**1.Назначение и классификация операционных систем.**

ОС управляет всеми устройствами компьютерной системы (процессорами, оперативной памятью, дисками, клавиатурой, монитором, принтерами, сетевыми устройствами и др.) и обеспечивает пользователя удобным интерфейсом для работы с аппаратурой.

Обычно ОС определяется через ее функции - под ОС понимают комплекс управляющих и обрабатывающих программ, который, с одной стороны, выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами, а с другой - предназначен для наиболее эффективного использования ресурсов вычислительной системы и организации надежных вычислений.

Другими словами, две основные функции (назначение) ОС:

1. предоставлять пользователю некую расширенную виртуальную машину, с которой легче работать (легче программировать), чем непосредственно с аппаратурой реального компьютера или реальной сети;

Для пользователя (приложения) работа с различными ресурсами (процессор, память, диски, периферийные устройства) выглядит так, как будто эти ресурсы находятся в его полном распоряжении, при этом в одной системе может выполняться одновременно несколько приложений, каждое со своим собственным набором ресурсов. В этом смысле говорят, что ОС реализует *виртуальную машину*, предоставляя приложениям средства многозадачности.

1. управлять ресурсами вычислительной системы.

**Классификация:**

1. По количеству одновременно работающих пользователей:

**Однопользовательские ОС** позволяют работать на компьютере только одному человеку. (однопользовательские однозадачные – пакетные, MS-DOS, однопользовательские многозадачные – Windows, Linux, Mac OS)

**Многопользовательские ОС** поддерживают одновременную работу на ЭМВ нескольких пользователей за различными терминалами. (многопользовательские многозадачные – сетевые ОС: NetWare, Unix, OS/2, Windows NT)

1. По числу процессов, одновременно выполняемых под управлением системы:

**Однозадачные ОС** поддерживают выполнение только одной программы в отдельный момент времени, то есть позволяют запустить одну программу в основном режиме.

**Многозадачные ОС** (мультизадачные) поддерживают параллельное выполнение нескольких программ, существующих в рамках одной вычислительной системы на некотором отрезке времени, то есть позволяют запустить одновременно несколько программ, которые будут работать параллельно, не мешая друг другу.

1. По количеству поддерживаемых процессоров (однопроцессорные, многопроцессорные):

**Многопроцессорные ОС** поддерживают режим распределения ресурсов нескольких процессоров для решения той или иной задачи. При многопроцессорном режиме работы два или несколько соединенных и примерно равных по характеристикам процессора совместно выполняют один или несколько процессов (программ или наборов команд). Цель такого режима – увеличение быстродействия или вычислительных возможностей.  
Многопроцессорные ОС разделяют на симметричные и асимметричные.

В **симметричных** ОС на каждом процессоре функционирует одно и то же ядро, и задача может быть выполнена на любом процессоре, то есть обработка полностью децентрализована. При этом каждому из процессоров доступна вся память.

В **асимметричных**ОС процессоры неравноправны. Обычно существует главный процессор (master) и подчиненные (slave), загрузку и характер работы которых определяет главный процессор.

1. По типу доступа пользователя к ЭВМ:

**ОС пакетной обработки**: в них из программ, подлежащих выполнению, формируется пакет заданий, вводимых в ЭВМ и выполняемых в порядке очередности с возможным учетом приоритетности.

**ОС разделения времени** обеспечивают одновременный диалоговый (интерактивный) режим доступа к ЭВМ нескольких пользователей на разных терминалах, которым по очереди выделяются ресурсы машины, что координируется операционной системой в соответствии с заданной дисциплиной обслуживания. Каждой программе, находящейся в оперативной памяти и готовой к исполнению, выделяется для исполнения фиксированный, задаваемый в соответствии с приоритетом пользователя интервал времени. Если программа не выполнена до конца за этот интервал, ее исполнение принудительно прерывается, и программа переводится в конец очереди.

**ОС реального времени** обеспечивают определенное гарантированное время ответа машины на запрос пользователя с управлением им какими-либо внешними по отношению к ЭВМ событиями, процессами или объектами. При таком режиме ЭВМ управляет некоторым внешним процессом, обрабатывая данные и информацию, непосредственно поступающую от объекта управления.

1. По разрядности кода операционной системы: восьмиразрядные, шестнадцатиразрядные, тридцатидвухразрядные, шестидесяти четырехразрядные:
2. По типу интерфейса (командные (текстовые), объектно-ориентированные (как, правило, графические):

**Пользовательский интерфейс** – это программные и аппаратные средства взаимодействия пользователя с программой или ЭВМ. Пользовательский интерфейс бывает командным и объектно-ориентированным*.*

**Командный интерфейс** предполагает ввод пользователем команд с клавиатуры при выполнении действий по управлению ресурсами компьютера. При этой технологии в качестве единственного способа ввода информации от человека к компьютеру служит клавиатура, а компьютер выводит информацию человеку с помощью монитора. Эту комбинацию (монитор + клавиатура) стали называть консолью.

