Оглавление

[1. Системное программирование: определение, назначение, применение. 1](#_Toc155736830)

[2. Фреймворк операционной системы: определение, назначение, применение, состав. Стандарт POSIX. 5](#_Toc155736831)

[3. Применение процессов в ОС Windows, API для работы с процессами. 7](#_Toc155736832)

[4. Методы межпроцессного взаимодействия в ОС Windows: обмен данными, синхронизация. 19](#_Toc155736833)

[5. Применение потоков в ОС Windows, API для работы с потоками, API для синхронизации потоков. 23](#_Toc155736834)

[6. Применение механизмов синхронизации в ОС Windows, API для синхронизации. 30](#_Toc155736835)

[7. 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. 38](#_Toc155736836)

[8. Работа с оперативной памятью в ОС Windows: API для работы с виртуальной памятью, API для работы с Heap. 45](#_Toc155736837)

[9. Механизм отображение файлов в памяти: последовательность системных вызовов Windows для создания образа файла в оперативной памяти, использование образа файла, как средства межпроцессного взаимодействия. 50](#_Toc155736838)

[10. Динамически вызываемые библиотеки: структура DLL-библиотеки, экспорт функций, загрузка динамической библиотеки, динамический вызов функций динамической библиотеки, создание и применение библиотеки импорта. 54](#_Toc155736839)

[11. 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. 58](#_Toc155736840)

[12. Управление пользователями и группами пользователей в Windows: понятие дискреционной системы безопасности, типы Windows-пользователей, группы пользователей, возможности API управления пользователями и группами. 65](#_Toc155736841)

[13. 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. 72](#_Toc155736842)

[14. Windows-консоль: определение, применение стандартных потоков для ввода/вывода в консоль, возможности API для управления консолью. 77](#_Toc155736843)

[15. Windows-сервисы: определение, назначение, применение, API. 81](#_Toc155736844)

[16. Асинхронные операции ввода вывода: понятие асинхронной операции ввода/вывода, особенности программирования асинхронного ввода/вывода. 84](#_Toc155736845)

[17. Порты завершения ввода/вывода: назначение, применение, API. 86](#_Toc155736846)

[18. Платформа Docker: архитектура, назначение, принципы устройства, файловая система UFS, контейнеры, образы, основные команды. 90](#_Toc155736847)

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

[Системное программное обеспечение — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)

<https://spravochnick.ru/programmirovanie/sistemnoe_programmirovanie/>

***Смелов***: Прикладное и системное программирование: пользователем прикладного программного обеспечения является конечный пользователь (не обязательно человек), **системное программное обеспечение является вспомогательным средством для разработки прикладного программного обеспечения (библиотеки функций, фреймворки, компиляторы, …).**

***Системное программирование*** — создание системного программного обеспечения; *системный программист* — разработчик системного программного обеспечения.

***Системное ПО*** – совокупность программ, предназначенных для поддержания работоспособности системы обработки информации и управления (компьютера и компьютерных сетей) или повышения эффективности ее использования.

**Систе́мное программное обеспечение** — программы, решающие задачи общевычислительного характера — вы­де­ле­ния и разделения ресурсов, доступа к устройствам, обеспечивающие среды для разработки, запуска и выполнения других программ.

Два подхода есть системного программирования:

1. Разработка на уровне API ОС

Этот подход заключается в создании программных приложений, используя функции и сервисы, предоставляемые API операционной системы. API - это интерфейс, который определяет, как программы могут взаимодействовать с операционной системой и другими приложениями. Программисты, использующие этот подход, напрямую работают с функциями и библиотеками, предоставляемыми операционной системой, чтобы создавать приложения.

2. Разработка приложений для других программистов

Этот подход включает в себя создание инструментов, библиотек, фреймворков и других ресурсов, которые другие программисты могут использовать для упрощения своей работы.

Этот подход означает создание приложений или библиотек, которые предоставляют удобные и высокоуровневые интерфейсы для других программистов. Вместо того, чтобы напрямую работать с API операционной системы, программисты создают абстракции и интерфейсы для облегчения задачи другим разработчикам.

ОС -

В отличие от [прикладного программного обеспечения](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), системное не решает конкретные практические задачи, а лишь обеспечивает работу других программ, предоставляя им сервисные функции, управляет аппаратными ресурсами вычислительной системы.

Основные признаки, отличающие системное программирование от прикладного:

* **системное программное обеспечение напрямую использует системные вызовы операционной системы;**
* пользователями разработанного системного программного обеспечения являются другие программисты;
* разработанное системное программное обеспечение многократно используется в различных прикладных приложениях.

Обычно системное программное обеспечение оформляется в виде библиотек функций, которые можно подключать к создаваемым прикладным программам.

Как правило, к системному программному обеспечению относятся операционные системы, утилиты, драйверы, системы программирования, системы управления базами данных, программы, предназначенные для разработки программного обеспечения, широкий класс [связующего программного обеспечения](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D1%83%D1%8E%D1%89%D0%B5%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5).

------------------------------------------------------------------------

К этой категории относятся системные программы, предназначенные для разработки программного обеспечения:

* [ассемблеры](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80" \o "Ассемблер) — компьютерные программы, осуществляющие преобразование программы в форме исходного текста на [языке ассемблера](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0) в [машинные команды](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) в виде [объектного кода](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4);
* [трансляторы](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80) — программы или технические средства, выполняющее трансляцию программы;
  + [компиляторы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) — Программы, переводящие текст программы на языке высокого уровня, в эквивалентную программу на машинном языке.
  + [интерпретаторы](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80) — Программы (иногда аппаратные средства), анализирующие команды или операторы программы и тут же выполняющие их;
* [компоновщики](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA) (редакторы связей) — программы, которые производят компоновку — принимают на вход один или несколько объектных модулей и собирают по ним исполнимый модуль;
* [препроцессоры](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) исходных текстов — это компьютерные программы, принимающие данные на входе, и выдающие данные, предназначенные для входа другой программы, например, такой, как компилятор;
* отла́дчики ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *debugger*) — модули среды разработки или отдельные программы, предназначенные для поиска ошибок в программах;
* [текстовые редакторы](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%80%D0%B5%D0%B4%D0%B0%D0%BA%D1%82%D0%BE%D1%80) — компьютерные программы, предназначенные для создания и изменения текстовых файлов, а также их просмотра на экране, вывода на печать, поиска фрагментов текста и т. п.;
  + специализированные [редакторы исходных текстов](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B4%D0%B0%D0%BA%D1%82%D0%BE%D1%80_%D0%B8%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D1%85_%D1%82%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2) — текстовые редакторы для создания и редактирования исходного кода программ. Специализированный редактор исходных текстов может быть отдельным приложением, или быть встроен в [интегрированную среду разработки](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8);
* [библиотеки подпрограмм](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) — сборники подпрограмм или объектов, используемых для разработки программного обеспечения;
* [редакторы графического интерфейса](https://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B5%D0%B4%D0%B0%D0%BA%D1%82%D0%BE%D1%80_%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D0%BA%D0%BE%D0%B3%D0%BE_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D0%B0&action=edit&redlink=1).

-----------------------------------------------------------------------

**Систе́мный вы́зов** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *system call*) — обращение [прикладной программы](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) к [ядру](https://ru.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) для выполнения какой-либо операции.

**Системный вызов (System Call)** в операционных системах представляет собой интерфейс между пользовательским приложением и ядром опера-ционной системы. Этот механизм позволяет приложениям взаимодейство-вать с ядром для выполнения привилегированных операций, таких как чте-ние/запись в файлы, создание процессов, управление памятью и другие. При-меры системных вызовов могут включать: Открытие файла Чтение/запись дан-ных в файл и создание процессов.

Для выполнения межпроцессной операции или операции, требующей доступа к оборудованию, программа обращается к ядру, которое, в зависимости от полномочий вызывающего процесса, исполняет либо отказывает в исполнении такого вызова.

Как правило, система предоставляет библиотеку или [API](https://ru.wikipedia.org/wiki/API), которые находятся между обычным приложением и ОС. Такая библиотека предоставляет программисту удобный интерфейс для работы с ОС в виде интерфейсных функций. Интерфейсные функции библиотеки предоставляют обычные соглашения о вызове функций для использования системных вызовов и делают системные вызовы более унифицированными. Обращение к функции такой библиотеки само по себе не вызывает переключения в режим ядра и является обычным вызовом подпрограммы.

------------------------------------------------------------------------

Системные вызовы могут быть сгруппированы в пять больших категорий:

1. Управление процессами
   * load
   * execute
   * end (exit), abort
   * создание процесса ([fork](https://ru.wikipedia.org/w/index.php?title=Fork_(operating_system)&action=edit&redlink=1) в Unix-like, NtCreateProcess в Windows\_NT [Native API](https://ru.wikipedia.org/wiki/Native_API))
   * завершение процесса
   * get/set process attributes
   * [wait](https://en.wikipedia.org/wiki/Wait_(operating_system)) время, события, [signal](https://en.wikipedia.org/wiki/Signal_(computing)) события
   * [allocate](https://en.wikipedia.org/wiki/Dynamic_memory_allocation), [free](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) memory
2. Работа с файлами
   * create file, delete file
   * open, close
   * read, write, reposition
   * get/set file attributes
3. Управление устройствами
   * request device, release device
   * read, write, reposition
   * get/set device attributes
   * logically attach or detach devices
4. Работа с информацией
   * get/set time or date
   * get/set system data
   * get/set process, file, or device attributes
5. Связь, коммуникация
   * create, delete communication connection
   * send, receive messages
   * transfer status information
   * attach or detach remote devices

------------------------------------------------------------------------

Что такое аппаратное прерывание, программное прерывание?

**Аппаратное** - возникает как реакция микропроцессоров на физический сигнал от некоторого устройства (клавиатура, системные часы, жесткий диск и т.д.), по времени возникновения эти прерывания асинхронны, т.е. происходят в случайные моменты времени

**Программное** - вызываются искусственно с помощью соответствующей команды из программы, предназначены для выполнения некоторых действий операционной системы, являются синхронными;

**Исключения:** Ошибки, возникающие в процессе выполнения программы, такие как деление на ноль или обращение к недопустимой области памяти, могут также привести к системным прерываниям.

**В Windows API нет прямого способа вызвать системное прерывание, как это делается в DOS с использованием функции int86()**. Это связано с тем, что современные операционные системы, включая Windows, работают в защищенном режиме, который изолирует приложения на уровне пользователя от низкоуровневого доступа к аппаратному обеспечению и системным ресурсам[1](https://www.cyberforum.ru/win-api/thread1096758.html)[2](https://ru.stackoverflow.com/questions/620907/bios-%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D1%8B%D0%B5-%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F-int-11h-int-12h-%D0%B2-%D0%A1-c).

Однако, если вам нужно взаимодействовать с аппаратным обеспечением или системными ресурсами, вы можете использовать различные функции и механизмы, предоставляемые Windows API. Например, для работы с портами ввода/вывода можно использовать функции ReadFile() и WriteFile() с дескриптором файла, открытым для соответствующего порта

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

***Фреймворк OS (Operating System Framework) –*** это набор программных компонентов, библиотек и инструментов, предназначенных для разработки операционных систем или создания приложений, работающих на операционных системах.

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

*То есть фреймворк ОС это набор библиотек которые предоставляют АПИ для работы с ОС. (АПИ это же интерфейс и фреймворк ОС как раз таки реализует этот интерфейс). А АПИ представляет собой функции через которые мы можем осуществлять системные вызовы, и там создавать потоки, процессы, файлы и тд*

[**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(API определяет стандартизированные способы взаимодействия приложений с операционной системой. Фреймворк операционной системы предоставляет реализацию этих стандартов, обеспечивая единый интерфейс для взаимодействия с ОС. API служит уровнем абстракции, который скрывает детали реализации операционной системы от разработчиков приложений. Разработчиками операционной системы Windows была создана библиотека функций, при помощи которых и происходит взаимодействие приложения с операционной системой, так называемые функции Программного интерфейса приложений (Application Program Interface, 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).

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.

[POSIX — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/POSIX)

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.

[**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).

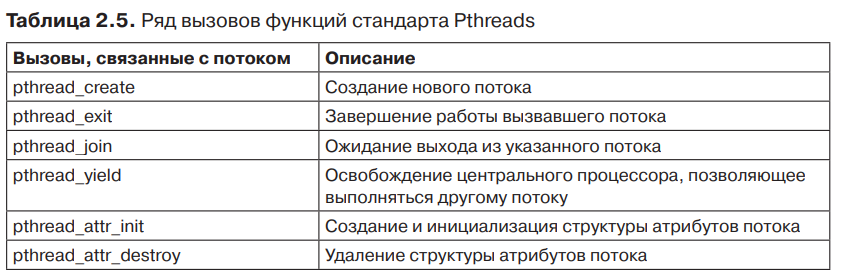
[POSIX также предназначен для использования как разработчиками приложений, так и системными разработчиками](https://en.wikipedia.org/wiki/POSIX).

**Стандарт состоит из четырёх основных разделов.**

* **Основные определения** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Base definitions*) — список основных определений и соглашений, используемых в спецификациях, и список заголовочных файлов [языка Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), которые должны быть предоставлены соответствующей стандарту системой.
* **Оболочка и утилиты** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Shell and utilities*) — описание утилит и командной оболочки [sh](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%B0%D1%8F_%D0%BE%D0%B1%D0%BE%D0%BB%D0%BE%D1%87%D0%BA%D0%B0_UNIX), стандарты [регулярных выражений](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F).
* **Системные интерфейсы** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *System interfaces*) — список [системных вызовов](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) языка [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)).
* **Обоснование** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Rationale*) — объяснение принципов, используемых в стандарте.

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/).



1. [**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).
2. [**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).
3. [**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).

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

Определение

**единица работы OS,** экземпляр выполняемой программы

под процессом понимается объект ядра, которому принадлежат системные ресурсы, используемые исполняемым приложением.

Каждый процесс в операционной системе Windows владеет следующими ресурсами:

виртуальным адресным пространством;

рабочим множеством страниц в реальной памяти;

маркером доступа, содержащим информацию для системы безопасности;

таблицей для хранения дескрипторов объектов ядра

И имеет идентификатор

Свойства процесса:

* процессу соответствует исполняемый программный файл;
* у процесса есть PID;
* у процесса есть Parent PID;
* у процесса есть приоритет
* 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;
* **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;

**OS**: **процесс OS** – единица работы OS - **объект ядра OS+адресное пространство**:

* создается ядром OS по системному вызову;
* адресное пространство (данные, программа, стек, куча);
* ресурсы: регистры, открытые дескрипторы (файлы), родительский процесс, перечень связанных (дочерних) процессов, реальные страницы памяти, виртуальное адресное пространство, маркеры доступа (безопасность)– содержит информацию по безопасности сеанса и идентифицирует пользователя, группу пользователей и пользовательские привилегии.;
* процесс может создавать (с помощью системного вызова) дочерние процессы, в общем случае может образовываться дерево процессов;
* обычно соответствует работающей программе (например, exe-файлу в Windows);
* OS хранит список (или таблицу) объектов работающих процессов Этот список содержит информацию о каждом активном процессе, включая его идентификатор процесса (PID), текущее состояние (например, выполняется, ожидает, приостановлен), приоритет, использование памяти, ссылки на открытые файлы и другие ресурсы, а также другую информацию, необходимую для управления процессом;
* при приостановке процесса в объекте процесса сохраняется вся информация (регистры, уведомления OS,… - контекст процесса), позволяющая возобновить работу процесса;
* процессу автоматически доступны три потока: ввода, вывода, вывод ошибок
* процессы изолированы друг от друга, что означает, что они не могут напрямую взаимодействовать друг с другом.;
* для обмена данными между процессами, применяется специальный механизм – механизм межпроцессного взаимодействия (IPC);

**OS**: линейное адресное пространство, станицы памяти, виртуальная память, сегменты памяти процесса.



**Применение процессов:**

**1. Исполнение программ:**

* Каждая запущенная программа представлена процессом. Процесс включает в себя код программы, данные, стек вызовов и другую информацию, необходимую для выполнения программы.

**2. Управление ресурсами:**

* Процессы обеспечивают изоляцию и управление ресурсами. Каждый процесс имеет свое собственное виртуальное адресное пространство, что позволяет избежать конфликтов между данными различных программ.
* Windows использует механизмы планирования процессов для эффективного использования процессорного времени и управления приоритетами выполнения.

**3. Многозадачность:**

* Windows поддерживает многозадачность, что означает, что одновременно могут выполняться несколько процессов. Каждый процесс может содержать несколько потоков выполнения, что позволяет эффективно использовать многозадачность на многоядерных системах.

**4. Системные службы и фоновые процессы:**

* Множество системных служб и фоновых процессов, таких как антивирусные программы, обновления системы, службы печати и другие, выполняются в виде отдельных процессов в фоновом режиме.

**5. Безопасность:**

* Каждый процесс имеет свои собственные права доступа, что способствует обеспечению безопасности системы. Разделение привилегий помогает предотвратить нежелательные вмешательства программ и защищает от несанкционированного доступа.

**Многопоточность**

***1. Приоритетная многопоточность* (Preemptive Multithreading):**

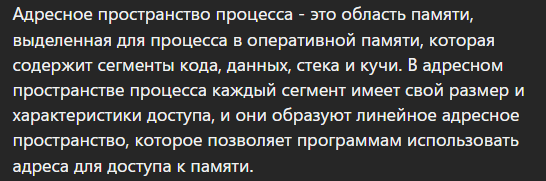
потоки могут быть принудительно приостановлены операционной системой (планировщиком) в любой момент, чтобы предоставить возможность выполнения другим потокам с более высоким приоритетом.

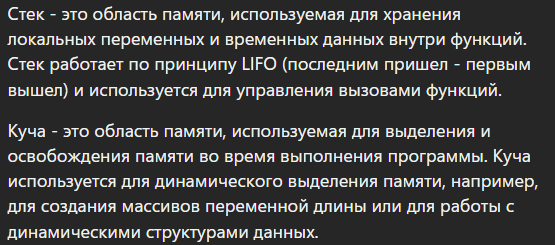
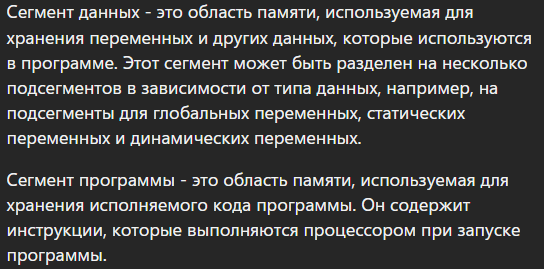
Операционные системы, такие как Windows и большинство десктопных Linux-систем, используют приоритетную многозадачность.

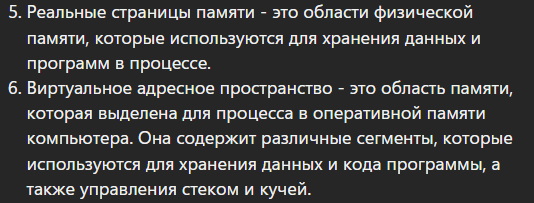
***2. Кооперативная многопоточность* (Cooperative Multithreading):**

Здесь потоки сами решают, когда они готовы освободить процессор и передать управление другим потокам. Операционная система не вмешивается в переключение контекста между потоками, и каждый поток должен самостоятельно учесть интересы других потоков.

Этот подход часто используется в системах реального времени и в некоторых мобильных операционных системах.







Контекст процесса

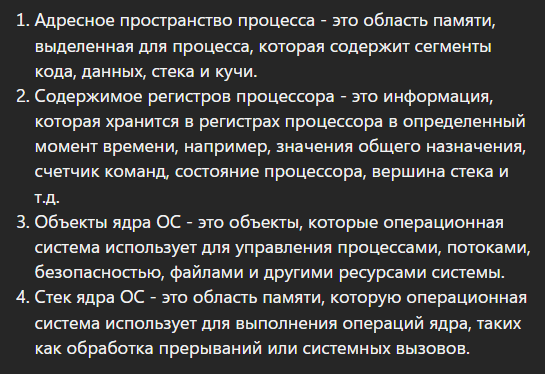
**Контекст процесса** - это набор данных, которые операционная система хранит о процессе в определенный момент времени. Эти данные включают в себя:

***адресное пространство процесса***,

***содержимое регистров процессора*** (например, регистры общего назначения, счетчик команд, состояние процессора, вершину стека и т.д.),

***объекты ядра ОС*** (такие как объекты процессов, потоков, безопасности, файлов и т.д.)

***стек ядра ОС***, который используется для выполнения операций ядра.



При переключении контекстов, ядро ОС сохраняет текущий контекст процесса в его объекте процесса (сохранение происходит в структуре данных, называемой "контекстом" или "контекстом выполнения".) и загружает контекст другого процесса для выполнения. Этот процесс называется **переключением контекстов.**

-- не понимаю про стек ядра

В ядре ОС есть специальный стек для переключения контекстов, который называется стеком ядра. Он используется для хранения информации о текущем контексте процесса и других объектов ядра ОС. При переключении контекстов, ядро ОС сохраняет текущую информацию о процессе на стеке ядра и загружает информацию о следующем процессе для выполнения.

--

Переключение контекстов является важной функцией операционной системы, так как позволяет ей эффективно управлять ресурсами системы и обеспечивать выполнение нескольких задач одновременно. Благодаря переключению контекстов, каждый процесс получает свою долю процессорного времени, что позволяет им работать параллельно и обеспечивает более эффективное использование ресурсов системы.

**OS: стандартные потоки ввода/вывода процесса:**

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

Идентификатор процесса

Когда процесс создается, операционная система присваивает ему уникальный PID и добавляет процесс в таблицу процессов операционной системы.

**PID** обычно представляет собой неотрицательное целое число. PID используется для идентификации процесса в операционной системе, для межпроцессного взаимодействия, для обнаружения и устранения проблем, связанных с процессами. Например, если процесс завис или перестал отвечать, можно использовать его PID для завершения процесса.

**OS**: **системные** **процессы:**

процессы запускаемые автоматически при запуске OS; Windows: windows-сервисы; Linux-демоны.

**Создание процесса в Windows CreateProcess**

Новый процесс в Windows создается вызовом функции CreateProcess, которая имеет следующий прототип:

BOOL CreateProcess(

LPCTSTR lpApplicationName, // имя исполняемого модуля

LPTSTR lpCommandLine, // командная строка

LPSECURITY\_ATTRIBUTES lpProcessAttributes, // защита процесса

LPSECURITY\_ATTRIBUTES lpThreadAttributes, // защита потока

BOOL bInheritHandles, // признак наследования дескриптора

DWORD dwCreationFlags, // флаги создания процесса

LPVOID lpEnvironment, // блок новой среды окружения

LPCTSTR lpCurrentDirectory, // текущий каталог

LPSTARTUPINFO lpStartUpInfo, // вид главного окна

LPPROCESS\_INFORMATION lpProcessInformation // информация о процессе

);

[**https://learn.microsoft.com/ru-ru/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa**](https://learn.microsoft.com/ru-ru/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa)

Функция **CreateProcess** создает новый процесс и первичный поток в этом процессе. Функция CreateProcess возвращает ненулевое значение, если процесс был создан успешно. В противном случае эта функция возвращает значение FALSE.

Параметры:

1. lpApplicationName определяет строку с именем исполняемого файла, который имеет тип exe и будет запускаться при создании нового процесса. Если эта строка равна nullptr, то имя исполняемого файла берется из lpCommandLine.

2. lpCommandLine (LPTSTR): Командная строка, которая передается процессу. Этот параметр может содержать имя исполняемого файла и его параметры.

3. lpProcessAttributes (LPSECURITY\_ATTRIBUTES): Указатели на структуры **SECURITY\_ATTRIBUTES**, которые определяют атрибуты безопасности для процесса (дескриптор безопасности), которая контролирует доступ к созданному процессу. Если NULL, процесс наследует дескриптор безопасности от родительского процесса.

4. lpThreadAttributes (LPSECURITY\_ATTRIBUTES): Защита потока (дескриптор безопасности), которая контролирует доступ к первичному потоку в созданном процессе. Если NULL, поток наследует дескриптор безопасности от родительского процесса.

5. bInheritHandles (BOOL): Признак наследования дескриптора. Если TRUE, процесс наследует дескрипторы от родительского процесса.

