[**1.Операционная система (ОС): архитектура компьютера с точки зрения программиста, определение, классификация, операционные системы реального времени, программные интерфейсы, BIOS, драйверы, утилиты, ядро, основные объекты ядра, POSIX, фреймворк, программные и аппаратные прерывания, системные вызовы.** 3](#_Toc146794777)

[**2.Процесс ОС: определение, ресурсы процесса, контекст процесса, идентификатор процесса, IPC, стандартные потоки ввода/вывода, системные процесса, создание процесса в Windows, HANDEL Windows-процесса, особенности создания процессов и работы с ними в Linux, дерево процессов, Idle-процесс, назначение процессоров процессам.** 9](#_Toc146794778)

[**3.Поток ОС: определение, отличие от процесса, контекст потока, идентификатор потока, основной поток процесса, потоковая функция, понятие многопоточности, понятие потокобезовасность, диспетчеризация потоков, модель 12 состояний потока, создание потока в Windows, управление потоками в Windows, HANDEL Windows-потока, особенности создания потоков и работы с ними в Linux, приостановка работы потока, реентерабельность потока, дерево потоков, назначение процессоров понятие fiber.** 12](#_Toc146794779)

[**4. Диспетчеризация потоков: циклическое планирование, приоритетное планирование, кооперативное планирование, понятие приоритета потока и процесса, базовый приоритет потока в Windows, особенности диспетчеризации потоков в системах реального времени, назначение приоритета процессу и потоку в Windows, динамическое (автоматическое) изменение приоритета потоков, распределение процессоров потокам, особенности диспетчеризации потоков в Linux.** 17](#_Toc146794780)

[**5.Управление памятью: упрощенные схемы чтения и записи данных в оперативную память, адресное пространство, виртуальная память, страничный свопинг, страничная память, таблица страниц, Memory Management Unit, Translation Lookaside Buffer, трансляция виртуальных адресов в реальные, смена контекста процесса, инвертированные таблицы, понятие рабочего набора, алгоритм замещения страниц LRU, структура адресного пространства Windows/32, распределение памяти для стека приложения в Windows, распределение heap-памяти в Windows, выделение виртуальной и освобождение памяти в Windows, управление рабочим множеством страниц в Windows, запрет свопинга в Windows, распределение и управление Heap-памятью, особенности управления памятью в Linux.** 21](#_Toc146794781)

[**6.Синхронизация потоков: определение, взаимная блокировка потоков, принцип реализации процесса синхронизации, механизмы синхронизации потоков в Windows, атомарные операции в Windows, особенности синхронизации потоков в Linux.** 30](#_Toc146794782)

[**7. Компьютерное время: принцип вычисления компьютерного времени, социальное и компьютерное время, единица измерения компьютерного времени, Universal Coordinated Time, POSIX-время, ожидающий таймер, состояния ожидающего таймера, периодические и непериодические таймеры, работа с ожидающими таймерами в Windows, особенности работы с ожидающим таймером в Linux.** 33](#_Toc146794783)

[**8.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.** 37](#_Toc146794784)

[**9. Механизм отображение файлов в памяти: последовательность системных вызовов Windows для создания образа файла в оперативной памяти, использование образа файла, как средства межпроцессного взаимодействия, особенности отображения файлов в linux.** 49](#_Toc146794785)

[**10.Динамически вызываемые библиотеки: структура DLL-библиотеки, экспорт функций,**   **загрузка динамической библиотеки, динамический вызов функций динамической библиотеки, создание и применение библиотеки импорта.** 51](#_Toc146794786)

[**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.** 54](#_Toc146794787)

[**12. Управление пользователями и группами пользователей в Windows: понятие дискреционной системы безопасности, типы Windows-пользователей, группы пользователей, возможности API управления пользователями и группами.** 56](#_Toc146794788)

[**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.** 62](#_Toc146794789)

[**14. Windows-консоль: определение, применение стандартных потоков для ввода/вывода в консоль, возможности API для управления консолью.** 64](#_Toc146794790)

[**15.Асинхронные операции ввода вывода: понятие асинхронной операции ввода/вывода, особенности программирования асинхронного ввода/вывода.** 67](#_Toc146794791)

[16. Платформа Docker: архитектура, назначение, принципы устройства, файловая система UFS, контейнеры, образы, основные команды. 69](#_Toc146794792)

[--\*\*. Windows-сервисы: понятие и назначения сервиса, структура сервиса, порядок разработки и принцип работы сервиса, команды управления сервисом. 74](#_Toc146794800)

# **1.Операционная система (ОС): архитектура компьютера с точки зрения программиста, определение, классификация, операционные системы реального времени, программные интерфейсы, BIOS, драйверы, утилиты, ядро, основные объекты ядра, POSIX, фреймворк, программные и аппаратные прерывания, системные вызовы.**

**Архитектура компьютера с точки зрения программиста**

С точки зрения программиста архитектура ЭВМ - это набор программно-доступных средств.

В современных ПК реализован магистрально-модульный принцип построения. Все устройства (модули) подключены к центральной магистрали, системной шине (единая магистраль), которая включает в себя адресную шину, шину данных и шину управления.



**Центральный процессор** - электронный блок либо интегральная схема, исполняющая машинные инструкции.

**Оперативная память** — часть системы компьютерной памяти, в которой во время работы компьютера хранится исполняемый машинный код (программы), а также входные, выходные и промежуточные данные, обрабатываемые процессором.

**Периферийные устройства:**

1. Ввода(мышь, клавиатура)
2. Вывода(Монитор, принтер)
3. Хранения(ввода/вывода): жесткий диск;

**Контроллер** – устройство, которое связывает периферийное оборудование или каналы связи с центральным процессором, освобождая процессор от непосредственного управления функционированием данного оборудования.(1) Внешние устройства работают значительно медленнее процессора, поэтому для организации параллельной работы процессора и внешних устройств в архитектуру компьютера включается устройство управления внешними устройствами, называемое контроллером.(2)

**Шина** обеспечивает канал связи для передачи данных и сигналов управления между основными компонентами компьютерной системы.

Состоит из:

1. **Шина управления** несет сигналы управления, синхронизации и координации для управления различными функциями в системе.
2. **Адресная шина** используется для указания ячеек памяти для передаваемых данных.
3. **Шина данных**, представляющая собой двунаправленный путь, передает фактические данные между процессором, памятью и периферийными устройствами.

**ОС**

**ОС** - комплекс взаимосвязанных программ, предназначенных для управления ресурсами компьютера и организации взаимодействия с пользователем.

**OS** - комплекс программ (утилиты, системные программы, ядро.), пользовательский интерфейс, интерфейс с аппаратурой (драйверы), программный интерфейс

Классификация

* Для мейнфреймов: (Много одновременных заданий, пакетная обработка данных), IBM OS/390, IBM Z/OS
* Серверные: Windows Server, Linux Server
* Сетевые: Novel NetWare
* Персональные: Windows, MacOS, Ubuntu
* Мобильные: iOS, Android, HarmonyOS
* Встроенные: Embedded Linux, QNX
* Реального времени: мягкое и жесткое реальное время, FreeRTOS
* Смарт-карт: MULTOS

**ОС реального времени**

Под ***реальным временем*** понимается количественная характеристика, которая может быть измерена реальными физическими часами.

***Жесткого реального времени* -** для которых нарушения равнозначны отказу системы.

***Системы мягкого реального времени*** - нарушения характеристик которых приводят лишь к снижению качества работы системы.

**Операционная система реального времени —** операционная система, реагирующая на внешние события в определенный промежуток времени.

Это операционная система, в которой успешность работы любой программы зависит не только от ее логической правильности, но и от времени, за которое она получила результат.

**Программные интерфейсы**

**Программные интерфейсы -** интерфейс между приложениями и ядром OS:

* OS API;
* OS Framework, .NET Framework, .NET CORE Framework;
* POSIX, IEEE Std 1003; IBM AIX, HP-UX, UnixWare, QNX Neutrino, iOS, FreeBSD;
* Windows API.

**BIOS**

**BIOS** - набор микропрограмм (в энергонезависимой), реализующих API для работы с аппаратурой компьютера и подключенными к нему устройствами.

После включения компьютера процессор читает код BIOS из ПЗУ, записывает его в ОЗУ (оперативную память) и передает управление коду BIOS и далее BIOS загружает загрузчик OS.

Предоставляет пользовательский интерфейс для настройки оборудования.(мб установка частоты процессора/оперативной памяти).

Выполняет следующие функции:

* тестирование оборудования (при включении, POST -Power-On Self-Test);
* поиск и подключение устройств типа plug and play;
* первоначальная загрузка операционной системы;
* API для работы с оборудованием (на этапе загрузки ОС, потом используются драйверы);
* активация в offline (SLIC — software licensing description table)
* пользовательский интерфейс для настройки оборудования (устройство для загрузки, частота, лимиты, перечень устройств и т.п.).

**Драйвер**

**Драйвер** - это программное обеспечение, которое стандартизирует доступ к физическому устройству, подключенному к компьютеру.

Драйвера пишут разработчики железа для того, чтобы подогнать его под стандарты операционной системы.

* HAL – Hardware Abstraction Layer – слой аппаратных абстракций – по сути драйвер материнской платы; реализуется в программном обеспечении, расположенном между аппаратурой и ядром OS (аппаратно-зависимая компонента); по разному реализован а разных OS;
* драйверы разрабатывают поставщики оборудования;
* драйверы для большинства устройств (материнских плат) – в составе дистрибутива OS;
* для периферии (принтеры, сетевых карты, мониторы) драйверы, как правило, поставляются вместе с оборудованием;
* в Linux драйвер является частью ядра, в Windows – это dll-файлы (расширение sys);
* в операционной системе есть процедура установки драйвера (в разных OS по-разному);
* DDK – Driver Development Kit – инструментарий для разработки драйверов (WDK – Windows Driver Kit);
* виртуальные драйверы;
* NDIS – Network Driver Interface Specification (Microsoft, 3Com), в основном применяется Microsoft.

**Утилиты**

**Утилита** — вспомогательная компьютерная программа в составе общего программного обеспечения для выполнения специализированных типовых задач.

**Независимые утилиты** - не требующие для своей работы операционной системы.

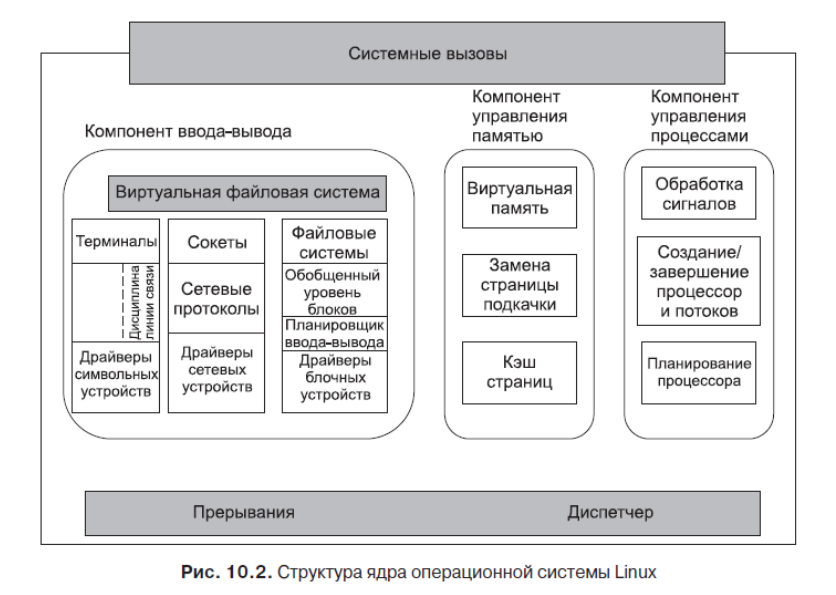
**Системные утилиты** - входящие в поставку ОС и требующие её наличия.

**Ядро ОС**

**Ядро -** центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации.

* резидентная часть программного обеспечения OS;
* всем ресурсам соответствует объекты OS;
* обработка аппаратных и программных прерываний;
* координированный доступ к ресурсам OS: процессорное время, оперативная память, устройства ввода вывода;
* единица работы – процесс (в простейшем случае приложение);
* создание и уничтожение процессов;
* межпроцессное взаимодействие;
* управление памятью;
* синхронизация процессов;
* диспетчеризация доступа к ресурсам OS;
* поддержка абстракции данных – файловая система;
* разная структура в разных OS: монолитная, модульная, микроядро, экзоядро, микроядро, …;
* в Windows ядро поддерживает графику, в Unix/Linux графика отдельный процесс;
* безопасность: аутентификация, авторизация.

**Объекты ядра**



**POSIX**

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

Основные задачи:

* Содействовать облегчению переноса кода прикладных программ на иные платформы
* Способствовать определению и унификации интерфейсов заранее при проектировании, а не в процессе их реализации.
* Сохранять по возможности и учитывать все главные, созданные ранее и используемые прикладные программы.
* Определять необходимый минимум интерфейсов прикладных программ для ускорения создания, одобрения и утверждения документов.
* Развивать стандарты в направлении обеспечения коммуникационных сетей, распределенной обработки данных и защиты информации.
* Рекомендовать ограничение использования бинарного (объектного) кода для приложений в простых системах.

**Фреймворк**

**Фреймворк ОС** - набор библиотек, которые позволяют нам использовать API OS.

**Фреймворк(приложения) —** *программная платформа*, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

**Программные и аппаратные прерывания**

**Прерывания** - это спец механизм, который обеспечивает остановку работы программ, которые выполнялись в текущий момент, и обеспечивает выполнение специальных программ, которые называются обработчики прерываний.

**Программные прерывания** - инициируются исполнением специальной инструкции в коде программы. Программные прерывания, как правило, используются для обращения к функциям встроенного программного обеспечения (firmware), драйверов и операционной системы.программные (частный случай внутреннего прерывания) — инициируются исполнением специальной [инструкции](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8) в коде [программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0).**(вывод символа на экран)**

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

**Приотиризация:** До окончания обработки прерывания обычно устанавливается запрет на обработку этого типа прерывания, чтобы процессор не входил в цикл обработки одного прерывания. Приоритизация означает, что все источники прерываний делятся на классы и каждому классу назначается свой уровень приоритета запроса на прерывание.

**Системный вызов**

**Системный вызов** - это специальный механизм с помощью которого мы можем вызывать функции ядра операционной системы.

**Системный вызов** - механизм вызова прикладной программой функции ядра OS.

Системный вызов выполняется через программные прерывания, указывается номер системного вызова и системный вызов обрабатывается. Надо сказать, что фактически все функции API в себе содержат системные вызовы.

Порядок выполнения системного вызова:

* Помещение параметров в стек вызова
* Вызов процедуры
* Помещение кода в регистр
* Переключение из пользовательского режима в режим ядра
* Проверка номера системного вызова
* Обработчик системного вызова обрабатывает
* Возврат управление процедуре из библиотеки
* Возврат управления пользовательской программе
* Очистка стека пользовательской программой

# **2.Процесс ОС: определение, ресурсы процесса, контекст процесса, идентификатор процесса, IPC, стандартные потоки ввода/вывода, системные процесса, создание процесса в Windows, HANDEL Windows-процесса, особенности создания процессов и работы с ними в Linux, дерево процессов, Idle-процесс, назначение процессоров процессам.**

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

**Проце́сс —** это идентифицируемая абстракция совокупности взаимосвязанных системных ресурсов на основе отдельного и независимого виртуального адресного пространства в контексте которой организуется выполнение потоков.

**Ресурсы процессора**

* регистры
* открытые 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.

**Контекст процесса**

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

Включает:

* содержимое адресного пространства
* содержимое регистров - общего назначения, счетчик команд, состояния процессора, вершина стека)
* стек ядра (для этого процесса)
* объекты ядра ОС

**Идентификатор процесса**

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

Как он выглядит в различный операционных система

32-разрядное число

**IPC**

**Межпроцессорное взаимодействие —** обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности IPC.

В Windows поддерживаются следующие механизмы IPC:

* Буфер обмена
* COM
* Копирование данных
* Сопоставление файлов
* Mailslots
* Каналы
* RPC
* Сокеты

Стандартные потоки ввода/вывода

потоки имеющие зарезервированные номера - дескрипторы (номера),  поток ввода (0), поток вывода (1), поток вывода ошибок (2).  Как правило (хотя и не обязательно), эти дескрипторы открыты уже в момент запуска задачи (исполняемого файла).

Системные процессы

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

Системные процессы являются частью ядра и всегда расположены в оперативной памяти.

Создание процесса в Windows

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

**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().

PID Идентификатор процесса - 32-разрядное число, которое фактические отображает номер объекта ОС.

**Fork()**

**fork()** — системный вызов в Unix-подобных операционных системах, создающий новый процесс (потомок), который является практически полной копией процесса-родителя, выполняющего этот вызов.

