**1. Назначение ОС. Функции ОС. ОС как многоуровневая виртуальная машина. Понятие процесса, ресурса. Эволюция ОС, тенденции в развитии ОС.**

**Операционная система** – набор программ, скоординировано управляющих ресурсами вычислительной системы (ВС) и обеспечивающих пользователю интерфейс для общения с ВС.

**Функции ОС**

1. Обеспечении запуска других программ, не входящих в ОС;

2. Организация файловой системы;

3. Обеспечение другими программами доступа к ресурсам ВС;

4. Управление доступом к ресурсам;

5. Организация пользовательского интерфейса;

6. Поддержка многозадачного и многопользовательского режима.

Задачи ОС а

1.Повышение эффективности использования компьютера, путем рационального управления его ресурсами.

2.Обспечение пользователю инструментов для управления ВС.

3.Обеспечение программисту ряда удобств, за счет предоставления для него расширенной виртуальной машины.

Многоуровневая виртуальная машина

Имеются в виду Уровни абстрагирования от деталей аппаратного устройства в ВС.

Уровни абстрагирования:

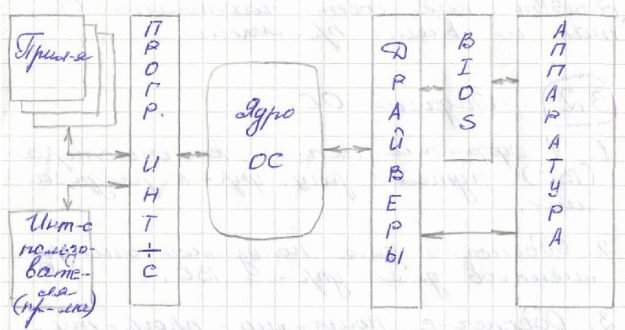
1.Аппаратный уровень, отсутствие вспомогательных средств.

2.Уровень BIOS, вызов функций Биос путем прерывания процессора

3.Функции DOS низкого уровня

4.Функции DOS высокого уровня

5.Уровень команд DOS



Процесс – программа в процессе выполнения.

Каждому процессу выделяется память под код, стек, данные, место на экране и т.д. В течение времени могут занимать и освобождать место.

Ресурс – все то, что может потребоваться программе в процессе её работы (блок опер. памяти, порт, диск, место на экране, процессорное время)

Планирование ресурсов – определение кому, когда и в каком количестве требуется выделить ресурс.

Отслеживание ресурсов – поддерживание оперативной информации свободен\занят ресурс и в каком количестве.

Процессы, порождённые при запуске программы несколько раз параллельно – экземпляры приложений.

Этапы развития ОС:

1. 1945-1955 – ОС не существовало, управление ВС вручную

2. 1955-1965 – появление систем пакетной обработки, разделение персонала на программистов и операторов

3. 1965-1980 – ЭВМ на микросхемах малой и средней интеграции, появление семейства программно-совместимых ЭВМ, режим мультипрограммирования (несколько программ на 1 процессоре), режим spooling (прием заданий в темпе поступления, обработка по мере возможности), многопользовательская ОС в режиме разделения времени (1 машина, много юзеров, иллюзия работы на разных ЭВМ)

4. 1980- ЭВМ на БИС и сверх БИС, основное внимание пользовательскому интерфейсу, бурное развитие сетей, многозад./многопольз. ОС, развитый графический интерфейс, сетевое взаимодействие, ПК

**2. Принципы построения ОС. Классификация ОС по различным признакам.**

* По особенностям алгоритмов управления ресурсами:

− многозадачные – однозадачные + управление доступом к ресурсам;

− однозадачные – предоставление пользователю виртуальной машины.

* По признаку количества пользователей:

− однопользовательская (DOS) – одновременная полноценная работа с 1-м пользователем;

− многопользовательская (unix) – одновременная полноценная работа с многими пользователями.

* По способу реализации многозадачности:

− не вытесняющая многозадачность – активный процесс явно/неявно должен отдавать управление другому процессу;

− вытесняющая многозадачность – активный процесс может быть прерван с целью передать управление другому процессу(таймер), решение о переключении принимает ОС, планировщик.

* По наличию поддержки многопроцессорной обработки

• Симметричные – все процессоры (ядра) одинаковы и выполняют, как системный, так и прикладной код (нет такого разделения на деле);

• Асимметричные (на одном процессоре системные код, на другом выполняются прикладные задачи).

* По особенностям аппаратных платформ:
* ОС ПК
* ОС Миникомпьютера
* ОС Кластеров
* ОС сетей ЭВМ

ОС больших машин более сложные. Используются планировщики с приоритетом.

*Кластер-*слабо связанная совокупность нескольких ВС, работающих совместно для выполнения общих приложений, физически представленная локальной сетью со специальными каналами передачи данных, представляется пользователю в виде единой ВС.

* По переносимости системы на другие платформы

− мобильные

− на конкретной платформе

* По особенности областей использования

− Система пакетной обработки – max количество запросов в единицу времени, запрос мб обработан долго, но пакет обрабатывается быстро. Обеспечивается сбалансирование загрузки всех устройств.

− Система разделения времени – макс быстрое получение отклика на запрос, удобство и эффективность работы пользователя. Каждый пользователь работает через терминал. Каждый терминал имеет отрезки времени, в течении которого система обрабатывает его команды

− Система реального времени – способность выдерживать реально заданные интервалы времени для получения результата. Не предназначены для запуска произвольных программ. Такие ОС работают со специальными программами с известными различными параметрами.

* По особенности методов построения:

− На основе монолитного ядра – в его составе функционируют большие процессы (потоки), которые отвечают за прерывания, сист. вызовы, выделение памяти, организацию файловой системы. Обычно эти потоки выполняются не вытесняющим образом. Менее надежный способ, но проще реализовать.

− На основе микроядра – исполняется планировщик процессов, а также имеется средство общения между процессами - сообщения. Характерна вытесняющая многозадачность. Более медленные чем первый, но более надежные.

**3. Многозадачность. Понятие процесса, контекста и дескриптора процесса. Виды многозадачности, их достоинства и недостатки. Жизненный цикл процесса. Многопоточность.**

**Многозадачность** – свойство ОС обеспечивать возможность параллельной обработки нескольких процессов.

**Процесс** – программа, набор машинных инструкций, запущенный и исполняемый в данный момент времени на ВС – программа в процессе выполнения.

**Дескриптор процесса** – информация о процессе необходимая для работы алгоритма планирования (идентификатор процесса, данные о приоритетах и привилегиях, расположение кодов сегмента и стека процесса).

**Контекст процесса** – информация, необходимая для ОС, чтобы многократно прерывать и восстанавливать исполнение кода процесса.

Контекст нужен процессу для его корректного выполнения, дескриптор нужен ОС для переключения процессов и планирования.

|  |  |
| --- | --- |
| Невытесняющая | Вытесняющая |
| − процесс выполняется до тех пор, пока программа сама, по своему алгоритму не отдаст управление | − решение о переключении принимает планировщик без участия процесса и в случайное время |
| + между переключениями процессов, процесс занимает ресурсы монопольно;  + задачи, требующие max использования процессора, могут быть более эффектно закодирован, если реже отдавать управление другим процессорам  + переключение контекстов простое и быстрое  − необходимость отдачи управления, частичное планирование процессов  − зацикливание одного процесса, приводит к краху системы  − управляемость системы, часто теряется на произвольные участки времени. | + зацикливание или зависание 1 процесса, не останавливает работы других  + система способна начать обработку действий пользователя (запроса) сразу  + процессы переключает система, можно делать циклы долгими  − более сложный медленный алгоритм переключения контекста  − возможность прерывания процесса в любой момент времени, требует применение средств защиты при работе с разделенными ресурсами |

К *невытесняющим* относятся, системы с относительным приоритетом.

Уничтожение

Аварийная ситуация

Создание

уничтож

Создание

|  |  |
| --- | --- |
| Жизненный цикл процесса (однопроцессорная ОС) | |
| ВЫПОЛНЕНИЕ  (1 процесс) | активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором |
| ОЖИДАНИЕ  (несколько процессов) | пассивное состояние процесса, не может выполняться по своим внутренним причинам - ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса |
| ГОТОВНОСТЬ  (несколько процессов) | также пассивное состояние процесса, заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс он готов выполняться, однако процессор занят выполнением другого процесса |
| ЗАВЕРШЕНИЕ | пассивное состояние; процесс завершил выполнение кода, освободил ресурсы, но ОС продолжает поддерживать служебную информацию, которую могут запросить другие процессы |
| Создание | Загрузка кода из исполнимого файла в память и выделение начальных ресурсов, прописанных в исполнимом файле , поставить процесс в очередь готовых, создать контекст и дескриптор |
| Уничтожение | Когда другие процессы, запросившие доступ к его системной информации, уведомляют, что он не требуется |

Планировщик работает с очередями процессов, к-е стоят из их дескрипторов. Каждый элемент очереди содержит указатель на сохраненный контекст и след./пред. элементы очереди.

Потоки выполнения – процедуры, исполняемые параллельно в рамках одного процесса. Все потоки в рамках одного процесса имеют общее адресное пространство, глобальные переменные, хэндлы открытых ресурсов. У каждого потока имеется свой стек и свой контекст. (включая состояние регистров и счетчик команд, указатель стека)

**Многопоточность** – свойство ОС, состоящее в том, что процесс может состоять из нескольких потоков, выполняющихся «параллельно».

Суть многопоточности – квазимногозадачность на уровне одного процесса

+упрощаются программы за счет использования общего адресного пространства

+меньше временных затрат на создание потока

+повышается производительность за счет распараллеливания процессорных операций и операций ввода\вывода

**4. Понятие ресурса ОС. Управление ресурсами. Проблемы управления ресурсами, связанные с многозадачностью ОС. Проблема синхронизации процессов. Средства синхронизации. Проблема взаимной блокировки процессов.**

**Ресурс** – все то, что может потребоваться в программе в процессе её работы (вв\выв, блок оперативной памяти, процессорное время).

|  |  |
| --- | --- |
| **Управление ресурсами** | |
| Планирование ресурсов | определение, кому, когда, в каком количестве необходимо выделить данный ресурс (это делает планировщик) |
| Отслеживание ресурса | поддерживание оперативной информации о том занят ли или освобожден ресурс, в каком количестве(мониторинг) |

**Проблемы взаимодействия**

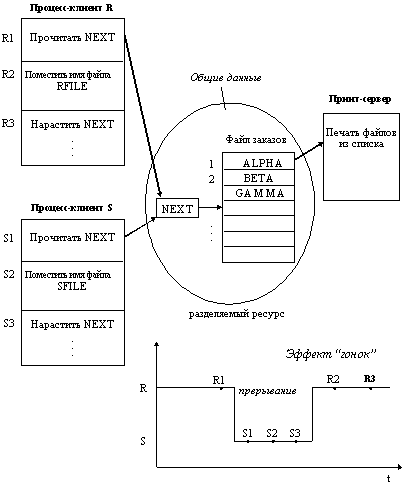
В современных многозадачных ОС каждому процессу выделяются изолированные, защищенные от доступа других процессов блоки памяти.

Передача информации через общие блоки памяти невозможна.

Для обмена данными используются механизмы ОС: сообщения, трубопроводы, почтовые ящики, разделяемые дисковые файлы.

**Проблема синхронизации**: необходимо не допускать одновременного доступа к параллельно выполняемым процессам с разделённым ресурсом. Актуально для вытесняющих алгоритмов.

Параллельная печать на принтере. Во втором случае файл от S не попадает на печать, т.к оба прочитали Next, а R запишет поверх S.

Гонки – два или более процесса обрабатывают разделяемые данные, и конечный результат зависит от отношения скоростей процессов. Средства синхронизации позволяют совершать блокировки доступа других программ к разделенным ресурсам.

**Критическая секция** − *часть программы в которой доступ осуществляется к разделяемым ресурсам;*

− участок кода одного из процессов использующийся в мультипрограммном режиме, в течение к-го выполнение определенных действий параллельно исполнимых процессов недопустимо.

− участок кода в пределах, которого алгоритм полагается на целостность данных, которая потенциально может быть нарушена параллельным исполнимым процессом.

В программе КС окружены специальными фрагментами кода или систем вызова они называются вх\выходом из КС или *критическими скобками.*  
Во время работы с КС доступ к ресурсу доложен иметь один процесс-принцип взаимного исключения.

**Варианты реализации Критических скобок (КС)**

* Запрет прерываний перед КС и их разрешение после (никогда не применяется)
* В случае зацикливания – беда
* Блокирует все процессы, а не только те, что могут помешать
* В многопроцессорной системе это не поможет
* ОС полноценно запрещает пользовательскому коду запрет прерывания
* Введение разделяемой блокирующей переменной

Перед КС существует переменная D глобальная, если она=0 то ресурс занят, если=1 –свободен

* Механизм событий

**Событие** – системный объект характеризуется состояниям (взведен\сброшен). ОС поддерживает возможность ожидания события, при этом если событие сброшено процесс поступает в очередь ожидания, а при взведённом – в очередь готовых. Wait & Reset(D) /пока не 1 & уст. в 0/ - КС – Set(D) /уст. в 1/

* Семафоры Дейкстры (можно реализовать доступ к разделенным ресурсам)

**Семафор** – системный объект, характеризуется целым неотрицательным состоянием и определены две операции:

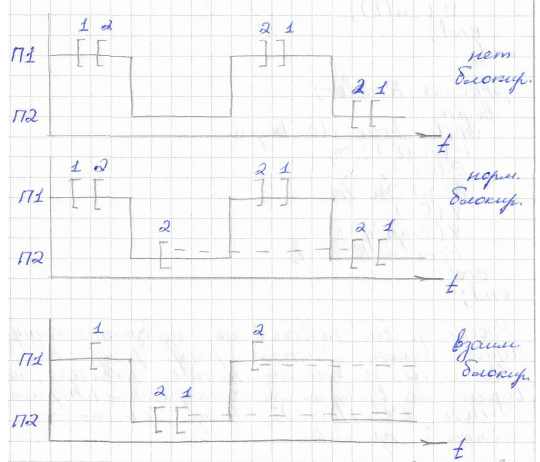
увеличение V(s):S++

уменьшение P(s): (s=0 => wait пока не s>0) или (s>0 => s--)

P(s) – KC – V(s)

**Проблема взаимной блокировки** возникает при одновременном использовании нескольких объектов синхронизации и при некорректном применении блокировки.

