**Экзаменационные вопросы**

**ПОИТ-3, зима 2023**

**Системное программирование**

1. Системное программирование: определение, назначение, применение.

Когда говорят о системном программировании, в большинстве случаев подразумевают разработку программ, имеющих один из трех признаков. Первый признак**: пользователем разработанного программного обеспечения является программист**. Иными словами, системный программист разрабатывает программное обеспечение для других программистов, часто называемых «проблемными». Второй признак: **разрабатываемое программное обеспечение является повторно используемым и, как правило, оформляется в виде библиотек функций и применяется в нескольких прикладных приложениях.** Можно сказать, что системный программист разрабатывает общее, универсальное программное обеспечение. Третий признак: **системное программное обеспечение напрямую использует системные вызовы операционной системы.**

Системные вызовы (в англоязычной литературе встречается сокращение syscall) — это вызовы функций, совершаемые из пользовательского пространства. Они направлены из приложений (например, текстового редактора или вашей любимой игры) к ядру. Смысл системного вызова — запросить у операционной системы определенную службу или ресурс. Когда программа работает в пользовательском режиме, у нее ограниченный доступ к системным ресурсам. Однако, иногда приложениям необходимо выполнять действия, требующие более высоких привилегий, таких как открытие файла, управление памятью или создание процессов или потоков.

**Системные вызовы (system call)** представляют собой интерфейс между пользовательским приложением и ядром операционной системы. Ядро операционной системы предоставляет набор функций, которые программы могут вызывать для выполнения определенных операций, таких как ввод-вывод, создание процессов, управление файлами и т. д.

**Неминуемо приходится применять системные вызовы. Поэтому, как правило, системное программирование подразумевает разработку библиотек универсальных функций, использующих системные вызовы операционной системы.**

[**Применение системного программирования — разработка операционных систем, драйверов устройств, компиляторов, интерпретаторов, отладчиков, эмуляторов, виртуальных машин, сетевых протоколов, систем безопасности, систем управления базами данных и других программ, которые работают на низком уровне абстракции и взаимодействуют с аппаратурой**](https://habr.com/ru/hubs/system_programming/)

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

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

**Другие определения:**

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

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

**Применение: Фреймворки операционной системы используются в различных областях, включая разработку мобильных и настольных приложений.**[**Они могут включать инструменты для работы с графикой, базами данных, сетевыми соединениями и другими функциями, которые обычно требуются приложениям**](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA)**.**

[**Состав: Фреймворк операционной системы может включать в себя API** ***Разработчиками операционной системы Windows была создана библиотека функций, при помощи которых и происходит взаимодействие приложения с операционной системой, так называемые функции Программного интерфейса приложений (Application Program Interface, API)*, библиотеки для работы с файлами и сетями, графические и пользовательские интерфейсы, системы управления памятью и другие компоненты**](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA)**.**

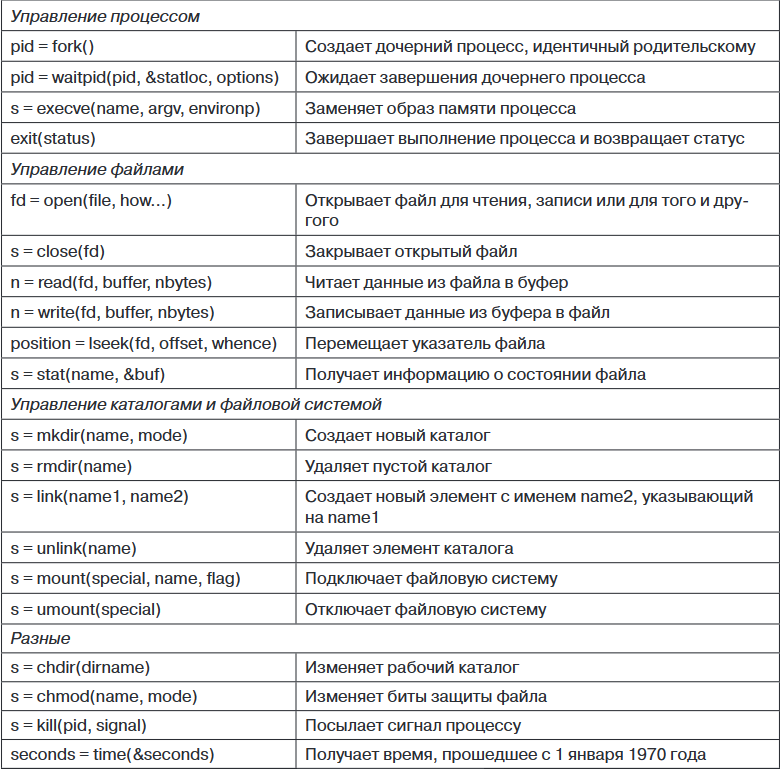
[***Примером фреймворка операционной системы может служить .NET Framework от Microsoft***](https://learn.microsoft.com/ru-ru/dotnet/framework/install/on-windows-10)***.***[***Этот фреймворк предоставляет набор библиотек и инструментов для разработки приложений под операционную систему Windows***](https://learn.microsoft.com/ru-ru/dotnet/framework/install/on-wind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

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

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

**Чтобы появилась возможность писать программы, работающие в любой UNIX-системе, Институт инженеров по электротехнике и электронике (IEEE) разработал стандарт системы UNIX, названный POSIX, который в настоящее время поддерживается большинством версий UNIX. Стандарт POSIX определяет минимальный интерфейс системных вызовов, который должны поддерживать совместимые с ним системы UNIX. Фактически на данный момент POSIX-интерфейс поддерживается также рядом других операционных систем.**

**В стандарте POSIX определено более 100 процедур, обеспечивающих обращение к системным вызовам. Некоторые наиболее важные из них и сгруппированные для удобства по категориям перечислены в табл*(ну тут линукс, но можно перечислить категории сущ.команд)***

****

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

**Основным понятием в любой операционной системе является процесс: абстракция, описывающая выполняющуюся программу.**

***При этом хотя в каждый конкретный момент времени центральный процессор работает только с одним процессом, в течение 1 секунды он может успеть поработать с несколькими из них, создавая иллюзию параллельной работы.***

**Процесс это просто** **экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных. Концептуально у каждого процесса есть свой, виртуальный, центральный процессор. Разумеется, на самом деле настоящий центральный процессор постоянно переключается между процессами, но чтобы понять систему, куда проще думать о наборе процессов, запу-**

**щенных в (псевдо) параллельном режиме, чем пытаться отслеживать, как центральный процессор переключается между программами. Это постоянное переключение между процессами называется мультипрограммированием, или многозадачным режимом работы.**

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

|  |
| --- |
| BOOL CreateProcess(  LPCTSTR lpApplicationName, // имя исполняемого модуля  LPTSTR lpCommandLine, // командная строка  LPSECURITY\_ATTRIBUTES lpProcessAttributes, // защита процесса  LPSECURITY\_ATTRIBUTES lpThreadAttributes, // защита потока  BOOL bInheritHandles, // признак наследования дескриптора  DWORD dwCreationFlags, // флаги создания процесса  LPVOID lpEnvironment, // блок новой среды окружения  LPCTSTR lpCurrentDirectory, // текущий каталог  LPST ARTUPINFO lpStartUpInfo, // вид главного окна  LPPROCESS\_INFORMATION lpProcessInformation // информация о процессе |

