**Экзаменационные вопросы**

**ПОИТ-3, зима 2023**

**Системное программирование**

2. Что это такое управление процессами синхронизации с фацловой системой, хип; посикс

3. процессы как применить, для чего параметры, как запустить, как остановить,

Как можно работать управлять процессами

Есть два процесса чтобы изолировать друг от друга, реализовать взаимодействие, либо переда обмен данными либо синхронизация, как обменяться данными между процессами

Неймд пайп 2 вида, образ файла, мейлслот, через файл

Синхронизация: мьютекс, семафор (критическая секция нет)

Через память дочерние процессы можем

4.примененик потоков, как можно применить, для чего применять потоки

Разница между ассинхронностью и параллельности

Два потока параллелльно одновременно на разных процесссах

Асинхронная операция – два этапа: заявка на исполнение, получение результата, между двумя событиями можно что-то делать

Ассинхронность - Схема по которой мы программируем

Если один процессор асинхронное не распаралелиться

5.в каком случае поток не работает, в очереди, заблокирован, суспенд(приостановлен), слип, слип плюс суспенд – можно пока не резьюм

Вайлотайл – переменная в многопоточном

Должен отклюять оптимизацию для переменной

6. синхронизация механизмы апи какими способами можно синхронизировать

7. сто можно сделать в файловой системе

Какие есть файловые сисетмы

Команды файловой системы

Мув копи

Асинхронная операция отличия от сихнроннной

7. для чего две памяти: виртуальная и хип

Alloc virtual heap

Постранично 4кБт килобайта – виртуальная – большие данные

Небольшие куски памяти – хип

Два списка в хипе

Что такое фрагментация в хипе

Упаковка хипа

Механизмы отображения что это такое как сделать послеовательность и длч еъчего

Что это такое такое как сделать в какой последовательности для чего

Для межпроцессорного обмена

10.длл как они проецируются в адресное простнрастно процесса

длм мейн какие параметы причина вызова

экспорт функции каким образом загруить получить указатель на ыункцию

библиотека импорта как загрузить для чего используется

для чего взаимодейтсвует импорт

11. спецификация ком

Ком сервер типы серверов, каким образом создаются компонеиы

Чем характеризивуется компоненты и инферфейсы

Два вида счетчика

12.чо такое дистанционная системва безопасности

Апи для ползователей

Группа уч записи

Что такое уч запись

Свзяь между группой и учю записью

13Структурная обработка ошибок что это такое какой механизм трай ексепт (3 варинат филтра)

Как можно генерировать исключение

Чем она характеризуется как все устроено

14. консоль

Все что можно сделать

Поменять заголовок

Размер

Позифиоировть курсор

Размер курсора

Создат несколько буферов

Работа с окном

15 виндовс сервис

Что такое как создать

Для сего он нужен

Как управлять как регистрировать

16. что это такое особенности выполнения

17. Принципы портов ввода выода

Как работают

В каких случаях эффективно применять эти порты

18.что можно сделать с платформлой докер

Как внутри устроено

Из чего состоит

Что можно записать в фацл какие команды выполнить

Как создать имедж как туда записать

1. Системное программирование: определение, назначение, применение.

системном программировании- разработка программ, имеющих один из трех признаков.

Первый признак: пользователем разработанного программного обеспечения является программист. Иными словами, системный программист разрабатывает программное обеспечение для других программистов, часто называемых «проблемными».

Второй признак: разрабатываемое программное обеспечение является повторно используемым и, как правило, оформляется в виде библиотек функций и применяется в нескольких прикладных приложениях. Можно сказать, что системный программист разрабатывает общее, универсальное программное обеспечение.

Третий признак: системное программное обеспечение напрямую использует системные вызовы операционной системы.

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: Создание процесса

Поэтому, как правило, системное программирование подразумевает разработку библиотек универсальных функций, использующих системные вызовы операционной системы. При разработке повторно используемого программного обеспечения, системный программист берет уже существующую или предлагает новую систему соглашений, которой оно должно соответствовать.

Соглашения могут быть оформлены в виде спецификаций или корпоративных стандартов. В этих документах, как правило, оговариваются принципы именования объектов (имена функций, параметров, переменных), структуры и типы используемых данных, система интерфейсов (группы функций, классифицированных по каким-то признакам) и т. д.

Примером такой спецификации может служить COM (Component Object Model – объектная модель компоненты) компании Microsoft. Практически вся прикладная часть программного обеспечения (например, Microsoft Office), входящая в операционную систему Windows, разработана в соответствии с этой спецификации Программист, освоивший эту спецификацию, получает возможность писать программное обеспечение, которое может использовать

Из чата!

Назначение системного программирования:

Работа с аппаратурой: Системное программирование позволяет взаимодействовать с аппаратурой компьютера, такой как процессоры, память, периферийные устройства.

Разработка операционных систем: Разработка компонентов операционных систем, включая ядро, драйвера устройств, системные библиотеки.

Оптимизация производительности: Системные программисты могут оптимизировать код и ресурсы для повышения производительности системы.

Работа с сетями: Разработка сетевых протоколов, серверов и других компонентов для обеспечения сетевой связи. Создание компиляторов и интерпретаторов: Разработка программ, преобразующих исходный код высокоуровневых языков в машинный код.

Управление ресурсами: Системное программирование занимается управлением ресурсами компьютерной системы, такими как память, процессорное время, файловая система.

Применение системного программирования:

1. ***Разработка операционных систем.*** Операционные системы, такие как Windows, Linux, MacOS, являются примерами системного программного обеспечения, поскольку они предоставляют интерфейс между аппаратной частью компьютера и другими приложениями.
2. ***Разработка драйверов устройств.***Драйвер устройства — это программа, которая позволяет операционной системе взаимодействовать с аппаратным обеспечением компьютера, таким как принтеры, сканеры, звуковые карты и т.д.
3. ***Разработка компиляторов.***
4. ***Разработка программного обеспечения встроенных систем.*** Встроенная система — это компьютерная система, которая управляет конкретным устройством, таким как автомобиль, телевизор или медицинское оборудование. Встроенные системы управления и контроля используются для автоматизации производства, управления транспортом
5. ***Разработка сетевого программного обеспечения.*** Сетевое программное обеспечение — это программа, которая обеспечивает взаимодействие между компьютерами в сети.
6. ***Разработка баз данных и систем управления базами данных, таких как Oracle, MySQL, PostgreSQL и др.*** Базы данных предназначены для хранения и организации больших объемов данных.
7. ***Системы безопасности, такие как антивирусы и фаерволы, также создаются с использованием системного программирования****.* .
8. ***Высокопроизводительные вычислительные системы, такие как суперкомпьютеры и высокопроизводительные вычислительные кластеры, используются для научных и инженерных расчетов, моделирования и анализа данных.***

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

**фреймворк**

Совокупность библиотек и среды выполнения, облегчающее разработку большого программного проекта.

набор библиотек и правила работы с ними, предназначены для интерфейсов выполнения программ с ядром ОС.

**Фреймворк – набор библиотек, посредник между вами и ядром ОС (дословно Смелов)**

набор библиотек и правила работы с ними, облегчающее разработку большого программного проекта. **посредник между вами и ядром ОС**

[**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).

**Применение**: Фреймворки операционной системы используются в различных областях, включая разработку мобильных и настольных приложений. [Они могут включать инструменты для работы с графикой, базами данных, сетевыми соединениями и другими функциями, которые обычно требуются приложениям](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).

[Примером фреймворка операционной системы может служить .NET Framework от Microsoft](https://learn.microsoft.com/ru-ru/dotnet/framework/install/on-windows-10). [Этот фреймворк предоставляет набор библиотек и инструментов для разработки приложений под операционную систему Windows](https://learn.microsoft.com/ru-ru/dotnet/framework/install/on-windows-10).

Важно отметить, что фреймворк отличается от библиотеки. Библиотека может быть использована в программном продукте просто как набор подпрограмм похожей функциональности, не влияя на архитектуру программного продукта и не накладывая на неё никаких ограничений. [В то время как фреймворк диктует правила построения архитектуры приложения, задавая на начальном этапе разработки поведение по умолчанию - «каркас», который нужно будет расширять и изменять согласно указанным требованиям](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA)

**POSIX** (англ. Portable Operating System Interface — переносимый интерфейс операционных систем) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов.

Таненбаум:

Стандарт POSIX определяет минимальный интерфейс системных вызовов, который должны поддерживать совместимые с ним системы UNIX. Фактически на данный момент POSIX-интерфейс поддерживается также рядом других операционных систем

[Стандарт POSIX определяет как системные, так и пользовательские уровни интерфейсов программирования приложений (API), а также командные оболочки и интерфейсы утилит, для обеспечения совместимости программного обеспечения (портативности) с вариантами Unix и другими операционными системами](https://en.wikipedia.org/wiki/POSIX).

[POSIX также предназначен для использования как разработчиками приложений, так и системными разработчиками](https://en.wikipedia.org/wiki/POSIX).

Вот еще некоторые интересные факты о POSIX:

1. [**Pthreads**: POSIX также определяет стандарт для многопоточности, известный как Pthreads1](https://habr.com/ru/articles/326138/). [Это позволяет разработчикам создавать приложения, которые могут выполняться в несколько потоков, что улучшает производительность и эффективность1](https://habr.com/ru/articles/326138/).
2. [**Стандартные потоки**: POSIX определяет стандартные потоки ввода-вывода, которые используются в системах типа UNIX2](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D1%8B%D0%B5_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B8). [Это включает стандартный ввод (stdin), стандартный вывод (stdout) и стандартный вывод ошибок (stderr)2](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D1%8B%D0%B5_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B8).
3. [**Совместимость с Unix**: POSIX был разработан для обеспечения совместимости программного обеспечения с вариантами Unix3](https://wiki5.ru/wiki/POSIX). [Это означает, что приложения, разработанные в соответствии со стандартом POSIX, могут работать на любой операционной системе, которая также соответствует этому стандарту3](https://wiki5.ru/wiki/POSIX).
4. [**Переносимость**: Одной из основных целей POSIX является обеспечение переносимости приложений на уровне исходного кода3](https://wiki5.ru/wiki/POSIX). [Это означает, что разработчики могут написать код один раз и запустить его на любой операционной системе, которая соответствует стандарту POSIX3](https://wiki5.ru/wiki/POSIX).

Что это такое управление процессами синхронизации с фацловой системой, хип; посикс

Для управления процессами синхронизации с файловой системой в операционной системе Windows можно использовать различные средства и подходы. Вот несколько из них:

* 1. Критические секции и объекты синхронизации:

В Windows можно использовать критические секции, мьютексы, семафоры и другие объекты синхронизации для управления доступом к общим ресурсам. Например, функции CreateMutex, CreateSemaphore, WaitForSingleObject и ReleaseMutex могут быть использованы для синхронизации между процессами.

Мьютексы (Mutex):

Мьютексы используются для защиты критических секций кода, которые могут выполняться только одним потоком или процессом в определенный момент времени.

Семафоры (Semaphore):

Семафоры представляют собой счетчики, которые могут использоваться для ограничения количества потоков или процессов, которые могут одновременно получить доступ к ресурсам.

Критические секции (Critical Sections):

Критические секции представляют собой участки кода, которые могут быть выполнены только одним потоком или процессом в определенный момент времени. В операционных системах Windows используется API функций EnterCriticalSection и LeaveCriticalSection для реализации критических секций.

* 1. События Windows:

Используйте события Windows (Event Objects) для сигнализации между процессами о наступлении определенных событий, таких как изменение файла.

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

**Процесс OS** –— единица работы OS (объект ядра OS + контекст процесса).

**Процесс** — **это** в выполняемая в данный момент программа

Единица работы OS. © Смелов

Абстрактное описание запущенной программы, контейнер с информацией о ней.

основные свойства процесса:

* процессу соответствует исполняемый программный файл;
* у процесса есть PID;
* у процесса есть Parent PID;
* в Windows: HANDLE – идентификатор объекта OS;
* в OS есть процесс инициализации (родитель для всех);
* запуск и управление процессом осуществляется с помощью системных вызовов;
* процессы изолированы друг от друга;
* процессу выделяется линейное адресное пространство (размер зависит от разрядности), сегменты: code, static, data, heap, stack;
* **контекст процесса** – данные, которые сохраняются при переключении процессов и предназначенные для продолжения работы;
* процессу автоматически доступны три потока: ввода, вывода, вывод ошибок.
* при запуске OS некоторые процессы загружаются и стартуют автоматически, как правило используются для внутреннего назначения;
* в составе ОS есть таблица, содержащая объекты ядра процессов (состояние, приоритет, указатели на другие объекты); есть средства OS позволяющие ее просматривать;

**Ресурсы**: регистры, открытые файлы, родительский процесс, перечень связанных (дочерних) процессов, реальные страницы памяти, виртуальное адресное пространство, маркеры доступа (безопасность)

**Контекст процесса:** адресное пространство, содержимое регистров (общего назначения, счетчик команд, состояния процессора, вершина стека, …), объекты ядра OS (объекты процессов, потоков, безопасности, файлов и пр.), стек ядра (для ее этого процесса). Переключение контекстов. В ядре специальный стек для переключения контекстов

**Process IDentifier, PID** — уникальный номер (идентификатор) процесса в многозадачной операционной системе (ОС). В ОС семейства Windows PID хранится в переменной целочисленного типа.

**inter-process communication, IPC(механизм межпроцессного взаимодействия)** — обмен данными между потоками одного или разных процессов. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

**Стандартные потоки ввода/вывода процесса** - потоки имеющие зарезервированные номера - дескрипторы (номера): поток ввода (0), поток вывода (1), поток вывода ошибок (2). Как правило, эти дескрипторы открыты уже в момент запуска задачи.

**Системные процессы**: процессы, запускаемые автоматически при запуске OS (**windows-сервисы, linux-демоны**).

Функция **CreateProcess (при вызове нужно имя exe)**, которая создает новый процесс с единственным потоком. Функция возвращает два отдельных дескриптора, по одному для процесса и потока, передавая их в структуре. Глобальные идентификаторы остаются уникальными для данного объекта на протяжении всего времени его существования и во всех процессах, тогда дескрипторов процесса может быть несколько и каждый из которых может характеризоваться собственным набором атрибутов, например определенными разрешениями доступа.

Новый процесс получает информацию об окружении, рабочем каталоге и иную информацию в результате вызова функции CreateProcess. По завершении этого вызова любые изменения характеристик родительского процесса никак не отразятся на дочернем процессе. Оба процесса полностью независимы друг от друга.

**HANDLE** — условно, адрес, по которому хранится информация по процессу, например такая как: время запуска, имя файла, ассоциированного с процессом, и даже тот же самый PID. Будучи однажды получен, HANDLE требует закрытия через CloseHandle().

**Процесс-потомок наследует следующие признаки родителя:**

* сегменты кода, данных и стека программы;
* таблицу файлов, в которой находятся состояния флагов дескрипторов файла, указывающие, читается ли файл или пишется. Кроме того, в таблице файлов содержится текущая позиция указателя записи-чтения;
* рабочий и корневой каталоги;
* реальный и эффективный номер пользователя и номер группы;
* приоритеты процесса (администратор может изменить их через nice);
* контрольный терминал;
* маску сигналов;
* ограничения по ресурсам;
* сведения о среде выполнения;
* разделяемые сегменты памяти.

**Потомок не наследует от родителя следующих признаков:**

* идентификатора процесса (PID, PPID);
* израсходованного времени ЦП (оно обнуляется);
* сигналов процесса-родителя, требующих ответа;
* блокированных файлов (record locking).

Бездействие системы ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) [system](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0)**idle** [process)](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) — [процесс,](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) выполняемый [процессором](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) в пространстве [ядра](https://ru.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) в случае, если нет других [процессов,](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) которые [процессор](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) мог бы выполнять.

**Дерево процессов -** при создании процессом-родителем процессов-потомков формируется дерево процессов, при котором появляется связь от потомка к родителю. Это значит, что при завершении процесса-родителя произойдет и завершение процессов-потомков. Такая связь единичная, то есть процесс-потомок ничего не знает о родителях процесса-родителя.

Применение процессов:

Применение процессов заключается в создании абстракции, которая позволяет осуществлять (псевдо) параллельные операции, даже если в системе присутствует всего один центральный процессор. Процессы превращают один центральный процессор в несколько виртуальных, позволяя операционной системе эффективно управлять многозадачностью.

Процессы в многозадачной системе позволяют центральному процессору быстро переключаться между различными процессами, предоставляя каждому из них долю процессорного времени. Хотя в каждый конкретный момент времени процессор работает только с одним процессом, за секунду он может успеть поработать с несколькими, создавая иллюзию параллельной работы.

Разработчики операционных систем создали концептуальную модель последовательных процессов, которая упрощает работу с параллельными вычислениями. В этой модели все программное обеспечение на компьютере представлено как последовательные процессы, каждый из которых представляет собой экземпляр выполняемой программы с собственным виртуальным центральным процессором. Это упрощает восприятие системы и облегчает работу с (псевдо) параллельными операциями.

Мультипрограммирование, или многозадачный режим работы, представляет собой постоянное переключение между процессами, что позволяет эффективно использовать ресурсы и создавать иллюзию одновременного выполнения задач.

**Особенности выполнения процессов на много(одно)ядерных процессорах:**

В операционных системах, таких как Windows, Linux и других, может выполняться одновременно несколько процессов. [Это свойство называется многозадачностью1](http://xn--80abdbnca6cjfb1beq.xn--p1ai/%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C/). Операционная система управляет выполнением процессов, распределяя им процессорное время. [Это может создать впечатление, что все процессы выполняются одновременно, особенно на многоядерных процессорах2](https://qna.habr.com/q/638662).

Однако стоит отметить, что на одноядерном процессоре в каждый конкретный момент времени выполняется только один процесс. [Операционная система быстро переключается между процессами, что создает иллюзию их параллельного выполнения2](https://qna.habr.com/q/638662).

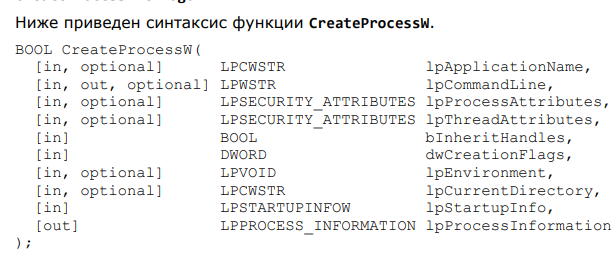
[Также, если запустить несколько однопоточных программ (процессов), то операционная система сама будет регулировать одновременное выполнение программ2](https://qna.habr.com/q/638662). [Это означает, что операционная система может использовать все ядра процессора для выполнения нескольких процессов одновременно2](https://qna.habr.com/q/638662).

[В случае многоядерных процессоров, несколько процессов могут действительно выполняться одновременно, поскольку каждое ядро может выполнять один процесс за раз2](https://qna.habr.com/q/638662). [Это позволяет значительно увеличить производительность системы при выполнении нескольких задач2](https://qna.habr.com/q/638662)

##### **Создание процесса в Windows**

В Windows одним вызовом функции Win32 CreateProcess создается процесс, и в него загружается нужная программа.

У этого вызова имеется 10 параметров, включая выполняемую программу, параметры командной строки для этой программы, различные параметры безопасности, биты, управляющие наследованием открытых файлов, информацию о приоритетах, спецификацию окна, создаваемого для процесса (если оно используется), и указатель на структуру, в которой вызывающей программе будет возвращена информация о только что созданном процессе.



**lpApplicationName**

Имя модуля для выполнения.

В строке можно указать полный путь и имя файла модуля для выполнения или указать частичное имя.

Параметр lpApplicationName может иметь значение NULL. В этом случае имя модуля должно быть первым маркером с разделителями пробелами в строке lpCommandLine.

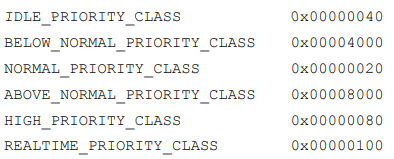
**lpCommandLine**

Командная строка для выполнения.