Опасность тупиков: при использовании нескольких объектов синхронизации одновременно потоки будут бесконечно ждать друг друга и никогда не выйдут из этого состояния

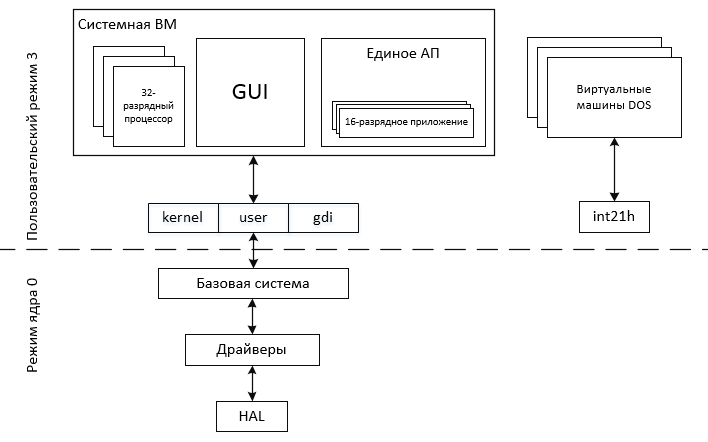
Методы борьбы:

1. Механизм контрольных точек (ставит программист). В контрольной точке сохраняется полное состояние процесса. Есть возможность отката при взаимной блокировке.

2. Два объекта синхронизации использовать в одинаковом порядке.

**5. Общая характеристика ОС Windows. Архитектура Windows.**

Структура – состав системы. Архитектура – принцип взаимодействия основных частей.



API

API

Windows – многопользовательская (имитация), многозадачная ОС с графическим интерфейсом. В ней использовано 4 варианта файловой системы (FAT 12, FAT 16, FAT 32, NTFS); вытесняющая многозадачность; многопоточность.

GUI: графический интерфейс пользователя

HAL: уровень абстрагирования от аппаратуры.

Все 16 разряд приложения делят между собой единое адресное пространство и не могут управляться в соответствии с принципом многозадачности.

32 разряд приложения выполняются в разных адресных пространствах в соответствии с принципом многозадачности.

В некоторых версиях Windows:

* Библиотека GDI32.dll – функции рисования, графические функции вывода (как на экран, так и на принтер).
* Библиотека USER32.dll – функции управления окнами, функции преобразования строк.
* Библиотека KERNEL32.dll – функции управления многозадачностью, функции обеспечения связи между процессами, поддержка самого механизма dll, поддержка файлового ввода/вывода.

Вирт.машины MS-DOS – дисковая операционная система - технология, позволяющая запускать 16/32-разрядные приложения [DOS](https://ru.wikipedia.org/wiki/DOS) и 16-разрядные приложения [Windows 3.x](https://ru.wikipedia.org/wiki/Windows_3.x) на процессорах [Intel 80386](https://ru.wikipedia.org/wiki/Intel_80386) и выше, когда уже запущена другая [операционная система](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0).

Базовая система

* + Планирование (выделение) памяти
  + Планирование потоков под выделение памяти
  + Подсистема управления файлами

**6. Шаблон программы для Windows. Понятие сообщения. Принципы создания окон и обработки сообщений.**

Когда в ОС возникает событие, требующие реакции со стороны программы, программа получает сообщение, программа должна постоянно обрабатывать сообщения. Сообщения всегда адресованы окнам. С каждым окном связана процедура, в которую передаются все входящие сообщения – оконная процедура. Сообщения могут ставиться в очередь/передаваться в оконную процедуру. Сообщения передаются через очередь, программа должна сама извлекать из очереди сообщения и передавать их в обработчик.

Сообщение – блок данных размером 28 байт фиксированной структуры:

type TMSG = record;

hwnd : THandle; //хэндл окна, которому адресовано сообщение

message: integer; //идентификатор сообщения (задаёт тип сообщения)

wParam, lParam: integer; //данные, передаваемые в сообщении

time: dWord; //время посылки сообщения

pt: TPoint; //положения курсора мыши в момент посылки сооб-я

end;

Оконная процедура реализуется реакцией на ВСЕ сообщения, адресованные некоторому окну или группе окон.

Прототип оконной процедуры:

function WndProc1(hwnd: Thandle; msg: integer; wparam, lparam: integer): integer; stdcall;

Цикл выборки и обработки сообщений предназначен для выборки из очереди сообщений посланных буф. способом и отправки их на обработку, занимается анализом очереди

var msg: TMSG;

while GetMessage (msg, 0, 0, 0) = true do begin //получить сообщение

TranslateMessage (msg); //Win транслирует сообщение от клавиатуры

DispatchMessage (msg); //Win вызовет оконную процедуру

end; //выход по wm\_quit, на которое GetMessage вернет false

GetMessage – обращается к очереди, если в ней нет сообщений, то ждет (поток в состоянии ожидания). Если есть – извлекает из очереди и помещает в структуру msg.

TranslateMessage – обеспечивает прохождение символьных сообщений (инф-я о набираемых буквах и цифрах).

DispatchMessage – вызывает оконную процедуру того окна, к-му адресовано сообщение.

С каждым окном связана оконная процедура. В потоке программы порождается окно, сообщения для этого окна помещаются в очередь сообщений потока, породившего окно.

Сообщения могут посылаться

* Не буфериз. способом, при посылке сообщения окну своего потока сразу запускается оконная процедура. Если другому – создается высокоприоритетная очередь небуфф. сообщений и она обрабатывается при каждом обращении этого потока к своей обычной очереди сообщений.

SendMessage(hwnd: THandle; msg: integer; wparam, lparam: integer): integer

- возвращает результат обработки сообщения. Обрабатывается долго, но сразу.

* Буферным способом, сообщения помещаются в очередь того потока, который создал окно получателя. Сообщения в очереди ожидают в состоянии FIFO. Поток должен извлекать сообщения из очереди и отправлять их на обработку в оконные процедуры.

PostMessage(hwnd: THandle; msg: integer; wparam, lparam: integer): longbool

– помещает сообщение в очередь. Результат – удалось\нет (хэндл несуществующего окна).

Очередь сообщения создается для каждого потока, который создавал окна или делал вызовы.

Все сообщения Windows имеют идентификаторы. Можно придумать и свои.

В программе можно выделить 3 главных части:

1. Инициализация (создание главного окна).

2. Цикл выборки и обработки сообщения

3. Оконные процедуры (реализ. обработку сообщения) – обработчики сообщений.

Общая структура шаблона программы:

function WndProc (…): stdcall;

begin

case msg of

wm\_paint: begin … end;

wm\_close: begin … end;

else

result:=DefWindowProc(…);

end;

end;

begin

{Регистрация оконного класса

Порождение окна

Вывод окна на экран

Цикл выборки и обработки сообщения}

end;

Все окна в Windows порождаются, как экземпляры некоторого оконного класса.

**Оконный класс** – некоторый блок памяти, описывает основные свойства и поведения окон, которые ему принадлежат. Каждый класс содержит ссылку на оконную процедуру, единую для всех окон класса. В системе есть набор предписанных классов, но можно создать собственный.

RegisterClassEX() – возвращает переменную типа nAtom (специальным образом зарегистрированная в системе строка, доступная по целому ID – 16 разр), которое можно использовать для идентификации оконного класса.

CreateWindow() – создание окна

CreateWindowEx()

Успех создание – ненулевой хэндл окна, используется для идентификации окна при последующих операциях с ним.

Все окна и классы программы автоматически уничтожаются при завершении программы.

**7. Окно как средство разделения пространства экрана между приложениями. Принципы отображения информации в окнах. Сообщение WM\_PAINT.**

Окно – средство разделения пространства экрана между приложениями. Принцип отображения информации в окнах. Сообщение WM\_PAINT.



Окно – средство разделения экрана на процессы. Координаты рисования можно задать любые, но рисование проводиться только в клиентской области.

Принципы рисования окон:

1. Использование буфера сохранения, когда окно появляется на экране все, что было за ним сохраняется в памяти. (Плохо для графических окон, применяется в Windows для вспомогательного меню.)

2. Для обычных окон буфер не используется, но существует требование: окно должно в любой момент времени быть способно перерисовать свое содержимое. ОС отслеживает, какие участки, каких окон требуют прорисовки и посылает этим окнам сообщение WM\_PAINT.

Сообщение WM\_PAINT посылается окну, когда необходимо отрисовать что-либо в клиентской области.

В параметрах иногда передается контекст устройства, но обычно он = 0.

Сообщение посылается буферизированным способом и имеет низкий приоритет. Извлекается из очереди, когда других сообщений нет.

Функция GetMessage не удаляет его из очереди. Оно удаляется из очереди, когда в окне перестает существовать недействительный регион и помещается в очередь, когда такой регион появляется.

На самом деле WM\_PAINT в очередь не помещается никогда, но, если очередь пуста при вызове GetMessage и недействительном регионе функция, как бы вызывает это сообщение.

Оно не должно посылаться программам с помощью Send\Post Message. Вместо этого мы вручную делаем окно недействительным InvalidateReсt или ValidateReсt.

**8. Средства Windows для отображения информации. GDI. Примитивы GDI. Объекты GDI. Основные группы функций GDI. Возможности Windows по выводу векторной и растровой графики, текстовой информации. Регионы.**

GDI – содержит все функции рисования. Они обеспечивают отображение информации на любом устройстве (в окне, на принтере). GDI поддерживает аппаратно независимую графику.

Примитивы GDI – разновидности графических объектов, к-е мб отображены.

* Перо(Pen) (отрезки, прямоугольники, эллипсы, окружности, дуги, ломанные линии)
* Кисть «Brush» – закрашенные области.
* Битовые образы” Bitmap” – двумерный набор битов, соответствующих пикселям устройства отображения.
* Текст «Font» – растровые шрифты, разн. размер, курсив, жирный, зачеркнутый, вывод под любым углом.

**Контекст устройства** – внутренняя структура Windows, содержащая информацию о характеристиках устройства отображения (глубина цвета, конфигурация региона отсечения, разрешающая способность). В контексте устройства всегда выбран только 1 объект каждого типа. Выбранный объект задает способ отрисовки своих примитивов. Объекты GDI самостоятельны и могут существовать вне контекста. Их можно создать/удалить в момент рисования и заранее/потом. Чтобы активизировать объект, нужно выбрать его в контекст. hOldPen:=SelectObject(hdc, hNewPen);

1) Возвращать контекст в том же состоянии, в каком получили

2) Объект создали – объект удалили. Не создавали – не удаляем. Если выбран в контексте, удалить не получится.

**Группы функций GDI**

* Функции получения и освобождения контекста устройства.

BeginPaint, EndPaint, GetDC (если здесь указать 0, то получается контекст для рисования на всем экране), ReleaseDC

* Функции получения информации о контексте устройства //GetTextMetrics
* Функции рисования линий, эллипсов, текста, областей
* Функции для работы с объектами GDI
* Функции для управления атрибутами контекста устройства(SetBkMode)

**Растровые операции**

При рисовании примитивов могут закрашиваться заданным цветом, либо цвет может формироваться, как результат логической функции, применяемый к текущему цвету SetRop2(hDC, Mode)

mode: R2\_COPYPEN//рисуем поверх «заменяя»

R2\_XORPEN//рисуем по операции или

*Регион* - плоская фигура, произвольной формы с заданными координатами (лев верх и правый ниж угол). Простой регион – прямоугольник. Если регион имеет сложную форму –это набор прямоугольников высотой 1 пиксель.

* Видимый регион - часть клиентской области окна, находящегося в пределах экрана и не скрывается под другим окном
* Недействительный регион (регион обновления) – часть клиентской области требует отрисовки.
* Регион отсечения – регион, в пределах к-го разрешено рисование (при перерисовке окна регион отсечения = недействительному).

**9. Общая характеристика средств для ввода информации, предоставляемых Windows: сообщения клавиатуры, мыши и таймера, оконные органы управления, меню, акселераторы.**

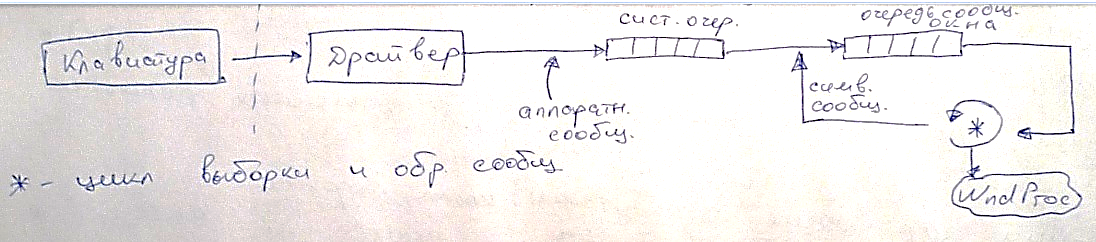
устройства ввода = мышь, клавиатуру, логический таймер.

1. **Клавиатура**

Пользователь нажимает клавишу, активное окно, имеющее фокус ввода, получает сообщение, содержащее информацию о нажатой клавише. То же происходит и при отпускании клавиши. Эти сообщения идут через очередь и делятся на:

Аппаратные - виртуальный код нажатой клавиши, идентифицируют нажатую кнопку, посылаются при нажатии всех клавиш. Содержат scan-cod клавиши и виртуальный код клавиши, определенный в Windows.

Символьные - при нажатии алфавитно-цифровых клавиш и содержит информацию о наборном символе с учетом текущего регистра и языковой раскладки. Символьные сообщения программа посылает сама себе посредством ф-ии TranslateMessage. Если приходит апп. сообщение, то при его обработке в TranslateMessage формируется символьное сообщение.



В очередь потока сообщения поступают по одному.

1. **Мышь**. (левая, правая, колесо).

Если пользователь производит манипуляции с мышью, то ОС посылает сообщения окну, над которым находится указатель мыши.

Общая логика прохождения сообщения

* Мышиные сообщения посылаются тому в поле, которого находиться курсор. Сообщение о нажатии мыши м не сопровождаться сообщением об отпускании.
* (очереди из сообщений мыши нет). В очереди находится только одно сообщение от мыши. Некоторые м пропадать.
* Если в поле окна происходят манипуляции с мышью, окну всегда посылается WM\_NCHITTEST. Цель – спросить у окна, где у нее сейчас курсор. Обработчик должен вернуть код зоны окна, в к-ой находится курсор.
* Существует возможность отследить отпускание кнопки мыши, даже если курсор за пределами окна.

SETCapture(hWnd) – после этого окно продолжает получать сообщения от мыши пока кнопка нажата, даже если курсор за пределами окна.

WM\_LBUTTONDOWSN:SETCapture(hWnd);//что бы установить захват

При отпускании кнопки REALEASECapture;//снять захват.

1. **Таймер**. Программа может запросить у системы получение сообщений WM\_TIMER. Для этого нужно создать логический таймер. Сообщение WM\_TIMER имеет низкий приоритет. Если программа не успевает обработать сообщения, то сообщение теряется (мб только одно в очереди).

Не может отслеживать точные интервалы времени, т.к. они идут через очередь; для их генерации используется системный таймер с периодом 55мс. У любого логического таймера есть числовой идентификатор. Сообщения таймера передаются в таймерные процедуры, либо обрабатываются обычным путем.