**Между процессом-потомком, порождаемым вызовом fork(), и процессом-родителем существуют различия:**

* PID процесса-потомка отличен от PID процесса-родителя;
* значению PPID процесса-потомка присваивается значение PID процесса-родителя;
* процесс-потомок получает собственную таблицу файловых дескрипторов, являющуюся копией таблицы процесса-родителя на момент вызова fork(); это означает, что открытые файлы наследуются, но если процесс-потомок, например, закроет какой-либо файл, то это не повлияет на таблицу дескрипторов процесса-родителя;
* для процесса-потомка очищаются все ожидающие доставки сигналы;
* временная статистика выполнения процесса-потомка в таблицах ОС обнуляется;
* блокировки памяти и записи, установленные в процессе-родителе, не наследуются.

После fork() процесс-потомок чаще всего выполняет системный вызов exec(), загружающий в пространство процесса новую программу (именно так, и только так, в Unix-системе выполняется запуск программы в отдельном процессе). Так, первый (нулевой) процесс Unix (ядро системы) создаёт свою копию, чтобы запустить init (процесс с PID = 1), который в свою очередь создаёт дочерние процессы для запуска инициализации системы и терминалов.

**Дерево процессов**

Процессы, попарно связанные дочерне-родительскими отношениями, формируют дерево процессов операционной системы.

Процессы знают только PID дочерних процессов, а процессов родителей – не знают.

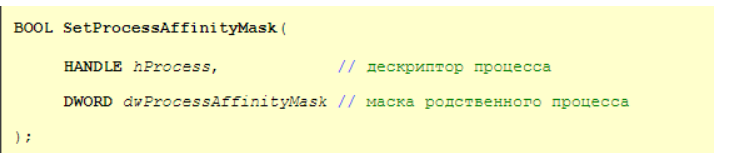
**Idle-процесс**

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

Количество потоков процесса «бездействие» равно количеству процессоров или ядер, имеющихся в компьютере. Процесс «бездействие» выполняется процессором/ядром, например, если в системе все потоки выполняются на другом процессоре/ядре.

**Назначение процессоров процессам**

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



# **3.Поток ОС: определение, отличие от процесса, контекст потока, идентификатор потока, основной поток процесса, потоковая функция, понятие многопоточности, понятие потокобезовасность, диспетчеризация потоков, модель 12 состояний потока, создание потока в Windows, управление потоками в Windows, HANDEL Windows-потока, особенности создания потоков и работы с ними в Linux, приостановка работы потока, реентерабельность потока, дерево потоков, назначение процессоров понятие fiber.**

**Определение**

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

**Поток (управления) OS** – последовательность инструкций, выполняемых процессором в выделенные OS интервалы времени.

**Отличия**

Поток часть процесса.

При создании процесса, автоматически создается основной (main) поток (выполняется функция ядра, создающая поток).

Как правило, каждый поток может работать (читать и писать) с одной и той же областью памяти, в отличие от процессов, которые не могут просто так получить доступ к памяти другого процесса.

Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие.

Поток — определенный способ выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса.

**Контекст потока**

* **программный код;**
* **набор регистров;**
* **стек памяти;**
* **стек ядра операционной системы;**
* **маркер доступа.**

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

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

**Идентификатор потока**

**Идентификатор -** уникально идентифицирует поток повсюду в системе. Идентификаторы допустимы со времени создания потока, и до тех пор, пока поток не завершит работу.

**Основной поток процесса**

При создании процесса, автоматически создается основной (main) поток (выполняется функция ядра, создающая поток).

**Потоковая функция**

Функция передаваемая в качестве аргумента при создании потока, код которой исполняется потоком.

**Понятие многопоточности**

**Многопоточность -** свойство платформы или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени.

**Потоковая безопасность**

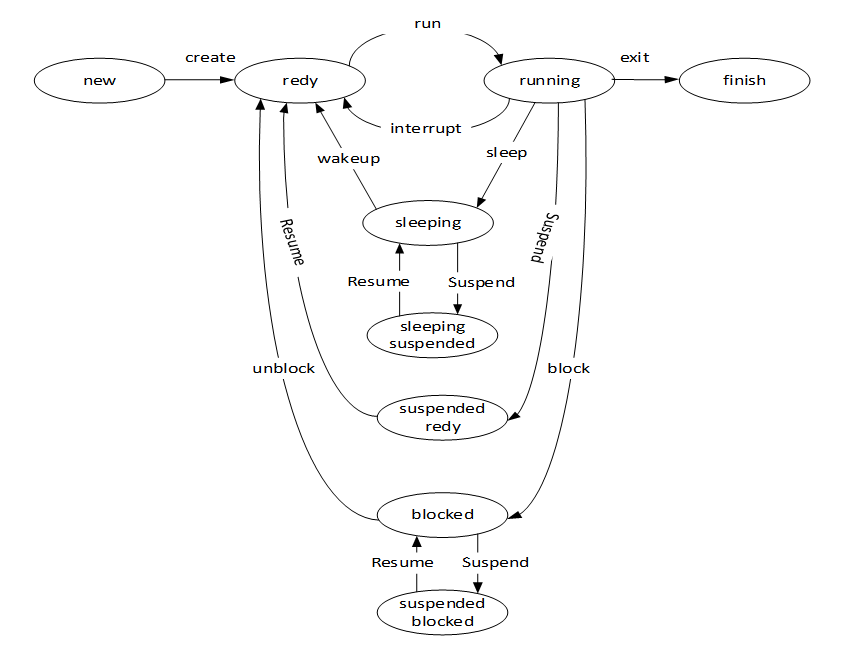
Это концепция программирования, применимая к многопоточным программам. Код потокобезопасен, если он функционирует исправно при использовании его из нескольких потоков одновременно. В частности, он должен обеспечивать правильный доступ нескольких потоков к разделяемым данным.

**Диспетчеризация**

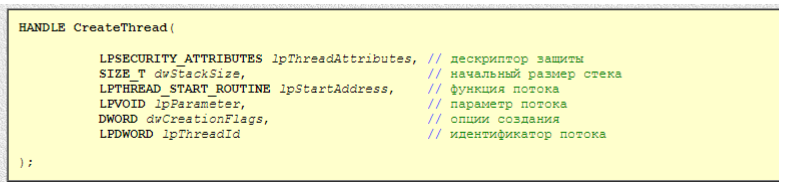
**Диспетчеризация -** заключается в реализации найденного в результате планирования (динамического или статистического) решения, то есть в переключении про­цессора с одного потока на другой. Прежде чем прервать выполнение потока, ОС запоминает его контекст, с тем, чтобы впоследствии использовать эту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

**Модель 12 состояния потока**

* **interrupt –** прерывание по окончанию кванта;
* **block –** блокировать до наступления события;
* **unblock –** ожидаемое событие наступило;
* **Suspend –** приостановить поток, Resume – возобновить поток;
* **sleep –** остановить поток на заданное время;
* **wakeup –** возобновить работу.



**Создание потока Windows**



**Управление потоками в Windows**

**Создание: CreateThread**

Любой поток процесса может сам завершить свое выполнение, вызвав функцию **ExitThread**, однако более обычным способом самостоятельного завершения потока является возврат из функции потока с использованием кода завершения в качестве возвращаемого значения.

Выполнение потока также может быть завершено другим потоком с помощью функции **TerminateThread**, однако освобождения ресурсов потока при этом не происходит. Поток, выполнение которого было завершено (напомним, что обычно поток должен самостоятельно завершать свое выполнение), продолжает существовать до тех пор, пока посредством функции CloseHandle не будет закрыт ее последний дескриптор.

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

Поток создается с приоритетом потока **THREAD\_PRIORITY\_NORMAL**. Используйте функции **GetThreadPriority** и **SetThreadPriority**, чтобы получать и установить приоритетное значение потока.

**Handle Windows-потока**

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

**HANDLE —** условно, адрес, по которому хранится информация по потоку. Будучи однажды получен, HANDLE требует закрытия через CloseHandle().

**Особенности создания потоков и работы с ними в Linux**

Для ядра Linux не существует отдельной концепции потоков. В ядре Linux потоки реализованы так же, как и обычные процессы. В ОС Linux нет никакой особенной семантики для планирования выполнения потоков или каких-либо особенных структур данных для представления потоков. **Поток** — это просто процесс, который использует некоторые ресурсы совместно с другими процессами. Каждый поток имеет структуру task\_struct и представляется для ядра  обычным процессом (который совместно использует ресурсы, такие как адресное пространство, с другими процессами).

Для операционной системы Linux потоки — это просто способ совместного использования ресурсов несколькими процессами (которые и так имеют достаточно малое время переключения контекста)

Допустим, у нас есть процесс, состоящий из четырех потоков. В операционных системах с явной поддержкой потоков должен существовать дескриптор процесса, который далее указывает на четыре потока. Дескриптор процесса описывает совместно используемые ресурсы, такие как адресное пространство и открытые файлы. Потоки описываются ресурсами, которые принадлежат только им. В ОС Linux, наоборот, существует просто четыре процесса и, соответственно, четыре обычные структуры task\_struct. Четыре процесса построены так, чтобы совместно использовать определенные ресурсы.

Потоки создаются с использованием функции **pthread\_create()**

Для возможности использования потоков в коде C/C++ в Linux компоновщику в процессе сборки нужно передать параметр **–lpthread**

**Приостановка работы потока**

**Windows:** SuspendThread() – приостановить, ResumeThread – восстановить работу

**Linux:** Ну тут хуита, нету нормальных механизмов

Можно сделать через пайп, что поток А повисает на чтении пайпа, а что бы возобновить его из потока Б отправляется сообщение

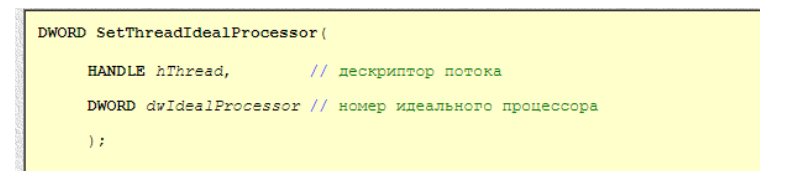
**Реентерабельность потока**

**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.

**Дерево потоков**

Потоки могут внутри себя создавать ещё потоки. Таким образом совокупность потоков имеет древовидную структуру.

**Назначение процессоров**



Номер привилегированного процессора для потока. Значение **MAXIMUM\_PROCESSORS** сообщает системе, что поток не имеет привилегированного процессора.

**Понятие fiber**

**Fiber –** механизм ручного планирования выполнения кода в рамках потока. Fiber использует кооперативные алгоритмы планирования – то есть оперативная система никогда не инициирует переключение контекста выполнения потока. Управление передается тогда, когда это нужно внутри кода выполняющемся в Fiber.

# **4. Диспетчеризация потоков: циклическое планирование, приоритетное планирование, кооперативное планирование, понятие приоритета потока и процесса, базовый приоритет потока в Windows, особенности диспетчеризации потоков в системах реального времени, назначение приоритета процессу и потоку в Windows, динамическое (автоматическое) изменение приоритета потоков, распределение процессоров потокам, особенности диспетчеризации потоков в Linux.**

**Циклическое планирование**

Каждому потоку предоставляется квант времени процессора. Когда квант заканчивается, поток переводится планировщиком в конец очереди, а управление передается следующему за ним потоку.

**Недостатки:**

* слишком малый квант времени (по сравнению с временем переключения контекстов) приводит к частому переключению потоков и снижению производительности;
* слишком большой квант может привести к увеличению времени ответа на интерактивный запрос.

На самом деле там не одна очередь, а их много. И каждая для своего процессора.

**Приоритетное планирование**

Это такой алгоритм планирования при котором каждому потоку присваивается приоритет, и управление передается потоку с самым высоким приоритетом.

Обычно потоки объединяют по приоритетам в группы, и применяют приоритетное планирование среди групп, а внутри группы используют циклическое планирование.

**Кооперативное планирование**

Алгоритм основан на том, что активному потоку позволяется выполняться, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению поток.

Это такой алгоритм планирования при котором поток получает столько процессорного времени, сколько он считает нужным. Таким образом, все потоки делят процессорное время, периодически передавая управление следующей задаче.

**Гарантированное планирование**

ОС гарантирует существующим потокам, что они получат гарантированную справедливую часть процессорного времени. n потоков, 1/n частей процессорного времени каждому.

**Понятие приоритета потока и процесса**

**Приоритет процесса -** число, ориентируясь на значение которого планировщик процессов может выдавать процессу больше или меньше процессорного времени.

**Приоритет потока —** величина, складывающаяся из двух составных частей: приоритета породившего поток процесса и собственно приоритета потока. Когда поток создается, ему назначается приоритет, соответствующий приоритету породившего его процесса.

* **Real time; 24**
* **Normal; 8**
* **High; 13**
* **Below normal; 6**
* **Above normal; 10**
* **Idle. 4**

**Базовый приоритет потока**

Сочетание класса приоритета процесса и приоритета потока, изменяется в пределах [1,31], по умолчанию – 8, приоритеты возрастающие (СМЕЛОВ)

****

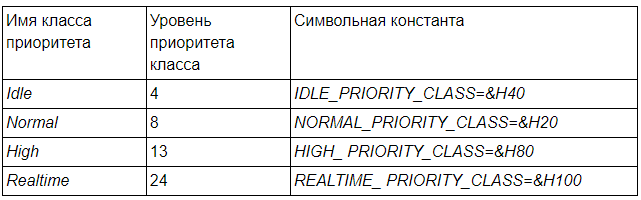
**Особенности диспетчеризации потоков в системах реального времени**

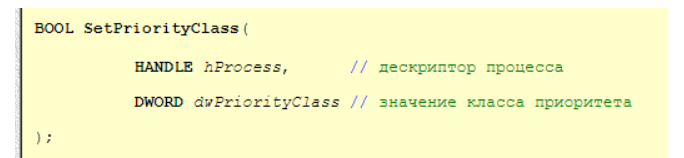
Любая система реального времени должна реагировать на сигналы управляемого объекта в течение заданных временных ограничений. Необходимость тщательного планирования работ облегчается тем, что в системах реального времени весь набор выполняемых задач известен заранее. Кроме того, часто в системе имеется информация о временах выполнения задач, моментах активизации, предельных допустимых сроках ожидания ответа и т. д.

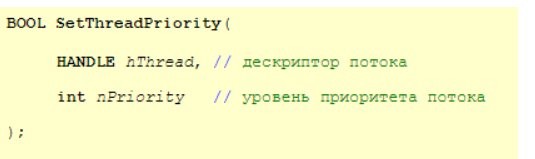
В системах реального времени так как потоки обычно создается для обработки какого-то события они ставятся в начало очереди. Потому что обработка каждого нового события является более приоритетной чем обработка всех предыдущих.

В задачах реального времени предъявляются специфические требования к методам диспетчеризации, поскольку передача управления потоку должна определяться критическим сроком его обслуживания (т.н. deadline-driven scheduling). В наибольшей степени этому требованию соответствует вытесняющая приоритетная многозадачность. Суть этого метода заключается в том, что как только поток с более высоким, чем у активного потока, приоритетом переходит в состояние готовности, активный поток вытесняется (т.с. из активного состояния принудительно переходит в состояние готовности) и управление передается более приоритетному потоку.

**Назначение приоритета процессу и потоку в Windows**

Происходит в два этапа. Во-первых, каждому процессу в момент создания присваивается класс приоритета. Узнать класс приоритета можно с помощью функции ***GetPriorityClass*,** а изменить - с помощью функции ***SetPriorityClass***. Ниже приведены имена классов приоритета процессов, уровни приоритета и константы, которые используются с этими вышеупомянутыми функциями (как и с функцией *CreateProcess*).  
  
При создании уровень приоритета потока по умолчанию устанавливается равным уровню класса приоритета процесса, создавшего данный поток. Тем не менее, можно использовать функцию ***SetThreadPriority***, чтобы изменить приоритет потока





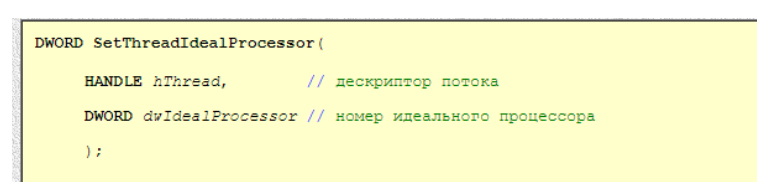
**Динамическое изменение приоритета потоков**

Иногда система изменяет уровень приоритета потока. Обычно это происходит в ответ на некоторые события, связанные с вводом выводом (например, на появление оконных сообщений или чтение с диска). Система повышает приоритет только тех потоков, базовый уровень которых находится в пределах 1-15 Именно поэтому данный диапазон называется «областью динамического приоритета» (dynamic priority range). Система не допускает динамического повышения приоритета потока до уровней реального времени (более 15) Поскольку потоки с такими уровнями обслуживают системные функции, это ограничение не дает приложению нарушить работу операционной системы.  
Значение динамического приоритета потока ограничено снизу его базовым приоритетом, верхней же границей является нижняя граница диапазона приоритетов реального времени.