**dwCreationFlags**

Флаги, управляющие классом приоритета и созданием процесса. Флаги — это всегда отдельные биты, поэтому объединять их надо с помощью операции «логическое ИЛИ».

Флаги приоритета следующие:



**IDLE\_PRIORITY\_CLASS**. Процесс, потоки которого выполняются только в том случае, если система простаивает и вытесняется потоками любого процесса, выполняющегося в классе с более высоким приоритетом. Примером является заставка. Класс неактивного приоритета наследуется дочерними процессами.

**NORMAL\_PRIORITY\_CLASS**. Процесс без особых потребностей в планировании.

**HIGH\_PRIORITY\_CLASS.** Процесс, выполняющий критически важные по времени задачи, которые необходимо выполнить немедленно, чтобы он выполнялся правильно. Потоки процесса класса с высоким приоритетом вытесняют потоки обычных или неактивных процессов класса приоритета. Примером является список задач, который должен быстро реагировать на вызов пользователем независимо от нагрузки на операционную систему. **REALTIME\_PRIORITY\_CLASS**. Процесс, имеющий наивысший возможный приоритет. Потоки процесса класса приоритета в режиме реального времени вытесняют потоки всех остальных процессов, включая процессы операционной системы, выполняющие важные задачи. Например, процесс в режиме реального времени, который выполняется более чем за очень короткий интервал, может привести к тому, что кэши диска не будут очищаться или мышь перестает отвечать на запросы.

**Возвращаемое значение**. Если функция выполняется успешно, возвращается ненулевое значение. Если функция выполняется неудачно, возвращается нулевое значение. Большинство процессов завершаются по окончании своей работы. Добровольно завершить собственный процесс в Windows можно вызовом функции ExitProcess. Другие процессы могут также завершить его вызовом функции TerminateProcess при наличии необходимых разрешений. Третий случай завершения процесса — при возникновении ошибки.

CreateProcess(

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

NULL, // Аргументы командной строки

NULL, // Дескриптор процесса не наследуется

NULL, // Дескриптор потока не наследуется

FALSE, // Флаг наследования

0, // Флаги создания

NULL, // Блок среды процесса

NULL, // Текущий каталог

&si,

&pi

)

Завершение процесса:

Процесс может завершить свою работу самостоятельно, вернувся из функции main

Также существуют инструменты для принудительного завершения процессов **taskkill**

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

Обмен данными - это процесс передачи информации между двумя или более процессами.

типа.

**inter-process communication, IPC(механизм межпроцессного взаимодействия)** — обмен данными между потоками одного или разных процессов. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

[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, критические секции и т.д](https://learn.microsoft.com/ru-ru/windows/win32/ipc/interprocess-communications)

1)Пайпы (Named Pipes):

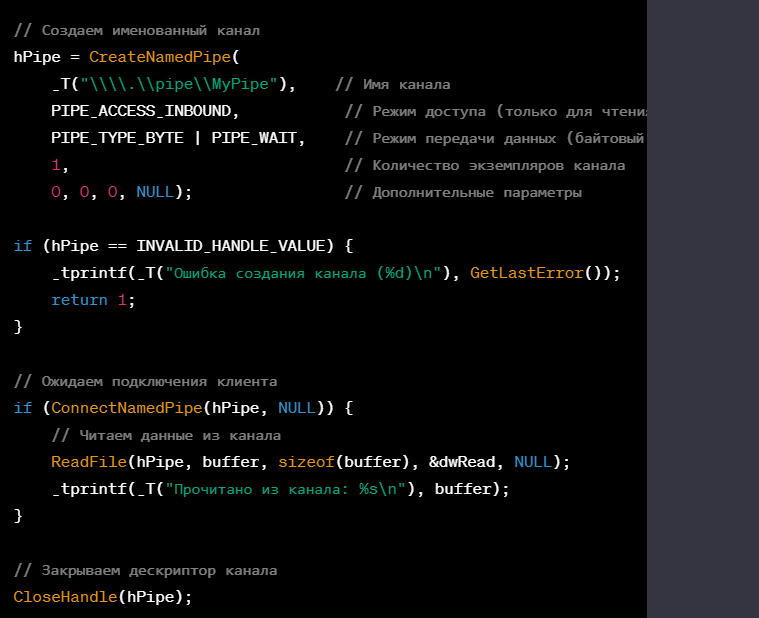
Обмен данными: Пайпы предоставляют именованный канал для обмена данными между процессами. Они могут использоваться как для анонимных, так и для именованных соединений.

Синхронизация: Пайпы поддерживают блокировку, что обеспечивает синхронизированный доступ к данным в разных процессах.

Один процесс (сервер) создает именованный канал, а другой процесс (клиент) открывает этот канал и отправляет или принимает данные.

* **Сервер:**
  1. Создает именованный канал с помощью функции **CreateNamedPipe**.
  2. Ожидает подключения клиента с помощью функции **ConnectNamedPipe**.
  3. После подключения сервер может читать и записывать данные в канал.
* **Клиент:**
  1. Открывает именованный канал с помощью функции **CreateFile**.
  2. После успешного открытия канала клиент может читать и записывать данные в канал.

Сервер:



Клиент:



* 1. [**Образ файла**4](https://lifehacker.ru/kak-otkryt-fajl-iso/)[5](https://hetmanrecovery.com/ru/recovery_news/what-is-an-iso-image-and-how-to-use-it.htm)[6](https://dic.academic.ru/dic.nsf/ruwiki/1070632)[7](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B7_%D1%82%D0%BE%D0%BC%D0%B0)[: Образ файла (или образ диска) - это файл, который содержит полную копию содержания и структуры файловой системы и данных, находящихся на диске, таком как компакт-диск, дискета или раздел жесткого диска6](https://dic.academic.ru/dic.nsf/ruwiki/1070632)[7](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B7_%D1%82%D0%BE%D0%BC%D0%B0). [Образ файла можно представить как полную копию всего, что хранится на физическом оптическом диске, таком как CD, DVD или Blu-ray диск, включая собственную файловую систему5](https://hetmanrecovery.com/ru/recovery_news/what-is-an-iso-image-and-how-to-use-it.htm). [Он представляет собой секторальную копию диска, при этом никакое дополнительное сжатие не используется5](https://hetmanrecovery.com/ru/recovery_news/what-is-an-iso-image-and-how-to-use-it.htm).
* [**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).

2)Межпроцессные Очереди Сообщений (Message Queues):

Обмен данными: который позволяет одному процессу отправлять сообщения другому процессу асинхронно. Этот механизм позволяет процессам обмениваться данными без явного использования разделяемой памяти или прямого вызова функций.

Общие черты межпроцессных очередей сообщений включают в себя:

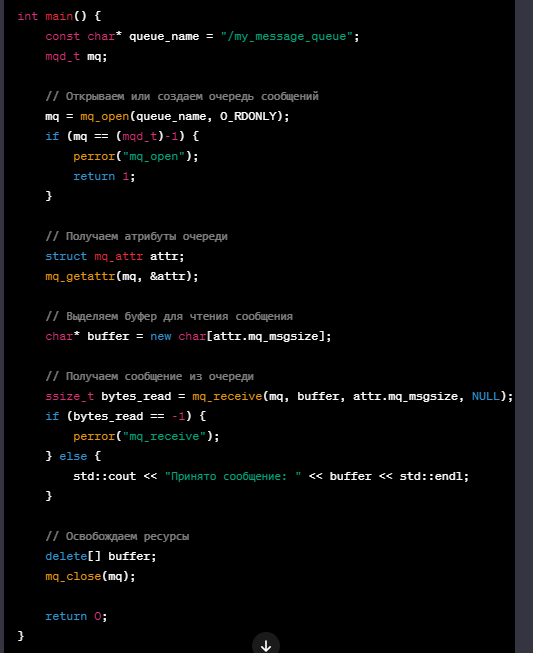
1. **Асинхронность:** Отправитель и получатель могут работать асинхронно, не ожидая ответа друг от друга.
2. **Отделение по времени:** Процессы могут отправлять и принимать сообщения в разные моменты времени.
3. **Гибкость:** Механизм позволяет передавать различные типы данных, включая структуры данных или даже файлы.

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

Сервер:



Клиент:

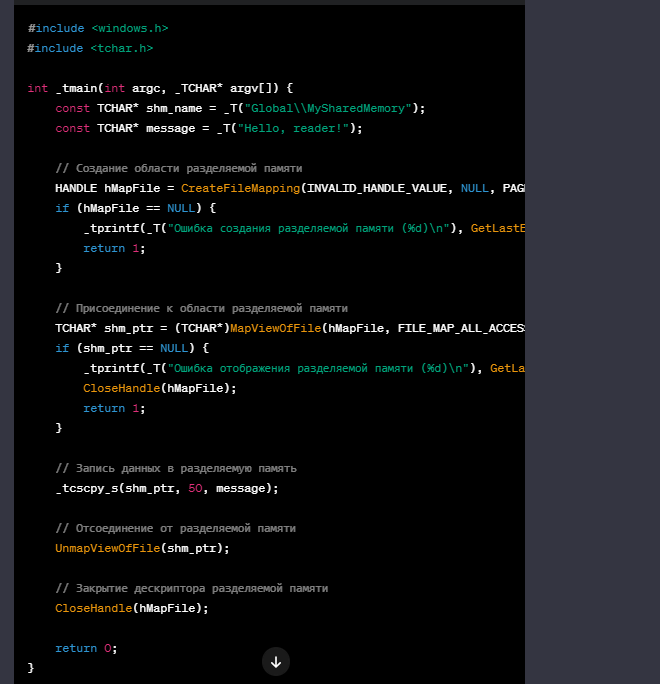


3)Разделяемая Память (Shared Memory):

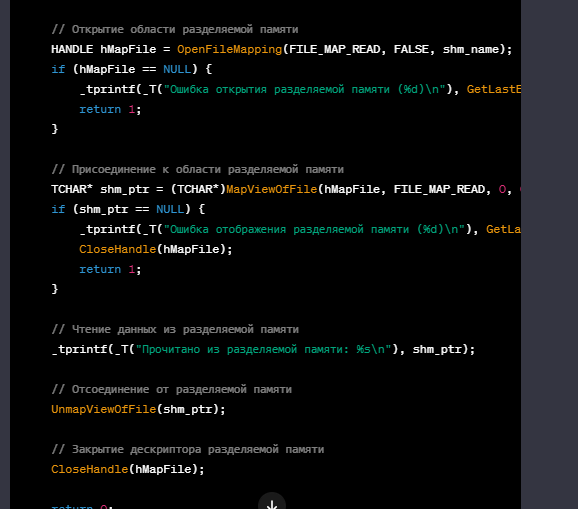
Обмен данными: Разделяемая память позволяет процессам разделять общий участок памяти, в котором они могут записывать и читать данные.

Синхронизация: Для избежания гонок данных, с разделяемой памятью обычно используются объекты синхронизации, такие как мьютексы и семафоры.

Писатель:



Читатель:



4)Мьютексы, Семафоры, Критические Секции:

Обмен данными: Эти объекты синхронизации также могут использоваться для согласования доступа к разделяемым ресурсам, обеспечивая безопасность данных между процессами.

Синхронизация: Мьютексы, семафоры и критические секции обеспечивают синхронизированный доступ к ресурсам.

5)События (Events):

Обмен данными: События могут использоваться для оповещения одного или нескольких процессов о наступлении события.

Синхронизация: События также могут служить механизмом синхронизации, позволяя процессам ожидать определенных событий.

6)RPC (Remote Procedure Call)(пример COM):

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.

### Основные понятия в RPC:

1. **Клиент и Сервер:**
   * **Клиент:** Процесс, который вызывает удаленные процедуры.
   * **Сервер:** Процесс, который предоставляет удаленные процедуры.
2. **Интерфейс:**
   * Определяет список процедур, которые могут быть вызваны удаленно. Обычно описывается в специальном языке описания интерфейса (IDL).
3. **Маршалинг и Демаршалинг:**
   * **Маршалинг:** Процесс упаковки параметров вызова процедуры в структуру данных, которая может быть передана по сети.
   * **Демаршалинг:** Процесс извлечения параметров из упакованной структуры данных.
4. Применение потоков в ОС Windows, API для работы с потоками, API для синхронизации потоков.

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

**Поток (управления) OS** – последовательность инструкций, выполняемых процессором в выделенные OS интервалы времени. Процессы и потоки связаны друг с другом, но при этом имеют существенные различия*.*

**Процесс** — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие (например файлы).

**Поток** использует то же самое пространства стека, что и процесс, а множество потоков совместно используют данные своих состояний. Как правило, каждый поток может работать с одной и той же областью памяти. У каждого потока есть собственные регистры и собственный стек, но другие потоки могут их использовать. Это определенный способ выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса.

**Поток может находиться в одном из трех состояний:**

1. Выполнение: процессор выполняет команды данного потока.
2. Ожидание: поток заблокирован по своим внутренним причинам
3. Готовность: поток готов, но процессор занят выполнением другого потока.

**Есть несколько разных понятий, связанных с областью параллельных вычислений.**

* Конкурентное исполнение (concurrency)
* Параллельное исполнение (parallel execution)
* Многопоточное исполнение (multithreading)
* Асинхронное исполнение (asynchrony)

Каждый из этих терминов строго определен и имеет четкое значение.

**Конкурентность (concurrency)**

Конкурентность (\*) (concurrency) - это наиболее общий термин, который говорит, что одновременно выполняется более одной задачи. Например, вы можете одновременно смотреть телевизор и комментить фоточки в фейсбуке. Винда, даже 95-я могла (\*\*) одновременно играть музыку и показывать фотки.

(\*) К сожалению, вменяемого русскоязычного термина я не знаю. Википедия говорит, что concurrent computing - это параллельные вычисления, но как тогда будет parallel computing по русски?

(\*\*) Да, вспоминается анекдот про Билла Гейтса и многозадачность винды, но, *теоретически* винда могла делать несколько дел одновременно. Хотя и не любых.

Конкурентное исполнение - это самый общий термин, который не говорит о том, каким образом эта конкурентность будет получена: путем приостановки некоторых вычислительных элементов и их переключение на другую задачу, путем действительно одновременного исполнения, путем делегации работы другим устройствам или еще как-то. Это не важно.

*Конкурентное исполнение* говорит о том, что за определенный промежуток времени будет решена более, чем одна задача. Точка.

**Параллельное исполнение**

Параллельное исполнение (parallel computing) подразумевает наличие более одного вычислительного устройства (например, процессора), которые будут *одновременно* выполнять несколько задач.

Параллельное исполнение - это строгое подмножество конкурентного исполнения. Это значит, что на компьютере с одним процессором параллельное программирование - невозможно;)

**Многопоточность**

Многопоточность - это один из способов реализации конкурентного исполнения путем выделения абстракции "рабочего потока" (worker thread).

Потоки "абстрагируют" от пользователя низкоуровневые детали и позволяют выполнять более чем одну работу "параллельно". Операционная система, среда исполнения или библиотека прячет подробности того, будет многопоточное исполнение конкурентным (когда потоков больше чем физических процессоров), или параллельным (когда число потоков меньше или равно числу процессоров и несколько задач физически выполняются одновременно).

**Асинхронное исполнение**

Асинхронность (asynchrony) подразумевает, что операция может быть выполнена кем-то на стороне: удаленным веб-узлом, сервером или другим устройством за пределами текущего вычислительного устройства.

Основное свойство таких операций в том, что начало такой операции требует значительно меньшего времени, чем основная работа. Что позволяет выполнять множество асинхронных операций одновременно даже на устройстве с небольшим числом вычислительных устройств.

**Схема работы выполнения асинхронной операции: заявка на получение – получения результата. Если один процессор - нет возможности выполнения асинхронных операций.**

**Асинхронная операция** может выполняться как в основном потоке программы, так и в другом. Кроме того, программист явно не знает, как будет выполняться данная операция. Например, при обращении к пулу потоков на платформе .Net компилятор самостоятельно определяет в каком потоке будет происходить выполнение программы.

**API для работы с потоками в С++:**

Функция [CreateThread](http://vsokovikov.narod.ru/New_MSDN_API/Process_thread/fn_createthread.htm) создает для процесса новый поток. Созданный поток должен определить начальный адрес кода, с которого новый поток должен исполняться. Как правило, начальный адрес — это название функции, определенной в коде программы. Эта функция получает единственный параметр и возвращает значение типа DWORD. Процесс может иметь одновременно несколько потоков, выполняющих ту же самую функцию.

*HANDLE CreateThread(*

*LPSECURITY\_ATTRIBUTES lpsa,*

*DWORD dwStackSize,*

*LPTHREAD\_START\_ROUTINE lpStartAddr,*

*LPVOID lpThreadParm,*

*DWORD dwCreationFlags,*

*LPDWORD lpThreadId*

*)*

lpsa — указатель на структуру атрибутов защиты.

dwStackSize — размер стека нового потока в байтах. Значению 0 – размер по умолчанию, равный размеру основного потока.

lpStartAddr — указатель на функцию, которая должна выполняться.

lpThreadParm — указатель, передаваемый потоку в качестве аргумента.

dwCreationFlags — если значение этого параметра установлено равным 0, то поток запускается сразу же после вызова функции CreateThread.

*lpThreadId* – идентификатор потока.

CREATE\_SUSPENDED приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности путем вызова функции ResumeThread.

По завершении выполнения потока память, занимаемая ее стеком, освобождается. В случае если поток был создан в библиотеке DLL, будет вызвана соответствующая точка входа DllMain с указанием флага DLL\_THREAD\_DETACH в качестве "причины" этого вызова.

*VOID ExitThread(*

*DWORD dwExitCode*

*)*

Любой поток процесса может сам завершить свое выполнение, вызвав функцию ExitThread.

Когда завершается выполнение последнего потока, завершается и выполнение самого процесса.

Выполнение потока также может быть завершено другим потоком с помощью функции TerminateThread, однако освобождения ресурсов потока при этом не происходит.

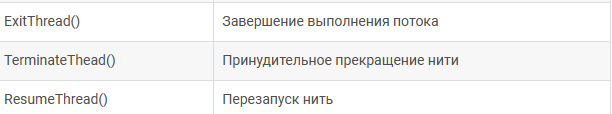
Поток, выполнение которого было завершено, продолжает существовать до тех пор, пока посредством функции CloseHandle не будет закрыт ее последний дескриптор. Любой другой поток, возможно и такой, который ожидает завершения другого потока, может получить код завершения потока.

Поток может увеличивать или уменьшать значение счетчика приостановок другого потока с помощью функций SuspendThread и ResumeThread.

Чтобы создавать поток, который запускается в виртуальном адресном пространстве другого процесса, используется функция [CreateRemoteThread](http://narovol.narod.ru/_tbkp/New_MSDN_API/fn_createremotethread.htm).

Поток создается с приоритетом потока THREAD\_PRIORITY\_NORMAL. Используйте функции [GetThreadPriority](http://narovol.narod.ru/_tbkp/New_MSDN_API/fn_getthreadpriority.htm) и [SetThreadPriority](http://narovol.narod.ru/_tbkp/New_MSDN_API/fn_setthreadpriority.htm), чтобы получать и установить приоритетное значение потока.

В Windows-программах не слишком широко применяются стандартные типы данных из С или С++, такие как int или char\*. Вместо них используются типы данных, определенные в различных библиотечных (header) файлах. Наиболее часто используемыми типами являются HANDLE, HWND, BYTE, WORD, DWORD, UNIT, LONG, BOOL, LPSTR и LPCSTR.



механизмы синхронизации в 6 вопросе

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

При одновременном доступе нескольких процессов (или нескольких потоков одного процесса) к какому-либо ресурсу возникает проблема синхронизации. Поскольку поток в Win32 может быть остановлен в любой, заранее ему неизвестный момент времени, возможна ситуация, когда один из потоков не успел завершить модификацию ресурса, но был остановлен, а другой поток попытался обратиться к тому же ресурсу. В этот момент ресурс находится в несогласованном состоянии, и последствия обращения к нему могут быть самыми неожиданными — от порчи данных до нарушения защиты памяти.