1) Создать таймер – вкл

2) Уничтожить таймер – выкл

3) Старый уничтожить, новый создать – изменить параметры.

1. **Controls.** Являются дочерними окнами создаются с помощью create window (дочерние окна главного окна). Являются экземплярами определенных в системе оконных классов.

Button-кнопка,radiobutton-переключатель,checkbox-флажок;edit-поле ввода,scrollbar-полосы прокрутки;combobox-выпадающий список;listbox-список строк;static-прямоугольная область для ввода информации.

1. **Меню** Как и дочерние окна управления, при различных действиях посылают программе wm\_command в то окно, где меню прикреплено: wparam=0|id (идентиф. пункта меню); lparam=0.

Меню бывает

* Главное(main)-отображаемое в виде строки или нескольких строк верху окна
* Подменю (drop-down) вертикальные строки, появляющиеся при выборе пунктов главного меню
* Всплывающее меню (pop-up) появляется при щелчке правой кнопки по тому или иному органу управления

Пункты меню

1.Разрешены-Enabled; - посылаются wm\_command

2.Запрещены-Disbled; - выглядят так же, как и разрешенные

3.Недоступны-Grayed

1. **Акселераторы** – описание комбинации горячих клавиш содержатся в таблице акселераторов. При нажатии акселератора окну приходят wm\_command (старший бит wparam=1, младший=id) с указателем идентификатора комбинации. Прием сообщений от акселераторов реализуется в цикле выборки сообщений из очереди с помощью функции TranslateAccelerator, которая принимает значение 1 или 0. Обрабатывается сообщение уже в оконной процедуре.

**10. Ресурсы Windows как средство хранения данных программы. Виды ресурсов. Принципы использования ресурсов.**

Особые дополнительные данные хранимые в теле исполнимых файлов .exe или .dll. Для доступа к ресурсам используются функции WinAPI, напоминающие ср-ва доступа к файлам.

Ресурсы обладают числовым или строковым идентификатором для загрузки.

Для ресурса существует язык описания ReassureScript файлы.

В виде ресурсов может быть любая информации не более 2 ГБ.

Стандартного формата

* Битовые образы
* Значки иконки (сейчас любого размера, глубина цвета 16 px, квадратный)
* Курсор мыши-32/32 2 цвета
* Строки (символьная строка, только числовые идентификаторы)
* Описание меню (можно описать полосовое так и отдельно стоящее меню)
* Описание горячих клавиш(акселераторы)
* Описание внешнего вида диалогового окна (dialog boxes)

Произвольный формат.

Файл ресурсов дб в явном виде упомянут в программе с использованием директивы $Resource ($R) – {$R myres.res}. Не следует называть файл ресурсов именем проекта (делфи создает такой ресурс автоматически). Ресурсы включаются в исполнимый файл на этапе его компоновки одновременно с файлами .OBJ.

Загрузка ресурсов: handle:=LoadXXX(hModule//откуда грузить ресурс hInstance//, ‘id\_res’ // pointer(n) – если числовой индекс//). Загруженный ресурс должен быть удален – DeleteOblect(handle).

**11. Библиотеки динамического связывания (DLL). Написание DLL на Delphi. Использование DLL. Способы загрузки DLL. DLL и многозадачность.**

DLL – библиотеки динамического связывания - файл, содержащий наборы готовых к исполнению процедур и ресурсы, в ОС винд это исполнимый файл. Ф-ии из DLL вызываются файлами ехе, а также из функций других DLL. Файл ехе или DLL, загруженный в память, - модуль. К ф-ям DLL м одновременно обращаться любое к-во модулей.

*Динамическое связывание* – функции не включены в код программы на этапе сборки, а вызываются в процессе работы программы из уже существующей библиотеки. Если программа использует DLL, то сама библиотека требуется на этапе использования, но не используется на этапе сборки (в отличие от других библиотек).

Способы подключения DLL

1. **На этапе загрузки программы** (load-time, linking) - статическое

Используемые из DLL процедуры и функции объявлены как внешние в соответствующей DLL. На этапе запуска проги загрузчик проверяет наличие необходимых DLL и процедур в них. Если их нет – ошибка загрузки и невозможно запустить прогу.

const testlib=’testlib.dll’

procedure proc1(x,y:integer); external testlib name ‘proc’; Pascal

1. **На этапе выполнения** (run-time linking) - динамическое

Используется в случаях: прогрмамме требуются ресурсы из DLL; ф-ии из DLL нужны лишь на коротком этапе работы; использование ф-ий из DLL необязательно.

hLib:=LoadLibrary(‘MyLib.dll’);

FreeLibrary(hLib); Если библиотеку не удается загрузить, то hLib=0. Доступ к процедуре или функции осуществляется по адресу.

@p:=GetProcAdress(hLib, ‘Proc1’);

Вызов осуществляется с использованием процедурной переменной

var p: procedure(x,y:integer); stdcall;

**Модули вызова dll**

stdcall, cdecl, pascal, register

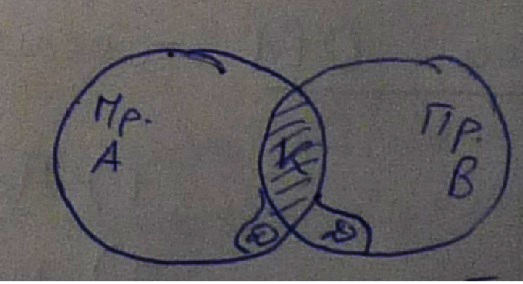
**Синтаксис исходного текста DLL**

**library** <имя>; *// вместо Program или Unit*  
**uses** <обычный список используемых модулей>;

<Объявление процедур и функций>

**exports**  
 Proc1 **index** 10,  
 Proc2 **name** 'ProcedureTwo',  
 Proc3,  
 .....;  
**begin**  
 <возможные начальные установки, напр. - для глоб. переменных>  
**end**.

Во фразе **exports** через запятую перечисляются идентификаторы предназначенных для вызова внешними программами, процедур. Фраза **exports** может встречаться в тексте сколько угодно раз и перемежаться с объявлением процедур, единственное условие – процедура должна сначала объявляться, а затем экспортироваться.

Глобальные переменные DLL, в отличие от модуля (Unit), не могут быть экспортированы. Кроме того, если DLL используется в нескольких процессах, то для каждого процесса создается отдельная копия глобальных переменных DLL, и поэтому глобальные переменные DLL в принципе не могут быть использованы для передачи данных между процессами даже в случае доступа к ним через функции самой DLL.

**12. Организация памяти в Windows. Виртуальная память. Виртуальные адресные пространства 32-разрядных процессов, структура адресного пространства. Достоинства и недостатки используемой организации памяти.**

Для 32/64-разрядных приложений Windows обеспечивает:

- многопоточность

- плоскую (не сегментированную) модель виртуальной памяти;

- организацию для каждого процесса изолированного адресного пространства;

- выполнение всех потоков в режиме вытесняющей многозадачности;

- поддержку средств синхронизации потоков – критических секций, семафоров, мутексов, событий;

- обмен данными между процессами

- система приоритетов

- приоритеты абсолютные и динамические

- виртуальная память со страничной организацией.

Сегментированная модель памяти.

Память разделяется на блоки (сегменты). Размер сегмента произвольный. С каждым сегментом связан селектор. При обращении к памяти указывается селектор и смещение внутри сегмента. Селектор содержит информацию о начале сегмента, правах доступа, присутствии сегмента в памяти и т.д. Механизм вычисления физ. адреса происходи аппаратно. Указатель (дальний): в нем выделены селектор и смещение. Ближний – смещение. Особенности: фрагментирование памяти при выгрузке сегментов на диск (разный размер сегментов); трудности в обработке больших объемов данных.

Альтернатива – линейная/плоская модель памяти. Указатель – единый адрес, не разбиваемый на сегмент и смещение. Для выгрузки информации на диск используется страничный механизм. Страница – блоки памяти по 4Кбайта. Особенности: удобно обрабатывать большие данные; страничный механизм решает проблему фрагментации памяти.

Проблема фрагментации памяти:

Сборка мусора – объединение свободных участков памяти в единый блок. Проблема – невозможность двигать занятые участки, т.к. приложения имеют указатели на эти участки. Решение – приложению выдается хэндл занятого участка, при обращении к блоку, ОС преобразует хэндл в адрес.

Виртуальная память предназначена для решения проблемы с нехваткой оперативной памяти. При обращении к ВП в машинных командах используются виртуальные адреса. Часть адресов соответствует адресам в ОЗУ. Другая – блоки данных на диске. При выполнении обращения по виртуальным адресам, обращаются к ОЗУ, если блока там нет, то он из диска перемещается в ОЗУ, затем происходит повторное обращение к ОЗУ. Механизм по мимо расширения объема оперативной памяти (за счет использования других видов памяти. Используя виртуальные адреса.) обеспечивает:

1.Организацию множества независимого адресного пространства (защита блока памяти от вмешательства параллельного процесса)

2.Рациональное использование ОЗУ

Недостатки сегментной модели

1.Сложные конструкции для адреса если размер блока данных> max размера сегмента (16разр.сист-64Кб 32разр сист-4Гб)

Фрагментация памяти при выгрузке на диск.

Плюсы линейной модели.

+ Адрес любой ячейки –целое число без разделений.

+Адрес следующей ячейки +1.

+Проще работать с большими данными.

+Страничный механизм позволяет решить проблемы с фрагментацией.

+Изолирование адресного пространство каждого процесса.

**13. Многозадачность и проблема организации обмена информацией между приложениями. Обзор средств Windows для межпроцессного обмена информацией.**

**Многозадачность** – это способность ОС параллельно (псевдопараллельно) обеспечивать обработку нескольких процессов.

**Многопоточность** – свойство приложения, что процесс, порожденный в ОС, может состоять из нескольких потоков, выполняемых параллельно.

**Потоки выполнения** (treads) – процедуры, выполняемые параллельно в рамках одного процесса. У них

* Общее адресное пространство (в рамках одного процесса)
* Общие глобальные переменные, хэндлы открытых ресурсов
* Свой стек и контекст у каждого потока

В современных ОС процессам выделяется изолированные адресные пространства. Т.е. адрес в адресном пр-ве 1 процесса, в другом приведет к другому участку памяти. Один процесс не может выставить запрос ячейки другого процесса. Передать данные от одного процесса к другому не так-то просто.

При реализации собственного протокола обмена сообщениями необходимо учитывать, что Windows "согласится" доставлять только сообщения, коды которых лежат в диапазоне C000h .. FFFFh.

Хотя программист может "волевым решением" установить, что его программы будут пользоваться сообщениями, допустим, с кодом D000h и E000h, но никто не гарантирует, что какие-то программы параллельно не посылают друг другу сообщения с такими кодами. Поэтому Windows в виде функции RegisterWindowMessage предоставляет возможность регистрировать сообщения по уникальному строковому идентификатору. Вероятность того, что различные группы приложений используют одни и те же строковые идентификаторы сообщений можно сделать достаточно низкой, если использовать в идентификаторах, например, название фирмы-производителя программного продукта.

Например, приложение может выполнить команду:

idMSG1:= RegisterWindowMessage('Borland\_Data\_exchange\_message');

При этом функция возвращает значение, однозначно соответствующее указанной строке, т.е. если другое приложение вызовет эту же функцию с такой же строкой в качестве параметра, то будет возвращено то же число, а для другой строки число гарантированно будет другим. Если по каким-то причинам зарегистрировать сообщение не удается (например, слишком длинная строка, хотя в документации такие ограничения не упомянуты), то возвращается 0, иначе – число в диапазоне C000h .. FFFFh. Зарегистрированные сообщения остаются зарегистрированными до завершения сеанса работы в Windows (выключения или перезагрузки компьютера).

Windows позволяет отправить любое сообщение окну системы, достаточно знать хэндл окна. Отправлять можно

1. Часто для начального установления связи одно из приложений широковещательно рассылает сообщение-приветствие, в котором указывается хэндл окна-отправителя. Широковещательная рассылка достигается при указании в первом параметре SendMessage константы HWND\_BROADCAST вместо хэндла окна. При получении приветствия (такое сообщение должно быть зарегистрировано как инициатором связи, так и предполагаемым партнером) другое приложение отвечает сообщением-подтверждением, в котором также указывается хэндл окна, которому следует направлять дальнейшие сообщения в процессе связи.

Например, два приложения зарегистрировали одни и те же сообщения:

msg1:= RegisterWindowMessage('Запрос');

msg2:= RegisterWindowMessage('Ответ');

msg3:= RegisterWindowMessage('Данные');

Затем первое приложение рассылает сообщение-запрос:

SendMessage(HWND\_BROADCAST, msg1, hClientWindow,0);

При получении такого сообщения (его получат все окна первого плана, т.е. не имеющие окон-собственников, всех прилохений) второе приложение запоминает хэндл запрашивавшего окна (в нашем случае он передавался в wParam) и отвечает, указывая хэндл своего окна, например так:

hOtherWindow:=wparam;

PostMessage(wparam, msg2, hServerWindow, 0);

Приняв такое сообщение, окно hClientWindow первого приложения готово обмениваться сообщениями с окном hServerWindow второго приложения, причем в каждом приложении известен хэндл окна-партнера из другого приложения.

hOtherWindow:=wparam;

И теперь в любом приложении может быть послано сообщение:

PostMessage(hOtherWindow,msg3, 10,20);

где hOtherWindow – хэндл окна-партнера, сохраненный при установлении связи.

При двустороннем обмене сообщениями всегда предпочтительно пользоваться функцией PostMessage (если нет особых причин использовать SendMessage), так как при этом исключается рекурсивный вызов оконной процедуры. Дело в том, что если в ответ на SendMessage одного приложения второе приложение пошлет ответ через SendMessage, то оконная процедура в первом приложении запустится еще раз (рекурсивно), хотя предыдущий код, посылавший SendMessage, еще не завершен. Это вполне допустимо, однако при плохом понимании программистом особенностей рекурсии способствует появлению логических ошибок в программе.

2. *ГЛОБАЛЬНЫЕ АТОМЫ*

Атом – специальным образом определенная строка с числовым идентификатором. Зная идентификатор можно запросить саму строку. В сообщение можно передать 8 байт информации (полезной). Отправитель регистрирует глобальный атом-строку доступную все процессам ОС, а получатель извлекает её. Размер глобальной таблицы атомов ограничен, следовательно, атомы после передачи должны быть удалены. Можно передать строки до 255байт.

3.*WM\_COPYDATA* Сообщение, позволяющее передать произвольные блоки данных (до 25б). В этом сообщении посылается указатель на некий блок данных, действительно только во время обработки этого сообщения. Запоминать указатель нет смысла. Нужно обработать или скопировать.

4. *Отображение файлов.* В памяти создание общей области разделяемой памяти для нескольких процессов. Для работы нужно FileMapping - отображающий объект и FileView - образ файла.

