**Системное программирование**

## **Системное программирование: определение, назначение, применение.**

Когда говорят о системном программировании, в большинстве случаев подразумевают разработку 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.

Может показаться, что выше перечислено три класса различного программного обеспечения. Но при разработке общего, повторно используемого программного обеспечения, неминуемо приходится применять **системные вызовы**. ***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. Именно благодаря такому подходу компании Microsoft удалось обеспечить эффективное взаимодействие и совместимость всех своих прикладных продуктов. Программист, освоивший эту спецификацию, получает возможность писать программное обеспечение, которое может использовать открытые интерфейсы для программного взаимодействия с продуктами Microsoft, а также организовать взаимодействие между загрузочными модулям собственного программного обеспечения.

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

Системные программисты разрабатывают:

* Низкоуровневые lib-ы
* ОС (Windows, Linux, MacOS)
* Драйверы устройств (позволяет взаимодействовать ОС с аппаратным оборудованием компьютера)
* Компиляторы (для различных ЯП: C, Java, C#)
* Сетевое ПО (взаимодействие устройств в сети)
* СУБД (Oracle, MySQL, MS SQL Server…)
* Системы безопасности (антивирусы, брандмауэры)
* ПО 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)

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

**Фреймворк ОС** - ***набор библиотек (функций) и порядок работы с ним, которые являются посредниками между разрабатываемым ПО и ядром ОС.***

Другие определения:

1. Совокупность библиотек и среды выполнения, облегчающее разработку большого программного проекта.
2. Набор библиотек и правила работы с ними, предназначены для интерфейсов выполнения программ с ядром ОС.
3. [Фреймворк операционной системы предоставляет структуру и набор API (Application Programming Interfaces), которые разработчики могут использовать для создания, тестирования и развертывания приложений, которые могут работать в определенной операционной системе](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA).

[**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.

(**Минимальный интерфейс системных вызовов** в контексте POSIX означает набор базовых системных вызовов, которые должны быть реализованы в любой операционной системе, стремящейся к совместимости с POSIX. Это включает в себя функции для работы с файлами, процессами, памятью и другими основными аспектами операционной системы)

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

[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).

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

(под API я так полагаю подразумеваются функции WINAPI для работы с процессами???)

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

**Процесс** — это выполняемая в данный момент программа / абстрактное описание запущенной программы, контейнер с информацией о ней.

Свойства процесса:

* процессу соответствует исполняемый программный файл;
* у процесса есть PID;
* у процесса есть Parent PID;
* у процесса есть приоритет (от 0 до 31)
* состояние (running, stop, waiting…)
* атрибуты безопаности
* 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;

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

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

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

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

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

## **стандартные потоки ввода/вывода**

Потоки имеющие зарезервированные номера - дескрипторы (номера), поток ввода (0), поток вывода (1), поток вывода ошибок (2).

## **системные процесса**

процессы запускаемые автоматически при запуске OS; Windows: windows-сервисы; Linux-демоны. Системные процессы являются частью ядра и всегда расположены в оперативной памяти.

**Системные процессы** – процессы, запускаемые системой, выполняют системный код.

**Пользовательские процессы** – процессы, запускаемые пользователем, выполняют собственный код и иногда обращаются к системным функциям.

**Применение процессов:**

Применение процессов заключается в создании абстракции, которая позволяет осуществлять (псевдо – если одноядерный процессор) 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**

Смелов:

Функция **CreateProcess,** которая создает новый процесс с единственным потоком. При вызове этой функции требуется указать имя файла исполняемой программы. Функция возвращает два отдельных дескриптора, по одному для процесса и потока, передавая их в структуре, которая указывается при вызове функции.

Таненбаум:

Существуют четыре основных события, приводящих к созданию процессов.

1. Инициализация системы.
2. Выполнение работающим процессом системного вызова, предназначенного для создания процесса.
3. Запрос пользователя на создание нового процесса.
4. Инициация пакетного задания

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

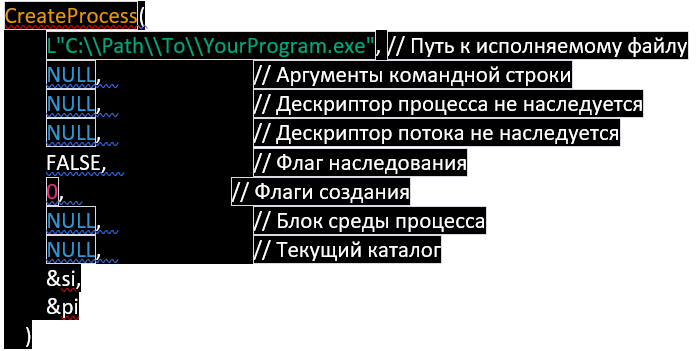
**HANDEL Windows-процесса**

Handle (рукоятка, привод) – 32-разрядное число, номер объекта ядра операционной системы.

## **Idle-процесс**

Idle process - процесс, выполняемый процессором в пространстве ядра операционной системы в случае, если нет других процессов, которые процессор мог бы выполнять.

[**Дочерний процесс** — это процесс, созданный другим процессом, который называется родительским процессом5](https://www.white-windows.ru/osnovnye-sistemnye-protsessy-windows-10-i-ih-naznachenie/)[6](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F).



2 последний startupInfo и processInfo(информация о запущенном процессе)

[**Остановка процесса:** Вы можете остановить процесс в Windows с помощью функции TerminateProcess1](https://learn.microsoft.com/ru-ru/windows/win32/sync/interprocess-synchronization)(или taskkill)[или команды Stop-Process в PowerShell2](https://winitpro.ru/index.php/2020/10/26/upravlenie-processami-powershell/).

Для работы с процессами в ОС Windows можно использовать различные API:

1. Windows API (WinAPI / WIN32?): Это набор функций, предоставляемых ОС Windows для работы с элементами операционной системы, включая процессы. Например, функции `CreateProcess`, `TerminateProcess`, `GetExitCodeProcess` и другие позволяют создавать, завершать и получать код завершения процесса соответственно.

2. Process and Thread Functions: Это набор функций, которые позволяют создавать и управлять процессами и потоками.

3. PSAPI Functions: Это набор функций, которые позволяют получать информацию о процессах и используемых ими ресурсах.

4. Tool Help Function: Это набор функций, которые позволяют перечислять процессы, потоки, модули и другие компоненты системы².

5. Windows Management Instrumentation (WMI): Это набор интерфейсов, которые позволяют получать информацию о системе и управлять ею. Win32 Provider предоставляет множество классов для получения системной информации, включая класс Win32\_Process.

Важно отметить, что эти API предоставляют низкоуровневый доступ к системе и требуют знания особенностей работы с процессами в Windows.

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

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

1. **Асинхронность:**
   * **Определение:** Асинхронное выполнение означает, что задачи могут выполняться независимо друг от друга, и выполнение одной задачи не блокирует выполнение другой. В программировании асинхронность часто реализуется с использованием колбэков, промисов или асинхронных функций.
   * **Пример:** Веб-запросы в браузере часто выполняются асинхронно, что позволяет продолжать выполнение других задач в то время, как ждем ответа от сервера.
2. **Параллельность:**
   * **Определение:** Параллельное выполнение означает, что несколько задач выполняются одновременно, фактически используя несколько ядер процессора или многозадачные возможности. Параллельность может быть достигнута как в многозадачных системах, так и на многоядерных процессорах.
   * **Пример:** Выполнение нескольких вычислений одновременно, используя несколько ядер процессора, является примером параллельности.

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

1. **Заявка на исполнение (Request for Execution):**
   * На этом этапе приложение инициирует асинхронную операцию, отправляя запрос на выполнение определенной задачи. Эта задача может быть, например, асинхронным вводом/выводом (I/O), асинхронным запросом к веб-сервису или любым другим видом асинхронной операции.
   * После отправки запроса на исполнение, приложение может продолжить свою работу без ожидания завершения асинхронной операции.
2. **Получение результата (Receiving the Result):**
   * На этом этапе приложение получает результат выполненной асинхронной операции. Это может быть событие, колбэк, оповещение или любой другой механизм, который уведомляет приложение о завершении операции.
   * После получения результата, приложение может обработать результат и продолжить выполнение своей логики.

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

[**1. NamedPipe (Именованный канал)**](https://learn.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication)[1](https://learn.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication)[2](https://learn.microsoft.com/en-us/windows/win32/ipc/named-pipes)[3](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%B5%D0%BD%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB)[: Именованный канал - это один из методов межпроцессного взаимодействия, который обеспечивает однонаправленную или двунаправленную связь между сервером канала и одним или несколькими клиентами канала1](https://learn.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication). [Все экземпляры именованного канала используют одно и то же имя канала, но каждый экземпляр имеет свои собственные буферы и дескрипторы и предоставляет отдельный канал для связи клиент/сервер2](https://learn.microsoft.com/en-us/windows/win32/ipc/named-pipes). [Именованные каналы поддерживают полнодуплексную связь по сети и несколько экземпляров сервера, обмен сообщениями и имитацию клиента, что позволяет подключающимся процессам использовать свой собственный набор разрешений на удаленных серверах1](https://learn.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication).

[**2. 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).

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

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

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

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

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

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

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

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

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

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

## **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** — это сущность в процессе, которую можно запланировать для выполнения. [Все потоки процесса совместно используют его виртуальное адресное пространство и системные ресурсы](https://learn.microsoft.com/ru-ru/windows/win32/procthread/about-processes-and-threads)

В ОС Windows каждый процесс предоставляет ресурсы, необходимые для выполнения программы. [Процесс имеет виртуальное адресное пространство, исполняемый код, открытые дескрипторы для системных объектов, контекст безопасности, уникальный идентификатор процесса, переменные среды, класс приоритета, минимальный и максимальный размер рабочего набора и по крайней мере один поток выполнения](https://learn.microsoft.com/ru-ru/windows/win32/procthread/about-processes-and-threads)[1](https://learn.microsoft.com/ru-ru/windows/win32/procthread/about-processes-and-threads).

**Процесс содержит 1 и более потоков.**

**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.

## основной поток процесса

При создании процесса, автоматически создается **основной (main) поток** (выполняется функция ядра, создающая поток). В простейшем – один поток процесса.

## **идентификатор потока**

**Идентификатор -** уникально идентифицирует поток повсюду в системе. Идентификаторы потоков доступны с момента создания потока и остаются действительными до его завершения.

**Маркер доступа** — это объект, описывающий контекст безопасности процесса или потока. Сведения в маркере включают удостоверение и привилегии учетной записи пользователя, связанной с процессом или потоком.

Данные, сохраняемые при переключении процессора с одного потока на другой и предназначенные для последующего возобновления выполнения данного потока;

## **понятие многопоточности**

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

Поддерживается OS, есть приоритетная и корпоративная многопоточность;

(***Многопоточность*** – модель (парадигма) программирования, состоящая в том, что процесс, порождённый в операционной системе, может состоять из нескольких одновременно выполняемых потоков.)

(Смелов, дословно)

## **понятие потокобезопасность**

**Tread-Safety: Потокобезопасность кода (программы) –** свойство программного кода (программы) корректно работать в нескольких потоках одновременно.

## **диспетчеризация потоков**

**Диспетчеризация потоков** - это процесс (механизм) управления выполнением потоков и реализации различных методов (способов) планирования в операционной системе. Диспетчеризация ответственна за выбор потока, который будет выполняться на процессоре в данный момент, и за управление временем, которое каждый поток получает для выполнения.

**Диспетчеризация -** заключается в реализации найденного в результате планирования (динамического или статистического) решения, то есть в переключении про­цессора с одного потока на другой. Прежде чем прервать выполнение потока, ОС запоминает его контекст, с тем, чтобы впоследствии использовать эту инфор­мацию для последующего возобновления выполнения данного потока. диспетчеризация потоков осуществляется OS или самим потоком.

**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**

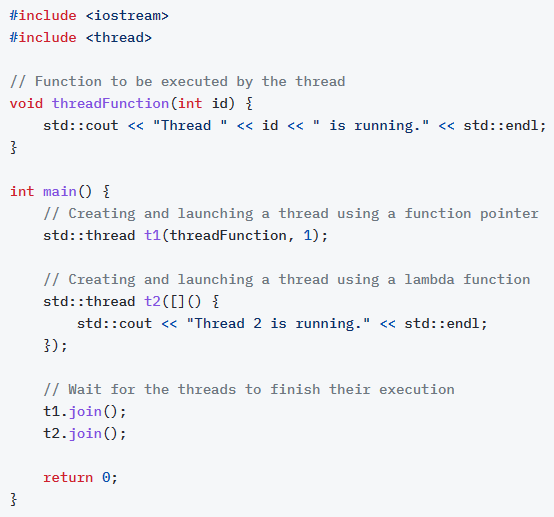
Асинхронность (asynchrony) подразумевает, что операция может быть выполнена кем-то на стороне: удаленным веб-узлом, сервером или другим устройством за пределами текущего вычислительного устройства.