Главной идеей, заложенной в основе синхронизации потоков в Win32, является использование объектов синхронизации и функций ожидания.

Объекты находиться в одном из двух состояний — Signaled или Not Signaled(ожидаем).

Функции ожидания блокируют выполнение потока до тех пор, пока заданный объект находится в состоянии Not Signaled.

Поток, когда использует ресурс ставит его в несигнальное состояние, по окончанию использование сбрасывает в сигнальное

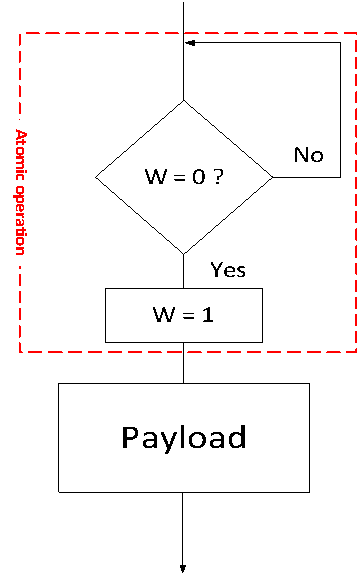
Остальные потоки- вызывают функцию ожидания, которая позволит дождаться освобождения ресурса.

Синхронизация - механизм упорядочивания выполнения программных блоков двух или более потоков.

**Синхронизация –** механизмустановления порядка работы потоков

Взаи́мная блокиро́вка — ситуация в многозадачной среде, при которой несколько процессов находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать свое выполнение.

принцип реализации синхронизации (общая память)



Механизмы синхронизации потоков в Windows:

- Critical section;

- Mutex;

- Semaphore;

- Atomic operation (interlocking function)

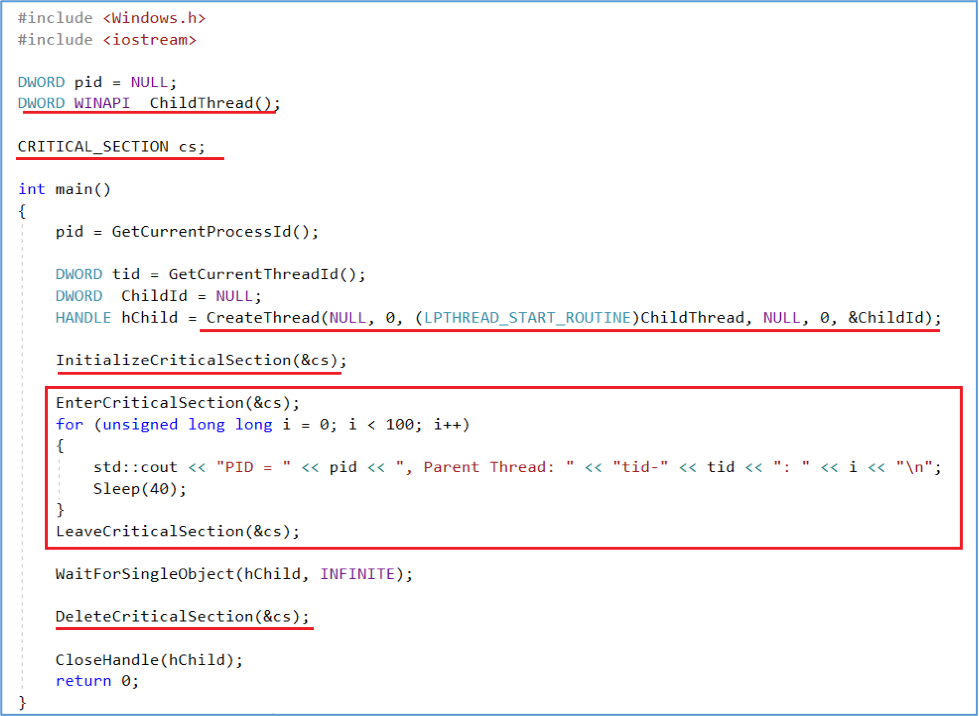
- Event;

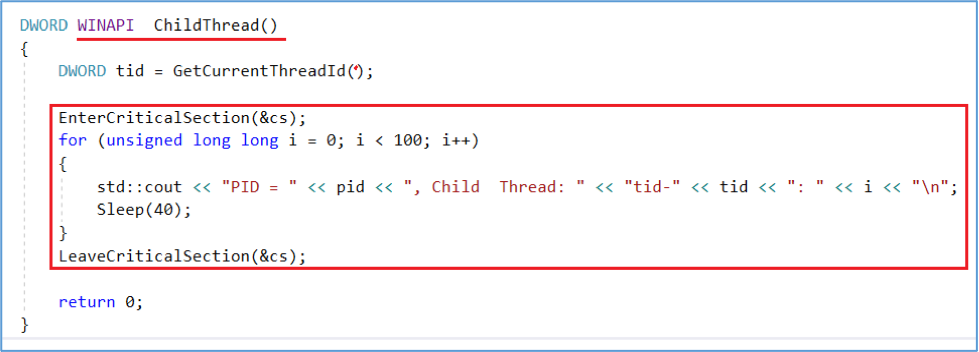
- Writable timer.

Critical section — механизм синхронизации нескольких потоков одного процесса. НЕ ЯВЛЯЕТСЯ объектом ядра OS.

**Критическая секция** – спец механизм, который может работать в рамках одного потока и не требует системного вызова.

Critical section for 2 threads:





**Мьютексы** – частный случай семафора.

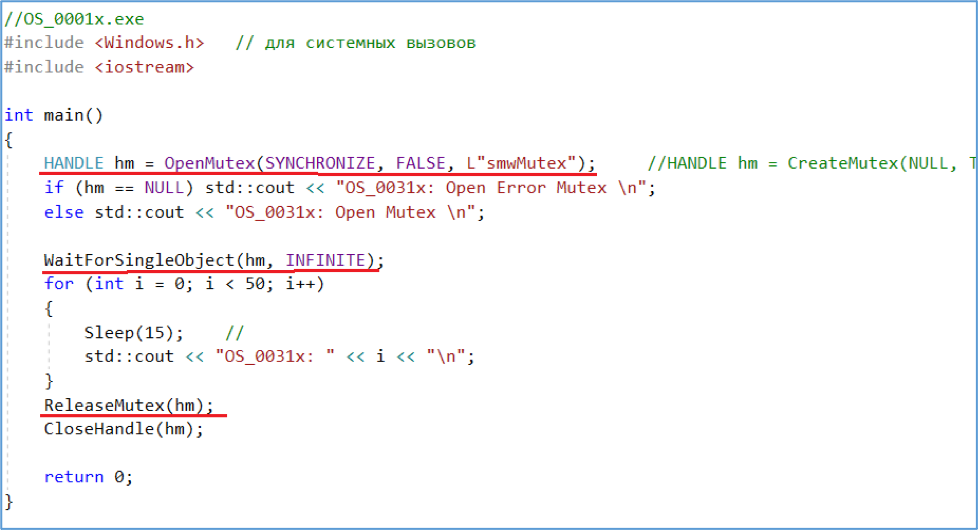
Mutex – механизм синхронизации нескольких потоков разных процессов, является объектом ядра OS. Как только хотя бы один процесс запрашивает владение мьютексом, он переходит в несигнальное состояние и остается таким до тех пор, пока не будет освобожден владельцем. Такое поведение позволяет использовать 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, критическая секция же принадлежит процессу и служит для синхронизации только его потоков.

* **Мьютекс:** Часто используется для синхронизации между процессами или потоками, работающими в разных процессах.
* **Критическая секция:** Применяется для синхронизации потоков внутри одного процесса.

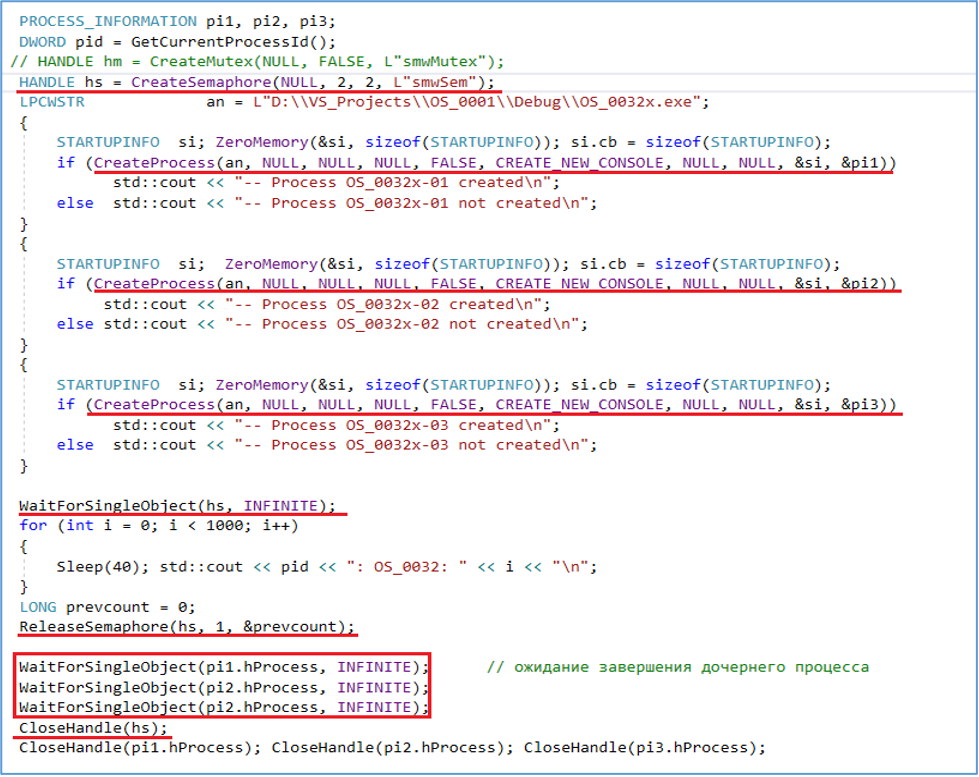
Mutex for 2 processes:

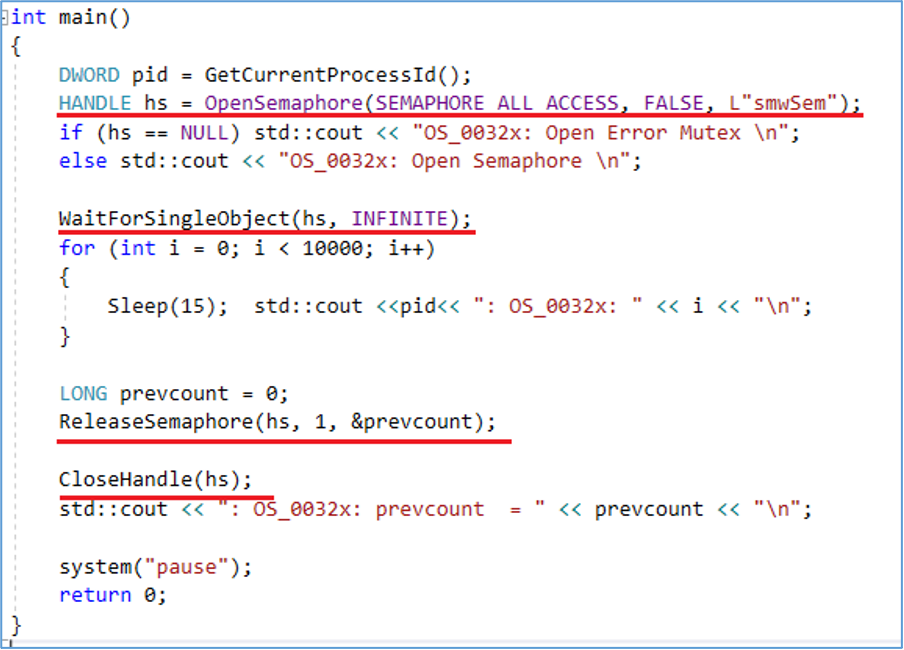




Semaphore - представляет собой счетчик, содержащий целое число в диапазоне от 0 до максимальной величины, заданной при его создании. Когда поток (или процесс) захватывает ресурс, счетчик семафора уменьшается. При достижении семафором значения 0 он переходит в несигнальное состояние, при любых других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

Semaphore for 4 processes:





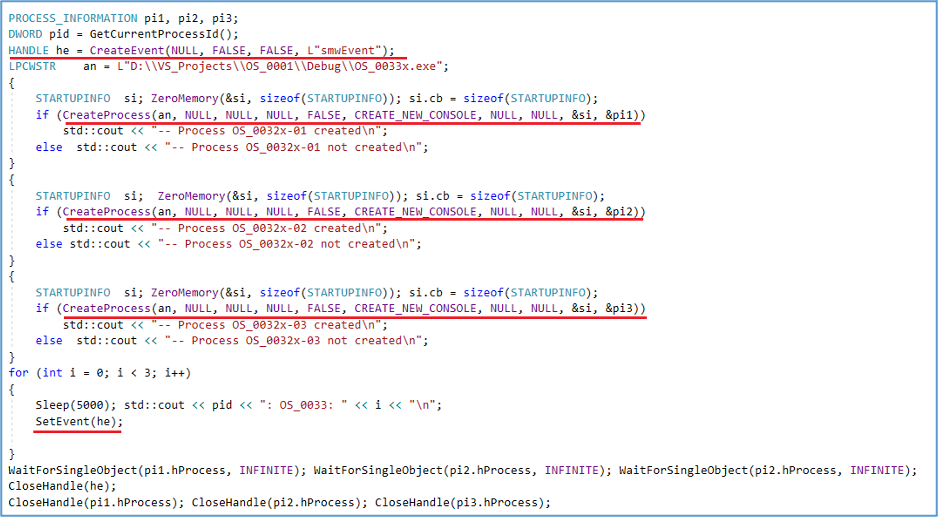
Event - объекты-события используются для уведомления ожидающих нитей (потоков, процессов) о наступлении какого-либо события. Различают два вида событий - с ручным и автоматическим сбросом.

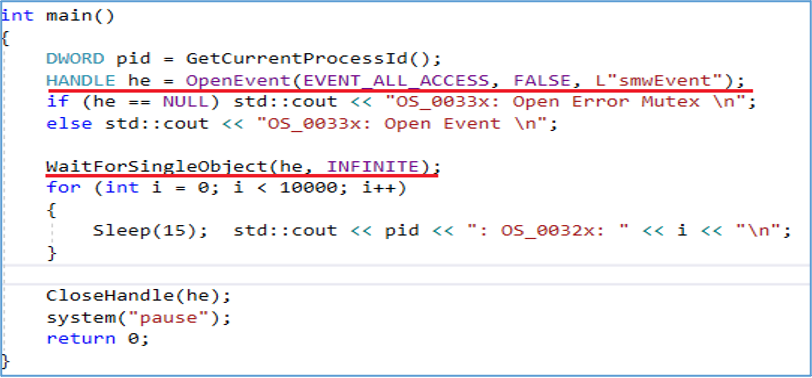
Ручной сброс осуществляется функцией ResetEvent. События с ручным сбросом используются для уведомления сразу нескольких нитей.

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.

Event, SetEvent, WaitForSingleObject



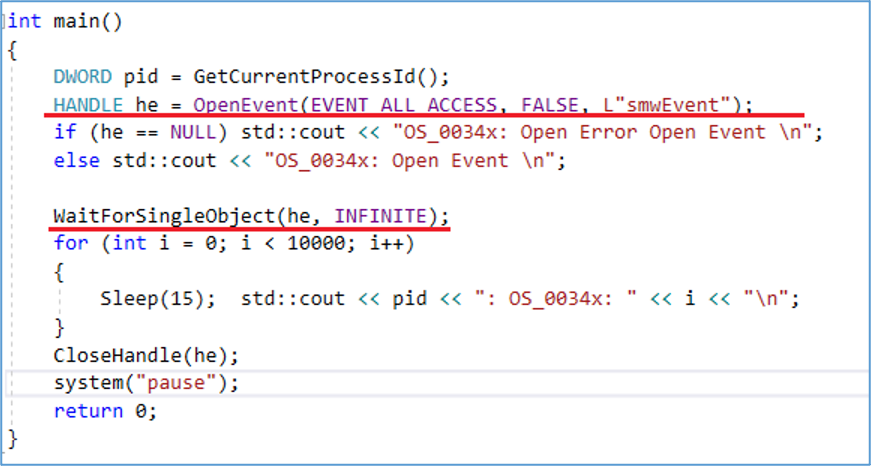


OS: Event, SetPulse, WaitForSingleObject

**SetEvent** используется для установки события в сигнальное состояние.

**WaitForSingleObject** используется для блокировки текущего потока до тех пор, пока событие не перейдет в сигнальное состояние или не истечет время ожидания.





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.

атомарные операции в Windows

**Атомарные операции** позволяют нам синхронизировать работу потоков для работы с общими областями памяти. Можем менять счётчики, значения и тд. Атомарные операции позволяют сделать так, чтобы изменение общих областей памяти выполнялось корректно.

Атомарная операция — 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.

синхронизация механизмы апи какими способами можно синхронизировать

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.

## определение файловой системы

порядок, определяющий способ организации, хранения и именования данных на носителях информации в компьютерах, а также в другом электронном оборудовании

**система управления файлами,** часть операционной системы, обеспечивающая доступ к файлам. Устанавливает связь между логическим представлением и физическим расположением данных (абстракция над данными).

## Файловая система: логическая и физическая организация данных

**Логическая организация файловой системы:**

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

Она включает в себя следующие аспекты:

* **Файлы и директории**: Файлы представляют собой логические единицы данных, которые могут содержать текст, изображения, звуки и другие типы информации. Директории служат для организации файлов в иерархической структуре. Логические пути и имена файлов и директорий позволяют пользователям и программам обращаться к данным.
* **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 и т. д.).
* **Метаданные**: Логическая организация также включает в себя метаданные, такие как атрибуты файла (например, дата создания и размер) и информацию о размещении файла в файловой системе.
* **Системы файлов**: Различные операционные системы используют разные системы файлов (например, NTFS в Windows, ext4 в Linux), которые определяют способы хранения и организации данных на диске.
* Типы файлов;
* Иерархическая структура файловой системы;
* Имена файлов;
* Атрибуты файлов;
* Логическая организация файла.

**Физическая организация файловой системы:**

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

Это включает в себя следующие аспекты:

* **Блоки и сектора**: Физический носитель информации разбивается на блоки или секторы, которые являются минимальными единицами хранения. Файлы размещаются в блоках, исходя из их размера.
* **Форматирование**: Физическая организация включает в себя процесс форматирования носителя информации, в ходе которого создаются структуры данных, необходимые для работы файловой системы (например, таблицы размещения файлов).
* **Дисковая геометрия**: Файловая система учитывает параметры физического диска, такие как количество секторов на дорожке и число головок, для оптимизации доступа к данным.
* Физическая организация и адресация файла;
* Физическая организация FAT;
* Физическая организация NTFS.

## отличие файловых систем

В зависимости от выбора пользователя будут меняться методы работы с файлами, обращения к конфигурации ядра и способы хранения данных в ОП.

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 системыg

Оглавление (или таблица содержимого) файловой системы представляет собой структуру данных, которая содержит информацию о расположении файлов и директорий в файловой системе. Она обычно используется для быстрого доступа к файлам и определения их местоположения на диске.