Использовать, когда процессы обмениваются большим количеством информации. Если несколько приложений отображают на памяти один и тот же файл, то для этого используется один и тот же адрес виртуальной памяти.

Действия А (создать: файл, FM, FV)

Действия B (открыть FM, создать FV)

Получить хэндл если оно уже создано

hMapping:=OpenFileMapping(0,false,’имя объекта’)

if Mapping:=0

HMapping:+CreateFileMappimg(-1,0,PAGE\_READ\_WRITE,’имя объекта’)

pData:=MapViewofFile(hMapping,FILE\_MAP\_ ALLACCESS,0,0,0//доступ ко всему отображению можно осуществлять по частям);

move(s,pData,256)

5. *БУФФЕР ОБМЕНА* (clipBoard) Информация заноситься только по команде пользователя. Пригоден для обмена информации между приложениями.

OprnClipBoard

EmptyClipBoard(очистить)

CloseClipBoard

RegisterClipBoard

**14. Многозадачность и проблема синхронизации. Обзор и характеристика средств синхронизации, предоставляемых Windows.**

**Многозадачность** – это способность ОС параллельно (псевдопараллельно) обеспечивать обработку нескольких процессов.

**Многопоточность** – свойство приложения, что процесс, порожденный в ОС, может состоять из нескольких потоков, выполняемых параллельно.

**Потоки выполнения** (treads) – процедуры, выполняемые параллельно в рамках одного процесса. У них

* Общее адресное пространство (в рамках одного процесса)
* Общие глобальные переменные, хэндлы открытых ресурсов
* Свой стек и контекст у каждого потока

Средства синхронизации предотвращают одновременный доступ к разделенным ресурсам (все то что нужно программе во время её работы)

1. **КРИТИЧЕСКИЕ** СЕКЦИИ –для синхронизации потоков в рамках одного процесса. Чтобы воспользоваться критической секцией, программа должна создать экземпляр переменной типа TRTLCriticalSection (захватив под него память из кучи или просто создав глобальную переменную этого типа):

var section: TRTLCriticalSection;

Информация внутри структуры должна быть предварительно проинициализирована с помощью вызова функции InitializeCriticalSection(section);

Далее потоки могут пытаться занять критическую секцию с помощью вызовов EnterCriticalSection(section) или if TryEnterCriticalSection(section) then ...

Первая функция просто реализует захват критической секции вызывающим ее потоком, при занятости КС поток переводится в состояние ожидания и ждет освобождения КС.

Вторая функция вместо перехода в состояние ожидания возвращает логическое значение FALSE, если критическая секция занята (и входит в КС с возвратом TRUE, если КС свободна).

После завершения действий, выполняемых в КС, поток должен освободить КС, вызвав LeaveCriticalSection(section);

Критических секций в программе может быть сколько угодно.

2. **Функции ожидания**

**procedure** Sleep(milliseconds:integer); – Эта функция переводит вызывающий ее поток в состояние ожидания на указанное количество миллисекунд (32-разрядное целое). При указании в качестве параметра 0 происходит просто досрочная передача управления другим потокам, находящимся в состоянии готовности. Поток, переведенный в состояние ожидания с помощью этой функции, не может быть возобновлен до истечения указанного времени.

3. **СЕМАФОР** – системный объект с которым связано целое неотрицательное состояние. Семафор считается взведенным (не требует ожидания), когда его значение отлично от нуля. Если значение семафора равно нулю, то обращение к функции ожидания с указанием его хэндла приводит к переводу потока в состояние ожидания. При любом обращении к функции ожидания с указанием хэндла взведенного семафора его (семафора) значение автоматически уменьшается.

Пример создания семафора:

**hSema:=CreateSemaphore(**

**nil,** // атрибуты защиты

**1,** // начальное значение

**1,** // максимальное значение

**'MySemaphore'** // имя семафора

**);**

**WaitForSingleObject** - Считается взведённым (не требует ожидание) если его состояние > 0, то декремент и вход в КС, или ждем ”>0”

**hSema:=OpenSemaphore(SEMAPHORE\_ALL\_ACSESSS\_TRUE,’Имя\_семафора’);** //если такой семафор существует мы получим его хэндл даже если он существует в каком то другом процессе, и получим 0 если его нет.

Release Semaphore (hSema,+1(к состоянию),nil(атрибуты защиты));

Что бы уничтожить семафор, все кто открывал его хэндлы должны их закрыть **CloseHandle(hSema)**

4. **СОБЫТИЯ(Event) –** системный объект, находящийся в логических состояниях 1 –взведен 0-сборошен.

События можно вызывать/сбрасывать в явном виде: Set/ResetEvent(hEvent). В составе автоматической КС должны и автоматически сбрасываться. Создаются и уничтожаются аналогично семафорам

hEvent:=CreateEvent(nil(защита),False(авто\_сброс), true(начальное\_событие), ’имя’)

hEvent:=OpenEvent(…)

5. **MUTEX** – системный объект, который может принадлежать(захвачен) потоку или не принадлежать (свободен) вообще. Осиротевший мутекс = свободному. Функция ожидания пытается захватить мутекс в рамках выполняемого потока. Если он захвачен, то поток ожидает, или захватывает и выполняет действие КС.

hMu:=CreateMutex()

hMu:=OpenMutex()

КС WaitFor SingleObjects(hMu)

RealseMutex(hMu);

CloseHandle(hM);

**15. Различные способы отслеживания временных интервалов в Windows-программе. Достоинства и недостатки.**

**Таймер.** Программа может запросить у системы получение сообщений WM\_TIMER. Для этого нужно создать логический таймер. Механизм этот не очень точен, не надежен. Сообщение WM\_TIMER имеет низкий приоритет. Если программа не успевает обработать сообщения, то сообщение теряется.

Стратегии работы с таймером

* Обработка сообщений WM\_TIMER в оконной процедуре
* Вызвать специальную функцию, обрабатывающую WM\_TIMER

GetTickCount //функция возвращающая время работы компьютера

**Высокочастотный таймер.** Если необходимо отслеживать интервалы можно воспользоваться доступом к счетчику по системной плате

C1,2-показания счетчика

QueryPerformanceController(var c:int64);

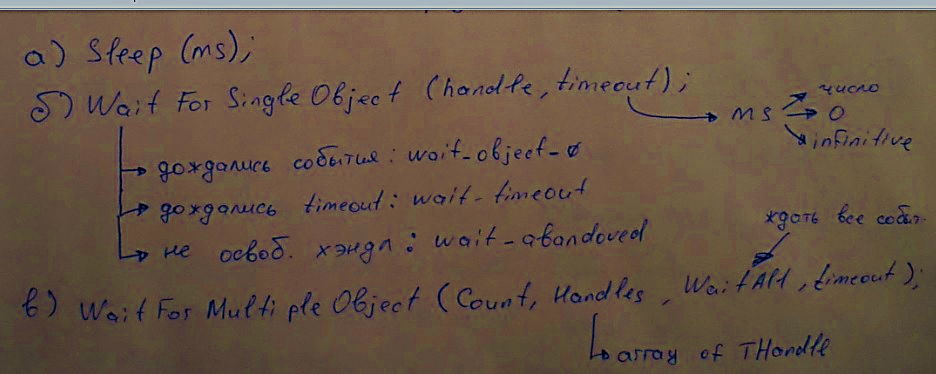
QueryPerformance[Frequency](https://www.google.ru/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CBwQFjAA&url=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FFrequency&ei=AJezVNisCuWgyAPJuYKYDQ&usg=AFQjCNESrz41bmQHnkKuc6yVco45Ntd7pQ&bvm=bv.83339334,d.bGQ)**(**var f:int64);

Функции ожидания позволяют перевести поток в режим ожидания на какое-то время или при наступлении какого-то события (нескольких).

Поток в состоянии ожидание:

1.не обрабатывает сообщения

2.его окна не обрабатывают сообщения, посланные через SendMsg



GetTickCount T=Tc1-Tc2 //количество мс прошедших со старта системы

**16. Модели памяти для 16- и 32-разрядных приложений Windows. Проблема фрагментации памяти.**

Сегментированная память разделяет блоки памяти на сегменты, каждому сегменту селектор, в котором уровень доступа, базовый адрес и т.п. При обращении к памяти всегда явно или неявно задаётся селектор и смещение.

Механизм вычисления адреса реализуется аппаратно. Размер сегментов разные, ограниченные разрядностью смещения.

Существует понятие ближнего и дальнего указателя. Дальний (селектор и смещение). Ближний только смещение.

Т.к. сегменты разные по размеру существует опасность сегментирования при выгрузке на диск. У коротких сегментов трудности при обработке больших объемов памяти.

Для 32 разрядных приложений используется плоская (линейная) модель-адрес выражается числом, не разделяясь на сегменты смещения. Адрес последующей ячейки +1. Все указатели ближние. При использовании линейной модели памяти применяется страничный механизм.

+отсутствие сегментации (упрощает написание алгоритмов с большими массивами)

Страничный механизм нет проблем с фрагментацией (т.к. все страницы одного размера).

Проблемы с фрагментацией памяти

Возникает при любом динамическом высвобождении памяти. Блоки свободные и занятые будут хаотично расположены (т.к. имеют разные размеры)

Приложениям всегда требуется непрерывные свободные блоки памяти

Возможна ситуация, когда размер sum памяти превышает, а свободного непрерывного блока нет. Перемещать нельзя блоки памяти, т.к. приложение использует для доступа к данным указатели. Доступ к блокам доложен осуществляется через хэндлы, а не указатели.

hMem:=GlobalAlloc(Gmem\_Moveable,1000000) //выделение перемещ.блока

Функция для создания указателя для доступа к памяти по хэндлу.

pData:=GlobalLock(hMem) //Запрещает перемещение и дает указатель

GlobalUnlock(hMem) //Разрешает перемещение и делает указатель недействительным

GlomalDiscard(hMem) //Освобождение блока памяти

Функции языков программирования для выделения\освобождения памяти работают с уже выделенными блоками памяти определенного разряда. При нехватке памяти могут вызвать системные функции WinApi для выделения\освобождения памяти.

**17. Особенности планирования потоков в Windows. Приоритеты процессов и потоков, обработка межпоточных сообщений.**

В рамках одного процесса Windows позволяет создаться множество потоков в режиме вытесняющей многозадачности. Когда запускается процесс порождается единственный главный поток, остальные создаются самостоятельно и исполняются параллельно главному. Для каждого потока своя очередь сообщений. Если поток ожидает события, то его надо перевести в состояние ожидания. Что бы это сделать нужно объявить функцию потока. Она будет запущенна параллельно основной. В режиме вытесняющей многозадачности

**Function** MeTread1(data:pointer):integer;stdcall//Загружаем целый указатель

var

begin   
 result :code//перед завершением

ExitThread(code)//выход из потока

Вызов потока

Htread:=CreateTread(nil,0,@MyTread1,nil,0,id); id – идент. потока, возвращаемое значение

TerminateTread(hTread); [\\ указатель](file:///\\указатель) терминала потока

Прерывание потока

1. Логическое завершение, процедура дошла до конца

2. Внутри процедуры ExitTread

3. TerminateTread – жесткое прерывание потока (вне потока). Не произойдет освобождение ресурсов. При завершении процессов все хэндлы должны автоматически закрываться CloseHandle. Убедиться, что поток завершился GetExitCodTread(HTread). Перевести в активный режим ResumeTread. Режим ожидания – SuspendThread. Режим ожидания WaitForObject(htread;infinite) – возвращает событие случилось, если поток завершился.

Передача сообщений между потоками

Очередь для каждого потока одна. Создаётся, когда поток вызывает ф-ии из user или GDI. Через очередь следующее сообщение адресуется окнам этим потоков. Если поток создает окна он должен иметь свой собственный цикл выборки\обработки сообщений SendMessage (если поток отправит *своему* собственному окну с его помощью – прямой выход *другому,* буферизируется и создает еще один высокий приоритет очереди из вне)

Если поток в состоянии ожидания, то он не отправляет свою очередь и все, кто шлют ему SendMessage тоже стоят. Не рекомендуется использовать эту функцию между потоками. В Windows действует абсолютный приоритеты, которые могут быть динамически изменены.

Потоки выполняются вытесняющим способом и действует система абсолютных приоритетов.

Приоритет потоков зависит от

1.Приоритета процесса (ProcessPriorthClass)

2.Приоритет потока TradPriority

Приоритет 0-31: 0-15-пользовательские, 16-31 realtime. Имеются 6 классов приоритета процессов и больше потоков

Если поток не получает управления в течении 5 секунд, т.к. у него малый приоритет то Windows повышает его до 15.

Планировщик – выделяет время. Система планировщика – отслеживает время TimeCawing

Нужно организовывать много потоков

1.Если программа требует длительного вычислений. Вычисления выносятся в другой поток. Программа в главном

2.Многоцессорная или многоядерная система если позволяет разрядность нескольких процессов, зависит от алгоритма

3.Когда необходимо ожидать готовность ресурса. (Поступать данных из низкоскоростного порта Windows)

**18. Инверсия приоритетов.**

Поток L имеет низкий приоритет, поток M — средний, поток H — высокий. Поток L захватывает mutex (объект, который принадлежит потоку или не принадлежит никому ABANDONED), и, выполняясь с удержанием mutex’а, прерывается потоком H, который пробудился по какой-то причине, и имеет более высокий приоритет. Поток H пытается захватить mutex.

В полученной ситуации поток H ожидает завершения текущей работы потоком M, ибо, пока поток M исполняется, низкоприоритетный поток L не получает управления и не может освободить mutex.

Устраняется повышением приоритета всех нитей, захватывающих данный mutex, до одного и того же высокого значения на период удержания mutexa. Некоторые реализации mutex’ов делают это автоматически.

**21. Особенности прохождения и обработки сообщения WM\_PAINT. Различные способы получения контекста устройства отображения.**

Сообщение WM\_PAINT посылается окну, когда необходимо отрисовать что-либо в клиентской области.

В параметрах (wparam) иногда передается контекст устройства, но обычно он = 0.

Сообщение посылается буферизированным способом и имеет низкий приоритет. Извлекается из очереди, когда других сообщений нет.

Функция GetMessage не удаляет его из очереди. Оно удаляется из очереди, когда в окне перестает существовать недействительный регион и помещается в очередь, когда такой регион появляется.

На самом деле WM\_PAINT в очередь не помещается никогда, но, если очередь пуста при вызове GetMessage и недействительном регионе функция, как бы вызывает это сообщение.

Оно не должно посылаться программам с помощью Send\Post Message. Вместо этого мы вручную делаем окно недействительным InvalidateReсt или ValidateReсt.