6. dwCreationFlags (DWORD): Флаги создания процесса. Определяют различные параметры создания процесса, такие как создание новой консоли, использование отдельной среды окружения и т. д.

*CREATE\_SUSPENDED* — указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функция ResumeThread.

*DETACHED\_PROCESS и CREATE\_NEW\_CONSOLE* — взаимоисключающие значения, которые не должны устанавливаться оба одновременно. Первый флаг означает создание нового процесса, у которого консоль отсутствует, а второй — процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса.

7. lpEnvironment (LPVOID): Блок новой среды окружения для создаваемого процесса. для передачи переменных среды новому процессу. Если NULL, процесс наследует среду окружения от родительского процесса.

8. lpCurrentDirectory (LPCTSTR): Текущий каталог (указатель на строку с путем) для создаваемого процесса. Если NULL, процесс использует текущий каталог родительского процесса.

9. lpStartUpInfo (LPSTARTUPINFO): Указатель на структуру **STARTUPINFO**, которая определяет атрибуты главного окна и потока нового процесса.

10. lpProcessInformation (LPPROCESS\_INFORMATION): указатель на структуру, в которую будут помещены возвращаемые функцией значения дескрипторов и глобальных идентификаторов процесса и потока. Структура PROCESS\_INFORMATION, о которой идет речь, имеет следующий вид:

typedef struct PROCESS\_INFORMATION {

HANDLE hProcess;

HANDLE hThread;

DWORD dwProcessId;

DWORD dwThreadId;

} PROCESS\_INFORMATION;

------------------------------------------------------------------------

Зачем процессам и потокам нужны еще и дескрипторы, если они снабжаются глобальными идентификаторами (ID)? Глобальные идентификаторы остаются уникальными для данного объекта на протяжении всего времени его существования и во всех процессах, тогда дескрипторов процесса может быть несколько и каждый из которых может характеризоваться собственным набором атрибутов, например определенными разрешениями доступа. В силу указанных причин одним функциям управления процессами требуется предоставлять идентификаторы процессов, а другим — дескрипторы. Кроме того, необходимость в дескрипторах процессов возникает при использовании универсальных функций, которые требуют указания дескрипторов. В качестве примера можно привести функции ожидания, обсуждаемые далее в этой главе, которые обеспечивают отслеживание переходов объектов различного типа, в том числе и процессов, указываемых с помощью дескрипторов, в определенные состояния. Точно так же, как и дескрипторы файлов, дескрипторы процессов и потоков должны закрываться сразу же после того, как необходимость в них отпала.

------------------------------------------------------------------------

Завершение процесса

Когда компилятор откомпилирует заданную ему программу, он осуществляет системный вызов, сообщающий операционной системе о завершении своей работы. Этим вызовом в UNIX является **exit**, а в Windows — **ExitProcess**

Процесс может завершить свою работу вызовом функции ExitProcess, которая имеет следующий прототип:

VOID ExitProcess( UINT uExitCode // код возврата из процесса );

При вызове функции ExitProcess завершаются все потоки процесса с кодом возврата, который является параметром этой функции. При выполнении этой функции система посылает динамическим библиотекам, которые загружены процессом, сообщение DLL\_PROCESS\_DETACH, которое говорит о том, что динамическую библиотеку необходимо отсоединить от процесса.

Один процесс может быть завершен другим при помощи вызова функции

**TerminateProcess**, которая имеет следующий прототип:

BOOL TerminateProcess(

HANDLE hProcess, // дескриптор процесса

UINT uExitCode // код возврата

);

Если функция TerminateProcess выполнилась успешно, то она возвращает ненулевое значение. В противном случае возвращаемое значение равно FALSE. Функция TerminateProcess завершает работу процесса, но не освобождает все ресурсы, принадлежащие этому процессу. Это происходит потому, что при выполнении этой функции система не посылает динамическим библиотекам, загруженным процессом, сообщение о том, что библиотеку необходимо отсоединить от процесса.

------------------------------------------------------------------------

В UNIX существует только один системный вызов для создания нового процесса — fork. Этот вызов создает точную копию вызывающего процесса. После выполнения системного вызова fork два процесса, родительский и дочерний, имеют единый образ памяти, единые строки описания конфигурации и одни и те же открытые файлы. И больше ничего

------------------------------------------------------------------------

Состояния процессов

три состояния, в которых может находиться процесс:

* выполняемый (в данный момент использующий центральный процессор);
* готовый (работоспособный, но временно приостановленный, чтобы дать возможность выполнения другому процессу);
* заблокированный (неспособный выполняться, пока не возникнет какое-нибудь внешнее событие).



Утилиты

1. Linux, /proc, перечень процессов

Функции процессов

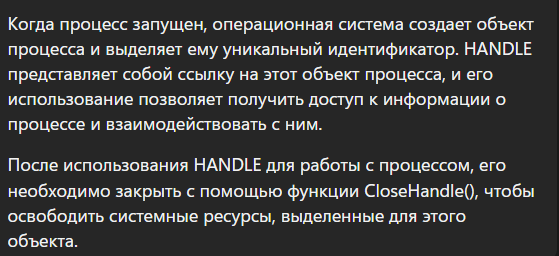
<https://learn.microsoft.com/ru-ru/windows/win32/procthread/process-and-thread-functions#process-functions>

BOOL GetProcessHandleCount( HANDLE hProcess, PDWORD pdwHandleCount)– Счетчики дескрипторов процессов

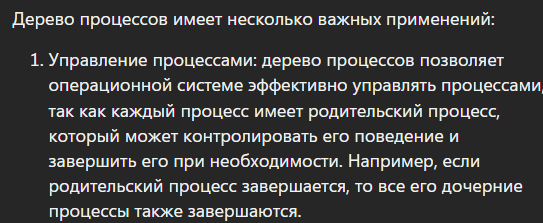
HANDLE Windows-процесса

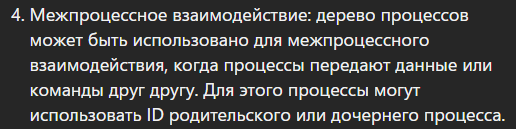
(СМЕЛОВ)**HANDLE WIndows-процесса** - 32-разрядное число, которое фактически отображает номер объекта ядра ОС.

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

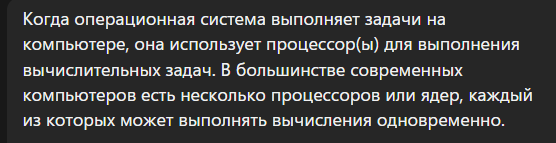


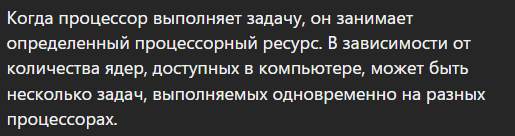
**Дерево процессов** - это **иерархическая структура**, которая представляет все запущенные в операционной системе процессы и связи между ними. Каждый процесс может иметь одного или несколько дочерних процессов, и таким образом, формируется дерево процессов.

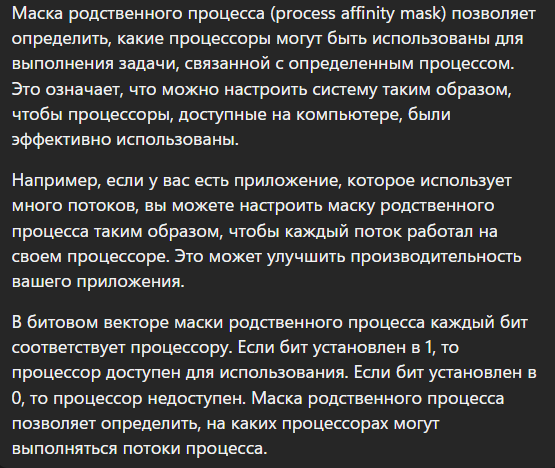




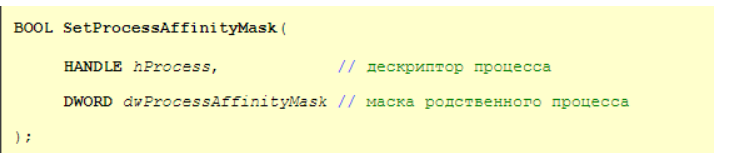
Назначение процессоров процессам







Маской родственного процесса (process affinity mask) называется **битовый вектор**, в котором **каждый бит обозначает процессор, на котором потокам процесса позволяется запускаться**.



* **Фоновые процессы** выполняют свою работу, когда нет активных пользовательских процессов. Обычно эти процессы следят за состоянием системы. Приоритет таких процессов устанавливается флагом IDLE\_PRIORITY\_CLASS.
* **Процессы с нормальным приоритетом** — это обычные пользовательские процессы. Приоритет таких процессов устанавливается флагом NORMAL\_PRIORITY\_CLASS. Этот приоритет также назначается пользовательским процессам по умолчанию. В Windows 2000 приоритет обычных пользовательских процессов может также устанавливаться флагами BELOW\_NORMAL\_PRIORITY\_CLASS или ABOVE\_NORMAL\_PRIORITY\_CLASS, которые соответственно немного повышают или понижают приоритет пользовательского процесса.
* **Процессы с высоким приоритетом** это такие пользовательские процессы, от которых требуется более быстрая реакция на некоторые события, чем от обычных пользовательских процессов. Приоритет таких процессов устанавливается флагом HIGH\_PRIORITY\_CLASS. Эти процессы должны содержать небольшой программный код и выполняться очень быстро, чтобы не замедлять работу системы. Обычно такие приоритеты имеют другие системы, работающие на платформе операционных систем Windows.
* **К последнему типу процессов относятся процессы реального времени**. Приоритет таких процессов устанавливается флагом REAL\_TIME\_PRIORITY\_CLASS. Работа таких процессов обычно происходит в масштабе реального времени и связана с реакцией на внешние события. Эти процессы должны работать непосредственно с аппаратурой компьютера.

**Большинство процессов завершаются по окончании своей работы. Когда компилятор откомпилирует заданную ему программу, он осуществляет системный вызов, сообщающий операционной системе о завершении своей работы. Этим вызовом в UNIX является exit, а в Windows — ExitProcess.**

*Приоритет процесса можно изменить при помощи функции SetPriorityClass*

*Узнать приоритет процесса можно посредством вызова функции GetPriorityClass*

*Иногда процессу требуется знать свой дескриптор, чтобы изменить какие-то свои характеристики. Например, процесс может изменить свой приоритет. Для этих целей в Win32 API существует функция GetCurrentProcess*

*Один процесс может быть завершен другим при помощи вызова функции TerminateProcess*

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

**Методы межпроцессного взаимодействия (МПВ) в ОС Windows — это способы обмена данными и согласования действий между процессами, работающими на одном или разных компьютерах. МПВ позволяют решать задачи параллельного и распределенного программирования, такие как координация, совместное использование ресурсов, распределение нагрузки и т.д. МПВ в ОС Windows можно классифицировать по следующим критериям:**

**Обмен данными — это способ передачи информации между процессами. Обмен данными может быть синхронным или асинхронным, в зависимости от того, ждут ли процессы подтверждения получения данных или нет. Обмен данными также может быть однонаправленным или двунаправленным, в зависимости от того, могут ли процессы отправлять и получать данные или только одно из этого. Наконец, обмен данными может быть точка-точка или широковещательный, в зависимости от того, сколько процессов участвует в коммуникации**

Методы обмена данными позволяют передавать информацию между процессами, которые могут работать на одном или разных компьютерах. К ним относятся:

* **Буфер обмена** — центральное хранилище для обмена данными между приложениями, которые поддерживают операции вырезания, копирования и вставки.
* **Сопоставление файлов** — механизм, который позволяет нескольким процессам совместно использовать один и тот же файл или часть файла, отображая его в свои адресные пространства.
* **Mailslots** — механизм, который позволяет отправлять и получать сообщения между процессами, работающими на одном или разных компьютерах в сети. Mailslots поддерживают только однонаправленную связь и не гарантируют доставку сообщений.

-------------------------------------------------------------------------------------------------

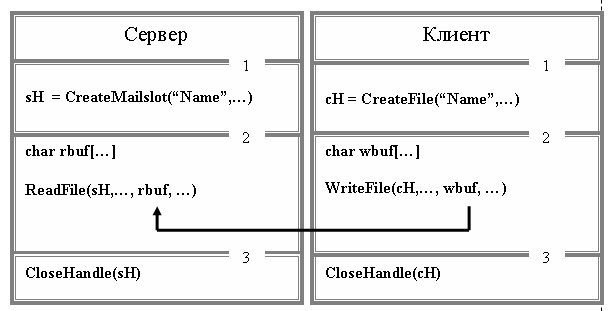
Механизм Mailslots (почтовый ящик) является одним из IPC-механизмов операционной системы Windows, позволяющий распределенные приложения архитектуры клиент-сервер в локальной сети TCP/IP.

**Почтовый ящик** (Mailslot) – это объект ядра операционной системы, который обеспечивает передачу данных от процессов-клиентов к процессам-серверам, выполняющимся на компьютерах в одной локальной сети.

Процесс, создающий почтовый ящик называется **сервером** почтового ящика. Процессы, которые связываются с почтовым ящиком, называются **клиентами** почтового ящика.

Каждый почтовый ящик имеет имя, которое определяется сервером при создании и используется клиентами для доступа. **Передача может осуществляться только сообщениями и в одном направлении – от клиента к серверу.** Обмен данными может происходить в синхронном и асинхронном режимах. Допускается создание нескольких серверов с одинаковым именем почтового ящика – в этом случае все отправляемые клиентом сообщения будут поступать во все почтовые ящики, имеющие имя, указанное клиентом. Однако, следует сказать, что такая рассылка сообщений возможна только в том случае, когда длина оправляемых сообщений не превышает 425 байт.

В том случае, если клиент отправляет сообщение размером меньше, чем 425 байт, то пересылка осуществляется без гарантии доставки. Пересылка сообщения размером более 425 байт возможна только от одного клиента к одному серверу.



1. Создание почтового ящика сервером. 2. Соединение клиента с почтовым ящиком. 3. Обмен данными через почтовый ящик. 4. Закрытие почтового ящика клиентом и сервером.

--------------------------------------------------------------------------------------------------

* **Сокеты Windows** — механизм, который позволяет устанавливать двустороннюю связь между процессами, работающими на одном или разных компьютерах в сети. Сокеты Windows поддерживают различные протоколы транспортного уровня, такие как TCP и UDP.

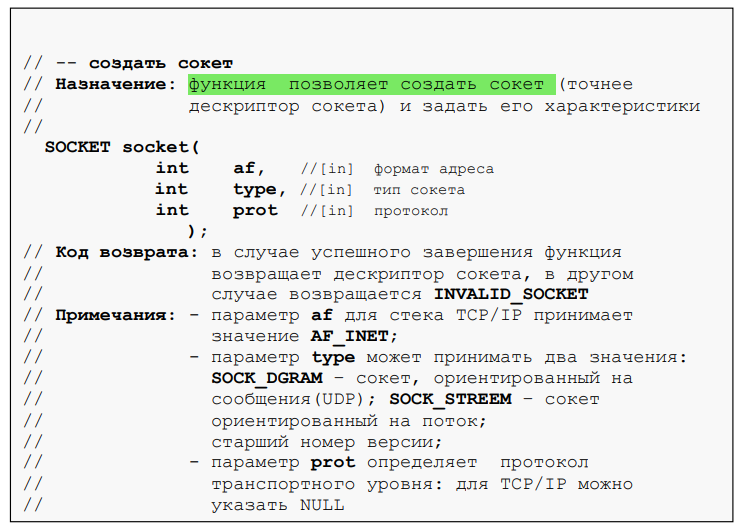
--------------------------------------------------------------------------------------------------

Сокет – совокупность IP-адреса и номера порта; используется для идентификации прикладного процесса в сети.

API сокетов – это название программного интерфейса для обмена данными между процессами, находящимися на одном или на разных объединенных сетью компьютерах.

Следует помнить, что одни и те же номера портов могут быть использованы как для протокола UDP, так и для протокола TCP.

В операционной системе Windows интерфейс сокетов имеет название **Windows Sockets API**. API сокетов включает в себя функции создания сокета (имеется в виду объект ОС, описывающий соединение), установки параметров сокета (сетевой адрес, номер порта и т.д.), функции создания канала и обмена данными между сокетами. Кроме того, есть набор функций, позволяющий управлять передачей данных, синхронизировать процессы передачи и приема данных, обрабатывать ошибки и т.п.



--------------------------------------------------------------------------------------------------

* **Именованные каналы** — механизм, который позволяет устанавливать двустороннюю или одностороннюю связь между процессами, работающими на одном или разных компьютерах в сети. Именованные каналы поддерживают синхронный и асинхронный режимы работы, а также механизмы безопасности и сжатия данных.

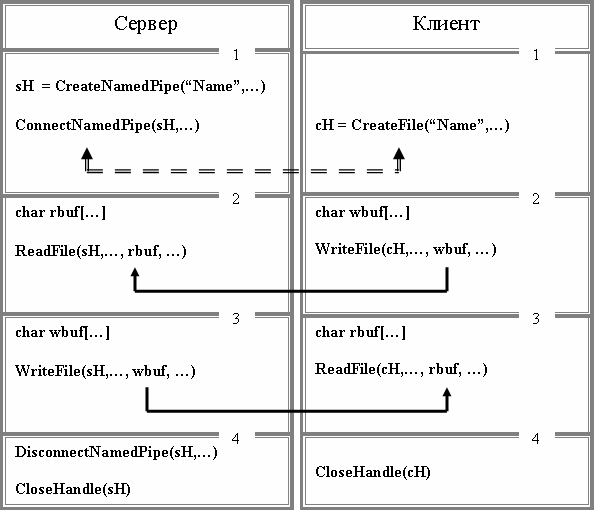
--------------------------------------------------------------------------------------------------

**Именованный канал** – это объект ядра операционной системы, который обеспечивает обмен данными между процессами, выполняющимися на компьютерах в одной локальной сети.

Процесс, создающий именованный канал, называется сервером именованного канала. Процессы, которые связываются с именованным каналом, называются клиентами именованного канала. Любой именованный канал идентифицируется своим именем, которое задается при создании канала.

Именованные каналы бывают: **дуплексные** (позволяющие передавать данные в обе стороны) и **полудуплексные** (позволяющие передавать данные только в одну сторону). Передача данных в именованном канале может осуществляться как **потоком**, так и **сообщениями**. **Обмен данными в канале может быть синхронным и асинхронным.**

Все функции Named Pipe API можно разбить на три группы: функции управления каналом (создать канал, соединить сервер с каналом, открыть канал, получить информацию об именованном канале, получить состояние канала, изменить характеристики канала); функции обмена данными (писать в канал, читать из канала, копировать данные канала) и функции для работы с транзакциями.



Теперь приведем порядок работы с именованными каналами:

создание именованного канала сервером:

соединение сервера с экземпляром именованного канала;

соединение клиента с экземпляром именованного

обмен данными по именованному каналу;

отсоединение сервера от экземпляра именованного канала;

закрытие именованного канала клиентом и сервером.

--------------------------------------------------------------------------------------------------

* **Очереди сообщений** — механизм, который позволяет отправлять и получать сообщения между процессами, работающими на одном или разных компьютерах в сети. Очереди сообщений поддерживают гарантированную доставку сообщений, приоритеты, транзакции и уведомления.

# 5. Применение потоков в ОС Windows, API для работы с потоками, API для синхронизации потоков.

1. **OS**: **поток (управления) OS** – объект ядра операционной системы, которому OS выделяет процессорное время. Наименьшая единица работы ядра OS.
2. **OS**: **поток (управления) OS** –последовательность инструкций, выполняемых процессором в выделенные OS интервалы времени.

**Системные потоки** (потоки ядра ОС) – выполняют различные сервисы ОС и запускаются ядром ОС, используются для реализации пользовательских потоков.

**Пользовательские потоки** – потоки, служащие для решения задач пользователя, и запускаемые приложением.

1. **Основные свойства потока:**

* поток – это объект OS;
* поток – средство диспетчеризации доступа к процессорному времени (квант примерно 20мс);
* поток – последовательность команд процессора;
* поток – наименьшая единица работы ядра OS;
* процесс – контейнер для потоков;
* процесс имеет как минимум один поток (main);
* создание потока осуществляется с помощью системного вызова;
* потоки в рамках одного процесса не изолированы, все ресурсы кроме процессорного времени – общие;
* для работы с потоками в OS есть специальный API;
* каждый поток имеет свой идентификатор;
* состояния потока: исполняется, готов к исполнению, блокирован, спит; приостановлен;
* код потока – потоковая функция (специфицирована в OS);
* диспетчеризация потоков осуществляется OS или самим потоком;
* контекст потока – данные необходимые для возобновления работы потока при его приостановке (диспетчеризация, синхронизация): программный код, набор регистров, стек памяти, оперативная память, стек ядра, маркер доступа);
* поток может создавать дочерние потоки и их завершать;
* потоки могут создавать дерево потоков;
* завершение родительского потока приводит к завершению всех его дочерних (требуется ожидание дочернего завершение потока);
* многопоточность – парадигма программирования, поддерживается OS, приоритетная и корпоративная многопоточность;
* API в разных OS отличается; Linux IEEE POSIX, NPTL (Native POSIX Thread Library),
* потокобезопасность кода (программы) **–** свойство программного кода (программы) корректно работать в нескольких потоках одновременно;
* реентерабельность кода (программы)– свойство одной копии программного кода работать в нескольких потоках одновременно;
* фибра – механизм для ручного планирования выполнения кода в рамках потока.

многопоточность

*1. Приоритетная многопоточность* (Preemptive Multithreading):

потоки могут быть принудительно приостановлены операционной системой (планировщиком) в любой момент, чтобы предоставить возможность выполнения другим потокам с более высоким приоритетом.

Операционные системы, такие как Windows и большинство десктопных Linux-систем, используют приоритетную многозадачность.

*2. Кооперативная многопоточность* (Cooperative Multithreading):

Здесь потоки сами решают, когда они готовы освободить процессор и передать управление другим потокам. Операционная система не вмешивается в переключение контекста между потоками, и каждый поток должен самостоятельно учесть интересы других потоков.

Этот подход часто используется в системах реального времени и в некоторых мобильных операционных системах.

***Диспетчеризация*** — это процесс переключения процессора с одного потока на другой соответственно плану. Работает по принципу FIFO, причем каждый процесс получает ограниченное процессорное время.

Диспетчеризация сводится к следующему:

* сохранение контекста текущего потока, который требуется сменить;
* загрузка контекста нового потока, выбранного в результате планирования;
* запуск нового потока на выполнение

------------------------------------------------------------------------

Бурда про многопоточность

------------------------------------------------------------------------

Применение потоков

Основная причина использования потоков заключается в том, что во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной. Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в квазипараллельном режиме.

*возможность использования параллельными процессами единого адресного пространства и всех имеющихся данных*. Эта возможность играет весьма важную роль для тех приложений, которым не подходит использование нескольких процессов (с их раздельными адресными пространствами).

Вторым аргументом в пользу потоков является ***легкость (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами.*** Во многих системах создание потоков осуществляется в 10–100 раз быстрее, чем создание процессов. Это свойство особенно пригодится, когда потребуется быстро и динамично изменять количество потоков

Третий аргумент в пользу потоков также касается производительности. Когда потоки работают в рамках одного центрального процессора, они не приносят никакого прироста производительности, но когда выполняются значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения

Тененбаум страницв 124

**В каждый момент времени поток может находиться в одном из шести состояний:**

* **Ready (готов) — поставлен в очередь и ожидает выполнения;**
* **Standby (ближайший) — готов быть выполненным следующим;**
* **Running (выполнение) — находится в режиме выполнения и взаимодействует с центральным процессором;**
* **Waiting (ожидание) — не выполняется, ожидая сигнала выполнения;**
* **Transition (промежуточное) — будет выполняться после того, как система загрузит его контекст;**
* **Terminated (завершен) — выполнение завершено, однако объект не удален.**

**Возможные приоритеты у потоков:**

**Приоритет процесса** – это числовое значение, определяющее важность и срочность выполнения данного процесса

базовый приоритет потока - это числовое значение, которое определяет относительную важность потока в контексте планирования выполнения процессов и потоков.