File System, файловая система, система управления файлами

1. NTFS (файловая система новой технологии): разработана Microsoft и используется в операционной системе Windows.
2. FAT (таблица размещения файлов/таблица распределения файлов): более старая файловая система, которая широко использовалась в MS-DOS и ранних версиях Windows.
3. exFAT (расширенная таблица размещения файлов): современная версия FAT, поддерживающая файлы большего размера.
4. HFS (иерархическая файловая система): разработана Apple и используется в операционной системе Macintosh.
5. APFS (Apple File System): современная файловая система, разработанная Apple и используемая в их новых операционных системах.
6. ext2, ext3, ext4: наиболее часто используемые файловые системы в операционной системе Linux.
7. XFS: высокопроизводительная файловая система, используемая во многих системах на базе Linux.
8. Btrfs: файловая система с копированием при записи, которая часто используется для моментальных снимков системы и резервного копирования.
9. ReFS (Resilient File System/ReiserFS): журналируемая файловая система, которая обычно использовалась в системах на базе Linux.

## файлы

набор логических записей © Смелов

## каталоги

файл содержащий информацию о месте расположения других файлов; © Смелов

Каталог - группа файлов на одном носителе, объединенных(сгруппированных) по какому-то признаку.

Каталог в файловой системе — это место, содержащее другие файлы и каталоги. Он также известен как папка. Каталоги используются для организации и хранения файлов в структурированном виде.

специальные каталоги: .(точка), .. (две точки);

## основные функции файловой системы

создание/удаление каталогов,

включение/исключение подкаталогов,

включение/исключение файла в каталог, создание/удаление файла, открытие/закрытие доступа к файлу, чтение/запись логических записей файла, установка (поддержка) указателя файла.

## буферы ввода/вывода

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

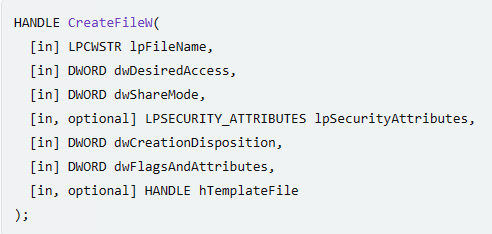
Буферы ввода-вывода используются для ускорения и оптимизации ввода-вывода данных. Они позволяют отложить чтение или запись данных для уменьшения числа вызовов низкоуровневых функций чтения/записи и снижения количества операций чтения/записи на диске. Это может улучшить производительность приложения и повысить эффективность использования ресурсов системы.

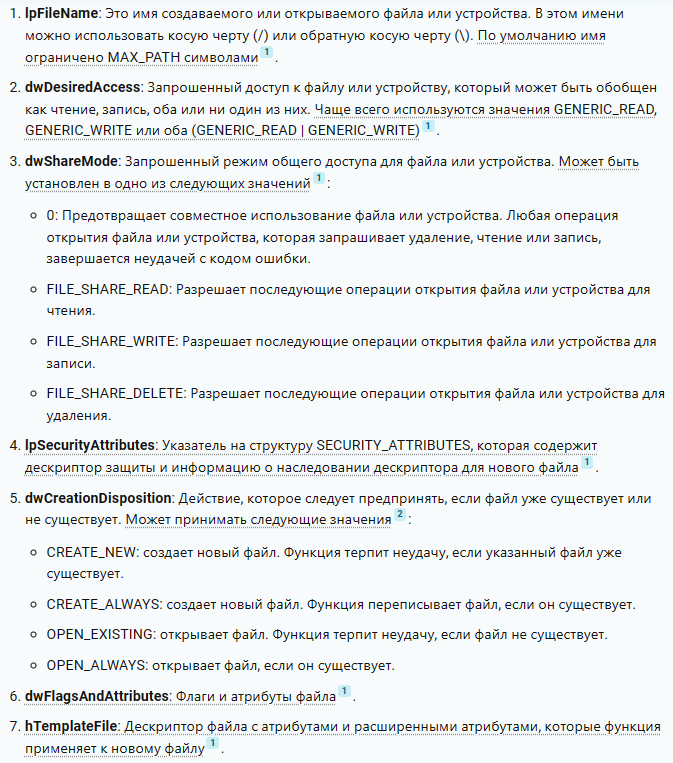
## кеширование ввода/вывода

Перемещение в быстродействующую память, наиболее часто используемых данных (обычно упреждающее чтение). В состав ядра OS входит специальная программа обеспечивающая кэширование данных. Кроме того кэширование осуществляет контроллер дисковода.

## основные функции API файловой системы

1. **CreateFile** (создание/чтение файла)





1. **Delete file**
2. **WriteFile**
3. **ReadFile**
4. **CopyFile**
5. **MoveFile**
6. **ReplaceFile** (заменяет файл в другом месте, сохраняя его атрибцты и текущую дату)
7. **LockFile**(блокировка на определённый диапазон байтов в файле)
8. **UnlockFile**
9. **LockFileEx**
10. **Windows. Функция OS API для удаления файла.**

DeleteFile

1. **Windows. Функция OS API для записи в файл.**

WriteFile

1. **Windows. Функция OS API для чтения файла.**

ReadFile

1. **Windows. Назначение и отличие функций OS API: CopyFile, MoveFile, ReplaseFile.**

CopyFile - копирует файл в другое место.

MoveFile - перемещает файл в другое место.

ReplaceFile - заменяет файл в другом месте, сохраняя его атрибуты и текущую дату.

1. **Windows. Перечислите функции OS API, которые изменяют текущее значение указателя позиции файла.**

SetFilePointer,

* **SetFilePointer** используется для перемещения указателя текущей позиции файла.
* Принимает в качестве параметров дескриптор файла, смещение относительно указанной позиции и флаг, указывающий, как интерпретировать это смещение

SetEndOfFile,

* спользуется для установки конечной позиции файла.
* Обычно используется для изменения размера файла. Если новый размер файла больше текущего, он будет заполнен нулями.

WriteFile.

* **WriteFile** используется для записи данных в файл.
* Принимает в качестве параметров дескриптор файла, указатель на буфер с данными для записи, количество байтов для записи и указатель на переменную, в которую будет записано фактическое количество записанных байтов.

1. **Windows. Перечислите функции OS API для блокировки и разблокировки файлов.**

LockFile,

В зависимости от режима блокировки : один поток блокирует следовательно безопасно выполняет отперации другие могут или только читать или вообще не могут получить доступ

* **LockFile** используется для установки блокировки на определенном диапазоне байтов в файле.
* Принимает в качестве параметров дескриптор файла, смещение и количество байтов для блокировки.

UnlockFile и

LockFileEx.

* **LockFileEx** предоставляет более гибкий способ управления блокировками и включает дополнительные опции.
* Принимает в качестве параметров дескриптор файла, режим блокировки (эксклюзивная или разделенная), смещение, количество байтов и другие параметры.
* Может использоваться для асинхронного ожидания блокировки.

## маркер файла

**Поясните понятие «маркер конца файла».**

"End-of-file marker" (маркер конца файла) - это специальный символ или флаг, который указывает на конец файла. Когда программа читает файл до достижения маркера конца файла, она знает, что файл закончился.

## текущая позиция файла

"File position pointer" (указатель позиции 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.

или??

**указатель позиции файла** - объект файловой системы, позиционирующий логическую запись.

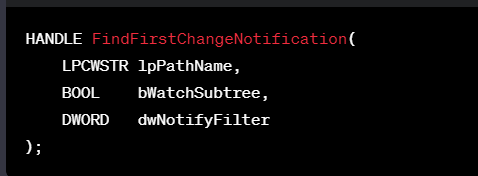
## блокировка файлов

"File lock" (блокировка файла) - это механизм, который запрещает другим процессам или программам изменять или использовать файл, который в данный момент заблокирован. Это используется для обеспечения целостности данных в файле.

## наблюдение за изменением в каталоге

Механизм "просмотра каталога" в Windows позволяет приложению получать уведомления об изменениях в каталоге, таких как создание, удаление или переименование файлов. Набор функций API, которые могут быть использованы для реализации этого механизма, включает в себя

FindFirstChangeNotification,



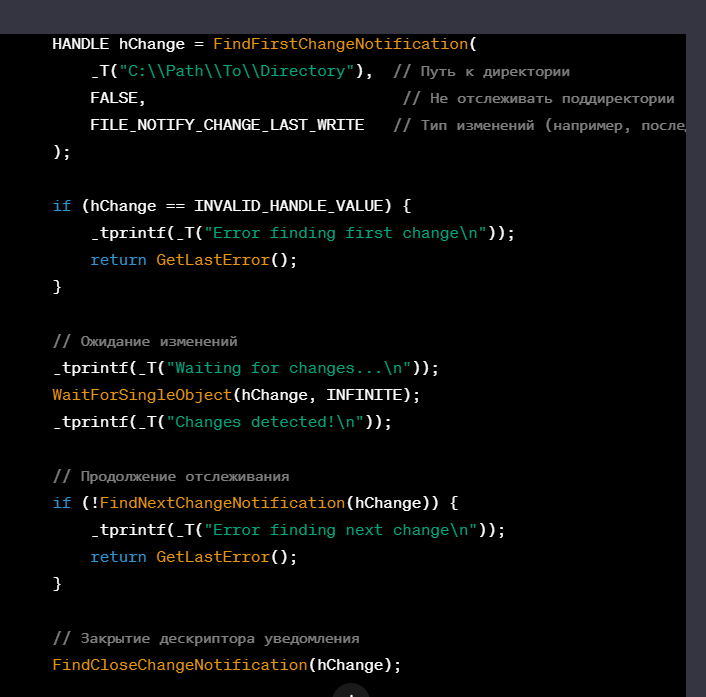
* **lpPathName**: Путь к директории, которую нужно отслеживать.
* **bWatchSubtree**: Если установлено в **TRUE**, отслеживаются изменения в поддиректориях.
* **dwNotifyFilter**: Фильтр, указывающий типы изменений, которые нужно отслеживать (например, **FILE\_NOTIFY\_CHANGE\_FILE\_NAME** для изменений имен файлов).

FindNextChangeNotification

ждет, пока произойдут изменения в отслеживаемой директории, и возвращает управление после того, как изменения произошли. Затем она может быть вызвана повторно для продолжения отслеживания.

FindCloseChangeNotification.

закрывает дескриптор уведомления о изменениях, освобождая связанные с ним ресурсы.



## 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.

### Ф**айловые системы**

**Основные: Ext, Jfs, RFS, xfs, Btrfs**

1. JFS (Journaled File System) — файловая система для операционной системы Linux. Это высокопроизводительная файловая система с журналированием, предназначенная для использования в системах корпоративного уровня. JFS поддерживает такие функции, как быстрое восстановление после сбоев, высокопроизводительный доступ к файлам и эффективное использование дискового пространства. Он предназначен для использования в больших системах хранения и оптимизирован для систем высокой доступности, включая конфигурации кластеризации и зеркалирования. JFS доступна в ядре Linux, начиная с версии 2.4.
2. Ext2/3/4: это распространенный формат файловой системы для Linux, является последовательным и надежным.
3. XFS: это быстрый и эффективный формат файловой системы для Linux, который подходит для больших объемов данных.
4. Btrfs: это новый формат файловой системы для Linux, предлагающий много инновационных возможностей, таких как управление копиями, управление резервными копиями и сжатие данных.
5. ReiserFS: это журналируемая файловая система для Linux. ReiserFS использует сбалансированную древовидную структуру данных для хранения и организации файлов, что обеспечивает эффективный доступ к файлам и управление ими. Он также поддерживает такие функции, как неограниченное количество жестких ссылок на файл, быстрое удаление и восстановление файлов, упаковка хвостов и эффективное хранение небольших файлов. Однако она была в значительной степени заменена другими файловыми системами, такими как Ext4, из-за проблем с производительностью и стабильностью.
6. 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** - , 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.

Основное **отличие**между оперативной и виртуальной памятью заключается в скорости и объеме.

**Оперативная память**обычно имеет меньший объем, но высокую скорость доступа, поскольку она находится близко к процессору.

**Виртуальная память** обычно имеет больший объем, но низкую скорость доступа, поскольку она находится на жестком диске.

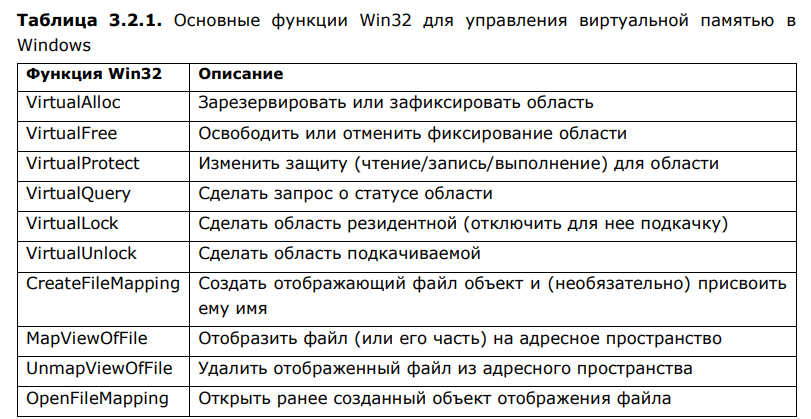
Кроме того, оперативная память является летучей, то есть теряет данные при отключении питания, а виртуальная память является постоянной, то есть сохраняет данные при отключении питания. Она является логическим адресным пространством, которое операционная система выделяет каждому процессу для доступа к памяти. Операционная система использует виртуальную память для управления доступом приложений к физической памяти и для обеспечения изоляции адресного пространства между процессами. Часть виртуальной памяти может быть загружена в физическую память, но это не означает, что вся виртуальная память соответствует физической памяти.

**Преимуществом**использования виртуальной памяти является то, что она позволяет запускать больше программ и обрабатывать больше данных, чем физически установлено оперативной памяти в компьютере. Таким образом, виртуальная память расширяет возможности компьютера и делает его более производительным.

**Недостатком**использования виртуальной памяти является то, что она замедляет работу компьютера из-за необходимости перемещать данные между оперативной и виртуальной памятью. Таким образом, виртуальная память требует дополнительных ресурсов и управления со стороны операционной системы.

[**Куча** - это область оперативной памяти компьютера, в которой возможно в динамическом режиме выделять память для различных программных нужд](https://tproger.ru/translations/programming-concepts-stack-and-heap). [Куча представляет собой непрерывную область памяти, поделённую на занятые и свободные области (блоки) различного размера](https://bing.com/search?q=). [Программы используют кучу для размещения динамически создаваемых структур данных](https://bing.com/search?q=).

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

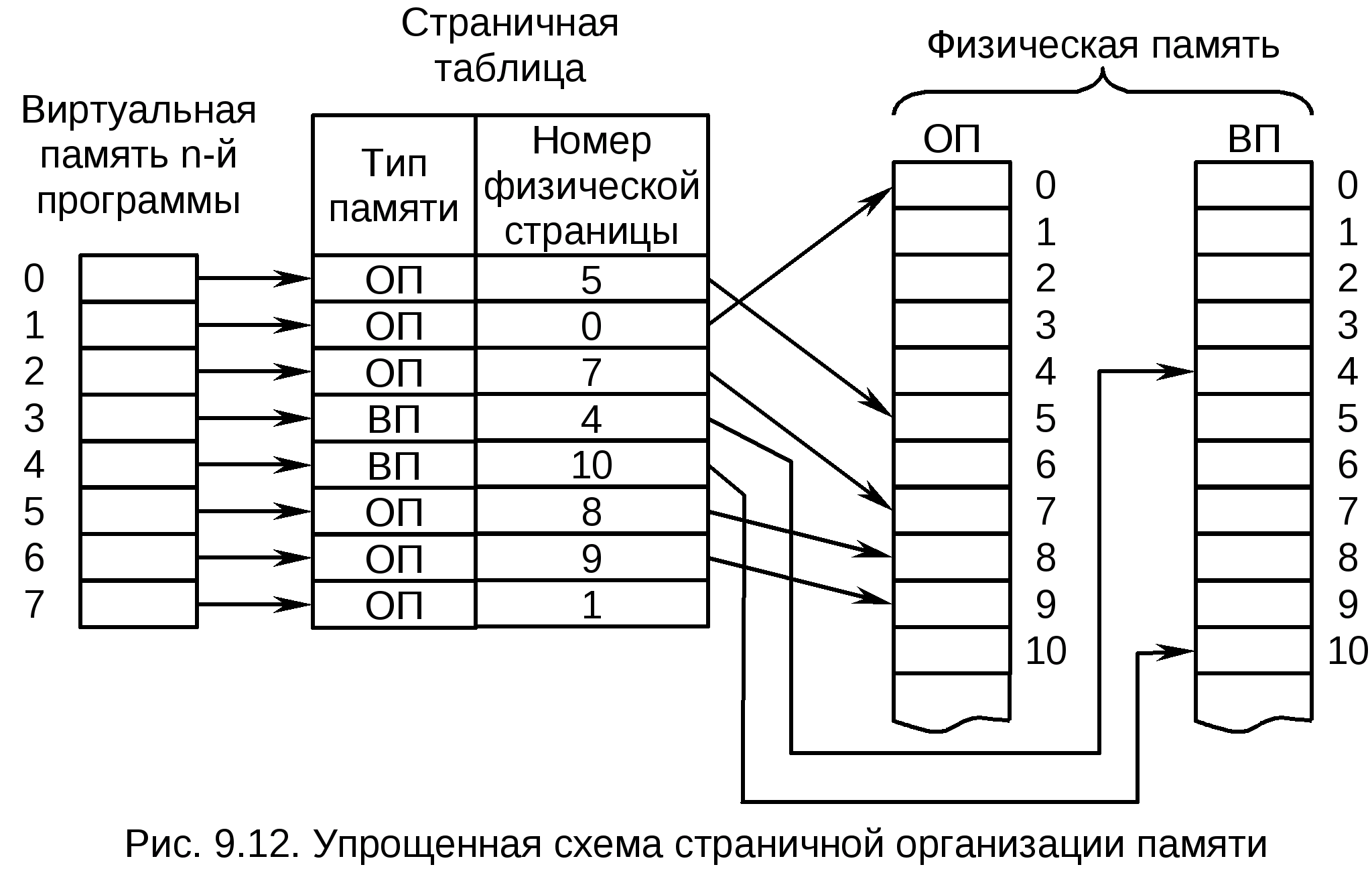


**Функции API OS для работы с heap:**

Каждый процесс имеет кучу по умолчанию, предоставляемую системой. Приложения, которые часто выделяют ресурсы из кучи, могут повысить производительность с помощью частных кучи [8].

**Частная куча** — это блок одной или нескольких страниц в адресном пространстве вызывающего процесса. После создания частной кучи процесс использует такие функции, как HeapAlloc и HeapFree, для управления памятью в этой куче.

Функции кучи также можно использовать для управления памятью в куче процесса по умолчанию с помощью дескриптора, возвращаемого функцией GetProcessHeap. Для этой цели новые приложения должны использовать функции кучи вместо глобальных и локальных функций.



Нет различий между памятью, выделенной из частной кучи, и памятью, выделенной с помощью других функций выделения памяти.

Функция **HeapCreate** создает частный объект кучи, из которого вызывающий процесс может выделять блоки памяти с помощью функции **HeapAlloc**. **HeapCreate** указывает как начальный, так и максимальный размер кучи. Начальный размер определяет количество зафиксированных (commited) страниц для чтения и записи, изначально выделенных для кучи. Максимальный размер определяет общее количество зарезервированных страниц. Эти страницы создают непрерывный блок в виртуальном адресном пространстве процесса, в который может расти куча. Дополнительные страницы автоматически фиксируются из этого зарезервированного пространства, если запросы **HeapAlloc** превышают текущий размер зафиксированных страниц, при условии, что физическое хранилище для него доступно. После фиксации страницы не удаляются до завершения процесса или до тех пор, пока куча не будет уничтожена путем вызова функции **HeapDestroy**.

Функция **HeapAlloc** выделяет указанное количество байтов из частной кучи и возвращает указатель на выделенный блок. Этот указатель можно использовать в функциях **HeapFree, HeapReAlloc, HeapSize и HeapValidate**.

Память, выделенная **HeapAlloc**, не перемещается. Адрес, возвращаемый **HeapAlloc**, действителен до освобождения или перераспределения блока памяти; Блок памяти не нужно блокировать.