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++:**

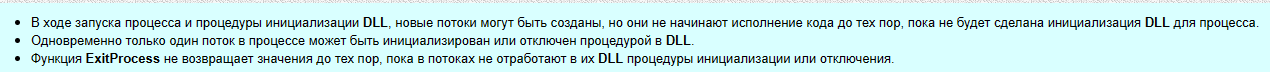


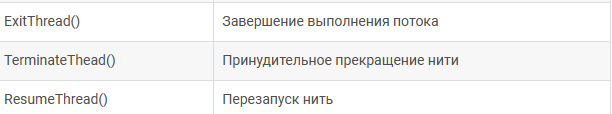
В C ++ потоки создаются с использованием класса std :: thread. Нить - отдельный поток исполнения; это аналогично тому, как помощник выполняет одну задачу, одновременно выполняя другую. Когда весь код в потоке выполняется, он завершается. Или с помощью функции **CreateThread:**



Если функция завершается успешно, величина возвращаемого значения - дескриптор нового потока.

Если функция завершается с ошибкой, величина возвращаемого значения - ПУСТО (NULL). Чтобы получать дополнительные данные об ошибках, вызовите [**GetLastError.**](https://narovol.narod.ru/_tbkp/New_MSDN_API/Debbag_error/fn_getlasterror.htm)





API для синхронизации в вопросе номер 6.

## **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. 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. В качестве разделяемого ресурса может выступать файл, network connection или какой-нибудь запрос.

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).

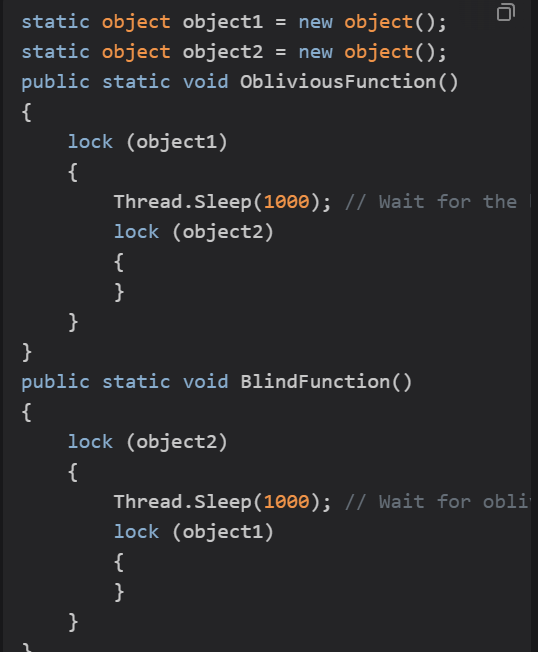
Функции ожидания блокируют выполнение потока до тех пор, пока заданный объект находится в состоянии Not Signaled.

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

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

**Синхронизация –** механизмустановления порядка работы потоков

**Взаимоблокировка (deadlock)** - это ситуация, когда два или более потоков блокируются и ждут друг друга, чтобы завершить свою работу. В результате ни один из потоков не может продолжить выполнение, что приводит к зависанию программы.



**Механизмы синхронизации потоков в Windows:**

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

*- Mutex;*

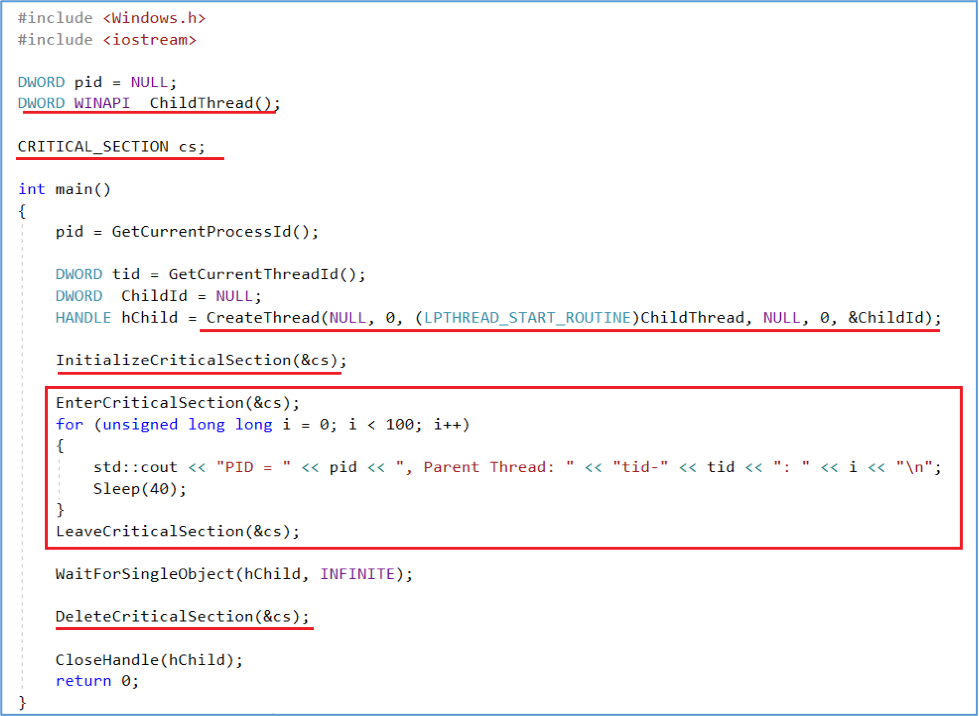
*- Semaphore;*

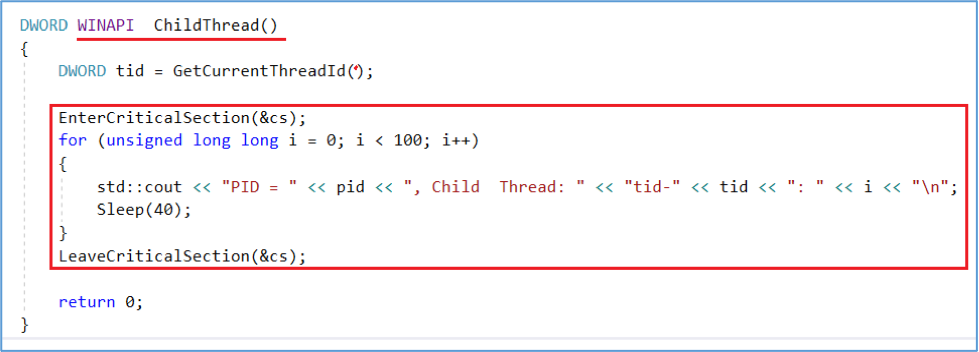
*- Atomic operation (interlocking function)*

*- Event;*

*- Writable timer.*

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



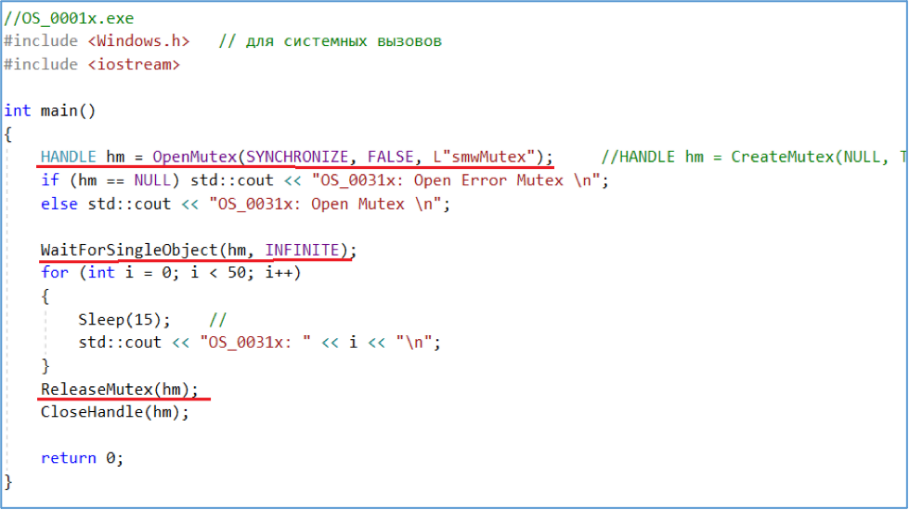


* 1. **Мьютекс** (англ. mutex, от **mut**ual **ex**clusion — «взаимное исключение») — это базовый механизм синхронизации. Он предназначен для организации взаимоисключающего доступа к общим данным для нескольких потоков с использованием барьеров памяти (для простоты можно считать мьютекс дверью, ведущей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

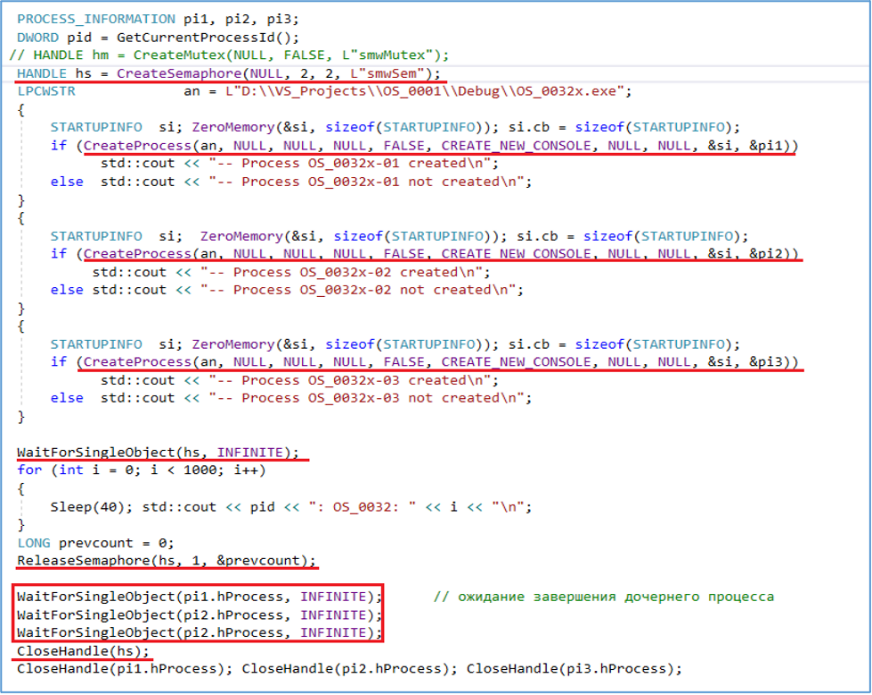
Как только хотя бы один процесс запрашивает владение мьютексом, он переходит в несигнальное состояние и остается таким до тех пор, пока не будет освобожден владельцем. Такое поведение позволяет использовать мьютексы для синхронизации совместного доступа нескольких процессов к разделяемому ресурсу.

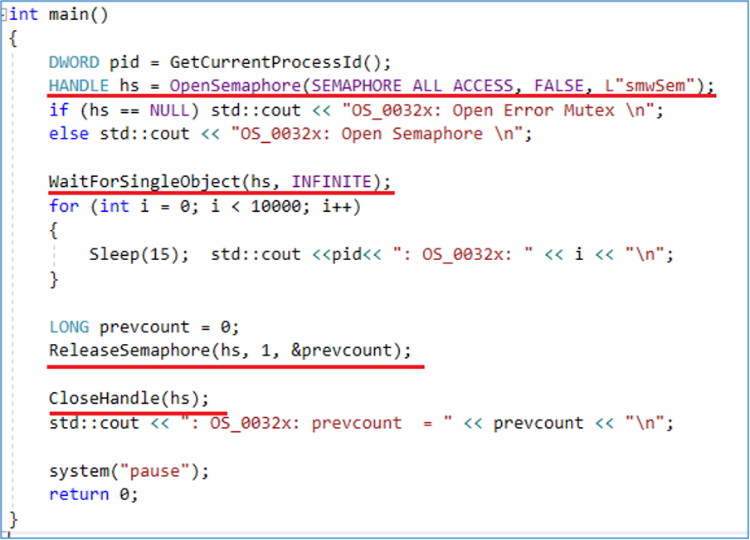
В ОС семейства [Windows](https://ru.wikipedia.org/wiki/Microsoft_Windows) разница между мьютексом и критической секцией в том, что мьютекс является объектом ядра и может быть использован несколькими процессами одновременно (**т.е. может выполнять синхронизацию процессов),** критическая секция же принадлежит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





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





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

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

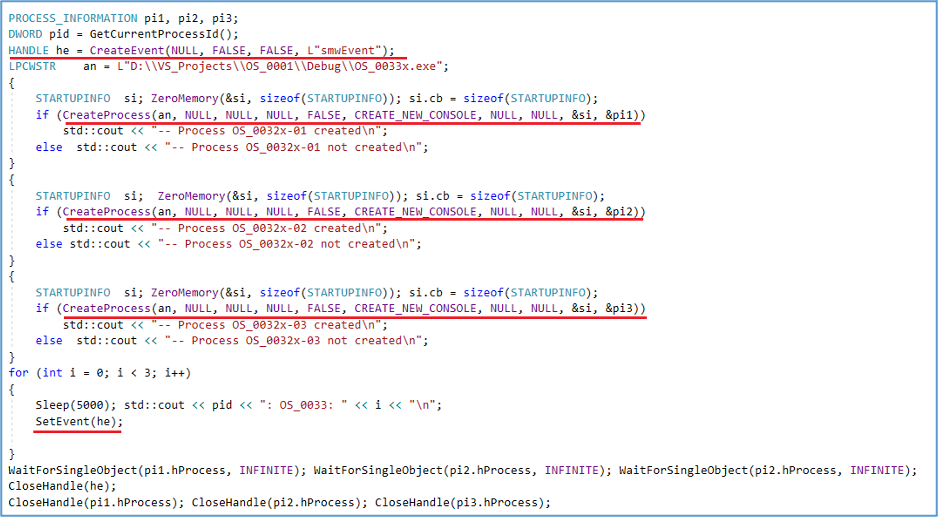
- Ручная событийная блокировка имеет два состояния (сигнальное и незавершенное), в то время как автоматическая событийная блокировка имеет только одно состояние (сигнальное).