Это приоритет потока, основанный на приоритете процесса и относительном приоритете потока. Он складывается на основании класса приоритета, породившего этот поток процесса и относительного класса приоритета потока. Диапазон изменения от 1 до 31 включительно ([1;31]), вычисляется по специальной таблице (таблицу надеюсь на листик переписывать по памяти не нужно будет...).

по умолчанию – 8



* **THREAD\_PRIORITY\_LOWEST — низший приоритет;**
* **THREAD\_PRIORITY\_BELOW\_NORMAL — приоритет ниже нормального;**
* **THREAD\_PRIORITY\_NORMAL — нормальный приоритет;**
* **THREAD\_PRIORITY\_ABOVE\_NORMAL — приоритет выше нормального;**
* **THREAD\_PRIORITY\_HIGHEST — высший приоритет.**

**Для создания потоков, выполняющихся в адресном пространстве\**

Создается поток функцией CreateThread, которая имеет следующий прототип:

HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES lpThreadAttributes, // атрибуты защиты DWORD dwStackSize, // размер стека потока в байтах LPTHREAD\_START\_ROUTINE lpStartAddress, // адрес функции

LPVOID lpParameter, // адрес параметра

DWORD dwCreationFlags, // флаги создания потока

LPDWORD pThreadId // идентификатор потока

);

При успешном завершении функция CreateThread возвращает дескриптор созданного потока и его идентификатор, который является уникальным для всей системы. В противном случае эта функция возвращает значение NULL

Параметр dwStackSize определяет размер стека, который выделяется потоку при запуске. Если этот параметр равен нулю, то потоку выделяется стек, размер которого по умолчанию равен 1 Мбайт. Это наименьший размер стека, который может быть выделен потоку. Если величина параметра dwStackSize меньше значения, заданного по умолчанию, то все равно потоку выделяется стек размером в 1 Мбайт. Операционная система Windows округляет размер стека до одной страницы памяти, который обычно равен 4 Кбайт.

Параметр lpStartAddress указывает на исполняемую потоком функцию. Эта функция должна иметь следующий прототип:

DWORD WINAPI имя\_функции\_потока(LPVOID lpParameters);

Параметр dwCreationFlags определяет, в каком состоянии будет создан поток. Если значение этого параметра равно 0, то функция потока начинает выполняться сразу после создания потока. Если же значение этого параметра равно **CREATE\_SUSPENDED**, то поток создается в подвешенном состоянии. В дальнейшем этот поток можно запустить вызовом функции ResumeThread.

Параметр lpThreadId является выходным, т. е. его значение устанавливает Windows. Этот параметр должен указывать на переменную, в которую Windows поместит идентификатор потока. Этот идентификатор уникален для всей системы и может в дальнейшем использоваться для ссылок на поток. Идентификатор потока главным образом используется системными функциями и редко функциями приложения. Действителен идентификатор потока только на время существования потока. После завершения потока тот же идентификатор может быть присвоен другому потоку

**Завершение потока**

Поток завершается вызовом функции ExitThread, которая имеет следующий прототип:

VOID ExitThread(

DWORD dwExitCode // код завершения потока

);

Эта функция может вызываться как явно, так и неявно при возврате значения из функции потока. При выполнении этой функции система посылает динамическим библиотекам, которые загружены процессом, сообщение DLL\_THREAD\_DETACH, которое говорит о том, что поток завершает свою работу.

Один поток может завершить другой поток, вызвав функцию TerminateThread, которая имеет следующий прототип:

BOOL TerminateThread(

HANDLE hThread, // дескриптор потока

DWORD dwExitThread // код завершения потока

);

Функция TerminateThread завершает поток, но не освобождает все ресурсы, принадлежащие этому потоку. Это происходит потому, что при выполнении этой функции система не посылает динамическим библиотекам, загруженным процессом, сообщение о том, что поток завершает свою работу. В результате динамическая библиотека не освобождает ресурсы, которые были захвачены для работы с этим потоком. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании потока.

**Приостановка потока**

Каждый созданный поток имеет счетчик приостановок, максимальное значение которого равно MAXIMUM\_SUSPEND\_COUNT. Счетчик приостановок показывает, сколько раз исполнение потока было приостановлено. Поток может исполняться только при условии, что значение счетчика приостановок равно нулю. В противном случае поток не исполняется или, как говорят, находится в подвешенном состоянии.

Исполнение каждого потока может быть приостановлено вызовом функции SuspendThread, которая имеет следующий прототип:

DWORD SuspendThread(

HANDLE hThread // дескриптор потока

);

Эта функция увеличивает значение счетчика приостановок на 1 и, при успешном завершении, возвращает текущее значение этого счетчика.

В случае неудачи функция SuspendThread возвращает значение, равное –1.

Отметим, что поток может приостановить также и сам себя. Для этого он должен передать функции SuspendThread свой псевдодескриптор, который можно получить при помощи функции GetCurrentThread.

Для возобновления исполнения потока используется функция ResumeThread, которая имеет следующий прототип:

DWORD ResumeThread(

HANDLE hThread // дескриптор потока

);

Приостановка

Поток может задержать свое исполнение вызовом функции Sleep, которая имеет следующий прототип:

VOID Sleep(

DWORD dwMilliseconds // миллисекунды

);

Единственный параметр функции Sleep определяет количество миллисекунд, на которые поток, вызвавший эту функцию, приостанавливает свое исполнение. Если значение этого параметра равно 0, то выполнение потока просто прерывается, а затем возобновляется при условии, что нет других потоков, ждущих выделения процессорного времени.

Псевдодескриптор текущего потока отличается от настоящего дескриптора потока тем, что он может использоваться только самим текущим потоком и, следовательно, может наследоваться другими процессами. Псевдодескриптор потока не нужно закрывать после его использования. Из псевдодескриптора потока можно получить настоящий дескриптор потока, для этого псевдодескриптор нужно продублировать, вызвав функцию DuplicateHandle

GetCurrentThread – получить псевдодескриптор текущего потока

*синхронизация ниже*

# 6. Применение механизмов синхронизации в ОС Windows, API для синхронизации.

**Смелов: синхронизация – механизм упорядочивания выполнения программных блоков двух или более потоков.**

**Синхронизация потоков –** это процесс координации выполнения нескольких потоков в многозадачной или многопоточной среде с целью обеспечения правильного порядка доступа к общим ресурсам.

**Синхронизация потоков –** механизм упорядочивания выполнения программных блоков двух или более потоков, позволяющий потокам согласовывать свою работу с общими ресурсами.

Объясните понятие «взаимная блокировка».

Взаимная блокировка (deadlock) – это ситуация, при которой два или более потока или процесса блокируются в ожидании ресурсов, удерживаемых другими потоками или процессами. Это приводит к тому, что они ожидают друг друга бесконечно, не выполняя никаких действий, что приводит к затруднению выполнения программы.

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

--------------------------------------------------------------

1. **Что такое потокобезопасность программного кода?**

Свойство программного кода (программы) корректно работать в нескольких потоках одновременно. Для обеспечения потокобезопасности кода часто используются механизмы синхронизации,

Потокобезопасность относится к возможности множества потоков (thread) безопасно работать с общими данными и ресурсами.

1. **Что такое реентерабельность кода?**

**Реентерабельность** — свойство одной копии программного кода работать в нескольких потоках одновременно. Реентерабельный код всегда потокобезопасен. Реентерабельный код не использует статическую память и не изменяет сам себя, все данные сохраняются в динамической памяти.

Реентерабельность относится к способности функции или участка кода выполняться безопасно и корректно при вызове из разных потоков (или рекурсивно) в одно и то же время, и каждый вызов будет работать независимо и безопасно, не влияя на состояние других вызовов

даже без явной синхронизации.

--------------------------------------------------------------

Методы синхронизации позволяют координировать работу процессов, которые совместно используют ресурсы, такие как память, файлы, устройства и т.д. К ним относятся:

* **Семафоры** — механизм, который позволяет ограничивать количество процессов или потоков, которые могут одновременно получить доступ к определенному ресурсу или выполнять определенную задачу.

---------------------

семафоры поддерживают счетчики, и когда значение этого счетчика больше 0, объект семафора находится в сигнальном состоянии. Если же значение счетчика становится нулевым, объект семафора переходит в несигнальное состояние

Создаются семафоры посредством вызова функции CreateSemaphore, которая имеет следующий прототип:

HANDLE CreateSemaphore(

LPSECURITY\_ATTRIBUTES lpSemaphoreAttribute, // атрибуты защиты

LONG lInitialCount, // начальное значение семафора

LONG lMaximumCount, // максимальное значение семафора

LPCTSTR lpName // имя семафора

);

Значение параметра lInitialCount устанавливает начальное значение семафора, которое должно быть не меньше 0 и не больше его максимального значения, которое устанавливается параметром lMaximumCount.

Параметр lpName может указывать на имя семафора или содержать значение NULL. В последнем случае создается безымянный семафор.

Если семафор с заданным именем уже существует, то функция CreateSemaphor возвращает дескриптор этого семафора, а функция GetLastError, вызванная после функции CreateSemaphor, вернет значение ERROR\_ALREADY\_EXISTS

Значение семафора уменьшается на 1 при его использовании в функции ожидания. Увеличить значение семафора можно посредством вызова функции ReleaseSemaphore, которая имеет следующий прототип:

BOOL ReleaseSemaphore(

HANDLE hSemaphore, // дескриптор семафора

LONG lReleaseCount, // положительное число, на которое

// увеличивается значение семафора

LPLONG lpPreviousCount // предыдущее значение семафора

);

Доступ к существующему семафору можно открыть с помощью одной из функций CreateSemaphore или OpenSemaphore. Если для этой цели используется функция CreateSemaphore, то значения параметров lInitialCount и lMaximalCount этой функции игнорируются, т. к. они уже установлены другим потоком, а поток, вызвавший эту функцию, получает полный доступ к семафору с именем, заданным параметром lpName.

Теперь рассмотрим функцию OpenSemaphore, которая используется в случае, если известно, что семафор с заданным именем уже существует. Эта функция имеет следующий прототип:

HANDLE OpenSemaphore(

DWORD dwDesiredAccess, // флаги доступа

BOOL bInheritHandle, // режим наследования

LPCTSTR lpName // имя семафора

);

* **Мьютексы** — механизм, который позволяет обеспечивать взаимное исключение между процессами или потоками, которые конкурируют за доступ к одному ресурсу или выполняют одну задачу.

--------------------------

Мьютекс находится в сигнальном состоянии, если он не принадлежит ни одному потоку. В противном случае мьютекс находится в несигнальном состоянии. Одновременно мьютекс может принадлежать только одному потоку. Потоки, ждущие сигнального состояния мьютекса, обслуживаются в порядке FIFO

Создается мьютекс вызовом функции CreateMutex, которая имеет следующий прототип:

HANDLE CreateMutex(

LPSECURITY\_ATTRIBUTES lpMutexAttributes, // атрибуты защиты

BOOL bInitialOwner, // начальный владелец мьютекса

LPCTSTR lpName // имя мьютекса

);

Если значение параметра bInitialOwner равно TRUE, то мьютекс сразу переходит во владение потоку, которым он был создан. В противном случае вновь созданный мьютекс свободен. Поток, создавший мьютекс, имеет все права доступа к этому мьютексу.

*Мьютекс захватывается потоком посредством любой функции ожидания, а освобождается функцией ReleaseMutex,* которая имеет следующий прототип:

BOOL ReleaseMutex (

HANDLE hMutex // дескриптор мьютекса

);

Для доступа к существующему мьютексу поток может использовать одну из функций CreateMutex или OpenMutex.

Функция CreateMutex используется в тех случаях, когда поток не знает, создан или нет мьютекс с указанным именем другим потоком. В этом случае значение параметра bInitialOwner нужно установить в FALSE, т. к. невозможно определить какой из потоков создает мьютекс. Если поток использует для доступа к уже созданному мьютексу функцию CreateMutex, то он получает полный доступ к этому мьютексу.

Для того чтобы получить доступ к уже созданному мьютексу поток может также использовать функцию OpenMutex, которая имеет следующий прототип:

HANDLE OpenMutex (

DWORD dwDesiredAccess, // доступ к мьютексу

BOOL bInheritHandle // свойство наследования

LPCTSTR lpName // имя мьютекса

);

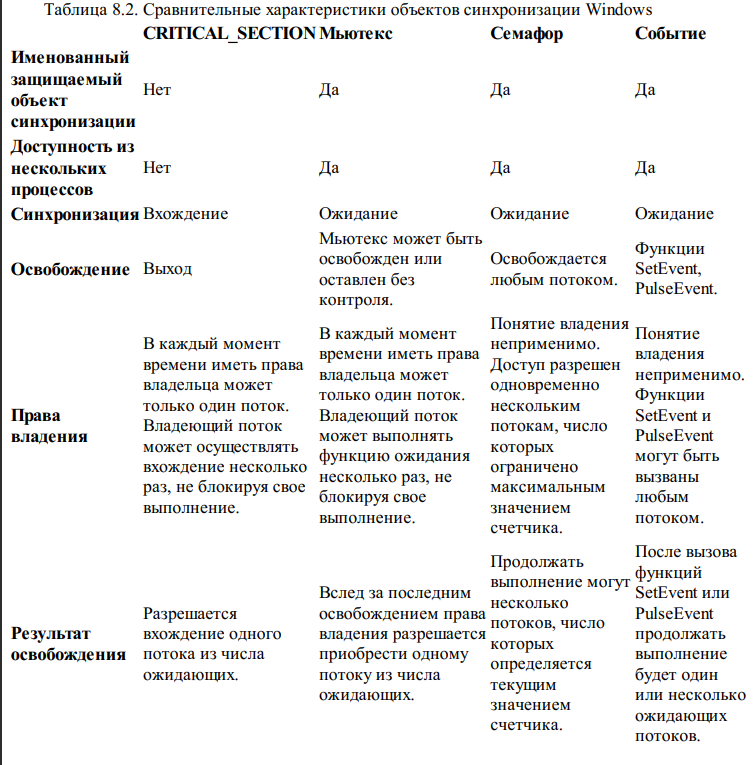
Параметр dwDesiredAccess этой функции может принимать одно из двух значений:

MUTEX\_ALL\_ACCESS — полный доступ

SYNCHRONIZE — синхронизация.

В первом случае поток получает полный доступ к мьютексу. Во втором слу

чае поток может использовать мьютекс только в функциях ожидания, чтобы захватить мьютекс, или в функции ReleaseMutex для его освобождения.

* **Критические секции** — механизм, который позволяет обеспечивать взаимное исключение между потоками одного процесса, которые конкурируют за доступ к одному ресурсу или выполняют одну задачу.
* **События** — механизм, который позволяет сигнализировать о наступлении определенных условий между процессами или потоками, а также ожидать этих условий. \
* **Последним из рассматриваемых нами типов объектов синхронизации ядра являются события (events). Объекты события используются для того, чтобы сигнализировать другим потокам о наступлении какого-либо события, например, о появлении нового сообщения. Важной дополнительной возможностью, обеспечиваемой объектами событий, является то, что переход в сигнальное состояние единственного объекта события способен вывести из состояния ожидания одновременно несколько потоков. Объекты события делятся на сбрасываемые вручную и автоматически сбрасываемые, и это их свойство устанавливается при вызове функции CreateEvent.**
* **• Сбрасываемые вручную события (manual-reset events) могут сигнализировать одновременно всем потокам, ожидающим наступления этого события, и переводятся в несигнальное состояние программно.**
* **• Автоматически сбрасываемые события (auto-reset event) сбрасываются самостоятельно после освобождения одного из ожидающих потоков, тогда как другие ожидающие потоки продолжают ожидать перехода события в сигнальное состояние.**
* **События используют пять новых функций: CreateEvent, OpenEvent, SetEvent, ResetEvent и PulseEvent.** Для открытия именованного объекта события используется функция **OpenEvent,** причем это может сделать и другой процесс. **Поток может установить событие в сигнальное состояние, используя функцию SetEvent. Если событие является автоматически сбрасываемым, то оно автоматически возвращается в несигнальное состояние уже после освобождения только одного из ожидающих потоков. В отсутствие потоков, ожидающих наступления этого события, оно остается в сигнальном состоянии до тех пор, пока такой поток не появится, после чего этот поток сразу же освобождается.**
* **С другой стороны, если событие является сбрасываемым вручную, то оно остается в сигнальном состоянии до тех пор, пока какой-либо поток не вызовет функцию ResetEvent, указав дескриптор этого события в качестве аргумента. В это время все ожидающие потоки освобождаются, но до выполнения такого сброса события другие потоки могут как переходить в состояние его ожидания, так и освобождаться. Функция PulseEvent освобождает все потоки, ожидающие наступления сбрасываемого вручную события, но после этого событие сразу же автоматически сбрасывается. В случае же использования автоматически сбрасываемого события функция PulseEvent освобождает только один ожидающий поток, если таковые имеются.**
* 
* **Таймеры** — механизм, который позволяет задавать отложенные или периодические действия между процессами или потоками, а также ожидать их выполнения.
* **Функции ожидания**

Функции ожидания в Windows это такие функции, параметрами которых являются объекты синхронизации. Эти функции обычно используются для блокировки потоков. Сама блокировка потока выполняется следующим образом. Если дескриптор объекта синхронизации является параметром функции ожидания, а сам объект синхронизации находится в несигнальном состоянии, то *поток, вызвавший эту функцию ожидания, блокируется до перехода этого объекта синхронизации в сигнальное состояние.*

Сейчас мы будем использовать только две функции ожидания WaitForSingleObject и WaitForMultipleObject.

Для ожидания перехода в сигнальное состояние одного объекта синхронизации используется функция WaitForSingleObject, которая имеет следующий прототип:

DWORD WaitForSingleObject(

HANDLE hHandle, // дескриптор объекта

DWORD dwMilliseconds // интервал ожидания в миллисекундах

);

Функция WaitForSingleObject в течение интервала времени, равного значению параметра dwMilliseconds, ждет перехода объекта синхронизации, дескриптор которого задается параметром hHandle, в сигнальное состояние.

Если значение параметра dwMilliseconds равно нулю, то функция только проверяет состояние объекта синхронизации. Если же значение параметра dwMilliseconds равно INFINITE, то функция ждет перехода объекта синхронизации в сигнальное состояние бесконечно долго.

В случае успешного завершения функция WaitForSingleObject возвращает одно из следующих значений:

WAIT\_OBJECT\_0 — объект перешел в сигнальное состояние;

WAIT\_ABANDONED — забытый мьютекс;

WAIT\_TIMEOUT — время ожидания истекло.

Значение WAIT\_OBJECT\_0 означает, что объект синхронизации находился или перешел в сигнальное состояние.

Значение WAIT\_ABANDONED означает, что объектом синхронизации являлся мьютекс, который не освободился завершившимся потоком. В этом случае мьютекс освобождается операционной системой и поэтому также переходит в сигнальное состояние. Такой мьютекс иногда называется забытый или заброшенный мьютекс (abandoned mutex).

Значение WAIT\_TIMEOUT означает, что время ожидания истекло, а объект синхронизации так и не перешел в сигнальное состояние.

В случае неудачи функция WaitForSingleObject возвращает значение WAIT\_FAILED.

------------------------------------------------------------------------

Mutex – это механизм синхронизации, который позволяет только одному потоку или процессу получить доступ к защищаемому ресурсу в определенное время. Он обеспечивает эксклюзивный доступ к ресурсу.

Semaphore – это механизм синхронизации, который позволяет ограничить количество потоков или процессов, имеющих доступ к общему ресурсу. Семафоры могут быть установлены на значение больше 1, что позволяет нескольким потокам получить доступ к ресурсу одновременно.

Мьютексы — это тоже объекты режима ядра, используемые для синхронизации, но они проще семафоров, поскольку не имеют счетчиков ( мьютекс объекта может захватить одновременно только один поток). В основе семафора лежит счётчик, над которым можно производить две [атомарные операции](https://ru.wikipedia.org/wiki/%D0%90%D1%82%D0%BE%D0%BC%D0%B0%D1%80%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F): увеличение и уменьшение значения на единицу. мьютекс может захватить одновременно только один поток, а семафор – несколько

Почему **mutex,** **semaphore, event** создают объект ядра OS, а **critical section** нет.

Critical section – механизм синхронизации нескольких потоков **одного** процесса, НЕ ЯВЛЯЕТСЯ объектом ядра OS

- Mutex, semaphore, event: Эти объекты создаются в ядре операционной системы, потому что они предоставляют межпроцессорную синхронизацию и должны быть видимыми для всех потоков в системе.

- Critical Section – это участок кода, который должен быть выполнен только одним потоком в определенное время. Критическая секция **не создает** объект ядра операционной системы, потому что она работает на уровне процесса и зависит от программной реализации среды выполнения. В отличие от мьютексов или семафоров, критическая секция предоставляет более легковесный и быстрый механизм синхронизации, но не обеспечивает такой же уровень защиты от состояний гонки, как мьютексы или семафоры.

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

**каталОг** – файл содержащий информацию о месте расположения других файлов;

--- Каталог - это структура для хранения файлов и других каталогов. Каталоги позволяют организовать файлы в иерархическую структуру, что облегчает их управление и поиск.

**OS: буферы ввода/вывода**: области памяти для хранения физически считанных данных; необходимы для устранения несоответствия между физическим и логическим чтением/записью. Сначала заполняется буфер кластерами(физически считанными данными), а затем осуществляется чтение логических файлов.

**---** [Буферы ввода-вывода используются для временного хранения данных при вводе или выводе1](https://bing.com/search?q=). [Они служат областью памяти, которая обеспечивает обмен данными между внешними устройствами и процессами в пределах компьютера1](https://bing.com/search?q=).

1. **Логическая организация данных:**
   * **Файлы и Каталоги (Директории):** Логическая организация начинается с файлов и каталогов. Файл — это набор данных, обычно сгруппированных вместе, имеющих имя и ассоциированных с определенным типом данных. Каталог — это структура для группировки файлов и других каталогов в иерархическую структуру.
   * **Иерархическая структура:** Файловая система обычно организована в виде иерархической структуры, где корневой каталог содержит подкаталоги и файлы, каждый из которых может содержать свои подкаталоги и файлы. Это обеспечивает логическую организацию данных в виде древовидной структуры.
   * **Путь к файлу:** Каждый файл в файловой системе может быть доступен по определенному пути, который указывает на его местоположение в иерархии каталогов. Например, **C:\Users\Username\Documents\File.txt**.
   * **Атрибуты файлов:** Логическая организация также включает в себя атрибуты файлов, такие как права доступа, время создания и последнего доступа, размер файла и др.
2. **Физическая организация данных:**
   * **Блоки и Сектора:** Файлы в физическом смысле хранятся на диске в виде блоков данных. Файловая система взаимодействует с устройствами хранения (например, жесткие диски, SSD) через блоки данных, которые, в свою очередь, могут быть организованы в сектора.
   * **Кластеризация:** Файлы могут занимать несколько физических блоков, и этот процесс называется кластеризацией. Он позволяет эффективнее использовать пространство на диске, но может привести к фрагментации.
   * **Файловые системы:** Физическая организация также зависит от используемой файловой системы. Например, файловые системы FAT, NTFS, ext4 и другие имеют различные методы организации данных и алгоритмы управления пространством на диске.
   * **Блоки управления файловой системой (File System Control Blocks):** Файловая система поддерживает структуры данных, такие как блоки управления файловой системой, которые содержат информацию о каждом файле, включая указатели на блоки данных, атрибуты и дополнительные метаданные.
3. **Кэширование и Буферизация:**
   * Файловые системы часто используют кэширование и буферизацию для оптимизации доступа к данным. Кэш содержит копии часто используемых данных в оперативной памяти, уменьшая время доступа к файлам.