Функции кучи можно использовать для создания частной кучи при запуске процесса, указав начальный размер, достаточный для удовлетворения требований процесса к памяти. Если вызов функции **HeapCreate** завершается сбоем, процесс может завершиться или уведомить пользователя о нехватке памяти; Однако, если он будет успешным, процесс будет гарантированно иметь необходимую память. Память, запрашиваемая методом **HeapCreate**, может быть непрерывной и может не быть непрерывной.

Память, выделенная в куче **HeapAlloc**, является непрерывной. Не следует выполнять запись или чтение из памяти в куче, за пределами диапазона памяти, выделенной **HeapAlloc**, и не следует предполагать связь между двумя областями памяти, выделенными **HeapAlloc**.

Вы не должны каким-либо образом ссылаться на память, которая была освобождена **HeapFree**. После освобождения памяти любая информация, которая могла быть в ней, исчезает навсегда. Если требуется информация, не освобождайте память, содержащую эти сведения. Вызовы функций, возвращающие сведения о памяти (например, HeapSize), не могут использоваться с освобожденной памятью, так как они могут возвращать фиктивные данные. Функция **HeapDestroy** уничтожает частный объект кучи. Он удаляет и освобождает все страницы объекта кучи, а также делает дескриптор кучи недействительным.

1. Механизм отображение файлов в памяти: последовательность системных вызовов Windows для создания образа файла в оперативной памяти, использование образа файла, как средства межпроцессного взаимодействия.

File Mapping - отображение файла в виртуальную память; отображенный файл называется File View (представлением файла).

**File Mapping** ­– механизм, позволяющий отобразить файл или его фрагмент в оперативной памяти. Этот образ можно использовать далее в нескольких процессах и для межпроцессного обмена данных. Нужен для ускоренной обработки данных.

Оперативная память(RAM) - это часть компьютерной памяти, которая используется для хранения данных и инструкций, которые процессор должен быстро обработать. Она является физической памятью, установленной на материнской плате компьютера.

Виртуальная память - это механизм операционной системы, который обеспечивает приложениям доступ к большому объему памяти, превышающему объем физической памяти (RAM) в компьютере. Она является логическим адресным пространством, которое операционная система выделяет каждому процессу для доступа к памяти. Операционная система использует виртуальную память для управления доступом приложений к физической памяти и для обеспечения изоляции адресного пространства между процессами. Часть виртуальной памяти может быть загружена в физическую память, но это не означает, что вся виртуальная память соответствует физической памяти.

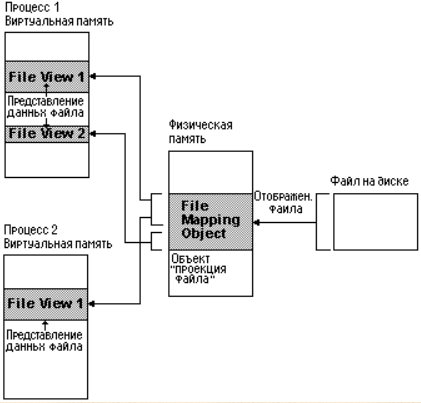
Нужен для:

1. ускорения работы
2. для межпроцессного взаимодействиями (можно разделить общий фрагмент памяти между разными процессами).

Можно загрузить из оперативной памяти как весь файл, так и его часть.

Каждый процесс имеет свое собственное виртуальное адресное пространство, поэтому данные, отображенные через файловое отображение, доступны только в пределах этого процесса. Однако, если несколько процессов отображают один и тот же файл, они могут совместно использовать данные через общую виртуальную адресную область.

Также можно создать несколько отображений файла в память, каждое из которых отображает свою часть файла. Например, можно создать два отображения файла, одно отображает первые 100 байт, а другое отображает следующие 100 байт. В этом случае два отображения будут использовать общий файл на диске, но каждое отображение будет отображать свою уникальную часть файла в память.



(С Википедии: File Mapping — это способ работы с файлами в некоторых ОС, при котором всему файлу или некоторой непрерывной его части ставится в соответствие определенный участок памяти)

При этом чтение данных из этих адресов фактически приводит к чтению данных из отображенного файла, а запись данных по этим адресам приводит к записи этих данных в файл.

Преимущества:

Дополнительным преимуществом использования отображения является меньшая, по сравнению с чтением/записью, нагрузка на ОС — дело в том, что при использовании отображений ОС не загружает в память сразу весь файл, а делает это по мере необходимости, блоками размером со страницу памяти (как правило, 4 килобайта). Таким образом, даже имея небольшое количество физической памяти (например, 32 мегабайта), можно легко отобразить файл размером 100 мегабайт или больше, и при этом для системы это не приведет к большим накладным расходам.

Файл, отображенный на память, удобен также тем, что можно достаточно легко менять его размер и при этом получать в своё распоряжение непрерывный кусок памяти нужного размера.

Более одного процесса могут использовать файл на диске как для чтения, так и для записи. Каждый процесс может создать новое представление, не отображая текущее представление файла.

Недостатки:

Использование отображений чревато замедлениями из-за страничных ошибок доступа. Допустим, страница, относящаяся к нужному файлу, уже лежит в кэше, но не ассоциирована с данным отображением. Если она была изменена другим процессом, то попытка ассоциировать её с отображением может закончиться неудачей и привести к необходимости повторно зачитывать данные с диска либо сохранять данные на диск.

Другой недостаток в том, что размер отображения зависит от используемой архитектуры. Теоретически, 32-битные архитектуры не могут создавать отображения длиной более 4 Гб.

последовательность системных вызовов Windows для создания образа файла в оперативной памяти

Шаг 1. Создайте или откройте файловый объект, представляющий файл на диске.

HANDLE hFile = CreateFile (TEXT ("datafile.txt"),

GENERIC\_READ | GENERIC\_WRITE,

0, // открытие с монопольным доступом

NULL, // без атрибутов безопасности

CREATE\_NEW, // создание нового временного файла

FILE\_FLAG\_DELETE\_ON\_CLOSE, // удаляем файл после отмены отображения представления

NULL);

* FileName – имя файла
* DesiredAcess – тип доступа к файлу (GENERIC\_WRITE – запись, GENERIC\_READ - чтение)
* SharedMode – Режим совместного использования (если стоит 0, то пока файл открыт, то его не может открыть другой процесс)
* SecurityAttributes – атрибуты защиты
* CreationDesposition – Вид операции (создание или открытие, или открытие и очистка)
* FlagsAndAtributes – флаги и атрибуты
* TemplateFile – файл шаблона, который предоставляет атрибуты и расширенные атрибуты для файла, который будет создан

Шаг 2: Создайте map объект для файла, который содержит информацию о том, как получить доступ к файлу и его размеру. После создания вышеуказанного файла мы используем его дескриптор и создаем его отображение в физической памяти.

HANDLE CreateFileMappingA (

HANDLE hFile,

LPSECURITY\_ATTRIBUTES lpFileMappingAttributes,

DWORD flProtect,

DWORD dwMaximumSizeHigh,

DWORD dwMaximumSizeLow,

LPCSTR lpName);

* hFile: Дескриптор открытого файла, который должен быть отображен в память.
* lpAttributes: Указатель на структуру SECURITY\_ATTRIBUTES, которая определяет, может ли объект отображения файла быть наследован процессом-потомком.
* flProtect: Режим защиты памяти для отображения файла.
* dwMaximumSizeHigh: Старшее слово максимального размера файла, который должен быть отображен в память.
* dwMaximumSizeLow: Младшее слово максимального размера файла, который должен быть отображен в память.
* lpName: Указатель на строку, которая задает имя объекта отображения файла. Если NULL, система создаст неименованный объект отображения файла.

слово" - это единица данных, которую компьютер может обрабатывать за один раз

Страница" - это фрагмент виртуальной памяти, который используется для хранения данных и кода программ.

Шаг 3. Отображение всего или части объекта отображения файлов из физической памяти в виртуальное адресное пространство вашего процесса.

LPVOID MapViewOfFile ( // получить адрес

HANDLE hFileMappingObject,

DWORD dwDesiredAccess,

DWORD dwFileOffsetHigh,

DWORD dwFileOffsetLow,

SIZE\_T dwNumberOfBytesToMap

);

Приложения могут переопределить поведение, вызвав функцию FlushViewOfFile, чтобы заставить систему немедленно выполнять дисковые транзакции для сопоставления файлов в фактической записи и представления.

Шаг 4. Очистка

4 (A) Отмените отображение объекта отображения файла из адресного пространства процесса. Вернитесь к предыдущим шагам и сначала удалите представления файлов из адресного пространства процесса.

BOOL UnmapViewOfFile (LPCVOID lpBaseAddress);

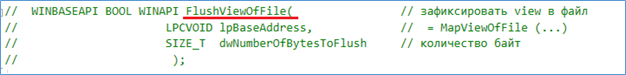
4 (B) Закройте объект сопоставления файлов. Этот шаг удаляет отображение файла из физической памяти.

CloseHandle (hFileMapping);

4 (C) Закройте файловый объект. Здесь закройте файл, открытый на диске, и освободите память. Поскольку на первом шаге мы устанавливаем флаг FILE\_FLAG\_DELETE\_ON\_CLOSE , файл будет удален после этого шага.

CloseHandle (hFile);

Также есть функция, которая “сбрасывает” виртуальную память обратно в файл.



использование образа файла, как средства межпроцессного взаимодействия

Для того, чтобы использовать представление файла в разных процессах, необходимо при создании Маппинга указать его имя, а в другом процессе вызывать функцию OpenFileMapping (с указанием имени Маппинга при создании).

HANDLE OpenFileMapping(

DWORD *dwDesiredAccess*, // режим доступа

BOOL *bInheritHandle*, // флажок наследования

LPCTSTR *lpName* // имя объекта

);

Пример:

Есть один процесс, который создает Маппинг через CreateFileMappingA с заданным именем (указывается заданное имя в последнем параметре CreateFileMappingA).

А второй процесс, который открывает существующий Маппинг с помощью OpenFileMapping с заданным именем (указывается заданное такое же имя в последнем параметре OpenFileMapping).

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

### Динамически вызываемые библиотеки

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.

#### Основные различия между DLL и статической библиотекой:

Механизм связывания: DLL является динамически подключаемой библиотекой, которая загружается в память приложения при его запуске. Статическая библиотека, напротив, является статически подключаемой библиотекой, которая компилируется в исполняемый файл во время линковки.

Общая память: DLL может использоваться несколькими приложениями, в то время как статическая библиотека связана только с одним приложением.

Обновление и модификация: DLL можно обновлять и модифицировать без перекомпиляции приложения. Если изменить статическую библиотеку, то необходимо перекомпилировать приложение, использующее эту библиотеку.

Размер: Размер DLL может быть значительно меньше, чем размер статической библиотеки, так как DLL содержит только код и данные, которые используются в приложении, а не все функции и данные из библиотеки, как это происходит в статической библиотеке.

Производительность: Загрузка DLL может занять некоторое время, но одна и та же DLL может использоваться несколькими приложениями, что может повысить производительность, если многие приложения используют одни и те же функции. Статическая библиотека не требует дополнительной загрузки в память, но ее размер может значительно увеличить размер исполняемого файла и ухудшить время запуска приложения.

Какой тип библиотеки использовать, зависит от конкретных требований проекта. Если необходима возможность обновления и модификации библиотеки без перекомпиляции приложения, то лучше использовать DLL. Если размер исполняемого файла критичен, и нет необходимости использовать одну и ту же библиотеку в нескольких приложениях, то статическая библиотека может быть более подходящим вариантом.

### структура DLL-библиотеки

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

набор подпрограмм библиотеки;

явное указание, какие подпрограммы должны экспортироваться из библиотеки.

### экспорт функций

**Extern “C”** указывает на то, что функция расположена в другом месте и что использует соглашение о вызове языка С. Используется, чтобы функции можно было легко использовать из других языков программирования или из языка C. Это обеспечивает согласованность и совместимость интерфейса DLL. Также позволяет вызывать функцию по указанному имени, а не сгенерированному автоматически.

При компиляции кода на C++ исходные файлы обычно проходят процесс "name mangling" (именование функций), который изменяет имена функций для поддержки особенностей языка C++, таких как перегрузка функций и пространства имен. Это делает функции в DLL недоступными для вызова из кода на C, который ожидает использования соглашения о вызове и именовании функций, используемого в C.

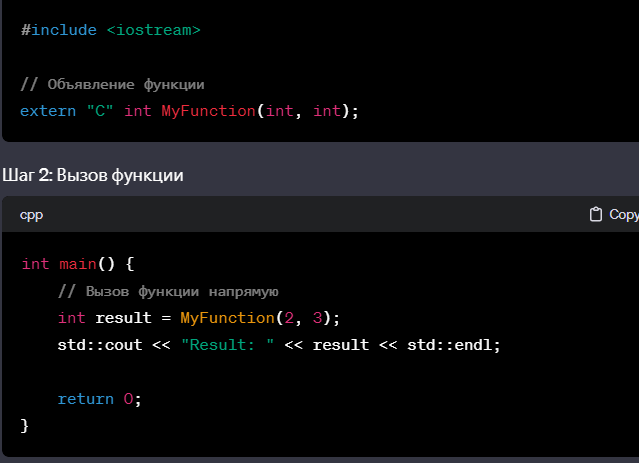
Ключевое слово **extern "C"** решает эту проблему, указывая компилятору C++ использовать соглашение о вызове и именовании функций, используемое в C. Это позволяет экспортированным функциям быть доступными из кода на C, а также из других языков, которые поддерживают соглашение о вызове C.

**\_\_declspec(dllexport)** - это ключевое слово для явного указания компилятору, что функция, класс или переменная должны быть экспортированы из исполняемого файла или DLL (динамической библиотеки) и стать доступными для использования другими модулями программы или внешними приложениями.

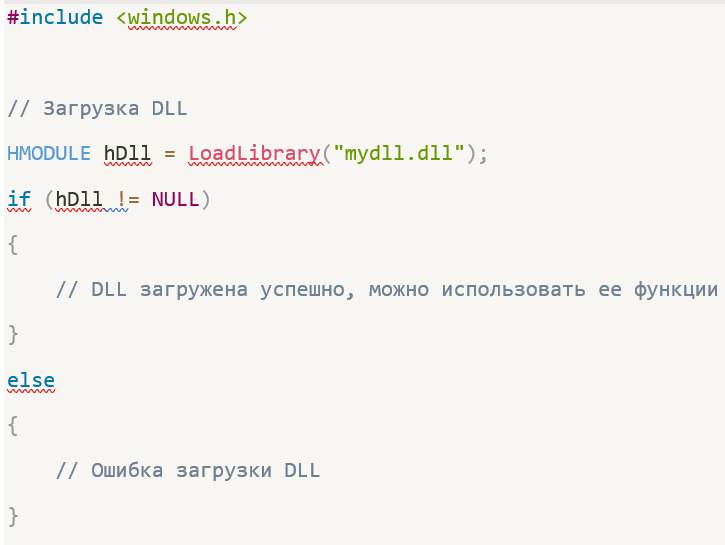
Когда используется **\_\_declspec(dllexport)**, компилятору сообщается, что определенный элемент программы (например, функция) должен быть видимым и доступным для других модулей, которые будут использовать ваш исполняемый файл или DLL. Это позволяет другим модулям программы или внешним приложениям вызывать и использовать экспортированные функции или обращаться к экспортированным переменным.

**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)
* **вная загрузка:**
  + Требуется явно загрузить библиотеку (**LoadLibrary**) и получить указатели на функции (**GetProcAddress**).
  + Позволяет более гибко управлять процессом загрузки и выгрузки библиотеки.
* **Неявная загрузка:**
  + Функции библиотеки вызываются как обычные функции.
  + Библиотека загружается автоматически при запуске программы.

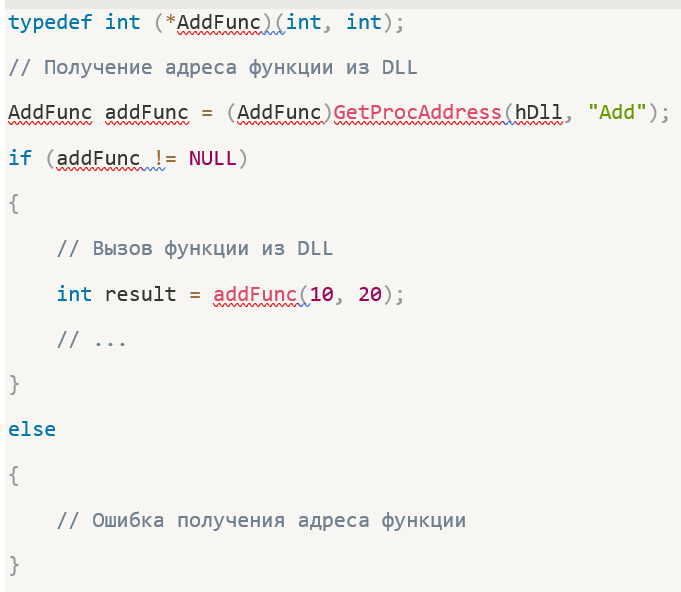
****

**LoadLibrary:**



**Динамический вызов функций динамической библиотеки:**

После загрузки DLL можно вызывать экспортированные функции, используя динамический вызов. Для этого в Windows используется функция GetProcAddress.



**Создание и применение библиотеки импорта:**

**Библиотека импорта (LIB)** - это файл, содержащий информацию о функциях и символах, которые можно использовать из DLL. Библиотека импорта упрощает динамическую загрузку DLL, обеспечивая статическую привязку к функциям **во время компиляции**. Она позволяет использовать более привычный синтаксис вызова функций.

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.

COM – спецификация, позволяющая разрабатывать программное обеспечения. Модель программного обеспечения. Разработана Microsoft. Спец инструкция по разработке кода, чтобы весь код писался одинаково. Стандарт предназначен для создания программного обеспечения на основе взаимодействующих компонентов.

COM-программирование – разработка COM-компонентов (объектов), программного обеспечения, имеющего модель COM.

COM-объект – специализированный объект времени исполнения (экземпляр).

Реализация одного экземпляра – сервер однокомпонентный, несколько типов объектов – многокомпонентный.

основные понятием СОМ

* является COM-компонент, представляющий собой программный модуль. Каждый компонент имеет свой уникальный 128-битный идентификатор в формате GUID (Global Unique Identifier).

создается и размещается в контейнере:

* + DLL-модуле
  + ; 2) EXE-модуле.
* COM-интерфейс. Интерфейс представляет собой набор абстрактных функций, имеющий аналогично COM-компонентам свой GUID-идентификатор. двух типов: стандартные и произвольные.
  + стандартными интерфейсами закреплены предопределенные GUID-идентификаторы(IUnknown).
  + Все остальные -производными (наследуют все методы) от IUnknown. Каждый компонент должен «реализовывать» как минимум стандартный интерфейс IUnknown.

Для каждого компонента необходимо подсчитывать сколько ссылок сделано на его интерфейс. Это нужно для того, чтобы знать, сколько клиентов подключены к нашей DLL. Если счётчик компонента == 0, он сам себя убивает.

COM-клиент создаёт объект класса. При вызове функции создания указывает идентификатор сервера и использует для этого OLE32.DLL. Необходимо знать CLSID сервера и в какой оболочке он находится.

COM-сервер: программный модуль, реализующий COM-объект.

В зависимости от типа контейнера и места его расположения:

* **CLSCTX\_INPROC\_SERVER** (DLL внутрипроцессный сервер);
* **CLSCTX\_LOCAL\_SERVER** (EXE-сервер за границами процесса, но та том же компьютере);
* **СLSCTX\_REMOTE\_SERVER** (EXE-сервер на удаленном компьютере).

Понятие позднего связывания программных модулей:

Раннее связывание (early binding):

В случае раннего связывания, связывание данных с кодом происходит на этапе компиляции программы, то есть на этапе, когда программа преобразуется из исходного кода в машинный код.