1. По типу использования ресурсов (сетевые, не сетевые):

**Сетевые ОС** предназначены для управления ресурсами компьютеров, объединенных в сеть с целью совместного использования данных, и предоставляют мощные средства разграничения доступа к данным в рамках обеспечения их целостности и сохранности, а также множество сервисных возможностей по использованию сетевых ресурсов.

По особенностям методов построения: монолитное ядро или микроядерный подход. Большинство ОС использует **монолитное ядро**, которое компонуется как одна программа, работающая в привилегированном режиме и использующая быстрые переходы с одной процедуры на другую, не требующие переключения из привилегированного режима в пользовательский и наоборот. Альтернативой является **построение ОС на базе микроядра**, работающего также в привилегированном режиме и выполняющего только минимум функций по управлению аппаратурой, в то время как функции ОС более высокого уровня выполняют специализированные компоненты ОС — серверы, работающие в пользовательском режиме. При таком построении ОС работает более медленно, так как часто выполняются переходы между привилегированным режимом и пользовательским, зато система получается более гибкой — ее функции можно наращивать, модифицировать или сужать, добавляя, модифицируя или исключая серверы пользовательского режима. Кроме того, серверы хорошо защищены друг от друга, как и любые пользовательские процессы.

**8.Синхронизация, механизмы синхронизации.**

Синхронизация процессов — приведение двух или нескольких процессов к такому их протеканию, когда определённые стадии разных процессов совершаются в определённом порядке, либо одновременно.

Синхронизация необходима в любых случаях, когда параллельно протекающим процессам необходимо взаимодействовать. Для её организации используются средства межпроцессного взаимодействия. Среди наиболее часто используемых средств — сигналы и сообщения, семафоры и мьютексы, каналы (англ. pipe), совместно используемая память.

Механизмами синхронизации являются средства операционной системы, которые помогают решать основную задачу синхронизации - обеспечивать координацию потоков, которые работают с совместно используемыми данными. Такие средства являются минимальными блоками для построения многопоточных программ, их называют синхронизационными механизмами.

**Семафор** - это совместно используемый неотъемлемый целочисленный счетчик, для которого задано начальное значение и определены следующие атомарные операции.

* Уменьшение семафора (down): если значение семафора больше нуля, его уменьшают на единицу, если же значение равно нулю, этот поток переходит в состояние ожидания до тех пор, пока оно не станет больше нуля. Эту операцию называют также ожиданиям - wait;
* Увеличение семафора (up): значение семафора увеличивается на единицу; когда при этом есть потоки, которые ожидают на семафоре, один из них выходит из ожидания и выполняет свою операцию уменьшения. Если на семафоре ожидают несколько потоков, то вследствие выполнения операции увеличения, его значение остается нулевым, но один из потоков продолжает выполнение. Эту операцию также называют сигнализацией - post.

Фактически значение семафора определяет количество потоков, которое может пройти через этот семафор без блокировки. Когда для семафора задано нулевое начальное значение, то он будет блокировать все потоки до тех пор, пока какой-то поток его не "откроет", выполнив операцию up. Операции up и down могут быть выполнены любыми потоками, имеющих доступ к семафору.

**Мьютекс -** реализует взаимное исключение. Его основная задача - блокировать все потоки, которые пытаются получить доступ к коду, когда этот код уже выполняет некоторый поток.

Мьютекс может находиться в двух состояниях: свободном и занятом. Начальным состоянием является «свободный». Над мьютекс возможны две атомарные операции.

* Занять мьютекс: если мьютекс был свободен, он становится занятым, и поток продолжает свое выполнение; если мьютекс был занят, поток переходит в состояние ожидания, выполнение продолжает другой поток. Поток, который занял мьютекс, называют владельцем мьютекс;
* Освободить мьютекс: мьютекс становится свободным; если на нем ожидают несколько потоков, из них выбирают один, он начинает выполняться, занимает мьютекс и входит в критическую секцию. В большинстве реализаций выбор потока будет случайным. Освободить мьютекс может только его владелец.

**Правила упрощенного параллелизма** предназначены для упрощения программирования на базе мьютексов. Они основываются на том очевидном факте, что мьютекс защищает не код критической секции, а совместно используемые данные внутри этой секции. Особенности работы упрощенного параллелизма:

* Каждая переменная, которую совместно использует более одного поток, должна быть защищена отдельным мьютексом;
* Перед каждой операцией изменения такой переменной, соответствующий мьютекс должен быть занят, а после изменения освобожден;
* Если надо работать одновременно с несколькими совместно используемыми переменными, необходимо занять все их мьютексы до начала работы и освободить их только после полного окончания работы.