**--------------------------------------------------------------------------------------------------------**

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

Она включает в себя следующие аспекты:

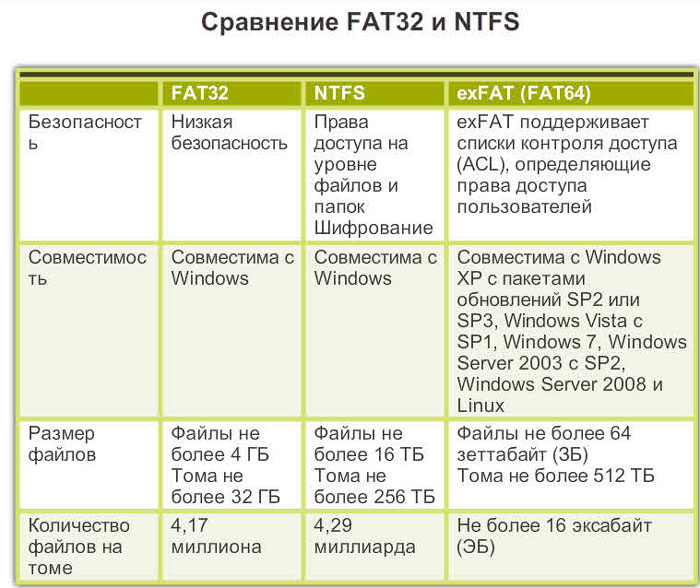
* **Файлы и директории**: Файлы представляют собой логические единицы данных, которые могут содержать текст, изображения, звуки и другие типы информации. Директории служат для организации файлов в иерархической структуре. Логические пути и имена файлов и директорий позволяют пользователям и программам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



**--------------------------------------------------------------------------------------------------------**

**Жесткие диски также называются жесткими магнитными дисками. Жесткие магнитные диски закреплены на стержне, который вращается с большой скоростью. Данные записываются на поверхностях жестких магнитных дисков с помощью магнитных головок, которые расположены над каждой дисковой поверхностью. Магнитной головке доступны только те данные, которые находятся на участке дисковой поверхности под или над ней. Все магнитные головки закреплены на одном стержне, который, совершая вращательные движения, перемещает головки по радиусам магнитных дисков в обоих направлениях.**

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

**Сектором называется наименьшая область (дуга) одной дорожки магнитного диска, которая может быть записана или считана магнитной головкой диска за его один полный поворот. Размер сектора равен 512 байт. Обычно плотность записи данных на диск является одинаковой для всех дорожек. Поэтому дорожки, находящиеся ближе к центру диска, содержат меньше секторов, чем дорожки, находящиеся ближе к краю диска. Одна дорожка жесткого диска может содержать от 380 до 700 секторов. Секторы каждой дорожки перенумерованы, причем эта нумерация начинается с 1. В начале каждого сектора хранится заголовок или префикс, который определяет начало и номер сектора. В конце каждого сектора хранится заключение или суффикс, который содержит контрольную сумму, необходимую для проверки целостности данных. Не так давно нумерация секторов одной дорожки не являлась последовательной. Секторы нумеровались через один сектор. Это было сделано для того, чтобы контроллер дискового устройства при доступе к жесткому магнитному диску не пропускал секторы при вращении.**

**В настоящее время скорость работы контроллеров жестких дисков сравнима со скоростью вращения диска, поэтому секторы каждой дорожки нумеруются последовательно. Так как дорожки содержат разное количество секторов, то скорость обмена данными с накопителем на жестких магнитных дисках зависит от номера дорожки. Это вызвано тем, что при постоянной угловой скорости вращения стержня с дисками линейная скорость перемещения секторов относительно головки различается для разных дорожек.**

**Эта скорость тем выше, чем дальше дорожка находится от центра диска. Поэтому диск разбивают на зоны. Одна зона содержит несколько цилиндров диска и на каждой из дорожек, входящих в одну зону, находится одинаковое количество секторов. Поэтому доступ к секторам одной зоны выполняется с одинаковой скоростью.**

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

**Файловая система — это порядок, определяющий способ организации, хранения и именования данных на носителях информации в компьютерах и другом электронном оборудовании. Файловая система определяет формат содержимого и способ физического хранения информации, которую принято группировать в виде файлов**

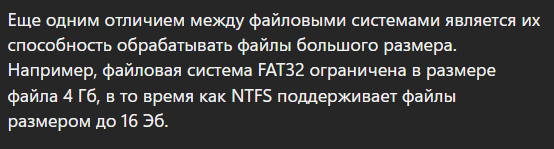
**Windows поддерживает на непосредственно подключенных устройствах файловые системы нескольких типов(тут не все, но те что ему надо):**

**1. Файловая система NT (NTFS) — современная файловая система, которая поддерживает длинные имена файлов, а также безопасность, устойчивость к сбоям, шифрование, сжатие, расширенные атрибуты, и позволяет работать с очень большими файлами и объемами данных. Заметьте, что на гибких дисках (флоппи-дисках, или дискетах) система NTFS использоваться не может. *полнофункциональная файловая система этого типа рекомендуется компанией Microsoft для использования в качестве основной***

**2. Файловые системы FAT и FAT32 (от File Allocation Table — таблица размещения файлов) происходят от 16-разрядной файловой системы (FAT16). FAT32 впервые была введена в Windows 98 для поддержки жестких дисков большого объема и других усовершенствованных возможностей; далее под термином FAT мы будем подразумевать любую из вышеуказанных версий. FAT является единственно доступной файловой системой для дисков (но не компакт-дисков), работающих под управлением Windows 9x, а также гибких дисков. Разновидностью FAT является TFAT — ориентированная на поддержку механизма транзакций версия, используемая в Windows СЕ. Постепенно FAT выходит из употребления и в большинстве случаев ее можно встретить лишь на устаревших системах.**

**Файловые системы Windows: FAT16, FAT32, NTFS.**

**Именование файлов: FAT32, NTFS - длинные имена (полные имена, до 255 символов), FAT16 - 8 символов и 3 символа расширение.**



**Логическая организация файловой системы включает в себя структуру каталогов и файлов, которые находятся на внешнем носителе.**

**Логическое представление данных: файл и запись.**

**Файл: набор логических записей.**

**Логическая запись: последовательность байт.**

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

**Физическая организация файловой системы определяет, как файлы будут храниться на физическом устройстве хранения.**

**функция CreateFile, используемая как для создания новых, так и для открытия существующих файлов.** Для закрытия доступа к файлу, как и для закрытия доступа к любому другому объекту ядра, используется функция CloseHandle, единственным параметром которой является дескриптор открытого файла. Для физического удаления файла с диска используется функция. **Для чтения и записи используются функции ReadFile и WriteFile.**

**Чтобы скопировать файл целиком, достаточно использовать одну функцию CopyFile. Функция CopyFile копирует существующий файл с заданным именем и присваивает копии указанное новое имя. Доступны две функции, позволяющие переименовывать, или "перемещать", файл. Эти же функции применимы и к каталогам. (Функции DeleteFile и CopyFile могут применяться только к файлам.) MoveFile и MoveFileEx**

**Создание и удаление каталогов осуществляется при помощи двух простых функций CreateDirectory и RemoveDirectory.**

**Часто несколько приложений имеют совместный доступ к одному и тому же файлу. При этом может потребоваться, чтобы приложение, которое изменяет данные в файле, имело к этому файлу монопольный доступ. Для этого необходимо блокировать весь файл или только его часть для доступа другим приложениям. Для этих целей используется функция LockFile. Для отмены блокировки области файла используется функция UnlockFile.**

**Для создания каталога используется функция CreateDirectory. Для поиска файлов, находящихся в каталоге, используются функции FindFirstFile и FindNextFile. Причем функция FindFirstFile находит первый файл, имя которого удовлетворяет образцу поиска, а функция FindNextFile находит последующие такие файлы. При этом отметим, что в образцах имен файлов для поиска могут использоваться символы-заместители ? и \*. Символ ? замещает один неизвестный символ в имени файла, а символ \* — любую последовательность символов.**

**В операционных системах Windows предусмотрены функции, позволяющие следить за изменениями, происходящими в каталогах. Для этих целей предназначены функции FindFirstChangeNotification и FindNextChangeNotification, которые отслеживают соответственно первое и последующие изменения, происходящие в каталоге.**

**После завершения наблюдения за изменениями в каталоге нужно вызвать функцию FindCloseChangeNotification, которая закрывает дескриптор наблюдения за изменениями в каталоге**

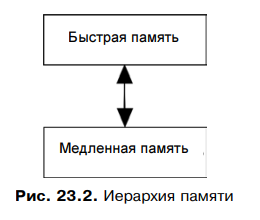
**Буфером ввода-вывода называется область оперативной памяти, предназначенная для временного хранения записей файла. Обычно длина буфера выбирается кратной длине кластера. Буферы ввода-вывода предназначены для решения двух задач:**

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

**Для решения этих задач при выводе данных файловая система сначала полностью заполняет буфер логическими записями, а затем дает команду внешнему устройству на запись данных на диск. При вводе данных система управления файлами сначала заполняет буфер кластерами, а затем управляет чтением логических записей из буфера в программу пользователя. Для ускорения ввода-вывода данных обычно используется несколько буферов ввода-вывода, которые организованы в кольцевую очередь. Во время работы пользовательского процесса с одним буфером файловая система параллельно осуществляет ввод или вывод данных в другие буферы.**

**КЭШ**

**Рассмотрим два соседних уровня иерархии памяти, которые показаны на рис. 23.2. При этом предположим, что память верхнего уровня работает быстрее, чем память нижнего уровня. Считаем, что нужные данные хранятся в памяти нижнего уровня, а механизм управления памятью обеспечивает передачу на верхний уровень тех данных, к которым наиболее вероятно обращение программы пользователя. В таких случаях говорят, что верхний уровень памяти работает как кэш по отношению к нижнему уровню памяти. Вначале этот термин применялся только к памяти самого высокого уровня, которая располагается между регистрами процессора и оперативной памятью. В настоящее время этот термин используется в более широком смысле для обозначения упреждающего ввода данных.**

****

**Кэширование ввода данных подразумевает, что система выполняет упреждающее чтение данных с магнитного диска без ожидания следующей команды на чтение данных из приложения. Это сокращает время на чтение записей файла, если они читаются приложением последовательно. В операционной системе Windows 2000 кэшированием ввода данных с диска занимается специальная программа ядра операционной системы, которая называется менеджером кэширования (Cash Manager).**

**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.

Про файловые системы в книге узбеков есть

И во 2 книге тоже норм

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

**Оперативная память** — это физическая память, используемая компьютером для временного хранения данных, к которым он должен иметь быстрый доступ во время выполнения задач. Вся информация, хранящаяся в оперативной памяти, теряется при выключении компьютера. Оперативная память используется для хранения кода программ, загруженных операционной системой, и данных, с которыми работают текущие процессы.

**Виртуальная память** — метод управления памятью процессора, который позволяет операционной системе использовать комбинацию оперативной памяти и файла на жестком диске для хранения данных, которые временно не активны или необходимы. Она создаёт видимость большего объема оперативной памяти, чем физически доступно на компьютере.

**OS: виртуальная память** (1961г) – метод управления памятью процессора, предназначенный для выполнения программ, которым выделяется адресное пространство превышающее доступный физический объем памяти компьютера.

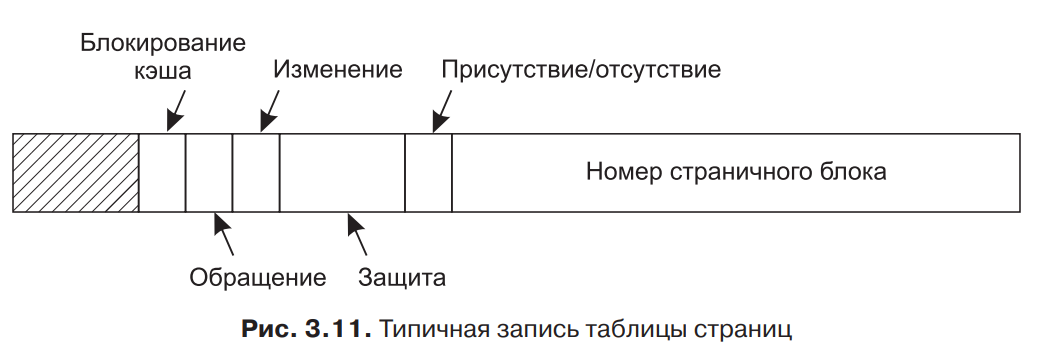
**Виртуальная память** — это способ работы с памятью, который позволяет запускать программы даже при нехватке оперативной памяти. Это достигается за счет перемещения данных из оперативной памяти в расположенный на жестком диске файл подкачки.  Виртуальная память обычно имеет больший объем, но низкую скорость доступа, поскольку она находится на жестком диске.

Связь между оперативной и виртуальной памятью состоит в том, что при нехватке оперативной памяти, операционная система начинает использовать виртуальную память, перемещая данные из оперативной памяти на жесткий диск. Этот процесс называется подкачкой. При необходимости доступа к данным, которые были перенесены в виртуальную память, операционная система перемещает их обратно в оперативную память.

При использовании виртуальной памяти виртуальные адреса не выставляются напрямую на шине памяти. Вместо этого они поступают в диспетчер памяти (Memory Management Unit (**MMU**)), который отображает виртуальные адреса на адреса физической памяти

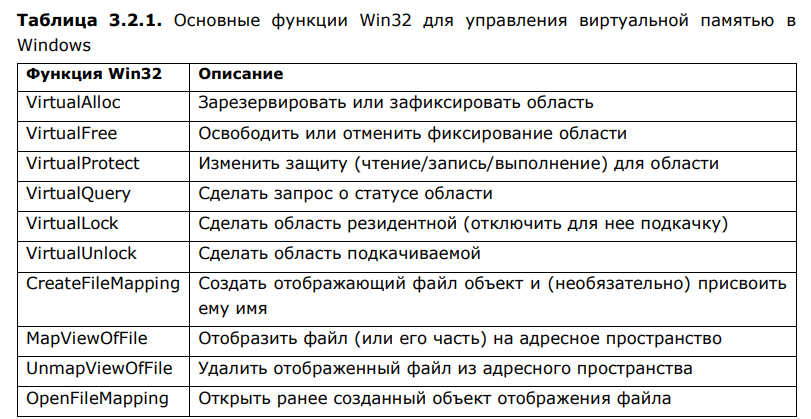
При простой реализации отображение виртуальных адресов на физические может быть сведено к следующему: виртуальный адрес делится на номер виртуальной страницы (старшие биты) и смещение (младшие биты)

Номер виртуальной страницы используется в качестве индекса внутри таблицы страниц, который нужен для поиска записи для этой виртуальной страницы. Из записи в таблице страниц берется номер страничного блока (если таковой имеется). Номер страничного блока присоединяется к старшим битам смещения, заменяя собой номер виртуальной страницы, чтобы сформировать физический адрес, который может быть послан к памяти



**Свопинг** — это процесс перемещения данных между оперативной памятью (RAM) и файлом подкачки (swap file) на жестком диске. Операционная система определяет, какие данные временно неактивны и могут быть перемещены с оперативной памяти на диск.

**OS: свопинг** –механизм OS обмена (вытеснения и загрузки) содержимым блоков оперативной физической памяти компьютера с устройством хранения данных с целью расширения адресуемого объема оперативной памяти компьютера. Механизм является аппаратно-программным



С точки зрения процесса, страницы его виртуальной памяти могут находиться в одном из трех состояний:

* свободны для использования (free);
* распределены процессу для использования (committed)
* зарезервированы, но не используются процессом (reserved);

Поясним эти состояния более подробно. Первоначально, при запуске процесса, все страницы виртуальной памяти считаются свободными, естественно кроме тех, в которые загружена сама программа. Чтобы распределить для использования свободные или зарезервированные страницы виртуальной памяти, процесс должен вызвать функцию VirtualAlloc. Только после успешного завершения этой функции процесс может использовать распределенную ему виртуальную память.

Параметр flAllocationType устанавливается программой и указывает на тип операции, которую выполняет функция **VirtualAlloc**. Значением этого параметра может быть любая комбинация следующих флагов:

MEM\_COMMIT — распределить память программе;

MEM\_RESERVE — зарезервировать область физической памяти

Параметр flProtect устанавливает атрибуты доступа к области виртуальной памяти, которые разрешают выполнять над страницами виртуальной памяти только определенные операции. Этот параметр может быть комбинацией следующих флагов:

PAGE\_READONLY — разрешает только чтение виртуальных страниц;

PAGE\_READWRITE — разрешает чтение и запись в виртуальных страницах;

PAGE\_EXECUTE — разрешает только исполнение кода в виртуальных с

 PAGE\_EXECUTE\_READ — разрешает исполнение и чтение кода в виртуальных страницах;

PAGE\_EXECUTE\_READWRITE — разрешает выполнение, чтение и запись виртуальных страниц;

PAGE\_NOACCESS — к виртуальным страницам нет доступа;

PAGE\_NOCACHE — виртуальные страницы можно не помещать в кэш.

Третье состояние характеризует виртуальные страницы как зарезервированные. Это значит, что эти виртуальные страницы зарезервированы процессом для дальнейшего использования и не будут выделяться системой для использования процессу без точного указания процессом их адреса. Следует отметить, что **при резервировании виртуальных страниц реальная память под эти страницы не выделяется.**

После завершения работы с виртуальной памятью ее необходимо освободить, используя функцию VirtualFree.

Параметр dwFreeType может принимать любую комбинацию следующих двух флагов:

MEM\_DECOMMIT — отменить распределение виртуальной памяти;

MEM\_RELEASE — освободить виртуальную

Если используется флаг MEM\_DECOMMIT, то память не освобождается, а остается в зарезервированном состоянии. Чтобы освободить зарезервированную виртуальную память, нужно установить флаг MEM\_RELEASE

Если некоторая область виртуальной памяти будет часто использоваться процессом, то можно запретить системе выгружать эти виртуальные страницы из реальной памяти, иначе говоря, запереть или блокировать эти виртуальные страницы в реальной памяти. Для этого нужно использовать функцию VirtualLock. При этом нельзя запирать в реальной памяти виртуальные страницы, к которым доступ запрещен, т. е. для которых установлен флаг PAGE\_NOACCESS. Кроме того, операционная система не позволяет процессу блокировать в реальной памяти более 30 виртуальных страниц сразу. Для отмены блокировки виртуальных страниц в реальной памяти используется функция VirtualUnlock

FillMemory – инициализировать память, CopyMemory – копировать блок памяти, MoveMemory – копировать перекрывающиеся блоков памяти, ReadProcessMemory – чтение памяти дочернего процесса, WriteProcessMemory – запись в память дочернего процесса.

Изменить атрибуты доступа к области виртуальной памяти можно при помощи вызова функции VirtualProtect,

Определить состояние области виртуальной памяти процесса можно при помощи вызова функции VirtualQuery (заполняет структуру):

typedef struct \_MEMORY\_BASIC\_INFORMATION

{

PVOID BaseAddress; // базовый адрес области виртуальной памяти

PVOID AllocationBase; // базовый адрес распределенной памяти

DWORD AllocationProtect; // атрибуты доступа к распределенной памяти

SIZE\_T RegionSize; // размер области с одинаковыми атрибутами

DWORD State; // состояние памяти

DWORD Protect; // атрибуты доступа к области виртуальной памяти

DWORD Type; // тип страниц в области виртуальной памяти

} MEMORY\_BASIC\_INFORMATION, \*PMEMORY\_BASIC\_INFORMATION;

Поле Type описывает тип страниц в области виртуальной памяти, информация о которой определяется. Это поле может принимать одно из следующих значений:

MEM\_IMAGE — исполняемый код;

MEM\_MAPPED — файл, проецируемый в память;

MEM\_PRIVATE — память, принадлежащая процессу.

Узнать о количестве страниц, которые входят в рабочее множество процесса, можно посредством вызова функции GetProcessWorkingSetSize

Куча

**heap** – фрагмент памяти адресного пространства (по умолчанию 1MB), предназначенный для динамического использования (malloc/free, new/delete)

**Heap** – область памяти адресного пространства, предназначенного для использования программной фрагментов динамически выделяемой памяти (malloc, new). По умолчанию – 1MB, из них 4K сразу забирает процесс. Можно установить стартовое значение величины HEAP в параметрах Linker. По мере new(malloc) размер HEAP прирастает. Память выделяется с учетом минимизации фрагментации.

Кучей или пулом памяти называется распределенная процессом область виртуальной памяти, используемая им для захвата и освобождения блоков памяти, размер которых меньше размера виртуальной страницы. Куча называется сериализуемой, если система синхронизирует доступ параллельно работающих потоков к этой куче. **В Windows каждая куча имеет свой дескриптор и, следовательно, является объектом ядра.**

Для каждого процесса Windows по умолчанию резервирует одну кучу размером в 1 Мбайт и сразу распределяет из нее 4 Кбайт виртуальной памяти для использования процессом. Функции malloc и free из стандартной библиотеки языка программирования C, а также операторы new и delete языка программирования С++ распределяют память из кучи, зарезервированной для процесса по умолчанию.

Дескриптор кучи, созданной для процесса по умолчанию, можно получить при помощи функции GetProcessHeap. Для динамического создания кучи используется функция HeapCreate, Для уничтожения кучи нужно использовать функцию HeapDestroy. Для распределения памяти из кучи используется функция HeapAlloc.

Если память, распределенная из кучи, больше не используется программой, то ее нужно вернуть обратно в кучу, т. е. — освободить. Для освобождения памяти, распределенной из кучи, используется функция HeapFree. Если куча не является сериализуемой, то параллельный доступ нескольких потоков к этой куче может нарушить ее непротиворечивое состояние и вызвать ошибку в работе приложения. Чтобы избежать такой ситуации, в операционных системах Windows предусмотрены функции HeapLock и HeapUnlock, которые блокируют параллельный доступ нескольких потоков к куче. Для того чтобы получить монопольный доступ к куче, поток должен вызвать функцию **HeapLock**.

Если куча заблокирована потоком при помощи функции **HeapLock** и другой поток вызывает какую-нибудь функцию для доступа к этой куче, то система переведет его в состояние ожидания до тех пор, пока поток, вызвавший функцию HeapLock, не вызовет функцию HeapUnlock

Для перераспределения памяти в куче используется функция HeapReAlloc

Чтобы получить информацию о состоянии кучи или отдельного блока виртуальной памяти, распределенного из кучи, можно использовать функцию HeapValidate

Для получения более подробной информации о состоянии кучи нужно использовать функцию **HeapWalk**, которая записывает информацию о блоках виртуальной памяти из кучи в структуру типа:

Если при работе с кучей распределяются и освобождаются блоки памяти разной длины, то со временем происходит **фрагментация** кучи, т. е. внутри кучи образуются свободные блоки. Это происходит потому, что длина свободного блока в куче не всегда соответствует длине распределяемого блока памяти. Фрагментация кучи приводит к двум последствиям. Во-первых, к неэффективному использованию памяти, т. к. внутри кучи находятся свободные неиспользуемые блоки виртуальной памяти, которые часто не могут быть использованы для распределения памяти из кучи из-за своей малой длины. Во-вторых, к более медленной работе с кучей, т. к. для распределения блока памяти требуется просмотр списка свободных блоков кучи.

Чтобы частично избежать этих последствий, необходимо выполнить уплотнение кучи, которое заключается в соединении последовательных свободных блоков памяти в куче в один блок большей длины и освобождении больших блоков свободной виртуальной памяти.

Для уплотнения кучи используется функция HeapCompact,

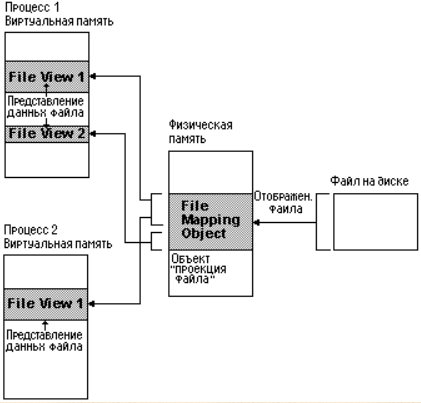
 API для работы с Heap: два двусвязных списка (список занятых, список свободных областей) создавать, перераспределять, проверять целостность.

ОТЛИЧИЕ ПОЛЬЗОВАТЕЛЬСКОЙ И СИСТМЕНОЙ КУЧИ

# 9. 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.

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);

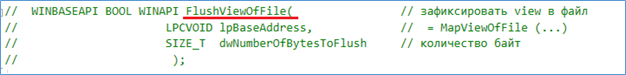
4 (B) Закройте объект сопоставления файлов. Этот шаг удаляет отображение файла из физической памяти.

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).