***Разница между процессом и программой довольно тонкая, но весьма существенная. Здесь нам, наверное, поможет какая-нибудь аналогия. Представим себе программиста, решившего заняться кулинарией и испечь пирог на день рождения дочери. У него есть рецепт пирога, а на кухне есть все ингредиенты: мука, яйца, сахар, ванильный экстракт и т. д. В данной аналогии рецепт — это программа (то есть алгоритм, выраженный в некой удобной форме записи), программист — это центральный процессор, а ингредиенты пирога — это входные данные. Процесс — это действия, состоящие из чтения рецепта нашим кулинаром, выбора ингредиентов и выпечки пирога. Теперь представим, что на кухню вбегает сын программиста и кричит, что его ужалила пчела. Программист записывает, на каком месте рецепта он остановился (сохраняется состояние текущего процесса), достает книгу советов по оказанию первой помощи и приступает к выполнению изложенных в ней инструкций. Перед нами процессор, переключенный с одного процесса (выпечки) на другой процесс, имеющий более высокую степень приоритета (оказание медицинской помощи), и у каждого из процессов есть своя программа (рецепт против справочника по оказанию первой помощи). После извлечения пчелиного жала программист возвращается к пирогу, продолжая выполнять действия с того места, на котором остановился. Ключевая идея здесь в том, что процесс — это своего рода действия. У него есть программа, входные и выходные данные и состояние. Один процессор может совместно использоваться несколькими процессами в соответствии с неким алгоритмом планирования, который используется для определения того, когда остановить один процесс и обслужить другой. В отличие от процесса программа может быть сохранена на диске и вообще ничего не делать.***

**С технической точки зрения во всех этих случаях новый процесс создается за счет уже существующего процесса, который выполняет системный вызов, предназначенный для создания процесса. Этим процессом может быть работающий пользовательский процесс, системный процесс, вызванный событиями клавиатуры или мыши, или процесс управления пакетными заданиями. Данный процесс осуществляет системный вызов для создания нового процесса. Этот системный вызов предписывает операционной системе создать новый процесс и прямо или косвенно указывает, какую программу в нем запустить. В Windows все происходит иначе: одним вызовом функции Win32 CreateProcess создается процесс, и в него загружается нужная программа.**

**Три состояния, в которых может находиться процесс:**

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

**Логически первые два состояния похожи друг на друга. В обоих случаях процесс желает выполняться, но во втором состоянии временно отсутствует доступный для этого процессор. Третье состояние коренным образом отличается от первых двух тем, что процесс не может выполняться, даже если процессору кроме него больше нечем заняться.**

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

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

*Приоритет процесса можно изменить при помощи функции SetPriorityClass*

*Узнать приоритет процесса можно посредством вызова функции GetPriorityClass*

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

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

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

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

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

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

* **Буфер обмена** — центральное хранилище для обмена данными между приложениями, которые поддерживают операции вырезания, копирования и вставки.
* **Сопоставление файлов** — механизм, который позволяет нескольким процессам совместно использовать один и тот же файл или часть файла, отображая его в свои адресные пространства.
* **Mailslots** — механизм, который позволяет отправлять и получать сообщения между процессами, работающими на одном или разных компьютерах в сети. Mailslots поддерживают только однонаправленную связь и не гарантируют доставку сообщений.
* **Сокеты Windows** — механизм, который позволяет устанавливать двустороннюю связь между процессами, работающими на одном или разных компьютерах в сети. Сокеты Windows поддерживают различные протоколы транспортного уровня, такие как TCP и UDP.
* **Именованные каналы** — механизм, который позволяет устанавливать двустороннюю или одностороннюю связь между процессами, работающими на одном или разных компьютерах в сети. Именованные каналы поддерживают синхронный и асинхронный режимы работы, а также механизмы безопасности и сжатия данных.
* **Очереди сообщений** — механизм, который позволяет отправлять и получать сообщения между процессами, работающими на одном или разных компьютерах в сети. Очереди сообщений поддерживают гарантированную доставку сообщений, приоритеты, транзакции и уведомления.

**Синхронизация — это способ обеспечения порядка и согласованности действий между процессами. Синхронизация может быть пассивной или активной, в зависимости от того, инициируют ли процессы синхронизацию самостоятельно или используют внешний сигнал. Синхронизация также может быть взаимной или невзаимной, в зависимости от того, требуется ли согласие всех процессов для синхронизации или нет. Наконец, синхронизация может быть состоянием или событием, в зависимости от того, основана ли синхронизация на проверке некоторого условия или на ожидании некоторого сигнала.**

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

* **Семафоры** — механизм, который позволяет ограничивать количество процессов или потоков, которые могут одновременно получить доступ к определенному ресурсу или выполнять определенную задачу.
* **Мьютексы** — механизм, который позволяет обеспечивать взаимное исключение между процессами или потоками, которые конкурируют за доступ к одному ресурсу или выполняют одну задачу.
* **Критические секции** — механизм, который позволяет обеспечивать взаимное исключение между потоками одного процесса, которые конкурируют за доступ к одному ресурсу или выполняют одну задачу.
* **События** — механизм, который позволяет сигнализировать о наступлении определенных условий между процессами или потоками, а также ожидать этих условий.
* **Таймеры** — механизм, который позволяет задавать отложенные или периодические действия между процессами или потоками, а также ожидать их выполнения.

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

***Многие текстовые процессоры обладают свойством автоматического сохранения всего файла на диск каждые несколько минут, чтобы уберечь пользователя от утраты его дневной работы в случае программных или системных сбоев или отключения электропитания. Третий поток может заниматься созданием резервных копий на диске, не мешая первым двум. Если бы программа была рассчитана на работу только одного потока, то с начала создания резервной копии на диске и до его завершения игнорировались бы команды с клавиатуры или мыши. Пользователь ощущал бы это как слабую производительность. Можно было бы сделать так, чтобы события клавиатуры или мыши прерывали создание резервной копии на диске, позволяя достичь более высокой производительности, но это привело бы к сложной модели программирования, основанной на применении прерываний. Программная модель, использующая три потока, гораздо проще. Первый поток занят только взаимодействием с пользователем. Второй поток по необходимости занимается переформатированием документа. А третий поток периодически сбрасывает содержимое ОЗУ на диск. Вполне очевидно, что три отдельных процесса так работать не будут, поскольку с документом необходимо работать всем трем потокам. Три потока вместо трех процессов используют общую память, таким образом, все они имеют доступ к редактируемому документу. При использовании трех процессов такое было бы невозможно.***

**Эти задачи не имеют своих собственных ресурсов — они развиваются в том же виртуальном адресном пространстве, могут пользоваться теми же файлами, виртуальными устройствами и иными ресурсами, что и данный процесс. Единственное, что им необходимо иметь в самостоятельном пользовании — это процессорный ресурс. Каждый процесс всегда состоит по крайней мере из одного потока, и только если имеется внутренний параллелизм, программист может разделить один поток на несколько параллельных. Потоки выполняются строго последовательно и имеют свой собственный программный счетчик и стек. Потоки, как и процессы, могут порождать потоки-потомки.**