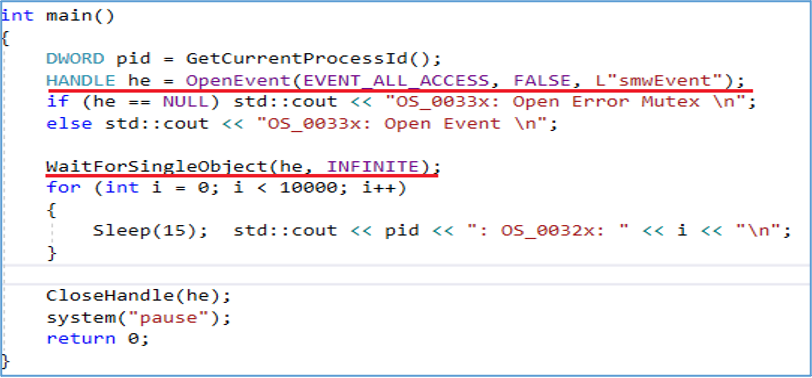
- Ручная событийная блокировка может использоваться для синхронизации между несколькими потоками или процессами, в то время как автоматическая событийная блокировка обычно используется для синхронизации внутри одного потока.

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

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

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





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

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

Для его создания используется функция CreateWaitableTimer.

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

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

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

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

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

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

## **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.**

**Файловая система** - порядок, определяющий способ организации, хранения и именования данных на носителях информации.

**Файловая система** — часть операционной системы, чтобы создать абстракцию над данными, с которыми мы собираемся работать в рамках данной ОС (Смелов на лекции)

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

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

Она включает в себя следующие аспекты:

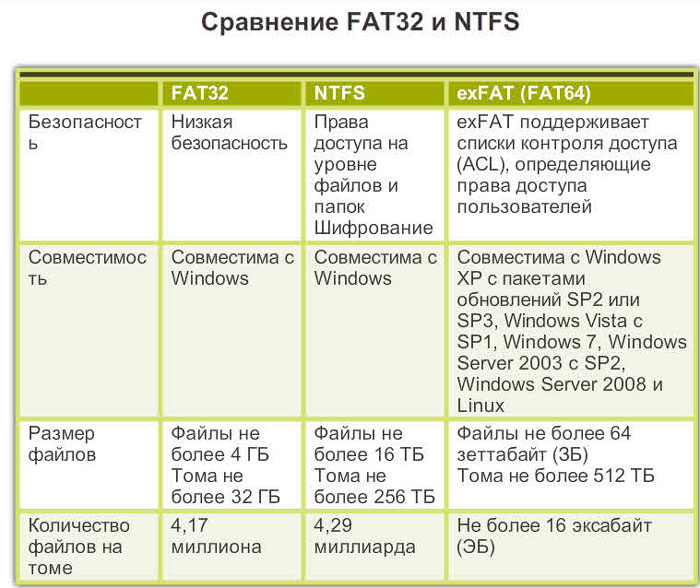
* **Файлы и директории**: Файлы представляют собой логические единицы данных, которые могут содержать текст, изображения, звуки и другие типы информации. Директории служат для организации файлов в иерархической структуре. Логические пути и имена файлов и директорий позволяют пользователям и программам 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** 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.

**Отличия файловых систем:**



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

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

**Оглавление файловой системы -** 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.

**Файл** – набор логический записей.

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

**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.

**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 вводы/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

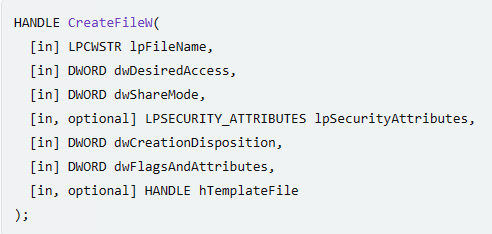
Буферы ввода-вывода используются для ускорения и оптимизации ввода-вывода данных. Они позволяют отложить чтение или запись данных для уменьшения числа вызовов низкоуровневых функций чтения/записи и снижения количества операций чтения/записи на диске. Это может улучшить производительность приложения и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

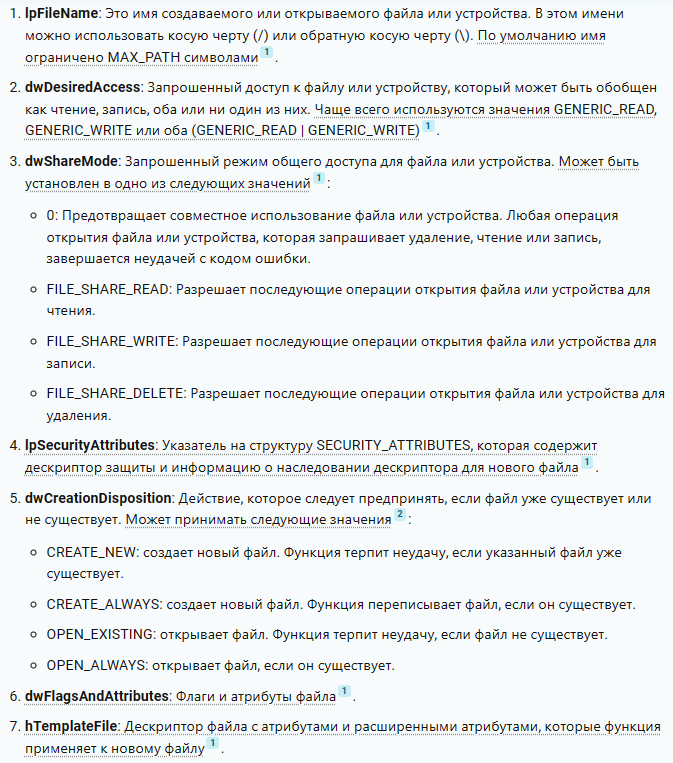
**Кеширование ввода/вывода:**

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:**

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





* **Delete file**
* **WriteFile**
* **ReadFile**
* **CopyFile**
* **MoveFile**
* **ReplaceFile** (заменяет файл в другом месте, сохраняя его атрибцты и текущую дату)
* **LockFile**(блокировка на определённый диапазон байтов в файле)
* **UnlockFile**
* **LockFileEx**

**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.

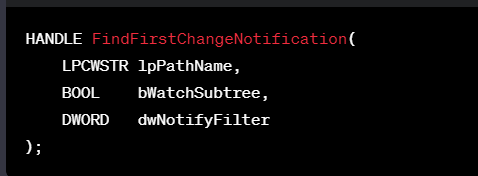
**File position pointer (указатель позиции файла)** - это специальный указатель, который отслеживает текущую позицию в файле, который был открыт для чтения или записи. Он показывает, где следует начинать чтение или запись данных в файл. Может быть изменен с помощью функций установки указателя позиции файла, таких как fseek() в C.

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

**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.

**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,



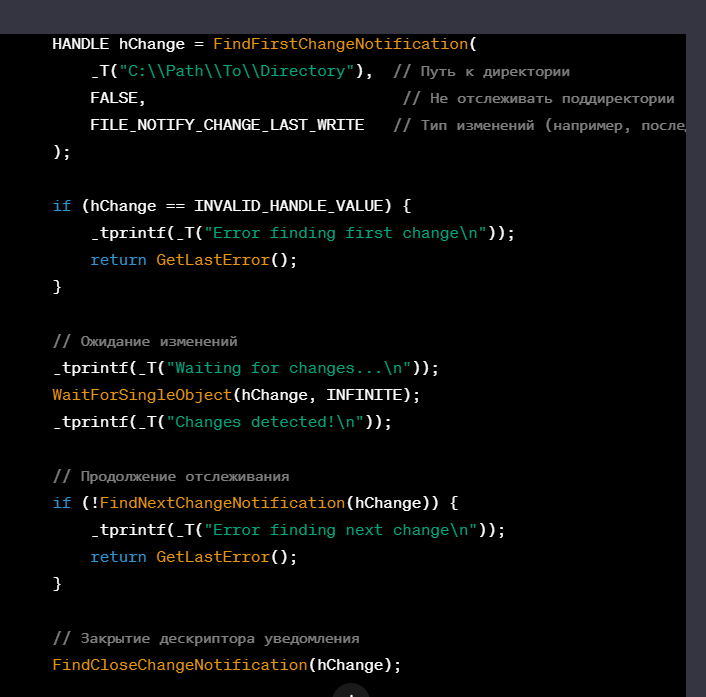
* **lpPathName**: Путь к директории, которую нужно отслеживать.
* **bWatchSubtree**: Если установлено в **TRUE**, отслеживаются изменения в поддиректориях.
* **dwNotifyFilter**: Фильтр, указывающий типы изменений, которые нужно отслеживать (например, **FILE\_NOTIFY\_CHANGE\_FILE\_NAME** для изменений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

**Файловые 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.

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

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

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

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

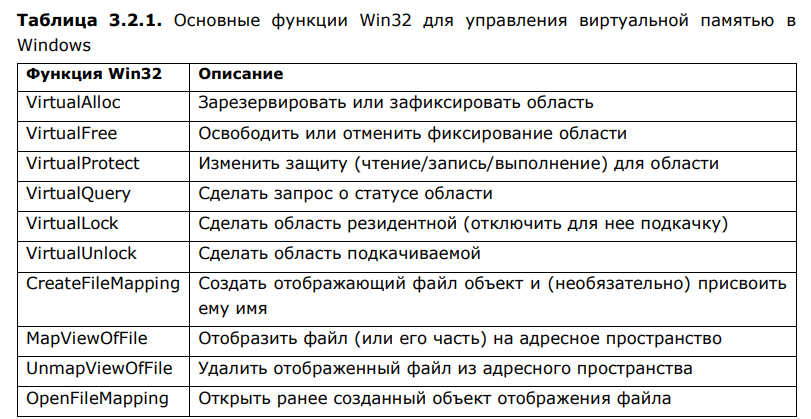
**Оперативная память**обычно имеет меньший объем, но высокую скорость доступа, поскольку она находится близко к процессору.

**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.

**Windows API (WinAPI):**

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

Примеры функций:

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

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

**HeapCreate**: Создает новую кучу.

**HeapDestroy**: Уничтожает указанную кучу.

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

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

Примеры функций:

**malloc**: Выделяет блок памяти.

**calloc**: Выделяет и обнуляет блок памяти.

**realloc**: Изменяет размер ранее выделенного блока.

**free**: Освобождает выделенный блок памяти.

**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.

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

Внутренняя фрагментация:

- Внутренняя фрагментация возникает, когда выделенный блок памяти немного больше, чем фактически требуется программе. Например, если программа запросила 10 байт, а ей выделено 16 байт, то внутренняя фрагментация составляет 6 байт.

- Это особенно часто происходит в системах, использующих фиксированный размер блоков для выделения памяти.

Внешняя фрагментация:

- Внешняя фрагментация возникает, когда существует множество свободного пространства, но оно разбито на разрозненные фрагменты, которые не могут быть использованы для удовлетворения запросов на выделение памяти.

- Это может быть вызвано выделением и освобождением памяти в произвольном порядке, что приводит к появлению небольших свободных областей между уже выделенными блоками.

Компромисс между внутренней и внешней фрагментацией является сложной задачей, и существует множество стратегий управления памятью для минимизации эффектов фрагментации. Например, алгоритмы выделения и освобождения памяти могут быть оптимизированы для снижения внутренней фрагментации, а алгоритмы сборки мусора могут использоваться для борьбы с внешней фрагментацией.

В контексте управления памятью и кучей (heap), термины **"упаковка" и "распаковка"** обычно используются в отношении сжатия или оптимизации использования памяти

Упаковка (Packing):

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

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

Распаковка (Unpacking):

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

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

Примеры операций упаковки и распаковки могут зависеть от конкретной реализации алгоритмов управления памятью. Например, алгоритмы сборки мусора могут включать в себя упаковку, перемещая выделенные объекты, чтобы освободить фрагментированное пространство и предотвратить появление фрагментации в будущем.

Однако, стоит отметить, что упаковка и распаковка могут быть дорогостоящими операциями, поскольку они могут потребовать перемещения данных в памяти, что влечет за собой дополнительные вычислительные затраты и возможно приводит к временному замедлению программы.

**Схема с двумя списками** (занятая и свободная области памяти) является распространенным методом управления памятью в системах, использующих кучу (heap). Этот метод включает в себя два основных списка:

1. **Список занятой памяти** (Allocated Memory List):

- **Определение**: Этот список содержит информацию о занятых блоках памяти, включая их размер и, возможно, другую метаинформацию (например, информацию о типе данных).

- **Структура**: Каждый элемент списка содержит информацию о блоке памяти, например, указатель на начало блока и его размер. Эти элементы образуют список, который представляет собой занятые области памяти в куче.

- **Использование**: Этот список позволяет программе отслеживать и управлять занятыми блоками памяти.