-----------------------------------------------------------------------------------------------------------------------------------------------------------

У ВЕРЫ

**Прежде чем перейти к описанию динамически подключаемых библиотек, рассмотрим механизм, который позволяет динамически выполнять это подключение. Этот механизм называется отображением содержимого файла (file mapping) в виртуальную память процесса.**

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

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

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

**После того как файл открыт, создается объект, отображающий этот файл в память. Под объектом, отображающим файл в память, можно понимать объект ядра операционной системы, который выполняет отображение файла в адресное пространство процесса. Можно также представить, что этот объект позволяет рассматривать файл, отображаемый в память, как файл подкачки. Для создания этого объекта используется функция CreateFileMapping**

**После того как создан объект, отображающий файл в память, файл или его часть должна быть отображена в память процесса. Другими словами, должен быть создан вид файла или его части в адресном пространстве процесса. Для отображения файла или его части в адресное пространство процесса используется функция MapViewOfFile**

**Если необходимо отобразить файл в адресное пространство процесса, начиная с некоторого заданного виртуального адреса, то для этой цели нужно использовать функцию MapViewOfFileEx**

**После окончания работы с видом файла в памяти нужно отменить отображение файла в адресное пространство процесса. Отмена отображения файла освобождает виртуальные адреса процесса. Следует особо отметить, что если отображение файла в адресное пространство процесса не отменено, то система продолжает держать отображаемый файл открытым до тех пор, пока существует его вид, независимо от того, закрыт этот файл функцией CloseHandle или нет. Для отмены отображения файла в память используется функция UnmapViewOfFile**

**Несколько процессов могут одновременно отображать один и тот же файл в свое адресное пространство. В этом случае операционная система обеспечивает согласованность содержимого файла для всех процессов, если доступ к этим данным осуществляется как к области виртуальной памяти процесса. То есть для доступа к файлу, который отображен в память, не используется функция WriteFile. Такая согласованность данных, хранящихся в файле, отображенном в память несколькими процессами, называется когерентностью данных. Однако следует отметить, что когерентность данных для файла, отображенного в память, не поддерживается в том случае, если этот файл отображается в адресное пространство процессов, которые выполняются на других компьютерах в локальной сети.**

**Так как один и тот же файл может быть отображен в память несколькими процессами, и система поддерживает когерентность таких отображений, то механизм отображения файлов в память может использоваться для обмена данными между процессами. Кроме того, в операционных системах Windows все остальные механизмы обмена данными между процессами базируются на отображении файлов в память. Поэтому можно сказать, что отображение файлов в память обеспечивает наилучшую производительность по сравнению со всеми остальными способами обмена данными между процессами**

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

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

**Динамически подключаемая библиотека (DLL, Dynamic Link Library) представляет собой программный модуль, который может быть загружен в виртуальную память процесса как статически, во время создания исполняемого модуля процесса, так и динамически, во время исполнения процесса операционной системой. Программный модуль, оформленный в виде DLL, хранится на диске в виде файла, который имеет расширение dll, и может содержать как функции, так и данные.**

**Для загрузки DLL в память используется механизм отображения файлов в память. Динамически подключаемые библиотеки предназначены, главным образом, для разработки функционально-замкнутых библиотек функций, которые могут использоваться разными приложениями. Это позволяет снизить затраты на разработку программного обеспечения, т. к. один и тот же программный код может использоваться разными разработчиками. Кроме того, динамически подключаемые библиотеки позволяют уменьшить объем используемой физической памяти при одновременной работе нескольких приложений, которые используют одну и ту же библиотеку. Это достигается благодаря механизму проецирования DLL в виртуальную память процессов, т. к. в этом случае все приложения разделяют один и тот же экземпляр исполняемого кода DLL, загруженный в физическую память**

**В отличие от исполняемых модулей, в которых эта функция называется main, в DLL главная функция называется DllMain и вызывается операционной системой при загрузке DLL в адресное пространство процесса и при создании этим процессом нового потока. Главное назначение функции DllMain заключается в инициализации DLL при ее загрузке, а также захвате и освобождении необходимых ресурсов при создании и завершении нового потока в процессе.**

**Теперь рассмотрим, как оформляются функции и переменные, которые DLL предоставляет в использование своим клиентам. Во-первых, заметим, что такие функции и переменные называются экспортируемыми. Для того чтобы сделать функцию или переменную экспортируемой, нужно определить их с модификатором extern "C" и квалификатором \_\_declspec(dllexport). Кроме того, экспортируемая переменная должна быть инициализирована. Модификатор extern "C" указывает компилятору на то, что функция или переменная должна иметь имя в стиле языка программирования С. То есть имя функции или переменной не будет искажаться путем добавления к нему обозначений типов данных из сигнатуры функции или определения переменной.**

**----------------------------------------------------------------------------------------------------------**

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.

**----------------------------------------------------------------------------------------------------------**

**Модификатор \_\_declspec(dllexport) указывает компилятору на то, что данная функция или переменная будет экспортироваться из DLL. Для определения экспортируемых из DLL функций и переменных можно использовать файл определения модуля, который должен иметь расширение def. Такие файлы включают инструкции, которые используются компоновщиком для создания исполняемого модуля или DLL. Однако эти файлы не обязательны для компоновки, т. к. большинство их параметров может быть задано при помощи определения опций компоновщика. При создании DLL def-файл содержит инструкции, которые определяют имена экспортируемых функций и переменных. Отметим, что если для описания экспорта из DLL используется файл определений, то квалификатор \_\_declspec(dllexport) перед именами функций и переменных в самой DLL использовать не нужно.**

**Для динамической загрузки DLL в виртуальную память процесса используются две функции LoadLibrary и LoadLibraryEx. Функция LoadLibraryEx отличается от функции LoadLibrary только тем, что позволяет управлять режимом загрузки библиотеки. Рассмотрим эти функции подробнее. Для загрузки динамически подключаемой библиотеки используется функция LoadLibrary,**

**Для отключения DLL от процесса используется функция FreeLibrary. Для одновременного завершения потока и отключения DLL используется функция FreeLibraryAndExitThread**

***В принципе, функция загрузки одной и той же DLL может вызываться в приложении несколько раз, что имеет смысл в многопоточном приложении. В этом случае при каждом новом вызове функции загрузки одной и той же DLL эта библиотека не загружается вновь, а используется счетчик ссылок на библиотеку, который при каждом вызове функции увеличивается на единицу. Соответственно, при каждом отключении этой DLL счетчик ссылок уменьшается на единицу. Динамически подключаемая библиотека выгружается только в том случае, если счетчик ссылок на эту библиотеку становится равным нулю***

**Если программа использует некоторые функции и переменные из DLL, то говорят, что она импортирует их из DLL. Для обеспечения доступа к импортируемым из DLL функциям и переменным используется функция GetProcAddress.**

**Статическая загрузка DLL отличается от динамической тем, что выполняется на этапе компоновки исполняемого модуля, а не на этапе его исполнения. Для этого компоновщик использует библиотеку импорта. Библиотека импорта — это файл с расширением lib, который создается компоновщиком при создании динамической библиотеки. *В библиотеке импорта содержатся ссылки на все экспортируемые из динамической библиотеки имена.* Для статической загрузки DLL необходимо выполнить следующую последовательность действий:**

**1. Создать любым из вышеуказанных способов DLL.**

**2. Поместить саму библиотеку и файл импорта этой библиотеки в каталог, из которого запускается приложение.**

**3. В меню Project выбрать пункт Settings, на котором выбрать закладку Link. После этого в окне Object/library modules ввести имя используемой библиотеки импорта.**

**4. Описать импортируемые из DLL имена в приложении.**

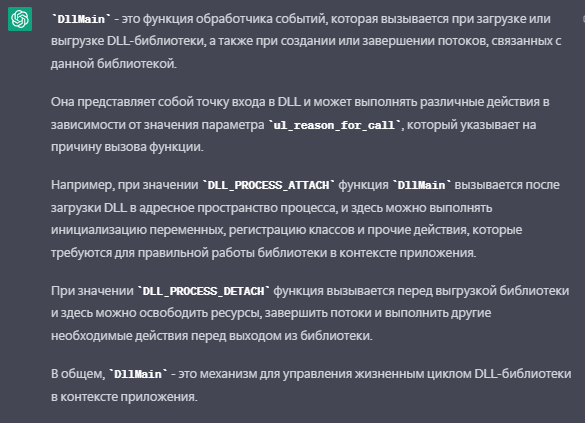
При неявном подключении (implicit linking) линкеру передается ***библиотека импорта*** (обычно имеет расширение lib), содержащая список переменных и функций DLL, которые могут использовать приложения. Обнаружив, что программа обращается хотя бы к одной из них, линкер добавляет в целевой exe-файл ***таблицу импорта***. Таблица импорта содержит список всех DLL, которые использует программа, с указанием конкретных переменных и функций, к которым она обращается. Позже, когда exe-файл будет запущен, загрузчик проецирует все DLL, перечисленные в таблице импорта, на адресное пространство процесса; в случае неудачи весь процесс немедленно завершается.

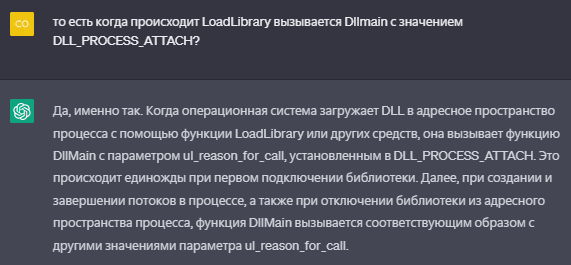
Поиск файлов DLL операционной системой осуществляется в следующей последовательности.

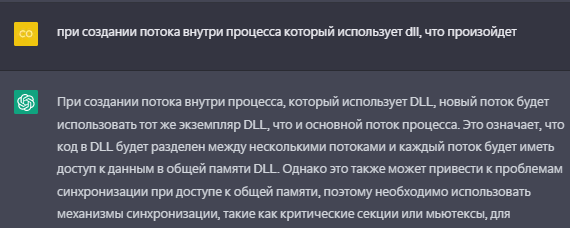
* **Каталог, в котором находится ЕХЕ-файл.**
* **Текущий каталог процесса.**
* **Системный каталог Windows.**

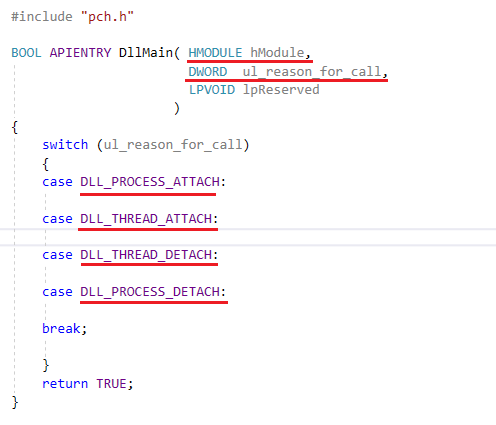
#pragma comment(lib,"MyDll.lib")

Каждый из способов имеет свои достоинства и недостатки. В случае неявного подключения все библиотеки, используемые приложением, загружаются в момент его запуска и остаются в памяти до его завершения (даже если другие запущенные приложения их не используют). Это может привести к нерациональному расходу памяти, а также заметно увеличить время загрузки приложения, если оно использует очень много различных библиотек. Кроме того, если хотя бы одна из неявно подключаемых библиотек отсутствует, работа приложения будет немедленно завершена. Явный метод лишен этих недостатков, но делает программирование более неудобным, поскольку требуется следить за своевременными вызовами LoadLibrary и соответствующими им вызовами FreeLibrary, а также получать адрес каждой функции через вызов GetProcAddress







****

# 11. 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 – модель программного обеспечения.**

**COM, или Component Object Model –** это ~~технология~~ модель, разработанная Microsoft. Используется для создания компонентов ПО, которые могут взаимодействовать между собой на основе стандартизированных интерфейсов.

**COM предназначен для создания программного обеспечения на основе взаимодействующих компонентов объекта, каждый из которых может использоваться во многих программах одновременно**

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

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

COM-объект (компонент) – это программный объект, созданный в соответствии с моделью компонентов (COM),

**Для идентификации компонента используется идентификатор CLSID.** – это уникальный идентификатор COM-компонента (используется для однозначной идентификации COM-объекта в системе.)

**GUID** - тип данных размером 128 бит, который используется для идентификации com-компонета или com-интерфейса

структура COM-клиента

1. CoInitialize(NULL);// инициализация библиотеки OLE32

2. HRESULT hr0 = CoCreateInstance(

CLSID\_CA, //идентификатор класса объекта, который вы хотите создать

NULL, //для передачи параметров конструктору

CLSCTX\_INPROC\_SERVER, // контекст, в котором будет создан объект

IID\_IUnknown, //идентификатор интерфейса, который вы хотите получить от созданного объекта.

(void\*\*)&pIUnknown); //сохранен указатель на интерфейс IUnknown, который будет получен после создания объекта

**3. Запрашиваем интерфейс**

pIUnknown->QueryInterface(IID\_IAdder, (void\*\*)&pIAdder)

4. Уменьшаем счётчик на 1  
 pIAdder->Release();

pIUnknown->Release();

5. CoFreeUnusedLibraries(); // завершение работы с библиотекой

Сервер

1. Интерфейс наследуюемый от IUnknown

2. Класс реализующий наш интерфейс и его методы + стандартные

3. Класс фабрика реализующий IClassFactory

4. В фабрике определяем CreateInstance, LockServer и еще QueryInterface, AddRef, Release

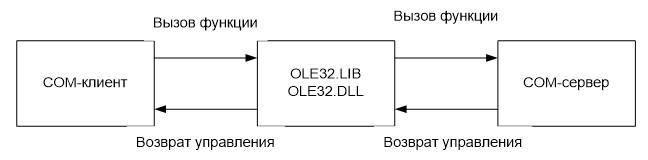
DllGetClassObject, DllCanUnloadNow

**Каждый компонент имеет уникальный идентификатор GUID (Globally Unique Identifier) — статистически уникальный 128-битный идентификатор. Его главная особенность — уникальность, которая позволяет создавать расширяемые сервисы и приложения без опасения конфликтов, вызванных совпадением идентификаторов и может одновременно использоваться многими программами.**

**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 (программный модуль, создающий COM-объект и использующий его методы) и COM-сервером (программный модуль, реализующий COM-объект.) выступает библиотека OLE32.DLL (библиотека импорта OLE32.LIB). Все функции OLE32.DLL возвращают значение типа HRESULT.**

****

**COM-сервер может иметь тип:**

* **CLSCTX\_INPROC\_SERVER (DLL внутрипроцессный сервер);**
* **CLSCTX\_LOCAL\_SERVER (EXE-сервер на том же компьютере),**
* **СLSCTX\_REMOTE\_SERVER (EXE-сервер на удаленном компьютере);**

**Интерфейсы используются для доступа к методам COM-объектов. Описывает один или несколько методов. Каждый интерфейс имеет свой идентификатор. В спецификации COM есть несколько стандартных интерфейсов (идентификатор и методы которого являются общедоступными), которые заранее прописаны в документации (например, IUnknown (базовый интерфейс для всех интерфейсов), IClassFactory(создаёт экземпляр COM-объекта)).**

**методы интерфейса IUnknown**

* **QueryInterface (получает id в ответ отправляет ссылку, можно получить указатель на другой интерфейс)**
* **AddRef (внутри компонента поддерживается счётчик, этот метод увеличивает на 1 счётчик ссылок на интерфейс)**
* **Release (уменьшает счётчик ссылок на интерфейс на 1)**

**Фабрика классов - Специальная компонента, задача которой создавать экземпляры объекта. Для каждого компонента своя фабрика. Фабрика классов реализует интерфейс IClassFactory**

**методы интерфейса IClassFactory**

* **CreateInstance создаёт экземпляры рабочих компонентов (создаёт инстанс CA). Для каждого сервера фабрика классов (ClassFactory) создаёт компонент. После создания инстанса, возвращает клиенту указатель на интерфейс IUnknown.**
* **LockServer(params). LockServer мб необходим, если необходимо эксклюзивное использование сервера (1 – сервер заблокирован, 0 – разблокирован).**

**Счетчик ссылок на интерфейсы - Для каждого компонента необходимо подсчитывать сколько ссылок сделано на его интерфейс. Это нужно для того, чтобы знать, сколько клиентов подключены к DLL. Если счётчик компонента == 0, он сам себя убивает. Увеличивается при использовании какого-либо метода, уменьшается после Release.**

**Поддерживают соглашение о вызове stdcall (аргументы передаются через стек, справа налево, очистку стека производит вызываемая подпрограмма); используется макрос STDMETHODCALLTYPE. Возвращают значение типа HRESULT (как функции OLE32). Исключение составляют методы AddRef и Release.**

**Клиенты не знают как СОМ-объект выполняет свои действия. СОМ-объект предоставляет свои услуги при помощи интерфейсов. В дополнение, приложению клиенту не нужно знать, где находится СОМ-объект. Технология СОМ обеспечивает прозрачный доступ независимо от местонахождения СОМ-объекта.**

**Когда клиент запрашивает услугу от СОМ-объекта, он передает СОМ-объекту идентификатор класса (CLSID). После передачи CLSID, СОМ-сервер должен обеспечить так называемую фабрику класса, которая создает экземпляры СОМ объектов.**

**В общих чертах, СОМ-сервер должен выполнять следующее:**

* **регистрировать данные в системном реестре Windows для связывания модуля сервера с идентификатором класса (CLSID);**
* **предоставлять фабрику СОМ-класса, создающую экземпляры СОМ-объектов;**
* **обеспечивать механизм, который выгружает из памяти серверы СОМ, которые в текущий момент времени не предоставляют услуг клиентам.**

**Экспортируемые стандартные функции:**

**Внутренний СОМ-сервер должен экспортировать четыре функции:**

* **function DllRegisterServer: HResult; stdcall;**
* **function DllUnregisterServer: HResult; stdcall;**
* **function DllGetClassObject (const CLSID, IID: TGUID; var Obj): HResult;**
* **function DllCanUnloadNow: HResult; stdcall;**
* **DllInstall используется только для установки и настройки приложений. Это не должно вызываться приложением**

**DllRegisterServer - применяется для регистрации DLL СОМ-сервера в системном реестре Windows. При регистрации СОМ-класса в системном реестре создается раздел в HKEY\_CLASSES\_ROOT\CLSID\{XXXXXXXX-XXXX-XXXX-xxxx-xxxxxxxx}, где число, записанное вместо символов х, представляет собой CLSID данного СОМ-класса. DllUnregisterServer - применяется для удаления всех разделов, подразделов и параметров, которые были созданы в системном реестре функцией DllRegisterServer при регистрации DLL СОМ-сервера.**

**DllGetclassObject - возвращает фабрику класса для конкретного СОМ-класса.**

**DllСanUnloadNow - применяется для определения, можно ли в настоящий момент времени выгрузить DLL СОМ-сервера из памяти.**

**Регистрация COM/DLL-сервера:**

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

**Формат командной строки REGSVR32:**

**Regsvr32 [/u] [/s] [/n] [/i[:строка\_команд]] DLL-файл**

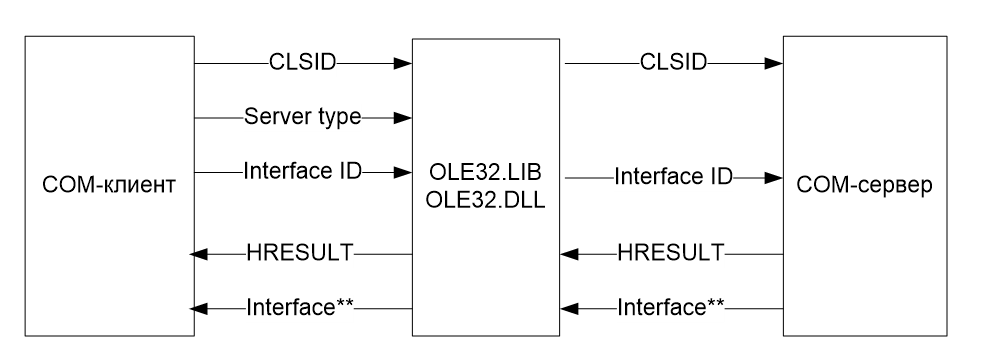
**/u — - вызывает DllUnInstall, отменяет регистрацию сервера**

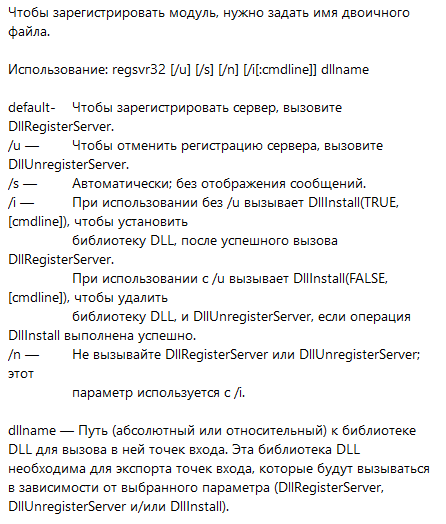
**/i — вызывает DllInstall, передавая ей в параметре необязательную строку\_команд;**

**/n — не вызывает DllRegisterServer; может использоваться с ключом /I;**

**/s – "тихий" режим; окна сообщений не отображаются;**

**Реестр Windows, или системный реестр — иерархически построенная база данных параметров и настроек в большинстве операционных систем Microsoft Windows.**





За стандартными интерфейсами закреплены предопределенные GUID-идентификаторы.

Далее в книжке Смелова  
1. Что такое COM? COM-программирование?

**COM, или Component Object Model** – это технология, разработанная Microsoft. Используется для создания компонентов ПО, которые могут взаимодействовать между собой на основе стандартизированных интерфейсов.

**COM-программирование**: разработка программного обеспечения, имеющего модель COM.

2. Что такое COM-объект(компонент)? CLSID?

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

COM-объект (компонент) – это программный объект, созданный в соответствии с моделью компонентов (COM), который предоставляет стандартизированный интерфейс для взаимодействия с другими объектами.

**CLSID** – это уникальный идентификатор COM-компонента (используется для однозначной идентификации COM-объекта в системе.)

3. Что такое GUID? Где применяется GUID? Размер GUID-идентификатора?

**GUID** - тип данных размером 128 бит, который используется для идентификации com-компонета или com-интерфейса

4. Какие типы COM-контейнеров бывают?

exe и dll

5. Что является клиентом и сервером в COM?

сервер – это программный модуль, который реализует COM-компоненты

клиент – это программный модуль, который создает экземпляры com-компонентов и использует их методы (программный модуль, создающий COM-объект и использующий его методы.)

\*в качестве COM-клиента может выступать COM-сервер.

6. Поясните понятия «однокомпонентный» и «многокомпонентный» COM-сервер.

com-сервер называется "однокомпонентым" если состоит из одного пользовательского компонента помимо стандартного компонента, который реализует интерфейс IClassFactory

"многокомпонентный" сервер состоит из нескольких пользовательских компонентов

7. Поясните типы COM-серверов: CLSCTX\_INPROC\_SERVER, CLSCTX\_LOCAL\_SERVER, СLSCTX\_REMOTE\_SERVER.

CLSCTX\_INPROC\_SERVER - COM-сервер выполняется в том же процессе, что и клиент. (dll-сервер внутрипроцессовый)

CLSCTX\_LOCAL\_SERVER - COM-сервер выполняется в отдельном процессе, отличном от клиента, но оба клиент и сервер находятся на одной и той же машине (локально). (exe-сервер)

СLSCTX\_REMOTE\_SERVER - exe-сервер, который работает на удаленной машине

8. Как называется имя библиотеки, обеспечивающей работу COM-приложений.

OLE32.DLL

9. Поясните назначение типа и структуру HRESULT.

HRESULT тип данных, который хранит информацию о результате вызова функции компонента

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

10. Что такое COM-интерфейс?

**COM-интерфейс** – абстракция для взаимодействия с COM-объектами (скрывает детали их внутренней реализации). Определяет набор методов, которые могут быть вызваны клиентским кодом для выполнения операций над объектом

**:** для доступа к методам COM-объекта применяются **интерфейсы** COM-объекта

11. Чем характеризуется COM-интерфейс?

1)идентификатор, который имеет тип GUID, 2) методы, 3) свойства

12. Что значит «стандартный» COM-интерфейс?