**Распределение процессоров потокам**

При работе на многоядерных процессорах, операционная система производит ротацию потоков между ядрами. Другими словами, переносит поток с более загруженных на менее загруженные ядра, что обеспечивает равномерную загрузку всех ядер системы.

Аналог афинити маски:



Позволяет ограничивать диспетчеризацию процессора

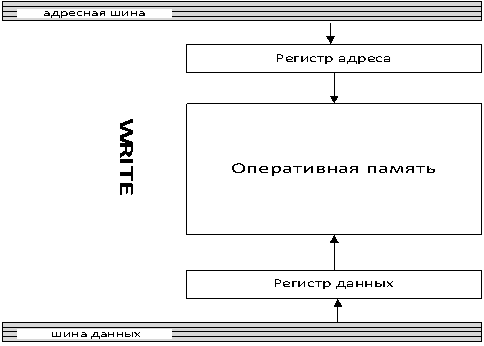
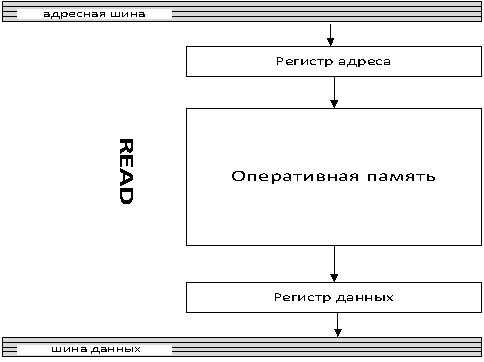
Номер привилегированного процессора для потока. Значение **MAXIMUM\_PROCESSORS** сообщает системе, что поток не имеет привилегированного процессора.

**Особенности диспетчеризации потоков в Linux**

* + **мультизадачная, несколько планировщиков потоков (разделение времени – OTHER,реального времени – FIFO/RR, бездействия - BATCH);**
  + **ядро работает с внутренними приоритетами, которые вычисляются из nice или статического приоритета;**
  + **потоки/процессы могут быть привязаны (маска) к процессору с помощью системного вызова;**
  + **ядро не различает процессы и потоки, общая системы нумерации, main-поток TID = PID;**
  + **OTHER, nice – любезность в пределах [-20, 19], приоритеты убывающие;**
  + **OTHER, nice может изменяться с помощью системных вызовов;**
  + **FIFO,статические приоритеты (возрастающие), нет квантования, уступка процессора, новый в начало очереди, подавляет OTHER, c RR по приоритетам;**
  + **RR, ,статические приоритеты (возрастающие), квантование, подавляет OTHER, c FIFO по приоритетам;**
  + **планировщики ввода/вывода, обычно для блоковых устройств (диски), назначаются при конфигурации, задача планировщика – повышение эффективности (приоритетность, гарантированное время, оптимизация поиска – лифтовые алгоритмы);**
  + **планировщики ввода/вывода, NOOP (простой, общая очередь), CFQ (очередь у каждого процесса, квантование), Deadline (гарантирует время получения данных), BFQ (усовершенствованный CFQ), MQ-Deadline – усовершенствованный Deadline, Kyber – для работы с быстрыми устройствами.**
  + **с помощью системных вызовов можно менять приоритет ввода/вывода (CFQ, BFQ).**
  + **можем приостанавливать, убивать, создавать потоки**
  + **потоки не входят в основную часть ядра, это отдельные библиотеки. (СМЕЛОВ) реентерабельность кода - когда один и тот же код может работать в нескольких потоках**

# **5.Управление памятью: упрощенные схемы чтения и записи данных в оперативную память, адресное пространство, виртуальная память, страничный свопинг, страничная память, таблица страниц, Memory Management Unit, Translation Lookaside Buffer, трансляция виртуальных адресов в реальные, смена контекста процесса, инвертированные таблицы, понятие рабочего набора, алгоритм замещения страниц LRU, структура адресного пространства Windows/32, распределение памяти для стека приложения в Windows, распределение heap-памяти в Windows, выделение виртуальной и освобождение памяти в Windows, управление рабочим множеством страниц в Windows, запрет свопинга в Windows, распределение и управление Heap-памятью, особенности управления памятью в Linux.**

**Упрощенные схемы чтения и записи**

****

**Адресное пространство**

**Адресное пространство -** непрерывный диапазон адресов выделяемый OS процессу; у каждого процесса свое адресное пространство.

**Виртуальная память**

**Виртуальная память –** метод управления памятью процессора, предназначенный для выполнения программ, которым выделяется адресное пространство превышающее доступный физический объем памяти компьютера.

**Страничный свопинг**

**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

**Страничная память**

**Страничная память –** реализации виртуальной памяти, при которой физическая память и адресное пространство разбивается на блоки (страницы), а также осуществляется страничный свопинг. Размеры страниц для X86-64: 4K, 2MB, 1GB.

В основе виртуальной памяти лежит идея, что у каждой программы имеется собственное адресное пространство, которое разбивается на участки, называемые страницами. Каждая страница представляет собой непрерывный диапазон адресов. Эти страницы отображаются на физическую память, но для запуска программы одновременное присутствие в памяти всех страниц необязательно(Книга)

**Таблица страниц**

**Таблица страниц —** это структура данных, используемая системой виртуальной памяти в операционной системе компьютера для хранения сопоставления между виртуальным адресом и физическим адресом. Виртуальные адреса используются выполняющимся процессом, в то время как физические адреса используются аппаратным обеспечением, или, более конкретно, подсистемой ОЗУ.

**Таблица страниц** является ключевым компонентом преобразования виртуальных адресов, который необходим для доступа к данным в памяти.

524288 × 3 byte = 1572864 = 1536K = 1.5MB – размер таблицы страниц для 2GB адресного пространства.

**Memmory Management Unit**

**MMU – Memory Management Unit –** диспетчер памяти – аппаратное (программируемое) устройство, входящее в состав процессора и предназначенное для трансляции виртуальных адресов оперативной памяти в реальные.

**Translation Lookaside Buffer**

**TLB(Translation Lookaside Buffer) –** буфер быстрого преобразования адреса; компонент MMU, предназначенный для вычисления реальных адресов, хранит 64 строки таблицы страниц, полностью таблица хранится во вторичной (диск) памяти без свопинга;

**Трансляция виртуальных адресов в реальные**

Блок управления памятью (MMU) ЦПУ хранит кэш недавно использованных отображений из таблицы страниц операционной системы. Это называется буфер ассоциативной трансляции (далее — TLB), который является ассоциативным кэшем.

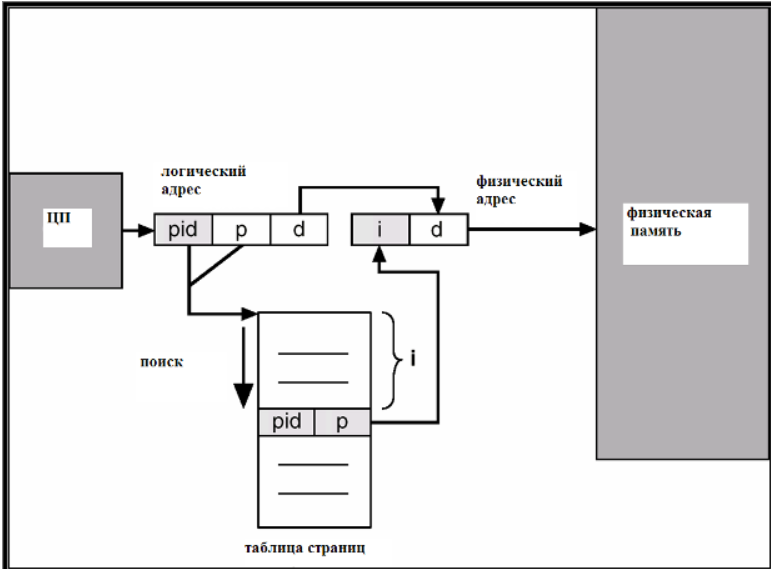
Когда виртуальный адрес необходимо преобразовать в физический адрес, то вначале просматривается TLB. Если совпадение найдено (называется «TLB hit») то возвращается физический адрес и доступ к памяти может продолжаться. Однако, если нет совпадения (называемого «TLB miss»), то либо модуль управления памятью, либо обработчик пропусков TLB операционной системы ищет сопоставление адресов в таблице страниц, чтобы узнать, существует ли сопоставление (называется «page walk»). Если сопоставление существует — то оно записывается обратно в TLB (это необходимо сделать, поскольку аппаратное обеспечение обращается к памяти через TLB в системе виртуальной памяти), и текущая команда перезапускается (что также может происходить параллельно).

**Смена контекста процесса**

Если целевая задача при переключении контекста использует отличный от предыдущего контекст памяти (каталог страниц и таблицы страницы), происходит очистка TLB, таким образом при последующих обращениях в память снижается производительность процессора (необходима подгрузка записей таблицы страниц в пустой TLB).

**Инвертированные таблицы**

Данный способ организации таблиц страниц предназначен для сокращения размеров таблиц страниц. В таблице страниц хранится один элемент для каждой реальной страницы, находящейся в памяти. К логическому адресу добавляется номер процесса (pid) владеющего данной страницей. По паре (номер процесса = pid, номер страницы = p) выполняется ассоциативный поиск в таблице страниц. Индекс найденного элемента таблицы i конкатенируется со смещением d внутри страницы, в результате получается физический адрес.



**Понятие рабочего набора**

**Рабочий набор -** это набор страниц памяти, которые в настоящее время принадлежат вашему процессу и не выгружены

**Алгоритм замещения страниц LRU**

**LRU (least recently used) —** это алгоритм, при котором вытесняются значения, которые дольше всего не запрашивались. Соответственно, необходимо хранить время последнего запроса к значению. И как только число закэшированных значений превосходит N необходимо вытеснить из кеша значение, которое дольше всего не запрашивалось.

**Структура адресного пространства Windows/32**

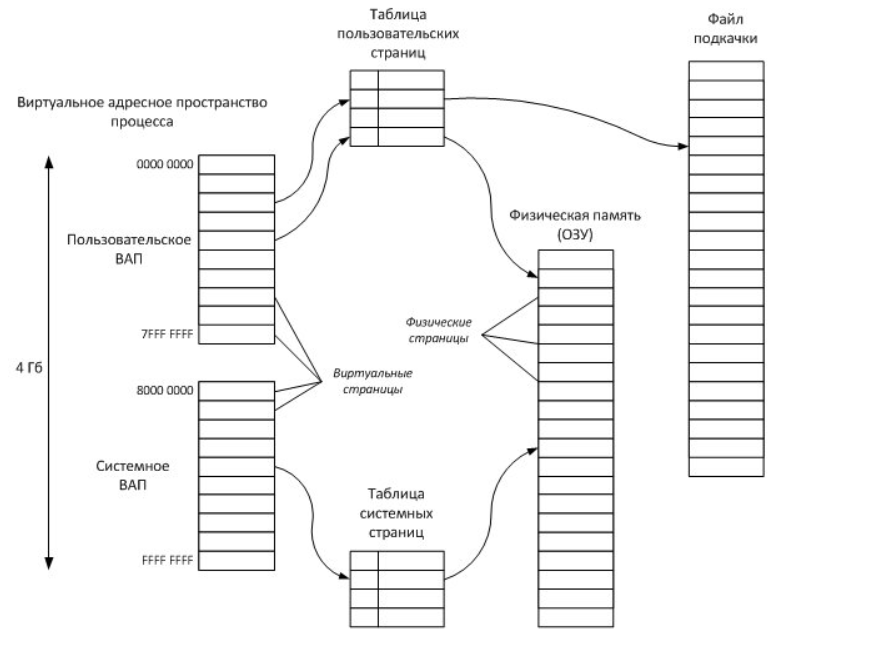


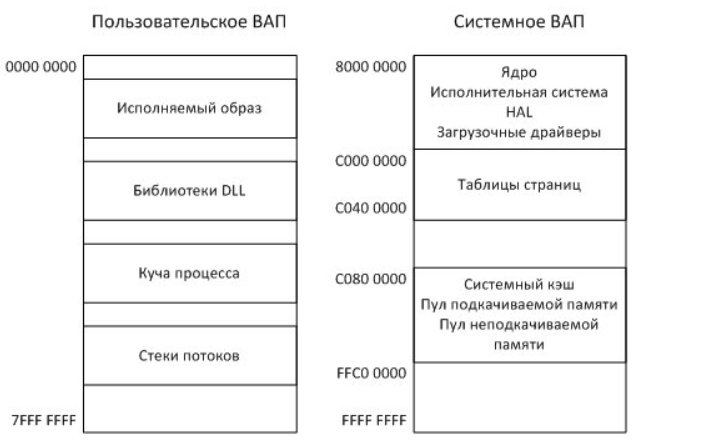
Схема реализации виртуальной памяти в 32-разрядной операционной системе Windows представлена на рис.11.1. Как уже отмечалось, процессу предоставляется виртуальное адресное пространство размером 4 ГБ, из которых 2 ГБ, расположенных по младшим адресам (0000 0000 – 7FFF FFFF), процесс может использовать по своему усмотрению (пользовательское ВАП), а оставшиеся два гигабайта (8000 0000 – FFFF FFFF) выделяются под системные структуры данных и компоненты (системное ВАП)

Виртуальная память делится на блоки одинакового размера – виртуальные страницы. В Windows страницы бывают большие (x86 – 4 МБ, x64 – 2 МБ) и малые (4 КБ). Физическая память (ОЗУ) также делится на страницы точно такого же размера, как и виртуальная память. Общее количество малых виртуальных страниц процесса в 32 разрядных системах равно 1 048 576 (4 ГБ / 4 КБ = 1 048 576).

Обычно процессы задействуют не весь объем виртуальной памяти, а только небольшую его часть. Соответственно, не имеет смысла (и, часто, возможности) выделять страницу в физической памяти для каждой виртуальной страницы всех процессов. Вместо этого в ОЗУ (говорят, "резидентно") находится ограниченное количество страниц, которые непосредственно необходимы процессу. Такое подмножество виртуальных страниц процесса, расположенных в физической памяти, называется рабочим набором процесса (working set).

Те виртуальные страницы, которые пока не требуются процессу, операционная система может выгрузить на диск, в специальный файл, называемый файлом подкачки (page file).

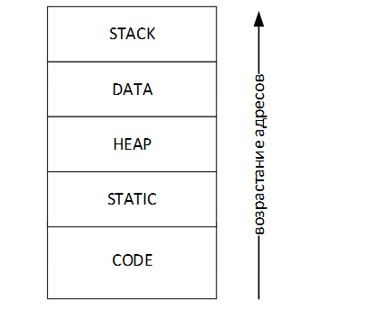
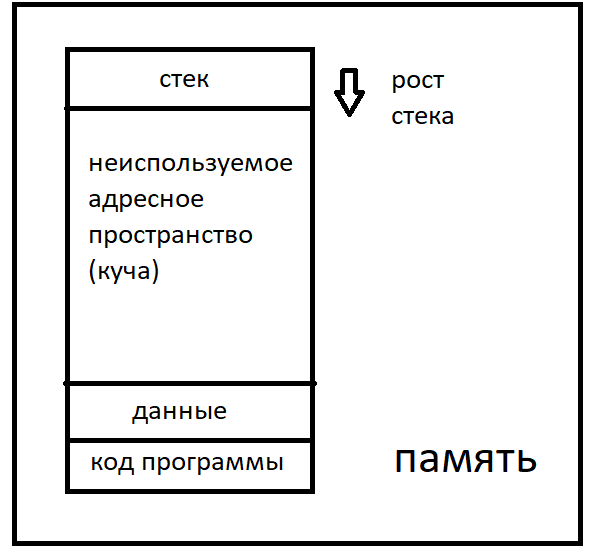
Каким образом процесс узнает, где в данный момент находится требуемая страница? Для этого служат специальные структуры данных – таблицы страниц (page table).



**Распределение памяти для стека в приложении Windows**

Начальный размер стека зависит от o/s. Начальный размер heap логически равен нулю, но имеет тенденцию расти почти сразу (по мере загрузки программы и общих библиотек).

Cтек и hepa будут назначены сегментам, которые могут быть расширены, так что физическая память для максимальных размеров не должна быть предварительно зарезервирована. Однако многие микропроцессоры не имеют аппаратного обеспечения для отображения памяти, и размеры должны быть предварительно зарезервированы (хотя иногда стек и heap растут навстречу друг другу, так что существует только один общий предел).