2. **Список свободной памяти** (Free Memory List):

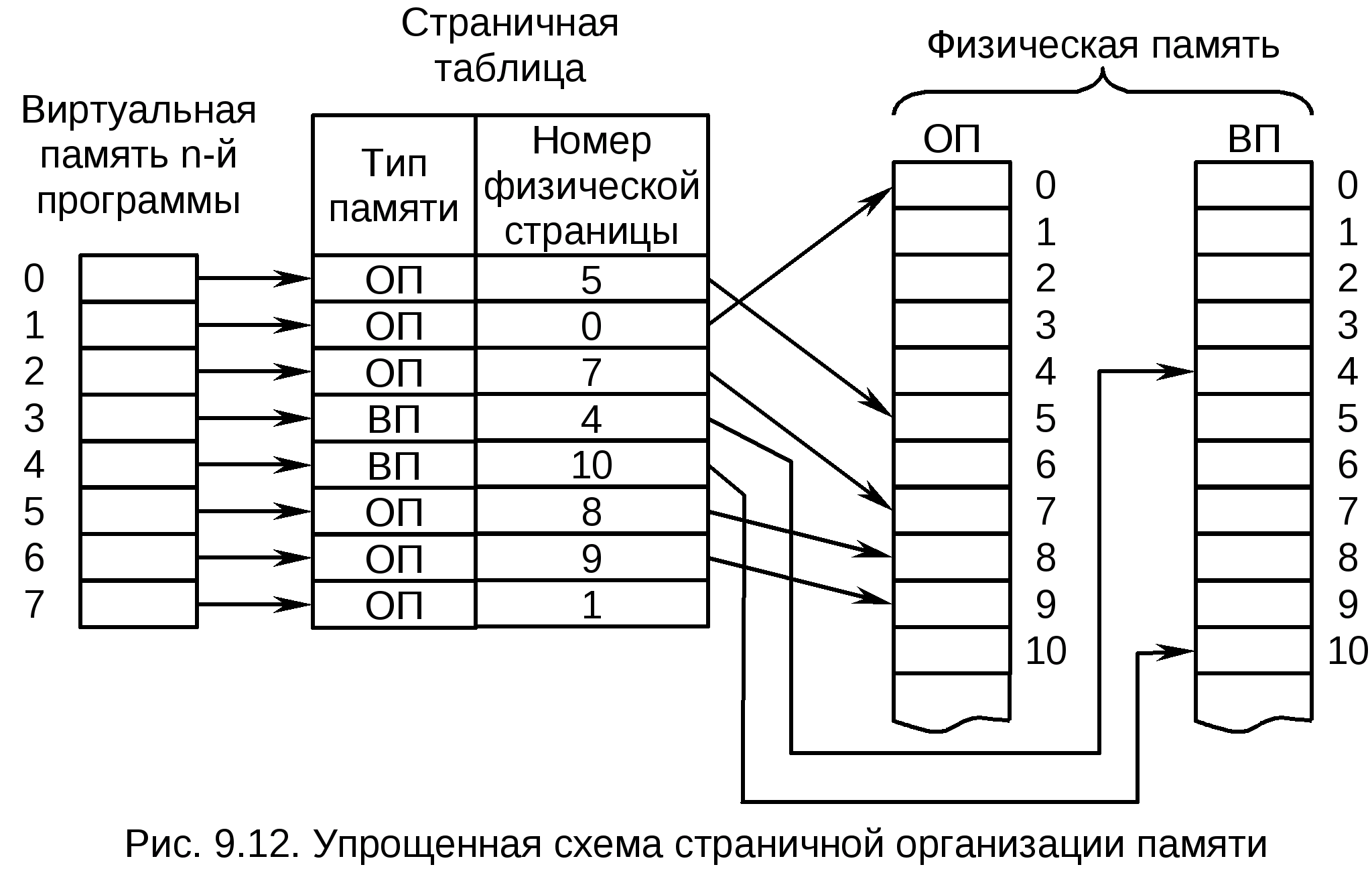
- **Определение**: Этот список содержит информацию о свободных блоках памяти, доступных для выделения. Эти блоки могут быть объединены вместе для уменьшения фрагментации.

- **Структура**: Каждый элемент списка содержит информацию о свободном блоке, такую как указатель на начало блока и его размер. Элементы списка представляют собой свободные области памяти в куче.

- **Использование**: Этот список позволяет эффективно находить свободные блоки памяти при запросе на выделение, а также освобождать память и добавлять свободные блоки в список.

Структура и реализация этих списков могут различаться в зависимости от конкретной системы управления памятью. Например, для списка свободной памяти могут использоваться различные алгоритмы, такие как "first fit" (выбор первого подходящего блока) или "best fit" (выбор наилучшего подходящего блока). Также возможно применение дополнительных оптимизаций для управления фрагментацией и повышения эффективности работы с кучей.

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.

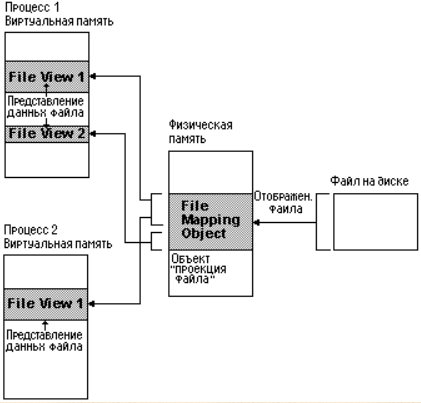
Память, выделенная в куче **HeapAlloc**, является непрерывной. Не следует выполнять запись или чтение из памяти в куче, за пределами диапазона памяти, выделенной **HeapAlloc**, и не следует предполагать связь между двумя областями памяти, выделенными **HeapAlloc**.

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.

## **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.



**File mapping:**

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

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

## **использование образа файла, как средства межпроцессного взаимодействия**

Сопоставление файлов можно использовать в качестве механизма межпроцессорного взаимодействия Для того, что бы пользоваться одним файлом в разных процессах, то один процесс должен создать или открыть файл, потом создать объект отображения (обязательно задать ему имя (не должно совпадать с объектами синхронизации)). Далее другие процессы используют функцию OpenFileMapping(), а потом MapViewOfFile() что бы сопоставить представление с адресным пространством процесса.

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.

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:**

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);

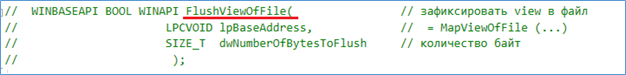
4 (B) Закройте объект сопоставления файлов. Этот шаг удаляет отображение файла из физической памяти.

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.



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).

## **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.

## **Различия 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.

## **Основное**

**Структура DLL-библиотеки 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.

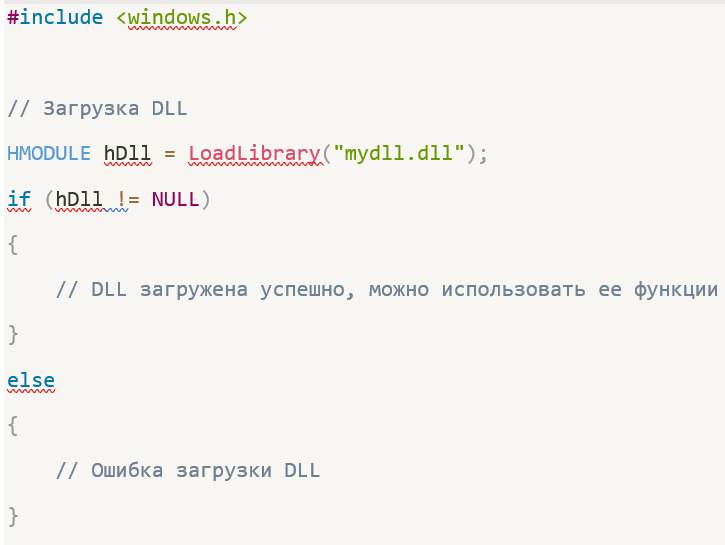
**\_\_declspec(dllexport)** - это ключевое слово для явного указания компилятору, что функция, класс или переменная должны быть экспортированы из исполняемого файла или DLL (динамической библиотеки) и стать доступными для использования другими модулями программы или внешними приложениями.

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:**

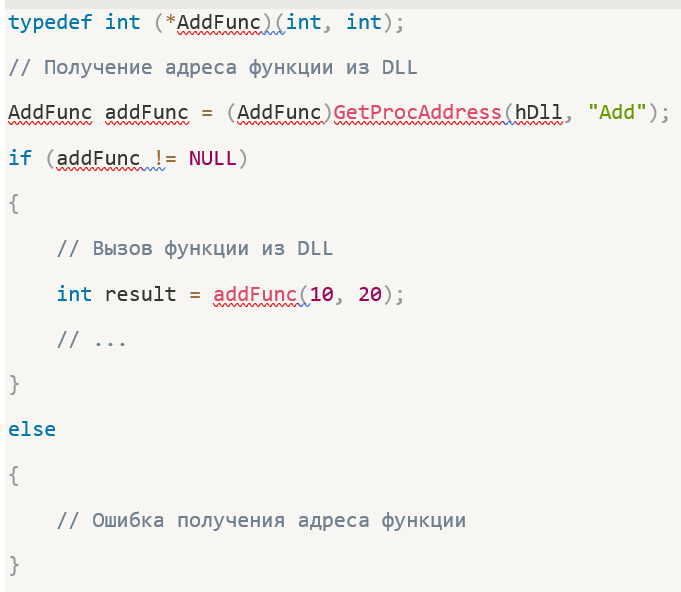
* + **Явная** (через LoadLibrary, а в таблицу экспорта будут заносится только те функции, переменные и т.д., которые мы в программе загрузим явным образом)
  + **Неявная** (через свойства проекта, в таблицу экспорта будут занесены ВСЕ функции из DLL)

**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.

## **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.

**COM (Component Object Model) -** это платформа для разработки компонентных приложений в операционных системах Windows.

**COM-программирование** – это способ разработки приложений, использующих COM-объекты.

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

**COM-объект (или компонент)** – это программный модуль, который реализует определенный функционал и предоставляет его в виде интерфейсов для использования другими приложениями.

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

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

**COM-объект**: создается и размещается в контейнере: **1) DLL-модуле; 2) EXE-модуле.**

**GUID** – тип идентификатора объекта. GUID обеспечивает уникальность идентификаторов COM-объектов и интерфейсов. Размер GUID-идентификатора составляет 128 бит (16 байт).

**CLSID** - это уникальный идентификатор COM-объекта, который позволяет ему быть идентифицированным в системе.

**Основные понятием СОМ**

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

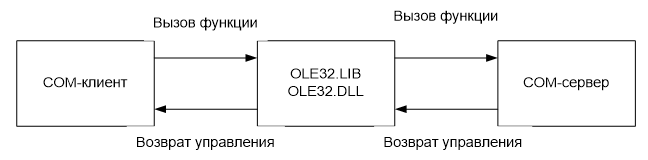
***Позднее связывание (late binding):***

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

## COM-интерфейс

COM-интерфейс (Component Object Model Interface) - это стандартизированный способ для объектов взаимодействовать друг с другом в системе Windows. Он определяет набор методов и свойств объекта, которые могут быть использованы другими объектами в системе. Применяется для доступа к методам COM-объекта.

## стандартные COM-интерфейсы

Стандартный COM-интерфейс - это интерфейс, который определен и поддерживается операционной системой Windows. Он является частью стандартных COM-библиотек, и может быть использован любым объектом, реализующим его интерфейс. Интерфейс, который описан в спецификации COM.

Iunknown, IclassFactory

**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:**

* ***QueryInterfac***e (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 CA, чтобы клиент мог получить к нему доступ.

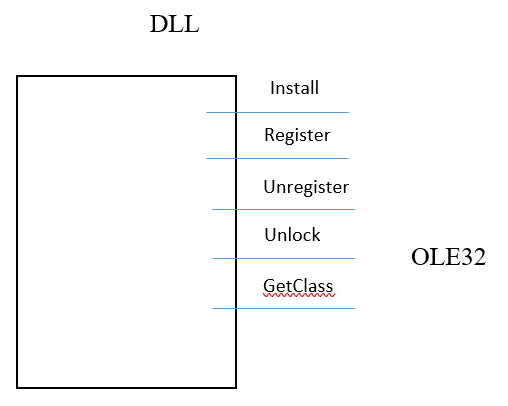
Сервер представляет из себя DLL, в которую экспортирует 5 функций, 3 используются для работы с ОС, 1 – для создания компонента.

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).



COM-объект состоит из двух компонентов (CF и CA). Он лежит в DLL.

В DLL общий счётчик для всех компонент.

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.

Чтобы работать с COM-компонентом со стороны клиента, клиент должен знать о CA следующие параметры: идентификатор CLSID, интерфейсы, которые он поддерживает. У каждого интерфейса необходимо знать id интерфейса, методы интерфейса.

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

DLL экспортирует следующие функции:

* ***Install***
* ***Register***
* ***Unregister***
* ***GetClassObject***
* ***CanUnloadNow***

Сервер должен зарегистрировать DLL в реестре. Теперь, когда клиент будет подключаться к OLE32 для работы с сервером, ему необходимо знать только CLSID. А OLE32 по этому CLSID залезет в реестр и возьмёт необходимую информацию о месте расположения DLL. После этого клиент передаёт OLE32 свои CLSID, IUnknown и IClassFactory.

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.



## **структура COM-клиента**

**Клиент** - это приложение, которое использует функциональность COM-объекта, предоставляемую через его интерфейсы.

или

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

COM-клиент взаимодействует с COM-компонентом через его интерфейсы. Для этого клиент должен выполнить следующие шаги:

* Получить указатель на интерфейс объекта, используя функцию, такую как **CoCreateInstance** или **CoGetClassObject**.
* Вызвать методы объекта, используя полученный интерфейс.