**Стандартный COM-интерфейс** – это интерфейс, описанный в спецификации COM, широко используются в COM для общих задач, все знают его IID.

13. Назовите два стандартных COM-интерфейса.

IUnknown

IClassFactory

14. Перечислите методы интерфейса IUnknown и поясните их назначение.

\* QueryInterface (запросить другой интерфейс по ID)

\* AddRef (увеличивает счетчик ссылок на интерфейс)

\* Release (уменьшает счетчик ссылок на интерфейс)

15. Что такое «фабрика классов» и для чего она нужна?

Объект, реализующий интерфейс IClassFactory, который создает экземпляры COM-объектов

16. Перечислите методы интерфейса IClassFactory и поясните их назначение.

\* CreateInstance –создает новый экземпляр COM-объекта и возвращает указатель на созданный объект

\* LockServer – Управляет блокировкой сервера

17. Что такое «счетчик ссылок на интерфейсы»? Для чего он нужен? Каким образом и когда этот счетчик увеличивается и уменьшается?

Счетчик ссылок на интерфейсы (Reference Count) – это механизм для отслеживания количества активных ссылок на интерфейс объекта.

Позволяет узнать когда можно удалить экземпляр компонента из памяти.

AddRef – увеличивает (InterlockedIncrement)

Release – уменьшает (InterlockedDecrement)

18. Какое соглашение о вызове и возврате должен обеспечивать метод COM-объекта? Какие методы являются исключением?

Методы COM-объектаподдерживают соглашение о вызове **stdcall** (аргументы передаются через стек, справа налево, очистку стека производит **вызываемая** подпрограмма); возвращают значение типа **HRESULT** (как функции OLE32).

**Исключение** составляют методы **AddRef** и **Release** интерфейса **IUnknown**.

19. Что должен «знать» COM-клиент, чтобы использовать COM-объект?

1) **CLSID** компонента 2) **тип DLL-сервера** (контейнера) 3) **ID** **интерфейсов** объекта.

20. Объясните в чем заключается процесс регистрации COM-объекта?

Записать в реест путь к dll и его CLSID

21. Поясните назначение утилиты regsvr32 и принцип ее работы.

для регистрации и отмены регистрации COM-серверов в системном реестре

Она вызывает функцию DllRegisterServer или DllUnregisterServer в COM-библиотеке для выполнения соответствующих действий.

22. Поясните назначение утилиты regedit.

позволяет просматривать и изменять записи рееста

23. Перечислите пять функций, которые экспортируются COM/DLL-контейнером. Поясните назначение этих функций.

DllRegisterServer - исп. regsvr32 для регистрации dll в реестре

DllUnregisterServer - исп. regsvr32 для удаления dll из реестра

DllInstall - исп. regsvr32 чтобы понять можно ли вызывать другие функции

DllGetClassObject - исп. OLE32; позволяет клиентскому коду получить указатель на класс-фабрику COM-объекта для получения компонента по CLSID

DllCanUnloadNow - исп. OLE32 чтобы узнать можно ли выгрузить dll

24. Назовите функцию COM-контейнера, которая вызывается OLE32 для получения указатель на фабрику классов.

DllGetClassObject

25. Назовите функцию фабрики классов, в которой создается объект компонента.

CreateInstance

26. Поясните назначение «счетчика экземпляров компонент». Где этот счетчик увеличивается и где уменьшается?

Показывает количество активных экземпляров компонентов

Счетчик увеличивается каждый раз, когда создается новый экземпляр объекта (обычно в методе **CreateInstance** фабрики классов) и уменьшается каждый раз, когда объект освобождается (при вызове **Release**).

Изменяется в конструкторе/деструкторе компонента

27. Назовите условие, при котором объект компонента удаляется.

когда счетчик ссылок на этот объект (счетчик экземпляров) достигает нуля.

счетчик ссылок на интерфейс == 0 и счетчик экземпляров компонента == 0

28. Объясните механизм блокировки COM-сервера (функция LockServer фабрики классов).

Механизм блокировки COM-сервера используется для управления его жизненным циклом. Если сервер заблокирован, он не будет выгружаться из памяти, даже если нет активных ссылок на COM-объекты.

**LockServer(TRUE)** счетчик активных блокировок увеличивается на 1. Если счетчик блокировки COM-сервера установлен в значение больше нуля, это предотвращает выгрузку сервера из памяти, даже если все клиенты освободили свои ссылки на объекты.

**LockServer(FALSE)**: клиент вызывает функцию **LockServer** с параметром **FALSE**. Это уменьшает счетчик блокировки на 1. Когда счетчик блокировки достигнет нуля (то есть, сервер разблокирован), COM-сервер может быть выгружен из памяти, если не существует активных клиентов.

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

* 1. [**Идентификация**](https://ru.wikipedia.org/wiki/%D0%98%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B)) — присвоение субъектам и объектам идентификатора и / или сравнение идентификатора с перечнем присвоенных идентификаторов.
  2. Аутентификация – проверка принадлежности субъекту предъявленного им идентификатора, подтверждающего личность.
  3. Авторизация – процесс проверки прав субъекта на выполнение некоторых действий.

Учётная запись — хранимая в компьютерной системе совокупность данных о пользователе, необходимая для его опознавания (аутентификации) и предоставления доступа к его личным данным и настройкам.

Пользователь операционной системы – это человек или системный процесс, который имеет доступ и использует ресурсы компьютерной системы через определенную учетную запись (логин), предоставленную операционной системой для идентификации и аутентификации.

Группа пользователей - логическая сущность, объединяющую несколько пользователей для управления доступом и разрешениями в операционной системе. В группе пользователей можно определить права на файлы, папки, приложения и другие ресурсы, а также ограничения, связанные с безопасностью

группу пользователей можно рассматривать как поименованный набор разрешений

**Дискреционная система безопасности** - это **система безопасности, которая определяет доступ к объектам в компьютерной системе** на основе разрешений, предоставленных владельцем объекта.

Это означает, что **владелец объекта контролирует, кто может получить доступ к этому объекту** и какие действия могут быть выполнены с ним.

**Дискреционная политика безопасности основывается на следующих принципах:**

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

КНИГА СМЕЛОВА

Любые действия, осуществляемые в операционной системе (выполнение программы, создание, изменение файла и т. п.), всегда выполняются от чьего-то лица. С другой стороны, все объекты операционной системы (потоки, процессы, объекты синхронизации, файлы

и т. п.) имеют своего владельца. На этом принципе построена дискреционная защита – контроль доступа, используемый в операционных

системах.

Прежде чем любой пользователь сможет работать в среде операционной системы Windows, он должен быть зарегистрирован администратором системы. Администратор системы – пользователь, обладающий специальными возможностями. При установке операционной

системы всегда создается первый администратор (задается его имя и

пароль), который в последствии может создавать (регистрировать)

других пользователей, в том числе и обладающих возможностями администратора.

Каждому пользователю операционной системы Windows соответствует учетная запись, которая хранится в базе данных менеджера

учетных записей. Эта база данных является частью системного реестра Windows.

Кроме учетных записей администратора и пользователей, созданных администратором, есть несколько специальных учетных записей,

используемых операционной системой для внутренних целей и сервисной поддержки программного и аппаратного обеспечения.

Каждый пользователь Windows имеет имя и может иметь пароль.

Эти данные позволяют нему подключиться к операционной системе.

С каждым объектом безопасности операционной системы (объекты,

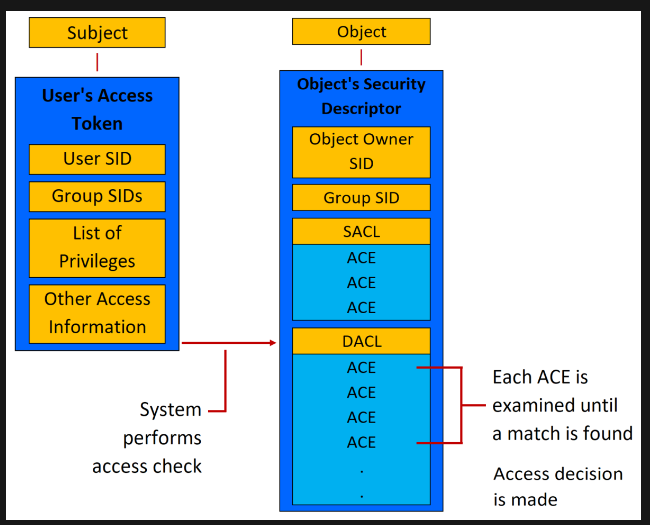
имеющие имя) связаны специальные списки, характеризующие права

доступа пользователей к этим объектам.

Для того чтобы просмотреть список учетных записей пользователей, создать, изменить свойства или удалить учетные записи, можно

воспользоваться оснасткой «Управление компьютером» mmc-консоли, доступ к которой можно получить через оп

В качестве активных субъектов этой модели безопасности рассматриваются процессы и потоки, каждый из которых работает от имени некоторого пользователя. Когда пользователь регистрируется и входит в систему, то для него создается маркер доступа (access token), который идентифицирует этого пользователя и содержит его привилегии. Каждый процесс, исполняемый от имени пользователя, имеет маркер доступа этого пользователя. Маркер доступа используется для контроля доступа процесса к объектам, которые называются в Windows охраняемыми объектами (securable objects). К охраняемым объектам относятся все объекты Windows, которые могут иметь имя. Кроме того, к охраняемым объектам относятся также потоки и процессы. Каждый охраняемый объект имеет дескриптор безопасности (security descriptor), который создается вместе с охраняемым объектом и содержит информацию, необходимую для защиты объекта от несанкционированного доступа. В дескрипторе безопасности идентифицируется владелец объекта, определяются пользователи и группы пользователей, которым разрешен или запрещен доступ к охраняемому объекту, а также информация для аудита доступа к объекту. Изменять информацию, заданную в дескрипторе безопасности, может только владелец объекта, которым по умолчанию является создатель этого объекта. При доступе к охраняемому объекту система сверяет информацию о пользователе, заданную в маркере доступа, с информацией, заданной в дескрипторе безопасности. Если в дескрипторе безопасности указано, что пользователю разрешен доступ к объекту, то процесс получает запрашиваемый доступ, в противном случае в доступе отказывается. Для хранения информации о пользователях, которым разрешен или запрещен доступ к охраняемым объектам, каждый дескриптор безопасности содержит список управления дискреционным доступом (Discretionary AccessControl List, DACL). Для управления аудитом доступа к объекту в дескрипторе безопасности хранится список управления системным доступом (System Access-Control List, SACL). Общее название для этих списков — списки управления доступом (Access-Control Lists) или сокращенно ACL.



Проверка прав доступа субъекта к объекту производится в момент открытия этого объекта в процессе субъекта.

Субъект - это то, от кого вы защищаете.

Объект - это то, что вы защищаете.

хзхз

***По умолчанию операционная система Windows NT создает три учетных записи:***

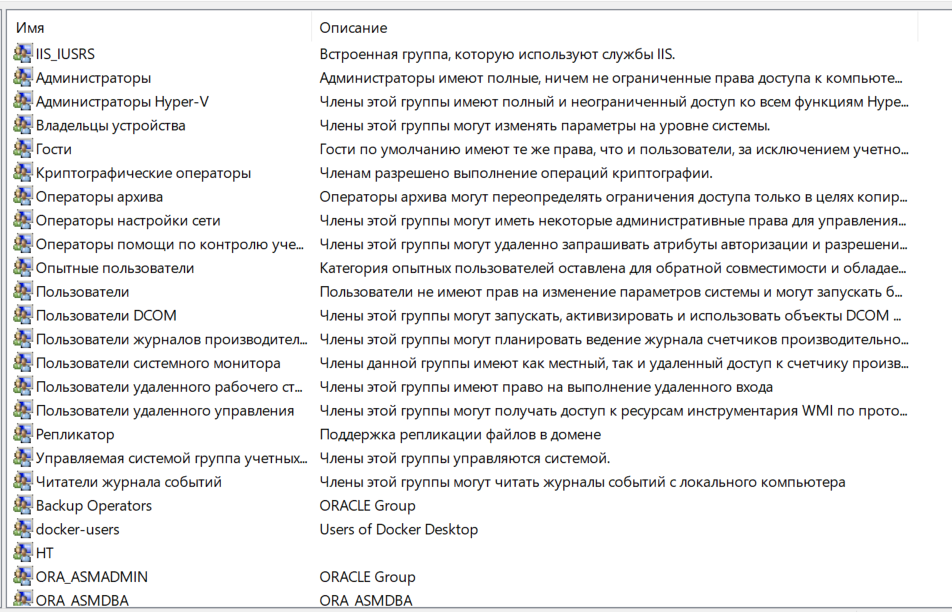
* ***Administrator — администратор;***
* ***Guest — гость;***
* ***System — система.***

***Учетная запись администратора предназначена для управления локальной системой, установленной на одном компьютере. По умолчанию эта учетная запись создается без пароля. Желательно, чтобы в дальнейшем администратор системы изменил имя пользователя и установил пароль для этой учетной записи. Учетную запись администратора системы нельзя заблокировать.***

***Администратор системы может создавать новые учетные записи.***

***Учетная запись гостя предназначена для обычного пользователя системы, который не имеет административных полномочий. Эта учетная запись также создается без пароля. Она не может быть удалена, но может быть переименована. Учетная запись системы используется самой операционной системой для выполнения различных задач, которые требуют аутентификации. Эта учетная запись также имеет административные привилегии.***

**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.

**В операционной системе Windows XP дополнительно создаются учетные записи:**

* **HelpAssistant — используется удаленным экспертом для регистрации на локальном компьютере, чтобы оказать помощь пользователю этого компьютера;**
* **SUPPORT\_xxxxxxxx — предназначен для сервисной поддержки и обслуживания аппаратного и программного обеспечения его производителями в режиме удаленного доступа.**

**Группа (group) это просто набор учетных записей пользователей, которые объединены по какому-либо признаку, например, пользователи одной группы могут работать в одном отделе. 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. Однако администратор системы может изъять некоторые права и полномочия у некоторых членов группы. Максимальное количество групп, в которые может входить пользователь, равно 1000**

**На платформе Windows NT различают три типа групп:**

* **глобальные группы;**
* **локальные группы;**
* **специальные группы.**

**Локальные и глобальные группы создаются администратором системы. Специальные группы создаются системой по умолчанию.**

***Глобальная группа используется для организации пользователей с целью упорядочения их доступа к ресурсам, находящимся в локальной сети вне домена, в котором создана эта группа. По умолчанию система создает следующие глобальные группы: Domain Admins — эта группа включает учетные записи всех администраторов внутри домена; Domain Guests — эта группа включает учетные записи всех гостей домена; Domain Users — эта группа учетных записей всех пользователей домена.***

**Локальная группа используется для организации доступа пользователей к ограниченному множеству ресурсов внутри домена. По умолчанию система создает следующие локальные группы:**

**Account Operators (операторы учетных записей) — члены этой группы могут создавать новые учетные записи пользователей;**

**Administrators (администраторы) — члены этой группы наделяются неограниченными полномочиями администратора системы;**

**Backup Operators (операторы резервирования) — членам этой группы разрешается создавать резервные копии файлов системы;**

**Guests (гости) — эта группа предназначена для ограничения полномочий разовых пользователей системы, члены этой группы имеют такие же права, как и члены группы Users;**

**Print Operators (операторы принтеров) — члены этой группы ответственны за управление принтерами системы: они могут добавлять, удалять и изменять принтеры (на сервере), используемые пользователями сети;**

**Power Users (полномочные пользователи) — члены этой группы обладают широкими правами, которые можно сравнить с административными правами, но не имеют власти над администраторами системы;**

**Replicator (репликаторы) — членам этой группы разрешается выполнять копирование каталогов, содержащих управляющую информацию, между компьютерами локальной сети;**

**Server Operator (операторы сервера) — члены этой группы могут управлять сервером, однако они не имеют полномочий, сравнимых с администраторами системы;**

**Users (пользователи) — члены этой группы являются зарегистрированными пользователями системы, они могут создавать другие локальные группы пользователей и управлять ими.**

***Специальные группы создаются системой для управления доступом к ресурсам. Членство в этих группах предопределено и не может быть изменено. Существуют следующие специальные группы: Creator/Owner (создатели и владельцы) — членами этой группы являются владельцы объектов; Everyone (любые учетные записи) — членами этой группы являются все учетные записи, зарегистрированные в системе, включая анонимную и пустую учетные записи;Interactive (интерактивные пользователи) — членами этой группы являются учетные записи, которые соответствуют пользователям, интерактивно работающим в системе; Network (сетевые пользователи) — членами этой группы являются учетные записи пользователей, которые работают с системой через локальную сеть; System (системные процессы) — эта специальная группа используется самой системой***

**Для создания учетной записи пользователя используется функция NetUserAdd, Отметим, что успешно эту функцию могут выполнить только те пользователи, которые являются администраторами или операторами учетных записей**

**Для получения информации о пользователе из его учетной записи используется функция NetUserGetInfo.**

**Для того чтобы перечислить учетные записи всех пользователей, зарегистрированных на сервере, используется функция NetUserEnum,**

**Для перечисления глобальных групп, которым принадлежит пользователь, используется функция NetUserGetGroups.**

**Для изменения содержимого учетной записи пользователя используется функция NetUserSetInfo.**

**Для изменения пароля пользователя используется функция NetUserChangePassword,**

**Для удаления учетной записи пользователя из базы данных менеджера учетных записей используется функция NetUserDel**

**Для создания учетной записи локальной группы в базе данных менеджера учетных записей используется функция NetLocalGroupAdd,**

**Для получения информации о локальной группе используется функция NetLocalGroupGetInfo. Отметим, что успешно эти функции могут выполнить только пользователи, которые являются администраторами или операторами учетных записей.**

**Для перечисления локальных групп, учетные записи которых зарегистрированны на заданном сервере, используется функция NetLocalGroupEnum. Для изменения имени локальной группы и комментариев о локальной группе используется функция NetLocalGroupSetInfo, для добавления новых членов в локальную группу используется функция NetLocalGroupAddMembers. Под установкой членов локальной группы понимаются следующие действия. Задается список членов, которые должны входить в локальную группу. После этого из локальной группы исключаются все члены, которые не принадлежат этому списку, и включаются только члены из списка. Для установки членов локальной группы используется функция NetLocalGroupSetMembers**

**Для перечисления членов локальной группы используется функция NetLocalGroupGetMembers Для удаления членов локальной группы используется функция NetLocalGroupDelMembers. Для удаления учетной записи локальной группы из базы данных менеджера учетных записей используется функция NetLocalGroupDel**

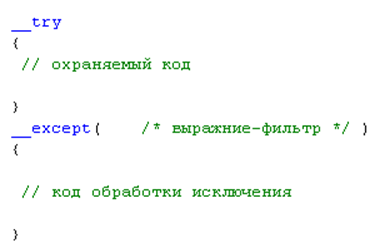
# 13. Структурная обработка ошибок в Windows: программное исключение, программные конструкции для обработки ошибок в Windows, фильтры, возможности API для структурной обработки ошибок, генерация ошибок, финальная обработка исключений.

**Исключение** – событие в программе, произошедшее во время ее выполнения, в результате которого нормальное выполнение программы становится не возможным.

В операционных системах Windows для этой цели предназначен механизм структурной обработки исключений (structured exception handling, SEH).

Смысл механизма структурной обработки исключений заключается в следующем. В программе выделяется блок программного кода, в котором может произойти исключение. Такой блок кода называется ***фреймом***, а сам код называется ***охраняемым кодом***. Затем, после фрейма вставляется программный блок, который обрабатывает происшедшее исключение. Этот блок называется ***обработчиком исключения***. После обработки исключения управление передается первой инструкции, следующей за обработчиком исключения.

Ключевое слово **\_\_try** отмечает фрейм, а ключевое слово **\_\_except** отмечает обработчик исключения.



Здесь выражение-фильтр принимает одно из следующих значений:

**EXCEPTION\_EXECUTE\_HANDLER** — управление передается обработчику исключений;

**EXCEPTION\_CONTINUE\_SEARCH** — система продолжает поиск обработчика исключения;

**EXCEPTION\_CONTINUE\_EXECUTION** — система передает управление в точку прерывания программы.

В выражении фильтра допускается использование функций **GetExceptionCode** и **GetExceptionInformation**, которые предоставляют информацию о происшедшем исключении.

Концептуально механизм структурной обработки исключений в Windows немного отличается от механизма обработки исключений, принятого в языке программирования С++. Дело в том, что механизм структурной обработки исключений был разработан раньше, чем принят стандарт языка С++. Кроме того, в отличие от языка программирования С++ *механизм структурной обработки исключений ориентирован не только на обработку программных исключений, но и на обработку аппаратных исключений.*

Получение кода исключения

Получить код происшедшего исключения можно при помощи функции **GetExceptionCode** (вызывается только в том случае, если исключение произошло), которая имеет следующий прототип:

DWORD GetExceptionCode(VOID);

Функция GetExceptionCode возвращает одно из следующих значений:

EXCEPTION\_ACCESS\_VIOLATION — попытка чтения или записи в виртуальную память без соответствующего права доступа;

EXCEPTION\_BREAKPOINT — встретилась точка останова;

EXCEPTION\_FLT\_DIVIDE\_BY\_ZERO — попытка деления на ноль в операции с плавающей точкой;

EXCEPTION\_FLT\_INVALID\_OPERATION — ошибка в операции с плавающей

точкой, для которой не предусмотрены другие коды исключения;

EXCEPTION\_FLT\_OVERFLOW — при выполнении операции с плавающей точкой произошло переполнение;

EXCEPTION\_INT\_DIVIDE\_BY\_ZERO — попытка деления на ноль в операции

с целыми числами;

EXCEPTION\_INT\_OVERFLOW — при выполнении операции с целыми

функции фильтра

Если требуется более детально обработать информацию об исключении, то в выражении-фильтре используют функцию, которая в этом случае называется функцией фильтра.

*В функции фильтра не разрешается вызывать функции GetExceptionCode и GetExceptionInformation, однако эти функции могут вызываться для инициализации параметров функции фильтра.*

Получение информации об исключении

Более подробную информацию об исключении можно получить при помощи вызова функции **GetExceptionInformation**, которая имеет следующий прототип:

LPEXCEPTION\_POINTERS GetExceptionInformation(VOID);

Эта функция возвращает указатель на структуру типа:

typedef struct **EXCEPTION\_POINTERS** {

PEXEPTION\_RECORD ExceptionRecord;

PCONTEXT Context;

} EXCEPTION\_POINTERS, \*PEXCEPTION\_POINTERS;

которая, в свою очередь, содержит два указателя: ExceptionRecord и Context на структуры типа **EXCEPTION\_RECORD и CONTEXT** соответственно.

В структуру типа CONTEXT система записывает содержимое всех регистров процессора на момент исключения.

Структура типа EXCEPTION\_RECORD имеет следующий формат:

typedef struct EXCEPTION\_RECORD {

DWORD ExceptionCode; // код исключения

DWORD ExceptionFlags,

struct \_EXCEPTION\_RECORD \*ExceptionRecord;

PVOID ExceptionAddress,

DWORD NumberParameters,

ULONG\_PTR ExceptionInformation[EXCEPTION\_MAXIMUM\_PARAMETERS];

} EXCEPTION\_RECORD, \*PEXCEPTION\_RECORD;

В нее система записывает информацию об исключении.

Поле *ExceptionFlags* может принимать одно из двух значений:

0 — которое обозначает, что после обработки исключения возможно возобновление выполнения программы;

EXCEPTION\_NONCONTINUABLE — которое обозначает, что после обработки исключения возобновление выполнения программы невозможно.

Поле *ExceptionAddress* содержит адрес инструкции в программе, на которой произошло исключение.

Поле *NumberParameters* содержит количество параметров, заданных в поле ExceptionInformation, которое является последним в этой структуре.

!Сделаем важное замечание о том, что функция GetExceptionInformation может вызываться только в выражении фильтра. Поэтому эта функция вызывается всегда только в том случае, если исключение произошло. Кроме того, структуры типа EXCEPTION\_POINTERS, EXCEPTION\_RECORD и CONTEXT действительны только на время вычисления выражения-фильтра. Чтобы использовать содержимое структур типа EXCEPTION\_RECORD и CONTEXT в блоке обработки исключения, его нужно сохранить в объявленных в программе переменных такого же типа.