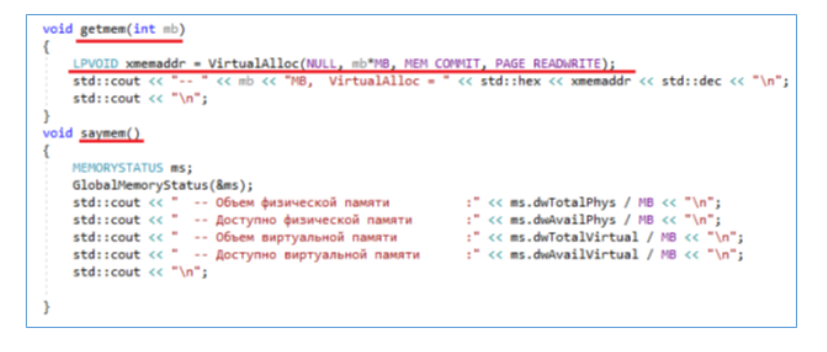
****

**Распределение heap-памяти в Windows**

**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.

**Выделение виртуальной и освобождение памяти в Windows**

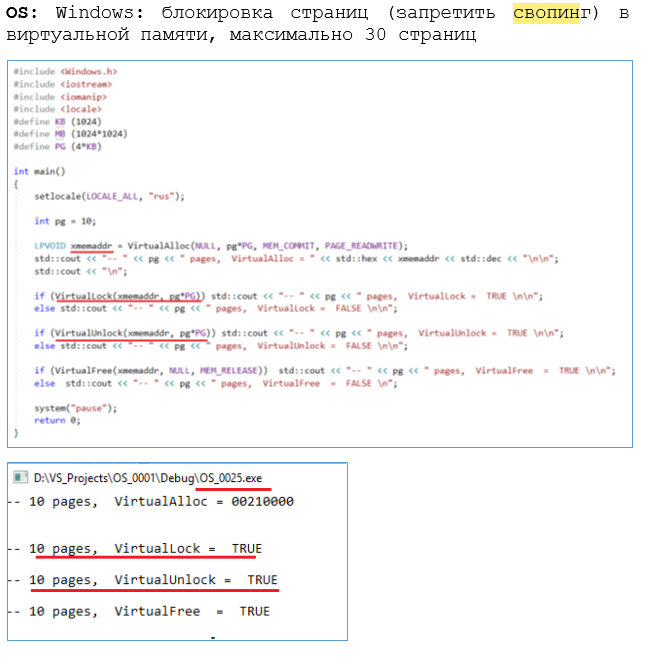
**VirtualAlloc**, выделить (MEM\_COMMIT) или резервировать (MEM\_RESERVE)память; только для чтения(PAGE\_READONLY), чтение/запись(PAGE\_READWRITE), исполняемый код (PAGE\_EXECUTE), граница 64К,кратно 4К (страница 4К), можно явно указать адрес.

**VirtualFree,** освободить (MEM\_RELEASE), отменить резервирование (MEM\_DECOMMIT).  


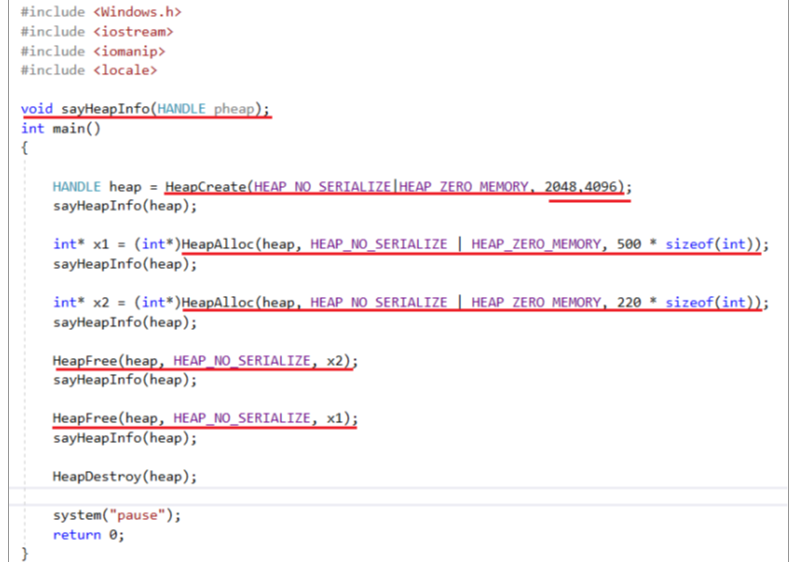
**Управление рабочим множеством страниц**



**Запрет свопинга в Windows**



**Распределение и управление Heap-памятью**

****

**Особенности управления памятью в Linux**

Когда происходит обращение к страницам памяти, которые некоторое или долгое время не использовались, т. е. к «неактивным» страницам, то ядро выполняет с ними несколько важных задач:

* - возвращает ссылки на эти страницы в соответствующей таблице страниц;
* - сбрасывает в нулевое значение время «неиспользования» этих страниц;
* - помечает эти страницы как «активные».

Со страницами, находящимися в виртуальной памяти не всё так однозначно. Дело в том, что для того, чтобы «активизировать» такие страницы, они должны быть предварительно прочитаны с диска.

Системное ядро комплектуется специализированными модулями. Которые содержат алгоритмы и даже целые технологии. С помощью которых система довольно эффективно «предсказывает», сколько может потребоваться памяти при разной степени активности и загруженности процессов. Эти алгоритмы имеют своей целью обеспечение процессов свободной памятью с максимальной эффективностью. Т. е. так, чтобы процессам как можно реже приходилось простаивать в «ожидании» выгрузки очередной страницы в свободную память. Таким образом, наблюдая за состоянием страничного обмена во время рабочей нагрузки системы, можно делать выводы о том, нужна ли ей дополнительная память. Если страничный обмен интенсивный — то однозначно следует установить дополнительные модули ОЗУ.

Если же происходит так, что процессам не хватает ни реальной физической, ни виртуальной памяти. Т. е. когда память полностью исчерпана, то система начинает завершать (а точнее уничтожать) целые процессы. Либо запрещает создание новых. Конечно в этом случае в первую очередь уничтожаются наиболее «безболезненные» для системы процессы. Однако в таких случаях даже «на глаз» и по собственным ощущениям видно что она большую часть времени тратит на управление памятью, а не на выполнение рабочих задач.

В Linux можно настроить параметр, который задаёт, насколько быстро ядро должно «отбирать»страницы памяти у процессов. Которым они менее нужны для процессов, которым они на данный момент необходимы. Этот параметр содержится в файле /proc/sys/vm/swappiness и по-умолчанию равен 60. Если задать его меньшим значением (например 0). То ядро будет забирать страницы процесса в самую последнюю очередь. Используя вместо этого любые другие варианты. Если это значение в пределах между 60 и 100. То страницы будут отбираться у процессов с более высокой вероятностью. Вариант с изменением данного параметра на самом деле говорит о том, что необходимо либо снизить нагрузку на систему, адаптировав её для других менее производительных задач, либо увеличить объём ОЗУ.

# **6.Синхронизация потоков: определение, взаимная блокировка потоков, принцип реализации процесса синхронизации, механизмы синхронизации потоков в Windows, атомарные операции в Windows, особенности синхронизации потоков в Linux.**

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

**Синхронизация потоков** – механизм, позволяющий потокам согласовывать свою работу с общими ресурсами.

**Взаимная блокировка потоков**

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

**Принцип реализации процесса синхронизации**

Ограничение одновременного доступа к ресурсам нескольких потоков.

**Механизмы синхронизации потоков в Windows**

**- Critical section ;**

**- Mutex;**

**- Semaphore;**

**- Atomic operation (interlocking function)**

**- Event;**

**- Writable timer.**

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

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

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

**Semaphore -** представляет собой счетчик, содержащий целое число в диапазоне от 0 до максимальной величины, заданной при его создании. Счетчик уменьшается каждый раз, когда поток успешно завершает функцию ожидания, использующую семафор, и увеличивается путем вызова функции ReleaseSemaphore. При достижении семафором значения 0 он переходит в несигнальное состояние, при любых других значениях счетчика его состояние — сигнальное. Такое поведение позволяет использовать семафор в качестве ограничителя доступа к ресурсу, поддерживающему заранее заданное количество подключений.

**Event -** объекты-события используются для уведомления ожидающих нитей о наступлении какого-либо события. Различают два вида событий - с ручным и автоматическим сбросом. Ручной сброс осуществляется функцией ResetEvent. События с ручным сбросом используются для уведомления сразу нескольких нитей. При использовании события с автосбросом уведомление получит и продолжит свое выполнение только одна ожидающая нить, остальные будут ожидать дальше.

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

Таймер ожидания может быть либо синхронизирующим (synchronization timer), либо сбрасываемым вручную уведомляющим (manual-reset notification timer) таймером. Синхронизирующий таймер связывается с функцией косвенного вызова, аналогичной процедуре завершения расширенного ввода/вывода, тогда как для синхронизации по сбрасываемому вручную уведомляющему таймеру используется функция ожидания.

**Атомарная операция**

**Атомарная операция —** операция, которая либо выполняется целиком, либо не выполняется вовсе; операция, которая не может быть частично выполнена и частично не выполнена.

**Атомарные операции в Windows -** это группа особых функций, названия которых начинаются с префикса Interlocked. Суть их в том, что каждая из них позволяет выполнить пару простых операций, но так, что они выполняются атомарно, то есть как бы «одним махом», так что их выполнение не может быть прервано другим потоком. Проиллюстрирую их использование на конкретном примере.

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

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

**void CReferenceCounter::AddRef( void ) {**

**::InterlockedIncrement( &m\_nReferences );**

**}**

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

**void CReferenceCounter::Release( void ) {**

**if( !::InterlockedDecrement( &m\_nReferences ) ) {**

**delete this;**

**}**

**}**

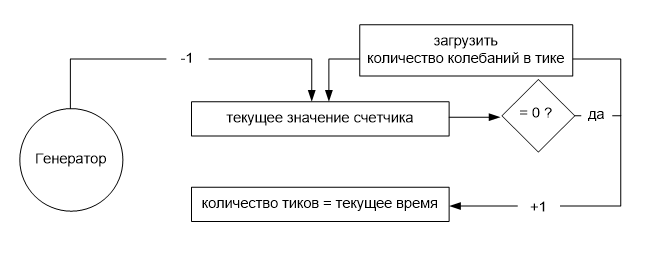
Таким образом, **InterlockedIncrement** и **InterlockedDecrement** выполняет атомарно сразу две операции: изменяют счетчик на единичку и сравнивают результат с нулём. Если бы эти функции не позволяли делать операцию сравнения, они вряд ли имели бы такую ценность, поскольку для сравнения с нулём нам пришлось бы снова прочитать значение счётчика из памяти, а в это время его мог бы изменить другой поток. Это общее свойство всех **Interlocked** функций: все они позволяют атомарно выполнить не одну, а как минимум две или даже три простых операции.

**Особенности синхронизации потоков в Linux**

Там сильно ограничено, фактически там есть только мьютексы и семафоры.

Для создания **потока** используется **библиотека pthread** и вызов pthread\_create, для синхронизации в этой же библиотеке описаны специальные объекты - **мьютексы**. **Мьютекс** - это объект который может принадлежать в некий момент времени только одному потоку и имеющий два состояния - занят и свободен. Поток пытающийся получить доступ к мьютексу в случае если последний занят будет остановлен системой до освобождения объекта. На этом собственно и основана синхронизация - перед использованием общего ресурса потоки сначала обращаются к мьютексу и в конечном счете выстраиваются в очередь.

# **7. Компьютерное время: принцип вычисления компьютерного времени, социальное и компьютерное время, единица измерения компьютерного времени, Universal Coordinated Time, POSIX-время, ожидающий таймер, состояния ожидающего таймера, периодические и непериодические таймеры, работа с ожидающими таймерами в Windows, особенности работы с ожидающим таймером в Linux.**



Другая разновидность часов создается из трех компонентов: кварцевого генератора, счетчика и регистра хранения. Если из кристалла кварца правильно вырезать пластину и подвести к ней напряжение, то ее можно заставить генерировать очень стабильный периодический сигнал, как правило, в диапазоне от нескольких сотен мегагерц до нескольких гигагерц в зависимости от выбранного кристалла. С помощью электронных схем этот опорный сигнал можно умножить на небольшое целое число, чтобы получить частоты до нескольких гигагерц или даже выше. В любом компьютере можно найти как минимум одну такую схему, которая обеспечивает различные компьютерные электронные схемы синхросигналом. Этот сигнал поступает в счетчик, заставляя его производить обратный отсчет до нуля. Когда значение счетчика становится нулевым, он выдает прерывание на центральный процессор.

**Социальное и компьютерное время**

Не монотонное: каждый год солнечный год увеличивается на 3 мс, люди измеряют время не точно, поэтому время от времени делались коррекции, например: в 1582 Папа Григорий XIII пропустил 10 дней календаря. Вычисление даты от Рождества по секундам не получится.

Поэтому эпоха Unix (POSIX-время) c 01.01.1970 0:00:00 в секундах. Используется 32 бита для представления числа. В 2038 г. счетчик перейдет в область отрицательных чисел. Секунда координации (по сообщению Международной службы вращения Земли, серверы точного времени): последняя секунда 30.06 или 31.12.

**Единица измерения компьютерного времени**

Тик))0)

Если про Windows мы говорим то это 1/1000 секунды

**Universal Coordinated Time**

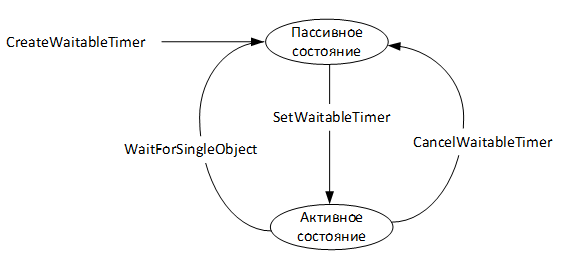
**Универсальное согласованное время** (на Гринвичском меридиане, раньше GMT – Greenwich Meridian Time), Международное бюро мер и весов (Париж), усредненное значение полученное на основе данных 50 лабораторий, оборудованных атомными часами (цезий-133) - TAI (International Atomic Time), расхождение с солнечными часами примерно 3мс (атомные часы отстают) в сутки, коррекция при ошибке в 800 мс.

**POSIX-время**

**Unix-время —** система описания моментов во времени, принятая в [Unix](https://ru.wikipedia.org/wiki/Unix) и других **POSIX-совместимых** [операционных системах](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). Определяется как количество секунд, прошедших с [полуночи](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%BD%D0%BE%D1%87%D1%8C) (00:00:00 [UTC](https://ru.wikipedia.org/wiki/%D0%92%D1%81%D0%B5%D0%BC%D0%B8%D1%80%D0%BD%D0%BE%D0%B5_%D0%BA%D0%BE%D0%BE%D1%80%D0%B4%D0%B8%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%B2%D1%80%D0%B5%D0%BC%D1%8F)) [1 января](https://ru.wikipedia.org/wiki/1_%D1%8F%D0%BD%D0%B2%D0%B0%D1%80%D1%8F) 1970 года (четверг); этот момент называют «[эпохой](https://ru.wikipedia.org/wiki/%D0%AD%D0%BF%D0%BE%D1%85%D0%B0_(%D0%B0%D1%81%D1%82%D1%80%D0%BE%D0%BD%D0%BE%D0%BC%D0%B8%D1%8F)) Unix» .

**Ожидающий таймер**

Объект синхронизации, два состояния: сигнальное – наступление заданного момента времени; несигнальное (активное и пассивное состояние) - ждет наступления заданного момента времени.



**Состояния ожидающего таймера**

Сигнальное и не сигнальное(активное и пассивное)

**Периодические и непериодические таймеры**

Таймер ожидания может быть либо синхронизирующим (synchronization timer), либо сбрасываемым вручную, уведомляющим (manual-reset notification timer) таймером. Синхронизирующий таймер связывается с функцией косвенного вызова, аналогичной процедуре завершения расширенного ввода/вывода, тогда как для синхронизации по сбрасываемому вручную уведомляющему таймеру используется функция ожидания.

Для начала потребуется создать дескриптор таймера, используя для этого функцию CreateWaitableTimer.

HANDLE CreateWaitableTimer(LPSECURITY\_ATTRIBUTES lpTimerAttributes, BOOL bManualReset, LPCTSTR lpTimerName);

Второй параметр, bManualReset, определяет, таймер какого типа должен быть создан — синхронизирующий или уведомляющий.

Первоначально таймер создается в неактивном состоянии, но с помощью функции SetWaitableTimer его можно активизировать и указать начальную временную задержку, а также длительность промежутка времени между периодически вырабатываемыми сигналами.