Обработка WM\_PAINT

var ps:TPaintStruct;

…

hdc:=BeginPaint(hwnd, ps);

//рисование

EndPaint(hwnd, ps); {освобождение контекста}

Описание структуры TPaintStruct

TPaintStruct: record

hdc: integer;

fErase: Boolean; {тру – необходимо стереть фон}

rcPaint: TRect; {координаты прямоугольника отсечения}

…

end;

Прямоугольник отсечения – мин прямоугольник, описанный вокруг региона отсечения. Анализируя его координаты, можно оптимизировать рисование (не рисовать части изображения, лежащие вне этого прямоугольника).

Действия ф-ии BeginPaint:

- настраивает один из свободных контекстов на работу с окном, возвращает хэндл этого контекста

- регион отсечения устанавливается по границам недействительного региона, а недействительный регион объявляется действительным, WM\_PAINT как-бы удаляется из очереди

Для перерисовки окна требуется объявить его клиентскую область недействительной с помощью ф-ии InvalidateRect.

Если необходимо немедленно перерисовать окно – UpdateWindow.

EndPaint – освобождение контекста устройства, разрывает его связь с окном.

Различные способы получения контекста устройства отображения:

Получение – hdc:=GetDC(hwnd); - выставляет регион отсечения по границам видимого региона

Освобождение – ReleaseDC(hwnd,hdc).

Для рисования не в клиентской области:

hdc:=GetWindowDC(hwnd);

…

ReleaseDC(hwnd,hdc);

Получение контекста экрана – hdc:=GetDC(0);

Получение контекста любого устройства кроме окна – hdc:=CreateDC(‘DISPLAY’,nil,nil,nil);

DISPLAY – получение контекста экрана

Удаление DeleteDC(hdc);

Получение характеристик контекста – GetDeviceCaps(hdc,index)

**23. Системы координат устройства отображения. Режимы отображения. Примеры работы с координатными системами.**

Ф-ии GDI работают в логической системе координат, шаг корд. сетки м не совпадать с пиксельной сеткой (по умолчанию совпадает)

З системы координат:

- абсолютная/физическая: нач. коорд. – левый верхний угол, у вниз, х вправо, шаг по осям 1 пиксель. В реальности закрашиваются точки, GDI не работает, только USER (отвечают за положение окна на экране).

- логическая: GDI работают. Шаг по осям произвольный, направление осей произвольное, нач. коорд. произвольное.

- относительная физическая: нач. коорд. произвольное, направление осей произвольное, шаг – 1 пиксель (никакие ф-ии не работают, нужна при прокрутке больших изображений в клиентской зоне, для этого достаточно изменить положение нач. коорд.)

Управление логической СК с помощью системы констант:

xViewOrg, yViewOrg – положение нк относительной физической ск в абсолютной физической ск.

xWinOrg, yWinOrg – положение нк относительной физической ск в логической ск.

xViewExt/xWinExt – коэф-т шага сетки логической ск по оси х

уViewExt/уWinExt – коэф-т шага сетки логической ск по оси у

Формулы перехода:

Физическая СК => Логическая СК: DPtoLP(hdc,rect//массив точек//, count);

LPtoDP(-//-);

GetClientRect(hwnd, rect)

Ф-ии управления СК:

SetViewportExtEx(hdc,xExt,yExt,size:TPoint);  устанавливает горизонтальные и вертикальные размеры области просмотра для контекста устройства при помощи использования заданных значений.

SetViewportOrgEx  определяет, какая точка устройства отображает начало координат окна (0,0).

SetWindowExtEx устанавливает горизонтальные и вертикальные размеры окна для контекста устройства при помощи использования указанных значений.

SetWindowOrgEx определяет, какая точка окна отображает начало координат области просмотра (0,0).

Режимы отображения:

Устанавливаются ф-ей SetMapMode(hdc,index);

index={MM\_TEXT – шаг по осям 1 пиксель, отсальное можно менять; MM\_LOENGLISH – шаг=0.1 доли дюйма; MM\_HIENGLISH – шаг=0.1 доли дюйма; MM\_LOMETRIC – шаг=0.1 мм; MM\_HIMETRIC – шаг=0.01 мм; MM\_ISOTROPIC – нк можно менять, шаг по осям по х и у дб одинаковы; MM\_ANISOTROPIC – всё можно менять}

Пример: MM\_ISOTROPIC w

SetMapMode(hdc, MM\_ISOTROPIC);

SetWindowExtEx(hdc,512,512,nil); h

SetViewportExtEx(hdc, w, h, nil);

SetViewportOrgEx((hdc, w div 2, h div 2, nil);

MoveToEx(hdc, -128, -128, nil);

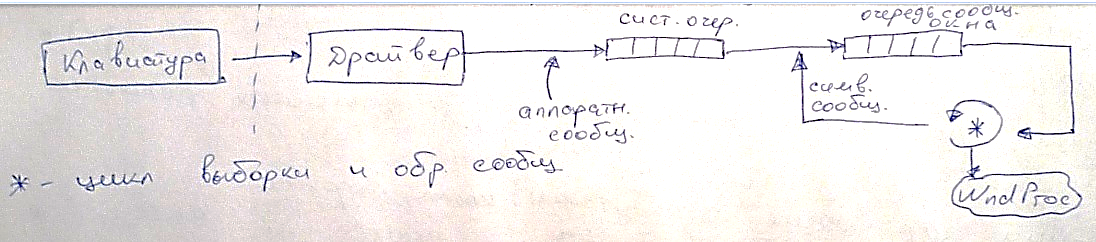
LineTo(hdc, 128,128);

**24. Сообщения от клавиатуры. Параметры, алгоритмы прохождения и обработки. Примеры работы с клавиатурой.**

Пользователь нажимает клавишу, активное окно, имеющее фокус ввода, получает сообщение, содержащее информацию о нажатой клавише. То же происходит и при отпускании клавиши. Эти сообщения идут через очередь и делятся на:

Аппаратные - виртуальный код нажатой клавиши, идентифицируют нажатую кнопку, посылаются при нажатии всех клавиш. Содержат scan-cod клавиши и виртуальный код клавиши, определенный в Windows.

Символьные - при нажатии алфавитно-цифровых клавиш и содержит информацию о наборном символе с учетом текущего регистра и языковой раскладки. Символьные сообщения программа посылает сама себе посредством ф-ии TranslateMessage. Если приходит апп. сообщение, то при его обработке в TranslateMessage формируется символьное сообщение.



В очередь потока сообщения поступают по одному.

При нажатии клавиши окну посылается сообщение WM\_KEYDOWN или WM\_SYSKEYDOWN, при отпускании — WM\_KEYUP или WM\_SYSKEYUP. Сообщения, в идентификаторах которых фигурирует слово "SYS"=клавиша + Alt.

Все перечисленные аппаратные сообщения от клавиатуры в параметре WPARAM имеют виртуальный код клавиши Windows. Эти коды имеют идентификаторы, начинающиеся с VK\_

В LPARAM содержится дополнительная информация о нажатии клавиши, структура которой такова:

|  |  |  |
| --- | --- | --- |
| Биты | Размер | Значение |
| 0..15 | 16 | Счетчик повторений при удержании |
| 16..23 | 8 | Скэн-код |
| 24 | 1 | Флаг принадлежности расширенной клавиатуре |
| 29 | 1 | Флаг контекста. "0" для обычных и "1" для системных сообщений. |
| 30 | 1 | Предыдущее состояние клавиши. "1" - была нажата, "0" - была отпущена. |
| 31 | 1 | Состояние клавиши. "1" - нажата (KEYDOWN), "0" - отпущена (KEYUP). |

Флаг принадлежности к расширенной клавиатуре выставляется в единицу, если нажаты такие клавиши, как правый Alt, Ctrl или Shift, "серые" клавиши со знаками арифметических операций и Enter, NumLock, а также клавиши управления курсором, не относящиеся к клавиатуре цифрового набора.

Флаг контекста показывает, нажата ли клавиша Alt в момент посылки сообщения. Когда активное окно минимизировано, оно получает все клавиатурные сообщения как системные, и чтобы определить, нажата ли действительно клавиша Alt, используется данный флаг.

**Символьные сообщения клавиатуры**

После обработки аппаратного сообщения в TranslateMessage в очередь могут быть помещены символьные сообщения WM\_CHAR, WM\_SYSCHAR, WM\_DEADCHAR и WM\_SYSDEADCHAR.

Символьные сообщения содержат код нажатого символа в WPARAM, а в LPARAM содержится та же информация, что и у породившего их аппаратного сообщения, в том числе — количество повторов символа.

DEADCHAR: окно получает два символьных сообщения: WM\_DEADCHAR с символом диакритического знака и, при последующем нажатии алфавитной клавиши, WM\_CHAR с символом с диакритическим знаком, например символом Ó . Если введенный после немого символа алфавитный символ не может иметь диакритического знака, окно получит не два, а три сообщения: WM\_DEADCHAR и WM\_CHAR с диакритическим знаком и WM\_CHAR с буквой алфавита.

**Состояние клавиатуры**

Возвращают не текущее состояние клавиатуры в момент их вызова, а состояние клавиатуры на момент посылки последнего извлеченного из очереди клавиатурного сообщения.

**function** GetKeyState(nVirtKey: Integer): SmallInt;

Функция GetKeyState возвращает логическое состояние клавиши с соответствующим виртуальным кодом. Старший бит 16-разрядного результата взведен, если клавиша нажата, и сброшен, если клавиша отпущена. Младший бит для клавиш управления режимом (CapsLock, NumLock, ScrollLock) показывает, активен соответствующий режим ("1") или нет ("0").

логическое состояние клавиатуры целиком. Возвращаемое логическое значение свидетельствует об успехе или неуспехе производимой операции.

**type** TKeyboardState = **array**[0..255] **of** Byte;

**function** GetKeyboardState(**var** KeyState: TKeyboardState): BOOL;

**function** SetKeyboardState(**var** KeyState: TKeyboardState): BOOL;

Старший бит соответствующего виртуальному коду клавиши байта взведен, когда клавиша нажата, и сброшен, когда отпущена. Младший бит по-прежнему показывает режим для клавиш управления режимом.

Для определения физического, т.е. аппаратного состояния клавиши в текущий момент служит функция GetAsyncKeyState с параметрами, аналогичными GetKeyState. Взведенный старший бит возвращаемого значения свидетельствует о том, что клавиша нажата, младший – что клавиша была нажата с момента предыдущего вызова GetAsyncKeyState из данного потока. Если фокус клавиатурного ввода в настоящий момент принадлежит окну другого потока, то функция возвратит нулевое значение.

**25. Сообщения от мыши. Параметры, алгоритмы прохождения и обработки. Примеры работы с мышью.**

**Мышь**. (левая, правая, колесо).

Если пользователь производит манипуляции с мышью, то ОС посылает сообщения окну, над которым находится указатель мыши.

Общая логика прохождения сообщения

* Мышиные сообщения посылаются тому в поле, которого находиться курсор. Сообщение о нажатии мыши м не сопровождаться сообщением об отпускании.
* (очереди из сообщений мыши нет). В очереди находится только одно сообщение от мыши. Некоторые м пропадать.
* Если в поле окна происходят манипуляции с мышью, окну всегда посылается WM\_NCHITTEST. Цель – спросить у окна, где у нее сейчас курсор. Обработчик должен вернуть код зоны окна, в к-ой находится курсор. Положение курсора задается в параметре lParam так же, как и в остальных сообщениях от мыши; параметр wParam не используется.
* Существует возможность отследить отпускание кнопки мыши, даже если курсор за пределами окна.

SETCapture(hWnd) – после этого окно продолжает получать сообщения от мыши пока кнопка нажата, даже если курсор за пределами окна.

WM\_LBUTTONDOWSN:SETCapture(hWnd);//что бы установить захват

При отпускании кнопки REALEASECapture;//снять захват.

С событиями от мыши связано 21 сообщение Windows. Если курсор мыши находится в рабочей области окна, окно получает следующие сообщения:

Движение курсора: WM\_MOUSEMOVE

Нажатие кнопок:

|  |  |  |  |
| --- | --- | --- | --- |
| **Кнопка** | **Нажатие** | **Отпускание** | **Двойное (второе) нажатие\*** |
| Левая | WM\_LBUTTONDOWN | WM\_LBUTTONUP | WM\_LBUTTONDBLCLK |
| Правая | WM\_RBUTTONDOWN | WM\_RBUTTONUP | WM\_RBUTTONDBLCLK |
| Средняя | WM\_MBUTTONDOWN | WM\_MBUTTONUP | WM\_MBUTTONDBLCLK |

\* Сообщения о двойных щелчках окна получают лишь в том случае, когда в стиле их оконного класса указан флаг CS\_DBLCLKS.

wParam - состояние кнопок, а также клавиш Ctrl и Shift клавиатуры; OR-комбинация констант MK\_LBUTTON, MK\_RBUTTON, MK\_MBUTTON, MK\_SHIFT и MK\_CONTROL.

lParam - координаты курсора в пикселах относительно левого верхнего угла рабочей области, X=loWord(lParam); Y=hiWord(lParam);

**Сообщения мыши нерабочей области**

Окно получает извещение о действиях с мышью и тогда, когда курсор находится на нерабочей области окна — на заголовке или на рамке. Названия этих сообщений имеют в своем составе аббревиатуру NC (non-client)

wParam - обозначает зону окна. lParam - как и для сообщений рабочей области — координаты курсора, но в координатной системе всего экрана, а не рабочей области окна.

**Пример 1 .Захват мыши**

Wm\_ButtonDown:begin

……

SetCapture(hWnd)

End;

WM\_LButtonUp:begin

…..

ReleaseCapture;

End;

**Пример 2**

WM\_NChittest:begin

Result:=DeFWindowProc(hwnd,msg,wparam,lparam);

If result HTClient then

Result:=HTCaption;

End;

Таким образом система будет считать клиентскую область заголовком, т.е окно-заголовок

**26. Организация логического таймера. Особенности прохождения и использования сообщения WM\_TIMER. Примеры программирования. Высокоточный таймер.**

**Таймер**. Программа может запросить у системы получение сообщений WM\_TIMER. Для этого нужно создать логический таймер. Сообщение WM\_TIMER имеет низкий приоритет. Если программа не успевает обработать сообщения, то сообщение теряется (мб только одно в очереди).

1) Создать таймер – вкл

2) Уничтожить таймер – выкл

3) Старый уничтожить, новый создать – изменить параметры.

Период логического таймера задается в миллисекундах и может лежать в интервале от 1 до $FFFFFFFF (что соответствует приблизительно 50 суткам). Однако для отсчета периода логических таймеров Windows использует аппаратный таймер компьютера, который срабатывает с периодом приблизительно в 55 мсек. Из этого следует, во-первых, что задание периода таймера менее 55 мсек. бессмысленно, так как программа в этом случае все равно будет получать одно сообщение таймера каждые 55 мсек. Во-вторых, задаваемый период логического таймера всегда округляется вниз до ближайшего значения, соответствующего целому количеству срабатываний аппаратного таймера.