Генерация программных исключений  
Механизм структурной обработки исключений в Windows позволяет генерировать программные исключения при помощи функции RaiseException, которая имеет следующий прототип:

VOID **RaiseException**(

DWORD dwExceptionCode, // код исключения

DWORD dwExceptionFlags, // флаг возобновляемого исключения

DWORD nNumberOfArguments, // количество аргументов

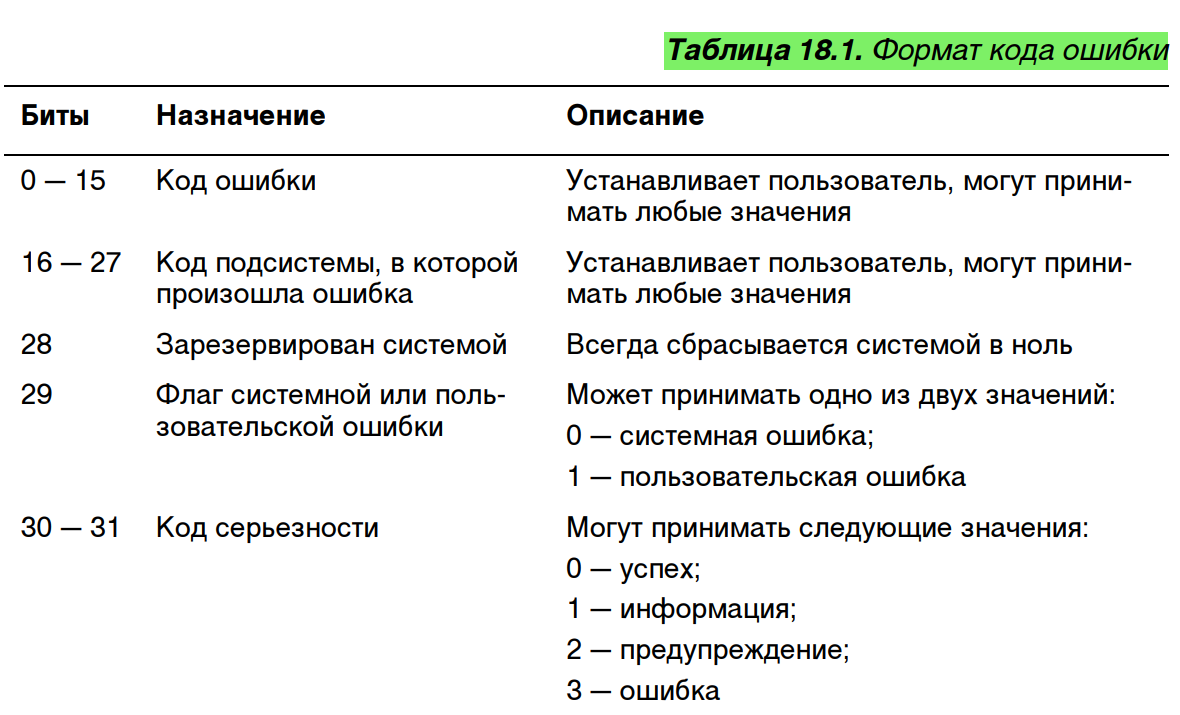
CONST ULONG\_PTR \*lpArgumens // массив аргументов

);

Параметр *dwExceptionFlag* может принимать одно из двух значений: 0 или EXCEPTION\_NONCONTINUABLE. Значение 0 означает, что после исключения возможно восстановление программы, вызвавшей это исключение. Если же параметр dwExceptionFlag имеет значение EXCEPTION\_NONCONTINUABLE, то выполнение программы не может быть возобновлено после исключения. В случае попытки возобновления программы система сгенерирует исключение с кодом EXCEPTION\_NONCONTINUABLE\_EXCEPTION.

Параметр *nNumberOfArguments* указывает количество параметров, передаваемых функции фильтра из блока обработки исключения. Это значение не может превышать величину EXCEPTION\_MAXIMUM\_PARAMETERS.

Параметр *lpArguments* задает параметры, которые передаются в функцию фильтра из блока обработки исключения. Это значение может быть установлено в NULL. В этом случае параметр nNumberOfArguments игнорируется.

Эта функция выбрасывает исключение с кодом, заданным параметром dwExceptionCode. Отметим, что код исключения следует формировать в соответствии с правилами, принятыми в операционных системах Windows.  


Если в программе произошло исключение, для которого не существует обработчика исключений, то в этом случае вызывается функция-фильтр системного обработчика исключений.

Обработка исключений с плавающей точкой

По умолчанию система отключает все исключения с плавающей точкой. Поэтому если при выполнении операции с плавающей точкой было получено число, которое не входит в диапазон представления чисел с плавающей точкой, то в результате система вернет NAN или INFINITY в случае слишком малого или слишком большого числа соответственно. Чтобы включить режим генерации исключений с плавающей точкой нужно изменить состояние слова, управляющего обработкой операций с плавающей точкой. Это можно сделать при помощи функции **\_controlfp**, которая имеет следующий прототип: unsigned int \_controlfp(

unsigned int new,

unsigned int mask

);

Для разрешения генерации исключений с плавающей точкой параметр mask должен принимать значение \_MCW\_EM

В параметре new для управления исключениями можно сбрасывать или устанавливать следующие значения:

\_EM\_INVALID — исключение EXCEPTION\_FLT\_INVALID\_OPERATION;

\_EM\_DENORMAL — исключение EXCEPTION\_FLT\_DENORMAL\_OPERAND;

\_EM\_ZERODIVIDE — исключение EXCEPTION\_FLT\_DIVIDE\_BY\_ZERO;

\_EM\_OVERFLOW — исключение EXCEPTION\_FLT\_OVERFLOW;

\_EM\_UNDERFLOW — исключение EXCEPTION\_FLT\_UNDERFLOW;

\_EM\_INEXACT — исключение EXCEPTION\_FLT\_INEXACT\_RESULT.

Обработка вложенных исключений

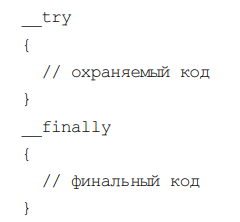
При использовании структурной обработки исключений возможно вкладывать блоки \_\_try и \_\_except в другой блок \_\_try. В этом случае если функция-фильтр внутреннего блока \_\_except возвращает значение EXCEPTION\_ CONTINUE\_SEARCH, то система удаляет все локальные объекты, принадлежащие текущим блокам \_\_try и \_\_except, и продолжает поиск обработчика исключений во внешних блоках \_\_try и \_\_except. Такая очистка стека от локальных объектов называется *глобальной раскруткой стека* или просто *раскруткой стека*

Встаивание SHE в механизм исключений С++

Наверное, это не надо, но в книге на странице 349 есть

Финальная обработка исключений

только теперь за блоком \_\_try следует код, который заключается в блок \_\_finally. Система гарантирует, что при любой передаче управления из блока \_\_try, независимо от того, произошло или нет исключение внутри этого блока, предварительно управление будет передано блоку \_\_finally. Такой способ обработки исключений называется финальная обработка исключений



Финальная обработка исключений используется для того, чтобы при любом исходе исполнения блока \_\_try освободить ресурсы, которые были захвачены внутри этого блока. Такими ресурсами могут быть память, файлы, критические секции и т. д.

**Недостатком** такой работы блока \_\_finally является то, что инструкция delete будет выполняться в любом случае, независимо от того, произошло исключение в блоке \_\_try или нет. Чтобы избежать такой ситуации, нужно проверить, как завершился блок \_\_try — нормально или нет.

проверка как завершился блок \_\_try – не надо

Управление из блока \_\_try может быть передано одним из следующих способов:

* нормальное завершение блока;
* выход из блока при помощи управляющей инструкции \_\_leave;
* выход из блока при помощи одной из управляющих инструкций return, break, continue или goto языка программирования С++;
* передача управления обработчику исключения.

В первых двух случаях считается, что блок \_\_try завершился нормально, а в последних двух случаях — ненормально.

Для того чтобы определить, как завершился блок \_\_try, используется функция **AbnormalTermination**, которая имеет следующий прототип:

BOOL AbnormalTermination(VOID);

В случае если блок \_\_try завершился ненормально, эта функция возвращает ненулевое значение, а в противном случае — значение FALSE. Используя функцию AbnormalTermination, ресурсы, захваченные в блоке \_\_try, можно освобождать только в том случае, если блок \_\_try завершился ненормально.

Сочетание блоков finally и except

Один try-блок может иметь только один блок finally или только один блок except, но не может иметь оба указанных блока одновременно. Поэтому нижеприведенный код вызовет появление ошибок на стадии компиляции.

\_\_try {

/\* Блок контролируемого кода. \*/

}

\_\_except (filter\_expression) {

/\* Блок обработчика исключений. \*/

}

\_\_finally {

/\* Так делать нельзя! Это приведет к ошибке на стадии компиляции. \*/

}

Вместе с тем, допускается вложение одного блока в другой, что используется довольно часто.

Обработчики завершения: - не надо

завершение процессов и потоков Обработчики завершения не выполняются, если выполнение процесса или потока было прекращено независимо от того, было ли это инициировано самим процессом путем использования функций ExitProcess или ExitThread, или вызвано извне, например, инициировано вызовом функций TerminateProcess или TerminateThread из другого места в программе. Поэтому ни одна из этих функций не должна вызываться процессом или потоком внутри блоков try… except или try…finally.

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

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

**Консолью называется интерфейс, который используется приложением для ввода-вывода текстовой информации. В этом случае приложение, которое использует консоль для обмена данными с пользователем, называется консольным приложением. Консольные приложения применяются главным образом в системном программировании для разработки различных сервисов и для обработки неустранимых ошибок, возникающих при работе графического приложения.**

**Консоль состоит из одного входного буфера и одного или нескольких буферов экрана. Входной буфер содержит информацию о событиях ввода. Каждое событие ввода описывается записью. Все записи упорядочены в очередь, которая хранится в буфере ввода.**

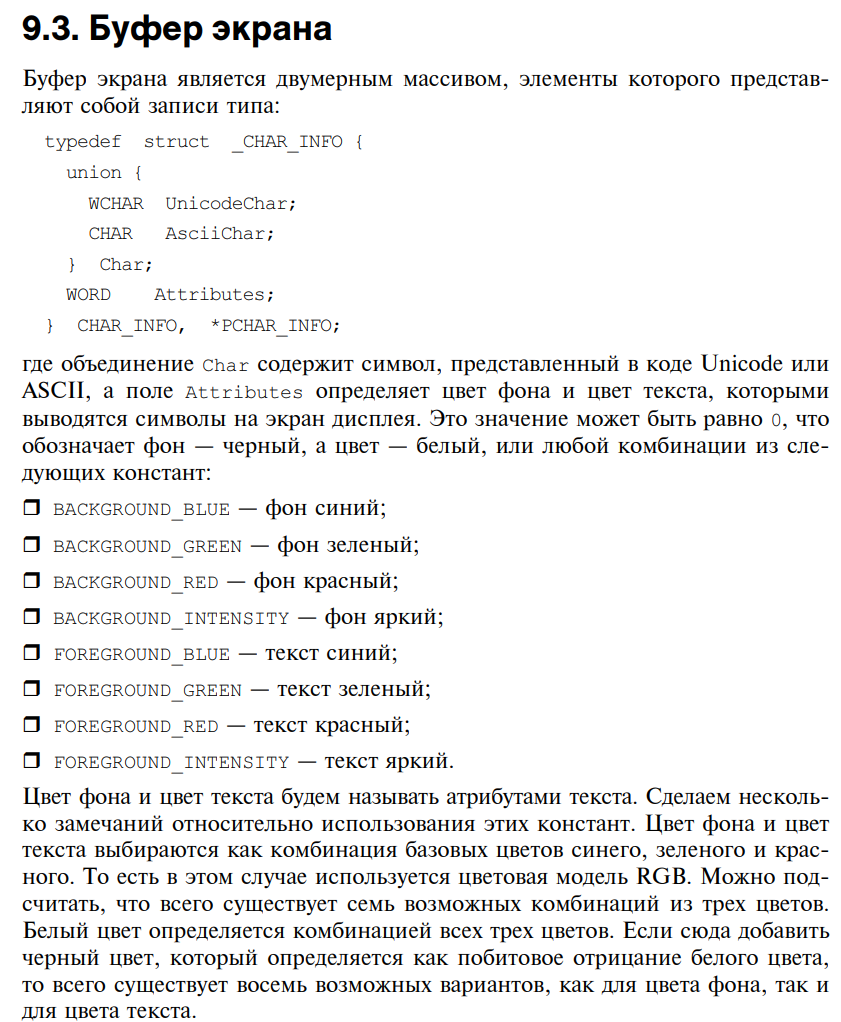
**Буфер экрана содержит информацию для вывода в окно приложения и является двумерным массивом, который содержит символы и данные о цвете.**

**Консоль обеспечивает два уровня ввода-вывода текстовой информации: высокий и низкий. Функции высокого уровня обеспечивают ввод-вывод символов с консоли, игнорируя остальные события. Функции низкого уровня обеспечивают обработку всех событий, связанных с консольным приложением**

**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.**

**Два последних события ввода, связанные с фокусом ввода и меню, обрабатываются системой и должны игнорироваться приложением.**

****

***Процесс может быть связан только с одной консолью*.**

**Новая консоль может создаваться одним из следующих двух способов.**

**Первый способ заключается в том, что при создании консольного процесса командой CreateProcess нужно установить флаг CREATE\_NEW\_CONSOLE. Отметим, что в этом случае, если консольный процесс создается из консольного приложения, а указанный флаг не установлен, то новый процесс присоединяется к консоли родительского процесса.**

**Второй способ заключается в использовании функции AllocConsole. Приложение освобождает консоль посредством вызова функции FreeConsole**

**При создании новой консоли система создает три дескриптора, которые обозначаются STDIN, STDOUT, STDERR и называются соответственно стандартными дескрипторами ввода, вывода и ошибки. Дескриптор STDIN связывается с буфером ввода, а дескрипторы STDOUT и STDERR связываются с буфером экрана. Эти дескрипторы используются в функциях, предназначенных для работы с консолью.**

**Дескриптор окна консоли можно получить, вызвав функцию GetConsoleWindow.**

**Для чтения заголовка окна консоли используется функция GetConsoleTitle.**

**Максимальный размер окна консоли можно определить, вызвав функцию GetLargestConsoleWindowSize**

**Буфер экрана может быть создан посредством вызова функции CreateConsoleScreenBuffer. Параметры буфера экрана можно определить с помощью функции GetConsoleScreenBufferInfo. Информацию о положении и видимости курсора можно получить, используя функцию GetConsoleCursorInfo**

**К функциям ввода-вывода высокого уровня относятся следующие функции: WriteConsole, ReadConsole, WriteFile и ReadFile.**

**Для чтения строки символов из входного буфера консоли экрана используется функция ReadConsole. Для записи строки символов в буфер экрана используется функция WriteConsole**

**В случае с консолью функции WriteFile, ReadFile, WriteConsole и ReadConsole читают и записывают символы потоком. Функции ReadFile и WriteFile работают только с символами, заданными в кодировке ASCII. Функции ReadConsole и WriteConsole отличаются от файловых функций только тем, что работают также с символами, заданными в кодировке Unicode. Кроме того, эти функции не обрабатывают управляющие символы при вводе-выводе на консоль.**

**Функции ввода низкого уровня работают непосредственно с записями входного буфера консоли. К ним относятся функции ReadConsoleInput, PeekConsoleInput, WriteConsoleInput, GetNumberOfConsoleInputEvents и FlushConsoleInputBuffer, работу с которыми мы сейчас рассмотрим. Для чтения записей из входного буфера используется функция ReadConsoleInput, для чтения записей из входного буфера консоли, не удаляя их оттуда, используется функция PeekConsoleInput.**

**Функция WriteConsoleInput предназначена для записи событий ввода во входной буфер консоли Для чтения количества записей, находящихся во входном буфере консоли используется функция GetNumberOfConsoleInputEvents,**

**Для определения количества кнопок у мыши используется функция GetNumberOfConsoleMouseButton**

**Функции вывода низкого уровня работают непосредственно с элементами буфера экрана. Эти функции можно разбить на три группы:**

**Чтение и запись последовательности символов:**

* + **ReadConsoleOutputCharacter — чтение последовательности символов из буфера экрана;**
  + **WriteConsoleOutputCharacter — запись последовательности символов в буфер экрана.**

**Заполнение буфера экрана заданным символом:**

* + **FillConsoleOutputCharacter — заполнение буфера экрана.**

**Чтение и запись прямоугольных областей символов:**

* + **ReadConsoleOutput — чтение прямоугольной области символов из буфера экрана;**
  + **WriteConsoleOutput — запись прямоугольной области символов в буфер 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.

# 15. Windows-сервисы: определение, назначение, применение, API.

**Смелов**

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

Обычно сервисы выполняют служебные функции, необходимые для какого-нибудь приложения. Иногда сервисы используются для доступа к внешним устройствам, они называются драйверами. В виде сервисов реализованы сетевые службы, такие как DNS, FTP, Telnet, DHCP и другие. Сам сервис может быть как консольным приложением, так и приложением с графическим интерфейсом.

Принципы разработки Windows-сервиса

Основное назначение функции **ServiceMain** – определить текущее состояние сервиса (структура ServiceStatus и функция **SetServiceStatus**), а также зарегистрировать обработчик управляющих команд (функция **RegisterServiceCtrlHandler**).

Обработчик управляющих команд – это функция, имеющая обычно имя **ServiceHandler** и предназначенная для обработки команд, поступающих сервису, например, через mmc-консоль. Именно эта функция определяет перечень команд, на которые реагирует сервис

**Сервис – это процесс, который выполняет служебные функции. То есть сервис это такая программа, которая запускается при загрузке операционной системы или в процессе ее работы по специальной команде и заканчивает свою работу при завершении работы операционной системы или по специальной команде.**

**Обычно сервисы выполняют определенные служебные функции, необходимые для работы приложений или какого-то конкретного приложения. Примером сервиса может служить фоновый процесс, который обеспечивает доступ к базе данных — такие сервисы также называются серверами. Другой тип сервисов — это программы, обеспечивающие доступ к внешним устройствам, такие сервисы называются драйверами. Как сервис также может быть реализован процесс, отслеживающий работу некоторого приложения, такие сервисы также называются мониторами.**

**Управляет работой сервисов специальная программа операционной системы, которая называется менеджер сервисов (Service Control Manager, SCM). Ниже перечислены функции, которые выполняет менеджер сервисов:**

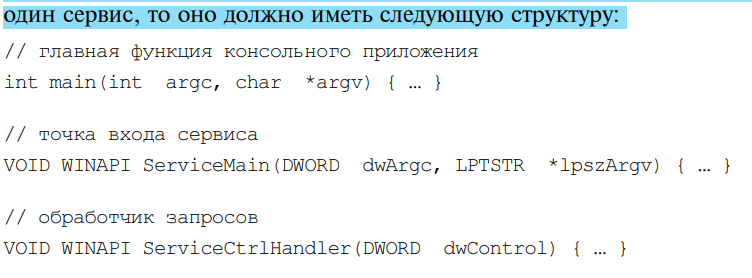
* **поддержка базы данных установленных сервисов;**
* **запуск сервисов при загрузке операционной системы;**
* **поддержка информации о состоянии работающих сервисов;**
* **передача управляющих запросов работающим сервисам;**
* **блокировка и разблокирование базы данных сервисов.**

**Для того чтобы менеджер сервисов знал о существовании определенного сервиса — его нужно установить.**

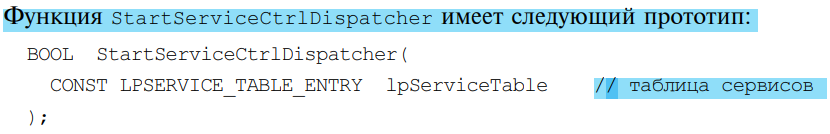
**Информация обо всех установленных хранится в реестре операционной системы Windows под ключом HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services.**

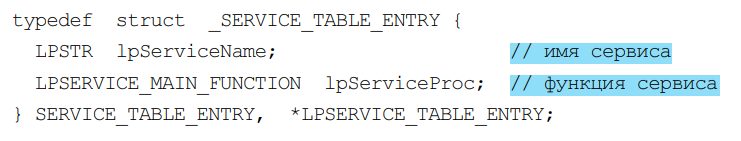
**Сервис может запускаться как операционной системой при загрузке, так и программно — из приложения.**

**Так как сервисы работают под управлением менеджера сервисов, то они должны удовлетворять определенным соглашениям, которые определяют интерфейс сервиса.** Сам сервис может быть как консольным приложением, так и приложением с графическим интерфейсом**. Кроме того, каждый сервис *должен содержать две функции обратного вызова*, которые вызываются операционной системой. Одна из этих функций определяет точку входа сервиса, т. е., собственно, и является сервисом, а вторая — должна реагировать на управляющие сигналы от операционной системы**

****

**1. Главной задачей функции main является запуск диспетчера сервиса, который является потоком и управляет этим сервисом.** StartServiceCtrlDispatcher **Диспетчер сервиса получает управляющие сигналы от менеджера сервисов по именованному каналу и передает эти запросы функции ServiceCtrlHandler, которая обрабатывает эти управляющие запросы. Если в приложении несколько сервисов, то для каждого сервиса запускается свой диспетчер и для каждого диспетчера определяется своя функция обработки управляющих запросов, которая выполняется в контексте соответствующего диспетчера сервисов. Запуск диспетчеров сервисов выполняется при помощи вызова функции StartServiceCtrlDispatcher. Для подключения обработчика запросов к сервису используется функция RegisterServiceCtrlHandler.**

****

****

**2. Функция, определяющая точку входа сервиса. Если определяется только один сервис, то эта функция обычно называется ServiceMain, хотя возможны и другие, более подходящие по смыслу имена точек входа сервисов. Если же в приложении определяется несколько сервисов, то естественно каждый из них должен иметь свое имя. Эта функция содержит два параметра, которые аналогичны параметрам функции main консольного приложения. Параметр dwArgc содержит количество аргументов в массиве lpszArgv, а сам этот массив содержит адреса строк. Причем первый из этих адресов указывает на строку, содержащую имя сервиса, а последующие аргументы передаются из функции StartService.**

!Для запуска обработчика управляющих команд от менеджера сервисов сервис должен использовать функцию RegisterServiceCtrlHandler,

RegisterServiceCtrlHandler(ServiceName, ServiceHandler)

! Для изменения состояния сервиса используется функция SetServiceStatus,

SetServiceStatus(hServiceStatusHandle, &serviceStatus

**3. Функция, определяющая обработчик управляющих запросов: Если определяется только один сервис, то эта функция обычно называется ServiceCtrlHandler. Если же в приложении определяется несколько сервисов, то естественно, что обработчик запросов для каждого сервиса должен иметь свое имя. Эта функция содержит только один параметр, который содержит код управляющего сигнала**

***Как уже было сказано, главной задачей функции main является запуск диспетчера сервиса для каждого из сервисов. Для запуска диспетчера используется функция StartServiceCtrlDispatcher, которая должна быть вызвана в течение 30 секунд с момента запуска программы main. Если в течение этого промежутка времени функция StartServiceCtrlDispatcher вызвана не будет, то последующий вызов этой функции закончится неудачей. Поэтому всю необходимую инициализацию сервиса нужно делать в самом сервисе, т. е. в теле функции ServiceMain.***

**Функция ServiceMain должна выполнить следующую последовательность действий:**

**1. Немедленно запустить обработчик управляющих команд от менеджера сервисов, вызвав функцию RegisterServiceCtrlHandler.**

**~~2. Установить стартующее состояние сервиса SERVICE\_START\_PENDING посредством вызова функции SetServiceStatus.~~**

**3. Провести локальную инициализацию сервиса.**

**4. Установить рабочее состояние сервиса SERVICE\_RUNNING посредством вызова функции SetServiceStatus.**

**5. Выполнять работу сервиса, учитывая состояния сервиса, которые могут изменяться обработчиком управляющих команд от менеджера сервисов. 6. После перехода в состояние останова SERVICE\_STOPPED выполнить освобождение захваченных ресурсов и закончить работу**