**COM-клиент** должен «знать»:

* CLSID объекта;
* тип DLL-сервера (контейнера);
* ID интерфейсов объекта, чтобы использовать COM-объект

## **структура COM/DLL-сервера**

Сервер - это приложение, которое реализует функциональность COM-объекта и предоставляет ее через интерфейсы для использования другими приложениями.

COM-сервер может быть однокомпонентным (реализующим один тип объектов) или многокомпонентным (реализующим несколько типов объектов).

COM-сервер может иметь тип:

* 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);

## **экспортируемые стандартные функции**

**DllCanUnloadNow** - Периодически вызывается СОМ для проверки возможности выгрузки DLL (проверяет отсутствие экземпляров объектов, хранящихся в DLL и проверки не заблокирована ли с помощью LockServer IClassFactory). Реализуется во внутризадачном сервере.

**DllGetClassObject** - Точка входа, реализуемая во внутризадачных серверах так, что интерфейсы их фабрик классов могут быть получены клиентами. (возвращает фабрику класса для конкретного СОМ-класса.)

**DllInstall** - Выполняет установку и настройку библиотеки DLL.

**DllRegisterServer**   - Добавляет записи в системный реестр. 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

**DllUnregisterServer** - Удаляет записи из системного реестра

СОМ-объекты должны поддерживать саморегистрацию с помощью экспортируемых СОМ-функций DllRegisterServer и DllUnRegisterServer

## **регистрация COM/DLL-сервера**

**утилита regsvr32** предназначена для регистрации и удаления COM-объектов (DLL) в системном реестре Windows. Эта утилита вызывает функции DllRegisterServer и DllUnregisterServer, которые определены в библиотеке DLL COM-объекта.

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

(в NOTION хорошо написано, как по мне <https://www.notion.so/Windows-Windows--f1763890c6af45b3812e25ec74ce1381> )

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

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

);

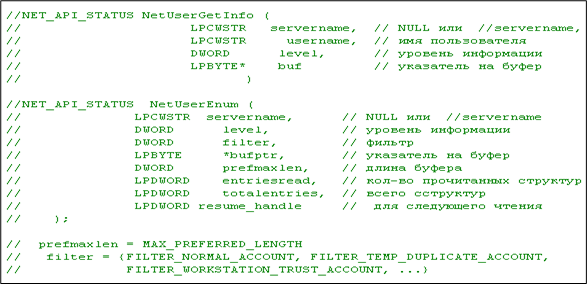
**2. Информация о пользователей**

****

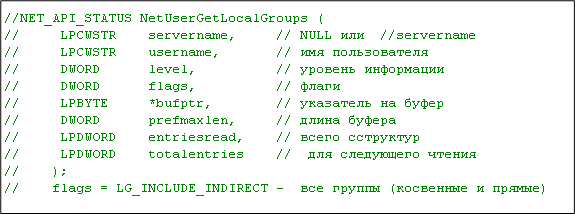
**3. Освободить память**

****

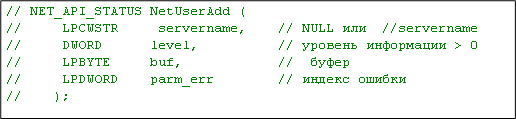
**4. Получить коллекцию пользователей**

****

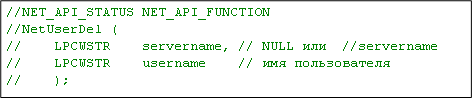
**5. Группы пользователя**

****

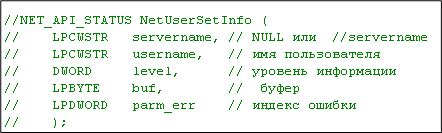
**6. Добавить пользователя**

****

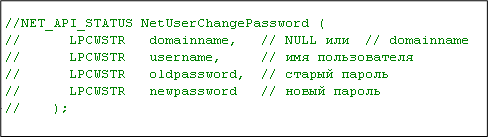
**7. Удалить пользователя**

****

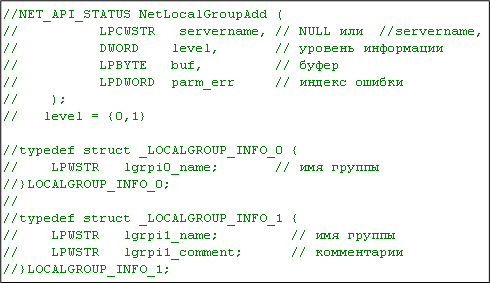
**8. Изменить информацию о пользователе**

****

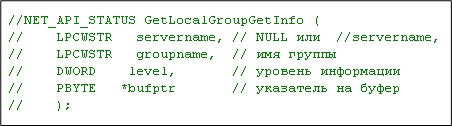
**9. Изменить пароль**

****

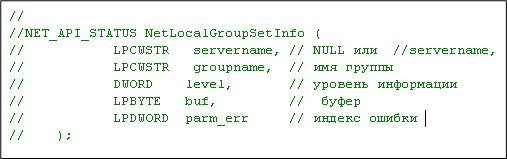
**10. Добавление группы**

****

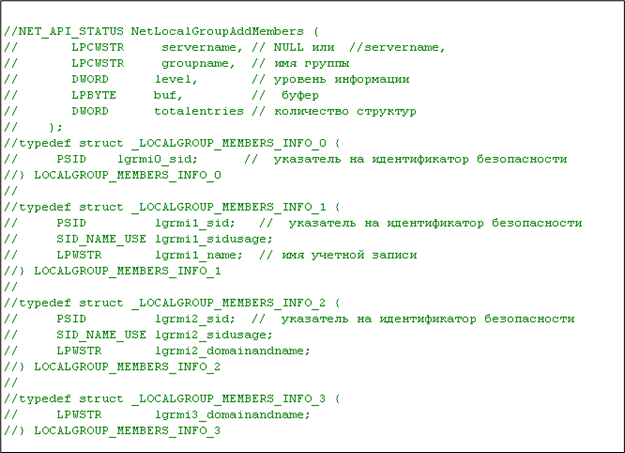
**11. Получить информацию о группе**

****

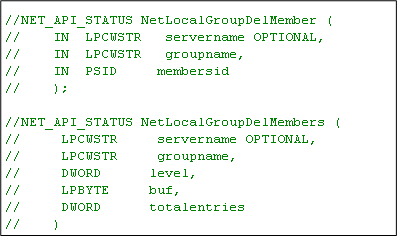
**12. Изменить информацию о группе**

****

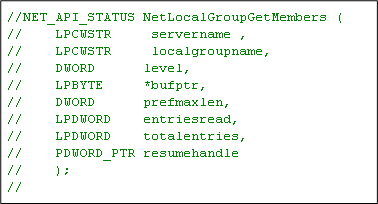
**13. Добавление членов локальной группы**

****

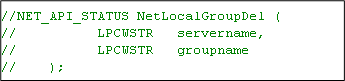
**14. Удаление членов группы**

****

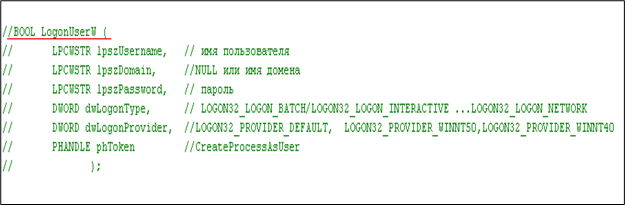
**15. Получить коллекцию членов группы**

****

**16. Удалить группу**

****

**17. Подключение пользователя**

****

## 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.

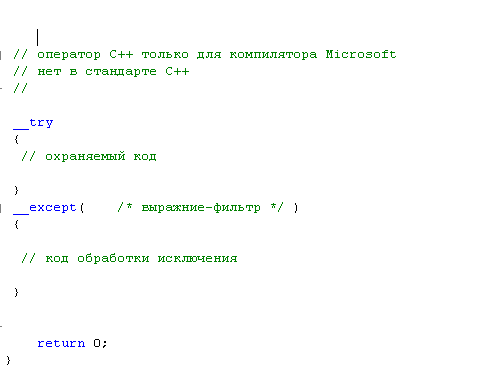
Исключения могут быть основаны на оборудовании или ПО. Структурированная обработка исключений полезна даже в том случае, когда приложения не могут полностью восстанавливаться после исключений. SEH позволяет отображать сведения об ошибках и захватывать внутреннее состояние приложения, чтобы помочь в диагностике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

**Программные конструкции для обработки ошибок**

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

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

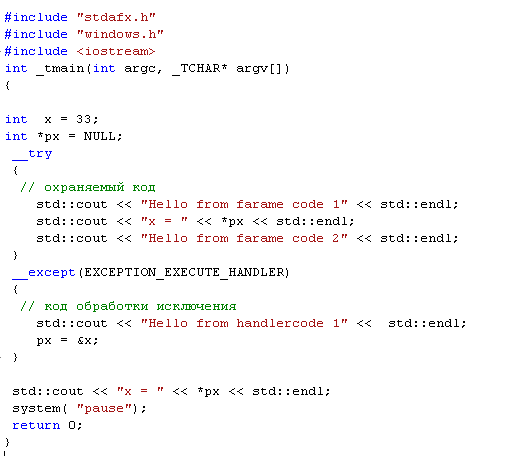
1. Сначала выполняется защищенный раздел \_\_try.
2. Если исключение при этом не возникает, выполнение переходит в инструкцию, стоящую после предложения \_\_except.
3. Если во время выполнения защищенного раздела возникает исключение или в любой подпрограмме вызывается защищенный раздел, \_\_except выражение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

****

нельзя goto в охраняемый код и в обработчик.

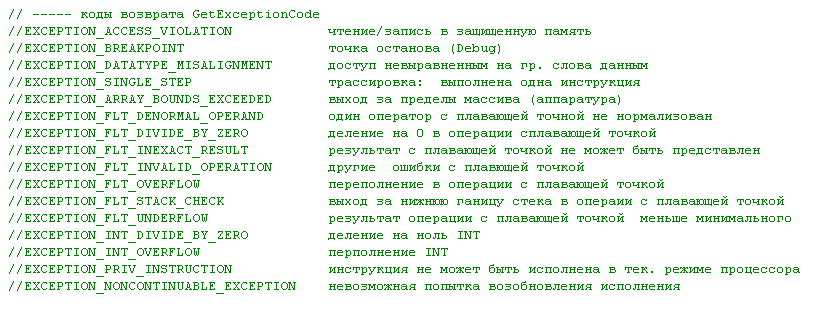
в выражении фильтра можно использовать две функции: GetExeptionCode, GetExceptionInformation.

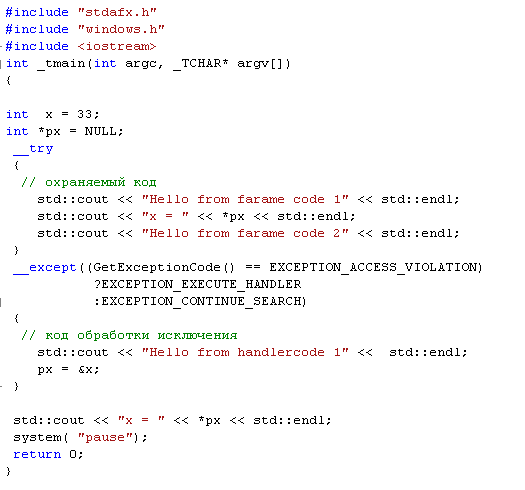
переменные, объявленные внутри {} – локальные

****

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

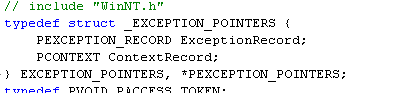
DWORD GetExceptionCode() – может использоваться только в выражении-фильтре (для определения дальнейших действий: обрабатывать, искать обработчик, вернуть управление в точку прерывания) или в блоке обработки исключения (для получения кода исключения).

****

****

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

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

****

****

ExceptionCode = GetExceptionCode()

ExceptionAddress – адрес инструкции

ExceptionFlag = 0 – возможно возобновление работы

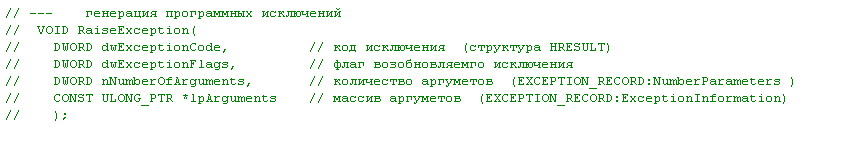
ExceptionFlag = EXEPTION\_NONCONTINUABLE – невозможно

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

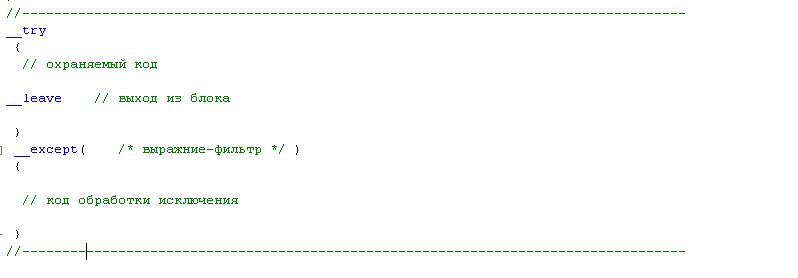
NumberParameters – количество параметров в ExcetptionInformation

ExcetptionInformation – массив 32-битных описателей исключения

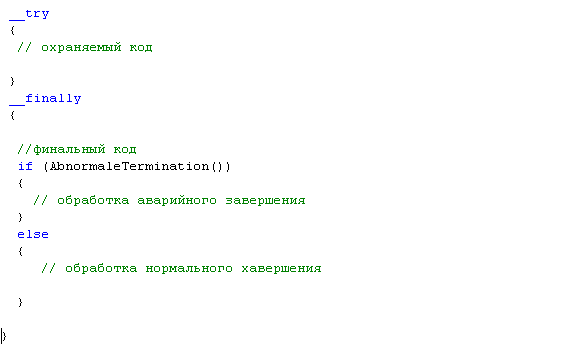
**SEH: Microsoft C++: генерация программных исключений**



SEH: Microsoft C++: передача управления

****

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

****

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 путём ввода с клавиатуры текстовых строк.