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

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

**Возможные приоритеты у потоков:**

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

**Для создания потоков, выполняющихся в адресном пространстве вызывающего процесса, предусмотрен системный вызов CreateThread. Любой поток процесса может сам завершить свое выполнение, вызвав функцию ExitThread. Выполнение потока также может быть завершено другим потоком с помощью функции TerminateThread, однако освобождения ресурсов потока при этом не происходит, обработчики завершения не выполняются и уведомления библиотекам DLL не посылаются.**

*Функции, используемые для получения идентификаторов (ID) и дескрипторов потоков, напоминают те, которые используются для аналогичных целей в случае процессов:*

* *GetCurrentThread — возвращает ненаследуемый псевдодескриптор вызывающего потока.*
* *GetCurrentThreadId — позволяет получить идентификатор потока, а не его дескриптор.*
* *OpenThread — создает дескриптор потока по известному идентификатору*

*Для изменения приоритета потока используется функция SetThreadPriority*

*Узнать уровень приоритета потока можно посредством вызова функции GetThreadPriority*

***синхронизация ниже***

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

**До сих пор нами были обсуждены только один механизм, обеспечивающий синхронизацию процессов и потоков друг с другом:**

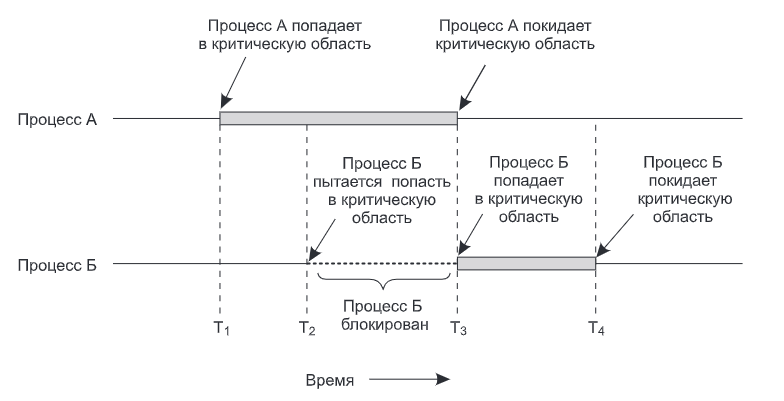
* **Поток, выполняющийся в контексте одного процесса, может дожидаться завершения другого процесса с использованием функции ExitProcess путем применения к дескриптору процесса функций ожидания WaitForSingleObject или WaitForMultipleObject .**

**Функция WaitForSingleObject позволяет приостановить выполнение потока до тех пор, пока не поступит сигнал от заданного объекта. Кроме того, в этой команде указывается максимальное время ожидания. Чтобы обеспечить бесконечное ожидание, в качестве временного интервала следует задать значение INFINITE. Если объект уже доступен или если он подает сигнал в течение заданного времени, функция WaitForSingleObject возвращает значение 0 и выполнение потока возобновляется. Но если заданный интервал времени прошел, а объект не подал сигнала, функция возвращает значение WAIT\_TIMEOUT.**

**Windows предоставляет четыре других объекта, предназначенных для синхронизации потоков и процессов. Три из них — мьютексы, семафоры и события — являются объектами ядра, имеющими дескрипторы. События используются также для других целей, например, для асинхронного ввода/вывода. Мы начнем обсуждение с четвертого объекта, а именно, объекта критического участка кода CRITICAL\_SECTION.**

**Та часть программы, в которой используется доступ к общей памяти, называется критической областью или критической секцией.**

**Для инициализации и удаления переменной типа CRITICAL\_SECTION используются, соответственно, функции InitializeCriticalSection и DeleteCriticalSection. Функция EnterCriticalSection блокирует поток, если на данном критическом участке кода присутствует другой поток. Ожидающий поток разблокируется после того, как другой поток выполнит функцию LeaveCriticalSection. Говорят, что поток получил права владения объектом CS, если произошел возврат из функции EnterCriticalSection, тогда как для уступки прав владения используется функция LeaveCriticalSection**



**Мьютекс — это совместно используемая переменная, которая может находиться в одном из двух состояний: заблокированном или незаблокированном. Следовательно, для их представления нужен только один бит, но на практике зачастую используется целое число, при этом нуль означает незаблокированное, а все остальные значения—заблокированное состояние. Объект взаимного исключения (mutual exception), или мьютекс (mutex), обеспечивает более универсальную функциональность по сравнению с объектом CRITICAL\_SECTION. Поскольку мьютексы могут иметь имена и дескрипторы, их можно использовать также для синхронизации потоков, принадлежащих различным процессам.**

**При работе с мьютексами мы будем пользоваться функциями CreateMutex, ReleaseMutex и OpenMutex. Функция OpenMutex открывает существующий именованный мьютекс. Эта функция дает возможность потокам, принадлежащим различным процессам, синхронизироваться так, как если бы они принадлежали одному и тому же процессу. Вызову функции OpenMutex в одном процессе должен предшествовать вызов функции CreateMutex в другом процессе. Функция ReleaseMutex освобождает мьютекс, которым владеет вызывающий поток. Если мьютекс не принадлежит потоку, функция завершается с ошибкой.**

**Семафор**

**Объекты второго из трех упомянутых в начале главы типов объектов синхронизации ядра — семафоры (semaphores), поддерживают счетчики, и когда значение этого счетчика больше 0, объект семафора находится в сигнальном состоянии. Если же значение счетчика становится нулевым, объект семафора переходит в несигнальное состояние. Потоки и процессы организуют ожидание обычным способом, используя для этого одну или несколько функций ожидания. При разблокировании ожидающего потока значение счетчика уменьшается на 1.**

**К функциям управления семафорами относятся CreateSemaphore, OpenSemaphore и ReleaseSemaphore, причем последняя функция может инкрементировать значение счетчика на 1 и более. Эти функции аналогичны своим эквивалентам, предназначенным для управления мьютексами. Семафор может быть освобожден любым потоком, а не только тем, который ожидает**

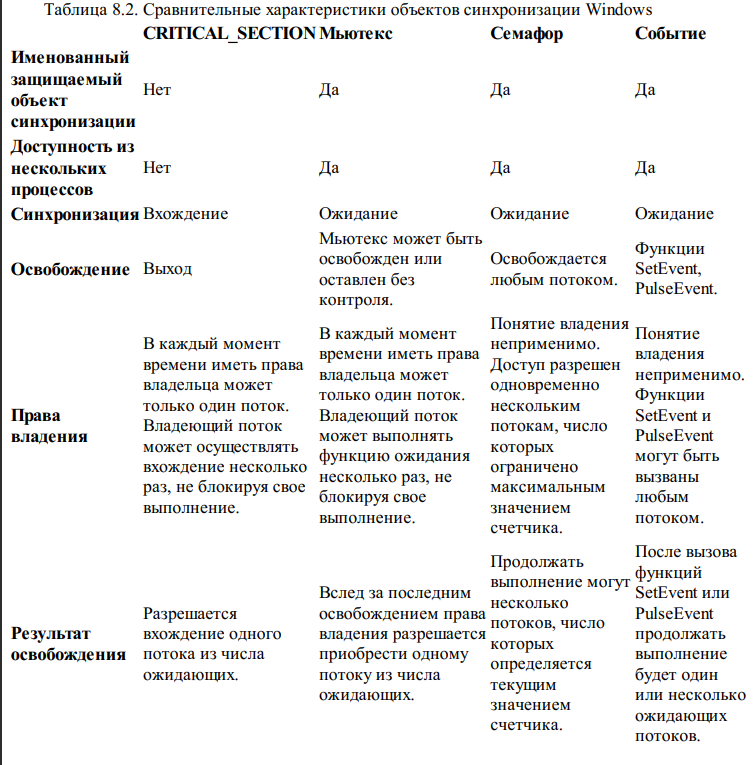
**Последним из рассматриваемых нами типов объектов синхронизации ядра являются события (events). Объекты события используются для того, чтобы сигнализировать другим потокам о наступлении какого-либо события, например, о появлении нового сообщения. Важной дополнительной возможностью, обеспечиваемой объектами событий, является то, что переход в сигнальное состояние единственного объекта события способен вывести из состояния ожидания одновременно несколько потоков. Объекты события делятся на сбрасываемые вручную и автоматически сбрасываемые, и это их свойство устанавливается при вызове функции CreateEvent.**