**Работа с сервисом**

**Для установки связи с менеджером сервисов и открытия доступа к базе данных сервисов используется функция OpenSCManager. После окончания работы с менеджером сервисов или сервисом нужно закрыть его дескриптор. Для этой цели предназначена функция CloseServiceHandle**

**Для установки сервисов в базу данных используется функция CreateService, Напомним, что после окончания работы с сервисом нужно закрыть его дескриптор, используя для этого функцию CloseServiceHandle. Отметим, что эта функция не уничтожает сервис, а только разрывает связь прикладной программы с этим сервисом. Для удаления сервиса из базы данных нужно использовать функцию DeleteService.**

**Для открытия доступа к уже установленному сервису используется функция OpenService. Для запуска сервиса используется функция StartService.**

**Для определения состояния сервиса используется функция QueryServiceStatus. Для определения конфигурации сервиса используется функция QueryServiceConfig. Для изменения конфигурации сервиса можно использовать функцию ChangeServiceConfig**

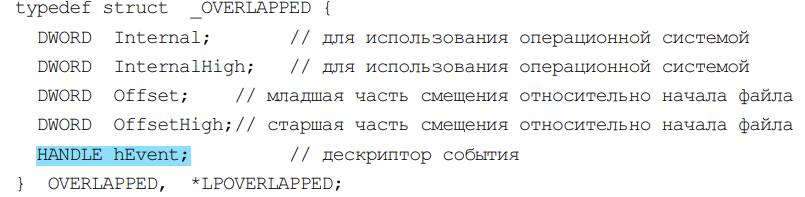
**Назовем внутренним именем сервиса то имя, под которым сервис хранится в базе данных сервисов, а также используется менеджером сервисов для ссылок на этот сервис. Узнать внутреннее имя сервиса можно при помощи функции GetServiceKeyName**

**Приложение может послать сервису управляющую команду, которая будет передана обработчику управляющих команд сервиса. Для этой цели используется функция ControlService**

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

В книжке

При синхронной записи данных в файл поток, выдавший команду записи, блокируется до тех пор, пока данные не будут записаны в файл или буфер. При асинхронной записи данных в файл, после выдачи команды на запись данных, поток не блокируется, а продолжает свою работу. Соответственно, при синхронном чтении данных из файла поток, выдавший команду чтения, блокируется до тех пор, пока данные не будут прочитаны из файла. При асинхронном чтении данных такой блокировки не происходит, и поток продолжает свое исполнение. В операционных системах Windows асинхронный ввод-вывод также называется перекрывающимся (overlapped) вводом-выводом. Для асинхронного ввода-вывода данных в операционных системах Windows используются те же функции ReadFile и WriteFile, что и для синхронного ввода-вывода. Однако в этом случае файл должен быть открыт в режиме FILE\_FLAG\_OVERLAPPED



**Первое, что необходимо сделать для организации асинхронного ввода/вывода, будь то перекрывающегося или расширенного, — это установить атрибут перекрывания (overlapped attribute) для файлового или иного дескриптора. Для этого при вызове CreateFile или иной функции, в результате которого создается файл, именованный канал или иной дескриптор, следует указать флаг FILE\_FLAG\_OVERLAPPED. Затем с помощью структуры OVERLAPPED (указываемой, например, параметром lpOverlapped функции ReadFile) можно указывать следующую информацию:**

**• Позицию в файле (64 бита), с которой должно начинаться выполнение операции чтения или записи.**

**• Событие (сбрасываемое вручную), которое будет переходить в сигнальное состояние по завершении соответствующей операции.**

**При выполнении вызова функций ввода/вывода это событие сразу же сбрасывается системой (устанавливается в несигнальное состояние). Когда операция ввода/вывода завершается, событие устанавливается в сигнальное состояние и остается в нем до тех пор, пока не будет использовано другой операцией ввода/вывода. Событие должно быть сбрасываемым вручную, если его перехода в сигнальное состояние могут ожидать несколько потоков (хотя в наших примерах используется всего один поток), и на момент завершения операции они могут не находиться в состоянии ожидания.**

**Эти структуры выступают в качестве необязательных параметров при вызове четырех приведенных ниже функций, которые могут блокироваться при завершении операций: ReadFile WriteFile TransасtNamedPipe ConnectNamedPipe**

**Возникает вопрос: как поток узнает о завершении асинхронной операции чтения или записи? Это можно сделать двумя способами. Первый способ заключается в том, что для этого можно использовать дескриптор файла, который устанавливается в несигнальное состояние после начала каждой асинхронной операции записи или чтения и переходит в сигнальное состояние после завершения асинхронной операции записи или чтения. Однако этот способ не может быть использован, если с одним файлом работает несколько асинхронных операций ввода-вывода. Так как в этом случае невозможно определить какая из асинхронных операций ввода/вывода завершилась. Второй способ заключается в использовании специального события, которое устанавливается операционной системой в сигнальное состояние при завершении асинхронной операции ввода-вывода. Дескриптор этого события должен находиться в структуре типа OVERLAPPED, адрес которой передается в функции асинхронного ввода-вывода.**

**Выполнить асинхронную запись данных можно при помощи функции WriteFile При асинхронной записи данных в файл эта функция возвращает ненулевое значение в том случае, если запись данных в файл уже завершилась до выхода из функции WriteFile. Если же функция возвращает FALSE, то нужно проверить код последней ошибки путем вызова функции GetLastError. Если эта функция возвращает значение ERROR\_IO\_PENDING, то это значит, что операция вывода данных еще не закончилась. *И тут и ниже передается эта структура типа оверлаппед последним параметром***

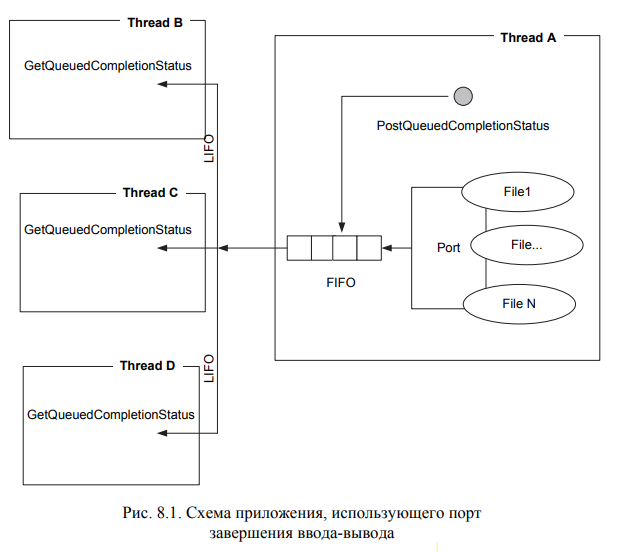
**Выполнить асинхронное чтение данных можно при помощи функции ReadFile При асинхронном чтении данных из файла эта функция возвращает ненулевое значение в том случае, если чтение данных из файла уже завершилось до выхода из функции ReadFile. Если же функция возвращает значение FALSE, то нужно проверить код последней ошибки путем вызова функции GetLastError. Если эта функция возвращает значение ERROR\_IO\_PENDING, то это значит, что операция ввода данных еще не закончилась.**

**Состояние, в котором находится асинхронная операция ввода-вывода, можно определить посредством функции GetOverlappedResult. Отменить асинхронную операцию передачи данных, которая еще не завершилась, можно посредством функции CancelIo. Возможен другой подход к оповещению потока о завершении асинхронной операции записи или чтения данных из файла. В этом случае операционная система не устанавливает событие, дескриптор которого задан в структуре типа OVERLAPPED, а, по завершении асинхронной операции ввода-вывода, вызывает функцию, которая называется процедурой завершения ввода-вывода (FileIOCompletionRoutine). Такой подход к оповещению завершения асинхронных операций ввода-вывода используется в функциях WriteFileEx и ReadFileEx. При этом отметим, что процедура завершения ввода-вывода вызывается только в том случае, если поток находится в настороженном состоянии. В противном случае процедура завершения ввода-вывода включается в очередь асинхронных процедур потока. Кроме того, процедуры завершения ввода-вывода могут вызываться в порядке отличном от порядка завершения асинхронных операций ввода-вывода.**

# 17. Порты завершения ввода/вывода: назначение, применение, API.

Смелов

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



Поток A использует порт завершения, связанный с дескрипторами N файлов. Поток в определенном порядке выполняет операции асинхронного ввода или вывода для этих файлов. Порт отслеживает завершение операций ввода или вывода для связанных с ним файлов. *При завершении каждой операции порт формирует очередной элемент выходной FIFO-очереди*. Элемент очереди содержит информацию о завершенной операции.

Потоки B, C и D предназначены для циклического считывания элементов выходной очереди порта. При получении из очереди элемента поток может определить файл, для которого была выполнена операция, значение текущей позиции файла и другую информацию.

ключ – (идентификатор) файла

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.

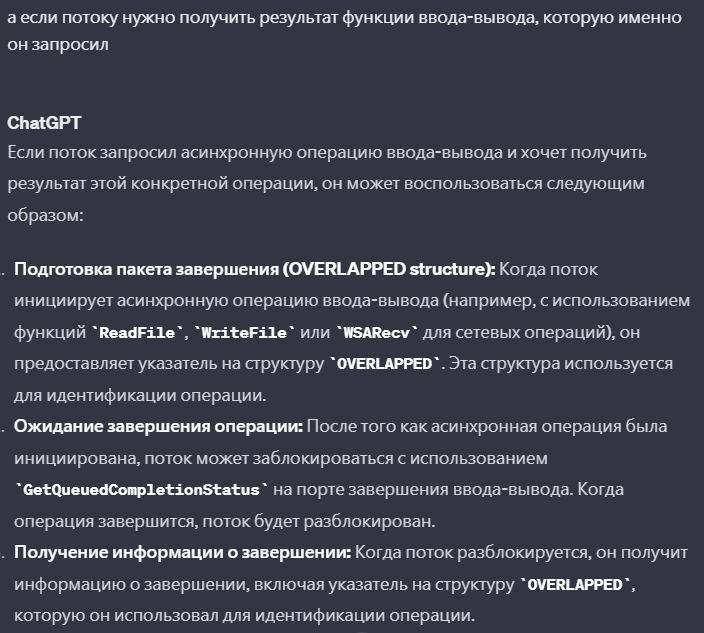
4. Запустить цикл операций асинхронного ввода или вывода для связанных файлов

**Сайт Майкрософт – можно не читать**

**Как работают порты завершения ввода-вывода**

Функция [**CreateIoCompletionPort**](https://learn.microsoft.com/ru-ru/windows/win32/fileio/createiocompletionport) создает порт завершения ввода-вывода и связывает с ним один или несколько дескрипторов файлов. Когда асинхронная операция ввода-вывода для одного из этих дескрипторов файлов завершается, пакет завершения ввода-вывода помещается в очередь в порядке FIFO в соответствующий порт завершения ввода-вывода. Одним из эффективных способов использования этого механизма является объединение точки синхронизации для нескольких дескрипторов файлов в один объект, хотя существуют и другие полезные приложения. Обратите внимание, что, хотя пакеты помещаются в очередь в порядке FIFO, они могут быть выведены из очереди в другом порядке.

Поток (созданный потоком main или сам поток main) использует функцию [**GetQueuedCompletionStatus**](https://learn.microsoft.com/ru-ru/windows/win32/api/ioapiset/nf-ioapiset-getqueuedcompletionstatus), чтобы ждать, пока пакет завершения будет помещен в очередь на порт завершения ввода-вывода, а не ожидает непосредственного завершения асинхронного ввода-вывода. Потоки, которые блокируют выполнение через порт завершения ввода-вывода, освобождаются в порядке последнего выхода (LIFO), а следующий пакет завершения извлекается из очереди FIFO порта завершения ввода-вывода для этого потока. Это означает, что при выпуске пакета завершения в поток система освобождает последний (последний) поток, связанный с этим портом, передав ему сведения о завершении для самого старого завершения ввода-вывода.



один поток может быть связан не более чем с одним портом завершения ввода-вывода.

Книжка

**Порт завершения ввода-вывода** — это, фактически, объект синхронизации, который оповещает параллельно работающие потоки о завершении асинхронных операций доступа к файлам или именованным каналам.

Для работы порта завершения ввода-вывода к нему необходимо подключить файлы, которые должны быть открыты в режиме асинхронного доступа. Причем для каждого файла устанавливается свой номер, который в этом случае называется *ключом завершения*.

При завершении асинхронной операции доступа к файлу операционная система посылает в порт завершения пакет, который содержит информацию о завершившейся асинхронной операции ввода-вывода, содержащую и ключ файла, для которого завершилась эта операция.

Порт содержит очередь пакетов, оповещающих о завершении асинхронных операций ввода-вывода. Обслуживается эта очередь по алгоритму FIFO (first in — first out), т. е. вошедший первым — выходит первым.

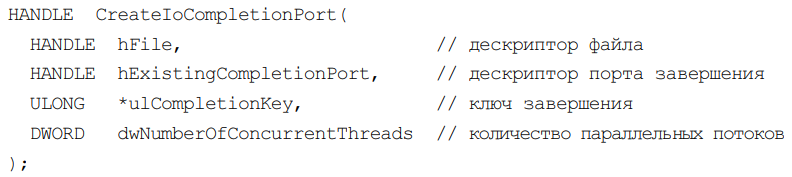
Для создания портов завершения ввода-вывода и подключения к нему файлов в операционных системах Windows используется функция CreateIoCompletionPort.

***Порт завершения ввода-вывода удаляется, когда закрывается его дескриптор***

Поток может узнать о том, что завершилась асинхронная операция ввода-вывода, вызвав функцию , в которой указывается дескриптор порта. Если очередь пакетов порта завершения не пуста, то поток получает пакет из этой очереди. В противном случае поток блокируется. *Каждый порт содержит очередь заблокированных потоков, которая обслуживается по алгоритму LIFO (lst in — first out), т. е. вошедший последним — выходит первым.*

При этом отметим, что ***порт завершения может одновременно обслуживать только заданное количество потоков, которое определяется одним из параметров функции CreateIoCompletionPort***. Остальные потоки, которые превысили этот предел, блокируются системой, до тех пор, пока количество обслуживаемых потоков не станет меньше заданного предела.

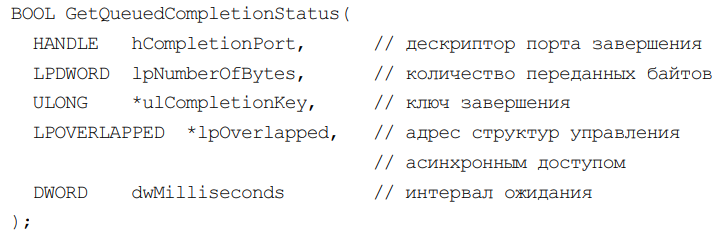
Для создания порта завершения ввода-вывода и подключения файлов к этому порту используется функция CreateIoCompletionPort



В параметре ulCompletionKey должен быть задан адрес числа без знака, которое будет обозначать номер файла. Это число называется *ключом завершения* и используется для идентификации файла, для которого завершилась асинхронная операция ввода-вывода.

Каждый новый вызов функции CreateIoCompletionPort задает новое максимальное количество потоков, которые могут одновременно подключиться к порту завершения ввода-вывода.

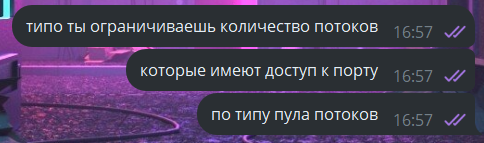
Для получения пакета из порта завершения ввода-вывода используется функция GetQueuedCompletionStatus,



unsigned long, в которую функция GetQueuedCompletionStatus поместит ключ завершения, т. е. номер файла, для которого завершилась асинхронная операции ввода-вывода

Следует отметить, что уведомление о завершении операции получит не обязательно тот же поток, который инициировал чтение или запись. Уведомление о завершении операции может быть получено любым ожидающим потоком. Поэтому необходимо, чтобы ключ мог идентифицировать дескриптор, связанный с завершившейся операцией.

Порты нужны чтобы уменьшить затраты ресурсов



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

Docker - платформа для разработки, доставки и запуска приложений в контейнерах. . Основное назначение – упростить развертывание приложения.

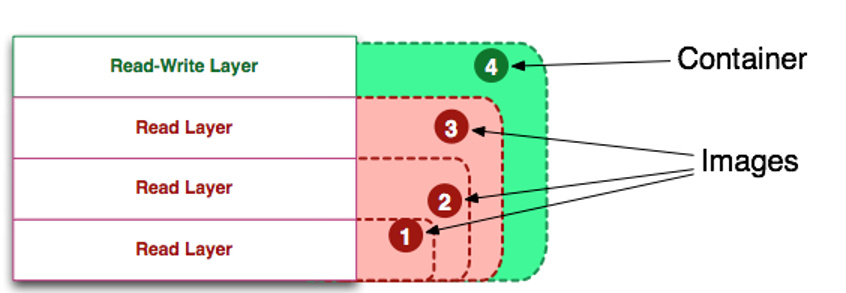
Позволяет «упаковать» приложение со всем его [окружением](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BE%D0%BA%D1%80%D1%83%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1)[[en]](https://en.wikipedia.org/wiki/Operating_environment) и зависимостями в контейнер,

Она включает в себя несколько ключевых компонентов. Давайте рассмотрим, как связаны Docker Engine и Docker Hub.

**Docker Hub** – облачный сервис для распространения **контейнеров**

**Docker Engine** – механизм (сервис и приложение) для создания и функционирования **контейнеров**.

Docker Engine - это ядро или основной компонент Docker. Это программное обеспечение, которое управляет контейнерами, и оно работает на хост-системе. Docker Engine обеспечивает возможность создания, запуска и управления контейнерами, а также взаимодействия с контейнеризированными приложениями.

image– read only template – шаблон для чтения.  


Docker Engine — это клиент-серверное приложение, которое позволяет управлять контейнерами и изображениями. Docker Engine включает в себя клиентский интерфейс, с помощью которого пользователи могут взаимодействовать с Docker, и серверную часть, которая управляет контейнерами и изображениями2.

Docker Engine состоит из следующих основных компонентов1:

* Сервер (Docker Daemon): Это долго работающее приложение, которое отвечает за создание, запуск и распределение Docker-контейнеров.
* Клиент (Docker Client): Это инструмент командной строки, который позволяет пользователям взаимодействовать с Docker Daemon.
* REST API: Docker Client использует REST API для передачи команд Docker Daemon.

Docker Engine необходим для функционирования и обеспечения связи между основными Docker-объектами (реестром, образами и контейнерами)3. Он позволяет создавать и управлять Docker-контейнерами, что делает его неотъемлемой частью экосистемы Docker1.

[**Реестр Docker** — это серверное приложение, которое хранит и позволяет распространять образы Docker](https://bing.com/search?q=%d1%87%d1%82%d0%be+%d1%82%d0%b0%d0%ba%d0%be%d0%b5+%d1%80%d0%b5%d0%b5%d1%81%d1%82%d1%80+%d0%b2+Docker). [Реестр Docker может быть частным и храниться в локальной среде или в общедоступном облаке3](https://learn.microsoft.com/ru-ru/dotnet/architecture/microservices/container-docker-introduction/docker-containers-images-registries)

Docker Hub, с другой стороны, — это общедоступный реестр. Docker Hub предоставляет репозиторий для хранения образов Docker и обмена ими

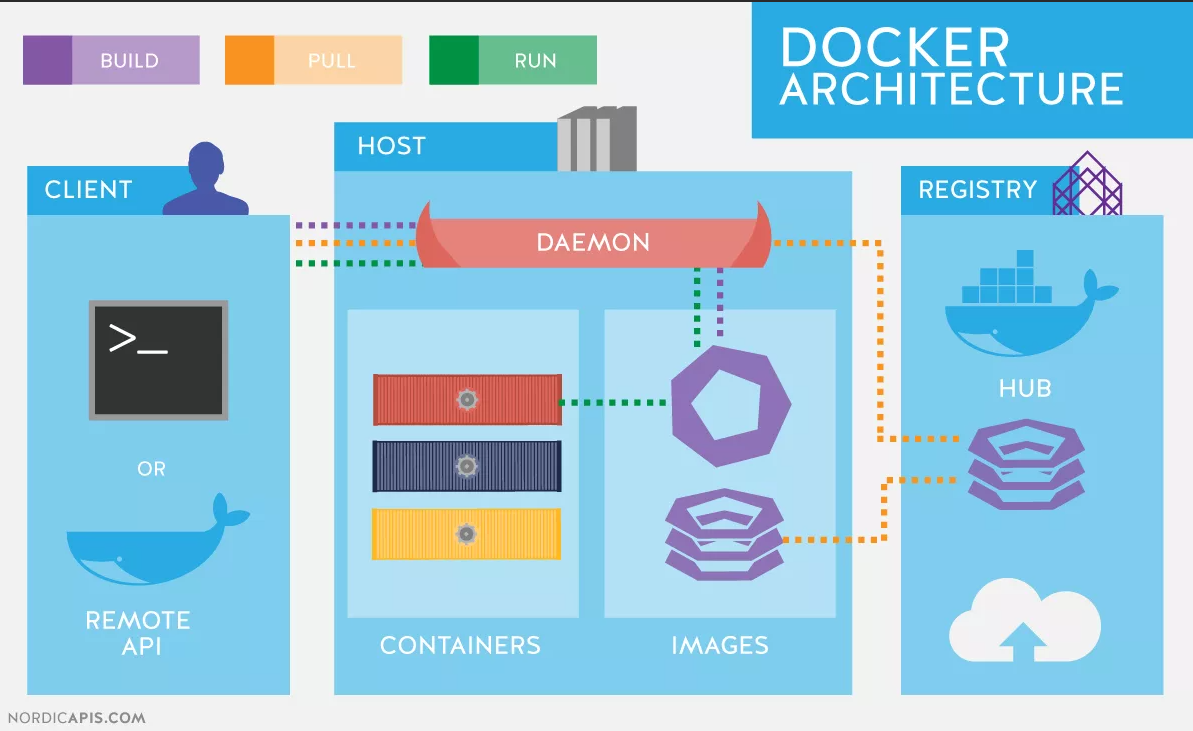
**Платформа Docker** является открытым программным обеспечением, которое предоставляет возможность упаковывать и запускать приложения в контейнерах. Docker использует стандартные контейнерные технологии, которые позволяют изолировать приложения и их зависимости от окружения.

**Архитектура Docker(Docker Engine)** состоит из нескольких компонентов:

* Docker daemon - отвечает за управление контейнерами и их запуск.
* Docker client - предоставляет интерфейс командной строки для взаимодействия с Docker daemon
* Docker registry - это централизованное хранилище образов Docker.
* Docker API - набор инструментов для взаимодействия с Docker daemon через программный интерфейс.
* Docker Images - это шаблоны для создания контейнеров. Они содержат все необходимое для запуска приложения, включая код, зависимости, переменные среды и настройки.
* Docker Containers - это экземпляры Docker Images. Контейнеры запускаются из образов и представляют собой изолированные исполняемые окружения, включая файловую систему, процессы и сетевые ресурсы. Docker Engine управляет жизненным циклом контейнеров.

Dockerfile - это текстовый файл, который содержит инструкции по созданию Docker-образа.

Инструкции **ENTRYPOINT** и **CMD** определяют команду, которая будет выполнена при запуске контейнера. **ENTRYPOINT** устанавливает команду, а **CMD** предоставляет аргументы, которые могут быть переопределены при запуске контейнера.



**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.

[Каждый Docker-образ состоит из серии слоев, которые представляют собой различные состояния файловой системы во время создания образа](https://bing.com/search?q=UFS+%d0%b2+Docker)[1](https://bing.com/search?q=UFS+%D0%B2+Docker). [Когда Docker запускает контейнер, он объединяет все эти слои в единую, согласованную файловую систему](https://bing.com/search?q=UFS+%d0%b2+Docker)[1](https://bing.com/search?q=UFS+%D0%B2+Docker).

**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// интерактивный терминал (**-it**)
* 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 (поместить в Docker Hub)
* commit - зафиксировать container в image
* docker build + путь к докер файл - построить image
* docker image rm – удалить image
* docker info – инфа
* docker-compose build
* docker-compose up – запуск