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

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

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

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

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

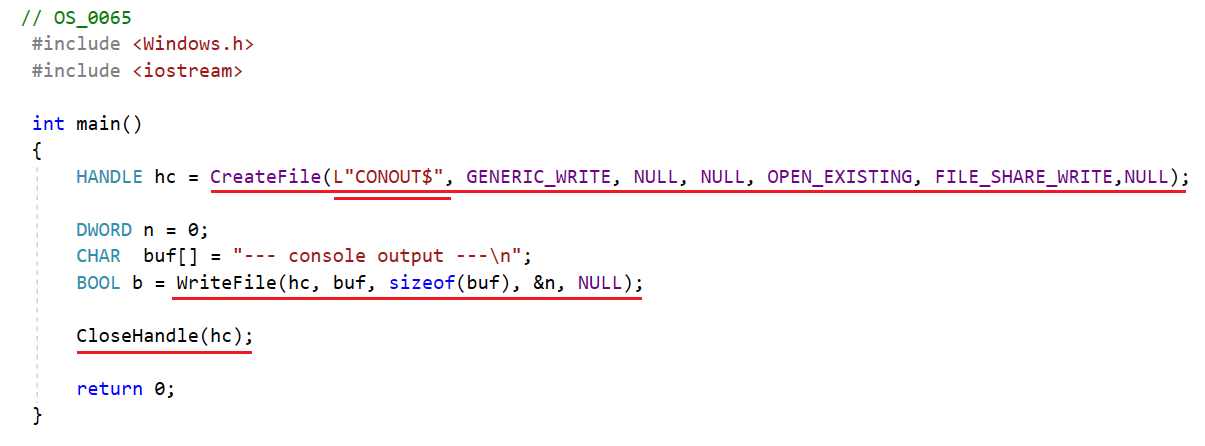
typedef struct \_CHAR\_INFO {  
union {  
WCHAR UnicodeChar;  
CHAR AsciiChar;  
} Char;  
WORD Attributes;  
} CHAR\_INFO, \*PCHAR\_INFO;

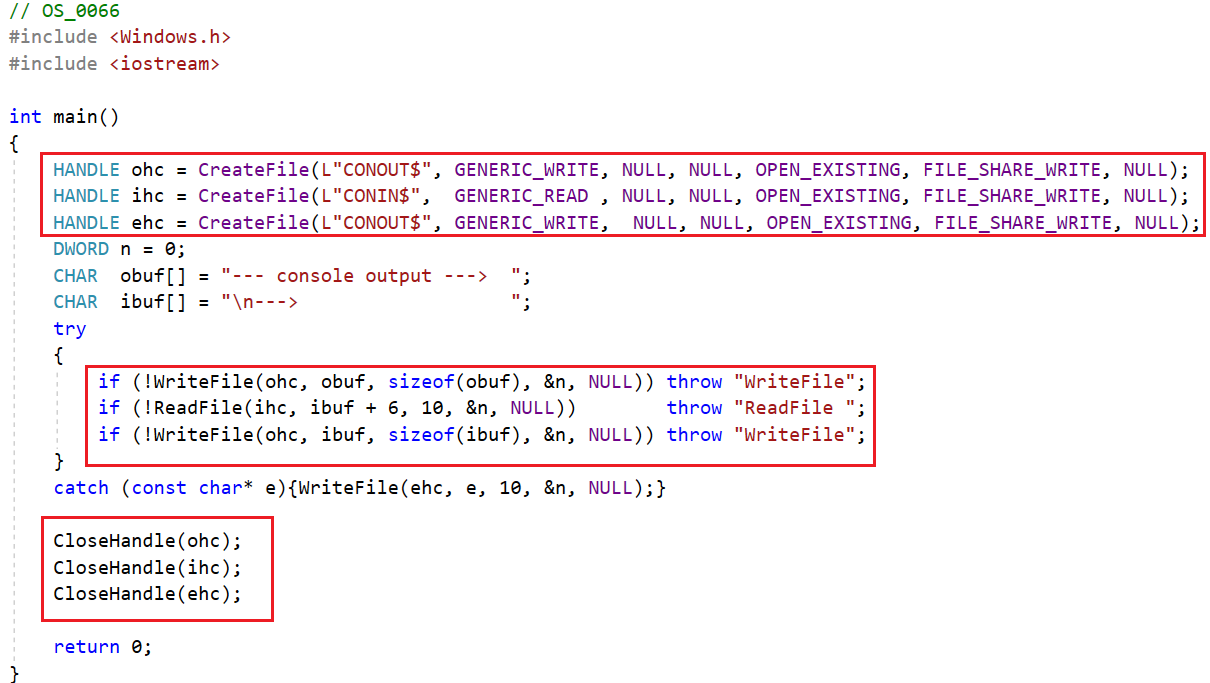
где объединение Char содержит символ, представленный в коде Unicode или ASCII, а поле Attributes определяет цвет фона и цвет текста, которыми выводятся символы на экран дисплея. Это значение может быть равно 0, что обозначает фон — черный, а цвет — белый, или любой комбинации из следующих констант: 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

Консоль — 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.

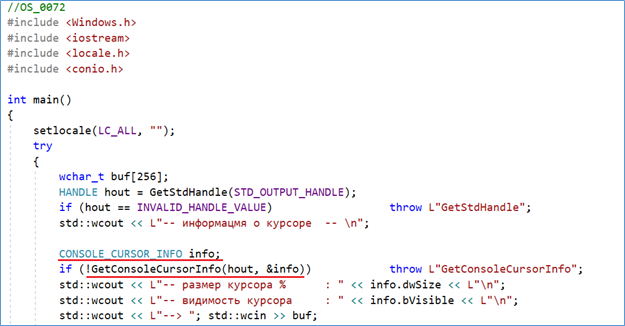
Стандартные потоки:

****

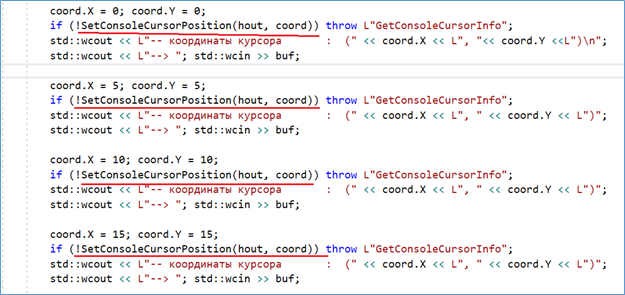
****

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



**Console: позиции курсора**

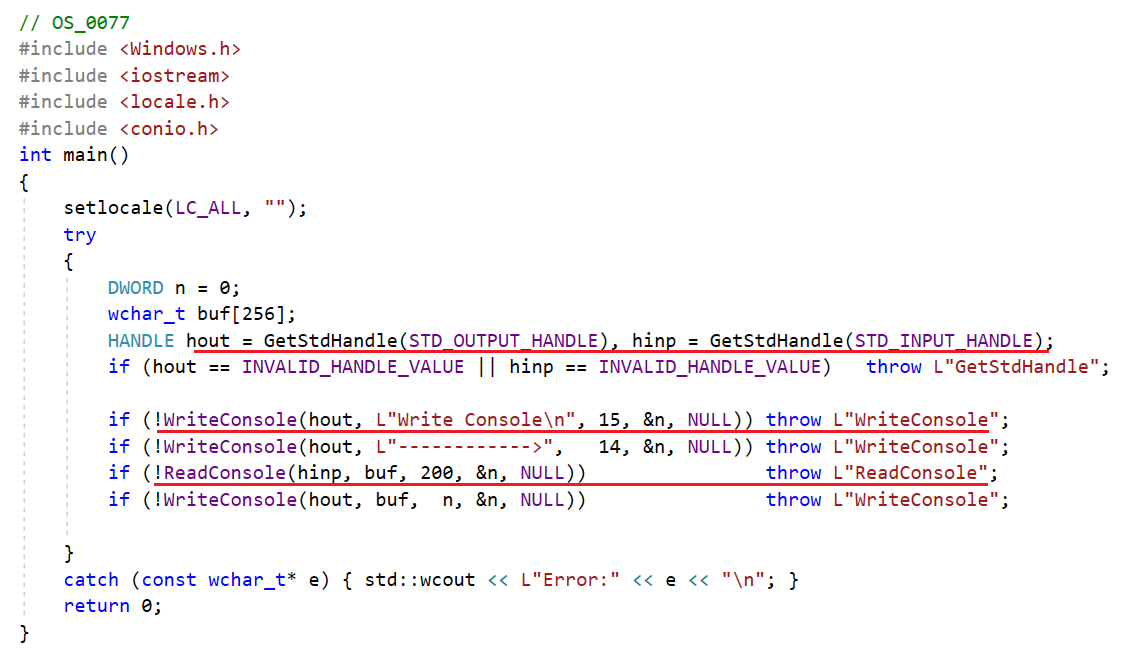


**Console: установка атрибутов консоли**



**Высокоуровневый ввод/вывод WriteConsole, ReadConsole**

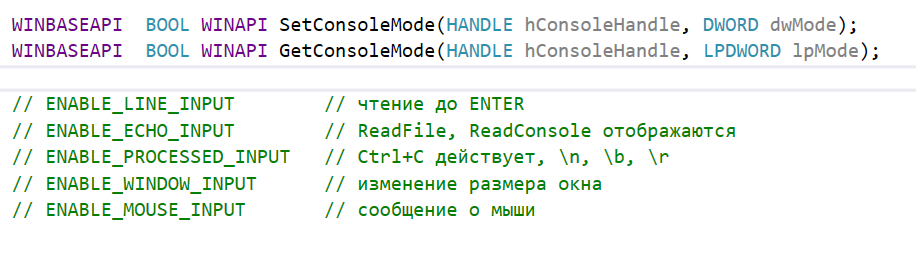
ReadFile, WriteFile – аналогичны WriteConsole, ReadConsole



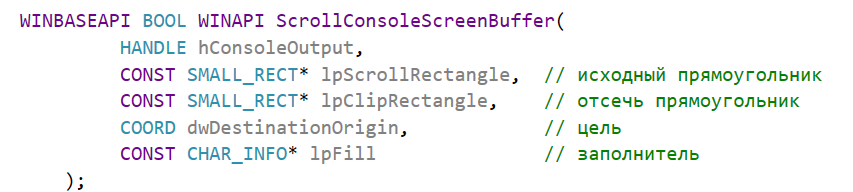
**Console: ввод низкого уровня ReadConsoleInput**



**режимы работы консоли**



**прокрутка буфера экрана**



SetConsoleTitle задаёт заголовок для текущего окна консоли.

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

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

GetCurrentConsoleFont извлекает сведения о текущем шрифте консоли.

FlushConsoleInputBuffer очищает входной буфер консоли.

AllocConsole выделяет новую консоль для вызывающего процесса.

AddConsoleAlias определяет псевдоним консоли для указанного исполняемого файла.

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

**Ответ 1:**

**Определение**:

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

**Назначение**:

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

**Применение**:

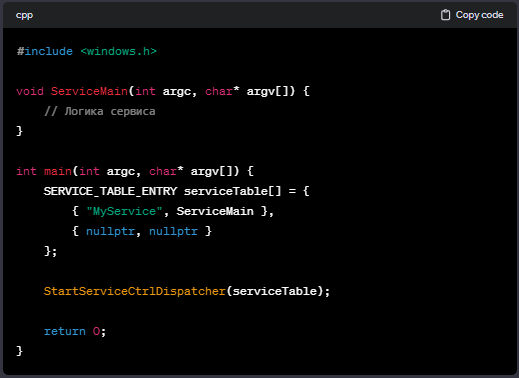
- Сервисы применяются в сценариях, где необходима постоянная и независимая от пользовательской активности работа. Например, службы безопасности, службы баз данных, службы печати и т. д.