*SetTimer(Hwnd(хэндл таймера),ID(идентификатор таймера в программе),T(период мс),lpTimerProc(необходимый указатель на процедуру реакции таймера мб nil);*

SetTimer возвращает числовой идентификатор таймера или 0, если по каким-либо причинам таймер не может быть создан.

*killTimer(hwnd,ID)*

Сообщение WM\_TIMER имеет следующие параметры:

wParam - числовой идентификатор таймера.

lParam - указатель на функцию обработки, если таковая имеется.

две возможности по обработке сообщений таймера: в оконной процедуре и в специальной процедуре обработки. В первом случае в качестве lpTimerFunc в SetTimer передается nil, и сообщение WM\_TIMER должно обрабатываться наравне со всеми остальными в общем операторе CASE оконной процедуры. Во втором случае в программе должна быть описана процедура со следующим набором параметров (имя — произвольное):

**procedure** TimerProc(hwnd:THandle; uMsg, idEvent, time: integer); **stdcall**;

Указатель на эту процедуру (@TimerProc) передается в качестве lpTimerFunc в SetTimer, и при каждом получении сообщения WM\_TIMER процедура DispatchMessage будет вызывать не оконную процедуру, а процедуру TimerProc. Параметры этой процедуры — хэндл окна, тип сообщения (всегда WM\_TIMER), числовой идентификатор таймера и системное время в момент получения сообщения.

Таймер для окна

*SetTimer(hwnd,2,100,nil);*

*killTimer(hwnd,3)*

Таймер для процедуры(всей программы)

*nID:=Settimer(0,0,1000,@TimerProc)*

*killTimer(0,nID);*

Пример

case msg of

*WM\_TIMEr: invalidateRect(hwnd,nil,true);//перерисовать окно*

Высокочастотный таймер. Позволяет запросить показания счетчика, считывающего импульсы высокой частоты. Если необходимо отслеживать интервалы можно воспользоваться доступом к счетчику по системной плате

C1,2-показания счетчика

### *QueryPerformanceConter(var c:int64);//показания счетчика в импульсах, тиках*

### *QueryPerformance*[*Frequency*](https://www.google.ru/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CBwQFjAA&url=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FFrequency&ei=AJezVNisCuWgyAPJuYKYDQ&usg=AFQjCNESrz41bmQHnkKuc6yVco45Ntd7pQ&bvm=bv.83339334,d.bGQ)*(var f:int64);//показания частоты*

### 27. Дочерние окна управления класса button (кнопка) и связанные с ними сообщения. Принципы и примеры использования.

Создание кнопки - создание дочернего окна со стилем BS\_

|  |  |
| --- | --- |
| **Стиль** | **Характеристика** |
| BS\_PUSHBUTTON | Обычная кнопка ("нажимаемая кнопка"). |
| BS\_DEFPUSBUTTON | Кнопка с более жирной рамкой. |
| BS\_CHECKBOX |  |
| BS\_AUTOCHECKBOX |  |
| BS\_3STATE | То же, что флажок, но возможно также "третье состояние", когда поле флажка отображается серым цветом. |
| BS\_AUTO3STATE |  |
| BS\_RADIOBUTTON |  |
| BS\_AUTORADIOBUTTON |  |
| BS\_GROUPBOX | Рамка с текстом в верхнем левом углу, обычно служит для визуального объединения радио-кнопок в группу. Само по себе не реагирует на события от клавиатуры и мыши. |
| BS\_OWNERDRAW | Кнопка пользователя, отрисовкой внешнего вида кнопки должна заниматься оконная процедура родительского окна, обрабатывая сообщение WM\_DRAWITEM. |

**Пример**

WM\_create :begin

CreateWindow(‘button’,’r1’,bs\_autoradiodibutton or ws\_tabstop or ws\_child…, idr1, …);

CreateWindow(‘button’,’r2’,тоже только нет ws\_tabstop);

CreateWindow(‘button’,’r3’,ws\_group or ws\_tabstop;

CreateWindow(‘button’,’r4’, без ws\_tabstop..);

CreateWindow(‘button’,’r5’,без ws\_tabstop..);

Hwndbuttoneclose:=CreateWindow(‘button’,’закрыть’,bs\_defpushbutton or ws\_tabstop и.т.д);

В диалоговое окно посылаются WM\_COMMAND.

При нажатии кнопок в сообщении WM\_COMMAND содержится код уведомления BN\_CLICKED (старший бит wparam) и id (младший бит wparam).

Родительское окно может посылать окнам кнопок управляющие сообщения:

|  |  |
| --- | --- |
| **Сообщение** | **Описание** |
| BM\_GETCHECK | Возвращает для флажков и индикаторов выбора состояние выбора: 0 (не помечено), 1 (помечено), 2 (третье состояние). |
| BM\_SETCHECK | Устанавливает состояние выбора для флажков и индикаторов выбора. В wParam должен содержаться код состояния (см. выше). |
| BM\_GETSTATE | Возвращает состояние как комбинацию битовых флагов:  биты 0 и 1 — содержат код, возвращаемый BM\_GETCHECK;  бит 2 — нажата (1) или отпущена (0) кнопка; бит 3 — имеет ли кнопка фокус. |
| BM\_SETSTATE | Для нажимаемых кнопок устанавливает состояние. wParam = 0 — кнопка отпущена, wParam = 1 — кнопка нажата. |
| BM\_SETSTYLE | Позволяет изменить стиль кнопки после ее создания. wParam - содержать новый стиль как комбинация битовых флагов, lParam - 1 - перерисовать кнопку, 0 - не перерисовывать. |

Например, изменить состояние окна-флажка с хэндлом hCheckBox1 на противоположное можно с помощью следующей конструкции:

SendMessage(hCheckBox1, BM\_SETCHECK,

SendMessage(hCheckBox1, BM\_GETCHECK,0,0) xor 1,

0);

**28. Дочерние окна управления класса edit (редактор) и связанные с ними сообщения. Принципы и примеры использования.**

одно- или многострочные текстовые редакторы. Могут иметь клавиатурный фокус и позволяют вводить текст, редактировать его при помощи клавиш управления, выделять при помощи клавиш и мыши и производить операции копирования и вставки с буфером обмена. Могут иметь собственные стандартные полосы прокрутки(включение флагов WS\_VSCROLL и WS\_HSCROLL в стиль окна редактирования).

Окно редактирования создается как экземпляр класса edit. В стиле флаги:

|  |  |
| --- | --- |
| ES\_MULTILINE | Многострочный редактор. |
| ES\_AUTOHSCROLL | Автоматически используется горизонтальная полоса прокрутки, если текст не умещается по ширине окна. При отсутствии этого флага производится автоматический перенос на следующую строку. |
| ES\_AUTOVSCROLL | Автоматически появляется вертикальная полоса прокрутки, если текст не умещается по высоте окна. |
| ES\_NOHIDESEL | Выделенный текст подсвечивается, даже когда окно теряет фокус, иначе при потере фокуса подсветка текста гасится. |

Перемещение фокуса по клавише Tab - необходимо выполнить дополнительные требования :

1. Оконная процедура для обработки сообщений по умолчанию должна использовать DefDlgProc вместо DefWindowProc.

2. Вследствие этого класс окна-диалога, содержащее органы управления, должен быть порожден с указанием параметра cbWindowsExtra больше или равного DLGWINDOWEXTRA = 30.

3. Т.к. DefDlgProc по умолчанию ничего не делает в ответ на сообщение WM\_CLOSE, то для обеспечения возможности закрытия окна диалога его оконная процедура должна в явном виде обрабатывать это сообщение, например, вызывая DestroyWindow(hwnd), как это делает DefWindowProc.

С помощью функции SetWindowText можно задать текст в окне редактирования, с помощью GetWindowtextLength и GetWindowText — получить текст. Многострочные окна редактирования воспринимают последовательность #13#10 как конец строки. Окна редактирования не обрабатывают символ табуляции (#9) и отображают его в виде вертикальной черты ("|"). Чтобы в обычном окне (не в диалоге) обрабатывать нажатия клавиш Tab, Shift-Tab, Enter как управляющие, необходимо вводить новую оконную процедуру для окон редактирования (см. ниже). Размер редактируемого текста ограничен примерно 32 килобайтами, при этом используется область памяти, выделенная приложению, содержащему окно редактора.

Окна редактора посылают родительскому окну сообщения WM\_COMMAND со следующими кодами уведомления в старшем слове wParam:

|  |  |
| --- | --- |
| EN\_SETFOCUS | Редактор получил фокус ввода |
| EN\_KILLFOCUS | Редактор потерял фокус ввода |
| EN\_CHANGE | Содержимое изменилось, посылается после вывода на экран |
| EN\_UPDATE | Содержимое изменилось, посылается перед обновлением экрана |
| EN\_ERRSPACE | Буфер редактирования переполнился |
| EN\_MAXTEXT | Буфер редактирования переполнился при вставке |

Программа может управлять окном редактора, посылая ему сообщения:

EM\_GETLINECOUNT — возвращает число строк в многострочном редакторе.

EM\_LINEINDEX, wParam = номер строки — возвращает смещение начала строки относительно начала буфера редактирования. Строки нумеруются с нуля. wParam=-1 позволяет узнать смещение строки, содержащей курсор.

EM\_LINELENGTH, wParam = номер строки — возвращает длину указанной строки.

EM\_GETLINE, wParam = номер строки, lParam = указатель на буфер — копирует указанную строку в заданный буфер, заканчивая ее нулем.

См. также сообщения WM\_CUT, WM\_COPY, WM\_PASTE, EM\_GETSEL, EM\_SETSEL, EM\_REPLACESEL, с помощью которых реализуются операции с буфером обмена.

**Пример 1.Замена первой буквы А в поле edit при люб внес изм в него.**

Var buffer:array[0..30]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

WM\_COMMAND:BEGIN

case Hiword(wparam)of

EN\_CHANGE:

Begin

GETWINDOWTEXT(GetDlgItem(hwnd,edit),buffer,30);

Buffer[0]:=’A’;

SetWindowText(GetDlgItem(hwnd,edit)buffer);

End;

**29. Дочерние окна управления класса listbox (список) и связанные с ними сообщения. Принципы и примеры использования.**

Список — набор текстовых строк, которые выводятся в прямоугольном прокручиваемом окне в один/несколько столбцов. Окно списка м получать фокус и способно обрабатывать сообщения клавиатуры. Когда окно списка имеет фокус, одна из строк отображается с пунктирной рамкой и считается имеющей фокус. Строки мб выделены цветом, в этом случае они считаются выбранными.

По умолчанию списки не посылают родительскому окну сообщения WM\_COMMAND; чтобы посылка извещений происходила, необходимо включать в стиль окна списка флаг LBS\_NOTIFY. Другими важными флагами стиля являются:

|  |  |
| --- | --- |
| LBS\_SORT | Строки списка автоматически сортируются |
| LBS\_MULTIPLESEL | Позволяет выделять несколько строк списка (список с множественным выбором) |

Для того, чтобы список имел рамку и полосу прокрутки, необходимо включить в стиль окна списка флаги WS\_BORDER и WS\_VSCROLL. (+ WS\_CAPTION, WS\_SIZEBOX, WS\_HSCROLL)

Для управления списком - сообщения, посылаемые окну списка при помощи функции SendMessage:

LB\_ADDSTRING, lParam = указатель на Z-строку — добавляет строку в конец несортированного списка, в сортированном списке строка оказывается на нужном месте в соответствии с алгоритмом сортировки.

LB\_INSERTSTRING, wParam = позиция, lParam = указатель на Z-строку — вставляет строку в указанной позиции, все остальные строки сдвигаются. Если список сортированный, пересортировка не производится. При wParam=-1 строка добавляется в конец списка.

LB\_DELETSTRING, wParam = позиция — удаляет строку в заданной позиции.

LB\_RESETCONTENT — полная очистка списка.

WM\_SETREDRAW, wParam = 0 - разрешить, 1 - запретить — устанавливает флаг, разрешающий или запрещающий перерисовку списка при изменении его содержимого. При последовательном добавлении или удалении нескольких строк полезно сбросить этот флаг перед началом операции и взвести по окончании.

LB\_GETCOUNT — возвращает количество элементов в списке.

LB\_SETCURSEL, wParam = номер строки — в списке с одиночным выбором делает указанную строку выбранной, при wParam=-1 ни одна строка не выбрана.

LB\_SELECTSTRING, wParam = номер первой строки, lParam = указатель на Z-строку для поиска — делает выбранной первую подходящую строку, начиная с wParam. Если wParam=-1, то поиск начинается с начала списка. Выбирается строка, начинающаяся со строки, задаваемой lParam. При невозможности найти подходящую строку возвращается LB\_ERR.

LB\_GETCURSEL — возвращает номер текущей выбранной строки или -1.

LB\_GETTEXTLEN, wParam = номер строки — возвращает длину указанной строки.

LB\_GETTEXT, wParam = номер строки, lParam = адрес буфера — записывает в буфер заданную строку списка, завершая ее нулем.

LB\_SETSEL, wParam = 0 - снять выделение, 1 - включить выделение, lParam - номер строки — в списках с множественным выбором выделяет или снимает выделение указанного элемента списка, если lParam=-1, то выделяется или снимается выделение у всех элементов списка.

LB\_GETSEL, wParam = номер строки — возвращает 0, если строка не выбрана, и другое значение, если выбрана.

Окна списков в свою очередь извещают родительские окна о действиях с ними путем посылки им сообщений WM\_COMMAND со следующими кодами уведомления в старшем слове wParam:

|  |  |
| --- | --- |
| LBN\_ERRSPACE | При добавлении строки не хватило памяти. |
| LBN\_SELCHANGE | Изменен текущий выбор. |
| LBN\_DBLCLICK | Был двойной щелчок мышью по списку. |
| LBN\_SETFOCUS | Список получил фокус. |
| LBN\_KILLFOCUS | Список потерял фокус. |

**30. Дочерние окна управления класса scrollbar (полоса прокрутки) и связанные с ними сообщения. Принципы и примеры использования полос прокрутки.**

Полоса прокрутки мб помещена в произвольном месте окна. Для этого необходимо создать ее как дочернее окно управления класса scrollbar, указав стиль SBS\_VERT или SBS\_HORZ. Полоса прокрутки при этом занимает весь указанный в параметрах CreateWindow (или MoveWindow) прямоугольник. Системные настройки ширины стандартных оконных полос прокрутки можно узнать, вызвав функции GetSystemMetrix(SM\_CYHSCROLL) или GetSystemMetrix(SM\_CXVSCROLL).

Обязательно следует устанавливать диапазон, так как для таких полос прокрутки по умолчанию принимается MIN=MAX=0. Оконные полосы: 0..100

Окна класса scrollbar, как и стандартные полосы прокрутки окна, посылают родительскому окну сообщения WM\_VSCROLL или WM\_HSCROLL вместо сообщений WM\_COMMAND, посылаемых остальными оконными органами управления. Wparam=position бегунка|операция. Отличия этих сообщений от сообщений, посылаемых оконными полосами прокрутки, состоят в том, что для оконных полос прокрутки lParam содержит 0, а для дочерних окон - полос прокрутки — хэндл такого дочернего окна.