**• Сбрасываемые вручную события (manual-reset events) могут сигнализировать одновременно всем потокам, ожидающим наступления этого события, и переводятся в несигнальное состояние программно.**

**• Автоматически сбрасываемые события (auto-reset event) сбрасываются самостоятельно после освобождения одного из ожидающих потоков, тогда как другие ожидающие потоки продолжают ожидать перехода события в сигнальное состояние.**

**События используют пять новых функций: CreateEvent, OpenEvent, SetEvent, ResetEvent и PulseEvent.** Для открытия именованного объекта события используется функция **OpenEvent,** причем это может сделать и другой процесс. **Поток может установить событие в сигнальное состояние, используя функцию SetEvent. Если событие является автоматически сбрасываемым, то оно автоматически возвращается в несигнальное состояние уже после освобождения только одного из ожидающих потоков. В отсутствие потоков, ожидающих наступления этого события, оно остается в сигнальном состоянии до тех пор, пока такой поток не появится, после чего этот поток сразу же освобождается.**

**С другой стороны, если событие является сбрасываемым вручную, то оно остается в сигнальном состоянии до тех пор, пока какой-либо поток не вызовет функцию ResetEvent, указав дескриптор этого события в качестве аргумента. В это время все ожидающие потоки освобождаются, но до выполнения такого сброса события другие потоки могут как переходить в состояние его ожидания, так и освобождаться. Функция PulseEvent освобождает все потоки, ожидающие наступления сбрасываемого вручную события, но после этого событие сразу же автоматически сбрасывается. В случае же использования автоматически сбрасываемого события функция PulseEvent освобождает только один ожидающий поток, если таковые имеются.**



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

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

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

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

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

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

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

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

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

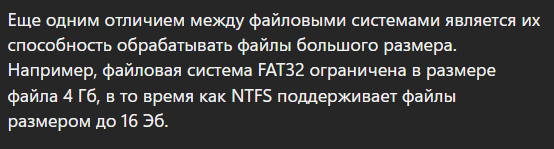
**Windows поддерживает на непосредственно подключенных устройствах файловые системы нескольких типов(тут не все, но те что ему надо):**

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

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

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

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





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

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

**Файл: набор логических записей.**

**Логическая запись: последовательность байт.**

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

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

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

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

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

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

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

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

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

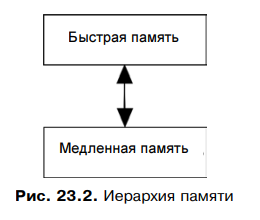
**Буфером ввода-вывода называется область оперативной памяти, предназначенная для временного хранения записей файла. Обычно длина буфера выбирается кратной длине кластера. Буферы ввода-вывода предназначены для решения двух задач:**

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

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

**КЭШ**

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

****

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

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

***Интегральные схемы, предназначенные для хранения программ и данных, называются физической памятью. Обычно под физической памятью мы понимаем память, к которой процессор может обращаться, используя адресную шину и шину данных, а внутренняя память самого процессора представляется регистрами. Каждый байт физической памяти имеет свой номер или индекс, который называется физическим адресом. При обращении к физической памяти процессор должен выставить на адресную шину физический адрес памяти, к которой он хочет получить доступ.***

***Под логической памятью процесса понимается массив байтов, к которым может обратиться процесс. Индекс каждого элемента этого массива называется логическим адресом. Так как логическая память процесса представляется линейным массивом байтов, то логический адрес процесса обычно называют линейным адресом. Так как в действительности процесс может работать только с данными в физической памяти, то во время работы процесса необходимо отображать логическую память процесса в физическую память компьютера. Обычно, прямое отображение невозможно по той простой причине, что объем логической памяти процесса превышает объем физической памяти компьютера. Для решения этой задачи физическую память компьютера дополняют памятью на дисках.***

**Полученную расширенную память называют виртуальной памятью, а адрес элемента этой памяти называют виртуальным адресом.** **С точки зрения процесса, страницы его виртуальной памяти могут находиться в одном из трех состояний:**

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

**Поясним эти состояния более подробно. Первоначально, при запуске процесса, все страницы виртуальной памяти считаются свободными, естественно кроме тех, в которые загружена сама программа. Чтобы распределить для использования свободные или зарезервированные страницы виртуальной памяти, процесс должен вызвать функцию VirtualAlloc. Только после успешного завершения этой функции процесс может использовать распределенную ему виртуальную память. Третье состояние характеризует виртуальные страницы как зарезервированные. Это значит, что эти виртуальные страницы зарезервированы процессом для дальнейшего использования и не будут выделяться системой для использования процессу без точного указания процессом их адреса. Следует отметить, что при резервировании виртуальных страниц реальная память под эти страницы не выделяется.**

**После завершения работы с виртуальной памятью ее необходимо освободить, используя функцию VirtualFree. Если некоторая область виртуальной памяти будет часто использоваться процессом, то можно запретить системе выгружать эти виртуальные страницы из реальной памяти, иначе говоря, запереть или блокировать эти виртуальные страницы в реальной памяти. Для этого нужно использовать функцию VirtualLock. Для отмены блокировки виртуальных страниц в реальной памяти используется функция VirtualUnlock**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Теперь рассмотрим, как оформляются функции и переменные, которые DLL предоставляет в использование своим клиентам. Во-первых, заметим, что такие функции и переменные называются экспортируемыми. Для того чтобы сделать функцию или переменную экспортируемой, нужно определить их с модификатором extern "C" и квалификатором \_\_declspec(dllexport). Кроме того, экспортируемая переменная должна быть инициализирована. Модификатор extern "C" указывает компилятору на то, что функция или переменная должна иметь имя в стиле языка программирования С. То есть имя функции или переменной не будет искажаться путем добавления к нему обозначений типов данных из сигнатуры функции или определения переменной. Модификатор \_\_declspec(dllexport) указывает компилятору на то, что данная функция или переменная будет экспортироваться из DLL. Для определения экспортируемых из DLL функций и переменных можно использовать файл определения модуля, который должен иметь расширение def. Такие файлы включают инструкции, которые используются компоновщиком для создания исполняемого модуля или DLL. Однако эти файлы не обязательны для компоновки, т. к. большинство их параметров может быть задано при помощи определения опций компоновщика. При создании DLL def-файл содержит инструкции, которые определяют имена экспортируемых функций и переменных. Отметим, что если для описания экспорта из DLL используется файл определений, то квалификатор \_\_declspec(dllexport) перед именами функций и переменных в самой DLL использовать не нужно.**