**API для создания и управления:**

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

**Использование WinAPI:**

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

Пример кода:

**Управление**:

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

**Регистрация**:

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

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

**Ответ 2:**

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

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

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

**Windows service состоит из:**

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

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

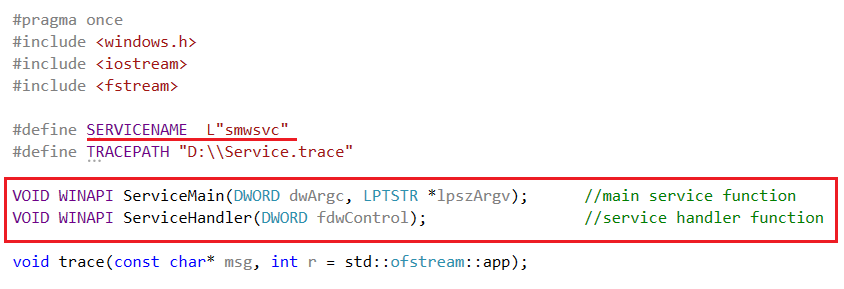
**Список служб находится** в реестре Windows «HKEY\_local\_machine\system\CurrentControlSet\Services».

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

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

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

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



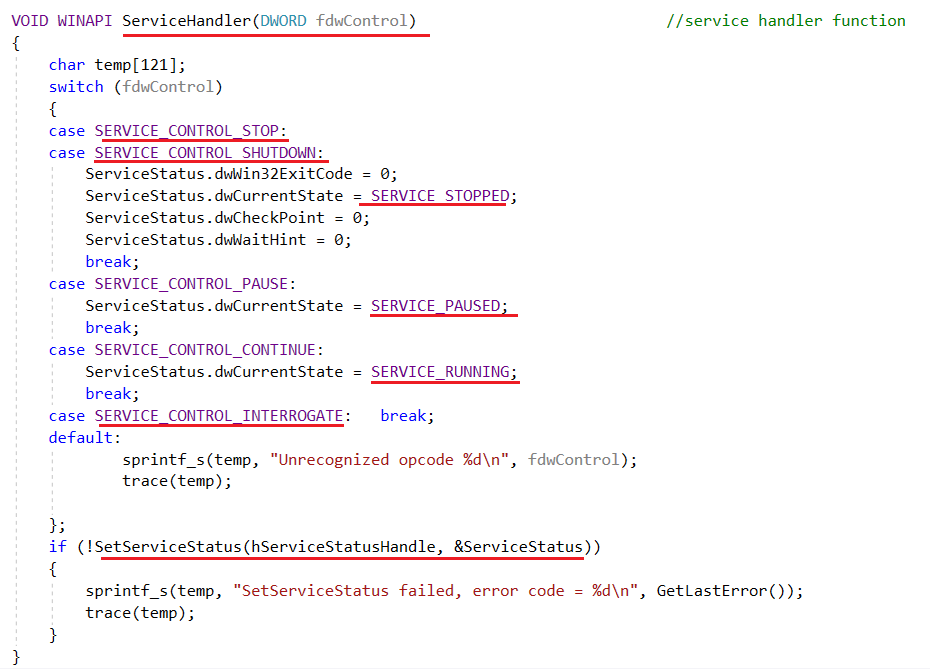


VOID WINAPI SvcMain(DWORD dwArgc, LPTSTR \*lpszArgv )

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

void WINAPI ServiceHandler(DWORD dwCode)

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



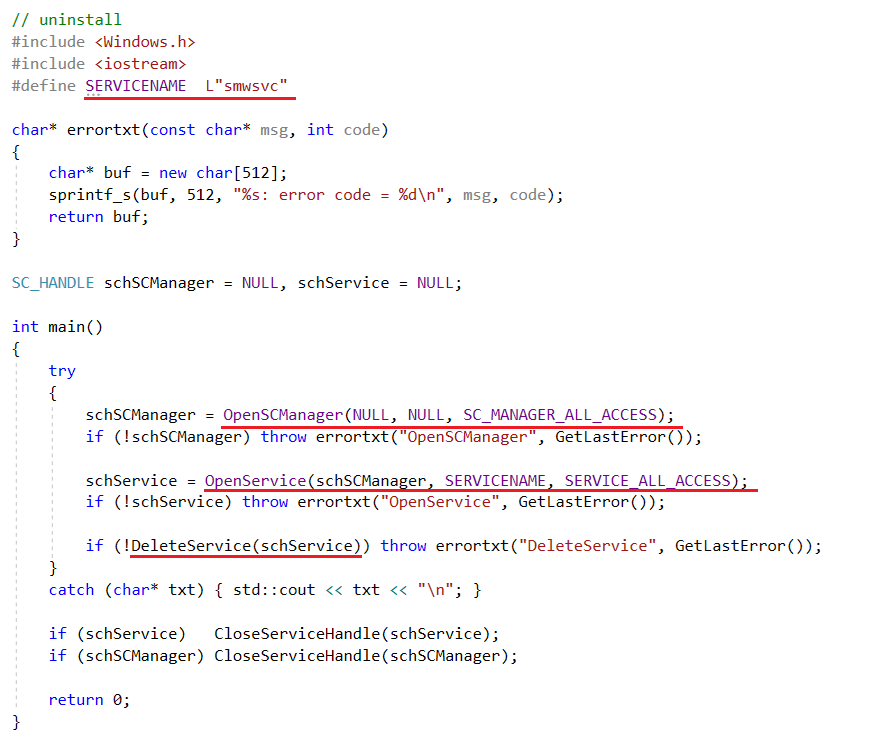
**создание сервиса, регистрация в реестре Windows.**



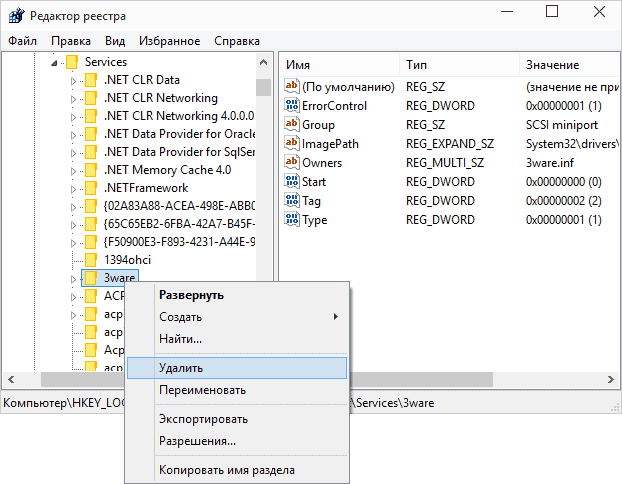


**удаление сервиса.**

**Кодом:**

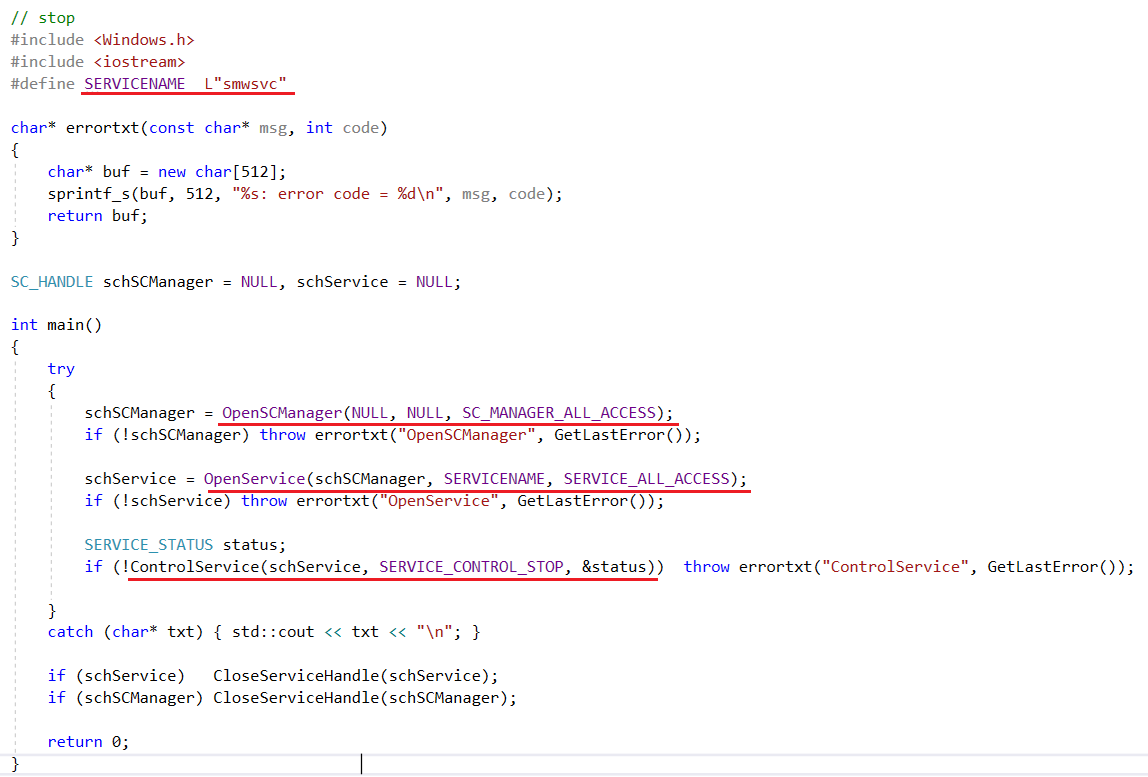


**Удалить службу Windows можно также и с помощью редактора реестра**

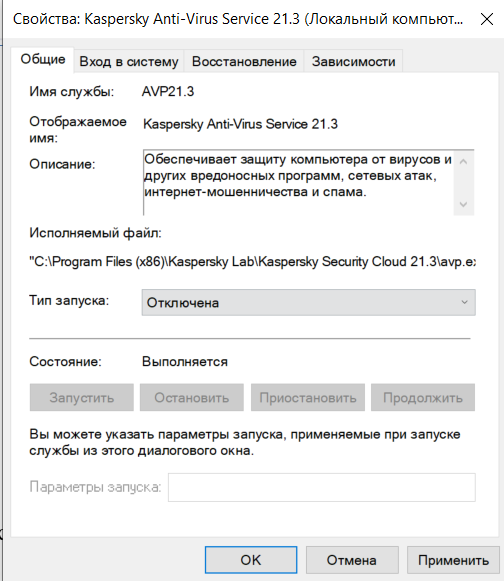


**Windows Services:** остановка сервиса.

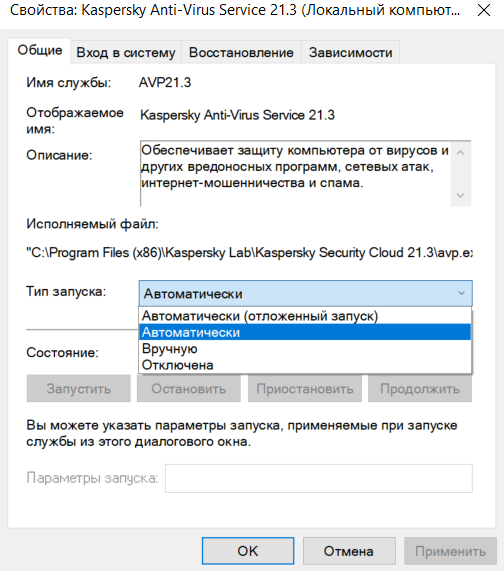
**Кодом:**



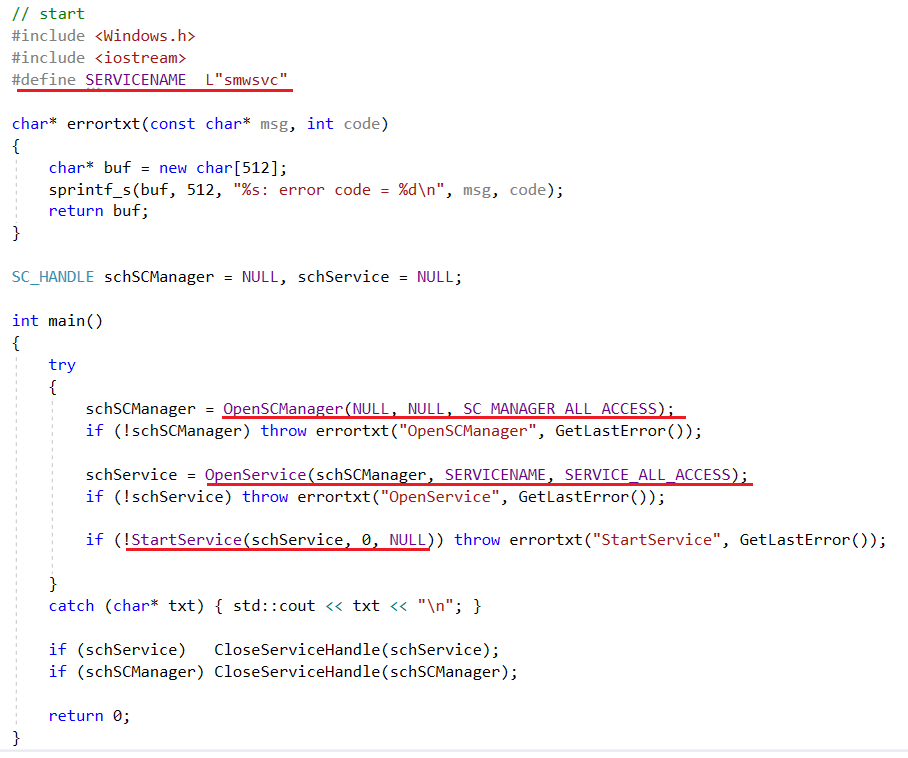
В службах:



**Windows Services:** старт сервиса.



Кодом:

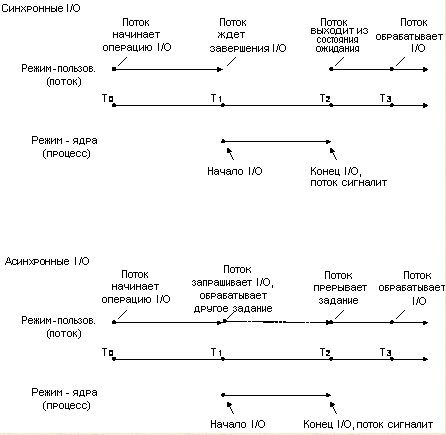


1. Асинхронные операции ввода вывода: понятие асинхронной операции ввода/вывода, особенности программирования асинхронного ввода/вывода.

