватывать внутреннее состояние приложения, чтобы помочь в диагностике проблемы.

**Программные конструкции для обработки ошибок**

Составной оператор после \_\_try предложения — *тело* или *защищенный* раздел.

\_\_except выражение также называется критерием *фильтра*. и в круглых скобках установлено соответствующее значение, называемое фильтром. Его значение определяет, как обрабатываются исключения. Обработчик задает действия, выполняемые при возникновении исключения во время выполнения раздела body. Выполнение происходит следующим образом:

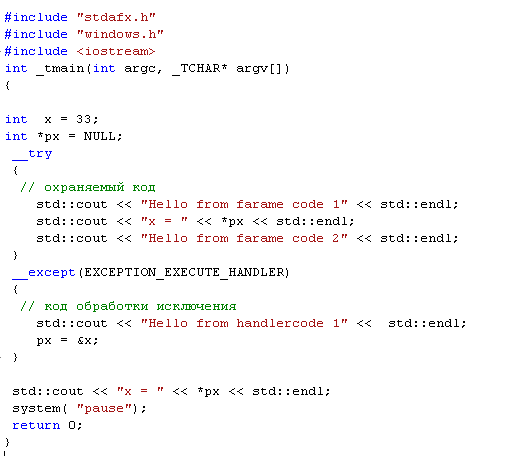
1. Сначала выполняется защищенный раздел \_\_try.
2. Если исключение при этом не возникает, выполнение переходит в инструкцию, стоящую после предложения \_\_except.
3. Если во время выполнения защищенного раздела возникает исключение или в любой подпрограмме вызывается защищенный раздел, \_\_except выражение вычисляется. Возможны три значения.
   1. EXCEPTION\_CONTINUE\_EXECUTION (-1) Исключение закрыто. Выполнение продолжается в точке, в которой возникло исключение.
   2. EXCEPTION\_CONTINUE\_SEARCH (0) исключение не распознано. Продолжайте выполнять поиск обработчика в стеке, сначала для содержащихся try-except инструкций, а затем для обработчиков со следующим высшим приоритетом.
   3. EXCEPTION\_EXECUTE\_HANDLER (1) распознано исключение. Передайте управление обработчику исключений, выполнив \_\_except составной оператор, а затем продолжайте выполнение после \_\_except блока.

****

нельзя goto в охраняемый код и в обработчик.

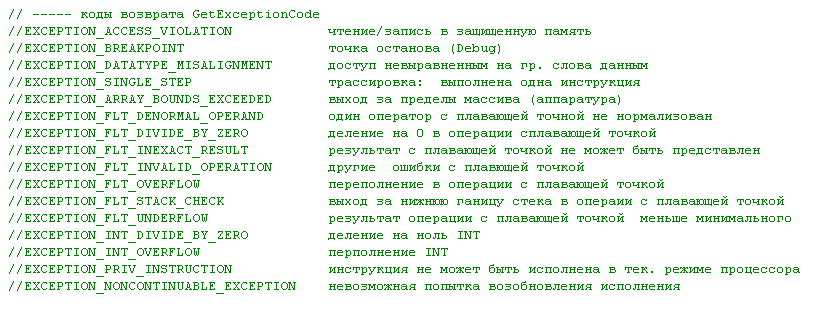
в выражении фильтра можно использовать две функции: GetExeptionCode, GetExceptionInformation.

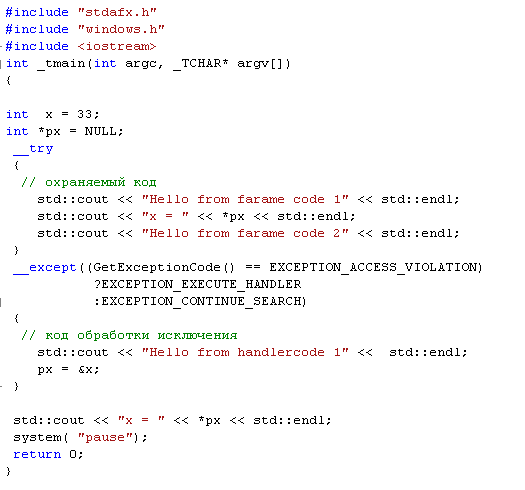
переменные, объявленные внутри {} – локальные

****

try/catch/throw ~ \_\_try/\_\_except: \_\_try/\_\_except разработан раньше, обрабатывает аппаратные ошибки.

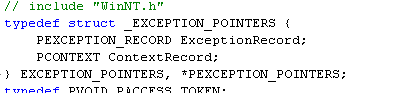
DWORD GetExceptionCode() – может использоваться только в выражении-фильтре (для определения дальнейших действий: обрабатывать, искать обработчик, вернуть управление в точку прерывания) или в блоке обработки исключения (для получения кода исключения).

****

****

В функции фильтра нельзя вызывать GetExeptionCode или GetExceptionInformation, но можно использовать для инициализации параметров этой функции в выражении-фильтре. Если будет сделана попытка возобновить процесс выполнения программы, то в общем случае это не удается: один оператор С++, как правило, состоит из нескольких инструкций процессора, а возврат осуществляется к инструкции, что может привести к зацикливанию

LPEXCEPTION\_POINTERS GetExceptionInformation() – может быть использована только в выражении фильтра

****

****

ExceptionCode = GetExceptionCode()

ExceptionAddress – адрес инструкции

ExceptionFlag = 0 – возможно возобновление работы

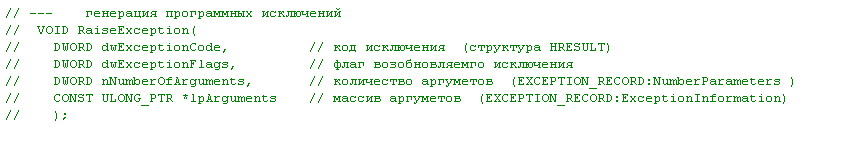
ExceptionFlag = EXEPTION\_NONCONTINUABLE – невозможно

\*ExceptionRecord = указатель на EXEPTION\_RECORD при вложенных исключениях

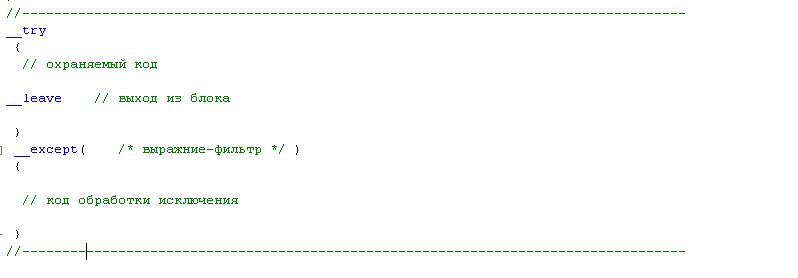
NumberParameters – количество параметров в ExcetptionInformation

ExcetptionInformation – массив 32-битных описателей исключения

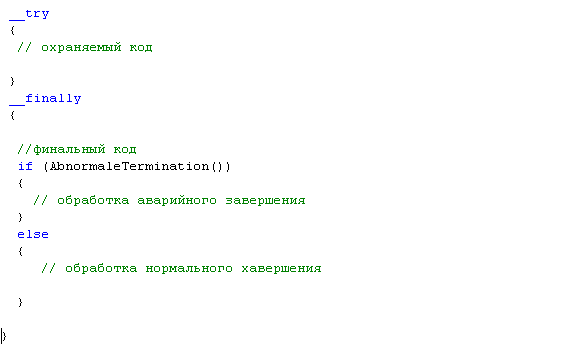
**SEH: Microsoft C++: генерация программных исключений**



SEH: Microsoft C++: передача управления

****

**SHE: Microsoft C++: финальная обработка исключений**

****

*Тут тоже чат:*

Исключения в Windows распространяются по стеку вызовов до тех пор, пока не будет найден подходящий обработчик исключений. Этот процесс включает в себя:

1. **Создание исключения**: Процессор или операционная система генерирует исключение.
2. **Поиск обработчика**: Система начинает поиск обработчика исключений, начиная с текущей функции и поднимаясь вверх по стеку вызовов.
3. **Выполнение обработчика**: Если подходящий обработчик найден, он выполняется. Если обработчик не найден, программа завершает свою работу.

#### Специальный код для каждого приложения

Каждое приложение в Windows может использовать SEH благодаря встроенной поддержке операционной системы и компилятора. Компилятор (например, MSVC) вставляет специальный код в программу для управления структурой SEH, что позволяет обрабатывать исключения на уровне языка C/C++.

Компилятор генерирует и вставляет инструкции для создания записей структурных обработчиков (SEH records) в стек вызовов. Эти записи используются для поиска и вызова соответствующих обработчиков исключений, когда они возникают.

1. Windows-консоль: определение, применение стандартных потоков для ввода/вывода в консоль, возможности API для управления консолью.

- *как можно применить API Console?*

*Что ты можешь сделать с консолью? Могу использовать несколько буферов 1 экран. Могу позиционировать. Могу менять курсор.*

*Ну и как можно применить консоль. С помощью этого api можно сделать собственный интерфейс. Разработать собственную cmd.*

**Windows-консоль (Интерфейс командной строки)** — разновидность текстового интерфейса между человеком и компьютером, в котором инструкции компьютеру даются в путём ввода с клавиатуры текстовых строк.

**Консоль —** это приложение, которое предоставляет службы ввода-вывода для приложений в символьном режиме.

**Консоль** – это три компонента: потоки ввода вывода, буфер и окно (Смелов на лекции)

Входной буфер консоли содержит очередь записей, которые описывают события ввода. События ввода подразделяются на следующие категории:

* + ввод с клавиатуры;
  + ввод с мыши;
  + изменение размеров окна;
  + изменение фокуса ввода;
  + события, связанные с меню.