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

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

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

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

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

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

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

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

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

**Отметим, что импортируемые из DLL имена должны иметь модификатор extern и квалификатор \_\_declspec(dllimport). Использование квалификатора \_\_declspec(dllimport) с прототипами функций не обязательно, но его нужно обязательно использовать при импорте имен переменных. Однако использование этого квалификатора с именами функций делает описание функций более понятным и ускоряет работу компоновщика.**

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

**Component Object Model (COM) – спецификация, позволяющая разрабатывать программное обеспечения. Модель программного обеспечения. Разработана Microsoft.**

**COM(ComponentObject Model)- модель программного обеспечения или набор методов или Технологический стандарт от компании Microsoft, предназначенный для создания программного обеспечения на основе взаимодействующих компонентов объекта, каждый из которых может использоваться во многих программах одновременно**

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

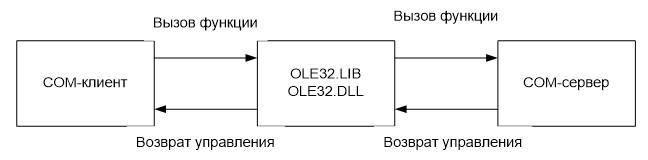
**COM-объект – специализированный объект времени исполнения (экземпляр). Для идентификации компонента используется идентификатор CLSID.**

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

**Раннее связывание (early binding): В случае раннего связывания, связывание данных с кодом происходит на этапе компиляции программы, то есть на этапе, когда программа преобразуется из исходного кода в машинный код.**

**Позднее связывание (late binding): В случае позднего связывания, связывание данных с кодом происходит в процессе выполнения программы, а не на этапе компиляции.**

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

****

**COM-сервер может иметь тип:**

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

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

**методы интерфейса IUnknown**

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

**СОМ-объекты представляют собой экземпляры CoClass. Напомним, что CoClass - это класс, поддерживающий один или более интерфейс. СОМ-объекты могут предоставлять только те услуги, которые определены в интерфейсах CoClass. Экземпляры CoClass создаются при помощи специального типа объекта, называемого фабрикой класса.**

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

**методы интерфейса IClassFactory**

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

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

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

**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). После передачи CLSID, СОМ-сервер должен обеспечить так называемую фабрику класса, которая создает экземпляры СОМ объектов.**

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

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

**Экспортируемые стандартные функции:**

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

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

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

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

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

**Регистрация COM/DLL-сервера:**

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

**Формат командной строки REGSVR32:**

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

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

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

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

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

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

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

**Политикой безопасности называется набор требований, выполнение которых обеспечивает безопасную работу информационной системы.**

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

* **для каждого объекта определяется набор операций, которые можно выполнять над этим объектом;**
* **субъект может выполнить операцию над объектом при условии, если он имеет право на выполнение этой операции;**
* **субъект, который имеет права на выполнение некоторых операций над объектом, может передать эти права другому субъекту.**

**Суть дискреционной политики безопасности заключается в том, что права доступа субъекта к объекту определяются другим субъектом, который имеет эти права. Другими словами, доступ субъектов к объектам разрешается другими субъектами или можно сказать, что оставлен на усмотрение другим субъектам, которые обладают этими правами.**

**Так как субъекты могут передавать права управления объектами другим субъектам, то должны быть определены правила, которым подчиняются субъекты при передаче таких прав. Набор таких правил определяет модель управления в дискреционной модели безопасности. Существуют четыре модели управления, которые могут использоваться в дискреционной модели безопасности:**

* **иерархическое управление (hierarchical control);**
* **управление правами доступа владельцем объекта (concept of ownership).**
* **либеральное управление (laissez-fair);**
* **централизованное управление (centralized control).**

**Кратко опишем каждую из этих моделей управления. В иерархической модели управления все субъекты, которые могут управлять правами доступа к объектам, упорядочиваются (иерархически), организуя при этом древовидную структуру. В вершине этой структуры находится администратор системы безопасности, который может управлять правами доступа ко всем объектам и наделять такими правами любого субъекта. Для остальных субъектов предполагается, что они могут управлять правами доступа только субъектов, которые находятся ниже их в иерархии. В модели управления, использующей понятие владельца объекта, управление правами доступа к объекту для всех субъектов выполняется именно владельцем этого объекта. Владельцем объекта, как правило, считается тот субъект, который создает этот объект. В такой модели управления также присутствует администратор системы безопасности, который может ограничить права владельца объекта на передачу прав доступа к объекту другим субъектам. Фактически в этом случае модель управления представляет собой двухуровневую, иерархическую модель управления. В либеральной модели управления передача прав доступа к объекту от субъекта, который обладает этими правами, к другому субъекту никак не контролируется. Как уже говорилось, такая модель управления является самой ненадежной. В централизованной модели управления правами управления доступом к объектам обладает только один субъект, как правило, это администратор системы. Таким образом, все вопросы по разрешению доступа субъектов к объектам системы разрешаются только администратором. Централизованная модель управления является наиболее надежной из рассмотренных моделей управления доступами к объектам.**

**Первый подход: Для каждого пользователя информационной системы администратором информационной системы задаются его возможности. При этом менеджер системы безопасности работает так, что каждому субъекту разрешается доступ к ресурсам только в соответствии с его возможностями.**

**Возможности каждого пользователя хранят в виде списка, каждый элемент которого содержит имена объектов и режимы доступа к этим объектам. Такой список называется профилем (profile) пользователя. Тогда субъект может открыть заданный режим доступа к объекту только в том случае, если этот объект и заданный режим присутствует в его профиле**

**При втором подходе матрицу управления доступами рассматривают по столбцам. Так как каждый столбец матрицы соответствует одному объекту, то можно сказать, что весь столбец матрицы доступов описывает всех субъектов, которые имеют доступ к этому объекту, и режимы доступа каждого субъекта. В этом случае при реализации столбец матрицы управления доступами хранится в виде списка, который называется список управления доступами (access control list). На практике такие списки обычно сокращенно называют ACL, а элементы этих списков — ACE (access control elements). Каждый элемент списка управления доступами содержит имя пользователя и разрешенные этому пользователю режимы доступа к объекту. Список управления доступами объекта обычно создается владельцем объекта или администратором информационной системы. При этом менеджер системы безопасности открывает субъекту требуемый доступ к объекту только в том случае, если его имя и требуемый режим доступа находятся в списке управления доступами для этого объекта.**

**Очевидно, что при такой реализации матрицы управления доступами легко решаются вопросы корректировки доступа к объектам и удаления объектов, однако возникают сложности с корректировкой информации о пользователях системы. Так, например, если пользователю запрещается доступ к ресурсам информационной системы, то его имя и режимы доступа нужно удалить из всех списков управления доступами для всех объектов, к которым этот пользователь имел доступ. Для упрощения решения этой задачи всех пользователей информационной системы разбивают на группы, предполагая, что каждая из этих групп может работать только с определенными объектами в заданных режимах доступа. Тогда в списки управления доступами вносят имена групп пользователей с соответствующими режимами доступа. Это значительно сокращает объем списков и упрощает их корректировку.**