**Рекурсивный мьютекс** - особый вид мьютекса. Он позволяет повторное занятие тем же потоком, а также отслеживает, какой поток пытается его занять.

Когда это не тот поток, который уже занимает, мьютекс ведет себя как обычный, и поток переходит в состояние ожидания. Когда же это тот самый поток, внутренний счетчик блокировок этого мьютекс увеличивают на единицу, и поток продолжает свое выполнение. В случае увольнения мьютекс внутренний счетчик уменьшается на единицу, для других потоков рекурсивный мьютекс будет разблокирован только тогда, когда счетчик дойдет до нуля (то есть, когда все его блокировки одним потоком будут сняты).

Рекурсивные мьютексы менее эффективны в реализации, не могут быть использованы вместе с условными переменными, поэтому обращаться к ним нужно только тогда, когда без этого нельзя обойтись. Например, библиотека функций может использовать такие мьютексы для того, чтобы избежать взаимных блокировок при повторном вызове таких функций одним потоком.

**Условной переменной** называют синхронизационные примитивы, позволяющие организовать ожидания выполнения условия внутри критической секции, заданной мьютексом. Условная переменная всегда связана с конкретным мьютексом и данными, защищенными этим мьютексом. Для условной переменной определены следующие операции:

1. Ожидания (wait). Дополнительным входным параметром эта операция принимает мьютекс, который должен находиться в закрытом состоянии. Вызов ожидания происходит в ситуации, когда не выполняется некоторое условие и нужны потоки для продолжения работы. Вследствие выполнения ожидания поток прекращается, а мьютекс открывается. Так другие потоки получают возможность войти в критическую секцию и изменить там данные, которые она защищает, возможно, выполнив условие, необходимое потоку. На этом операция ожидания не заканчивается - ее завершит другой поток, вызвав операцию сигнализации после того, как условие будет выполнено.
2. Сигнализация (signal). Эту операцию поток должен выполнить после того, как войдет в критическую секцию и завершит работу с данными. Эта операция проверяет, нет ли потоков, ожидающих условной переменной, и если такие потоки есть, переводит один из них в состояние готовности. В результате восстановления поток завершает выполнение операции ожидания и блокирует мьютекс. Если нет ни одного потока, который ожидает условной переменной, операция сигнализирования не делает ничего, и информацию о ее выполнении в системе не сохраняют.
3. Широковещательная сигнализация (broadcast) отличается от обычной тем, что перевод в состояние готовности и восстановление выполняют для всех потоков, ожидающих этой условной переменной, а не только для одного из них.

**12.Библиотеки динамической компоновки DLL**

Применение библиотек динамической компоновки на практике подразумевает наличие в них специальных функций, которые могут быть вызваны из других приложений или DLL. Такие функции называются **экспортируемыми**. Вызывающее приложение определенным образом получает информацию об адресе и характеристиках экспортируемой функции (импортирует функцию), а затем осуществляет собственно ее вызов с передачей соответствующих параметров. Таким образом, приложение получает возможность задействовать все преимущества, которые дает применение библиотек динамической компоновки при разработке программного обеспечения.

Как правило, типичная DLL содержит некоторое количество внутренних функций, выполняющих реальную работу, а также несколько экспортируемых функций, которые играют роль интерфейса между DLL и приложениями. При этом экспортируемые функции могут вызывать внутренние функции DLL, что, вообще говоря, является хорошей практикой, так как обеспечивает инкапсуляцию и высокую степень внутренней связности в DLL.

Стандартной точкой входа для DLL, т.е. аналогом WinMain для приложений является функция DllEntryPoint() (еще она может называться DllMain() ). Она применяется достаточно редко – как правило, в случаях, когда требуется выполнять какие-то инициализационные действия при загрузке DLL, поскольку в этом случае DllEntryPoint() вызывается автоматически. Тем не менее, DllEntryPoint() является обязательным элементом библиотеки DLL, хотя и содержит обычно только оператор выхода.

Функция GetSomeString() является внутренней функцией DLL и не экспортируется. Таким образом, другие приложения не имеют прямого доступа к ее коду. В приведенном примере функция GetSomeString() выполняет некоторую полезную работу, т.е. возвращает определенную строку, которая используется вызывающим приложением.

Функция Test() является экспортируемой, т.е. она доступна другим приложениям. Для этого она должна быть описана со специальным модификатором, который определяется используемой версией компилятора (в приведенном примере **\_\_declspec (dllexport)**). При использовании этого модификатора информация о функции будет помещена в таблицу экспорта DLL и станет доступна внешним приложениям.