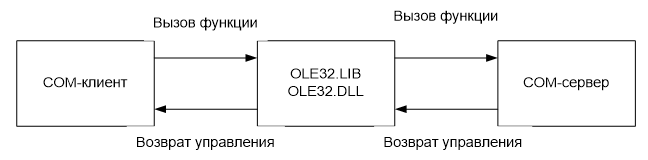
Позднее связывание (late binding):

В случае позднего связывания, связывание данных с кодом происходит в процессе выполнения программы, а не на этапе компиляции.

При создании объекта посредником между COM-клиентом и COM-сервером выступает библиотека *OLE32.DLL* (библиотека импорта *OLE32.LIB,* библиотека импорта нужная для того, чтобы на этапе компоновки можно было работать с какой-то библиотекой посредством статических вызовов).

С помощью экспортируемых функций библиотеки клиент передает информацию об идентификаторе компонента, типе сервера (Server type), идентификаторах интерфейсов (Interface ID). Именно OLE32.DLL по идентификатору CLSID через реестр операционной системы определяет место расположения контейнера компонента, загружает и инициализирует его.

Все функции OLE32.DLL возвращают значение типа HRESULT. Он используется для идентификации результата выполнения.



HRESULT — это тип возвращаемого значения COM-функции / метода

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

Структура COM-клиента:

COM-клиент создаёт объект класса. При вызове функции создания указывает идентификатор сервера и использует для этого OLE32.DLL. Необходимо знать CLSID сервера и в какой оболочке он находится.

**COM-клиент**: программный модуль, создающий COM-объект и использующий его методы.

**COM-клиент**: в качестве COM-клиента может выступать COM-сервер.

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



Интерфейсы используются для доступа к методам COM-объектов.

1. **COM-объект** можетиметьодин или несколько интерфейсов.
2. **К**аждый интерфейс включает один или несколько методов.
3. **К**аждый интерфейс имеет идентификатор, который имеет тип **GUID**.
4. **К**аждый COM-объект обязан поддерживать стандартный **интерфейс** **IUnknown** (у негостандарт. id**)**

В спецификации COM есть несколько стандартных интерфейсов, которые заранее прописаны в документации (IUnknown, IClassFactory) и которые имеют известные для всех id и методы.

IUnknown является базовым для любого другого интерфейса (интерфейс всех интерфейсов).

Методы IUnkown:

* QueryInterface (получает id в ответ отправляет ссылку; (ссылку) интерфейсы по IID)
* AddRef (внутри компонента поддерживается счётчик, этот метод увеличивает на 1 счётчик ссылок на интерфейс, если счётчик равен 0 – никто не использует компонент)
* Release (уменьшает счётчик ссылок на интерфейс на 1)

Внутри компонента есть специальная компонента, которая называется фабрика класса CF.

Задача – создавать объекты этого класса (в данном случае CA). Для каждого компонента своя фабрика. Фабрика классов реализует интерфейс IClassFactory. Интерфейсы поставляются из библиотеки OLE32 (для CA – IUnknown, CF – IClassFactory).

Методы интерфейса IClassFactory:

– CreateInstance(возвращает экземляр пользовательского компонента)

– LockServer(запрещает разрушение экземпляра фабрики классов

CreateInstance создаёт экземпляры рабочих компонентов (создаёт инстанс CA). Для каждого сервера фабрика классов создаёт компонент. После создания инстанса, возвращает клиенту указатель на интерфейс IUnknown.

LockServer мб необходим, если необходимо эксклюзивное использование сервера (1 – сервер заблокирован, 0 – разблокирован).

Сервер мб заблокирован с помощью OLE32 либо сами можем заблокировать.

Задача ClassFactory – создавать экземпляры компонентов для серверов.

IClassFactory создаёт экземпляр объекта IUnknown, чтобы клиент мог получить к нему доступ.

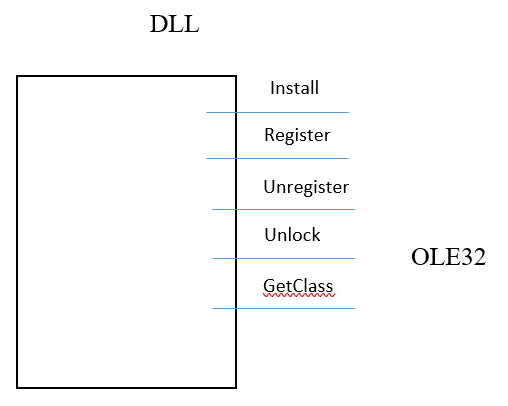
Сервер представляет из себя DLL, в которую экспортирует 5 функций, 3 используются для работы с ОС, 1 – для создания компонента.

DLL находятся в реестре (можно найти в реестре)

DllRegisterServer – регистрация DLL

DllUnregisterServer – удаляет DLL из реестра ОС

Для регистрации DLL необходимо знать 4 параметра (CLSID, friendly name, версию indProgID, progID) и hModule (по нему можно получить место расположения DLL).



Экспортировать функцию из DLL можно нескольким способами:

– def-файл, в котором указаны функции, которые нужно экспортировать.

– модификатор extern

Между клиентом и сервером стоит библ. OLE32. Целью сервера является создание COM-объекта CA.

DLL – изолированный программный модуль, который можно загрузить и извлечь указатели на функции, которые эта библиотека экспортирует.

Экспортируемые стандартные функции:

Внутренний СОМ-сервер должен экспортировать четыре функции:

function **DllRegisterServer**: HResult; stdcall; (применяется для регистрации DLL СОМ-сервера в системном реестре Windows. При регистрации СОМ-класса в системном реестре создается раздел в HKEY\_CLASSES\_ROOT\CLSID\{XXXXXXXX-XXXX-XXXX-xxxx-xxxxxxxx}, где число, записанное вместо символов х, представляет собой CLSID данного СОМ-класса)

function **DllUnregisterServer**: HResult; stdcall; (применяется для удаления всех разделов, подразделов и параметров, которые были созданы в системном реестре функцией DllRegisterServer)

function **DllGetClassObject** (const CLSID, IID: TGUID; var Obj): HResult; (возвращает фабрику класса для конкретного СОМ-класса.)

function **DllCanUnloadNow**: HResult; stdcall; (применяется для определения, можно ли в настоящий момент времени выгрузить DLL СОМ-сервера из памяти. Возвращает значение FALSE, т. е. DLL выгрузить нельзя. Если ни один СОМ-объект данной DLL не используется, то функция возвращает значение TRUE.)

**DllInstall** используется только для установки и настройки приложений. Это не должно вызываться приложением

Регистрация COM/DLL-сервера **:**

Утилита **regsvr32.exe** является стандартной программой командной строки для регистрации и отмены регистрации элементов управления OLE, ActiveX и библиотек DLL в реестре Windows.

Для работы утилиты требуются повышенные привилегии, поэтому окно командной строки должно вызываться с правами администратора.

regsvr32 -i:xxxx OS12\_COM.dll

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

[Порядок выполнения **верификации** обычно следующий](https://skillbox.ru/media/code/identifikatsiya-autentifikatsiya-avtorizatsiya-chem-oni-razlichayutsya/):

Идентификация систем



1. **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 пользователя1](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/).





1. [**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)

Дискреционная система безопасности – модель разграничения доступа, которая 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 субъектов;
* в компьютерной системе существует привилегированный пользователь, обладающий правом полного доступа к любому объекту.

Последнее свойство определяет невозможность существования в компьютерной системе потенциально недоступных объектов, владелец которых отсутствует. Но реализация права полного доступа к любому объекту посредством предварительного назначения себя его владельцем не позволяет привилегированному пользователю (администратору) использовать свои полномочия незаметно для реального владельца 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.

возможности API управления пользователями и группами

**1. Имя текущего пользователя**

BOOL GetUserName

(

LPTSTR lpBuffer, // указатель на буфер

LPDWORD nSize // получает размер возвращает количество символов

);

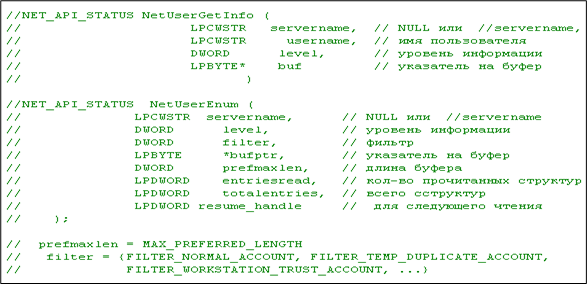
**2. Информация о пользователей**

****

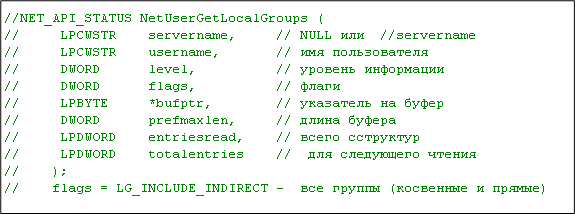
**3. Освободить память**

****

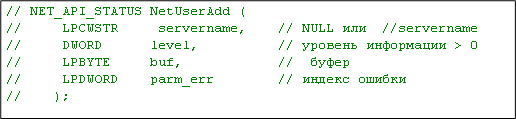
**4. Получить коллекцию пользователей**

****

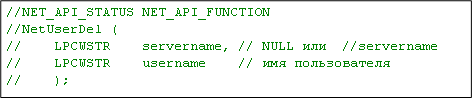
**5. Группы пользователя**

****

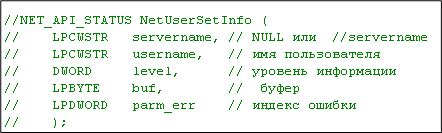
**6. Добавить пользователя**

****

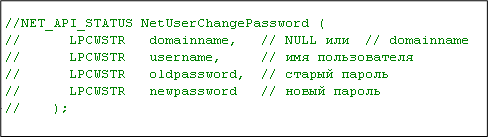
**7. Удалить пользователя**

****

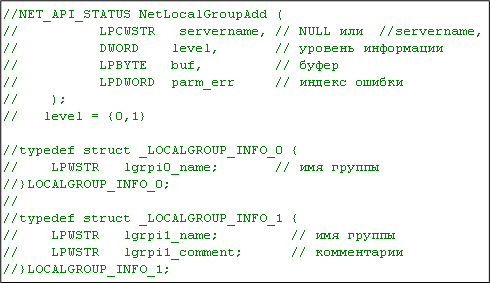
**8. Изменить информацию о пользователе**

****

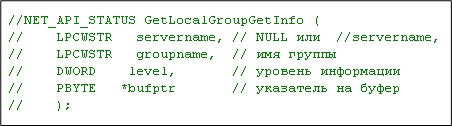
**9. Изменить пароль**

****

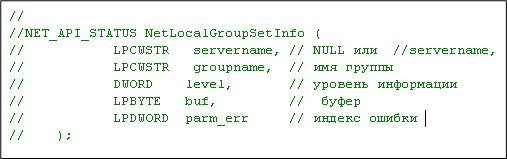
**10. Добавление группы**

****

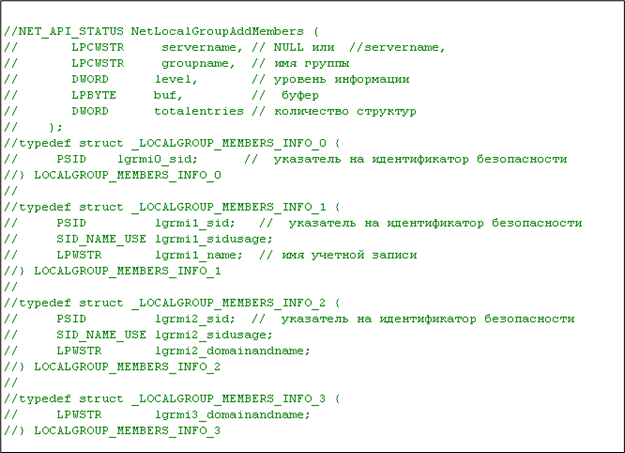
**11. Получить информацию о группе**

****

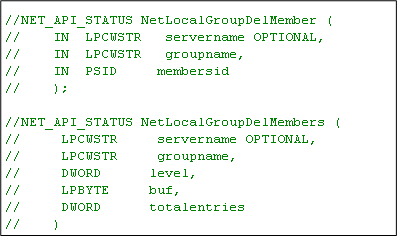
**12. Изменить информацию о группе**

****

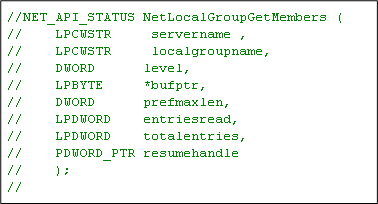
**13. Добавление членов локальной группы**

****

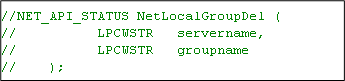
**14. Удаление членов группы**

****

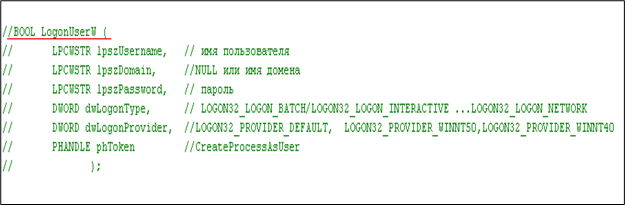
**15. Получить коллекцию членов группы**

****

**16. Удалить группу**

****

**17. Подключение пользователя**

****

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.

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.

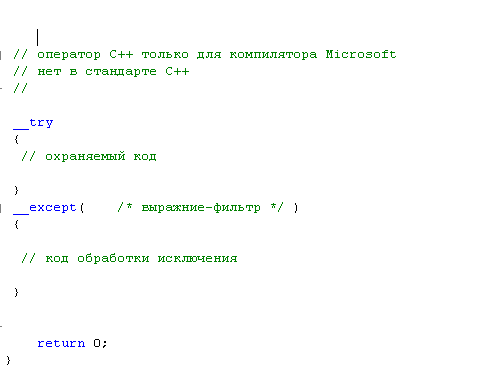
SEH: Обработчик исключения – блок программного кода, который обрабатывает исключение.

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.

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:

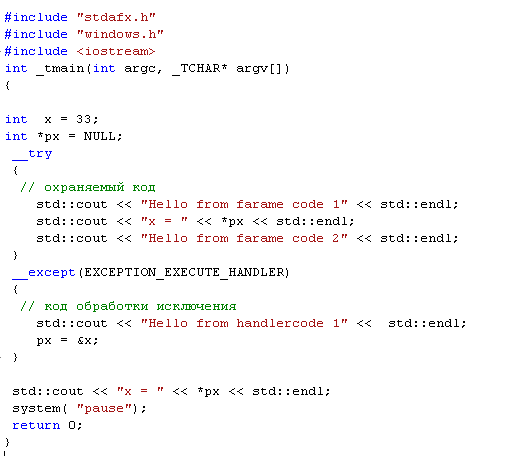
1. Сначала выполняется защищенный раздел \_\_try.
2. Если исключение при этом не возникает, выполнение переходит в инструкцию, стоящую после предложения \_\_except.
3. Если во время выполнения защищенного раздела возникает исключение или в любой подпрограмме вызывается защищенный раздел, \_\_except выражение 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.

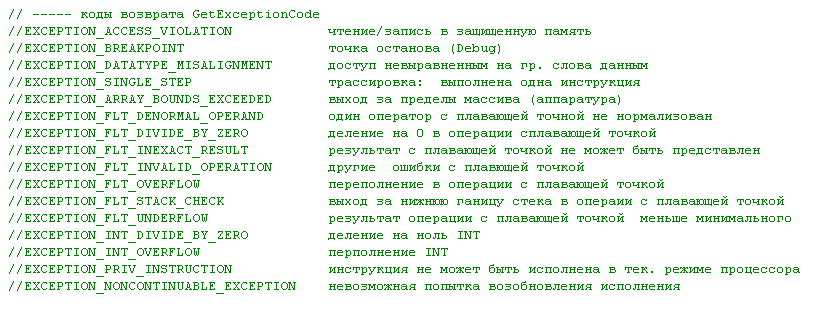
в выражении фильтра можно использовать две функции: GetExeptionCode, GetExceptionInformation.

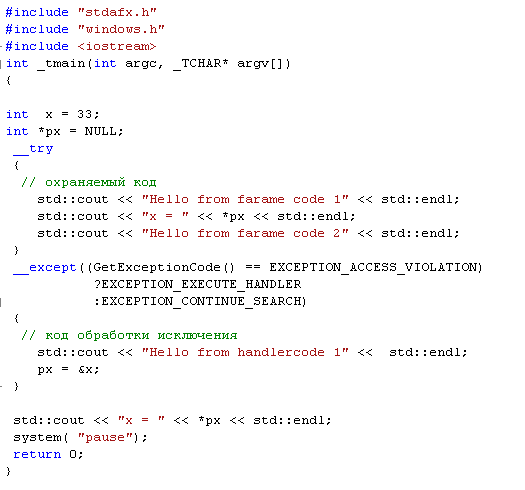
переменные, объявленные внутри {} – локальные

****

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

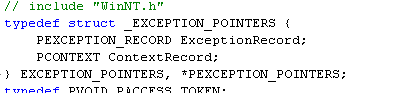
DWORD GetExceptionCode() – может использоваться только в выражении-фильтре (для определения дальнейших действий: обрабатывать, искать обработчик, вернуть управление в точку прерывания) или в блоке обработки исключения (для получения кода исключения).

****

****

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

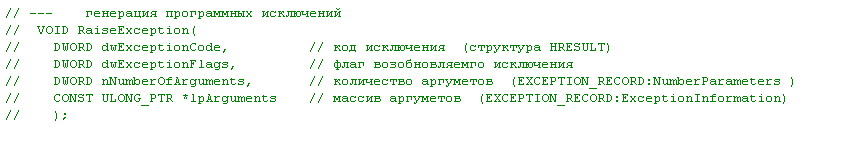
ExceptionFlag = EXEPTION\_NONCONTINUABLE – невозможно

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

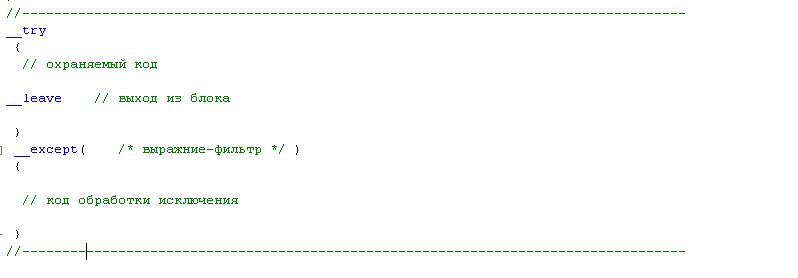
NumberParameters – количество параметров в ExcetptionInformation

ExcetptionInformation – массив 32-битных описателей исключения

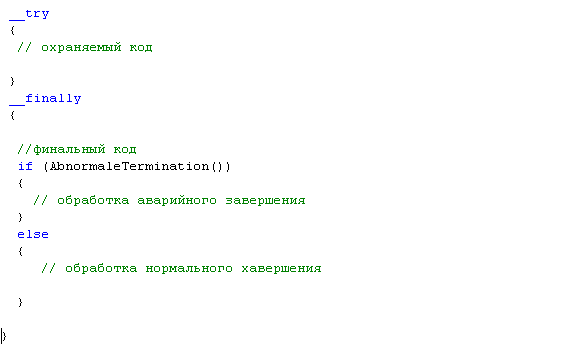
SEH: Microsoft C++: генерация программных исключений



SEH: Microsoft C++: передача управления

****

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

****

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

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

**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 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.

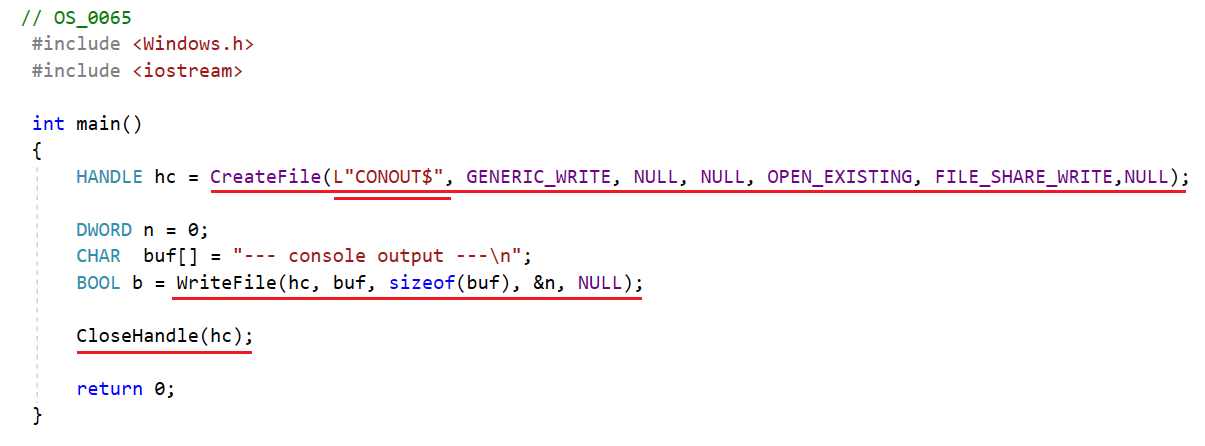
Консоль — это приложение, которое предоставляет службы ввода-вывода для приложений в символьном режиме.