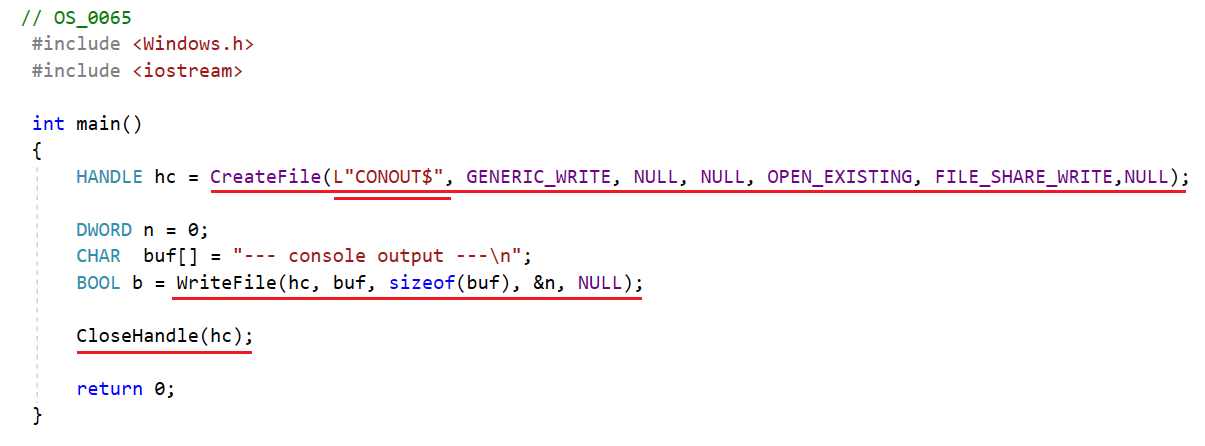
Буфер экрана является двумерным массивом, элементы которого представляют собой записи типа:

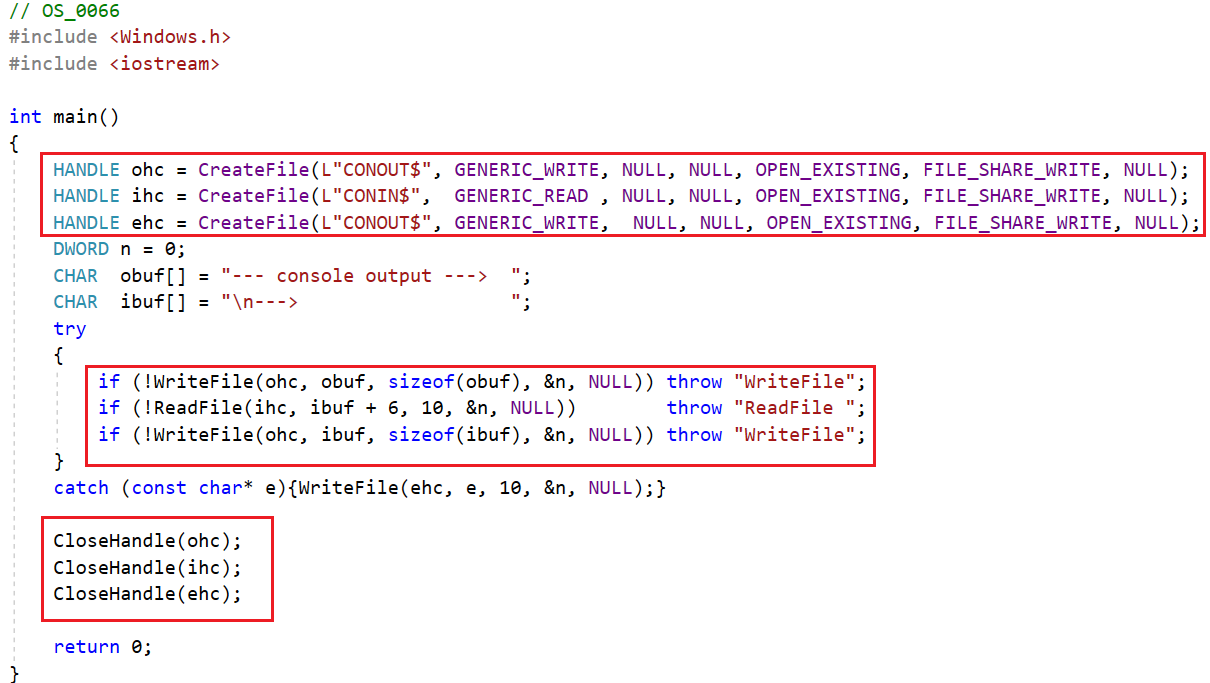
typedef struct \_CHAR\_INFO {  
union {  
WCHAR UnicodeChar;  
CHAR AsciiChar;  
} Char;  
WORD Attributes;  
} CHAR\_INFO, \*PCHAR\_INFO;

где объединение Char содержит символ, представленный в коде Unicode или ASCII, а поле Attributes определяет цвет фона и цвет текста, которыми выводятся символы на экран дисплея. Это значение может быть равно 0, что обозначает фон — черный, а цвет — белый, или любой комбинации из следующих констант:  
BACKGROUND\_BLUE — фон синий;  
BACKGROUND\_GREEN — фон зеленый;  
BACKGROUND\_RED — фон красный;  
BACKGROUND\_INTENSITY — фон яркий;  
FOREGROUND\_BLUE — текст синий;  
FOREGROUND\_GREEN — текст зеленый;  
FOREGROUND\_RED — текст красный;  
FOREGROUND\_INTENSITY — текст яркий.  
Цвет фона и цвет текста будем называть атрибутами текста. Сделаем несколько замечаний относительно использования этих констант. Цвет фона и цвет текста выбираются как комбинация базовых цветов синего, зеленого и красного. То есть в этом случае используется цветовая модель RGB. Можно подсчитать, что всего существует семь возможных комбинаций из трех цветов.  
Белый цвет определяется комбинацией всех трех цветов. Если сюда добавить черный цвет, который определяется как побитовое отрицание белого цвета, то всего существует восемь возможных вариантов, как для цвета фона, так и для цвета текста.

Консоль состоит из входного буфера и одного или нескольких буферов экрана. Входной буфер содержит очередь входных записей, каждая из которых содержит сведения о событии ввода. Очередь ввода всегда включает события ключа и нажатия клавиши. Он также может включать события мыши (перемещения указателя и нажатия кнопки и выпуски) и события, в течение которых действия пользователя влияют на размер активного буфера экрана. Буфер экрана — это двумерный массив символьных и цветовых данных для вывода в окне консоли. Консоль может совместно использоваться любым количеством процессов.

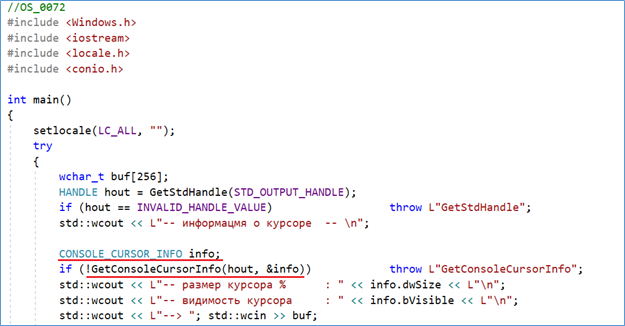
Стандартные потоки:

****

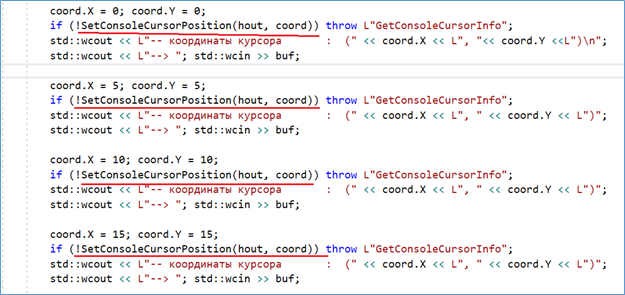
****

Функция [CreateFile](https://docs.microsoft.com/ru-ru/windows/win32/api/fileapi/nf-fileapi-createfilea) позволяет процессу получить дескриптор для входного буфера консоли и активного буфера экрана, даже если STDIN и STDOUT были перенаправлены. Чтобы открыть дескриптор для входного буфера консоли, укажите значение CONIN$ при вызове CreateFile. Укажите значение CONOUT$ при вызове CreateFile, чтобы открыть дескриптор для активного буфера экрана консоли. CreateFile позволяет указать доступ только для чтения и записи в возвращаемом дескрипторе.

Console: параметры консоли



**Console: позиции курсора**

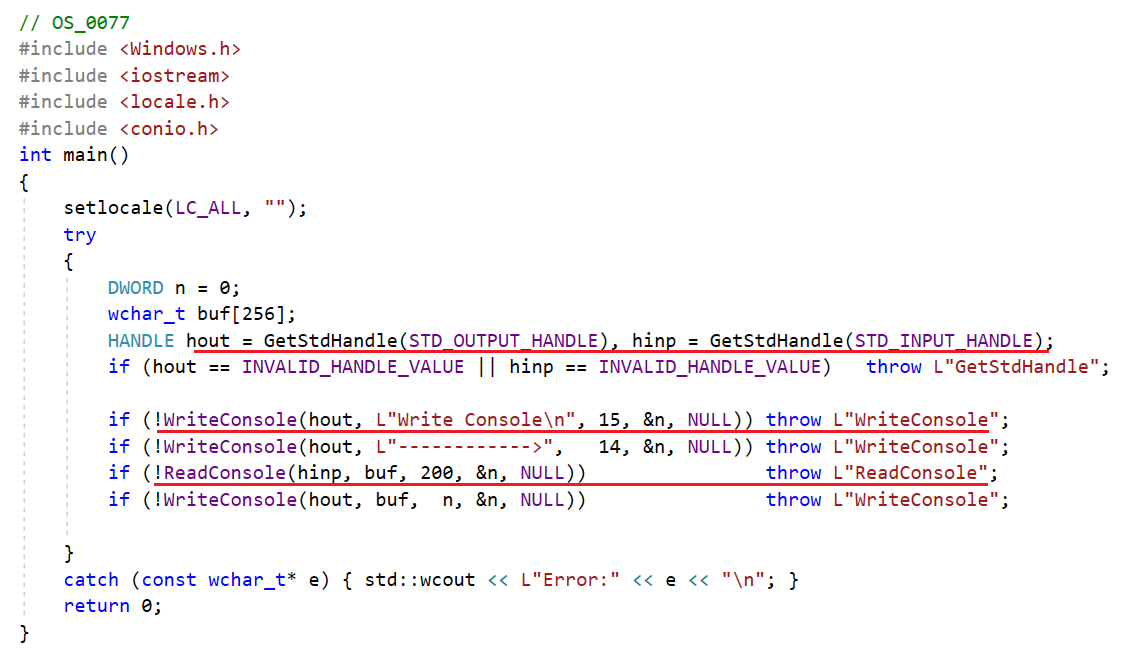


**Console: установка атрибутов консоли**



**Высокоуровневый ввод/вывод WriteConsole, ReadConsole**

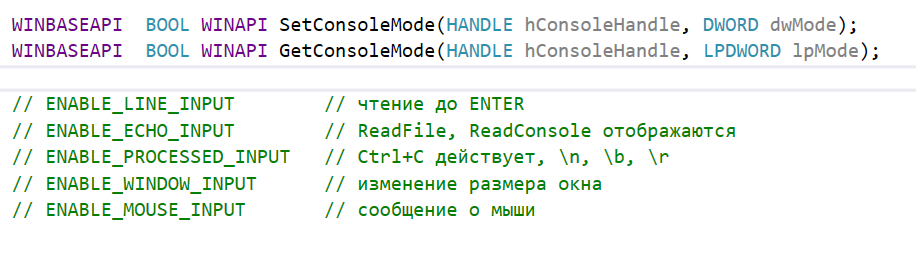
ReadFile, WriteFile – аналогичны WriteConsole, ReadConsole