**Прежде чем пользователь сможет работать в среде операционных систем Windows NT/2000/XP, он должен быть зарегистрирован администратором системы. При регистрации пользователя администратором создается учетная запись пользователя (user account), которая хранится в базе данных менеджера учетных записей (Security Account Manager, SAM). Эта база данных является частью реестра Windows и доступ к ней имеет только администратор системы. Регистрация пользователя выполняется на уровне домена локальной сети, что упрощает управление доступом к ресурсам этой сети. Вообще доменом называется группа компьютеров в локальной сети, которые разделяют общую базу данных учетных записей пользователей.**

**Существуют четыре типа учетных записей, а именно:**

* **учетная запись пользователя;**
* **учетная запись группы пользователей;**
* **учетная запись компьютера;**
* **учетная запись домена.**

**Учетная запись компьютера используется для регистрации компьютеров в домене и создается при подключении компьютера к домену. Также как пользователь должен войти в систему на компьютере, так и компьютер должен при включении войти в домен. Учетная запись домена используется для регистрации домена в других доменах. Учетные записи пользователя, компьютера и домена имеют пароли. Доступ к паролю пользователя имеют администратор и сам пользователь. Паролем домена управляет администратор, а паролем компьютера управляет контроллер доменов (Primary Domain Controller, PDC).**

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

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

***Учетная запись администратора предназначена для управления локальной системой, установленной на одном компьютере. По умолчанию эта учетная запись создается без пароля. Желательно, чтобы в дальнейшем администратор системы изменил имя пользователя и установил пароль для этой учетной записи. Учетную запись администратора системы нельзя заблокировать. Администратор системы может создавать новые учетные записи. Учетная запись гостя предназначена для обычного пользователя системы, который не имеет административных полномочий. Эта учетная запись также создается без пароля. Она не может быть удалена, но может быть переименована. Учетная запись системы используется самой операционной системой для выполнения различных задач, которые требуют аутентификации. Эта учетная запись также имеет административные привилегии.***

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

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

**Группа (group) это просто набор учетных записей пользователей, которые объединены по какому-либо признаку, например, пользователи одной группы могут работать в одном отделе. При этом отметим, что одна учетная запись пользователя может входить более чем в одну группу. Каждая группа имеет свою учетную запись и наделена своими правами и полномочиями. Эти права и полномочия передаются каждому члену группы. Однако администратор системы может изъять некоторые права и полномочия у некоторых членов группы. Максимальное количество групп, в которые может входить пользователь, равно 1000**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Для получения информации о пользователе из его учетной записи используется функция NetUserGetInfo. Для того чтобы перечислить учетные записи всех пользователей, зарегистрированных на сервере, используется функция NetUserEnum, Для перечисления глобальных групп, которым принадлежит пользователь, используется функция NetUserGetGroups. Для изменения содержимого учетной записи пользователя используется функция NetUserSetInfo. Для изменения пароля пользователя используется функция NetUserChangePassword, Для удаления учетной записи пользователя из базы данных менеджера учетных записей используется функция NetUserDel**

**Для создания учетной записи локальной группы в базе данных менеджера учетных записей используется функция NetLocalGroupAdd, Для получения информации о локальной группе используется функция NetLocalGroupGetInfo. Отметим, что успешно эти функции могут выполнить только пользователи, которые являются администраторами или операторами учетных записей.**

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

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

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

**Исключением называется событие, которое произошло во время выполнения программы, в результате совершения которого дальнейшее нормальное выполнение программы становится невозможным**

**В операционных системах Windows для этой цели предназначен механизм структурной обработки исключений (structured exception handling, SEH). Смысл механизма структурной обработки исключений заключается в следующем. В программе выделяется блок программного кода, в котором может произойти исключение. Такой блок кода называется фреймом, а сам код называется охраняемым кодом. Затем, после фрейма вставляется программный блок, который обрабатывает происшедшее исключение. Этот блок называется обработчиком исключения. После обработки исключения управление передается первой инструкции, следующей за обработчиком исключения**

**Очевидно, что для того чтобы использовать этот механизм в программе, в язык программирования С++ нужно ввести новые ключевые слова. Такими ключевыми словами являются \_\_try и \_\_except, *которые расширяют список стандартных ключевых слов языка программирования С++ и различаются только компилятором фирмы Microsoft.* Ключевое слово \_\_try отмечает фрейм, а ключевое слово \_\_except отмечает обработчик исключения.**

**Сделаем два важных замечания относительно механизма структурной обработки исключений. Во-первых, не допускается использование оператора goto для передачи управления внутрь фрейма или обработчика исключения. Во-вторых, в выражении фильтра допускается использование функций GetExceptionCode и GetExceptionInformation, которые предоставляют информацию о происшедшем исключении. Параметр выражение\_фильтра в операторе except вычисляется сразу же после того, как возникает исключение. В качестве выражения может выступать литеральная константа, вызов функции фильтра (filter function) или условное выражение. В любом случае выражение должно возвращать одно из следующих трех значений:**

**1. EXCEPTION\_EXECUTE\_HANDLER — система выполняет операторы блока обработки исключений, как показано на рис. 4.1 (см. программу 4.1). Это соответствует обычному случаю.**

**2. EXCEPTION\_CONTINUE\_SEARCH — система игнорирует данный обработчик исключений и пытается найти обработчик исключений в охватывающем блоке, продолжая этот процесс аналогичным образом до тех пор, пока не будет найден обработчик исключений.**

**3.EXCEPTION\_CONTINUE\_EXECUTION — система немедленно возвращает управление в точку, в которой возникло исключение. В случае некоторых исключений дальнейшее выполнение программы невозможно, но если такие попытки делаются, то генерируется повторное исключение**

**Получить код происшедшего исключения можно при помощи функции GetExceptionCode. Отметим, что функция GetExceptionCode может вызываться только в выражении-фильтре или в блоке обработки исключения. Следовательно, эта функция вызывается всегда только в том случае, если исключение произошло. Отсюда можно определить назначение функции GetExceptionCode. Если эта функция вызывается в выражении фильтра, то она используется для того, чтобы определить выполняет ли текущий обработчик исключения обработку исключений с данным кодом или нужно продолжить поиск подходящего обработчика исключения. Если же функция GetExceptionCode вызывается в блоке обработки исключения, то она также предназначена для проверки кодов исключений, которые обрабатывает текущий обработчик исключения, но в этом случае поиск другого обработчика исключений не выполняется.**

**Если для принятия решения об обработке исключения требуется более детально обработать информацию об исключении, то в выражении-фильтре используют функцию, которая в этом случае называется функцией фильтра. В функции фильтра не разрешается вызывать функции GetExceptionCode и GetExceptionInformation, однако эти функции могут вызываться для инициализации параметров функции фильтра. Более подробную информацию об исключении можно получить при помощи вызова функции GetExceptionInformation**

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

**Механизм структурной обработки исключений в Windows позволяет генерировать программные исключения при помощи функции RaiseException**

**Если в программе произошло исключение, для которого не существует обработчика исключений, то в этом случае вызывается функция-фильтр системного обработчика исключений, которая выводит на экран окно сообщений с предложением пользователю закончить программу аварийно или выполнить отладку приложения. Системная функция-фильтр UnhandledExceptionFilter**

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

**В реализации языка программирования С++ фирмой Microsoft, т. е. в Visual C++, предусмотрен механизм, который позволяет использовать механизм структурной обработки исключений в механизме обработки исключений, используемом в С++. Для этой цели была разработана функция \_set\_se\_translator. Эта функция устанавливает в системе функцию, которая называется функцией-транслятором, назначение которой состоит в том, чтобы преобразовывать структурные исключения в исключения языка программирования С++. Если функция-транслятор установлена, то она вызывается всегда при выбросе структурного исключения. В функции-трансляторе можно использовать инструкцию throw языка программирования С++, которая будет выбрасывать исключение С++ нужного типа.**