Управление полосой прокрутки осуществляется путем посылки сообщений SBM\_SetPos. + ф-ии GetScrollPos, GetScrollRange, SetScrollPos и SetScrollRange, что и для стандартных оконных полос прокрутки, однако в параметре hWnd следует указывать хэндл самой полосы прокрутки, а в nBar — константу SB\_CTL

В отличие от стандартных оконных полос прокрутки, окна класса scrollbar могут получать фокус и обрабатывать нажатия клавиш клавиатуры.

|  |  |
| --- | --- |
| Клавиша | wParam сообщения VM\_xSCROLL |
| Home | SB\_TOP |
| End | SB\_BOTTOM |
| Page Up | SB\_PAGEUP |
| PageDown | SB\_PAGEDOWN |
| Стрека влево, стрелка вверх | SB\_LINEUP |
| Стрелка вправо, стрелка вниз | SB\_LINEDOWN |

**32. Синтаксис описания ресурсов и примеры программирования. Использование битовых образов, значков, курсоров мыши, строк и ресурсов произвольного формата.**

Файлы ресурсов — это бинарные файлы, имеющие определенный формат; для создания и редактирования ресурсов используется компилятор ресурсов (например в Delphi — brc32.exe), позволяющий на основе текстового описания ресурсов (обычно это фалы с расширением .RC) построить бинарный файл .RES.

**Курсоры, значки и битовые образы**

Битовый образ — это изображение в формате BMP. Можно использовать изображения любого размера и глубины цвета.

Значок (icon) — это растровое изображение размером 32х32 или 16х16 пикселов, монохромное или 16-цветное (в современных версиях Windows поддерживается и большая глубина цвета). В дополнение к возможным 2 или 16 цветам, пикселы значка могут быть прозрачными и инвертирующими. Значки хранятся в файлах с расширением .ICO.

Курсор мыши — это монохромное/цветное растровое изображение размером 32х32 пиксела. Вершина (hotspot) в виде пары координат относительно левого верхнего угла битового образа курсора. Курсоры хранятся в виде файлов с расширением .CUR.

При использовании компилятора ресурсов каждый значок, битовый образ или курсор обычно описываются в виде ссылки на соответствующий файл ресурса в формате

<идентификатор> <ТИП> <файл>

например:

picture1 BITMAP pic1.bmp

hand CURSOR hand.cur

appicon ICON icon1.ico

picture1, hand и appicon — идентификаторы битового образа, курсора мыши и значка соответственно. Вместо строковых идентификаторов при описании ресурсов могут использоваться целые положительные 16-разрядные числа. Кроме того, описание битового образа ресурса может располагаться не в отдельном файле, а в файле .RC в виде шестнадцатеричного дампа памяти.

Для использования ресурсов в программе их необходимо загрузить. Битовые образы загружаются с помощью функции

**function** LoadBitmap(hInstance: HINST; lpBitmapName: PAnsiChar): HBITMAP;

Возвращает хэндл битового образа или 0 при неудаче. Первый параметр - хэндл экземпляра приложения, из исполнимого файла которого производится загрузка ресурса. Второй параметр - указатель на строковой идентификатор битового образа, либо его числовой идентификатор, если в файле ресурса образ описывалась с номером. Windows определяет, передан указатель на строку или числовой идентификатор, проверяя старшее слово переданного указателя: если передано 16-битное число, то старшее слово нулевое, указатель на область данных программы всегда имеет ненулевое старшее слово.

После использования каждый загруженный с помощью LoadBitmap битовый образ должен быть уничтожен с помощью функции DeleteObject.

Для загрузки значков используется функция

**function** LoadIcon(hInstance: HINST; lpIconName: PChar): HICON;

В отличие от функции LoadBitmap, если указанный значок уже был ранее загружен, вторая его копия не будет создаваться и будет возвращен хэндл уже загруженного значка. Значки не нуждаются в уничтожении.

Возможность отобразить значок в контексте устройства с помощью функции DrawIcon(hdc,x,y,hIcon);

Курсоры загружаются при помощи функции **function** LoadCursor(hInstance: HINST; lpCursorName: PAnsiChar): HCURSOR; - аналогична функции LoadIcon. Хэндл курсора может использоваться при описании классов окон. Кроме того, при обработке сообщения WM\_MOUSEMOVE имеется возможность анализировать положение мыши и менять форму курсора при помощи функции SetCursor(hCursor); - Если эту функцию не вызывать, то будет использован курсор, определенный в оконном классе.

Пусть имеется файл описания ресурсов, содержащий описание:

123 BITMAP pic1.bmp

ic1 ICON icon1.ico

Тогда соответствующие битовый образ и значок должны быть загружены при помощи вызовов:

hbmp := LoadBitmap(hInstance, pointer(123) );

hicon := LoadIcon(hInstance, 'ic1');

....................... операции ................

DeleteObject(hbmp);

Идентификаторы должны быть уникальными для однотипных ресурсов.

**Таблицы строк символов**

Использование этого вида ресурсов - перевод с английского на русский язык.

Символьные строки организуются в таблицы и объявляются с использованием ключевого слова STRINGTABLE следующим образом:

STRINGTABLE

{

<id1>, "<string1>"

.....................

}

STRINGTABLE

{

1, "Ошибка времени выполнения"

2, "Недопустимая операция"

10, "Нормальное завершение"

}

При работе с Delphi: константы, идентифицирующие символьные строки, могут быть описаны в отдельном файле в соответствии с синтаксисом языка Паскаль в виде секции **const,** и затем включены в исходный текст программы с помощью директивы компилятора $INCLUDE ($I), а в файл .RC — с помощью #include. Можно также объявить модуль (unit) языка Паскаль, интерфейсная часть которого состоит из объявлений констант, и подключить его к программе в разделе **uses**, а к описанию ресурсов — с помощью #include.

————— константы ресурсов: RESCONST.PAS —————

**unit** resconst;

**interface**

**const**

IDS\_Runerror = 1;

IDS\_InvalidOp = 2;

IDS\_Normalterm = 10;

**implementation end**.

———————— описание ресурсов —————————

#include "resconst.pas"

STRINGTABLE

{

IDS\_RUNERROR, "Ошибка времени выполнения"

IDS\_INVALIDOP, "Недопустимая операция"

IDS\_NORMALTERM, "Нормальное завершение"

}

—————————————————————————————————————

Строки загружаются с помощью функции LoadString в готовый буфер:

**function** LoadString(hInst: HINST; uID: UINT; lpBuffer: PChar; nBufferMax: Integer): Integer;

Здесь hInst — хэндл экземпляра приложения, из исполнимого файла которого загружается строка, uID — целочисленный идентификатор строки, lpBuffer — указатель на буфер для приема данных, nBufferMax — размер буфера. Если загружаемая строка превышает размер буфера, она будет усечена. Функция возвращает количество прочитанных символов. Загружаемая строка, даже усеченная, всегда заканчивается нулевым символом.

**Ресурсы пользователя**

Имеется возможность сохранить в виде ресурса и подключить к исполняемому модулю произвольные данные. Для этого в файле описания ресурсов должна быть ссылка на файл, содержащий подключаемые данные, например:

data1 MYDATA example1.dat

В качестве идентификатора и типа данных может быть указана произвольная строка. Загрузить такие данные можно при помощи вызова

hData := LoadResource (hInstance, FindResource(hInstance, 'data1', 'MYDATA') );

Для работы с самими данными необходим не хэндл, а указатель, что выглядит примерно так:

pData := LockResource(hData);

*{действия с pData}*

FreeResource(hData);

LoadResource, FindResource, LockResource и FreeResource

**33. Синтаксис описания ресурсов и примеры программирования. Использование меню и таблиц акселераторов из ресурсов.**

Простейшим способом включения меню в интерфейс Windows-программы является описание его в файле ресурсов и последующая его загрузка оттуда. Меню, как и дочерние окна управления, при различных действиях пользователя посылает программе сообщения WM\_COMMAND.

виды меню: (main menu), (drop-down menu), (popup menu)

Пункты меню мб "разрешены" (enabled), "запрещены" (disabled) или "недоступны" (grayed — серые). При выборе пользователем разрешенных пунктов меню происходят действия (открывается всплывающее меню или программе посылается сообщение WM\_COMMAND). Запрещенные и недоступные пункты меню также м выбрать, но никаких действий не происходит. Запрещенные пункты меню выглядят так же, как разрешенные.

Описание меню в файле ресурсов производится согласно следующему шаблону:

*<имя\_меню>* **MENU**

**{***[<список элементов меню>]***}**

<имя\_меню> — это строка или число, служащее для идентификации меню в программе при описании класса окна. Список элементов меню состоит из описаний пунктов меню или всплывающих меню.

Пункт меню (которому не соответствует подменю) описывается как

**MENUITEM "***<текст>***",** *<идентификатор>* *[***,** *<признаки>]*

Если пункту меню соответствует подменю, то оно описывается как

**POPUP "***<текст>***"** *[***,** *<признаки>]*

{*[<список элементов меню>]*}

<Текст> - строка, которая отображается в описываемом пункте. В текст мб включен знак "&"; символ текста, следующий за амперсантом, будет отображаться подчеркнутым и при нажатии его на клавиатуре совместно с клавишей Alt будет выбираться этот пункт меню.

<идентификатор> — число, передаваемое в программу в параметре сообщения WM\_COMMAND при выборе этого пункта меню. Удобно использовать внешние определения идентификаторов, как это было при обсуждении строковых таблиц, и подключать их к описанию ресурсов директивой #include. Имена констант для идентификаторов меню принято начинать с символов "IDM\_", например IDM\_EDIT.

Признаков элементов главного меню - флаги, которые можно объединять с помощью символа дизъюнкции ("ИЛИ") языка С ("|" — "трубопровод"):

GRAYED - пункт недоступен и выводится серым.

INACTIVE - пункт неактивен (не генерирует WM\_COMMAND), но отображается обычным образом, несовместимо с GRAYED.

MENUBREAK - следующие пункты меню отображаются с новой строки.

HELP - этот и все следующие пункты меню "прижимаются" к правому краю строки меню.

В описании всплывающего меню могут встречаться как подменю, так и пункты меню, описываемые точно так же с помощью директив POPUP и MENUITEM. Кроме того, внутри всплывающих меню можно нарисовать горизонтальную черту, включив в качестве описания строки меню строку **MENUITEM SEPARATOR**

Может дополнительно использоваться признак CHECKED, означающий, что пункт меню выбран и отмечен галочкой.

**Использование меню**

wndclass.lpszMenuName := 'MyMenu'; имя, как оно описано в ресурсе

или, если при описании меню использовался числовой идентификатор, например, 12:

wndclass.lpszMenuName := pointer(12);

или

wndclass.lpszMenuName := '#12';

Все окна этого класса будут иметь указанное меню. Чтобы указать свое меню для конкретного окна при его создании, необходимо загрузить описание меню из ресурса:

hMenu := LoadMenu(hInstance, 'MyMenu');

и затем передать полученный хэндл меню hMenu в качестве девятого параметра функции CreateWindow.

Наконец, имея хэндл загруженного описания меню, можно изменить меню уже существующего окна, вызвав функцию

SetMenu(hwnd, hMenu);

а просто узнать хэндл меню окна — с помощью функции

GetMenu(hWnd);

Любое связанное с окном меню уничтожается при уничтожении окна. Однако загруженные, но не связанные с окнами меню должны быть явно удалены при помощи DestroyMenu(hMenu).

**Таблица акселераторов**

В этой таблице задаются комбинации клавиш, нажатие которых приводит к генерации сообщений WM\_COMMAND с тем или иным кодом.

В файле описания ресурсов таблицы акселераторов определяются следующим образом:

*<имя\_таблицы>* **ACCELERATORS**

**{***[определения комбинаций клавиш]***}**

В качестве имени таблицы можно указывать символьный или числовой идентификатор, как и для меню. Определения клавиш — это строки одного из четырех возможных форматов:

**"***<символ>***",** *<числовой\_идентификатор>**[***, SHIFT***] [***, CONTROL***] [,* **ALT***]*

**"^***<символ>***",** *<числовой\_идентификатор>* *[***, SHIFT***] [***, CONTROL***] [,* **ALT***]*

*<ASCII-код>***,** *<числовой\_идентификатор>***, ASCII** *[***, SHIFT***] [***, CONTROL***] [***, ALT***]*

*<виртуальный\_код>***,** *<числовой\_идентификатор>***, VIRTKEY** *[***, SHIFT***] [***, CONTROL***] [***, ALT***]*

Числовой идентификатор быстрой клавиши передается в параметрах сообщения WM\_COMMAND. Слова SHIFT, CONTROL и ALT - указанные регистровые клавиши образуют комбинацию с клавишей, код которой указывается в первом параметре.

Первый способ - что в первом параметре указан символ, соответствующий алфавитно-цифровой клавише, второй — что это комбинация символа с клавишей Ctrl, в третьем случае первым параметром должен быть числовой код буквы, в последнем случае — виртуальный код клавиши Windows.

**Использование таблицы акселераторов**

Для использования описанных в файле ресурсов комбинаций клавиш их необходимо загрузить, а также несколько модифицировать цикл выборки сообщений.

Загрузка таблицы акселераторов и получение ее хэндла осуществляется с помощью функции LoadAccelerators:

hAccel := LoadAccelerators(hInstance, 'MyAccel');

Цикл обработки сообщений дополняется вызовом функции проверки комбинаций клавиш по таблице:

**while** getMessage(msg, 0,0,0) **do**

**if not** TranslateAccelerator(hwnd, hAccel, msg) **then begin**

TranslateMessage(msg);

DispatchMessage(msg);

**end**;

TranslateAccelerator проверяет, является ли сообщение msg сообщением клавиатуры и, если так, проводит поиск комбинации нажатых клавиш в таблице акселераторов. Если комбинация найдена, вызывается оконная процедура окна hwnd и ей передается сообщение WM\_COMMAND с кодом, идентифицирующим нажатую комбинацию, функция при этом возвращает TRUE. Для сообщений, не приводящих к посылке сообщения WM\_COMMAND при трансляции, функция возвращает FALSE.

Параметр hwnd функции TranslateAccelerator сосредоточивает обработку акселераторов в одной оконной процедуре: какое бы окно не являлось получателем сообщения msg (хэндл окна получателя содержится в теле сообщения), сообщение WM\_COMMAND будет передано в оконную процедуру принудительно указанного окна hwnd. В качестве hwnd программа обычно должна передавать либо хэндл главного окна, либо хэндл ее активного окна.