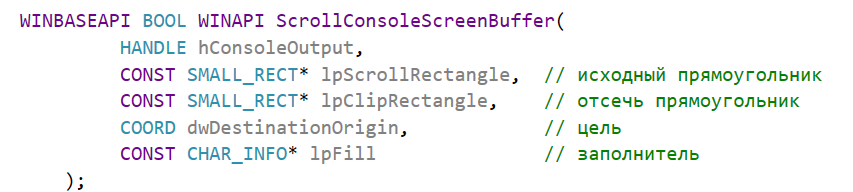
**Console: ввод низкого уровня ReadConsoleInput**



**режимы работы консоли**



**прокрутка буфера экрана**



SetConsoleTitle задаёт заголовок для текущего окна консоли.

WriteConsole записывает строку символов в буфер экрана консоли, начиная с текущего положения курсора.

ReadConsole считывает входные символы из буфера ввода консоли и удаляет его из буфера.

GetCurrentConsoleFont извлекает сведения о текущем шрифте консоли.

FlushConsoleInputBuffer очищает входной буфер консоли.

AllocConsole выделяет новую консоль для вызывающего процесса.

AddConsoleAlias определяет псевдоним консоли для указанного исполняемого файла.

*Чет чат еще написал:*

**Что ты можешь сделать с консолью?**

1. **Ввод и вывод данных**:
   * **Чтение данных** с клавиатуры и **вывод информации** на экран.
   * Перенаправление ввода и вывода в файлы и другие устройства.
2. **Управление текстом и атрибутами экрана**:
   * Изменение **цвета текста** и **фона**.
   * Управление стилями текста, такими как мигающий текст или яркость.
3. **Управление курсором**:
   * **Перемещение курсора** по экрану.
   * **Получение и установка позиции курсора**.
4. **Работа с буфером экрана**:
   * Чтение и запись данных в **буфер экрана**.
   * Создание, изменение размеров и переключение между буферами экрана.
5. **Обработка ввода**:
   * Управление режимами ввода (например, переход в режим без эха).
   * Обработка **клавиатурных и мышиных событий**.
6. **Управление консольным окном**:
   * Изменение размеров консольного окна.
   * Управление заголовком окна.
7. **Управление процессами и командами**:
   * Выполнение команд и скриптов.
   * Управление системными службами и процессами.

**Как можно применить консоль?**

1. **Автоматизация задач**:
   * **Автоматизация повторяющихся задач** через скрипты и командные файлы (batch files).
   * **Пример**: Ежедневное создание резервных копий данных с помощью командного файла.
2. **Администрирование системы**:
   * **Удаленное управление и администрирование** серверов и рабочих станций.
   * **Пример**: Запуск диагностики системы или установка обновлений на удалённом сервере через SSH.
3. **Разработка и тестирование**:
   * Быстрая и **эффективная отладка** и тестирование программ без необходимости сложных графических интерфейсов.
   * **Пример**: Тестирование новых функций программного обеспечения с использованием командной строки.
4. **Управление файлами и каталогами**:
   * Быстрое и **эффективное управление файлами** и папками (копирование, перемещение, удаление).
   * **Пример**: Массовое переименование файлов в каталоге с помощью командных файлов.
5. **Мониторинг и диагностика**:
   * **Мониторинг системных ресурсов** и диагностика проблем.
   * **Пример**: Использование команды **tasklist** для просмотра запущенных процессов и их использования памяти.
6. **Низкое потребление ресурсов**:
   * Консольные приложения потребляют меньше ресурсов по сравнению с графическими интерфейсами.
   * **Пример**: Запуск скриптов на сервере без графического интерфейса для экономии ресурсов.
7. **Обработка и фильтрация данных**:
   * **Фильтрация и обработка данных** на лету.
   * **Пример**: Использование команд **find** и **sort** для обработки текстовых данных.
8. **Создание и управление пользовательскими интерфейсами**:
   * Разработка простых **текстовых интерфейсов** для взаимодействия с пользователем.
   * **Пример**: Создание меню для выбора опций в командном файле.
9. Windows-сервисы: определение, назначение, применение, API.

**Ответ 1:**

**Определение**:

- Windows-сервис (Windows service) - это программа, предназначенная для выполнения в фоновом режиме в операционной системе Windows. Сервисы обычно запускаются при загрузке системы и выполняют свои задачи независимо от активности пользователей.

**Назначение**:

- Сервисы выполняют различные задачи, такие как обслуживание системы, предоставление функциональности для других программ, мониторинг и так далее.

**Применение**:

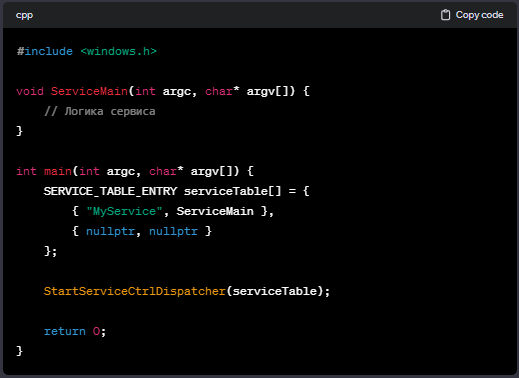
- Сервисы применяются в сценариях, где необходима постоянная и независимая от пользовательской активности работа. Например, службы безопасности, службы баз данных, службы печати и т. д.

**API для создания и управления:**

- Для создания и управления сервисами в Windows, используется Win32 API и функции, такие как `CreateService`, `OpenService`, `StartService`, `ControlService` и другие. Существует также API, предназначенный специально для работы с сервисами - Service Control Manager API.

**Использование WinAPI:**

- Создание сервиса на C++ обычно включает в себя использование Win32 API для работы с сервисами. Например, функции `CreateService`, `OpenService`, и т. д.

Пример кода:

**Управление**:

- Сервисы управляются с использованием инструментов и команд служб в Windows, таких как `sc` (службы командной строки), `services.msc` (Управление службами), а также с использованием API для управления службами.

**Регистрация**:

- Сервисы регистрируются в Service Control Manager (SCM) с использованием функций API, таких как `CreateService`.

Создание и управление Windows-сервисами - это часть разработки системного программного обеспечения, и требует внимания к многим аспектам, таким как жизненный цикл службы, обработка ошибок, безопасность и другие.

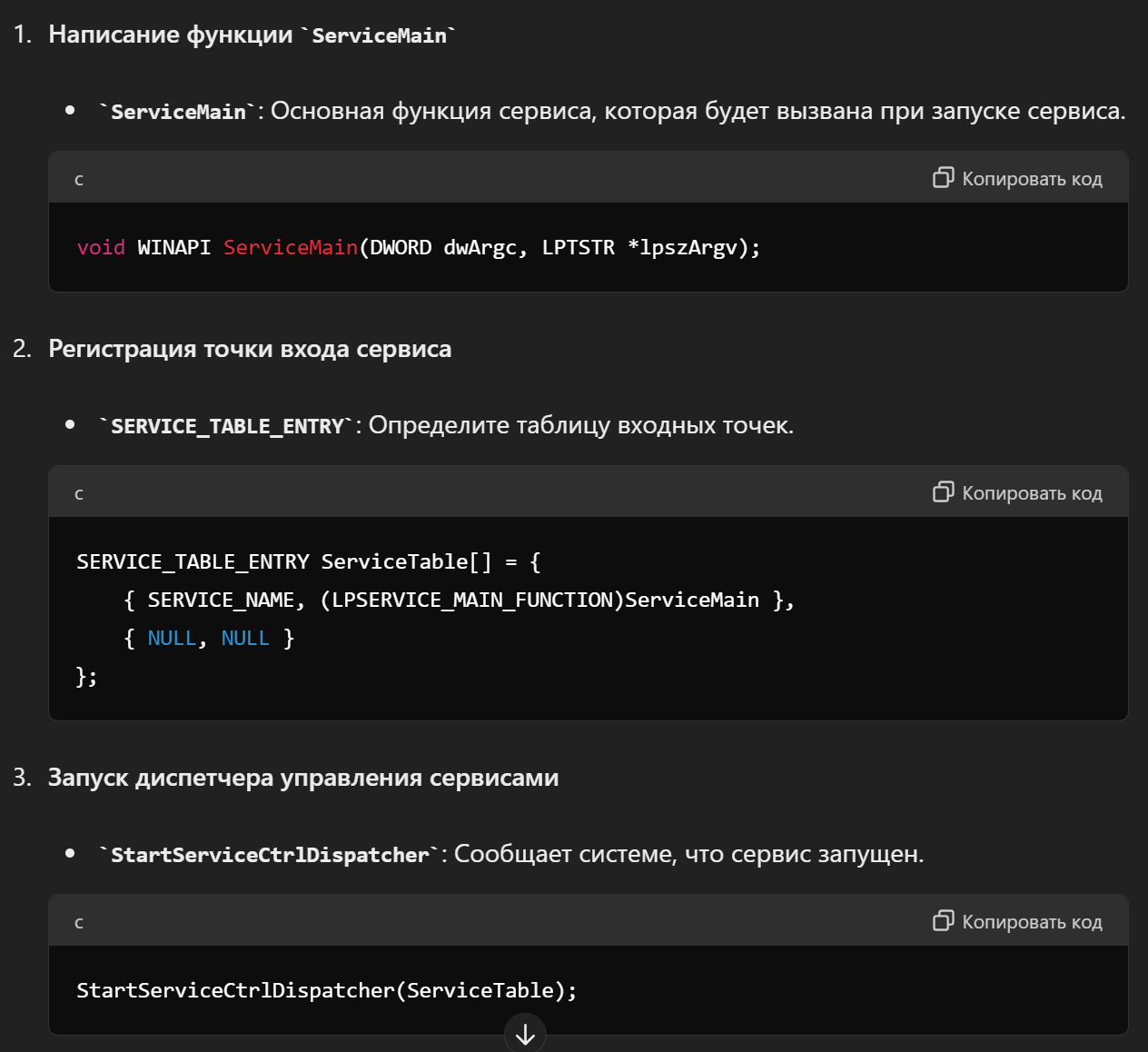
СПРАШИВАЕТ СМЕЛОВ:

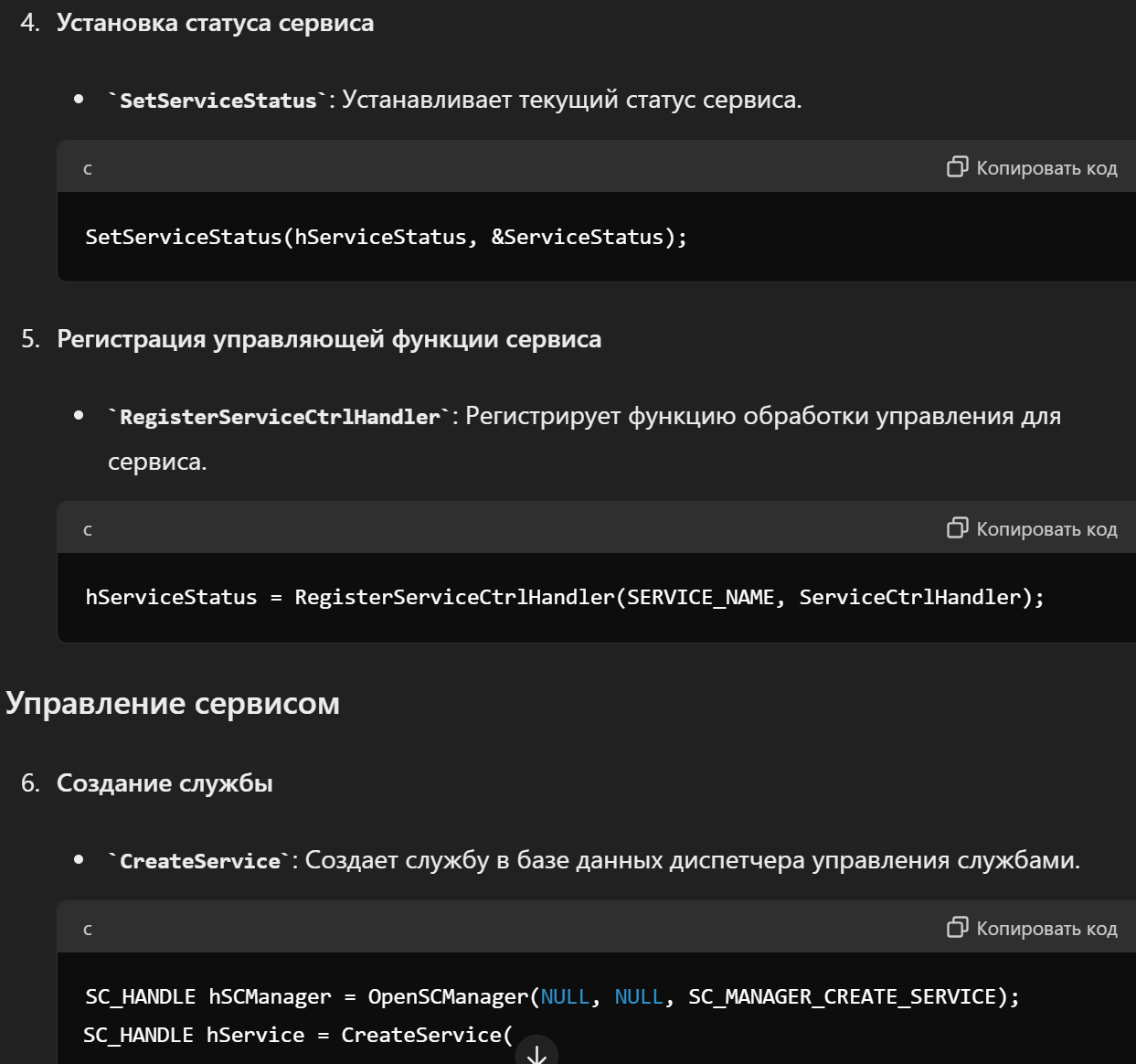
**Порядок создания: написать код, зарегистрировать в диспетчере управления службами, запуск/остановка (в службах).**

**Что можно сделать: install, delete, start, stop, pause, continue, query status (запросить статус), configure (изменение параметров/настроек).**

**Управлять можно через интерфейс ОС, а можно через приложение.**

**API**



****



**Ответ 2:**

**Сервис** - программа, которая запускается при загрузке ОС или любого приложения.

**Сервис** - процесс, выполняющий служебные функции;

Обычно сервис обеспечивает фоновый процесс(**сервер**), работу с внешним устройством (**драйвер**), следит за работой приложений (**монитор**).

**Windows service состоит из:**

***ServiceMain*** - имеет доступ к аргументам командной строки для службы так, как выполняет основная функция консольного приложения.

***ServiceHandler* -** функция обработчика, которая вызывается диспетчером управления, когда процесс обслуживания получает запрос управления от программы управления службой.

**Список служб находится** в реестре Windows «HKEY\_local\_machine\system\CurrentControlSet\Services».

**Сервисы поддерживаются** Windows 2000/XP/2003/Vista/Windows7/ Windows8/ Windows10.

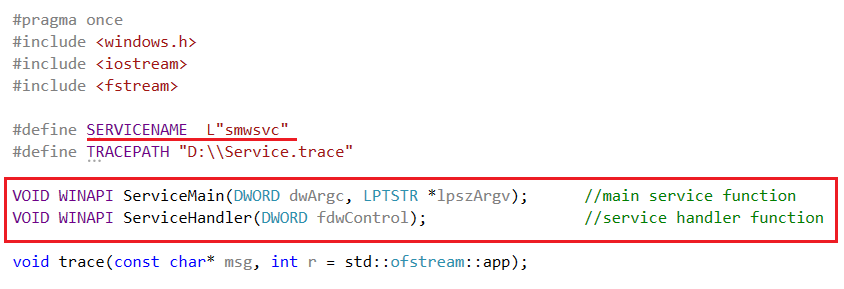
**Service Control Manager (SCM) –** Диспетчер управления службами

* поддержка БД установленных сервисов;
* запуск сервисов при загрузке ОС;
* информирование о состоянии работающего сервиса;
* передача запросов работающим сервисам;
* блокировка и разблокировка базы данных сервисов

**Windows Services:** приложение-процесс сервиса, приложение, которое регистрирует 2 функции обратного вызова. Функция **StartServiceCtrlDispatcher** должна быть вызвана в течении 30мс с момента старта приложения.

есть две функции - ServiceMain (точка входа для сервиса. ее две основные ф-ии это заргеать хендлер и установить статус для службы) и Handler (обрабатывает статусы службы).

callback - ф-ия обратного вызова. это ф-ии servicemain и registerHandler (она вызвается в serviceMain). Table в HTService/main.cpp - это таблица пар-значений имя сервиса - ф-ия сервиса. null null - для конца таблицы



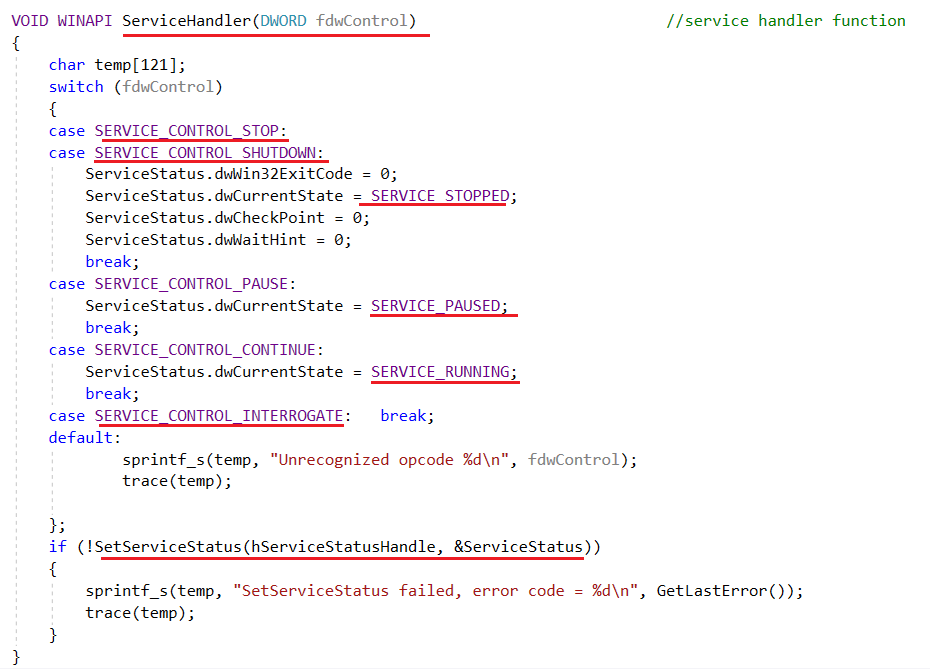


VOID WINAPI SvcMain(DWORD dwArgc, LPTSTR \*lpszArgv )