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

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

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



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

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

Способ 1.

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

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 структуру из пункта 2.

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

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

6.Продвинуть Offset, OffsetHigh.

Способ 2.

1.создать (объявить) структуру \_OVERLAPPED;

2.создать событие Event с автоматическим сбросом;

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

4.WriteFile/Read: lpOverlapped;

5.GetLastError == ERROR\_IO\_PENDING:

6.WaitForSingleObject(hEvent, time);

7.продвинуть Offset, OffsetHigh.

**Асинхронное блокирование файлов**

Функции ***UnlockFile*** и ***UnlockFileEx***

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.

Ответ 2:

Порты завершения ввода/вывода (I/O Completion Ports):

**Назначение и применение:**

I/O Completion Ports (I/OCP) предназначены для эффективной обработки асинхронных операций ввода/вывода в многозадачных приложениях. Они обеспечивают механизм обратного вызова (callback), который срабатывает при завершении асинхронной операции.

**API**:

- В Windows для работы с I/O Completion Ports используется набор API, включая функции `CreateIoCompletionPort`, `GetQueuedCompletionStatus`, `PostQueuedCompletionStatus` и другие.

**Принцип работы:**

**Создание порта завершения ввода/вывода:**

- Используется функция `CreateIoCompletionPort` для создания порта. Этот порт связывается с одним или несколькими устройствами ввода/вывода или с одним или несколькими потоками.

**Асинхронные операции:**

- Операции ввода/вывода выполняются асинхронно с использованием соответствующих функций, таких как `ReadFile`, `WriteFile`, `WSARecv`, `WSASend` и др.

**Ожидание завершения:**

- Поток, связанный с портом завершения, может ожидать завершения асинхронных операций с помощью функции `GetQueuedCompletionStatus`.

**Обработка завершения:**

- Когда асинхронная операция завершается, соответствующий обработчик (callback) вызывается с информацией о завершении операции.

**PostQueuedCompletionStatus:**

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

**Для чего нужен:**

**Эффективность обработки множества операций:**

- I/OCP обеспечивают эффективное управление множеством асинхронных операций ввода/вывода, позволяя эффективно обрабатывать большое количество операций в многозадачных приложениях.

**Масштабируемость:**

- I/OCP обеспечивают масштабируемость за счет эффективного управления потоками и ресурсами ввода/вывода.

**Уменьшение накладных расходов:**

- Использование I/OCP позволяет уменьшить накладные расходы, связанные с ожиданием завершения операций.

**Когда эффективно применять:**

**Сетевое программирование:**

- В сетевых приложениях, особенно при работе с большим количеством соединений.

**Ввод/вывод в файловых операциях:**

- В приложениях, выполняющих множество операций ввода/вывода с файлами.

**Системы обработки событий:**

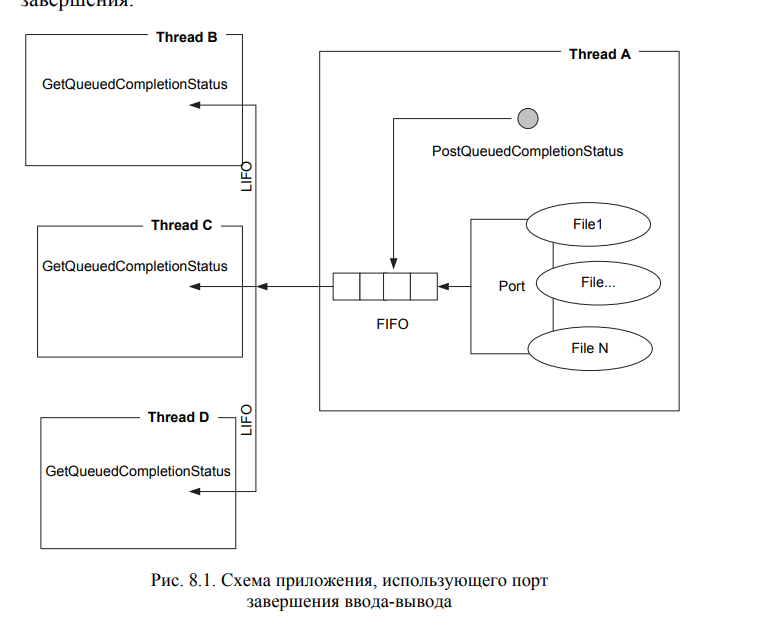
- В системах, где важна обработка асинхронных событий, таких как серверы, обрабатывающие большое количество запросов.

I/O Completion Ports предоставляют мощный и эффективный механизм для работы с асинхронными операциями ввода/вывода в Windows и являются ключевым элементом для создания производительных и отзывчивых многозадачных приложений.

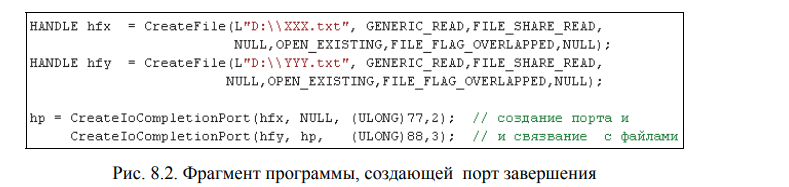
|  |
| --- |
| #include <iostream>#include <Windows.h>#include <vector>  #define MAX\_CONCURRENT\_FILES 3  struct FileContext { HANDLE hFile; OVERLAPPED overlapped; char buffer[1024];};  HANDLE hCompletionPort;  void ReadFileAsync(FileContext\* fileContext) {  ReadFile(fileContext->hFile, fileContext->buffer, sizeof(fileContext->buffer), NULL, &fileContext->overlapped);  }  int main() {  **// Создаем Completion Port**  hCompletionPort = CreateIoCompletionPort(INVALID\_HANDLE\_VALUE, NULL, 0, MAX\_CONCURRENT\_FILES);  if (hCompletionPort == NULL) {  std::cerr << "Error creating completion port: " << GetLastError() << std::endl; return 1; }  **// Открываем несколько файлов**  std::vector<FileContext> fileContexts(MAX\_CONCURRENT\_FILES);    for (int i = 0; i < MAX\_CONCURRENT\_FILES; ++i) {  fileContexts[i].hFile = CreateFile(L"example.txt", GENERIC\_READ, 0, NULL, OPEN\_EXISTING,  FILE\_FLAG\_OVERLAPPED, NULL);  if (fileContexts[i].hFile == INVALID\_HANDLE\_VALUE) {  std::cerr << "Error opening file: " << GetLastError() << std::endl; return 1; }  **// Связываем файл с Completion Port**  if (CreateIoCompletionPort(fileContexts[i].hFile, hCompletionPort, i, 0) == NULL) {  std::cerr << "Error associating file with completion port: " << GetLastError() << std::endl;eturn 1; }  **// Инициализируем асинхронное чтение**  ZeroMemory(&fileContexts[i].overlapped, sizeof(OVERLAPPED));  ReadFileAsync(&fileContexts[i]);  }  **// Ожидаем завершение асинхронных операций**  while (true) {  DWORD bytesRead;  ULONG\_PTR key;  LPOVERLAPPED overlapped;  if (!GetQueuedCompletionStatus(hCompletionPort, &bytesRead, &key, &overlapped, INFINITE)) {  std::cerr << "Error in GetQueuedCompletionStatus: " << GetLastError() << std::endl; return 1; }  **// Обработка завершенной операции**  FileContext\* fileContext = &fileContexts[key];  std::cout << "Read " << bytesRead << " bytes from file." << std::endl;  **// Повторяем асинхронное чтение**  ReadFileAsync(fileContext);  } return 0;} |

Ответ 1:

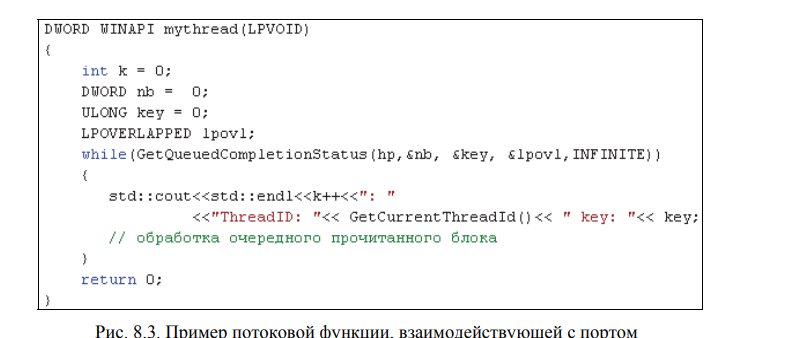
**Порт завершения** – это объект операционной системы, предназначенный для синхронизации работы параллельно работающих потоков с операциями асинхронного ввода-вывода. На рис. 8.1 изображена схема приложения, использующего порт завершения.



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 45 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



Следует обратить внимание на функцию CreateIOComplectionPort, предназначенную для создания порта, первый параметр которой – дескриптор файла, второй – дескриптор порта (используется, если с портом связывается более одного файла), а предпоследний параметр – ключ (идентификатор) файла, который считывается из очереди в рабочих потоках. На рис. 8.3 представлен пример потоковой функции, взаимодействующей с портом ввода-вывода. Функция в цикле вызывает функцию GetQueuedComplectionStatus, считывающую первый элемент очереди. 46 В том случае, если очередь пуста, поток переводится в состояние ожидания до появления элемента, указывающего на завершение операции асинхронного ввода-вывода. Параметры функции позволяют получить значение ключа файла, указатель на структуру OVERLAPPED, количество обработанных (считанных или записаны) байт.



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.

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

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

[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)

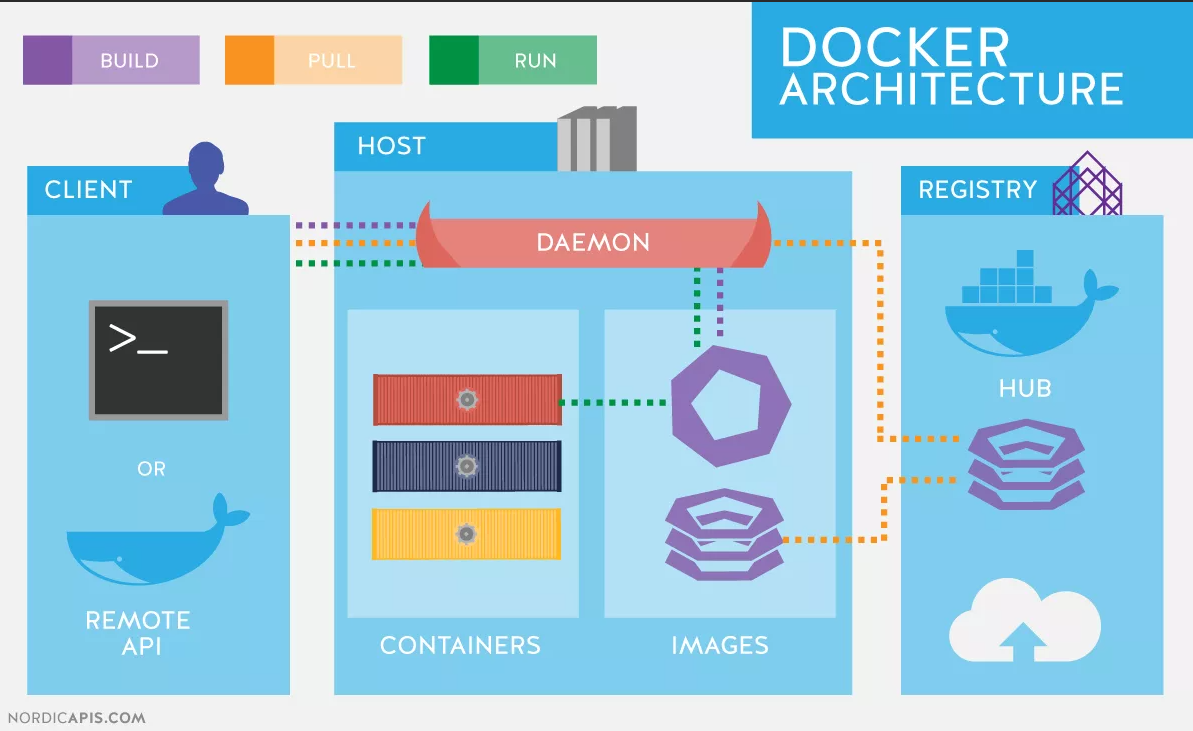
**Архитектура Docker** состоит из нескольких компонентов:

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

**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.

**Контейнер** - это самый маленький элемент в мире Docker. Приложения запускаются внутри контейнера. Это основной компонент Dockera.

**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 могут быть запущены, остановлены, удалены и перезапущены с помощью основных команд Docker.

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

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

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

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

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

**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.