BOOL SetWaitableTimer(HANDLE hTimer, const LARGE\_INTEGER \*pDueTime, LONG IPeriod, PTIMERAPCROUTINE pfnCompletionRoutine, LPVOID lpArgToCompletionRoutine, BOOL fResume);

hTimer — действительный дескриптор таймера, созданного с использованием функции CreateWaitableTimer.

Второй параметр, на который указывает указатель pDueTime, может принимать либо положительные значения, соответствующие абсолютному времени, либо отрицательные, соответствующие относительному времени, причем фактические значения выражаются в единицах времени длительностью 100 наносекунд, а их формат описывается структурой FILETIME.

Величина интервала между сигналами, указываемая в третьем параметре, выражается в миллисекундах. Если это значение установлено равным 0, то таймер переводится в сигнальное состояние только один раз. При положительных значениях этого параметра таймер является периодическим и срабатывает периодически до тех пор, пока его действие не будет прекращено вызовом функции CancelWaitableTimer. Отрицательные значения указанного интервала не допускаются.

Четвертый параметр, pfnCompletionRoutine, применяется в случае синхронизирующего таймера и указывает адрес процедуры завершения, которая вызывается при переходе таймера в сигнальное состояние и при условии, что поток переходит в состояние дежурного ожидания.

При вызове этой процедуры в качестве одного из аргументов используется указатель, определяемый пятым параметром, plArgToComplretionRoutine.

Установив синхронизирующий таймер, вы можете перевести поток в состояние дежурного ожидания путем вызова функции SleepEx, чтобы обеспечить возможность вызова процедуры завершения. В случае сбрасываемого вручную уведомляющего таймера следует организовать ожидание перехода дескриптора таймера в сигнальное состояние. Дескриптор будет оставаться в сигнальном состоянии до следующего вызова функции SetWaitableTimer..

Последний параметр, fResume, связан с режимами энергосбережения.

Функция CancelWaitableTimer используется для отмены действия вызванной перед этим функции SetWaitableTimer, но при этом не изменяет сигнальное состояние таймера. Чтобы это сделать, необходимо в очередной раз вызвать функцию SetWaitableTimer.

**Особенности работы с ожидающим таймером в Linux**

По большому счету там есть этот таймер как вы помните ведется несколько часов несколько подсчетов.

**Linux:** четыре типа часов: REALTIME – системное время(настенное), MONOTONIC – с начала загрузки OS(монотонно возрастает), PROCESS – процессорное время( затраченное процессом), THREAD – процессорное время(затраченное потоком).

**Timerfd\_create() -** Вызов timerfd\_create() создаёт новый объект таймера и возвращает файловый дескриптор, который ссылается на таймер. В аргументе clockid задаются часы, которые используются для хода таймера; значение должно быть CLOCK\_REALTIME или CLOCK\_MONOTONIC.

**Ну в другом доке все))0)**

# **8.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 -** это часть операционной системы, назначение которой состоит в том, чтобы обеспечить пользователю удобный интерфейс при работе с данными, хранящимися на диске, и обеспечить совместное использование файлов несколькими пользователями и процессами.

**Логическая и физическая организация данных**

**Логическая организация файловой системы:**

* Типы файлов;
* Иерархическая структура файловой системы;
* Имена файлов;
* Атрибуты файлов;
* Логическая организация файла.

**Физическая организация файловой системы:**

* Физическая организация и адресация файла;
* Физическая организация FAT;
* Физическая организация NTFS.

**Отличия файловых систем**

Они отличаются в основном набором символов для ввода информации, которая хранится в оглавлении файловой системы, поддерживаются или не поддерживаются каталоги.

Там еще есть различия по объему файлов и тд.

**Оглавление файловой системы**

Оглавление файловой системы FAT строится таким образом, что в его элементе информация для связи с местом хранения файла состоит всего лишь из номера первого кластера для того файла, который именует данный элемент оглавления. Собственно таблица FAT в своих элементах может хранить:

* код свободного кластера (обычно это значение 0);
* условное значение для сбойного кластера;
* номер следующего кластера размещения;

**Файлы**

**Файл -** именованная область данных на носителе данных. Имеет собственный формат - способ организации данных внутри файла.

**Каталоги**

**Каталог** - файл содержащий информацию о месте расположения других файлов ( спец каталоги .(текущий)..(родительский))

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

**Основные функции файловой системы**

Создание файлов, создание каталогов, премещения и так далее

**Буферы ввода/вывода**

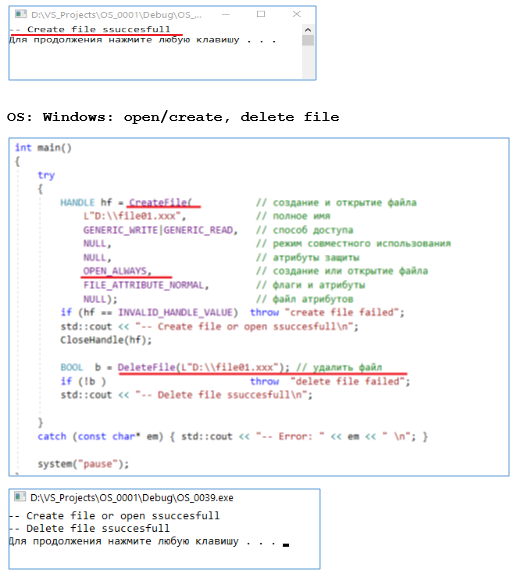
Области памяти для хранения физически считанных данных. Необходимы для устранения несоответствия между физическим и логическим чтением/записью. Сначала заполняется буфер кластерами (физически считанными данными), а затем осуществляется чтение логических файлов.

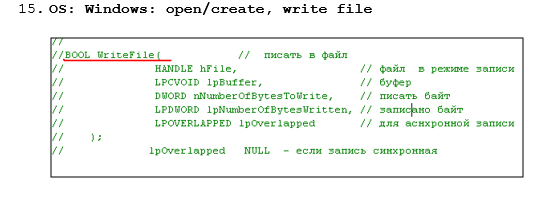
**Кэширование ввода/вывода**

**Кэширование -** перемещение в быстродействующую память, наиболее часто используемых данных (обычно упреждающее чтение).В состав ядра Windows входит специальная программа обеспечивающая кэширование данных. Кроме того кэширование осуществляет контроллер дисковода.

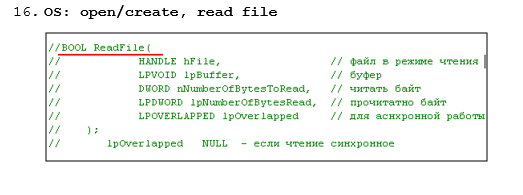
**Основные функции API файловой системы**



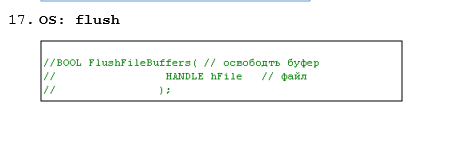




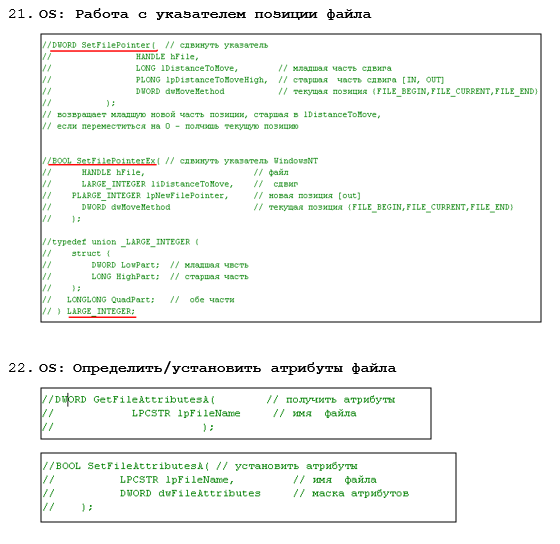


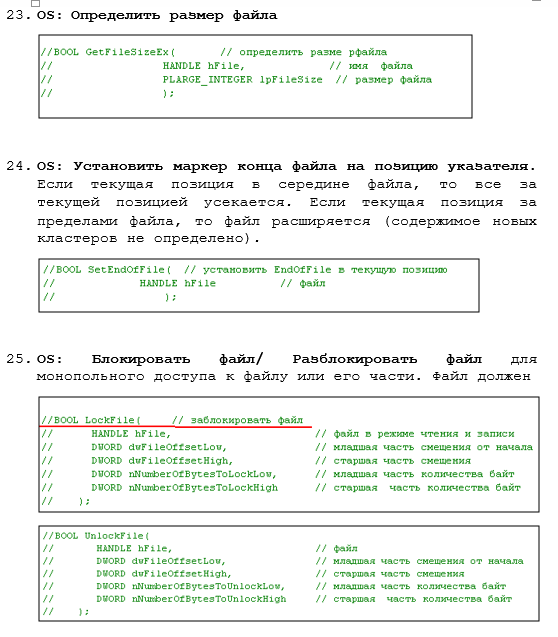


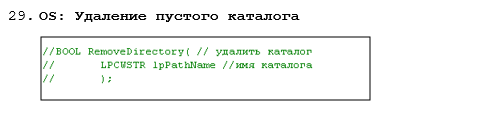












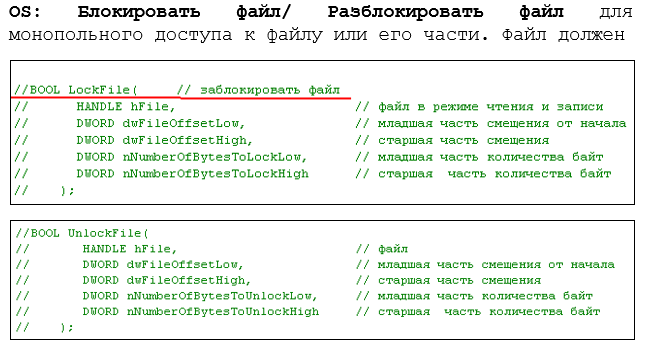
**Маркер файла**

EOF (end of file) индикатор, означающий что данные в источнике закончились.

**Текущая позиция файла**

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

**Блокировка файлов**



**Наблюдение за изменением в каталоге**



**Особенности устройства файловой системы в Linux**

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

**Ext2, Ext3, Ext4 или Extended Filesystem** - это стандартная файловая система для Linux. Она была разработана еще для Minix. Она самая стабильная из всех существующих, кодовая база изменяется очень редко и эта файловая система содержит больше всего функций. Версия ext2 была разработана уже именно для Linux и получила много улучшений. В 2001 году вышла ext3, которая добавила еще больше стабильности благодаря использованию журналирования. В 2006 была выпущена версия ext4, которая используется во всех дистрибутивах Linux до сегодняшнего дня. В ней было внесено много улучшений, в том числе увеличен максимальный размер раздела до одного экзабайта.

**JFS или Journaled File System** была разработана в IBM для AIX UNIX и использовалась в качестве альтернативы для файловых систем ext. Сейчас она используется там, где необходима высокая стабильность и минимальное потребление ресурсов. При разработке файловой системы ставилась цель создать максимально эффективную файловую систему для многопроцессорных компьютеров. Также как и ext, это журналируемая файловая система, но в журнале хранятся только метаданные, что может привести к использованию старых версий файлов после сбоев.

**ReiserFS** - была разработана намного позже, в качестве альтернативы ext3 с улучшенной производительностью и расширенными возможностями. Она была разработана под руководством Ганса Райзера и поддерживает только Linux. Из особенностей можно отметить динамический размер блока, что позволяет упаковывать несколько небольших файлов в один блок, что предотвращает фрагментацию и улучшает работу с небольшими файлами. Еще одно преимущество - в возможности изменять размеры разделов на лету. Но минус в некоторой нестабильности и риске потери данных при отключении энергии. Раньше ReiserFS применялась по умолчанию в SUSE Linux, но сейчас разработчики перешли на Btrfs.

**XFS** - это высокопроизводительная файловая система, разработанная в Silicon Graphics для собственной операционной системы еще в 2001 году. Она изначально была рассчитана на файлы большого размера, и поддерживала диски до 2 Терабайт. Из преимуществ файловой системы можно отметить высокую скорость работы с большими файлами, отложенное выделение места, увеличение разделов на лету и незначительный размер служебной информации.

**Btrfs** или B-Tree File System - это совершенно новая файловая система, которая сосредоточена на отказоустойчивости, легкости администрирования и восстановления данных. Файловая система объединяет в себе очень много новых интересных возможностей, таких как размещение на нескольких разделах, поддержка подтомов, изменение размера не лету, создание мгновенных снимков, а также высокая производительность. Но многими пользователями файловая система Btrfs считается нестабильной. Тем не менее, она уже используется как файловая система по умолчанию в OpenSUSE и SUSE Linux.

# **9. Механизм отображение файлов в памяти: последовательность системных вызовов Windows для создания образа файла в оперативной памяти, использование образа файла, как средства межпроцессного взаимодействия, особенности отображения файлов в linux.**

**Механизм отображения файлов в памяти**

**Отображение файла в память (на память)** — это способ работы с файлами в некоторых операционных системах, при котором всему файлу или некоторой непрерывной его части ставится в соответствие определённый участок памяти (диапазон адресов оперативной памяти). При этом чтение данных из этих адресов фактически приводит к чтению данных из отображенного файла, а запись данных по этим адресам приводит к записи этих данных в файл. Отображать на память часто можно не только обычные файлы, но и файлы устройств.

**Последовательность системных вызовов для создания файла в оперативной памяти**

**Первый шаг** – открытие или создание файла. Для этого используется функция CreateFile со следующими параметрами:

* 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 – темплейт файл, который предоставляет атрибуты и расширенные атрибуты для файла, который будет создан

**Второй шаг** – создать или открыть именованный или безымянный объект отображения в памяти файла с помощью функции CreateFileMappting():

* hFile – дескриптор (хэндл) файла
* SecurityAttributes – атрибуты защиты
* Protect – определяет страничную безопасность объекта отображения в памяти
* Нижняя и верхняя границы файлов
* Имя объекта

**Третий шаг** – сопоставить представление сопоставления файла с адресным пространством вызывающего процесса MapViewOfFile()

* Хэндл объекта отображения
* Тип доступа – FILE\_MAP\_ALL\_ACCESS, FILE\_MAP\_READ, FILE\_MAP\_WRITE
* Всякая еще хуита

**Использование образа файла, как средства межпроцессного взаимодействия**

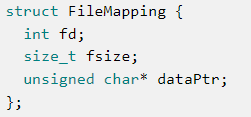
Сопоставление файлов можно использовать в качестве механизма межпроцессорного взаимодействия

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

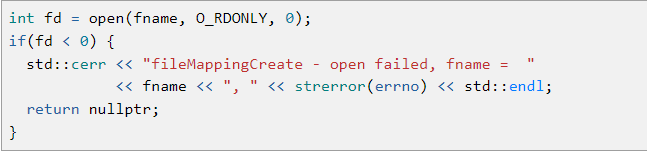
**Особенности отображения файлов в Linux**

Для отображения файлов в Linux идут другие шаги

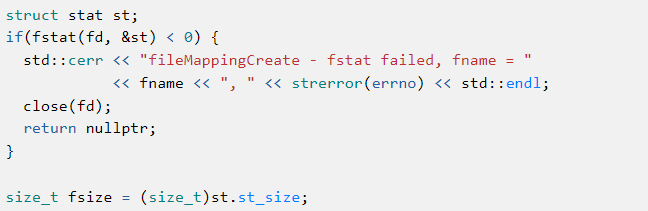
**Первый шаг** – создать структуру, которая будет хранить дескриптор фала, размер и указатель на участок памяти с отображением.



**Второй шаг** – получаем дескриптор фала открывая его с помощью функции open();



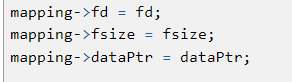
**Третий шаг** – узнать размер файла с помощью функции fstat



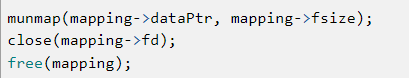
**Четвертый шаг** – создаем отображение файла в память с помощью функции mmap()



**Пятый шаг** – заполняем структуру FileMapping и возвращаем указатель на нее в качестве результата



**Шестой шаг** – в конце работы закрыть дескриптор и освободить ресурсы



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

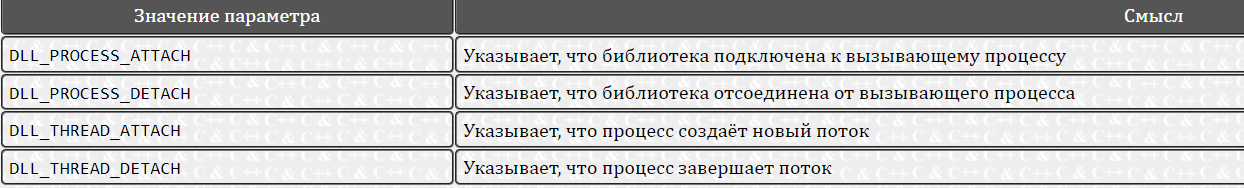
**Динамическая библиотека (DLL)** - это библиотека динамической компоновки, которая выступает в роли общего хранилища ресурсов и функций для других приложений.