**1-ый обратный вызов: основная (main) функция сервиса**

void WINAPI ServiceHandler(DWORD dwCode)

**2-ой обратный вызов: обработчик управляющих команд**



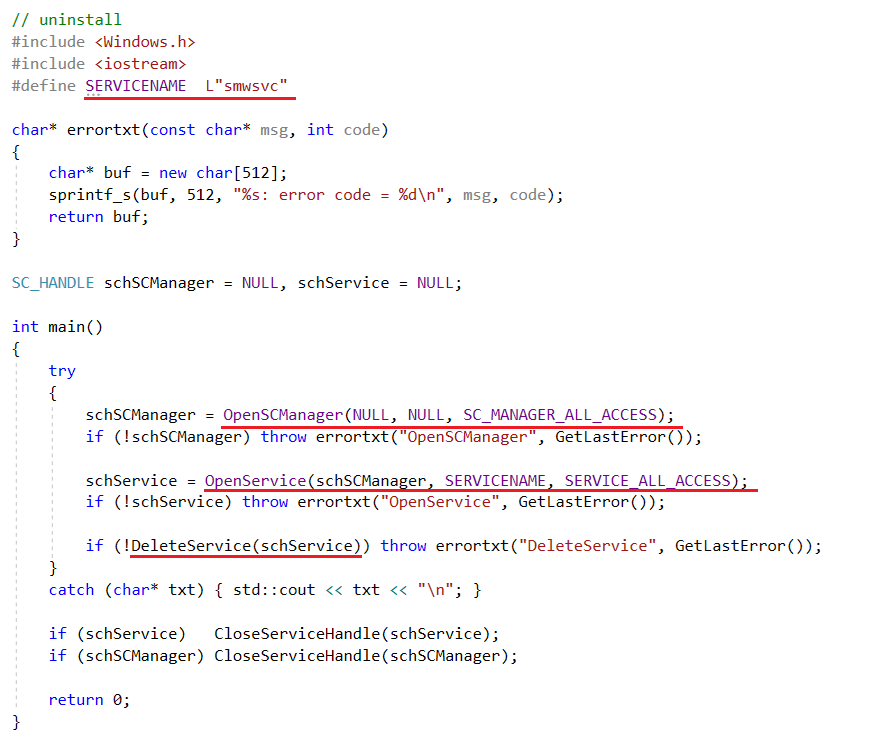
**создание сервиса, регистрация в реестре Windows.**



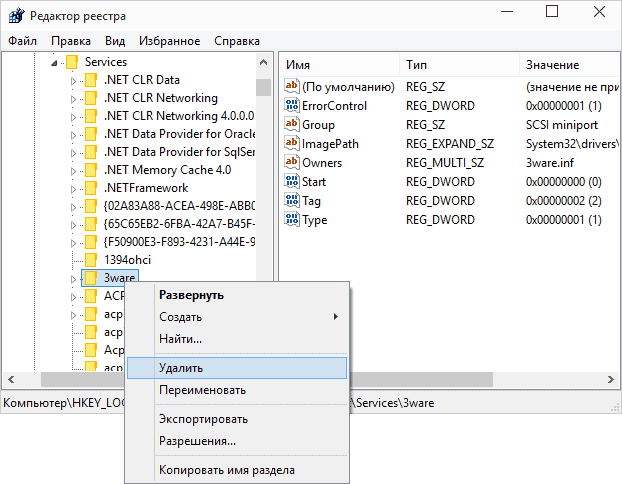


**удаление сервиса.**

**Кодом:**

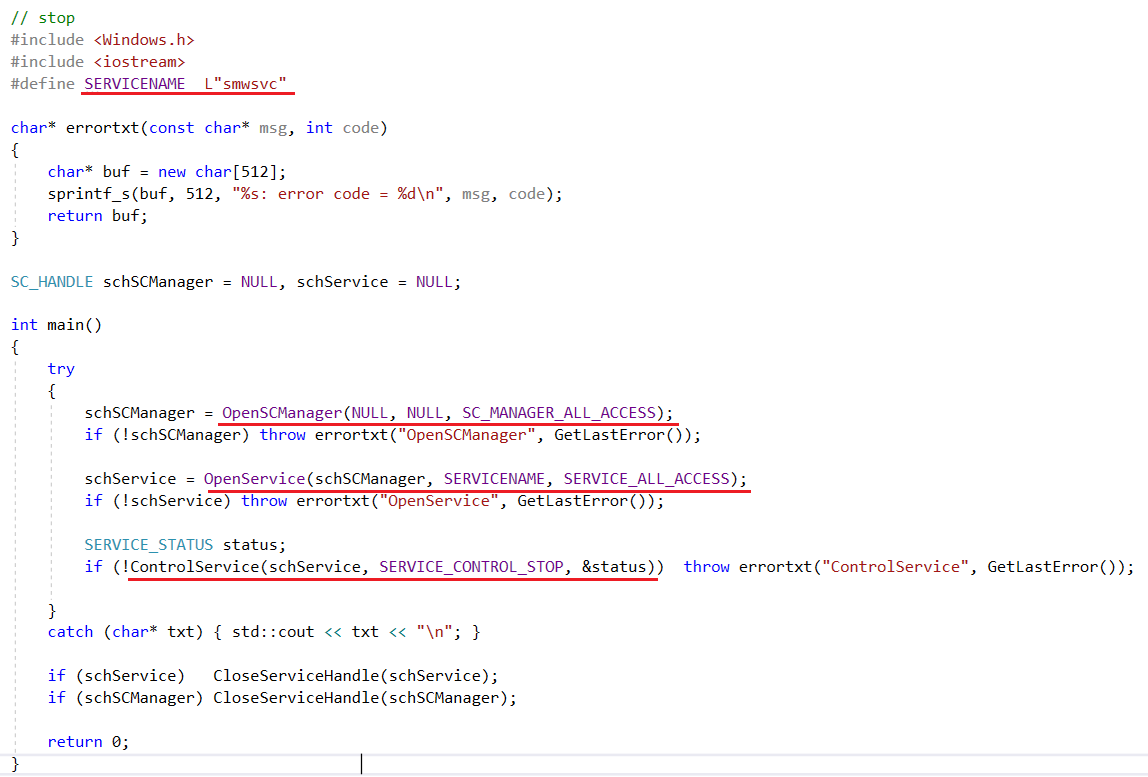


**Удалить службу Windows можно также и с помощью редактора реестра**

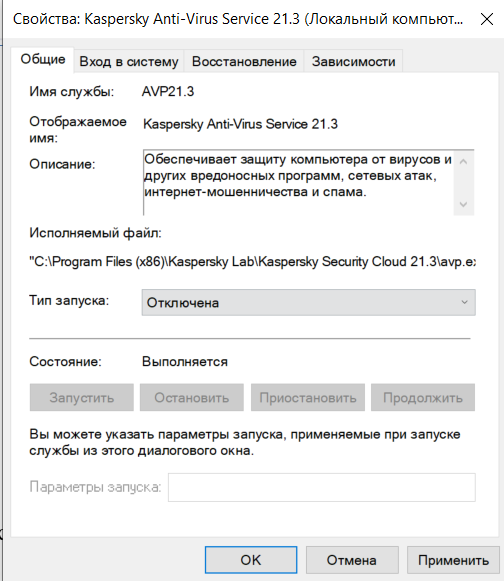


**Windows Services:** остановка сервиса.

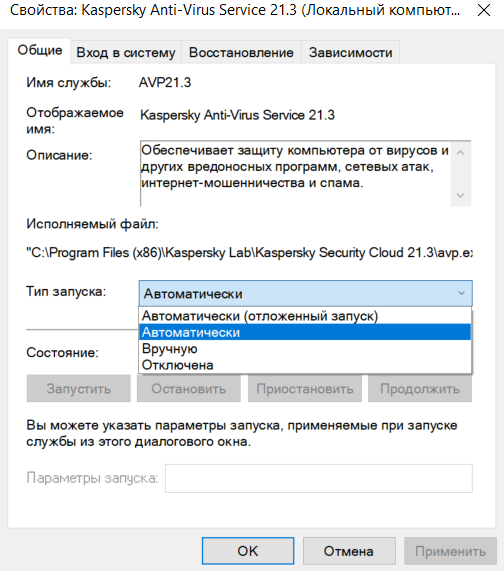
**Кодом:**



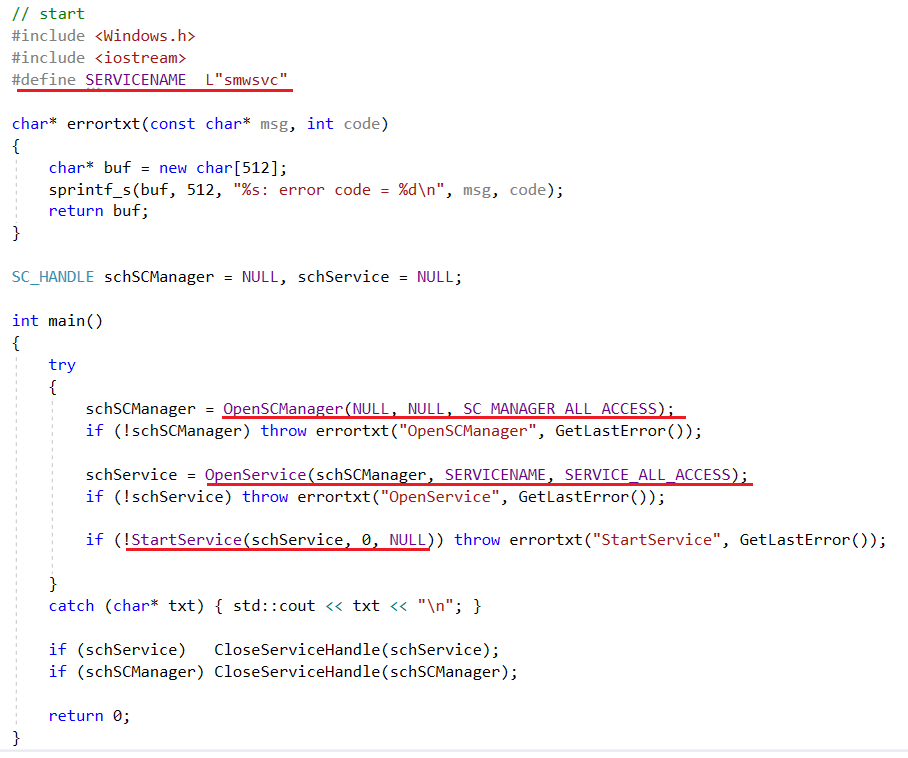
В службах:



**Windows Services:** старт сервиса.



Кодом:



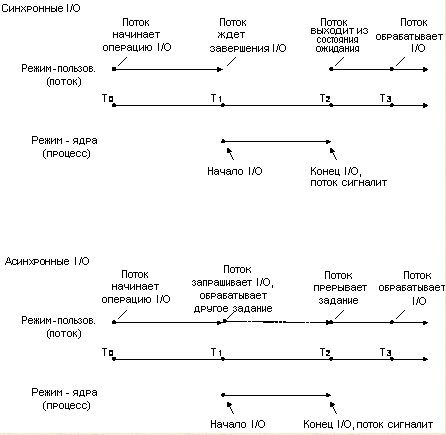
1. Асинхронные операции ввода вывода: понятие асинхронной операции ввода/вывода, особенности программирования асинхронного ввода/вывода.

**!!Разница между асинхронностью и мультипоточностью:  
Асинхронность – механизм, который даётся программистам какой-то системой программирования (async/await). В ОС асинхронности нет, есть синхронизация и мультипоточность. Асинхронность придумана для того, чтобы удобнее было использовать мультипоточность.**

**Асинхронные операции ввода/вывода** позволяют совместить долгие операции ввода-вывода с обработкой компьютера.

Имеется два типа синхронизации ввода-вывода (I/O) файлов: синхронный ввод-вывод файла и асинхронный ввод-вывод файла. Асинхронный еще называется, как перекрывающий(overlapped) ввод-вывод.

При синхронном вводе-выводе файла поток запускает операцию ввода/вывода и немедленно вводит ждущее состояние до тех пор, пока, запрос ввода-вывода не завершит работу. Поток, выполняющий асинхронный ввод-вывод файла, отправляет запрос на ввод-вывод данных ядру. Если запрос принят ядром, поток продолжает обрабатывать другое задание до тех пор, пока ядро не подаст сигналы потоку, что операция ввода/вывода полностью завершилась. Тогда поток прерывает работу со своим текущим заданием и обрабатывает данные от операции ввода/вывода по мере необходимости.



В ситуациях, когда ожидается запрос на ввод-вывод, который займет большое количество времени, такое как обновление или резервное копирование большой базы данных, асинхронный ввод-вывод как правило - хороший способ оптимизировать эффективность обработки. Однако, для относительно быстрых операций ввода/вывода, непроизводительные издержки обработки запросов ядра на ввод-вывод и сигналов ядра могут сделать асинхронный ввод-вывод менее выгодным, особенно если должны делаться много быстрых операций ввода/вывода. В этом случае, синхронный ввод-вывод будет лучше.

OVERLAPPED структура: Используется для управления асинхронными операциями. Содержит информацию о смещении файла и событии завершения операции.

Шаги которые необходимо выполнить для асинхронного ввода/вывода (два способа, первый при единственной операции ввода/вывода, второй при нескольких):

Способ 1.

1.при открытии файла (т.е при вызове функции CreateFile) нужно передать в параметр dwFlagsAndAttributes значение FILE\_FLAG\_OVERLAPPED (включить перекрывающиеся операции ввода-вывода в ресурсе связи).

2.создать (объявить) структуру \_OVERLAPPED. [OVERLAPPED (minwinbase.h) - Win32 apps](https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-overlapped)

3.при вызове функций WriteFileEx/ReadFileEx: в параметр lpOverlapped нужно передать указатель на структуру из пункта 2.

4.GetLastError == ERROR\_IO\_PENDING: если вызовем до окончания ввода-вывода;

5.WaitForSingleObject(HANLE, time); HANDLE - это HANDLE файла полученного при вызове CreateFile.

6.Продвинуть Offset, OffsetHigh.

Способ 2.

1.создать (объявить) структуру \_OVERLAPPED;

2.создать событие Event с автоматическим сбросом;

3.hEvent = Event (записать событие в \_OVERLAPPED);

4.WriteFileEx/ReadRileEx: lpOverlapped;

5.GetLastError == ERROR\_IO\_PENDING:

6.WaitForSingleObject(hEvent, time);

7.продвинуть Offset, OffsetHigh.

**Асинхронное блокирование файлов**

LockFileEx и UnlockFileEx позволяют асинхронно блокировать и разблокировать участки файла (либо весь, либо часть), обеспечивая синхронизацию доступа между процессами. Без «Ex» - синхронно.

Определить состояние асинхронного ввода/вывода.

Функция GetOverlappedResult

Отмена операции ввода/вывода.

Функция CancelTo

Проверить завершение асинхронной операции.

Функция HasOverlappedIoCompleted

Функции завершения

ReadFileEx/WriteFileEx в них можно передать callback который выполнится после окончания операции ввода/вывода.

Порты завершения — это специальный механизм, который позволяет обрабатывать результаты асинхронного ввода-вывода. Их преимущество в том, что создается пул потоков, которые будут обрабатывать результаты ввода-вывода. Целесообразно применять в приложениях в которых есть частые асинхронные операции (не нужно создавать поток на каждую операцию).

Асинхронные порты ввода-выводы – механизм, позволяющий запустить несколько потоков и операций ввода-вывода, чтобы осуществить многопоточную операцию ввода-вывода из нескольких файлов. Многопоточная обработка асинхронных операций ввода-вывода.

Функции для работы с портами:

CreateIoCompletionPort - создать порт завершения

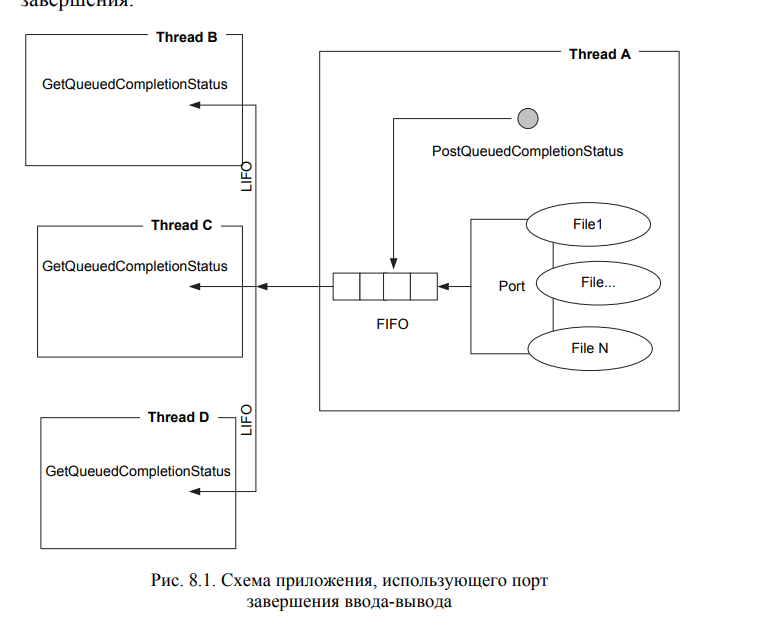
GetQueuedCompletionStatus - получить пакет порта завершения из очереди.

PostQueuedCompletionStatus - послать пакет в очередь порта завершения.

1. Порты завершения ввода/вывода: назначение, применение, API.

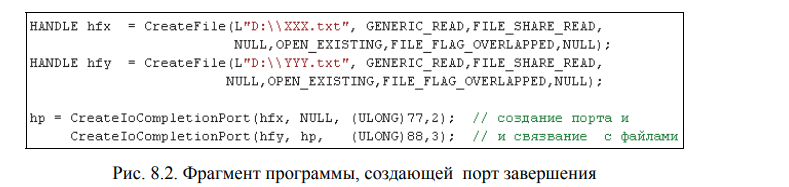
**Порт завершения** – это объект операционной системы, предназначенный для синхронизации работы параллельно работающих потоков с операциями асинхронного ввода-вывода.

**Последовательность**: описать файлы, привязать порты, далее строится очередь – эта операция должна быть асинхронной (идёт сообщение о том, что выполнилась операция асинхронно, она попадает в порт), на другой стороне очереди навешиваются потоки, которые считывают данные. Порты завершения ввода/вывода (IOCP) следует применять в высоконагруженных сетевых приложениях, таких как веб-серверы, файловые серверы и серверы баз данных, где требуется эффективное управление множеством асинхронных операций ввода/вывода для достижения высокой производительности и масштабируемости.



На рисунке изображены четыре потока (A, B. C, D) некоторого Windows-процесса. Поток A использует порт завершения, связанный с дескрипторами N файлов. Поток в определенном порядке выполняет операции асинхронного ввода или вывода для этих файлов. Порт отслеживает завершение операций ввода или вывода для связанных с ним файлов. При завершении каждой операции порт формирует очередной элемент выходной FIFO-очереди. Элемент очереди содержит информацию о завершенной операции. Потоки B, C и D предназначены для циклического считывания элементов выходной очереди порта. При получении из очереди элемента поток может определить файл, для которого была выполнена операция, значение текущей позиции файла и другую информацию. Таким образом, порты завершения ввода-вывода используются в тех случаях, когда формирование блока выводимых данных или обработка блока входных данных требует значительных затрат процессорного времени. Причем обработка каждого блока может выполняться независимо друг от друга. В этом случае достигается с одной стороны распараллеливание операций ввода-вывода с вычислительными операциями процессора, с другой стороны – распараллеливание обработки нескольких блоков данных (это эффективно, если компьютер имеет несколько процессоров).