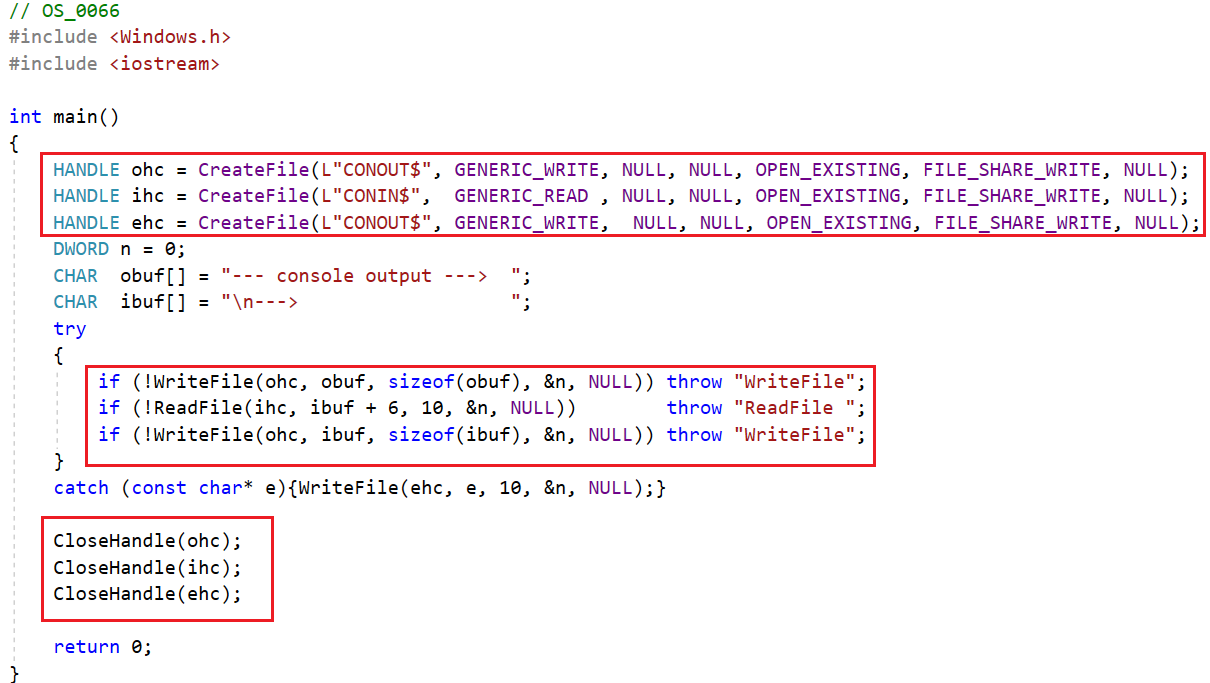
Консоль состоит из входного буфера и одного или нескольких буферов экрана.

* 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.

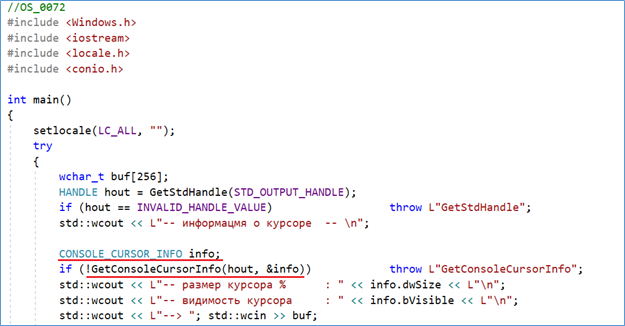
Стандартные потоки:

****

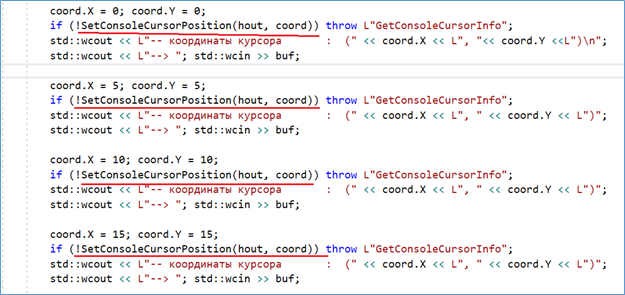
****

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 GetCurcorInfo



**Console: позиции курсора**

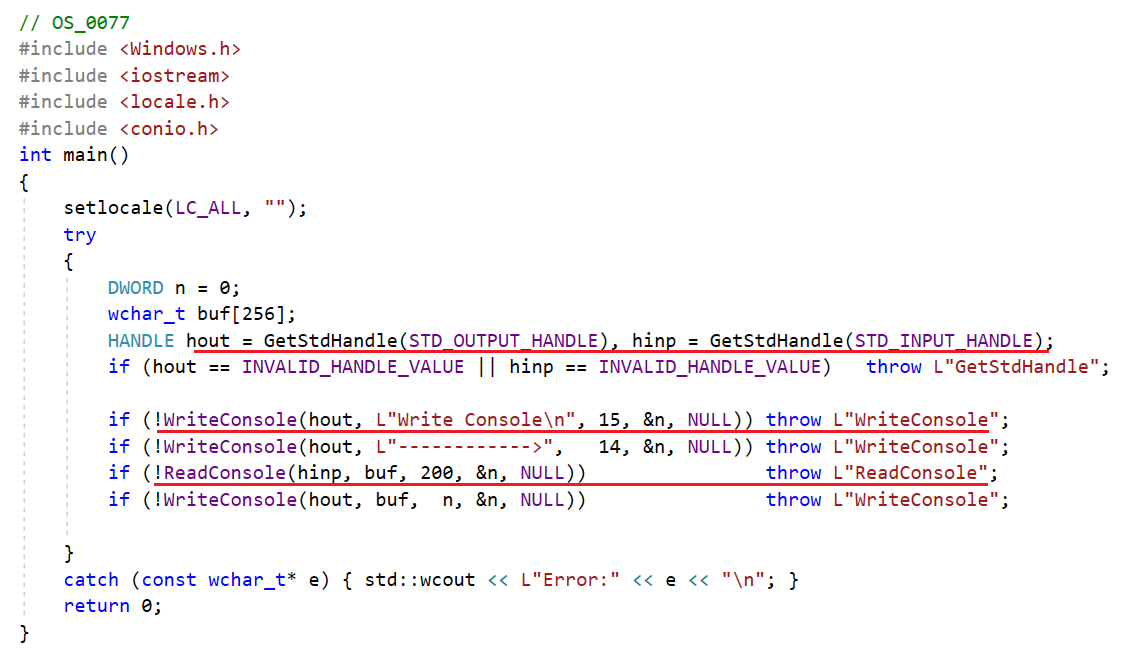


**Console: установка атрибутов консоли**

Ук34

**Высокоуровневый ввод/вывод WriteConsole, ReadConsole**

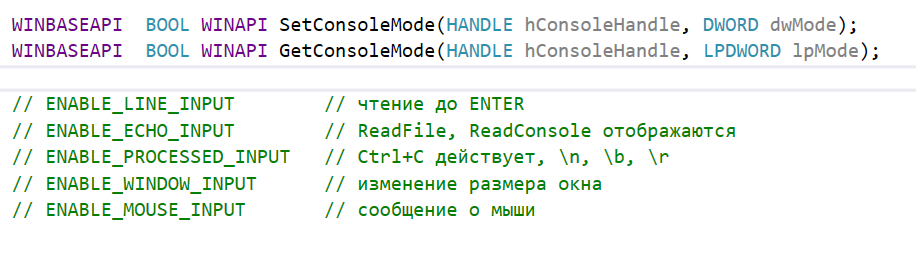
ReadFile, WriteFile – аналогичны WriteConsole, ReadConsole



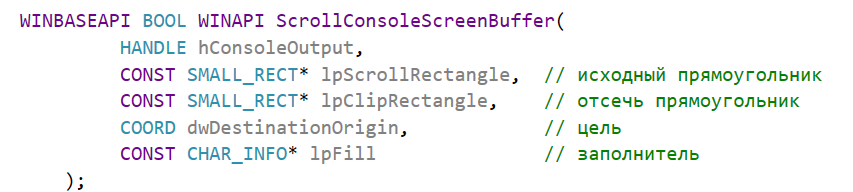
**Console: ввод низкого уровня ReadConsoleInput**



**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.

1. Windows-сервисы: определение, назначение, применение, API.

Windows-сервис – это процесс операционной системы, выполняющий служебные функции. Как правило, сервис стартует при загрузке операционной системы и заканчивает свою работу при завершении работы операционной системы. Часто сервисы называют службами.

Обычно сервисы выполняют служебные функции, необходимые для какого-нибудь приложения. Иногда сервисы используются для доступа к внешним устройствам, они называются драйверами.

В виде сервисов реализованы сетевые службы, такие как DNS, FTP, Telnet, DHCP и другие. Сам сервис может быть как консольным приложением, так и приложением с графическим интерфейсом.

**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.

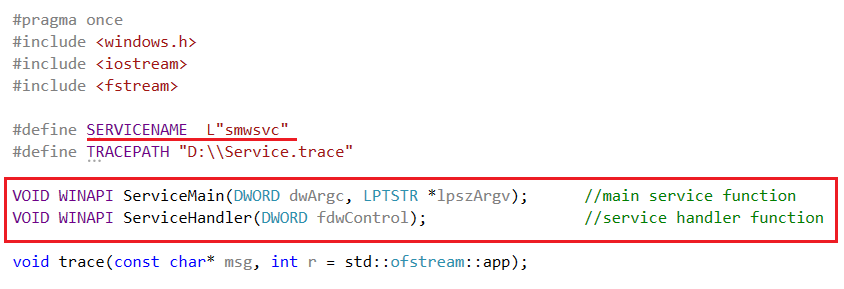
**Список служб находится** в реестре Windows «HKEY\_local\_machine\system\CurrentControlSet\Services».

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

Для просмотра действующих на локальном компьютере Windows сервисов может быть использована mmc-оснастка «**SCM**

**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. 

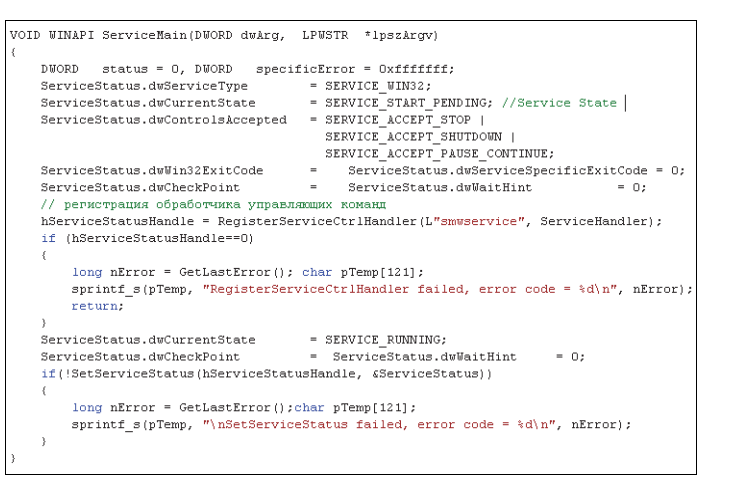
Обычно windows-сервисы разрабатываются как консольное приложение. Приложение, как правило содержит несколько функций предназначенных для:

установки (регистрации) сервиса,

запуска и

остановки сервиса.

Обычно точка входа сервиса имеет имя ServiceMain назначение– определить текущее состояние сервиса (структура ServiceStatus и функция SetServiceStatus), а также зарегистрировать обработчик управляющих команд (функция SetServiceStatus).



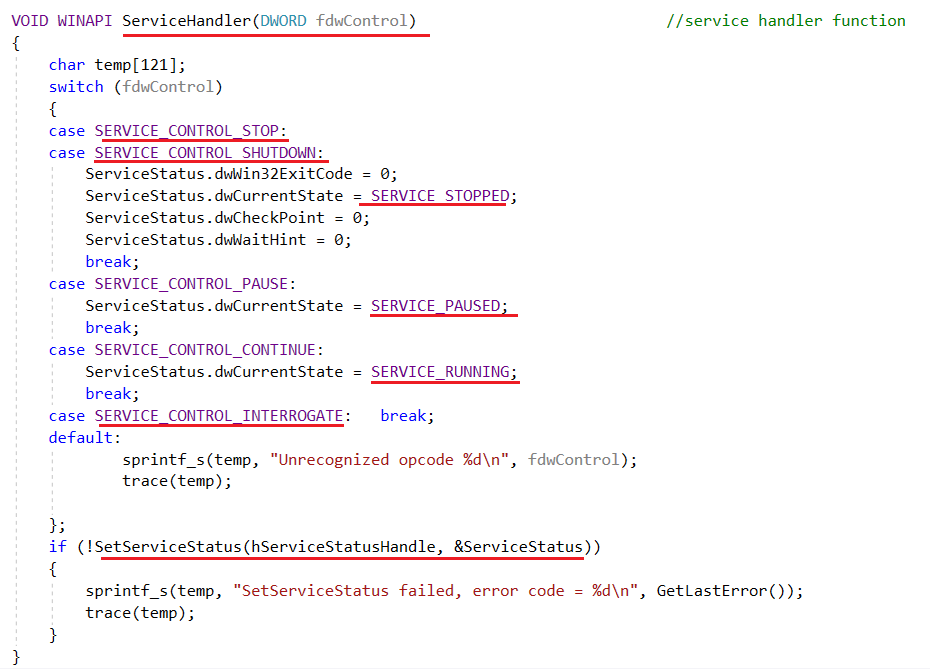
Обработчик управляющих команд – это функция ServiceHandler предназначенная для обработки команд, поступающих сервису, например, через mmc-консоль. Именно эта функция определяет перечень команд, на которые реагирует сервис. В зависимости от команды устанавливается новый статус сервиса.

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**



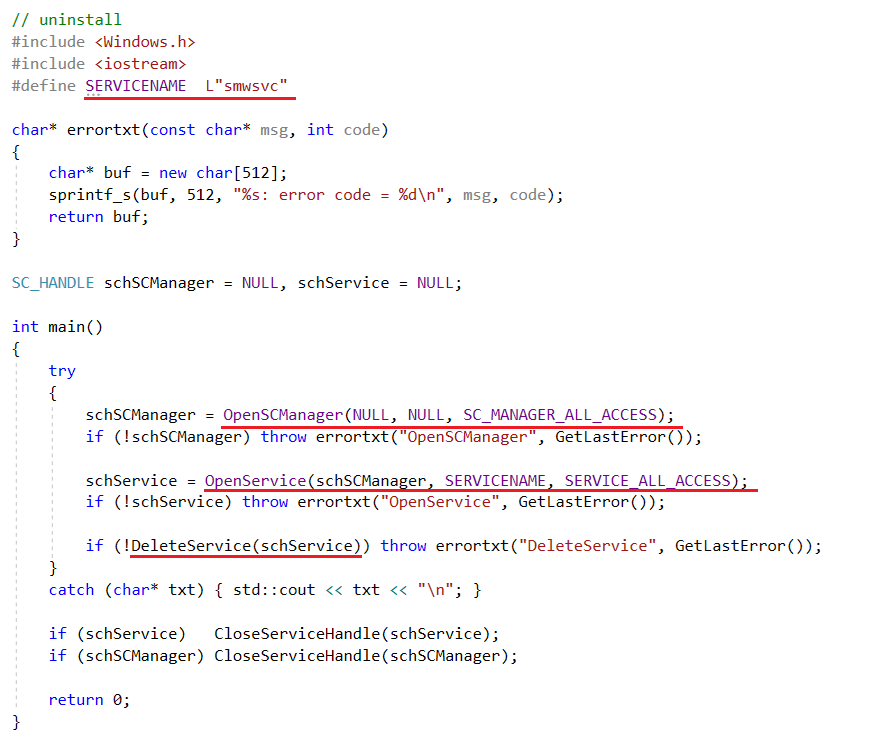
**создание сервиса, регистрация в реестре Windows.**