DLL при использовании загружается в оперативную память ДЛЯ ВСЕХ, без копирования по отдельности в каждый использующий ее процесс, в отличии от статической библиотеки .lib, которая при сборке приложения полностью копируется в него. При использовании динамической сборки в приложение встраивается только ссылка на DLL файл в системе, а когда она понадобится приложение благодаря библиотеке импорта сопоставит ее с областью памяти приложения.

**Структура**

**Функция LibMain –** получает управление при загрузке библиотеки в память имеет следующие параметры :

* hInstDll – хэндл DLL модуля. Значение – базовый адрес DLL
* Reason – код, который обозначает с какой целью была вызвана библиотека.
* lpReserved – параметр зарезервирован для внутреннего использования Windows



**Экспортируемые функции (можно отнести к структуре)**

Функции, которые будут экспортированы для других приложений помечаются как: extern "C" \_\_declspec(dllexport)

**Extern “C”** указывает на то, что функция расположена в другом месте и что использует соглашение о вызове языка С

**\_\_declspec(dllexport)** – в объектный файл (файл с машинным кодом) добавляется директива импорта. Это облегчает работу с DLL тем, что не нужно создавать .def файл, в котором описываются функции, которые будут доступны для использования.

**Загрузка динамической библиотеки**

Прежде чем вызывать подпрограммы, включенные в dll-библиотеку, их надо импортировать в приложение. Это можно сделать двумя способами – путём объявления внешней процедуры или функции или динамически с помощью Win32 API функций. Но в любом случае подпрограммы из dll-библиотеки будут импортированы только во время работы приложения.

Чтобы загрузить dll-библиотеку, операционная система должна найти ее. Поиск осуществляется в следующих местах:

* Директория, из которой загружено приложение, требующее dll-библиотеку;
* Текущая директория;
* Системная директория (обычно C:\Windows\System32);
* Системная директория для 16-битных приложений (обычно C:\Windows\System);
* Windows-директория;
* Директории, указанные в переменной окружения PATH.

Последовательность просмотра директорий может быть разной, это зависит от настроек, но, в общем, просматриваются перечисленные директории до нахождения требуемой библиотеки. Если ни в одной из перечисленных директорий библиотека не найдена, приложение получает уведомление об ошибке.

**Динамический вызов функций динамической библиотеки**

Чтобы использовать библиотеку DLL посредством явного связывания, необходимо вызвать функцию для явной загрузки библиотеки DLL во время выполнения. Для явного связывания с библиотекой DLL приложение должно выполнить следующие действия:

* Вызвать LoadLibrary или аналогичную функцию для загрузки библиотеки DLL и получения дескриптора модуля.
* Вызвать GetProcAddress, чтобы получить указатель для каждой экспортированной функции, которую вызывает приложение. Поскольку приложения вызывают функции DLL с помощью указателя, компилятор не создает внешние ссылки и нет необходимости связываться с библиотекой импорта. Тем не менее необходимо использовать инструкцию typedef или using, определяющую сигнатуру вызова для вызываемых экспортированных функций.
* По завершении работы с библиотекой DLL вызовите FreeLibrary.

**Создание и применение библиотеки импорта**

Чтобы использовать библиотеку DLL посредством неявного связывания, исполняемые файлы клиента должны получить от ее поставщика следующие файлы:

* Один или несколько файлов заголовка (с расширением h), в которых содержатся объявления экспортированных данных, функций и классов C++ в библиотеке DLL. Все классы, функции и данные, экспортируемые из библиотеки DLL, должны быть отмечены в файле заголовка как \_\_declspec(dllimport).
* Библиотека импорта, которая связывается с исполняемым файлом. Библиотека импорта создается компоновщиком при построении DLL. Дополнительные сведения см. в разделе Использование LIB-файлов в качестве входных данных для компоновщика.
* DLL-файл библиотеки.

Чтобы использовать данные, функции и классы в библиотеке DLL посредством неявного связывания, исходный файл клиента должен содержать файлы заголовков, в которых они объявляются. С точки зрения кода вызовы экспортированных функций выполняются так же, как и в случае с любыми другими функциями.

При вызове исполняемого файла операционная система должна иметь возможность найти DLL-файл. Это означает, что при установке приложения необходимо либо развернуть нужную библиотеку DLL, либо проверить ее наличие.

Библиотека импорта создается либо на основе dll-файла библиотеки, либо на основе файла определения модуля (.def), используемого для создания DLL-библиотеки.

После компиляции dll Visual Studio создаст библиотеку импорта в виде lib-файла, расположенного в том же каталоге, что и исходный dll или def-файл. Этот файл необходимо включить в проект создаваемого вами приложения, пользующегося функциями DLL-библиотеки.

# **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.**

**Спецификация COM**

**Объектная модель компонента Майкрософт (COM)** определяет стандарт взаимодействия для создания многократно используемых библиотек программного обеспечения, взаимодействующих во время выполнения.

Библиотеки COM можно использовать без необходимости их компиляции в приложение.

**Понятие позднего связывания**

**Позднее связывание -** подключение программного модуля во время исполнения программы

**COM-интерфейс**

**COM-интерфейс -** Средство, с помощью которого объект COM предоставляет свои функциональные возможности (службы) для внешних клиентов. Объект COM снабжает интерфейсом каждый набор методов и свойств. Любой объект COM имеет один или более интерфейсов.

Представляет собой абстрактный класс, который наследуется от IUnknown и содержит в себе сигнатуры методов.

**Стандартные COM-интерфейсы**

**IUnknown -** Этот интерфейс, который должны поддерживать все COM объекты, включает следующие подпрограммы:

* **QueryInterface -** возвращает клиенту указатель на запрошенный интерфейс IID. С помощью полученного указателя клиент может вызвать любой из реализованных методов интерфейса. В качестве IID клиент может указать тип класса интерфейс – в этом случае компилятор самостоятельно извлекает соответствующий GUID.
* **AddRef**
* **Release**

Методы AddRef и Release используются для того, чтобы объект COM мог самостоятельно отслеживать продолжительность своего существования. Эти методы просто изменяют число ссылок на объект. Когда число ссылок на объект становится равным нулю, объект удаляется из памяти COM сервером.

**IClassFactory** - Стандартный интерфейс, который поддерживает фабрики класса для создания компонентов

Он наследуется от IUnknown так что реализует его методы, а так же еще 2 метода:

* CreateInstance - создает экземпляр компонента
* LockServer - обеспечивающий блокировку программы сервера в памяти. Блокируя сервер для других программ, клиент получает уверенность в том, что доступ к нему будет быстро получен. Обычно это делается в целях повышения производительности.

**Структура COM-клиента**

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

**COM-клиент** должен «знать»:

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

**CLSID** - идентификатор объекта(GUID)

COM-сервер может иметь тип:

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

Шаги работы с COM со стороны клиента:

* CoInitialize(NULL) – инициализация OLE32.dll – библиотека обеспечивающая работу COM-приложений
* CoCreateInstance() – Функция для создания COM-объекта на клиенте. Параметры – Идентификатор объекта, тип сервера, идентификатор интерфейса, выходной параметр куда будет помещен объект
* Release – функция, которая используется для декремента счетчика ссылок на объект.
* CoFreeUnusedLibraries – завершение работы с библиотекой

**Структура COM-сервера**

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

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

COM-сервер может иметь тип: CLSCTX\_INPROC\_SERVER (DLL внутрипроцессный сервер); CLSCTX\_LOCAL\_SERVER (EXE-сервер за границами процесса, но та том же компьютере); CLSCTX\_REMOTE\_SERVER (EXE-сервер на удаленном компьютере).

СОМ-сервер представляет собой приложение или библиотеку, которая предоставляет услуги приложению-клиенту или библиотеке. СОМ-сервер содержит один или более СОМ-объектов, где СОМ-объекты выступают в качестве наборов свойств, методов и интерфейсов.

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). CLSID - всего лишь GUID, который применяется при обращении к СОМ-объекту. После передачи CLSID, СОМ-сервер должен обеспечить так называемую фабрику класса, которая создает экземпляры СОМ-объектов.

**Экспортируемые стандартные функции**

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

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

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

**DllRegisterServer** - Добавляет записи в системный реестр. Для регистрации DLL СОМ-сервера в системном реестре Windows. При регистрации СОМ-класса в системном реестре создается раздел в HKEY\_CLASSES\_ROOT\CLSID\{XXXXXXXX-XXXX-XXXX-xxxx-xxxxxxxx}, где число, записанное вместо символов х, представляет собой CLSID данного СОМ-класса

**DllUnregisterServer** - Удаляет записи из системного реестра

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

**Регистрация COM сервера**

Процесс регистрации заключается в добавлении clsid компонента и ссылки на dll в реестре

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

* Ищет в библиотеке точку входа и вызывает функцию DllRegisterServer либо DllUnRegisterServer данной библиотеки и смотрит на возвращаемый результат.
* Ищет в библиотеке точку входа и вызывает функции DllInstall / DllUnInstall.

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

**Понятие дискреционной системы безопасности**

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

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

Дискреционное разграничение доступа реализуется обычно в виде матрицы доступа, строки которой соответствуют субъектам компьютерной системы, а столбцы — ее объектам. Элементы матрицы доступа определяют права доступа субъектов к объектам. В целях сокращения затрат памяти матрица доступа может задаваться в виде списков прав субъектов (для каждого из них создается список всех объектов, к которым разрешен доступ со стороны данного субъекта) или в виде списков контроля доступа (для каждого объекта информационной системы создается список всех субъектов, которым разрешен доступ к данному объекту).

**Типы Windows-пользователей**

Все учетные записи в зависимости от своих функциональных возможностей разделяются на **три типа**: **администратор** компьютера, **ограниченная учетная запись** и **гостевая учетная запись**.

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

Учетная запись **администратора** позволяет выполнять следующие операции:

* Создание, удаление и редактирование учетных записей пользователей (в том числе и собственной учетной записи);
* Операции со своим паролем (создание, редактирование, удаление);
* Установка и удаление программ и оборудования, редактирование их параметров и свойств;
* Чтение всех общих файлов;
* Внесение изменений в конфигурацию на уровне системы.

Что касается **гостевой записи**, то она формируется автоматически в процессе установки системы, и предназначена для сторонних пользователей, не имеющих на данном компьютере собственных учетных записей. Под учетной записью гостя нет доступа к файлам, папкам, параметрам и приложениям, которые защищены паролем. С помощью соответствующих ссылок можно включать и отключать учетную запись гостя (в первом случае сторонние пользователи смогут войти в систему, во втором – нет).

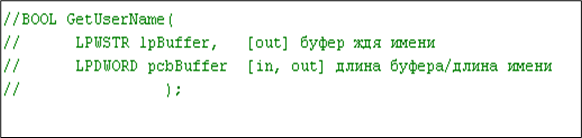
**Группы пользователей**

**Группа пользователей** – это собрание учетных записей **пользователей** **Windows**, имеющих одинаковые права безопасности.

* **Администраторы**. Неограниченный доступ.
* **Операторы архива**. Члены данной группы имеют права создания резервной копии даже тех объектов, к которым не имеют доступа.
* **Опытные пользователи**. Толку от них немного, так как группа включена только для совместимости с предыдущими версиями
* **Пользователи системного монитора**. Есть чудесная вещь под названием *Системный монитор(perfmon.msc*), с помощью которого 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**. Пользователи этой группы смогут входить в систему через удаленный рабочий стол.
* **Пользователи журналов производительности**. 4-ая группа дает только поверхностный доступ к *Системному монитору*. Данная группа дает более полные права.
* **Пользователи DCOM**. Пользователи группы могут манипулировать объектами распределенной модели DCOM.
* **Криптографические операторы**. Члены данной группы могут выполнять криптографические операции.

**Возможности API управления пользователями и группами**

**1. Имя текущего пользователя**

****

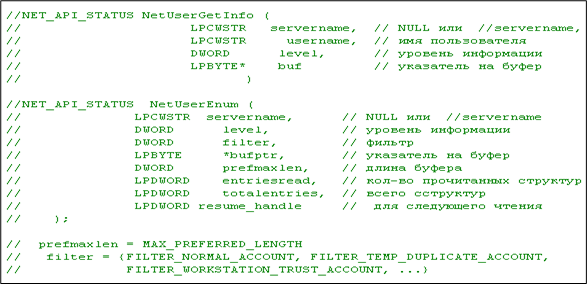
**2. Информация о пользователей**

****

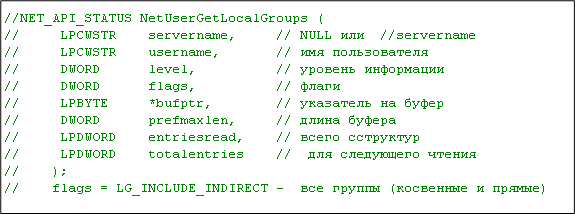
**3. Освободить память**

****

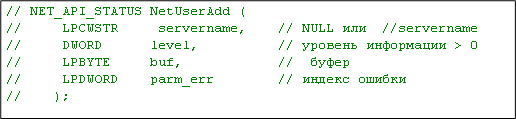
**4. Получить коллекцию пользователей**

****

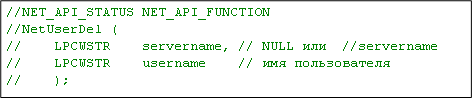
**5. Группы пользователя**

****

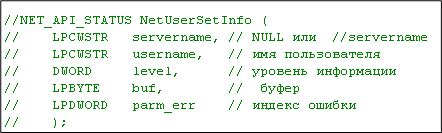
**6. Добавить пользователя**

****

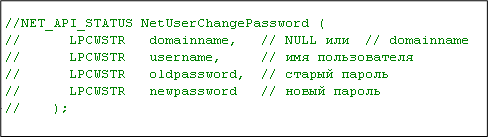
**7. Удалить пользователя**

****

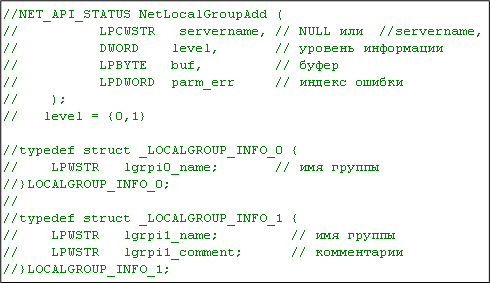
**8. Изменить информацию о пользователе**

****

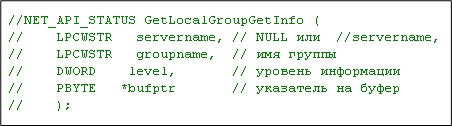
**9. Изменить пароль**

****

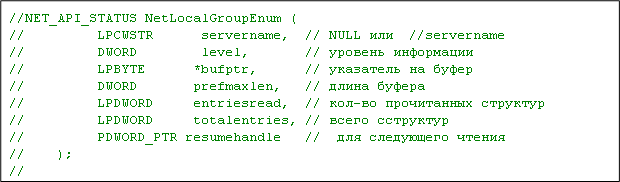
**10. Добавление группы**

****

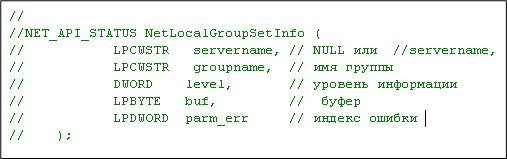
**11. Получить информацию о группе**

****

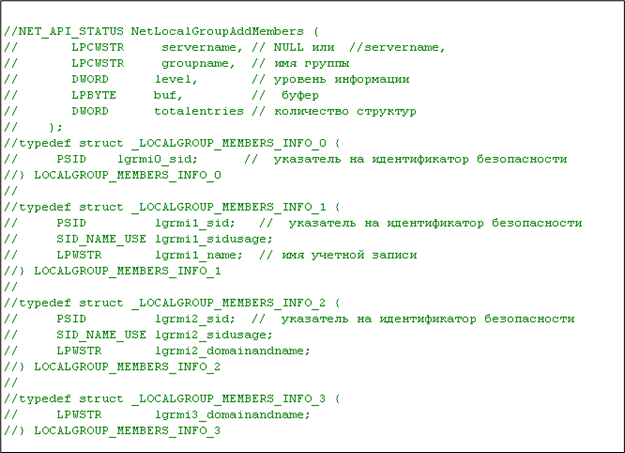
**12. Получить коллекцию групп**

****

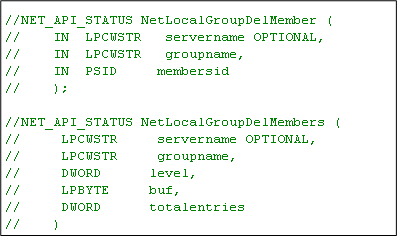
**13. Изменить информацию о группе**

****

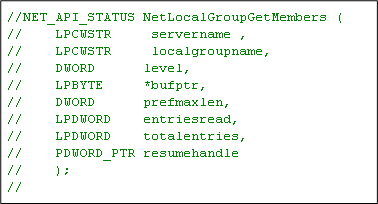
**14. Добавление членов локальной группы (исп. INFO\_3)**

****

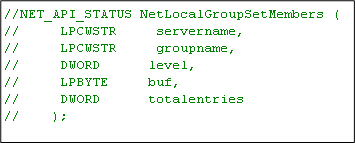
**15. Удаление членов группы**

****

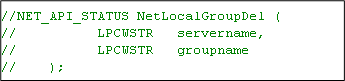
**16. Получить коллекцию членов группы**

****

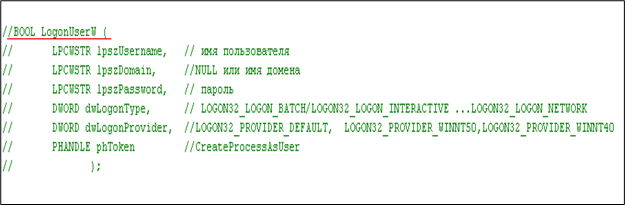
**17. Установка членов локальной группы**

****

**18. Удалить группу**

****

**19. Подключение пользователя**

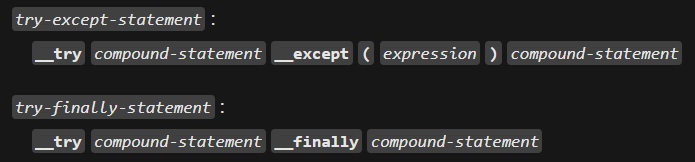
****

# **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.**

**SEH (structured exception handling) –** механизм структурной обработки событий в Windows.

**Исключение –** событие в программе, произошедшее во время ее выполнения, в результате котор ого нормальное выполнение программы становится невозможным.