Создание порта завершения ввода-вывода и связывания его с файлами:



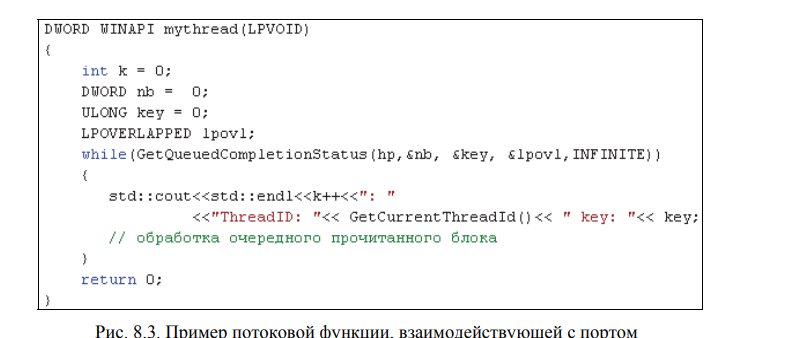
*HANDLE FileHandle: Дескриптор файла, устройства или процесса, который будет ассоциирован с портом завершения. Когда операция ввода/вывода завершится, результат будет помещен в порт завершения.*

*HANDLE ExistingCompletionPort: Если этот параметр равен NULL, функция создает новый порт завершения. Если не NULL, функция ассоциирует FileHandle с существующим портом завершения.*

*ULONG\_PTR CompletionKey: Ключ завершения, ассоциированный с FileHandle. Этот ключ будет возвращен при завершении операции ввода/вывода в порту завершения.*

*DWORD NumberOfConcurrentThreads: Число потоков, которые будут ожидать завершения операций ввода/вывода в порту завершения. Каждый поток получит уведомление о завершении операции в порте завершения.*

Следует обратить внимание на функцию CreateIOComplectionPort, предназначенную для создания порта, первый параметр которой – дескриптор файла, второй – дескриптор порта (используется, если с портом связывается более одного файла), а предпоследний параметр – ключ (идентификатор) файла, который считывается из очереди в рабочих потоках. Функция в цикле вызывает функцию GetQueuedComplectionStatus, считывающую первый элемент очереди. В том случае, если очередь пуста, поток переводится в состояние ожидания до появления элемента, указывающего на завершение операции асинхронного ввода-вывода. Параметры функции позволяют получить значение ключа файла, указатель на структуру OVERLAPPED, количество обработанных (считанных или записаны) байт.



- Функция `PostQueuedCompletionStatus` позволяет добавлять дополнительные сообщения в очередь завершения, которые также могут быть обработаны.

Таким образом, приложение, применяющее порт завершения, должно выполнить следующие действия:

1. Открыть один или несколько файлов в асинхронном режиме.

2. Создать порт и связать его с открытыми файлами.

3. Создать несколько потоков, которые считывают очередь порта.

4. Запустить цикл операций асинхронного ввода или вывода для связанных файлов.

**ГДЕ ПРИМЕНЯЕТСЯ: MS SQL Server, NGINX, IIS.**

1. Платформа Docker: архитектура, назначение, принципы устройства, файловая система UFS, контейнеры, образы, основные команды.

**Платформа** - набор инструментариев и библиотек для разработки.

**Платформа Docker** является открытым программным обеспечением, которое предоставляет возможность упаковывать и запускать приложения в контейнерах. Docker использует стандартные контейнерные технологии, которые позволяют изолировать приложения и их зависимости от окружения.

**Архитектура Docker** состоит из нескольких компонентов:

**Docker daemon** - отвечает за управление контейнерами и их запуск.

**Docker client** - предоставляет интерфейс для взаимодействия с Docker daemon

**Docker registry (реестр)** - это централизованное хранилище образов Docker.

**Docker API** - набор инструментов для взаимодействия с Docker daemon через программный интерфейс.

**Назначение** платформы Docker заключается:

* в упаковке и запуске приложений в контейнерах.
* Docker позволяет изолировать приложения и их зависимости от окружения путем использования стандартных контейнерных технологий.
* Платформа Docker является открытым программным обеспечением, которое предоставляет возможность разработчикам быстро и просто создавать, развертывать и масштабировать приложения в контейнерах.

**Принципы устройства Docker** базируются на использовании контейнерной технологии, которая позволяет изолировать приложения и их зависимости от окружения. Контейнеры в Docker используют **файловую систему UFS**, которая позволяет управлять файлами и директориями в контейнере. **Каждый контейнер создается из образа**, который может содержать необходимые приложения и зависимости. Основные команды Docker позволяют управлять контейнерами и образами, запускать и останавливать приложения, а также работать с Docker registry для хранения и обмена образами.

**UFS - это файловая система**, используемая в Docker для объединения нескольких файловых систем в одну общую. UFS предоставляет возможность создания слоев файловых систем, которые можно объединять вместе, чтобы создавать и управлять контейнерами.

При создании контейнера Docker использует принцип UFS, чтобы собрать слои файловых систем в специальную структуру, называемую контейнером. Каждый слой содержит только изменения по сравнению с предыдущим слоем или базовым образом контейнера. Это позволяет сэкономить пространство на диске и обеспечить быструю загрузку и развертывание контейнеров.

**Принцип CoW (Copy-on-Write)** является основной технологией, используемой UFS для управления слоями файловых систем. Он позволяет создавать "только для чтения" копии файлов или директорий и изменять только те части, которые действительно нужно изменить.

Когда вы создаете или изменяете файл в контейнере Docker, CoW применяет следующий процесс:

* Если файл находится в верхнем слое контейнера, то изменения записываются непосредственно в этот слой.
* Если файл находится в нижних слоях, то CoW создает копию файла только для чтения в верхнем слое и применяет изменения к этой копии. Исходный файл в нижнем слое остается неизменным.

В контексте Docker, слои (layers) представляют собой наборы изменений файловой системы, которые объединяются для создания контейнерного образа. Каждый слой содержит файлы и директории, а также метаданные, связанные с этими файлами.

Принцип UFS = CoW (Copy-on-Write) является ключевой особенностью UFS. Он используется для обеспечения эффективного механизма копирования файлов и директорий в UFS. Когда файл или директория в UFS копируется, на самом деле происходит создание ссылки на исходный объект, а не его фактическое копирование. Это позволяет экономить пространство на диске и ускоряет операцию копирования.

Таким образом, благодаря принципу CoW, каждый слой файловой системы Docker остается неизменным, если он не был изменен в более поздних слоях. Это увеличивает эффективность использования дискового пространства и ускоряет операции чтения и записи в контейнерах Docker.

COW работает в UFS и в снапшотах в SQL Server.

**Клиент** - запускается в командной строке и подключается к локальной (удаленной) службе докера (Docker daemon)

**Docker Daemon** - это служба которая отвечает за все задачи по обработке запросов клиентом. Host - сама служба докер запущена на каком-то хосте (компьютере)

**Контейнер** — это исполняемый экземпляр образа. Вы можете создавать, запускать, останавливать, перемещать или удалять контейнеры с помощью Docker API или CLI. Вы можете подключить контейнер к одной или нескольким сетям, подключить к нему хранилище или даже создать новый образ на основе его текущего состояния.

**Контейнеры в Docker** используются для изоляции приложений и их зависимостей от окружения. Они позволяют запускать приложения в отдельном окружении, что обеспечивает их стабильную работу и повышает безопасность. Каждый контейнер работает в своем собственном пространстве имен, которое изолирует его от других контейнеров и хостовой системы.

**Контейнеры в Docker бывают двух типов**: системные контейнеры и приложения контейнеры.

Системные контейнеры используются для запуска и управления системными сервисами и инфраструктурой, такой как базы данных, кэши и т. д. Эти контейнеры предназначены для использования в качестве инфраструктуры, то есть они запускаются и работают постоянно, в отличие от приложений контейнеров, которые запускаются только при необходимости. В системных контейнерах обычно устанавливаются службы, которые необходимы для работы приложений, такие как базы данных, кэши, мониторинг и т.д.

Приложения контейнеры используются для запуска приложений и их зависимостей в изолированном окружении. Эти контейнеры создаются для каждого приложения, которое нужно запустить, и могут быть запущены и остановлены по мере необходимости. Каждый контейнер содержит все необходимые компоненты для запуска приложения, включая код приложения, зависимости и настройки окружения.

**Image (образ)** - это шаблон только для чтения с инструкциями по созданию контейнера Docker. Вы можете создавать свои собственные образы или использовать только те, которые созданы другими и опубликованы в реестре. Чтобы создать собственный образ, вы создаете Dockerfile с простым синтаксисом для определения шагов, необходимых для создания образа и его запуска. Каждая инструкция в Dockerfile создает слой в образе. Когда вы меняете Dockerfile и перестраиваете образ, перестраиваются только те слои, которые изменились. Это часть того, что делает образы такими легкими, маленькими и быстрыми по сравнению с другими технологиями виртуализации.

**Репозиторий** - в нем находятся различные версии образа. Реестре - в нем находятся различные репозитории.

**Образы Docker** хранятся в Docker registry, который может быть как публичным, так и частным. Публичный реестр Docker Hub содержит множество образов, которые можно использовать для создания контейнеров. Частный реестр Docker позволяет хранить и обмениваться образами внутри организации.

Основные команды Docker позволяют управлять контейнерами, образами и реестрами. Некоторые из них:

* docker run - запускает новый контейнер
* docker stop - останавливает контейнер
* docker ps - выводит список запущенных контейнеров (флаг -a – и остановленные)
* docker images - выводит список доступных образов
* docker pull - загружает образ из реестра или другого репозитория
* docker push - отправляет образ в реестр
* docker start – запуск остановленного контейнера (/stop)
* docker rm – удаление контейнера
* docker rmi – удаление образа
* docker exec – выполнение команды внутри контейнера

Docker Hub - это облачный репозиторий контейнеров, который предоставляет доступ к образам Docker для быстрого развертывания приложений и сервисов в контейнерах. Он предоставляет централизованное хранилище образов, интеграцию с CI/CD платформами, возможность автоматического сбора образов из исходного кода, а также инструменты для совместной работы и управления доступом. Разработчики и команды могут использовать Docker Hub для обмена образами контейнеров и управления ими на всех этапах разработки, тестирования и развертывания приложений.