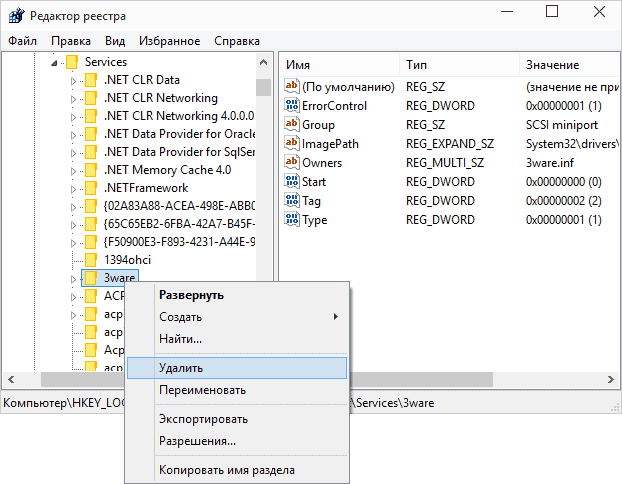


**удаление сервиса.**

**Кодом:**

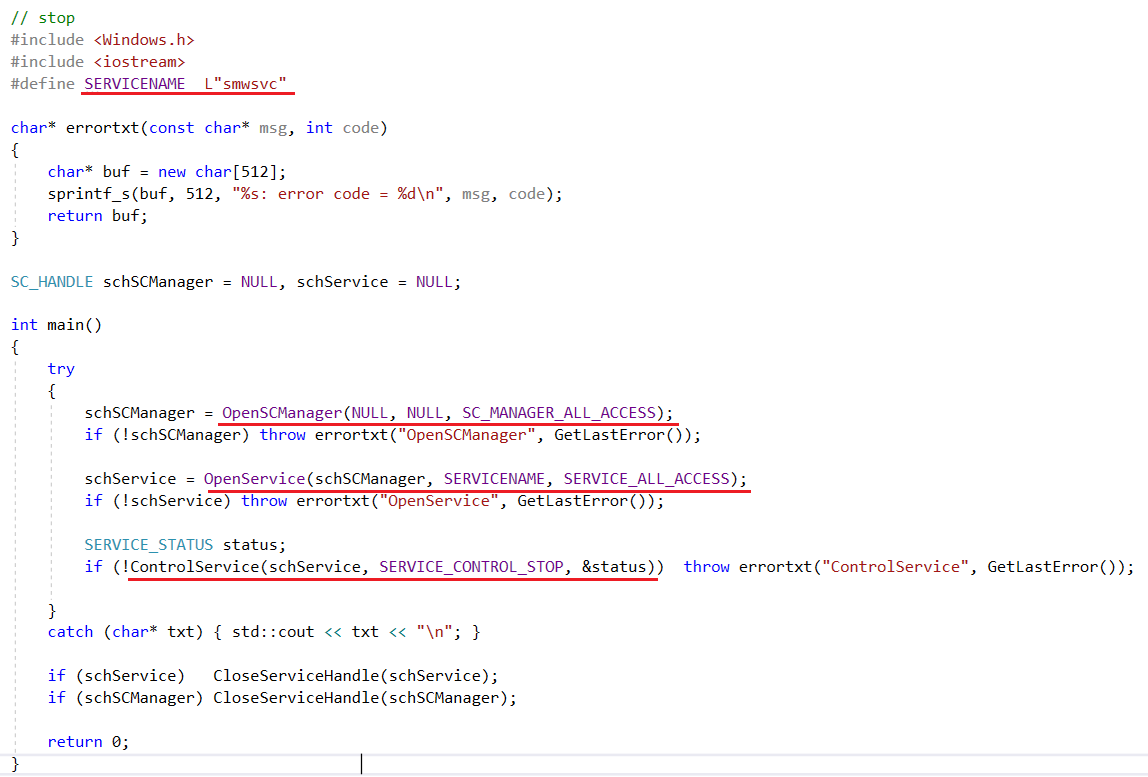


**Удалить службу Windows можно также и с помощью редактора реестра**

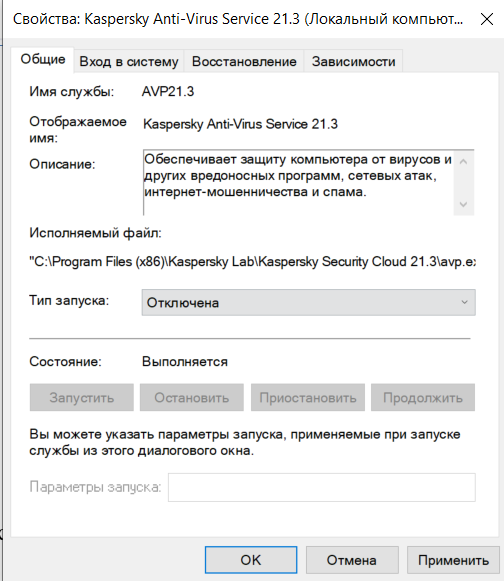


**Windows Services:** остановка сервиса.

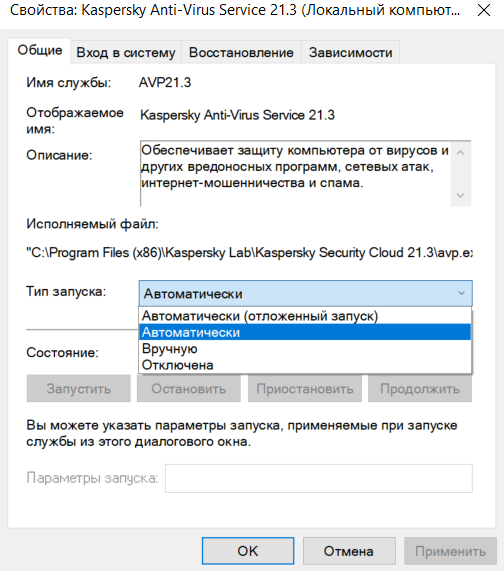
**Кодом:**



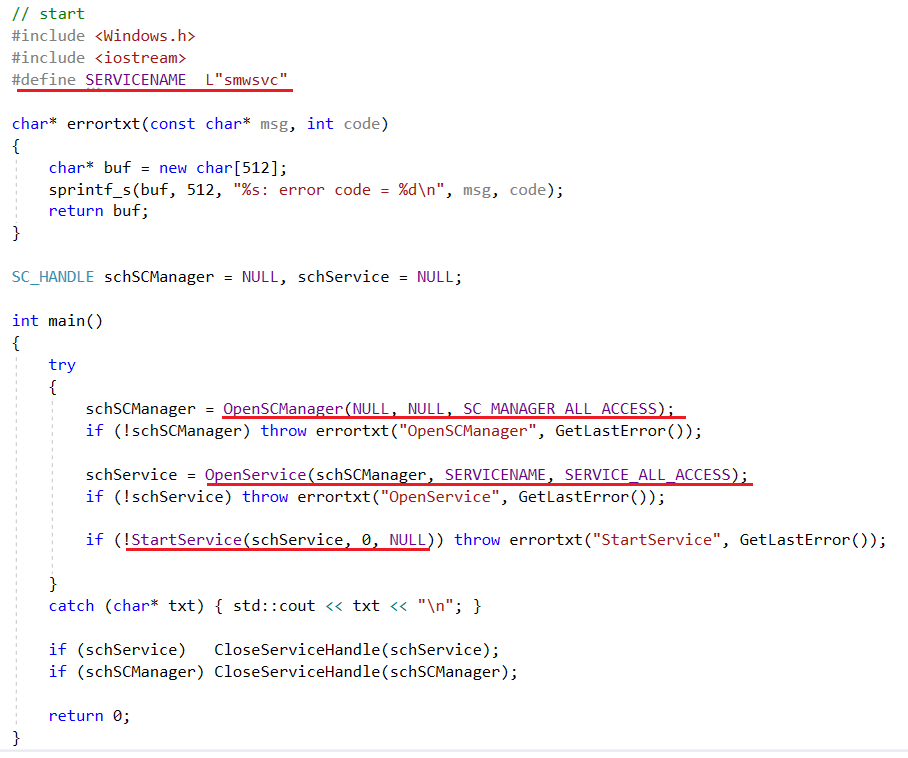
В службах:



**Windows Services:** старт сервиса.



Кодом:



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

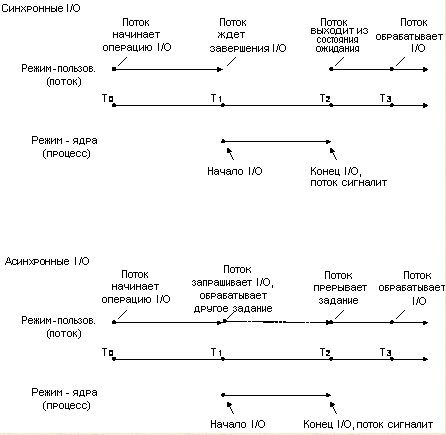
Позволяют совместить долгие операции ввода-вывода с обработкой компьютера.

Имеется два типа синхронизации ввода-вывода (I/O) файлов:

* синхронный ввод-вывод файла
* асинхронный ввод-вывод файла.(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.

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

Способ 1.( при единственной операции ввода/вывода)

1.при открытии файла (т.е при вызове функции CreateFile) нужно передать в параметр dwFlagsAndAttributes значение FILE\_FLAG\_OVERLAPPED.

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 WriteFile/ReadFile: в параметр lpOverlapped нужно передать указатель на структру 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

Реализация:

// Шаг 2: Определение структуры \_OVERLAPPED

OVERLAPPED g\_Overlapped = {0};

// Шаг 4: Обработка завершения асинхронной операции

void CALLBACK FileIOCompletionRoutine(DWORD dwErrorCode, DWORD dwNumberOfBytesTransfered, LPOVERLAPPED lpOverlapped)

{

if (dwErrorCode == 0)

{

\_tprintf(\_T("Асинхронная операция завершена успешно. Байтов передано: %d\n"), dwNumberOfBytesTransfered);

}

else

{

\_tprintf(\_T("Асинхронная операция завершена с ошибкой. Код ошибки: %d\n"), dwErrorCode);

}

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

// Шаг 1: Открытие файла с использованием FILE\_FLAG\_OVERLAPPED

HANDLE hFile = CreateFile(

\_T("example.txt"), // Имя файла

GENERIC\_READ, // Запрос на чтение

0, // Разделитель доступа

NULL, // Атрибуты безопасности (по умолчанию)

OPEN\_EXISTING, // Открыть существующий файл

FILE\_FLAG\_OVERLAPPED, // Использовать асинхронные операции

NULL // Дескриптор файла (null, так как мы открываем новый файл)

);

// Шаг 3: Вызов функции ReadFile с использованием структуры \_OVERLAPPED

DWORD dwBytesRead;

TCHAR buffer[100];

// Устанавливаем позицию чтения на начало файла

SetFilePointer(hFile, 0, NULL, FILE\_BEGIN);

// Устанавливаем структуру \_OVERLAPPED

g\_Overlapped.Offset = 0; // // Младшие 32 бита смещения

g\_Overlapped.OffsetHigh = 0; // // Старшие 32 бита смещения

if (ReadFile(

hFile, // Дескриптор файла

buffer, // Буфер для чтения

sizeof(buffer), // Размер буфера

&dwBytesRead, // Количество прочитанных байт

&g\_Overlapped)) // Указатель на структуру \_OVERLAPPED

{

\_tprintf(\_T("Чтение завершено синхронно (не ожидается)\n"));

}

else

{

// Шаг 4: Проверка на ERROR\_IO\_PENDING

if (GetLastError() == ERROR\_IO\_PENDING)

{

\_tprintf(\_T("Чтение запущено асинхронно (ожидание завершения)\n"));

// Шаг 5: Ожидание завершения асинхронной операции

DWORD dwWaitResult = WaitForSingleObject(hFile, INFINITE);

if (dwWaitResult == WAIT\_OBJECT\_0)

{

\_tprintf(\_T("Операция завершена успешно\n"));

}

else

{

\_tprintf(\_T("Операция завершена с ошибкой (код %d)\n"), GetLastError());

}

}

else

{

\_tprintf(\_T("Ошибка при запуске асинхронного чтения (код %d)\n"), GetLastError());

}

}

// Шаг 6: Продвижение указателя файла

SetFilePointer(hFile, dwBytesRead, NULL, FILE\_CURRENT);

// Закрытие дескриптора файла

CloseHandle(hFile);

return 0;

}

Способ 2. (при нескольких операциях ввода/вывода)

1.создать (объявить) структуру \_OVERLAPPED;

2.создать событие Event с автоматическим сбросом;

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

4.WriteFile/Read: lpOverlapped;

5.GetLastError == ERROR\_IO\_PENDING:

6.WaitForSingleObject(hEvent, time) ;

7.продвинуть Offset, OffsetHigh.

#include <Windows.h>

#include <tchar.h>

// 1. Создание (объявление) структуры \_OVERLAPPED

OVERLAPPED g\_Overlapped = {0};

// 2. Создание события с автоматическим сбросом

HANDLE g\_hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

int \_tmain(int argc, \_TCHAR\* argv[])

{

// 3. Присвоение события \_OVERLAPPED

g\_Overlapped.hEvent = g\_hEvent;

// 4. Вызов функции WriteFile с использованием структуры \_OVERLAPPED

HANDLE hFile = CreateFile(

\_T("example.txt"), // Имя файла

GENERIC\_WRITE, // Запрос на запись

0, // Разделитель доступа

NULL, // Атрибуты безопасности (по умолчанию)

OPEN\_ALWAYS, // Открыть файл (или создать, если не существует)

FILE\_FLAG\_OVERLAPPED, // Использовать асинхронные операции

NULL // Дескриптор файла (null, так как мы создаем новый файл)

);

TCHAR buffer[] = \_T("Hello, Async IO!");

// Устанавливаем структуру \_OVERLAPPED

g\_Overlapped.Offset = 0;

g\_Overlapped.OffsetHigh = 0;

if (WriteFile(

hFile, // Дескриптор файла

buffer, // Буфер для записи

sizeof(buffer), // Размер буфера

NULL, // Количество записанных байт (не используется с асинхронными операциями)

&g\_Overlapped)) // Указатель на структуру \_OVERLAPPED

{

\_tprintf(\_T("Запись завершена синхронно (не ожидается)\n"));

}

else

{

// 5. Проверка на ERROR\_IO\_PENDING

if (GetLastError() == ERROR\_IO\_PENDING)

{

\_tprintf(\_T("Запись запущена асинхронно (ожидание завершения)\n"));

// 6. Ожидание завершения асинхронной операции

DWORD dwWaitResult = WaitForSingleObject(g\_hEvent, INFINITE);

if (dwWaitResult == WAIT\_OBJECT\_0)

{

\_tprintf(\_T("Операция завершена успешно\n"));

}

else

{

\_tprintf(\_T("Операция завершена с ошибкой (код %d)\n"), GetLastError());

}

}

else

{

\_tprintf(\_T("Ошибка при запуске асинхронной записи (код %d)\n"), GetLastError());

}

}

// 7. Продвижение указателя файла

SetFilePointer(hFile, sizeof(buffer), NULL, FILE\_CURRENT);

// Закрытие дескриптора файла и события

CloseHandle(hFile);

CloseHandle(g\_hEvent);

return 0;

}

Асинхронное блокирование файлов. Функции UnlockFile(только синхронные) и UnlockFileEx (: Позволяет выполнять как синхронные, так и асинхронные операции разблокировк)

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:

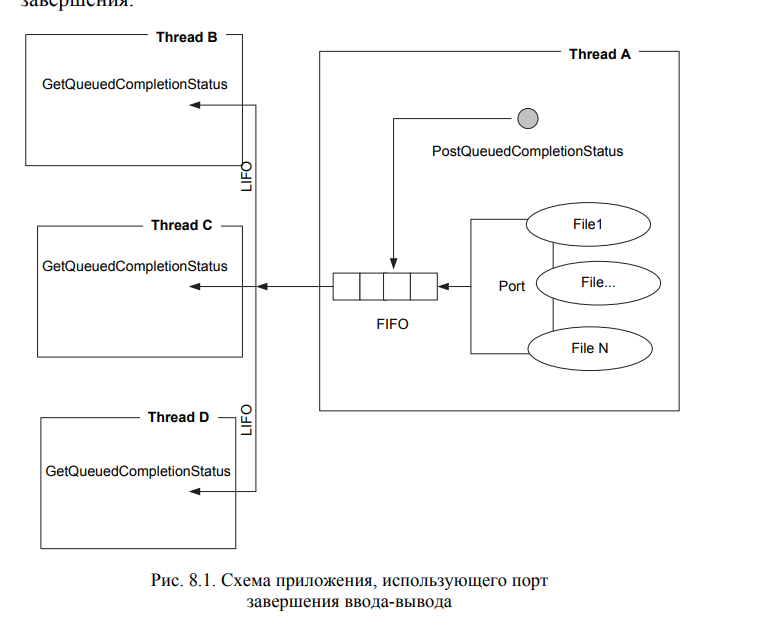
CreateIoCompletionPort - создать порт завершения

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. На рис. 8.1 изображена схема приложения, использующего порт завершения.



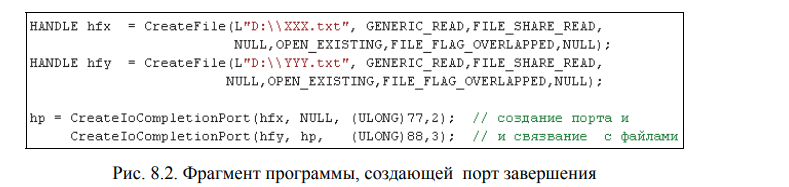
На рисунке изображены четыре потока (A, B. C, D) некоторого Windows-процесса. Поток A использует порт завершения, связанный с дескрипторами N файлов. Поток в определенном порядке выполняет операции асинхронного ввода или вывода для этих файлов.

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.

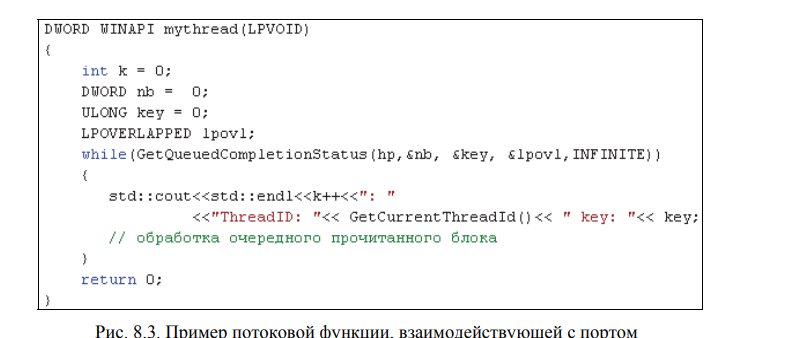
Таким образом, порты завершения ввода-вывода используются в тех случаях, когда формирование блока выводимых данных или обработка блока входных данных требует значительных затрат процессорного времени. Причем обработка каждого блока может выполняться независимо друг от друга. В этом случае достигается 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).

8.1.2. Принципы разработки приложениий с применением порта завершения На рис. 8.2 приведен фрагмент программы, демонстрирующий создание порта завершения ввода-вывода и связывания его с файламA



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.

На рис. 8.3 представлен пример потоковой функции, взаимодействующей с портом ввода-вывода. 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 записанны) байт



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.

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

предоставляет возможность упаковывать и запускать приложения в контейнерах. Docker использует стандартные контейнерные технологии, которые позволяют изолировать приложения и их зависимости от окружения. Клиент-серверная

[Уроки Docker для начинающих / #1 – Контейнеры и введение в Докер. Для чего он нужен? (youtube.com)](https://www.youtube.com/watch?v=dNS61T4MmlM&list=PL0lO_mIqDDFX1c0JHogP5YuZdOVawoepS)

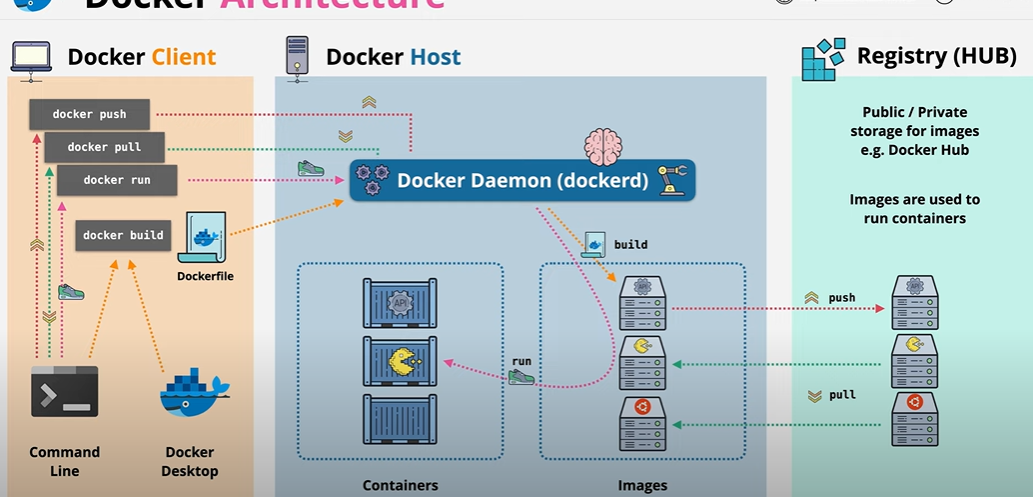
**Архитектура Docker** состоит из нескольких компонентов:

Docker daemon - отвечает за управление контейнерами и их запуск (rest api которое принимает запросы от клиента и передает из дальше).

Docker client - предоставляет интерфейс для взаимодействия с Docker daemon

Docker registry - 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.

**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 Docker** создаются из **образов**, которые содержат все необходимые приложения и зависимости.

**Контейнеры в Docker бывают двух типов**:

* системные контейнеры
* 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.

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

Реестр - в нем находятся различные репозитории.

**Образ Docker** - это набор инструкций для создания контейнера. Он может содержать код приложения, зависимости, настройки окружения и другие компоненты, необходимые для запуска приложения в контейнере. Образы Docker можно сравнить с шаблонами, которые используются для создания контейнеров. Образы могут быть созданы из других образов, что упрощает процесс разработки и обеспечивает консистентность окружения.

**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
* docker images - выводит список доступных образов
* docker pull - загружает образ из реестра
* docker push - отправляет образ в реестр

Кроме того, существуют дополнительные команды для работы с сетями, томами, переменными окружения и другими аспектами Docker платформы.