Таким образом, после успешной сборки DLL и заполнения таблицы экспорта внешние приложения смогут вызывать функцию Test() и, соответственно, получать в свое распоряжение строку, возвращаемую функцией GetSomeString(), т.е. задействовать функционал, предоставляемый библиотекой DLL.

**13.Явное и неявное связывание библиотек динамической компоновки.**

Вызов функций из DLL существенно отличается в зависимости от того, применяется неявное или явное связывание библиотеки DLL с приложением. При **неявном связывании** информация о вызываемой функции и библиотеке DLL, в которой она реализована, помещается в таблицу импорта вызывающего приложения, и оно получает всю необходимую информацию для вызова функции. Это действие выполняется на этапе компоновки приложения с помощью т.н. библиотеки импорта. Такая библиотека создается с помощью специальной утилиты LIB. Полученная библиотека импорта содержит сведения о функциях, экспортируемых из DLL. Таким образом, она является посредником между DLL, которая экспортирует функции, и EXE-файлом, который эти функции использует.

Поскольку вызываемая из DLL функция является внешней по отношению к вызывающему приложению, она должна обязательно иметь прототип в вызывающей программе, определенный в точном соответствии с его описанием в DLL. При этом прототип обязательно должен иметь модификатор **extern**, показывающий, что функция является внешней по отношению к данному модулю, и ее код следует искать где-то во внешних модулях (в данном случае в DLL).

Как видно, приложение содержит прототип функции Test(), совпадающий с ее описанием в коде библиотеки DLL (за исключением модификатора \_\_declspec (dllexport)), и имеющий модификатор **extern.** Это дает возможность при компоновке с использованием соответствующей библиотеки импорта напрямую осуществлять вызов функции DLL, неявно связанной с приложением, как показано в коде функции WinMain(). Данная схема вызова удобна тем, что не требует от программиста никаких дополнительных усилий для связи с DLL и вызова нужной функции.

При **явном связывании** вызов функции осуществляется иначе и требует от программиста следующих явных действий:

* загрузка DLL (через LoadLibrary())
* получение адреса нужной функции (через GetProcAddress())
* выгрузка DLL (через FreeLibrary())

Как видно, вызов функции при явном связывании осуществляется значительно сложнее и требует, в частности, иного описания функции в самой библиотеке DLL.

Наличие в описании модификатора **extern “C”** продиктовано особенностями явного связывания, согласно которым вызывающей программе должно быть известно внутреннее имя функции в DLL. Дело в том, что компиляторы С++ используют т.н. расширение имен (name mangling), предназначенное для поддержки перегруженных функций. При расширении имен внутреннее имя функции отличается от имени в тексте программы, поскольку к нему дописываются специальные символы расширения. Таким образом, попытка вызвать функцию из DLL по имени будет неудачной (имя функции, переданное вызывающим приложением, и внутреннее имя функции будут отличаться). Один из способов обхода данной проблемы – это отключение расширения имен, которое достигается с использованием специального модификатора **extern “C”**. В этом случае внутреннее имя функции будет совпадать с именем функции в коде вызывающего приложения.

Вызывающее приложение определяет указатель на функцию, имеющую тот же тип, что и функция Test(), которая в этом примере вызывается из DLL. Это необходимо, так как вызов функции при явном связывании происходит по ее указателю. Для удобства дальнейшего использования создается тип такого указателя, имеющий имя PFN.

Для вызова функции Test() из DLL вызывающее приложение вначале должно загрузить библиотеку DLL, содержащую эту функцию, в адресное пространство процесса, что осуществляется посредством функции WinAPI LoadLibrary() как показано в приведенном примере. При этом в переменной hMod сохраняется дескриптор модуля DLL, загруженной в адресное пространство процесса.

Далее посредством вызова функции WinAPI GetProcAddress() выполняется получение адреса функции Test() внутри модуля DLL, ранее загруженной в процесс вызывающего приложения. Полученный адрес приводится к ранее определенному типу PFN. Так как теперь известен адрес функции Test(), то выполняется ее фактический вызов, в результате которого в переменную str возвращается строка, представляющая собой результат выполнения функций из библиотеки DLL. Данная строка выводится в MessageBox, на чем использование функционала DLL приложением и завершается. Так как теперь ранее загруженная DLL более не требуется, то происходит ее выгрузка из адресного пространства процесса вызывающего приложения посредством функции WinAPI FreeLibrary(), после чего вызывающее приложение завершает работу. Следует особо отметить, что явное связывание при больших усилиях, связанных с вызовом функций из DLL, обеспечивает значительную гибкость и лучший контроль со стороны программиста, чем неявное связывание.