**В операционных системах Windows существует еще один способ обработки исключений, суть которого заключается в следующем. Код, при исполнении которого возможен выброс исключения, как и в случае с фреймовой обработкой исключений, заключается в блок \_\_try. Но только теперь за блоком \_\_try следует код, который заключается в блок \_\_finally. Система гарантирует, что при любой передаче управления из блока \_\_try, независимо от того, произошло или нет исключение внутри этого блока, предварительно управление будет передано блоку \_\_finally. Такой способ обработки исключений называется финальная обработка исключений. Очевидно, что, как и в случае с фреймовой обработкой исключений, финальная обработка исключений требует поддержки как компилятора, так и операционной системы. Кроме того, в язык программирования С++ добавляется новое ключевое слово \_\_finally. Финальная обработка исключений используется для того, чтобы при любом исходе исполнения блока \_\_try освободить ресурсы, которые были захвачены внутри этого блока. Такими ресурсами могут быть память, файлы, критические секции и т.д. Один try-блок может иметь только один блок finally или только один блок except, но не может иметь оба указанных блока одновременно.**

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

**нормальное завершение блока;**

**выход из блока при помощи управляющей инструкции \_\_leave;**

**выход из блока при помощи одной из управляющих инструкций return, break, continue или goto языка программирования С++;**

**передача управления обработчику исключения.**

**В первых двух случаях считается, что блок \_\_try завершился нормально, а в последних двух случаях — ненормально. Для того чтобы определить, как завершился блок \_\_try, используется функция AbnormalTermination. В случае если блок \_\_try завершился ненормально, эта функция возвращает ненулевое значение, а в противном случае — значение FALSE. Используя функцию AbnormalTermination, ресурсы, захваченные в блоке \_\_try, можно освобождать только в том случае, если блок \_\_try завершился ненормально**

**Как и в случае фреймовой обработки исключений, можно вкладывать блоки \_\_try и \_\_finally в другой блок \_\_try. В этом случае, если внутри самого внутреннего блока \_\_try произошло исключение, то, как и в случае фреймовой обработки исключений, выполняется раскрутка стека. Если самые внутренние блоки \_\_try и \_\_finally вложены в другие блоки \_\_try с финальной обработкой исключений, то при раскрутке стека управление передается всем вложенным блокам \_\_finally в порядке, обратном их вложенности**

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

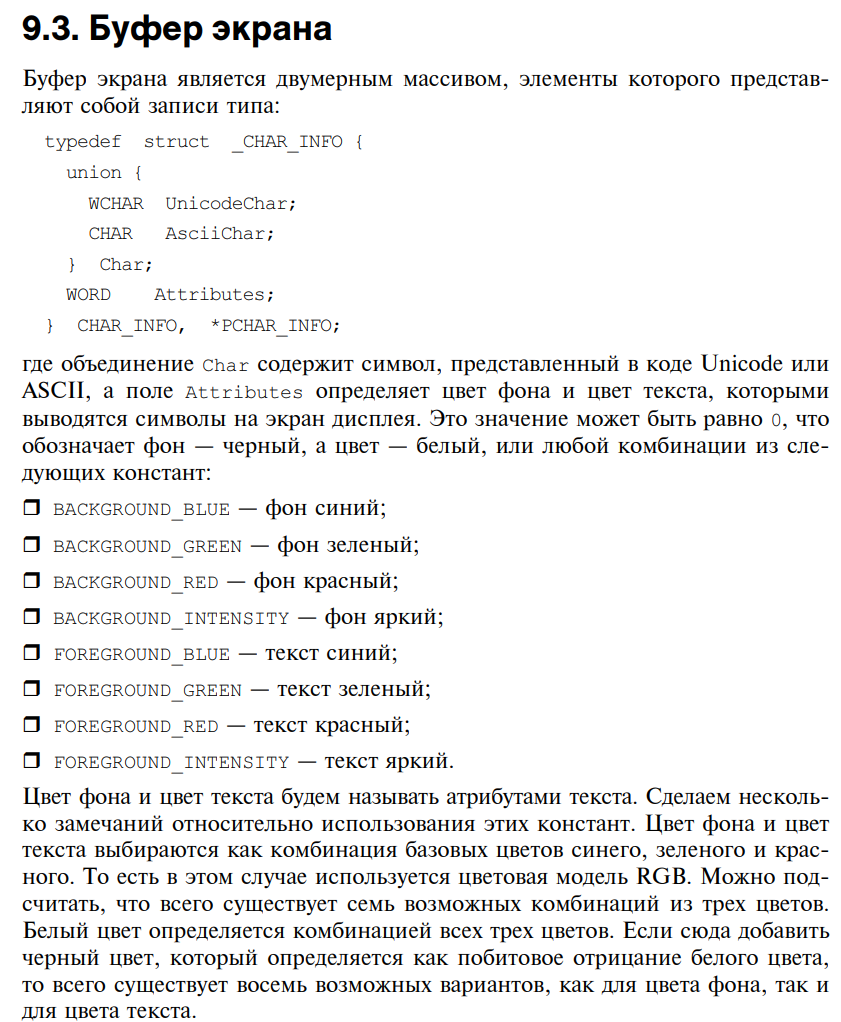
**Консолью называется интерфейс, который используется приложением для ввода-вывода текстовой информации. В этом случае приложение, которое использует консоль для обмена данными с пользователем, называется консольным приложением. Консольные приложения применяются главным образом в системном программировании для разработки различных сервисов и для обработки неустранимых ошибок, возникающих при работе графического приложения. Консоль состоит из одного входного буфера и одного или нескольких буферов экрана. Входной буфер содержит информацию о событиях ввода. Каждое событие ввода описывается записью. Все записи упорядочены в очередь, которая хранится в буфере ввода. Буфер экрана содержит информацию для вывода в окно приложения и является двумерным массивом, который содержит символы и данные о цвете.**

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

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

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

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

****

**Процесс может быть связан только с одной консолью. Новая консоль может создаваться одним из следующих двух способов. Первый способ заключается в том, что при создании консольного процесса командой CreateProcess нужно установить флаг CREATE\_NEW\_CONSOLE. Отметим, что в этом случае, если консольный процесс создается из консольного приложения, а указанный флаг не установлен, то новый процесс присоединяется к консоли родительского процесса. Второй способ заключается в использовании функции AllocConsole. Приложение освобождает консоль посредством вызова функции FreeConsole**

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

**Дескриптор окна консоли можно получить, вызвав функцию GetConsoleWindow. Для чтения заголовка окна консоли используется функция GetConsoleTitle. Максимальный размер окна консоли можно определить, вызвав функцию GetLargestConsoleWindowSize**