РАЗНИЦА IMAGE И КОНТЕЙНЕРА: основное отличие заключается в том, что образ Docker - это статичный шаблон, в то время как контейнер - это динамически создаваемое и запускаемое окружение на основе этого шаблона.

**Image (Образ)**:

* **Создание и управление**: Вы можете создавать собственные образы с помощью Dockerfile, комбинируя различные инструкции для создания нужной конфигурации.
* **Хранение и распространение**: Образы могут быть сохранены в репозиториях, таких как Docker Hub, для обмена ими между разработчиками или использования на разных системах.

**Процесс старта контейнера**:

* **Создание контейнера**: При запуске контейнера на основе образа, Docker создает экземпляр контейнера, который работает на основе указанных параметров.
* **Инициализация**: Docker инициализирует контейнер, выполняя команды, указанные в Dockerfile или в командной строке.
* **Запуск приложений**: Контейнер может запускать одно или несколько приложений, определенных внутри образа.
* **Проброс портов**:
  + Для проброса портов используется опция **-p** или **--publish** при запуске контейнера.
  + Пример: **docker run -p 8080:80 nginx** - проброс порта 8080 на хосте на порт 80 в контейнере nginx.
* **Связь между контейнерами**:
  + Можно использовать сети Docker для связи контейнеров.
  + Docker Compose также может использоваться для объединения контейнеров в один сервис.
* **Docker Compose**:
  + Инструмент для определения и запуска многоконтейнерных Docker-приложений.
  + Позволяет описывать конфигурацию приложения в YAML-файле.
* **Volume в Docker**:
  + Это механизм для сохранения и передачи данных между контейнерами и хостом.
  + Виды: **named volume** (именованный том) и **bind mount** (привязка каталога хоста к каталогу контейнера).
  + **docker volume create <имя\_тома>**: Создание именованного тома.

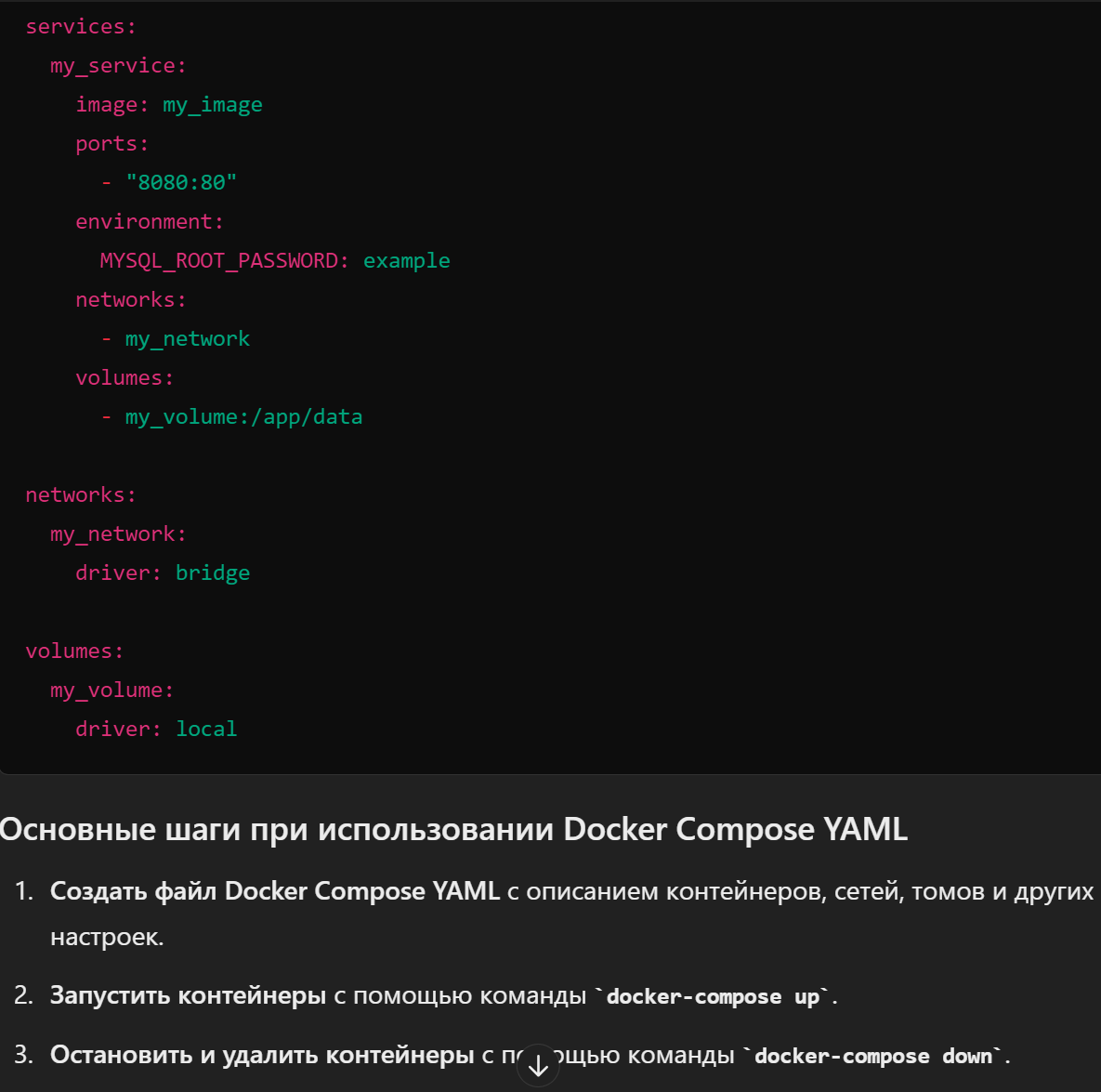
docker run -d --name my\_container -v my\_volume:/path/in/container

* **Dockerfile**:
  + Файл, содержащий инструкции для создания Docker-образа.
  + Основные команды: **FROM**, **RUN**, **COPY**, **ADD**, **CMD**, **ENTRYPOINT**, **EXPOSE**, **WORKDIR**.

1. Платформа Docker: процесс разработки Image.

Слова Смелова: Берётся готовый Image из DockerHub, для него делается DockerFile (указывается его *from*), потом туда дописывается что-то своё (команды), можно выполнить команды внутри контейнера (которые будут инсталлировать); docker build, указывается dockerfile и он сгенерит новый image. Указывая на него *from* его снова можно использовать. Сказать про .yaml (конфигурационный файл – описываются запускаемые сервисы).

Файлы YAML в Docker используются для определения и настройки Docker-контейнеров и их сетевых окружений с помощью Docker Compose.



Процесс разработки собственного Docker образа включает несколько этапов:

Выбор базового образа: Определите, на каком базовом образе вы хотите основать свой собственный. Это может быть образ операционной системы (например, Ubuntu, Alpine), образ для определенного языка программирования или фреймворка (например, Python, Node.js), или специализированный образ для конкретного приложения.

Создание Dockerfile: Напишите Dockerfile, который описывает все шаги, необходимые для создания вашего образа. В Dockerfile вы определяете базовый образ, копируете файлы и каталоги, устанавливаете зависимости, выполняете конфигурацию и настройку, запускаете команды для установки и настройки вашего приложения.

Сборка образа: С помощью команды docker build вы собираете Docker образ из Dockerfile. Docker анализирует Dockerfile и выполняет инструкции, описанные в нем, для создания образа.

Тестирование образа: Запустите контейнер из вашего образа с помощью команды docker run и убедитесь, что ваше приложение работает корректно внутри контейнера. Вы можете провести ручное тестирование, а также автоматизированные тесты, чтобы убедиться, что контейнер работает как ожидается.

Оптимизация образа: Оптимизируйте свой Docker образ, уменьшив его размер и улучшив производительность. Используйте многоэтапную сборку, удаляйте неиспользуемые зависимости и файлы, уменьшайте слои образа и т.д.

Публикация образа: Если ваш образ будет использоваться другими людьми или на других системах, опубликуйте его в Docker Hub или в другом репозитории контейнеров. Это позволит другим разработчикам и системным администраторам загружать и использовать ваш образ.

**DOCKERFILE:**

# Используем базовый образ

FROM ubuntu:latest

# Устанавливаем необходимые пакеты

RUN apt-get update && apt-get install -y python3 python3-pip

# Устанавливаем рабочую директорию

WORKDIR /app

# Копируем файлы приложения в рабочую директорию

COPY . /app

# Устанавливаем зависимости

RUN pip3 install -r requirements.txt

# Открываем порт 80

EXPOSE 80

# Устанавливаем переменные окружения

ENV APP\_ENV=production

# Задаем команду для выполнения при запуске контейнера

CMD ["python3", "app.py"]

1. Платформа Docker: процесс разработки серверного приложения.

Если разрабатывается сервер в docker, то нужно выпустить его наружу, чтоб можно было делать к нему запросы, нужно порты пробросить, как работать с папкой; нужно вывести БД наружу (обычно в volume папку); как работать с папкой.

Чтобы сервер, работающий в контейнере Docker, был доступен извне и можно было делать к нему запросы, нужно пробросить порты контейнера на хостовую машину. Это можно сделать при запуске контейнера с помощью параметра -p. Например: **docker run -d -p 8080:80 <имя\_вашего\_образа>**

Чтобы работать с папкой на хостовой машине из контейнера, вы можете использовать механизмы монтирования томов или проброса путей (bind mounts). Например, для монтирования папки /host/folder на хостовой машине в папку /container/folder в контейнере, вы можете использовать параметр -v при запуске контейнера. Теперь содержимое папки /host/folder на хостовой машине будет доступно внутри контейнера по пути /container/folder. (использованием volume)

Чтобы вывести базу данных наружу, обычно используются Docker volumes. Docker volume - это специальный механизм для хранения данных вне контейнера, который обеспечивает сохранность данных даже при удалении контейнера. Вы можете создать Docker volume с помощью команды docker volume create и затем использовать его при запуске контейнера, указав параметр -v:

**docker volume create mydatabase**

**docker run -d -v mydatabase:/var/lib/mysql <имя\_вашего\_образа>**