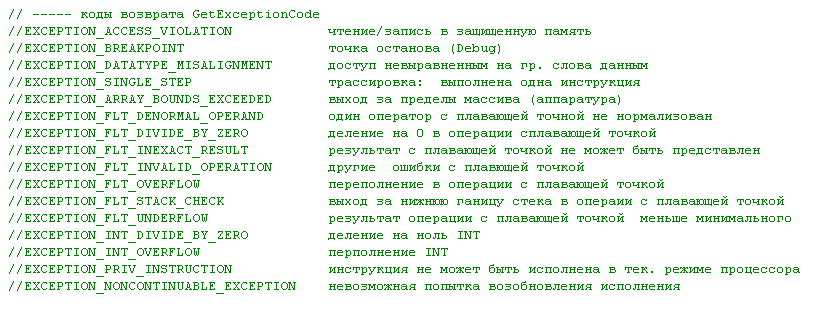
**Программные конструкции**



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

**Фильтры**

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

****

**GetExceptionInformation() -** Извлекает независимое от компьютера описание исключения и сведения о состоянии компьютера, которое существует для потока при возникновении исключения. Эту функцию можно вызывать только из выражения фильтра обработчика исключений.

**Возможности API для структурной обработки ошибок**

**UnhandledExceptionFilter() -** Определяемая приложением функция, которая передает необработанные исключения отладчику, если процесс отлаживается. В противном случае он необязательно отображает окно сообщения об ошибке приложения и вызывает выполнение обработчика исключений. Эта функция может быть вызвана только из выражения фильтра обработчика исключений.

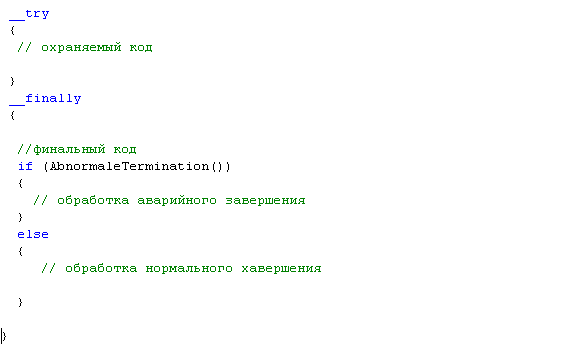
**SetUnhandledExceptionFilter() -** Позволяет приложению заменять обработчик исключений верхнего уровня каждого потока процесса.

После вызова этой функции, если в процессе, который не отлаживается, возникает исключение, и исключение попадает в необработанный фильтр исключений, этот фильтр вызовет функцию фильтра исключений, указанную параметром lpTopLevelExceptionFilter.

**Генерация ошибок**

**RaiseException()**

**Финальная обработка исключений**

****

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

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

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

**Применение стандартных потоков для ввода/вывода**

Функция CreateFile позволяет процессу получить дескриптор для входного буфера консоли и активного буфера экрана, даже если STDIN и STDOUT были перенаправлены. Чтобы открыть дескриптор для входного буфера консоли, укажите значение CONIN$ при вызове CreateFile. Укажите значение CONOUT$ при вызове CreateFile, чтобы открыть дескриптор для активного буфера экрана консоли. CreateFile позволяет указать доступ только для чтения и записи в возвращаемом дескрипторе.

**Возможности API для управления консолью**

**GetStdHandle()** – получить дескриптор консоли

**GetConsoleCursorInfo()** – извлекает информацию о размерах и виде курсора для заданного экранного буфера. Записывает результат в структуру CURSOR\_CONSOLE\_INFO

**SetConsoleCursorInfo() –** устанавливает размер и вид курсора для заданного экранного буфера консоли

**SetConsoleCursorPosition() –** устанавливает позицию курсора в заданном экранном буфере консоли

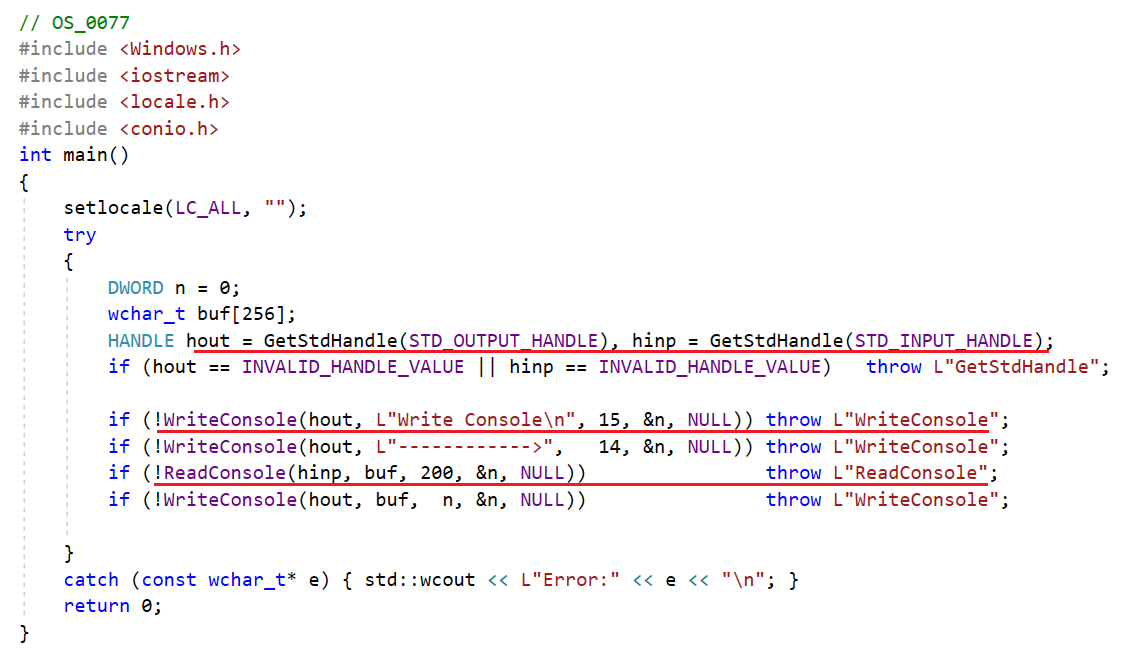
**SetConsoleActiveScreenBuffer() –** устанавливает новый активный буфер консоли

**FillConsoleOutputAttribute**

**SetConsoleTextAttribute()**

**SetConsoleTitle()**

**WriteConsole, ReadConsole –** высокоуровневый ввод/вывод

****

**ReadConsoleInput() – низкоуровневый ввод**

****

**WriteConsoleInput – низкоуровневый вывод**

****

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

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

* синхронный ввод - вывод (I/O) файла
* асинхронный ввод - вывод (I/O) файла.

Асинхронный ввод - вывод (I/O) файла также называется как перекрывающий(overlapped) ввод - вывод.

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

**Особенности программирования асинхронного ввода/вывода**

Первый вариант

* Открыть файл с помощью CreateFile с атрибутом FILE\_FLAG\_OVERLAPPED
* Объявить структуру \_OVERLAPPED – хранит информацию, которая используется в операции асинхронного ввода/вывода
* При вызове WriteFile/ReadFile в качестве одного из параметров передать указатель на структуру \_OVERLAPPED
* Использовать WaitForSingleObject(FileHandle, time) для ожидания завершения

Второй вариант

* Открыть файл с помощью CreateFile с атрибутом FILE\_FLAG\_OVERLAPPED
* Создать Event с автоматическим сбросом
* Объявить структуру \_OVERLAPPED – хранит информацию, которая используется в операции асинхронного ввода/вывода
* Поместить объект события в \_OVERLAPPED
* При вызове WriteFile/ReadFile в качестве одного из параметров передать указатель на структуру \_OVERLAPPED
* Использовать WaitForSingleObject(EventObject, time) для ожидания завершения

**Асинхронное блокирование файлов**

**LockFile() –** блокирует файл или его какой-то регион байтов и дает доступ к этому региону только вызывающему процессу.

**UnlockFile() –** разблокирует доступ к файлу или региону байт

**Определить состояние асинхронной операции**

**GetOverlappedResult() -** извлекает результаты асинхронной (перекрывающей) операции с указанным файлом, именованным каналом или с устройством коммуникации.

**HasOverlappedIoCompleted() –** функция, которая проверяет закончилась ли асинхронная операция

**ReadFileEx/WriteFileEx –** функции, которые позволяютсчитать или записать асинхронно а так же имеют параметр – ссылка на функцию (callback) которая исполнится после операции чтения/записи

**Порты завершения**

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

**CreateIOComplitionPort()** - создает порт завершения ввода-вывода и связывает один или несколько дескрипторов файлов с этим портом.

Каждая завершившаяся тем или иным образом операция ввода/вывода, назначенная на данный порт завершения, сообщает о своем возврате с помощью специального системного пакета, который поступает в очередь завершенных запросов порта. Для обработки этой очереди и предназначены наши рабочие потоки. Достигается это с использованием функции **GetQueuedCompletionStatus()**

После того как вы закроете все сокеты, необходимо завершить все рабочие потоки порта завершения. Для этого воспользуемся функцией **PostQueuedCompletionStatus**(), которая отправит потоку пакет, заставляющий прекратить работу.

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

## Архитектура

Архитектура докера: Docker использует клиент-серверную архитектуру.  
Docker - это сервис для запуска приложений в контейнере

## Назначение

Задачи которые решает Docker:

1. все приложения запускаются в изолированной среде
2. легко запускать приложения на разных серверах (компьютерах)
3. все зависимости приложения устанавливаются внутри контейнеров
4. приложение внутри контейнера можно легко масштабировать путем увеличения количества контейнеров, то есть одно и тоже приложение можно запустить в разных контейнерах
5. очень удобно использовать в процессе разработки приложений (все зависимости находятся внутри контейнера)

## Принципы устройства

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)

## Образы

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.

## Контейнеры

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.

## Основные команды для работы с Docker

Docker version - показывает информацию о клиенте и сервере Docker

Docker ps -a -показывает список запущенных и остановленных контейнеров (без -a только запущенные)

Docker Images - список всех образов которые находятся на компьютере

Docker run hello-world - создает и запускает контейнер (hello-world - название образа)

Docker pull hello-world - скачает образ

Docker run hello-world:latest можно указать версию образа

Docker rm id - удалить контейнер

Docker container prune - удалить всю историю контейнеров

Docker container inspect Id контейнера - выводит все дали определенного контейнера

Docker run -d nginx - создает и запускает контейнер в фоновом режиме (nginx - название образа)

Docker stop id - остановить контейнер

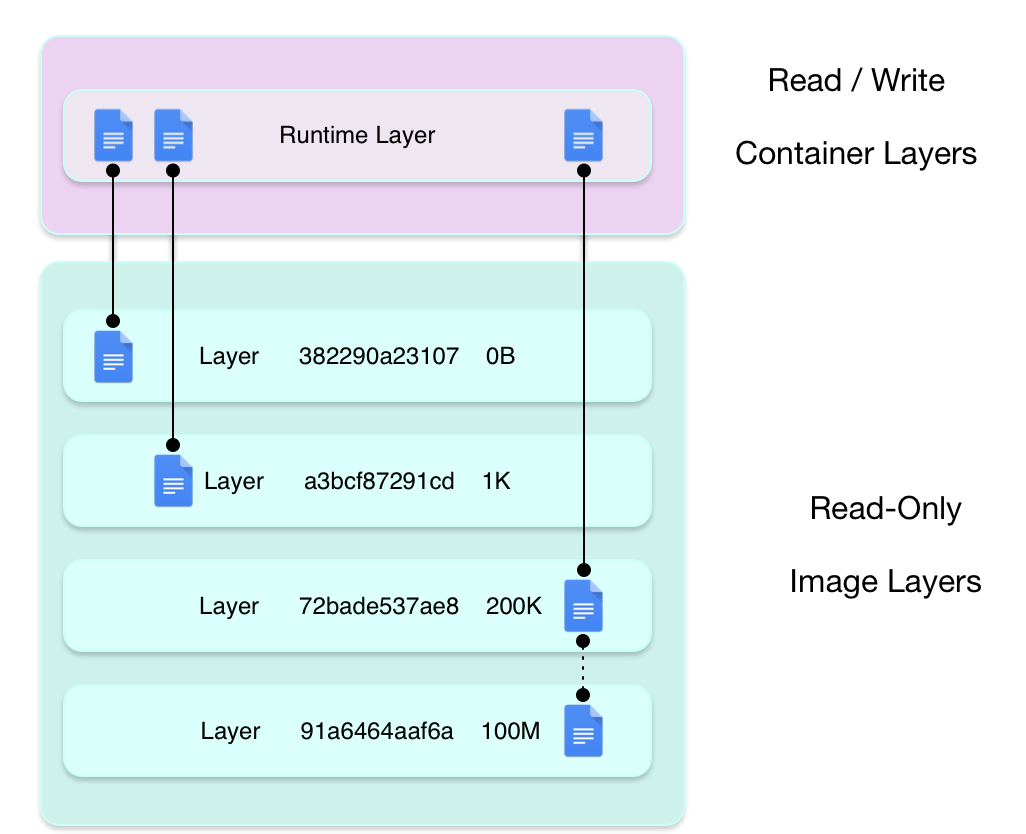
Docker kill id - процесс будет остановлен моментально

Docker exec -it id bash - запустит дополнительный процесс уже в запущенном процессе; exec - выполняет команду в запущенном контейнере; -it - опции подключения интерактивного терминала; bash - название процесса

Docker run -d --name my\_nginx nginx - запустить контейнер nginx с именем my\_nginx

docker run -d  -p 8080:80 nginx - запускаем контейнер с образом nginx в фоновом режиме с публицкией порта и указанием внешнего порта и внутреннего порта.

## Файловая система UnionFS



**UnionFS** — вспомогательная [файловая система](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), производящая [каскадно-объединённое монтирование](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%81%D0%BA%D0%B0%D0%B4%D0%BD%D0%BE-%D0%BE%D0%B1%D1%8A%D0%B5%D0%B4%D0%B8%D0%BD%D1%91%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BE%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) других файловых систем. Он позволяет прозрачно накладывать файлы и каталоги отдельных файловых систем, известных как ветки, образуя единую согласованную файловую систему.  
Содержимое каталогов с одним и тем же путем в объединенных ветвях будет отображаться вместе в одном объединенном каталоге в новой виртуальной файловой системе.

**монтирование объединения** — это способ объединения нескольких каталогов в один, который, по-видимому, содержит их объединенное содержимое .