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

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

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

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

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

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

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

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

**Чтение и запись последовательности символов:**

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

**Заполнение буфера экрана заданным символом:**

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

**Чтение и запись прямоугольных областей символов:**

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

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

**Сервис это процесс, который выполняет служебные функции. То есть сервис это такая программа, которая запускается при загрузке операционной системы или в процессе ее работы по специальной команде и заканчивает свою работу при завершении работы операционной системы или по специальной команде. Обычно сервисы выполняют определенные служебные функции, необходимые для работы приложений или какого-то конкретного приложения. Примером сервиса может служить фоновый процесс, который обеспечивает доступ к базе данных — такие сервисы также называются серверами. Другой тип сервисов — это программы, обеспечивающие доступ к внешним устройствам, такие сервисы называются драйверами. Как сервис также может быть реализован процесс, отслеживающий работу некоторого приложения, такие сервисы также называются мониторами.**

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

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

**Для того чтобы менеджер сервисов знал о существовании определенного сервиса — его нужно установить. Информация обо всех установленных хранится в реестре операционной системы Windows под ключом HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services. Сервис может запускаться как операционной системой при загрузке, так и программно — из приложения. Если сервис больше не нужен, то его нужно удалить из базы данных операционной системы. Для работы с сервисами в операционных системах Windows предназначены специальные функции из Win32 API.**

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

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

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

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

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

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

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

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

**3. Провести локальную инициализацию сервиса.**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Порт завершения ввода-вывода — это, фактически, объект синхронизации, который оповещает параллельно работающие потоки о завершении асинхронных операций доступа к файлам или именованным каналам. В дальнейшем мы будем говорить только о файлах, подразумевая, что все сказанное справедливо и для именованных каналов.**

**Для работы порта завершения ввода-вывода к нему необходимо подключить файлы, которые должны быть открыты в режиме асинхронного доступа. Причем для каждого файла устанавливается свой номер, который в этом случае называется ключом завершения. При завершении асинхронной операции доступа к файлу операционная система посылает в порт завершения пакет, который содержит информацию о завершившейся асинхронной операции ввода-вывода, содержащую и ключ файла, для которого завершилась эта операция. Порт содержит очередь пакетов, оповещающих о завершении асинхронных операций ввода-вывода. Обслуживается эта очередь по алгоритму FIFO (first in — first out), т. е. вошедший первым — выходит первым.**

**Для создания портов завершения ввода-вывода и подключения к нему файлов в операционных системах Windows используется функция CreateIoCompletionPort. Порт завершения ввода-вывода удаляется, когда закрывается его дескриптор. Поток может узнать о том, что завершилась асинхронная операция ввода/вывода, вызвав функцию GetQueuedCompletionStatus, в которой указывается дескриптор порта. Если очередь пакетов порта завершения не пуста, то поток получает пакет из этой очереди. В противном случае поток блокируется. Каждый порт содержит очередь заблокированных потоков, которая обслуживается по алгоритму LIFO (lst in — first out), т. е. вошедший последним — выходит первым. При этом отметим, что порт завершения может одновременно обслуживать только заданное количество потоков, которое определяется одним из параметров функции CreateIoCompletionPort. Остальные потоки, которые превысили этот предел, блокируются системой, до тех пор, пока количество обслуживаемых потоков не станет меньше заданного предела.**

**Другой особенностью порта завершения является то, что он может получать пакеты не только от системы, по завершении асинхронной операции ввода/вывода, но также и от приложения. Для посылки пакетов в порт завершения предназначена функция PostQueuedCompletionStatus. Этот прием может использоваться для оповещения потоков о внешних событиях. Для создания порта завершения ввода-вывода и подключения файлов к этому порту используется функция CreateIoCompletionPort. Для получения пакета из порта завершения ввода-вывода используется функция GetQueuedCompletionStatus**

**Для посылки приложением пакета в порт завершения ввода-вывода исполь зуется функция PostQueuedCompletionStatus**

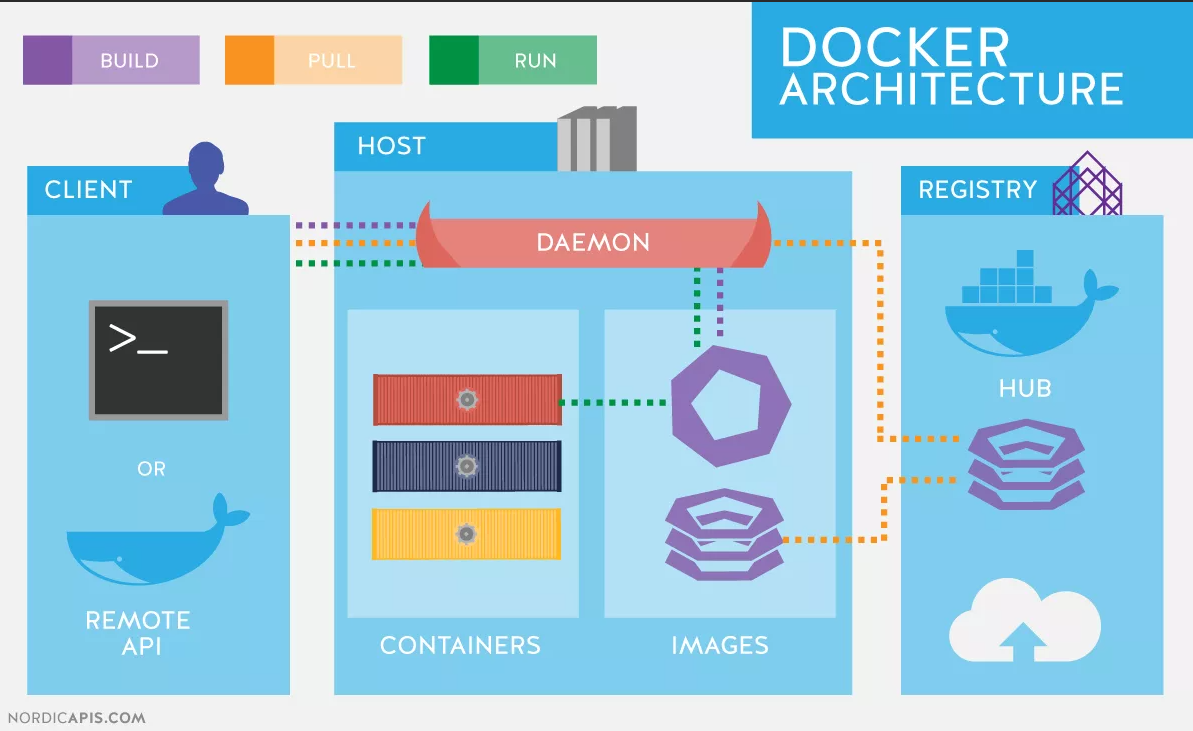
1. Платформа Docker: архитектура, назначение, принципы устройства, файловая система UFS, контейнеры, образы, основные команды.

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

**Архитектура Docker** состоит из нескольких компонентов:

* Docker daemon - отвечает за управление контейнерами и их запуск.
* Docker client - предоставляет интерфейс для взаимодействия с Docker daemon
* Docker registry - это централизованное хранилище образов Docker.

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



**Назначение** платформы Docker заключается:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Контейнеры Docker** создаются из **образов**, которые содержат все необходимые приложения и зависимости. Контейнеры могут быть запущены, остановлены, удалены и перезапущены с помощью основных команд Docker.

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

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

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

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

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

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

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

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

* docker run - запускает новый контейнер
* docker stop - останавливает контейнер
* docker ps - выводит список запущенных контейнеров
* docker images - выводит список доступных образов
* docker pull - загружает образ из реестра
* docker push - отправляет образ в реестр

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

Доцент каф. ИСиТ В.В. Смелов