Сообщение WM\_COMMAND, порождаемое при нажатии быстрой клавиши, имеет следующие параметры: loword(wParam) - числовой идентификатор комбинации клавиш из таблицы акселераторов; hiword(wparam) = 1; lParam = 0;

Если акселератор соответствует тому или иному пункту меню (имеет тот же идентификатор), то помимо сообщения WM\_COMMAND той же оконной процедуре посылаются сообщения активизации меню, т.е. WM\_INITMENU, WM\_INITMENUPOPUP, WM\_MENUSELECT. Если клавиша соответствует запрещенному пункту меню, то сообщения WM\_COMMAND и прочие не посылаются

**34. Синтаксис описания ресурсов и примеры программирования. Использование диалогов из ресурсов.**

Ресурсы позволяют описать вид контролов внутри диал. окна. Диалоговые окна содержат дочерние окна управления (кнопки, поля ввода текста, радио-кнопки и кнопки-флажки), в описании вида и положения которых и состоит описание диалога.

Необходимо процедуру диалога. Принципиальное отличие между диалоговой процедурой и обычной оконной процедурой - диалоговой процедуре передаются не все сообщения, которые получает оконная процедура.

**function** DlgProc(hDlg: HWND; Msg: UINT;   
 wParam: WPARAM; lParam: LPARAM): **longint**; **stdcall**;

Функция обрабатывает сообщение - возвращает ненулевое значение, иначе — 0. Сообщения, которые не обрабатываются, не должны передаваться ни в DefWindowProc, ни в DefDlgProc, т.к. их обработка сосредоточена в оконной процедуре диалога. Вызов DefDlgProc внутри диалоговой процедуры - рекурсия (оттуда снова будет вызвана диалоговая процедура). Процедура диалога обрабатывает два сообщения: WM\_INITDIALOG (посылается диалоговому окну при его инициализации), и WM\_COMMAND — сообщения от дочерних окон управления.

Если в ответ на WM\_INITDIALOG процедура диалога возвращает TRUE, то Windows дает фокус первому из элементов управления диалога (его хэндл передается в wParam); при инициализации диалога можно изначально дать фокус другому элементу управления с помощью функции SetFocus, при этом обработчик WM\_INITDIALOG должен вернуть FALSE.

При попытке использовать SetFocus проблема: т.к. диалоговое окно создается Windows, то в программе неизвестны хэндлы дочерних органов управления. Эту информацию можно получить, зная числовой идентификатор органа управления, задаваемый в файле ресурса, при помощи функции **function** GetDlgItem(hDlg: THandle; id: integer): THandle;

Сообщения WM\_COMMAND от дочерних окон управления…

Для посылки сообщений органам управления в API реализована функция

**function** SendDlgItemMessage(hDlg: HWND; nIDDlgItem: Integer;  
 Msg: UINT; wParam: WPARAM; lParam: LPARAM): Longint;

работающая аналогично SendMessage, при этом получателем сообщения является дочернее окно диалога hDlg с идентификатором nIDDlgItem (при этом не требуется знать хэндл этого органа управления).

В окнах диалога Windows автоматически поддерживает интерфейс клавиатуры, позволяющий перемещать фокус от одного дочернего окна управления к другому при помощи клавиши Tab и клавиш управления курсором.

Диалог загружается и активизируется при помощи функции DialogBox. Возврат из этой функции не производится, пока в ответ на какое-либо событие в диалоговой процедуре не будет вызвана функция EndDialog, т.е. функция DialogBox активизирует модальный диалог.

**function** DialogBox(hInstance: HINST; lpTemplate: PChar;  
 hWndParent: HWND; lpDialogFunc: pointer): Integer;

lpTemplate — Z-строка, содержащая имя ресурса диалога, hWndParent — хэндл окна-собственника диалога (обычно — главного окна программы, хотя можно указать 0), lpDialogFunction — указатель на процедуру диалога. Возвращаемое значение определяется программистом при вызове EndDialog внутри процедуры диалога.

**function** EndDialog(hDlg: HWND; nResult: Integer): BOOL;

hDlg — хэндл окна диалога (передается в процедуру диалога в качестве параметра), nResult — код завершения диалога, возвращаемый в вызывающую диалог программу. Определены некоторые стандартные коды, например ID\_OK, ID\_CANCEL, ID\_ABORT, ID\_IGNORE, ID\_YES, ID\_NO, ID\_CLOSE. Диалог должен завершаться явным вызовом EndDialog в ответ на нажатие некоторой кнопки (обычно это Ok и Cancel), а также при получении диалоговой процедурой сообщения WM\_CLOSE.

<имя> DIALOG <x>,<y>,<w>,<h>

[STYLE <оконные стили>]

[CAPTION «заголовок»]

{[<описание контрола>]

}

<тип контрола> <id> [, “текст”], <x>, <y>,<w>,<h>[,<стиль>]

def push button 101, “OK”, 100,50,100,20,ws\_tabstop

listbox 105, 200,50,100,400,ws\_tabstop

DialogBox(hInstance,'DIALOG1',0,@DlgProc);

DIALOG1 – имя в ресурсах

**35. Организация обмена информацией между процессами при помощи сообщений. Примеры программирования.**

**Сообщение WM\_COPYDATA**

Это сообщение позволяет передать блок информации от одного процесса другому. При этом в процессе обработки этого сообщения, и только в это время, передаваемый блок данных становится "видимым" из адресного пространства процесса-получателя этого сообщения и доступным для чтения. В документации к Windows сказано, что процесс-получатель не имеет права модифицировать информацию внутри блока, и он должен скопировать нужную информацию в самостоятельно выделенную область памяти, если эта информация нужна вне обработчика сообщения WM\_COPYDATA.

Сообщение WM\_COPYDATA всегда должно посылаться небуферизованным способом с помощью функции SendMessage. Отправлять его с помощью PostMessage недопустимо. Чтобы воспользоваться этим сообщением, нужно знать хэндл окна-получателя, относящегося к другому процессу, поэтому процессы предварительно должны "установить соединение" – например, с помощью широковещательной рассылки сообщений.

wParam - хэндл окна, отправившего данные; в принципе здесь может фигурировать хэндл любого окна процесса-отправителя.

lParam - указатель на структуру следующего вида:

TCopyDataStruct = **packed record**

dwData: integer;

cbData: dword;

lpData: pointer;

**end**;

dwData – произвольное 32-разрядное значение, lpData – указатель на передаваемый блок данных, cbData – размер блока, адресуемого lpData.

Процесс-отправитель должен заполнить поля этой структуры, установив в lpData адрес блока данных, которые необходимо передать в другой процесс. Windows перед вызовом обработчика этого сообщения в другом процессе произведет необходимые действия по переносу данных в адресное пространство получателя, и в обработчике поле lpData окажется указывающим на блок передаваемых данных уже в адресном пространстве получателя. Этот указатель действителен только во время обработки сообщения, т.е. пока не завершилась функция SendMessage в процессе-отправителе.

С помощью этого сообщения можно передавать данные между любыми 16- и 32-разрядными приложениями.

**36. Организация обмена информацией между процессами при помощи отображения файлов на память.**

Пользовательская программа просит ОС отобразить определенный дисковый файл в некоторый диапазон адресов своего виртуального адресного пространства, при этом создается особый управляющий объект, называемый отображением файла (file mapping). После этого при чтении и записи по адресам памяти, на которые отображен файл, программа читает и записывает содержимое соответствующего дискового файла. Это достигается за счет того, что страницы памяти, соответствующие адресам отображения, выгружаются ОС не в системный файл подкачки, а в указанный файл; программа на самом деле читает и пишет в страницы памяти, в которые было считано содержимое файла при создании отображения и откуда информация будет возвращена в файл после закрытия отображения.

Сам по себе такой способ работы с файлом может иногда оказаться полезен, однако основное применение возможность отображения файлов в память находит не по прямому назначению (для работы с файлами), а для обмена информацией между приложениями. Это связано с тем, что один и тот же файл могут отобразить в память несколько приложений одновременно. При этом операционная система обеспечивает, чтобы изменения, сделанные в отображенном файле одним процессом, были отражены в образе файла, которую "видят" другие процессы. Фактически это достигается за счет того, что для отображения одного и того же файла в разных процессах используется одна и та же разделяемая физическая область памяти. При этом виртуальные адреса отображений в разных процессах могут не совпадать.

Необходимы два объекта ОС: отображающий объект (file mapping) и образ файла (file view). М рассматривать образ файла как диапазон адресов виртуального адресного пространства процесса, используемых для работы с файлом, а отображающий объект – как страницы физической памяти, связанные с файлом. В связи с этим отображающий объект создается один раз в одном из взаимодействующих процессов, а образы файла – в каждом процессе с использованием существующего отображающего объекта.

Для создания отображающего объекта нужен хэндл открытого файла. Обычно файл специально создается для этой цели с помощью функции CreateFile, при этом желательно, чтобы создающий процесс создавал файл с эксклюзивным доступом (другие процессы не могли бы его открыть/испортить). Возвращает она хэндл файла:

hFile := CreateFile(  
 'myfile.map', // имя файла  
 GENERIC\_READ or GENERIC\_WRITE, // доступ на чтение-запись  
 0, // эксклюзивный доступ  
 0, // атрибуты защиты, не поддерживается Windows 95, только NT  
 CREATE\_ALLWAYS, // создавать заново  
 FILE\_ATTRIBUTE\_NORMAL);

Затем может быть создан отображающий объект:

hMapping := CreateFileMapping(  
 hFile, // хэндл открытого файла  
 0, // атрибуты защиты, не поддерживается Windows 95, только NT  
 PAGE\_READ\_WRITE,  
 0, 4096, // 4096 байт, если 0 - размер равен размеру файла  
 'MyMapping' // имя-идентификатор объекта  
 );

После этого может быть создан образ файла в адресном пространстве процесса, функция возвращает указатель на начало блока памяти размером, в нашем случае, 4096 байт:

pData := MapViewOfFile(  
 hMapping, // хэндл отображающего объекта  
 FILE\_MAP\_ALL\_ACCESS, // доступ на чтение-запись  
 0,0,0 // доступ ко всему отображению (можно задать часть)  
 );

Далее можно производить в области, адресуемой указателем pData действия по чтению и записи информации. Например, можно переписать оттуда 256 байт в короткую строку с именем S:

Move(pData^, S, 256);

Другие процессы должны не создавать отображающий объект, а получать хэндл готового по имени:

hMapping := OpenFileMapping(0,false,'MyMapping');

Далее тоже вызывается MapViewOfFile.

При использовании данного механизма для обмена данными имеется возможность отказаться от использования собственного файла-основы и отображать системный файл подкачки. Для этого в функции CreateFileMapping в качестве первого параметра можно задать –1 (т.е. $FFFFFFFF). В этом случае обязательно необходимо задавать размер отображения.

Образ файла должен уничтожаться после использования функцией

UnmapViewOfFile(pData);

в качестве аргумента указывается указатель на образ файла, полученный с помощью MapViewOfFile.

Отображающий объект и сам файл (если он создавался) закрываются с помощью функции CloseHandle:

CloseHandle(hMapping);  
 CloseHandle(hFile);

Обычно разделение памяти осуществляется между тесно связанными программами, входящими в состав одного пакета. Например, некоторые сервисные функции пакета могут быть реализованы в отдельных программах-утилитах, работающих как самостоятельно, так и под управлением главной программы пакета. В этом случае проблемы "кому создавать отображающий объект" обычно не возникает – его создает главная программа, а вызываемая ей программа-утилита использует готовый объект через OpenFileMapping.

Если же разделение памяти должно осуществляться между равноправными приложениями (например – разными экземплярами одного и того же приложения), то можно использовать следующий подход:

1. Приложение пытается открыть объект с помощью OpenFileMapping.

2. Если возвращено нулевое значение, то объект не существует, поэтому его надо создать именно в этом экземпляре приложения, а затем удалить при помощи CloseHandle.

**40. Страничный механизм организации памяти Windows. Декодирование линейного 32-разрядного виртуального адреса.**

При страничной организации виртуальной памяти память выделяется процессам одинаковыми небольшими блоками. Размер страниц = 4 кбайта. Каждая страница может находиться либо в оперативной памяти, либо на диске.

Особые управляющие структуры - таблицами страниц и каталогами страниц. Эти структуры аппаратно используются процессором для преобразования виртуальных адресов в физические. При этом биты 32-разрядного линейного адреса разделяются на три группы и используются следующим образом:

Адрес

биты D (31..22) биты P (21..12) биты A (11..0)

Каталог таблиц Таблица страниц Страница

адрес таблицы

данные

адрес страницы

CR3

Каждая страница имеет размер 4 килобайта, и поэтому начинается с адреса, в котором младшие 12 бит нулевые (12 бит адресуют 4 кбайта).

ОС поддерживает таблицы страниц, в которых содержатся 32-разрядные дескрипторы страниц. Каждая таблица страниц сама по себе является страницей и занимает 4 кбайта. Старшие 20 бит дескриптора содержат старшие 20 бит физического адреса, с которого начинается страница (младшие 12 бит этого адреса нулевые). В оставшихся 12 битах дескриптора кодируются различные признаки, в частности – находится ли страница в памяти или на диске, можно ли производить запись в нее, была ли такая запись и т.д.

Для каждого процесса ОС создает каталог таблиц страниц (тоже 4 кбайт, т.е. одна страница), с помощью которого процесс обращается к выделенным ему страницам памяти. Физический адрес этого каталога - в спецрегистре процессора CR3, и изменять его может только код, исполняемый в режиме ядра. Каждый элемент каталога таблиц содержит 32-разрядный дескриптор таблицы, в котором содержатся 20 старших бит ее адреса.

Когда в какой-либо машинной команде производится обращение по 32-разрядному адресу, процессор берет 20 разрядов адреса каталога из CR3, приписывает к ним десять битов поля D и два нуля и по полученному адресу извлекает дескриптор таблицы страниц. Затем берется 20 старших битов дескриптора, к ним приписываются 10 битов поля P, два нуля, и по полученному физическому адресу из таблицы страниц извлекается дескриптор страницы.

Если биты признаков дескриптора страницы содержат информацию о том, что страница находится в памяти, то происходит обращение к странице по физическому адресу, получаемому комбинацией 20 старших битов дескриптора и 12 битов поля А адреса. Если же страница отсутствует в памяти (находится на диске), то происходит аппаратное прерывание, обслуживая которое операционная система должна обеспечить подкачку нужной страницы с диска в память.

На самом деле процесс формирования физического адреса происходит без многократного обращения к таблицам дескрипторов в памяти, так как в процессоре есть средства для кэширования этой информации (хранения ее в сверхоперативной памяти внутри процессора).

Нетрудно видеть, что такой механизм управления памятью исключает проблему фрагментации физической памяти: страницы, образующие непрерывный блок в виртуальном адресном пространстве, могут быть как угодно разбросаны в физической памяти и загружаться при подкачке с диска в любое место. Вместе с тем, проблема фрагментации виртуальной памяти страничным механизмом не устраняется.