Это позволяет файловой системе отображаться как доступная для записи, но фактически не разрешает запись для изменения файловой системы, также известную как [**копирование при записи.**](https://en.wikipedia.org/wiki/Copy-on-write)

Механизм **копирования при записи** ([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) *Copy-on-write, COW*) используется для оптимизации многих процессов, происходящих в [операционной системе](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), таких как, например, работа с [оперативной памятью](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) или [файлами](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB) на диске.

Идея подхода *copy-on-write* заключается в том, что при чтении области данных используется общая копия, в случае изменения данных — создается новая копия

UnionFS (Объединенная файловая система) является механизмом объединения нескольких файловых систем в одну единую файловую систему. Он позволяет создавать логические представления файловых систем, где данные могут быть совмещены или переопределены из разных источников.

Принцип работы UnionFS состоит в том, что несколько файловых систем объединяются в одну, и при обращении к файлам система проходит по каждой файловой системе в определенном порядке, пока не найдет соответствующий файл. Это позволяет создавать иерархические структуры файловых систем, где файлы из разных источников (например, разных дисков или сетевых ресурсов) могут быть объединены и представлены как одно целое.

Преимущества UnionFS:

**Объединение различных источников данных:** UnionFS позволяет объединять файлы из разных источников в одну файловую систему. Например, вы можете объединить файловую систему только для чтения и файловую систему для записи, чтобы создать логическую файловую систему, которая позволяет читать данные из одного источника и записывать данные в другой.

**Переопределение файлов:** UnionFS позволяет переопределить файлы, то есть, если файл существует в нескольких источниках, вы можете определить, какой файл будет виден при обращении к нему. Это полезно, например, при создании слоев файловой системы для версионирования или для внесения изменений в существующие файлы.

**Эффективное использование ресурсов:** UnionFS позволяет использовать файлы из разных источников без необходимости физического копирования или перемещения. Это позволяет сэкономить дисковое пространство и упростить управление файлами.

UnionFS является основой для многих систем контейнерной виртуализации, таких как Docker, где он используется для объединения слоев образов контейнеров и создания множественных файловых систем внутри контейнера. Это позволяет контейнерам быть эффективными, портативными и легко обновляемыми.

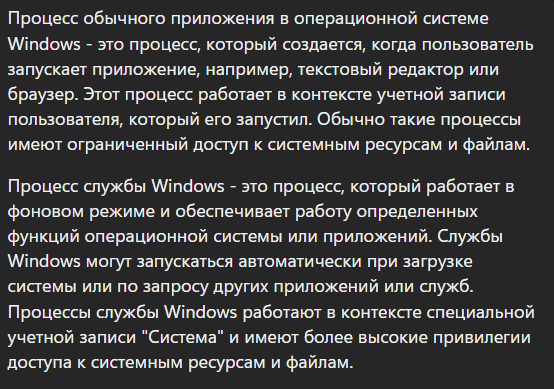
# --\*\*. Windows-сервисы: понятие и назначения сервиса, структура сервиса, порядок разработки и принцип работы сервиса, команды управления сервисом.

## Определение

**Windows service -** процесс, выполняющий служебные функции;

Программа, которая запускается при загрузке операционной системы;

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



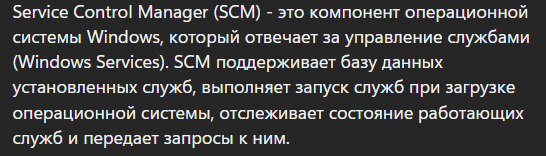
1. **Windows Services:** сервис может быть запущен при загрузке ОС или из приложения.

2. **Windows Services:** проблемы с отладкой (трассировка в файл).

Windows Services могут иметь проблемы с отладкой и трассировкой в файл. Это связано с тем, что сервисы запускаются в фоновом режиме и не имеют пользовательского интерфейса, что затрудняет их отладку. Для решения этой проблемы можно использовать специальные инструменты для отладки Windows Services, которые позволяют выполнять трассировку в файл и анализировать логи для выявления ошибок.

**Service Control Manager (SCM)**

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



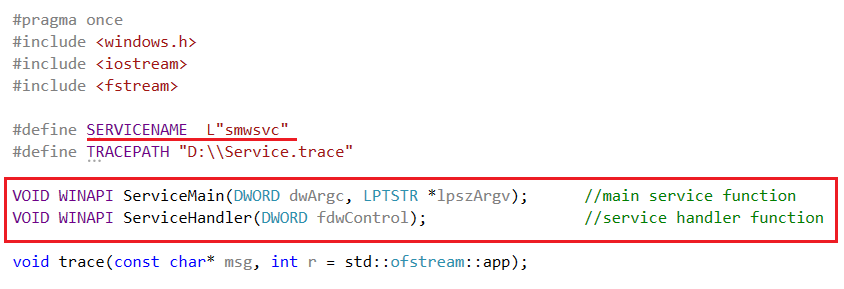
## Структура

**ServiceMain -** имеет доступ к аргументам командной строки для службы так, как выполняет основная функция консольного приложения. Первый параметр содержит число аргументов, передаваемых службе во втором параметре. Всегда будет хотя бы один аргумент. Второй параметр — это указатель на массив строковых указателей. Первым элементом в массиве всегда является имя службы.

Функция ServiceMain должна немедленно вызвать функцию **RegisterServiceCtrlHandlerEx**, чтобы определить функцию HandlerEx, которая обработает запросы на управление. Затем, она должна вызвать функцию SetServiceStatus, чтобы отправить информацию о статусе диспетчеру управления службами. После этих вызовов, функция должна завершить инициализацию службы.

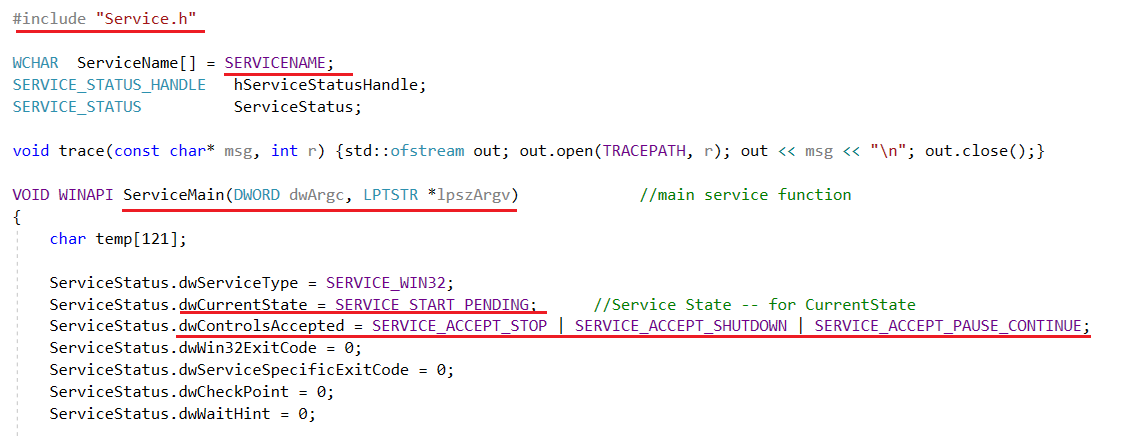
**ServiceHandler -** функция обработчика, которая вызывается диспетчером управления, когда процесс обслуживания получает запрос управления от программы управления службой

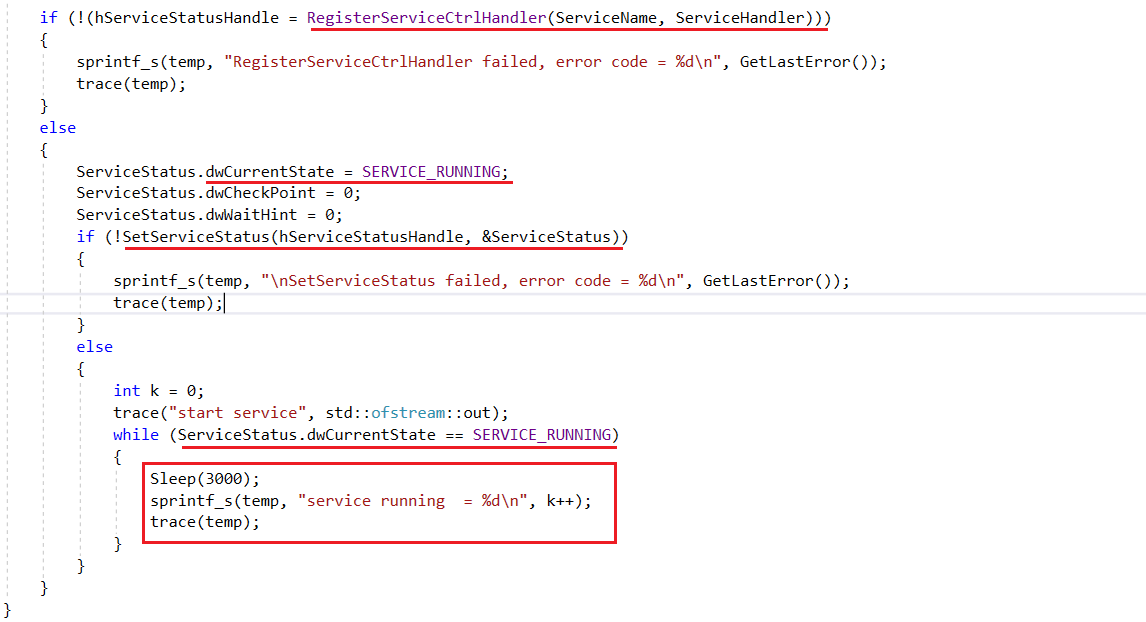
**Windows Services:** приложение-процесс сервиса, приложение, которое регистрирует 2 функции обратного вызова. Функция **StartServiceCtrlDispatcher** должна быть вызвана в течении 30мс с момента старта приложения.



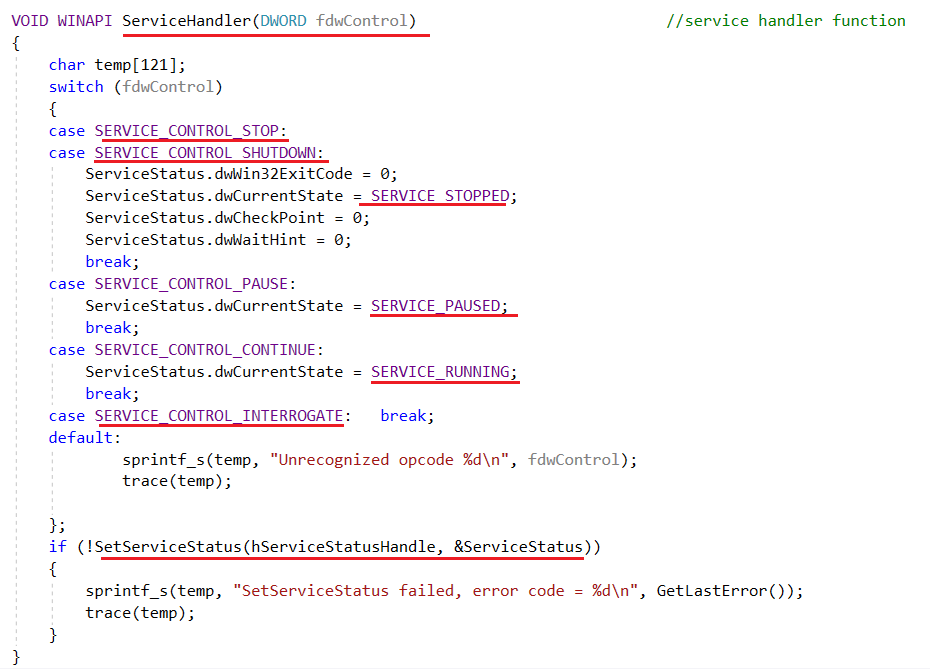


**1-ый обратный вызов: основная (main) функция сервиса**





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

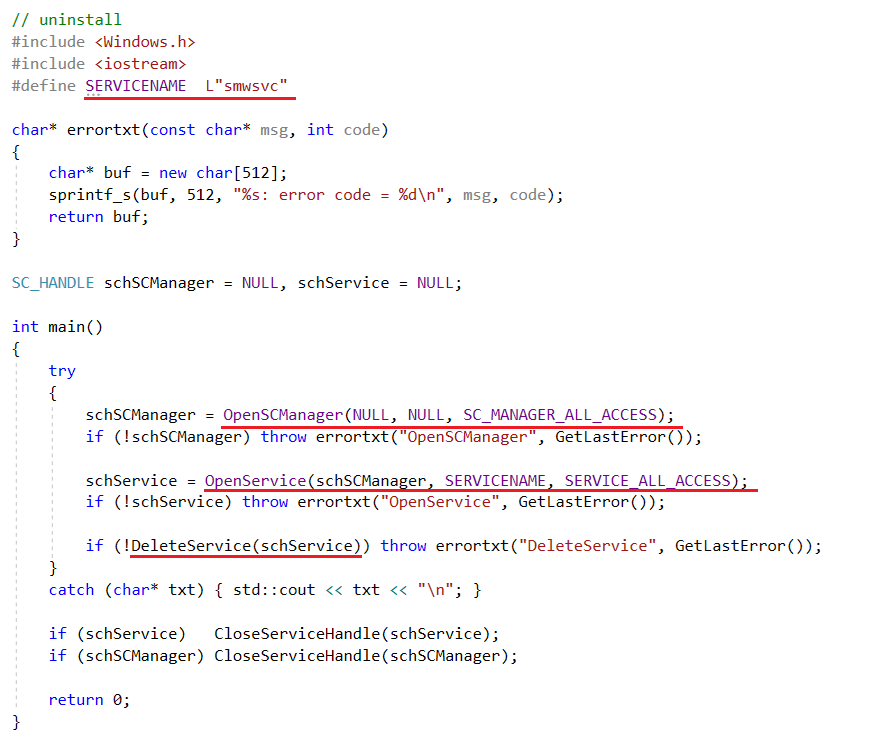


## Порядок разработки

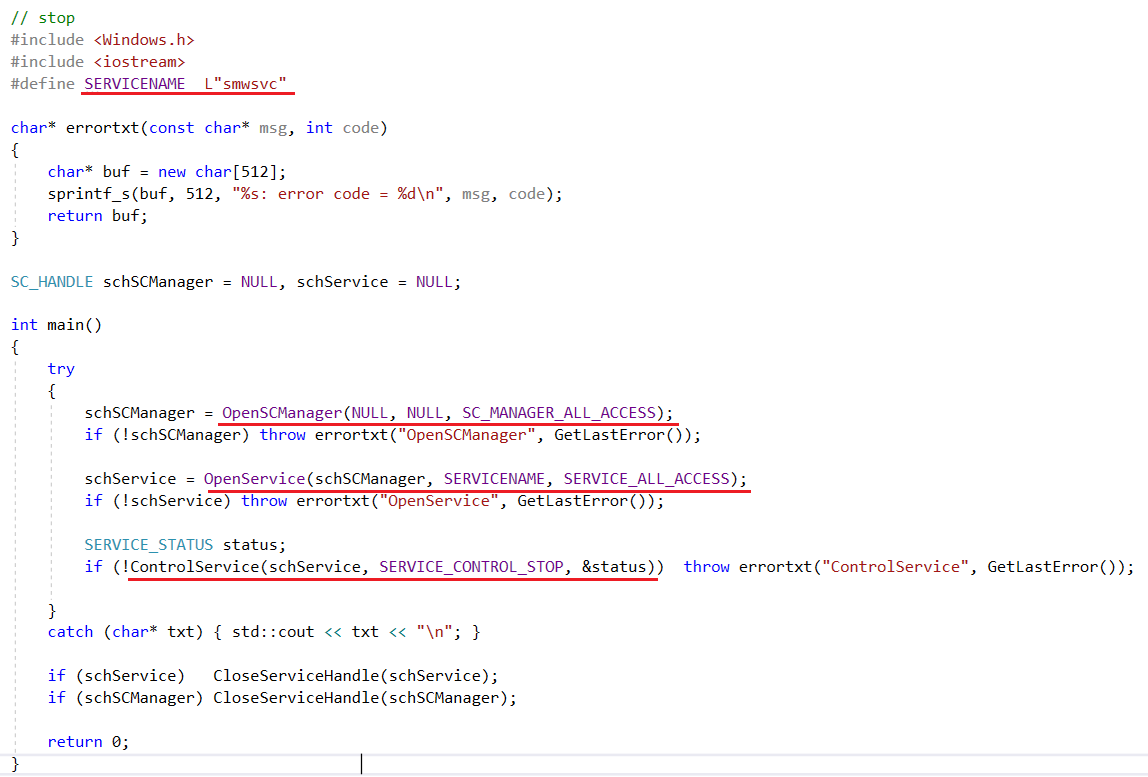
**Windows Services:** создание сервиса, регистрация в реестре Windows.



**Windows Services:** удаление сервиса.



**Windows Services:** остановка сервиса.



**Windows Services:** старт сервиса.

